(19) 대한민국특허청(KR)
(12) 공개특허공보(A)

(11) 공개번호 10-2024-0011738
(43) 공개일자 2024년01월26일

(51) 국제특허분류(Int. Cl.)
G06F 16/901 (2019.01) *G06F 16/22* (2019.01)
(52) CPC특허분류
G06F 16/9027 (2019.01)
G06F 16/2246 (2019.01)
(21) 출원번호 10-2023-7043146
(22) 출원일자(국제) 2022년05월18일
심사청구일자 없음
(85) 번역문제출일자 2023년12월13일
(86) 국제출원번호 PCT/US2022/029722
(87) 국제공개번호 WO 2022/251009
국제공개일자 2022년12월01일
(30) 우선권주장
21175921.2 2021년05월26일
유럽특허청(EPO)(EP)

(71) 출원인
마이크로소프트 테크놀로지 라이센싱, 엘엘씨
미국 워싱턴주 (우편번호 : 98052) 레드몬드 원 마이크로소프트 웨이
(72) 발명자
드라고예비치 알렉산다르
미국 워싱턴 98052-6399 레드몬드 원 마이크로소프트 웨이 마이크로소프트 테크놀로지 라이센싱, 엘엘씨
리우 준이
미국 워싱턴 98052-6399 레드몬드 원 마이크로소프트 웨이 마이크로소프트 테크놀로지 라이센싱, 엘엘씨
(74) 대리인
김태홍, 김진희

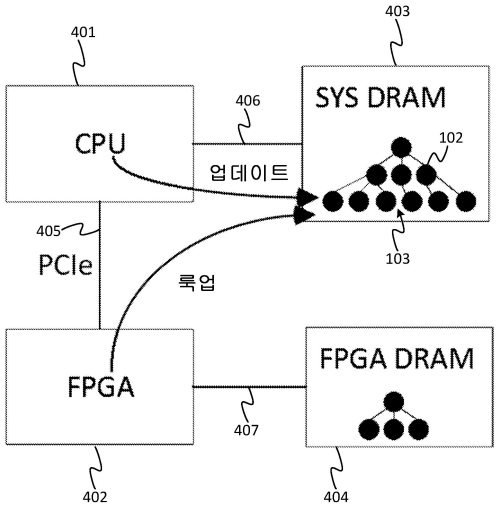
전체 청구항 수 : 총 15 항

(54) 발명의 명칭 **트리 기반 데이터 구조**

(57) 요 약

라이터는 트리의 잎 노드에 항목들을 기록하고, 리더는 잎 노드로부터 항목들을 판독한다. 각각의 노드는 개개의 제1 블록과 제2 블록을 포함하며, 제1 블록은 키 순서로 정렬된 개개의 잎의 복수의 항목들을 포함한다. 잎에 새로운 항목들을 기록할 때, 라이터는 식별된 잎 노드의 제2 블록에 새로운 항목들을, 키 순서로 정렬되지 않고, 기록되는 순서로 기록한다. 잎으로부터 하나 이상의 타깃 항목들을 판독할 때, 리더는 하나 이상의 타깃 항목들에 대해 잎을, a) 제1 블록에 이미 정렬되어 있는 항목들의 순서, 및 b) 리더가 제1 블록의 항목들과 관련하여 키에 의해 제2 블록의 항목들을 정렬하는 것에 기초하여 검색한다.

대 표 도 - 도4



명 세 서

청구범위

청구항 1

시스템으로서,

트리 구조를 포함하는 데이터 구조를 저장하는 메모리 - 상기 트리 구조는 각각이 노드 ID를 갖는 복수의 노드들을 포함하고, 일부 노드들은 잎 노드(leaf node)들이고 다른 노드들은 내부 노드(internal node)들이며, 각각의 내부 노드는 상기 트리 구조에서 하나 이상 자식들의 개개의 세트의 부모이고, 각각의 자식은 잎 노드 또는 또 다른 내부 노드이고, 각각의 잎 노드는 자식이지만 부모는 아니며, 상기 잎 노드들 각각은 키-값 쌍(key-value pair)을 각각이 포함하는 하나 이상의 항목(item)들의 개개의 세트를 포함하고, 상기 내부 노드들 각각은 그 개개의 자식들 각각의 노드 ID를 상기 개개의 자식에 의해 포함된(encompassed) 키들의 범위에 매핑함 -;

상기 잎 노드들에 항목들을 기록하도록 마련된 라이터(writer); 및

상기 잎 노드들로부터 항목들을 판독하도록 마련된 리더(reader)

를 포함하고;

상기 잎 노드들 각각은 개개의 제1 블록과 개개의 제2 블록을 포함하고, 상기 제1 블록은 상기 메모리의 주소 공간에 키 순서로 정렬된(sorted) 상기 개개의 잎 노드의 복수의 상기 항목들을 포함하고;

상기 라이터는, 상기 트리 구조에 새로운 항목들을 기록할 때, 상기 트리 구조를 통해 상기 키들의 노드 ID들에의 매핑을 따름으로써 어느 잎 노드에 기록할지 식별하고, 상기 식별된 잎 노드의 상기 제2 블록에 상기 새로운 항목을, 키 순서로 정렬되지 않고, 기록되는 순서로 기록하도록 구성되고;

상기 리더는, 상기 트리 구조로부터 하나 이상의 타깃 항목들을 판독할 때, 상기 트리 구조를 통해 상기 키들의 노드 ID들에의 매핑을 따름으로써 어느 잎 노드로부터 판독할지 결정하고, 그런 다음 상기 하나 이상의 타깃 항목들에 대해 상기 결정된 잎 노드를, a) 상기 제1 블록에 이미 정렬되어 있는 상기 항목들의 순서, 및 b) 상기 리더가 상기 제1 블록의 상기 항목들과 관련하여 키에 의해 상기 제2 블록의 항목들을 정렬하는 것에 기초하여 검색하도록 구성되는, 시스템.

청구항 2

제1 항에 있어서,

상기 라이터는, 상기 시스템의 컴퓨터 판독가능 스토리지에 저장되어 있고 상기 시스템의 하나 이상의 프로세서에서 실행되도록 마련된 소프트웨어로 구현되는, 시스템.

청구항 3

제1 항 또는 제2 항에 있어서,

상기 리더는 PGA, FPGA 또는 전용 하드웨어 회로로 구현되는, 시스템.

청구항 4

제1 항 내지 제3 항 중 어느 한 항에 있어서,

상기 트리 구조는 B-트리의 형태를 취하는, 시스템.

청구항 5

제4 항에 있어서,

상기 B-트리는 항목들이 중간 노드(intermediate node)들에는 저장되지 않는 B+ 트리인 것인, 시스템.

청구항 6

제1 항 내지 제5 항 중 어느 한 항에 있어서,

상기 잎 노드들 및/또는 내부 노드들 중 적어도 일부에서, 상기 개개의 제1 블록은 세그먼트들로 나뉘고, 상기 노드는 복수의 쇼트컷(shortcut)들을 포함하는 쇼트컷 블록을 더 포함하고 - 각각의 쇼트컷은 i) 상기 개개의 제1 블록의 해당 세그먼트의 경계에서의 키의 표시, 및 ii) 상기 잎 노드 내의 상기 해당 세그먼트에 대한 오프셋을 포함함 -;

상기 리더는, 제1 블록의 상기 검색을 수행할 때, 키에 기초하여 상기 하나 이상의 항목들을 포함할 수 있는 세그먼트들만을 검색하기 위해 상기 쇼트컷들을 사용하도록 구성되는, 시스템.

청구항 7

제1 항 내지 제6 항 중 어느 한 항에 있어서,

상기 라이터는 기록 시, 상기 잎 노드들 중 적어도 하나에 기록할 때, 각각의 항목과 연관된 백 포인터(back pointer)를 상기 개개의 제2 블록에 포함시키도록 구성되고 - 각각의 백 포인터는 상기 제2 블록에서의 상기 연관된 항목과 비교하여 다음으로 큰 또는 다음으로 작은 키를 갖는 상기 제1 블록에서의 항목에 대한 오프셋을 포함함 -;

상기 리더는, 제2 블록의 상기 정렬을 수행할 때, 상기 개개의 제1 블록과 관련하여 키에 의해 상기 제2 블록의 상기 항목들을 정렬하기 위해 상기 백 포인터들을 사용하도록 구성되는, 시스템.

청구항 8

제1 항 내지 제7 항 중 어느 한 항에 있어서,

상기 라이터는 기록 시, 상기 잎 노드들 중 적어도 하나에 기록할 때, 상기 제2 블록에서의 상기 항목들 각각에 대해, 상기 항목을 상기 제2 블록에 기록할 때 제2 블록에서의 현재 키와 관련하여 상기 항목의 키의 순서를 나타내는 순서 힌트(order hint)들을 포함시키도록 구성되고;

상기 리더는, 제2 블록의 상기 정렬을 수행할 때, 상기 개개의 제1 블록과 관련하여 키에 의해 상기 제2 블록의 상기 항목들을 정렬하기 위해 상기 순서 힌트들을 사용하도록 구성되는, 시스템.

청구항 9

제8 항에 있어서,

상기 리더는 상기 순서 힌트들을,

- 상기 개개의 제2 블록에서의 상기 항목들을, 그것들이 정렬되어 있는 순서로 처리함으로써; 그리고

- 상기 순서 힌트들을 사용하여 상기 항목들을 어레이로 정렬함으로써 - 상기 어레이는 상기 제2 블록 내에 상기 항목들의 오프셋을 저장함 -

사용하도록 구성되고;

순서 힌트 i를 갖는 각각의 항목이 처리될 때, 상기 제2 블록에서의 그 오프셋은 상기 어레이에서 포지션 i에 삽입되고, 포지션 $j \geq i$ 에서의 모든 항목들은 오른쪽으로 이동되어 상기 전체 제2 블록이 처리된 후, 상기 어레이는 상기 항목들의 오프셋을 상기 정렬된 순서로 포함하도록 되는, 시스템.

청구항 10

제1 항 내지 제9 항 중 어느 한 항에 있어서,

상기 데이터 구조는 상기 노드 ID들을 개개의 메모리 주소에 매핑하는 페이지 테이블(page table)을 포함하고;

상기 리더는 결정된 노드 ID의 노드로부터 판독할 때, 상기 페이지 테이블을 사용하여 메모리 내 상기 결정된 노드의 메모리 주소를 결정하도록 구성되고;

상기 라이터는, 노드를 기록, 분할 또는 병합하기 위해 상기 노드를 업데이트할 때, 상기 노드의 업데이트된 버전을 새로운 메모리 주소에 기록하고, 그런 다음 일단 기록되면, 상기 새로운 메모리 주소를 가리키도록 상기 페이지 테이블에서의 상기 개개의 메모리 주소를 업데이트하도록 구성되는, 시스템.

청구항 11

제1 항 내지 제10 항 중 어느 한 항에 있어서,

각각의 제2 블록은 최대 크기를 가지며;

상기 라이터는, 상기 식별된 노드의 상기 개개의 제2 블록이 상기 최대 크기를 초과하게 할 것인 새로운 항목을 기록하는 경우, 상기 제2 블록을 상기 개개의 제1 블록과 병합하도록 구성되는, 시스템.

청구항 12

제11 항에 있어서,

상기 데이터 구조는 상기 노드 ID들을 개개의 메모리 주소에 매핑하는 페이지 테이블을 포함하고;

상기 리더는, 결정된 노드 ID의 노드로부터 판독할 때, 상기 페이지 테이블을 사용하여 메모리 내 상기 결정된 노드의 상기 메모리 주소를 결정하도록 구성되고;

상기 라이터는, 상기 개개의 제2 블록과 제1 블록을 병합하기 위해 노드를 업데이트할 때, 상기 병합을 포함하는 상기 노드의 업데이트된 버전을 새로운 메모리 주소에 기록하고, 그런 다음 일단 병합이 완료되면, 상기 새로운 메모리 주소를 가리키도록 상기 페이지 테이블에서의 상기 개개의 메모리 주소를 업데이트하도록 구성되는, 시스템.

청구항 13

제1 항 내지 제12 항 중 어느 한 항에 있어서,

각각의 잎 노드는 상기 개개의 잎 노드의 키들에 이웃하는 키들의 범위를 포함하는 하나 이상의 형제 잎 노드들의 노드 ID를 추가로 표시하고;

상기 리더는, 범위 스캔(range scan)을 수행하여 키들의 범위로부터 항목들을 판독하고, 하나보다 많은 잎 노드에 걸쳐 있는 항목들의 범위로부터 판독하는 범위 스캔을 수행할 때,

- 상기 트리 구조를 통해 상기 키들의 노드 ID들에의 매핑을 따름으로써 상기 스캔된 범위에 있는 키들 중 하나를 포함하는 상기 잎 노드들 중 하나를 결정하고, 그런 다음

- 상기 잎 노드들 중 상기 하나에 표시된 바와 같은 상기 형제 잎 노드들 중 적어도 하나의 ID를 사용하여 상기 스캔된 범위에 있는 적어도 하나의 다른 키를 포함하는 적어도 하나의 다른 잎 노드를 결정하도록

구성되는, 시스템.

청구항 14

방법으로서,

트리 구조를 포함하는 데이터 구조를 메모리에 저장하는 단계 - 상기 트리 구조는 각각이 노드 ID를 갖는 복수의 노드들을 포함하고, 일부 노드들은 잎 노드들이고 다른 노드들은 내부 노드들이며, 각각의 내부 노드는 상기 트리 구조에서 하나 이상 자식들의 개개의 세트의 부모이고, 각각의 자식은 잎 노드 또는 또 다른 내부 노드이고, 각각의 잎 노드는 자식이지만 부모는 아니며, 잎 노드들 각각은 키-값 쌍을 각각 포함하는 하나 이상의 항목들의 개개의 세트를 포함하고, 내부 노드들 각각은 그 개개의 자식들 각각의 노드 ID를 상기 개개의 자식에 의해 포함된 키들의 범위에 매핑함 -;

상기 잎 노드들에 항목들을 기록하는 단계; 및

상기 잎 노드들로부터 항목들을 판독하는 단계

를 포함하고,

상기 잎 노드들 각각은 개개의 제1 블록과 개개의 제2 블록을 포함하고, 상기 제1 블록은 상기 메모리의 주소 공간에 키 순서로 정렬된 상기 개개의 잎 노드의 복수의 상기 항목들을 포함하고;

상기 기록하는 단계는, 상기 트리 구조에 새로운 항목을 기록할 때, 상기 트리 구조를 통해 상기 키들의 노드 ID들에의 매핑을 따름으로써 어느 잎 노드에 기록할지 식별하는 단계, 및 상기 식별된 잎 노드의 상기 제2 블록

에 상기 새로운 항목을, 키 순서로 정렬되지 않고, 기록되는 순서로 기록하는 단계를 포함하고;

상기 판독하는 단계는, 상기 트리 구조로부터 하나 이상의 타깃 항목들을 판독할 때, 상기 트리 구조를 통해 상기 키들의 노드 ID들에의 매핑을 따름으로써 어느 잎 노드로부터 판독할지 결정하는 단계, 및 그런 다음 상기 하나 이상의 타깃 항목들에 대해 상기 결정된 잎 노드를, a) 상기 제1 블록에 이미 정렬되어 있는 상기 항목들의 순서, 및 b) 상기 리더가 상기 제1 블록의 상기 항목들과 관련하여 키에 의해 상기 제2 블록의 상기 항목들을 정렬하는 것에 기초하여 검색하는 단계를 포함하는, 방법.

청구항 15

비일시적 컴퓨터 판독가능 매체 또는 매체들에 구현된 컴퓨터 프로그램으로서, 하나 이상의 프로세서에서 실행될 때 제14 항의 상기 기록하는 단계 및/또는 판독하는 단계를 수행하도록 구성된 코드를 포함하는, 컴퓨터 프로그램.

발명의 설명

기 술 분 야

배 경 기 술

[0001] B-트리와 같은 트리 구조(tree structure)가 키-값 쌍(key-value pair)들(각각의 쌍은 저장되는 콘텐트(content)인 값과 상기 값을 인덱싱하는 데 사용되는 키를 포함함)을 저장하기 위한 인덱스 데이터 구조(index data structure)로 사용될 수 있다. 트리의 각각의 잎 노드(leaf node)는 키-값 쌍들인 다수의 항목(item)들을 포함한다. 잎에서 트리 위로 더 올라가면, 트리의 각각의 내부 노드는 그것의 자식들(잎 또는 다른 내부 노드일 수 있음) 각각에 포함된(encompassed) 키 값들의 범위의 표시를 포함한다. 새로운 항목이 트리에 추가될 때, 라이터(writer)는 트리 구조를 사용하여 어느 잎이 새로운 항목의 키가 속하는 키들의 범위를 포함하는지 결정한다. 이 잎이 가득 차있지 않으면, 새로운 항목은 기존(existing) 잎에 추가될 수 있다. 그러나 잎이 가득 차있으면(잎 및 내부 노드는 일반적으로 바이트 단위의 최대 크기를 가짐) 잎은 분할될 것이다. 즉, 새로운 잎이 생성된다. 이는 또한 두 잎의 키 범위를 적절하게 레퍼런싱하기 위해 이전 잎과 새로운 잎의 부모 노드(parent node)를 업데이트하는 것을 의미한다. 부모에 대한 해당 업데이트가 부모를 그것의 최대 크기를 초과하게 하는 경우, 해당 부모 자체가 분할되고 조부모(grandparent)에 대한 레퍼런스가 업데이트되는 식으로 된다. 항목들이 삭제되는 경우 이는 또한 잎들 또는 내부 노드들을 병합하는 것을 포함할 수 있다.

[0002] 주어진 애플리케이션에서 라이터 또는 리더(reader)가 수행해야 하는 다수의 작업이 있다. 리더는 특정 키를 사용하여 개별 항목을 판독할 수 있거나, 범위 스캔(range scan)을 수행하여 키들의 범위를 판독할 수 있다. 라이터는 새로운 엔트리(entry)를 기록하거나, 기존 엔트리를 수정하거나, 엔트리를 삭제할 수 있다.

[0003] 인덱스 데이터 구조는 광범위한 소프트웨어 시스템, 예를 들어 데이터 저장소(store) 및 데이터베이스, 파일 시스템, 운영 체제 등에서 사용된다. 인덱스 데이터 구조는 여러가지 작업들 중에서도, 키-값 쌍들을 저장하고 키-값 쌍들의 룩업(lookup), 스캔(scan), 삽입(insert) 및 삭제(delete)를 지원한다. B-트리는 하나의 그러한 인덱스 데이터 구조이다. B-트리는 인오더(in-order) 인덱스이고 이는 스캔 작업도 지원한다는 것을 의미한다. 스캔 작업은 지정된 범위의 키들로 트리에 저장된 모든 키-값 쌍들을 리턴한다.

발명의 내용

해결하려는 과제

[0004] B-트리와 같은 트리 기반 데이터 구조에서는 라이터와 리더 간의 작업량을 밸런싱하는 것이 바람직하다. 예를 들어, 새로운 엔트리들이 단순히 시간순(chronological order)으로 추가되고 키에 의해 정렬되지 않는 경우, 이것은 라이터에게는 매우 빠른 작업이다. 그러나 리더는 그러면 개별 판독 또는 범위 스캔을 수행하기 위해 판독-시간에 항목들을 정렬해야 한다(적어도 범위 스캔은 정렬(sorting)을 필요로 한다 - 개별 판독은 가능할 수도 있지만, 정렬을 사용하여 룩업과 범위 스캔 모두 동일한 방식으로 구현하는 것이 더 간단함). 반면, 라이터가 기록을 수행할 때마다 모든 새로운 엔트리들을 정렬된 순서로 배치하면, 이것은 리더가 판독하는 것을 매우 빠르게 할 것이지만, 기록-시간에 라이터에게 더 큰 부담을 줄 것이다. 이 두 가지 접근방식 사이에 절충안을 제

공하는 것이 바람직하다.

과제의 해결 수단

[0005] 본 명세서에 개시된 일 양태에 따르면, 데이터 구조를 저장하는 메모리를 포함하는 시스템이 제공되고, 데이터 구조는 각각 노드 ID를 갖는 복수의 노드들을 포함하는 트리 구조를 포함한다. 일부 노드들은 잎 노드들이고 다른 노드들은 내부 노드들이며, 각각의 내부 노드는 트리 구조에서 하나 이상 자식들의 개개의 세트에 대한 부모이다. 각각의 자식은 잎 노드이거나 또 다른 내부 노드이며, 각각의 잎 노드는 자식이지만 부모는 아니다. 잎 노드들 각각은 키-값 쌍을 각각 포함하는 하나 이상 항목들의 개개의 세트를 포함하며, 내부 노드들 각각은 그것의 개개의 자식들 각각의 노드 ID를 개개의 자식에 포함된 키들의 범위에 매핑한다. 시스템은 잎 노드들에 항목들을 기록하도록 마련된(arranged) 소프트웨어, 하드웨어 또는 이들의 조합으로 구현된 라이터; 및 잎 노드들로부터 항목들을 판독하도록 마련된 소프트웨어, 하드웨어 또는 이들의 조합으로 구현된 리더를 더 포함한다. 잎 노드들 각각은 개개의 제1 블록과 개개의 제2 블록을 포함하고, 제1 블록은 메모리의 주소 공간에 키 순서로 정렬된 개개의 잎 노드의 복수의 항목들을 포함한다. 라이터는 트리 구조에 새로운 항목들을 기록할 때, 트리 구조를 통해 키들의 노드 ID들에의 매핑에 따라 어느 잎 노드에 기록할지 식별하고, 식별된 잎 노드의 제2 블록에 새로운 항목들을 키 순서로 정렬되지 않고(rather than), 기록되는 순서로 기록하도록 구성된다. 리더는 트리 구조로부터 하나 이상의 타깃 항목들을 판독할 때, 트리 구조를 통해 키들의 노드 ID들에의 매핑에 따라 어느 잎 노드로부터 판독할지 결정하고, 그런 다음 하나 이상의 타깃 항목들에 대해 상기 결정된 잎 노드를, a) 제1 블록에 이미 정렬되어 있는 항목들의 순서, 및 b) 리더가 제1 블록의 항목들과 관련하여(relative to) 키에 의해 제2 블록의 항목들을 정렬하는 것에 기초하여 검색하도록 구성된다.

[0006] 실시예에서, 제1 블록은 제2 블록보다 더 큰 최대 크기를 가질 수 있고, 따라서 제1 블록은 "대형 블록"으로, 제2 블록은 "소형 블록"으로 지칭될 수 있다. 실시예에서 라이터는 소프트웨어로 구현될 수 있지만 리더는 커스텀(custom) 하드웨어로, 예를 들어 프로그래머블 게이트 어레이(programmable gate array, PGA) 또는 필드 프로그래머블 게이트 어레이(field programmable gate array, FPGA)로 구현될 수 있다. 그러나 보다 일반적으로 라이터는 하드웨어(예를 들어, 하드웨어 가속기)로 구현될 수도 있으며/있고 리더는 소프트웨어로 구현될 수도 있다.

도면의 간단한 설명

[0007] 본 명세서에 개시된 실시예의 이해를 돕고 그러한 실시예가 어떻게 실행될 수 있는지 설명하기 위해, 단지 예로서 다음 도면들을 참조한다:

도 1은 예시적인 키 구간 [14, 22]에 대한 포괄적 스캔 형태의 범위 스캔을 개략적으로 예시하고, 이에 의해 리턴되는 항목들의 키에 의해 형성된 구간은 입력 구간 내에 속하는 가장 큰 서브세트(subset)이다.

도 2는 예시적인 키 구간 [14, 22]에 대한 커버링 스캔(covering scan) 형태의 범위 스캔을 개략적으로 예시하고, 이에 의해 리턴되는 항목들의 키에 의해 형성된 구간은 입력 구간을 포함하는 가장 작은 슈퍼세트(superset)이다.

도 3은 예시적인 키 구간(14, 22)에 대한 커버링 스캔 형태의 범위 스캔을 개략적으로 예시한다.

도 4는 본 명세서에 개시된 실시예에 따른 시스템의 개략적인 블록도이다.

도 5는 트리 구조를 포함하는 데이터 구조를 개략적으로 예시한다.

도 6은 소형 블록과 대형 블록을 포함하는 노드를 개략적으로 예시한다.

도 7은 소형 블록에 새로운 항목을 삽입하는 방법을 개략적으로 예시한다.

도 8은 소형 블록을 정렬하기 위해 순서 힌트를 포함시키는 방법을 개략적으로 예시한다.

도 9는 순서 힌트에 기초하여 소형 블록을 정렬하는 방법을 개략적으로 예시한다.

도 10은 4-요소 소형 블록 정렬기의 개략적인 회로도이다.

도 11은 삽입 작업 동안 대형 블록과 소형 블록을 병합하는 방법을 개략적으로 예시한다.

도 12는 삽입 동안 잎 노드를 분할하는 방법을 개략적으로 예시한다.

발명을 실시하기 위한 구체적인 내용

[0008] 본 개시는 각각의 노드가 제1 블록과 제2 블록으로 구성된 노드 레이아웃을 제공한다. 바람직하게는, 제1 블록은 제2 블록보다 큰 최대 크기(비트(bits) 또는 바이트(bytes)로)를 갖고, 따라서 제1 블록 및 제2 블록은 각각 대형 블록 및 소형 블록으로 지칭될 수 있다. 다음은 예로서 대형 블록과 소형 블록을 참조하여 설명될 것이지만, 이하의 어디든 원칙적으로 더 일반적으로 "제1 블록"과 "제2 블록"으로 각각 대체될 수 있다.

[0009] 대형 블록은 정렬되고 이는 노드를 효율적으로 검색할 수 있게 한다. 이 맥락에서 정렬된다(sorted)는 것은 항목들이 키에 의해 정렬됨을 의미하여, 키에 의한 항목들의 순서가 항목들이 메모리의 주소 공간(구현에 따라 가상 또는 물리적 주소 공간)에 나타나는 순서와 동일하게 만들어진다. 반면에 소형 블록은 노드에 대한 최신 삽입 및 삭제를 포함하며 시간순으로 있으며, 이는 신속한 업데이트를 가능하게 한다. 선택가능한 것은 전체 노드를 정렬되지 않은 상태로 유지하여 업데이트 성능은 빠르지만 검색 성능은 저조하게 되는 것; 또는 전체 노드가 정렬되어 검색 성능은 빠르지만 업데이트 성능은 저조하게 되는 것이다.

[0010] 선택적으로 쇼트컷 키(shortcut key)가 대형 블록에서 사용될 수 있다. 이것은 전체 노드 중 단지 작은 부분을 검사하기 위한 노드 검색을 허용하여 성능이 향상된다.

[0011] 또 다른 대안적인 또는 추가적인 최적화로서, 실시예에서 "백 포인터(back pointer)들"이 소형 블록에 포함될 수 있다. 이것들은 키들을 비교하지 않고 소형 블록과 대형 블록에 있는 항목들 간에 순서를 확립할 수 있게 한다. 이는 전체 키들을 비교할 필요가 없기 때문에 효율적인 구현(예를 들어, 하드웨어로)이 가능해진다.

[0012] 쇼트컷 키 및/또는 백 포인터와 함께 또는 독립적으로 사용될 수 있는 또 다른 선택적인 최적화로서, 실시예에서 "순서 힌트(order hint)들"이 소형 블록에 포함될 수 있다. 소형 블록에서의 순서 힌트들은 키들을 비교하지 않고 소형 블록에 서 항목들의 정렬되는 순서를 확립하기 위한 매우 효율적인 하드웨어를 가능하게 한다.

[0013] 또 다른 선택적인 최적화는 복잡한 업데이트의 동기화(synchronization)를 제공하는 것이다. 이 경우 노드들이 분할되거나 병합될 때, 라이터는 서브-트리의 복사본(예를 들어, 스레드-프라이빗(thread-private) 복사본)을 생성하고 그런 다음 그것을 페이지 테이블(page table)의 단일 포인터 업데이트로 스와핑(swapping)함으로써 트리를 업데이트한다.

[0014] 실시예에서, 라이터는 하나 이상의 프로세서에서 실행되는 소프트웨어로 구현될 수 있지만, 리더는 특수 하드웨어로 구현될 수 있다. 상기 하드웨어는 FPGA, PGA 또는 심지어 ASIC(애플리케이션 특정 집적 회로(application specific integrated circuit))과 같은 전용(즉, 고정-기능) 회로를 포함할 수 있다.

[0015] 인덱스 데이터 구조를 구현하는 특수 하드웨어는 데이터 구조 작업을 실행하기 위한 전용 파이프라인(pipeline)을 갖는 하드웨어이다. 이것은 명령어 세트를 구현하고 해당 명령어 세트를 사용하여 데이터 구조 작업을 구현하는 프로그램을 실행하는 범용 CPU와 대조된다. 특수 하드웨어는 몇 가지 이점을 갖는다: 와트(watt)당 처리량이 더 높고, 예측 가능성이 더 높고 레이턴시(latency)가 더 짧으며, CPU에서 소프트웨어로 실행되는 동일한 기능성보다 저렴할 수 있다. 특수 하드웨어를 사용하는 경우의 주된 단점은 설계하기 더 어렵고, 소프트웨어보다 구축하는 데 더 많은 시간이 소요되며, 전용 하드웨어의 경우 구축 후에는 변경할 수 없으므로 일반적으로 매우 광범위하게 사용되는 기능성을 위해서만 사용된다.

[0016] 그러나 본 개시의 범위는 소프트웨어에서의 기록 및 하드웨어에서의 판독 구현으로 제한되지 않는다는 점에 유의한다. 다른 가능성으로, 라이터는 FPGA나 ASIC과 같은 일부 하드웨어 형태로 구현될 수 있고/있거나 리더는 소프트웨어로 구현될 수 있다. 본 명세서에 개시된 실시예는 두 분야의 최상의 것으로부터 이익을 얻는 인메모리(in-memory) B-트리를 제공했다: 모든 작업들이 CPU 상의 소프트웨어에서 실행될 수 있는 한편, 룩업 및 스캔 작업은 설계된 특수 하드웨어에서 실행될 수 있다. 많은 작업량에서, 룩업 및 스캔이 지배적인 작업이고, 이는 대부분의 작업이 하드웨어 오프로드(offload)로부터 이익을 얻을 수 있음을 의미한다. 룩업 및 스캔은 업데이트 작업보다 간단하여, 전체 시스템이 더 빠르게 구축되고 사용될 수 있음을 의미한다. 하나의 예시적인 구현에서 메모리는 DRAM 기반 호스트 메모리일 수 있고(그러나 이는 제한되지 않음), 다른 예에서 B-트리는 예를 들어, SSD 또는 3D-Xpoint 드라이브와 같은 스토리지 NVM 디바이스에 저장될 수 있다.

[0017] 위에 언급된 기술의 예들이 이제 도면을 참조하여 더 상세히 설명될 것이다.

[0018] 다음은 B-트리의 관점에서 설명될 것이지만, 보다 일반적으로 다양한 타입의 판독 및 기록 작업, 대형 및 소형 블록들의 사용, 쇼트 컷 키, 백 포인터, 순서 힌트, 및/또는 복잡한 업데이트(분할 및 병합)의 동기화와 같은

본 명세서에 설명된 아이디어들은 키-값 쌍들을 저장하기 위한 임의의 트리 구조에 적용될 수 있다.

[0019] **트리 구조(TREE STRUCTURE)**

[0020] B-트리와 같은 키-값 저장소(store)는 데이터를 키-값 쌍(*항목*이라고도 함)으로 저장한다. 항목은 가변 크기일 수 있지만, 구현에 따라 데이터 구조는, 예를 들어 512 바이트(bytes)까지와 같이, 항목의 총 허용 크기에 제한을 둘 수 있거나, 일부 구현에서는 구현을 단순화하기 위해 그것들을 고정된 크기로 만들 수도 있다. 각각의 키는 단일의 개개의 값과 연관되어 있고; 동일한 키를 갖는 다수의 키-값 쌍들은 허용되지 않는다.

[0021] 도 1, 도 4 및 도 5는 예로서 B-트리와 같은 트리 구조(103)의 요소들을 예시한다. 트리(103)는 키-값 저장소로 사용될 수 있는 메모리에 구현된 데이터 구조의 한 형태이다. 트리(103)는 복수의 노드들(102)을 포함하며, 그것들 중 일부는 잎 노드들(102L)이고 다른 것들은 내부 노드(102I)이다. 각각의 노드(102)는 개개의 노드 ID를 가지며, 이는 본 명세서에서 논리(logical) ID(LID)라고도 지칭될 수 있다. 각각의 잎 노드(102L)는 하나 이상 항목들(101)의 개개의 세트를 포함하며, 각각의 항목(101)은 키-값 쌍을 포함한다(예시로서 도 1은 범위 스캔을 경험하는 노드(102)를 도시하지만, 이것은 제한적인 것이 아니며 판독 작업의 한 타입에 대한 단지 하나의 예이다). 키-값 쌍은 항목의 인덱스 역할을 하고 항목의 콘텐트인 값에 매핑되는 키를 포함한다. 도면에서 항목들(101) 내부에 제시된 숫자들은 예시적인 키이다(실제로 키는 훨씬 더 큰 숫자로 실행될 수 있지만 이것은 단지 예시일 뿐이다). 각각의 내부 노드(102I)는 내부 노드의 하나 이상의 자식들의 개개의 세트를 지정한다. 일단 트리가 구축되면, 내부 노드들 중 적어도 하나 또는 일부는 각각 하나보다 많은 자식을 가질 것이다(그렇지만 트리에 하나의 잎만 있는 경우, 예를 들어 방금 생성된 경우, 뿌리 노드(root node)는 하나의 자식만 가짐). 각각의 내부 노드(102I)는 또한 어느 키들의 범위가 그것의 개개의 자식들 각각에 포함되는지를 표시한다. 나중에 설명될, 포인터들 사이에 키들이 끼워 넣어진(sandwiched) 도 5에 도시된 특정 구성은 이 매핑을 지정하는 단지 하나의 가능한 예일 뿐이다.

[0022] 매핑이 구현되는 수단에 관계없이, 내부 노드들(102I)에 있는 노드 ID들은 따라서 부모 노드와 자식 노드 사이의 에지(edge)들을 정의하고, 이에 의해 트리 구조를 형성한다. 각각의 내부 노드(102I)는 적어도 하나의 개개의 자식의 부모이다. 일단 트리가 구축되면, 내부 노드들 중 적어도 하나 또는 일부는 각각 다수의 개개의 자식들의 부모이다(그렇지만 위에서 언급된 것처럼, 트리에 잎이 하나만 있는 경우 뿌리는 단일 자식을 가질 것임). 주어진 부모의 자식은 잎 노드(102L) 또는 또 다른 내부 노드(102I)일 수 있다. 잎 노드들(102L)은 부모가 아닌 자식일 뿐이다(즉, 잎들은 트리의 바텀(bottom)이거나, 다른 말로 하면 트리 가지들의 말단(terminus)임).

[0023] 본 명세서에서 키가 자식에 "포함(encompassed)"된다고 하는 것은 다음 중 하나를 의미한다: 자식이 해당 키를 갖는 항목(키-값 쌍)을 포함하는 잎 노드(102)임; 또는 자식이, 그것dl 노드 ID들을 키들에 매핑하는 것을 통해, 트리 계층에서 하나 이상의 레벨(또는 "세대") 아래에 해당 키를 갖는 항목을 포함하는 잎 노드(102I)로 궁극적으로 이어지는 또 다른 내부 노드(102I)임.

[0024] 트리 상단의 내부 노드들(102I) 중 하나는 뿌리 노드이다. 라이터 또는 리더가 특정 키를 사용하여 항목에 기록하거나 이로부터 판독하려면, 라이터 또는 리더는 뿌리의 어느 자식이 필요한 키를 포함하는지 발견하기 위해 뿌리 노드에 지정된 노드 ID들의 키 범위들에의 매핑을 쿼리함으로써(querying) 시작한다. 해당 자식 자체가 또 다른 내부 노드(102I)이면, 라이터 또는 리더는 다음 세대의 어느 자식이 필요한 키를 포함하는지 발견하기 위해 해당 노드에 지정된 매핑을, 필요한 키를 갖는 항목을 포함하는 잎 노드(102L)를 발견할 때까지 쿼리하는 식으로 한다.

[0025] 실시예에서 키 비교는 C memcmp 함수의 의미론(semantics)을 따를 수 있으며, 이는 정수 비교 의미론을 유지하기 위해 정수 키들이 빅 엔디안(big-endian) 포맷으로 저장되어야 함을 의미한다. 그러나 이것은 단지 하나의 가능한 예일 뿐이다.

[0026] 실시예에서 B-트리는 잎 노드들(102L)만이 키-값 쌍들을 포함하고 내부 노드들(102I)은 노드-ID 대 키-범위 매핑만을 포함하는 B+ 트리의 형태를 취할 수 있다.

[0027] **판독/기록 작업**

[0028] B-트리와 같은 트리 구조는 개별 룩업 및/또는 범위 스캔과 같은 판독 작업 타입을 지원할 수 있다. 그것은 삽입, 수정 및/또는 삭제와 같은 기록(업데이트) 작업 타입을 지원한다. 그러한 작업들의 예시적인 의미론이 하기에 설명된다.

[0029] **룩업(Lookup):** 이것은 키를 인수(argument)로 취하고, 트리에 있는 경우 키와 연관된 값을 리턴한다. 그렇지 않

으면 "찾을 수 없음(not-found)" 상태를 리턴한다.

[0030] **스캔(Scan):** 이것은 두 개의 키를 입력 인수로 취하고 폐구간 [*로우(low)-키, 하이(high)-키*]에 있는 키들을 갖는 트리로부터의 모든 키-값 쌍들을 리턴한다. 실시예에서는 두 가지 타입의 스캔, 즉 *포괄적 스캔(inclusive scan)*과 *커버링 스캔(covering scan)*이 지원된다. 도 1, 도 2 및 도 3은 차이점을 예시한다. (l,u)는 하한 l과 상한 u가 트리에 있어도 결과에 포함되지 않음을 의미한다. [l, u]는 하한 l과 상한 u가 트리에 있으면 포함됨을 의미한다. l은 포함되지만 u는 포함되지 않는 [l, u)와, l은 포함되지 않지만 u는 포함되는 (l, u]의 조합도 가능하다.

[0031] 포괄적 스캔(도 1)에 있어서, 리턴된 항목들의 키들로 형성된 구간은 입력 구간 내에 속하는 가장 큰 서브세트이다. "36 내지 80의 키를 갖는 모든 항목들을 가져오기"와 같은 데이터베이스 쿼리를 제공할 때 포괄적 스캔이 사용될 수 있다. 키가 36인 항목이 없더라도 결과는 키가 35인 항목을 포함하지 않는다. 커버링 스캔(도 2 및 도 3)에 있어서, 리턴된 항목들의 키에 의해 형성된 구간은 입력 구간을 포함하는 가장 작은 슈퍼세트이다. 도 2의 경우, 키 22를 갖는 항목이 리턴되지만 도 3에서는 리턴되지 않는다. 커버링 스캔은 "파일의 시작 부분에서 오프셋(offset) 범위 550 - 800에 데이터를 저장하는 디스크 블록들을 가져오기"와 같은 스토리지 메타데이터(meta-data)를 쿼리할 때 유용하다. 결과는 550 이전의 가장 높은 키와 800 이후의 가장 작은 키(즉, 지정된 범위의 바로 양측에 있는(either side) 키)를 포함하고, 이는 요청된 범위의 데이터를 저장하는 모든 디스크 블록들이 리턴됨을 의미한다.

[0032] **삽입(Insert):** 이것은 삽입할 새로운 키-값 쌍을 취한다. 지정된 키를 갖는 항목이 트리에 존재하지 않으면, 키-값 쌍이 트리에 삽입된다. 그렇지 않으면, 작업은 트리를 업데이트하지 않고 "이미 존재함" 상태를 리턴한다.

[0033] **수정(Modify):** 이것은 업데이트할 기존의 키-값 쌍을 취한다. 지정된 키를 갖는 항목이 트리에 있으면, 값이 입력 인수의 값으로 업데이트된다. 값의 크기는 이전과 상이할 수 있음에 유의한다. 그렇지 않으면, 작업은 트리를 업데이트하지 않고 "찾을 수 없음" 상태를 리턴한다.

[0034] **삭제(Delete):** 이것은 제거할 항목의 키를 취한다. 항목이 트리에 있으면 그것은 제거된다. 그렇지 않으면, 작업은 "찾을 수 없음" 상태를 리턴한다.

[0035] **시스템 아키텍처(SYSTEM ARCHITECTURE)**

[0036] 도 4는 실시예에 따른 예시적인 시스템을 도시한다. 시스템은 라이터(401), 리더(402), 시스템 메모리(403) 및 선택적으로 리더(402)의 로컬 메모리(404)를 포함한다. 트리 구조(103)는 시스템 메모리(403)에 저장된다. 라이터(401)와 리더(402)는 시스템 메모리(403)에 동작 가능하게(operatively) 결합된다. 예를 들어, 리더(402)는 PCIe 버스와 같은 버스(405)를 통해 라이터(401)에 동작 가능하게 결합될 수 있으며; 반면 라이터(401)는 버스를 통한 통신을 필요로 하지 않는 별도의 로컬 또는 전용 연결(406)을 통해 시스템 메모리(403)에 동작 가능하게 결합될 수 있다. 라이터(401)는 로컬 연결(406)을 통해 트리(103)에 기록하기 위해 시스템 메모리(403)에 액세스하도록 마련될 수 있지만, 리더(402)가 트리로부터 판독하기 위해서는 버스(405)를 통해 메모리(403)에 액세스해야 할 수 있다. 리더(402)는 그 자신의 로컬 연결(407)을 통해 그것의 로컬 메모리(404)에 연결될 수 있으며, 이에 의해 리더의 로컬 메모리(404)에 트리(103)의 일 부분을 캐싱하도록(cache) 마련될 수 있다.

[0037] 메모리(403)는 임의의 적절한 형태를 취할 수 있으며 하나 이상의 메모리 유닛에 구현된 하나 이상의 메모리 매체를 포함할 수 있다. 예를 들어. 메모리 매체는 SRAM(static random access memory) 또는 DRAM(dynamic RAM), EEPROM(electrically erasable and programmable ROM), 플래시 메모리 등과 같은 전자 실리콘 기반 매체; 또는 자기 디스크 또는 테이프와 같은 자기 매체; 또는 재기록 가능한 광학 매체 또는 합성 생물학적 스토리지(synthetic biological storage)와 같은 더욱 색다른 형태; 또는 임의의 이들 및/또는 다른 타입들의 임의의 조합을 포함할 수 있다. 메모리(403)가 구현되는 메모리 유닛(들)은 라이터(401)와 동일한 칩 상의 온칩(on-chip) 메모리 유닛, 또는 라이터(401)와 동일한 보드 상의 별도의 유닛, 또는 라이터(401)와 동일한 하우징(housing)이나 랙(rack)에 구현될 수 있거나 별도의 하우징이나 랙의 상이한 위치에 구현될 수 있는 SSD(solid state drive) 또는 하드 드라이브와 같은 외부 유닛 등; 또는 이들의 임의의 조합을 포함할 수 있을 것이다. 예시된 바와 같이, 하나의 특정 실시예에서, 메모리(403)는 라이터(401)와 동일한 보드 또는 칩 상에 구현될 수 있는 DRAM을 포함한다. 예를 들어, 하나의 특정 예에서, 시스템은 CPU가 상주하는 보드에 꽂혀 있는 DRAM 보드를 사용한다. 그러나 이것은 제한적이지 않다.

[0038] 리더의 로컬 메모리(404)는 사용되는 경우 임의의 하나 이상의 유닛에 임의의 적절한 매체를 포함하는 임의의 적절한 형태를 취할 수도 있다. 예시된 바와 같이, 하나의 특정 실시예에서, 리더의 로컬 메모리(404)는 리더

(402)에 통합될 수 있거나 리더(402)와 동일한 칩 또는 보드에 구현될 수 있는 DRAM을 포함한다.

[0039] 실시예에서, 라이터(401)는 시스템의 비일시적 컴퓨터 판독가능 스토리지에 저장된 소프트웨어로 구현되고 시스템의 하나 이상의 프로세서, 예를 들어 CPU에서 실행되도록 마련되고; 반면 리더(402)는 FPGA, PGA 또는 심지어 전용(고정-기능) 하드웨어 회로, 또는 조합의 형태를 취할 수 있는 하드웨어로 구현된다. 다음은 그러한 실시예의 관점에서 설명될 수 있지만, 하기에 설명된 기술 중 임의의 기술에 대해 라이터(401)가 하드웨어로 및/또는 리더(402)가 소프트웨어로 대신 구현될 수 있다는 것이 배제되지 않는다는 것이 이해될 것이다.

[0040] 라이터(401)(및/또는 이것이 소프트웨어로 구현되는 대체 실시예에서는 리더)의 소프트웨어를 저장하는 데 사용되는 스토리지는 위에서 설명된 것과 같은 임의의 적절한 형태를 취할 수 있거나, 대안적으로 전자 실리콘 기반 ROM 또는 CD ROM과 같은 광학 매체 등과 같은 판독 전용 메모리(read only memory, ROM)를 취할 수 있다. 하나 이상의 프로세서는 예를 들어, 범용 CPU(central processing unit); 또는 GPU(graphics processing unit), DSP(digital signal processor), 암호 프로세서(cryptoprocessor) 또는 AI 가속기 프로세서 등과 같은 애플리케이션 특정 또는 가속기 프로세서; 또는 이들 중 임의의 것 및/또는 다른 종류의 임의의 조합을 포함할 수 있다.

[0041] 예시된 바와 같이 하나의 특정 실시예에서, 라이터(401)는 CPU 상에서 실행되는 소프트웨어로 구현되고 리더(402)는 FPGA로 구현된다. 하기의 실시예들은 그러한 예의 관점에서 설명될 수 있지만, 이것은 제한적이지 않다는 것이 이해될 것이다.

[0042] B-트리는 시스템 메모리(403)에 데이터를 저장한다. 실시예에서 이는 시스템 메모리(일반적으로 수백 GB)만큼 큰 트리를 저장할 수 있게 하지만 리더(402)(예를 들어, FPGA)가 PCIe(405)를 통해 시스템 메모리에 액세스하는 비용이 발생한다. 리더(예를 들어, FPGA)(402)는 실행 속도를 높이기 위해 그것의 로컬 메모리(예를 들어, DRAM)(404)에 트리의 상위 레벨의 캐시(cache)를 유지할 수 있다. 하나의 예시적인 구현에서, 리더(402)는 그것의 로컬 레지스터 파일에 뿌리 노드의 LID와 트리 레벨들의 수를 기록한다.

[0043] B-트리는 더 큰 트리를 저장할 수 있도록 보조 스토리지에도 데이터를 저장할 수 있도록 설계될 수 있다. 매우 낮은 레이턴시(low-latency) NVMe(Non-Volatile Memory Express) 디바이스가 등장하면서 낮은 레이턴시 시나리오에서도 이것이 매력적인 옵션이 되었다.

[0044] 실시예에서, FPGA(402)는 룩업(lookup) 및 스캔(scan) 작업만을 수행하고 업데이트 작업은 CPU(401)에 의해 수행된다. CPU와 FPGA 간의 동기화는 경량인 것이 바람직하다. B-트리 노드들이 분할되거나 병합될 때만 PCIe 버스(405)를 통한 통신이 필요할 수 있다(더 상세한 내용은 데이터 구조 섹션 참조). FPGA에서 판독 작업만 지원하는 것은 판독-중심(dominated) 사용 사례들이 모든 작업을 하드웨어로 구현하지 않고 하드웨어 오프로드의 이익을 얻을 수 있게 한다.

[0045] **데이터 구조(DATA STRUCTURE)**

[0046] 실시예에서 B-트리(103)는 B+ 트리이며, 이는 키-값 쌍들(101)이 잎들(102L)에만 저장되고, 한편 내부 노드들(102I)은 키 표시자들 및 자식 노드들에 대한 포인터들을 저장한다는 것을 의미한다.

[0047] 도 5는 2-레벨 트리의 예를 도시하지만, 여기에 설명된 원리들은 임의의 레벨들의 수(즉, 부모와 자식의 임의의 레벨들의 수)를 갖는 트리에 적용될 수 있다.

[0048] 트리(103)를 포함하는 데이터 구조는 페이지 테이블(501)을 더 포함한다. 페이지 테이블(501)은 노드 ID들(논리 ID들, LID들이라고도 함)을 개개의 노드들(102)이 시스템 메모리(403)에 저장되는 실제 메모리 주소들에 매핑한다. 이러한 방식으로 라이터(401) 또는 리더(402)는 페이지 테이블(501)에서 노드 ID에 기초하여 주소를 룩업함으로써, 그것의 노드 ID(LID)에 기초하여 메모리(403)에서 노드(102)를 찾을 수(locate) 있다.

[0049] 언급된 바와 같이, 각각의 내부 노드(102I)는 그것의 자식 노드 ID들 각각을 개개의 자식에 포함된 키들의 범위의 표시에 매핑하는 정보를 포함한다. 도 5는 이것이 어떻게 구현될 수 있는지에 대한 하나의 예를 예시한다.

[0050] 이 예에서, 각각의 내부 노드(102I)는 노드 포인터들(502) 및 키 표시자들(503)의 세트를 포함한다. 각각의 노드 포인터(502)는 노드 ID(논리 ID, LID)의 관점에서 지정된 자식 노드에 대한 포인터이다. 이 예에서 각각의 키 표시자(503)는 자식에 포함된 범위의 경계에서의 키들 중 하나를 지정한다. 각각의 내부 노드(102I)에서, 포인터 왼쪽에 키 l의 표시자(503)가 있고 그것의 오른쪽에 키 u가 있는 포인터(502)는 구간 [l, u)에 키들을 저장하는 서브트리를 가리킨다. 또한, 각각의 내부 노드(102I)는 내부 노드에서 가장 낮은 키보다 더 낮은 키들을 갖는 서브트리에 대한 포인터(502)를 저장한다. 이 가장 왼쪽 포인터는 노드의 헤더(603)에 저장된다(예를

들어, 도 6 참조). 즉, 각각의 키 표시자는 한 쌍의 포인터들 사이에 끼워 넣어져 있으며, 키 표시자는 키 표시자의 양측에 있는 포인터들에 의해 가리켜진 자식들에 포함된 범위들 사이의 경계를 지정한다. 다르게 말하면, 내부 노드들은 가장 왼쪽 자식 LID를 저장한다. 그것들은 기본적으로 X 개 키와 X+1 개 포인터를 저장한다. 가장 왼쪽은 +1 포인터이다. 도 5에 도시된 특정 예에서, 구간 [22, 65)에 키가 있는 새로운 항목은 중간 잎에 삽입될 것이고; 22 미만의 키들을 갖는 항목들은 가장 왼쪽 잎에 삽입될 것이고; 65 보다 크거나 같은 키들을 갖는 항목들은 가장 오른쪽 잎에 삽입될 것이다.

[0051] 자식들을 키 범위에 매핑하기 위한 도 5에 도시된 구성은 단지 하나의 예일 뿐이라는 점이 이해될 것이다. 또 다른 예는 모든 경계 키들을 먼저 노드에 저장하고 그런 다음 그것들 후에 모든 LID들이 따른다.

[0052] 일부 실시예에서, 각각의 잎(102L)은 또한 범위 스캔의 구현을 단순화하기 위해 그것의 왼쪽에 있는 잎과 그것의 오른쪽에 있는 잎에 대한 포인터를 저장할 수 있다.

[0053] B-트리(103)는 (임의의 다른 B-트리와 마찬가지로) 바텀으로부터 성장한다. 삽입하는 동안 새로운 항목(101)이 그것이 매핑되는 잎(102)에 맞지 않으면 잎이 두 개로 분할된다. 이전 잎들 항목들은 반으로 분할되고 새로운 잎에 대한 포인터가 부모 노드에 삽입된다. 부모 노드가 가득 차면, 그것도 분할된다. 분할은 트리의 뿌리까지 반복하여(recursively) 계속될 수 있다. 뿌리가 분할되면, 새로운 뿌리가 생성되고 이는 트리의 높이를 증가시킨다. 모든 항목들이 삭제되어 잎이 비어 있게 되면, 잎은 부모로부터 삭제된다. 대안적으로, 또 다른 옵션은 B-트리의 공간 불변(space invariant)을 유지하기 위해 잎들이 절반 미만으로 채워지면 잎들을 병합하는 것이다.

[0054] **노드 레이아웃(NODE LAYOUT)**

[0055] 실시예에서, B-트리 노드들(102)은 DRAM 페이지 크기에 맞춰 조정(align)하기 위해 크기가 8 KB일 수 있다. 그것들은 리더(402)(예를 들어, FPGA)가 물리적 메모리 주소를 사용하여 노드들(102)에 액세스할 수 있도록 운영 체제에 의해 이동될 수 없는 고정된 메모리(pinned memory)에 할당될 수 있다. 그러나 이것이 필수적인 것은 아니다.

[0056] 내부 노드들(102I)은 바람직하게는 자식 노드들의 주소를 직접 저장하지 않는다. 대신, 그것들은 각각의 노드에 대해 논리 식별자(LID)들을 저장한다. 일 예에서 LID는 크기가 6 바이트일 수 있으며, 이는 트리의 최대 크기를 2^61 바이트로 제한한다. 노드의 LID에서 노드의 가상 및/또는 물리적 주소로의 매핑은 페이지 테이블(501)에 유지된다. 페이지 테이블(501)은 시스템 메모리(403)와 FPGA-부착 메모리(404) 모두에 저장될 수 있다. 새로운 노드의 매핑이 생성되거나 노드의 매핑이 변경될 때, 라이터(401)(예를 들어, CPU)는 시스템 메모리(403)에 있는 테이블(501)을 업데이트한다. 캐시가 리더 측에 유지되는 경우, 라이터(401)는 또한 리더의 부착된 메모리(404)에 있는 테이블의 복사본을 업데이트하도록 리더(402)(예를 들어, FPGA)에 (예를 들어, PCIe를 통해) 지시(command)를 발행한다. 대안적으로 리더는 업데이트를 위해 트리를 폴링(poll)할 수 있다.

[0057] LID를 사용하여 노드를 주소 지정(addressing)하는 것은 인다이렉션(indirection)의 레벨을 제공하여, 예를 들어 NVMe 디바이스에 노드들을 저장할 수 있게 한다. 업데이트 작업은 일부 경우에 노드들의 기록 시 복사(copy-on-write)를 수행할 수 있으므로, 트리에 대한 액세스를 동기화하는 데도 도움이 된다.

[0058] 실시예에서, 내부 노드들(102I)과 잎 노드들(102L)의 레이아웃은 동일하다. 예를 들어, 도 6을 참조한다. 노드 헤더(603)는 예를 들어 노드(102)의 처음 32 바이트에 저장될 수 있다. 헤더(603)는 노드(102)의 타입(내부 또는 잎), 노드에서 사용된 바이트 수, 락 비트(lock bit) 및 노드에 대한 액세스를 동기화하기 위한 노드 버전 번호를 지정하는 필드, 및/또는 하기에 논의된 다른 여러 필드를 포함할 수 있다. 도 5에 도시된 것과 같은 예에서, 내부 노드(102I)의 헤더(603)는 또한 가장 왼쪽 자식의 LID를 저장한다. 잎 노드(102L)의 헤더는 왼쪽 형제와 오른쪽 형제의 LID를 저장할 수 있다. 키와 값은 가변 크기일 수 있으므로 블롭(blob)으로 저장될 수 있다. 각각의 블롭은 예를 들어, 그것의 크기를 지정하는 2-바이트 헤더를 갖는다. LID는 크기가 알려져 있으므로 헤더 없이 저장될 수 있다. 값들은 순차적 액세스(sequential access) 성능을 향상시키기 위해 B-트리 노드들(102L)에 인라인(inline)으로 저장될 수 있다. 512 바이트보다 큰 값은 B-트리(103) 외부에 저장될 수 있으며, 상기 값에 대한 포인터는 잎(102L)에 저장된다.

[0059] 위에 예시된 노드 크기 등과 같은 특정 구현 파라미터들의 선택에 관계없이, 본 명세서에 개시된 실시예에 따르면 각각의 잎 노드(102L)는 2 개의 블록, 즉 *대형 블록*(601)과 *소형 블록*(602)을 포함한다. 이를 구현하기 위해 블록들 사이의 경계에 대한 포인터가 노드의 헤더(603)에 저장될 수 있다. 포인터는 예를 들어, 대형 블록(601)의 시작에 대한 오프셋으로 표현될 수 있다. 대형 블록(601)은 항목들을 정렬된 순서로 저장하고, 소형 블록

(602)은 최근 업데이트의 로그(log)를 저장한다. 소형 블록(602)이 문턱값(예를 들어, 512 바이트로 설정)보다 커지면, 그것은 대형 블록(601)과 병합된다. 노드(102L)를 대형 블록과 소형 블록으로 분할함으로써, 업데이트할 때마다 노드를 정렬하는 오버헤드 없이, 판독 작업은 대부분 정렬된 데이터에 액세스하는 것에서 이점을 얻는다.

[0060] 소형 블록(602)의 엔트리들은 새로 삽입된 항목들이거나 삭제 마커(marker)들일 수 있다. 선택적인 최적화로서, 소형 블록(602)의 각각의 엔트리는 대형 블록(601)에 있는 항목에 대한 포인터(예를 들어, 2 바이트 길이)를 추가로 저장할 수 있다. 편리한 레이블(label)로서, 이는 "백 포인터(back pointer)"라고 지칭될 수 있다(그렇지만 용어 "백(back)"은 반드시 물리적 주소 공간 등에서 임의의 특정 방향으로 제한되는 것을 의미하지는 않음). 하나의 예시적인 구현에서, 새로 삽입된 항목(101)에 대해, 상기 포인터는 소형 블록(602)에 추가된 상기 새로운 항목의 키보다 큰 키를 갖는 대형 블록(601)에 있는 제1 항목을 가리킨다. 또는 삭제 마커에 대해, 포인터는 삭제된 항목을 가리킨다. 포인터는 노드(102) 내에서 오프셋으로 표현될 수 있다. 그것은 키들을 비교하지 않고 대형 블록과 소형 블록에 있는 항목들(101) 사이의 순서를 확립하기 위해 리더(402)에 의해 사용될 수 있으며, 이는 특히 판독이 하드웨어로 구현되는 경우 유용한 최적화이다.

[0061] 또 다른 선택적 최적화로서, 소형 블록(602)에 있는 각각의 항목은 항목 삽입 시에, 정렬된 소형 블록에 그것의 인덱스를 저장하는 필드(예를 들어, 1 바이트 길이)를 포함할 수 있다. 즉, 이것은 삽입될 때의 항목 순서이다. 이것은 "순서 힌트(order hint)"라고 지칭될 수 있다. 이러한 인덱스들은 현재 정렬된 순서를 재구성하기 위한 검색 중에 소형 블록을 스캐닝하는 동안 리더(402)에 의해 "리플레이(replayed)"된다. 순서는 리더(402)의 임시 "인다이렉션(indirection)" 어레이에, 예를 들어 FPGA에 저장되고, 이는 정렬된 순서로 항목들에 액세스하는 데 사용된다. 이것은 특히 하드웨어에 구현되면, 정렬을 더욱 효율적으로 만들고 상당한 레이턴시를 가져오지는 않는다.

[0062] 도 7은 소형 블록(602)에 새로운 항목(101)을 삽입하는 단계들의 예를 도시한다. 단계 a)는 삽입 전의 노드(102)를 도시한다. 단계 b)에서, 항목(101)이 소형 블록(602)에 복사된다. 그것은 아직 리더(402)에 보이지 않는다. 단계 c)에서, 노드 헤더(603)에 표시된 데이터의 크기는 새로운 항목을 포함하도록 변경된다.

[0063] 또 다른 선택적 최적화로서, 노드에서 키 검색을 최적화하기 위해 대형 블록(601)은 대략 동일한 크기의 *세그먼트(segment)*들로 나뉘어질 수 있다. 세그먼트들의 경계에 있는 키들은 경계에 대한 포인터와 함께 *쇼트컷 블록(shortcut block)*(604)에 저장되며, 이는 노드 헤더(603) 바로 뒤에 저장될 수 있다. 키 검색은 쇼트컷 블록을 스캔하고 키를 포함할 수 있는 세그먼트들을 식별함으로써 시작된다. 검색에서는 해당 세그먼트들과 소형 블록만 검사하며, 이는 메모리로부터 판독되는 데이터의 양을 줄인다. 예를 들어, 성능 병목현상이 리더(예를 들어, FPGA)(402)와 시스템 메모리(403) 사이의 PCIe 대역폭에 있는 경우, 이러한 최적화는 성능을 크게 향상시킨다. 32-바이트 이하의 키에 대해, 검색은 8 KB 노드로부터 대략 1.5 KB의 데이터를 판독하며, 이는 전체 노드를 스캔하는 보다 간단한(straightforward) 구현에 비해 성능이 5 배 향상된다. 쇼트컷 키들은 대형 및 소형 블록(601, 602)의 병합 동안에 선택될 수 있다.

[0064] **노드 캐시(NODE CACHE)**

[0065] FPGA(402)는 그것에 직접 부착된 수 기가바이트(gigabytes)의 DRAM(404)(예를 들어, 4 GB)을 가질 수 있다. 이 메모리는 모든 사용 사례에서 전체 B-트리(103)를 저장할 만큼 충분히 크지 않을 수 있지만, 처음 몇 개 레벨들에 대해 충분히 클 수 있으므로 실시예에서는 레이턴시를 줄이고 B-트리의 처리량을 높이기 위해 캐시로 사용될 수 있다. 예를 들어, 캐시(404)는 메타데이터(meta-data)의 양을 줄이기 위해 예를 들어 8 KB 노드 입도(node granularity)로 할당될 수 있지만, 데이터는 PCIe(405)를 통해 판독되는 원하지 않는 데이터의 양을 줄이기 위해 256-바이트 청크(chunk)로 페치(fetch)될 수 있다. FPGA(402)는 캐시(404)에 있는 청크들을 추적하기 위해 각각의 캐싱된 노드(102)에 대한 32-비트 점유 비트맵(bitmap)을 유지할 수 있다. 캐시(404)와 시스템 메모리(403) 사이의 페이지 일관성은 페이지 매핑이 업데이트될 때마다 캐싱된 페이지들을 무효화함으로써 유지될 수 있다. CPU(401) 상의 소프트웨어는 페이지 매핑이 업데이트될 때 PCIe(405)를 통해 페이지 테이블 업데이트 지시를 전송할 수 있고, FPGA(402)는 점유 비트맵을 클리어(clear)함으로써 페이지를 무효화할 수 있다.

[0066] **페이지 테이블(PAGE TABLE)**

[0067] 리더(402)(예를 들어, FPGA)는 그것의 로컬 메모리(404), 예를 들어 FPGA 디바이스(402)에 부착될 수 있는 DRAM에 저장된 페이지 테이블(501)의 로컬 복사본을 유지할 수 있다. 실시예에서 테이블에서의 각각의 엔트리는 FPGA 메모리에서는 8 바이트이지만, 시스템 DRAM에서는 16 바이트이다. 이것은 실시예에서 시스템 DRAM에서의

복사본은 두 개의 주소(가상 및 물리적)를 저장하고 FPGA DRAM 메모리에서의 복사본은 하나의 주소(물리적 주소만)를 저장하기 때문이다.

[0068] 페이지 테이블(501)에서의 엔트리들은 그것들의 개개의 LID에 매핑된 노드들(102)의 메모리 주소를 저장한다. 대체로 이들은 구현에 따라 물리적 주소 또는 가상 주소일 수 있다. 실시예에서 라이터는 가상 메모리 매핑을 사용하고 라이터 측의 페이지 테이블(501)은 그것들의 개개의 ID에 매핑된 노드들의 가상 주소를 저장한다. 페이지 테이블의 복사본이 FPGA(402)에 유지되는 실시예에서, FPGA 페이지 테이블 복사본에서의 엔트리들은 그것들의 개개의 LID에 매핑된 노드들의 물리적 주소를 저장할 수 있다. 예를 들어, 8-바이트 엔트리와 4 GB의 DRAM으로, 시스템은 최대 4 TB의 트리를 지원하고, 이는 오늘날 서버의 메인 메모리 트리에 충분한 크기이다. 더 큰 트리는 리더(402)(예를 들어, FPGA)가 시스템 메모리(403) 또는 NVMe 디바이스에서의 페이지 테이블(501)에 액세스하도록 하거나, 또는 B-트리 노드 크기를 증가시킴으로써 지원될 수 있다.

[0069] 페이지 테이블(501)은 모든 가능한 실시예에서 절대적으로 필수적인 것은 아니다. 덜 선호되는 대안은 내부 노드들(102I)에서의 노드 포인터로서 메모리 주소를 직접 사용하는 것이다.

[0070] **룩업(LOOK-UP)**

[0071] 룩업은 뿌리에서 시작하고 각각의 레벨의 내부 트리 노드들(102I)을 순회하여 다음 레벨의 노드(102)에 대한 포인터를 찾는다. 각각의 룩업은 잎(102L)에서 종료된다. 잎이 키를 포함하면, 그것은 연관된 값을 리턴한다. 그렇지 않으면, 그것은 키가 트리에 저장되어 있지 않음을 나타내는 상태를 리턴한다.

[0072] 내부 노드(102I)에 액세스할 때, 룩업은 처음에, 노드 헤더(603) 및 쇼트컷 블록(604)을 포함하는 노드(102)의 제1 부분(예를 들어, 처음 512 바이트)을 페치할 수 있다. 룩업은 쇼트컷 블록(604)을 검색하여 타깃 키보다 작거나 같은 마지막 키를 찾는 것으로 시작한다. 그것은 대형 블록(601)의 세그먼트를 찾기 위해 쇼트컷 항목의 포인터를 따라간다. 검색은 메모리로부터 대형 세그먼트를 페치하고 그것을 검색하여 타깃 키보다 작거나 같은 마지막 키를 찾는다. 다음으로(또는 병행하여), 룩업은 메모리로부터 소형 블록(602)을 페치하고 타깃 키보다 작거나 같은 마지막 키를 검색한다. 백-포인터가 사용되는 경우, 룩업은 키들을 비교하지 않고 소형 블록과 대형 블록에서 발견된 항목들(101) 중 더 큰 것에 저장된 포인터를 따른다 - 소형 블록 항목의 백 포인터가 대형 블록 항목 바로 뒤를 가리키는 경우, 그것은 소형 블록의 포인터를 따르고; 그렇지 않으면, 그것은 대형 블록의 포인터를 따른다. 타깃 키가 대형 블록과 소형 블록에 있는 첫번째 키 둘 모두 보다 작은 경우, 룩업은 가장 왼쪽 포인터를 따른다.

[0073] 룩업이 잎(102I)에 도달하면, 그것은 내부 노드와 유사한 방식으로 그것을 검색한다. 주된 차이점은 소형 블록과 대형 블록에서 정확히 일치하는 것(exact match)을 찾는다는 점이고, 이는 대형 블록과 소형 블록에 있는 항목들을 정렬할 필요가 없다는 것을 의미한다.

[0074] **소형 블록 검색:** 소형 블록(602)의 검색은 항목들(101)을 검사하기 전에 정렬함으로써 시작할 수 있다. 정렬함으로써, 소형 블록의 검색은 타깃 키보다 큰 키를 갖는 첫번째 항목을 발견하자마자 중단될 수 있다. 실시예에서, 소형 블록 정렬은 키들을 비교하지 않는다. 대신 그것은 각각의 소형 블록 항목에 저장된 *순서 힌트* 필드를 사용하여 순서를 확립한다. 순서 힌트 필드는 항목 삽입 시 정렬되는 소형 블록에 항목의 인덱스를 저장한다. 현재 정렬된 순서를 재구성하기 위해 소형 블록을 스캔하는 동안 인덱스들이 "리플레이"된다. 확립된 순서는 항목들을 복사하지 않고, 임시 인다이렉션 어레이에, 예를 들어 FPGA 레지스터에 저장된다. 인다이렉션 어레이는 정렬된 순서로 항목들에 액세스하는 데 사용된다. 정렬은 특히 하드웨어에서 상당한 레이턴시를 가져오지는 않으며, 쇼트컷 블록(604)과 대형 블록(601)을 검색하는 동안 병렬로 수행될 수 있다. 소프트웨어에서, 정렬 순서는 작은 인다이렉션 어레이에 유지될 수 있다.

[0075] 도 8은 노드(102)로의 예시적인 삽입 시퀀스(sequence)를 예시한다. 각각의 항목(101)에 보여지는 숫자는 그것의 키를 나타낸다. 대형 블록(601)에서의 요소들은 명확성을 위해 생략되어 있다. 항목 위의 숫자는 그것의 순서 힌트를 나타낸다. 단계 a)에서 노드의 소형 블록(602)은 처음에는 비어 있다. 단계 b)에서, 예로서, 키 90을 갖는 항목이 삽입되고 소형 블록에서 첫 번째 항목이므로 순서 힌트 0이 할당된다. 다음 단계 c)에서, 키 60을 갖는 항목이 삽입된다. 그것은 소형 블록에서 가장 작은 항목이므로 순서 힌트 0이 할당된다. 기존 항목들에 있는 순서 힌트는 변경되지 않는다. 순서 힌트는 새로운 항목을 삽입하기 전에 확립된, 소형 블록의 정렬된 순서로부터 결정된다. 다음으로 단계 d)에서, 키 30을 갖는 항목이 삽입된다. 그것은 다시 소형 블록에서 가장 작으므로 그것의 순서 힌트도 0이다. 단계 e)에서 최종 항목의 키는 45이고 그것의 순서 힌트는 1이다.

[0076] 도 9는 소형 블록(602)을 정렬하는 것을 예시한다. 도면은 인다이렉션 어레이(901)의 상태를 보여준다. 인다이

렉션 어레이는 소형 블록에서의 항목들의 오프셋을 저장하지만, 도면에서 각각의 오프셋은 설명을 위해 그것의 키로 표현된다.

[0077] 소형 블록(602)을 정렬하기 위해, 항목들(101)은 그것들이 저장된 순서대로 처리되고 그것들의 순서 힌트가 인다이렉션 어레이에서의 항목들을 정렬하는 데 사용된다. 인다이렉션 어레이(901)는 소형 블록에서의 항목들의 오프셋을 저장한다. 순서 힌트 i를 갖는 항목이 처리될 때, 소형 블록에서의 그것의 오프셋은 인다이렉션 어레이의 포지션 i에 삽입된다. 포지션 j > i에서의 모든 항목들은 오른쪽으로 이동된다. 전체 소형 블록이 처리된 후, 인다이렉션 어레이는 정렬된 순서로 항목들의 오프셋을 포함한다. 도 9는 도 8에 도시된 소형 블록에 대한 인다이렉션 어레이 정렬을 예시한다. 하나의 구현에서 도면의 각각의 단계는 키 비교와 병렬로, FPGA에서 단일 사이클로 완료된다. 첫번째 단계 a)에서는, 항목 90이 첫번째 슬롯에 삽입된다. 다음 단계 b)에서는, 모든 항목들이 오른쪽으로 이동되고 순서 힌트 0을 갖는 항목 60이 첫번째 슬롯에 삽입된다. 다음 단계 c)에서는, 모든 항목들이 다시 오른쪽으로 이동되어 순서 힌트 0을 갖는 항목 30을 위한 공간을 만든다. 마지막 단계 d)에서는, 항목 60과 항목 90이 오른쪽으로 이동되고 순서 힌트 1을 갖는 항목 45가 인덱스 1을 갖는 슬롯에 삽입된다. 최종 상태 e)는 단계 a)-d) 이후의 인다이렉션 어레이의 상태를 보여준다.

[0078] 도 10은 인다이렉션 어레이 정렬기의 예시적인 구현을 도시한다. 4-항목 와이드 어레이(4-item wide array)가 도시된다. 정렬기는 9-비트 폭의 레지스터들을 포함한다. 레지스터에 대한 입력은 그것의 출력, 즉, 그 왼쪽으로의 레지스터의 출력이나, 또는 입력 오프셋 값이다. 레지스터 입력은 입력 인덱스를 레지스터의 (상수) 인덱스와 비교함으로써 선택된다. 그것들이 동일하면, 입력 오프셋이 선택되고; 입력 인덱스가 레지스터의 인덱스보다 작으면, 왼쪽으로의 레지스터의 출력이 선택되고; 입력 인덱스가 더 크면, 레지스터의 출력이 그 자체에 기록된다. 이 예시적인 구현은 소형 블록에 삭제 항목을 갖는 경우를 다루지 않지만, 이를 제공하기 위해 레지스터에 대한 입력은 또한 그 오른쪽으로의 레지스터의 출력을 취하도록 조정될 수 있다.

[0079] 물론 도 10에 도시된 레이아웃은 하나의 가능한 하드웨어 구현일 뿐이며, 이 분야의 기술자는 본 명세서에 개시된 내용이 주어지면 종래의 설계 도구를 사용하여 다른 것을 설계할 수 있다는 것이 이해될 것이다. 또한 리더측(401)에서 소프트웨어 구현도 가능하다.

[0080] CPU(401) 상에서의 정렬 구현은 유사한 접근방식을 따를 수 있다. 인다이렉션 어레이는 크기가 작고 LI 캐시에 편안하게 들어맞으며, 따라서 작업 속도가 빠르다. 주된 차이점은 어레이에 있는 항목들을 오른쪽으로 이동하는 데 다수의 사이클이 소요된다는 것이다.

[0081] **범위 스캔(RANGE SCAN)**

[0082] 범위 스캔 작업은 위에서 설명된 바와 같이 뿌리로부터 트리(103)를 순회하여 타깃 범위의 시작 키를 찾는다. 그런 다음 그것이 범위의 종료 키나 트리 끝에 도달할 때까지 시작 키로부터 앞으로(forward) 스캔한다.

[0083] 실시예에서, 잎들(102L) 사이를 이동하기 위해 범위 스캔은 *형제 포인터(sibling pointer)들*을 사용할 수 있다. 그러한 실시예에서, 각각의 잎(102L)은 그것의 왼쪽 및 오른쪽 형제들에 대한 포인터를 유지하여, 양방향으로 범위 스캔을 수행하는 것을 가능하게 한다. 그러나 형제 포인터들의 사용이 필수적인 것은 아니다. 형제 포인터들이 사용되지 않으면, 스캔은 대신 다음 잎을 찾기 위해 트리 위로 다시 이동한다. 대부분의 경우 그것은 첫번째 부모만 보고(look at) 다음 형제의 LID를 취할 수 있다. 그러나 부모의 끝까지 갔다면 그것의 형제를 찾기 위해 그것의 부모에게 갈 것이다. 스캔은 뿌리까지 계속 올라가야 할 수도 있다.

[0084] 범위 스캔은 키에 의해 정렬된 항목들(101)을 리턴한다. 항목들을 정렬된 상태로 유지하기 위해, 잎(102L)의 스캔은 대형 및 소형 블록(601, 602)을 함께 처리할 수 있다. 커버링 스캔에서, 스캔은 먼저 대형 및 소형 블록 모두에서 범위의 시작보다 작거나 같은 가장 높은 키를 갖는 항목을 찾는다. "표준(standard)" 포괄적 스캔은 범위의 시작보다 크거나 같은 가장 낮은 키에서 시작한다는 점을 제외하고 유사하다. 어느 쪽이든, 그런 다음 스캔은 구간의 끝을 찾고 구간에 있는 항목들을 리턴하면서 대형 및 소형 블록 모두에서의 노드를 앞으로 스캔한다. 오른쪽 경계보다 큰 키를 갖는 항목이 대형 및 소형 블록 모두에서 발견되면 스캔이 중단된다. 스캔하는 동안, 다음 작은 항목과 다음 큰 항목이 모두 구간에 있으면, 검색은 작은 항목에서의 백 포인터를 사용하여 어느 것을 다음에 리턴할지 결정할 수 있고; 작은 항목이 큰 항목을 가리키는 경우, 작은 항목은 리턴되고 검색은 다음 작은 항목으로 이동하며; 그렇지 않으면, 큰 항목이 리턴되고 검색은 다음 큰 항목으로 이동한다. 쇼트컷 키(604)를 처리하기 위해, 스캔은 현재 큰 세그먼트의 끝을 추적할 수 있다. 이 경우 다음 세그먼트 시작 부분에서 키는 쇼트컷 블록으로부터, 그리고 값은 대형 블록으로부터 검색된다. 세그먼트 중간에서, 키 및 값은 대형 블록에 있다.

[0085] **삽입(INSERT)**

[0086] 삽입 작업은 새로운 항목(101)을 삽입할 노드(120L)를 찾기 위한 트리 순회(tree traversal)로 시작된다. 순회는 룩업 동안의 순회와 유사하다. 차이점은 실시예에서 삽입 및 수정 작업의 의미론을 구현하기 위해 버전에 관계없이 모든 항목들을 판독한다는 것이다. 노드를 업데이트하기 전에, 라이터(401)는 그것을 기록 잠금(write lock)으로 잠글 수 있고, 이는 노드의 상태가 순회 동안 관찰된 상태임을 보장할 수 있다(실시예에서 메모리(403)는 다양한 타입의 잠금, 기록 잠금 및 판독 잠금을 지원할 수 있으며, 기록 잠금의 사용이 반드시 판독 잠금의 사용을 의미하는 것은 아님에 유의한다). 예를 들어 기록 잠금은 노드 헤더의 32-비트 락 워드(lock word)에 저장될 수 있고, 여기서 락 워드는 락 비트와 노드가 업데이트될 때 증가되는 31-비트 노드 시퀀스 번호(node sequence number)로 구성된다. 라이터(401)는 순회 동안과 동일하게 시퀀스 번호를 유지하면서, 락 워드에 대한 원자적(atomic) 비교-및-스왑 명령(compare-and-swap instruction)을 사용하여 락 비트를 원자적으로 설정함으로써 기록 잠금을 획득할 수 있다. 비교-및-스왑이 성공하면, 이것은 순회 이후 노드가 변경되지 않았으므로 삽입이 진행될 수 있음을 의미한다. 비교-및-스왑이 실패하면, 순회 이후 노드가 변경되었으므로 라이터는 트리 순회를 다시 수행하여 작업을 재시도한다.

[0087] **빠른-경로 삽입(fast-path insert):** 하나의 일반적인 경우에 삽입은 노드(102L)를 분할하거나 대형 및 소형 블록(601, 602)을 병합할 필요가 없다. 이 일반적인 경우의 삽입은 적절한 위치에서(in place) 수행될 수 있다(도 7). 라이터(401)는 소형 블록(602)의 끝 이후에 새로운 항목을 먼저 복사하고 그런 다음 새로운 항목(101)을 포함하도록 소형 블록의 크기를 조정함으로써 새로운 항목을 원자적으로 추가한다. 이를 위해, 라이터(401)는 기록 잠금을 획득한 후 다음의 단계들을 수행할 수 있다: 잠금을 획득한 후 항목을 소형 블록에 복사하고; 노드의 크기를 업데이트하고; 노드의 시퀀스 번호를 증가시키고; 노드를 잠금 해제함. 실시예에서 헤더에 있는 락 워드와 소형 블록의 크기는 동일한 64-비트 워드를 공유할 수 있으므로, 라이터(401)는 단일 명령으로 노드 버전을 증가시키고, 노드의 크기를 업데이트하며, 잠금을 해제할 수 있다. 새로운 항목은 헤더(603)에 현재 지정된 바와 같이 소형 블록의 끝을 지나(beyond) 저장되기 때문에, 동시(concurrent) 리더들은 새로운 항목이 복사되는 동안 그것을 관찰하지 않는다. 그들은 항목이 없거나 완전히 기록된 항목을 갖는 노드를 관찰한다.

[0088] 실시예에서 리더(402)(예를 들어, FPGA)는 잎 노드들(102L)을 캐싱하지 않는다. 만약 캐싱했다면, 라이터(401)는 PCIe(405)를 통해 지시를 발행하여 캐시(404)를 무효화해야 하며, 이는 일반적인 경우의 삽입에 추가적인 오버헤드를 가져올 것이다.

[0089] **대소 병합(large-small merge):** 소형 블록(602)이 너무 커지면 라이터(401)는 대형 및 소형 블록(601, 602)을 병합한다(도 11 참조). 이는 노드(102L)에 대해 새로운 메모리 버퍼를 할당하고 노드의 모든 항목들(101)을 새로운 버퍼에 있는 대형 블록으로 정렬한다. 실시예에서, 새로운 항목은 (새로운 소형 블록이 아닌) 병합된 대형 블록에 추가된다. 즉, 새로운 노드의 소형 블록은 병합 후 비어 있다. 원칙적으로는 다른 방법(즉, 새로운 항목이 새로운 소형 블록에 추가되는 첫번째 항목임)으로 수행될 수도 있지만, 새로운 항목을 대형 블록에 즉시 저장하면 시간이 지남에 따른 대소 병합들이 약간 줄어들게 된다.

[0090] 쇼트컷 키들(604)을 사용하는 실시예에서, 라이터(401)는 항목들을 정렬하는 동안 쇼트컷 키들을 선택한다(대형 블록은 변경할 수 없기(immutable) 때문에 그 시점에서 쇼트컷 키들을 선택함). 각각의 처리된 항목에 대해, 라이터(401)는 예를 들어 지금까지 복사된 바이트 수, 복사할 나머지 바이트, 및/또는 키와 값의 평균 크기에 기초하여 그것을 쇼트컷에 넣을지 또는 대형 블록에 넣을지를 결정한다. 그것은 바람직하게는 대형 블록 세그먼트들을 비슷한 크기로 유지하면서 쇼트컷 키들의 수를 최대화한다.

[0091] 모든 항목들이 새로운 버퍼에 복사될 때, 라이터(401)는 페이지 테이블(501)에 있는 노드의 LID 매핑을 새로운 버퍼의 주소로 원자적으로 대체한다. LID 매핑을 업데이트하기 위해, 실시예에서 라이터(401)는 페이지 테이블의 CPU 및 FPGA 복사본 모두에서 LID 엔트리를 업데이트한다. 그것은 LID 엔트리에 대한 소프트웨어 잠금을 획득하고, FPGA에 페이지 테이블 업데이트 지시를 발행하고, FPGA 지시가 완료된 후 소프트웨어 잠금을 해제한다. 노드의 LID는 동일하게 유지되므로 부모 노드는 변경될 필요가 없다. 마지막으로, 라이터는 이전(old) 메모리 버퍼를 가비지 수집 목록(garbage collection list)에 넣고, 잠금을 해제하고, 그것의 헤더에 "삭제됨" 플래그를 설정하여 동시 라이터들이 이전 버퍼를 업데이트하지 않도록 보장한다. 진행 중인 작업은 여전히 이전 버퍼로부터 판독할 수 있지만, 작업이 삭제된 노드를 업데이트해야 하는 경우 재시도(retry)한다. 재시도 시에, 새로운 노드 매핑이 사용될 것이다.

[0092] 사용되는 가비지 수집은 동시 시스템들에 대한 표준 명시적 가비지 수집(standard explicit garbage collection), 예를 들어, RCU, 에포크(epoch)-기반 메모리 관리자, 또는 해저드 포인터(hazard pointer)와 같

은 임의의 적절한 구성일 수 있다. 도 11은 삽입 작업 동안 대형 블록과 소형 블록을 병합하는 예를 도시한다. 노드(102L)에 새로운 항목을 삽입하면 소형 블록(602)이 그것의 최대 크기를 초과하게 하여 소형 블록은 이제 새로운 항목과 함께 대형 블록(601)으로 병합될 것이다. 단계 a)는 가장 오른쪽 노드(102L)에 삽입되기 전의 트리(103)를 보여준다. 단계 b)는 새로운 버퍼 할당 및 그것으로의 병합을 보여준다. 노드에 대한 이전 포인터(파선)가 삽입된다. 단계 c)에서, 페이지 테이블은 새 버퍼를 가리키도록 업데이트되고, 이전 버퍼는 진행 중인 임의의 작업에 필요하지 않으면 가비지 수집(garbage collection, GC)으로 배치된다.

[0093] 이 접근방식의 이점은 병합과 같은 복잡한 업데이트가 원래 메모리 주소에 있는 노드의 이전 인스턴스에 대한 판독 잠금을 설정하지 않고도 수행될 수 있다는 것이다(그렇지만 일부 작업들은 여전히 기록 잠금을 사용할 수 있고, 이는 다수의 라이터들만 방지하고, 하나의 라이터와 하나 이상의 리더가 노드에 액세스할 수 있도록 허용함). 라이터(401)가 새로운 버퍼에서 노드(102L)를 업데이트하는 중이지만 페이지 테이블이 업데이트되기 전에, 리더(402)가 노드(102L)를 판독하려고 시도하면, 리더(402)는 페이지 테이블(501)에 현재 지정된 대로 기존 메모리 주소로부터 이전 인스턴스를 단순히 판독한다. 새로운 버퍼(즉, 새로운 메모리 위치)에 대한 기록이 완료되면, 라이터(401)는 그러면 페이지 테이블(501)을 신속하게 업데이트하여 노드의 LID에 대한 개개의 페이지-테이블 엔트리를 새로운 메모리 주소로 전환할 수 있다. 실시예에서 이는 단일 원자적 작업으로 행해질 수 있다. 임의의 후속 판독은 그러면 업데이트된 페이지 테이블 엔트리에 기초하여 새로운 주소로부터 노드의 새로운 인스턴스를 판독할 것이다. 노드 분할이나 노드 병합과 같은 다른 복잡한 업데이트에 대해 유사한 접근방식이 사용될 수 있다.

[0094] 새로운 메모리 버퍼가 LID에 대해 할당될 때, 그것의 시퀀스 번호는, 상기 버퍼가 그것을 삭제하고 재사용하기 전에 어떠한 작업에 의해서도 도달할 수 없다는 것이 보장되므로 안전하게 0으로 설정될 수 있다. 예시적으로 일부 예시적인 크기를 살펴보면, 소형 블록이 512 바이트보다 커지고 소형 블록 엔트리들의 크기가 적어도 10 바이트일 때(삭제 엔트리 크기는 10 바이트이고, 삽입 엔트리 크기는 적어도 13 바이트임) 대형 및 소형 블록(601, 602)이 병합되면, 버퍼의 최대 버전은 41 이며, 이는 31-비트 시퀀스 번호는 결코 랩 어라운드(wrap around)하지 않는다는 것을 의미한다. 실제로 일 구현에서 가장 작은 크기는 실제로 평균 6 바이트이지만, 1-바이트 엔트리라도 시퀀스 번호는 오버플로(overflow)하지 않을 것이다.

[0095] **노드 분할(NODE SPLITTING)**

[0096] 잎(102L)에 삽입을 위한 공간이 충분하지 않은 경우, 라이터(401)는 잎을 두 개로 분할하고 새로운 항목에 대한 포인터를 부모에 삽입한다(도 12). 부모가 가득 차있으면, 라이터(401)는 부모도 분할한다. 이러한 방식으로 분할은 뿌리까지 전파될 수 있다. 업데이트되었지만 분할되지 않은 내부 노드(102I)는 분할의 뿌리라고 불리운다. 라이터는 페이지 테이블(501)에서 분할의 뿌리에 대한 매핑을 업데이트하여 분할의 뿌리 아래의 전체 서브트리를 교체한다. 임의의 데이터를 업데이트하기 전에, 라이터는 작업에 의해 업데이트되고 있는 모든 노드들(102)에 대한 기록 잠금을 획득할 수 있다(이는 다른 라이터들이 간섭하는 것을 방지하기 위한 잠금임). 노드 버전 불일치(mismatch)로 인해 잠금이 실패하면, 라이터는 획득된 모든 기록 잠금을 해제하고 작업을 다시 시작한다. 라이터는 또한 작업을 완료하기 위해 필요한 모든 메모리 버퍼들을 할당한다. 그것이 분할하는 각각의 노드에 대해 두 개의 새로운 노드(새로운 LID와 새로운 버퍼를 가짐)들과, 분할의 뿌리에 대해 메모리 버퍼를 할당한다. 시스템 메모리 부족으로 인해 메모리 할당이 실패하는 경우, 라이터는 포기하고 호출자(caller)에게 적절한 상태 코드를 리턴할 수 있다. 임의의 잠금을 획득하고 메모리를 할당한 후, 라이터는 잎으로부터 노드들을 처리하여 그것들을 새로 할당된 버퍼들로 분할한다. 분할로 인해 생성된 두 노드 각각은 원래 노드로부터의 데이터의 대략 절반을 갖는 것으로 된다. 라이터는 분할하면서 대형 및 소형 블록들을 병합한다. 분할의 뿌리에 도달하면, 라이터는 그것을 새로운 메모리 버퍼에 복사한다. 그것은 새로운 자식 노드들에 대한 포인터와 노드들 간 경계에서의 키를 포함시키기 위해 새로운 버퍼를 수정한다. 두 서브트리를 스와핑하기 위해, 라이터는 페이지 테이블에서 분할의 뿌리에 대한 매핑을 그것의 새로운 메모리 버퍼의 주소로 업데이트한다. 이러한 방식으로 서브트리를 스와핑하는 것은 리더들이 이전의 서브트리나 새로운 서브트리를 관찰하는 것을 보장한다. 그런 다음 라이터는 분할된 모든 노드들의 메모리 버퍼들, 그것들의 LID 및 분할의 뿌리에 대한 이전의 메모리 버퍼를 가비지 수집 목록에 넣는다. 마지막으로, 그것은 모든 메모리 버퍼들의 잠금을 해제하고 그것들을 삭제된 것으로 마킹한다.

[0097] 도 12는 삽입 동안에 노드를 분할하는 예를 도시한다. 파선 박스들은 새로 할당된 노드 버퍼들을 나타낸다. 채워진 박스들은 새로운 LID와 메모리 버퍼를 갖는 새로운 노드들을 나타낸다. 단계 a)는 분할 전의 트리(103)를 도시한다. 가장 오른쪽 노드가 분할되고 있다. 분할의 뿌리는 트리의 뿌리이다. 단계 b)에서, 분할을 위해 두 개의 새로운 노드가 할당되고 분할의 뿌리에 새로운 버퍼가 할당된다. 단계 c)에서, 분할의 뿌리에 대한 매핑이

업데이트되고 노드들이 가비지 수집(GC) 목록에 추가된다.

[0098] 실시예에서, 라이터는 또한 스캔 작업 동안에 사용되는, 형제 잎 포인터들을 업데이트한다. 그것은 형제 잎들을 잠그고 새로운 서브트리에서 스와핑한 후 포인터들을 업데이트한다. 형제 포인터들과 분할의 뿌리가 원자적으로 업데이트되지 않더라도, 리더들은 트리의 일관된 상태를 관찰한다.

[0099] 트리의 뿌리가 새로운 항목에 맞지(fit) 않는 경우, 라이터는 그것을 분할하고 새로운 뿌리를 생성하여 트리 높이를 증가시킨다. 라이터는 단지 분할의 뿌리에 대한 메모리 버퍼 대신 새로운 뿌리에 대해 LID와 메모리 버퍼를 할당하는 것을 제외하고, 위에서와 같이 분할을 실행한다. 그것은 이전 뿌리의 왼쪽 절반으로 설정된 가장 왼쪽 포인터와 이전 뿌리의 오른쪽 절반을 가리키는 단일 항목을 갖는 것으로 새로운 뿌리를 초기화한다. 그런 다음 그것은 리더(예를 들어, FPGA)에 뿌리와 트리 높이를 업데이트하도록 지시를 전송할 수 있다. 하나의 예시적인 구현에서, 리더(402)는 뿌리 노드의 LID와 트리 레벨들의 수를 캐시(404)가 아닌 그것의 로컬 레지스터 파일에 기록한다. 캐시에 있는 데이터 복사본, 예를 들어 이전의 뿌리 노드는, 그것의 LID가 또 다른 물리적 노드에 대해 재사용될 때 결국 무효화되거나, 가능하게는 축출될 것이며 이는 조금 있으면 이전 데이터는 액세스될 수 없기 때문이다. 따라서 캐시는 캐싱된 노드 데이터의 일관성을 유지하는 데에만 집중할 필요가 있다.

[0100] **삭제(DELETE)**

[0101] 항목(101)을 삭제하는 것은 새로운 항목을 삽입하는 것과 유사하다. 라이터(401)는 삭제 엔트리(delete entry)를 소형 블록(602)에 삽입한다. 상기 엔트리는 그것이 삭제하고 있는 항목을 가리키므로, 삭제되는 항목은 룩업 작업 동안 무시된다. 대형 및 소형 블록(601, 602)을 병합하는 동안, 삭제된 항목들이 차지했던 공간은 회수된다. 노드(102)가 비어 있으면, 그것은 삭제된다. 노드 삭제는 노드 분할과 유사하게 잎(102L)으로부터 트리(103) 위로 진행될 수 있다. 실시예에서 그 원자성(atomicity)은 위에서 설명된 것과 동일한 기술을 사용하여 보장된다.

[0102] **수정(MODIFY)**

[0103] 수정 작업은 삽입과 삭제의 조합으로 수행될 수 있다. 라이터(401)는 이전 항목에 대해 삭제 엔트리를, 그리고 소형 블록(602)에 새로운 항목을 추가하고, 그것들을 노드의 헤더(603)를 업데이트하여 원자적으로 게시한다.

[0104] **결론**

[0105] 위의 실시예는 단지 예로서 설명되었다는 것이 이해될 것이다. 보다 일반적으로, 본 명세서에 개시된 일 양태에 따르면, 시스템으로서 트리 구조를 포함하는 데이터 구조를 저장하는 메모리 - 트리 구조는 각각 노드 ID를 갖는 복수의 노드들을 포함하고, 일부 노드들은 잎 노드들이고 다른 노드들은 내부 노드들이며, 각각의 내부 노드는 상기 트리 구조에서 하나 이상 자식들의 개개의 세트에 대한 부모이고, 각각의 자식은 잎 노드 또는 또 다른 내부 노드이고, 각각의 잎 노드는 자식이지만 부모는 아니며, 잎 노드들 각각은 키-값 쌍을 각각 포함하는 하나 이상의 항목들의 개개의 세트를 포함하고, 내부 노드들 각각은 그것의 개개의 자식들 각각의 노드 ID를 상기 개개의 자식에 포함된 키들의 범위에 매핑함 - 를 포함하는 시스템이 제공된다. 시스템은 잎 노드들에 항목들을 기록하도록 마련된 라이터; 및 잎 노드들로부터 항목들을 판독하도록 마련된 리더를 더 포함한다. 잎 노드들 각각은 개개의 제1 블록과 개개의 제2 블록을 포함하고, 제1 블록은 메모리의 주소 공간에 키 순서로 정렬된 개개의 잎 노드의 복수의 항목들을 포함한다. 라이터는 트리 구조에 새로운 항목들을 기록할 때, 트리 구조를 통해 키들의 노드 ID들에의 매핑을 따라 어느 잎 노드에 기록할지 식별하고, 식별된 잎 노드의 제2 블록에 새로운 항목을, 키 순서로 정렬되지 않고, 기록되는 순서로 기록하도록 구성된다. 리더는 트리 구조로부터 하나 이상의 타깃 항목들을 판독할 때, 트리 구조를 통해 키들의 노드 ID들에의 매핑을 따라 어느 잎 노드로부터 판독할지 결정하고, 그런 다음 하나 이상의 타깃 항목들에 대해 상기 결정된 잎 노드를, a) 제1 블록에 이미 정렬되어 있는 항목들의 순서, 및 b) 리더가 제1 블록의 항목들과 관련하여 키에 의해 제2 블록의 항목들을 정렬하는 것에 기초하여 검색하도록 구성된다.

[0106] 실시예에서 라이터는 상기 시스템의 컴퓨터 판독가능 스토리지에 저장되어 있고 상기 시스템의 하나 이상의 프로세서에서 실행되도록 마련된 소프트웨어로 구현될 수 있다. 실시예에서 리더는 PGA, FPGA 또는 전용 하드웨어 회로로 구현될 수 있다. 그러나 대안적으로 라이터는 하드웨어로 구현될 수 있고/있거나 리더는 소프트웨어로 구현될 수 있고, 또는 어느 것이든 소프트웨어와 하드웨어의 조합으로 구현될 수 있다.

[0107] 실시예에서 트리 구조는 B-트리의 형태, 예를 들어 항목들이 중간 노드들에는 저장되지 않는 B+ 트리의 형태를 취할 수 있다.

[0108] 실시예에서, 잎 노드들 및/또는 내부 노드들 중 적어도 일부에서, 개개의 제1 블록은 세그먼트들로 나뉘고 노드는 복수의 쇼트컷들을 포함하는 쇼트컷 블록을 더 포함할 수 있고, 각각의 쇼트컷은 i) 개개의 제1 블록의 해당 세그먼트의 경계에서의 키의 표시, 및 ii) 잎 노드 내의 해당 세그먼트에 대한 오프셋을 포함한다. 그러한 실시예에서, 리더는 제1 블록에 대한 상기 검색을 수행할 때, 키에 기초하여 하나 이상의 항목들을 포함할 수 있는 세그먼트들만을 검색하기 위해 쇼트컷들을 사용하도록 구성될 수 있다.

[0109] 실시예에서, 라이터는 기록 시, 잎 노드들 중 적어도 하나에 기록할 때, 각각의 항목과 연관된 백 포인터를 개개의 제2 블록에 포함시키도록 구성되고, 각각의 백 포인터는 제2 블록에서의 상기 연관된 항목과 비교하여 다음으로 큰 또는 다음으로 작은 키를 갖는 제1 블록에서의 항목에 대한 오프셋을 포함한다. 그러한 실시예에서, 리더는 제2 블록의 상기 정렬을 수행할 때, 개개의 제1 블록과 관련하여 키에 의해 제2 블록의 항목들을 정렬하기 위해 백 포인터들을 사용하도록 구성될 수 있다.

[0110] 실시예에서, 라이터는 기록 시, 잎 노드들 중 적어도 하나에 기록할 때, 제2 블록에서의 항목들 각각에 대해, 상기 항목을 제2 블록에 기록할 때 제2 블록에서의 현재 키와 관련하여 항목의 키의 순서를 나타내는 순서 힌트들을 포함시키도록 구성된다. 그러한 실시예에서, 리더는, 제2 블록의 상기 정렬을 수행할 때, 개개의 제1 블록과 관련하여 키에 의해 제2 블록의 항목들을 정렬하기 위해 순서 힌트들을 사용하도록 구성될 수 있다.

[0111] 예를 들어, 리더는 순서 힌트들을, 개개의 제2 블록에서의 항목들을, 그것들이 정렬되어 있는 순서로 처리하고; 순서 힌트들을 사용하여 상기 항목들을 어레이로 정렬함으로써 사용하도록 구성되고, 어레이는 제2 블록 내에 상기 항목들의 오프셋을 저장한다. 순서 힌트 i를 갖는 각각의 항목이 처리될 때, 제2 블록에서의 그것의 오프셋은 어레이에서 포지션 i에 삽입되고, 포지션 $j \geq i$ 에서의 모든 항목들은 오른쪽으로 이동되어 상기 전체 제2 블록이 처리된 후, 어레이는 항목들의 오프셋을 상기 정렬된 순서로 포함하도록 된다.

[0112] 실시예에서, 데이터 구조는 노드 ID들을 개개의 메모리 주소에 매핑하는 페이지 테이블을 포함할 수 있다. 이 경우, 리더는 결정된 노드 ID의 노드로부터 판독할 때, 페이지 테이블을 사용하여 메모리 내 상기 결정된 노드의 메모리 주소를 결정하도록 구성된다. 일부 그러한 실시예에서, 라이터는 노드를 기록, 분할 또는 병합하기 위해 노드를 업데이트할 때, 노드의 업데이트된 버전을 새로운 메모리 주소에 기록하고, 그런 다음 일단 기록되면, 새로운 메모리 주소를 가리키도록 페이지 테이블에서의 개개의 메모리 주소를 업데이트하도록 구성될 수 있다.

[0113] 실시예에서, 각각의 제2 블록은 최대 크기를 가질 수 있다. 라이터는 식별된 노드의 개개의 제2 블록이 최대 크기를 초과하게 하는 새로운 항목을 기록하는 경우, 제2 블록을 개개의 제1 블록과 병합하도록 구성될 수 있다.

[0114] 실시예에서, 데이터 구조가 노드 ID들을 개개의 메모리 주소에 매핑하는 페이지 테이블을 포함하는 경우, 라이터는 개개의 제2 블록과 제1 블록을 병합하기 위해 노드를 업데이트할 때, 병합을 포함하는 노드의 업데이트된 버전을 새로운 메모리 주소에 기록하고, 그런 다음 일단 병합이 완료되면, 새로운 메모리 주소를 가리키도록 페이지 테이블에서의 개개의 메모리 주소를 업데이트하도록 구성될 수 있다.

[0115] 실시예에서 각각의 잎 노드는 개개의 잎 노드의 키들에 이웃하는 키들의 범위를 포함하는 하나 이상의 형제 잎 노드들의 노드 ID를 추가로 표시할 수 있다. 또한, 리더는 범위 스캔을 수행하여 키들의 범위로부터 항목들을 판독하도록 구성될 수 있다. 그러한 실시예에서, 리더는 하나보다 많은 잎 노드에 걸쳐 있는 항목들의 범위로부터 판독하는 범위 스캔을 수행할 때, 트리 구조를 통해 키들의 노드 ID들에의 매핑을 따라 스캔된 범위에 있는 키들 중 하나를 포함하는 잎 노드들 중 하나를 결정하고, 그런 다음 상기 잎 노드들 중 상기 하나에 표시된 형제 잎 노드들 중 적어도 하나의 ID를 사용하여 스캔된 범위에 있는 적어도 하나의 다른 키를 포함하는 적어도 하나의 다른 잎 노드를 결정하도록 구성될 수 있다.

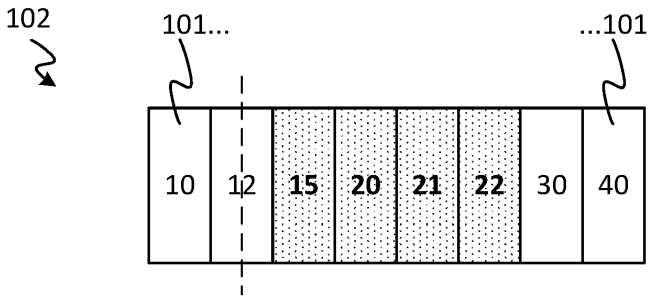
[0116] 본 명세서에 개시된 또 다른 양태에 따르면, 본 명세서에 개시된 임의의 실시예에 따른 메모리, 라이터 및/또는 리더의 동작들을 포함하는 방법이 제공된다.

[0117] 또 다른 양태에 따르면, 비일시적 컴퓨터 판독가능 매체 또는 매체들에 구현된 컴퓨터 프로그램으로서, 하나 이상의 프로세서에서 실행될 때 라이터의 상기 기록하는 단계 및/또는 리더의 상기 판독하는 단계를 수행하도록 구성된 코드를 포함하는 컴퓨터 프로그램이 제공될 수 있다.

[0118] 개시된 기술의 다른 변형 또는 응용은 일단 본 개시 내용이 주어지면 이 분야의 기술자에게 명백해질 수 있다. 본 개시의 범위는 설명된 실시예에 의해 제한되지 않으며 첨부된 청구범위에 의해서만 제한된다.

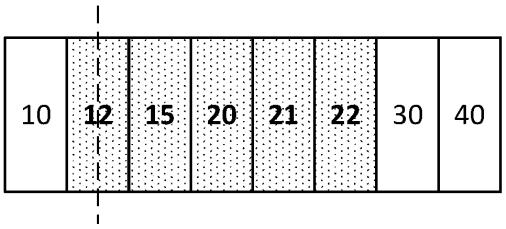
도면

도면1



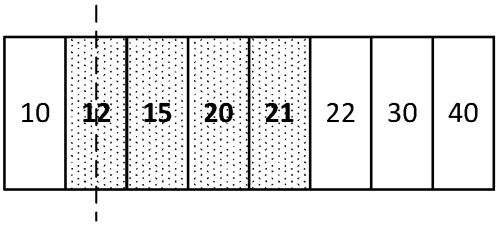
[14, 22]에 대한 포괄적 스캔

도면2



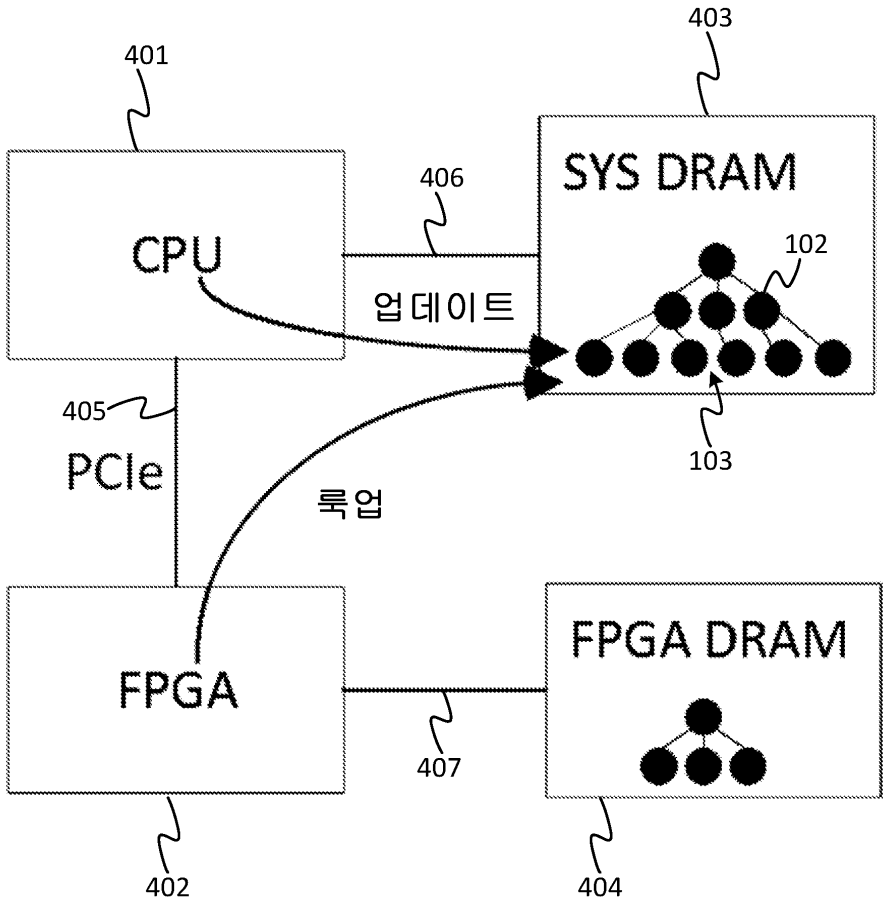
[14, 22]에 대한 커버링 스캔

도면3

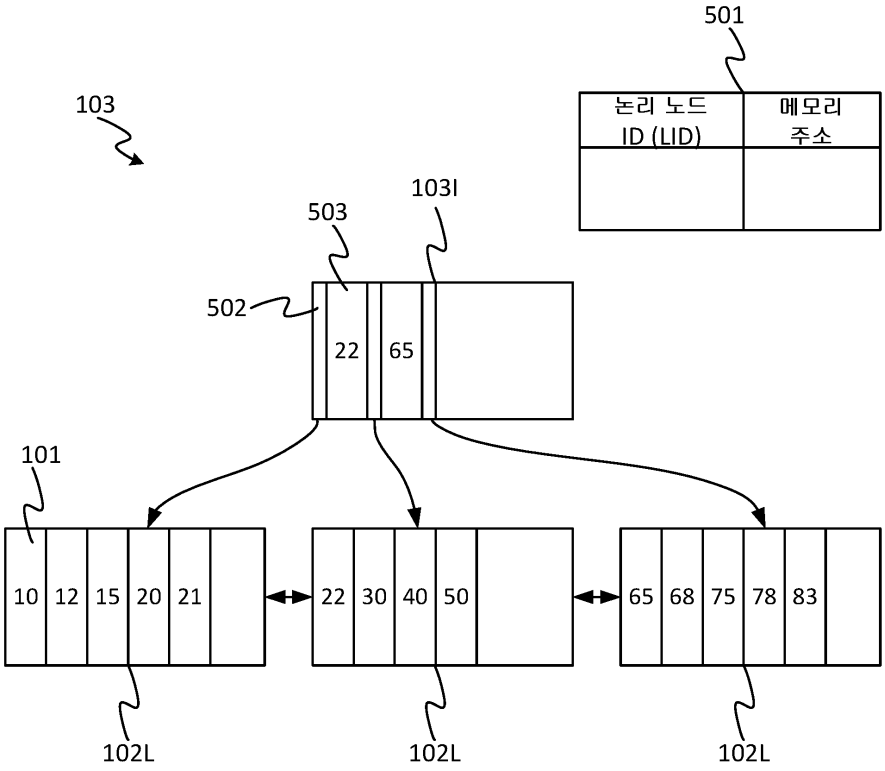


(14, 22)에 대한 커버링 스캔

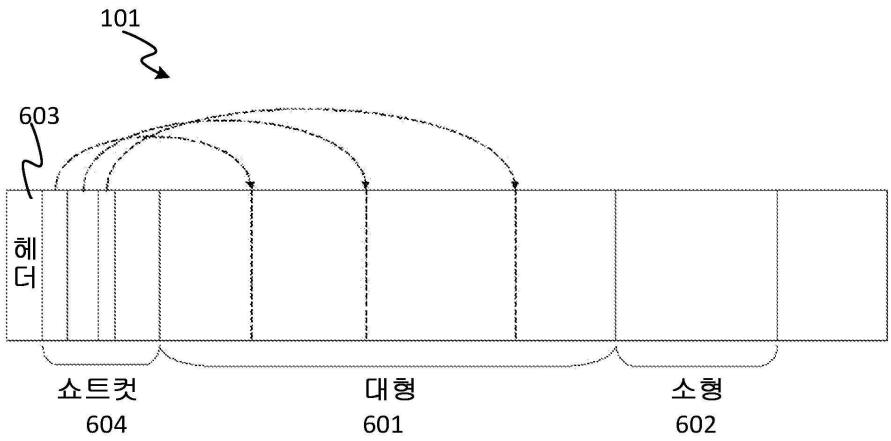
도면4


401
403
CPU
406
SYS DRAM
업데이트
102
405
PCIe
103
룩업
FPGA
FPGA DRAM
407
402
404

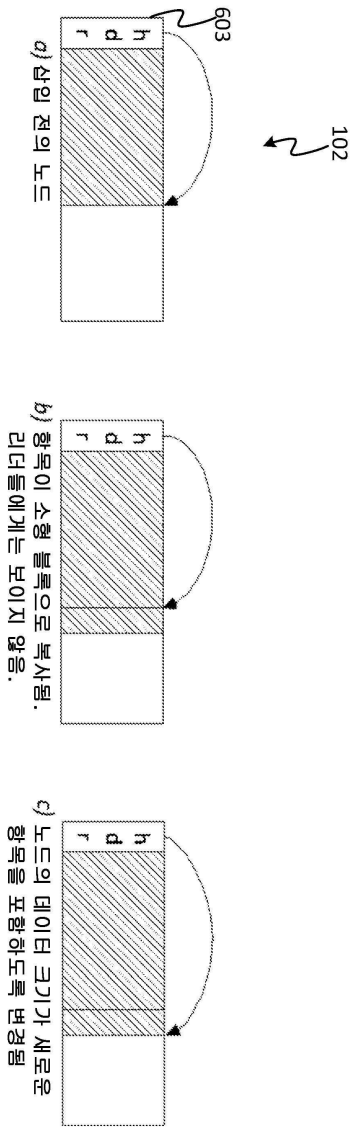
도면5


501
논리 노드 ID (LID)
메모리 주소
103
103I
503
502
22
65
101
10
12
15
20
21
22
30
40
50
65
68
75
78
83
102L
102L
102L

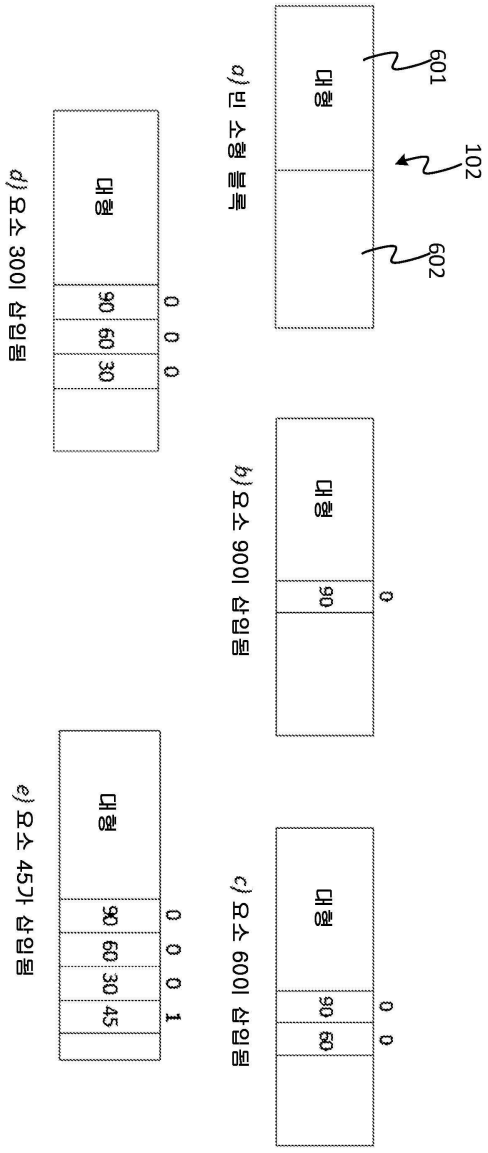
도면6


101
603
헤더
쇼트컷
604
대형
601
소형
602

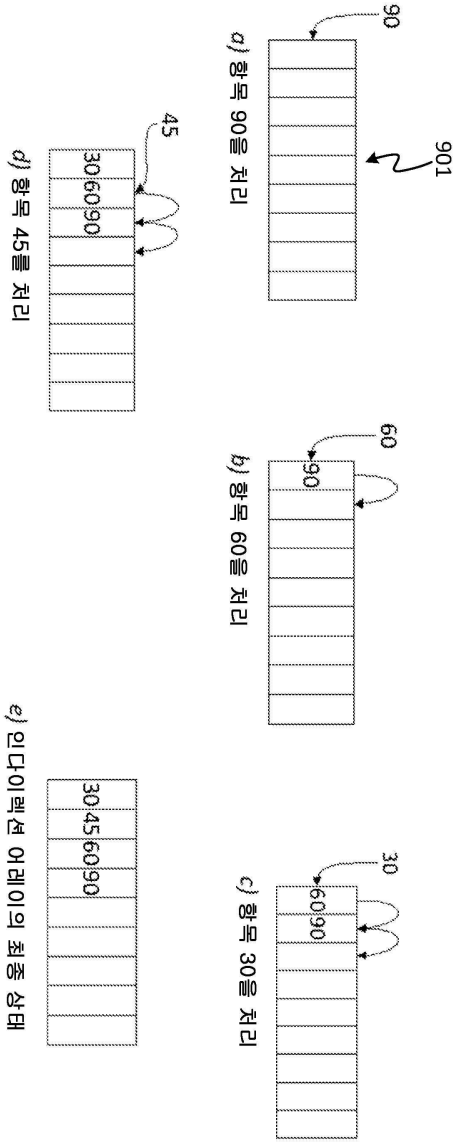
도면7


603
h
d
r
102
a) 삽입 전의 노드
b) 항목이 소형 블록으로 복사됨.
리더들에게는 보이지 않음.
c) 노드의 데이터 크기가 새로운
항목을 포함하도록 변경됨

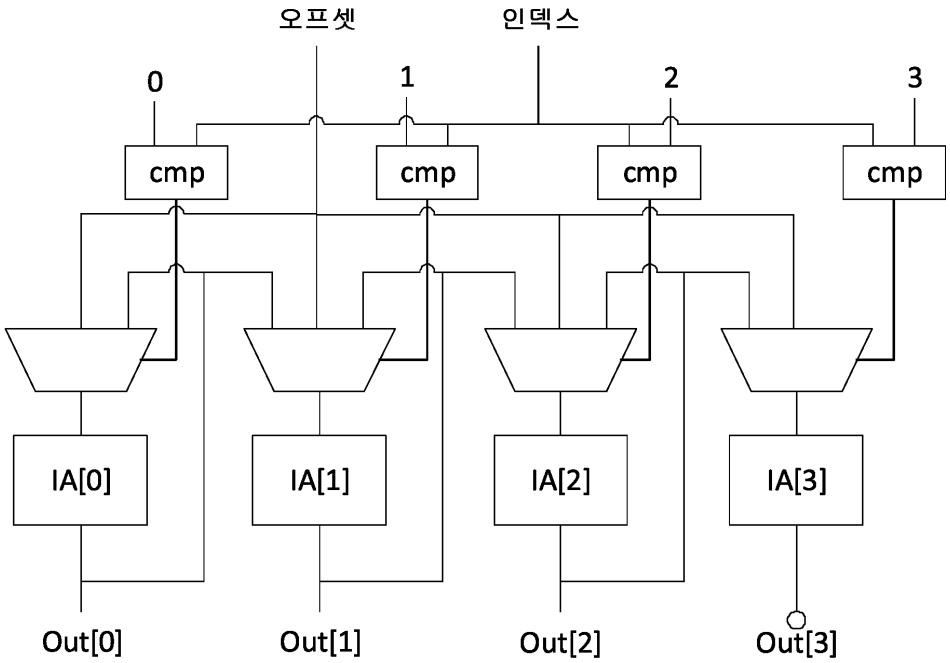
도면8


a) 비 요소 올림 삽입
대형
601
102
602
b) 요소 900I 삽입물
대형
90
0
c) 요소 600I 삽입물
대형
90
60
0 0
d) 요소 300I 삽입물
대형
90
60
30
0 0 0
e) 요소 45기 삽입물
대형
90
60
30
45
0 0 0 0 1

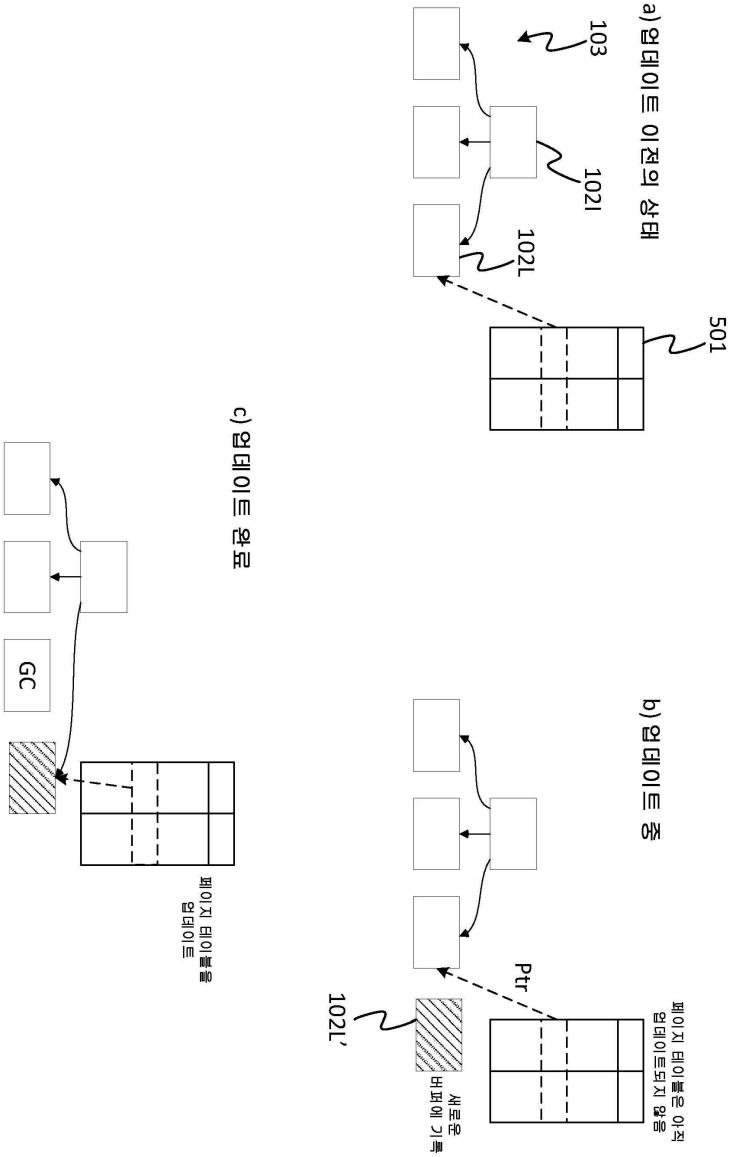
도면9


a) 항목 90을 처리
90
901
b) 항목 60을 처리
60
90
c) 항목 30을 처리
30
60
90
d) 항목 45를 처리
45
30
60
90
e) 인다이렉션 어레이의 최종 상태
30
45
60
90

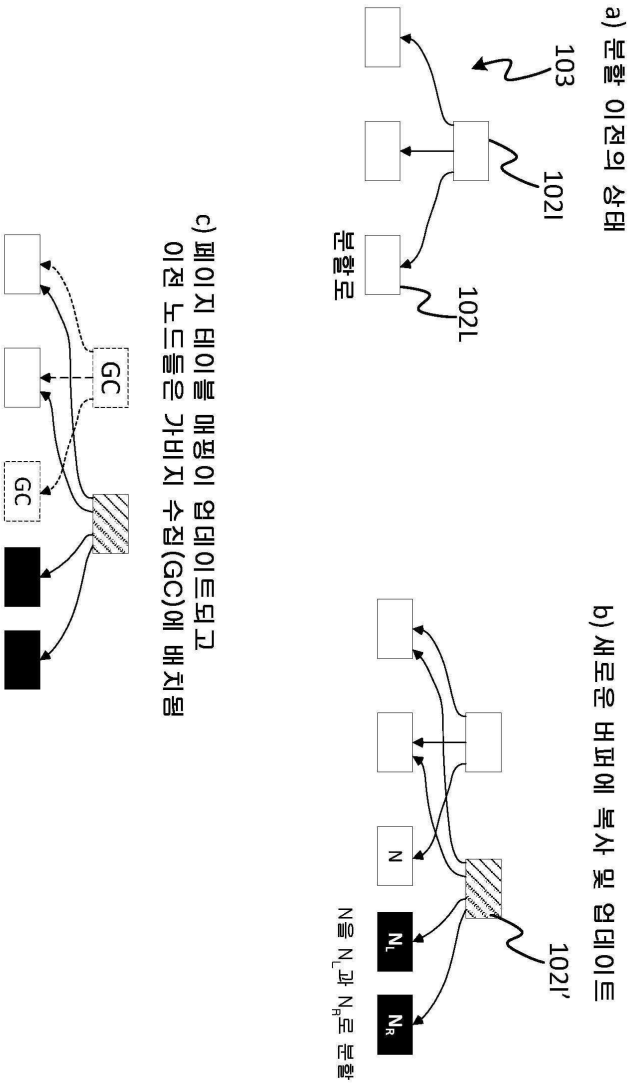
도면10


오프셋
인덱스
0
1
2
3
cmp
cmp
cmp
cmp
IA[0]
IA[1]
IA[2]
IA[3]
Out[0]
Out[1]
Out[2]
Out[3]

도면11


a) 업데이트 이전의 상태
103
1021
102L
501
b) 업데이트 중
Ptr
1021'
새로운 버퍼에 기록
페이지 테이블을 아직 업데이트되지 않음
c) 업데이트 완료
GC
페이지 테이블을 업데이트

도면12


a) 분할 이전의 상태
103
102l
분할로
102L
b) 새로운 버퍼에 복사 및 업데이트
N
N_L
N_R
102l'
N를 N_L과 N_R로 분할
c) 페이지 테이블 매핑이 업데이트되고
이전 노드들은 가비지 수집(GC)에 배치됨
GC
GC