LB+tree: Lock-free 기법을 적용한 Lazy-Balanced+tree 자료구조

장창수[1], 김욱희[2], 이성진[3], 김재호[3]

[1]경상국립대학교 항공우주 및 소프트웨어 공학부

[2]건국대학교 컴퓨터공학부

[3]경상국립대학교 소프트웨어공학과

windowsu980@gmail.com, wookhee@konkuk.ac.kr, insight@gnu.ac.kr, jaeho.kim@gnu.ac.kr

LB+tree: Lazy-Balanced+tree Index with Lock-free mechanism

Changsu Jang[1], Wookhee Kim[2], Seongjin Lee[3], Jaeho Kim[3]

[1]School of Aerospace and Software Engineering, Gyeongsang National University

[2]School of Computer Science and Engineering, Konkuk University

[3]Department of Software Engineering, Gyeongsang National University

요 약

수십~수백 개의 코어를 탑재한 매니코어 시스템의 도입으로 스레드 동기화 중요성이 부각되고 있다. 특히, 데이터베이스 인덱스로 널리 사용되는 B+tree의 동시성 성능 향상 요구가 더욱 증대된다. B+tree의 업데이트 연산은 트리의 재균형 과정을 수반하며, 이로 인해 많은 수의 스레드가 동시에 접근할 경우 심각한 병목현상이 발생한다.

본 논문에서는 이러한 문제를 해결하기 위해 트리의 재균형 과정에서 발생하는 오버헤드를 최소화하고, 동기화를 위한 lock의 범위를 줄이며, 다중 버전 동기화 방안을 도입하는 전략으로 동시성 성능을 극대화하는 변형된 B+tree 데이터 구조인 LB+tree (Lazy balanced B+tree)를 제안한다. LB+tree는 기존 구조와 비교해 동시성 향상뿐만 아니라 성능 병목현상을 효과적으로 완화하며 매니코어 환경에서도 높은 처리 성능을 제공한다.

1. 서 론

매니-코어 시스템의 확산으로 소프트웨어의 병렬 처리와 확장성 확보가 중요해지면서, 데이터베이스 인덱스 구조, 특히 널리 사용되는 B+tree의 병렬 성능 개선이 중요한 과제로 떠오르고 있다. 기존 B+tree의 락 기반 및 락 프리 동기화 방식은 매니-코어 환경에서 락 경합이나 CAS 실패 등으로 인해 병목 현상이 발생하며, 확장성에 한계를 드러낸다.

이에 따라 동기화 전략으로 락 기반, 락 프리, 다중 버전, 트랜잭셔널 메모리 기반 기법들이 제안되어 왔으며, 그중 다중 버전 기반인 RCU[1]는 읽기 동시성은 우수하나, 쓰기에서 락 의존성이 크고, 프로그래밍 복잡도가 높다는 한계가 있다. 이를 보완한 RLU[2]와 MV-RLU[3]는 보다 유연한 동시성과 단순한 프로그래밍 인터페이스를 제공한다.

본 논문에서는 MV-RLU를 활용한 새로운 B+tree 구조인 LB+tree를 제안한다. LB+tree는 기존 구조를 유지하면서 재균형 연산(SMO)을 지연 및 국소화하여 쓰기 병목을 줄이고, MV-RLU 기반 다중 버전 접근을 통해 읽기 동시성을 극대화한다. 이는 BW-tree[4]처럼 전체 구조를 재설계하지 않고도 성능 병목을 해결하고, 매니-코어 환경에서 효율적인 인덱스 구조를 제공할 수 있는 대안을 제시한다.

본 연구에서 제안하는 LB+tree는 기존 B+tree 구조를 유지하면서, Split/Merge Operation(SMO)을 지연 및 국소화하여 쓰기 병목을 완화하는 것을 목표로 한다. SMO 지연으로 인한 구조적 불일치는 락 프리 기법과 구조적 수정으로 해결하며, 높은 동시성을 유지하도록 설계되었다. 논문에서는 LB+tree의 기본 연산과 SMO 과정을 기존 방식과 비교하며, 향후 다양한 워크로드와 멀티코어 환경에서의 성능 평가를 통해 그 효용성을 검증할 예정이다.

2. LB+tree의 구조와 알고리즘

2.1. LB+tree 구조

<그림 1>은 LB+tree의 전체 구조를 보여준다. LB+tree는

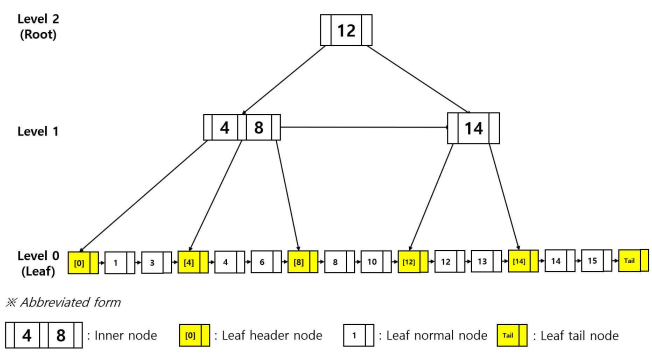

그림 1. Abbreviated form of LB+tree structure

두 가지 형태의 노드 타입으로 구성되어 있다. Level 1부

터 Root까지는 Inner 노드, Level 0은 Leaf 노드로 이루어진다. 각 노드는 다음 노드를 가리키는 포인터를 포함하고 있어, 동일 레벨 간에 linked-list 형태로 연결된다.

<그림 2>의 Inner 노드는 state, count, level, min_key, entry_array, next_node 등의 필드를 포함한다. 여기서 state는 SMO 예정 여부를 나타내며, count는 자식 노드 수, level은 해당 노드의 레벨, min_key는 첫 번째 자식의 key 값이다. entry_array는 자식 노드의 key 및 포인터 정보를 저장하며, next_node는 같은 레벨의 다음 노드를 가리킨다.

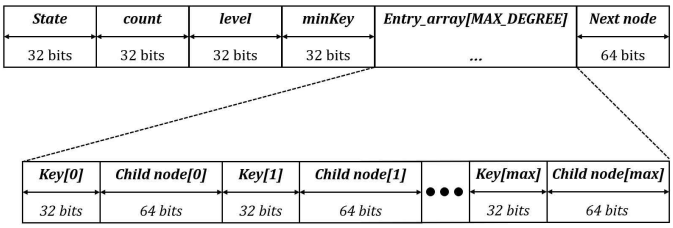

그림 2. LB+tree Inner-node data structure

<그림 3>의 Leaf 노드의 type 필드는 노드가 header, normal, tail 중 어떤 타입인지를 구분하며, normal 및 tail 노드는 key 값과 다음 노드 포인터를 포함한다. header 노드는 추가로 state와 count 정보를 저장하며, count는 해당 header와 tail 사이의 normal 노드 수를 나타낸다.

type	state	count	Key	Next node
32 bits	32 bits	32 bits	32 bits	64 bits

그림 3. LB+tree Leaf-node data structure

2.2. LB+tree 기본 연산 (Basic Operation)

LB+tree의 write/ read연산은 비교적 단순한 구조로 이루어져 있다. 이 중 Find_header()는 RLU 인터페이스를 활용해 inner 노드 탐색 과정의 일관성을 보장하며, 주어진 key 범위를 포함하는 header 노드를 반환한다. 이후 각 연산은 header 노드 내의 leaf 영역에서 수행된다.

write 연산은 header 노드의 count 값을 수정하기 위해 header 노드와 인접 노드 2개에 대한 쓰기 락을 동시에 획득해야 한다. 락 획득에 실패하면, 연산은 롤백 되고 root 노드부터 다시 시작된다. 연산이 성공한 경우에는 header 노드의 count 값을 기준으로 split 또는 merge 조건을 검사하고, 임계값을 초과하거나 미달 시 해당 노드의 상태를 변경하여 SMO준비 상태로 전환한다.

2.3. LB+tree 분할, 병합 연산 (Split Merge Operation)

기존 A Lock-Free[5] B+tree 구조에서는 split/merge 연산 수행 시, 그림 4와 같이 해당 inner 노드([14])가 split/merge operation이 종료될 때까지 child 노드들이 모두 write-lock에 걸려 병목 현상을 유발한다.

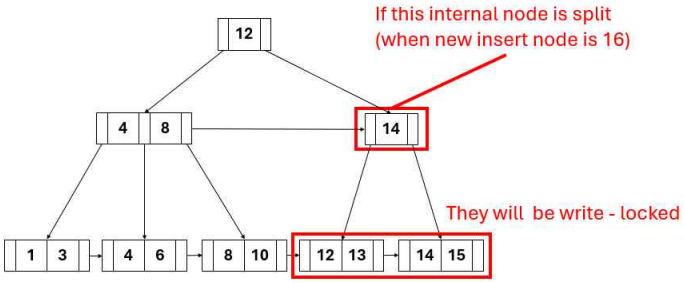

그림 4. The Bottle-neck of A Lock-free B+tree

LB+tree는 이러한 병목 문제를 해결하기 위해, Split 연산을 4단계로 분리하고 일부 연산을 지연 동기화 (Lazy Synchronization) 방식으로 처리한다. 그림 5는 split연산의 단계별 구조를 보여준다.

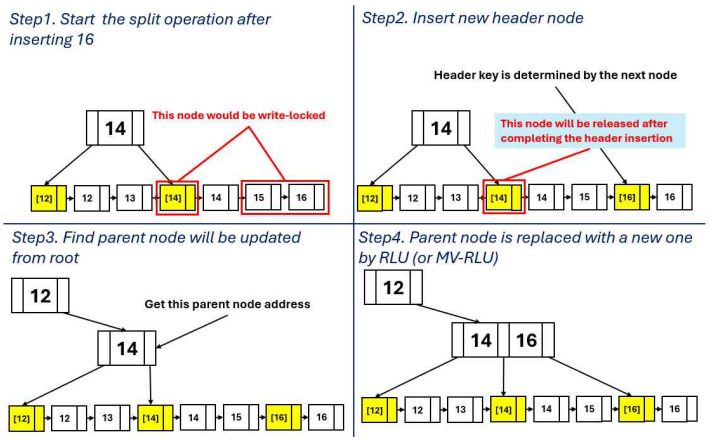

그림 5. Split Algorithm sequence

- step 1 ~ step 2: 새로운 header 노드의 삽입 등 즉시 적용 되는 연산
- step 3 ~ step 4: parent 노드 갱신, 노드 교체등 지연 적용되는 동기화 연산

한편, merge 연산은 split과 반대 방향으로 진행된다. LB+tree에서는 먼저 부모 노드에서 SMO 상태가 'FREE'인 child partner 노드를 탐색한 후, parent 노드 병합 대상 entry를 삭제하고 남은 header 노드 정보를 기반으로 entry에서 삭제된 child 노드를 병합하고 연산을 종료한다. 이러한 구조 덕분에 split 또는 merge 연산이 진행 중인 상황에서도, 그림 5와 같이 key 17과 같은 새로운

입력은 갱신 이전의 구조를 우회하여 탐색 및 삽입을 수행할 수 있다. 이는 모든 연산이 즉시 완료되지 않더라도, 일부 결과를 우선 반영하고 나머지는 지연 처리하는 방식으로, 동시성 환경에서도 일관성과 연산 가능성을 유지할 수 있도록 설계되었다.

3. 실험

본 연구에서는 삭제 연산을 제외한 기본 연산(Insert, Update, Search, Range-Search)에 대한 성능을 선행적으로 평가하였다. 실험은 100부터 1,000,000 범위의 키 값을 가지는 항목(Item)을 대상으로, 멀티스레드 개수를 점진적으로 증가시키며 10,000ms 동안 수행되었다. 각 연산의 workload 구성 비율은 다음과 같다: Insert 30%, Update 5%, Search 60%, Range-Search 5%.

3.1. 실험 환경

실험은 다음 사양의 워크스테이션에서 수행되었다.

- cpu : Intex Xeon Gold 6258R X 2
  (28cores, 56threads, 2.70GHz)
- L3 Cache : 77Mb
- Memory : DDR4-2933 ECC Registered 32GB X 12
- OS : Ubuntu 20.04.6 LTS
- Compiler: g++ 9.4.0 with -03 option

3.2. 실험 결과

X축은 스레드 개수, Y축은 실행 횟수를 나타내며, 스레드가 증가함에 따라 LB+tree의 자료구조 성능도 점진적으로 향상되었다. 싱글 스레드와 비교했을 때, 56 멀티 스레드 환경에서 Insert는 173%, Update는 322%, Search

는 325%, Scan은 185% 성능 향상을 보였다.

4. 결론

본 연구에서는 LB+tree의 성능 향상을 위해 Split/Merge Operation(SMO)을 지연하고 국소화하는 방안을 제안하였으며, RLU를 활용해 멀티 스레드 환경에서의 성능을 선행적으로 평가하였다. 그 결과, 싱글 스레드 환경에 비해 멀티 스레드 환경에서 성능이 크게 향상됨을 확인할 수 있었다. 향후 에는 다양한 워크로드와 멀티코어 환경에서 다른 자료구조와의 성능 비교를 통해 LB+tree의 효용성을 추가 적으로 검증할 계획이다.

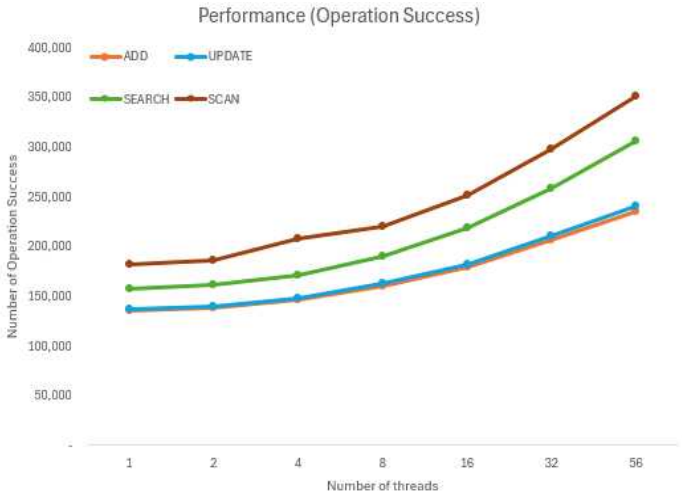

그림 4. LB+tree 스레드 개수 별 Basic Operation 성능

참고문헌

[1] McKenney P. E., RCU (read-copy update), Linux Journal, vol. 2001, no. 93, pp. 5, 2001.

[2] Matveev A., Shavit N., Felber P., Marlier P., Read-Log-Update: A lightweight synchronization mechanism for concurrent programming, Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP), pp. 168-183, 2015.

[3] Lev Y., Matveev A., Shavit N., MV-RLU: Scaling read-log-update with multi-versioning, Proceedings of the USENIX Annual Technical Conference (USENIX ATC), pp. 103-115, 2017.

[4] Levandoski J. J., Lomet D. B., Sengupta S., The Bw-Tree: A B-tree for new hardware platforms, Proceedings of the IEEE 29th International Conference on Data Engineering (ICDE), vol. 2013, pp. 302-313, 2013.

[5] A. Braginsky and E. Petrank, "A Lock-Free B+Tree," in Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 58-67, 2012. doi: 10.1145/2312005.2312016