Front. Comput. Sci., 2025, 19(9): 199607
https://doi.org/10.1007/s11704-024-40509-4

RESEARCH ARTICLE

LRP: learned robust data partitioning for efficient processing of large dynamic queries

Pengju LIU[1,2], Pan CAI[1,2], Kai ZHONG[1,2], Cuiping LI (✉)[1,2], Hong CHEN[1,2]

1 Engineering Research Center of Database and Business Intelligence, Ministry of Education, Beijing 100872, China
2 School of Information, Renmin University of China, Beijing 100872, China

Abstract The interconnection between query processing and data partitioning is pivotal for the acceleration of massive data processing during query execution, primarily by minimizing the number of scanned block files. Existing partitioning techniques predominantly focus on query accesses on numeric columns for constructing partitions, often overlooking non-numeric columns and thus limiting optimization potential. Additionally, these techniques, despite creating fine-grained partitions from representative queries to enhance system performance, experience from notable performance declines due to unpredictable fluctuations in future queries. To tackle these issues, we introduce LRP, a **l**earned **r**obust **p**artitioning system for dynamic query processing. LRP first proposes a method for data and query encoding that captures comprehensive column access patterns from historical queries. It then employs Multi-Layer Perceptron and Long Short-Term Memory networks to predict shifts in the distribution of historical queries. To create high-quality, robust partitions based on these predictions, LRP adopts a greedy beam search algorithm for optimal partition division and implements a data redundancy mechanism to share frequently accessed data across partitions. Experimental evaluations reveal that LRP yields partitions with more stable performance under incoming queries and significantly surpasses state-of-the-art partitioning methods.

Keywords data partitioning, data encoding, query prediction, beam search, data redundancy

1 Introduction

Data partitioning is a pervasive concept in our daily, with a simple analogy being the organization of goods within supermarkets into distinct sections for easier navigation. This principle is extensively adopted in database management systems (DBMS) as a crucial step in database physical design [1–3] to enhance the efficiency of data retrieval and management. Horizontal partitioning (HP) [4–9] stands out as a critical branch of data partitioning that aims for minimal row granularity during data division to optimize data management. It involves selecting commonly used column distributions or mining the query-data access relationship as partitioning features to finely allocate table data into smaller partition files, thereby accelerating queries. For example, in Fig. 1①, if a two-dimensional tablespace $\mathcal{D}$ (400 MB in size) is partitioned based on the areas targeted by queries (black dots), we only require accessing two partitions R_1 and R_2 (165 MB in total) to fetch all the required data. This method circumvents the need to scan the entire tablespace, which often occurs when using simplistic data-driven partitioning strategies, such as uniform or random division of the tablespace.

Recent HP studies [7–14] regard query acceleration as the primary optimization objective, extracting valuable predicate conditions to greedily split given $\mathcal{D}$ for maximizing data skipping. However, there are two major limitations:

(1) Current research only considers numeric column-related predicates, making the split partitions not fully adaptable to real querying areas. For example, by comparing Figs. 1① and 1②, the measured querying boundaries (black dots) are larger than the real querying ones (green dots) due to the neglect of non-numeric predicates. This discrepancy results in an inaccurate data layout (solid boxes, 165 MB) that scans an additional 25 MB of data compared to the optimal layout (dotted boxes, 145 MB). Thus, considering predicates related to non-numeric data is crucial for partitioning, especially for tables like TPC-H [15], which have a few numeric columns. In such tables, 73.6% of the columns are non-numeric, and 86% of queries utilize these columns in filter conditions.

(2) Another limitation is that most studies demonstrate superior performance for static queries but are difficult to adapt to query shifts. Only a few studies [11,13,14] adapt to dynamic query workload through periodic data re-partitioning, which can be a costly operation for DBMS. For example, in Fig. 1③, we have created two partitions R_1 and R_2, for old queries (black dots) in $\mathcal{D}$. However, as new queries (red dots) arrive, the amount of scanned data can increase from 165 to 400 MB. This occurs because the partially queried areas outside R_1 and R_2 are randomly allocated to different partitions, potentially requiring a search of the entire table. By assessing the similarity of querying boundaries between new and old queries (introduced from [9]), we categorize new

Received May 21, 2024; accepted September 8, 2024

E-mail: licuiping@ruc.edu.cn

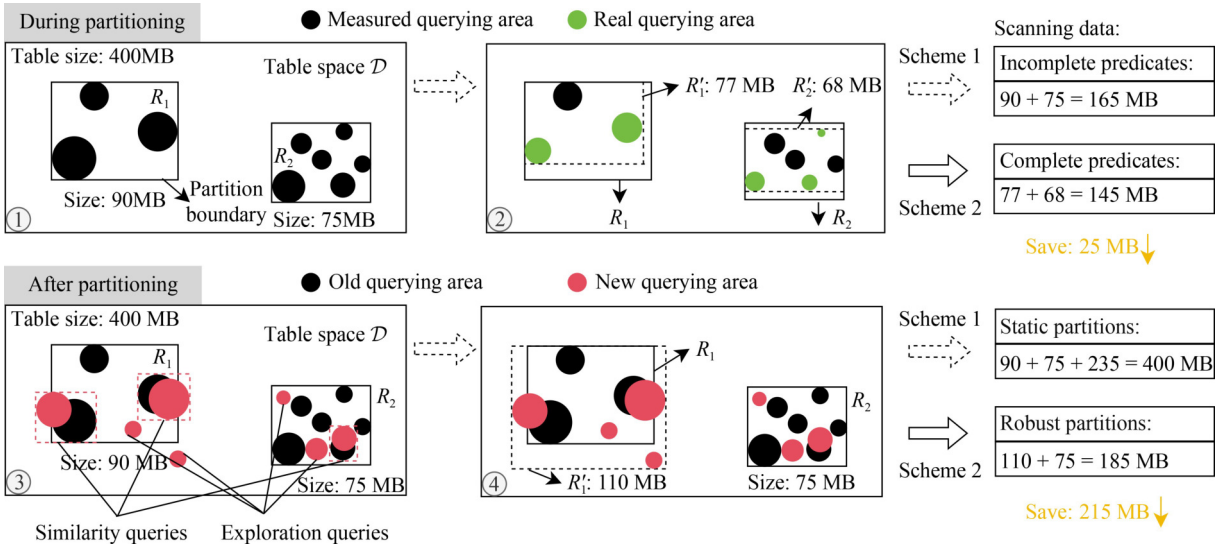

Fig. 1 LRP Motivation Examples: comparing the amount of scanning data between ideal and existing partitioning scheme during query execution on the given tablespace $\mathcal{D}$. We considered two critical time points during the partitioning cycle

queries into two types: similarity queries (red dotted boxes) and exploratory queries (other red dots). If the robust partition R'_1 is created in advance for incoming similarity and exploratory queries (see Fig. 1④), we can save 215 MB of data scanning without incurring the expensive re-partitioning required to update R_1 for query shifts.

Overall, there are three primary challenges. The first challenge arises from the absence of non-numeric-type predicates, which alters the candidate split condition set for partitioning and consequently impacts the order in which the tablespace is split. Hence, *how to fully incorporate the complete predicate conditions in the partition construction process* (**C1**) is crucial. The second challenge involves the dynamic nature of collected workloads; *it is essential to predict query distribution changes accurately to create robust partitions* (**C2**) that mitigate the need for frequent re-partitioning. The third challenge, a common objective in previous studies, is *refining predicate-based tablespace partitioning* (**C3**) to improve query performance.

Our proposed solution. We propose a **L**earned **R**obust **P**artitioning system (LRP) that utilizes historical query logs to generate robust data layouts. Firstly, LRP identifies queries with similar characteristics submitted at distinct times, then designs complete data encoding schemes and a logical tree structure to embed these raw queries into structured query vectors. This vector representation incorporates various types of predicate conditions and their logical relationships (addressing **C1**). Secondly, LRP employs two neural networks, multi-layer perceptron (MLP) [16] and long short-term memory (LSTM) [17], selected based on the temporal features of the queries, to predict incoming similarity queries. LRP also introduces a novel loss function that replaces the traditional Mean Squared Error (MSE) with a partition semantic correlation error to improve prediction accuracy (addressing **C2**). Thirdly, LRP decodes the predicted query vectors into candidate predicates and utilizes a beam search algorithm to determine the locally optimal predicate condition for each tablespace split. Furthermore, LRP introduces an effective data redundancy mechanism that replicates frequently accessed data to minimize query access contention at each partition (addressing **C3**). To enhance the robustness of fixed layouts against exploratory queries, LRP employs a data-driven KD-Tree [4] to partition the remaining tablespace after predicate-based splitting.

It is important to note that, similar to QdTree [8] and PAW [9], our approach does not directly optimize multi-table queries, such as their join phases. Instead, it indirectly optimizes them by reducing the number of blocks required for joins.

Contributions. In summary, we make the following contributions:

- We propose multiple data and query encoding schemes to capture the often-overlooked access features of non-numeric columns. Then, we present a logical tree for the transformation of raw queries into vectors.
- We design two optional predictive networks, along with a loss function integrating partition semantics, to accurately predict changes in similarity queries, ensuring robust layout creation over these predictions.
- We employ beam search and KD-Tree strategies to find the optimal predicate allocation order for splitting a given tablespace. Moreover, we design a data redundancy policy that aims to minimize query access contention at a negligible storage cost.
- Through rigorous testing on benchmark datasets, we demonstrate that our method outperforms existing techniques in both performance and robustness.

2 Preliminaries

In this section, we introduce the relevant concepts of queries and partitioning, followed by defining the two main problems to be addressed. Next, we present the related work. Table 1 summarizes the necessary notations used in this paper.

Table 1 Table of notation definitions

Notation	Description
E	Relation table E with z columns $\{c_1,\ldots,c_z\}$.
$\mathcal{D}$ and $H_{\mathcal{D}}$	Table dataset and its domain representations.
l_i, u_i	Lower and upper bounds of ith domain in $H_{\mathcal{D}}$.
Q and H_Q	Set of queries $\{q_1,\ldots,q_n\}$ and their vector representations.
$l_i^{(q)}$, $u_i^{(q)}$	Lower and upper bounds of ith domain in the query vector H_q.
$\Delta(H_{q_i}, H_{q_j})$	Query distance between two query vectors H_{q_i} and H_{q_j}.
$\phi(H_{q_i}, H_{q_j})$	Overlapping area between two query vectors H_{q_i} and H_{q_j}.
Q_H, Q_P, Q_F	Sets of historical, predicted, and future similarity queries.
Q_E	Set of exploratory queries.
$Q_i \to Q_j$	One-to-one query mappings between two query sets Q_i and Q_j.
$\mathbb{P}$ and $\mathcal{P}$	Set of data layouts, and a single data layout containing n partitions $\{R_1,\ldots,R_n\}$.
T	A partition tree structure, where each node V_i correspond to its instantiated partition R_i.
$\mathbf{T}_{lg}$	A logical predicate tree, composed of logical nodes $\mathbf{V}_r$ and predicate nodes $\mathbf{V}_p$.
$\mathcal{L}_{mse}$, $\mathcal{L}_{wd}$	Mean squared error; new loss function that integrates MSE and partition correlation error.
$\mathbf{S_{ss}}$, $\mathbf{S_P}$, $\mathbf{S_M}$	Candidate split set; predicate condition-based split set; median condition-based split set.

2.1 Similarity queries

The DBMS optimizer generates a query plan to execute a user query efficiently. Within this plan, filter operations are typically executed first, determining the satisfied data rows through predicate pushdown. Given the complete predicates extracted from a query q and a table dataset $\mathcal{D}$ with z columns $c_1,...,c_z$, we can characterize its query feature as the column domains for the satisfied rows, represented in vector form $H_q \in \mathbb{R}^{2z}$, i.e.,

$$H_q = \left[l_i^{(q)}, u_i^{(q)}\right]_{i=1}^{z},$$

where $l_i^{(q)}, u_i^{(q)}$ denote the lower and upper bounds of the ith column domain for the queried data by q. Similarly, we represent the domains of the entire table as $H_{\mathcal{D}} = [l_i, u_i]_{i=0}^{z} \in \mathbb{R}^{2z}$, where l_i, u_i denote the lower and upper bounds of the ith column data.

In real-world scenarios, many new and old queries often share the same filter columns and similar predicate conditions [9]. This leads to certain rows being repeatedly accessed by user queries, as illustrated by the red dotted boxes in Fig. 1③. We define two optional metrics to measure the similarity feature between queries q_1, q_2:

1) The first optional similarity metric is the overlapping data area $\phi(H_{q_1}, H_{q_2})$ accessed by the two queries, which can be measured by calculating the size (in bytes) of the set of co-accessed rows.

$$\phi\left(H_{q_1}, H_{q_2}\right) = \sum_{\mathcal{D}[i]\in\mathcal{D}} \forall j : \left(l_j^{(q_1)} \leqslant \mathcal{D}[i,j] \leqslant u_j^{(q_1)}\right) \wedge \left(l_j^{(q_2)} \leqslant \mathcal{D}[i,j] \leqslant u_j^{(q_2)}\right)$$

where the ith row in $\mathcal{D}$ is denoted as $\mathcal{D}[i]$, with a length of $|D[i]|$, and $\mathcal{D}[i,j]$ as the jth column value of $\mathcal{D}[i]$.

2) The second similarity metric evaluates the access distance, denoted as $\Delta(H_{q_1}, H_{q_2})$, between the two queries on each column dimension. This is calculated by finding the maximum difference between column boundaries.

$$\Delta\left(H_{q_1}, H_{q_2}\right) = \frac{\max_{i=1}^{z}\left(\left|u_i^{(q_1)} - u_i^{(q_2)}\right| + \left|l_i^{(q_1)} - l_i^{(q_2)}\right|\right)}{2}.$$

Then we can define how to identify queries that satisfy different levels of similarity.

Definition 1 (δ-similar queries). Let δ be a distance threshold. Consider historical load ($\mathbb{Q}_H$) and future load ($\mathbb{Q}_F$) with distinct submission timestamps. If we can find two sets Q_H and Q_F, for any $q_h \in Q_H \subseteq \mathbb{Q}_H$, there always exists a mapping $q_h \to q_f (q_f \in Q_F \subseteq \mathbb{Q}_F)$, such as $\max\left(\frac{\Delta(H_{q_h},H_{q_f})}{\Delta(H_{q_h})}, \frac{\Delta(H_{q_h},H_{q_f})}{\Delta(H_{q_f})}\right) \leqslant \delta$ or $\min\left(\frac{\phi(H_{q_h},H_{q_f})}{\phi(H_{q_h})}, \frac{\phi(H_{q_h},H_{q_f})}{\phi(H_{q_f})}\right) \geqslant 1-\delta$, then we deem them as δ-similar. The remaining exploratory queries can be obtained as $Q_E = \mathbb{Q}_F - Q_F$. Here, the computation method of $\Delta(H_q)$ and $\phi(H_q)$ is defined as follows:

$$\Delta\left(H_q\right) = \max_{1\leqslant i\leqslant z}\left|u_i^{(q)} - l_i^{(q)}\right|,$$

$$\phi\left(H_q\right) = \sum_{\mathcal{D}[i]\in\mathcal{D}} \forall_{1\leqslant j\leqslant z} : \left(l_j^{(q)} \leqslant \mathcal{D}[i,j] \leqslant u_j^{(q)}\right).$$

2.2 Horizontal partitioning layout

A query-aware data layout $\mathcal{P}$ consists of disjoint partitions $(R_1,...,R_m)$ created from query samples $Q(q_1,...,q_n)$. Each partition R_i is materialized as block files with sizes limited to $[b_{\min}, 2b_{\min}]$ ($b_{\min}$ = 64 MB in HDFS [18]). The I/O cost of processing query q over $\mathcal{P}$ is evaluated as the total size of accessed partitions, denoted as $C(\mathcal{P}, q)$. To build a data layout, recent works [8,9,12–14,19] propose a partition tree structure, similar to an index tree, which has been proven effective in guiding data allocation. We give its definition as follows:

Definition 2 (Partition index tree). The partition index tree (T) acts as a router, allocating data to specific partitions. Creating T starts with a root node covering the entire table space, and we select a feasible predicate to split it into

multiple child nodes. Each leaf node is then processed sequentially until no more child nodes can be generated. In T, all nodes maintain metadata to guide query skipping and support data routing. Each leaf node (V_i) is ultimately materialized as a single partition (R_i), with a node size $|V_i|$, which is determined by the number of rows it contains. This node size ranges from $[V_{\min}, V_{\max}]$, i.e.,

$$|V_i| \in \left[\frac{b_{\min}}{\max(|\mathcal{D}[1]|\cdots|\mathcal{D}[n]|)}, \frac{2\times b_{\min}}{\max(|\mathcal{D}[1]|\cdots|\mathcal{D}[n]|)}\right].$$

Next, we formally outline the two main problems to be addressed in this paper.

Problem 1 (Robust partitioning). Given a training set $Q_H \to Q_F$ and a test set of $\overline{Q_H} \to \overline{Q_F}$, each mapping satisfies δ-similarity, we consider two cases: (1) Q_H is evenly divided into an ordered sequence based on query submission time; (2) Submission time is not recorded, rendering all queries unordered. Our goal is to find an optimal model $\mathcal{M}'$ that fits the $Q_H \to Q_F$ mapping, enabling it to generate a predicted load $\overline{Q_P}$ based on $\overline{Q_H}$. This $\overline{Q_P}$ can maximize cumulative co-accessed data area with $\overline{Q_F}$ while ensuring that a robust data layout built upon $\overline{Q_P}$ has the smallest query cost difference versus the layout generated by $\overline{Q_F}$, i.e.,

$$\mathcal{M}' = \text{argmax}_{\mathcal{M}}\phi\left(\mathcal{M}(\overline{Q_H}|Q_{H\to F}), \overline{Q_F}\right),$$
$$\text{s.t. } C\left(\mathcal{P}_{[\mathcal{M}'(\overline{Q_H})]}, \overline{Q_F}\right) - C\left(\mathcal{P}_{[\overline{Q_F}]}, \overline{Q_F}\right) \text{ is minimized.}$$

Problem 2 (Optimal layout). Given Q_P, Q_F, Q_E, and n relation tables, this problem asks for constructing an individual partition tree T_i for the ith table, followed by routing its table data $\mathcal{D}_i$ on T_i to create a robust data layout $\mathcal{P}_i$, such that the total I/O cost of executing $Q_E + Q_F$ (i.e., Q_{E+F}) over all layouts $\mathbb{P}\{\mathcal{P}_1, ..., \mathcal{P}_n\}$ is minimized. This process is formulated as follows:

$$\mathcal{P}_i \Leftarrow \text{Route}\,(T_i(Q_P), \mathcal{D}_i),\ 1 \leqslant i \leqslant n,$$
$$\mathbb{P}' = \text{argmin}_{\mathbb{P}} \sum_{i=1, \mathcal{P}_i \in \mathbb{P}}^{n} C(\mathcal{P}_i, Q_{E+F}).$$

2.3 Related work

Data-driven partitioning. In most popular database products such as TiDB [20], ClickHouse [21], and Snowflake [22], partitioning rules based on data distribution are still recommended as the preferred option. This method is suitable for data with prior statistics but requires careful selection of partition boundaries, including hash and range partitioning [4], along with partition maintenance structures like SMA [23], Zone Maps [24]. They exhibit low sensitivity and high adaptability to rapidly changing load scenarios, but with relatively lower performance. Conversely, our LRP ensures both superior performance and minimized partition updates to accommodate dynamic workloads.

Query-driven partitioning. There are two approaches for creating query-driven partitioning rules. Classifier-based methods [7,10] extract representative predicate conditions from historical loads, cluster tuples based on the similarity of their satisfying predicates, and subsequently compute classification features for each cluster. Tree-based methods [8,9,12–14,19] incrementally build a partition tree by selecting numeric-type predicates with the maximum query skipping benefit as the split condition at each tree expansion stage. Jigsaw [25] provides optimal data skipping with tetris-shaped partitions managed by logical segments, yet it requires multiple hash tables for frequent tuple reconstruction. Unlike prior studies, LRP is the first tree partitioner to identify additional non-numeric predicates and implement a refined beam search policy for selecting better predicate split combinations.

Adaptive partitioning. To cope with dynamic loads, [11,13,14,19] manage re-partitioning based on the filling of a predefined-length query window, while [8] relies on periodically monitoring distribution differences between new and old data. Learning-based methods [26–30] use RNN/RL-style models to monitor and predict query distribution changes, aiding in calculating potential re-partitioning benefits. Another solution is to try creating a robust layout to reduce the re-partitioning frequency. PAW [9] introduces a similar-load scenario and search for a fixed expansion value to fit query changes, which cannot always guarantee high layout quality. LRP addresses this by training prediction networks that incorporate partition evaluation factors into the loss function.

3 LRP overview

3.1 LRP framework

Overview. Figure 2 shows the architecture of LRP framework with a three-stage task.

Step 1 — Query feature embedding. Initially, LRP reads raw queries from real system logs and encodes non-numeric column data using predefined coding rules based on the table schema. The non-numeric columns in the raw table domain are then updated based on the encoding scheme of the corresponding columns. Next LRP extracts complete logical predicate trees (defined in Subsection 4.2) from queries, encodes them, and serializes them into structured query vectors by a level-by-level logical domain computation.

Step 2 — Similarity load prediction. Depending on whether the submission time is known, these query vectors are classified as ordered or unordered. Accordingly, LRP adopts the corresponding network, LSTM for ordered and MLP for unordered, to predict changes in query vectors.

Steps 3–5 — Data layout construction. To create the data layout instance, LRP decodes predicted vectors into logical predicates, followed by a step-by-step construction of the partition tree (T). Starting from the root node, LRP generates the candidate predicate split set for the current node and applies a beam search-based split policy to select the optimal split condition for creating child nodes. The T is incrementally expanded until all leaf nodes meet the partition size requirements. LRP then further reallocates leaf node data by replicating frequently accessed data across multiple partitions for greater data skipping. All table data is routed by

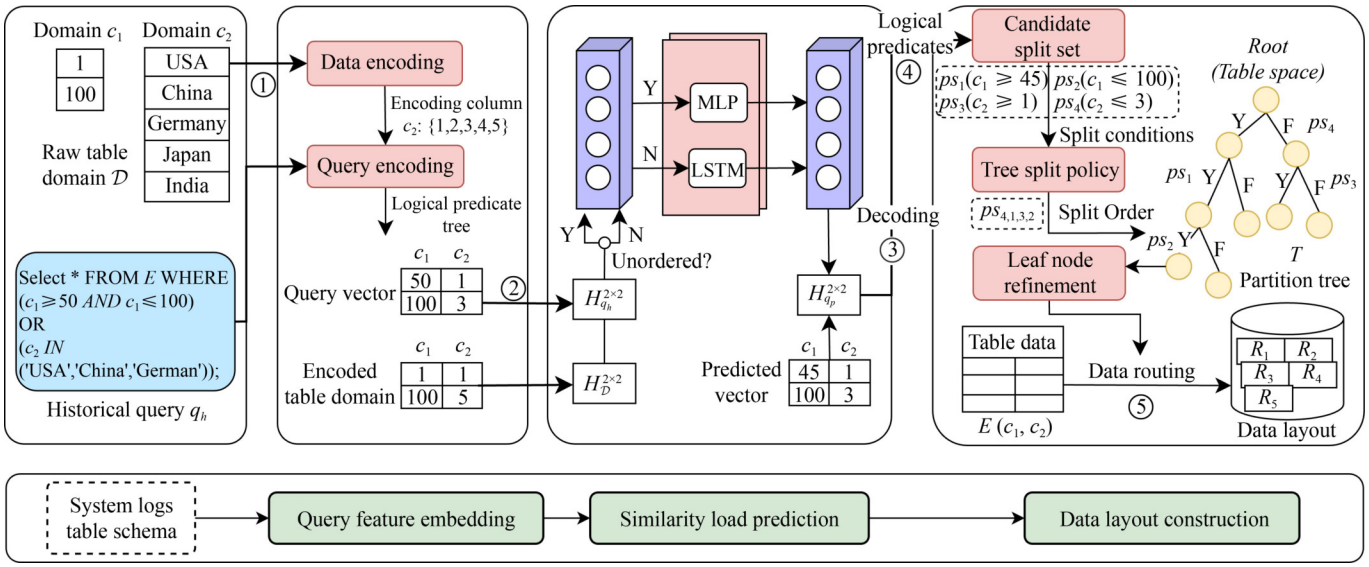

Fig. 2 The overall framework of LRP system

T to leaf nodes, which are subsequently materialized into specified partition files.

Query processing. When processing a new query, the query optimizer uses the integrated T to identify required partitions, developing an appropriate query plan and forwarding it to the query executor to obtain the result set.

Example 1 Given a query q_h and the domain for table $E(c_1, c_2)$ in Fig. 2, where c_1 (numeric column) and c_2 (categorical column) respectively use min-max values and distinct values as their raw domain representations. First, LRP will use a dictionary table to encode the country column c_2 because it is an enumeration type, i.e., {1:'USA', 2:'China', 3:'Germany', 4:'Japan', 5:'India'}, to obtain a new numeric table domain, i.e., $[[1,100],[1,5]]$. Subsequently, all predicates are extracted from q_h and those related to the encoded columns are processed; for instance, 'c_2 *in* $(USA, China, Germany)$' is converted to '$1 \leqslant c_2 \leqslant 3$'. Through logical computation among predicates, LRP obtains the vector representation of q_h as $H_{q_h} = [[50,100],[1,3]]$. Since individual queries lack temporal order, LRP will select the MLP network to generate the vector prediction $H_{q_p} = [[45,100],[1,3]]$, which is then decoded into four predicates (ps_1, ps_2, ps_3, ps_4). Employing a split order $\{ps_4 \Rightarrow ps_1 \Rightarrow ps_3 \Rightarrow ps_2\}$ recommended by the tree split policy, LRP constructs a four-layer partition tree (T) comprising five leaf nodes. These leaf nodes are further refined by replicating high-frequency hot data among partitions. Finally, by routing the entire table data through T, five partition files $(R_1, R_2, R_3, R_4, R_5)$ are created as the final data layout.

3.2 Role of LRP

LRP primarily targets boosting partitioning performance while emphasizing robustness under dynamic loads. It is applicable in the following scenarios:

- **Static optimal layout.** By utilizing more comprehensive predicate features and refining the table space partitioning strategy, LRP functions as an efficient static partitioning algorithm when the load prediction module is disabled.
- **Complementary to re-partitioning.** Although LRP creates robust layouts, it does not eliminate the need for re-partitioning operations to maintain system stability during significant performance declines (e.g., an increase in exploratory queries). Instead, LRP aims to reduce the frequency of re-partitioning and can also work collaboratively with existing re-partitioning methodologies [13,31] rather than conflicting with them.

4 Data encoding and query vector extraction

In this section, we fully leverage non-numeric column data's access properties to generate complete logical predicates as partitioning features, which can then be embedded into unified query vectors for subsequent load prediction.

4.1 Data encoding phase

There are various types of non-numeric columns in a table schema, including date strings, enumeration data, and other fixed-length and variable-length texts. We design multiple encoding functions to convert them into numeric data, facilitating the computation of their column domains based on the encoded data distribution.

Encoding strategy. As shown in Fig. 3①, for a non-numeric column c:

1) If c is of date type, a date formatting function is applied, i.e., $\mathcal{D}[*,c] = \text{TimeStamp}(\mathcal{D}[*,c])$.

2) If c is of enumeration type or if $\frac{\#\text{distinct}(\mathcal{D}[*,c])}{\#\mathcal{D}[*,c]}$ is less than a constant ϑ_C, we directly utilize a dictionary encoding strategy by creating a finite-sized dictionary table $Dict_c$ for all distinct values $(v^0, ..., v^n)$ in column c, i.e., $Dict_c[v^0] = 0, \ldots, Dict_c[v^n] = n$.

3) If c has a maximum text length less than the predefined constant ϑ_L (classified as Complex Type-1), a trie-based

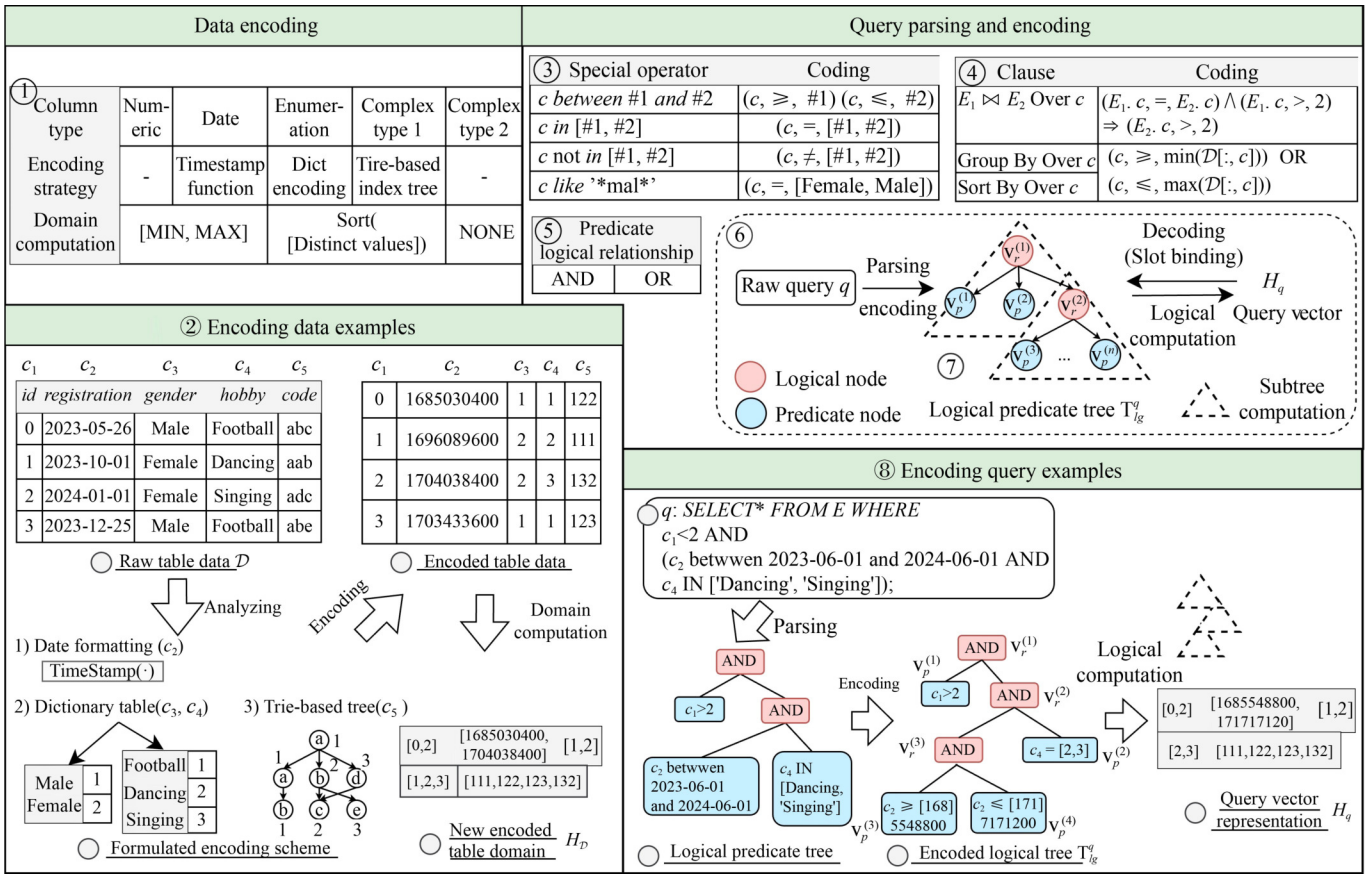

Fig. 3 Data and query encoding scheme: This illustration includes a detailed example demonstrating the transformation of various datasets and queries with different predicate types into a unified vector representation

index tree [32] is constructed, which is suitable for long and non-categorical strings. The encoding process starts at the root node, identifying the encoding value of the ith letter in a given string from nodes at depth i, ultimately generating a sequence as its encoding key.

4) Otherwise (classified as Complex Type-2), no encoding is performed.

We determine suitable values for the above constants ($\vartheta_C = 0.1$, $\vartheta_L = 15$) through empirical judgment and experimental validation.

Encoding Key Allocation Order. For each encoding column, we avoid allocating continuous dictionary keys to column values in alphabetical order. Instead, our goal is to ensure that the most frequently co-accessed texts are allocated continuous keys, facilitating subsequent domain simplification of non-numeric columns. To achieve this, we first record the co-occurrence frequency of column values referenced by queries, sort them in descending order, and then determine the allocation priority for each column value accordingly. For example, in Fig. 2, if the countries 'USA' and 'Germany' are frequently queried together, we modify the encoding scheme of ('USA', 'China', 'Germany') from (1, 2, 3) to (1, 3, 2).

Domain computation. We then compute the column domains for the encoded table dataset, which supports non-numeric column-based node splitting in partition trees (see Section 6).

1) For numeric and date columns (c_i), we can directly use the min/max functions to compute their domain boundaries, i.e., $[l_{c_i}, u_{c_i}] \Rightarrow [\min(\mathcal{D}[*, c_i]), \max(\mathcal{D}[*, c_i])]$.

2) For encoded text columns (c_j), we adopt the distinct encoding keys as their list-type domain representation, i.e., $[l_{c_j}, u_{c_j}] \Rightarrow \text{sorted}(\text{distinct}(\mathcal{D}[*, c_j]))$.

3) For unencoded text columns (c_k), whose data distribution cannot be effectively quantified, we use 'None' as the domain identifier, i.e., $[l_{c_k}, u_{c_k}] \Rightarrow [\text{None}, \text{None}]$.

Example 2 Fig. 3② illustrates how a simple raw table with 4 rows and 5 columns is encoded. For each non-numeric column of $c_2, \ldots, c_5$, we employ distinct encoding schemes. This includes converting the date column (c_2) using a general timestamp function, constructing dictionary tables for the enumeration columns (c_3,c_4), and building a trie-based tree for the irregular text column (c_5). After encoding, the table data consists only of pure numerical values. We then utilize the three domain computation rules to represent the domains for each column, forming the encoded numeric table domain.

4.2 Query encoding phase

Predicate formatting. Given any predicate, we represent it as a triplet (μ, op, ν), consisting of the column (μ), operator (op), and condition value (ν). To our knowledge, previous studies [6,8–10,14] have predominantly focused on predicate conditions involving comparison operators (e.g., '⩾', '⩽', '=', and '≠') associated with numeric columns. To support more predicate types, we extract and format all predicates in a query from three parts: special operators, query clauses, and logical

relationships.

1) Special operators. To utilize all operators as the partition tree split condition, certain special operators, including set operators and pattern matching operators, must be formatted into unified comparison operators. As shown in Fig. 3③, '*c between* #1 *and* #2' is converted to two predicates: $(c,\geqslant,\#1)$ and $(c,\leqslant,\#2)$; '*c in* [#1,#2]' and '*c not in* [#1,#2]' are easily converted to $(c,=,[\#1,\#2])$ and $(c,\neq,[\#1,\#2])$; if c is a common enumeration column (e.g., gender), '*c like* *mal*' is converted to $(c,=,[\text{Female},\text{Male}])$ by identifying all column values that satisfy the given wildcard.

2) Query clauses. Fig. 3④ designs the formatting rules for three common clauses. In cases involving a *Join* operation (e.g., two tables $E_1 \bowtie E_2$) on the c column, where they share a common join key c, we can distribute all predicate conditions related to c to each table to identify potential predicates. For *Group By* and *Order By* operations on the c column, the column data needs to participate in comparison operators such as '$\leqslant$' and '$\geqslant$' to complete the grouping and sorting of c.

3) Logical relationship (shown in Fig. 3⑤). We consider the logical relationships (AND, OR) among single predicates or predicate groups, which is crucial for accurately encoding query vectors and consequently facilitating the search for the queried leaf nodes in partition trees.

Query embedding. For any query, extracting all queried rows from the sampling table so as to calculate its vector representation is expensive; instead, we propose a lightweight query embedding strategy that allows obtaining its vector representation without query pre-execution. This strategy consists of two steps (see Fig. 3⑥):

1)Query Encoding. Using extracted predicates and logical relationships, we can construct a logical tree structure ($\mathbf{T}_{lg}$) (see Fig. 3⑦) composed of logical nodes ($\mathbf{V}_r$) and predicate nodes ($\mathbf{V}_p$). Child nodes (including $\mathbf{V}_p$ and sub-trees) sharing the same parent node ($\mathbf{V}_r$) imply that they satisfy the corresponding logical relationship (AND, OR). Subsequently, all condition values related to non-numeric columns in $\mathbf{V}_p$ are encoded using predefined data encoding schemes.

2) Logical computation. We leverage the encoded $\mathbf{T}_{lg}$ for efficient vector calculations, which are conducted via a bottom-up traversal on $\mathbf{T}_{lg}$, as depicted in Fig. 3⑦. When traversing to each node level, all sibling nodes along with their parent node are grouped into a subtree, and then column domains are computed by performing logical operations among predicates. The process begins with the lowest level subtree (depicted by dashed triangles), sequentially visiting subtrees at higher levels, and cumulatively updating domain values until reaching the root node.

Example 3 As shown in Fig. 3⑧, to embed a query q, we first extract a 3-level logical tree $\mathbf{T}_{lg}^{q}$ and then format these predicates with non-numeric condition values or non-comparison operators using the encoding structures (Fig. 3①) and predicate conversion rules (Figs. 3③ and 3④). Next, logical calculations are performed over $\mathbf{T}_{lg}^{q}$ to obtain the final query vector. Specifically, the subtree $\mathbf{T}_{lg}^{q}(\mathbf{V}_r^{(3)}:\mathbf{V}_p^{(3:4)})$ at the bottom is executed first in terms of the column c_2, i.e., $(c_2,\geqslant,1.68\times10^9)\wedge(c_2,\leqslant,1.71\times10^9)\Rightarrow c_2[1.68\times10^9,1.71\times10^9]$. Subsequently, the subtree $\mathbf{T}_{lg}^{q}(\mathbf{V}_r^{(2:3)}:\mathbf{V}_p^{(2:4)})$ is executed over c_4. Since c_4 is not referenced by the left subtree $\mathbf{T}_{lg}^{q}(\mathbf{V}_r^{(3)}:\mathbf{V}_p^{(3:4)})$, we can select the c_4's table domain (i.e., $[1,2,3]$) as its replacement, then $(c_4,=,[2,3])\wedge(c_4,=,[1,2,3])\Rightarrow c_4[2,3]$. Next, we perform $\mathbf{T}_{lg}^{q}(\mathbf{V}_r^{(1:2)}:\mathbf{V}_p^{(1)})$ over c_1, i.e., $(c_1,<,2)\wedge c_1[0,2]\Rightarrow c_1[0,2)$. For c_3 and c_5, since they are not involved in any predicates, their domains are also replaced by the corresponding table domains. Finally, we concatenate all domains to generate the final query vector.

Query vector decoding. To create partitions over these vector predictions H_{Q_P}, we need to pre-decode them into predicate conditions. Considering the one-to-one mapping relationship between H_{Q_H} and H_{Q_P} ($H_{Q_H}\rightarrow H_{Q_P}$), we can employ a slot-binding strategy before predicting queries. We first identify boundary values in H_{Q_H} with equivalent relationships to its corresponding predicate nodes of $\mathbf{T}_{lg}$, performing slot binding. When decoding vectors, we only fill domain values of H_{Q_P} into these pre-saved slots to rewrite logical predicates.

5 Predicting similarity-load

In this section, we initially define a loss function to reflect the prediction effectiveness on robust partitioning. Subsequently, we employ two predictive networks to learn the similarity changes of given mappings ($H_{Q_H}\rightarrow H_{Q_F}$), generating H_{Q_P}.

5.1 Prediction loss incorporating partition semantics

As described in Subsection 4.1, there are three distinct domain types. To achieve a unified model input, we must process the list-type and None-type domains of non-numeric columns (c_i) to match the dimensionality of the min–max domain. This ensures consistency in the dimensions of all domains within the query vector.

Vector preprocessing. Before defining the loss function, we process query vectors as follows:

1) If $H_{Q_H}^{c_i}$ is a list-type domain, intuitively, predicting its changes is challenging due to its larger number of elements compared to the min–max domain. In contrast, predicting only its domain boundary changes can significantly improve accuracy, with minimal errors in the query predicate features of column c_i. As shown in Eq. (1), we use the min–max values of all dictionary keys in $H_{Q_H}^{c_i}$ and consider the minimum interval between keys as boundary values to continuous the c_i's discrete domain. This is because these discrete domains are typically derived from query predicates, comprising co-accessed column values, and are more likely to be allocated continuous key values based on the key allocation order mentioned in Subsection 4.1.

$$H_{Q_H}^{c_i}\Rightarrow\left[k^0,...,k^n\right]\Rightarrow[\min(k^{0..n})-0.5\times|k^1-k^0|,\ \max(k^{0..n})+0.5\times|k^1-k^0|]. \tag{1}$$

where $k^0,...,k^n$ are obtained keys after encoding c_i.

2) If $H_{Q_H}^{c_i}$ is a None-type domain, as it does not contain any specific knowledge, we apply a simple numeric encoding, i.e., $H_{Q_H}^{c_i}\Rightarrow[0,0]$.

Loss function design. We do not directly use the mean square error (MSE) between H_{Q_P} and H_{Q_F} as the loss function, as MSE implies that the two query vector values are closely matched, but this does not necessarily mean that the partitions constructed on them have similar quality. Instead, we incorporate both the MSE and partition semantic information to accurately estimate potential partition performance over H_{Q_F}. Specifically, LRP tailors a function $\mathcal{L}_{wd}$ for reliable loss estimation by using a weighted deviation to account for the impact of different dimension deviations between H_{Q_P} and H_{Q_F} on partition quality.

Figure 4 shows all possible intersection positions of H_{q_f} (white rectangle) and H_{q_p} (green rectangle) in a 2D table (i.e., $z=2$), which is categorized by the number of edges (N_{eg}) in H_{q_f} not covered by H_{q_p}. Intuitively, we formulate it as follows:

$$N_{eg} = 2z - \sum_{i=1}^{z}\left[f^o\left(l_i^{(q_f)}, l_i^{(q_p)}\right) + f^o\left(u_i^{(q_p)}, u_i^{(q_f)}\right)\right],$$

where the function $f^o(x,y) = \begin{cases} 1, & x \geqslant y \\ 0, & \text{otherwise} \end{cases}$.

As N_{eg} increases from 0 to 4, the number of queried areas outside of H_{q_p} increases. Consequently, each H_{q_f} has a higher probability of accessing an extra number of partitions in the H_{q_p}-based data layout, resulting in a higher loss value. Moreover, a smaller intersection area between H_{q_f} and H_{q_p} should also correspond to a higher loss value. Based on these two observations, $\mathcal{L}_{wd}$ is primarily composed of two parts of losses, i.e.,

$$\begin{aligned}\mathcal{L}_{wd}\left(H_{q_f}, H_{q_p}\right) &= \mathcal{L}_{wd}^{(1)}\left(H_{q_f}, H_{q_p}\right) + \mathcal{L}_{wd}^{(2)}\left(H_{q_f}, H_{q_p}\right) \\ &= \frac{\phi(H_{q_p}) - \phi(H_{q_p}, H_{q_f})}{\phi(H_{q_p})} \\ &\quad + \overline{w_2}\frac{\phi(H_{q_f}) - \phi(H_{q_f}, H_{q_p})}{\phi(H_{q_f})},\end{aligned}$$

where $\mathcal{L}_{wd}^{(1)}\left(H_{q_f}, H_{q_p}\right)$ indicates the prediction inaccuracy of q_p, affecting the created partition size. Conversely, $\mathcal{L}_{wd}^{(2)}\left(H_{q_f}, H_{q_p}\right)$ represents the uncovered area of H_{q_f}, determining N_{eg} as well as the number of future accessed partitions. Intuitively, $\mathcal{L}_{wd}^{(2)}$ should be assigned a higher weight, as increasing the number of accessed partitions has a greater impact on the overall cost than the size of accessed partitions. Thus, we assign weights to them as $1:\overline{w_2}$ ($\overline{w_2} > 1$). The weight $\overline{w_2}$ is dynamic, influenced by the increase in N_{eg} and the ratio $\gamma(q_p)$, which is the query density of q_p (denoted as ρ_{q_p}) relative to the table's average query density (denoted as $\rho_{\mathcal{D}}$). A higher $\gamma(q_p)$ typically indicates that, under the same N_{eg}, more partitions will be accessed in the future. Therefore, we represent $\overline{w_2}$ as $1 + \lambda \times \gamma(q_p) \times N_{eg}$, where λ is a hyper-parameter and $\gamma\left(q_p\right)$ is calculated as follows:

$$\gamma\left(q_p\right) = \frac{\rho_{q_p}}{\rho_{\mathcal{D}}} = \frac{\sum_{q_i \in Q_F}\phi\left(H_{q_p}, H_{q_i}\right)}{\prod_{j=1}^{z}\left(u_j^{(q_p)} - l_j^{(q_p)}\right)} \times \left(\frac{\sum_{q_i \in Q_F}\phi\left(H_{q_i}\right)}{\prod_{j=1}^{z}\left(u_j - l_j\right)}\right)^{-1}.$$

In this formula, we represent the ρ_{q_p} by using the proportion of the cumulative query access area within the q_p area. The $\rho_{\mathcal{D}}$ is estimated by the proportion of the cumulative access area of all queries to the total table area.

5.2 Design of load predictor

In this subsection, we first categorize historical query vectors based on their temporal information, and then design suitable prediction networks for each category.

Load classification. We identify query submission times from logs. If H_{Q_H} is distributed across continuous time intervals, and there exists a query sequence at each interval that satisfies δ-similarity with adjacent sequences, it is referred to as ordered load; otherwise, it is unordered, i.e.,

$$H_{Q_H} = \begin{cases} \left[H_{Q_h}^{(t_1)}, \ldots, H_{Q_h}^{(t_m)}\right], & H_{Q_H} \text{ is ordered,} \\ \left[H_{q_h}^{(1)}, \ldots, H_{q_h}^{(m)}\right], & \text{otherwise,} \end{cases}$$

where $H_{Q_h}^{(t_i)} = \left\{H_{q_h}^{(t_i,j)} | j = 1, \ldots, n\right\}$, $i = 1, \ldots, m$.

Prediction dimensionality reduction. To facilitate model convergence, we shift from predicting the changes of each domain boundary in the query to predicting the expansion ratio τ_i ($i = 1, ..., z$) for each domain radius. This reduces the model output dimension from $2z$ to z (50% ⇓). During model training, this shift will lead to an increase in $\mathcal{L}_{wd}^{(1)}$ but a decrease in N_{eg}. Nevertheless, since N_{eg} plays a dominant role in reducing the $\mathcal{L}_{wd}^{(2)}$ value, while $\mathcal{L}_{wd}^{(1)}$ has a smaller weight coefficient, the impact on the final $\mathcal{L}_{wd}$ value is limited. Overall, this shift, by reducing the prediction dimensionality, also stabilizes the decrease of N_{eg}, thereby promoting a greater decrease in total loss on the test set.

Model structure. Next, we introduce the load predictor, as showcased in Fig. 5, to fit the query changes in training samples $H_{Q_H} \rightarrow H_{Q_F}$, taking H_{Q_H} as input and generating the output H_{Q_P}.

[**Step 1** Normalization process.] Due to significant differences in the domain ranges across different columns, we first normalize the structured vector H_{q_h} to the 0–1 intervals based on the table domains $H_{\mathcal{D}}$, as follows:

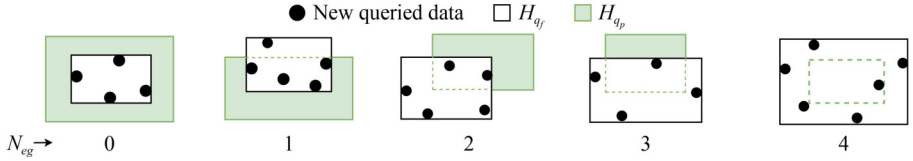

Fig. 4 Five distinct intersection cases of H_{q_f} and H_{q_p} in a 2D tablespace. The black dot denotes the data queried by H_{q_f}

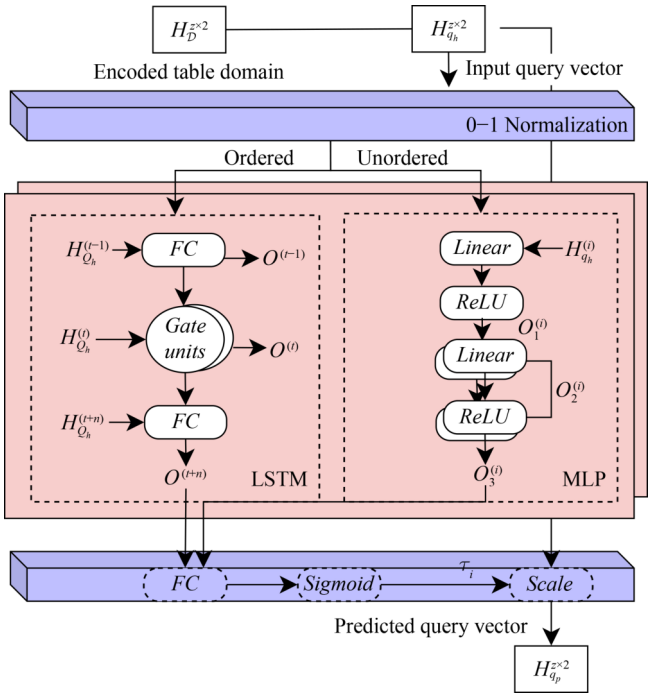

Fig. 5 Network architecture design for load predictor

$$H_{q_h} = \mathrm{Norm}(H_{q_h}, H_{\mathcal{D}}) = \bigcup_{i=1}^{z} \left[\frac{l_i^{(q_h)} - l_i}{u_i - l_i}, \frac{u_i^{(q_h)} - l_i}{u_i - l_i} \right].$$

[**Step 2** Estimation process.] Next, for ordered loads, to learn dependency relationships among continuous sequences $\left[H_{Q_h}^{(t-1)}, H_{Q_h}^{(t)}, H_{Q_h}^{(t+1)}\right]$, we utilize an LSTM network [17] comprising multiple hidden layers and memory cells, each containing input, forget, and output gate. This enables the network to adaptively store and update long-term dependency information regarding query domain changes. For unordered loads, given the one-to-one mappings $H_{q_h}^{(i)} \to H_{q_f}^{(i)}$, we directly employ a simple MLP network [16] composed of stacked linear layers with a ReLU activation function to fit the query changing trends. Finally, we use a fully-connected (FC) layer to map the obtained prediction features ($O^{(t+n)}$ or $O_3^{(i)}$) to an expansion ratio τ and apply a Sigmoid activation function to transform the output range to $[0, 1]$. This is then input into the scale-layer to extend H_{q_h}, yielding

$$H_{q_p} = \mathrm{Scale}\left(H_{q_h}, \tau\right) = \bigcup_{i=1}^{z} \left[l_i^{(q_h)} - r_i\tau_i,\ u_i^{(q_h)} + r_i\tau_i \right],$$

where $r_i = \frac{1}{2}\left[l_i^{(q_h)} + u_i^{(q_h)}\right]$.

[**Step 3** Model training.] The model is trained to aggregate the features of the query vectors with non-linear transformations and iteratively update the network weights Θ to minimize the loss $\mathcal{L}_{wd}$ between H_{Q_F} and H_{Q_P}. We use L2 regularization [33] to prevent overfitting. Assuming the optimal network weights are $\bar{\Theta}$, we can obtain $H_{q_p}^{(i)}$ using $f_{\mathrm{MLP}}\left(H_{q_h}^{(i)} \in H_{Q_H}; \bar{\Theta}\right)$ or $f_{\mathrm{LSTM}}\left(\left[H_{Q_h}^{(t,i)}\right]_{t=1}^{m} \in H_{Q_H}; \bar{\Theta}\right)$.

6 Robust partitioning layout construction

In this section, we first present the construction process of a partition tree (a.k.a., logical partition structure) based on obtained query predictions. This process is divided into three phases: parsing, expansion, and optimization. Subsequently, we describe how this tree is integrated into the DBMS.

6.1 Parsing phase: candidate split set generation

To construct a partition tree, we need to pre-generate a candidate split set (denoted as $\mathbf{S_{ss}}$) before each node split. Starting from the root node, we first obtain predicate set ($\mathbf{S_P}$) decoded from the vector predictions H_{Q_P}. If $\mathbf{S_P} = \varnothing$, inspired by KD-Tree [4], we proceed to derive the column median condition set ($\mathbf{S_M}$) from the node data distribution; otherwise, we set $\mathbf{S_M} = \varnothing$. The two sets are then merged as the split set for the current node, i.e., $\mathbf{S_{ss}} = \{ps \in \mathbf{S_P}\} \bigcup \{ms \in \mathbf{S_M}\}$, where ps denotes the predicate-based split and ms denotes the median-based split. At each split, only one element in $\mathbf{S_{ss}}$ is selected as the split condition. After splitting this node, we need to reallocate the remaining predicates for each child node, an operation denoted as $Reallocate(\mathbf{S_{ss}})$. Specifically, we filter the feasible predicates for the child nodes, checking whether each predicate can split a given child node into nodes that satisfy the size constraints, and then update the median conditions for each child node.

The use of split conditions. Given a node V_i and the split condition $s_j(\mu, op, \nu)$, we denote the split operation as $Split(V_i, s_j)$. If the split column μ is of (encoded) numeric type, we execute the expression '$\mu\ op\ \nu$' (where $op \in \{=, \neq, >, <, \leqslant, \geqslant\}$) to split the node data. Rows that satisfy the condition are directed to the left child node; otherwise, they are directed to the right. If μ is an unencoded text type, the expression '$\mu\ op\ \nu$' represents an invalid split and is treated as a special

Algorithm 1 Partition tree construction via the recursive function *BeamSearch* (*BS*)

Input: tree T and its decoded queryset $T.Q_P$, candidate number n_c, depth limit h_{max}.
Output: partition tree T, total query cost C_{total}.

```
CanSplit = 0;
T.root.S_ss = T.Q_P(S_M) + T.Q_P(S_P);
while CanSplit++ ≤ 0 do
    for V_i (size ∈ [V_min, V_max]) of T.leaves do
        V_i.S_ss ← Sort( B_skip(s_1) > B_skip(s_2),
                   ∀s_1, s_2 ∈ V_i.S_ss );
        C_min = +∞, s_min = ∅;
        for condition s_i of top-n_c items in S_ss do
            if h_max > 0 then
                C_split ← BS (T, Q_P, n_c, h_max-1);
            else
                C_split = |V_i.Q_P| × |V_i| − B_split(s_i);
            if C_split < C_min then
                C_min = C_split, s_min = s_i;
        if C_min < |V_i.Q_P+E| × |V_i| then
            V_l, V_r ← Split(V_i, s_min);
            V_l.S_ss, V_r.S_ss ← Reallocate(S_ss);
            CanSplit = 0;
return T and C_total = Σ_{q_i∈T.Q_P} C(T, q_i)
```

median split, i.e., we randomly split the node data and evenly distribute it to the two child nodes.

6.2 Expansion phase: beam-search tree split

During the entire tree extension, we aim to find an optimal split order by assigning appropriate split conditions to nodes at each depth, as the current depth's condition selection impacts both the next depth's decision and the overall split order of the tablespace.

Basic idea. We adopt a refined greedy policy, beam search (*BS*), which first explores local split condition combinations of multiple tree depths at each step, and then selects the split condition corresponding to the current depth from the combination with the highest skipping benefit $\mathcal{B}_{skip}$. Here, $\mathcal{B}_{skip}$ represents the reduction in query cost before and after splitting. This process involves two important parameters: (1) n_c represents the number of split condition combinations explored in each decision-making step; (2) h_{max} is the number of split conditions in each combination, i.e., the additional explored maximal tree depth at the current node.

Beam search-based tree split. Algorithm 1 provides the execution details of *BS*. We first traverse each available leaf node V_i (initially only the root node) of the partition tree T in turn (line 4). Then, we generate a dynamically changing $\mathbf{S_{ss}}$ during the node split process (lines 2,5,16) and sort $\mathbf{S_{ss}}$ by $\mathcal{B}_{skip}$ (line 5). Subsequently, the top-n_c items in $\mathbf{S_{ss}}$ are selected as the exploration conditions, allowing n_c distinct candidate split paths for V_i (lines 7–13). Each path recursively executes the same beam search subprocess (denoted as $BS^{(2)}$) to return a query cost (C_{split}) of accessing the current child nodes. Here, each $BS^{(2)}$ will continue to extend more secondary paths (i.e., $BS^{(3)}$) until $BS^{(h_{max})}$ is executed. Next, we choose the current split condition (s_{min}) from the optimal path among the n_c paths, splitting V_i into child nodes V_l and V_r, and updating the $\mathbf{S_{ss}}$ for them (lines 14–17). After splitting, we proceed to the next available leaf node and repeat this process. The algorithm terminates when no more leaf nodes can be split (line 3).

6.3 Optimization phase: node shape refinement via data replication

In horizontal partitioning, row data acts as the minimal construction unit of partition files. This can result in significant query access contention between leaf nodes, especially for dense queries, where different parts of each row are likely to be accessed by different queries. To mitigate the contention, LRP introduces data replication for frequently accessed rows, thereby creating super nodes, which are defined as follows:

Definition 3 (*Super Node*). Leaf nodes maintain distinct data domains to ensure each row is routed to only one partition. However, when a small batch of rows from one partition is copied into another, this rule is broken, and the modified partition becomes a redundant partition. We refer to its corresponding tree node as a super node, which preserves a primary domain and multiple secondary domains to route non-redundant and redundant data independently.

We denote the replication process of data from node V_i to V_j as $V_i \rightarrow V_j$, where V_j is the super node and is assigned an additional secondary domain to maintain the redundant data. When more data is replicated from V_i to V_j, any intersecting secondary domains are merged to save storage space for repeated redundant data. When searching for leaf nodes that match a given query, the primary domain continues to function as the main router. However, adjustments are made to super nodes by effectively utilizing secondary domains, thereby minimizing the number of reads required for leaf nodes.

Example 4 Given a two-dimensional table and a decoded query set $Q_P(q_{1\sim3})$, Fig. 6(a) displays a partition tree T is expanded using the split order $\{ps_1, ps_2, ms_4, ps_3, ms_5\}$, where $ps_{1\sim3}$ and $ms_{4\sim5}$ represent predicate and median splits extracted from Q_P and the node dataset, respectively. The final T contains six leaf nodes. Specifically in the tablespace, these nodes are instantiated six partitions $R_{5\sim8,10,11}$ (see Fig. 6(b)). Based on the primary domain of partitions, q_1 needs to access $\{R_8, R_{10}\}$. If we replicate the green-shaded data area S_1 from R_8 to R_{10} (i.e., $V_8 \rightarrow V_{10}$) and add the data domain of S_1 as the secondary domain of R_{10} (see Fig. 6(c)), then q_1 only needs to access R_{10} because q_1's scanning data in R_8 satisfies the secondary domain of R_{10}. Similarly, when creating $R_{10} \rightarrow R_6$, the access plan for q_2 can be changed from $\{R_6, R_{10}\}$ to $\{R_6\}$.

Replicate node creation. We stipulate that a node will be considered for replication if the ratio of redundant data to the

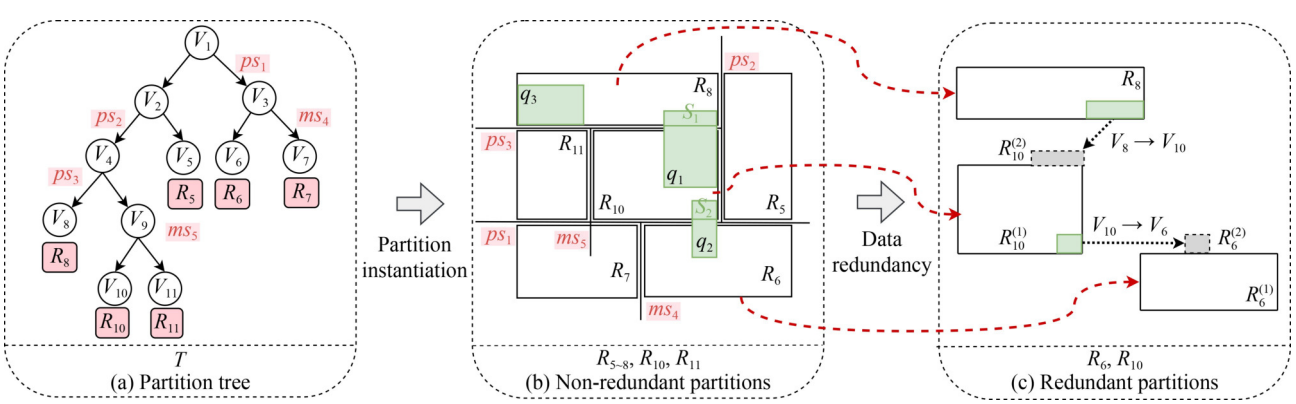

Fig. 6 Reducing unnecessary data reads for q_1 and q_2 by replicating small proportion of data areas (S_1, S_2) to R_{10} and R_6

entire node's data is below the set threshold θ. Generally, the more data is replicated, the greater the data skipping benefit $\mathcal{B}_{skip}$. However, if replicating an equal amount of data does not yield a proportional increase in $\mathcal{B}_{skip}$ (e.g., when the super node serves only a few queries), then this replication operation is not recommended due to its high data maintenance and storage costs with minimal $\mathcal{B}_{skip}$. To address this trade-off, we first define a function $y = F(\theta)$ to represent the change in the ratio of skipping benefit to redundant data proportion as θ increases, i.e.,

$$y = F(\theta) = \frac{w_s \times \mathcal{B}_{skip}}{w_r \times \frac{|\mathcal{D}_r(\theta)|}{|\mathcal{D}|}},$$

where w_s and w_r respectively are user-defined weights for $\mathcal{B}_{skip}$ and $\frac{|\mathcal{D}_r(\theta)|}{|\mathcal{D}|}$ (we set $w_s : w_r = 1 : 1$); $\mathcal{D}_r(\theta)$ is all redundant data when the replication threshold is θ; $\mathcal{D}$ denotes the table data. Since we aim to achieve greater skipping benefits with as low θ as possible, we can search for the inflection point of the y-curve's growth as the optimal $\overline{\theta}$, i.e., solving $F(\theta, y)'' = \frac{d^2 y}{d\theta^2} = 0$. It signifies that when $\theta > \overline{\theta}$, increasing $\mathcal{D}_r$ at the same proportion will result in limited growth in $\mathcal{B}_{skip}$.

Super node selection. We categorize the visited nodes into two groups: replicate nodes (V_X) with data to be replicated and candidate nodes (V_Y) without such redundant data. For each query, after deciding these replicate nodes (satisfying $\overline{\theta}$), we must select which node among the candidate nodes will serve as the super nodes. Generally, to maximize $\mathcal{B}_{skip}$, the V_Y with the fewest query accesses is preferred as the super node to avoid additional reads of redundant data for irrelevant accessing queries. Therefore, for each pair of nodes V_X and V_Y, we define a benefit ($\mathcal{B}_{allocate}$) for $V_X \to V_Y$ to reflect its allocation priority. The $\mathcal{B}_{allocate}$ arises from the skipped querying node size after replicating, minus the additional read size of redundant data for other irrelevant queries, that is

$$\mathcal{B}_{allocate}(V_X \to V_Y) = \left(|V_X| - \left|\widetilde{\mathcal{D}}_r(V_Y)\right|\right) \times \left|Q^{V_X \to V_Y}\right| - \left|\mathcal{D}_r^{V_X \to V_Y}\right| \times O(V_Y),$$

where V_X is the skipped node for all queries $Q^{V_X \to V_Y}$ that benefit from the creation of $V_X \to V_Y$; $\widetilde{\mathcal{D}}_r(V_Y)$ indicates the existing redundant data in V_Y before allocation; $\mathcal{D}_r^{V_X \to V_Y}$ represents the data replicated from V_X to V_Y; $O(V_Y)$ represents the number of queries accessing V_Y.

Next, we select the top element with the highest benefit, $\overline{V_Y}$, as the super node $(V_X \to \overline{V_Y})$, updating its $\widetilde{\mathcal{D}}_r(\overline{V_Y})$ accordingly, i.e., $\widetilde{\mathcal{D}}_r(\overline{V_Y}) += \mathcal{D}_r^{V_X \to \overline{V_Y}}$. Each $\overline{V_Y}$ can be allocated once or multiple times. This allocation process is repeated until positive benefits cannot be obtained.

6.4 Functionality of the LRP partition tree

The LRP partition tree T serves two primary functions: (1) Routing support: Each row in the raw table is processed, matching with leaf nodes via the node metadata of T, and subsequently written to the corresponding partition files (a.k.a., physical partitions). (2) Partition filtering: When a new query is submitted, its encoded logical tree $\mathbf{T}_{lg}$ is executed across all relevant partition trees. The logical predicates at each level of $\mathbf{T}_{lg}$ are examined from bottom to top to identify the satisfied leaf nodes of T. This limits the search space for partition files and assists the query optimizer in formulating an efficient plan.

7 Efficiency analysis of LRP design

In this section, we analyze the execution efficiency of four phases within LRP and compare it with two classical partitioning algorithms (QdTree [8], PAW [9]). The results are shown in Table 2.

1) Data and query feature extraction: QdTree incurs minimal overhead during predicate feature extraction due to the absence of a query prediction module, thus eliminating the need for query vectorization. In contrast, PAW requires traversing the entire sampling table to pre-execute queries for generating query vectors. LRP, however, only encodes non-numeric columns and performs logical computations, resulting in lower costs compared to PAW.

2) Query prediction: The training time for LRP's prediction module depends on the number of epochs, introducing additional costs due to weight updates across stacked network layers. Conversely, PAW is faster as it only necessitates multiple iterations of heuristic determinations to incrementally identify the optimal τ.

3) Partition tree construction: The selection of candidate predicates during each tree split process is the primary time bottleneck. Both QdTree and PAW employ a similar greedy strategy. However, LRP explores a larger solution space through beam search in each predicate selection round, potentially leading to exponential growth in time complexity if

Table 2 Time complexity analysis

Partitioning phase	Partitioning method		
	Qd-Tree	PAW	LRP(Ours)
① Data and query feature extraction	$O(N_Q)$	$O(z \times N_{\text{samples}} \times N_Q)$	$O(z_{\text{encoding}} \times N_{\text{samples}} + N_Q)$
② Similarity load prediction	–	$O\left(\frac{z \times N_Q}{\tau_{\text{step}}}\right)$	$O(z \times N_Q \times N_{\text{net}} \times N_{\text{epoch}})$
③ Partition tree construction	$O(N_Q \times 2^{h_{\text{tree}}})$	$O(N_Q \times 2^{h_{\text{tree}}})$	$O\left(N_Q \times 2^{h_{\text{tree}}} \times \frac{n_c^{h_{\max}-1}}{n_c - 1}\right)$
④ Data routing for partition instantiation	$O\left(\frac{z \times N_{\text{row}} \times h_{\text{tree}}}{N_{\text{machine}} \times B \times W_{\text{r/w}}}\right)$	$O\left(\frac{z \times N_{\text{row}} \times h_{\text{tree}}}{N_{\text{machine}} \times B \times W_{\text{r/w}}}\right)$	$O\left(\frac{z \times N_{\text{row}} \times h_{\text{tree}}}{N_{\text{machine}} \times B \times W_{\text{r/w}}}\right)$

Note: Column count (z), encoded column count (z_{encoding}), query count (N_Q), predicate count (N_P), original table row count (N_{row}), sampling row count (N_{samples}), encoding or decoding cost ($O(1)$ when using hash tables), prediction network layer count (N_{net}), SSD read/write coefficient ($W_{\text{r/w}}$), partition tree height (h_{tree}), beam width (n_c), machine count (N_{machine}), network bandwidth (B), and maximum explored tree depth ($h_{\max}$). τ_{step} denotes the average increment of τ per iteration in PAW's search for the optimal vector expansion ratio.

$h_{\max}$ is not properly set.

4) Data routing: All methods share the same partition instantiation process. To deploy partitions, all row data must be routed through the partition tree and written to specified block addresses. Here, we primarily considered the volume of data written and network factors, without accounting for finer-grained factors such as cache hit rate.

Overall, both PAW and LRP involve query encoding and prediction, leading to higher execution overhead compared to QdTree. When comparing LRP and PAW, LRP incurs lower costs only during the feature extraction phase, but its overall overhead is greater. However, as illustrated in Table 2②③, this issue can be partially mitigated by setting appropriate hyperparameters (e.g., keeping $h_{\max}$ as small as possible, removing redundant intermediate network layers to reduce N_{net}, etc.) or by employing multithreading to build different partition sub-trees in parallel. Furthermore, as shown in Table 2④, when routing large-scale datasets to create partition files (i.e., $N_{\text{row}} \gg N_{\text{samples}}$), the time scale differences among the algorithms have a smaller impact on overall system performance.

8 Experiments

In this section, we show the evaluation results of LRP system from three aspects: (1) We compare the scanning ratios of LRP (including its variants) with two classical partitioning algorithms (QdTree, PAW) and the optimal reference for raw and numeric tables under static loads; (2) We separately evaluate the table scanning ratios of different prediction methods combined with the same or different partitioners under δ-similarity loads. We also explore the impact of different mixing ratios of exploratory and similarity queries on partitioner performance; (3) We further evaluate the above experimental configurations in a real Spark cluster.

8.1 Experimental setup

We conduct extensive experiments on a Spark cluster with four computer nodes (3.8 GHz CPUs, 32 GB RAM, 1 TB disk), utilizing HDFS [18] as the underlying file system and Parquet files [34] to store partitioned block data. To speed up data routing, a distributed acceleration framework Ray [35] is utilized to fully leverage the computational resources of each machine. The neural networks (prediction module) are trained on a Tesla P40 GPU with a 24 GB frame buffer.

Dataset and static workload. (1) TPC-H benchmark [15] (50 GB data) comprises 8 tables (62 columns in total) and 22 query templates. (2) TPC-DS [36] (62 GB data) is used to simulate the sales operations of a department store. Compared to TPC-H, it offers a more comprehensive schema, including 7 fact tables and 14 dimension tables (a total of 429 columns), as well as 99 query templates. (3) JOB benchmark [37] is a 13 GB real-world IMDB dataset containing 12 tables (134 columns in total) and 33 query templates. (4) ClickBench [38] (20 GB data) is derived from an actual traffic platform, featuring a large table (105 columns, 100M records) and 43 query templates. Here, query templates are used to generate arbitrary number of static synthetic queries.

Similarity workload. Following PAW [9], we collect 4250 static queries as Q_H for TPC-H and JOB, and generate the mappings $Q_H \rightarrow Q_F$, named TPC-D and JOB-D, respectively. To be specific, we divide each tablespace into $2z$ regions and extend each domain of Q_H with a random distance threshold $\delta \in \text{uniform}(0.8\Gamma, \Gamma)$, where $\Gamma = \lambda(1 + \frac{i}{2z})$, $\lambda = 0.01$, and i represents that Q_H falls within the ith region. For the ordered queries, we set $\lambda \in [0.005, 0.015]$ and $\lambda \propto t$. For each benchmark, we randomly shuffle δ-similar queries and split them into training/validation/test sets by 7:2:1.

Evaluation metrics. (1) *Access Ratio* indicates the ratio of accessed data to the entire table size. (2) *Query Latency* refers to the end-to-end response time for processing queries. (3) *Model Execution Time* denotes the time taken to generate logical partition structures, including query parsing, query prediction, and partition tree creation.

Baseline methods.

- QdTree [8] generates the candidate split set using only numeric predicate conditions during the creation of the partition tree. Its split policy consistently selects the predicate condition that maximizes data skipping benefit at each extension step.
- PAW [9] optimizes the division of small nodes in QdTree by merging multiple steps of predicate conditions into a single-step split condition.
- LB-Cost represents a theoretical **l**ower **b**ound **cost** in an ideal scenario where all accessed data per query is searched and directly written to separate block files.
- TH$^+$ is a refined version of PAW [9]'s prediction module. It selects the optimal distance threshold $\bar{\delta}$ from all possible values for extending domain boundaries in Q_H, aiming to minimize the MSE between Q_P and Q_F.
- ML$^+$ is an abbreviation for the independent MLP or LSTM predictor component in LRP, aiming to minimize the access ratio when executing Q_F over Q_P-based partitions.
- LRP-E and LRP-S are two variants of LRP, each with a component removed to conduct an ablation study. They remove the column encoding and data redundancy modules from LRP, respectively.

8.2 Exp-1: Static scenario experiments

Evaluation on pure numeric tables. We first conduct experiments on pure numeric tables, i.e., by removing all non-numeric column data. Figure 7(a) reveals that LRP-S and LRP-E gains over a 8% and 15.7% reduction in table access ratio than PAW across all benchmarks, respectively. Meanwhile, Fig. 8 provides detailed comparisons for representative tables in each benchmark. All cases follow the same performance ranking: LB-Cost > LRP-E(LRP) > LRP-S > PAW > QdTree, where LRP is equivalent to LRP-E since the encoding component is invalid when non-numeric column data is excluded from the raw tables. These results have three-fold reasons: (1) LRP-S outperforms PAW, demonstrating that our beam search policy can identify better split orders among numerous condition combinations; (2) LRP-E outperforms LRP-S, indicating that super nodes help

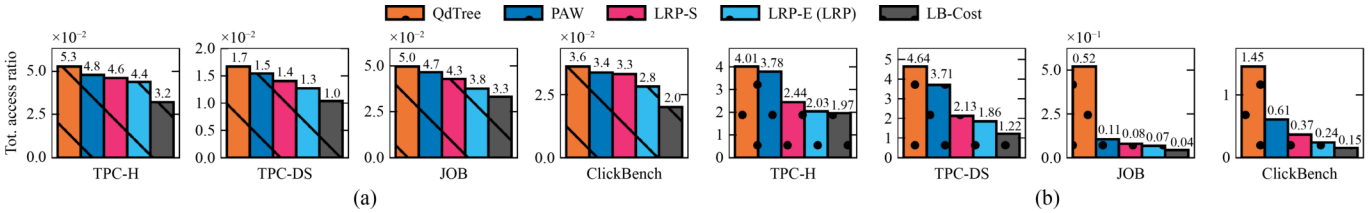

Fig. 7 Performance comparisons with baselines in two scenario types: (a) numerical benchmarks that retain only purely numeric column data and related predicates; (b) raw benchmarks that use complete datasets and predicates

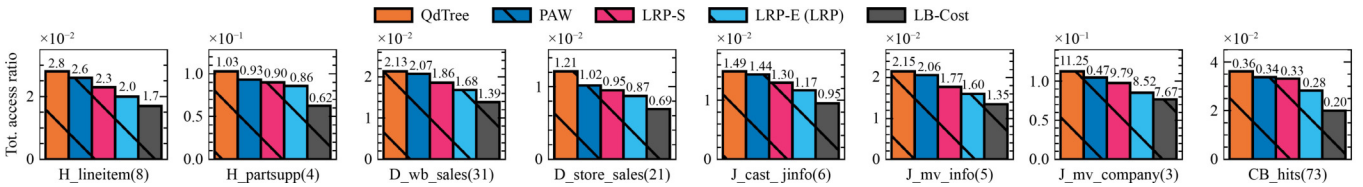

Fig. 8 We continue to explore the data access ratio for each table when only numerical columns are retained, using different partitioning algorithms. The prefixes 'H_', 'D_', 'J_', and 'CB_' represent the abbreviations for the TPC-H, TPC-DS, JOB, and ClickBench benchmarks, respectively. The number of numeric columns for each table is shown in parentheses

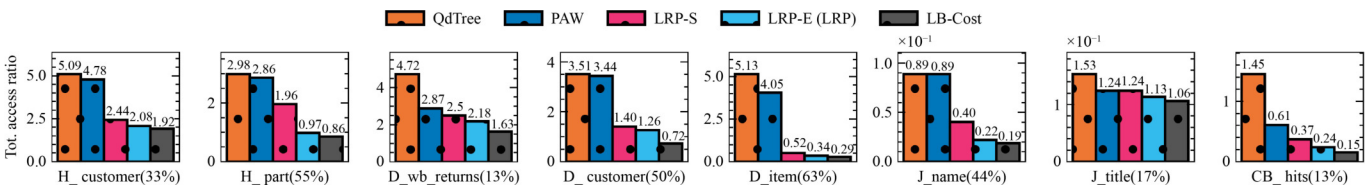

Fig. 9 Comparison of data access ratios against algorithm baselines on raw tables. Values in parentheses indicate the percentage of non-numeric predicates among all predicates

resolve query access contention caused by row-granularity limitations when handling high-density queries on tables like D_wb_sales and D_store_sales; (3) The performance differences between the algorithms are stable and do not vary significantly across different datasets or numerical predicates.

Evaluation on raw tables. When experimenting on raw tables, Fig. 7(b) reveals larger performance discrepancies among the algorithms compared to the numeric environments shown in Fig. 7(a). Detailed comparisons for specific tables are provided in Fig. 9. LRP-S and LRP reduce the data access ratio by 38.9% and 48.8%, respectively, compared to PAW. This reduction is especially pronounced in tables with a high proportion of predicates, such as H_part, D_item, and J_name (average 64% ⇓). This is because when more textual predicates (especially those involving costly join operations) are mixed with numeric predicates, LRP and LRP-S are still able to effectively capture the complete query access patterns. In contrast, QdTree and PAW perform worse due to their incomplete query information, rendering numeric predicate-based splits inefficient. This inefficiency is also reflected in their similar table scanning ratios in cases like H_part, D_customer, and J_name, where a large proportion of text predicates are present. Moreover, the result that LRP outperforms LRP-S indicates that LRP's allowance for data redundancy among partitions further enhances performance on raw tables. These factors contribute to LRP exhibiting more significant reductions in scanning data compared to the other baselines.

Evaluation on model overhead. Regarding the raw tables, Fig. 10(a) evaluates the model execution time for each

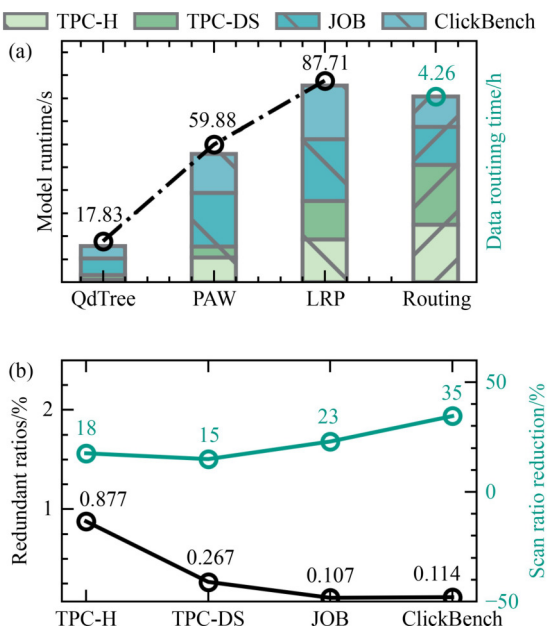

Fig. 10 Comparison of model overhead metrics on raw tables: (a) Model execution time and data routing time; (b) reduction in data access caused by data duplication

algorithm and their average data routing time when deploying physical partitions. Despite LRP being 17.87 s slower than PAW in logical partition generation due to its additional data encoding and redundancy modules, their cumulative processing time (163 s) is far lower than the data routing time (4.26 h). Thus, this time gap can be considered negligible. Figure 10(b) displays the changes in the data redundant ratio. We observe that SuperNode minimizes block addressing and

scanning by replicating a small portion of data. Specifically, LRP yields an average scanning ratio reduction of 22.7% (i.e., $\frac{18+15+23+35}{4}\%$) over LRP-S, with only a 0.0034 data redundancy ratio. This is because queries with more textual predicates are more likely to access dispersed, smaller amounts of data, thereby promoting the $\mathcal{B}_{skip}$ of redundant partitions.

Evaluation on model adaptability. We also conduct model sensitivity tests with varying workload configuration. Figure 11(a) generates TPC-H queries ranging from 50 to 1000, increasing the query density and requiring finer partitions. LRP initially achieves a 6.7% performance improvement over PAW, eventually reaching 29%. Figure 11(b) explores the impact of maximum query range: an increasing queried range results in higher access ratios and a larger performance gap between LRP and PAW. However, at 0.3 or above, the quality of all layouts begins to deteriorate until reaching 1, at which point the quality of all layouts become consistent, with their queried data being stored in the same number of blocks. Figure 11(c) restricts the maximum number of columns accessed by each query. As shown, the more columns involved in the predicates, the fewer the average rows typically meet the conditions, resulting in a smaller proportion of data scanned. This does not affect predicate selection during partition tree construction, thus having minimal impact on algorithm performance. Figure 11(d) explores two extreme query distributions. Under uniform distribution, each algorithm effectively partitions rows to accommodate each query, with LRP performing best. When queries are densely distributed, some table areas cannot be effectively partitioned due to contention among queries, reducing the performance of all algorithms. However, LRP still maintains the lowest access ratio, benefiting from data replication.

8.3 Exp-2: δ-similarity scenario experiments

In addition to the provided TH$^+$ and ML$^+$ predictors, we establish two additional prediction baselines: *N/A*, representing no prediction, and OPT, referring to the predicted load satisfying $Q_P = Q_F$.

Evaluation on prediction approaches. Figure 12 displays the Q_P-based layout performance for various prediction methods, along with the performance deviation (shaded areas) across single-table layouts. Regardless of whether the same partitioner (e.g., LRP) or distinct partitioners (i.e., QdTree, PAW, LRP, and LB-Cost) are used to create layouts based on the given predictions, our ML$^+$ predictor always outperforms the best alternative, achieving performance improvements of 46% and 72% compared to TH$^+$ in unordered and ordered loads, respectively. Additionally, ML$^+$ shows narrower shaded boundaries, indicating better optimization for each sub-layout. Moreover, compared to ordered loads, TH$^+$ proves less effective in unordered loads with fewer historical queries, while ML$^+$ exhibits greater stability.

Case study. Figures 13(a)–13(c) and 13(d)–13(f) present 3D visualizations of query prediction cases from TPC-D and JOB-D, respectively, across dimensions c_1, c_2, and c_3. In Figs. 13(a) and 13(d), a close-up of a randomly selected query is shown, with the coordinate system adjusted for clarity. Blue and red boxes represent the H_{Q_P} generated by TH$^+$ and ML$^+$, respectively, while the green boxes depict the actual H_{Q_F}. In

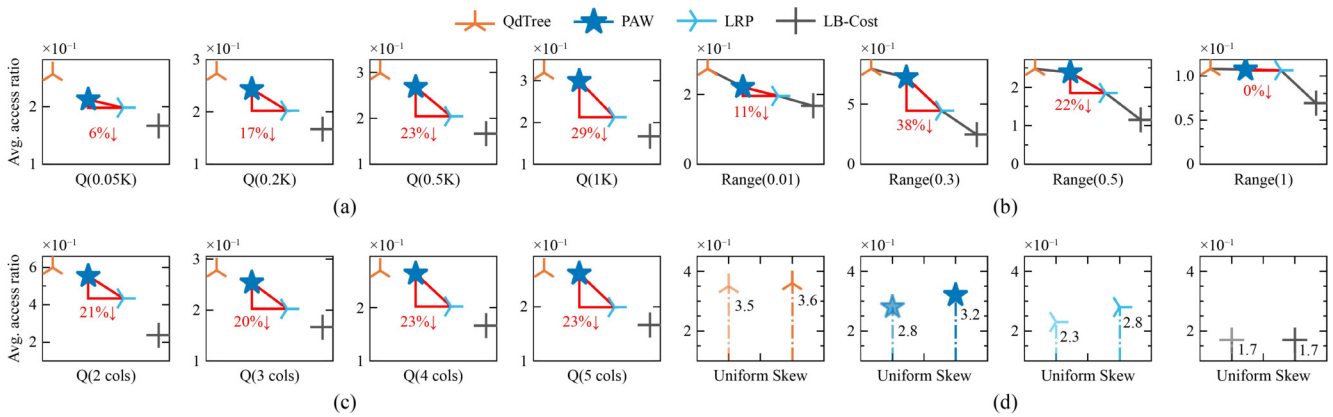

Fig. 11 Sensitivity analysis of the model: (a) Varying the number of queries; (b) varying the maximal queried column domain range; (c) varying the maximum number of co-accessed columns allowed per query; (d) varying query distribution characteristics, including uniform and skew distributions, under the constraint of the same number of queries

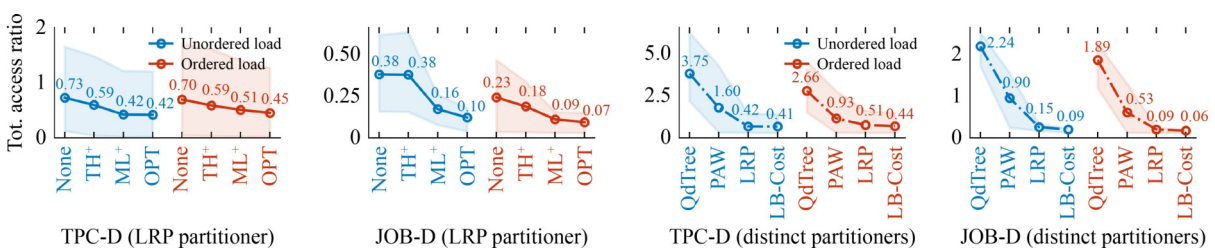

Fig. 12 Comparison of inference layout performance in TPC-D and JOB-D environments

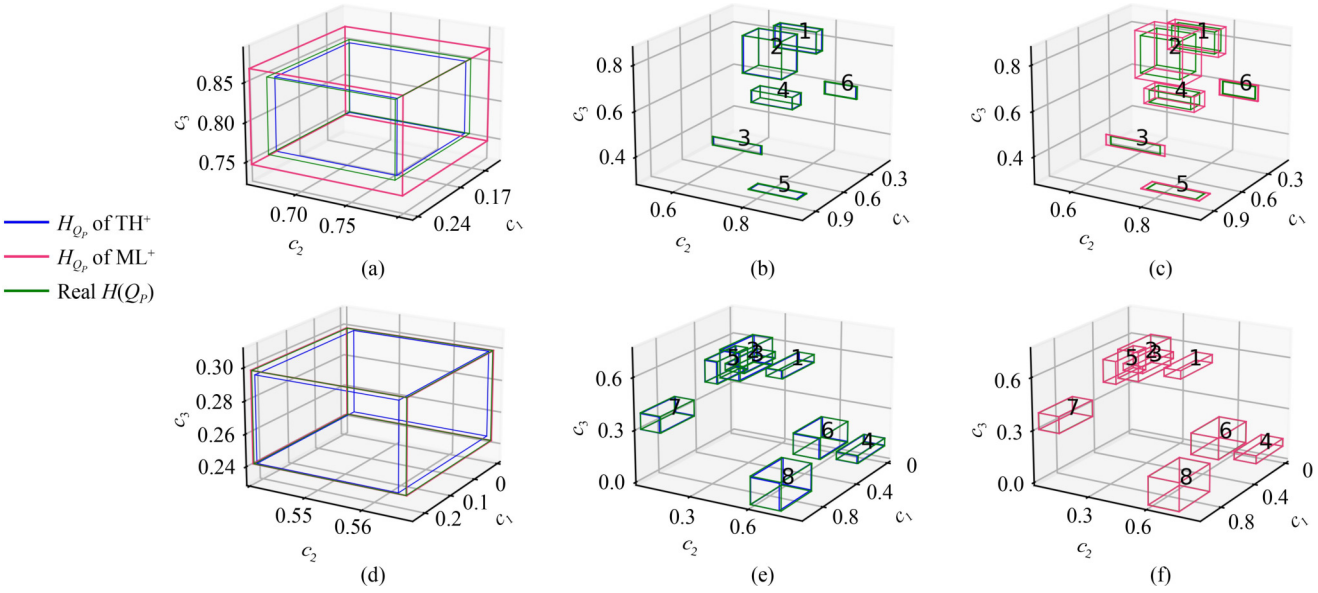

Fig. 13 Case study: visualization of prediction results from TH+ and ML+ on (a)–(c) TPC-D and (d)–(f) JOB-D samples

Fig. 13(a), although TH+'s H_{Q_P} aligns more closely with H_{Q_F} compared to ML+, i.e., $\mathcal{L}_{mse}(\text{TH}^+) < \mathcal{L}_{mse}(\text{ML}^+)$, ML+ achieves a smaller prediction loss ($\mathcal{L}_{wd}(\text{ML}^+) < \mathcal{L}_{wd}(\text{TH}^+)$) due to its N_{eg} value being 0. Intuitively, if the red box is used as the basis for constructing partition files, then for queries requiring scanning the green box, only one file needs to be accessed to complete the task. Similarly, in Fig. 13(d), the N_{eg} of ML+ is 0, which is lower than the 4 of TH+. This suggests that ML+ predicts domain values that are slightly larger than the real H_{Q_F}, making robust partitioning feasible.

Figures 13(b)–13(c) and 13(e)–13(f) randomly select six and eight query cases, respectively, with their prediction metrics compared in Table 3. In all cases, partitions based on ML+'s predictions consistently yield fewer scanned blocks ($\text{ML}^+ : \text{TH}^+ \Rightarrow 45 < 198$) and lower data scan ratios ($0.0045 < 0.018$). This can be attributed to ML+'s smaller N_{eg} ($0.36 < 4.86$), leading to a smaller $\mathcal{L}_{wd}$ value ($0.79 < 8.45$), irrespective of the $\mathcal{L}_{mse}$ value ($3\times10^{-4} > 9\times10^{-5}$).

Evaluation on the exploratory queries. Figure 14 shows that LRP mitigates performance degradation when more exploratory queries are mixed into TPC-D and JOB-D test queries. As the random percentage increases from 0% to 85%, the access ratio gap between PAW and QdTree expands from

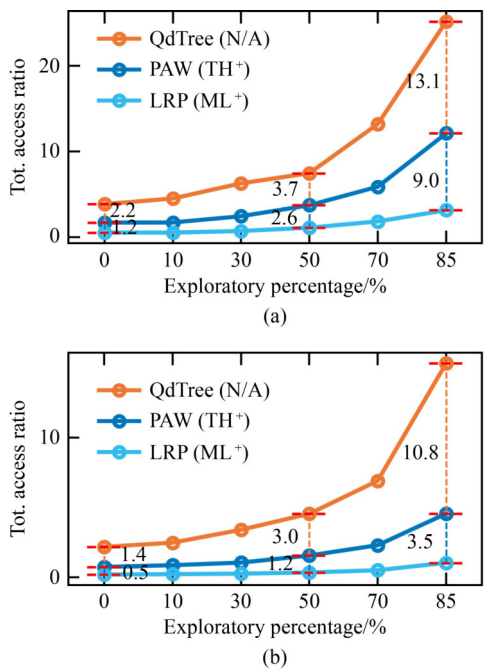

Fig. 14 Performance comparison when mixing more exploratory queries into TPC-D and JOB-D load environments. (a) TPC-D; (b) JOB-D

Table 3 Comparison of prediction metrics and access ratio reductions for selected query vectors from TPC-D and JOB-D

Prediction metric	Method	Query vector case (TPC-D)						Query vector case (JOB-D)								Average
		#1	#2	#3	#4	#5	#6	#1	#2	#3	#4	#5	#6	#7	#8	
MSE loss($\mathcal{L}_{mse}$)/10^{-5}	TH+	4.66	6.66	2.21	2.55	4.61	1.1	9.56	15.9	11.6	12.1	12.5	15.2	12.3	18.8	**9.27**
	ML+	60.3	19.6	46.7	59.5	45.7	26.7	0.21	0.55	0.44	0.31	0.21	0.53	0.51	0.39	31.3
Prediction loss($\mathcal{L}_{wd}$)	TH+	10.5	10.3	10.5	10.3	10.4	10.4	7.17	7.12	7.11	7.03	7.01	7.01	7.00	7.05	8.49
	ML+	0.41	0.45	0.41	0.45	0.43	0.44	3.36	1.70	3.36	0.03	0.03	0.03	0.04	0.02	**0.79**
#Uncovered edges(N_{eg})	TH+			6 (all cases)							4 (all cases)					4.86
	ML+			0 (all cases)				2	1	2	0	0	0	0	0	**0.36**
#Scanning blocks	TH+	163	399	39	124	384	16	180	258	328	96	283	150	345	15	198
	ML+	38	107	6	24	11	7	24	88	214	16	64	32	8	4	**45**
Access ratio ($/10^{-2}$)	TH+	0.94	2.31	0.22	0.72	2.22	0.40	2.14	3.07	3.90	1.14	3.37	1.78	4.11	0.17	1.89
	ML+	0.22	0.61	0.03	0.14	0.06	0.04	0.28	1.02	2.50	0.19	0.75	0.37	0.09	0.05	**0.45**

2.2 to 13.1 in TPC-D, and from 1.4 to 10.8 in JOB-D. Similarly, the access ratio gap between LRP and PAW expands from 1.2 to 9 in TPC-D and from 0.5 to 3.5 in JOB-D. This improvement is attributed to PAW/LRP's use of data distribution-based median splits, which enhances their layout adaptability to exploratory queries compared to QdTree's sole use of predicate splits. Moreover, LRP's consideration of median splits for encoded non-numeric column data further boosts its adaptability.

8.4 Exp-3: efficiency analysis in Spark cluster

There is a positive correlation between the access ratio and query latency. Reducing the data access ratio typically results in decreased query latency due to fewer accessed partition files, less scanned metadata and column data on disk, and less frequent merging of scan results.

To assess the quality of constructed layouts, we utilized Spark-SQL, averaging five executions for each indicator measurement. Table 4 presents statistical data such as averages, minimums, percentiles, regarding query latency. Our findings are as follows. (1) Static Scenarios: For both numeric and raw datasets, LRP consistently outperforms other baselines across various statistical indicators. The optimizations in the partition tree and the complete consideration of logical predicates make LRP achieve an average query latency reduction of 15.9% and 47.5% for the two types of datasets, respectively, compared to PAW. LRP exhibits more significant performance improvements in raw tables, similar to the trends observed in Fig. 7(b). By effectively leveraging text-type predicates as partitioning features to partition data, LRP accelerates filter operations, especially in queries with a higher proportion of non-numeric columns. (2) Dynamic Scenarios: In TPC-D and JOB-D, LRP consistently achieves greater performance improvements compared to static scenarios, attributable to its robust layout supported by predictive networks with a novel loss function. This function allows ML^+ to establish a positive correlation between prediction loss and layout robustness while minimizing prediction loss. Compared to PAW, LRP achieves query response times that are 2.49× faster ($\frac{5.13\ s}{2.06\ s}$) on TPC-D and 3.17× faster ($\frac{1.70\ s}{0.54\ s}$) on JOB-D.

9 Conclusion and future work

In this paper, we propose an end-to-end LRP system. LRP begins by designing a comprehensive data and query encoding scheme to extract valuable query access patterns across both numeric and non-numeric column data types. It then trains MLP/LSTM networks to minimize loss values by integrating partition semantic information, thereby predicting future query pattern shifts. Finally, it implements beam search-based tree-splitting and node redundancy policies to generate a partition tree, which can materialize a robust data layout tailored to predicted query patterns. Experimental results on Spark demonstrate that our method significantly outperforms existing solutions, achieving a 49.20% reduction in query latency for static queries and a 64.15% reduction in dynamic

Table 4 Testing real query latency on: (1) TPC-H, TPC-DS, JOB, and ClickBench benchmarks with only numeric column data; (2) Raw TPC-H, TPC-DS, JOB, and ClickBench benchmarks; (3) TPC-D and JOB-D benchmarks designed for similarity scenarios. The **red** denotes the best result, excluding the optimal reference, while the **blue** denotes the second-best result

Benchmark	Method	Query latency/s					Benchmark	Method	Query latency/s				
		75th	90th	95th	Min	Mean			75th	90th	95th	Min	Mean
Numeric TPC-H	QdTree	1.53	1.59	1.79	0.72	1.21	Numeric JOB	QdTree	3.51	3.84	4.01	0.83	2.90
	PAW	0.97	0.98	0.98	0.69	0.97		PAW	1.20	1.27	1.33	0.81	1.08
	LRP (ours)	**0.87**	**0.90**	**0.93**	**0.66**	**0.85**		LRP (ours)	**0.96**	**0.96**	**0.97**	**0.81**	**0.92**
	LB-Cost	0.74	0.77	0.79	0.54	0.69		LB-Cost	0.76	0.77	0.77	0.70	0.74
	Imp./%	**10.6**	**7.7**	**5.0**	**4.7**	**⇑11.9**		**Imp./%**	**20.4**	**24.3**	**27.1**	**0**	**⇑14.4**
Numeric TPC-DS	QdTree	4.26	6.61	7.59	0.21	5.41	Numeric ClickBench	QdTree	1.14	1.58	2.22	0.27	0.83
	PAW	1.82	2.43	2.51	0.12	3.23		PAW	0.66	0.68	0.74	0.27	0.65
	LRP (ours)	**0.94**	**1.72**	**2.12**	**0.12**	**2.57**		LRP (ours)	**0.55**	**0.56**	**0.56**	**0.25**	**0.54**
	LB-Cost	0.88	1.09	1.14	0.07	1.28		LB-Cost	0.25	0.27	0.30	0.17	0.24
	Imp./%	**48.4**	**29.1**	**15.6**	**0.4**	**⇑20.3**		**Imp./%**	**16.4**	**17.3**	**23.9**	**6.7**	**⇑17.1**
Raw TPC-H	QdTree	11.22	15.16	18.02	0.36	5.96	Raw JOB	QdTree	2.48	3.80	4.63	0.28	1.15
	PAW	8.30	10.41	11.46	0.19	4.66		PAW	1.29	1.59	1.68	0.24	0.79
	LRP (ours)	**3.96**	**4.95**	**5.47**	**0.16**	**2.38**		LRP (ours)	**0.61**	**0.73**	**0.78**	**0.23**	**0.44**
	LB-Cost	3.07	3.73	4.11	0.10	1.69		LB-Cost	0.38	0.47	0.52	0.13	0.26
	Imp./%	**52.3**	**52.5**	**52.2**	**13.6**	**⇑48.9**		**Imp./%**	**52.8**	**54.1**	**53.7**	**5.2**	**⇑44.5**
Raw TPC-DS	QdTree	3.68	5.57	7.23	0.35	4.83	Raw ClickBench	QdTree	8.90	9.47	9.69	1.75	8.72
	PAW	1.81	2.81	2.98	0.21	3.47		PAW	4.24	4.64	4.84	1.30	5.07
	LRP (ours)	**1.21**	**1.75**	**2.05**	**0.21**	**2.00**		LRP (ours)	**1.92**	**2.28**	**2.31**	**0.40**	**2.32**
	LB-Cost	0.78	1.01	1.06	0.15	1.15		LB-Cost	1.43	1.79	1.88	0.16	1.52
	Imp./%	**33.3**	**37.5**	**31.2**	**0**	**⇑42.3**		**Imp./%**	**54.9**	**50.8**	**52.3**	**68.9**	**⇑54.2**
TPC-D	QdTree	11.97	14.80	15.95	1.67	9.93	JOB-D	QdTree	5.22	6.72	8.48	0.97	3.15
	PAW	6.96	8.39	9.34	0.21	5.13		PAW	2.19	2.69	3.01	0.58	1.70
	LRP (ours)	**2.04**	**2.26**	**2.35**	**0.17**	**2.06**		LRP (ours)	**0.81**	**1.13**	**1.44**	**0.25**	**0.54**
	LB-Cost	1.14	1.22	1.27	0.13	1.16		LB-Cost	0.26	0.28	0.29	0.18	0.24
	Imp./%	**70.7**	**73.1**	**74.8**	**20.3**	**⇑59.8**		**Imp./%**	**62.9**	**58.2**	**52.2**	**56.7**	**⇑68.5**

environments.

However, some aspects of LRP still require further refinement, which we plan to address in future work. (1) Join Optimization: When selecting tree-splitting conditions, we did not fully consider the cost of data shuffling, which often exceeds scanning costs. Therefore, predicate and median conditions related to Join columns should be weighted more heavily to reduce shuffling overhead. (2) Load Balancing: Although table data has been allocated to partition files, the distribution of these files across machines needs further optimization. Future work will focus on optimizing the physical placement of files based on access frequency and other factors to ensure load balancing and maximize the utilization of cluster resources.

Acknowledgements This work was supported by the National Key Research and Development Program of China (Grant No. 2023YFB4503600) and the National Natural Science Foundation of China (Grant Nos. U23A20299, 62072460, 62172424, 62276270, and 62322214).

Competing interests The authors declare that they have no competing interests or financial conflicts to disclose.

References

1. Taylor R W, Sacca D, Wiederhold G. Database partitioning in a cluster of processors. ACM Transactions on Database Systems (TODS), 1985, 10(1): 29−56
2. Copeland G, Alexander W, Boughter E, Keller T. Data placement in bubba. In: Proceedings of 1988 ACM SIGMOD International Conference on Management of Data. 1988, 99−108
3. Stöhr T, Märtens H, Rahm E. Multi-dimensional database allocation for parallel data warehouses. In: Proceedings of the 26th International Conference on Very Large Data Bases. 2000, 273−284
4. Bentley J L. Multidimensional binary search trees used for associative searching. Communications of the ACM, 1975, 18(9): 509−517
5. Zhan C, Su M, Wei C, Peng X, Lin L, Wang S, Chen Z, Li F, Pan Y, Zheng F, Chai C. AnalyticDB: real-time OLAP database system at alibaba cloud. Proceedings of the VLDB Endowment, 2019, 12(12): 2059−2070
6. Papadomanolakis S, Ailamaki A. AutoPart: automating schema design for large scientific databases using data partitioning. In: Proceedings of the 16th International Conference on Scientific and Statistical Database Management. 2004, 383−392
7. Sun L, Franklin M J, Krishnan S, Xin R S. Fine-grained partitioning for aggressive data skipping. In: Proceedings of 2014 ACM SIGMOD International Conference on Management of Data. 2014, 1115−1126
8. ang Z, Chandramouli B, Wang C, Gehrke J, Li Y, Minhas U F, Larson P Å, Kossmann D, Acharya R. Qd-tree: learning data layouts for big data analytics. In: Proceedings of 2020 ACM SIGMOD International Conference on Management of Data. 2020, 193−208
9. Li Z, Yiu M L, Chan T N. PAW: data partitioning meets workload variance. In: Proceedings of the 38th IEEE International Conference on Data Engineering. 2022, 123−135
10. Sun L, Franklin M J, Wang J, Wu E. Skipping-oriented partitioning for columnar layouts. Proceedings of the VLDB Endowment, 2016, 10(4): 421−432
11. Li C, Markl V, Aly A M, Mahmood A R, Hassan M S, Aref W G, Ouzzani M, Elmeleegy H, Qadah T. AQWA: adaptive query workload aware partitioning of big spatial data. Proceedings of the VLDB Endowment, 2015, 8(13): 2062−2073
12. Aly A M, Elmeleegy H, Qi Y, Aref W. Kangaroo: workload-aware processing of range data and range queries in hadoop. In: Proceedings of the 9th ACM International Conference on Web Search and Data Mining. 2016, 397−406
13. Lu Y, Shanbhag A, Jindal A, Madden S. AdaptDB: adaptive partitioning for distributed joins. Proceedings of the VLDB Endowment, 2017, 10(5): 589−600
14. Ding J, Minhas U F, Chandramouli B, Wang C, Li Y, Li Y, Kossmann D, Gehrke J, Kraska T. Instance-optimized data layouts for cloud analytics workloads. In: Proceedings of 2021 International Conference on Management of Data. 2021, 418−431
15. TPC-H benchmark. See tpc.org/tpch/ website, 1999.
16. Rosenblatt F. Principles of neurodynamics: perceptrons and the theory of brain mechanisms. The American Journal of Psychology, 1963, 76(4): 705−707
17. Hochreiter S, Schmidhuber J. Long short-term memory. Neural Computation, 1997, 9(8): 1735−1780
18. Shvachko K, Kuang H, Radia S, Chansler R. The hadoop distributed file system. In: Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST). 2010, 1−10
19. Shanbhag A, Jindal A, Madden S, Quiane J, Elmore A J. A robust partitioning scheme for ad-hoc query workloads. In: Proceedings of 2017 Symposium on Cloud Computing. 2017, 229−241
20. Huang D, Liu Q, Cui Q, Fang Z, Ma X, Xu F, Shen L, Tang L, Zhou Y, Huang M, Wei W, Liu C, Zhang J, Li J, Wu X, Song L, Sun R, Yu S, Zhao L, Cameron N, Pei L, Tang X. TIDB: a raft-based HTAP database. Proceedings of the VLDB Endowment, 2020, 13(12): 3072−3084
21. ClickHouse: an open-source columnar database management system. See clickhouse.com/docs/en/observability/managing-data website, 2016
22. Dageville B, Cruanes T, Zukowski M, Antonov V, Avanes A, Bock J, Claybaugh J, Engovatov D, Hentschel M, Huang J S, Lee A W, Motivala A, Munir A Q, Pelley S, Povinec P, Rahn G, Triantafyllis S, Unterbrunner P. The snowflake elastic data warehouse. In: Proceedings of 2016 International Conference on Management of Data. 2016, 215−226
23. Moerkotte G. Small materialized aggregates: a light weight index structure for data warehousing. In: Proceedings of the 24th International Conference on Very Large Data Bases. 1998, 476−487
24. Graefe G. Fast loads and fast queries. In: Proceedings of the 11th International Conference on Data Warehousing and Knowledge Discovery. 2009, 111−124
25. Kang D, Jiang R, Blanas S. Jigsaw: a data storage and query processing engine for irregular table partitioning. In: Proceedings of 2021 International Conference on Management of Data. 2021, 898−911
26. han A, Yan X, Tao S, Anerousis N. Workload characterization and pre diction in the cloud: a multiple time series approach. In: Proceedings of 2012 IEEE Network Operations and Management Symposium. 2012, 1287−1294
27. Pavlo A, Angulo G, Arulraj J, Lin H, Lin J, Ma L, Menon P, Mowry T C, Perron M, Quah I, Santurkar S, Tomasic A, Toor S, Van Aken D, Wang Z, Wu Y, Xian R, Zhang T. Self-driving database management systems. In: Proceedings of the 8th Biennial Conference on Innovative Data Systems Research. 2017, 1
28. Ma L, Van Aken D, Hefny A, Mezerhane G, Pavlo A, Gordon G J. Query-based workload forecasting for self-driving database management systems. In: Proceedings of 2018 International Conference on Management of Data. 2018, 631−645
29. Hilprecht B, Binnig C, Röhm U. Learning a partitioning advisor for cloud databases. In: Proceedings of 2020 ACM SIGMOD International Conference on Management of Data. 2020, 143−157
30. Zhou X, Li G, Feng J, Liu L, Guo W. Grep: a graph learning based database partitioning system. Proceedings of the ACM on Management of Data, 2023, 1(1): 94
31. Jindal A, Dittrich J. Relax and let the database do the partitioning online. In: Proceedings of the 5th International Workshop on Business Intelligence for the Real-Time Enterprise. 2011, 65−80
32. Wang J, Chai C, Liu J, Li G. Face: a normalizing flow based cardinality estimator. Proceedings of the VLDB Endowment, 2021, 15(1): 72−84

33. Hoerl A E, Kennard R W. Ridge regression: biased estimation for nonorthogonal problems. Technometrics, 1970, 12(1): 55−67
34. Bertino E, Atzeni P, Tan K L, Chen Y, Tay Y C, Melnik S, Gubarev A, Long J J, Romer G, Shivakumar S, Tolton M, Vassilakis T. Dremel: interactive analysis of web-scale datasets. Proceedings of the VLDB Endowment, 2010, 3(1–2): 330−339
35. Ray: an open source framework to build and scale your ML and Python applications. See docs.ray.io/en/latest/ website, 2017
36. TPC-DS benchmark. See www.tpc.org/tpcds/ website, 2005
37. JOB benchmark. See developer.imdb.com/non-commercial-datasets/ website, 2016
38. ClickBench benchmark. See github.com/ClickHouse/ClickBench website, 2019

Pengju Liu is currently pursuing his PhD degree at the School of Information, Renmin University of China, China. He received his B.S. degree in information management and information system from Dalian Maritime University, China, in 2020. His research interests include adaptable database partitioning, and load forecasting.

Pan Cai is a PhD candidate at School of Information, Renmin University of China, China. Her research interests include query optimization, learned index, and machine learning. She is particularly interested in the structural design of learned indexes on multidimensional data.

Kai Zhong is a PhD student at School of Information, Renmin University of China, China, advised by Professor Cuiping Li. He received his BS degree in Information and Computing Science at School of Information, Huazhong Agricultural University, China in June 2022. His research lie in the field of AI4DB and Machine Learning.

Cuiping Li is currently a professor at the Renmin University of China, China. She received her PhD degree from Chinese Academy of Sciences, China in 2003. Prior to that, she earned her BS and MS degrees from Xi'an Jiaotong University, China in 1994 and 1997, respectively. She is a distinguished member of CCF. Her main research interests are machine learning for database and distributed query optimization.

Hong Chen is currently a professor at the Renmin University of China, China. She received her PhD degree from Chinese Academy of Sciences, China in 2000. Before that, she received her BS and MS degrees from Renmin University of China, China in 1986 and 1989, respectively. She received Second Prize of the National Award for Science and Technology Progress in 2018. She is a committee member of CCF and CIC. Her research interests include database technology and high-performance computing.