# CP3. 저장소와 검색



# Part 1. 데이터 시스템의 기초

## 3. 저장소와 검색

### ▼ 개요

- 데이터베이스가 데이터를 저장하는 방법과 데이터를 요청했을 때 다시 찾을 수 있는 방법을 설명한다.
- DB 가 저장과  검색을 내부적으로 처리하는 방법을 개발자가 주의해야하는 이유는 뭘까?
  - 대개 개발자가 처음부터 자신의 저장소 엔진을 구현하기보다는 사용 가능한 여러 저장소 엔진 중에 애플리케이션에 적합한 엔진을 선택하는 작업이 필요하다. (즉, 내부 엔진은 어떻게 동작하는지, 적합한 엔진을 사용하는 방법은 어떻게 되는지를 알아야 **선택할 수 있는 안목**을 기를 수 있다.)
- 트랜잭션 작업부하에 맞춰 최적화된 저장소 엔진과 분석을 위해 최적화된 엔진 간에는 큰차이가 있다. (트랜잭션 처리나 분석)
- 분석에 최적화된 저장소 엔진 (컬럼 지향 저장소)
- 이번장에, **RDB와 NoSQL 로 불리는 DB에 사용되는 저장소 엔진**을 설명하고 **로그 구조(Log-structured) 계열 저장소**와, **(B트리 같은) 페이지 지향(page-oriented) 계열 저장소 엔진**을 추가로 검토한다.

### ▼ 데이터베이스를 강력하게 만드는 데이터 구조

- 가장 간단한 DB

```
#!/bin/bash

db_set () {
    echo "$1, $2" >> database
}

db_get () {
    grep "^$1," database | sed -e "s/^$1,//" | tail -n 1
}
```

  - key value 저장소 구조에 db_get 시 가장 최근 값을 반환.

- 일반적으로, 파일 추가 작업은 매우 효율적이기 때문에, db_set 은 좋은 성능을 보여준다
  db set과 마찬가지로 많은 DB는 내부적으로 추가 적용 (append-only) 데이터 파일인 **로그(log)**[여기서는 연속된 추가 전용 레코드를 의미한다.] 를 사용한다.
- 반면에, db_get은 많은 레코드가 있을 경우 성능이 매우 좋지 않다. 매번 키를 찾을 때 마다 전체 DB 를 스캔한다.

- **색인 (index) - mysql index, PostgreSQL index**
  - DB에서 특정 키의 값을 효율적으로 찾기 위해서는 다른 데이터 구조가 필요하다. 바로 **색인**
  - 일반적인 개념은 어떤 부가적인 메타데이터를 유지 하는 것
  - 색인은 기본 데이터에서 파생된 추가적인 구조이다.
  - DB는 색인의 추가와 삭제를 허용한다.
    이 작업은 데이터베이스의 내용에는 영향을 미치지 않는다. 단지 질의 성능에만 영향을 준다.
    - 쓰기 과정에 오버 헤드가 발생한다. (인덱스 테이블에도 추가적인 적재가 필요함)
    - 색인을 잘 선택했다면 읽기 질의 속도가 향상한다.
  - **해쉬 색인**
    - 키 값 저장소는 대부분 프로그래밍 언어에서 볼 수 있는 사전 타입과 유사하다. 보통 hashMap, hashTable 으로 구현한다.
    - 가장 간단한 색인 전략은 키를 데이터 파일의 바이트 오프셋에 매핑해 인메모리 해시 맵을 유지하는 전략이다.


      - 파일에 새로운 key-value 를 추가할 때마다 방금 기록한 데이터의 오프셋을 반영하기 위해 해쉬 맵도 갱신해야한다.
      - 값을 조회할 때는 해시 맵을 사용해 인메모리 오프셋을 찾아 해당 위치를 구하고 값을 읽는다.
      - 이 방법은 매우 단순해 보이지만, 실제로 많이 사용하는 접근법이다. ((비트캐스크)가 기본적으로 사용하는 방식)
    - 비트캐스크는 해시 맵을 전부 멤모리에 유지하기 때문에 사용 가능한 램(RAM)에 모든 키가 저장된다는 조건을 전제로 고성능 읽기, 쓰기를 보장한다.
      - 비트캐스크 같은 저장소 엔진은 각 키의 값이 자주 갱신되는 상황에 매우 적합하다.
        - 키당 쓰기 수가 많지만, 메모리에 모든 키를 보관할 수 있다.
    - 지금처럼 파일에 항상 추가만 한다면 결국 디스크 공간 부족이 된다.
      - 해결책은, 특정 크기의 세그먼트(분할, 구간, 세부부분 등)로 로그를 나누는 방식이 좋다.

- 특정 크기에 도달하면 세크먼트 파일을 닫고 새로운 세그먼트 파일에 이후 쓰기를 수행한다. 세그먼트 파일들에 대해
  **컴팩션**을 수행할 수 있다.
  - 컴팩션은 로그에서 중복된 키를 버리고 각 키의 최신 갱신 값만 유지
  - 컴팩션은 보통 세그먼트를 더 작게 만들기 때문에 동시에 여러 세그먼트들을 병합할 수 있다.
- 세그먼트가 쓰여진 후에는 절대 변경할 수 없기 때문에 병합할 세그먼트는 새로운 파일을 만든다.
- 고정된 세그먼트의 병합과 컴팩션은 백그라운드 스레드에서 수행할 수 있다.
- 컴팩션을 수행하는 동안 이전 세그먼트 파일을 사용해 읽기와 쓰기 요청의 처리를 정상적으로 계속 수행 가능하다.
- 병합 과정이 끝난 이후에는 읽기 요청은 이전 세그먼트 대신 새로 병합한 세그먼트를 사용하게끔 전환한다. 이전 세그먼트 파일은 간단히 삭제한다.
- 이제 각 세그먼트는 키를 파일 오프셋에 매핑한 자체 인메모리 해시 테이블을 갖는다. (키의 값은 최신 세그먼트 해시맵 확인 → 그 다음 세그먼트 해시맵을 조회)
- 이런 간단한 생각을 실제로 구현하려면 세부적으로 많은 사항을 고려해야한다.

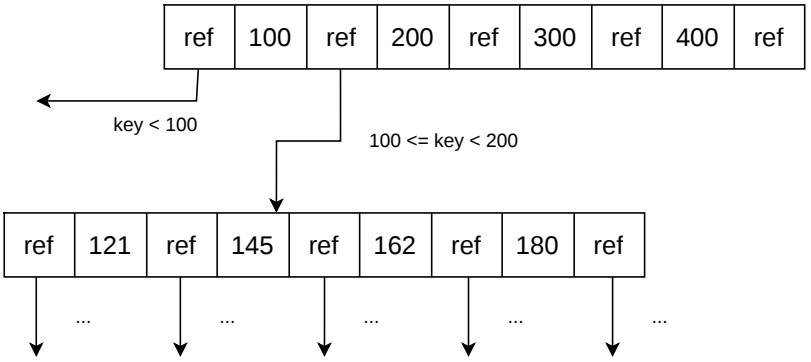
- 고려 사항
  - 파일 형식
    - CSV 는 로그에 가장 적합한 방식이 아니다.
    - 바이트 단위의 문자열 길이를 부호화한 다음 원시 문자열을 부호화하는 바이너리 형식을 사용하는 편이 더 빠르고 간단.
  - 레코드 삭제
    - 키와 관련된 값을 삭제하려면 데이터 파일에 특수한 삭제 레코드(a.k 툼스톤)을 추가해야 한다.
    - 로그 세그먼트가 병합시, 삭제 레코드는 병합 과정에서 삭제된 키의 이전 값을 무시한다.
  - 고장 복구
    - DB 가 재시작되면 인메모리 해시 맵은 손실된다.
    - 원칙적으로는 전체 세그먼트 파일을 처음부터 끝까지 읽고 키에 대한 최신 값의 오프셋을 확인해서 각 세그먼트 해시 맵을 복원할 수 있다. 하지만 세그먼트 파일이 크면 해시 맵 복원은 오랜 시간이 걸릴 수 있고, 이는 서버 재시작을 고통스럽게 만든다.
      비트 캐스크는 각 세그먼트 해시 맵을 메모리로 조금 더 빠르게 로딩할 수 있게 스냅숏을 디스크에 저장해 복구 속도를 높인다.
  - 부분적으로 레코드 쓰기
    - 데이터베이스는 로그에 레코드를 추가하는 도중에도 죽을 수 있다. 비트캐스크 파일은 체크섬을 포함하고 있어 로그의 손산됭 부분을 탐지해 무시할 수 있다.
  - 동시성 제어
    - 쓰기를 엄격하게 순차적으로 로그에 추가할 때 일반적인 구현 방법은 하나의 쓰기 쓰레드만 사용하는 것이다.

        - 데이터 파일 세그먼트는 추가 전용이거나 불변이므로 다중 스레드로 동시에 읽기를 할 수 있다.
    - 추가 전용 로그 (하나의 쓰기 쓰레드) 일 때 장점
        - 추가와 세그먼트 병합은 순차적인 쓰기 작업이기 때문에 보통 무작위 쓰기보다 빠르다.
        - 세그먼트 파일이 추가 전용이나 불변이면 동시성과 고장 복구는 훨씬 간단하다.
            - 값을 덮어쓰는 동안 DB가 죽는 경우에, 이전 값 부분과 새로운 값 부분을 포함한 파일을 나누어 함께 냅두기 때문에 걱정 필요 없다.
        - 오래된 세그먼트 병합은 시간이 지남에 따라 조각화되는 데이터 파일 문제를 피할 수 있다.
    - 해시 테이블 색인 제한 사항
        - 해시 테이블은 메모리에 저장해야 하므로 키가 너무 많으면 문제.
            - 원칙적으로 디스크에 해시 맵을 유지할 수 있지만 불행하게도 디스크 상의 해시 맵에 좋은 성능을 기대하긴 어려움.
              무작위 IO 가 많이 필요하고 디스크가 가득 찬 경우 확장 비용이 비싸고 해시 충돌 해소를 위해 성가신 로직이 필요
        - 해시 테이블은 범위 질의에 효율적이지 않다.
- **SS 테이블과 LSM 트리**
    - **SS**(sorted string) **테이블**이란 로그 구조화 저장소 세그먼트 파일의 형식에 일련의 키 값 쌍을 키로 정렬 하는 것이다.
    - SS 테이블을 해시 색인을 가진 로그 세그먼트 비교시 장점
        - 세그먼트 병합은 파일이 사용 가능한 메모리보다 크더라도 간단하고 효율적이다. (병합 정렬 알고리즘과 유사)
            - 입력 파일들을 함께 읽고 각 파일의 첫 번째 키를 본다(정렬된 순서)
            - 가장 낮은 키를 출력 파일로 복사
            - 이 과정을 반복하면 새로운 키로 정렬된 병합된 세그먼트 파일이 생성된다.
            - 여러 입력 세그먼트에 동일한 키가 있는 경우, 가장 최근 세그먼트만 유지하고 오래된 세그먼트는 버린다.
        - 파일에서 특정 키를 찾기 위해 더는 메모리에 모든 키의 색인을 유지할 필요가 없다.
            - 찾아야 하는 색인이 없더라도 등록된 주변 색인으로 위치를 대강 알 수 있다. (정렬되어 있음으로)
        - 읽기 요청은 요청 범위 내에서 여러 키 값을 스캔해야 하기 때문에 해당 레코드들을 블록으로 그룹화하고 디스크 쓰기 전에 압축한다.
            - 그러면 희소 인메모리 색인의 각 항목은 압축된 블록의 시작을 가리키게 된다. 디스크 공간을 절약한다는 점 외에도 압축은 IO 대역폭 사용도 줄인다.
- **SS 테이블 생성과 유지**
    - 디스크 상에 정렬된 구조를 유지하는 일은 가능하지만(B 트리) 메모리에 유지하는편이 훨신 쉽다.
        - 쓰기 과정시 (레드 블랙 트리, AVL 트리)이런 데이터 구조를 이용하면 임의 순서로 키를 삽입하고 정렬된  순서로 해당 키를 다시 읽을 수 있다.
    - 저장소 엔진을 다음과 같이 만들 수 있다.

        - 쓰기가 들어오면 인메모리 균형 트리 데이터 구조(레드 블랙 트리 등)에 추가한다. (인 메모리 트리는 **멤테이블**이라고도 한다)
        - 멤 테이블이 보통 수 메가바이트 정도의 임곗값보다 커지면 SS 테이블 파일로 디스크에 기록한다.
          트리가 이미 키로 정렬된 키-값 쌍을 유지하고 있기 떄문에 효율적으로 수행할 수 있다.
          새로운 SS 테이블 파일은 데이터의 가장 최신 세그먼트가 된다.
          SS테이블을 디스크에 기록하는 동안 쓰기는 새로운 멤테이블 인스턴스에 기록한다.
        - 읽기 요청을 제공하려면 먼저 멤테이블에서 키를 찾아야한다.
          그 다음 디스크 상의 가장 최신 세그먼트에서 찾는다. 그 다음으로 두번째 오래된 세그먼트… 이렇게 찾는다.
        - 가끔 세그먼트 파일을 합치고 덮어 쓰여지거나 삭제된 값을 버리는 병합과 컴팩션 과정을 수행한다. (백그라운드)
    - 문제
        - 데이터베이스 고장나면 아직 디스크로 기록되지 않고 멤테이블에 있는 가장 최신 쓰기는 손실된다.
        - 이런 문제를 피하기 위해서는 이전 절과 같이 매번 쓰기를 즉시 추가할 수 있게 분리된 로그를 디스크 상에 유지해야 한다.
        - 이 로그는 손상 후 멤테이블을 복원할 때만 필요하기 때문에 순서가 정렬되지 않아도 문제되지 않는다.
        - 멤테이블을 SS 테이블로 기록하고 나면 해당 로그를 버릴 수 있다.
- **LSM 트리 (**로그 구조화 병합 트리) (Log-Structured Merge-Tree)
    - SS 테이블의 형식으로 디스크에 key-value 데이터를 컴팩션(병합정렬(merge sort)) 사용해 키의 최신 값만 유지하는 색인 방식
    - 기본 개념으로 백그라운드에서 연쇄적으로 SS 테이블을 지속적으로 병합하는 것을 의미함.
    - 이 개념은 데이터셋이 가능한 메모리보다 훨씬 더 크더라도 여전히 효과적이다.
      데이터가 정렬된 순서로 저장돼 있다면 범위 질의를 효율적으로 실행할 수 있다.
      이 접근법의 디스크 쓰기는 순차적이기 때문에 LSM 트리가 매우 높은 쓰기 처리량을 보장할 수 있다.
- **성능 최적화**
    - LSM 트리 최적화 - **블룸 필터**
        - LSM 알고리즘은 데이터베이스에 존재하지 않는 키를 찾는 경우 느릴 수 있음.
          멤테이블을 확인한 다음 키가 존재하지 않는다는 사실을 확인하기 전에는 가장 오래된 세그먼트까지 거슬러 올라가야 한다.
        - 이런 종류의 접근을 최적화하기 위해서 저장소엔진은 보통 **블룸 필터**를 추가적으로 사용한다
        - 블룸 필터는 집합 내용을 근사한 메모리 효율적 데이터 구조다.
          블룸 필터는 키가 데이터베이스에존재하지 않음을 알려주므로 존재하지 않는 키를 위한 불필요한 디스크 읽기를 많이 절약 가능하다.
    - **크기 계층(size-tiered) 컴팩션 과 레벨 컴팩션 (leveled compaction)** - SS 테이블을 압축하고 병합하는 순서와 시기를 결정하는 전략 중 일반적인 것
        - 크기 계층 컴팩션
            - 상대적으로 좀 더 새롭고 작은 SS 테이블을 상대적으로 오래됐고 큰 SS테이블에 연이어 병합한다.
        - 레벨 컴팩션

        - 키 범위를 더 작은 SS 테이블로 나누고 오래된 데이터는 개별 “레벨”로 이동하기 때문에 컴팩션을 점진적으로 진행해 디스크 공간을 덜 사용한다.

- **B 트리**
    - 거의 대부분의 RDB의 표준 색인 구현으로, 많은 비관계형 DB에서도 사용한다.
    - 가장 널리 사용되는 색인 구조이고 LSM 색인과는 상당히 다르다.
      비슷한 점, B 트리는 SS 테이블과 같이 키로 정렬된 키-값 쌍을 유지하기 때문에 키-값 검색과 범위 질의에 효율적이다.
    - LSM 색인은 DB를 일반적으로 수 메가바이트 이상의 **가변 크기를 가진 세그먼트**로 나누고 항상 순차적으로 세그먼트를 기록한다.
    - B 트리는 전통적으로 4KB 크기(때로 더 큰)의 **고정 크기 블록**이나 **페이지**로 나누고 한 번에 한 번에 하나의 페이지에 읽기 또는 쓰기를 한다. 디스크가 고정 크기 블록으로 배열되기 때문에 이런 설계는 근본적으로 하드웨어와 더 밀접한 관련이 있다.




        - 각 페이지는 주소나 위치를 이용해 식별할 수 있다. 이 방식으로 하나의 페이지가 다른 페이지를 참조할 수 있다. (페이지 참조가 페이지 트리를 구성할 수 있다.)
        - 한 페이지는 B 트리의 **루트(root)** 로 지정된다. 색인에서 키를 찾으려면 루트에서 시작한다. 페이지는 여러 키와 하위 페이지의 참조를 포함한다.
          각 하위 페이지는 키가 계속 이어지는 범위를 담당하고 참조 사이의 키는 해당 범위 경계가 어디인지 나타낸다.
        - 최종적으로는 개별 키(**리프 페이지**(leaf Page))를 포함하는 페이지에 도달한다.이 페이지는 각 키의 값을 포함하거나 값을 찾을 수 있는 페이지 참조를 포함한다.
        - B 트리의 한 페이지에서 하위 페이지를 참조하는 수를 **분기 계수**라고 부른다.(ref 의 수 위에 예시에선 5)
        - B 트리에 존재하는 키의 값을 갱신하려면 키를 포함하고 있는 리프 페이지를 검색하고 페이지의 값을 바꾼 다음 페이지를 디스크에 다시 기록한다. 새로운 키를 추가하려면 새로운 키를 포함하는 범위의 페이지를 찾아 해당 페이지에 키와 값을 추가한다. 새로운 키를 수용한 페이지에 충분한 여유 공간이 없다면 페이지 하나를 반쯤 채워진 페이지 둘로 나누고 상위 페이지가 새로운 키 범위의 하위 부분들을 알 수 있게 갱신한다.
            - 이 알고리즘은 트리가 계속 균형을 유지하는 것을 보장한다. n 개의 키를 가진 B 트리는 깊이가 항상 O(log n) 이다.

## ▼ 트랜잭션 처리나 분석
## ▼ 컬럼 지향 저장소
## ▼ 정리