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.
  - RDB 모델의 기본키 색인이 대표적
  - 가장 간단한 색인 전략은 키를 데이터 파일의 바이트 오프셋에 매핑해 인메모리 해시 맵을 유지하는 전략이다.

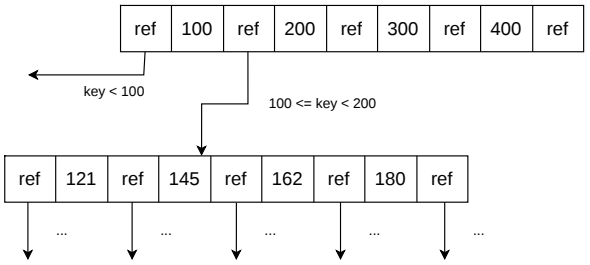
- 파일에 새로운 key-value 를 추가할 때마다 방금 기록한 데이터의 오프셋을 반영하기 위해 해쉬 맵도 갱신해야한다.
- 값을 조회할 때는 해시 맵을 사용해 인메모리 오프셋을 찾아 해당 위치를 구하고 값을 읽는다.
- 이 방법은 매우 단순해 보이지만, 실제로 많이 사용하는 접근법이다. ((비트캐스크)가 기본적으로 사용하는 방식)

- 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.

- 신뢰할 수 있는 B 트리 만들기
    - B 트리의 기본적인 쓰기 동작은 새로운 데이터를 디스크 상의 페이지에 덮어쓴다. 이 동작은 덮어쓰기가 페이지 위치를 변경하지 않는다고 가정한다.
      즉, 페이지를 덮어쓰더라도 페이지를 가르키는 모든 참조는 온전하게 남는다.
    - 일부 동작은 여러 다양한 페이지의 덮어쓰기를 필요로 한다. 예를 들어 삽입 때문에 페이지가 너무 많아, 페이지를 나눠야 한다면 분할된 두 페이지를 기록하고 두 하위 페이지의 참조를 갱신하게끔 상위 페이지를 덮어쓰기 해야 한다.
      일부 페이지만 기록하고 DB 가 고장나면 결국 색인이 훼손되기 때문에 매우 위험한 동작이다. (고아 페이지 - 어떤 페이지와도 부모 관계가 없는 페이지)
    - DB 고장 상황에서 스스로 복구할 수 있게 만들려면 일반적으로 디스크 상에 **쓰기 전 로그 (write-ahead-log, WAL)(재실행 로그)**라고 하는 데이터 구조를 추가해 B 트리를 구현한다.
      쓰기 전 로그는 트리 페이지에 변경된 내용을 적용하기 전에 모든 B 트리의 변경 사항을 기록하는 추가 전용 파일이다.
      DB 가 고장 이후 복구시 일관성 있는 상태로 B 트리를 다시 복원하는데 사용한다.
    - 같은 자리의 페이지를 갱신하는 작업은 동시성 제어를 필요로 한다. 그렇지 않다면 스레드가 일관성이 깨진 상태의 트리에 접근할 수 있다.
      동시성 제어는 보통
      래치(Latch)(가벼운 잠금)로 트리의 데이터 구조를 보호한다.
    - 위 내용들에서 LSM 트리와 매우 다른점 을 보여준다.
      LSM 트리의 색인은 파일에 추가만 할 뿐이고 같은 위치의 파일 변경은 하지 않는다.
      LSM 은 동시성 문제가 없다. 백그라운드에서 모든 병합을 수행하고 원자적으로 새로운 새그먼트로 변경하기 때문이다.

- B 트리 최적화
    - 페이지 덮어 쓰기와 고장 복구를 위한 WAL 유지 대신 일부 DB 는 쓰기 시 복사(copy -on-write scheme)를 사용한다. 변경된 페이지는 다른 위치에 기록하고 트리에 상위 페이지의 새로운 버전을 만들어 새로운 위치를 가르키게 한다. 이 접근 방식은 동시성 제어에도 유용하다.
    - 페이지에 전체 키를 저장하는 게 아니라 키를 축약해 쓰면 공간을 절약할 수 있다. 특히 트리 내부 페이지에서 키가 키 범위 사이의 경계 역할을 하는 데 충분한 정보만 제공하면 된다. 페이지 하나에 키를 더 많이 채우면 트리는 더 높은 분기 계수를 얻는다. 그러면 트리 깊이 수준을 낮출 수 있다.

- 일반적으로 페이지는 디스크 상 어디에나 위치할 수 있다. 키 범위가 가까운 페이지들이 디스크 상에 가까이 있어야 할 필요가 없기 때문이다. 질의가 정렬된 순서로 키 범위의 상당 부분을 스캔해야 한다면 모든 페이지에 대해 디스크 찾기가 필요하기 때문에 페이지 단위 배치는 비효율적이다. 따라서 많은 B 트리 구현에서 리프(leadf) 페이지를 디스크 상에 연속된 순서로 나타나게끔 트리를 배치하려 시도한다. 하지만 트리가 커지면 순서를 유지하기 어렵다. 반대로 LSM 트리는 병합하는 과정에서 저장소의 큰 세그먼트를 한 번에 다시 씀으로, 디스크에서 연속된 키를 유지하기가 더 쉽다.
- 트리에 포인터를 추가한다. 예를 들어 각 리프 페이지가 양쪽 형제 페이지에 대한 참조를 가지면 상위 페이지로 다시 이동하지 않아도 순서대로 키를 스캔 할 수 있다.
- **프랙탈 트리(fractal tree)** 같은 B 트리 변형은 디스크 찾기를 줄이기 위해 로그 구조화 개념을 일부 빌렸다.

- B 트리와 LSM 트리 비교
  - 간단한 표

	LSM 트리	B 트리
구현 난이도	쉬움	어려움
읽기 성능	나쁨 (각 컴팩션 단계에 있는 여러 가지 데이터구조와 SS 테이블을 모두 조회 해야함)	좋음
쓰기 성능	좋음	나쁨 (트리 하나로 구현됨으로 쓰기 이벤트시 동시성 제어 로직을 추가 해야함)

  - LSM 저장소 장점
    - B 트리 색인은 모든 데이터 조각을 최소한 두 번 기록해야 한다. 쓰기 전 로그 한 번과 트리 페이지에 한 번(페이지 분리 시 다시 기록)이다.
    - LSM 색인 또한 SS 테이블의 반복된 컴팩션과 병합으로 인해 여러 번 데이터를 다시 쓴다. DB에 쓰기 한 번이 DB 수명 동안 디스크에 여러 번의 쓰기를 야기하는 이런 효과를 **쓰기 증폭(write amplification)**이라 한다.
    - 쓰기가 많은 애플리케이션에서 성능 병목은 DB가 디스크에 쓰는 속도 일 수 있다.
      쓰기 증폭은 바로 성능 비용이다. 저장소 엔진이 디스크에 기록할수록 디스크 대역폭 내 처리할 수 있는 초당 쓰기는 점점 줄어든다.
    - LSM 트리는 보통 B 트리보다 쓰기 처리량을 높게 유지할 수 있다.
      LSM 트리가 상대적으로 쓰기 증폭이 더 낮고, 트리에서 여러번 덮어쓰는 것이 아닌, 순차적으로 컴팩션된 SS 테이블 파일을 쓰기 때문임.
    - LSM 트리는 압축률이 더 좋다.
      B 트리 저장소 엔진은 파편화로 인해 사용하지 않는 디스크 공간 일부가 남는다. 페이지를 나누거나 로우가 기존 페이지에 맞지 않을 때 페이지의 일부 공간은 사용하지 않게 된다.
      LSM 트리는 페이지 지향적이지 않고 주기적으로 파편화를 업애기 위해 SS 테이블을 다시 기록하기 때문에 저장소 오버헤드가 더 낮다.
  - LSM 저장소 단점
    - 컴팩션 과정이 때로는 진행 중인 읽기와 쓰기의 성능에 영향을 준다.
      저장소 엔진은 컴팩션을 점진적으로 수행하고 동시 접근의 영향이 없게 수행하려 한다. (컴팩션 연산이 끝날 때까지 요청을 대기)
    - 높은 쓰기 처리량이 디스크의 쓰기 대역폭(유한함)에 관련된다. 초기 쓰기 (로깅, 멤테이블을 디스크로 방출)와 백그라운드에서 수행되는 컴팩션 스레드가 이 대역폭을 공유해야 한다. DB 가 커질 수록 컴팩션을 위해 더 많은 디스크 대역폭이 필요함.
    - 쓰기 처리량이 높음에도 컴팩션 설정을 주의 깊게 하지 않으면 컴팩션이 유입 쓰기 속도를 따라갈 수 없다. 이 경우 디스크 상에 병합되지 않은 세그먼트 수는 디스크 공간이 부족할 때까지 증가한다. (세그먼트 파일이 증가해 읽기 성능도 저하됨)
  - B 트리의 장점
    - 각 키가 색인의 한 곳에만 정확하게 존재한다는 점이다. 반면 로그 구조화 저장소 엔진은 다른 세그먼트에 같은 키의 다중 복사본이 존재할 수 있다. 이런 측면 때문에 강력한 트랜잭션 시멘틱를 제공하는 DB에는 B 트리가 훨씬 매력적이다. 많은 관계형 DB에서 트랜잭션 격리(transactional isolation)는 키 범위의 잠금을 사용해 구현한 반면 B 트리 트리 색인에서는 트리에 직접 잠금을 포함한다.
  - B 트리는 DB 아키텍처에 깊게 뿌리내렸다. 그리고 많은 작업 부하에 대해 지속적으로 좋은 성능을 제공함으로 B 트리가 사라질 가능성은 거의 없다.
    새로운 데이터 저장소에서는 로그 구조화 색인이 점점 인기를 끌고 있다.

- **기타 색인 구조**
  - **보조 색인(secondary index)**
    - RDB 에서 보통 효율적으로 조인을 수행하는 데 결정적인 역할을 한다.
    - 키-값 색인에서 쉽게 생성할 수 있다. 기본키(PK) 색인과의 주요 차이점은 키가 고유하지 않다는 점이다. 즉 같은 키를 가진 많은 로우(문서, 정점)이 있을 수 있다.
      - 해결방법
        - 색인의 각 값에 일치하는 로우 식별자 목록을 만드는 방법
        - 로우 식별자를 추가해 각 키를 고유하게 만드는 방법이다.
  - 색인 안에 값 저장하기
    - 색인에서 키는 질의가 검색하는 대상이지만 값은 다음의 두 가지 중 하나에 해당한다. 값은 질문의 실제 로우(문서, 정점)거나 **다른 곳에 저장된 로우를 가르치는 참조**다. 후자의 경우 로우가 저장된 곳을 **힙 파일(heap file)**이라 하고 특정 순서 없이 데이터를 저장한다. 힙 파일 접근은

일반적인 방식이다. 여러 보조 색인이 존재할 때 데이터 중복을 피할 수 있기 때문이다. 각 색인은 힙 파일에서 위치만 참조하고 실제 데이터는 일정한 곳에 유지한다.

- 힙 파일 접근 방식은 키를 변경하지 않고 값을 갱신할 때 효율적이다. 새로운 값이 이전 값보다 많은 공간을 필요로 하지 않으면 레코드를 제자리에 덮어쓸 수 있다. 새로운 값이 많은 공간을 필요로 한다면 상황은 조금 더 복잡해진다. 힙에서 충분한 공간이 있는 새로운 곳으로 위치를 이동해야 하기 때문이다. 이런 경우 모든 색인이 레코드의 새로운 힙 위치를 가리키게끔 갱신하거나 이전 힙 위치에 전방향 포인터를 남겨둬야 한다.
- 색인에서 힙 파일로 다시 이동하는 일은 읽기 성능에 불이익이 너무 많기 때문에 어떤 상황에서는 색인 안에 바로 색인된 로우를 저장하는 편이 바람직하다. 이를 **클러스터드 색인(clustered index)** 라고 한다.
- 클러스터드 색인(색인 안에 모든 로우 데이터를 저장)과 비클러스터드 색인(색인 안에 데이터의 참조만 저장) 사이의 절출안을 커버링 색인(covering index) 나 포괄열이 있는 (index with included column)이라 한다. 이 색인은 색인 안에 테이블의 칼럼 일부를 저장한다. 이렇게 하면 색인만 사용해 일부 질의에 응답이 가능하다.

  - 다중 칼럼 색인
    - 가장 일반적인 유형은 **결합 색인(concatenated index) - (RDB 복합키)**이라 한다. 결합 색인은 하나의 칼럼에 다른 칼럼을 추가하는 방식으로 하나의 키에 여러 필드를 단순히 결합한다.
    - 다차원 색인은 한 번에 여러 칼럼을 질의하는 조금더 일반적인 방법이다. 특히 지리 공간 데이터(lat, lon)에 주용하게 사용된다. B 트리나 LSM 트리 색인은 이런 유형의 질의에 효율적으로 응답할 수 없다.
      - 일반적인 방법은 R 트리 처럼 전문 공간 색인을 사용하는 것
      - B 트리로 지리 공간 데이터를 색인하려면, 이차원 위치를 공간 채움 곡선을 이용해 단순 숫자로 변환한 다음 일반 B 트리 색인을 사용하는 것
  - 전문 검색과 퍼지 색인
    - 앞선 색인들은 정확한 값이나 정렬된 키의 값의 범위를 질의 한다. 철자가 틀린 단어와 같이 유사한 키에 대해서는 검색 할 수 없기 때문에 **애매모호한(fuzzy) 질의**에는 다른 기술이 필요하다.
      - ex) 전문 검색 엔진은 일반적으로 특정 단어를 검색할 때 해당 단어의 동의어로 질의를 확장한다. 그리고 단어의 문법적 활용을 무시하고 동일한 문서에서 서로 인접해 나타난 단어를 검색하거나 언어학적으로 텍스트를 분석해 사용하는 등 다양한 기능을 제공한다.

- 인메모리 DB (**모든 것을 메모리에 보관)**
  - 지속성을 목표로 하고, 그 목표를 이루기 위해 특수 하드웨어를 사용하거 디스크에 변경 사항의 로그를 기록하거나 디스크에 주기적인 스냅숏을 기록하거나 다른 장비에 인메모리 상태를 복제하는 방법이 있다.
  - 인메모리 DB가 재시작 되는 경우 특수 하드웨어를 사용하지 않는다면 디스크나 네트워크를 통해 복제본에서 상태를 다시 적재해야 한다.
  - 디스크는 전적으로 지속성을 위한 추가 전용 로그이고 읽기는 전적으로 메모리에서 제공한다.
  - 성능 장점은 디스크에서 읽지 않아도 된다는 사실 때문은 아니다. 디스크 기반 저장소 엔진도 운영체제가 최근에 사용한 디스크 블록을 메모리에 캐시하기 때문에 충분한 메모리를 가진 경우에는 디스크에서 읽을 필요가 없다. 오히려 인메모리 데이터 구조를 디스크에 기록하기 위한 형태로 부호화하는 오버헤드를 피할 수 있어서 더 빠를 수도 있다.
  - 성능 외에도 인메모리 DB 는 디스크 기반 색인으로 구현하기 어려운 데이터 모델을 제공한다. ex) 레디스는 우선순위 큐와 셋 같은 다양한 데이터 구조를 DB 같은 인터페이스로 제공한다. 또한 메모리에 모든 데이터를 유지하기 때문에 구현이 비교적 간단하다.
  - 인메모리 DB 아키텍처가 디스크 중심 아키텍처에서 발생하는 오버헤드 없이 가용한 메모리보다 더 큰 데이터셋을 지원하게끔 확장할 수 있다. 소위 **안티 캐싱(anti-cacheing)** 접근 방식은 메모리가 충분하지 않을 때 가장 최근에 사용하지 않은 데이터를 메모리에서 디스크로 내보내고 나중에 다시 접근할 때 메모리에 적재하는 방식으로 동작한다.

▼ 트랜잭션 처리나 분석

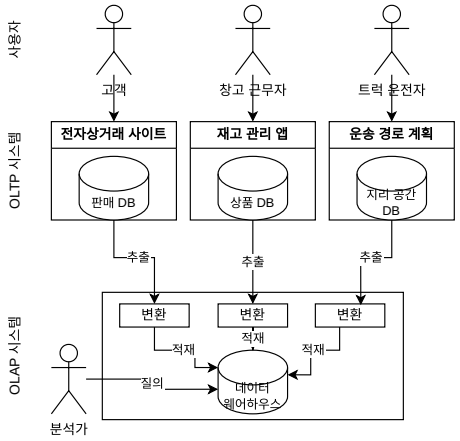
- **트랜잭션**
  - **논리 단위 형태로서 읽기와 쓰기 그룹을 나타낸다.**
  - 트랜잭션이 반드시 ACID (원자성(atomicity), 일관성(consistency), 격리성(isolation), 지속성(durability)) 속성을 가질 필요는 없다.
    트랜잭션 처리는 주기적으로 수행되는 일괄 처리 작업과 달리 클라이언트가 지연 시간이 낮은 읽기와 쓰기를 가능하게 한다는 의미다.
  - **온라인 트랜잭션 처리(online transcation processing, OLTP)**
    - 보통 애플리케이션은 색인(index)을 사용해 일부 키에 대한 적은 수의 레코드를 찾는다. 레코드는 사용자 입력을 기반으로 삽입되거나 갱신된다.
      이런 애플리케이션은 대화식이기 때문에 이 접근 패턴을 온라인 트랜잭션 처리라고 한다.
  - **데이터 분석(data analytic)**
    - 데이터 분석은 트랜잭션과 접근 패턴이 매우 다르다. 보통 분석 질의는 많은 수의 레코드를 스캔해 레코드당 일부 칼럼만 읽어 집계 통계를 계산해야 한다.
      이런 질의는 비즈니스 분석가가 작성하여 경연진이 더 나은 의사 결정을 도와주는 보고서를 제공한다. (비즈니스 인텔리전스)
    - 이런 데이터베이스 사용 패턴을 **온라인 분석 처리(online analytic processing, OLAP)** 라고 한다.
  - 특성 비교

특성	트랜잭션 처리 시스템 (OLTP)	분석 시스템 (OLAP)
주요 읽기 패턴	질의당 적은 수의 레코드, 키 기준으로 가져옴	많은 레코드에 대한 집계

주요 쓰기 패턴	임의 접근, 사용자 입력을 낮은 지연 시간으로 기록	대규모 불러오기(bulk import, ETL) 또는 이벤트 스트림
주요 사용처	웹 애플리케이션을 통한 최종 사용자/소비자	의사결정 지원을 위한 내부 분석가
데이터 표현	데이터의 최신 상태	시간이 지나며 일어난 이벤트 이력
데이터셋 크기	기가바이트에서 테라바이트	테라바이트에서 페타바이트

    - 처음에는 OLTP, OLAP 를 위해 동일한 DB를 사용했다.
      이후에 OLTP 시스템을 분석 목적으로 사용하지 않고 개별 DB에서 분석을 수행하는 경향을 보였다. 이 개별 DB를 **데이터 웨어하우스(Data warehouse)** 라 부른다.

- **데이터 웨어하우징**
    - OLTP 시스템은 대부분 운영이 중요하기 때문에, 일반적으로 높은 가용성과 낮은 지연 시간의 트랜잭션 처리를 기대한다. → DB 관리자는 OLTP DB를 보호하려고 한다. → 비즈니스 분석가가 OLTP DB에 즉석 분석 질의(자원을 많이 먹음 - 비쌈)를 실행하는 것을 꺼려한다. 분석 질의가 데이터셋의 많은 부분을 스캔해 이와 동시에 실행되는 트랜잭션의 성능을 저하시킬 가능성이 있기 때문이다.
    - 반대로 **데이터 웨어하우스**는 분석가들이 OLTP 작업에 영향을 주지 않고 마음껏 질의할 수 있는 개별 DB다. 데이터 웨어하우스는 사내의 모든 다양한 OTLP 시스템에 있는 데이터의 읽기 전용 복사본이다. 데이터는 OLTP DB에서 (주기적인 데이터 덤프나 지속적인 갱신 스트림을 사용해) **ETL(Extract-Transform-Load)** 과정을 거쳐서 적재된다.
    - ETL



        - 분석을 위해 OLTP 시스템에 직접 질의하지 않고 개별 데이터 웨어하우스를 사용하는 큰 장점은 분석 접근 패턴에 맞게 최적화할 수 있다는 점이다.

- **OLTP DB 와 데이터 웨어하우스의 차이점**
    - SQL 은 일반적으로 분석 질의에 적합하기 때문에 데이터 웨어하우스의 데이터 모델은 일반적인 관계형 모델을 사용한다.
    - 데이터 웨어하우스와 관계형 OLTP DB 는 둘 다 SQL 질의 인터페이스를 지원하기 때문에 비슷해 보인다. 하지만 각각 매우 다른 질의 패턴에 맞게 최적화됨으로, 시스템 내부는 완전히 다르다.

- **분석용 스키마: 별 모양와 눈꽃송이 모양 스키마**
    - 데이터 모델은 트랜잭션 처리 영역에서 애플리케이션의 필요에 따라 광범위하고 다양하게 사용된다. 반면 분석에서는 데이터 모델의 다양성이 훨씬 적다. 많은 데이터 웨어하우스는 **별 모양 스키마(star schema)(차원 모델링(dimensional modeling)** 로 알려진 상당히 정형화된 방식을 사용한다.
    - **사실 테이블 (fact table)**
        - 스키마 중심에 있는 테이블
        - row 는 특정 시각에 발생한 이벤트에 해당한다.
    - **차원 테이블(dimension table)**
        - 사실 테이블의 다른 컬럼은 다른 테이블 가르키는 외래 키 참조다.
        - 날짜와 시간도 보통 차원 테이블에서 표현된다.
    - 사실 테이블의 각 로우는 이벤트를 나타내고 차원 테이블은 이벤트의 속성인 누가, 언제, 어디서, 무엇을, 어떻게, 왜 등을 나타낸다.
    - 별 모양 스키마
        - 테이블 관계가 시각화될 때 사실 테이블이 가운데에 있고 차원 테이블로 둘러싸고 있다는 사실에서 비롯됐다.
    - 눈꽃송이 모양 스키마
        - 별 모양 스키마의 변형을 칭하며, 차원이 하위차원으로 더 세분화된다. (더 정규화되어있다)
    - 일반적으로 데이터 웨어하우스에서 테이블은 보통 폭이 매우 넓다.

▼ 컬럼 지향 저장소

- **칼럼 지향 저장소**
  - 사실 테이블은 칼럼이 보통 100개 이상이지만 일반적인 데이터 웨어하우스 질의는 한 번에 4, 5개 칼럼만 접근한다.
  - EX) 사람들이 요일에 따라 신선 과일을 사고 싶어하는지 사탕을 더 사고 싶어하는지 분석

```
SELECT
    dim_date.weekday, dim_product.category,
    SUM(fact_sales.quantity) as quantity_sold
FROM fact_sales
    JOIN dim_date ON fact_sales.date_key = dim_date.date_key
    JOIN dim_product ON fact_sales.product_sk = dim_product.product_sk
WHERE
    dim_date.year = 2013 AND
    dim_product.category IN ('Fresh fruit', 'Candy')
GROUP BY
    dim_date.weekday, dim_product.category;
```

    - 대부분의 OLTP DB 에서 저장소는 로우 지향 방식으로 데이터를 배치한다. 테이블에서 한 로우의 모든 값은 서로 인접하게 저장된다.
    - fact_sales.date_key와 fatc_sales.product_sk 칼럼 모두 혹은 둘 중 하나에 색인이 있다고 하자. 이 색인은 저장소 엔진에 특정 날짜나 특정 제품의 모든 판매 내용을 찾을 수 있는 위치를 알려준다. 하지만 로우 지향 저장소 엔진이 위 SQL 을 처리하기 위해서는 여전히 디스크로 부터 (컬럼 전체가 있는)모든 로우를 메모리로 적재한 다음 구문을 해석해 필요한 조건을 충족하지 않은 로우를 필터링해야 한다. 이 작업은 오래 걸릴 수 있다.
  - **칼럼 지향 저장소는 모든 값을 하나의 로우에 함께 저장하지 않는 대신 각 칼럼별로 모든 값을 함께 저장한다.** 각 컬럼을 개별 파일에 저장하면 질의에 사용되는 칼럼만 읽고 구분 분석하면 된다. 이 방식을 사용하면 작업량이 많이 줄어든다.
  - fact_sales 테이블

date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
140102	69	4	NULL	NULL	1	13.99	13.99
140102	69	5	19	NULL	3	14.99	9.99
140102	69	5	NULL	191	1	14.99	14.99
140102	74	3	23	202	5	0.99	0.89
140103	31	2	NULL	NULL	1	2.49	2.49
140103	31	3	NULL	NULL	3	14.99	9.99
140103	31	3	21	123	1	49.99	39.99
140103	31	8	NULL	233	1	0.99	0.99

    - 칼럼 저장소 배치

      date_key file contents: 140102, 140102, 140102, 140102, 140103, 140103, 140103 …
      product_sk file contents: 69, 69, 69, 74, 31, 31, 31, 31

      …

    - 칼럼 지향 저장소 배치는 각 칼럼 파일에 포함된 로우가 모두 같은 순서인 점에 의존한다.
- **칼럼 압축**
  - 질의에 필요한 칼럼을 디스크에서 읽어 적재하는 작업 외에도 데이터를 압축하면 디스크 처리량을 더 줄일 수 있다. 다행히 칼럼 지향 저장소는 대개 압축에 적합하다.
  - 많은 값이 반복해서 나타나면 압축을 하기 아주 적합하다. 칼럼 데이터에 따라 다양한 압축 기법을 사용할 수 있다. 그 중 한 가지 기법은 데이터 웨어하우스에서 특히 효과적인 **비트맵 부호화(bitmap encoding)** 이다.

```
칼럼 값:
product_sk: 69, 69, 69, 74, 31, 31, 31, 31
가능한 각 값에 대한 비트맵:
product_sk=31: 0 0 0 0 1 1 1 1
product_sk=69: 1 1 1 0 0 0 0 0
product_sk=74: 0 0 0 1 0 0 0 0
런 렝스 부호화(run-length encoding):
product_sk=31: 4 4   (0이 4개, 1이 4개)
product_sk=69: 0 3   (0이 0개, 1이 3개, 나머지 0)
producy_sk=74: 3 1   (0이 3개, 1이 1개, 나머지 0)
```

    - 보통 칼럼에서 고유 값의 수는 로우 수에 비해 적다. (ex - 판매 거래 row 는 수십억 개가 있을 수 있지만, 고유 제품은 100,000 만 있을 수 있다). 그러면 n 개의 고유 값을 가진 칼럼을 가져와 n개의 개별 비트맵으로 변환할 수 있다. 고유 값 하나가 하나의 비트맵이고 각 로우는 한 비트를 가진다. 만약 로우가 해당 값을 가지면 비트는 1이고 아니면 0 이다

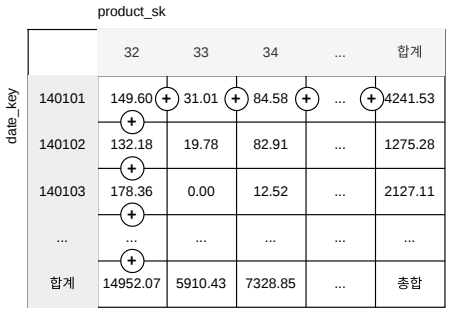
    - n이 매우 작으면 이 비트맵은 로우당 하나의 비트로 저장할 수 있다. 하지만 n 이 더 크면 대부분의 비트맵은 0이 더 많다. 이런 경우 비트맵을 추가적으로 런 렝스 부호화 할 수 있다. 이 방법을 사용함녀 칼럼의 부호화를 현저히 줄일 수 있다.
    - 이런 비트맵 색인은 데이터 웨어하우스에서 일반적으로 사용되는 질의 종류에 매우 적합하다.

```
WHERE product_sk IN (31, 74);
    product_sk = 31, product_sk = 74 에 대한 비트맵 2 개를 적재하고
    두 비트맵의 비트를 OR 계산한다. 매우 효율적으로 수행가능

WHERE product_sk = 31 AND store_sk = 3
    product_sk = 31, stroe_sk = 3으로 비트맵을 적재하고 비트맵의 비트 AND 계산
    이 계산은 각 칼럼에 동일한 순서로 로우가 포함되기 때문에 가능하다.
    한 컬럼의 비트맵에 있는 k 번째 비트는 다른 칼럼의 비트맵에서 k번째 비트와 같은 로우.
```

- **메모리 대역폭과 벡터화 처리**
    - 수백만 로우를 스캔해야 하는 데이터 웨어하우스 질의는 디스크로부터 메모리로 데이터를 가져오는 대역폭이 큰 병목이다.
        - 하지만 이 병목이 유일한 것은 아니다. 분석용 DB 개발자는 메인 메모리에서 CPU 캐시로 가는 대역폭을 효율적으로 사용하고 CPU 명령 처리 파이프라인에서 분기 예측 실패와 버블을 피하며 최신 CPU에서 단일 명령 다중 데이터 명령을 사용하게끔 신경 써야 한다.
    - 디스크로부터 적재할 데이터 양 줄이기 외에도 칼럼 저장소 배치는 CPU 주기를 효율적으로 사용하기에 적합하다.
        - 예를 들어 질의 엔진은 압축된 칼럼 데이터를 CPU의 L1 캐시에 딱 맞게 덩어리로 나누어 가져오고 이 작업을 (함수 호출이 없는)타이트 루프에서 반복한다. CPU는 함수 호출이 많이 필요한 코드나 각 레코드 처리를 위해 분기가 필요한 코드보다 타이트 루프를 훨씬 빨리 실행할 수 있다.
        - 칼럼 압축을 사용하면 같은 양의 L1 캐시에 칼럼의 더 많은 로우를 저장할 수 있다. 앞에서 설명한 비트 AND 와 OR 같은 연산자는 압축된 칼럼 데이터 덩어리를 바로 연산할 수 있게 설계할 수 있다. 이런 기법을 **벡터화 처리** 라고 한다.
- **칼럼 저장소의 순서 정렬**
    - 칼럼 저장소에서 로우가 저장되는 순서가 반드시 중요하지는 않다. 삽입된 순서로 저장하는 방식이 가장 쉽다. 새로운 로우를 삽입하는 작업은 각 칼럼 파일에 덧붙여 추가하는 것을 의미하기 때문이다.
    - 하지만 이전의 SS 테이블에서 했던 것처럼 순서를 도입해 이를 색인 매커니즘으로 사용할 수 있다. 각 칼럼을 독립적으로 정렬할 수는 없다. 그렇게 하면 더 이상 컬럼의 어떤 항목이 동일한 로우에 속하는지 알 수 없기 때문이다.
    - **칼럼별로 저장됐을지라도 데이터는 한 번에 전체 로우를 정렬해야 한다**. DBA는 공통 질의에 대한 지식을 사용해 테이블에서 정렬해야 하는 칼럼을 선택할 수 있다. ex) 질의가 시간 범위를 목표로 한다면 1차 정렬 키를 date_key 로 하는게 맞다. 그러면 질의 최적화기는 지난 달에 해당하는 로우만 스캔할 수 있으며 모든 로우를 스캔하기 보다 **훨씬 빠르다.**
    - 첫 번째 칼럼에서 같은 값을 가진 로우들의 정렬 순서를 두 번째 칼럼에서 정할 수 있다. ex) 질의가 date_key 가 1차 정렬키 라면 같은 날짜에 판매한 제품을 함께 그룹화 하게끔 product_st 를 보조 정렬 키로 하는게 합리적이다. (특정 날짜에 판매한 제품을 그룹화, 필터링 질의에 도움된다.)
    - **정렬된 순서의 또 다른 장점은 칼럼 압축에 도움이 된다**. 기본 정렬 칼럼에 고유 값을 많이 포함하지 않는다면 정렬한 후 기본 정렬 칼럼은 연속해서 같은 값을 연속해서 길게 반복된다. 그럼 런 렝스 부호화는 엄청 효율이 좋아진다. (첫 번째 정렬 키에서 가장 강력함)
- **다양한 순서 정렬**
    - 다양한 질의는 서로 다른 정렬 순서의 도움을 받으므로 같은 데이터를 다양한 방식으로 정렬해 저장한다면 어떨까? 하나의 장비가 고장나도 데이터를 잃지 않으려면 데이터를 여러 장비에 복제해 두는 작업이 필요하다. 복제 데이터를 서로 다른 방식으로 정렬해서 저장하면 질의를 처리할 때 질의 패턴에 가장 적합한 버전을 사용할 수 있다.
    - 칼럼 지향 저장에서 여러 정렬 순서를 갖는 것은 로우 지향 저장에서 여러 2차 색인을 갖는 것과 약간 비슷하다.
      하지만 로우 지향 저장은 한 곳(힙 파일이나 클러스터드 색인)에 모든 로우를 유지하고 2차 색인은 일치하는 로우를 카르키는 포인터만 포함한다는 점이 큰 차이점이다. 칼럼 저장에서는 일반적으로 데이터를 가리키는 포인터가 없고, 단지 값을 포함하는 칼럼만 존재한다.
- **칼럼 지향 저장소에 쓰기**
    - 데이터 웨어하우스에서 이런 최적화(칼럼 저장소의 순서 정렬 등)는 합리적이다. 왜냐면, 대부분의 작업은 분석가가 수행하는 대량의 읽기 전용 질의이기 때문이다. 칼럼 지향 저장소, 압축, 정렬은 모두 읽기 질의를 더 빠르게 하지만 쓰기를 어렵게 한다는 단점이 있다.
    - B 트리 사용과 같은 제자리 갱신 접근 방식은 압축된 칼럼에서는 불가능하다. 정렬된 테이블의 중간에 있는 로우에 삽입을 원하는 경우 모든 칼럼 파일을 재작성해야 한다. 로우는 칼럼 안의 위치에 따라 식별되므로 삽입은 모든 칼럼을 일관되게 갱신해야 한다.
    - 다행히도 이번 장의 앞부분에서 LSM 트리라는 좋은 해결책을 이미 설명했다. 모든 쓰기는 먼저 인 메모리 저장소로 이동해 정렬된 구조에 추가하고 디스크에 쓸 준비를 한다. 인메모리 저장소가 로우 지향인지 칼럼 지향인지는 중요하지 않다. 충분한 쓰기를 모으면 디스크의 칼럼 파일에 병합하고 대량으로 새로운 파일에 기록한다.
    - 질의는 디스크의 칼럼 데이터와 메모리의 최근 쓰기를 모두 조사해 두 가지를 결합해야 한다. 질의 최적화기는 이런 구별을 사용자에 드러내지 않는다. 분석가의 관점에서 삽입, 갱신, 삭제로 수정하는 데이터는 후속 질의에 즉시 반영한다.
- 집계 테이블 큐브와 구체화 뷰
    - 모든 데이터 웨어하우스가 칼럼 저장이 필수는 아니다. 전통적인 로우 지향 DB와 기타 아키텍처도 사용된다. 하지만 칼럼 저장소는 즉석 분석 질의에 대해 상당히 빠르기 때문에 급속하게 인기를 얻고 있다.
    - 간략하게 언급하고 넘어갈 만한 데이터 웨어하우스의 다른 측면으로 구체화 집계가 있다. 앞에서 설명한 데이터 웨어하우스 질의는 보통 SQL 에 COUNT, SUM, AVG, MIN, MAX 같은 집계 함수를 포함한다. 동일한 집계를 많은 다양한 질의에서 사용한다면 매번 원시 데이터를 처리하는 일은 낭비다. 자주 질의되는 일부 COUNT, SUM 을 캐시하는건 어떨까?

- 이런 캐시를 만드는 한 가지 방법이 **구체화 뷰(materialized view)**다. 관계형 데이터 모델에서는 이런 캐시를 대개 표준 (가상) 뷰로 정의한다. 표준 뷰는 테이블 같은 객체로 일부 질의의 결과가 내용이다. 차이점으로 구체화 뷰는 디스크에 기록된 질의 결과의 실제 복사본이지만, 가상 뷰는 단지 질의를 작성하는 단축키일 뿐이다. 가상 뷰에서 읽을 때 SQL 엔진은 원래 질의로 즉석에서 확장하고 나서 질의를 처리한다.
- 원본 데이터를 변경하면 구체화 뷰를 갱신해야 한다. 구체화 뷰는 원본 데이터의 비정규화된 복사본이기 때문이다. DB는 이 작업을 자동으로 수행할 수 있다. 하지만 이런 갱신으로 인한 쓰기는 비용이 비싸기 때문에 OLTP DB에서는 구체화 뷰를 자주 사용하지 않는다. 데이터 웨어하우스는 읽기 비중이 크기 때문에 구체화 뷰를 사용하는 것이 합리적이다.
- **데이터 큐브(data cube)** 또는 **OLAP 큐브**라고 알려진 구체화 뷰는 일반화된 구체화 뷰의 특별 사례다.



  - 이제 각 사실은 2차원 테이블에만 외래 키를 가진다고 가정해보자, 위에서 외래 키가 date, product 이다. 각 셀은 날짜와 제품을 결합한 모든 사실의 속성의 집계 값을 포함한다. 각 로우나 칼럼에 같은 집계를 적용할 수 있고 1차원으로 축소한 요약(날짜와는 관계없는 제품별 판매량이나 제품과 관계없는 날짜별 판매량)을 얻을 수 있다.
  - 장점은 특정 질의를 효과적으로 미리 계산했기 때문에 해당 질의를 수행시 매우 빠르다.
  - 단점은 원시 데이터에 질의하는 것과 동일한 유연성이 없다는 점이다.

▼ 정리

- DB 가 어떻게 저장과 검색을 다루는지 근본적인 내용을 다뤘다.
  - B 트리, LSM 트리의 저장, 검색 방식, 특징
- 트랜잭션 처리 최적화(OLTP), 분석 최적화(OLAP)
  - OLTP 시스템은 사용자 대면임으로 대량의 요청을 받을 수 있다. 부하를 처리하기 위해 보통 애플리케이션이 각 질의마다 작은 수의 레코드만 다룬다. 애플리케이션은 키의 일부만 사용하는 레코드를 요청하고 저장소 엔진은 요청한 키의 데이터를 찾기 위해 색인을 사용한다. 이 경우는 대개 디스크 탐색이 병목이다.
  - 데이터 웨어하우스와 유사한 분석 시스템은 최종 사용자가 아닌 비즈니스 분석가가 사용한다. OLTP 시스템보다 훨씬 더 적은 수의 질의를 다루지만, 각 질의는 대개 매우 다루기 어렵고 짧은 시간에 수백만 개의 레코드를 스캔해야 한다. 이 경우 대개 디스크 대역폭이 병목이다. 칼럼 지향 저장소는 이런 종류의 작업 부하를 처리시 적합하다.
  - OLTP 관점
    - 로그 구조화 관점에서 파일에 추가와 오래된 파일의 삭제만 허용하고 한 번 쓰여진 파일은 절대 갱신하지 않는다. (LSM 트리, 레벨 DB, HBASE 등)
      비교적 최근에 개발되었고, 핵심은 임의 접근 쓰기를 체계적으로 디스크에 순차 쓰기로 변경된 것.
    - 제자리 갱신 관점에서 덮어쓰기를 할 수 있는 고정 크기 페이지의 셋으로 디스크를 다룬다. (B 트리)
  - OLAP 관점 (칼럼 저장소)
    - 질의가 많은 수의 로우를 순차적으로 스캔해야 한다면 색인을 사용하는 방법은 적절하지 않다.
    - 대신 질의가 디스크에서 읽는 데이터의 양을 최소화하기 위해 데이터를 매우 작게 부호화하는 일이 중요해졌다.