

# PAW: Data Partitioning Meets Workload Variance

Zhe Li

Department of Computing  
Hong Kong Polytechnic University  
richie.li@connect.polyu.hk

Man Lung Yiu

Department of Computing  
Hong Kong Polytechnic University  
csmlyiu@comp.polyu.edu.hk

Tsz Nam Chan

Department of Computer Science  
Hong Kong Baptist University  
edisonchan@comp.hkbu.edu.hk

**Abstract**—In distributed storage systems (e.g., HDFS, Amazon S3, Databricks), partitioning is applied on a dataset in order to enhance performance and availability. Recently, partitioning methods have been designed to optimize the query performance of partitions with respect to the historical query workload. Nevertheless, in practice, future query workloads may deviate from the historical query workload, thus deteriorating the performance of existing partitioning methods.

To fill this research gap, we model the variance of future query workloads from the historical query workload, then exploit this characteristic to produce partitions that perform well for future query workloads. In addition, we explore the space of irregular shaped partition regions to further optimize the query performance. Experimental results on TPC-H and real datasets show that our proposal is up to 70× more efficient than the state-of-the-art method.

## I. INTRODUCTION

In many applications, distributed systems (e.g., Hadoop, Spark) are employed to store and query a vast amount of data. These systems adopt block-based storage (e.g., HDFS, Amazon S3, Databricks), which partitions a dataset into large data blocks<sup>1</sup>, then provides access at the block level. During query processing, it suffices to fetch the data blocks whose semantic descriptions intersect with the query region. Recent studies [1]–[5] have used the historical query workload to construct partitions in order to reduce the query cost.

We conceptually visualize three scenarios for partitioning in Figure 1, based on the similarity between the historical query workload and future query workloads. The historical query workload  $Q_H$  is denoted by a red dot, whereas future query workloads in different time periods  $Q_{F1}, Q_{F2}, Q_{F3}$  are represented by blue dots.

- **The most specific:** The first scenario (Figure 1a) assumes that future workloads have exactly the same distribution as the historical workload. The state-of-the-art solution is the Qd-tree [1], which aims to optimize the performance of partitions with respect to the historical workload only. However, the above assumption is seldom true in practice, causing performance deterioration on future workloads, as we shall discuss shortly.
- **The most generic:** The third scenario (Figure 1c) considers exploratory queries [6]–[9]; future workloads are

unpredictable and may vary significantly from the historical workload. The typical solution is to partition the dataset based on data distribution (e.g.,  $k$ -d tree, R-tree), like in SpatialHadoop [5].

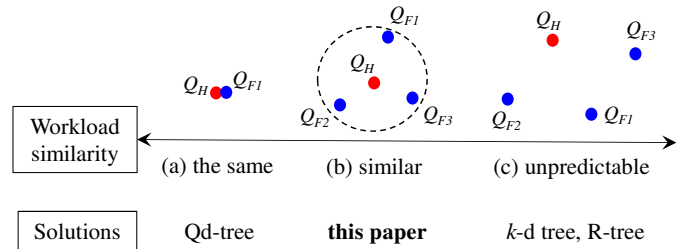


Fig. 1: The similarity between the historical workload  $Q_H$  and future workloads  $Q_{F1}, Q_{F2}, Q_{F3}$ , in three scenarios

In this paper, we explore the alternative scenario in Figure 1b, where future workloads are similar to the historical workload. Observe that concepts in machine learning (e.g., model, training set, testing set) [10] are analogous to the corresponding concepts in our problem (i.e., partition layout, historical workload, future workload). *Overfitting* occurs when a model is optimized to fit the training set (e.g.,  $Q_H$ ) closely; however, such a model does not *generalize* well to the testing set (e.g.,  $Q_{F1}, Q_{F2}, Q_{F3}$ ).

### A. Limitations of existing solutions

Unfortunately, our scenario cannot be efficiently addressed by the solutions from other scenarios. Obviously, the solutions from the third scenario (e.g.,  $k$ -d tree, R-tree) are inefficient because they do not exploit any workload information.

The best solution from the first scenario suffers from *overfitting* when it is used in our scenario. We exemplify this concern in Figure 2 on a dataset with two attributes  $A$  and  $B$ . By building the Qd-tree [1] on the historical workload  $Q_H = \{q_1, \dots, q_4\}$ , we obtain the partition layout  $\mathcal{P} = \{P_1, \dots, P_5\}$  shown in Figure 2a. The future workload is  $Q_F = \{q'_1, \dots, q'_4\}$ , where  $q'_i$  is similar to  $q_i$ . Although the partition layout  $\mathcal{P}$  is efficient for the historical workload, it incurs high cost for the future workload. For instance, the I/O cost for  $q_1$  in Figure 2a is 220MB only; however, the I/O cost for  $q'_1$  in Figure 2b is 790MB because  $q'_1$  intersects all partitions. This phenomenon is similar to the concept of *overfitting* in machine learning [10].

This work was supported by grant GRF 152050/19E from the Hong Kong RGC.

<sup>1</sup>For example, the typical size of a data block in HDFS is 128MB.

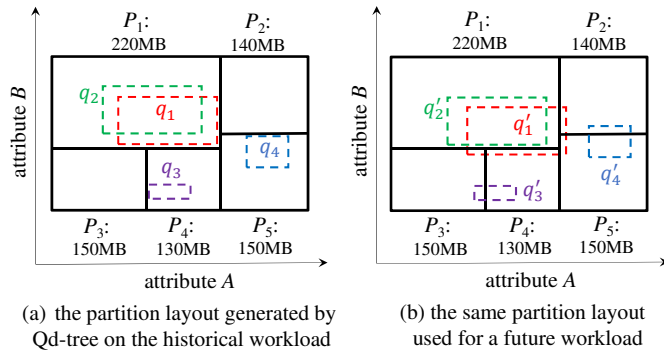


Fig. 2: An example of historical queries and future queries in workload-aware partitioning

In addition, existing partitioning methods produce rectangular partitions only. Frequently-used partitions (e.g.,  $P_1$  in Figure 2a) may cover rarely-queried regions and thus deteriorate the query performance. We demonstrate an alternative partition layout in Figure 3, which contains three rectangular partitions ( $P_1, P_2, P_3$ ) and an irregular shape partition ( $P_4$ ). Our idea is to enlarge  $P_4$  (in any shape) as much as possible in order to cover rarely-queried regions, and thus reduce the sizes of more frequently-used partitions (e.g.,  $P_1, P_2, P_3$ ). Observe that the query cost of query  $q_2$  is reduced from 220MB in Figure 2a to 160MB in Figure 3.

TABLE I: Methods for partitioning

Method	✓ Support	∇ No optimization	× No support
Method	Qd-tree [1]	k-d tree [11]	PAW (ours)
Exact workload (Fig. 1a)	✓	∇	✓
Similar workload (Fig. 1b)	∇	∇	✓
Unpredictable workload (Fig. 1c)	∇	✓	✓
Workload-aware	✓	×	✓
Data-aware	×	✓	✓
Irregular shape partition	×	×	✓

Table I summarizes the properties of methods appeared in Figure 1. Although our solution focuses on the second scenario (Figure 1b), it can also be adapted to the first scenario (Figure 1a) and the third scenario (Figure 1c).

### B. Our challenges and contributions

Our key challenges in this paper are as follows. (1) How to generalize the historical workload to future workloads? (2) How to utilize the generalized workload to produce an efficient partition layout? (3) How to exploit irregular shape partitions to boost the query performance?

To tackle the above challenges, we make the following contributions:

- To avoid overfitting, we formulate a data partitioning problem that considers the generalization of the historical workload.
- We propose Partitioning Aware of Workload Variance (PAW), a workload-aware partition technique that is robust to future queries. It exploits irregular shape partitions to reduce the query cost.

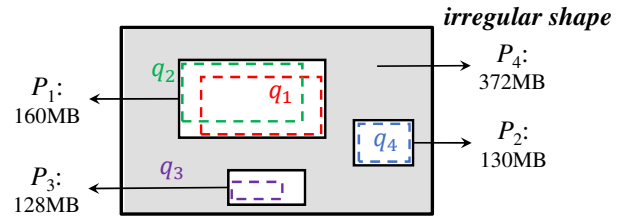


Fig. 3: An example of using irregular shape partitions for historical queries.

- We develop two plugin optimizations to further boost the query performance.
- Extensive experiments are conducted to validate the performance of the proposed methods. PAW outperforms the state-of-the-art method Qd-tree [1] by up to 70×.

The rest of the paper is organized as follows. We first review the related work in Section II. Next, we introduce preliminaries in Section III. Then, we present our PAW partitioning algorithms in Section IV, followed by two plugin optimizations in Section V. Lastly, we present our experiments in Section VI and conclude in Section VII.

## II. RELATED WORK

Data partitioning is common in databases and big data processing systems. We first review existing partitioning techniques in Section II-A, then discuss partition pruning in Section II-B.

### A. Partitioning

We classify existing partitioning techniques into three categories (i.e., independent, data-aware, and workload-aware).

**Independent partitioning** assigns records to partitions based on a mapping function that does not rely on the data distribution nor query workloads. Range partitioning and hash partitioning are frequently used in database systems [12], [13], Hive [14], and Spark [15].

**Data-driven proposals** mainly utilize traditional indexing to generate partition layouts, such as B-tree [16], k-d tree [11] and R-tree [17]. For example, SpatialHadoop [5] partitions data according to spatial locality with a R-tree like mechanism. Notice these approaches partition the dataset only according to the data distribution and provide relatively stable performance to queries with different focus. However, these methods are inefficient when the query workloads follow certain distribution.

**Workload-aware approaches** exploit the workload to produce partition layouts. Auto-tuning and AutoAdmin methods [18]–[24] attempt to find a good physical design of a database according to a given workload. These physical designs include index selection, compression options, update policy, buffering options and partitioning options for data organization. However such methods may take much time (or iteration) in exploring a huge space of parameters. On the contrary, adaptive physical design methods [25]–[27] generate the physical design incrementally and adaptively with periodic update to amortize the exploration cost. However, all these

methods generate partitions with simple horizontal or hash split on certain selected dimensions, which again may have poor performance when the query workloads involve many dimensions.

Adaptive partitioning techniques such as AQWA [4] and Amoeba [2] consider the scenario that future workloads shift gradually over time. To be adaptive, they continuously update partitions as future queries arrive (in a streaming manner), which imposes expensive overheads on updating partitions. However, their techniques are not efficient in our scenario where the workloads vary within a limited scope (i.e., the dotted scope in Figure 1b). In our problem setting, there is no need to update the descriptors of partitions, provided that the future workload is sufficiently similar to the historical workload.

Sun et al. [28] formulate the “MaxSkip” partitioning problem and prove it NP-hard. They propose a bottom-up clustering approach which iteratively merges unique binary feature vectors that produce the lowest penalty. The experimental study in [1] has demonstrated that Qd-tree [1] outperforms the solution in [28] by at most 61 times. Note that both [1] and [28] have not dealt with the overfitting problem and our scenario (see Figure 1b).

Some techniques use graph-based workload modeling to generate partition layouts [21], [29], [30]. However, such works are used to optimize the cost caused by mutual network transfer or data shuffling, which differ from our cost target (i.e., minimizing data scan).

Recently, machine learning based methods are getting increasingly popular. SageDB [31] provides a vision of future database which replaces most of its components by machine learning alternatives, including the partition layout generator. Both Yang et al. [1] and Hilprecht et al. [32] provide reinforcement learning agents to find suitable partition layout. However the former has similar performance with its heuristic based alternative and the latter can only produce coarse-grained partitioning (e.g., hash).

Ding et al. [33] leverage sideways information generated from analytic queries with join operations to improve partitioning on multi-table scenarios, which is orthogonal to ours (i.e., single dataset / table partitioning) and could be combined with existing partitioning approaches.

### B. Query Evaluation with Partition Pruning

Partition pruning has been frequently used to improve query performance. Specifically, one may maintain some aggregate information such as min, max, count and sum for each dimension in each partition, which is known as small materialized aggregates (SMAs) [34]. Then, the SMAs will determine whether a partition contains the query result or not and directly prune the partitions that have no overlap with the given query. Among all the SMAs, the min-max based aggregate is most popular and have been used in various databases like Oracle [35], PostgreSQL [36] and SQLServer [37] to skip irrelevant partitions. Besides these industry databases, many research

works [1], [4], [28], [38] also design their algorithms based on partition pruning.

## III. PRELIMINARIES

### A. Problem Definition

In this paper, we consider multi-dimensional range queries on a dataset  $\mathcal{D}$  with dimensionality  $d_{max}$ . According to [1], [2], SQL queries with unary predicates (in the WHERE clause) can be converted to range queries. We shall elaborate the details in Section III-B.

A partition layout  $\mathbb{P}$  is a collection of disjoint partitions (say,  $P_1, P_2, \dots, P_n$ ) so that their union equals to the dataset  $\mathcal{D}$ . Each partition  $P_i$  has a descriptor to indicate the ranges of records stored in  $P_i$ .

Many block-based storage systems (e.g., HDFS, Amazon S3, Databricks) impose a minimal size  $b_{min}$  on a block (e.g., 128MB in HDFS). We adopt this constraint so that the generated partitions will not be too small. Thus, we require the size of each partition to be at least  $b_{min}$ . As a remark, in this paper we only care about logical partitions (i.e., which partition to place for each record). We leave the underlying storage layer to decide the physical placement of blocks and the replication policy.

The I/O cost of a query  $q$  on a partition layout  $\mathbb{P}$ , denoted by  $Cost(\mathbb{P}, q)$ , is defined as the total size of the partitions in  $\mathbb{P}$  that intersect with  $q$ . Then, we define the I/O cost of a query workload  $Q$  (i.e., a set of queries) on  $\mathbb{P}$  by summing the cost of each query in  $Q$ .

$$Cost(\mathbb{P}, q) = \sum_{p_i \in \mathbb{P}, p_i \cap q \neq \emptyset} p_i.size \quad (1)$$

$$Cost(\mathbb{P}, Q) = \sum_{q \in Q} \sum_{p_i \in \mathbb{P}, p_i \cap q \neq \emptyset} p_i.size \quad (2)$$

Generalization is the key to address overfitting. Our idea is to generalize the historical workload  $Q_H$  into a collection of possible workloads that are sufficiently similar to  $Q_H$ . Based on this idea, we formulate the distance between the historical workload  $Q_H$  and the future workload  $Q_F$ . First, we define the distance between two queries as the maximal difference between them in any dimension. This definition is inspired by the  $L_\infty$  norm in the vector space.

*Definition 1 (Distance between two queries):* The distance between two queries  $q_i$  and  $q_j$ , denoted by  $dist_{\square}(q_i, q_j)$ , is defined as

$$dist_{\square}(q_i, q_j) = \max_{d=1..d_{max}} \max(|q_i.l_d - q_j.l_d|, |q_i.u_d - q_j.u_d|), \quad (3)$$

where  $q_i.l_d$  and  $q_i.u_d$  denote the lower and upper values of  $q_i$  on the  $d$ -th dimension.

Next, we propose the following test on whether two workloads  $Q_H$  and  $Q_F$  are similar. This definition allows different workload sizes, but requires  $|Q_F|$  to be divisible by  $|Q_H|$ .

*Definition 2 ( $\delta$ -similar workloads):* Let  $\delta$  be a distance threshold. Two workloads  $Q_H$  and  $Q_F$  are said to be  $\delta$ -similar if there exists a matching  $\mathcal{M} \subset Q_F \times Q_H$  such that: (i) each pair  $(q'_i, q_j) \in \mathcal{M}$  satisfies  $dist_{\square}(q'_i, q_j) \leq \delta$  (ii) each  $q'_i \in Q_F$

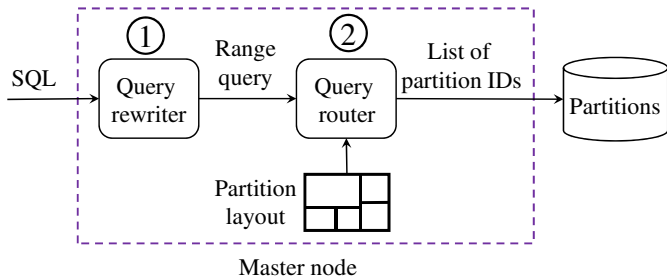


Fig. 4: The query framework of PAW

appears exactly once in  $\mathcal{M}$ , and (iii) each  $q_j \in Q_H$  appears exactly  $\frac{|Q_F|}{|Q_H|}$  times in  $\mathcal{M}$ .

The last condition guarantees that the matching is not dominated by any particular query in  $Q_H$ . As a remark, the system parameter  $\delta$  should be chosen by system administrators. We also provide some heuristics to estimate  $\delta$  from the historical workload (to be discussed in Section IV-E). We leave for future work the alternative definitions of  $\delta$ -similar workloads.

We now formally define our problem as follows.

**Problem 1 (Partitioning for Worst-Case Future Workload):** Given a historical workload  $Q_H$ , a dataset  $\mathcal{D}$ , and a threshold  $\delta$ , this problem asks for the partition layout  $\mathbb{P}$  such that (i) the size of each partition  $P_i \in \mathbb{P}$  is at least  $b_{min}$ , and (ii) it minimizes the following worst-case cost

$$WCost(\mathbb{P}) = \max_{\delta\text{-similar}(Q_H, Q_F)} \frac{1}{|Q_F|} \cdot Cost(\mathbb{P}, Q_F),$$

for any possible future workload  $Q_F$  that is  $\delta$ -similar to  $Q_H$ . The factor  $\frac{1}{|Q_F|}$  is used to obtain the average cost per query.

Our problem can address overfitting because it considers a collection of possible workloads. It is also challenging because it involves infinite possible instances of the future workload  $Q_F$ .

### B. Query Processing

We illustrate the flow of query processing in Figure 4. The master node (of the cluster) manages the meta information of the partition layout in memory (e.g., descriptors and sizes of partitions).

First, when a SQL query is received, the *query rewriter* converts it into one or more range queries. For example, a SQL query with `WHERE A >= 10 AND B <= 50` corresponds to the range query  $[10, \infty) \times (-\infty, 50]$ . As another example, a SQL query with `WHERE A >= 10 OR B <= 50` can be decomposed into two disjoint range queries  $[10, \infty) \times (-\infty, \infty)$  and  $(-\infty, 10) \times (-\infty, 50]$ .

Second, the *query router* examines the partition layout to find the relevant partitions, computes the union list of partition IDs (in case of multiple sub-queries), and then sends the list of partition IDs to the underlying storage system.

## IV. CONSTRUCTION OF PAW

In this section, we present our techniques to construct the partition layout for Problem 1. First, in Section IV-A,

we simplify our partitioning problem but without sacrificing the query cost. Next, we propose split methods in Sections IV-B and IV-C. Then, we summarize our construction algorithm in Section IV-D. Finally, we discuss how to deal with unknown  $\delta$  in Section IV-E.

### A. Reducing the Problem

Recall that, in Problem 1, the equation of  $WCost(\mathbb{P})$  involves infinite possible instances of the future workload  $Q_F$ . To simplify this problem, we plan to rewrite that equation so that it involves one workload only.

Our idea is to extend each historical query  $q_i \in Q_H$  into a new query  $q_i^*$  by  $\delta$  along all directions, i.e.,

$$[q_i^*.l_d, q_i^*.u_d] = [q_i.l_d - \delta, q_i.u_d + \delta]$$

An example is shown in Figure 5.

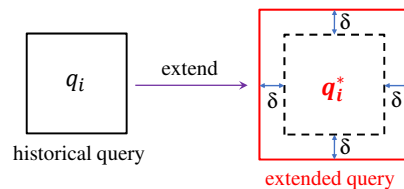


Fig. 5: An example of extended query

Next, we construct the *worst-case workload*  $Q_F^*$  as the collection of  $q_i^*$ . The following lemma shows that  $WCost(\mathbb{P})$  can be expressed by using a single workload ( $Q_F^*$ ).

**Lemma 1:** For any partition  $\mathbb{P}$ , we have:

$$WCost(\mathbb{P}) = \frac{1}{|Q_F^*|} \cdot Cost(\mathbb{P}, Q_F^*)$$

*Proof:* Let  $Q_F$  be a workload such that it is  $\epsilon$ -similar to  $Q_H$ . We first consider the case that  $Q_F$  and  $Q_H$  have the same size, so that we ignore the factor  $\frac{1}{|Q_F^*|}$  in the equation of  $WCost(\mathbb{P})$ .

According to Definition 2, there exists one-to-one matching  $\mathcal{M}$  between  $Q_F$  and  $Q_H$  such that each pair  $(q'_i, q_j) \in \mathcal{M}$  satisfies  $dist_{\square}(q'_i, q_j) \leq \delta$ . Without loss of generality, we assume that  $q_i \in Q_H$  corresponds to  $q'_i \in Q_F$  in the matching  $\mathcal{M}$ .

Observe that  $q_i^*$  (in  $Q_F^*$ ) contains the maximal query range with respect to  $dist_{\square}(q_i^*, q_i) = \delta$ . Thus, we conclude that  $q_i^*$  contains  $q'_i$ . Next, we get  $Cost(\mathbb{P}, Q_F^*) \geq Cost(\mathbb{P}, Q_F)$ , and then obtain:  $WCost(\mathbb{P}) = Cost(\mathbb{P}, Q_F^*)$ .

The above arguments can be generalized to  $Q_F$  with other sizes, except that each query in  $Q_H$  corresponds to  $\frac{|Q_F|}{|Q_H|}$  queries in  $Q_F$ . ■

In subsequent sections, we shall use the worst-case workload  $Q_F^*$  to build partitions.

### B. Partitioning with Irregular Shapes

In this section, we exploit irregular shapes in data partitioning to lower the query cost by reducing the sizes of frequently-queried regions as much as possible.

In subsequent examples, we begin from an initial partition  $P_o$  and denote the worst-case workload as  $Q_F^* = \{q_1^*, q_2^*, \dots\}$ .

First, we examine an ideal case in which the queries in  $Q_F^*$  are mutually disjoint. Figure 6a shows the initial partition  $P_o$  and the queries in  $Q_F^*$ . Our idea is to divide  $P_o$  into four partitions as shown in Figure 6b.  $P_1, P_2, P_3$  can be used to answer  $q_1^*, q_2^*, q_3^*$ , respectively. The remaining records are placed in an irregular shape partition ( $P_4$ ).

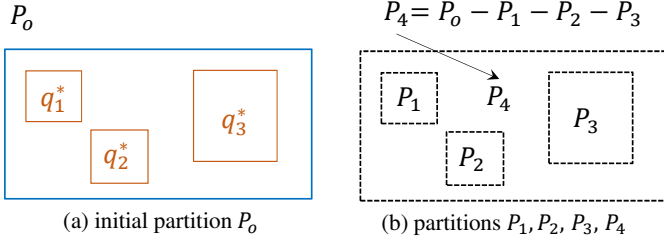


Fig. 6: Optimal partitions for disjoint queries

Next, we consider a more complicated case in which some queries in  $Q_F$  intersect, as shown in Figure 7a. Our idea is to merge intersecting queries (e.g.,  $q_1^*, q_4^*, q_5^*$ ) into a *minimum bounding rectangle* (MBR), then create a grouped partition (e.g.,  $GP_1$ ) for records in that MBR. With this idea, we obtain three grouped partitions ( $GP_1, GP_2, GP_3$ ) and an irregular partition ( $IP$ ) in Figure 7b. Recall that our partitioning problem imposes a constraint on the size of each partition (i.e., at least  $b_{min}$ ). If any partition has a size below  $b_{min}$ , then we expand it in order to satisfy the size constraint, like in Figure 7c.

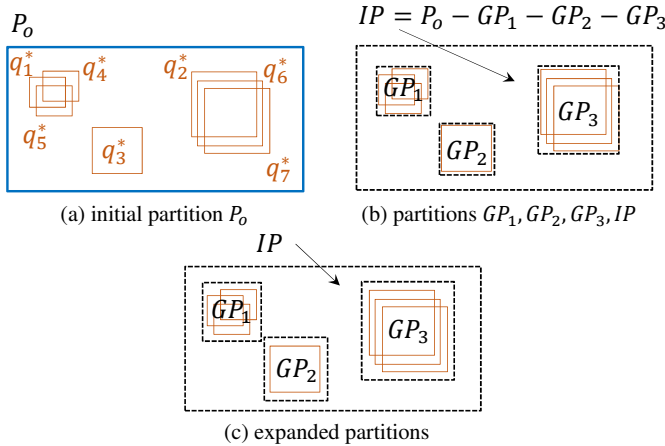


Fig. 7: Multi-Group Split

We elaborate how to expand a partition in Figure 8. Conceptually, a rectangular region  $GP$  can be expressed as  $GP.\vec{c} \pm GP.\vec{r}$ , where the center vector  $GP.\vec{c}$  and the radius vector  $GP.\vec{r}$  are defined as follows.

$$GP.\vec{c} = \left[ \frac{GP.l_d + GP.u_d}{2} \right]_{d=1..d_{max}}$$

$$GP.\vec{r} = \left[ \frac{GP.u_d - GP.l_d}{2} \right]_{d=1..d_{max}}$$

With these concepts, the region  $GP$  can be enlarged by a factor  $f$  along all dimensions via the following expression.

$$GP' = GP.\vec{c} \pm f \cdot GP.\vec{r}$$

For example, in Figure 8, the region  $GP'$  is obtained by enlarging  $GP$  with the factor  $f = 1.5$ .

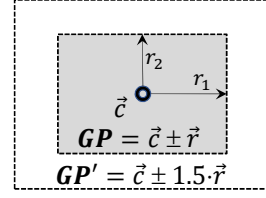


Fig. 8: Expand a partition

Our next step is to find the minimal factor  $f$  such that the size of  $GP'$  is at least  $b_{min}$ . Given a record  $x$ , we define its *relative position* in region  $GP$  as follows.

$$F_{GP}(x) = \max_{d=1..d_{max}} \frac{|x_d - GP.\vec{c}_d|}{GP.\vec{r}_d}$$

Consider the initial partition  $P_o$  in which  $GP$  is split from. We sort the records in  $P_o$  in the ascending order of  $F_{GP}(x)$ , then follow this order to assign records to  $GP$  until satisfying the size constraint  $b_{min}$ .

We summarize our split method in Algorithm 1. At lines 2-4, we create grouped partitions, like in Figure 7b. When the size of a grouped partition  $GP_i$  is below  $b_{min}$ , we expand it by using the aforementioned idea (at line 5). At line 7, we test whether all grouped partitions are mutually disjoint. If so, then we return them together with an irregular partition  $IP$  as the result. Otherwise, an empty result is returned to indicate a failure split.

---

#### Algorithm 1 Multi-Group Split

---

**Input:** partition  $P_o$ , workload  $Q_F^*$ , min block size  $b_{min}$

- 1:  $\mathbb{P} \leftarrow \emptyset$
- 2: check the intersection relationships of queries in  $Q_F^*(P_o)$
- 3: **for** each group of intersecting queries in  $Q_F^*(P_o)$  **do**
- 4:   create a grouped partition  $GP_i$
- 5:   expand  $GP_i$  until  $GP_i.size \geq b_{min}$
- 6:   insert  $GP_i$  to  $\mathbb{P}$
- 7: **if**  $\forall GP_i, GP_j \in \mathbb{P}, GP_i \cap GP_j = \emptyset$  **then**
- 8:    $IP \leftarrow P_o - \bigcup_{GP_i \in \mathbb{P}} GP_i$
- 9:   **if**  $IP.size \geq b_{min}$  **then**
- 10:     **return**  $\mathbb{P} \cup \{IP\}$
- 11: **return**  $\emptyset$

---

#### C. Rectangular Split

In some cases, rectangular partitions may perform better than irregular partitions. Therefore, we introduce this complementary query-driven partition technique. To illustrate this, we consider the example in Figure 9, where the minimum block size  $b_{min}$  is 128 MB. Observe that  $q_1^*$  and  $q_2^*$  have partial intersection, so the MBR of these two queries is huge.

Applying the method in Section IV-B on this example, the solution degenerates into a single partition  $P$ , as shown in

Figure 9a. The query cost can be reduced to 465MB if we divide  $P$  into two irregular partitions as shown in Figure 9b. Interestingly, if we employ only rectangular partitions like in Figure 9c, the query cost can be further reduced to 420MB.

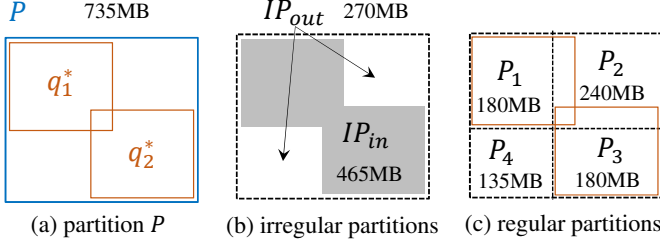


Fig. 9: A case where rectangular partitions perform better

As such, we also consider existing query-driven partitioning methods that are axis-parallel and produce rectangular partitions. Specifically, for each dimension  $d$ , we consider a possible split at (i) the median of  $P$  in this dimension, and (ii) the lower and upper values of each query in  $Q_F^*(P)$ . Observe that (ii) has also been used in the Qd-tree [1].

We summarize this split method in Algorithm 2.

#### Algorithm 2 Axis-Parallel Split

---

**Input:** partition  $P$ , workload  $Q_F^*$ , min block size  $b_{min}$

- 1: let  $\mathcal{APS}$  be the set of axis-parallel splits
- 2:  $\mathbb{P}_{best} \leftarrow \emptyset$ ;  $cost_{best} \leftarrow \infty$
- 3: **for**  $split_i$  in  $\mathcal{APS}$  **do**
- 4:    $\mathbb{P}' \leftarrow$  divide  $P$  by using  $split_i$
- 5:   **if**  $\forall p_i \in \mathbb{P}', p_i.size \geq b_{min}$  **then**
- 6:     **if**  $cost_{best} > Cost(\mathbb{P}', Q_F^*(P))$  **then**
- 7:        $\mathbb{P}_{best} \leftarrow \mathbb{P}'$ ;  $cost_{best} \leftarrow Cost(\mathbb{P}', Q_F^*(P))$
- 8: **return**  $\mathbb{P}_{best}$

---

#### D. The Construction Algorithm

We propose a recursive algorithm to construct partitions (see Algorithm 3). Its parameters include: (i) the initial partition  $P_o$ , (ii) the worst-case workload  $Q_F^*$  as described in Section IV-A, and (iii) the split policy  $\Psi$  which will be discussed later. In the root recursive call,  $P_o$  is initialized to the MBR of the dataset  $\mathcal{D}$ . In lines 1-6, we apply each split function  $SF_i$  on the initial partition  $P_o$ , measure the cost of the resulting partition layout  $\mathbb{P}'$ , and update the best layout  $\mathbb{P}_{best}$  when  $\mathbb{P}'$  yields a lower cost. If  $\mathbb{P}_{best}$  is better than  $P_o$ , then we replace  $P_o$  by  $\mathbb{P}_{best}$  and invoke a recursive call on each partition in  $\mathbb{P}_{best}$ . However if  $P_i$  is irregular shape, the partitioning process terminates once it is generated as an irregular shape partition contains no queries in  $Q_F^*$  and no splits could reduce its cost if it is already 0.

Note that, at line 2, the split policy  $\Psi$  is used to decide the list of applicable split methods for  $P_o$ . Ideally, both irregular split (Alg. 1) and rectangular split (Alg. 2) should be tried at line 2, in order to optimize the query cost. However, Alg. 1 is time-consuming and it is effective on small to medium sized partitions. To control the trade-off between the quality and

construction time, we recommend to use the following split policy  $\Psi$ :

$$\Psi(P_o) = \begin{cases} \{\text{Alg. 1, Alg. 2}\}, & \text{if } P_o.size \geq \alpha \cdot b_{min} \\ \{\text{Alg. 2}\} & \text{if } P_o.size \geq 2 \cdot b_{min} \\ \emptyset & \text{otherwise} \end{cases} \quad (4)$$

where  $\alpha$  is a constant larger than 1. We shall test the effect of  $\alpha$  in our experimental study.

---

#### Algorithm 3 PAW-Construction

---

**Input:** the initial partition  $P_o$ , the worst-case workload  $Q_F^*$ , the split policy  $\Psi$

- 1:  $\mathbb{P}_{best} \leftarrow \emptyset$ ;  $cost_{best} \leftarrow \infty$
- 2: **for**  $SF_i$  in  $\Psi(P_o)$  **do**
- 3:    $\mathbb{P}' \leftarrow SF_i(P_o)$
- 4:   **if**  $\mathbb{P}' \neq \emptyset$  **then**
- 5:     **if**  $Cost(\mathbb{P}', Q_F^*(P_o)) < cost_{best}$  **then**
- 6:        $\mathbb{P}_{best} \leftarrow \mathbb{P}'$ ;  $cost_{best} \leftarrow Cost(\mathbb{P}', Q_F^*(P_o))$
- 7: **if**  $cost_{best} < Cost(P_o, Q_F^*(P_o))$  **then**
- 8:   **for**  $P_i$  in  $\mathbb{P}_{best}$  **do**
- 9:     add  $P_i$  as a child of  $P_o$
- 10:   PAW-Construction( $P_i, Q_F^*, \Psi$ )

---

Figure 10 demonstrates some running steps of our construction algorithm. The relationships between different recursive calls (and partitions) can be visualized as the *partition tree*, in Figure 10d. The leaf nodes correspond to the final partitions  $P_A, P_B, P_C, P_D$  in the storage system. The partitions for non-leaf nodes are not stored physically; instead, we only maintain their descriptors in the master node in order to facilitate query routing.

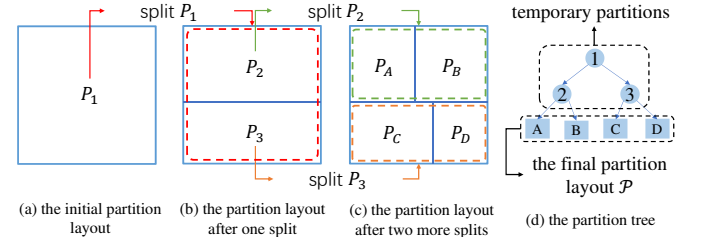


Fig. 10: Example of PAW construction

As a remark, Algorithm 3 is a greedy heuristics algorithm; it makes locally optimal choices but does not guarantee the globally optimal solution. Alternative search strategies (e.g., beam search [39]) may be applied to broaden the search space and revise Algorithm 3. We leave this issue for future work.

#### E. Dealing with Unknown $\delta$

In real applications, the distance threshold  $\delta$  can be unknown. We propose a simple heuristics to estimate the threshold by using the historical workload  $Q_H$ . First, we simulate the past and future by dividing  $Q_H$  equally into two workloads  $Q_{H1}$  and  $Q_{H2}$ , according to their timestamps in the system. Next, we compute the minimal value (say,  $\delta'$ ) such that  $Q_{H1}$  and  $Q_{H2}$  are  $\delta'$ -similar (see Definition 2). Then, we use  $\delta'$  as the distance threshold.

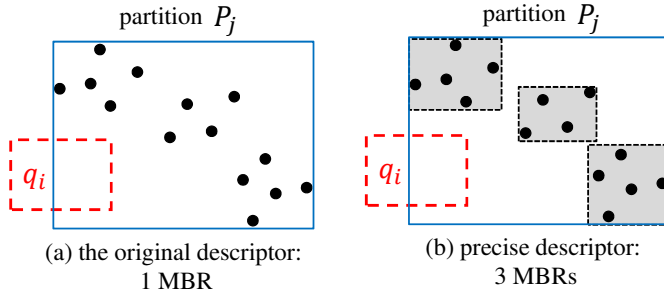


Fig. 11: Descriptors for a partition  $P_j$

Note that Algorithm 3 stops partitioning when a region does not intersect with any query in  $Q_F^*$ . Some partitions can be huge and may cause performance deterioration if some future queries partially intersect with them. This happens when the estimated  $\delta'$  is less than the real  $\delta$ . To alleviate this issue, we keep split the leaves of a partition tree by data-aware partitioning (e.g.,  $k$ -d tree partitioning) until reaching the finest size (i.e.,  $[b_{min}, 2b_{min})$ ).

## V. PLUGIN MODULES

We shall present two plugin modules to boost the query performance, after constructing the partition layout. They are orthogonal to the techniques proposed in Section IV.

### A. The Precise Descriptor Module

This module represents partitions by using more precise descriptors in order to enhance the pruning power during query processing.

As an example, consider the partition  $P_j$  in Figure 11, where black dots denote the actual records stored in  $P_j$ . We compare the original descriptor of  $P_j$  (in Figure 11a) and the precise descriptor of  $P_j$  (in Figure 11b), by using the query  $q_i$ . The original descriptor of  $P_j$  is just the MBR of  $P_j$ , which intersects with the query  $q_i$ . Thus, we still need to scan the data stored in  $P_j$ . In Figure 11b, we use a more precise descriptor of  $P_j$ , i.e., a set of 3 gray MBRs that collectively cover all records in  $P_j$ . Since  $q_i$  does not intersect with any gray MBR, we avoid scanning the data stored in  $P_j$ .

Existing data partitioning algorithms may be used to construct the above precise descriptor. In our implementation, we adopt the R-tree construction algorithm to extract a given number of MBRs (say  $N_{mbr}$ ) from a partition. Like the partition layout, the precise descriptors of partitions are stored in the memory in the master node (cf. Figure 4). During query processing, the master node checks whether a query (say  $q_i$ ) intersects with any MBR in the precise descriptor of partition  $P_j$ . If there is no intersection, we skip the scanning of  $P_j$ .

As a remark, a precise descriptor occupies only  $16 \cdot d_{max} \cdot N_{mbr}$  bytes, where  $N_{mbr}$  is the number of MBRs in the precise descriptor, and  $d_{max}$  is the number of dimensions. The extra memory required is very small compared to the size of a partition (e.g., 128MB).

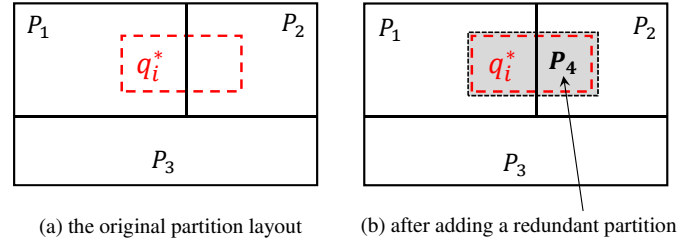


Fig. 12: Plugin redundant partition

### B. The Storage Tuner Module

This module exploits the available storage space to construct extra partitions in order to reduce the query cost. Obviously, these extra partitions overlap with some partitions in the original partition layout  $\mathbb{P}$ .

We proceed to illustrate our idea. In Figure 12a, the original partition layout contains three partitions  $P_1, P_2, P_3$ . A query  $q_i^* \in Q_F^*$  intersects with both  $P_1$  and  $P_2$ . Thus, the I/O cost of  $q_i^*$  is  $P_1.size + P_2.size$ . In Figure 12b, suppose that the gray partition  $P_4$  is included as an extra partition. Note that  $P_4$  can be used to answer  $q_i^*$  at a lower cost.

Next, we elaborate how to construct extra partitions, subject to the available storage space. For each query  $q_j^*$  in the worst-case workload  $Q_F^*$ , we define its extra partition  $RP_j$  as the result size of  $q_j^*$ . Then, the gain of  $RP_j$  is defined as follows.

$$Gain(RP_j) = \frac{\sum_{q_i^* \in Q_F^*, q_i^* \subseteq RP_j} (Cost(\mathbb{P}, q_i^*) - RP_j.size)}{RP_j.size} \quad (5)$$

We implement a greedy algorithm to select extra partitions in descending order of  $Gain(RP_j)$ , until occupying all the available storage space. When a partition  $RP_j$  is selected, we include it into  $\mathbb{P}$  and update the gain of any other extra partition.

At the query time, the master node first checks whether the query lies within some extra partitions. If yes, then we consult the corresponding extra partition. Otherwise, the query is processed with the original PAW partition layout.

## VI. EXPERIMENTAL EVALUATION

We conduct extensive experiments to evaluate the performance of PAW. Our key findings are highlighted as follows:

- PAW incurs much lower I/O cost than Qd-tree ( $2\times$  to  $70\times$  better).
- The I/O cost of PAW is only within  $1.5\times$  of the theoretical lower bound cost in most of the tested cases.

### A. Experimental Setting

**Platform.** Experiments are conducted on a 4-node Spark cluster connected by LAN. Each node has 8 vCPUs (Intel Skylake, 2.6GHz), 16GB RAM, and 1TB HDD. All data is stored in the Parquet file format in HDFS. For each method, we measure the average end-to-end response time (per query) and the average I/O cost (per query).

**Methods.** We compare our proposed method PAW with two existing methods: Qd-tree [1] and  $k$ -d tree [11]. For PAW, we disable our plugin modules in Section V by default. For Qd-tree [1], we implement its greedy version because it is deterministic and yields comparable performance (to the reinforcement learning variant) in our experiments. For  $k$ -d tree [11], we implement the standard version which chooses split dimensions by round-robin and splits from the median. In addition, we also compare with the theoretical lower bound cost, denoted by LBCost, which scans exactly the query result.

**Datasets.** The TPC-H benchmark<sup>2</sup>, with scale factor, is used to generate the `lineitem` table. This table contains 600 million records (75GB) and 8 numerical attributes. We observe that the generated records are uniformly distributed.

We also use a real dataset OSM [40], which contains 100 million records and 2 numerical attributes (latitude and longitude). The records in this dataset follow a skewed distribution.

Following Qd-tree [1], we use a sampled subset (fixed to 6M rows from the dataset) to generate the logical partition layout, then route the full dataset to its corresponding partitions and write each partition to disk. In Table II, we show the breakdown of construction time for 3 different TPC-H dataset sizes (i.e., 8GB, 38GB, and 75GB). The logical partition layout could be generated quickly due to sampling, while routing (i.e., distribute records to partitions) and I/O dominate 90%–99% of the construction time. Therefore, all methods have similar construction time on the same dataset size. It takes approximately 2.4 hours and 0.2 hours to construct partitions on TPC-H (75GB) and OSM, respectively.

TABLE II: Partition construction time for the TPC-H dataset with 3 sizes (8 GB, 38GB, and 75GB)

Methods	Layout generation time (s)	Routing and I/O time (s)		
		8GB	38GB	75GB
Qd-tree	8.68	730	3794	7899
$k$ -d tree	3.43	754	3594	8703
PAW	22.09	737	3880	8769

**Query workloads.** Following the literature [4], [5], [20], [22], [29], we implement the following two generators to produce the historical workload  $Q_H$ . The uniform generator generates historical queries according to the data domain. The skewed generator picks a fixed number of centers and then generates historical queries by the Gaussian mixture model.

The future workload  $Q_F$  is generated randomly within a given distance threshold from  $Q_H$ . By default, we use a total of 100 queries; 50% in  $Q_H$  and 50% in  $Q_F$ . The parameters for generating the above workloads are shown in Table III.

### B. A Case Study on Partition Layouts

Before diving into detailed comparisons, we visualize the partition layouts generated by different methods. For the sake of visualization, we plot Figures 13 and 14 in the 2-dimensional space. Partition boundaries are shown as green lines, whereas query regions are drawn as red boxes.

<sup>2</sup><http://www.tpc.org/tpch/>

TABLE III: Default query properties

Property	Default value
Common workload generator properties	
# $Q$ : the number of queries	100
#dims: the number of dimensions in queries	4
$\delta$ : the distance threshold	1% of domain length
$\gamma$ : the maximal query range	10% of domain length
Skewed workload generator properties	
# $C$ : the number of query centers	10
$\sigma$ : the standard deviation	10% of $\gamma$

For  $k$ -d tree, the historical workload (Figure 13a) and the future workload (Figure 14a) have similar query cost. The reason is because each partition has been split to its minimum size (i.e.,  $[b_{min}, 2b_{min})$ ). However, a query may partially intersect some partitions and waste I/O cost on scanning records that are not in the query result.

Qd-tree produces the partitions based on the borders of historical queries (see Figure 13b). This could optimize the query cost with respect to the historical workload. However, slightly different future queries may deteriorate the query cost significantly (see Figure 14b).

PAW extends historical queries and applies multi-group splits. Thus, it produces just the right partitions to tolerate variances in the future workload. Each future query is highly likely to fall into a single partition in Figure 14c.

### C. Scalability on TPC-H

In Figure 15, we vary the data size of TPC-H, then plot the average I/O cost (per query) and the average end-to-end query response time (per query). in the Spark cluster. PAW outperforms Qd-tree by  $10\times$ .

Observe that, when the I/O cost is extremely high, the end-to-end query response time grows sub-linearly. This is attributed to the optimizations implemented in the underlying system (e.g., row group based pruning [41], caching). These optimizations (and engineering tricks) are orthogonal to our problem. In subsequent experiments, we measure the I/O cost only because it is platform independent.

### D. Robustness to Various Parameters

In this subsection, we measure the *scan ratio* as the ratio of the average I/O cost to the dataset size.

**The effect of the number of query dimensions.** In Figure 16, we vary the number of dimensions used in queries, by selecting the first few attributes (from the `lineitem` table in TPC-H) except the primary key. Note that partitions store all the dimensions of records. The I/O cost of PAW is within  $1.5\times$  of the theoretical lower bound cost in all tested cases. Qd-tree incurs the highest I/O cost because the future workload deviates from the historical workload and Qd-tree partitions are more likely to have partial intersections with the future workload. When the number of query dimensions increases, the query selectivity drops and thus all methods yield lower I/O cost in general. The spike in the I/O cost of  $k$ -d tree, from 2 to 3 query dimensions, happens because  $k$ -d tree chooses split



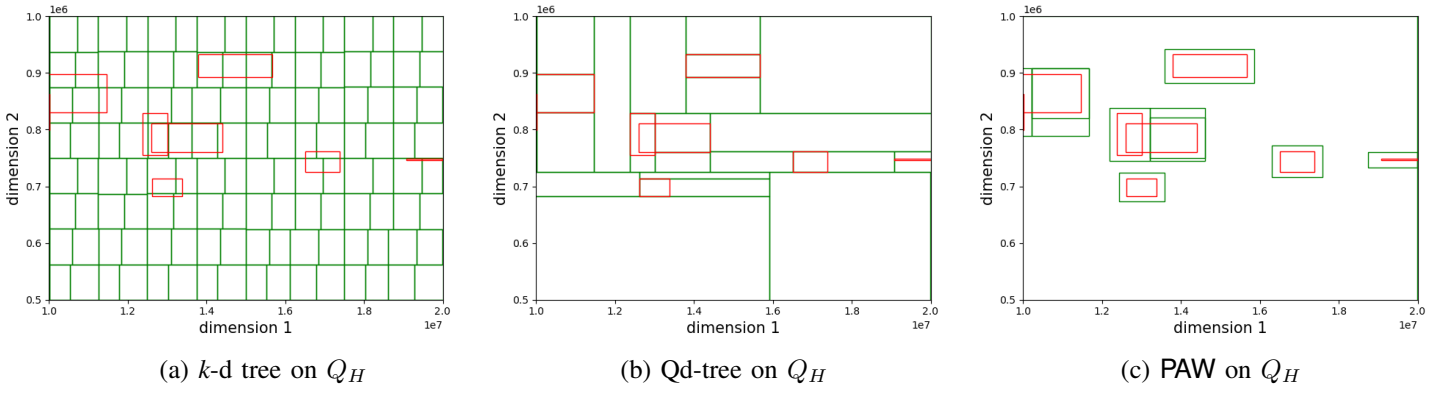


Fig. 13: The historical workload and the partition layout of all methods, on TPC-H

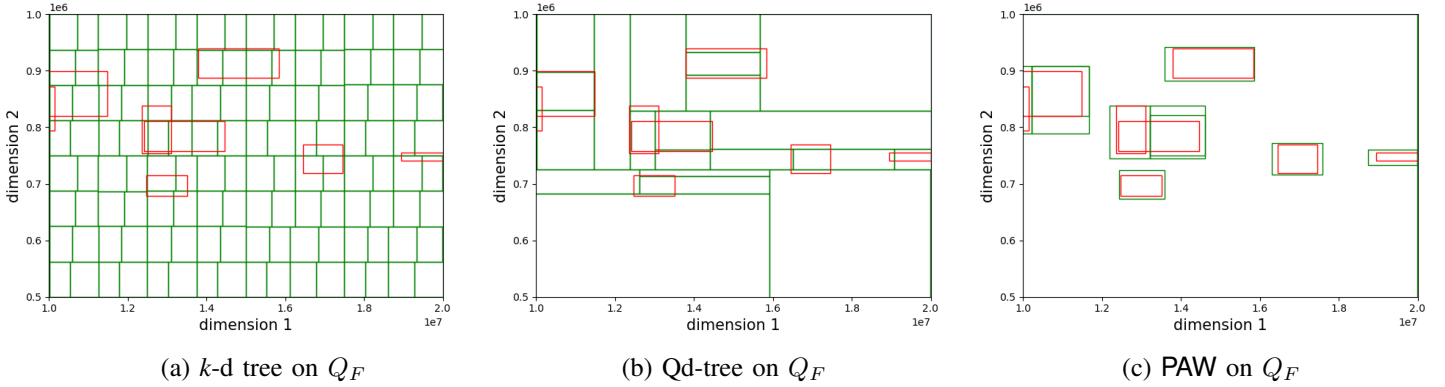


Fig. 14: The future workload and the partition layout of all methods, on TPC-H

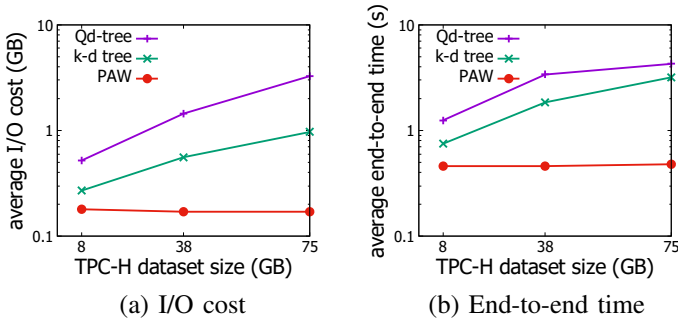


Fig. 15: Average I/O cost and end-to-end time, on TPC-H

dimensions by round-robin and the 3rd chosen split dimension has a much smaller domain than other attributes.

**The effect of the query range.** In Figure 17, we vary the maximal query range. As expected, most of the methods incur higher I/O cost as query range increases. Even when the query range is small, the queries in the future query workload may partially intersect large partitions in Qd-tree, as we have demonstrated in Figure 14b. This phenomenon renders the I/O cost of Qd-tree unnecessary high. PAW remains the best method.

**The effect of workload size.** In Figure 18, we vary the historical workload size. PAW and  $k$ -d tree are not very sensitive to this parameter. The I/O cost of PAW is close to the theoretical lower bound cost. In contrast, Qd-tree behaves differently on different data distributions. For a uniform dataset (i.e., TPC-H), a larger historical workload size implies more partitions in Qd-tree, thus leading to more saving in the I/O cost. For a skewed dataset (i.e., OSM), Qd-tree may compute a pathological layout as shown in Figure 14b, where future queries are likely to intersect with multiple partitions. When the number of historical queries is extremely high, the scan ratio of Qd-tree is close to PAW due to the large number of partitions.

PAW outperforms Qd-tree by 70 $\times$  on TPC-H and 10 $\times$  on OSM.

**The effect of the distance threshold.** In Figure 19, we vary the distance threshold  $\delta$  between the historical workload and the future workload. As expected, a larger  $\delta$  implies a higher I/O cost. Again, PAW is the best method and its I/O cost is close to the theoretical lower bound cost in most of the cases. Note that a larger  $\delta$  could also widen the average query range of future queries, leading to a higher scan ratio of all methods.

**The effect of the workload distribution.** As shown in Figure 20, the I/O cost of most methods rise slowly when we change from the uniform workload to the skewed workload.

For a skewed workload, queries are more likely to have overlapping ranges, rendering it harder to generate efficient partition layouts. The I/O cost of  $k$ -d tree remains stable because its partition layout does not rely on the historical workload.

**The effect of skewed workload parameters.** Next, we test the effect of skewed workload parameters: (i) the number of query centers, and (ii) the standard deviation of query range. In Figure 21a, when the number of query centers rises, the I/O cost begins to drop because more partitions will be produced. However, when the number of query centers becomes too high (e.g., 50), overlapping occurs among many queries, thus leading to higher query cost. Note that PAW is not sensitive to this parameter. Figure 21b plots the I/O cost of methods while varying the standard deviation of the query range. PAW performs the best and it is the least sensitive to this parameter.

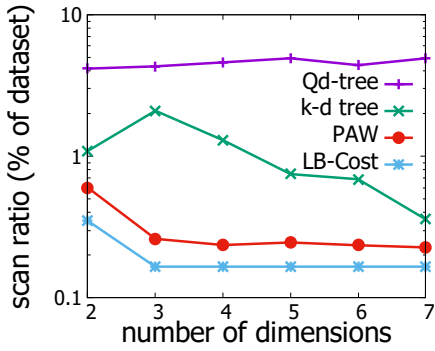


Fig. 16: Average I/O cost on TPC-H, varying the number of query dimensions

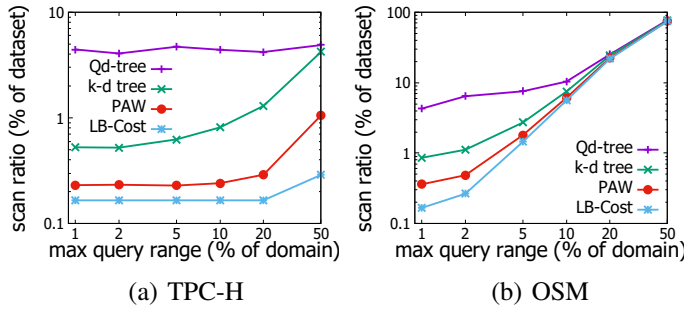


Fig. 17: Average I/O cost on two datasets, varying the maximal query range

### E. The Effect of Unknown Distance Threshold

We proceed to compare two variants of PAW: (i) PAW, which is given the (real) distance threshold  $\delta$ , (ii) PAW-unknown, which estimates  $\delta$  from the historical workload according to Section IV-E. Figure 22a shows the I/O of these two variants of PAW, as well as the theoretical lower bound cost. For uniform workloads, PAW-unknown is worse than PAW by 3 to 4 $\times$ ; nevertheless, the scan ratio (per query) is

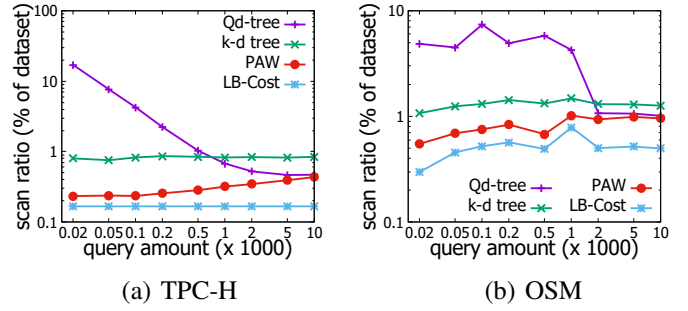


Fig. 18: Average I/O cost on two datasets, varying the number of queries

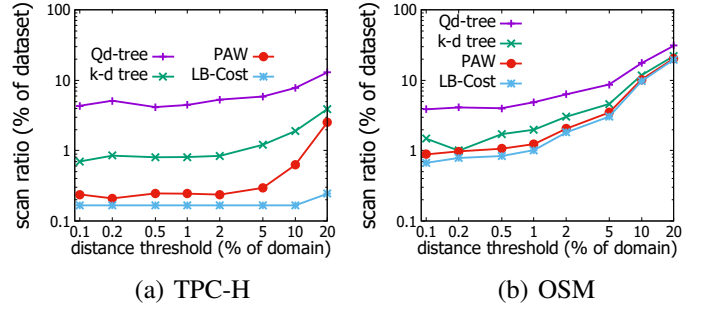


Fig. 19: Average I/O cost on two datasets, varying the distance threshold  $\delta$

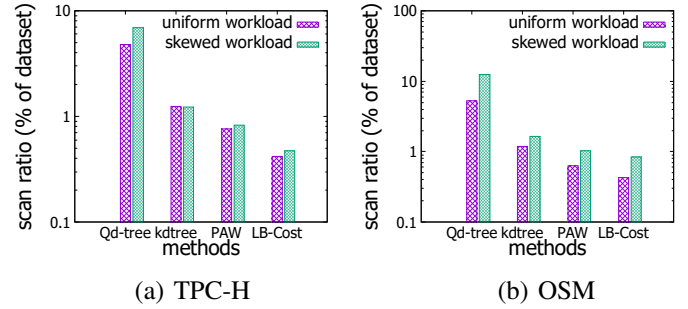


Fig. 20: Average I/O cost on two datasets, varying the distribution of queries

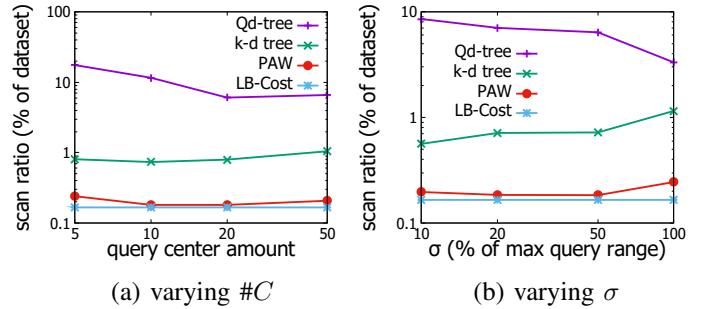


Fig. 21: Average I/O cost on TPC-H, varying the parameters in skewed workload

still below 1% of the entire dataset. For skewed workloads, the performance of PAW-unknown is comparable to PAW.

In the next experiment, we simulate the unpredictable scenario (in Figure 1c) by replacing X% of queries in the future workload with random queries. The historical workload is uniformly distributed. We turn on the data-aware optimization (as discussed in Section IV-E) for PAW. In Figure 22b, we vary the percentage of random queries (in the future workload). Observe that the performance of PAW degrades gracefully from 0% to 100%. In the worst case (i.e., 100% of random), PAW is comparable to a  $k$ -d tree. Thanks to the data-aware optimization in Section IV-E, PAW can support the unpredictable scenario shown in Figure 1c.

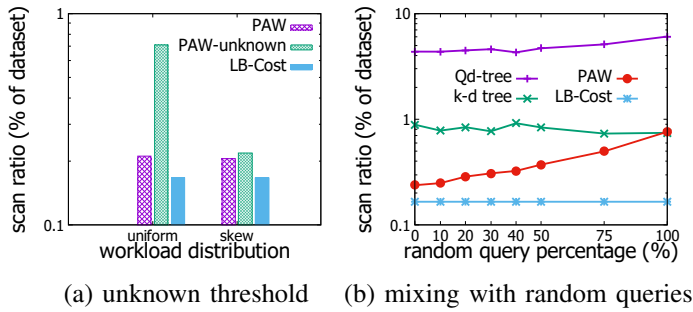


Fig. 22: Average I/O cost on TPC-H dataset, using the techniques in Section IV-E

### F. Experiments on Plugin Modules

In this subsection, we test the plugin modules described in Section V.

First, we enable the precise descriptor plugin in PAW and vary the number of MBRs per precise descriptor. According to Figure 23a, 3 MBRs per precise descriptor would be sufficient to reduce the query I/O cost by 10%.

Second, we enable the storage tuner plugin in PAW and vary the amount of disk space for storing redundant partitions. According to Figure 23b, when we have 20% more extra space, the query I/O cost of PAW drops by 10%.

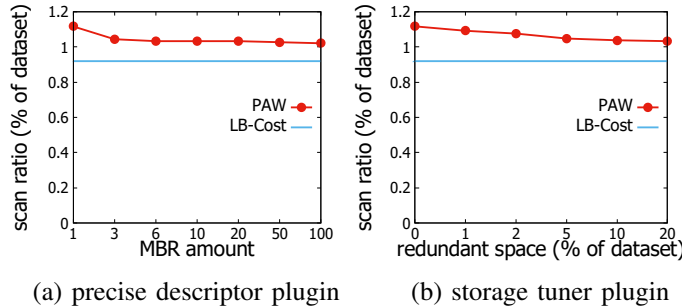


Fig. 23: Average I/O cost on OSM, using plugin modules

### G. Experiments on the Special Case $\delta = 0$

In this subsection, we consider the special case  $\delta = 0$ , which corresponds to the ‘same workload’ scenario in Figure 1a. This is also the scenario where Qd-tree is designed for.

Even in this special case, PAW still outperforms Qd-tree because we exploit irregular shape partitions (in Multi-Group Split). We rerun our experiments on this setting.

Table IV gives the I/O cost and end-to-end time in Spark cluster when  $\delta = 0$  under the default setting. Observe that PAW is 6 $\times$  more efficient than  $k$ -d tree and 1.2 $\times$  more efficient than Qd-tree.

TABLE IV: Query cost on default settings when  $\delta = 0$

Scenario	Measure	$k$ -d tree	Qd-tree	PAW
$\delta = 0$	I/O cost (GB)	0.81	0.18	0.15
$ D  = 75GB$	end-to-end time (s)	3.11	0.63	0.50

In Figure 24a, we vary the number of query dimensions. Similar to the case when  $\delta \neq 0$ , the query cost of all methods decrease, which correspond to the selectivity of queries. In Figure 24b, we vary maximum query range. As expected, all methods increase the cost as query range increases. In Figure 24c, we vary number of historical queries.  $k$ -d tree is still insensitive to this parameter. When there are many queries, the query distribution becomes more similar to the uniform distribution. Thus, the I/O costs of both Qd-tree and PAW increase as the number of queries increases. In Figure 24d, we test the effect of different query distributions. The result is similar to the  $\delta \neq 0$  case.

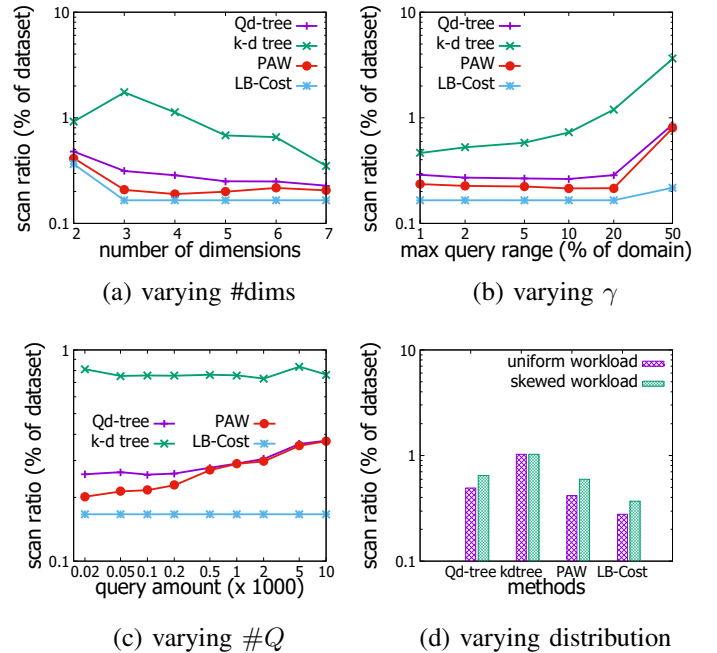


Fig. 24: Average I/O cost on TPC-H dataset, varying query parameters, fixing  $\delta = 0$

In Figure 25, we rerun the plugin optimizations. For precise descriptor plugin (Figure 25a), its performance is similar to

the case when  $\delta \neq 0$  (i.e., Figure 23a). For storage tuner plugin, since now the extra partitions are generated using  $Q_F^*$  equivalently ( $Q_F^* = Q_H$  when  $\delta = 0$ ), the plugin keep reducing the cost for all method with increasing redundant space until the lower bound is met. Notice under the above two plugin optimizations, PAW still achieves the lowest cost and is close to the theoretical lower bound in all cases.

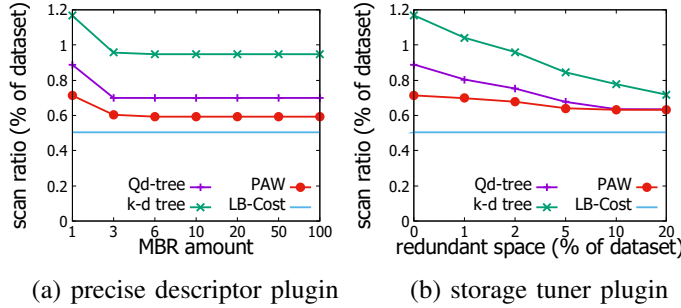


Fig. 25: Average I/O cost on OSM, using plugin modules, fixing  $\delta = 0$

## VII. CONCLUSION

In this paper, we study the workload-aware partitioning problem that aims at reducing the query cost for any unknown future workload  $Q_F$ . Although existing works assume  $Q_F$  are exactly the same as the historical workload  $Q_H$ , we allow them to be slightly different (within a distance threshold). We formally define such concept  $\delta$ -similar workload and show that the worst case query cost is achieved when  $Q_F = Q_F^*$ , which could be derived from the historical queries  $Q_H$ . Our proposed partitioning based on  $Q_F^*$  allows us to avoid overfitting, causing the future workload to intersect with fewer partitions in PAW than in the Qd-tree (see Figure 14).

In order to further reduce the query cost on  $Q_F^*$ , we propose the Multi-Group Split partition methods to maximize the pruning of non-queried regions, which is also the first partition method that exploits irregular shape partitions and non-axis parallel splits. We combine Multi-Group Split with existing split methods to form a general partition framework PAW. Our experiment results show that PAW could be 10X faster than the state-of-the-art method in query response time in default settings and up to 70X more efficient in certain workload conditions.

In the future, we could have the following explore directions. (1) Algorithms 1 and 2 are based on range queries. How to support more SQL and analytic query operations (e.g., KNN) that could benefit from partitioning? (2) Other than the I/O cost, how to take the storage layer's data placement and network latency issues into one cost model and generate partitions according to it? (3) When more split functions are considered, how to automatically determine their apply conditions?

## REFERENCES

- [1] Z. Yang, B. Chandramouli, C. Wang, J. Gehrke, Y. Li, U. F. Minhas, P. Larson, D. Kossmann, and R. Acharya, "Qd-tree: Learning data layouts for big data analytics," in *SIGMOD*, 2020, pp. 193–208.
- [2] A. Shanbhag, A. Jindal, S. Madden, J. Quiané-Ruiz, and A. J. Elmore, "A robust partitioning scheme for ad-hoc query workloads," in *SoCC*, 2017, pp. 229–241.
- [3] A. M. Aly, H. Elmeleegy, Y. Qi, and W. G. Aref, "Kangaroo: Workload-aware processing of range data and range queries in hadoop," in *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, 2016, pp. 397–406.
- [4] A. M. Aly, A. R. Mahmood, M. S. Hassan, W. G. Aref, M. Ouzzani, H. Elmeleegy, and T. Qadah, "AQWA: adaptive query-workload-aware partitioning of big spatial data," *PVLDB*, vol. 8, no. 13, pp. 2062–2073, 2015.
- [5] A. Eldawy and M. F. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data," in *ICDE*, 2015, pp. 1352–1363.
- [6] A. Bozzon, M. Brambilla, S. Ceri, and D. Mazza, "Exploratory search framework for web data sources," *Vldb J.*, vol. 22, no. 5, pp. 641–663, 2013.
- [7] M. Singh, M. J. Cafarella, and H. V. Jagadish, "Dbexplorer: Exploratory search in databases," in *EDBT*, 2016, pp. 89–100.
- [8] B. Qarabaqi and M. Riedewald, "Merlin: Exploratory analysis with imprecise queries," *IEEE TKDE*, vol. 28, no. 2, pp. 342–355, 2016.
- [9] O. B. El, T. Milo, and A. Somech, "Automatically generating data exploration sessions using deep reinforcement learning," in *SIGMOD*, 2020, pp. 1527–1537.
- [10] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, <http://www.deeplearningbook.org>.
- [11] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975.
- [12] IBM, "Multidimensional clustering tables," [https://www.ibm.com/support/producthub/db2/docs/content/SSEPGG\\_11.5.0.com.ibm.db2.luw.admin.partition.doc/doc/c0021560.html](https://www.ibm.com/support/producthub/db2/docs/content/SSEPGG_11.5.0.com.ibm.db2.luw.admin.partition.doc/doc/c0021560.html).
- [13] MySQL, "Subpartitioning," <https://dev.mysql.com/doc/mysql-partitioning-excerpt/8.0/en/partitioning-subpartitions.html>.
- [14] Hive, <https://hive.apache.org/>.
- [15] Spark, "Parquet Files," <https://spark.apache.org/docs/latest/sql-data-sources-parquet.html>.
- [16] D. Comer, "The ubiquitous b-tree," *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, 1979.
- [17] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD*, 1984, pp. 47–57.
- [18] S. Agrawal, N. Bruno, S. Chaudhuri, and V. R. Narasayya, "Autoadmin: Self-tuning database systems technology," *IEEE Data Eng. Bull.*, vol. 29, no. 3, pp. 7–15, 2006.
- [19] J. Rao, C. Zhang, N. Megiddo, and G. M. Lohman, "Automating physical database design in a parallel database," in *SIGMOD*, 2002, pp. 558–569.
- [20] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki, "Slalom: Coasting through raw data via adaptive partitioning and indexing," *PVLDB*, vol. 10, no. 10, pp. 1106–1117, 2017.
- [21] A. Pavlo, C. Curino, and S. B. Zdonik, "Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems," in *SIGMOD*, 2012, pp. 61–72.
- [22] E. Wu and S. Madden, "Partitioning techniques for fine-grained indexing," in *ICDE*, 2011, pp. 1127–1138.
- [23] S. Agrawal, V. R. Narasayya, and B. Yang, "Integrating vertical and horizontal partitioning into automated physical database design," in *SIGMOD*, 2004, pp. 359–370.
- [24] M. Athanassoulis, K. S. Bøgh, and S. Idreos, "Optimal column layout for hybrid workloads," *PVLDB*, vol. 12, no. 13, pp. 2393–2407, 2019.
- [25] S. Idreos, M. L. Kersten, and S. Manegold, "Database cracking," in *CIDR*, 2007, pp. 68–78.
- [26] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap, "Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores," *PVLDB*, vol. 5, no. 6, pp. 502–513, 2012.
- [27] N. Bruno and S. Chaudhuri, "An online approach to physical design tuning," in *ICDE*, 2007, pp. 826–835.
- [28] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin, "Fine-grained partitioning for aggressive data skipping," in *SIGMOD*, 2014, pp. 1115–1126.

- [29] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," *PVLDB*, vol. 3, no. 1, pp. 48–57, 2010.
- [30] J. Zhou, N. Bruno, and W. Lin, "Advanced partitioning techniques for massively distributed computation," in *SIGMOD*, 2012, pp. 13–24.
- [31] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan, "Sagedb: A learned database system," in *CIDR*, 2019.
- [32] B. Hilprecht, C. Binnig, and U. Röhm, "Towards learning a partitioning advisor with deep reinforcement learning," in *aiDM@SIGMOD*, 2019, pp. 6:1–6:4.
- [33] J. Ding, U. F. Minhas, B. Chandramouli, C. Wang, Y. Li, Y. Li, D. Kossmann, J. Gehrke, and T. Kraska, "Instance-optimized data layouts for cloud analytics workloads," in *SIGMOD*. ACM, 2021, pp. 418–431.
- [34] G. Moerkotte, "Small materialized aggregates: A light weight index structure for data warehousing," in *VLDB*, 1998, pp. 476–487.
- [35] Oracle, "Oracle documentation," <https://docs.oracle.com/en/>.
- [36] PostgreSQL, "Postgresql documentation," <https://www.postgresql.org/docs/>.
- [37] P. Larson, C. Clinciu, C. Fraser, E. N. Hanson, M. Mokhtar, M. Nowakiewicz, V. Papadimos, S. L. Price, S. Rangarajan, R. Rusanu, and M. Saubhasik, "Enhancements to SQL server column stores," in *SIGMOD*, 2013, pp. 1159–1168.
- [38] L. Sun, M. J. Franklin, J. Wang, and E. Wu, "Skipping-oriented partitioning for columnar layouts," *PVLDB*, vol. 10, no. 4, pp. 421–432, 2016.
- [39] F. C. Dictionary, "beam search <http://foldoc.org/beam+search>."
- [40] "OpenStreetMap dataset," <https://registry.opendata.aws/osm/>.
- [41] "Parquet Filter Pushdown," <https://drill.apache.org/docs/parquet-filter-pushdown/>.