Qd-tree: Learning Data Layouts for Big Data Analytics

Group Members: Songming Fan and Tianling Gong

Introduction and Problem Definition

How the need of Qd-tree comes out

1. Increasing Data Volumes:
   The last decade has seen an exponential increase in the amount of data collected, emphasizing the need for efficient data analytics. This makes the need to run queries on large datasets increasingly important.

2. Limitation of the previous method
   More recently, most of researches focus on the method of organizing data in each block, focusing on reducing query time within a block. Ignoring how to reduce the time of doing I/O.

3. Current systems partition data by arrival time or range, which fails to minimize the number of data blocks accessed by a query. This highlights a gap in optimizing for query-specific data layouts.

The goal of Qd-Tree

Aiming to reduce I/O costs by learning optimal data layouts tailored to specific query workloads

What Qd-Tree does in high level

1. Generate a tree like structure to assign the data into block.

2. Using the tree like structure to do queries.

Problem Definition

I/O cost is related to the tuple need to be scanned. Given a set of tuples V, aiming to partition them into multiple blocks, such that the number of tuples required to scan fora workload is minimized. Suppose the block partitioning is $P = \{P1, . . ., Pk\}$, the worklord is W .

$$C(P_i) = |P_i| \sum_{q \in W} S(P_i, q)$$

Where $S(P_i, q) = 1 \; or \; 0$

$$C(P) = \sum_{P_i \in P} C(P_i)$$

In that case, minimize I/O cost translate to maximized $C(P)$, the total number of tuples skipped for the given workload W.

Problem Definition

Problem 1 (MaxSkip Partitioning).
Given a set V of tuples, a workload W of queries, and a minimal block size b, find a partitioning P to maximize C(P), s.t. $|Pi| \geq b$ for all $Pi \in$ P

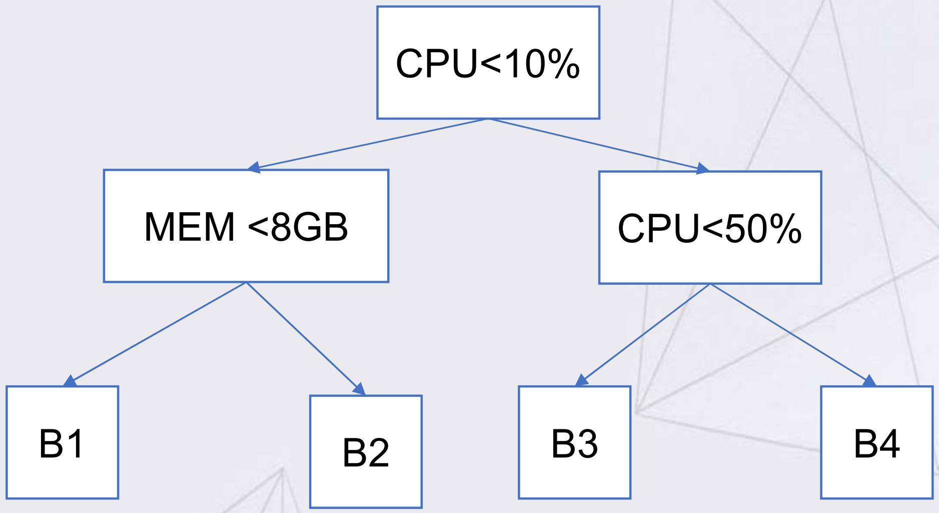
Problem 2 (Learned MaxSkip Partitioning).
Given a set V of tuples, a workload W of queries, a skipping function S, and a minimal block size b, find a partitioning function F, such that for the next V tuples ingested, the partitioning P generated by $F(V)$ maximizes C(P).

Qd-Tree structure

Routing Data

The overall strategy is to use a qd-tree to assign data to blocks on storage. Each record "arrives" at the root and is recursively routed down

Records are stored with an additional block ID (BID) field to denote the block they belong to, and the dataset is partitioned by this field.

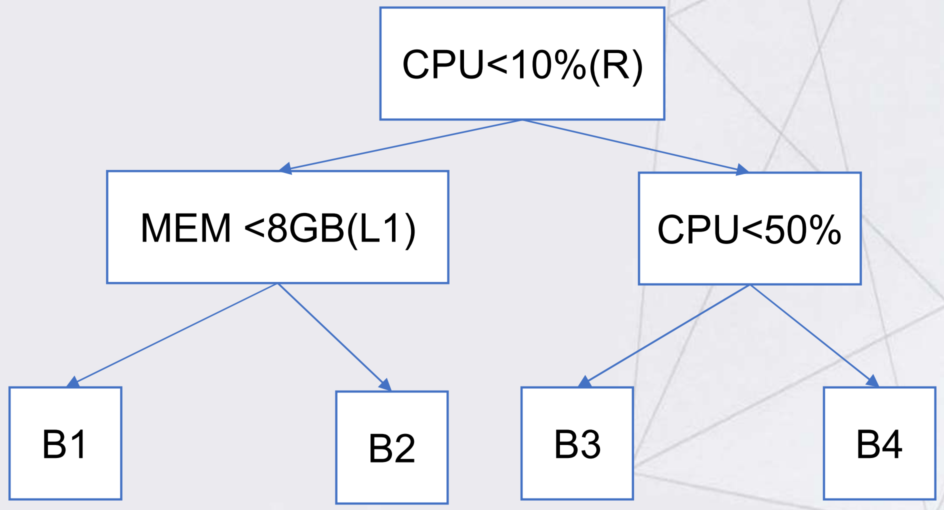


Semantic Description of Nodes

Two kind of comparisons need to handle:

1. Range comparisons: It can be easily handled by using a 2*N array. Like the R can be described as [0,MAX_CPU), [0,MAX_MEM)

2. Equality comparisons: Some column in the database can be called categorical column. Like a “color” column.

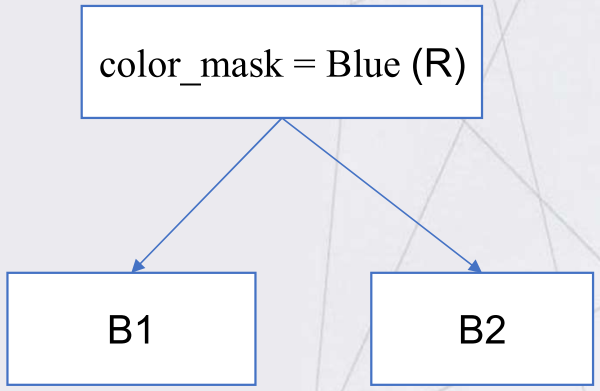


Semantic Description of Nodes

2. Equality comparisons: We can use a array with length D array to handle each category column. For example, for Color column: $priority \in$ {Red , Blue, White}

R.color_mask: $priority \in$ [1,1,1]

There is a cut on root with color_mask = Blue, and create two leaf node called B1(left) and B2(right).



B1.color_mask: $priority \in$ [1,1,1]
B2.color_mask: $priority \in$ [1,0,1]

Why not use B1.color_mask: $priority \in$ [0,1,0]?

How to get Candidate Cuts?

Using simple treatment

Since we are given a target workload *W* of queries, we simply parse them through a standard SQL planner and take all pushed-down unary predicates as allowed cuts.

```
SELECT T
FROM TABLE T
WHERE (T.a<10 OR T.b<90) AND (T.c IN (1,4))
```

We can get three cuts from this query: T.a<10; T.b<90 ; T.c IN (1,4).

How to do queries?

Two ways are given

1. Using qd-tree to do search:
   From root to leaf, find the BID in the leaf that fit the query.

   Queries need to be reconstructed to fit the qd-tree node

2. Scan the leaf node: **This is the author actually doing in experiments**
   We can just scan all the leaf nodes(block) of the qd-tree, check whether its description intersects with the query

Greedy construction of Qd-tree

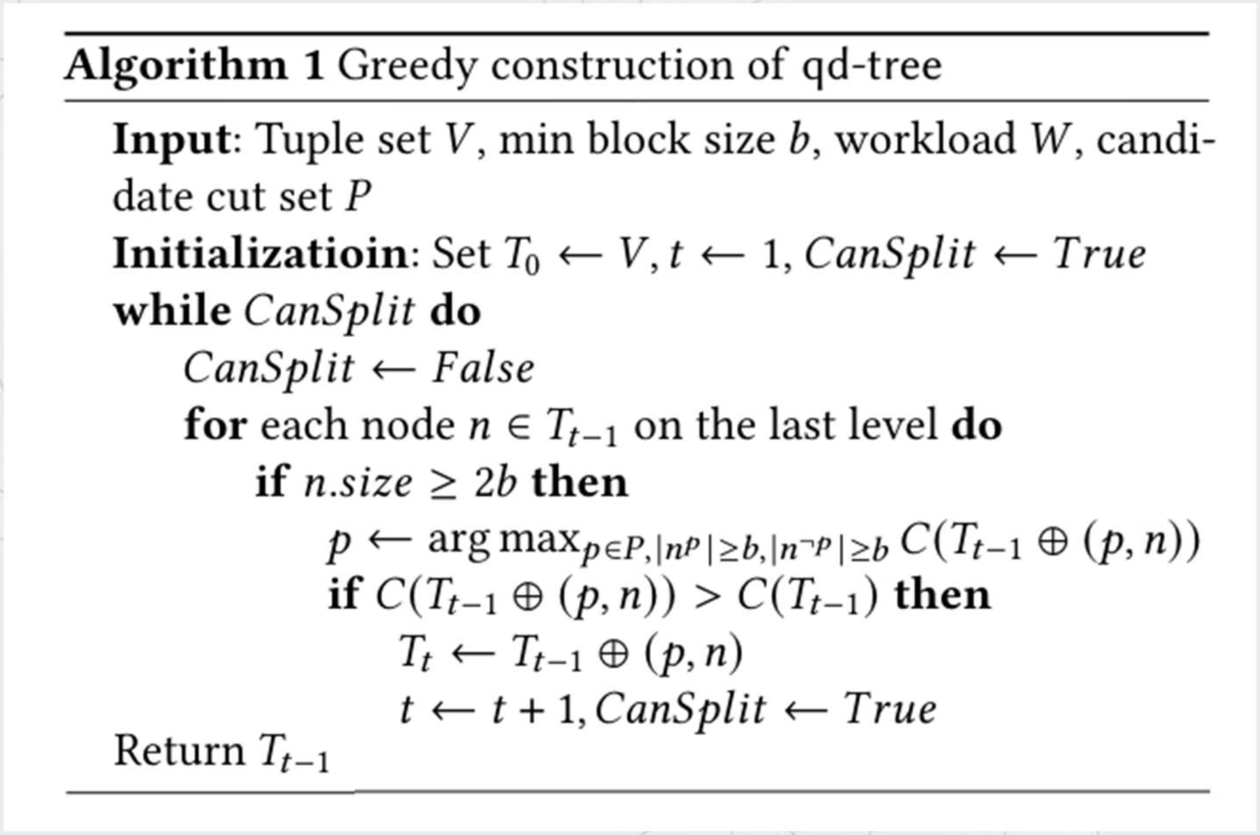
Why use greedy approach and how it works?

The construction of a qd-tree is an NP-hard combinatorial optimization problem. Greedy algorithms are a typical family of solutions that are usually efficient and make locally optimal choices. It can give a qd-tree not so bad.

To present the algorithm, we define an action $a = (p, n)$ as applying cut p to node n in qd-tree T. The result of action a is denoted as $T \oplus a = T \oplus (p, n)$.

What greedy algorithm do is try all possible cut p on node n and choose the one can maximize $C(T \oplus (p, n))$

How it works?



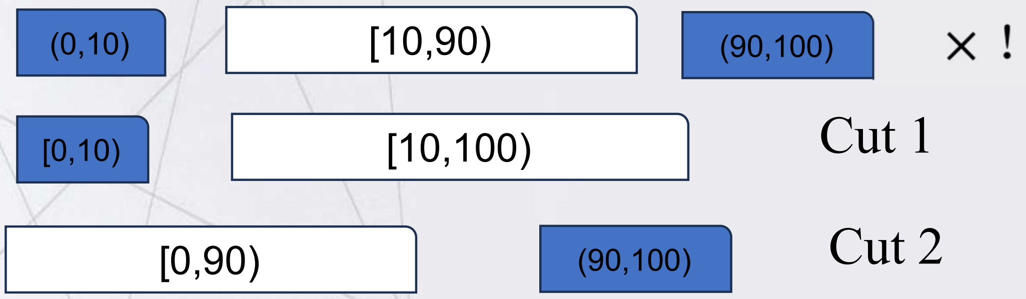
Algorithm 1 Greedy construction of qd-tree

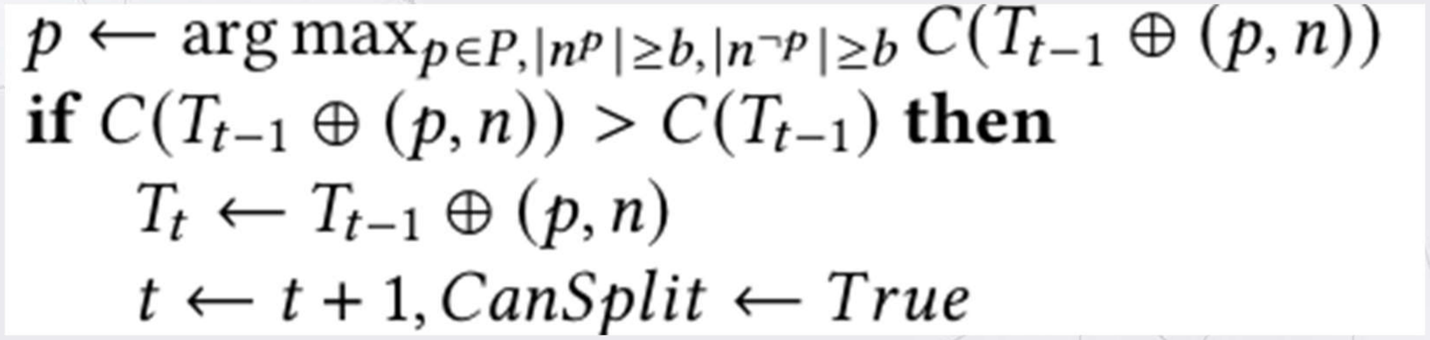
```
**Input**: Tuple set $V$, min block size $b$, workload $W$, candidate cut set $P$
**Initializatioin**: Set $T_0 \leftarrow V, t \leftarrow 1, CanSplit \leftarrow True$
**while** $CanSplit$ **do**
    $CanSplit \leftarrow False$
    **for** each node $n \in T_{t-1}$ on the last level **do**
        **if** $n.size \geq 2b$ **then**
            $p \leftarrow \arg\max_{p \in P, |n^p| \geq b, |n^{\neg p}| \geq b} C(T_{t-1} \oplus (p, n))$
            **if** $C(T_{t-1} \oplus (p, n)) > C(T_{t-1})$ **then**
                $T_t \leftarrow T_{t-1} \oplus (p, n)$
                $t \leftarrow t + 1, CanSplit \leftarrow True$
Return $T_{t-1}$
```

The problem of greedy approach

Can’t handle work lord with just disjunctive range queries.

Like a simple query: Q1: cpu<10 OR cpu>90





```
$p \leftarrow \arg\max_{p \in P, |n^p| \geq b, |n^{\neg p}| \geq b} C(T_{t-1} \oplus (p, n))$
if $C(T_{t-1} \oplus (p, n)) > C(T_{t-1})$ then
    $T_t \leftarrow T_{t-1} \oplus (p, n)$
    $t \leftarrow t + 1, CanSplit \leftarrow True$
```

Qd-Tree Using Deep RL

Qd-Tree Using Deep RL

Issues with techniques for optimization

Greedy Technique does not handle well for global optimization.

Memoized Search (Dynamic Programming) is not viable as high-dimensional search space is too large, approximate dynamic programming required.

What’s a better approach?

Approximate, accelerated, and incremental memoized search performed by deep RL

Qd-Tree Using Deep RL

WOODBLOCK, a deep RL agent.

How it works?

Construct routing trees for a target dataset and workload

Learns to identify better cuts through rewards

Deploys best tree after a timeout is reached or a number of tree are attempted.

Qd-Tree Using Deep RL

What are the benefits of using WOODBLOCK?

Featurization the states and uses the model to make prediction about rewards given current state

Sampling subsets of follow-up actions of search states and updating model from rewards

Produces better trees and efficient deployment of solution

Qd-Tree Using Deep RL

Greedy construction relies on tree-submodularity

- Disjunctive range and candidate cut fails to satisfy and cannot skip
- Must choose the cut on disk, neither query 1 nor query 2 can be satisfy, lacks skipping ability.

RL produces better partitioning and handles candidate cuts and large number of data dimension.

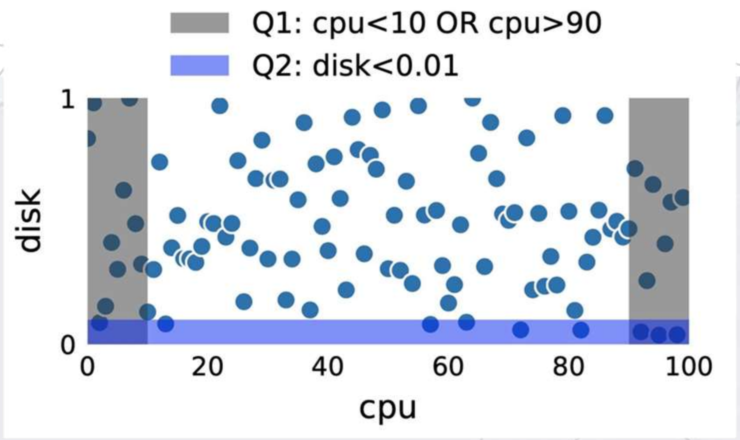


Figure 3: A dataset with disjunctive queries. Regions se lected by Q1/Q2 are shown in grey/blue. The candidate cut are: {cpu<10, cpu>90, disk<0.01}. The first two cuts canno skip any query, so Greedy opts for the third cut, resulting i a scan ratio of 50.5%. Woodblock is not limited by the form of queries; it produces a layout with a scan ratio of 10.4%, 4.8× improvement. Discussion in Section 5.1.

- Block 1: $disk < 0.01$
- Block 2: $(disk \geq 0.01) \wedge (cpu > 90)$
- Block 3: $(disk \geq 0.01) \wedge (cpu < 10)$
- Block 4: $(disk \geq 0.01) \wedge (cpu \leq 90) \wedge (cpu \geq 10)$

WOODBLOCK

WOODBLOCK

Application of RL algorithm:

Variables:

Tree Construction Markov Decision Process (MDP)

State Space: subspace of the entire data space of the relation under optimization

Action Space: set of allowed cuts.

Core of WOODBLOCK: 2 learnable networks parameterized by θ

Policy network: returns a probability distribution over action space $\pi_\theta : S \rightarrow A$

Value network: returns the expected cumulative reward from a given state $V_\theta : S \rightarrow \mathbb{R}$

WOODBLOCK

Process of construction of qd-tree:

- The agent starts construction of one tree with a root node
- Agent transitions into next states.
- Once a stopping condition is reached, described next, a qd-tree is completed.

Higher skipping ratio is achieved through cuts made by the refined θ.

WOODBLOCK

Stopping Condition

Each leaf block must contain at least *b* records.

If the resultant children contain more than *b* records by approximation, then agent can make a cut on current node.

This approximation is achieved by testing the cut on a data sample.

After cut is made, begin evaluation of the cut on the data sample as a cut is made, and resulted in a subset of records that satisfy it and a subset that does not.

When current node has no legal cuts left in action space, stop cutting on it and a leaf is form.

WOODBLOCK

Calculating Reward for WOODBLOCK to learn

Number of skipped records under node n across all queries

$$S(n) := \begin{cases} C(\text{n.records}) & \text{if } n \text{ is a leaf} \\ S(\text{n.left}) + S(\text{n.left}) & \text{otherwise} \end{cases}$$

For every action, a reward is assigned

$$R((n, p)) := S(n)/(|W| \cdot |\text{n.records}|)$$

WOODBLOCK

How are the networks implemented?

Policy network and the value network have shared weights, which are two fully-connected layers, 512 units each, with ReLU activation.

Output layer:

- Policy network: a $|A|$-dimensional linear projection
- Value network: a scalar projection.

Each state is featurized as the concatenation of n.range and n.categorical_mask.

Framework Extension

Framework Extension

Advanced Cuts

Fast evaluation in the process of tree construction is enabled by single-column predicates

Qd-tree can be extended to support binary cuts of the form (attr1, op, attr2)

Append a new component to each node’s description

n.adv_cuts: a bit vector of size |AC|

Framework Extension

Data Overlap

Qd-tree also supports duplicating data.

The (N + 1)-record block is not further cut because of the minimum block size constraint

Construction algorithms with a relaxed cutting condition which allows one of the children to have size smaller than the constrained block size.

The lucky block would be further cut into an N-record one and a block with a single record.

Modifications to node metadata still maintain and the semantic descriptions preserve completeness.

No extra storage cost.

Framework Extension

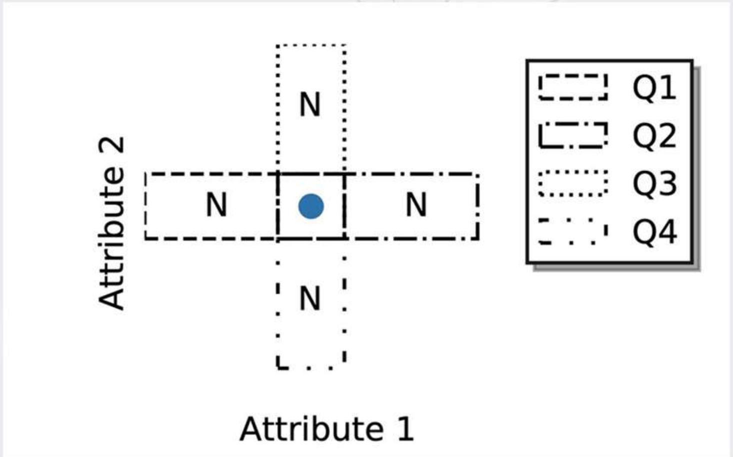


Figure 4: A scenario where significant data skipping is gained by replicating a single record. Each query selects $N+1$ records. The queries only overlap in the one tuple placed at the center. If the space is naively cut in a binary fashion, 3 out of 4 queries each reads N extra tuples. By handling overlap, qd-tree replicates the singleton record to all four N-record regions, so no queries touch unnecessary records.

Framework Extension

Data and Query Routing

Data routed the same way, except a row routed to all matching blocks.

Candidate set of blocks includes all blocks that overlap with the query rectangle.

With overlap, the set of blocks scanned when evaluating a query may contain duplicate rows.

Prune away the other blocks because a block covers the query rectangle.

When scanning block ID, ignore tuples match block’s semantic description with ID < i.

Framework Extension

Solution to Data Replication: Two-tree approach

Full-copy data replication is an approach to overlap when utilizing extra storage.

Using Greedy or RL to learn a qd-tree optimized for the full workload

Building a second qd-tree tuned for the queries that has the issue of the worst skippability under the qd-tree learnt.

Modifying the reward function for RL through accounting for the existence of the first tree.

Changes to be made:

- Picking one of the two trees that maximizes the skippability for each query
- Calculating the number of skipped tuples
- Summing up skipped tuples for all queries in the workload

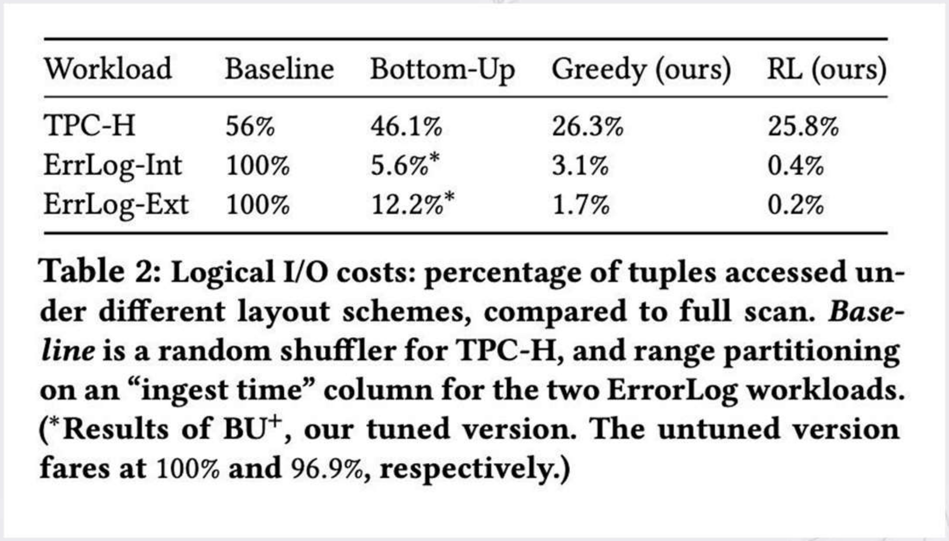
Weak Points

Due to the random cut that is made initially, WOODBLOCK's performance at the start can be suboptimal. The agent need to gradually learn through reward function, identifying profitable cuts and to optimize data skipping.

A large portion of tree construction time is taken up by routing records and calculation of rewards, therefore in most occasions, only CPUs are used for the WOODBLOCK agent.

Evaluations and Conclusions

Evaluation



Workload	Baseline	Bottom-Up	Greedy (ours)	RL (ours)
TPC-H	56%	46.1%	26.3%	25.8%
ErrLog-Int	100%	5.6%*	3.1%	0.4%
ErrLog-Ext	100%	12.2%*	1.7%	0.2%

Table 2: Logical I/O costs: percentage of tuples accessed under different layout schemes, compared to full scan. *Baseline* is a random shuffler for TPC-H, and range partitioning on an "ingest time" column for the two ErrorLog workloads. (*Results of BU^+, our tuned version. The untuned version fares at 100% **and** 96.9%**, respectively.)**

Evaluation

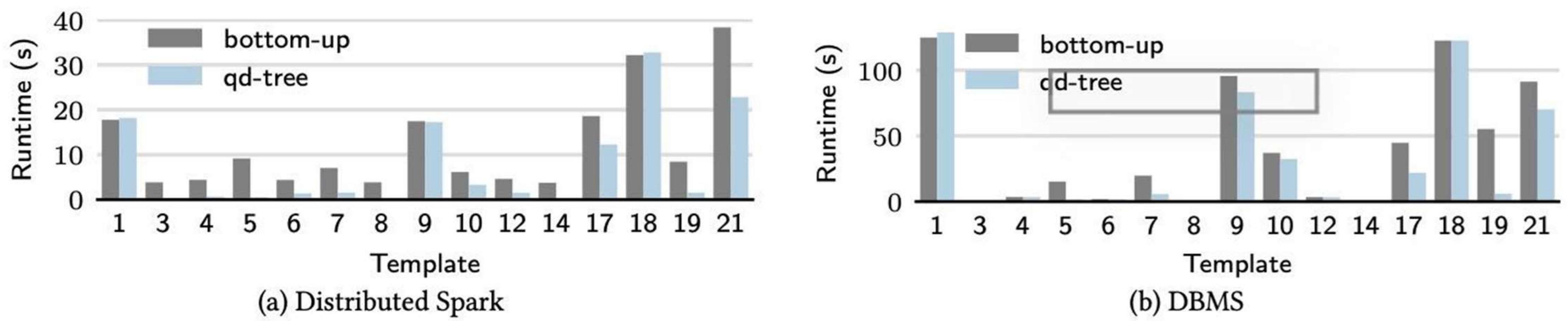
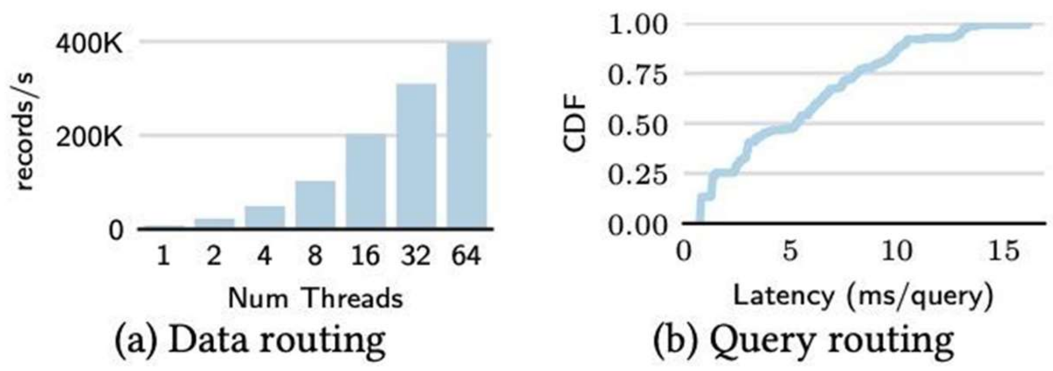


Figure 5: TPC-H execution runtimes, grouped by each template.

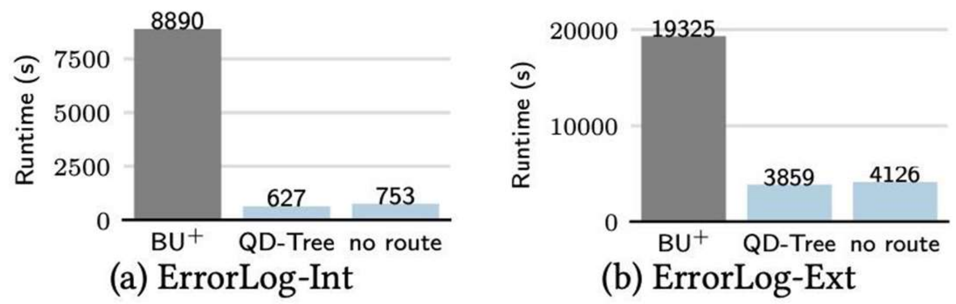
Evaluation

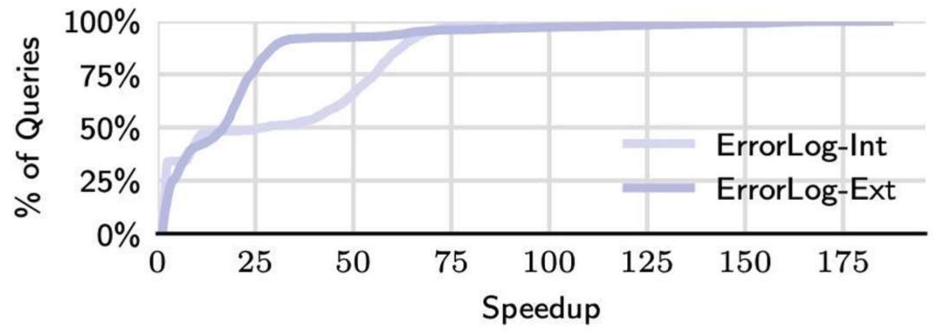


(a) Data routing (b) Query routing

Figure 6: Performance of routing data and queries.

Evaluation





(c) CDF of per-query speedups over BU$^+$

Figure 7: ErrorLog execution runtimes.

Evaluation

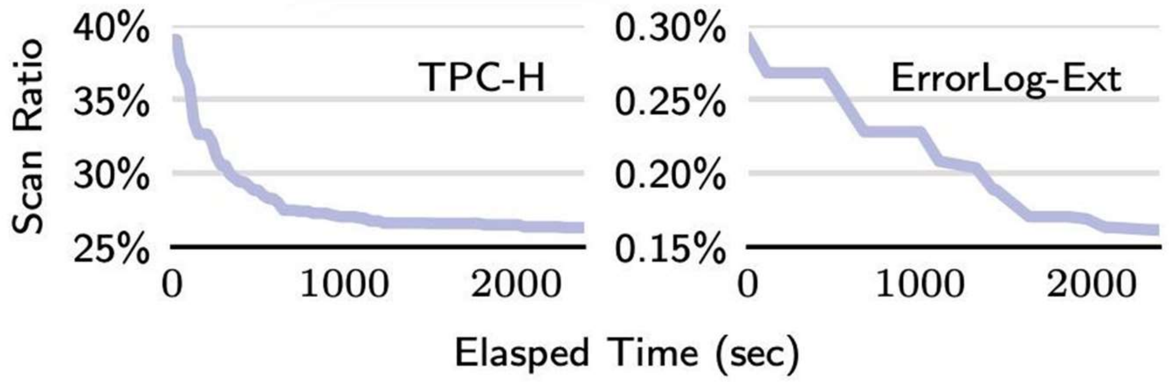


Figure 8: Learning curve of Woodblock. On TPC-H, most quality improvement is learned in the first ~10 minutes; on ErrorLog-Ext, high quality is achieved immediately (.3%) and continuously improved when given more time budget. The trend for ErrorLog-Int is similar (not shown).

Evaluation

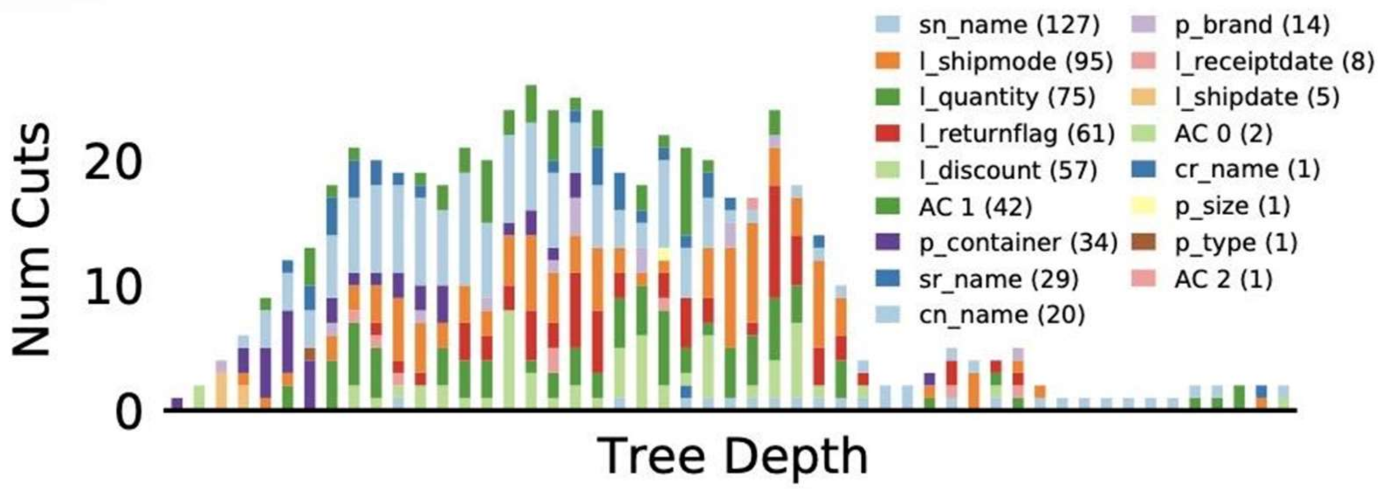


Figure 9: A Woodblock-produced top-performing qd-tree for TPC-H. The number after each legend indicates the total number of cuts on that column (or advanced cut).

Conclusions

qd-tree is implemented as a lightweight Python library

Woodblock, the RL agent, is implemented using Ray RLlib, a scalable reinforcement learning library.

Compared to current blocking schemes, qd-tree can provide physical speedups of more than an order of magnitude

Reaches 2 times of the lower bound for data skipping based on selectivity

Provides complete semantic descriptions

In future work, strategies for pruning away the redundant work are explored

as well as the extension to more than two trees in data replication.