1. 블록 단위 I/O

기본적으로 데이터 I/O는 Block 단위로 수행된다.

* Oracle을 포함한 모든 DBMS에서 I/O는 블록(SQL Server 등 다른 DBMS는 페이지라는 용어를 사용) 단위로 이루어진다. 즉 하나의 레코드를 읽더라도 레코드가 속한 블록 전체를 읽는다. SQL 성능을 좌우하는 가장 중요한 성능지표는 액세스하는 블록 개수이며, 옵티마이저의 판단에 가장 큰 영향을 미치는 것도 액세스해야 할 블록 개수다. 블록 단위 I/O는 버퍼 캐시와 데이터 파일 I/O 모두에 적용된다.

- 데이터 파일에서 DB 버퍼 캐시로 블록을 적재할 때
- 데이터 파일에서 블록을 직접 읽고 쓸 때
- 버퍼 캐시에서 블록을 읽고 쓸 때
- 버퍼 캐시에서 변경된 블록을 다시 데이터 파일에 쓸 때

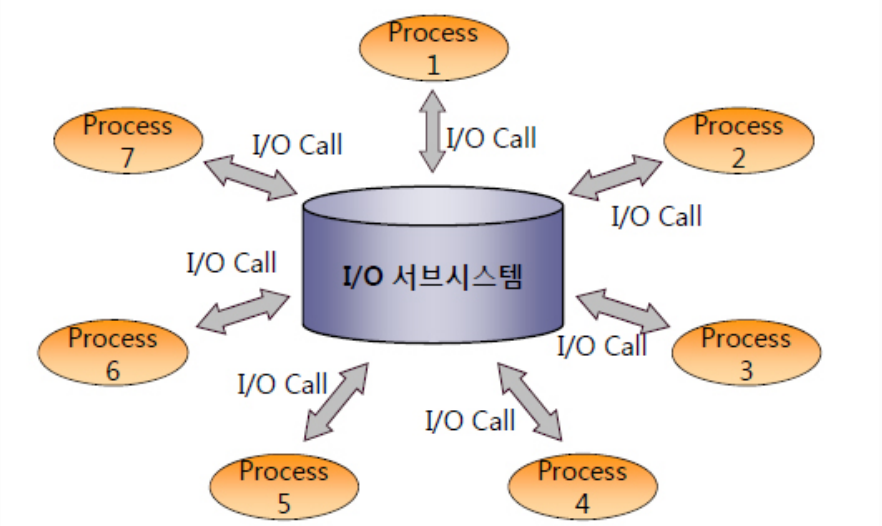
정확하의 중요성!

조회할 때 일반적으로 'full scan' 방식을 사용한다.
만약 어떤 컬럼이 들어가는 데이터의 특정 data value의 크기가 크다면, 한 block에 들어가는 레코드의 개수는 적다.
이에 따라 조회해야할 block 수가 많아지고, 해당 데이터 베이스 성능이 약화된다.

2. 메모리 I/O vs. 디스크I/O

가. I/O 효율화 튜닝의 중요성

* 디스크를 경유한 데이터 입출력은 디스크의 액세스 암(Arm)이 움직이면서 헤드를 통해 데이터를 읽고 쓰기 때문에 느린 반면, 메모리를 통한 입출력은 전기적 신호에 불과하기 때문에 디스크를 통한 I/O에 비해 비교할 수 없을 정도로 빠르다. 모든 DBMS는 읽고자 하는 블록을 먼저 버퍼 캐시에서 찾아보고, 없을 경우에만 디스크에서 읽어 버퍼 캐시에 적재한 후 읽기/쓰기 작업을 수행한다. 물리적인 디스크 I/O가 필요할 때면 서버 프로세스는 시스템에 I/O Call을 하고 잠시 대기 상태에 빠진다. 디스크 I/O 경합이 심할수록 대기 시간도 길어진다. ([그림 Ⅲ-1-12] 참조)



[그림 Ⅲ-1-12] 디스크 I/O 경합

* 모든 데이터를 메모리에 올려 놓고 사용할 수 있다면 좋겠지만 비용과 기술 측면에 한계가 있다. 메모리는 물리적으로 한정된 자원이므로, 결국 디스크 I/O를 최소화하고 버퍼 캐시 효율을 높이는 것이 데이터베이스 I/O 튜닝의 목표가 된다.

먼저, DBMS의 작동 원리 입니다.
평소에 **데이터는 하드 디스크의 데이터 파일에 저장**해 두었다가
필요한 시점에 메모리로 복사 해 옵니다.(이때의 메모리 = 데이터베이스 버퍼 캐시)

여기서 중요한 것은 100개의 칼럼을 가진 테이블에
Select를 통해 1개의 컬럼을 수행할 경우에도 100개의 컬럼을 모두 접근하게 됨으로 조심해야하며,
정규화가 중요합니다.

↑ 정규화의 중요성2.

핵심!!!

테이블에 저장된 데이터를 읽는 방식은 두 가지이다.

① Table Full Scan은 해당 테이블에 전체 블록을 읽어서 사용자가 원하는 데이터를 찾는 방식이다.
② 그리고 Index Range Scan은 인덱스를 이용하여 데이터를 일정부분읽어서 ROWID로 테이블 레코드를 찾아가는 방식이다. ROWID는 테이블 레코드가 디스크 상에 어디 저장됐는지를 가리키는 위치 정보이다.

이제 여기서 자주 들었던 Join의 종류가 나오기 시작합니다.

1) Neste Loop Join(가장 기본적인 Join 기법)
2)Sort_Merge join
3) Hash Join(CBE에서만 가능)

1. Nested Loop Join

선행 테이블을 Scan한 결과의 조건 컬럼 내 데이터를 후행테이블 Scan 조건으로 활용한다.

먼저 예시 쿼리를 적어보겠습니다.

```
select
      e.ename, d.dname
from
      emp e, dept d
where
      e.deptno = d.deptno
```

위의 SQL을 실행하면 순서는 다음과 같습니다.

1) 사원·부서 테이블을 메모리에 복사
2) 사원 테이블에서 사원이름 꺼내서 '임시 작업공간'으로 가져 가져감(인덱스 상황이나 다른 요소에 따라 순서가 변경 될 수도 있음)
3) 부서 테이블에서 해당 부서명을 찾으러 가는데 그때 위 SQL의 where조건을 보고 해당 조건에 맞는 데이터를 찾아서 부서명을 가져옴
4) 한 행의 작업이 끝나면 다시 처음 테이블로 돌아가서 두번째 행의 이름을 PGA(메모리 영역으로 생각)로 가져옴
5) 다시 부서 테이블에 가서 두번째 부서번호와 동일한 부서번호를 가진 부서명을 꺼내옴

PGA

Inner loop.

Outer loop.

이와같은 과정이 먼저 읽었던 사원 테이블의 데이터가 끝이 날때까지 작업이 반복됩니다.(LOOP)
그래서 이 Join을 Nested Loop Join이라고 합니다.

먼저 읽은 테이블의 행의 수만큼 Join이 수행됩니다.(중요)
먼저 읽는 테이블이 Join의 성능을 결정 합니다.
그래서 Driving Table(선행 테이블)이라 하고 나중에 읽는 테이블은 Driven Table(후행 테이블) 이라 합니다.

(=Outer Table) (=Inner Table)

- 단점 : `RANDOM ACCESS`가 발생한다.
  RANDOM ACCESS가 많이 발생하면, 성능 지연이 발생한다.

2. Sort-Merge join

```
select
      e.ename, d.dname
from
      emp e, dept d
where
      e.deptno = d.deptno
```

다시한번 위의 쿼리를 확인해 보겠습니다.

emp 테이블의 ename의 값을 가져오고 이를 통해 dept 테이블의 dname의 값을 찾게 됩니다.
이때 where 절에 있는 조건을 보고 그 조건에 맞는 dname을 가져 오게 됩니다.
즉, where절에 잘못된 조건을 줄 경우나 조건을 안 줄 경우에 올바른 데이터를 가져오지 못합니다.

만약 dept 테이블의 데이터가 1억건 이라면
emp의 ename 값을 찾기위해 1억건을 읽어야 합니다.
(emp 테이블에 데이터가 10개가 있다면 10억건의 데이터를 읽어야 합니다.)

그래서 join과 더불어 필수적으로 언급되는 것이 인덱스 입니다.
인덱스가 존재한다면 해당 테이블을 전부 읽지 않고도 데이터를 찾은 후 테이블로 가서 바로 데이터를 읽을 수 있습니다.

이때 인덱스가 없을 경우에도 빨리 해당 데이터를 찾아서 결과를 출력해야 하는 경우에 Sort-Merge join 방법을 사용합니다.
Sort-Merge 방법은
where 조건을 기준으로(deptno) 정렬합니다.
그리고 서로 같은 값(deptno)를 비교하여 값을 가져 옵니다.
말그대로 Sort 한 후 그 결과를 Merge 해서 데이터를 찾게 됩니다.
단점은 Sort 할 때 시간이 너무 오래 걸린다는 점 입니다. 그래서 이를 보완하기 위하여 Hash Join을 사용합니다.

- 'SORT_AREA'라는 메모리 공간에 조인할 두 테이블을 로딩하고 SORT를 수행한다.
- 두 개 테이블에 대해 SORT가 완료되면, 두 테이블을 병합한다.

Hash Join

Hash Join은 양쪽 테이블 모두 Join 컬럼에 인덱스가 없을 경우에 사용 합니다.
(Sort-Merge Join이 시간이 너무 오래 걸리기 때문)

방법

1) 두 테이블 중에서 범위가 좁은 테이블을 메모리로 가져옵니다.
2) Join 조건 칼럼의 데이터를 Hash 함수에 넣어서 나온 Hash Value 값으로 Hash Table을 생성합니다.
3) 후행 테이블의 Join 조건을 Hash 함수에 넣어서 Hash Value를 생성하고 이 값을 선행 테이블의 Hash Table의 값과 비교하여 같은 값을 찾아 매칭합니다.

Sort-Merge Join과 Hash Join은 둘 다 모든 테이블을 다 읽는다는 부분은 동일하지만 Sort-Merge Join에서 실행하는 정렬은 실행하지 않습니다.
이 차이는 약 2배 이상의 성능차이가 나게 됩니다.