Introduction

LSM-Tree에서 flush와 compaction을 포함한 background process가 table-based statistics을 파괴 하기 때문에 전통적인 replace cache방식(LRU,LFU등)이 무효(cache invalidation)로 된 현상이 있음. 이러한 현상이 일어날때 LSM-Tree storage engine은 성능의 급격한 변동을 일으키고 시스템의 안정성을 저하시키며 사용자의 조작도 영향을 이르킬수 있음.

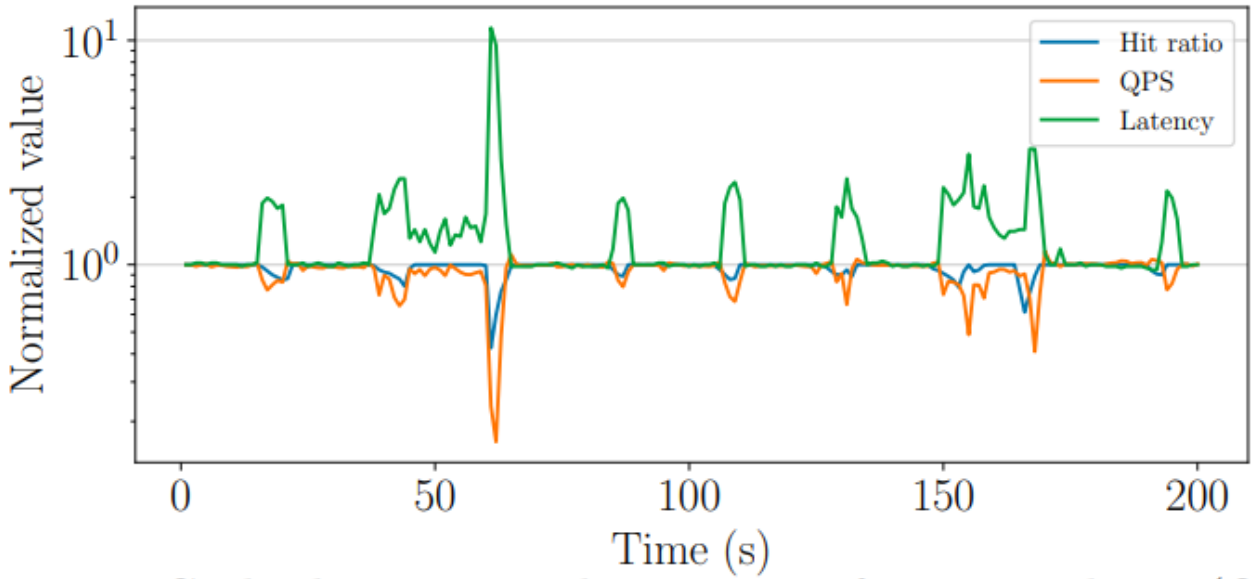


Figure 1: Cache hit ratio and system performance churn (QPS and latency of 95th percentile) caused by cache invalidations.

Background and preliminary

FORMULATION 2. ***Cache invalidation problem.*** *Given a database cache* $C = \{r_0, r_1, ..., r_{L-1}\}$ *determined by a cache replacement policy, and a set of records participate in a compaction (or flush)* $M_i = \{r_i^0, r_i^1, \cdots, r_i^{n_i-1}\}$, *the invalidated cache can be represented as* $|C \bigcap M_i|$, *where* M_i *and* n_i *refer to a compaction (or flush) and the number of records moved by it,* $n_i, i \in N$. *The cache invalidation problem is defined to minimize the invalidated cache for a compaction or flush.*

cache invalidation 생기는 이유:

- storage engine의 compaction와 flush로 인해 생긴 Mi 가 원래의 block 통계에 영향을 주기 때문에 기존에 있던 replace cache가 무효로 됨.
- 이러한 문제로 영향을 받지 않은 통계기준---key range based 로 통해 compaction & flush 과정에서 machine learning을 통해 방문할 key range를 예측하고 미리 일어와 cache invalidation을 방지함.

Design

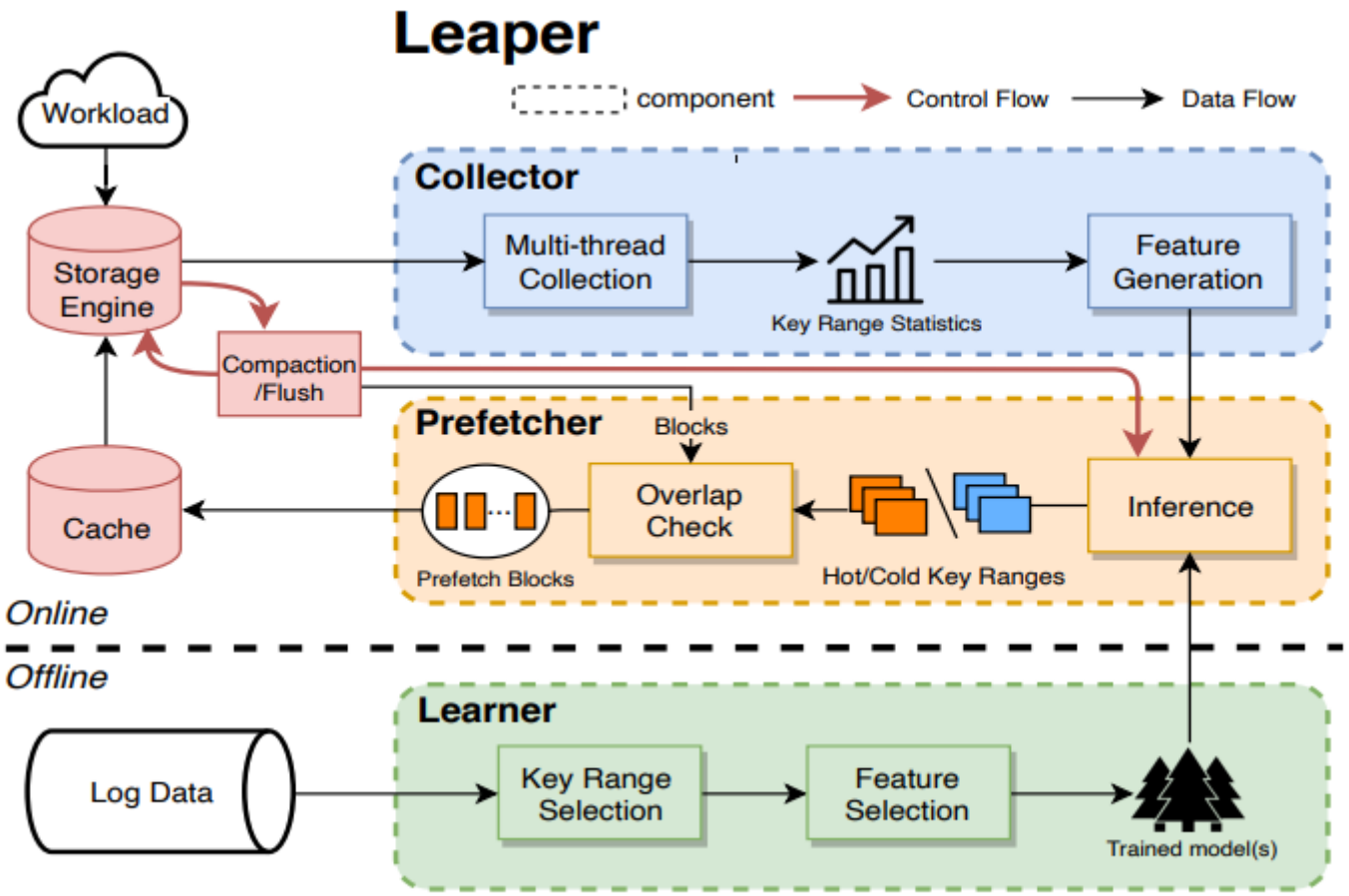


Figure 3: Workflow of LSM-tree storage engine with Leaper.

Leaper = Learned prefetcher

3개 module로 구성됨

- Learner : log에서 key의 방문 기록을 가지고 cost를 줄이기 위해 연속된 key는 key range로 변경해서 classification모델로 학습. Learner은 workload의 특성을 파악하고 key range의 크기를 결정함. Learner가 model을 훈련시키고 시간 따라서 미래의 방문할 key range를 예측. Gradient Boosting Decision Tree (GBDT)를 사용함.
- Collector : multi thread 방식으로 online access data를 수집.
- Prefetcher : collector가 제공한 feature 을 입력 & learner가 제공한 모델을 통해 미래 access 할 key range를 예측. 예측결과로 prefetcher가 모든 data block 을 overlap 처리후 cache에 제장 & access

하지 않은 block은 제거.

Key Range Selection

Algorithm 1: Key Range Selection

Input: Total key range T, initial size A , access information and decline threshold α

Output: Most suitable granularity A^*

1 Initialize access matrix M ($N \times \frac{T}{A}$ bits for N time intervals and $\frac{T}{A}$ key ranges) for key range size A;

2 Define the number of zeroes in M as $Z(A)$;

3 **while** $2 \cdot Z(2A) > \alpha \cdot Z(A)$ **do**

4 $\quad A \leftarrow 2A$;

5 Binary search to find the maximum value A^* satisfying $A^* \cdot Z(A^*) > \alpha A \cdot Z(A)$ from A to $2A$;

6 **return** A^*

- classification model의 data를 만들기 위해 data block의 access information을 회득해야함. 하지만 disk에 있는 data block들이 merge되고 compaction되면서 각 data block의 상세를 정확히 알수 없음. key range selection은 연속으로 된 key를 range로 만들어 DB가 작업을 수행할때 변경없이 사용할수 있도록 함. cost 감소 & lsm-tree의 data 구조와 부합 & range query성능이 좋음.

Overlap Check

Key range selection 으로 생성된 key range는 data block하고 대응한 관계가 없기 위해 Prefetcher로 통해 예측후 flush와 compaction이 실행되면 어느 block에서 read할지 확정해야 함.

Algorithm 3: Check Overlap Algorithm

Input: Target blocks $\{(A_i, B_i)\}_{i=1}^{m}$, hot key ranges $\{(a_j, b_j)\}_{j=1}^{n}$

Output: Prefetch Data T

/* Binary Search: $O(nlogm) < O(m)$ */

$start = A_1, end = A_m$;

for a_j *in hot key ranges* **do**

  Binary Search for $A_i \leqslant a_j < A_{i+1}$ from start to end;

  while $b_j > A_{i+1}$ **do**

    if $Min(B_i, b_j) \geqslant a_j$ **then**

      T.Add((A_i, B_i));

    $i \leftarrow i + 1$;

  $start = A_{i+1}$;

/* Sort-merge: $O(m) \leqslant O(nlogm)$ */

for A_i *in target blocks and* a_j *in hot key ranges* **do**

  if $Min(B_i, b_j) \geqslant Max(A_i, a_j)$ **then**

    T.Add((A_i, B_i));

  if $B_i < b_j$ **then**

    $i \leftarrow i + 1$;

  else if $B_i > b_j$ **then**

    $j \leftarrow j + 1$;

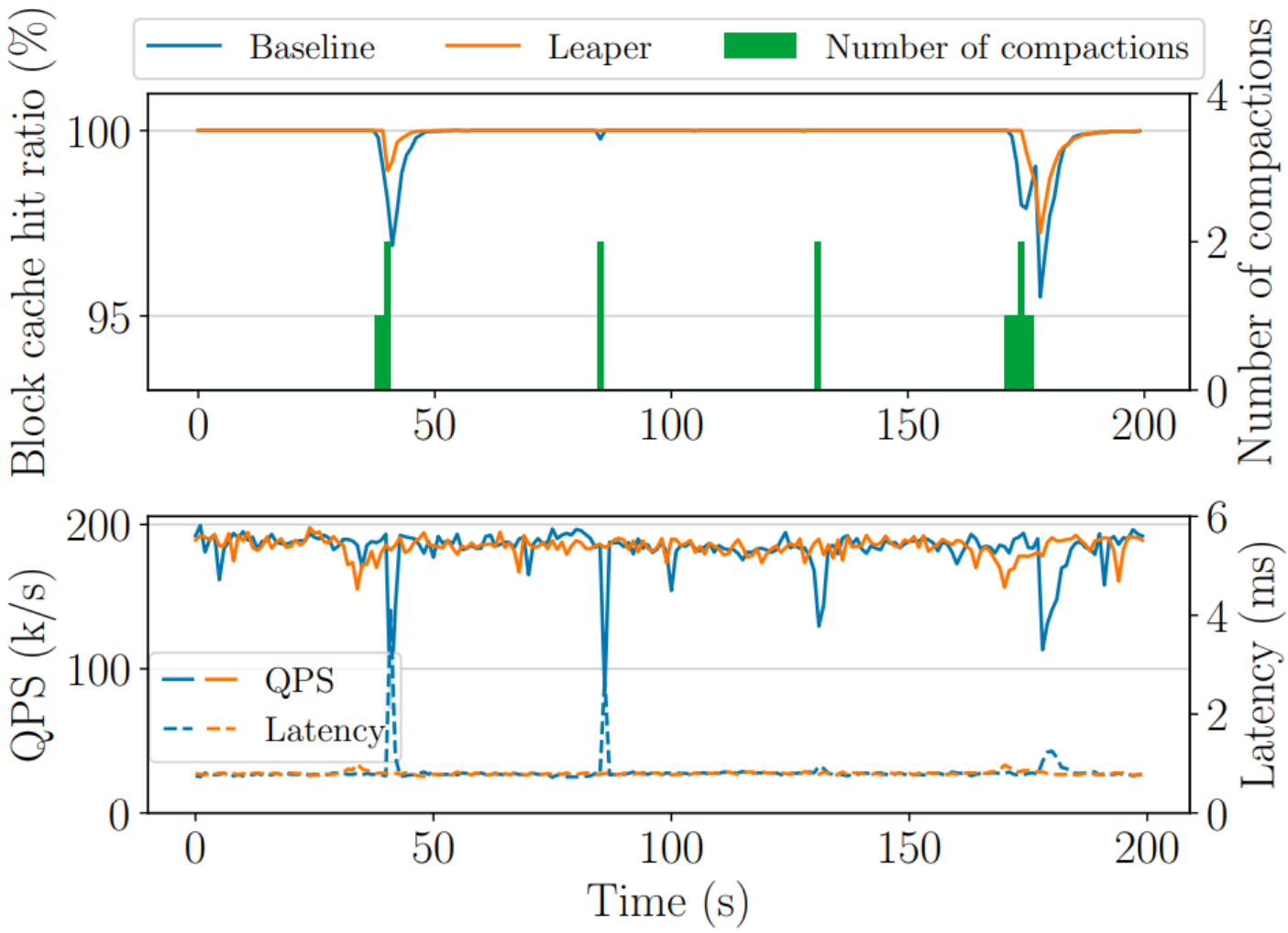
  else

    $i \leftarrow i + 1, j \leftarrow j + 1$;

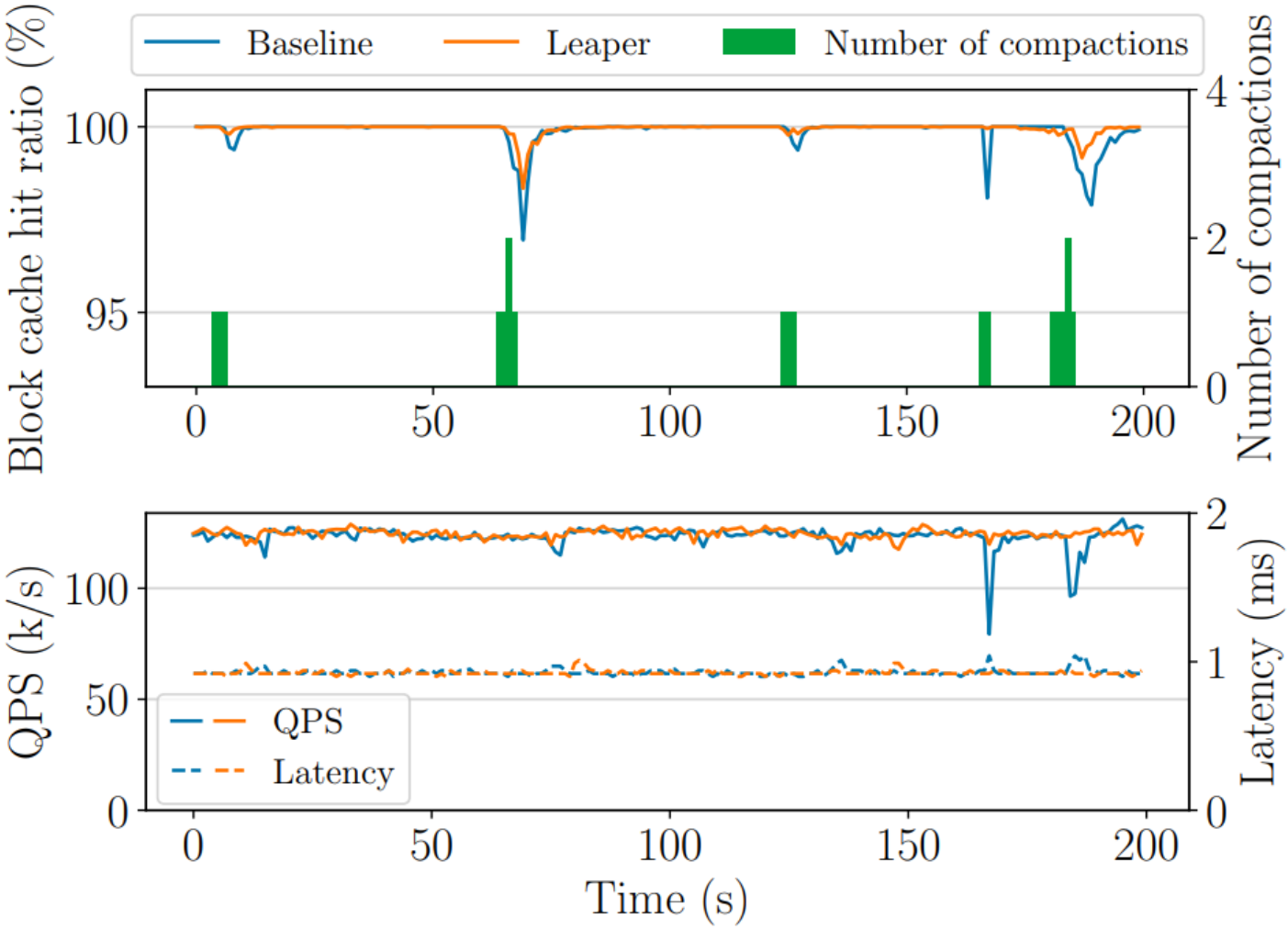
return T

이 알고리즘은 binary search & merge sort를 종합되어 hot data block을 찾는데 더 적은 cost를 소비함.

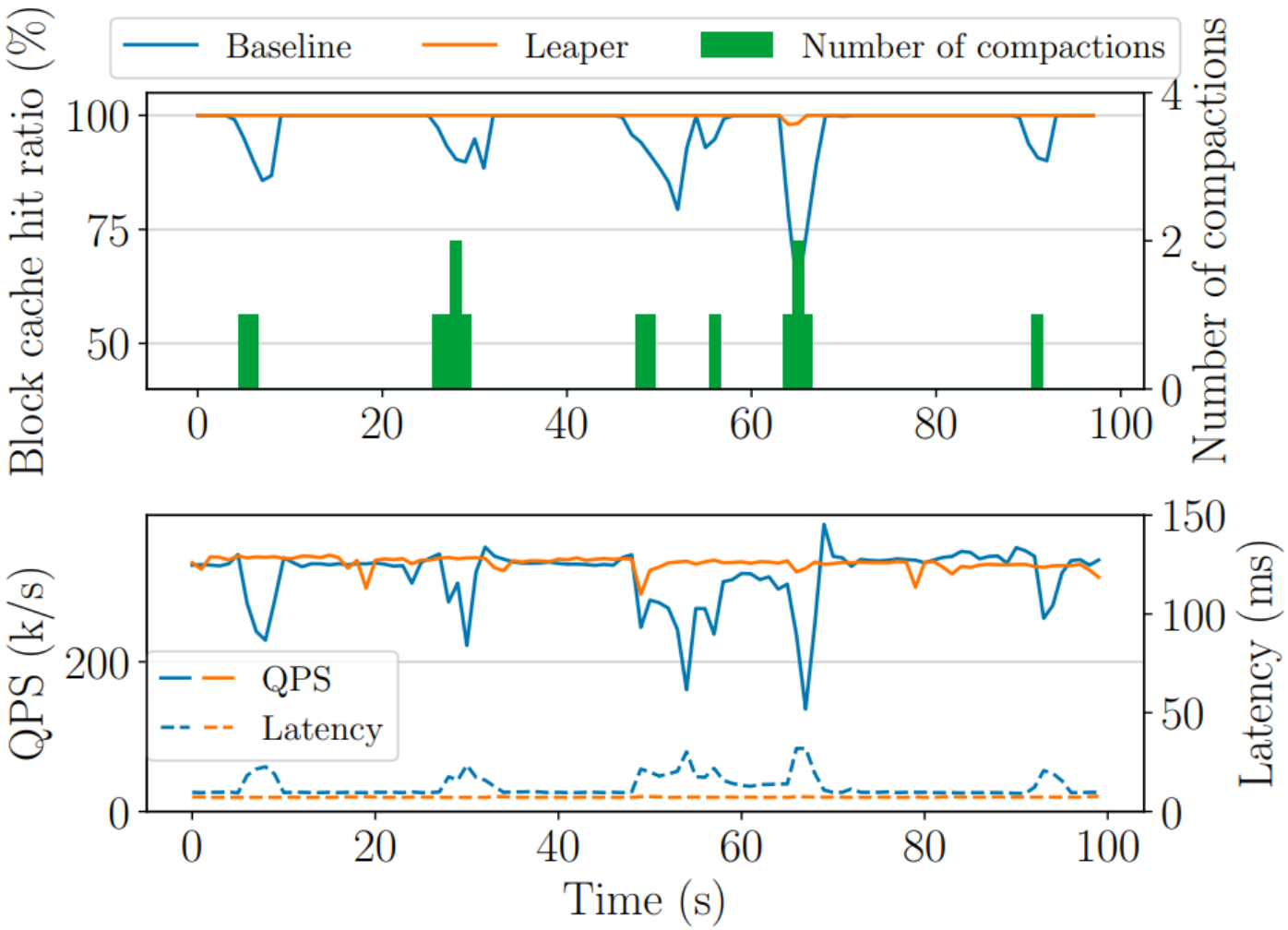
Evaluation



(a) E-commerce



(b) Instant messaging



(c) Synthetic workload

LSM-Tree based storage engine 의 cache invalidation문제를 파악하고 machine learning을 통해 해결.