A Distributed Learned Hash Table

Shengze Wang[1], Yi Liu[1], Xiaoxue Zhang[2], Liting Hu[1], Chen Qian[1]

[1]*University of California Santa Cruz,* [2]*University of Nevada Reno*

{shengze, yliu634, liting, cqian12}@ucsc.edu, xiaoxuez@unr.edu

Abstract—**Distributed Hash Tables (DHTs) are pivotal in numerous high-impact key-value applications built on distributed networked systems, offering a decentralized architecture that avoids single points of failure and improves data availability. Despite their widespread utility, DHTs face substantial challenges in handling range queries, which are crucial for applications such as LLM serving, distributed storage, databases, content delivery networks, and blockchains. To address this limitation, we present LEAD, a novel system incorporating learned models within DHT structures to significantly optimize range query performance. LEAD utilizes a recursive machine learning model to map and retrieve data across a distributed system while preserving the inherent order of data. LEAD includes the designs to minimize range query latency and message cost while maintaining high scalability and resilience to network churn. Our comprehensive evaluations, conducted in both testbed implementation and simulations, demonstrate that LEAD achieves tremendous advantages in system efficiency compared to existing range query methods in large-scale distributed systems, reducing query latency and message cost by 80% to 90%+. Furthermore, LEAD exhibits remarkable scalability and robustness against system churn, providing a robust, scalable solution for efficient data retrieval in distributed key-value systems.**

I. INTRODUCTION

Key-value data management across distributed computing systems plays a crucial role in supporting large-scale Internet applications, including the emerging area of large language model (LLM) serving [1]–[6]. Distributed Hash Tables (DHTs) have been widely used for decentralized data management [7]–[9]. A DHT is a distributed data structure adept at performing storage and retrieval operations of key-value pairs across a decentralized network of nodes. DHTs mitigate the limitations of centralized architectures by eliminating single points of failure and distributing data loads across numerous nodes, thereby enhancing data availability and network efficiency [10]. State-of-the-art systems like InterPlanetary File System (IPFS) [11], Cassandra [12], [13], Tor [14], Namecoin [15], and Bittorrent [16], have exemplified the integration of DHTs in ensuring scalable and fault-tolerant data management in distributed networked systems.

The problem. Despite their widespread adoption and inherent advantages, DHT-based systems encounter significant challenges, particularly when handling complex queries such as range queries, which are important functions in applications such as KV-cache sharing in LLM serving [2]–[4], [17], distributed file systems and databases [11], [12], edge-cloud systems [18], [19], and blockchain systems [20], [21]. Current DHT systems are primarily optimized for single-key lookups. DHTs use a uniformly random hash function to distribute keys

into random locations, hence similar keys will be mapped to completely different storage locations. This feature of DHT will introduce two major limitations for range queries. First, all keys in the queried range need to be searched to ensure the completeness of the query. Second, these keys will be mapped to different locations based on the hash function. Accessing these locations will cause a high cost of network traffic. In the literature, efforts to improve range query performance in distributed systems have led to limited solutions. Armada [22] uses a partition tree model within the FissionE topology [23]. DBST [24] integrates binary search trees for range queries. MARQUES [25] employs space-filling curves in a multi-level overlay structure, bringing increased overhead and scalability issues. RQIOT [26] explores the idea of using order-preserving hashing to improve range query efficiency, yet how to design such a hash method, especially in a dynamic distributed system, is unclear. These solutions cannot completely resolve the two limitations of range queries in DHT.

Our solution. To address the critical issue – enabling efficient range queries for distributed networked systems – we introduce **LEAD** (LEArned DHT), a novel system that first integrates machine learning models with DHT frameworks to enhance the performance of range queries evidently. Drawing on the learned indexes proposed in recent years [27], which suggests that indexes could be conceptualized as "models" that predict the position of a key within a dataset, we argue that **a learned model can replace the hash function to distribute keys in networked systems**. By learning the cumulative distribution function (CDF) of keys, we can **maintain the inherent order of these keys** while mapping them to a decentralized group of nodes, making similar keys be placed in close locations. Hence the two limitations of random hash functions can be completely resolved. To minimize inference overhead and reduce the prediction error, we adapt the Recursive Model Index (RMI) structure [28] to train the learned model.

However, the idea of learning models to maintain the key relationships while disturbing keys consistently in DHT-based systems poses several challenges. First, we need to devise a strategy for managing key mapping and peer addressing, as well as utilizing the relationships between keys to conduct range queries efficiently. Second, the distributed environment is highly dynamic and characterized by frequent network churns; this requires the protocol to quickly adapt to network changes. Third, as the network expands and new data are introduced, the previously established Cumulative Distribution Function (CDF) on which the model was trained may no longer accurately

represent the new data distribution. Consequently, the learned model might not distribute data as uniformly as traditional hash functions, posing additional challenges for load balancing.

In response to these challenges, our protocol, LEAD, elaborates on the methodologies for applying learned models within DHT-based systems, focusing on the following aspects:

(1) We first introduce the concept of the Learned Hash Function under the realm of distributed key-value systems. We detail the strategy to map and retrieve keys with learned models for DHT-based systems. This approach renovates traditional hash functions that map keys to random positions, allowing LEAD to maintain the inherent order of keys and enhance range query performance.

(2) LEAD is designed to adapt dynamically to frequent changes in the system such as database size increases, node joins, and departures. It employs mechanisms that rapidly update the overlay routing tables and maintain the learned models, ensuring the system remains robust and efficient even in highly volatile environments. We propose a distributed model update method termed the Federated Recursive Model (FRM).

(3) LEAD incorporates a load-balancing model called *Shadow Balancer* using virtual nodes to allocate keys in an even manner that prevents overloading specific nodes, thus enhancing overall system performance and scalability.

(4) We conduct comprehensive evaluations of LEAD's performance in both implementation on real networked systems and simulations. The evaluation spans various network conditions, scales, and topologies, along with diverse datasets and data volumes. Our assessment demonstrates LEAD significantly outperforms existing baseline methods in range query efficiency, reducing latency by more than tenfold compared to traditional methods in current DHT-based systems. Additionally, LEAD exhibits remarkable scalability and resilience to network churn, maintaining logarithmic efficiency in single-key query performance.

(5) We conduct two timely case studies demonstrating LEAD's effectiveness in real-world applications that require efficient range queries: key-value cache management for LLM serving and the InterPlanetary File System (IPFS).

Beyond the immediate motivation of accelerating range queries in classical DHT deployments, the same order-preserving learned hash that powers LEAD unlocks a diverse set of emerging workloads: it can collocate semantically close embeddings in vector databases that serve retrieval-augmented LLMs, shard the rapidly growing key–value caches and adapter weights of distributed transformer inference without a central router, deliver geo-temporal IoT telemetry and edge-AI models to nearby gateways for low-latency analytics, adapt CDN object placement to shifting popularity skew in real time, and provide an range index across heterogeneous blockchains. These broader scenarios underscore LEAD's potential as a general storage substrate for next-generation, data-intensive distributed networked systems and motivate the design choices detailed in the rest of the paper.
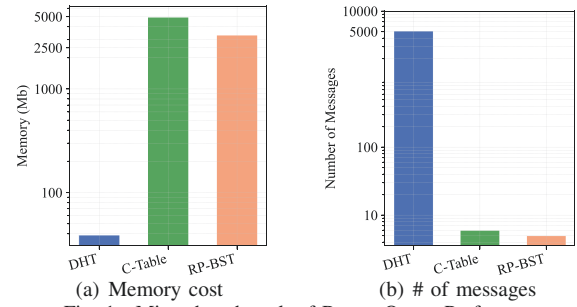

Fig. 1. Micro benchmark of Range Query Performance

II. BACKGROUND AND MOTIVATION

We conducted a focused micro-benchmark to expose the inherent trade-offs of three representative baselines: DHT-based key-value system with Chord [7] (DHT), the centralized range-location mapping table (C-Table), and the Range-Partition Binary Search Tree (RP-BST), whose details are presented in [29]. Chord utilizes a ring-like hashing space to manage key-value pairs and is highly efficient for single-key lookups due to its logarithmic routing efficiency. However, it struggles with range queries, which often require traversing multiple nodes sequentially, thereby increasing latency and message cost. We also implemented DBST [24] as an RP-BST overlay. Each node maintained a BST interval and two routing pointers (left/right). We evaluated the number of messages required to complete range queries and their memory overhead — critical metric affecting response time and the efficiency of data retrieval in distributed environments. The experimented system includes 100 nodes with 200 million key-value pairs from the 'osmc64' dataset (described in Section IV-A) and executed range queries for a range covering 2,000 keys after a given key. As depicted in Fig. 1, the centralized table substantially reduces the number of messages required to resolve range queries compared to DHT; however, it imposes a higher memory burden on the system. RP-BST–style overlays also improve messaging efficiency; however, they incur considerable memory consumption and control-plane complexity. Crucially, they offer limited resilience to network churn and impose high costs for index maintenance. Furthermore, both the centralized table and RP-BST overlay require a dedicated coordinator to maintain and synchronize metadata, introducing an additional bottleneck in distributed deployments. This underscores the necessity for a solution like LEAD, which aims to merge the advantages of both solutions.

III. LEAD DESIGN

A. System Overview

The section describes the LEAD system. It details the methodologies employed in LEAD for key mapping using its *Learned Hash Function*, addressing peers during node joins and departures, data retrieval mechanisms tailored for queries, and balancing the loads. Additionally, the protocol outlines stabilization and recovery strategies to handle system dynamics.

Fig. 2 presents the system design of LEAD. At a high level, physical nodes within the system are virtualized into multiple virtual nodes, each functioning as independent peers within a structured overlay network. Central to each peer is the learned
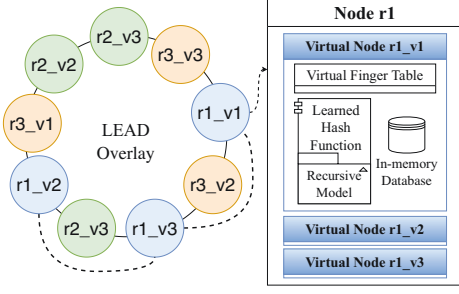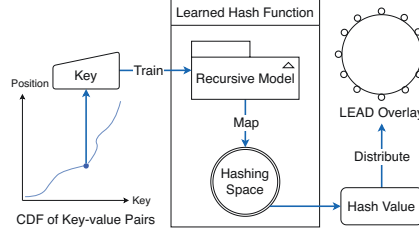
Fig. 2. LEAD System Design

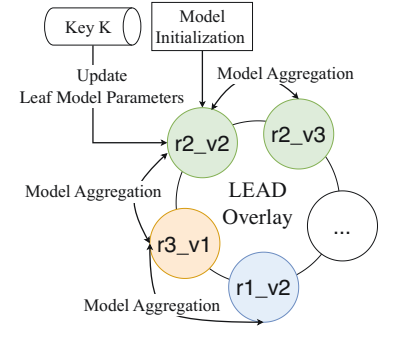Fig. 3. Key mapping with a learned hash function

Fig. 4. Decentralized Model Update

model utilized for efficient and in-order key mapping. This is complemented by a consistent hashing function employed specifically for peer addressing. Each peer also maintains a virtual finger table, the component for storing updated routing information and facilitating effective data queries. Additionally, peers are equipped with an in-memory database dedicated to the storage and rapid retrieval of key-value pairs.

B. Key Mapping with a Learned Hash Function

LEAD uses a learned hash function for key mapping, as showed in Fig. 3. Unlike traditional hash functions, which aim to map keys to random values within a specified range, the learned hash function strategically maps keys to order-preserving values in a hashing space. Utilizing the cumulative distribution function (CDF) of keys managed on the network, it maintains the inherent order of these keys while mapping them to a hashing space. This preservation of key relationships enhances systems with the capability for in-order data retrieval.

We employ the Recursive Model Indexes (RMI) structure [28] to implement the learned hash function in LEAD. In Section IV-C9, we will show that RMI provides the lowest latency compared to other learned models. The RMI structure is a hierarchy of models, where at each stage the model determines the appropriate child model to engage for a specified key. At the leaf level, models predict the relative position of a key within a dataset. A scale factor, S, is then applied to translate this relative position into a hashing space comprising H hash values. For instance, considering a two-stage RMI trained on N key-value pairs, the learned hash function, denoted as $LearnedHASH$, can be articulated as follows:

$$LearnedHASH(key) = \left\lfloor \frac{N}{H} \times f_2^{\left\lfloor \frac{B^a \times f_1(x)^b}{N} \right\rfloor}(K) \right\rfloor \quad (1)$$

[a]B referred to as the branching factor that determimines the number of "buckets" that data is divided into by the stage-one model

[b]f_i referred to as the ith stage model

$LearnedHASH$ is trained by optimizing the parameters of the given model by minimizing the squared error of its predictions. Specifically, a model k at stage ρ, denoted by $f_\rho^{(k)}$, is trained with the following loss function [27]:

$$L_\rho = \sum_{(x,y)^a} (f_\rho^{\left\lfloor \frac{M_\rho^b \times f_{\rho-1}(x)}{N} \right\rfloor}(x) - y)^2 \quad (2)$$

[a]x is the key, $y \in [0, N)$ is its relative position within a dataset
[b]Number of models at stage ρ

We introduce three systems-level optimizations that are critical for a fully-decentralized overlay for the vanilla RMI: (i) Auto-Model Selection. At bootstrap time a peer runs a lightweight mountain-climbing probe adapted from the learned-index tuner —that trains candidate leaf predictors on a 1% sketch of its local key sample, ranks them by 99-th-percentile prediction error and instantiates the for an optimal trade-off between model size and prediction error. (ii) The vanilla RMI assumes a fixed target domain. In practice, node joins and virtual-node churn change the effective density of the overlay, so a leaf that once mapped to may need only half that span an amount of updates later. Each peer therefore attaches a 2-field anchor ⟨offset,scale⟩ to its leaf model: the on-line gradient update adjusts offset to keep the median key centered and dials scale up/down with a 2-bit PID controller so the 95 % key-quantile always ends near the right edge of the peer's virtual-ID window. (iii) LEAD integrates a Federated Recursive Model (FRM) within its Learned Hash Function, enabling collaborative learning among peers for dynamic model updates. This decentralized design ensures load balancing and seamless request handling during model updates, as detailed in Section III-E3. $LearnedHASH$ maps each key to a hash value within the same hashing space used for peer addressing. While hash collisions for different keys are permissible, the hash value's primary role is to distribute the key across the network, not to serve as a unique identifier. Each key-value pair, identified by the key K, is assigned to the first peer whose VID (as detailed in Section III-D) either equals or follows the hash value produced by $LearnedHASH(K)$.

Model initialization and re-training. We assume the system starts with a small number (<10) of nodes with a limited amount of data. Hence, the very first model training can be conducted on an arbitrary node without causing a scalability problem. Then more nodes and data join the system, hence **one of the key contribution of LEAD is to adjust the network for newly joined nodes and re-train the learned model for new data**. The re-training mechanism will be detailed in Sec. III-E3.

C. Load balancing with virtual nodes

Achieving balanced load distribution in distributed key-value systems remains challenging. These systems contain heterogeneous nodes, with varied storage capacity and network bandwidth. Additionally, nodes may experience resource shortages

due to higher-priority tasks or hotspots (popular data items that attract many requests). These factors undermine the randomization and uniformity that consistent hashing aims for, leading to uneven load distribution, bottlenecks, and inefficiencies within the system. To address these challenges, LEAD employs a load balancing model called *Shadow Balancer*, which utilizes virtual nodes to optimize key distribution across the network and alleviate hotspot effects. As illustrated in Fig. 2, each physical node is virtualized into multiple virtual nodes, with each operating as an independent peer within the network. To facilitate efficient peer addressing and data retrieval processes, this design also leverages consistent hashing to ensure that these virtual peers are distributed as evenly as possible across the hashing space. The operational policy of the *Shadow Balancer* is formalized as follows:

(1) Each node virtualizes itself into k virtual peers, where k is adjustable according to the node's capabilities.
(2) In response to resource bottlenecks, a node plans the departure of virtual peers that manage fewer requests.

Even in resource-constrained environments, the Shadow Balancer adds minimal overhead. See [29] for its detailed analysis.

D. Peer Addressing

Along with the learned hash function, LEAD employs a consistent hashing mechanism known as $PeerHASH$ to assign an m-bit identifier, denoted as VID, to each peer in the network. Specifically, our implementation of LEAD utilizes a universal hash function as $PeerHASH$. Each physical node, referred to as N, hosts one or more virtual nodes, collectively called V. These virtual nodes are assigned unique port numbers, enabling direct inter-peer communication without intermediaries. The VID for each peer can be derived by hashing a concatenation of the corresponding node's IP address and its port number using $PeerHASH$. Every V maintains its own set of network routing information in a structure known as *virtual finger table*. In a hashing space holding h hashing values, the table holds $\lfloor \log h \rfloor$ entries, with each entry comprising a VID and the corresponding node's IP address. Similar to Chord, each ith entry in the virtual finger table of a virtual node V identifies the first node, S, that succeeds V by at least 10^{i-1} positions in the hashing space for peer addressing. We defines the $Successor(x)$ as the first peer whose VID is equal to or follows a hash value x in the peer addressing space. Consequently, the ith entry of the virtual finger table of V, denoted as $vfinger(i)$, can be formalized as

$$vfinger(i) = successor(VID + 10^{i-1}) \qquad (3)$$

1) Node Joins and Departs: To maintain the status of V in a dynamic network, each peer V must preserve the status of its successor. The process for a node (N) to join the network is outlined in the following procedures:

(1) **Initialization:** A new node initializes itself either as the first node in an empty network or by obtaining information about an existing peer (V_0) that is part of the network.
(2) **Node virtualization:** The node N creates n virtual nodes (Vs) and assigns them n unique ports. Their Virtual IDs (VIDs) are then generated using the $PeerHASH$.

(3) **Successor Discovery:** Each virtual node V dispatches a Remote Procedure Call (RPC) to V_0 to lookup for $Successor(VID)$ and obtain its knowledge of the network, including the successor's predecessor, successor, and virtual finger table. The lookup mechanics for $Successor(VID)$ are further detailed in Section III-E1.
(4) **Status Acknowledge:** Upon identifying its successor peer $Successor(VID)$, the virtual node V establishes itself within the network by setting $Successor(VID)$ as its immediate successor and adopting $Successor(VID)$'s current predecessor as its own. Subsequently, V issues RPC to $Successor(VID)$ instructing it to update its predecessor record to V. Concurrently, V sends another RPC to the predecessor of $Successor(VID)$, requesting an update of its successor record to V. Moreover, V copies the 2nd to i^{th} entries of the virtual finger table from $Successor(VID)$, which accelerates its initialization and stabilizes its initial operations within the network.
(5) **Key Transfer:** Once the virtual node V has successfully joined the network, it initiates the key transfer process: V requests $Successor(VID)$ to transfer the appropriate key-value pairs that fall within its responsibility range.

For planned node departures, the node N notifies the immediate successors and predecessors of its managed virtual nodes V. Subsequently, these virtual nodes V transfer their key-value pairs to their predecessors.

2) Virtual Finger Table Update: Accurate and up-to-date routing information is crucial for the efficiency and reliability of LEAD. LEAD maintains the peer addressing information in virtual finger tables. Periodically, each peer updates its virtual finger table by sending RPCs across the network to obtain each entry's latest successor and their status. Additionally, events such as node joins, departures, and failures trigger the affected nodes to update their virtual finger tables.

E. Data Retrieval

1) Single Key Lookup: The distributed single key lookup process in LEAD aims to locate the immediate successor of a key by identifying the first peer on the network whose VID equals or follows the hash value of the given key in the hashing space. P consults its virtual finger table to execute an optimal jump towards the key's hash identifier. This involves selecting the farthest preceding peer in the finger table that does not exceed the key's identifier, assuming this peer possesses closer or direct knowledge of the key. The query is then routed to this selected node, which follows the same procedure. This iterative process continues until the query reaches the peer responsible for managing the key, denoted as S. Upon locating the key, S dispatches an RPC directly back to P with the requested data, effectively completing the retrieval process with enhanced efficiency and minimized latency.

2) Range query: LEAD leverages the Learned Hash Function to distribute keys across the network while preserving their relationships in order-preserving hash values. Range queries in LEAD are handled based on the order-preserving key mapping by the Learned Hash Function. To execute a range query for a
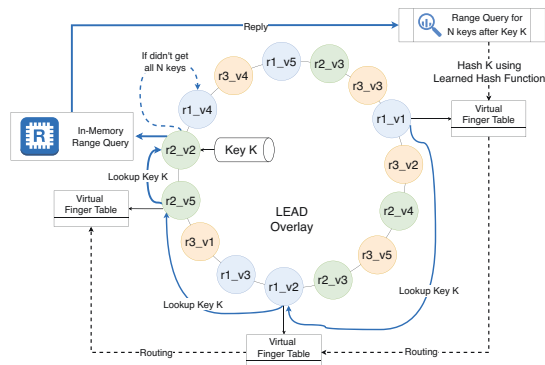
Fig. 5. Range Query in LEAD

sequence of n successive key-value pairs starting from key K, the initiating peer P first applies the Learned Hash Function to hash K (as shown in Fig. 5), yielding the hash value L_K. Using the single-key lookup mechanism described in Section III-E1, P locates the peer S responsible for K. Once the query reaches S, S performs a local range query within its in-memory database to retrieve the sequence of key-value pairs. If S holds only a portion of the required sequence, it forwards the remaining query to its successor. This forwarding process is repeated, moving through the chain of successors, until all n keys are retrieved. The final peer to fulfill the range query then sends the complete set of results back to the initiating peer P.

3) Model Update: While the learned hash function in LEAD efficiently distributes new key-value pairs across the network, challenges arise when this model no longer aligns with the overall Cumulative Distribution Function (CDF) of the keys managed across the network. Such misalignment can lead to increased hash collisions and an uneven distribution of key-value pairs, potentially overloading specific network peers. As detailed in Section IV-C10, the Learned Hash does not necessitate updates until new key-value pairs constitute up to 40% of the network's storage for the tested datasets. **Sub-optimized learned hash functions do not impact the correctness of system operations**, but they may affect load balancing if there is a significant logarithmic discrepancy between the learned hash function and the current data distribution. Model updates can help optimize the workload balancing across the network. To effectively manage these discrepancies, LEAD is proposed with the Federated Recursive Model (FRM) within its Learned Hash Function, promoting decentralized and cooperative learning among peers for dynamic model updates. As showed in Fig. 4, FRM incorporates the hierarchical structure of Recursive Models, with each peer in the network incrementally refining its segment of the leaf models based on locally observed data changes. The L0 layer in the FRM structure performs approximate predictions to identify the leaf model for specific keys. The hierarchical structure maintains stability in the L0 parameters when the model captures the approximate CDF of existing data. As such, when new keys are integrated into the network, the focus of FRM is on refining the corresponding leaf models for the unlearned keys. Each peer operates with two versions of the Recursive Model: one active in the current Learned Hash Function and another reserved for updates. The system

continuously monitors key distribution across the network in a decentralized manner through the tracking of the proportion of new key-value pairs integrated since the last model update at the peer level. When a new key-value pair is introduced to the network, the corresponding peer calculates the median index of keys it manages to determine the relative index for training, using its copy of the model designated for updates. The peer then selects the appropriate leaf model based on L0 layer predictions. Once the leaf model is identified, the peer refines this model. The relative index for training each new key is calculated by determining the median index of immediate keys currently managed by the peer. Given a network comprising n peers, with k keys distributed through the Learned Hash Function, we explore the scenario where m additional keys are introduced. To ascertain the proportion of these new keys observed by any given peer causes the total new keys on the network to exceed a predefined threshold t, we can model this expectation as $\frac{m}{k+m}$, assuming a relatively balanced load across the network. Then, we can achieve the threshold at t of the new key-value pairs observed by a peer, where the total new keys on the network exceed t of the total keys managed on the network since the last update with high probability. During the early phase of the LEAD network, when only a few peers are present, a randomly selected training peer is designated to initialize FRM. This initial coordinator is selected based on criteria such as computational power and network load. Once chosen, all peers in the network transfer their key-value pairs to this node. The central node then performs batch training to establish the initial parameters for the learned hash function. The process begins with a lightweight Model-Scout module that benchmarks multiple candidate leaf families (e.g., Linear, RadixSpline). A quick mountain-climbing search is then used to tune the model parameters, aiming for an optimal trade-off between model size and prediction error. Upon successful training, the model is adopted by other peers on the network through the stabilization process as discussed below. Peers are actively monitoring the proportion of new key-value pairs joined since the last model update. Once the proportion of new key-value pairs observed exceeds a threshold - specifically, 40% as identified in our empirical study in Section IV-C10, the peer flags the readiness status for the model update as true in its heartbeat message. Upon a peer being ready for a model update and detecting that a majority of its neighbors on the successor and predecessor list (e.g., 90%) are also flagged for updates, it takes the role of a transient coordinator. Then, it sends the flagged neighbors a Remote Procedure Call (RPC) to request confirmation of status and transfer of parameters. When such RPC is received by a peer, it pushes the updated leaf parameters to the transient coordinator, acknowledges readiness for the model update, and then resets its update-ready status, ensuring no redundant or conflicting update processes occur. During the parameter transfer, **only the segments that have changed are pushed to minimize data transfer size—for instance, only about 12 KB for approximately 1000 linear leaf model parameters and their segments stored in 32-bit**

format. After receiving acknowledgments from its neighbors, the transient coordinator aggregates the updated leaf model parameters from these peers through the averaging operation. Once the new model is consolidated, a new version number will be assigned to facilitate network-wide recognition and adoption. Peers in LEAD periodically check for the latest model version via heartbeat messages with their neighbors. The sectional transient inconsistency caused by updates does not compromise the continuous service of the LEAD system, as peer-addressing relies on an independent hash function. Moreover, **during cooperative model updates, the system remains operational**; only a subset of peers performs asynchronous updates on the leaf models. This is targeted at specific key segments and occurs until significant data changes are detected. Thus, the integrity of the system is preserved.

F. Stabilization and Failures Recovery

Handling system churn – where nodes frequently join or depart – is crucial for sustaining system integrity and performance. LEAD is designed to adapt rapidly to these changes through robust stabilization and failure recovery mechanisms. The correctness of LEAD is dependent on the current knowledge of its successors and predecessors within the network. Additionally, the efficiency of query handling is contingent upon the timeliness and accuracy of the virtual finger tables. To maintain this information, each peer periodically stabilizes themselves in the network through successor and predecessor verification, heartbeat communications, and virtual finger table maintenance. Building on its stabilization mechanisms, LEAD incorporates resilient failure recovery strategies to address peer failures. The details are presented in the [29].

IV. EVALUATION

This section presents the evaluation of LEAD through both testbed implementation and large-scale simulations, along with real-world case studies.

A. Methodology

Hardware and environments. The testbed implementation comprises nine virtual machines in public clouds, including three types of machines: one with two Intel Xeon Silver 4314 2.40 GHz 16-Core CPUs and 128GB of DDR4 2666MHz memory; one with an Intel Xeon E5-2687W v4 3.00GHz 12-Core CPU and 32GB of DDR4 2400MHz memory; and the other with an Intel Core i7-7700 3.60GHz 4-Core CPU and 16GB of DDR4 2400MHz memory. They communicate through the Internet. Each virtual machine runs 10 virtual nodes in the overlay hence the overlay includes 90 peers in total. We utilize Redis for in-memory key-value storage on peers.

The simulator we built, called p2psim+, is based on a publicly-available discrete event-driven simulator p2psim [30] running on an Ubuntu 22.04 LTS desktop with an AMD Ryzen 7 3700X 3.6 GHz 8-Core CPU, complemented by 32GB of DDR4 3200MHz RAM across two 16GB modules. P2psim is widely recognized and utilized within the community [31]. We added over 3,000 lines of C++ code to enhance the simulator. These extensions include the integration of LEAD, the support for user-defined network topologies, customized network behavior observers, and scalability enhancements for large experiments. We utilize the implementation of RMIs in Rust [28]. We will publish p2psim+ upon the acceptance of this paper.

Datasets. We leverage four real-world datasets from the SOSD benchmark [32], each consisting of 200 million 64-bit unsigned integers as keys. The datasets encapsulate a broad spectrum of data distributions and sources, described as follows:
 (1) 'osmc64': uniformly sampled OpenStreetMap Cell IDs
 (2) 'face64': randomly sampled Facebook user IDs
 (3) 'amzn64': Amazon book sale popularity data
 (4) 'wiki64': Wikipedia article edit timestamps

To accurately emulate real-world network topologies in our simulations, we incorporate the PlanetLab Dataset from the Network Latency Datasets [33]. This dataset captures round-trip times (RTTs) between 490 nodes dispersed across the PlanetLab network. Specifically, we employ the "PlanetLabData_1" as the latency model to construct the PlanetLab topology.

Baselines. We use four baseline methods in our experiments: the batch query approach on Chord [7] DHT with batch sizes of either 100 or 1000, and the recent work Marques [25]. We let Chord batch single-key queries together and send them as one or multiple consolidated requests across the network. Marques [25] is a recent enhancement on Chord [7] for range query efficiency. We exclude DBST [24] from direct comparison, as it relies on a centrally constructed binary search tree and incurs high overhead—analyzed in Section II—that renders it unsuitable for decentralized environments. Similarly, RQIOT [26] assumes centralized order-preserving hashing without providing a decentralized construction mechanism. Neither DBST nor RQIOT offer open-source implementations, further limiting their applicability in reproducible and fair comparison within our distributed system framework.

B. Testbed Performance

Fig. 6 presents the latency benchmark results obtained from the real-machine testbed implementation. For LEAD, a pre-trained two-layer model incorporating both linear and cubic layers is employed. For each experimental run, we inserted 200 million 64-bit unsigned integers from each of the four datasets. Then, we conducted range queries for ranges with varying numbers of keys, from 500 to 10,000, subsequent to a specified key. To ensure the reliability of the results, each query was repeated ten times, and we calculated the average latency for each data point. As demonstrated in Fig. 6, as the query range expands, LEAD maintains near-constant latency for range queries. In contrast, both the Batch Query method and Marques exhibit rapidly increasing latencies. For instance, in the experiment using the 'osmc64' dataset, a range query for 500 keys resulted in latencies of 259 ms for Batch Query with a batch size of 1000 and 557 ms for Marques, while LEAD efficiently resolved the query in just 145 ms. As the query range extended to 4,000 keys, the latency for Batch Query escalated to over 1,300 ms and for Marques to over 750 ms. Such latencies become prohibitive for most high-throughput applications. LEAD continued to deliver results in less than 150 ms, showcasing its superior performance and scalability.
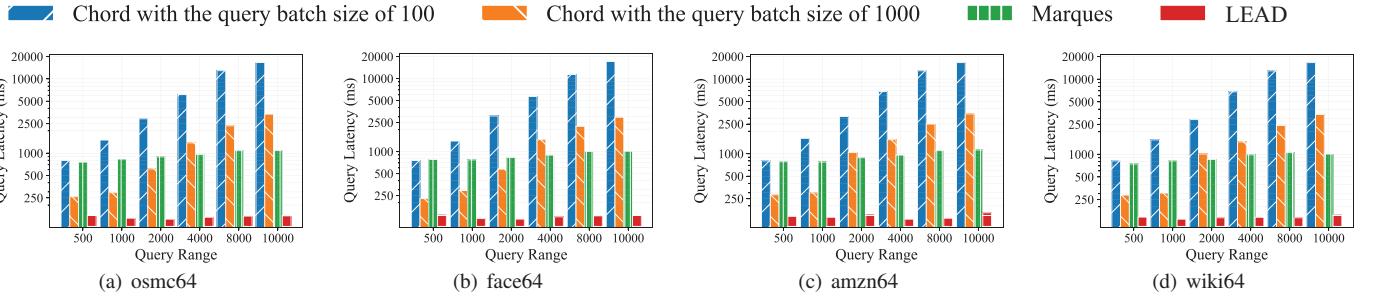
Fig. 6. Latency of range queries on various datasets in the real-machine testbed
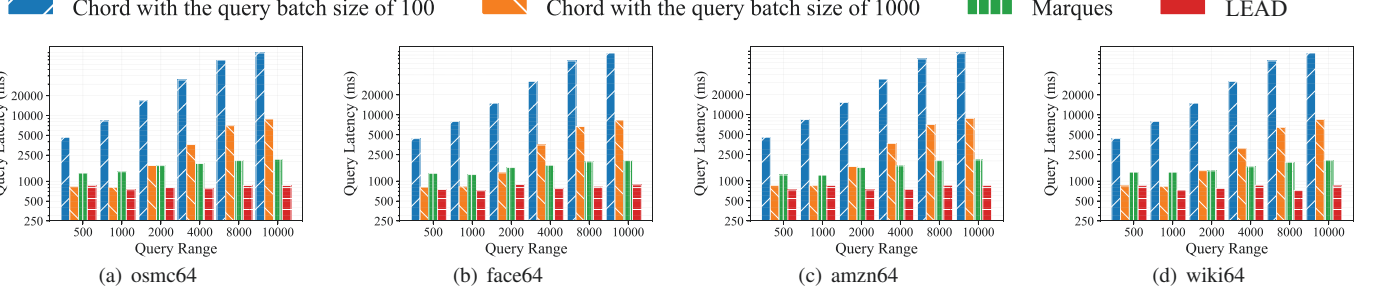

Fig. 7. Latency of range queries on various datasets from large-scale simulations
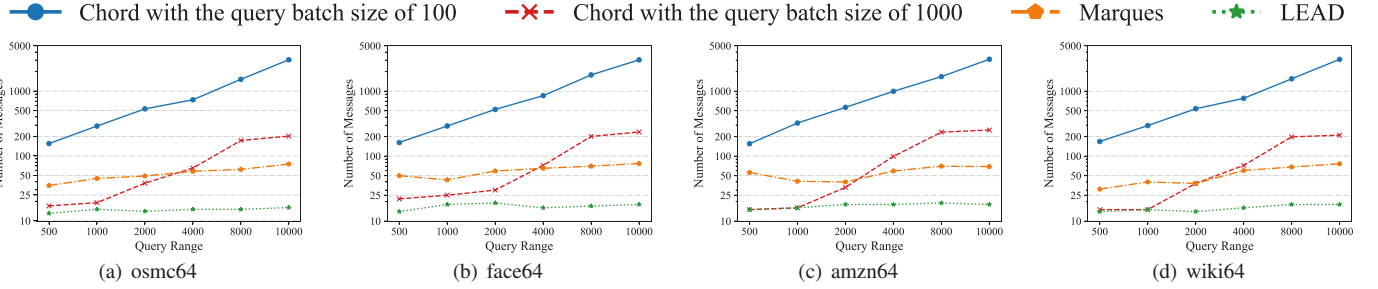

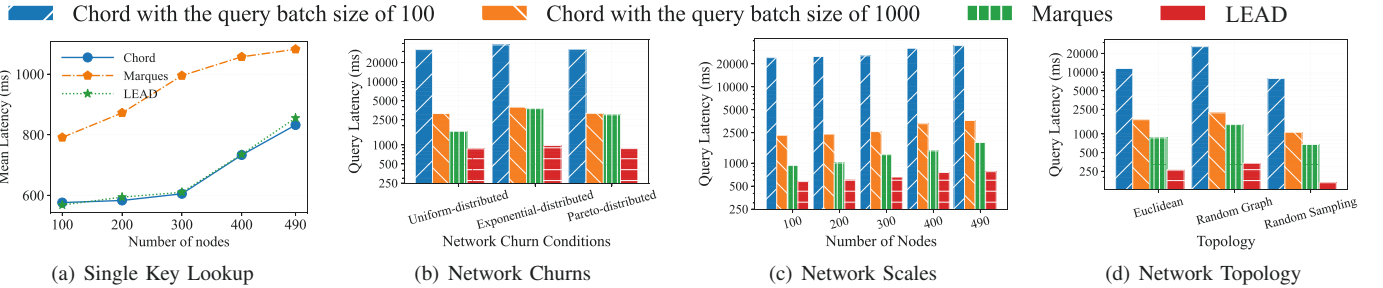Fig. 8. Number of messages of each range query on various datasets.

Fig. 9. Latency of range queries under various conditions

C. Simulation Results

The simulated system consists of 490 nodes configured according to the PlanetLab topology. Each node operates 10 virtual nodes. By default, we employed the pretrained two-layer models for LEAD, which incorporates both linear and cubic layers. Each simulation spanned a logical duration of 120 minutes. To emulate the dynamic nature of real-world distributed systems, node lifetimes were modeled with a uniform distribution, averaging 80 logical minutes. The network dynamics were initiated by exiting nodes from the network after their lifespan concluded and rejoining them following a uniformly distributed interval, averaging 10 logical minutes. Each time a node exited and rejoined, its routing state was

reset to preserve network integrity. Furthermore, to adapt to network changes effectively, the stabilization timer for each peer was set to 1000 logical ms, enabling regular updates to their finger tables and stabilization of their successor states. Range queries were conducted at regular intervals of five logical minutes throughout the simulation. Each query aimed to retrieve a sequence of N keys subsequent to a specified key M. For each query, we documented both the latency and the number of routing steps incurred. Following the completion of each test run, we calculated the average values for these metrics.

1) Range query performance: Fig. 7 illustrates the range query latency obtained from the simulation. Each experimental cycle involved inserting 200 million 64-bit unsigned in-

tegers from one of four distinct datasets: 'osmc64', 'face64', 'amzn64', and 'wiki64'. As the query range extends, the near-constant latency exhibited by LEAD underscores its substantial superiority in query latency compared to other baseline methods across all datasets tested. Again, the results show that LEAD significantly reduces the range query latency.

2) Query messages: To complement our latency analysis, we quantified the number of messages for each range query executed. Fig. 8 depicts the number of messages for range queries required across various test configurations, elucidating LEAD's optimized path efficiency for range queries. LEAD costs much fewer messages compared to the other baselines. For example, when the query range is 5000, LEAD only costs < 15 messages per query, while Marques needs > 50 messages and Chord, even with batching, requires > 200 messages for size 1000 and > 1000 messages for size 100. **LEAD reduces the query messages by over 80%.** We observe that LEAD typically incurs an amount of messages similar to those of a single-key lookup, which is logarithmic relative to network size.

3) Single-key performance: Alongside evaluating range query performance, we scrutinized the single-key lookup performance of each baseline method, utilizing the 'osmc64' dataset as a representative example. Fig. 9(a) demonstrates LEAD upholds competitive performance with Chord in single-key query latency. This is attributed to its adherence to the foundational design of Chord. On the other hand, Marques's multi-level overlay structure introduces more than a 50% increase in latency for single-key queries compared to the original Chord.

4) Network churn resistance: As illustrated in Fig. 9(b), the resilience of LEAD is demonstrated through its ability to maintain continuous service performance under various network churn conditions. The test setup involved populating the system with 200 million key-value pairs from the 'osmc64' dataset and executing range queries for 4,000 keys. Then we emulated the network dynamics through exiting nodes from the network after their lifespan concluded and rejoining them in intervals that followed uniform, exponential, or Pareto distributions.

5) Network scale: To assess scalability, we varied the network size from 100 to 490 nodes. We utilized the 'osmc64' dataset, which consists of 200 million key-value pairs, to measure latency by executing range queries for 4,000 keys. Fig. 9(c) illustrates that LEAD consistently outperforms other baseline methods across all network sizes evaluated.

6) Network topology: Fig. 9(d) demonstrates that LEAD consistently surpasses other baseline methods in range query latency across all evaluated network topologies. In continuation of our scalability testing, with the network size held constant at 490 nodes, we assessed the performance of LEAD across three synthetic network topologies: 1) Euclidean, in which the latencies between nodes were modeled by their distances in a two-dimensional Euclidean space; 2) Random graph; and 3) Random sampling, in which the inter-node latencies were randomly assigned within a range.

7) Load balancing: Figure 10 compares the load distribution of a traditional Chord DHT setup against LEAD integrated with our *Shadow Balancer*, which enables 10 virtual nodes
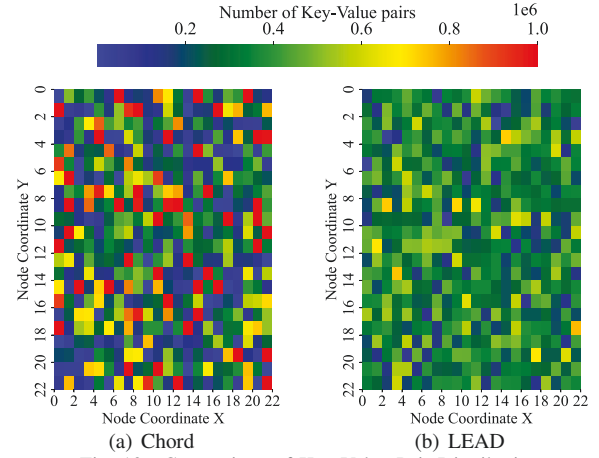

Fig. 10. Comparison of Key-Value Pair Distribution

per physical node. The heat map representation shows that LEAD significantly enhances load balancing within the network. LEAD with Shadow Balancer exhibits a more uniform green color across the network, indicating a well-balanced load among nodes. As depicted in Fig. 11, increasing the number of virtual nodes decreases the standard deviation, suggesting better load dispersion. Specifically, the inflection point at 10 virtual nodes per node in the PlanetLab topology indicates an optimized balance. Beyond this point, additional virtual nodes do not significantly improve load balancing, thereby identifying 10 virtual nodes per node as an ideal configuration for the established network.

TABLE I
RECURSIVE MODEL EVALUATION

Model	Maximum Log2 Error	Average Log2 Error	Size (Mb)
Linear	25.79	18.51	0.75
Radix	21.28	12.79	1.75
Cubic	18.63	9.82	12.00

8) Learned models: LEAD leverages the Recursive Mode structure for fast and accurate order-preserving key mapping with the learned hash function. Selecting the optimal model type during the training phase is crucial to minimize the prediction error, thereby ensuring LEAD achieves optimal load distribution. Fig. 12 illustrates the results of range query latency on the 'osmc64' dataset for queries ranging from 500 to 10,000 keys using three typical models as detailed in Table I. For the 'osmc64' dataset, our evaluations revealed that both the Radix and Cubic models can aptly fit its distribution, showcasing effective performance in managing range queries. In contrast, although the linear model offers benefits in terms of smaller model size, it results in increased error bounds, which can adversely affect the system performance.

9) Learned Indexes with LEAD: In formulating the learned model for LEAD, we assessed various learned index structures, including RMI [27], Radix Spline Indexes [34], and Piece-wise Geometric Model Indexes (PGM) [35]. Our evaluations, depicted in Figure 13, highlight the Recursive Model structure's consistent performance advantage across various query ranges when integrated with LEAD. Consequently, the Recursive Model structure is the preferred choice for LEAD, ensuring efficient and accurate range query handling.
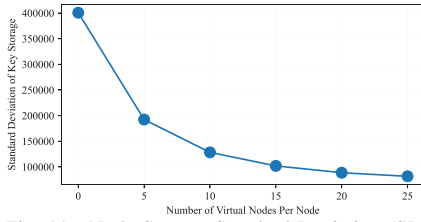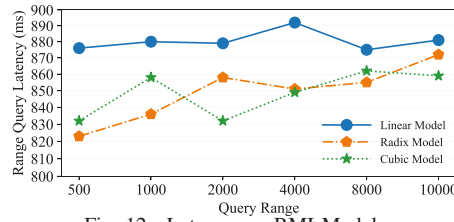
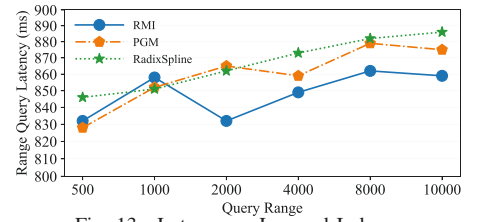Fig. 11. Node Storage Standard Deviation (SD.)

Fig. 12. Latency vs. RMI Models
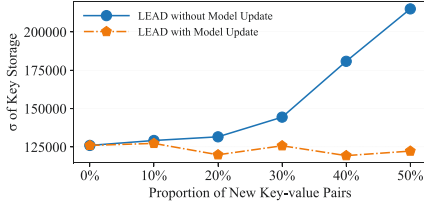

Fig. 13. Latency vs Learned Indexes
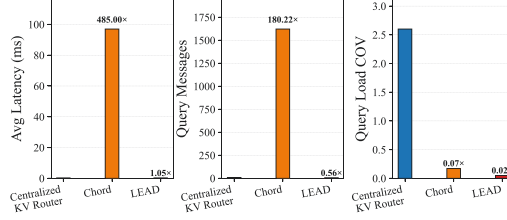

Fig. 14. SD. of Storage with Model Update

Fig. 15. KV Cache Management
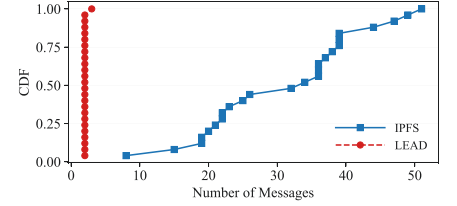

Fig. 16. CDFs of the message cost in IPFS

10) Model Update: We randomly selected a portion of the 'osm' dataset as the training set, treating the remainder as unlearned, new key-value pairs. These new pairs were then distributed across the network, alongside the existing data. To quantify the impact of introducing new data on network load balancing, we recorded the standard deviation of key-value pairs stored per node as the proportion of new entries increased. This metric was assessed for both versions of LEAD: one without model updates and one with dynamic model updates enabled. Fig. 14 illustrates the effects of new key-value pair integration on load balancing across the network. The results indicate that the key mapping with our learned hash function remains stable, with minimum impacts on load balancing, even as new key-value pairs constitute up to 40% of the network's storage. Fig. 14 also shows that our dynamic model update mechanism sustains optimal load balance across the network, demonstrating LEAD's robust adaptability to data changes within the network.

Our real-world testbed confirms that **maintaining LEAD's learned hash function adds only marginal overhead to a vanilla Chord control plane**. At the evaluated scale—one RMI instance per peer collectively managing 200 million 64-bit key–value pairs—the Linear and Radix models occupy under 2 MB of DRAM, while the Cubic model remains below 12 MB. **Model Update is likewise cheap and invoked only after the system observes a 40 % drift in new keys**: every key insertion triggers a gradient update that averages 2.1 µs on our testbed—roughly 0.2 % of end-to-end insertion latency and completely hidden by network delay. The model re-synchronisation exchanging $\leq$ 60 KB—about one-third of the bytes a peer already spends during a finger-table refresh cycle. Overall, CPU, memory, and network overheads for the RMI instance stay below 4 %, 0.1 %, and 3 %, respectively.

D. Case study I : KV Cache Management for LLM Severing.

In LLM serving, key–value (KV) caches retain the attention keys and values of earlier tokens for reuse rather than recomputation. Hence how to share and re-use existing KV caches is a crucial problem [2]–[4], [17], [36]. We consider a distributed LLM inference system where multiple nodes (GPU workers) collaboratively serve incoming queries. Each node caches KV blocks [36] from sequences it has processed, and nodes cooperate to serve future queries that may need those cached blocks, similar to a CDN. The goal is to leverage the existing KV cache instead of recomputing from scratch, by retrieving cached KV blocks from the network. KV blocks belonging to the same shared prefix can be stored on the same or nearby nodes. Since an LLM needs the KV for all prior tokens in the sequence, LEAD can fetch a whole span of positions in one efficient range query. That query is routed only to the node(s) responsible for that contiguous key range, obviating any need for system-wide broadcasts or gathers. We experiment a distributed inference system consisting of eight worker nodes—each equipped with a single NVIDIA A100 80 GB GPU running the Llama-3 8B model. The KV cache is managed using the PagedAttention scheme with a fixed 16-token block size [36]. Every token is issued a composite key. Our evaluation workload is Long-DocQA [37], which comprises 776 lengthy documents paired with 6,400 questions. LooGLE constructs each prompt by prefixing a full document to its associated question; after Zipf-0.6 sampling, prompts average 10,985 tokens [17], [38]. We simulate a production environment where cold KV blocks can be offloaded to host RAM, resulting in over 500 million KV blocks under management. In the absence of local shared-prefix caching, each inference request must retrieve on average more than 700,000 blocks from the network. We compare LEAD against two baselines: (i) Centralized KV Router: maintains a global index of KV locations across all workers. (ii) Chord-based DHT: uses a distributed hash table overlay for KV storage. Assuming a 10 GbE TCP/IP data-center network with a 100 µs average round-trip latency, we measure each system's average block-lookup latency, total message count, and cache-hit rate. As Fig. 15 illustrates, LEAD only add a marginal latency increase to the ideal centralized case, while delivering the resilience and scalability of a fully decentralized design—dramatically outperforming a traditional DHT. Notably, the centralized router suffers from severe query load imbalance, as evidenced by a high query CoV—the coefficient of variation in the number

of messages handled per node during range queries. All three systems achieve comparable cache hit rates under identical query correctness guarantees. These results highlight LEAD's foundation for enabling workload-aware, multi-tiered caching across heterogeneous GPU clusters, supporting dynamic inference pipelines and fault-tolerant, network-aware KV-cache management in scalable LLM serving infrastructures.

E. Case study II : InterPlanetary File System (IPFS).

InterPlanetary File System (IPFS) [11] is a distributed content delivery network that stores, retrieves, and locates data based on the Content Identifiers (CIDs) of its actual content rather than its name or location. With millions of daily content retrievals, IPFS supports numerous third-party applications, demonstrating its broad utility and impact. However, the traditional DHTs used in IPFS, i.e., Kademlia [39], face challenges in handling range queries, which are essential for efficiently retrieving sequences of data blocks or related files. To gauge LEAD's benefit in a production-style CDN, we forked the reference go-ipfs daemon [40] and replaced only its routing module with a LEAD overlay, leaving libp2p, Bitswap, and the block-store unmodified. We emulated an IPFS network with 100 peers using the PlanetLab topology. We generated 100 million synthetic key-value pairs to represent the logical units structuring the metadata of resources (files). For LEAD, the CIDs were managed by the learned hash function within SHA-1's hashing space. The key operation tested was a typical user request for a resource. In this scenario, an edge server retrieves all the blocks containing the metadata for the requested resource, which consists from 1,000 to 3,000 blocks in our test case. For the emulated IPFS, the server divided the 1,000 lookup queries into 10 batches and sent them over the network. In contrast, for LEAD, the query was optimized as a supported meta-operation range query. We recorded the number of hops required to resolve the request for both systems. As illustrated in Fig. 16, the CDFs of the retrieval hops for the emulated IPFS and LEAD indicate a significant reduction in the number of messages required to complete data block sequences retrieval when using LEAD.

V. Related Work

Range query in DHTs. Current DHT systems have significant limitations in handling range queries. These systems are inherently designed for exact key-based queries, and therefore, their hashing mechanisms lose the semantic relationship between keys—necessary for range queries. Significant efforts to facilitate efficient range queries in distributed networked systems have introduced innovative concepts while also revealing inherent limitations. Early attempts to reconcile hash-based load-balancing with ordered access bolted auxiliary data structures onto a vanilla DHT: Prefix-Hash-Trees (PHT) [41] and Range Search Trees [42] layer Chord-style fingers with a try that must be eagerly split and merged on every insert, leading to high control traffic per update and poor churn tolerance. Armada [22] utilizes a partition tree model and a tailored algorithm within the FissionE [23] topology to enhance range query efficiency. Nevertheless, its reliance on a customized

DHT scheme restricts its broader applicability. Similarly, DBST system [24] integrates binary search tree structures to provide efficient range queries for ordered data. These tree constructions are assumed to be centralized and are not applicable to large-scale distributed systems. MARQUES [25] employs space-filling curves within a multi-level overlay structure derived from Chord [7], targeting enhanced performance for range queries. Nonetheless, the complexity involved in managing this structured network overlay can substantially introduce overheads and pose scalability challenges. The latest work, RQIOT [26], tried to employ order-preserving hashing to handle range queries. However, how to implement such a hash method, especially in a dynamic distributed system, is unclear.

Learned Index Structures and Hash Functions. Recent research has reimagined traditional indexing by conceptualizing indexes as predictive models that estimate the position of a key within a dataset [27], [35], [43]–[47]. These learned index structures combine machine learning techniques with classical data structures to accelerate key lookups. Kraska et al. [27] proposed the Recursive Model Index (RMI) to address the inaccuracy of using a single model to approximate the dataset's CDF. There has been growing interest in learned hash functions, where models are trained to map keys to hash buckets in a data-aware manner. Prior works on locality-sensitive hashing (LSH) [48]–[50] have explored model-driven hash functions for approximate nearest neighbor search. More recently, Sabek et al. [51] demonstrated that learned models can achieve comparable or even fewer hash collisions than traditional hash functions. However, while these approaches show promise, integrating learned index structures or hash functions into decentralized systems such as Distributed Hash Tables remains unexplored—an opportunity that LEAD seeks to address.

VI. Conclusion

This paper introduces LEAD, a novel distributed key-value storage and lookup system designed to enhance the efficiency of range queries by incorporating learned models with DHTs. LEAD includes the detailed design of training and updating learned models, implementing single-key and range queries, achieving load balancing, and dealing with system churns. Extensive evaluations on both testbed implementation and simulations demonstrate that LEAD significantly reduces the latency and message cost of performing range queries by by 80% to $90\%+$, compared to existing DHT-based solutions. LEAD can maintain system consistency under dynamic changes and various system conditions.

We believe LEAD opens a completely new field for further research on integrating learned models with distributed systems. The implementation details and supplementary material are available at https://github.com/ShengzeWang/LEAD and [29].

REFERENCES

[1] Y. Wang, Y. Chen, Z. Li, Z. Tang, R. Guo, X. Wang, Q. Wang, A. C. Zhou, and X. Chu, "Towards efficient and reliable llm serving: A real-world workload study," *arXiv e-prints*, pp. arXiv–2401, 2024.

[2] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 611–626.

[3] L. Zheng *et al.*, "Sglang: Efficient execution of structured language model programs," in *Proc. of NeurIPS*, 2024.

[4] Y. Cheng, K. Du, J. Yao, and J. Jiang, "Do large language models need a content delivery network?" *arXiv preprint arXiv:2409.13761*, 2024.

[5] Y. Liu, H. Li, Y. Cheng, S. Ray, Y. Huang, Q. Zhang, K. Du, J. Yao, S. Lu, G. Ananthanarayanan, M. Maire, H. Hoffmann, A. Holtzman, and J. Jiang, "Cachegen: Kv cache compression and streaming for fast large language model serving," in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024.

[6] F. Fang, Y. Hua, S. Wang, R. Zhou, Y. Liu, C. Qian, and X. Zhang, "Gentorrent: Scaling large language model serving with an overley network," 2025. [Online]. Available: https://arxiv.org/abs/2504.20101

[7] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM*, vol. 31, no. 4, 2001.

[8] J. Zarrin, R. L. Aguiar, and J. P. Barraca, "Resource discovery for distributed computing systems: A comprehensive survey," *Journal of parallel and distributed computing*, vol. 113, pp. 127–166, 2018.

[9] A. Passarella, "A survey on content-centric technologies for the current internet: Cdn and p2p solutions," *Computer Communications*, vol. 35, 2012.

[10] M. Coluzzi, A. Brocco, P. Contu, and T. Leidi, "A survey and comparison of consistent hashing algorithms," in *2023 IEEE ISPASS*, 2023.

[11] D. Trautwein, A. Raman, G. Tyson, I. Castro, W. Scott, M. Schubotz, B. Gipp, and Y. Psaras, "Design and evaluation of ipfs: a storage layer for the decentralized web," in *ACM SIGCOMM 2022*, 2022, pp. 739–752.

[12] Apache, "Apache cassandra: Open source nosql database." [Online]. Available: https://cassandra.apache.org/

[13] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS*, vol. 44, no. 2, pp. 35–40, 2010.

[14] "The Tor Project | Privacy & Freedom Online — torproject.org," https://www.torproject.org/, [Accessed 13-05-2025].

[15] Namecoin, "Namecoin." [Online]. Available: https://www.namecoin.org/

[16] L. BitTorrent, "Bittorrent: The world's most popular torrent client." [Online]. Available: https://www.bittorrent.com/

[17] V. Srivatsa, Z. He, R. Abhyankar, D. Li, and Y. Zhang, "Preble: Efficient distributed prompt scheduling for llm serving," *arXiv preprint arXiv:2407.00023*, 2024.

[18] M. Chiang and T. Zhang, "Fog and IoT: An Overview of Research Opportunities," *IEEE Internet of Things Journal*, 2016.

[19] X. Li, M. Wang, S. Shi, and C. Qian, "VERID: Towards Verifiable IoT Data Management," in *Proceedings of ACM/IEEE IoTDI*, 2019.

[20] D. Marijan and C. Lal, "Blockchain verification and validation: Techniques, challenges, and research directions," *Computer Science Review*, vol. 45, p. 100492, 2022.

[21] Z. Nie, J. Li, F. Duan, and Y. Lu, "A collaborative ledger storing model for lightweight blockchains based on chord ring," *The Journal of Supercomputing*, vol. 80, no. 4, pp. 5593–5615, 2024.

[22] D. S. Li, J. Cao, X. C. Lu, and K. C. C. Chan, "Efficient range query processing in peer-to-peer systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 1, pp. 78–91, 2009.

[23] D. Li, X. Lu, and J. Wu, "Fissione: a scalable constant degree and low congestion dht scheme based on kautz graphs," in *IEEE INFOCOM 2005*, 2005.

[24] S. Ahmed, A. Shome, and M. Biswas, "Dbst: A scalable peer-to-peer distributed information system supporting multi-attribute range query," in *2021 ICSCT*, 2021, pp. 1–6.

[25] A. Sen, A. S. M. S. Islam, and M. Y. S. Uddin, "Marques: Distributed multi-attribute range query solution using space filling curve on dths," in *2015 NSysS*, 2015, pp. 1–9.

[26] B. Djellabi, M. Younis, and M. Amad, "Effective peer-to-peer design for supporting range query in internet of things applications," *Computer Communications*, vol. 150, pp. 506–518, 2020.

[27] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *ACM SIGMOD 2018*, 2018, pp. 489–504.

[28] R. Marcus, E. Zhang, and T. Kraska, "Cdfshop: Exploring and optimizing learned index structures," in *ACM SIGMOD 2020*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020.

[29] S. Wang, Y. Liu, X. Zhang, L. Hu, and C. Qian, "A distributed learned hash table," 2025. [Online]. Available: https://arxiv.org/abs/2508.14239

[30] T. Gil, F. Kaashoek, J. Li, R. Morris, and J. Stribling, "p2psim, a simulator for peer-to-peer protocols," 2003.

[31] J. Risson and T. Moors, "Survey of research towards robust peer-to-peer networks: Search methods," *Computer Networks*, vol. 50, no. 17, 2006.

[32] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "Sosd: A benchmark for learned indexes," *arXiv preprint arXiv:1911.13014*, 2019.

[33] R. Zhu, B. Liu, D. Niu, Z. Li, and H. V. Zhao, "Network latency estimation for personal devices: A matrix completion approach," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 724–737, 2017.

[34] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "Radixspline: a single-pass learned index," in *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, ser. aiDM '20. New York, NY, USA: Association for Computing Machinery, 2020.

[35] P. Ferragina and G. Vinciguerra, "The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds," *VLDB 2020*, 2020.

[36] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 611–626.

[37] J. Li, M. Wang, Z. Zheng, and M. Zhang, "Loogle: Can long-context language models understand long contexts?" *arXiv preprint arXiv:2311.04939*, 2023.

[38] F. Fang, Y. Hua, S. Wang, R. Zhou, Y. Liu, C. Qian, and X. Zhang, "Gentorrent: Scaling large language model serving with an overley network," *arXiv preprint arXiv:2504.20101*, 2025.

[39] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *International Workshop on Peer-to-Peer Systems*. Springer, 2002, pp. 53–65.

[40] Ipfs, "Ipfs/kubo: An ipfs implementation in go." [Online]. Available: https://github.com/ipfs/kubo

[41] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker, "Prefix hash tree: An indexing data structure over distributed hash tables," in *Proceedings of the 23rd ACM symposium on principles of distributed computing*, vol. 37. St. John's Newfoundland, Canada, 2004.

[42] J. Gao and P. Steenkiste, "An adaptive protocol for efficient support of range queries in dht-based systems," in *Proceedings of the 12th IEEE International Conference on Network Protocols, 2004. ICNP 2004*. IEEE, 2004, pp. 239–250.

[43] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann *et al.*, "Alex: an updatable adaptive learned index," in *ACM SIGMOD 2020*, 2020, pp. 969–984.

[44] B. Lu, J. Ding, E. Lo, U. F. Minhas, and T. Wang, "Apex: a high-performance learned index on persistent memory," *arXiv preprint arXiv:2105.00683*, 2021.

[45] J. Wu, Y. Zhang, S. Chen, J. Wang, Y. Chen, and C. Xing, "Updatable learned index with precise positions," *arXiv preprint arXiv:2104.05520*, 2021.

[46] C. Tang, Y. Wang, Z. Dong, G. Hu, Z. Wang, M. Wang, and H. Chen, "Xindex: a scalable learned index for multicore data storage," in *25th ACM SIGPLAN*, 2020, pp. 308–320.

[47] P. Li, Y. Hua, J. Jia, and P. Zuo, "Finedex: a fine-grained learned index scheme for scalable and concurrent memory systems," *VLDB 2021*, 2021.

[48] J. Wang, T. Zhang, N. Sebe, H. T. Shen *et al.*, "A survey on learning to hash," *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 4, pp. 769–790, 2017.

[49] M. Turčaník and M. Javurek, "Hash function generation by neural network," in *2016 NTSP*, 2016, pp. 1–5.

[50] J. Wang, J. Wang, N. Yu, and S. Li, "Order preserving hashing for approximate nearest neighbor search," in *21st ACM Multimedia*, 2013, pp. 133–142.

[51] I. Sabek, K. Vaidya, D. Horn, A. Kipf, M. Mitzenmacher, and T. Kraska, "Can learned models replace hash functions?" *Proc. VLDB Endow.*, vol. 16, no. 3, p. 532–545, nov 2022.