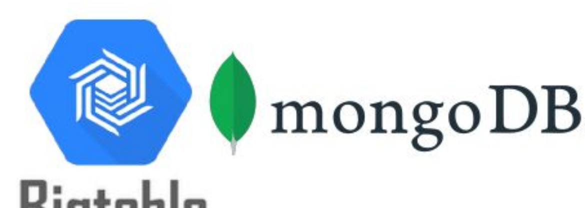
Learning to Optimize LSM-trees:
Towards A Reinforcement Learning based Key-Value Store for Dynamic Workloads

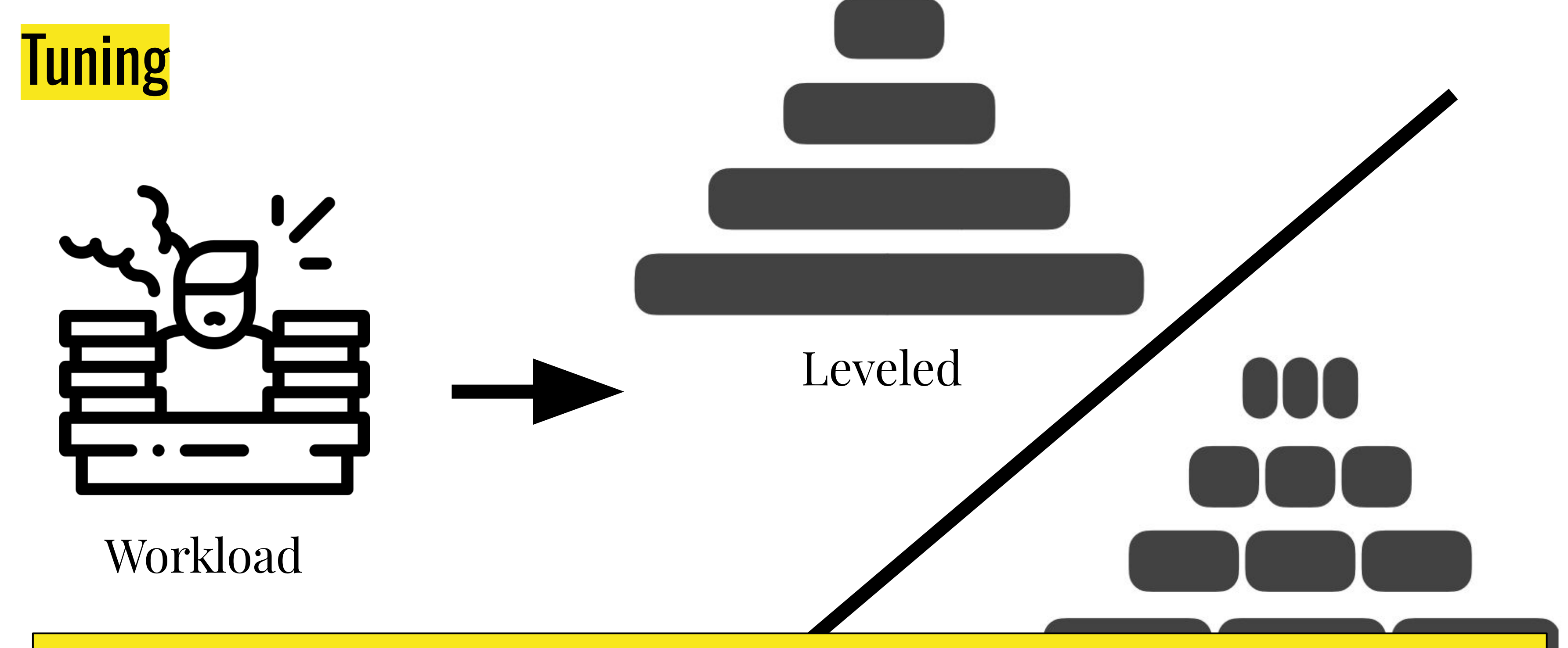
Mo, Chen, Luo, Shan — SIGMOD 24’

Presenters: Alex Ott, James Chen

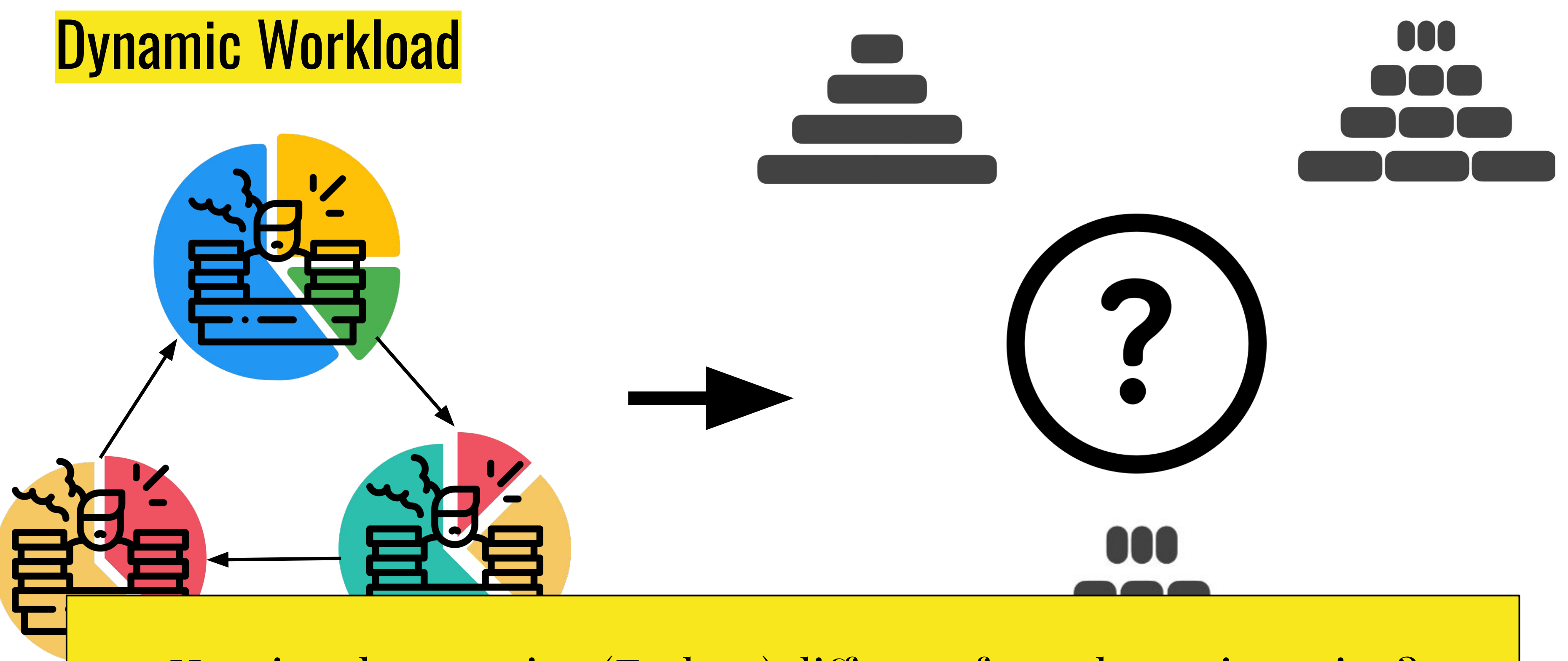
Why Use an LSM?



Tuning



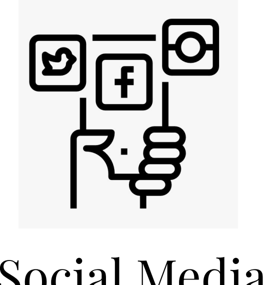
What happens when the workload changes over time?



How is robust tuning (Endure) different from dynamic tuning?

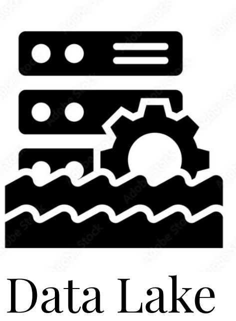
Motivation

Real world workloads are *dynamic*



Social Media

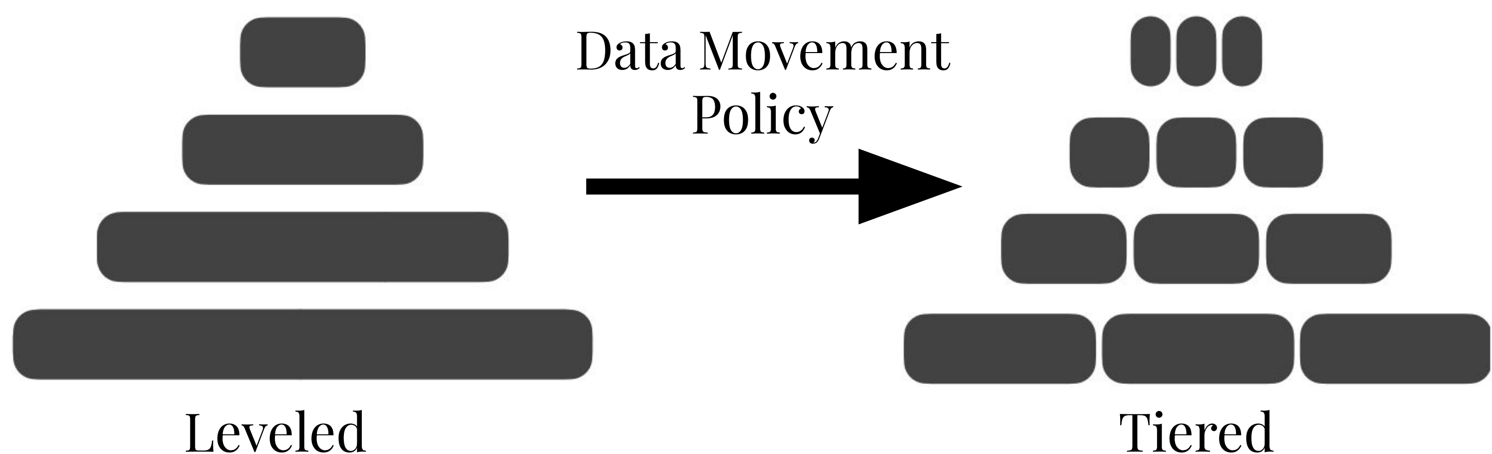
Data Warehouse



Data Lake

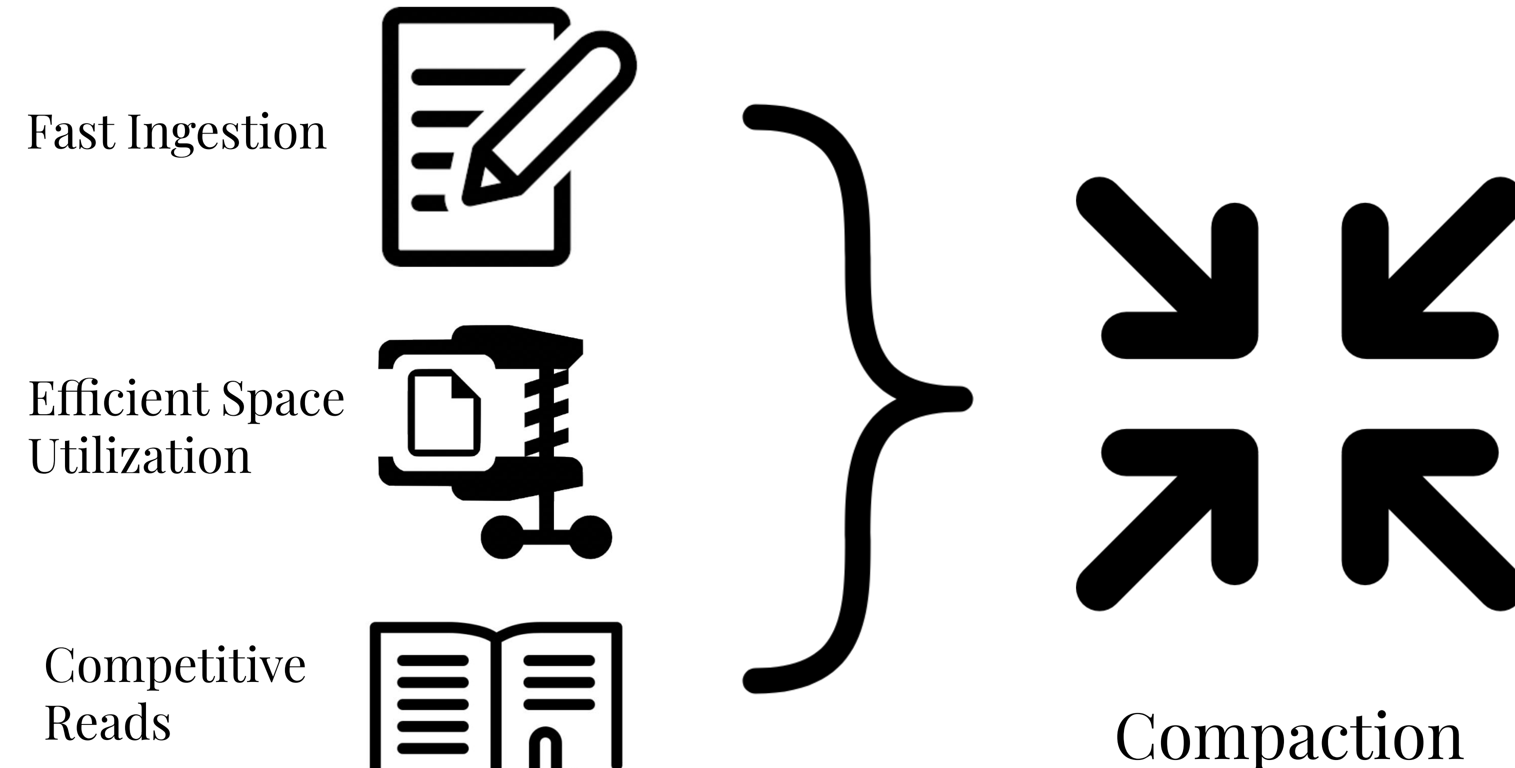
How to build a **workload-aware** LSM that adapts in **real-time**?

Intuition



Background

Compactions Drive Data Layout



Compaction

What is Machine Learning?

how to learn, not
what to do

Generalizable

Data, Data, and Data

Too much data

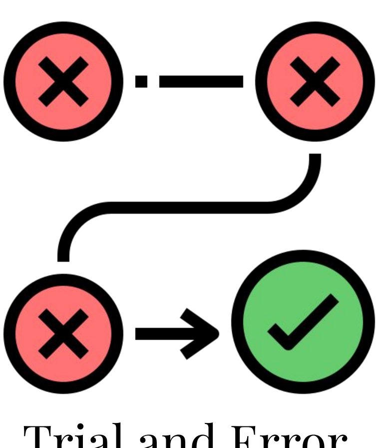
machine learning

unsupervised learning

supervised learning

reinforcement learning

Reinforcement Learning (RL) Mental Model



Trial and Error

REINFORCEMENT LEARNING MODEL

State (S_t)

Agent

Reward (R_t)

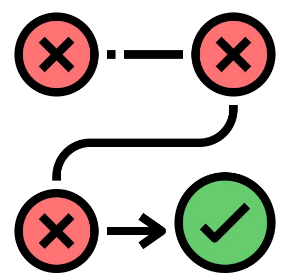
Action (A_t)

$R_{(t+1)}$

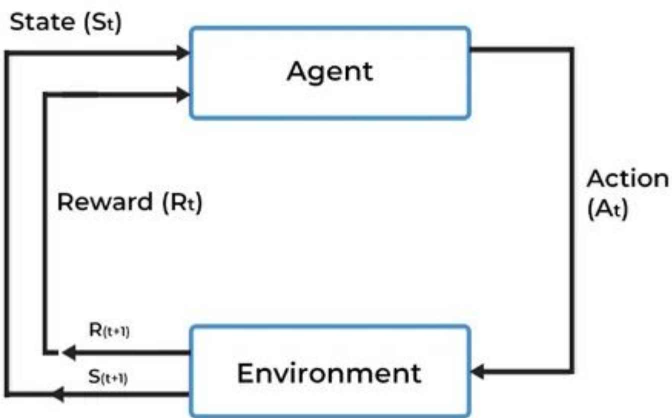
$S_{(t+1)}$

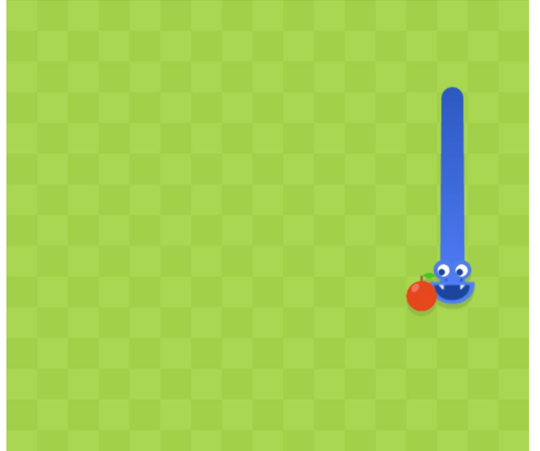
Environment

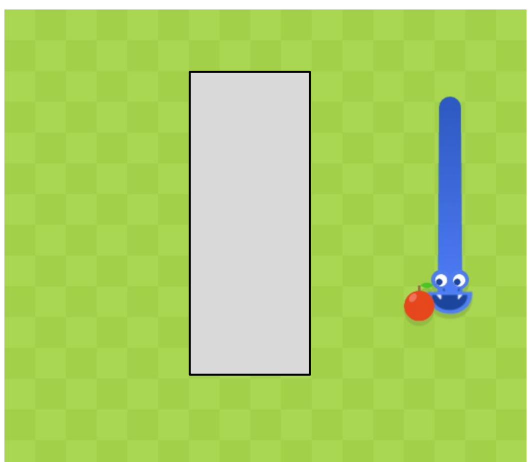
Reinforcement Learning (RL) Mental Model

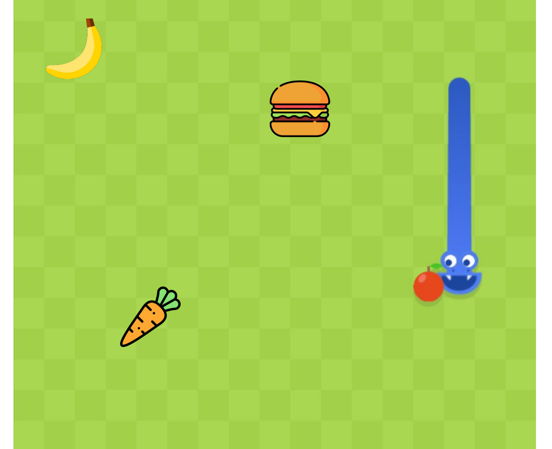


Trial and Error







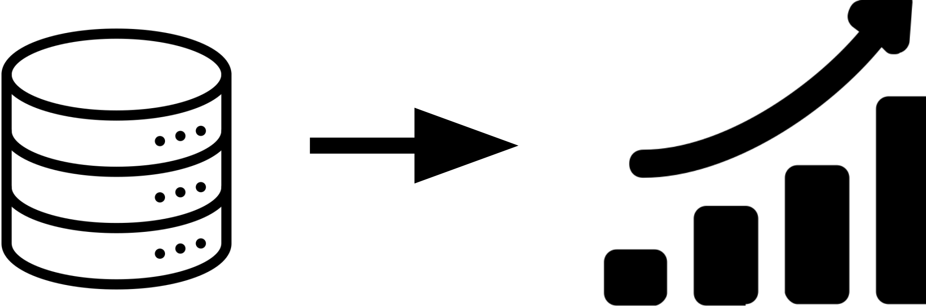


Advantages of RL?

No prior data

Challenges?

Need data to improve



RusKey

RL in RusKey

	Agent	**Environment**	**Actions**	**State**	**Reward**
Snake Game	Snake character	2D grid	Turn left or right	Snake and apple position	+ food - Running into the wall
LSM					

Key Components

2 Environment

5 Reward

FLSM-tree Based Database

Application Workload

Mission

Lookups

Updates

Composition

4 State

End-to-end stats

FLSM-Tree

Level

1

2

3

...

Level stats

Stats Collector

Flexible Transition

K K K

K'

3 Action

Tuning strategy

LERP Model

Get State & Reward

End-to-end latency

Level latency

Current state
Tuning strategy
Next State
Reward

Store

Replay Buffer

1 Agent

Sample & Update

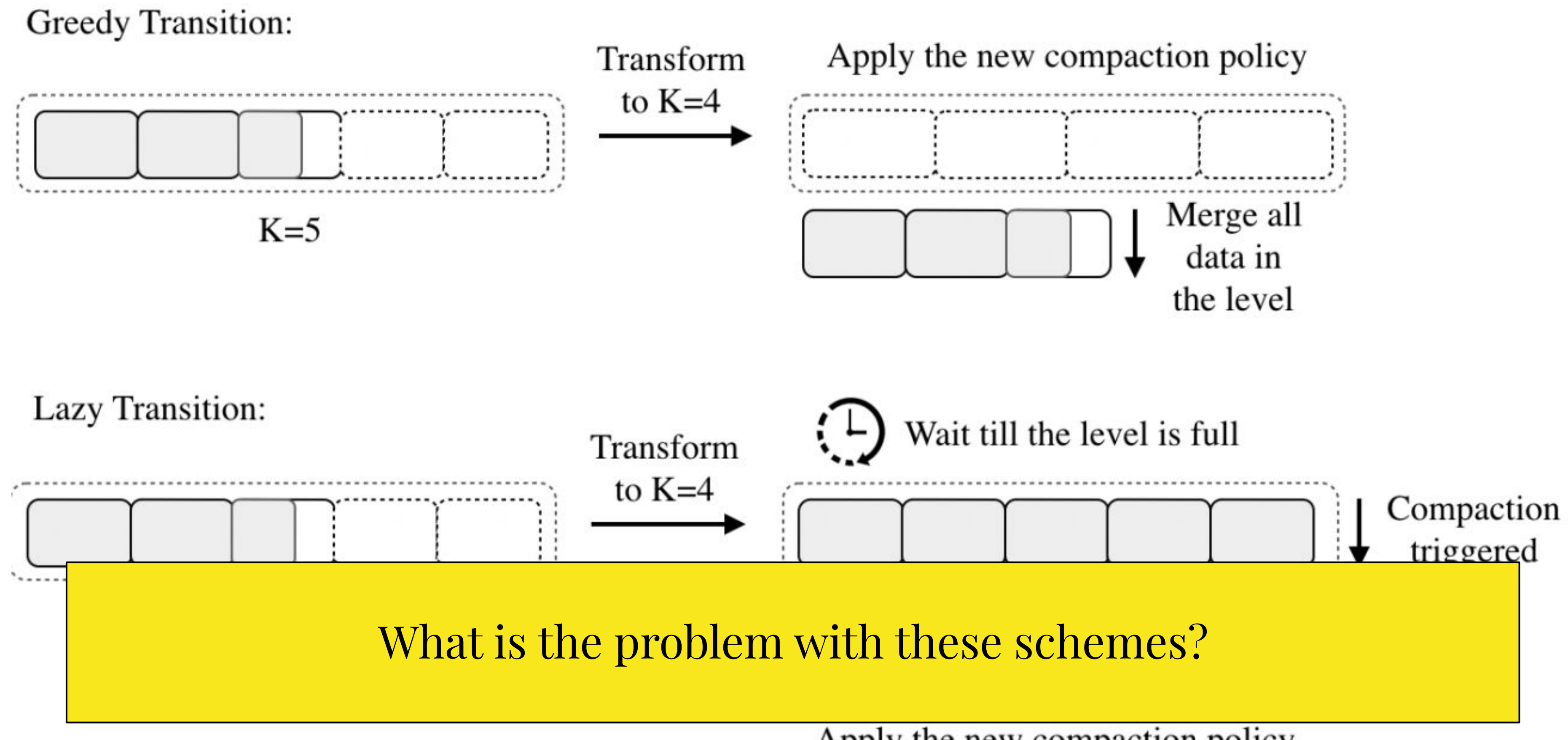
Actor

Update

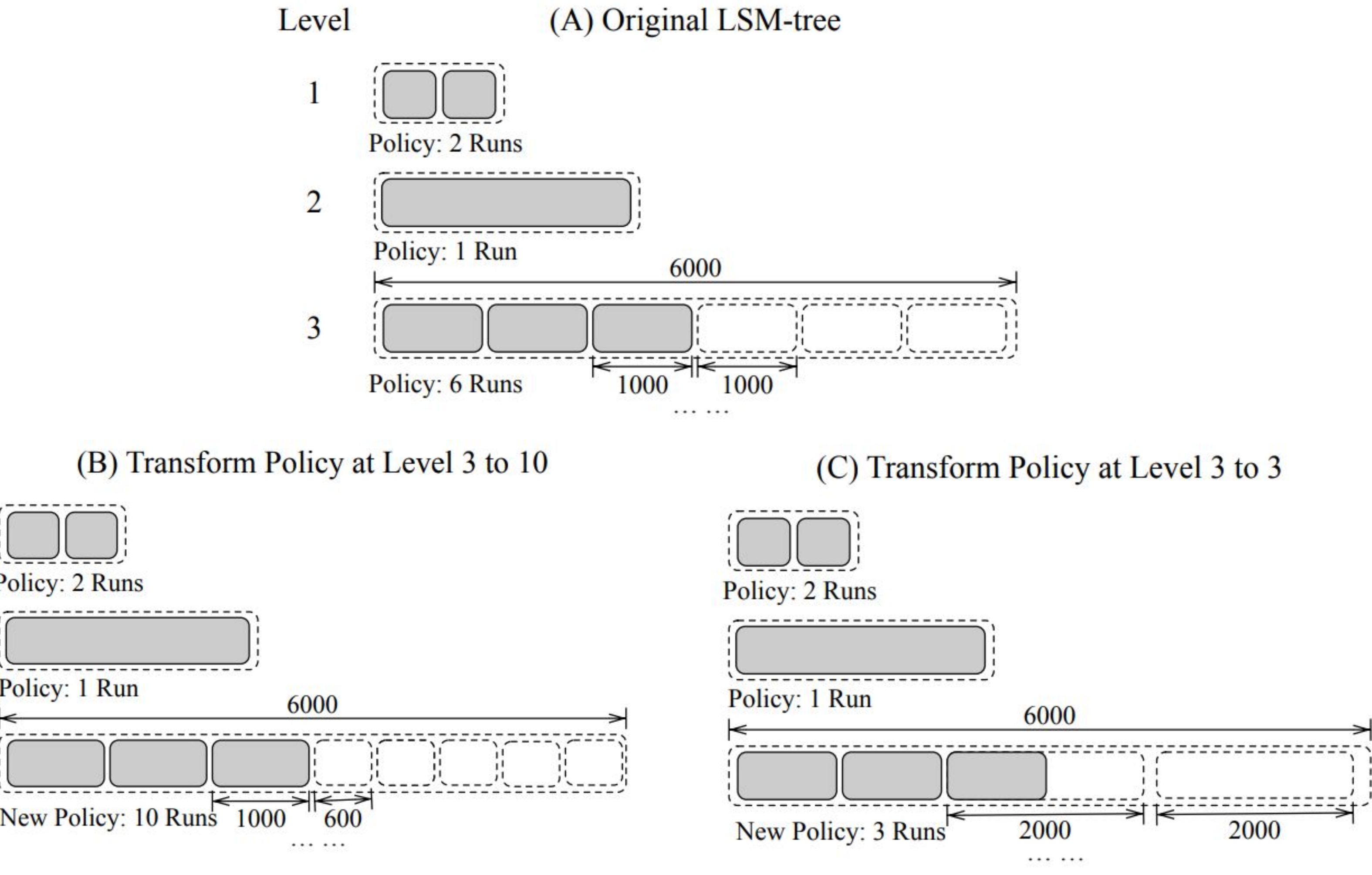
Critic

Fig. 1. The components and workflow of RusKey.

Transitions: Greedy, Lazy, and Something in Between



What is the problem with these schemes?


Level
(A) Original LSM-tree
1
Policy: 2 Runs
2
Policy: 1 Run
6000
3
Policy: 6 Runs
1000
1000
(B) Transform Policy at Level 3 to 10
Policy: 2 Runs
Policy: 1 Run
6000
New Policy: 10 Runs
1000
600
(C) Transform Policy at Level 3 to 3
Policy: 2 Runs
Policy: 1 Run
6000
New Policy: 3 Runs
2000
2000

LERP Framework

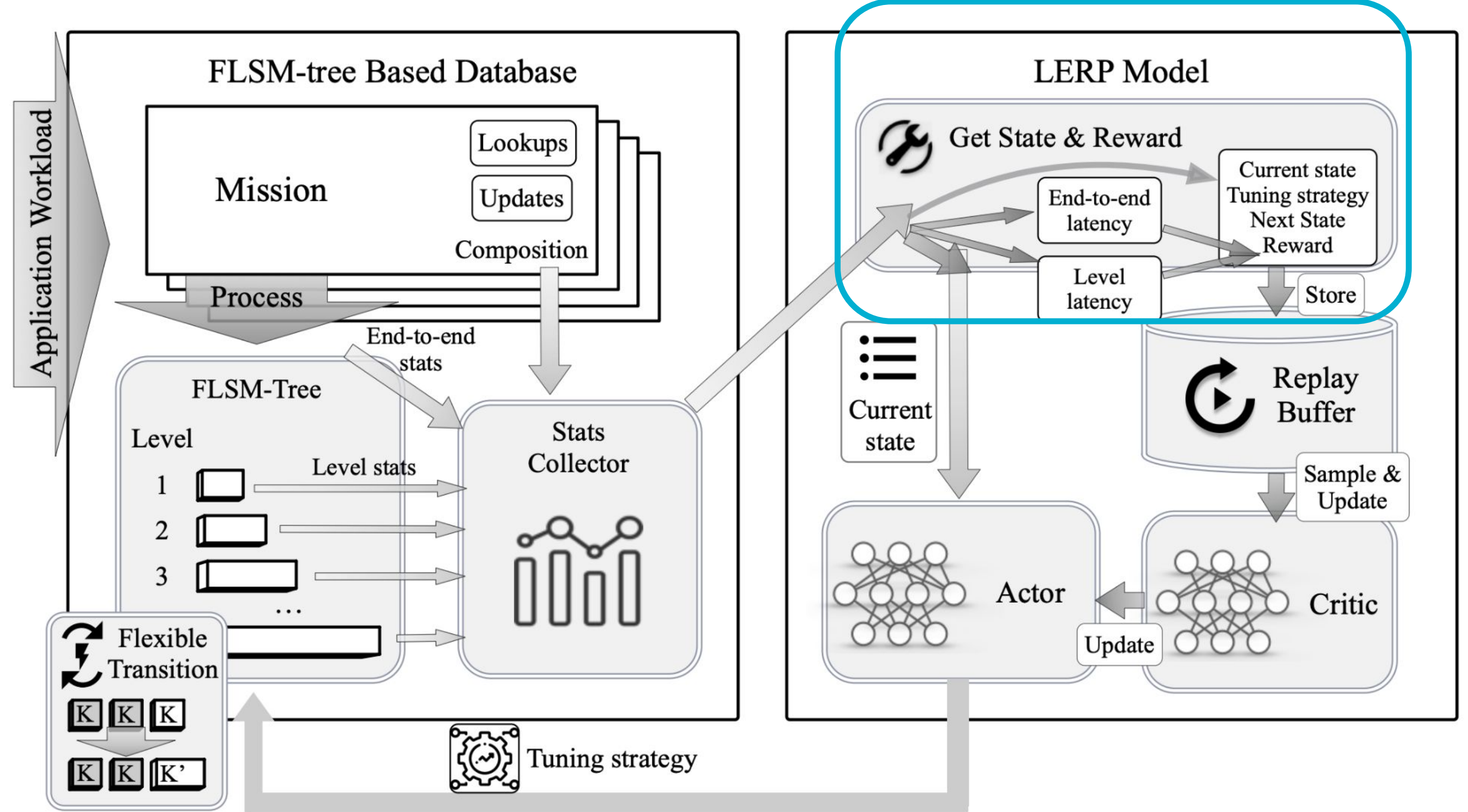


Fig. 1. The components and workflow of RusKey.

LERP Framework

Adjust compaction policy

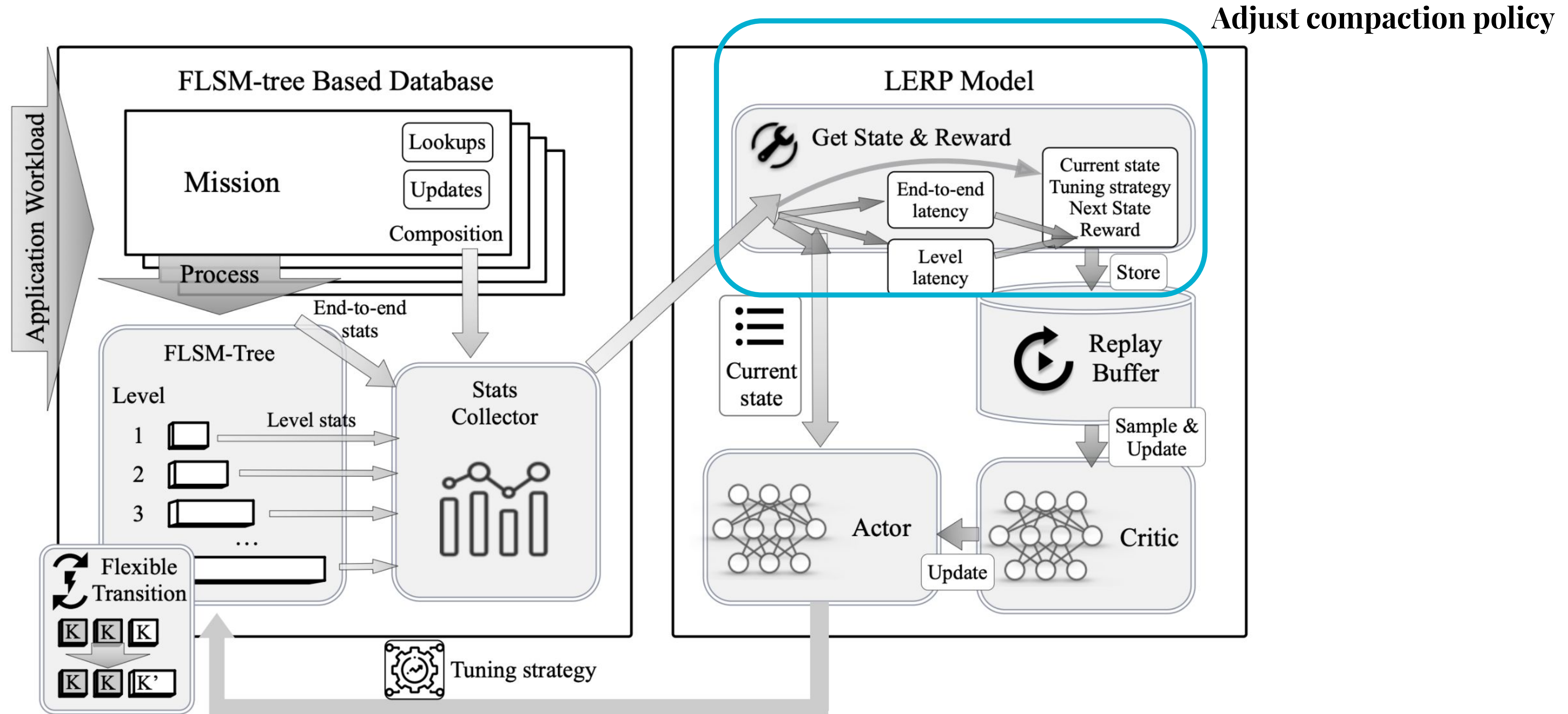


Fig. 1. The components and workflow of RusKey.

LERP Framework

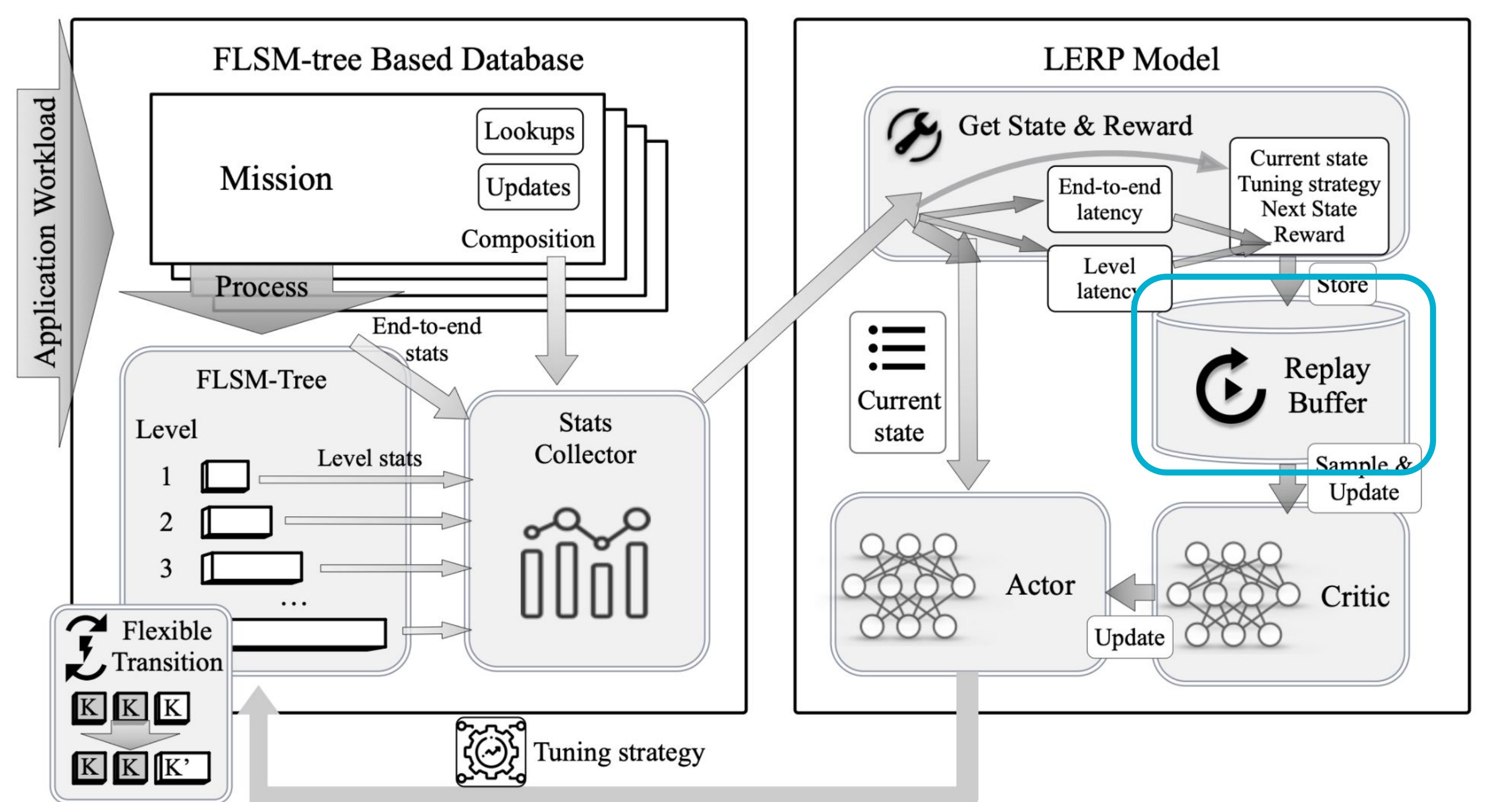


Fig. 1. The components and workflow of RusKey.

Experience samples:

- Action of the policy change
- FLSM- tree state
- Reward for the action

LERP Framework

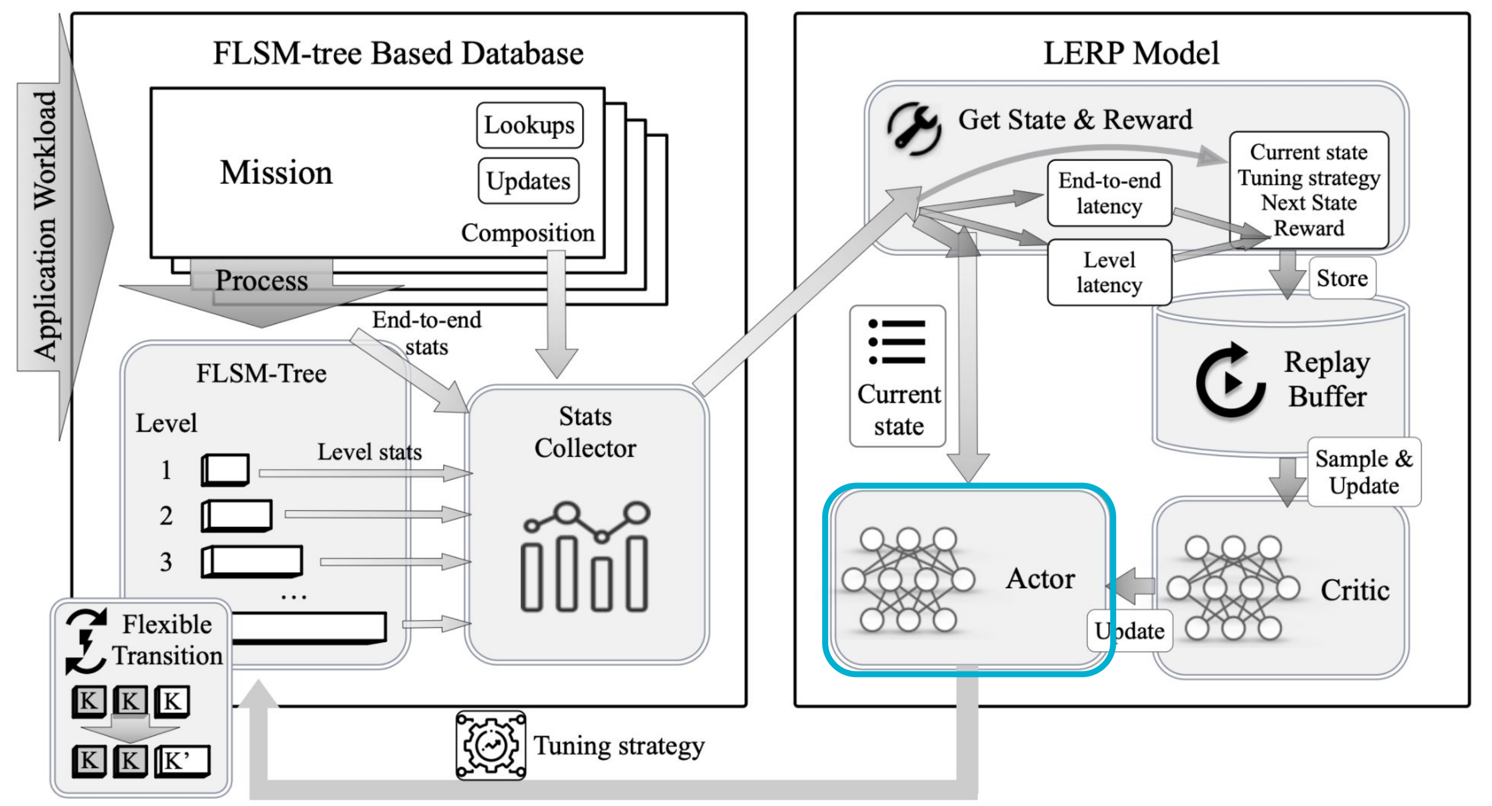


Fig. 1. The components and workflow of RusKey.

Selecting the best compaction strategy

LERP Framework

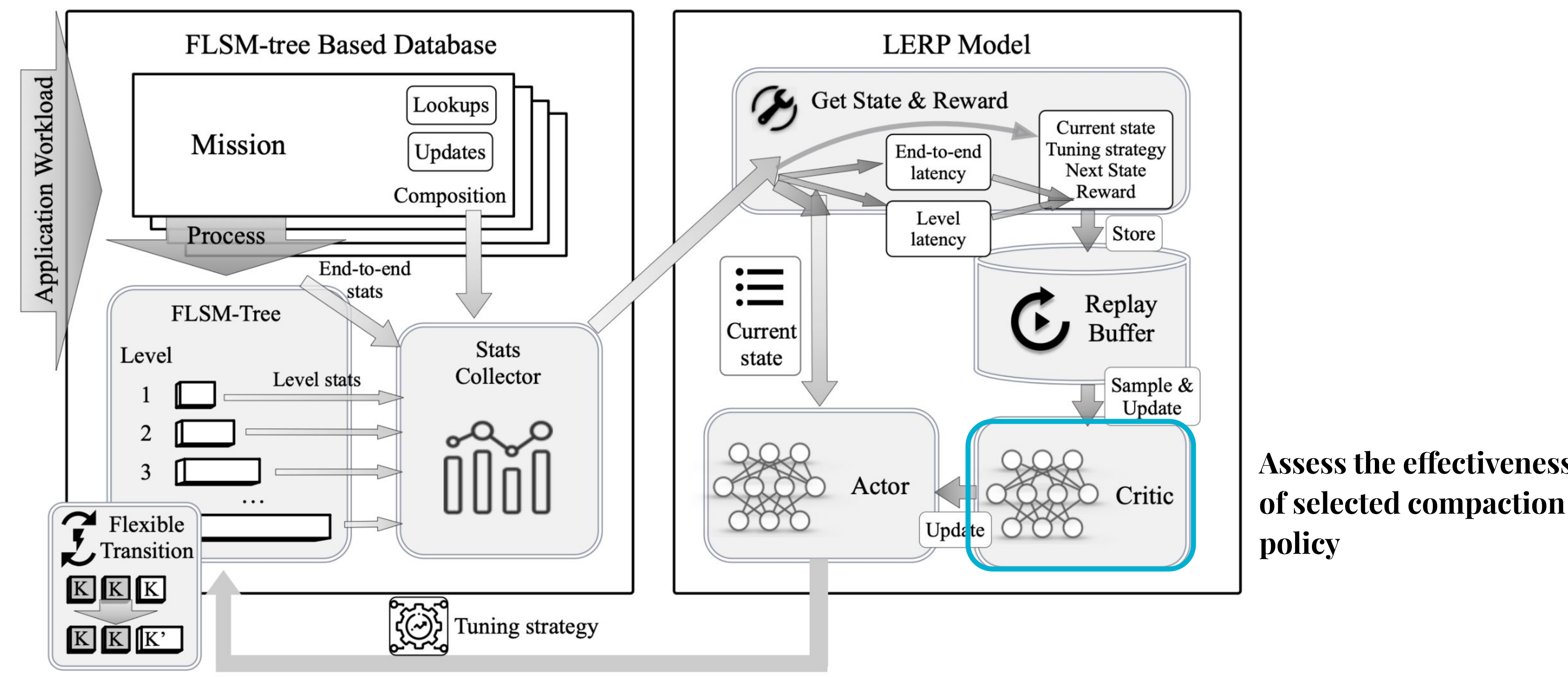


Fig. 1. The components and workflow of RusKey.

Assess the effectiveness of selected compaction policy

LERP Framework

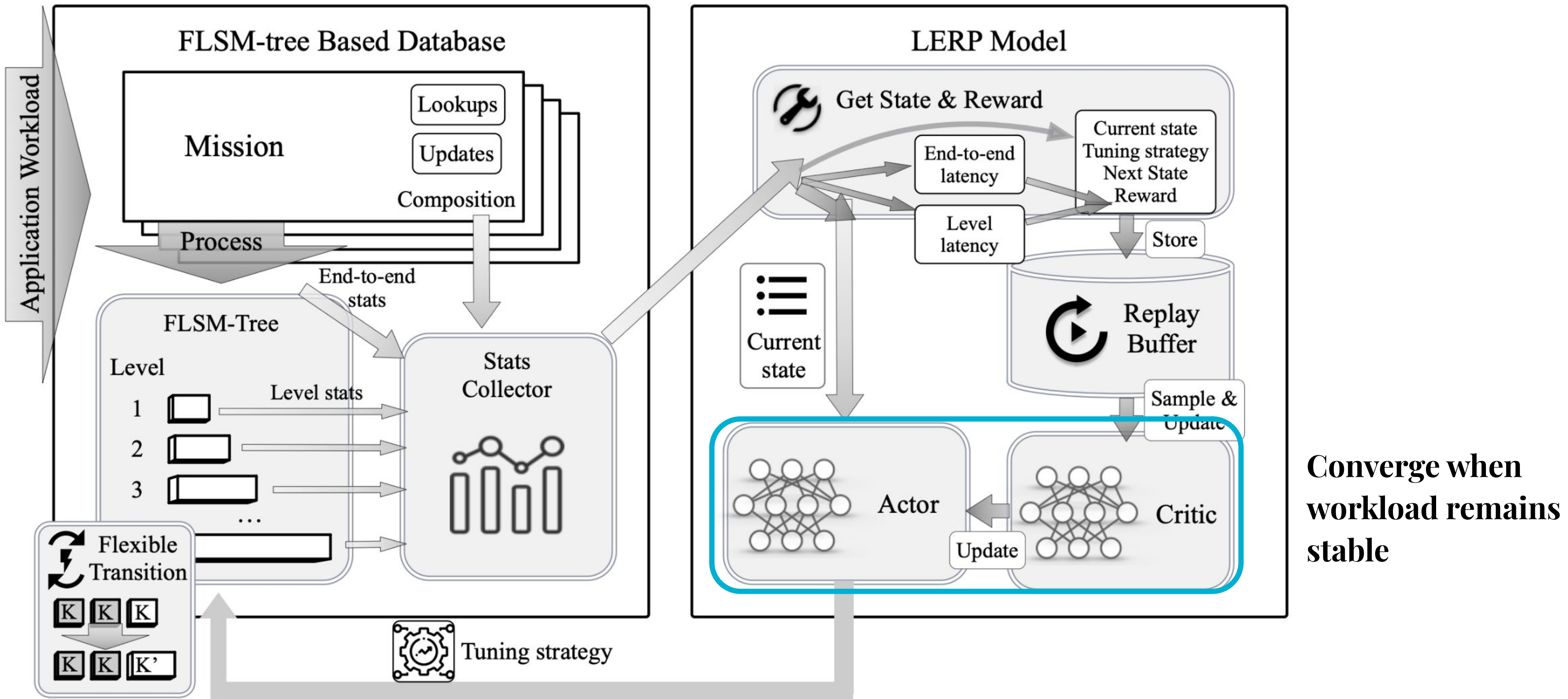


Fig. 1. The components and workflow of RusKey.

Converge when workload remains stable

LERP Framework

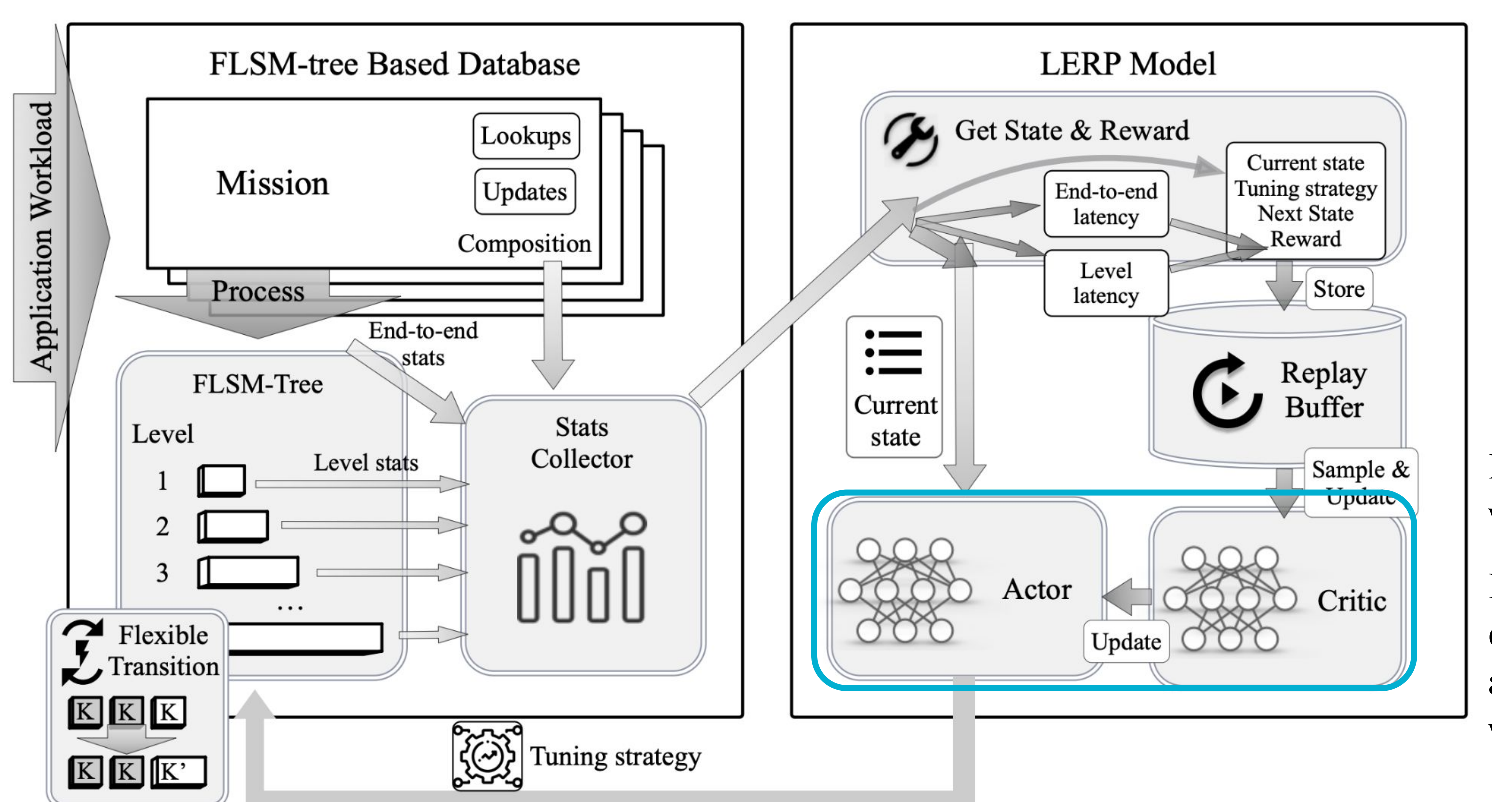


Fig. 1. The components and workflow of RusKey.

Exit convergence when workload changes.

LERP explores new compaction strategy adapted to new workload.

Level Based RL Model

State

LSM-tree configuration

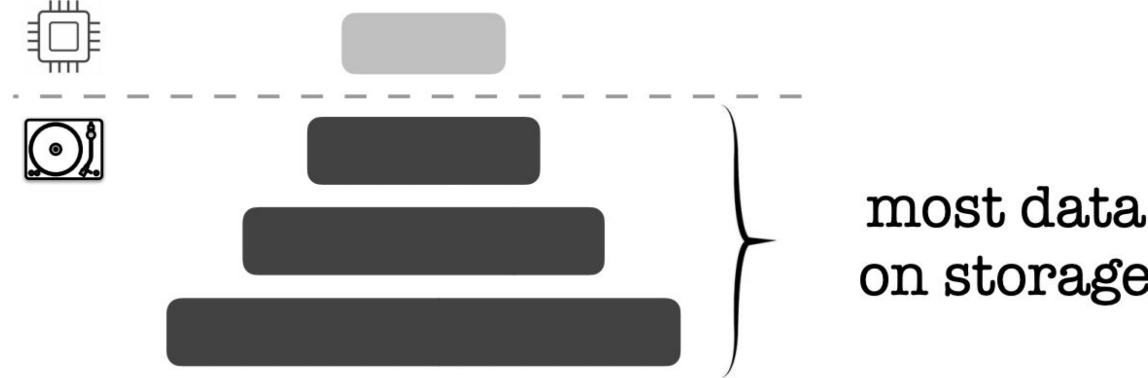
Action Space

Increase or decrease in compaction policies (level specific)

Reward

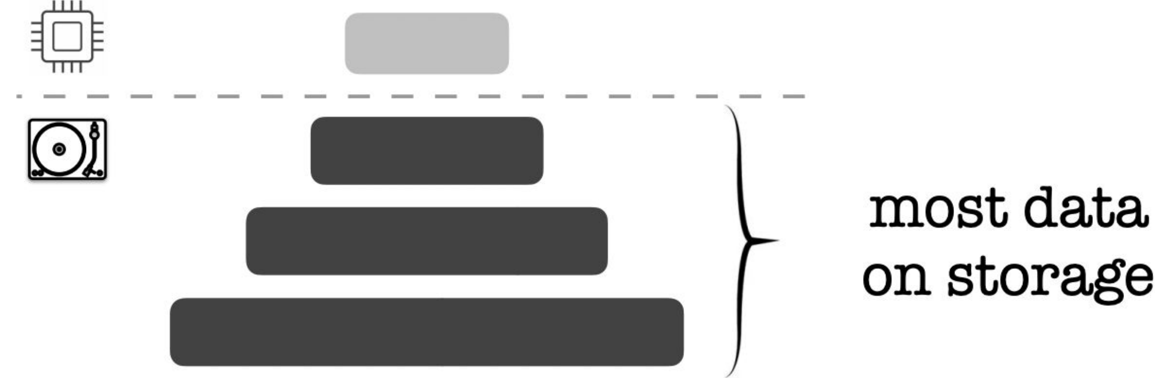
Minimize latency

Why Policy Propagation to Optimize Training?



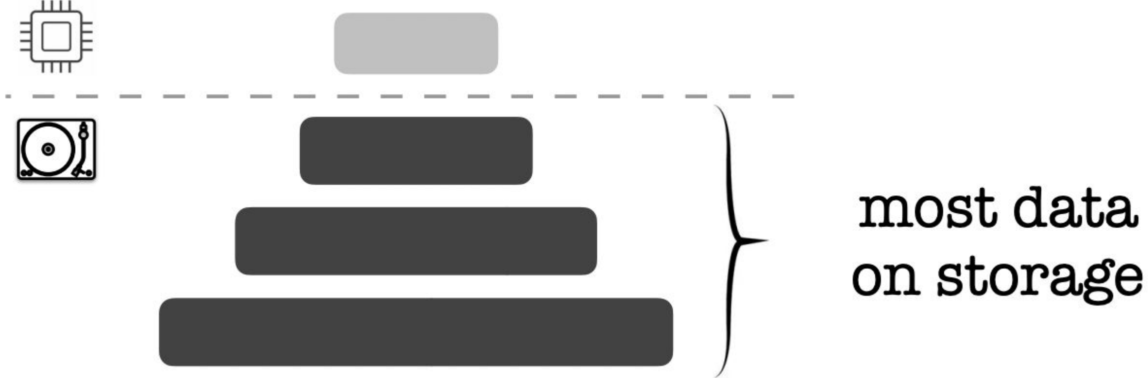
Deeper level size is exponentially larger

Why Policy Propagation to Optimize Training?



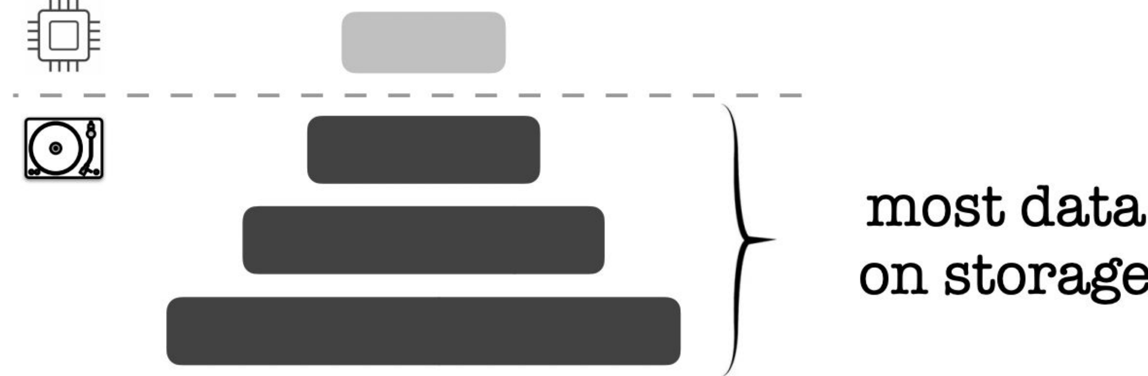
Deeper level size is exponentially larger
Compaction happens less frequently at deeper level

Why Policy Propagation to Optimize Training?



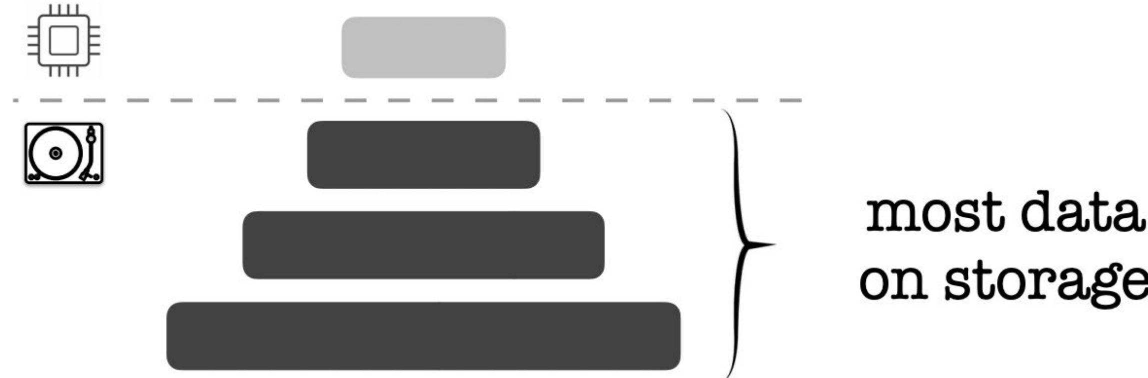
Deeper level size is exponentially larger
Compaction happen less frequently at deeper level
Less available training data

Why Policy Propagation to Optimize Training?



Deeper level size is exponentially larger
Compaction happen less frequently at deeper level
Less available training data
Deeper level needs more training data

Why Policy Propagation to Optimize Training?



How to learn the compaction policy at a deeper level?

Policy Propagation based on Cost Analysis

Uniform Bits-per-Key

Assigns the same bits-per-key to BF (per level)

R/W cost ratio is similar across level

Monkey Allocation

Shallow level assigns more bits-per-key than deeper level

Policy Propagation based on Cost Analysis

Uniform Bits-per-Key

Assigns the same bits-per-key to BF (per level)

R/W cost ratio is similar across level

Monkey Allocation

Shallow level assigns more bits-per-key than deeper level

Policy Propagation based on Cost Analysis

Uniform Bits-per-Key

Assigns the same bits-per-key to Bloom Filter (BF) (per level)

R/W cost ratio is similar across level
R/W amplification is same across level

Monkey Allocation

Shallow level assigns more bits-per-key than deeper level

Policy Propagation based on Cost Analysis

Uniform Bits-per-Key

Assigns the same bits-per-key to BF (per level)

Use RL model to learn the policy of level 1
Propagate the policy to all levels

Monkey Allocation

Shallow level assigns more bits-per-key than deeper level

Policy Propagation based on Cost Analysis

Uniform Bits-per-Key

Assigns the same bits-per-key to BF (per level)

Monkey Allocation

Shallow level assigns more bits-per-key than deeper level

R/W cost ratio is different

Policy Propagation based on Cost Analysis

Uniform Bits-per-Key

Assigns the same bits-per-key to BF (per level)

Monkey Allocation

Shallow level assigns more bits-per-key than deeper level

R/W cost ratio is different
Policy can vary across level

Policy Propagation based on Cost Analysis

Uniform Bits-per-Key

Assigns the same bits-per-key to BF (per level)

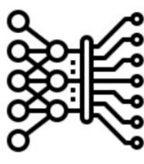
Monkey Allocation

Shallow level assigns more bits-per-key than deeper level

Infer remaining level policy based on the preceding two levels policy

Results

Experiment Design



Hardware:

Intel Xeon Gold 6326@2.9 GHz CPU, NVMe SSD, Ubuntu 22.04 OS.

Initial Data Load:

100 million kv entries

Key Size: 128B

Value Size: 896B

Mission:

100 million operations

(lookup or updates)

divided into 2000 mission

Baseline Comparison

- **Aggressive Compaction (K = 1):**
  - low read cost
  - high write amplification
- **Lazy Compaction (K = 10):**
  - high read cost
  - low write amplification
- **Moderate Compaction (K = 5):**
  - Balance between aggressive and lazy
  - moderate read/write amplification

K

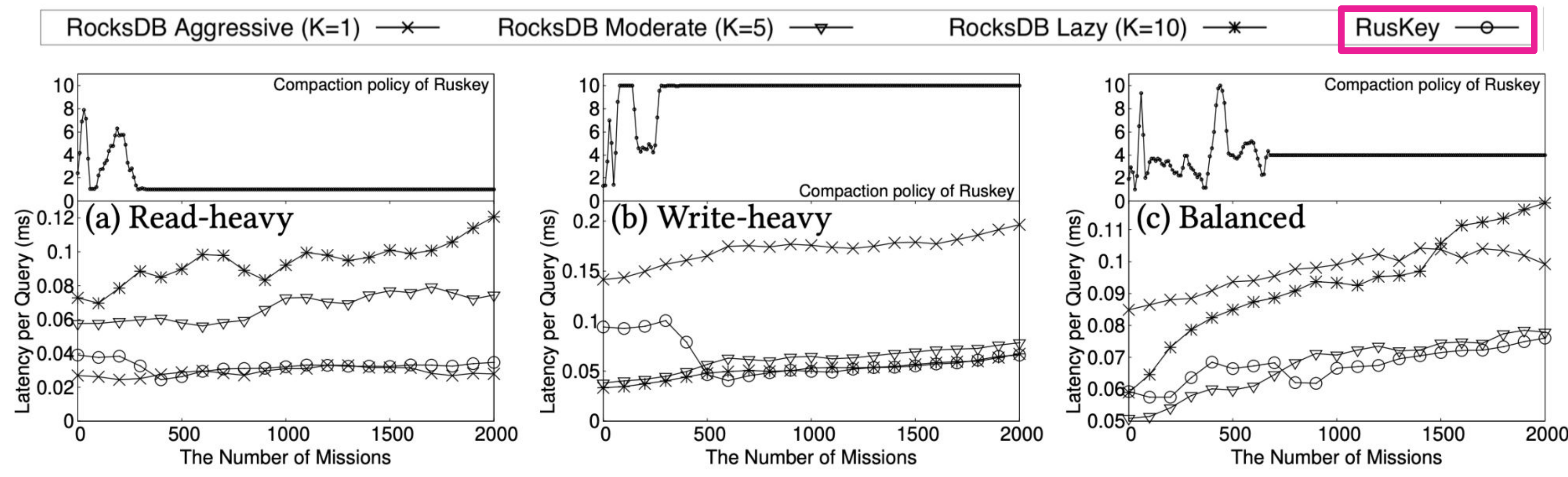
Max Number of sorted runs per level

Workload

- **Workload Types**:
  - read-heavy (10% update)
  - balanced (50% update)
  - write-heavy (90% update),
  - write-inclined (70% update)
  - read-inclined (30% update)

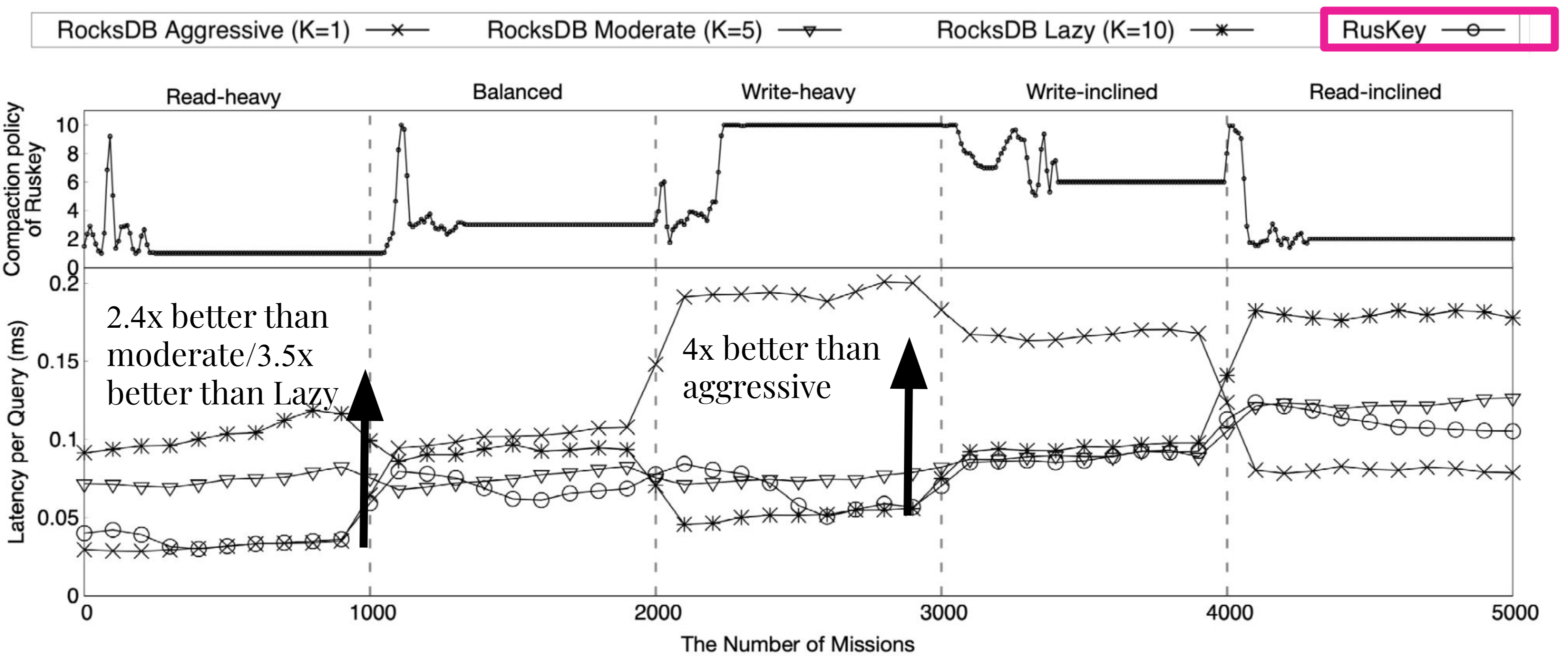
Each section includes 50 million operations which are divided into 1000 missions with 50000 operations for each.

Evaluation - Static Workload



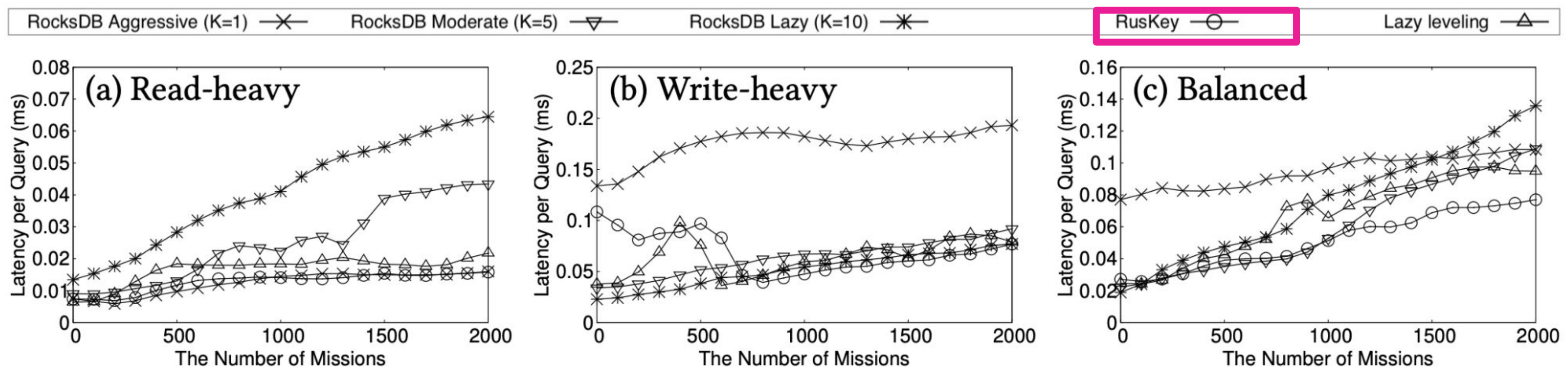
Ruskey achieves the lowest latency per query across all static workloads as missions increase

Evaluation - Dynamic Workload



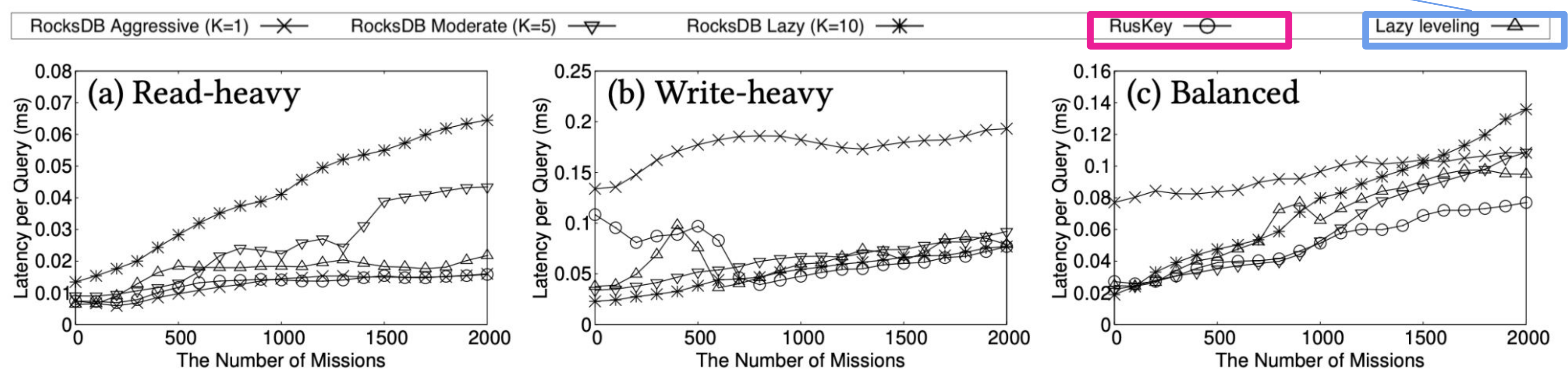
RusKey maintains near-optimal latency across all dynamic workload sessions, while other baselines exhibit sub-optimal performance in at least one session.

Ruskey Under Monkey Scheme



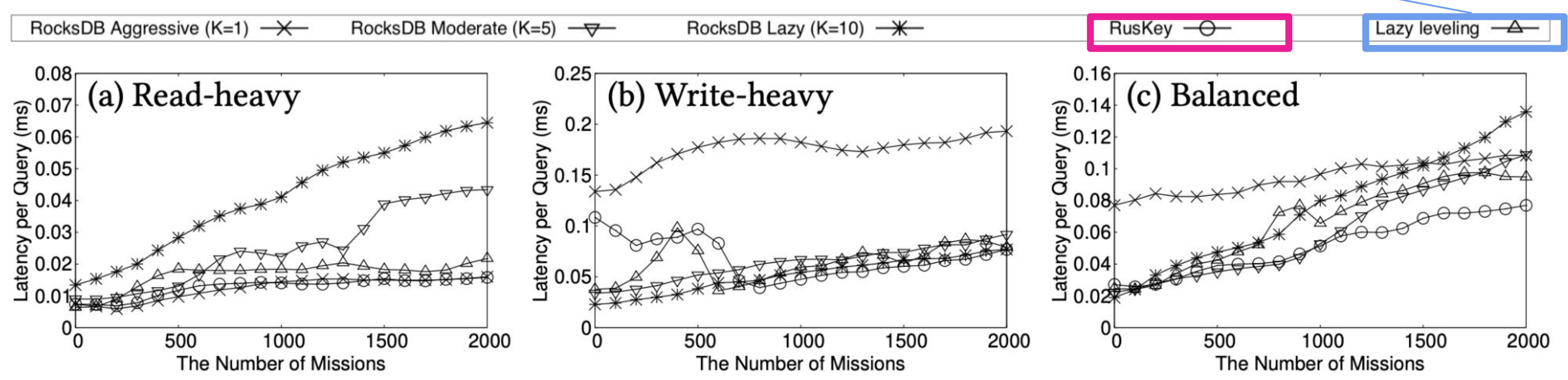
Compare RusKey with the baselines under the Monkey scheme with the same workload setting

Ruskey Under Monkey Scheme



Compare RusKey with the baselines under the Monkey scheme with the same workload setting

Ruskey Under Monkey Scheme



Ruskey and Lazy Leveling both achieve near-optimal performance, but Ruskey outperforms Lazy Leveling in every workload since it adopts novel policy setting through policy propagation

Compaction policy setting of RusKey

End-to-end latency (s)
4000
3000
2000
1000
0
Aggressive *Moderate* *Lazy* *Lazy-leveling* *RusKey*

By-level latency (s)
900
800
700
600
500
400
300
200
100
0
Write latency
Read latency
K=10 K=10 K=10 K=8 K=10 K=3 K=1 K=1
Lazy-leveling *RusKey* *Lazy-leveling* *RusKey* *Lazy-leveling* *RusKey* *Lazy-leveling* *RusKey*
Level 1 Level 2 Level 3 Level 4

25 million operations of r/w balanced workload under the Monkey scheme

Compaction policy setting of RusKey

The running time of processing all the operation

End-to-end latency (s)
4000
3000
2000
1000
0
Aggressive *Moderate* *Lazy* *Lazy-leveling* *RusKey*

By-level latency (s)
900
800
700
600
500
400
300
200
100
0
Write latency
Read latency
K=10 K=10 K=10 K=8 K=10 K=3 K=1 K=1
Lazy-leveling *RusKey* *Lazy-leveling* *RusKey* *Lazy-leveling* *RusKey* *Lazy-leveling* *RusKey*
Level 1 Level 2 Level 3 Level 4

25 million operations of r/w balanced workload under the Monkey scheme

Compaction policy setting of RusKey

The running time of processing all the operation

End-to-end latency (s)
4000
3000
2000
1000
0
Aggressive *Moderate* *Lazy* *Lazy-leveling* *RusKey*

By-level latency (s)
900
800
700
600
500
400
300
200
100
0
Write latency
Read latency
K=10 K=10 K=10 K=8 K=10 K=3 K=1 K=1
Lazy-leveling *RusKey* *Lazy-leveling* *RusKey* *Lazy-leveling* *RusKey* *Lazy-leveling* *RusKey*
Level 1 Level 2 Level 3 Level 4

25 million operations of r/w balanced workload under the Monkey scheme
Different Compaction policy at each level (Ruskey)

Compaction policy setting of RusKey

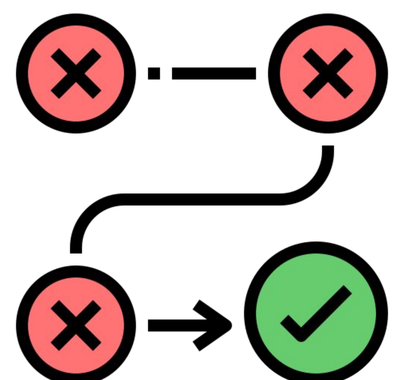
The running time of processing all the operation

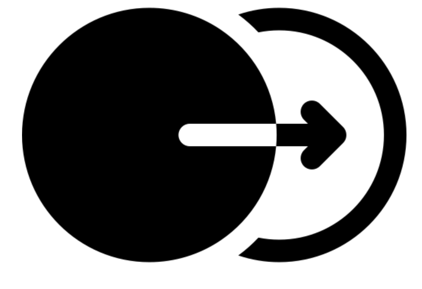
End-to-end latency (s)
4000
3000
2000
1000
0
Aggressive Moderate Lazy Lazy-leveling RusKey

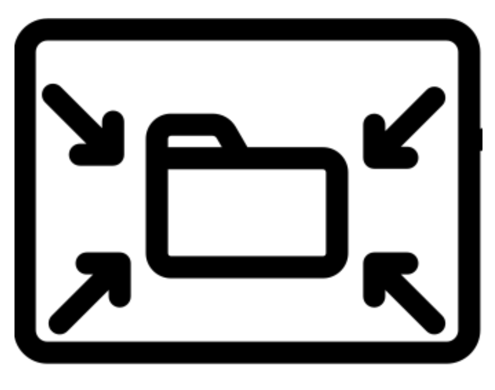
By-level latency (s)
900
800
700
600
500
400
300
200
100
0
Write latency
Read latency
K=10 K=10 K=10 K=8 K=10 K=3 K=1 K=1
Lazy-leveling RusKey Lazy-leveling RusKey Lazy-leveling RusKey Lazy-leveling RusKey
Level 1 Level 2 Level 3 Level 4

RusKey achieves optimal end-to-end and by-level latency by self-tuning its compaction policy under a balanced workload

Summary







Reinforcement Learning

Efficient Transitions

Reduce Required Data