

2026.01.28

---

# SkipList & HashTable

강호현, 이재열, 김민구, 리저우웨원, 최규빈

# 목차

- 1 개요
- 2 데이터 구조
- 3 조회
- 4 실험
- 5 결론
- 6 출처

# 01

## 개요

### 학습 목표

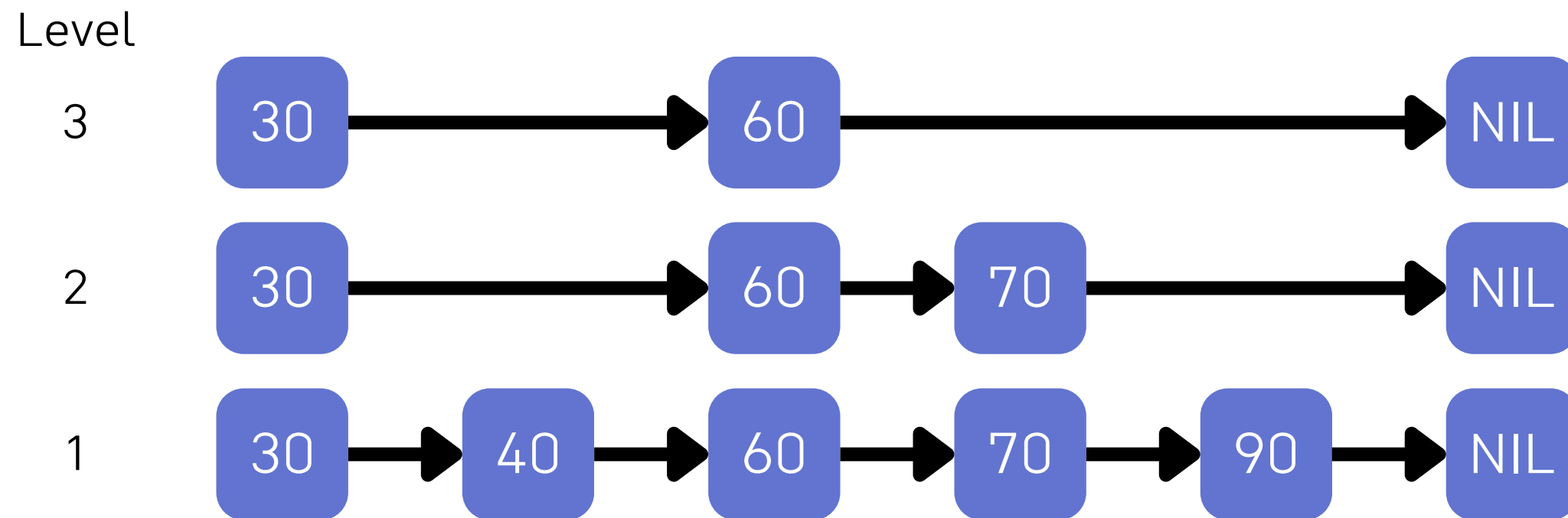
- RocksDB의 MemTable에서 사용하는 데이터 구조에 대한 이해
- 각 데이터 구조의 특징 및 성능 분석

### MemTable

- RocksDB에서 데이터가 디스크(SST 파일)로 저장되기 전 거치는 In-Memory 버퍼
- 사용하는 데이터 구조(Index) 따라 처리량(Throughput)과 지연 시간(Latency)이 결정
- 기본 구현은 skiplist 기반. 그 외에도 HashLinkedList, HashSkipList 또는 Vector로 구현 가능

# 02

## 데이터 구조 - SkipList



### 특징

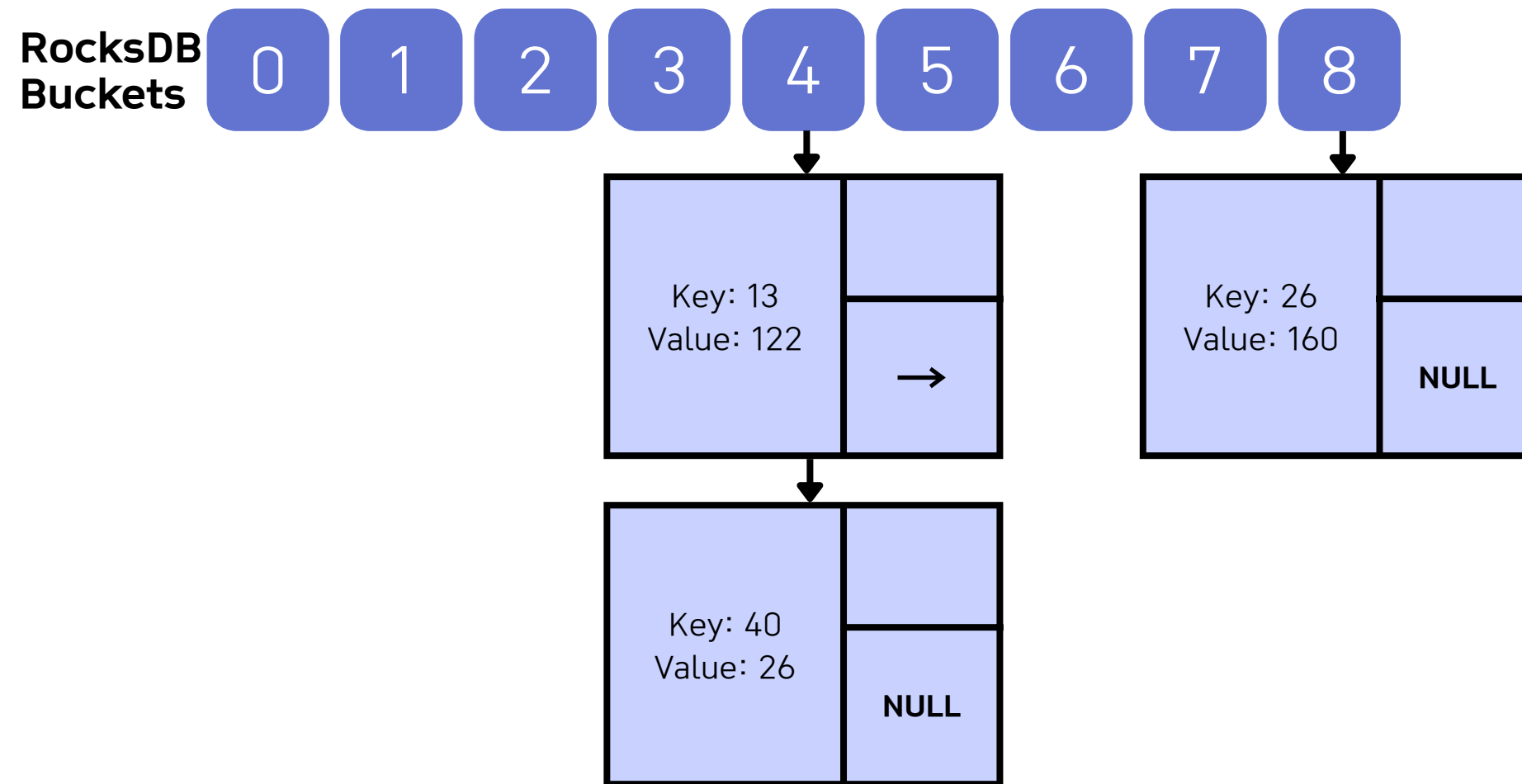
- 데이터가 삽입될 때마다 확률적으로 층(Level)을 쌓아 정렬된 상태를 유지
- 인덱스 유형: 이진 탐색(binary search)

### 장점

- 읽기와 쓰기, 임의 접근, 순차 스캔 모두에서 전반적으로 우수한 성능 제공
- Concurrent Insert와 Insert with Hint 기능 제공

# 02

## HashTable

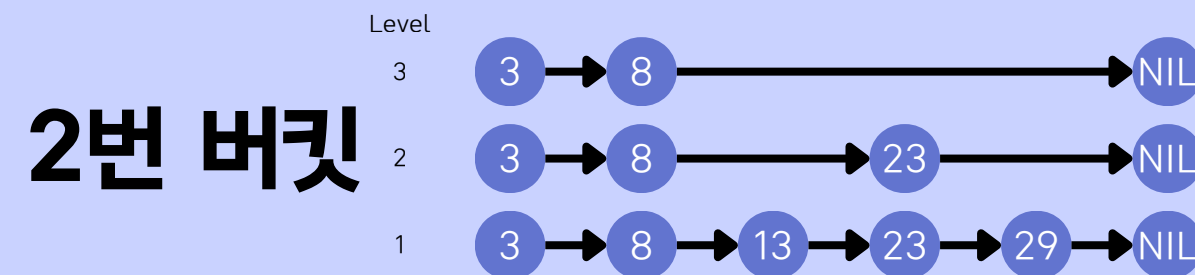
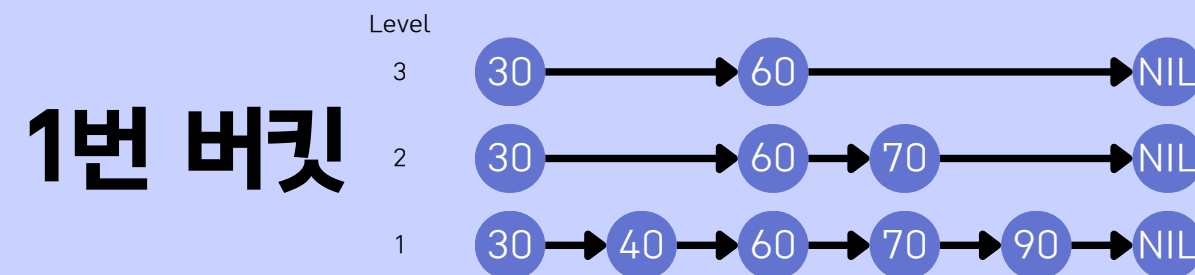


### 특징

- 해시 함수를 통해 Key를 배열의 인덱스에 매핑
- 해시 충돌(Hash Collision) → RocksDB에서는 Separate Chaining(체이닝 기법)을 사용
- 장점: 점 조회 속도 빠름
- 단점: 범위 조회 속도 느림

# 02

## 데이터 구조 - HashSkipList



### 특징

- 해시 버킷 내부에서 데이터를 스킵 리스트(SkipList)로 정렬하여 유지
- 인덱스 유형: 해시 + 이진탐색(hash + binary search)

### 장점

- 해시의 빠른 접근 속도와 SkipList의 정렬 기능을 결합하여 접두사(Prefix) 기반 조회 높은 효율

# 02

## 데이터 구조 - HashLinkedList

1번 버킷

30 → 40 → 60 → 70 → 90 → NIL

2번 버킷

15 → 24 → 31 → 85 → NIL

3번 버킷

1 → 3 → 5 → 7 → 11 → NIL

### 특징

- 해시 버킷 내부에서 데이터를 단방향 연결 리스트(LinkedList)로 정렬하여 유지
- 인덱스 유형: 해시 + 선형탐색(hash + linear search)

### 장점

- 특정 키를 찾는 점 조회(Point Lookup)에서 높은 효율
- 낮은 메모리 오버헤드

# 03

## 조회 - 점 조회(Point Lookup / Get)

### 특징

- 특정 단일 키의 값 조회
- ‘점프 방식’의 접근을 통한 빠른 검색 범위 축소
- 인덱스의 탐색 효율성을 측정.

### 매커니즘

- 현재 노드의 다음 노드 Key가 목표 Key보다 작으면 오른쪽으로 이동(동일 레벨 전진)
- 다음 노드 Key가 목표 Key보다 크거나 NIL이면 아래쪽 레벨(Level N-1)로 이동

# 03

## 조회 - 범위 조회(Range Lookup)

### 특징

- 범위 내 연속된 키의 값 조회
- 인덱스의 정렬 유지 능력을 측정

### 매커니즘

- Seek (위치 탐색): 점 조회 로직으로 범위의 시작 Key (Start Key) 탐색
- Next (순회): 최하위 레벨(Level 1)에 도달 시, 연결 리스트 포인터를 따라 순차적으로 순회하며 종료 Key를 만날 때까지 읽음

# 04

## 실험 - 개요

### 실험 목표

- 목표: 데이터구조 별 데이터양과 조회 방식에 따른 처리량 변화 확인
- 진행 방식: 각 케이스 별 벤치마크 실행 후 처리량(Ops/sec) 기록

### 실험 내 고정된 변인

- 키(Key) 크기: 16 bytes
- 값(Value) 크기: 100 bytes
- 압축 방식 (Compression): None (테스트용)

### 실험 내 조작한 변인

- 데이터 구조(SkipList, HashSkipList, HashLinkedList) – 3
- 데이터양(1M, 5M, 10M) – 3
- 조회 방식(Point Lookup, Range Lookup) – 2
- $3 * 3 * 2 =$  총 18회의 벤치마크

# 04

## 실험 - 개요

### 실험 환경(System Specs)

- CPU: Intel(R) Core(TM) i7-10700K CPU @ 3.80GHz (16 cores) / Intel x86\_64 (16코어)
- RAM: 31GB(사용 가능: 26GB)
- OS: Ubuntu 22.04.5 LTS / Ubuntu 22.04.5 LTS
- RocksDB Version: 10.7 / RocksDB 버전 10.7List

### 환경 A (High Memory)

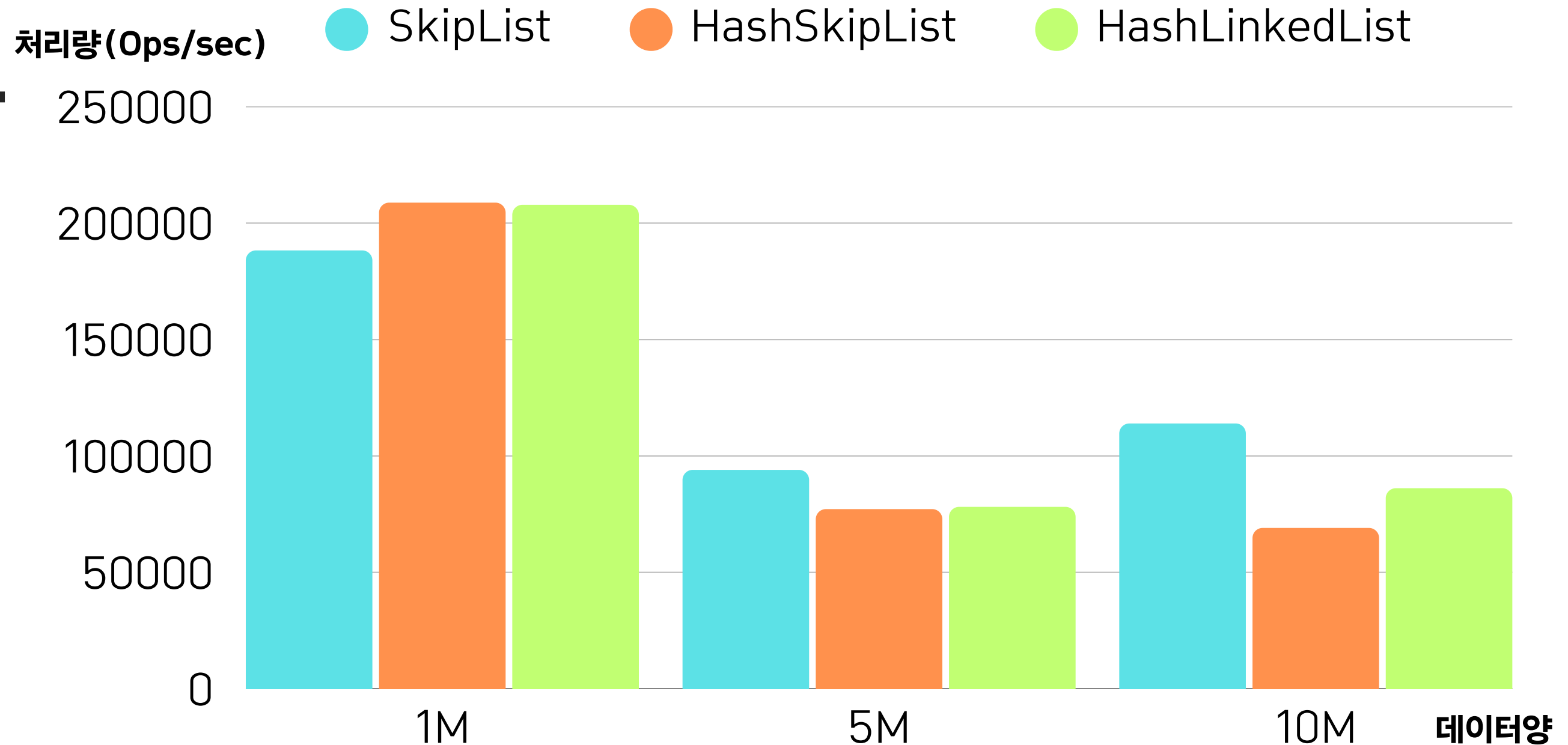
- 31GB RAM
- CPU 연산 능력이 충분하고 디스크 I/O 간섭이 없는 이상적인 환경 모방

### 환경 B (Low Spec)

- 4GB RAM + Limited CPU
- 자원이 제한된 실제 물리적 환경 모방

# 04

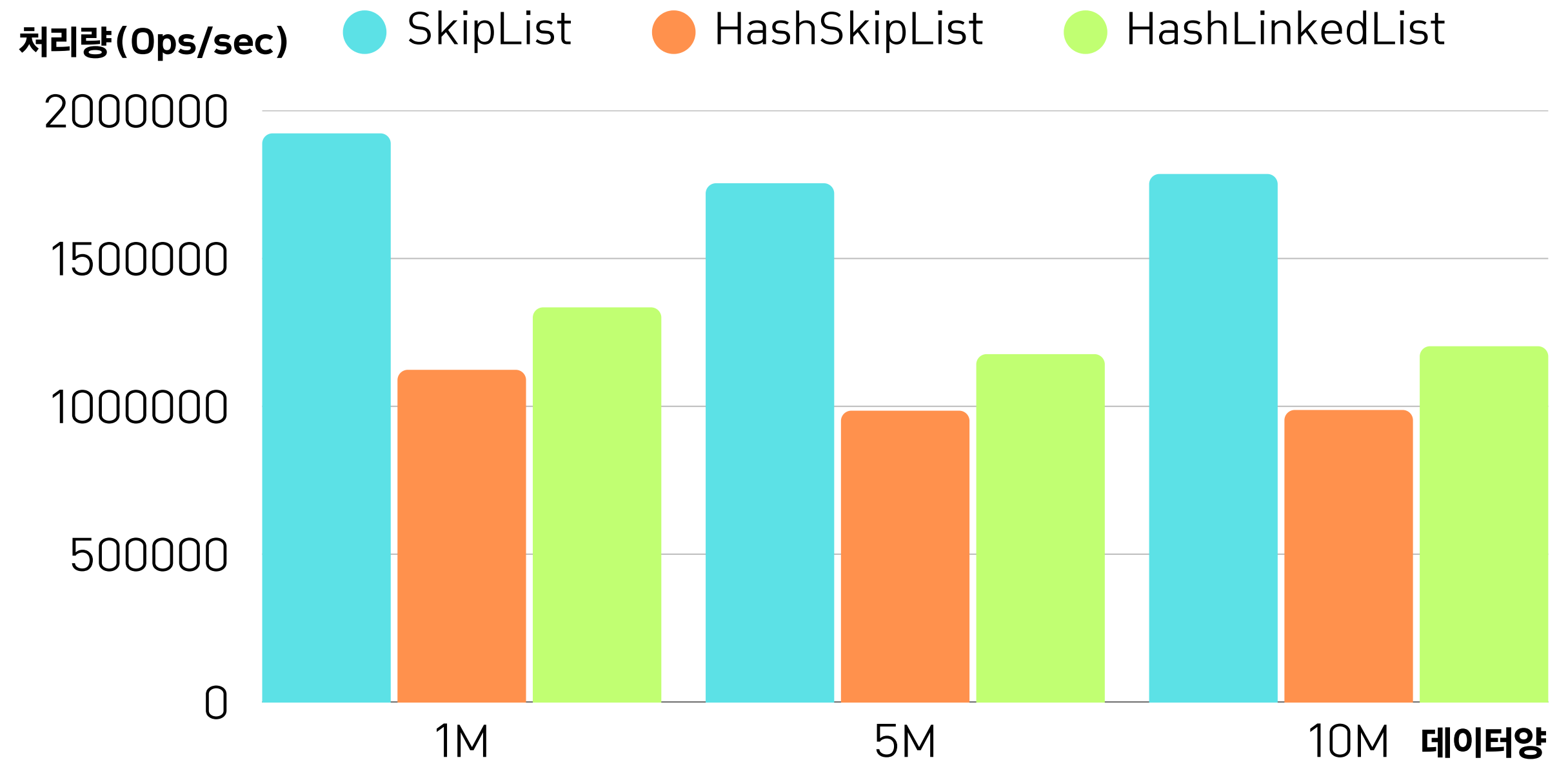
## 실험 - 측정 결과



- 소형 데이터에서 HashSkipList의 소폭 우위
  - 데이터 증가에 따라 Hash 구조의 성능은 뚜렷한 하락세를 보임
  - SkipList는 5M 구간의 캐시 병목구간을 지나 10M 구간에서도 강한 회복력과 안정성을 보임
- 대용량 데이터에서는 해시 충돌(Hash Collision) 심화되는 Hash 구조의 한계

# 04

## 실험 - 측정 결과



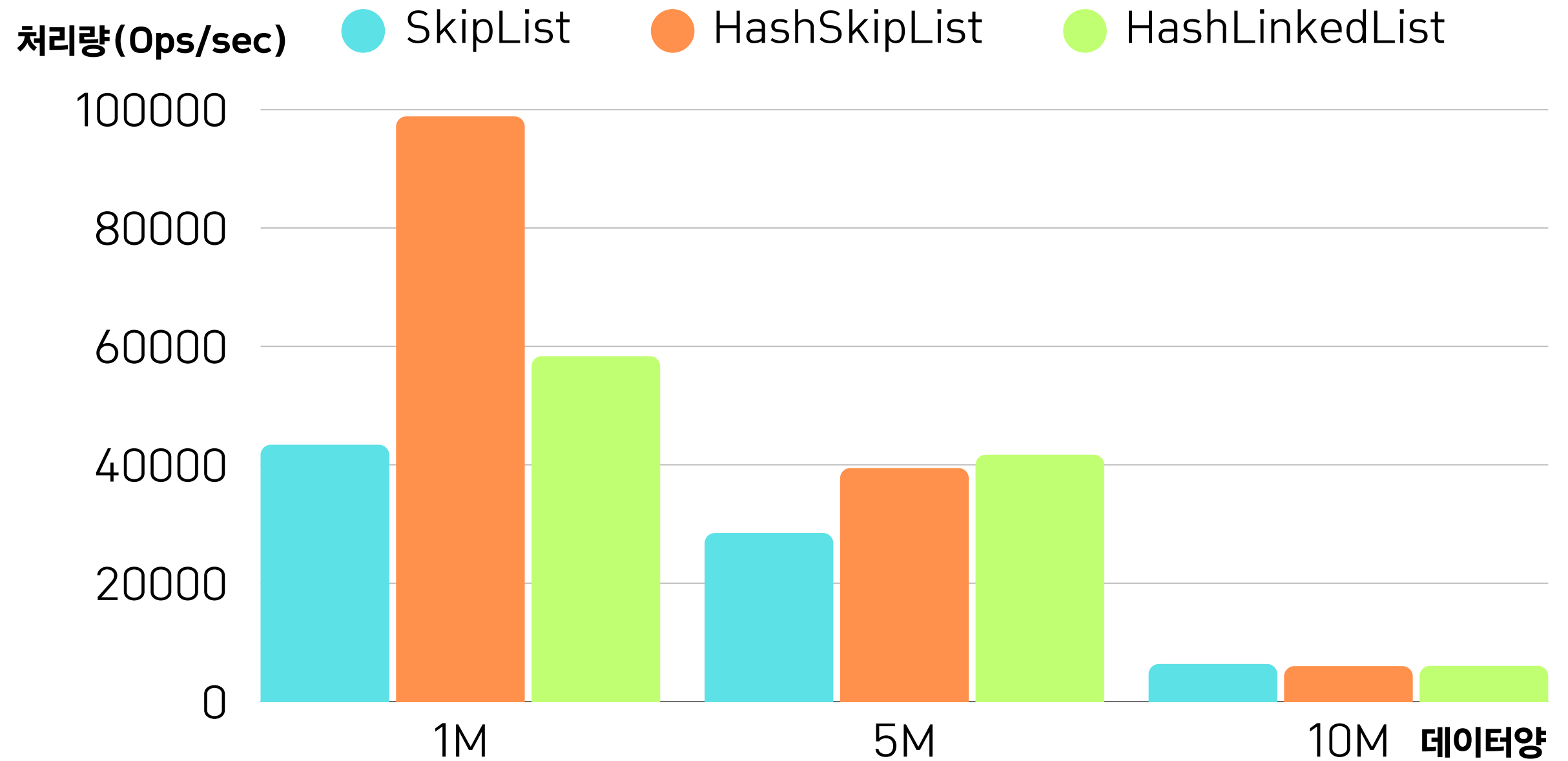
- 전 구간 SkipList의 압도적 우위

→ 구조적 차이 때문

Hash 구조는 본질적으로 비정렬(Unordered) 상태이므로,  
범위 검색 시 비싼 CPU 정렬 오버헤드(Sorting Overhead) 지불  
반면 SkipList는 태생적으로 정렬되어 있어 선형 스캔만 수행

# 04

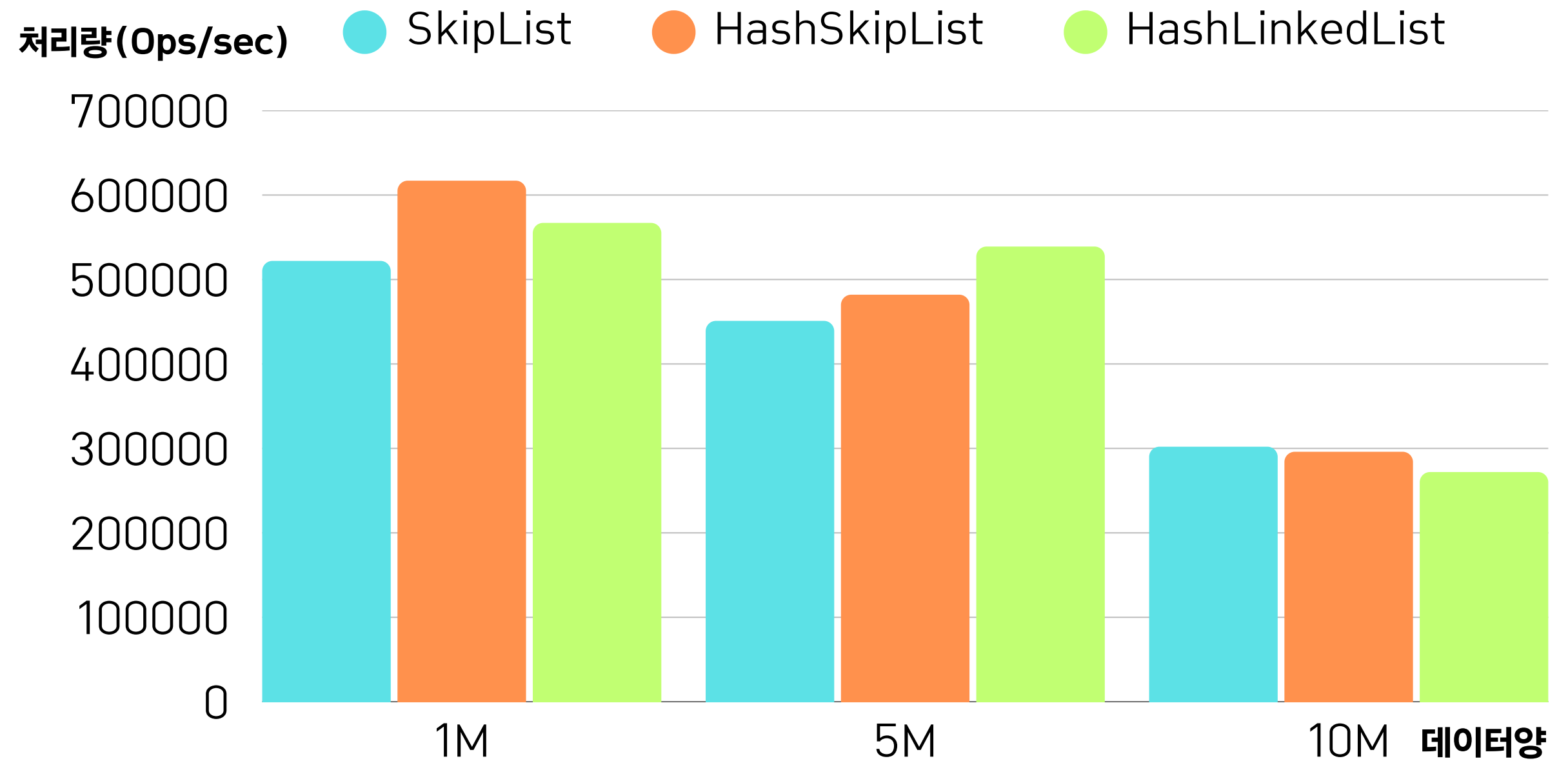
## 실험 - 측정 결과



- 소형 데이터에서 HashSkipList의 성능이 나머지에 비해 뚜렷한 차이(약 2.3배)
- 대형 데이터에서 모든 데이터 구조의 성능이 6,000 OPS 수준으로 거의 동일
- 구간에 따라 HashSkipList - HashLinkedList - SkipList 순 우위

# 04

## 실험 - 측정 결과



- 전 구간 비교적 비등
- 구간에 따라 HashSkipList – HashLinkedList – SkipList 순 우위

# 04

## 실험 - 분석

### 저사양 환경의 데이터 우위 원인

- 1M(메모리 여유): HashSkipList - 빠른 속도
- 5M(메모리 Tight): HashLinkedList - 공간 효율성
- 10M(메모리 넘침): SkipList - 안정성

### 고사양 - 저사양 환경의 결과 차이

- 저사양 머신에서는 메모리 지연 시간(Memory Latency)이 최대 병목 = 노드 크기가 커져서 얻는 손해보다 메모리 접근 횟수를 줄이는게 효율적
- 고사양 머신에서는 CPU의 L3 캐시(L3) 크기 증가 = 메모리 접근 횟수 차이 적음 → 시간 차이 미미

# 05

## 결론

### 모든 경우에서 우월한 단일 데이터 구조 X

- 앞선 요인들로 인해 데이터 크기, 조회 방법에 따라서도 우위가 갈림
- 데이터 규모 및 주로 사용할 조회 방식에 따른 데이터 구조 선택이 필요!

### 가장 안정적인 데이터 구조: SkipList

- 소규모 점 조회에서는 Hash 구조가 우위를 보임
- 그러나, 사양과 관계 없이 많은 데이터를 처리하는 경우 SkipList가 점 조회, 범위 조회 모두 우월함
- 범용 키-값 저장소(RocksDB 등)는 일반적으로 대용량 데이터 처리와 범위 검색(Range Scan)이 주가 되므로, SkipList가 가장 균형 잡힌 최적의 해답

# 06

## 자료 출처

- <https://github.com/facebook/rocksdb/wiki/Memtable>
- **Paper:** Pugh, W. (1990). "Skip Lists: A Probabilistic Alternative to Balanced Trees."
- **Paper:** O'Neil, P., et al. (1996). "The Log-Structured Merge-Tree (LSM-Tree)."

# THANK YOU

---