﻿**6주차 인덱싱(2)**

**1. 해시 인덱스 개요**

* 정확한 값 검색에 최적화된 인덱스
* 해시 함수를 통해 키 값을 버킷에 매핑
* *(검색 키 → 해시 함수 → 버킷 주소 → 레코드)*
* 범위 검색에는 부적합 ❌

**2. 순서 인덱스 vs 해시 인덱스 비교표**

****

**3. 정적 해싱 (Static Hashing)**

* 고정 크기의 해시 테이블
* 해시 함수: 키 → 버킷
* 충돌 시 오버플로 체인으로 연결 (Closed Addressing)
* 단점
* 버킷 초과 시 성능 저하
* 공간 낭비 가능
* 주기적인 재해싱 필요 (비용 큼)

**4. 동적 해싱 (Dynamic Hashing)**

**✅ 확장성 해싱 (Extendible Hashing)**

* 해시 함수의 prefix를 사용해 버킷 주소 테이블 접근
* 버킷이 꽉 차면 버킷 분할 or 테이블 확장
* i > ij: 하위 절반 포인터 변경 → 새 버킷 분할
* i = ij: 테이블 확장 후 분할

**5. 확장성 해싱 삽입 & 삭제**

* 삽입: 충돌 발생 시 분할 → 테이블 확장
* 삭제: 삭제 엔트리 삽입 후 병합 가능
* 조건 만족 시 버킷 주소 테이블 크기 축소도 가능

**6. 쓰기 최적화 인덱스 구조**

**✅** LSM 트리 (Log-Structured Merge Tree)

* 쓰기 효율을 극대화한 구조
* 메모리(L0) → 디스크(L1, L2, ...)로 병합하며 저장
* L0은 메모리 내 정렬된 구조 (쓰기 빠름)
* L1 이상은 디스크 내 B+-트리 형태

🔁 삽입 흐름

1. L0에 저장
2. L0 용량 초과 시 → L1에 병합
3. L1도 초과 시 → L2로 병합 ... 재귀적 진행

❌ 삭제/갱신 처리

* 삭제: '삭제' 엔트리 삽입 후 병합 시 제거
* 갱신: 삭제 + 삽입 조합

🔍 단점

* 여러 계층 탐색 필요 → 검색 성능 낮음
* 병합 비용 큼

**7. LSM 변형 구조**

**✅** 단계별 병합 인덱스 (Stepped-Merge Index)

* 각 계층에 여러 개 트리 존재
* 쓰기 성능↑, 검색 성능↓
* 블룸 필터로 검색 효율 향상
* 사용 예시: Google BigTable, Cassandra, MongoDB, MyRocks 등

**8. O/X 핵심 포인트**

* 해시 인덱스는 범위 검색에 유리하다 → ❌
* 확장성 해시는 해시 테이블 전체 복사 없이 버킷만 분할할 수 있다 → ⭕

**9. 시험 전날 체크리스트**

* 해시 인덱스 vs 순서 인덱스 차이
* 정적 vs 동적 해싱 차이
* 확장성 해싱의 버킷 분할 조건
* LSM 트리 구조 및 삽입/병합 흐름
* LSM과 B+-트리 차이점
* 단계별 병합 인덱스 개념 + 블룸 필터

﻿