碩士學位 請求論文

指導敎授 南 範 碩

비휘발성 메모리를 위한 LSM 트리의 점층적 컴팩션을 통한 공간 지역성 확보

成均館大學校 一般大學院

소프트웨어學科

鄭 柱 元

碩士學位 請求論文

指導敎授 南 範 碩

비휘발성 메모리를 위한 LSM 트리의 점층적 컴팩션을 통한 공간 지역성 확보

Progressive Compaction for Enhancing Spatial Locality in LSM Trees for Non-Volatile Memory

成均館大學校 一般大學院

소프트웨어學科

鄭 柱 元

碩士學位 請求論文

指導敎授 南 範 碩

비휘발성 메모리를 위한 LSM 트리의 점층적 컴팩션을 통한 공간 지역성 확보

Progressive Compaction for Enhancing Spatial Locality in LSM Trees for Non-Volatile Memory

이 論文을 工學 碩士學位請求論文으로 提出합니다.

2024 年 4 月 日

成均館大學校 一般大學院

소프트웨어學科

鄭 柱 元

이 論文을 鄭 柱 元의 工學
碩士學位 論文으로 認定함.

2024 年 6 月 日

審査委員長 _____

審査委員 _____

審査委員 _____

목차

그림목차

논문요약

비휘발성 메모리를 위한 LSM 트리의 점층적 컴팩션을 통한 공간 지역성 확보

대부분의 바이트 단위의 접근이 가능한 키-밸류 스토어는 비휘발성 메모리를 위해 설계되었으며, 8바이트 크기의 포인터 업데이트를 통해 키를 삽입, 업데이트 또는 삭제하는 구조 수정 연산 방식을 활용한다. 이러한 제자리 업데이트 방식은 쓰기 증폭 문제를 완화하는 데 도움이 되지만, 과도한 제자리 업데이트는 공간 지역성을 감소시켜 읽기 성능이 저하된다.

본 연구에서는 비휘발성 메모리에 최적화된 SkipBlockList를 제안하며, 이는 바이트 단위의 접근이 가능한 키-밸류 스토어의 스킵리스트 인덱스를 블록 장치에 최적화된 SSTable로 직렬화하는 중간 계층으로 동작한다. SkipBlockList는 단일 노드 내에 여러 키를 관리하며, 스킵리스트와 B-트리의 장점을 결합한다. SkipBlockList를 적용한 키-밸류 스토어인 BlockListDB는 제자리 업데이트 방식을 사용하여 큰 단위의 입출력을 효과적으로 줄이면서 동시에 노드 내 공간 지역성을 키-밸류의 복사를 통해 향상시킨다. 광범위한 성능 연구를 통해, BlockListDB가 최신 LSM 트리를 크게 앞선다는 것을 보여준다.

주제어 : 비휘발성 메모리, LSM 트리, 키-밸류 스토어, 스킵 리스트, 공간 지역성

제1장 서 론

LSM 트리는 MemTable이라는 메모리 내의 인덱스를 사용하여 키-밸류 쌍을 버퍼링한다[13]. 많은 양의 데이터가 버퍼에 축적되어 큰 디스크 대역폭을 사용하면, MemTable은 SSTable이라는 단일 파일로 디스크에 플러시된다. 디스크에 플러시된 SSTable의 수가 증가함에 따라, SSTable들 사이에 겹치는 키 범위가 증가하여 검색 성능이 저하된다. 이러한 SSTable의 키 범위 겹침을 완화하기 위해, LSM 트리는 백그라운드에서 컴팩션이라는 병합 정렬을 수행하여 큰 정렬을 생성한다. 블록 장치의 특성으로 인해, SSTable의 병합 정렬 시 일반적으로 키-밸류를 복사하는 방식이 사용되며, 이로 인해 많은 키-밸류 쌍이 디스크에 다시 쓰여진다. 결과적으로, 이 쓰기 증폭 문제는 컴팩션 성능을 저하시키고, 병합 정렬 성능의 저하로 인해 겹치는 SSTable의 수가 증가하여 검색 성능이 더욱 악화된다. 이러한 문제를 완화하기 위해, 쓰기 지연 메커니즘이 도입되어, 클라이언트의 삽입을 일시적으로 중지한다.

키-밸류 스토어에서 쓰기 증폭을 줄이기 위한 여러 연구가 수행되었다[1, 11]. 특히, 많은 연구들[1, 11, 12]이 바이트 단위의 접근이 가능한 비휘발성 메모리에서 캐시라인이나 XPLine과 같은 작은 입출력 단위를 사용하여 병합 정렬을 수행하는 것을 탐구[8]했다.

비휘발성 메모리의 작은 입출력 단위를 활용하기 위한 여러 방법이 제안되었다. ListDB[12]는 append-only 로그 항목을 점진적으로 스킵리스트 노드로 변환한다. 또한, ListDB는 'Zipper compaction'라는 제자리 병합 정렬 알고리즘을 통해 스킵리스트가 반복적으로 병합 정렬될 때도 키-밸류 쌍이 다시 작성되는 것을 방지함으로써 쓰기 증폭을 최소화한다. 하지만 제자리 병합 정렬 알고리즘은 쓰기 증폭을 효과적으로 줄이지만, 키 근접성에 관계없이 키-밸류 삽입 순서에 의해

비휘발성 메모리상의 위치가 결정되므로 낮은 공간 지역성을 겪는다. 특히, 인덱스가 개별 키-밸류 쌍을 가리키는 경우나 개별 키-밸류 쌍이 인덱스 내에서 독립적인 노드로 관리될 경우 메타데이터 관리 측면에서 단점이 존재한다. 특히 키-밸류 쌍의 크기가 작을 때 메타데이터의 크기가 데이터 자체보다 커지는 문제를 갖는다.

인텔이 Optane DCPMM의 생산을 중단하며, 새롭게 등장하는 차세대 비휘발성 메모리는 작은 용량의 NVDIMM 또는 내부 버퍼로 소량의 DRAM을 사용하는 MS-SSD일 것으로 예상된다. 따라서 비휘발성 메모리를 DRAM과 블록 장치 사이에 위치한 저장 계층으로 간주하는 것이 합리적이다. 이는 SSTable로의 쉬운 변환을 가능하게 하는 비휘발성 메모리 계층을 필요로 한다.

본 연구는 바이트 단위의 접근이 가능한 스킵리스트와 블록 장치에 최적화된 SSTable 사이의 중간 인덱스 계층의 역할을 하는 SkipBlockList라는 새로운 데이터 구조를 제안한다. SkipBlockList는 락프리 스킵리스트와 동일한 구조 수정 연산 방식을 사용함으로써 락프리 스킵리스트의 장점을 물려받는다. 스킵리스트는 본질적으로 공간 지역성이 낮고 각 키-밸류 쌍마다 하나 이상의 포인터가 필요하여 상당한 양의 메타데이터가 필요하지만, SkipBlockList는 단일 노드 내 배열에 여러 키-밸류 쌍을 저장함으로써 공간 지역성을 향상시키고 연결된 전체 노드의 수를 줄인다

이러한 SkipBlockLists를 바탕으로, 본 연구는 DRAM과 블록 장치 저장소 사이의 비휘발성 메모리(NVMM)를 위한 새로운 계층을 도입한다. LSM 트리는 각각 DRAM과 블록 장치에 최적화된 데이터 구조를 활용하지만, 메모리에서 저장소로 데이터를 이동할 때 데이터의 직렬화를 위한 상당한 비용이 발생하며 비휘발성 메모리의 지연시간이 낮은 이점을 제한하게 된다. 노드당 하나의 키-밸류 쌍이 저장되는 스킵리스트는 DRAM에서 사용되며, 블록 장치에서는 수십 MB 크기의 배열이 활용된다. 비휘발성 메모리 계층을 위해, 중간 형태의 SkipBlockLists가 사용되어 세밀한 단위의 스킵리스트를 굵은 단위의 큰 배열로

쉽게 변환할 수 있도록 돕는다.

제2장 연구배경

2-1. LSM 트리와 쓰기 증폭 문제

LSM 트리는 블록 장치에 대한 쓰기를 위해 최적화된 멀티쓰레드 인덱싱 구조이다. 이 구조는 MemTable이라고 알려진 작은 메모리 버퍼를 사용하여 키-밸류 쌍을 축적하고 저장소에 플러시한다. MemTable은 스킵리스트와 같은 메모리 내 인덱스를 사용하여 키-밸류 쌍을 정렬하며, 이를 통해 포인터와 같은 최소한의 메타데이터를 업데이트함으로써 키-밸류 쌍을 정렬할 수 있다. 메모리 버퍼가 한계 용량에 도달하면, 정렬된 키-밸류 쌍은 정렬된 시퀀스, 즉 배열로 변환된다. 결과적으로 생성된 정렬된 시퀀스는 순차적으로 블록 장치 저장소에 쓰여진다. 추가적으로, 각 블록의 키 범위에 대한 인덱스 정보가 파일에 추가되며, 이것을 SSTable이라고 한다.

동시에, 백그라운드 쓰레드는 저장소에 저장된 SSTable들을 병합 정렬하여 점차 하나의 커다란 정렬된 시퀀스를 형성한다. 이러한 병합 정렬 과정은 컴팩션이라고 부르며, SSTable들 사이의 키 범위 겹침을 제거하고 하나의 정렬된 시퀀스를 유지하여 검색 성능을 향상시킨다. RocksDB와 HBase와 같은 키-밸류 스토어는 블록 장치의 대역폭을 효율적으로 사용하기 위해 SSTable의 최대 크기를 제한한다. 기본적으로 RocksDB는 SSTable의 크기를 64MB로 설정한다. 반면, LevelDB는 SSTable 크기를 4MB로 설정하고 있다. 이 컴팩션 과정은 키-밸류 데이터가 반복적으로 새로운 SSTable로 복사되어야 한다. SSTable의 크기가 커짐에 따라 겹치는 키 범위가 증가하고 더 많은 키-밸류 쌍이 병합 정렬된다. 블록 장치는 각각의 키-밸류 쌍에 대한 세밀한 입출력 수행할 수 없기 때문에, 컴팩션 과정은 키 범위가 겹치는 SSTable을 읽고 쓰는 과정을 반복하며, 이는

상당한 쓰기 증폭 문제를 초래한다[11].

2-2. 비휘발성 메모리의 키-밸류 스토어

인텔의 Optane DCPMM은 비휘발성 메인 메모리 장치로, DRAM의 바이트 단위 접근성과 저장 장치의 지속성을 모두 포함한다. Optane DCPMM의 이러한 특성은 그 기능을 활용하기 위해 최적화된 다양한 데이터 구조와 키-밸류 스토어의 연구를 촉진하였다. 특히 4KB 블록에서 업데이트를 수행하는 널리 알려진 B-트리와 해시 테이블과 같은 인덱싱 구조들은 8 바이트, 캐시라인 또는 XPLine의 더 세밀한 단위의 업데이트를 수행하도록 재설계 되었으며, 이는 Optane DCPMM의 특성을 더욱 잘 활용할 수 있게 되었다.[2, 6, 7, 14, 15, 16] LSM 트리 역시 비휘발성 메모리의 특성을 활용하도록 재설계 되었다.[9, 10, 12]

NoveLSM은 비휘발성 메모리에 영구적인 MemTable을 도입하여 집중적인 쓰기 작업이 들어올 때 이를 버퍼링하고 키-밸류를 SSTable 형태로 직렬화하는 오버헤드를 줄인다. 큰 영구적 MemTable을 사용함으로써, NoveLSM은 플러시 작업의 빈도를 감소시킨다. SLM-DB[9]는 여러 레벨로 이루어진 SSTable 대신 바이트 단위의 접근이 가능한 단일 레벨의 영구적 인덱스를 사용하도록 제안한다. 블록 장치와 달리, NVMM의 우수한 쓰기 성능은 여러 레벨의 필요성을 제거하고 바이트 단위의 접근이 가능한 영구적인 인덱스 사용을 가능하게 한다. 이러한 접근 방식은 이후 진행된 연구인 Prism[5]에서 따르고 있다.

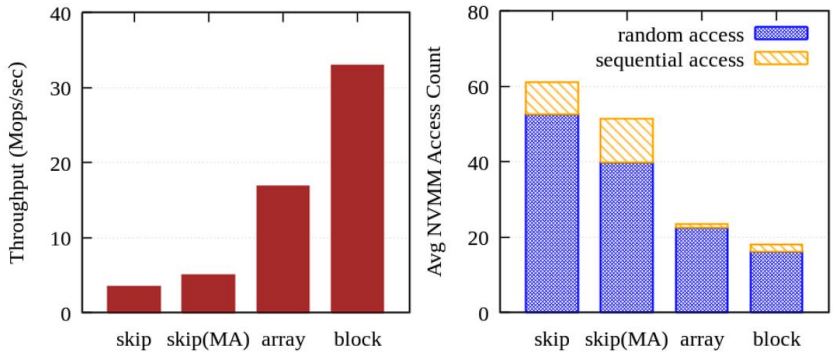
비휘발성 메모리의 쓰기 성능은 블록 장치보다 우수하지만 여전히 DRAM에는 미치지 못한다. 따라서 많은 키-밸류 스토어[4, 9, 10]들이 DRAM을 적극적으로 활용하는 디자인을 선택한다. 그러나 DRAM은 휘발성이므로, 시스템 중단 시의 복구를 보장하기 위해 MemTable에 키-밸류 쌍을 저장할 때 로깅이 필요하다. 또한 메모리 내 인덱스에 쓰여진 키-밸류 쌍은 이후 영구적인 저장소로 플러시된다. 동일한 데이터가 두 번 쓰여지는 사실을 고려할 때, ListDB[12]는 비휘발성 메모리에 저장된 로그 엔트리를 인덱스 노드로 변환하는 통합 로깅

기법을 제안한다. SLM-DB와 달리, ListDB는 비휘발성 메모리에서의 플러싱을 가속화하기 위해 두개의 레벨로 이루어진 구조를 사용한다. 즉, 레벨 1 인덱스는 비휘발성 메모리에서 관리되는 최종 인덱스이며, 레벨 0 인덱스는 DRAM에서 플러시 되었지만 아직 레벨 1 인덱스로 병합 정렬이 일어나지 않은 임시 인덱스이다.

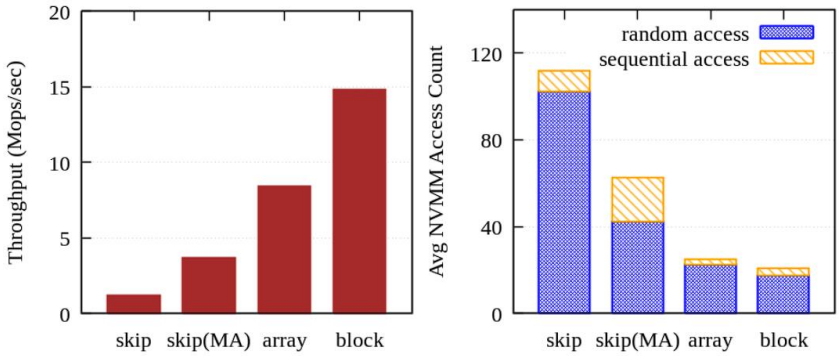
2-3. 제자리 업데이트 방식과 지역성의 관계

바이트 단위의 접근이 가능한 인덱싱 구조는 제자리 업데이트 방식을 활용하여 8바이트 포인터 업데이트를 통해 키를 삽입, 업데이트 또는 삭제한다. 이 제자리 업데이트 방식은 쓰기 증폭 문제를 완화하는 데 도움을 주지만, 과도한 제자리 업데이트는 공간 지역성을 감소시켜 읽기 성능을 저하시킨다. 즉, 키-밸류 쌍이 저장되는 순서가 키의 순서와 일치하지 않는다. 예를 들어, 스킵리스트에서 키-밸류 쌍의 메모리(또는 비휘발성 메모리) 위치는 메모리 할당 순서(즉, 삽입 순서)에 의해 결정되며, 키 값과는 무관하다.

스킵리스트의 크기가 증가함에 따라, 스킵리스트 노드를 방문할 때마다 캐시라인이나 XPLine을 읽어야 한다. 스킵리스트 노드를 이동하는 과정은 대부분 랜덤 읽기를 일으키기 때문에 캐시 미스가 자주 발생한다. 스킵리스트 노드 간 이동에서 발생하는 랜덤 읽기와 낮은 메모리 접근 지역성은 스킵리스트의 낮은 검색 성능에 큰 영향을 준다. 스킵리스트의 낮은 메모리 지역성 문제는 범위 쿼리를 처리할 때 특히 심화된다. 이는 범위 쿼리를 처리하기 위해서 스킵리스트 하단 레벨의 포인터를 따라 키-밸류 쌍을 읽을 때마다 랜덤 메모리 접근이 발생하기 때문이다. 이 문제는 특히 MemTable을 SSTable로 변환할 때 두드러진다.



(a) Point Query



(b) Scan Query

그림 2-1 비휘발성 메모리 인덱스 구조에 따른 탐색 성능

위의 실험은 비휘발성 메모리에서 전통적인 스킵리스트 구조의 사용과 공간 지역성의 관계성을 보여준다. 실험의 비교군은 전통적인 스킵리스트 구조, 정렬된 배열 구조, 마지막으로 본 논문에서 제시하는 SkipBlockList 구조를 사용하였다. 이 중 'SkipList(MA)'는 전통적인 스킵리스트 구조를 사용하지만, 키를 순차적으로 사용하여 일정량의 공간지역성을 얻은 상태를 나타낸다. 각각 Uniform한 분포의 무작위 키를 10억개씩 로드 한 뒤, 1억개의 부분 및 범위 탐색 쿼리를 수행하였다.

부분 탐색의 결과, 전통적인 스킵리스트 구조는 공간지역성을 충분히 확보한 다른 자료구조와 다르게 다량의 비휘발성 메모리 임의 접근을 발생시킨다. 이것은

전통적인 스킵리스트의 구조 상 노드의 모든 순회 과정이 공간지역성이 낮은 포인터 추적을 통해 이루어지기 때문이다. 비휘발성 메모리 임의 접근의 증가에 따라 부분 탐색 성능이 저하된다. 하지만 순차적인 삽입을 통해 일정량의 공간 지역성을 확보했을 때 비휘발성 메모리의 접근 횟수가 감소하고 부분 탐색 성능이 증가한다. 이것은 최하단 높이를 탐색할 때 한번의 비휘발성 메모리 접근으로 여러 노드를 읽을 수 있기 때문이다. 이러한 차이는 범위 탐색 쿼리를 수행할 때 더욱 두드러진다. 공간지역성을 확보한 자료구조일수록 비휘발성 메모리 접근 횟수가 크게 감소하게 된다. 이를 통해 전통적인 스킵리스트 구조의 낮은 공간지역성이 탐색 성능을 저하시키는 것을 확인할 수 있다.

제3장 BlockListDB

본 연구에서 제안하는 디자인인 BlockListDB는 비휘발성 메모리에 최적화된 키-밸류 스토어이다. BlockListDB는 중간 계층의 인덱싱 구조인 SkipBlockLists를 사용하여 점차적으로 스킵리스트를 SSTable로 변환한다. 이러한 중간 계층을 도입함으로써, BlockListDB는 구조 수정 연산을 통해 쓰기 성능을 최적화하고 캐시라인에 친화적인 방식으로 정확성과 원자성을 보장하며 메모리의 공간 지역성을 향상시킨다. 아래 내용은 BlockListDB의 주요 디자인을 상세히 설명하고 있다.

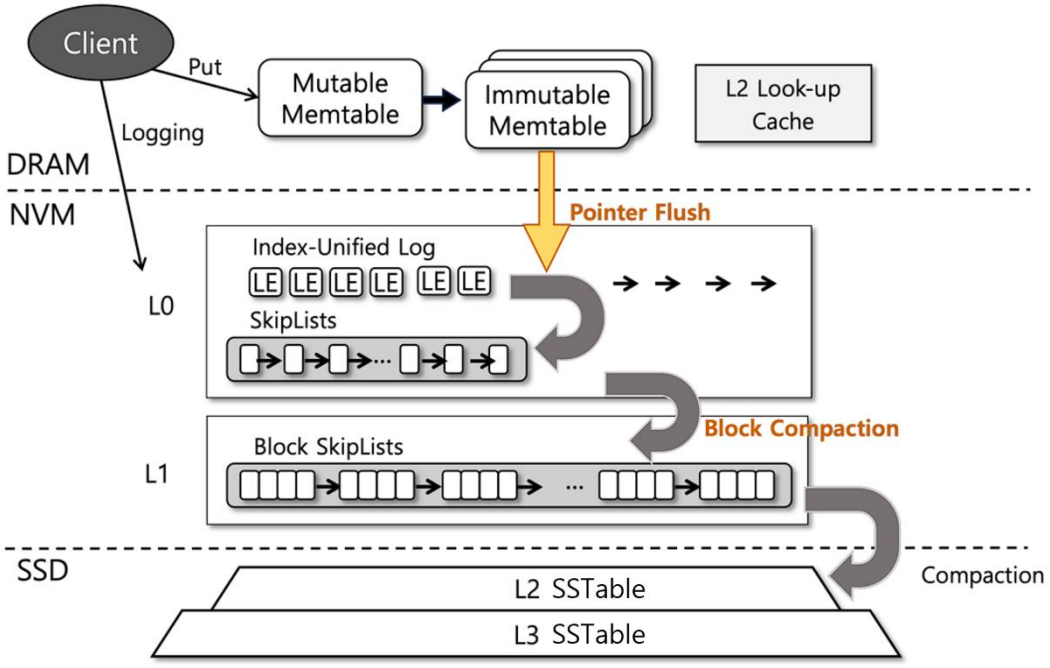


그림 3-1 BlockListDB의 아키텍쳐

그림은 BlockListDB의 아키텍쳐를 보여준다. 전통적인 LSM 트리 기반 키-밸류 스토어와 마찬가지로, BlockListDB는 락프리 스킵리스트를 사용하여 MemTable과 Immutable MemTable을 구성한다. MemTable을 비휘발성 메모리로 플러시할 때, BlockListDB는 Index-Unified Logging (IUL) 기법[12]을 사용한다. 즉, 키-밸류 쌍이 가변 MemTable에 삽입될 때 비휘발성 메모리의 Index-Unified Log의 형태로 로깅되며, 추후 스킵리스트 노드로 변환된다. 나중에 MemTable이 Immutable MemTable이 되어 비휘발성 메모리로 플러시될 때, 스킵리스트의 포인터만 비휘발성 메모리에 작성한다. 이 작업을 통해 IUL을 스킵리스트 노드로 변환한다. IUL은 비휘발성 메모리의 바이트 단위의 접근성을 활용하여 동일한 키-밸류 쌍의 중복 쓰기를 방지하고 쓰기 증폭을 크게 개선하는 역할을 한다.

Immutable MemTable이 비휘발성 메모리로 플러시될 때, 어떤 PMTable과도 병합 정렬되지 않는다. 이는 PMTable 간의 병합 정렬 작업으로 인해 플러시 성능이 영향을 받지 않도록 보장한다. 또한, IUL은 eADR을 지원하지 않는 환경에서도 포인터를 복사할 때 clflush를 호출하지 않는다. 이는 키의 순서를 지정하는 포인터가 복구 가능한 메타데이터이기 때문이다. IUL에서 스킵리스트 노드로 변환된 키-밸류 쌍의 인덱싱 구조는 L0 PMTable라고 부르며, 이것은 BlockListDB에서 비휘발성 메모리의 첫 번째 레벨을 나타낸다. 이 L0 PMTable은 일시적으로 키-밸류 쌍을 저장하는 영구적인 버퍼의 역할을 하며, PMTable 간 키 범위 겹침을 최소화하기 위해 백그라운드 쓰레드에 의해 다음 레벨로 병합 정렬된다.

L0 PMTable은 키의 개수는 적지만 넓은 키 범위를 가지고 있다. 낮은 공간 지역성을 가진 소수의 키를 많은 키를 포함하고 있는 다음 레벨의 PMTable로 컴팩션 하는 것은 많은 횟수의 랜덤 쓰기를 요구한다.

L0 PMTable과 IUL의 목적은 DRAM 내의 MemTable을 빠르게 비워서 쓰기 작업이 멈추지 않도록 하는 것이다. 반면 다음 레벨인 L1 PMTable의 주된 목적은

정렬된 시퀀스 간 키 범위 겹침을 제거하여 검색 성능을 향상시키는 것이다. L1의 하위 레벨은 SSD를 이용한 확장이 가능한 레벨로, 전통적인 LSM 트리 구조의 SSTable 형태로 키-밸류를 저장한다. L1에서 L2로 컴팩션이 일어나는 시점은, 비휘발성 메모리의 공간이 가득 차거나 L1 PMTable의 크기가 최대 크기에 도달했을 때이다.

그러나 ListDB에서 스킵리스트를 사용할 때, 쓰기 증폭 문제가 발생한다. 이는 스킵리스트 노드의 삽입 시 서로 다른 캐시라인에 존재하는 이전 노드들의 포인터를 업데이트해야 하기 때문이다. 이 문제를 해결하기 위해, 본 연구는 스킵리스트의 변형인 SkipBlockList를 제안한다.

제4장 SkipBlockList

4-1. 노드 레이아웃

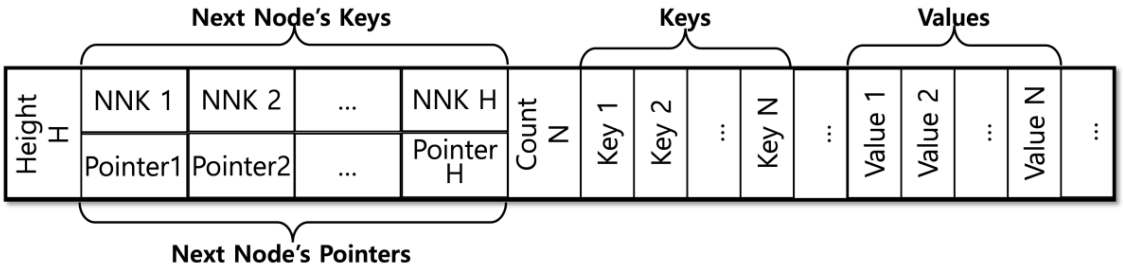


그림 4-1 SkipBlockList 노드 레이아웃

그림은 SkipBlockList 노드의 물리적인 레이아웃을 보여준다. 메모리 영역 상 공간 지역성을 높이기 위해 SkipBlockList는 그림에서 보여지듯 단일 노드에 여러 키-밸류 쌍을 저장한다. 단일 노드에 여러 키-밸류 쌍을 저장하는 것 외에도 각 높이에서 다음 노드의 가장 작은 키를 저장하는데, 이를 NNK(Next Node's Key)라고 한다. NNK는 B+트리의 내부 노드의 키와 유사하며, 다음 노드를 방문할 필요가 있는지를 결정한다.

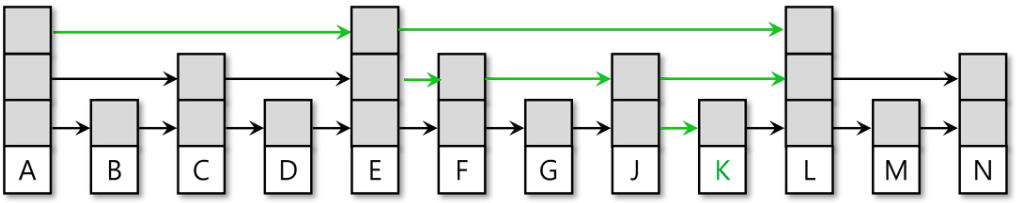
전통적인 스킵리스트 구조는 하나의 키-밸류 쌍에 대해 여러 포인터가 필요하며, 이로 인해 노드의 높이가 h일 때 메타데이터 크기는 8 X h 가 된다. 따라서 키-밸류 쌍의 크기가 작고 노드의 높이가 클 때 일반적으로 메타데이터의 크기가 데이터 크기보다 커지게 된다.

반면 SkipBlockList는 하나의 노드에 훨씬 더 많은 키-밸류 쌍을 저장하고 각 노드에는 소수의 포인터만 있다. 또한 SkipBlockList는 노드의 높이 h에 비해 하나의 노드 안에 훨씬 더 많은 키-밸류 쌍(N)을 저장한다. 따라서 h 개의 포인터와 h 개의 NNK를 가진 노드에 N 개의 키-밸류 쌍을 저장할 때,

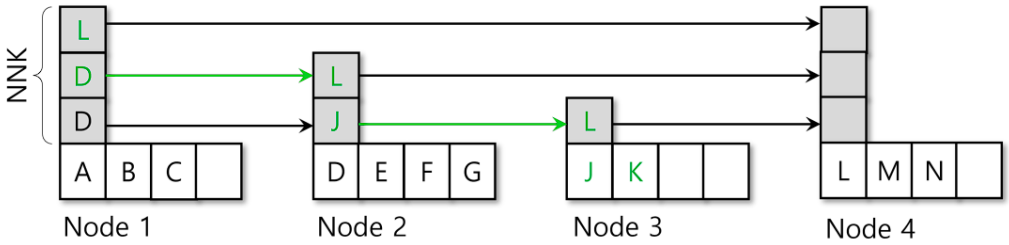
메타데이터의 비율은 (h * (8 + 키의 크기) /N * (키의 크기 + 값 크기))가 된다. N이 h보다 훨씬 클 경우, NNK로 인한 메타데이터 크기 증가는 무시할 수 있는 수준으로 줄일 수 있다. 또한 전통적인 스킵리스트에서 각 키-밸류 쌍이 h 개의 포인터를 가지는 것과 달리, SkipBlockList는 N 개의 키-밸류 쌍이 h 개의 포인터를 공유함으로써 총 메타데이터 크기를 크게 줄인다. 또한 NNK에 필요한 공간은 키의 접두부를 사용하거나 델타 인코딩을 사용하여 더욱 줄일 수 있다. Pugh가 제안한 1/4의 분기 계수를 사용하는 경우를 고려하면 전통적인 스킵리스트 구조에서 8 바이트의 키와 8 바이트의 밸류를 저장하는 경우, 포인터는 전체 인덱스 크기의 약 57%를 차지한다. 반면 SkipBlockList가 노드 당 32개의 키-밸류 쌍을 저장하는 경우, 메타데이터(포인터와 NNK)가 차지하는 비율은 4%로 감소한다.

SkipBlockList는 B+트리와 유사하게 다음 노드의 키 범위를 관리한다. 그러나 여전히 스킵리스트의 확률적 구조를 유지하며 연결 리스트를 기반으로 하고 있어, 락프리 삽입/삭제 작업이 가능하다는 장점이 있다. 뿐만 아니라, 스킵리스트 구조를 유지하기 때문에 NUMA 지역성에서 이점을 얻기 위해 ListDB의 Braided Skiplist 기법을 활용할 수 있다. 게다가 스킵리스트 구조는 병렬적으로 수행되고 있는 읽기 쿼리를 차단하지 않으면서 삽입을 가능하게 한다.[12]

4-2. SkipBlockList의 탐색



(a) SkipLists



(b) SkipBlockLists

그림 4-2 SkipList와 SkipBlockList의 탐색 과정

SkipBlockList는 다음 노드의 키(NNK)가 다음 노드를 방문하지 않고도 탐색을 가능하게 함으로써 역추적을 피할 수 있게 한다. 예를 들어, 그림의 (b)에서 타겟 키 I를 검색할 때, 현재 노드인 노드 1에 저장된 최상단 높이의 NNK(J)를 비교하고 노드 4를 방문하지 않고 다음 높이로 내려간다. 다음 높이의 NNK(D)가 목표 키 I보다 작기 때문에 노드 2로 이동한다. 같은 방식으로, 노드 2에서는 로컬 NNK(J)를 목표 키와 비교하여 노드 4를 방문하지 않고 최하단 높이로 내려가 노드 3으로 이동한다. 노드 3에서는 다시 최하단 높이의 NNK(J)와 비교하여, 그것이 타겟 키보다 크기 때문에, 최종적으로 현재 노드의 키-밸류 쌍 배열에서 타겟 키를 검색한다.

전통적인 스킵리스트에서 탐색 알고리즘은 각 높이의 역추적을 필요로 한다. 탐색은 첫번째 노드의 최상단 높이에서 시작된다. 그리고 현재 높이의 다음 노드로 이동하면서 타겟 키보다 크거나 같은 키를 가진 노드를 찾거나, 해당 높이의

마지막 노드에 도달할 때까지 계속 이동한다. 타겟 키보다 큰 키를 가진 노드를 찾으면, 이전 노드로 돌아가 다음 높이로 내려가고 이 과정을 반복한다. 이러한 역추적은 대량의 랜덤 메모리 접근을 발생시켜 스킵리스트의 탐색 성능을 저하시킨다.

그림의 (a)에서 초록 화살표로 표시된 전통적인 스킵리스트의 탐색 경로와 비교할 때, 노드의 접근 횟수는 두 가지 이유에 의해 크게 감소한다. 첫째, 노드의 총 수가 노드당 평균 키 수인 N에 의해 감소된다. 둘째, NNK는 불필요한 노드 방문을 효과적으로 방지한다.

4-3. SkipBlockList의 키 삽입

SkipBlockList의 키 삽입 알고리즘은 B+트리 알고리즘과 매우 유사하다. 키-밸류 쌍을 삽입할 때 노드에 여유 공간이 있다면 해당 위치에 삽입되며, B+트리와 유사하게 노드 내에서 키의 순서를 유지한다.

바이트 단위의 데이터 접근이 가능한 비휘발성 메모리의 경우, SkipBlockList는 wB+트리[3]의 메타데이터 섀도잉 방법을 활용할 수 있다. 이러한 방식에서는 키-밸류 쌍이 임의의 여유 공간에 저장되고, 그 오프셋이 키의 순서에 따라 정렬된다. 다른 방법으로는, FAST and FAIR B+트리[7]와 유사하게, 중간에 비어있는 공간을 만들거나 삭제하기 위해 키-밸류 쌍을 하나씩 이동할 수도 있다. 이 방법은 쓰기 의존성이 있는 저장 명령어가 순서를 변경하지 않는다는 사실을 활용한다. wB+트리는 항상 clflush를 3-4번 호출하는 반면, FAST and FAIR B+트리에서는 키가 삽입되는 위치에 따라 clflush 호출 횟수가 달라진다. 트리 노드의 크기가 작을 때, FAST and FAIR B+트리는 wB+트리에 비해 적은 clflush 호출을 발생시킨다. 그러나 트리 노드가 클 때는 많은 이동 연산으로 인해 clflush 호출 횟수가 증가하여 성능이 저하된다.

3세대 인텔 Xeon 프로세서에서 지원하는 확장 ADR(eADR)은 CPU 캐시 영역도 전원 손실 시 보호되는 영역에 포함되도록 보장한다. 가시성이 지속성과

동등해지므로, 쓰기 작업이 전역적으로 가시적이면 지속적이다. eADR은 빠르고 지속 가능한 락프리 인덱스 데이터 구조를 구현하기 쉽게 만든다. eADR 지원을 통해 clflush의 명시적 호출이 필요 없다. 따라서 clflush 호출 횟수를 줄이는 것보다 실행되는 명령어의 수를 최소화하고 구현이 간단한 방식을 선택하는 것이 유리하다.

4-4. SkipBlockList의 노드 분할

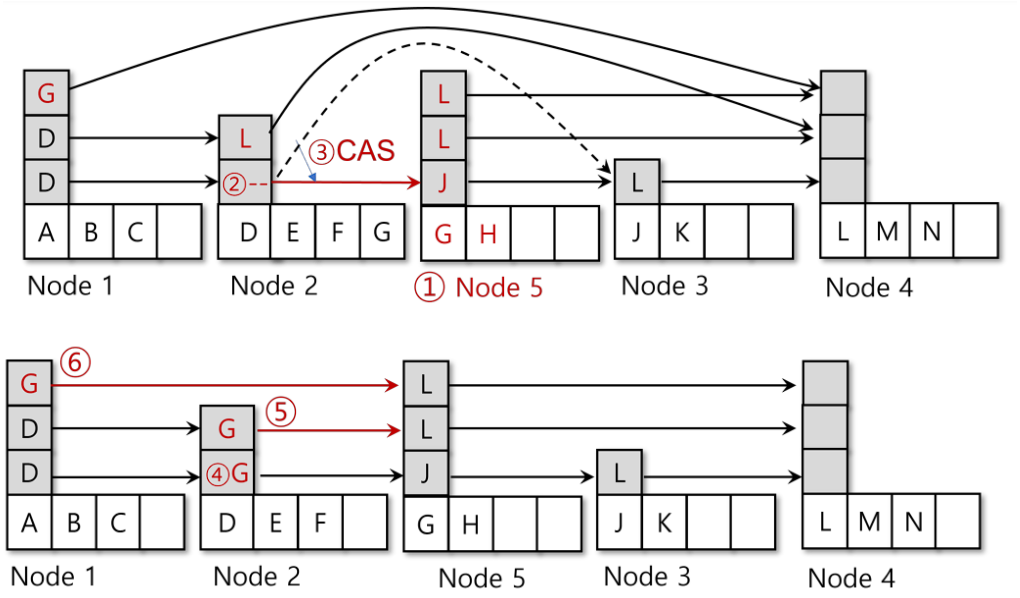


그림 4-3 SkipBlockList의 노드 분할 과정

SkipBlockList의 분할 알고리즘은 스킵리스트와 B+트리의 하이브리드 형태이다. 노드가 오버플로우되면, 확률적으로 선택된 높이의 새 노드를 생성하고 키의 중간값보다 큰 키를 가진 키-벨류 쌍을 새 노드로 복사한다. 이는 B+트리와 유사하다. 또한 각 높이의 선행 노드의 NNK와 포인터도 새 노드로 복사된다. 그림의 (c)에 보인 예에서, 1단계에서 노드 1의 최상위 레벨 NNK(G)와 노드 2의

2번째 및 3번째 레벨 NNK(L과 J)가 노드 5로 복사된다. 그런 다음 각 높이의 선행 노드의 NNK와 포인터를 업데이트하여 새 노드의 가장 작은 키와 주소를 가리키도록 한다. 최하단 높이에 다음 노드가 추가된 후, 새 노드로 복사된 키-밸류 쌍은 오버플로우 노드에서 삭제될 수 있다.

스킵리스트의 정확성을 보장하는 불변성은 상위 높이의 리스트가 하위 높이의 리스트의 부분집합이라는 것이다. 따라서 새 노드는 기존 SkipBlockList의 최하단 높이에 먼저 추가되어야 하며, 그 다음에는 차례로 더 높은 높이에 하나씩 추가되어야 한다.

4-5. SkipBlockList의 키 삭제

LSM 트리에서는 툼스톤이 사용된다. 이것은 키를 삭제하는 알고리즘의 필요성을 없애준다. 그러나, SkipBlockList는 삭제 작업을 지원하는 독립적인 인덱스이다.

키의 삭제로 인한 노드 사용도가 저하되지 않는 수준에서, 삭제 알고리즘은 B+트리의 삭제와 유사하게 동작한다. 메타데이터 섀도잉을 사용할 경우, 해당 메타데이터가 삭제되어 키-밸류 쌍이 차지하던 공간을 비어있는 공간으로 전환한다. 쉬프트 방식을 사용할 때는 인접한 키-밸류 쌍이 한 칸씩 이동하여 삭제된 키-밸류 쌍이 사용하던 공간을 채운다.

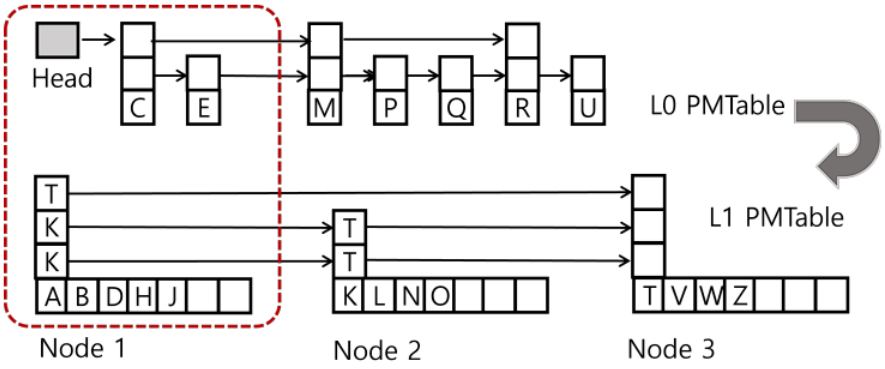
제5장 Block Compaction

BlockListDB에서 L0 PMTable은 Index-Unified Logging (IUL)에 의해 생성된 전통적인 스킵리스트 구조이고, L1 PMTable은 SkipBlockList 구조이다. Block Compaction은 L0에서 L1로의 컴팩션을 말하며 이 과정은 L0의 전통적인 스킵리스트를 L1의 SkipBlockList 구조와 병합한다. 이 과정에서 IUL에서 사용된 Braided Skiplist 구조[12]를 SkipBlockList에서도 활용하여 NUMA 지역성을 향상시킨다.

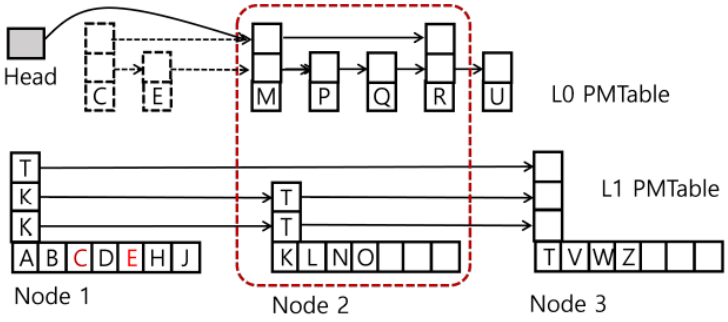
Block Compaction은 동시에 L0 및 L1 PMTable을 순회한다. 즉, 키 비교 결과에 따라 각 PMTable의 포인터를 다음 노드로 이동시킨다. 이때 두 스킵리스트에 저장된 키를 순서대로 방문할 때 L0의 키는 순차적으로 정렬되어 있다는 사실을 활용하여 첫 노드부터 반복적으로 순회할 필요가 없이 탐색 중인 노드에서 다음 키의 탐색을 이어 나갈 수 있다.

제자리 병합 방식의 Zipper Compaction과 달리, BlockListDB에서의 L0에서 L1로의 컴팩션은 L0 PMTable에서 L1 PMTable로 키-밸류 쌍을 복사한다. 이는 특히 키-밸류 쌍이 클 때 쓰기 작업의 양을 증가시켜 쓰기 증폭 문제를 일으킨다. 그러나 ListDB의 Zipper Compaction[12]과 같이 스킵리스트의 포인터를 업데이트하여 제자리 병합을 수행하는 것은 하나의 키-밸류 쌍의 병합을 위해 여러 포인터를 업데이트해야 한다는 것을 의미한다. 이로 인해 랜덤 쓰기가 발생하고 쓰기 성능이 크게 저하된다.

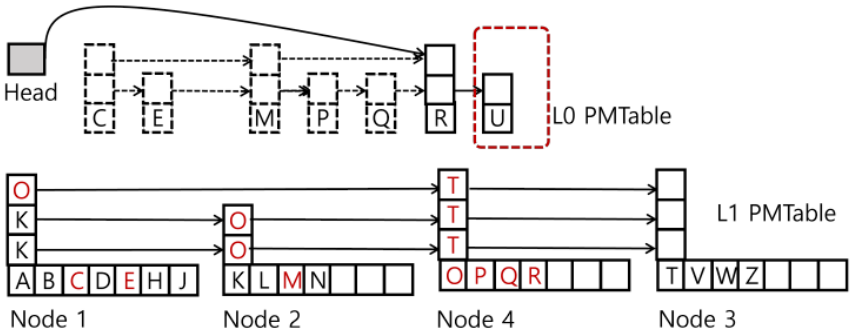
각 L0 키에 대해, 컴팩션 쓰레드는 L0 키가 삽입될 SkipBlockList 노드를 찾는다. 해당 노드에 여유 공간이 있다면, 그곳에 키-밸류 쌍을 삽입한다. 공간이 없다면, 노드는 분할된다. L1 PMTable에 여러 L0 키를 일괄적으로 삽입할 때, 여러 L0 키-밸류 쌍이 하나의 SkipBlockList 노드 내에 저장될 수 있다.



(a) Adding C and E to Node 1



(b) Adding M, P, Q, R, and S to Node 2 Causes a Split



(c) Head Moves to Top-level Predecessor

그림 5-1 Block Compaction 과정

그림은 L0에서 L1로의 Block Compaction 예시를 보여준다. 컴팩션 쓰레드는 L0 PMTable의 첫 번째 노드인 C와 L1 PMTable의 첫 번째 노드인 노드 1을 읽는다. 이는 C를 노드 1의 NNK, 즉 최상단 높이의 T, 두번째 높이의 K, 그리고 하단 높이의 K와 비교한다. C가 K보다 작기 때문에, C를 노드 1의 비어있는 공간에 삽입하고 다음 L0 노드인 E를 방문한다. E 또한 K보다 작으므로, E를 노드 1의 비어있는 공간에 삽입하고 다음 노드인 M을 방문한다. M이 최하단 높이의 NNK인 K보다 크기 때문에, 노드 1에 추가할 키-밸류 쌍이 더 이상 없다고 판단한다. 따라서, 노드 1의 키 개수를 두 개만큼 증가시키고 노드 1의 오프셋 배열을 업데이트하여 C와 E가 포함되도록 한다. 그런 다음, 하단 높이의 다음 포인터를 따라 L1 PMTable의 노드 2로 이동한다. L1 PMTable에서 다음 노드로 이동한 후, L0 PMTable의 최상단 높이의 선행 노드가 새로운 헤드 노드가 된다.

마찬가지로 컴팩션 쓰레드는 L0 PMTable에서 M, P, Q, R, S를 차례로 방문하며 이들을 모두 노드 2에 저장하려고 시도한다. 그러나 노드 2에는 충분한 공간이 없어 그림 (c)에서 보이는 것처럼 노드의 분할이 발생한다.

제6장 Lookup Cache

BlockListDB는 DRAM 공간 일부를 Second Chance Lookup Cache로 사용한다. Second Chance Lookup Cache는 비휘발성 메모리 상 키의 주소가 키와 함께 저장되어 비휘발성 메모리 상에서 L1 PMTable 탐색 과정 일부를 생략할 수 있도록 한다. Second Chance Lookup Cache에 사용되는 자료 구조에 따라, 메타데이터를 저장하기 위한 추가적인 공간이 필요하다. 키의 크기가 작을수록 키에 비해 메타 데이터 저장을 위해 필요한 공간이 늘어나게 된다.

BlockListDB는 정렬된 배열을 Second Chance Lookup Cache로 사용하다. 이것은 동일한 캐시 공간에 최대한 많은 수의 키를 저장할 수 있게 한다. 하지만 정렬된 배열을 캐시로 사용할 때 키 삽입에 대한 오버헤드 문제와 탐색 성능에 대한 한계가 존재한다.

정렬된 배열에 키를 하나씩 삽입한다면 각 삽입에 대해 삽입 공간을 확보하기 위한 쉬프트 과정이 필요하다. 하지만 BlockListDB는 키를 하나씩 삽입하는 방식을 사용하지 않고 각 L0에서 L1으로 컴팩션이 끝날 때 마다 정렬된 배열 전체를 업데이트 하는 방식을 사용한다. 배열을 업데이트 할 때, 우선 배열을 모두 활용할 수 있는 SkipBlockList의 제일 높은 높이를 타겟 높이로 선정한다. 이후 타겟 높이의 노드들을 순회하며 각 노드의 제일 작은 키를 정렬된 배열에 삽입한다.

정렬된 배열의 탐색 속도는 이진 탐색의 속도로 제한되어 있다. 따라서 정렬된 배열에 존재하는 키의 개수가 증가한다면 Trie와 같은 다른 자료 구조의 탐색 성능을 따라잡지 못할 것이다. BlockListDB의 Second Chance Lookup Cache는 정렬된 배열의 탐색 속도를 증가시키기 위한 Learned Index 방식을 제공한다. Learned index는 정렬된 배열의 탐색 속도를 증가시키는 것을 목적으로 고안된 데이터셋의 분포를 반영하는 인덱스로 매우 적은 메모리 공간만을 필요로 한다.

따라서 BlockListDB는 적은 캐시 공간을 활용하여 Learned Index를 학습하는 데에 사용한다. 하지만 만약 데이터셋의 분포가 Learned Index를 사용하기에 적합하지 않은 경우를 고려해 Learned Index를 학습하는 중 설정된 최대 선분 개수보다 많은 선분을 학습하는 순간 데이터셋의 분포가 학습에 적합하지 않다고 판단하고 학습을 중단한다. 이 경우에는 캐시를 탐색할 때 이진 탐색 방식을 사용한다.

제7장 성능 평가

7-1. 실험 환경

실험 환경은 다음과 같다. 인텔 Xeon Gold 5215 CPU (2.50 GHz, 소켓당 20 vCPUs)가 장착된 4 소켓 NUMA 서버에서 실험을 진행했다. 해당 서버는 DDR4 DRAM 256 GB(16x 16 GB)와 2 TB의 Optane DCPMM(16x 128 GB, 소켓 당 4개의 DCPMM 및 4개의 DRAM, App-Direct 모드 사용)으로 구성하였다. SSD 장치는 Samsung Z-SSD(SZ985) 800GB를 사용하였다. 해당 서버는 독점적인 디렉토리 일관성 프로토콜을 지원한다. 모든 구현은 gcc 7.5.0을 사용하여 -O3 최적화로 컴파일 되었다.

BlockListDB의 성능을 크게 두 가지로 나누어 평가했다. 첫째, BlockListDB의 디자인 각 부분이 성능에 끼치는 영향을 분석하였다. 둘째, BlockListDB의 성능을 기존 연구인 ListDB와 비교하여 전반적인 성능을 측정하였다.

실험 진행 전 ListDB에 존재하는 성능 버그를 최적화하여 비교하였다. ListDB의 Zipper Compaction 스캔 단계에서 상위 높이의 포인터를 사용하지 않고 하단 높이의 포인터만을 수정하는 버그이다.

본 실험은 야후에서 제공하는 YCSB 벤치마크를 사용하여 진행되었다. 모든 실험에 사용된 키-밸류의 크기는 각각 8 바이트로 동일하다. 또한 모든 범위 탐색 쿼리는 범위에 해당하는 50개의 키에 대한 밸류를 탐색한다.

7-2. 비휘발성 메모리 접근 횟수 분석

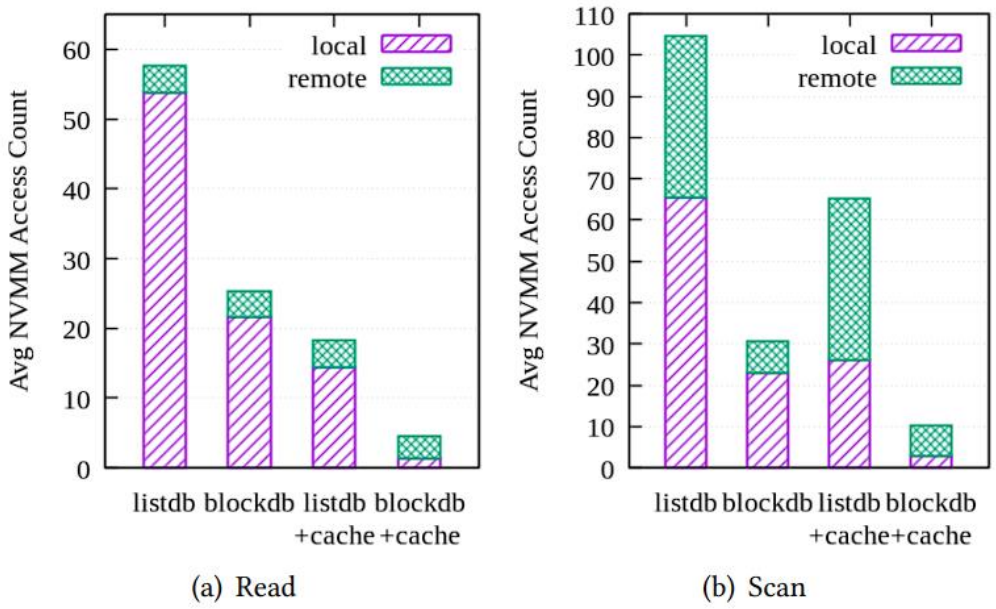


그림 7-1 탐색 과정에서 비휘발성 메모리 접근 횟수

SkipBlockList의 성능 향상의 주된 원인은 비휘발성 메모리 접근 횟수의 감소이다. 이를 증명하기 위하여, ListDB와의 비교 분석을 통해 BlockListDB에서 부분 및 범위 탐색 시 발생하는 비휘발성 메모리 접근 횟수를 측정했다. Zipfian 분포를 기반으로 한 1억 개의 키-밸류 쌍을 로드 한 후 실험을 진행하였다. 이후, YCSB Workload C와 E에 따라 100% 읽기 및 100% 스캔 작업으로 구성된 1억 개의 쿼리를 처리하였다. 실험은 80개의 클라이언트 쓰레드를 활용하여 진행한다.

비교 대상으로, ListDB와 FillDB는 각각 45MB의 Second Chance Lookup Cache를 사용하거나 사용하지 않는 조건으로 나누어 비교하였다. 그래프에서 'local'은 동일한 NUMA 노드 상에서의 비휘발성 메모리 접근 횟수를, 'remote'는 서로 다른 NUMA 노드에 대한 비휘발성 메모리 접근 횟수를 의미한다.

BlockListDB는 모든 유형의 탐색 시 ListDB에 비해 비휘발성 메모리 접근 횟수가 감소한 것을 알 수 있다. 부분 탐색 시 캐시를 사용하지 않는 경우,

BlockListDB는 ListDB에 비해 비휘발성 메모리 접근 횟수가 57% 감소한다. 이는 SkipBlockList가 노드의 전체 개수가 적고 NNK를 통해 불필요한 비휘발성 메모리 접근을 피할 수 있기 때문이다. 캐시를 사용했을 때 비휘발성 메모리 접근 횟수의 더욱 큰 차이를 보인다. 이것은 많은 키를 담을 수 있는 배열 형태의 캐시를 사용하고 SkipBlockList의 전체 노드 개수 또한 적기 때문에 SkipBlockList의 탐색 과정의 대부분을 생략할 수 있기 때문이다.

범위 탐색에서는 비휘발성 메모리 접근 횟수 차이가 더욱 두드러진다. 이것은 하나의 노드에 여러 개의 키를 저장하는 SkipBlockList는 한번의 비휘발성 메모리 접근으로 여러 키를 가져올 수 있기 때문이다. 따라서 찾고자 하는 키의 개수만큼 최하단 노드의 순회가 필요한 전통적인 스킵리스트와 다르게 SkipBlockList는 최하단 노드를 적게 순회한다. Braided Skiplist 구조에서 최하단 노드의 순회는 다른 NUMA 노드로의 접근을 유발하기 때문에 탐색 성능에 치명적이다. SkipBlockList 구조는 다른 NUMA 노드로의 접근을 최소화 하여 효과적으로 범위 탐색 성능을 개선한다.

7-3. 각 디자인 요소에 대한 성능 분석

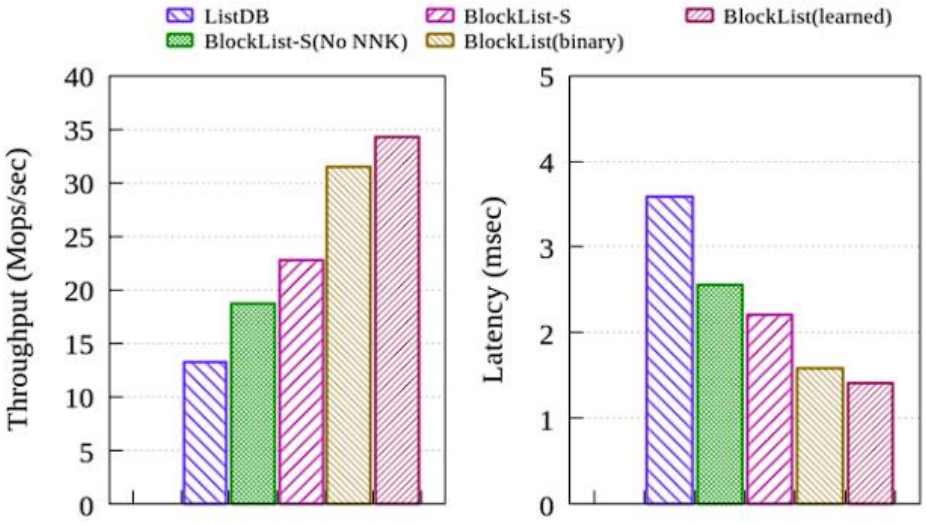


그림 7-2 BlockListDB의 디자인 요소에 따른 성능

이 실험은 BlockListDB의 각 디자인 요소가 탐색 성능을 얼마나 향상시켰는지에 대한 실험이다. Zipfian 분포의 YCSB Workload C 실험과 동일한 설정을 사용하여 10억개의 키-밸류 쌍을 로드 한 뒤 10억 개의 부분 탐색 쿼리를 처리하였고, 60개의 클라이언트를 사용하여 실험을 진행하였다. ListDB는 논문에서 제시한 SIZE 정책의 캐시를 사용하였다. BlockListDB-S(No NNK)는 ListDB에서 하나의 노드에 여러 개의 키를 배열 형태로 저장한 버전이며, SkipBlockList에서 NNK만 적용하지 않은 버전이다. 이 구조를 적용했을 때, ListDB에 비해 41%의 쓰루풋 증가를 보였다. BlockListDB-S는 ListDB에 SkipBlockList 구조를 적용한 버전으로, NNK를 사용하여 ListDB 대비 72%의 쓰루풋 향상을 이끌어냈다.

BlockListDB(binary search)와 BlockListDB(learned index)는 각각 SIZE 방식의 캐시가 아닌 배열 방식의 캐시를 적용한 버전이다. BlockListDB(binary search)는 배열 캐시 내에서 이진 탐색을 사용하여 키를 찾는 방식이고, BlockListDB(learned index)는 배열 캐시에 Learned Index 방식으로 학습된 모델을 이용하여 키를 찾는 방식이다. SIZE 방식의 캐시를 배열 캐시를 변경했을

때 최소 37%의 쓰루풋 증가를 보여준다. 이는 배열 형태의 캐시가 더 많은 키들을 저장할 수 있기 때문이다. 여기에 Learned Index 방식을 적용하면 데이터셋의 분포에 따라 배열을 더욱 빠르게 탐색할 수 있다.

결과적으로, SkipBlockList의 모든 설계 요소를 적용하였을 때 ListDB 보다 부분 탐색 쿼리를 처리하는 성능을 159% 향상시키는 결과를 보인다.

7-4. 컴팩션 방식에 따른 성능 분석

BlockListDB는 SkipBlockList를 위한 Block Compaction 기법을 활용한다. 새로운 컴팩션 방식이 기존 컴팩션 방식인 Zipper Compaction 과 비교하여 어떠한 성능 차이를 나타내고 이것이 복합적인 워크로드를 수행할 때 어떤 영향을 주는지 확인한다.

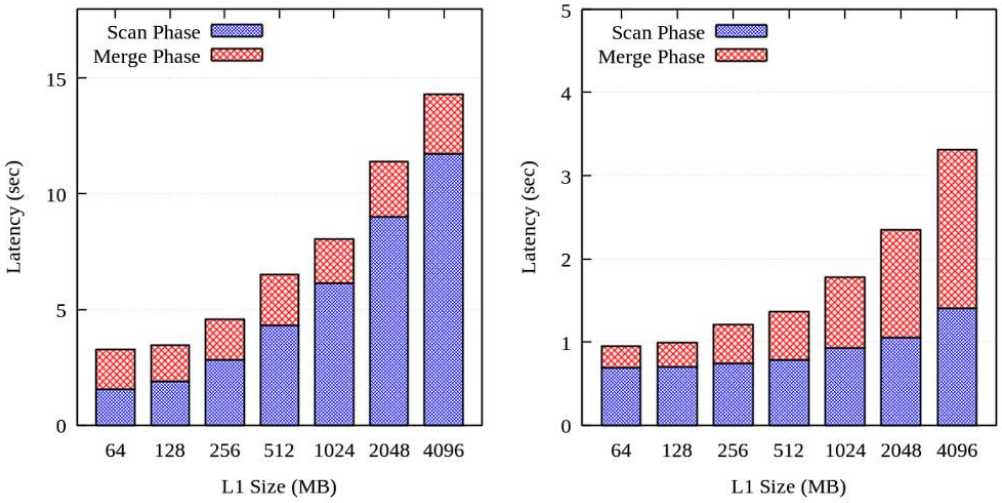


그림 7-3 컴팩션 방식에 따른 L0 컴팩션 소요 시간

첫번째 실험은 컴팩션 방식 자체의 성능을 비교한 실험이다. L1 PMTable의 크기에 따라 L0 PMTable 하나가 병합되는 데 걸리는 시간을 측정하였다. Zipfian

분포의 YCSB Workload A를 기반으로 워크로드를 생성하였다. L1 PMTable로 병합되는 하나의 L0 PMTable 크기는 64MB로 설정되었다. 다른 작업이 성능 측정에 영향을 주지 않기 위해 클라이언트 요청이 존재하지 않는 환경에서 1개의 백그라운드 쓰레드를 사용하여 컴팩션을 진행하였다.

컴팩션은 크게 두가지 단계로 구분된다. ‘Scan Phase’는 각각 L0 PMTable의 키를 삽입할 L1 PMTable의 위치를 탐색하거나 기록하는 탐색 단계이다. Zipper Compaction에서 스택에 각 포인터의 업데이트 정보를 기록하는 것과 Block Compaction에서 SkipBlockList를 순회하는 과정이 이에 해당한다. ‘Merge Phase’는 실제로 키의 삽입이 이루어지는 병합 단계이다. 컴팩션에 걸리는 총 시간을 컴팩션의 두가지 단계로 구분 지어 측정하였다.

동일한 조건에서 Block Compaction이 Zipper Compaction보다 컴팩션을 완료하는 데 걸리는 시간이 현저히 낮은 것을 보여준다. 컴팩션 성능 차이의 가장 큰 이유는 탐색 단계에서 소요된 시간에서 드러난다. L1 PMTable의 크기가 4GB까지 늘어났을 때 Zipper Compaction은 탐색 단계에 약 12초가 소요되는 반면, Block Compaction은 탐색 단계에 단 1.5초 가량 소요된다. 이러한 이유는 비휘발성 메모리에서 전통적인 스킵리스트 구조는 SkipBlockList에 비해 탐색 성능이 좋지 않기 때문이다. 특히 L1 PMTable을 연속적으로 탐색하는 작업은 범위 탐색 성능과 비례하나, 앞선 실험에서 보여지듯 전통적인 스킵리스트 구조는 범위 탐색 시 잦은 비휘발성 메모리의 접근으로 인해 성능이 저하되는 문제가 존재한다.

반면 SkipBlockList는 병합 단계에서 소요되는 시간의 비율이 늘어나는 모습을 보여준다. 이러한 이유는 L1 PMTable의 크기가 커짐에 따라 하나의 SkipBlockList 노드안에 여러 개의 키가 삽입되는 경우가 줄어들기 때문이다. 하지만 여전히 큰 크기의 스택을 사용하고 키의 병합마다 여러 개의 포인터 업데이트를 일으키는 Zipper Compaction 보다 빠른 소요 시간을 보여준다.

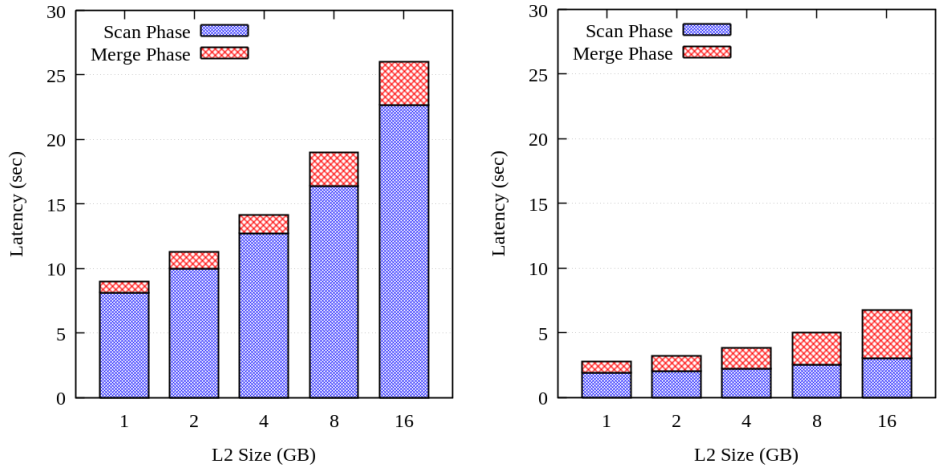
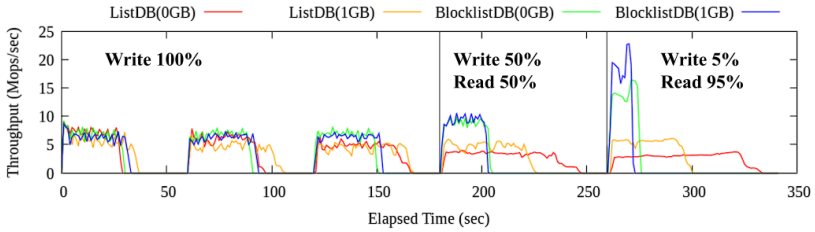
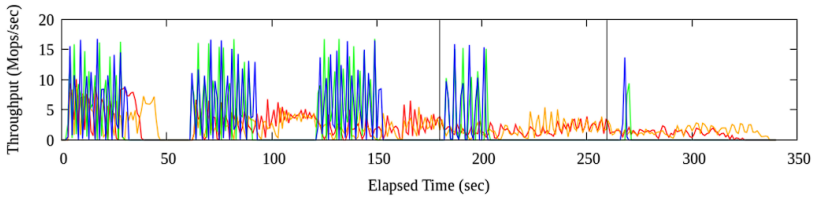


그림 7-4 컴팩션 방식에 따른 L1 컴팩션 소요 시간

마찬가지로 L1 PMTable을 각각 L2 SSTable 형식으로 변환할 때의 컴팩션 오버헤드를 분석한 실험이다. 두 컴팩션 모두 PMTable의 스캔 과정을 포함하기 때문에 전통적인 스킵리스트 구조를 사용한 ListDB의 경우 BlockListDB에 비해 L2 SSTable로 변환하는 컴팩션 오버헤드가 큰 모습을 보여준다.



(a) Forground Task Throughput



(b) Background Compaction Throughput

그림 7-5 복합 워크로드의 시간에 따른 작업 쓰루풋

두 번째 실험은 컴팩션 방식에 따른 성능 차이가 복합적인 워크로드를 처리하는 실제 환경에서 간접적으로 주는 영향을 측정하는 실험이다. Zipfian 분포의 YCSB Workload C를 기반으로 워크로드를 생성하였다. 우선 2억개의 쓰기 100%로 구성된 워크로드를 60초마다 3번 수행한다. 이후 80초마다 각각 2억개의 쓰기 50%, 읽기 50%로 구성된 워크로드와 2억개의 쓰기 5%, 읽기 95%로 구성된 워크로드를 수행한다. 클라이언트가 체감하는 포그라운드 작업의 쓰루풋과 컴팩션에 대한 백그라운드 작업의 쓰루풋을 시간의 흐름에 따라 측정하였다. 비교군은 ListDB와 BlockListDB에 대해 각각 1GB의 캐시를 사용했거나 사용하지 않은 버전을 비교하였다.

실험 결과 BlockListDB의 빠른 컴팩션 속도가 준수한 탐색 성능을 보장하는 것이 나타난다. ListDB의 경우, 쓰기 위주의 작업을 할 때 느린 Zipper Compaction이 마무리 되지 못하는 모습을 나타낸다. 이렇게 마무리되지 못한 컴팩션은 키 범위가 겹치는 L0 PMTable을 남겨두게 되고 이것은 탐색 성능 저하로 이어진다. 또한 컴팩션을 위해 비휘발성 메모리의 쓰기 대역폭을 사용하여 클라이언트의 쓰기 작업 또한 지연시키는 모습을 확인할 수 있었다. 반면, BlockListDB는 대부분 워크로드의 수행이 끝날 때 컴팩션 작업도 완료되는 모습을 보여준다. 컴팩션 작업의 완료는 키 범위가 겹치는 L0 PMTable를 제거하여 탐색 성능을 크게 개선하는 역할을 한다.

7-5. 블록 장치로의 컴팩션 성능 분석

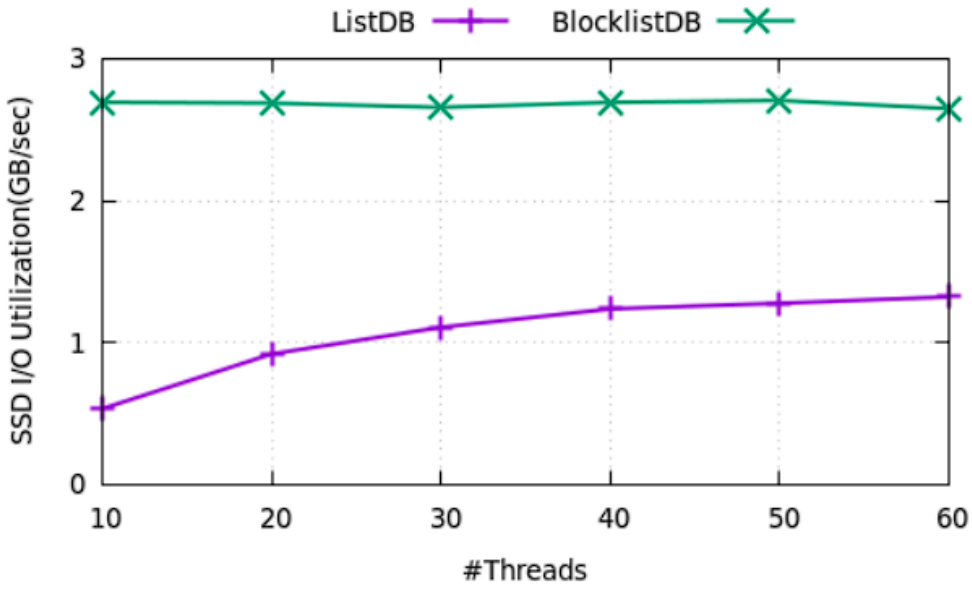


그림 7-6 블록 장치로의 컴팩션 시 IO 대역폭 활용도

BlockListDB의 SkipBlockList는 비휘발성 메모리와 블록 장치 사이의 중간 인덱스 계층의 역할을 하기 위해 고안되었다. 이 실험은 SkipBlockList가 중간 인덱스 계층의 역할을 수행하는지 검증하는 실험이다. 블록 장치로의 컴팩션 시 IO 대역폭을 최대로 활용하고 있는 지 여부를 통해 블록 장치로의 컴팩션에 최적화 되었는지 여부를 확인할 수 있다.

Zipfian 분포의 YCSB Workload A를 기반으로 4억개의 로드를 수행하였다. 이후 컴팩션을 통해 생성된 L1 PMTable을 블록 장치인 SSD로 컴팩션을 진행하며 평균적인 IO 대역폭 활용도를 측정한다. 10개부터 60개까지 백그라운드 쓰레드 수를 조절하였다.

실험 결과 BlockListDB는 블록 장치의 최대 대역폭인 2.7GB/s의 대역폭을 모두 활용하여 컴팩션을 진행한 반면, ListDB는 쓰레드 수가 최대인 경우에도 절반 수준의 대역폭 활용도를 보여주었다. IO 대역폭 활용도의 차이가 나는 이유는 ListDB가 사용하는 전통적인 스킵리스트 구조의 낮은 범위 탐색 성능 때문이다.

블록 장치로 컴팩션을 하기 위해서 L1 PMTable의 키들을 범위 탐색해야 한다. 이때 ListDB의 L1 PMTable에 대한 범위 탐색 성능이 최신 블록 장치의 빠른 대역폭을 따라잡지 못한다. 따라서 블록 장치로의 컴팩션 시 스킵리스트의 느린 범위 탐색 성능에서 병목 현상이 일어나게 되는 것이다. 반면 범위 탐색 성능이 최적화된 SkipBlockList의 경우는 블록 장치의 대역폭에서 병목 현상이 일어나게 된다. 이것은 SkipBlockList가 블록 장치로의 컴팩션에 최적화되었다는 것을 의미하고, 더 빠른 차세대 SSD를 적용한 구조에서도 블록 장치의 IO 대역폭을 최대로 활용할 수 있는 가능성을 보여준다.

7-6. YCSB 벤치마크 성능 분석

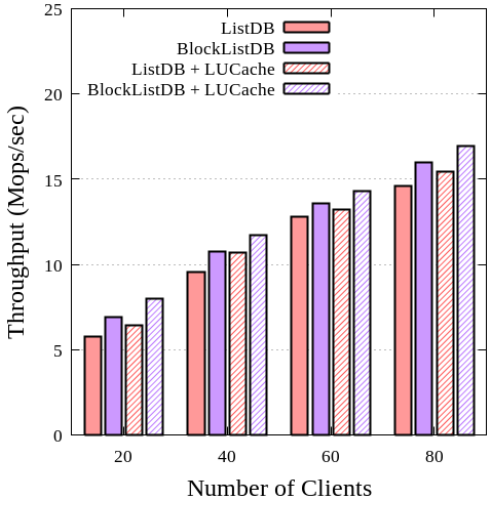


그림 7-7(a) YCSB 로드의 성능

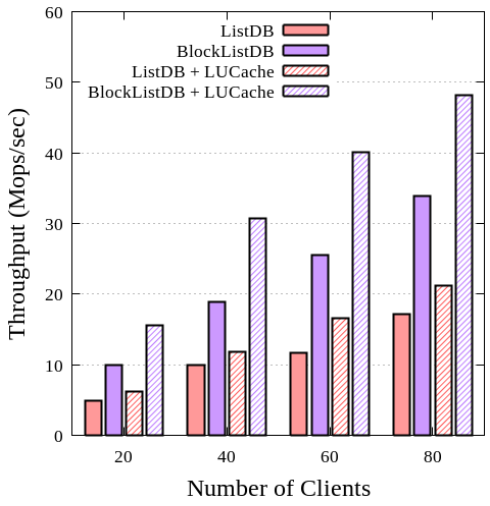


그림 7-7(b) YCSB 워크로드 A의 성능

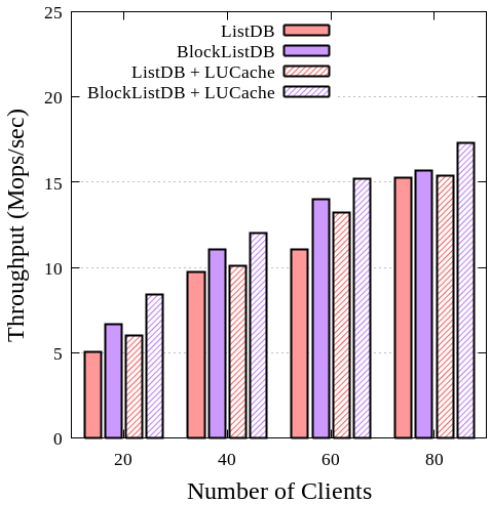


그림 7-7(c) YCSB 워크로드 B의 성능

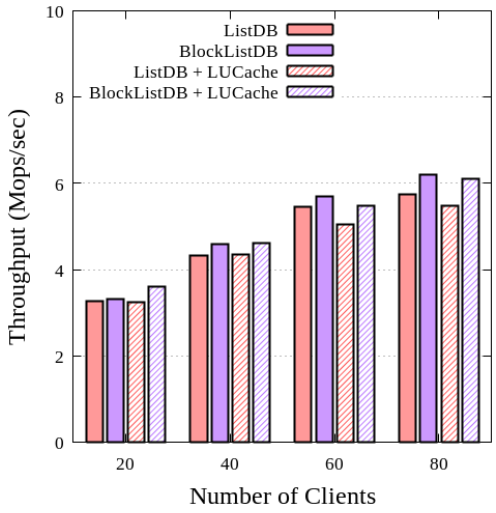


그림 7-7(d) YCSB 워크로드 C의 성능

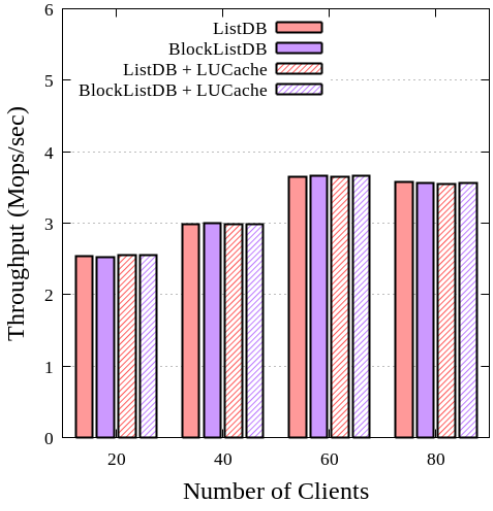


그림 7-7(e) YCSB 워크로드 D의 성능

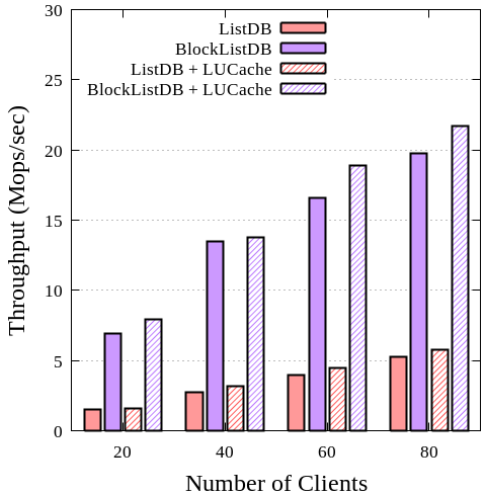


그림 7-7(f) YCSB 워크로드 E의 성능

이 실험은 전반적인 성능을 측정하기 위해 다양한 YCSB 벤치마크의 워크로드를 수행하여 성능을 측정하였다. 워크로드는 Zipfian 분포를 사용하여 1억개의 데이터를 미리 로드한 뒤, 1억개의 쿼리를 수행하였다. 비교군은 ListDB와 BlockListDB에 대해 45MB의 Second Chance Lookup Cache를 사용한 버전과

사용하지 않은 버전을 비교하였다. 쿼리를 보내는 클라이언트 쓰레드의 수를 20, 40, 60, 80으로 조절하여 실험을 진행하였다. 컴팩션을 진행하는 백그라운드 쓰레드의 수는 클라이언트 쓰레드의 수의 절반으로 조정되었다.

실험 결과, 모든 워크로드에 대해 BlockListDB의 성능이 개선된 것을 확인할 수 있다. 특히 이러한 결과는 탐색 쿼리를 주로 수행하는 워크로드 C와 E에서 더욱 두드러진다. 본 논문에서 BlockListDB를 위해 제안한 다양한 디자인이 탐색 성능을 효율적으로 향상 시킨 것을 알 수 있다. 워크로드 C의 경우, 클라이언트 쓰레드를 80개까지 사용하고 캐시를 활성화하였을 때, 95%의 성능 향상을 나타내었다. 워크로드 C에 비해 워크로드 E의 성능 향상 폭이 더욱 크게 드러났다. 이러한 이유는 공간 지역성의 확보가 범위 쿼리의 성능 향상에 크게 기여하기 때문이다.

제8장 결론

본 연구는 비휘발성 메모리와 블록 장치의 중간 계층의 역할을 하는 SkipBlockList를 제안하여 바이트 단위의 접근이 가능한 인덱스인 스킵리스트를 효율적으로 SSTable 형태로 변환시킬 수 있도록 하였다. SkipBlockList는 낮은 공간지역성으로 인해 탐색 성능이 저하되는 제자리 병합 방식의 스킵리스트의 문제를 해결한다. SkipBlockList는 비휘발성 메모리의 장점을 최대화하고, 락프리 스킵리스트의 특성을 이용하여 공간 지역성을 높이며, 키-밸류 쌍의 관리와 메타데이터 크기 문제를 효과적으로 해결한다.

SkipBlockList를 중간 계층으로 활용하고 있는 이기종 저장소를 위한 키-밸류 스토어인 BlockListDB는 계층 별로 역할을 구분하여 서로 다른 인덱스 구조를 활용한다. BlockListDB의 최근 데이터를 저장하는 상위 레벨인 L0에는 쓰기 지연 문제를 완화하기 위해 쓰기에 최적화 된 LIstDB의 IUL 방식을 활용한다. 중간 레벨인 L1는 SkipBlockList 구조를 통해 공간 지역성을 확보하여 탐색 성능을 개선하고 블록 장치로의 컴팩션을 효율적으로 수행할 수 있도록 한다. 하위 레벨인 L2 이하의 레벨은 대용량의 블록 장치를 활용할 수 있게 하여 확장성을 높이는 역할을 한다.

SkipBlockList의 적용을 통해 탐색 성능이 크게 개선되었을 뿐 아니라, 개선된 탐색 성능은 컴팩션 과정에서 탐색으로 소요되는 시간을 감소시켜 컴팩션 오버헤드를 줄여주는 효과를 보인다. 추가적인 탐색 성능 향상을 위해 활용될 수 있는 정렬된 배열 형태의 Second Chance Lookup Cache를 적용하였을 때 최대의 탐색 성능 향상을 보여준다.

참 고 문 헌

[1] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Chang sheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In 2020 USENIX Annual Technical Conference (USENIX ATC 20). USENIX Association, 17-31. https://www.usenix.org/conference/atc20/presentation/yao

[2] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in Non-volatile Main Memory. Proceedings of the VLDB Endowment 8, 7 (Feb. 2015), 786-797.

[3] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. Proceedings of the VLDB Endowment 8, 7 (2015), 786-797.

[4] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 1077-1091.

[5] Yongju Song, Wook-Hee Kim, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. Prism: Optimizing Key-Value Store for Modern Heterogeneous Storage Devices. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023). Association for Computing Machinery, New York, NY, USA, 588-602.

[6] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and

Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD). 371-386.

[7] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST). 187-200.

[8] Intel Optane Persistent Memory [n. d.]. Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html. ([n. d.]).

[9] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST). 191-205.

[10] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In Proceedings of the 2018 USENIX Annual Technical Conference (USENIX TC). 993-1005.

[11] Dongui Kim, Chanyeol Park, Sang-Won Lee, and Beomseok Nam. 2020. BoLT: Barrier-Optimized LSM-Tree. In Proceedings of the 21st International Middleware Conference (Middleware). 119-133.

[12] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young ri Choi, Alan Sussman, and Beomseok Nam. 2022. ListDB: Union of Write-Ahead Logs and Persistent SkipLists for Incremental Checkpointing on Persistent Memory. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). USENIX Association, Carlsbad, CA, 161-177.

https://www.usenix.org/conference/osdi22/presentation/kim
[13] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-structured Merge-tree (LSM-tree). Acta Informatica 33, 4 (June 1996), 351-385.
[14] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST). 257-270.
[15] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. Proceedings of the VLDB Endowment 13, 4 (Dec. 2019), 574-587.
[16] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In Proceedings of the 17th USENIX Conference on File and Storage (FAST). 31-44.

ABSTRACT

Progressive Compaction for Enhancing Spatial Locality in LSM Trees for Non-Volatile Memory

Chung Ju Won
Department of Computer Science Engineering
Sungkyunkwan University

Most of byte-addressable key-value stores designed for NVMM leverage the in-place structural modification operations (SMOs) to insert, update, or delete keys through 8-byte pointer updates. While this in-place SMOs help mitigate the write amplification problem, excessive in-place updates lead to decreased spatial locality, resulting in degraded read performance. In this study, we propose SkipBlockLists optimized for NVMM as an intermediate layer for serializing byte-addressable in-memory SkipList indexes into SSTables optimized for block devices. SkipBlockLists manage multiple keys within a single node, combining the advantages of SkipLists and B-trees. BlockListDB effectively reduces the amount of coarse-grained IO by using in-place SMO, while improving locality within node through copy-on-write.

Keywords: Non-Volatile Main Memory, LSM-Tree, Key-Value Store

碩士學位請求論文

휘발성 메모리를 위한 LSM 트리의 점층적 컴팩션을 통한 공간 지역성 확보

2024

鄭柱元