In-Memory DB 성능 최적화를 위한 Hybrid Learned Index 설계 및 검증

In-Memory Team
강호현, 김민구, 리저우웨원, 이재열, 최규빈

DANKOOK UNIVERSITY

CONTENTS

01
연구배경

RocksDB 개요

- LSM-Tree 기반의 Key-Value Store로, 빠른 쓰기 성능을 제공
- MemTable을 활용하여 메모리에서 데이터를 정렬 유지하며, 대규모 데이터 환경에서도 효율적으로 작동할 수 있도록 설계

이전 실험

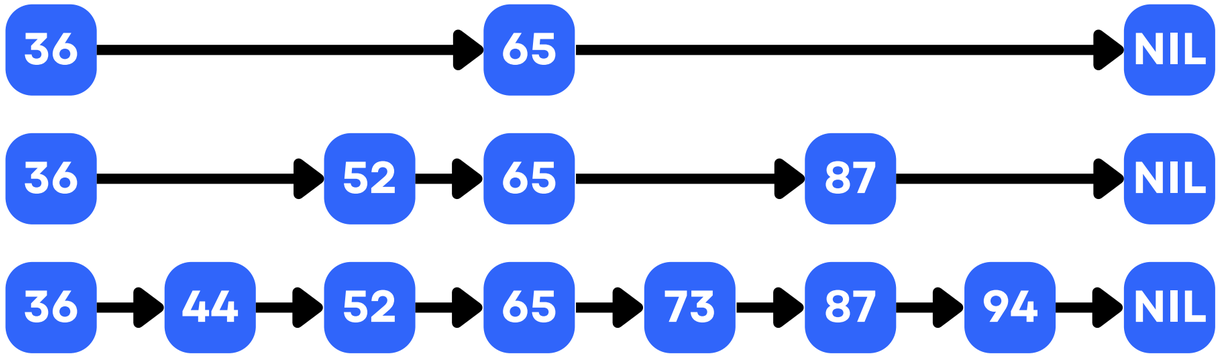
- 자료구조 SkipList, Hash SkipList, Hash LinkedList 성능 비교
- 대용량 데이터에서 SkipList가 가장 안정적인 모습을 확인

문제 제기

- SkipList가 가장 안정적이었지만 데이터가 커질수록 포인터 탐색 비용(Pointer Chasing)이 급증하는 모습을 보임
- 이는 불필요한 메모리 접근으로 인한 Cache Miss가 발생해서 성능 저하의 주원인이 됨

02 Learned Index?

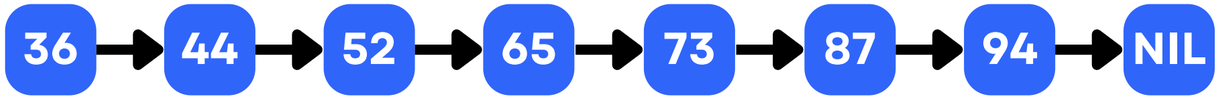
기존 탐색 방식



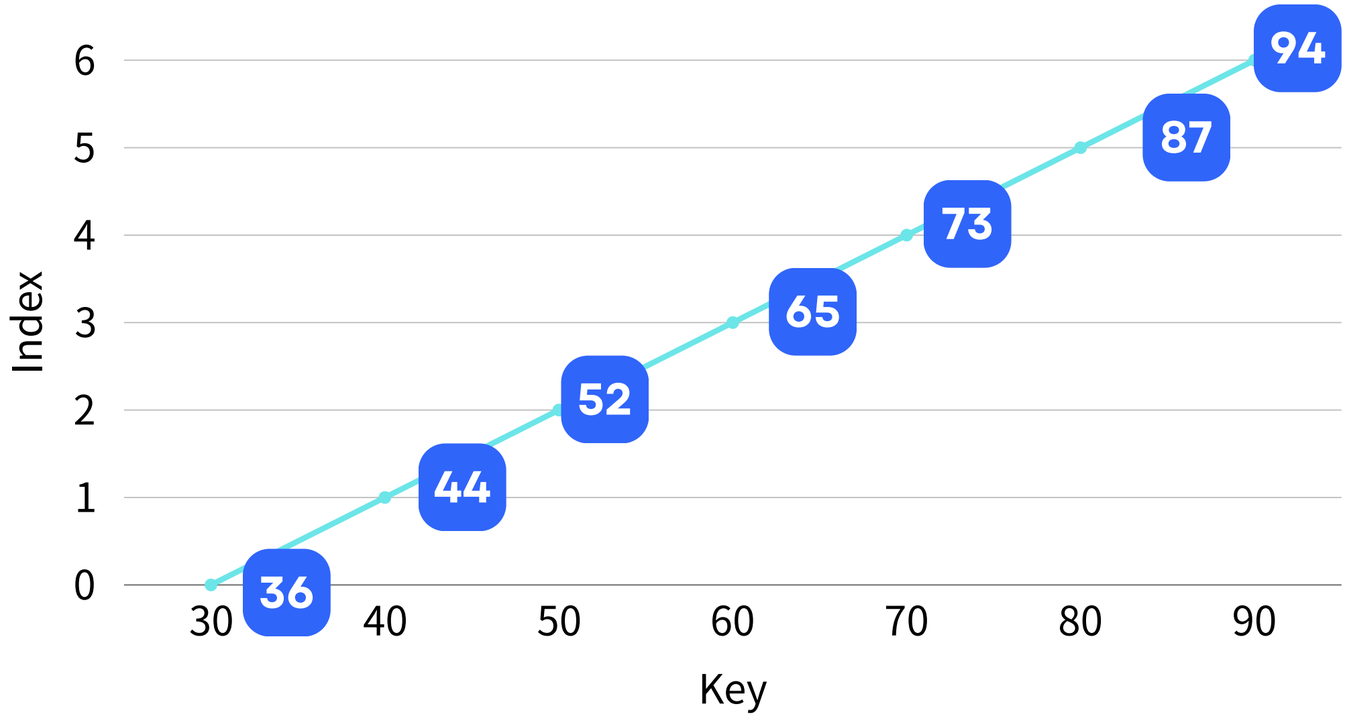
- 각 레벨에서 key를 비교하며 왼쪽→오른쪽→아래 방향으로 이동하며 탐색
- 이 또한 충분히 효율적인 탐색 방식이지만, 데이터가 많아질수록 탐색 시간 증가 - O(logN)

02 Learned Index?

Learned Index 방식



- 정렬된 배열 안의 키와 인덱스의 관계를 그래프로 표현하면...



- **Index = (Key * 0.1) - 3**의 직선 그래프로 근사값 계산 가능
- 처음부터, 혹은 절반씩 줄이면서 탐색할 필요 X → 근사값 근처만 탐색하면 끝
- 머신 러닝을 이용해 이보다 훨씬 복잡한 관계의 Key-Index도 함수 생성 가능

02 Learned Index?

Learned Index 방식

장점

- 인덱스 데이터 크기가 작음(포인터/노드 오버헤드↓) → 메모리 효율 증가
- 탐색 범위가 작아져 대량 데이터 읽기 속도↑

단점

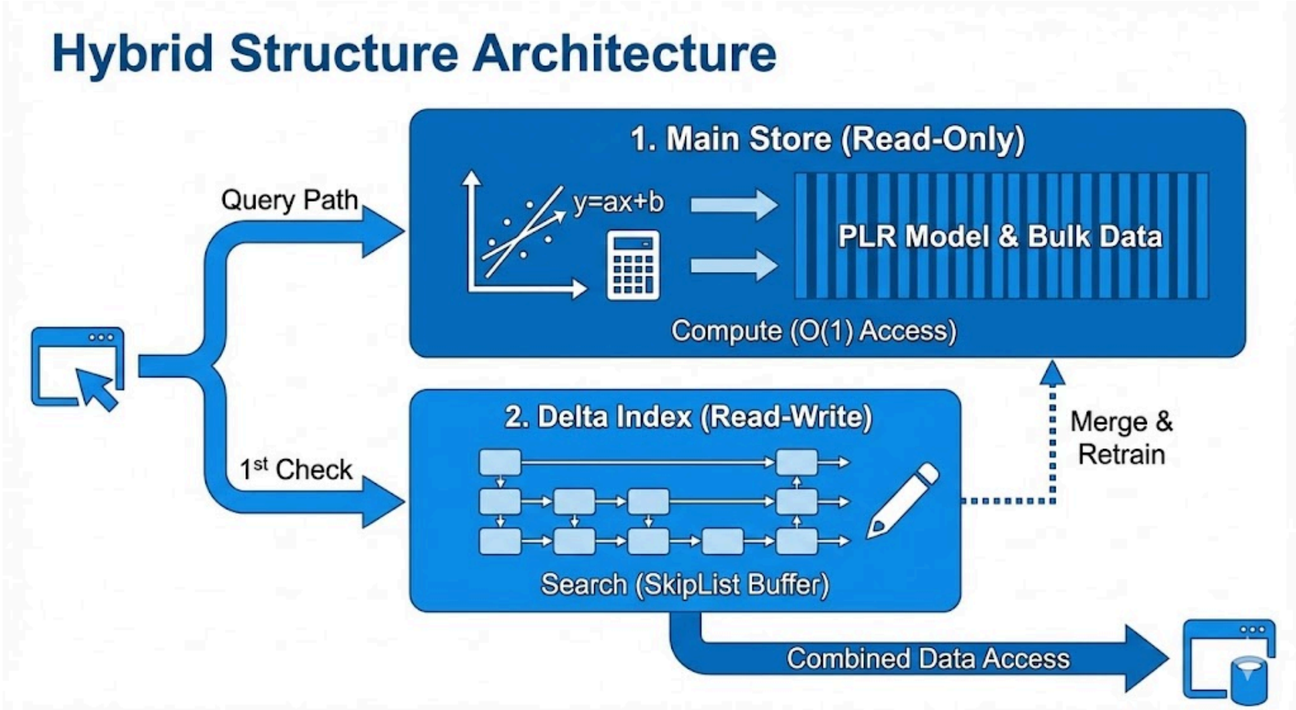
- 데이터 분포 변경시 인덱스 생성, 모델 재학습 필요(Build Time 비용 ↑)
  → 잦은 업데이트가 발생하는 환경에서는 Build Time으로 인한 손해↑

02
해결방안

해결방안

- 핵심 아이디어: 탐색(Search)하지 말고 계산(Compute)하자.
- 도입 기술: Learned Index (기계 학습 기반 인덱싱)
- 제안 아키텍처 (Hybrid Structure)
  1. Main Store (Read-Only): 대용량 데이터는 PLR (선형 회귀) 모델로 관리 (O(1) 접근).
  2. Delta Index (Read-Write): 실시간 데이터는 소규모 SkipList 버퍼로 관리.

     정적 데이터의 조회 속도와 동적 데이터의 쓰기 유연성을 결합.



03
실험설계

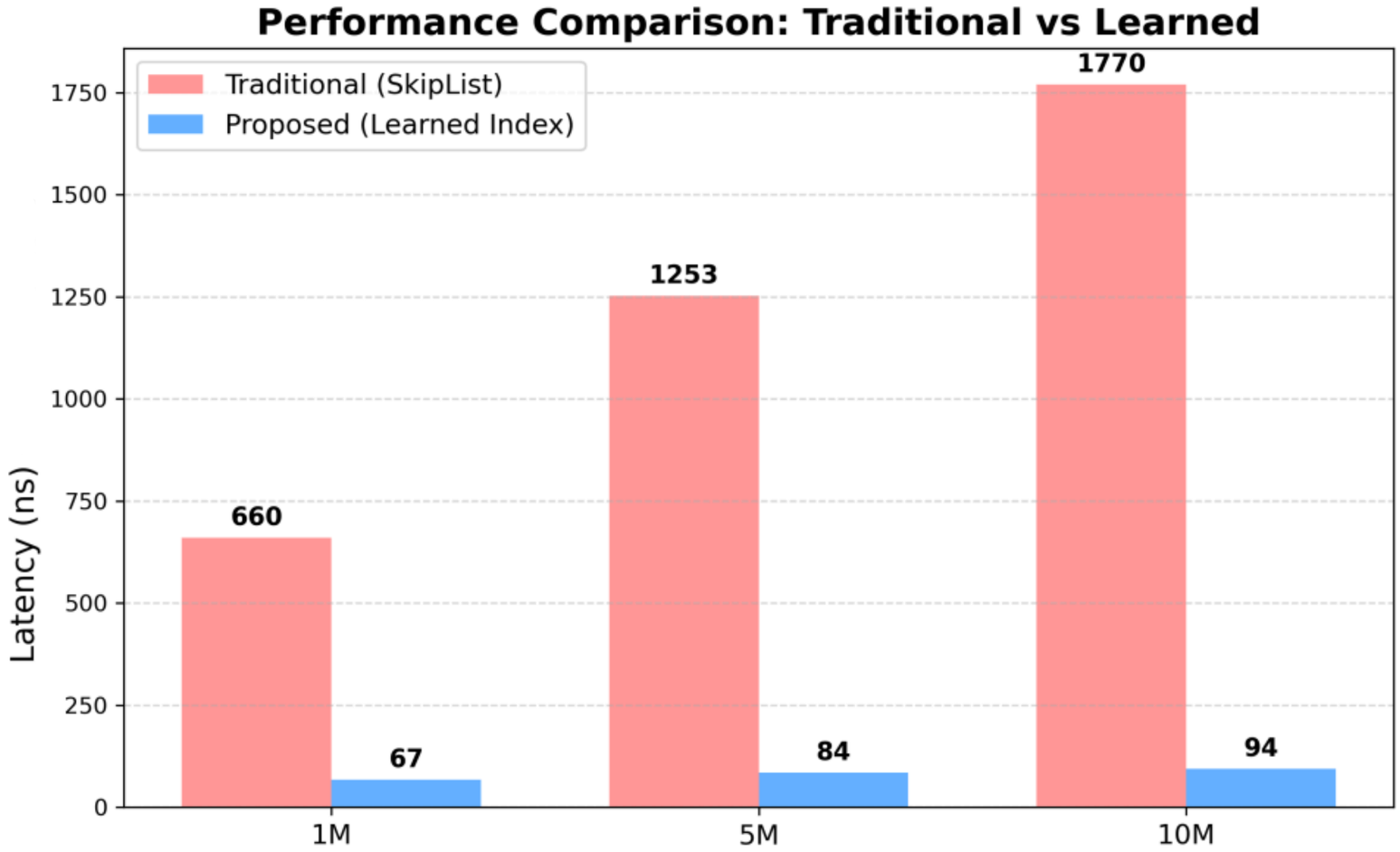
실험 환경 (Experimental Setup)

환경	VirtualBox, Ubuntu Linux, 4GB RAM (제한된 메모리 환경 가정).
데이터	총 데이터: 100만 개 (1M), 500만 개(5M), 1,000만 개 (10M). Key: Random Gap 분포 (비선형성 부여). Value: 128 Bytes Payload (현실적인 워크로드 반영).
비교 방식	Baseline: 기존 SkipList (RocksDB Default). Proposed: Hybrid Index (PLR 비율 100% 적용 시).

04

실험 결과 I

속도 검증



- **제안 모델이 기존 대비 약 18.8배 빠른 속도를 기록함. (1,770ns vs 94ns)**

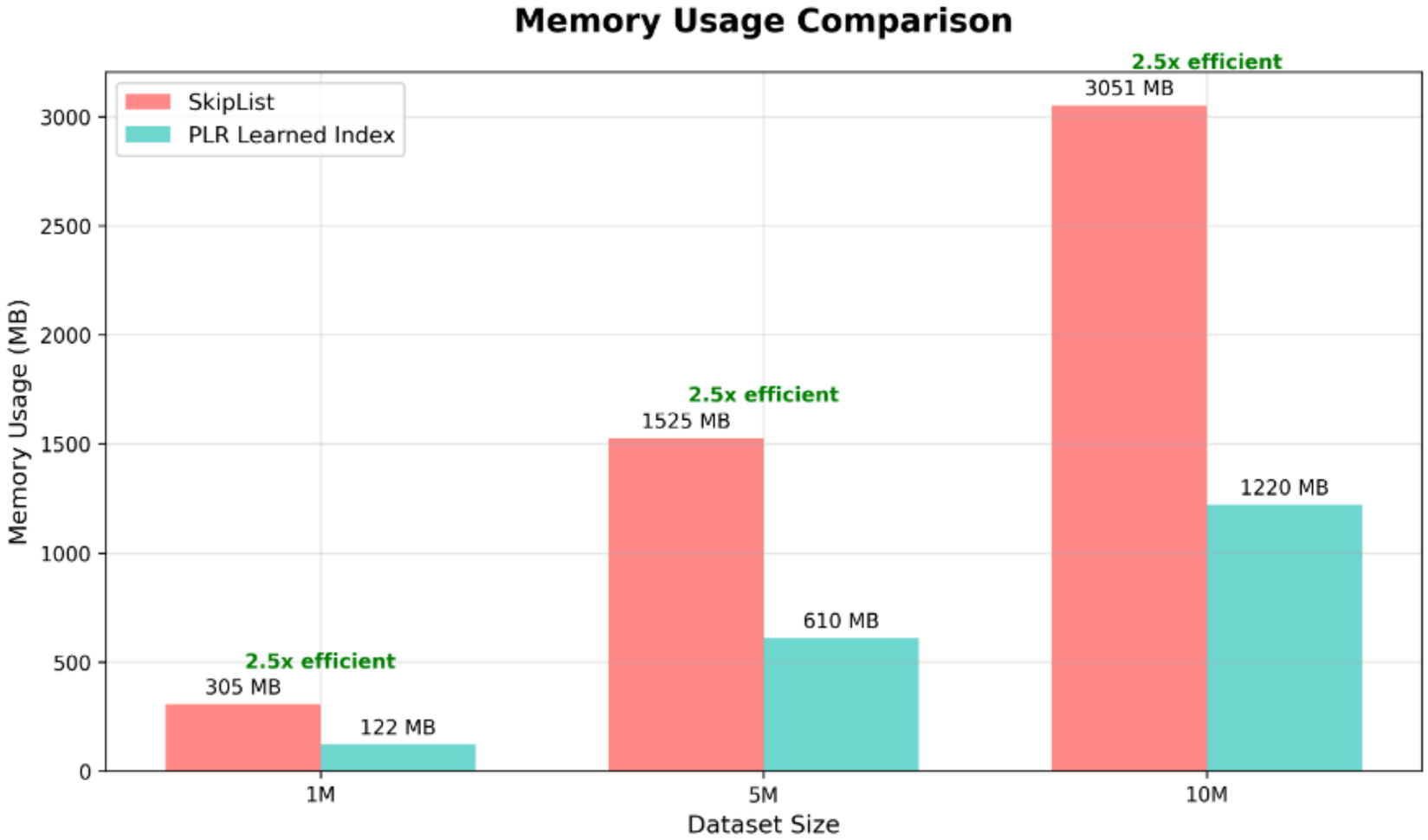
SkipList: O(log N) 복잡도로 데이터 증가 시 성능 저하 뚜렷 (1M → 10M 시 2.7배 느려짐).
Learned Index: O(1) 복잡도로 데이터 양과 무관하게 일정한 성능 유지 (Scalability 확보).

결론: CPU Cache Hit율을 극대화하여 메모리 병목 현상을 해결함.

05

실험 결과 II

효율성 검증



- **동일한 데이터를 저장하는 데 메모리를 약 2.5배 절약함 (2.5x Efficient).**

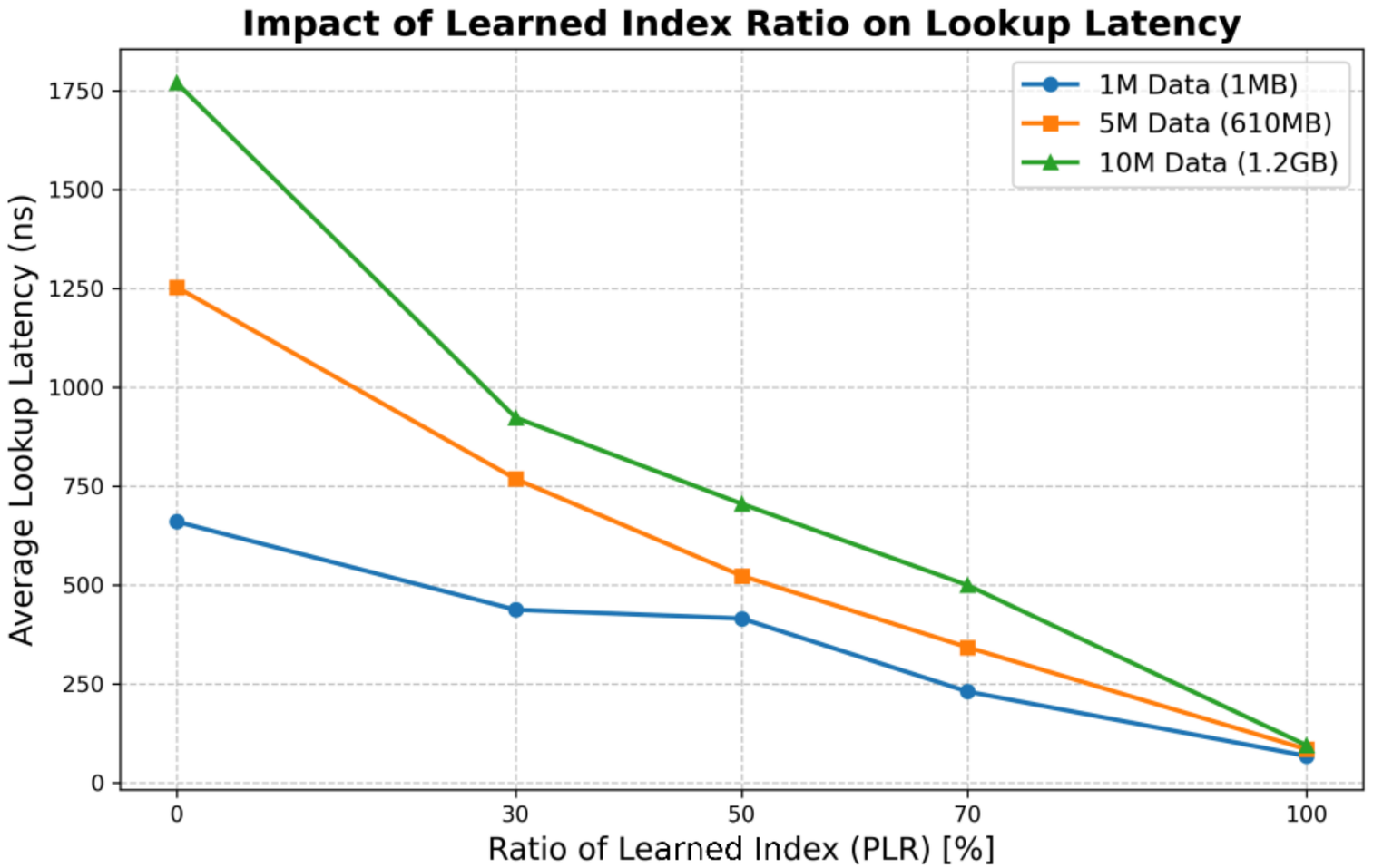
SkipList: 각 노드마다 다수의 포인터(Pointer, 8 Bytes)를 유지해야 하므로 오버헤드가 큼.
Learned Index: 데이터를 연속된 배열(Vector)에 저장하고 포인터를 제거하여 공간 낭비 최소화.

제한된 램(4GB) 환경에서 더 많은 데이터를 처리할 수 있음.

05

실험 결과 II

효율성 검증



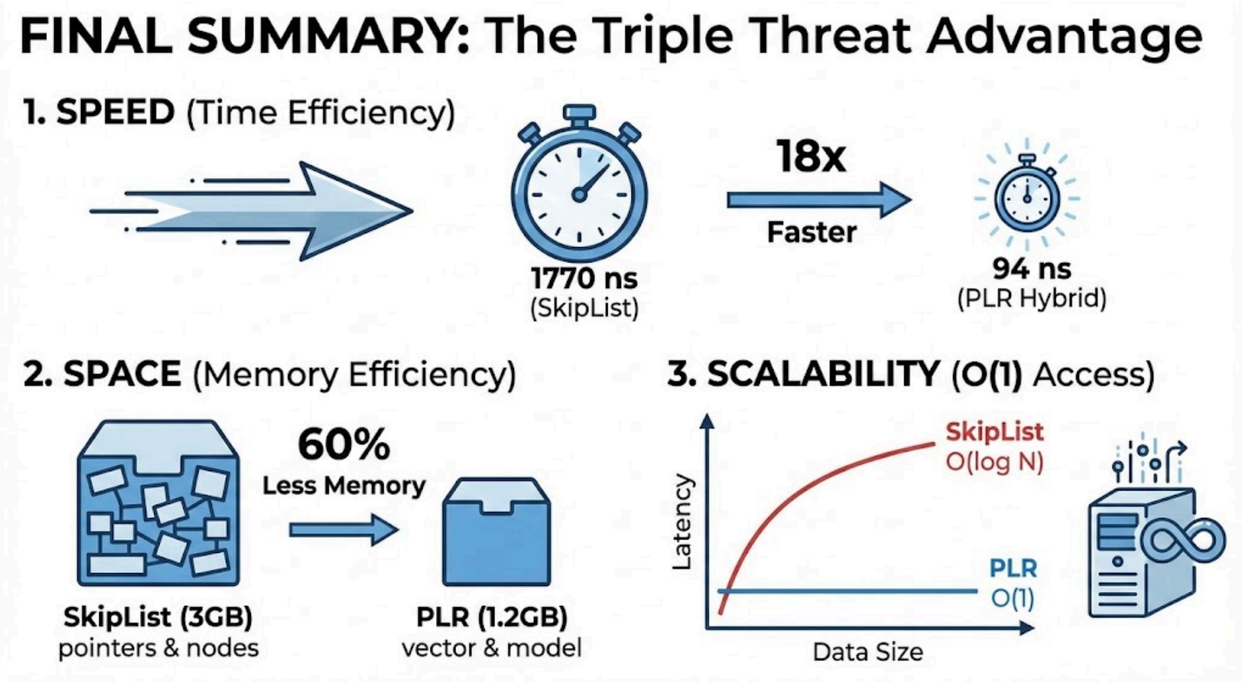
- **데이터 규모가 클수록(Big Data) 성능 격차가 더 벌어짐 (최대 22.6배 가속).**

단순히 작은 데이터셋에서만 빠른 것이 아니라, 실제 대용량 DB 환경에서 더욱 효과적인 알고리즘임을 보임.

06
결론

최종 요약

- 속도: 조회 Latency 18배 이상 단축 (Time Efficiency).
- 공간: 메모리 사용량 60% 감소 (Space Efficiency).
- 확장성: 데이터 증가에 따른 성능 저하 없음 (O(1)).

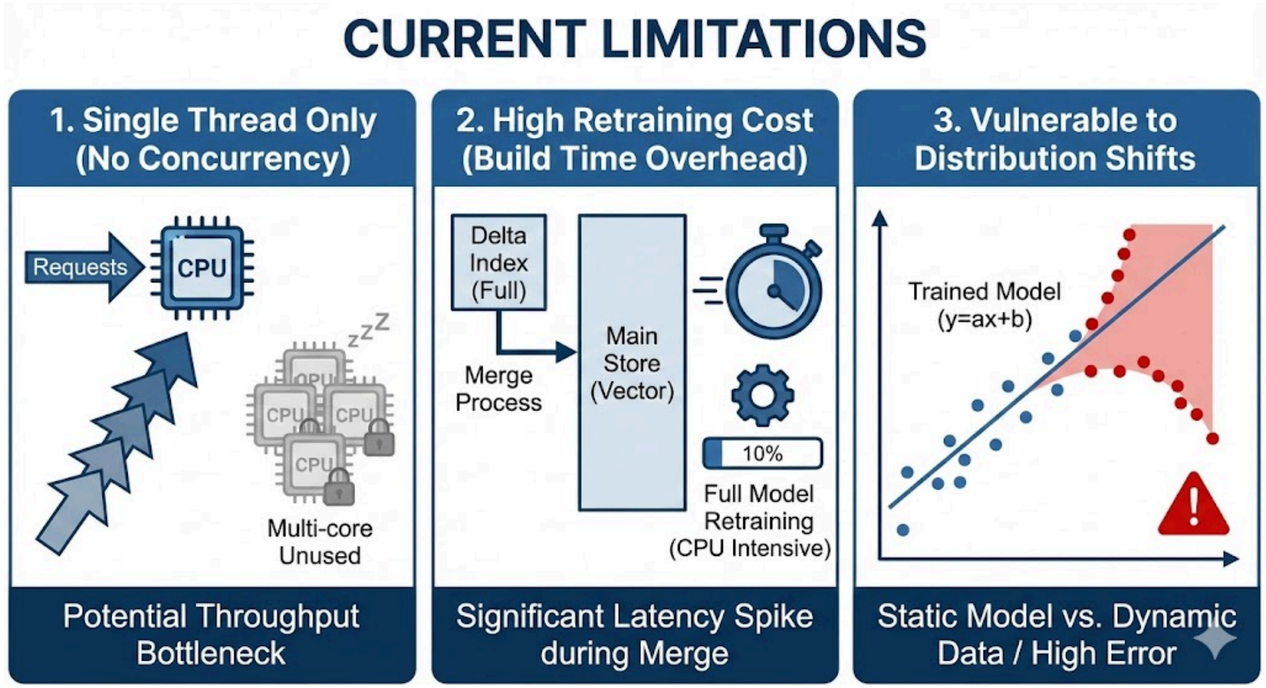


06

결론

한계점 (Limitations)

- 동시성 제어(Multi-thread) 미구현.
- Delta Index 임계치 초과 시 병합(Merge)하고 PLR 모델의 재학습 비용 문제.
- 데이터 복잡도에 따른 비선형적 데이터 처리.

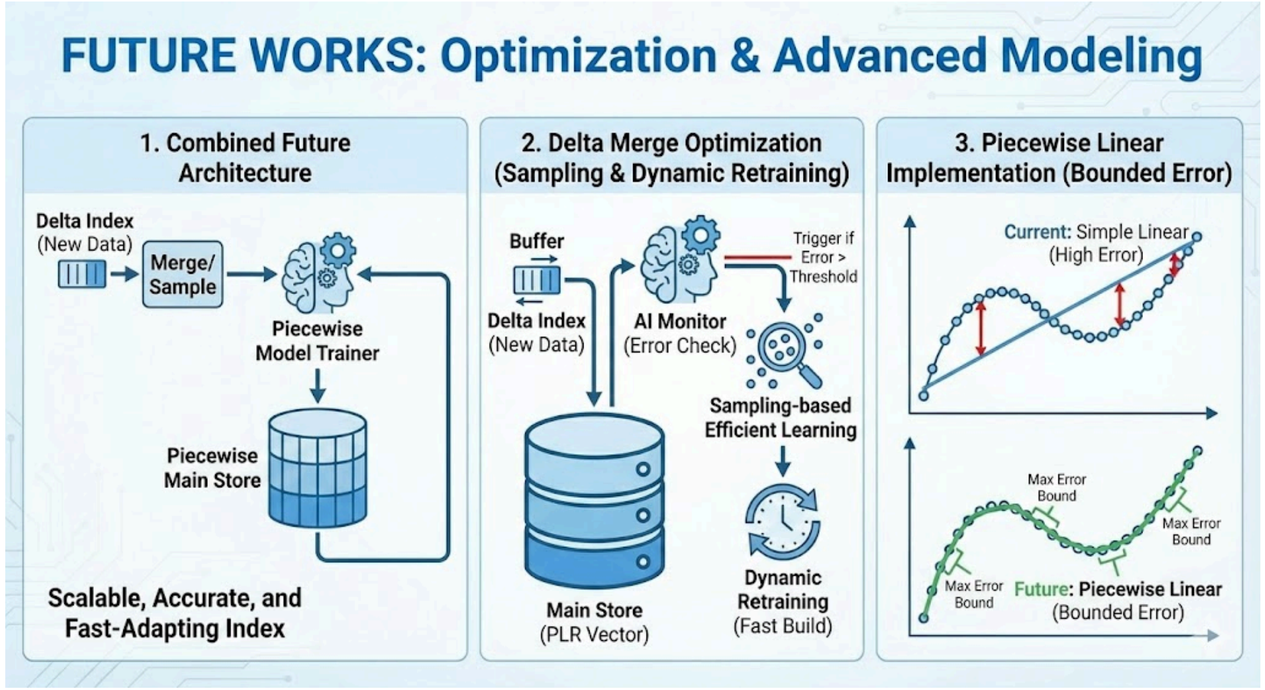


06

결론

향후 계획 (Future Work)

- Delta Index 병합(Merge) 정책 최적화
- 샘플링 기반 학습 (Sampling-based Efficient Learning) & 동적 재학습(Dynamic Retraining) 구현.
- 지정한 오차를 넘지 않도록 Piecewise(구간별) Linear을 구현.



참고 문헌

1. The Case for Learned Index Structures (SIGMOD 2018)

저자: Google (Tim Kraska, Jeff Dean 등)

내용 : B-Tree 같은 전통적인 인덱스를 머신러닝 모델(Learned Index)로 대체
→ 본 실험에선 B-Tree가 아닌 SkipList로 다룸

2. Bourbon: A Learned Index for Log-Structured Merge-trees (OSDI 2020)

저자: Dai et al.

내용 : RocksDB(LSM-Tree)에 Learned Index를 디스크에 저장된 파일(SSTable)을 타겟으로 실제로 적용
→ 본 실험에선 디스크가 아닌 메모리(MemTable)부터 최적화를 시도함

3. PGM-index: Fully Dynamic Compressed Learned Indexes (VLDB 2020)

저자: G. Ferragina et al. (University of Pisa)

내용 : 데이터를 직선(y=ax+b)으로 표현하되, 지정한 오차(E)를 절대 넘지 않도록 최적의 구간을 자동으로 나눔
→ 본 실험의 비선형적 데이터 처리의 결함을 해결하기 위해 참고함.

4. XIndex: A Scalable Learned Index for Multicore Data Stores (PPoPP 2020)

저자: H. Tang et al. (Shanghai Jiao Tong University)

내용 : 여러 쓰레드가 동시에 데이터를 읽고 쓸 때 Learned Index가 어떻게 작동해야 하는가를 다룸 (멀티코어 CPU에 집중)
→ 본 실험과 가장 유사한 구조(Main + Delta)를 다루는 논문. (다만 본 실험은 MemTable의 기본 구조에 집중)

5. Can Learned Indexes be Built Efficiently? A Deep Dive into Sampling Trade-offs (SIGMOD 2024)

저자: Minguk Choi, Seehwan Yoo, Jongmoo Choi (Dankook University)

내용 : Learned Index의 가장 큰 문제점인 느린 빌드 시간(Build Time)을 해결하기 위해 샘플링(Sampling) 기법을 도입. 전체 데이터를 학습하지 않고 일부만 샘플링하여 학습해도, 오차 범위(Error Bound)를 보장하며 빌드 시간을 획기적으로 단축할 수 있음을 증명함.
→ 본 연구와의 연계: 향후 과제인 '동적 재학습(Dynamic Retraining)' 구현 시, 이 논문의 샘플링 기법을 적용하여 Delta Index 병합 비용을 최소화할 계획임.

Thank You For Listening!