데이터베이스: 강의노트 07

A. Silberschatz, H. Korth, S. Sudarshan
Database System Concepts,
Fourth Edition, McGraw-Hill, 2002.

Part IV. Data Storage and Querying

11 저장과 파일 구조

11.5 저장장치 접근

- 데이터베이스는 여러 파일로 사상되어 디스크에 저장되며, 이 파일들은 보통 테이프에 백업된다.
- 각 파일은 고정된 크기의 저장 단위인 블록으로 분할되어 있으며, 이 단위는 저장뿐만 아니라 전송의 단위가 된다.
- 한 블록에는 보통 여러 개의 데이터 항목이 저장된다.
- 하나의 데이터 항목이 여러 블록에 걸쳐 저장되면 이 데이터 항목을 접근하기 위한 디스크 입출력이 많아진다.
- 데이터베이스의 저장장치 접근 측면에서의 목적은 **디스크와 주기억장치 간에 블록 이동을 최소화하는 것**이다.
- 디스크 접근을 최소화하는 한 가지 방법은 많은 블록을 주기억장치에 유지하는 것이다. 그러나 모든 블록을 모두 주기억장치에 유지할 수 없다. **버퍼**는 디스크에 있는 블록의 복사본을 유지하는 데이터베이스가 사용하는 주기억장치 영역이다.
- 데이터베이스의 하위시스템 중 버퍼 공간의 할당을 책임지는 하위시스템을 **버퍼 관리자(buffer manager)**라 한다.

11.5.1 버퍼 관리자

- 디스크에 있는 블록이 필요하면 버퍼 관리자에게 요청을 한다. 요청한 블록이 이미 버퍼에 있으면 블록의 주소를 주고, 없으면 블록을 버퍼에 적재한 다음에 그것의 주소를 준다. 이 때 필요하면 적재되어 있는 기존 블록을 덮어쓸 수 있다. 덮어쓰이게 되는 블록이 디스크에 반영되지 않은 수정을 가지고 있으면 덮어쓰기 전에 디스크를 갱신해야 한다.
- 데이터베이스의 크기는 매우 크므로 버퍼 관리자는 운영체제의 주기억장치 관리자 보다 더 복잡한 기술이 필요하다.
  - 교체 알고리즘: 운영체제는 보통 **LRU(Least-Recently Used)** 방식을 사용한다. 그러나 데이터베이스는 이 방식보다 향상된 방법을 적용할 수 있다.
  - 고정 블록(pinned block): 디스크에 쓰여지지 못하도록 제한되어 있는 블록을 말한다. 보통 시스템 실패(crash)로부터 복구하기 위해서는 블록이 디스크에 쓰여지지 시기를 제한해야 한다.
  - 블록의 강제 쓰기(forced output of blocks): 보통 블록은 그 블록이 교체되어야 할 때까지 디스크에 쓰여지지 않는다. 그러나 때로는 교체되지 않아도 디스크에 강제로 써야 할 경우가 있다.

11.5.2 버퍼 교체 알고리즘

- 버퍼 교체 알고리즘의 목적은 디스크 접근을 최소화하는 것이다. 그러나 미래 접근을 정확하게 예측할 수 없으며, 일반적으로 기존 접근을 바탕으로 미래 접근을 추측할 수밖에 없다.
- LRU: 가장 오래전에 사용된 블록을 교체하는 방법
- LFU(Least Frequently Used): 가장 적게 사용된 블록을 교체하는 방법
  - 문제점: 오래 전에는 자주 사용되었지만 최근에는 거의 사용되지 않는 블록은 계속 교체되지 않는다.
- 운영체제는 주기억장치를 관리할 때 LRU 방법을 주로 사용한다. 그러나 데이터베이스는 운영체제보다는 더 정확하게 미래의 접근 패턴을 예측할 수 있다. 특히 가까운 미래에 대해서는 정확하게 예측이 가능한 경우가 많다.
- 예) 다음과 같은 관계 대수 표현식을

$$borrower \bowtie customer$$

처리하는 의사코드(pseudocode)가 다음과 같다고 하자.

*borrower*의 각 투플 b마다
  *customer*의 각 투플 c마다
    b[고객명]=c[고객명]이면
      새 투플 x를 다음과 같이 정의하라:
      x[고객명]=b[고객명];
      x[대출번호]=b[대출번호];
      x[고객-도시]=c[고객-도시];
      x[고객-거리]=c[고객-거리];
      x를 *borrower* $\bowtie$ *customer*에 포함

두 관계가 서로 다른 파일에 저장되어 있다고 가정하자.

*borrower*의 한 투플을 처리하였으면 더 이상 그 투플은 필요없다. 즉, *borrower*의 한 블록을 모두 사용하였으면 더 이상 이 블록은 필요없다. 따라서 *borrower*의 한 블록을 다 처리했으면 이 블록

은 즉시 교체될 수 있다. 이런 교체 전략을 **즉시-교체(toss-immediate)** 방법이라 한다.

*customer*의 투플은 *borrower*의 투플과 달리 각 *borrower*의 투플을 처리할 때마다 계속 사용된다. 즉, *customer*의 한 블록은 조인이 완료될 때까지 계속 필요하다. 그런데 저장공간이 부족하여 교체되어야 하면 가장 최근에 참조한 블록을 교체하는 것이 가장 바람직하다. 이것은 가장 최근에 참조한 블록이 가장 나중에 다시 참조되기 때문이다. 이런 교체 전략을 **MRU(Most-Recently Used)** 방법이라 한다.

- 버퍼 관리자는 통계적 자료를 참고할 수 있다.
  - 데이터베이스에서는 관계들에 대한 논리적, 물리적 스키마에 관한 정보를 유지하는 데이터 사전(data dictionary)을 가장 많이 접근한다. 따라서 버퍼 관리자는 데이터 사전 블록은 교체하지 않는다.
- 여러 사용자가 병행으로 데이터베이스를 사용하고 있다면 병행 제어(concurrency-control) 하위시스템은 데이터베이스의 일관성을 유지하기 위해 특정 요청들을 지연시킬 수 있다. 버퍼 관리자가 지연된 요청들의 내용을 알면 교체 알고리즘을 적용할 때 고려할 수 있다.
- 실패 복구(crash-recovery) 하위시스템은 교체 알고리즘에 엄격한 제약을 가한다. 어떤 블록이 수정되었으면 버퍼 관리자는 이 블록을 실패 복구 하위시스템의 허락 없이는 디스크에 쓸 수 없다.

11.6 파일 조직

- 파일은 논리적으로 일련의 레코드로 조직되어 있다.
- 레코드는 파일의 블록으로 사상된다.
- 블록의 크기는 디스크의 물리적 특성에 의해 결정된다. 그러나 레코드의 크기는 데이터에 따라 가변적이다. 보통 관계마다 다른 크기의 레코드를 사용한다.
- 보통 하나의 파일에는 같은 크기의 레코드만 저장한다.

11.6.1 고정된 크기의 레코드

- 예) *account* 관계의 한 레코드의 정의

```
typedef struct {
        char account_number[10];
        char branch_name[22];
        double balance;
} account;
```

  char 형이 1 바이트이고, double 형이 8 바이트이면 *account* 레코드의 크기는 40 바이트이다.

- 가장 단순한 방법: 첫 40 바이트에 첫 레코드, 그 다음 40 바이트에 그 다음 레코드를 저장하는 방법.
  - 문제점
    - 삭제가 어렵다. 삭제된 레코드를 표시하거나 다른 레코드로 채워야 한다. 마지막 레코드로 채울 수 있지만 이것은 두 개의 블록 접근이 필요하다. 보통 삭제보다는 삽입이 많으므로 빈 레코드는 나중에 삽입에 의해 채워지도록 하는 것이 더 바람직하다. 그러나 단순 표시는 채울 위치를 찾기가 어렵기 때문에 효율적이지 못하다.
    - 블록 크기가 40 바이트의 배수가 아니면 블록 경계에 걸친 레코드가 존재하게 된다.
- 일반적인 방법: 파일 앞부분에 파일 헤더를 사용하여 파일에 관한 여러 정보를 저장하는 방법
  - 보통 이 헤더에 비어 있는 첫 번째 레코드의 위치만 유지하고, 비어 있는 첫 레코드에는 비어 있는 그 다음 레코드의 위치를 유지한다. 즉, 연결 리스트로 비어 있는 레코드를 관리한다.
  - 삽입 요청에 대해서는 이 연결 리스트를 활용하며, 비어 있는 레코드가 없으면 파일 끝에 레코드를 추가한다.

11.6.2 가변 크기의 레코드

- 가변 크기의 레코드를 저장해야 할 경우
  - 한 파일에 여러 레코드 타입을 저장해야 하는 경우
  - 레코드 내에 하나 이상의 필드가 가변 크기를 가질 경우
  - 필드의 반복을 허용하는 레코드 타입을 저장해야 하는 경우: 다중값 속성
- 예) 한 지점의 계좌 정보를 관리하기 위한 가변 크기의 레코드

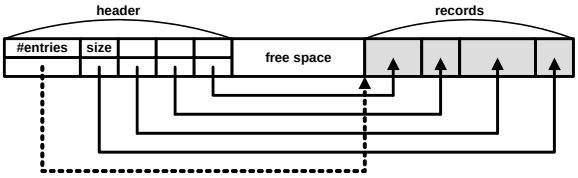
```
typedef struct {
        char account_number[10];
        double balance;
} account;

typedef struct {
        char branch_name[22];
        account account_info[];
} account-list;
```

  여기서 account_info는 가변 길이의 배열이다.

11.6.2.1 바이트 문자열 표현

- 레코드의 끝에 레코드의 끝을 표시하거나, 레코드 시작에 레코드의 길이를 저장하는 방법
- 단점
  - 삭제된 레코드의 공간을 다시 사용하기가 어렵다. 단편화가 발생한다.

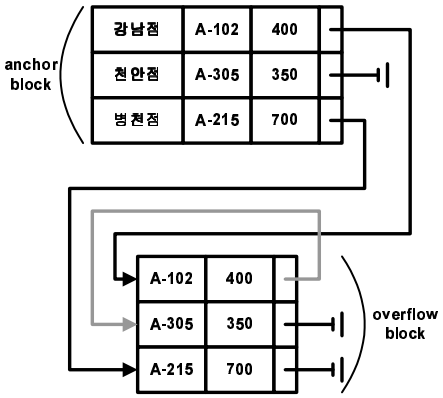


<그림 11.1> 슬롯 페이지 구조

  - 레코드에 데이터를 추가하기가 어렵다.

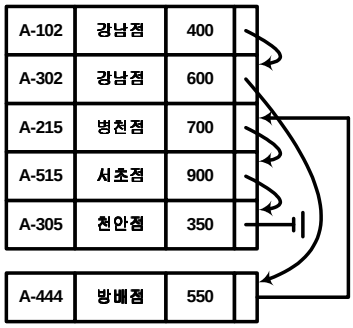
- 슬롯 페이지 구조(slotted-page structure)
  - 블록의 시작에 다음과 같은 정보를 보관한다.
    - 헤더에 있는 레코드의 엔트리의 수
    - 빈 공간의 끝 위치
    - 블록에 저장되어 있는 각 레코드의 위치와 크기
  - 실제 레코드는 블록의 끝부터 저장된다.
  - 빈 공간은 연속된 공간으로 항상 헤더와 저장되어 있는 마지막 레코드 사이에 유지된다.
  - 레코드가 삭제되거나, 한 레코드의 크기가 변경되면 블록을 재정리한다.

11.6.2.2 고정된 크기 표현



<그림 11.2> 시작 블록과 넘침 블록 구조

- 세 가지 방법
  - 한 레코드의 최대 크기가 정의되어 있으면 모든 레코드마다 이 크기를 차지하도록 저장하는 방법으로, 할당되었지만 사용되지 않는 공간은 특수 값으로 채우거나 레코드의 끝을 나타내는 값으로 채운다.
    - 대부분의 레코드가 최대 크기에 근접한 크기를 가질 경우에는 유용한 방법이다.
    - 문제점: 공간이 많이 낭비될 수 있다.
  - 고정된 크기의 레코드를 연결 리스트 형태로 연결하여 사용하는 방법



<그림 11.3> 순차 파일에서 삽입

    - 추가적으로 사용하는 레코드의 가변 필드 부분을 제외한 나머지 공간은 낭비된다.
  - 블록을 시작 블럭(anchor block)과 넘침 블럭(overflow block)으로 나누어 사용하는 방법
    - 시작 블럭에는 레코드의 고정 크기 필드와 가변 필드의 한 값만 저장한다.
    - 넘침 블럭에는 가변 필드의 값들만 연결 리스트 형태로 저장한다.
    - 시작 블럭과 넘침 블럭의 레코드 크기가 다르다.

11.7 파일 내에서 레코드 조직

- 힙 파일 조직(heap file organization): 레코드는 파일 내에 임의의 위치에 저장될 수 있다. 보통 하나의 관계는 하나의 파일에 저장한다.
- 순차 파일 조직(sequential file organization): 레코드는 지정된 필드(검색키)를 기준으로 정렬되어 저장된다.
- 해시 파일 조직(hashing file organization): 레코드의 몇 필드의 값을 해시 함수의 입력으로 사용하여 해시값을 계산한 다음에 이 값을 기준으로 저장된다.
- 클러스터 파일 조직(clustering file organization): 서로 다른 관계를 하나의 파일에 저장하며, 이 때 관련있는 데이터는 같은 블록에 저장된다.

11.7.1 순차 파일 조직

- 순차 파일 조직은 어떤 **검색키(search key)**를 기준으로 레코드들이 정렬되어 저장된다.
- 어떤 속성도 검색키가 될 수 있으며, 속성의 집합이 검색키가 될 수 있다. 꼭 주키가 검색키가 될 필요는 없다.
- 삽입: 물리적 순서와 논리적 순서가 일치하게 저장되어 있으므로 원래 삽입되어야 하는 위치에 삽입하기가 어렵다. 이를 위해 나머지 레코드를 모두 이동할 수 있지만 이것은 성능 측면에서 바람직하지 않다. 따라서 연결 리스트를 사용한다.

레코드가 삽입할 위치와 가장 가까운 곳에 있는 빈 공간에 삽입하고 포인터 값을 조정한다.

  - 물리적 순서와 논리적 순서가 많이 벗어나면 디스크 입출력 때문에 성능이 매우 떨어진다. 따라서 주기적으로 물리적 순서와 논리적 순서를 일치시켜 주어야 한다.

- 순차 파일 조직은 관계를 특정 순서로 읽어야 할 때 편리하며, 질의를 처리할 때도 도움이 될 수 있다.

11.7.2 클러스터 파일 조직

- 대부분의 관계형 데이터베이스는 각 관계를 별도 파일에 저장한다. 이것은 각 관계는 보통 고정된 크기의 레코드를 사용하며, 고정된 크기의 레코드는 단순한 파일 조직을 이용하여 쉽게 저장할 수 있기 때문이다. 더구나 단순한 파일 조직을 사용하면 이것을 조작하기 위한 코드도 단순하며, 코드의 양도 적다.

- 데이터베이스의 덩치가 커지고 복잡해지면 하나의 파일에 여러 관계를 저장하여 성능을 향상시키는 것이 필요할 수 있다.

- 대용량 데이터베이스는 파일 관리를 운영체제에게 의존하지 않고, 하나의 파일에 모든 관계를 저장하고, 이 파일을 스스로 관리한다.

- 예) 다음 질의를 고려하여 보자.

  select 계좌번호,고객명,고객-도시,고객-거리
  from *depositor,customer*
  where *depositor*.고객명=*customer*.고객명

  이 질의는 *depositor*와 *customer*를 자연 조인한다. 따라서 *depositor*의 각 투플마다 같은 고객명을 가진 *customer* 투플을 찾아야 한다. 최악의 경우 각 레코드가 다른 블록에 위치해 있으면 많은 디스크 접근이 필요하게 된다. 클러스터 파일 조직은 *depositor*의 각 투플을 관련 *customer*의 투플 근처에 저장한다. 따라서 보다 빠르게 그리고 적은 디스크 접근으로 조인을 할 수 있다. 그러나 이렇게 조직화되면 어떤 조인은 빠르게 처리할 수 있지만 다른 질의는 오히려 늦어질 수 있다. 이것을 극복하기 위해 *customer*의 투플들은 연결 리스트 형태로 연결하여 사용한다.

- 클러스터 파일 조직은 시스템에서 많이 사용될 질의를 기준으로 최적화되어 저장된다.

11.8 데이터 사전의 저장

- 데이터 사전에 유지되어야 하는 정보
  - 관계의 이름
  - 각 관계의 속성 이름
  - 각 속성의 도메인과 크기
  - 데이터베이스에 정의된 뷰의 이름과 그것의 정의
  - 무결성 제약 조건
  - 인가된 사용자 목록
  - 각 사용자의 통계 정보
  - 패스워드와 같은 사용자 인증 정보
  - 각 관계에 있는 투플의 수
  - 각 관계가 저장되어 있는 저장 구조
  - 각 관계가 저장되어 있는 파일
  - 한 파일에 모든 관계가 저장되어 있다면 관계의 각 투플이 파일 내에 어디에 저장되어 있는지에 대한 정보
  - 색인 정보