1장

1강 DBMS 아키텍처 개요

- 쿼리 평가 엔진
    - 쿼리 평가 엔진은 사용자로부터 입력받은 SQL 구문을 분석하고, 어떤 순서로 기억장치의 데이터 접근할지를 결정한다. 이때 결정되는 계획을 '실행 계획'이라고 한다. 이러한 실행 계획에 기반을 둬서 데이터에 접근하는 방법을 '접근 메서드'라고 부른다.
- 버퍼 매니저
    - DBMS는 버퍼라는 특별한 용도로 사용하는 메모리 영역을 확보해둔다. 이 메모리를 관리하는 것이 버퍼 매니저다. 디스크를 관리하는 디스크 용량 매니저와 함께 연동되어 작동한다.
- 디스크 용량 매니저
    - 데이터베이스는 가장 많은 데이터를 다루는 소프트웨어다. 또한 웹 서버 또는 애플리케이션 서버는 실행되는 동안에만 저장하면 되지만, 데이터베이스는 데이터를 영구적으로 저장해야한다.
    - 디스크 용량 매니저는 어디에 어떻게 데이터를 저장할지 관리하며, 데이터의 읽고 쓰기를 제어한다.

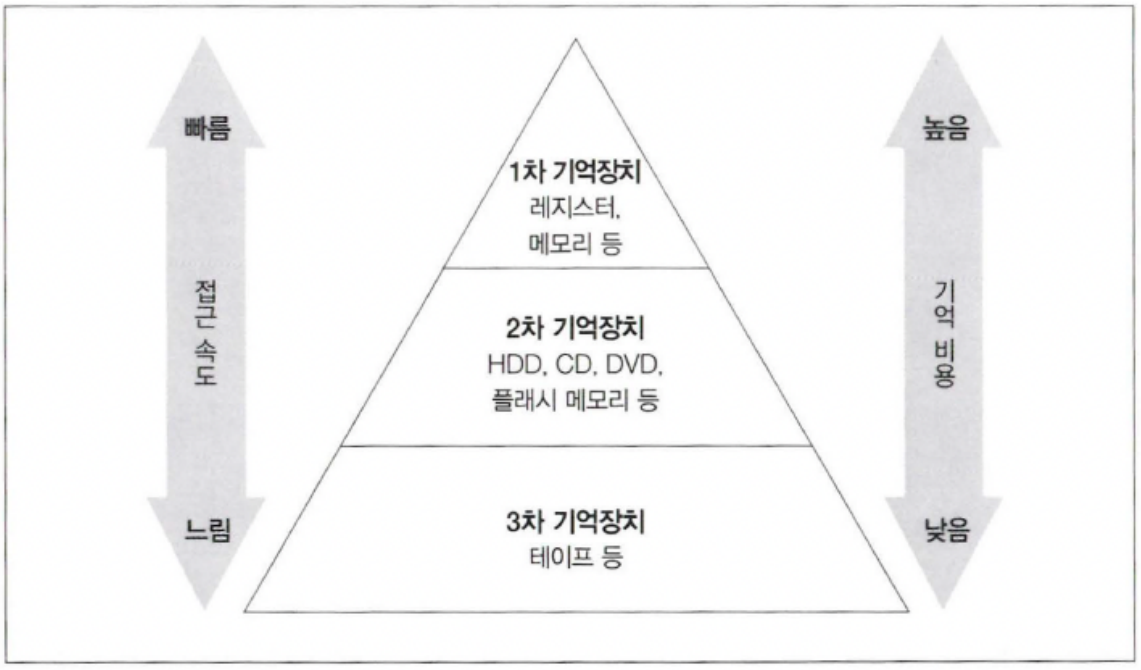
- 트랜잭션 매니저 / 락 매니저
    - 여러명의 유저가 동시에 데이터베이스에 접근하게 되면 DBMS 내부에서 트랜잭션이라는 단위로 관리된다. 이러한 트랜잭션의 정합성을 유지하면서 실행시키고, 필요한 경우 데이터에 락을 걸어 다른 사람과의 요청을 대기시키는것이 트랜잭션 매니저와 락 매니저의 역할이다.
- **리커버리 매니저**
    - 시스템에 장애를 대비하여 데이터를 정기적으로 백업하고, 문제가 일어났을 때 복구하는 기능을 수행하는것이 리커버리 매니저.

2강 DBMS와 버퍼

- 메모리는 한정된 희소 자원이다.
- 반면 데이터베이스가 메모리에 저장하고자 하는 데이터는 너무 많다. 따라서 데이터를 버퍼에 어떠한 식으로 확보할 것인가에 대해 트레이드오프가 발생한다.

기억장치의 계층

그림 1-2 기억장치의 계층



> 기억장치의 계층

일반적으로 기억장치는 기억 비용에 따라 1차부터 3차까지의 계층으로 분류한다. 기억 비용이란 데이터를 저장하는데 소모되는 비용을 나타낸다.

- 1차 기억장치
  - 레지스터, 메모리 등
- 2차 기억장치
  - HDD, CD, DVD, 플래시 메모리 등
- 3차 기억장치
  - 테이프 등

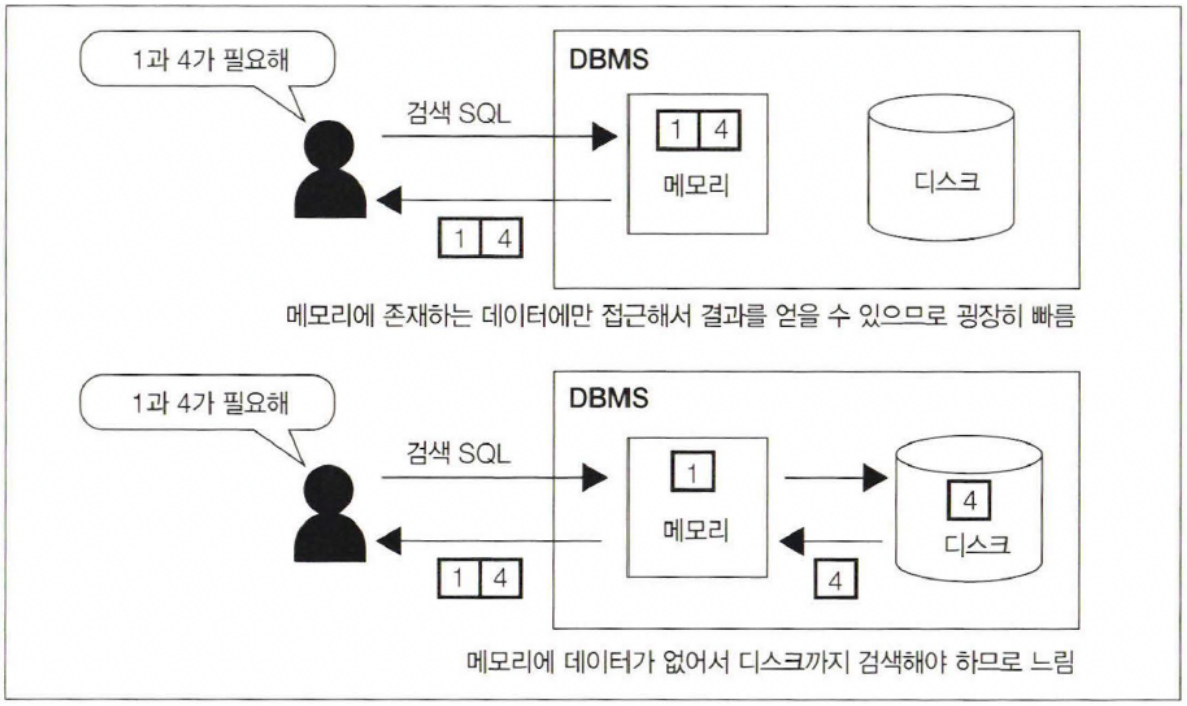
기억 비용이 저렴하다고 해서 좋은 기억장치가 아니다. 피라미드의 아래 면적이 큰 것은 '같은 비용으로 저장할 수 있는 데이터 용량이 많다'를 뜻한다.

많은 데이터를 저장하려면 속도를 잃고, 속도를 얻고자 하면 많은 데이터를 영속적으로 저장하기 힘들다는 트레이드오프가 발생한다.

DBMS와 저장장치의 관계

- 하드디스크(HDD)
  - DBMS가 데이터를 저장하는 매체는 대부분 하드디스크입니다. 용량, 비용, 성능의 관점에서 다른 선택지도 있지만, 대부분의 경우 HDD를 사용합니다.
  - 데이터베이스는 대부분의 시스템에서 범용적으로 사용되는 미들웨어이므로, 모든 상황에서 평균적인 성능을 보장하는 매체를 선택하는 것이 일반적입니다. 이는 DBMS가 데이터를 디스크 외의 장소에 저장하지 않는다는 의미는 아닙니다. 오히려 대부분의 DBMS는 디스크 외의 장소에도 데이터를 저장합니다.
- 메모리
  - 메모리는 디스크에 비해 저장 비용이 많이 듭니다. 따라서 하드웨어 1대에 탑재할 수 있는 양이 크지 않습니다. 일반적인 데이터베이스 서버에서는 메모리 양이 한두 자리 수 정도입니다. 아무리 많아도 100GB를 넘지 않는데, 이는 테라바이트 단위의 용량을 가진 HDD와 비교하면 매우 작은 크기입니다. 따라서 대규모 상용 시스템의 데이터베이스 내부 데이터를 모두 메모리에 올리는 것은 불가능합니다.
- 버퍼를 활용한 성능 향상
  - DBMS가 일부 데이터를 메모리에 저장하는 이유는 성능 향상을 위함입니다. 자주 접근하는 데이터를 메모리에 저장하면, 동일한 SQL 구문을 실행할 때 디스크에서 데이터를 가져올 필요 없이 메모리에서 빠르게 데이터를 검색할 수 있습니다.

버퍼



- 디스크 접근을 줄이면 큰 성능 향상을 기대할 수 있습니다. 이는 일반적인 SQL 구문 실행 시간의 대부분이 저장소 I/O에 소모되기 때문입니다.
- 성능 향상을 위해 데이터를 저장하는 메모리를 **버퍼** 또는 **캐시**라고 합니다. 이는 사용자와 저장소 사이에서 SQL 구문의 디스크 접근을 감소시켜주는 역할을 합니다. 주로 물리적인 매체로 메모리가 사용되며, 이는 하드디스크 상의 데이터에 접근하는 것보다 훨씬 빠릅니다.
- 고속 접근이 가능한 버퍼에 '데이터를 어떻게, 얼마나 오래 올릴지'를 관리하는 것이 DBMS의 버퍼 매니저의 역할입니다.

메모리 위에 있는 두 개의 버퍼

DBMS가 데이터를 유지하기 위해 사용하는 메모리는 크게 2 종류 이다.

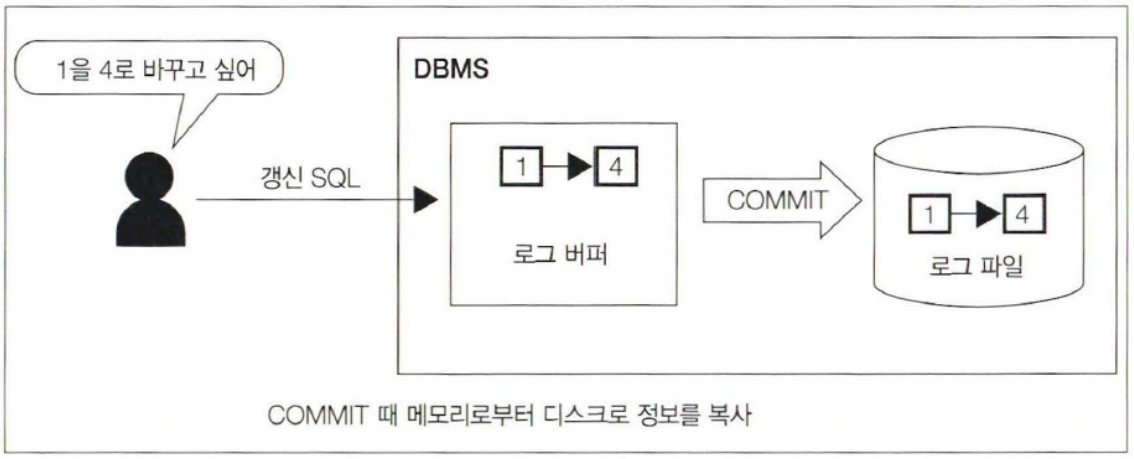
- 데이터 캐시
- 로그 버퍼

		Oracle 11gR2	PostgreSQL 9.3	MySQL 5.7(InnoDB)
데이터 캐시	명칭	데이터베이스 버퍼 캐시	공유 버퍼	버퍼 풀
	매개변수	DB_CACHE_SIZE	shared_buffers	innodb_buffer_pool_size
	초깃값	5MB×CPU 개수×그래뉼 크기(SGA_TARGET이 설정 되어 있지 않은 경우 48MB)	128MB	128MB
	설정값 확인 명령어 예	SELECT value FROM v$parameter WHERE name='db_cache_size';	show shared_ buffers;	SHOW VARIABLES LIKE 'innodb_buffer_pool_size';
	비고	SGA 내부에 확보	–	–
로그 버퍼	명칭	REDO 로그 버퍼	트랜잭션 로그 버퍼	로그 버퍼
	매개변수	LOG_BUFFER	wal_buffers	innodb_log_buffer_size
	초깃값	512KB 또는 128KB× CPU_COUNT 중에 큰 것	64KB	8MB
로그	설정값 확인 명령어 예	SELECT value FROM v$parameter WHERE name='log_buffer';	show wal_buffers;	SHOW VARIABLES LIKE 'innodb_log_buffer_size';
	비고	REDO 로그 버퍼	–	InnoDB 엔진 사용 시에만 적용

대부분의 DBMS는 두 개의 역할을 하는 메모리 영역을 가지고 있다.

> MySQL의 경우 데이터 캐시는 버퍼 풀로 명칭한다, 로그 버퍼는 그대로

- 데이터 캐시
  - 데이터 캐시는 디스크에 있는 데이터의 일부를 메모리에 유지하기 위한 영역입니다.
  - SELECT 구문의 결과 값이 전부 데이터 캐시에 있으면, 저속 저장소인 디스크에 접근할 필요가 없어 응답 속도가 빠릅니다.
  - 하지만, 데이터가 캐시에 없다면 디스크까지 접근해야 해서 응답 속도가 느려집니다.
- 로그 버퍼



- 로그 버퍼는 갱신 처리(INSERT, DELETE, UPDATE, MERGE)와 관련됩니다.
- DBMS는 갱신 관련 SQL 구문을 사용자로부터 받으면, 즉시 저장소에 있는 데이터를 변경하지 않습니다. 먼저 로그 버퍼에 변경 정보를 보내고, 나중에 디스크에서 변경을 수행합니다.
- 이런 방식으로, 데이터베이스의 갱신 처리는 SQL 구문의 실행 시점과 저장소에 갱신하는 시점 사이의 차이를 가지는 비동기 처리입니다.
- SQL 구문을 실행할 때 저장소 파일을 바로 변경하는 것이 간단할 수 있습니다. 하지만 DBMS는 시점의 차이를 두는 이유가 있습니다. 그 이유는 성능 향상입니다.
- 저장소는 검색뿐만 아니라 갱신할 때도 상당한 시간이 소요됩니다. 그러므로, 저장소 변경이 끝날 때까지 기다리면 사용자는 오랫동안 대기하게 됩니다. 그래서 DBMS는 메모리에 갱신 정보를 받은 시점에서 사용자에게 해당 SQL 구문이 '끝났다'고 통지하고, 내부적으로 관련 처리를 계속 수행합니다.

메모리의 성질이 초래하는 트레이드오프

- 휘발성
  - 메모리에는 데이터의 영속성이 없습니다. 하드웨어의 전원을 끄면 메모리에 저장된 모든 데이터가 사라집니다.
  - DBMS를 종료하고 다시 시작하면 버퍼에 있는 모든 데이터가 사라집니다. 따라서, DBMS에 장애가 발생하여 프로세스가 다운되면 메모리에 있는 모든 데이터가 소실됩니다.
  - 이런 이유로, 영속성이 보장되지 않으면 디스크를 완전히 대체하는 것은 불가능합니다.
- 휘발성의 문제점

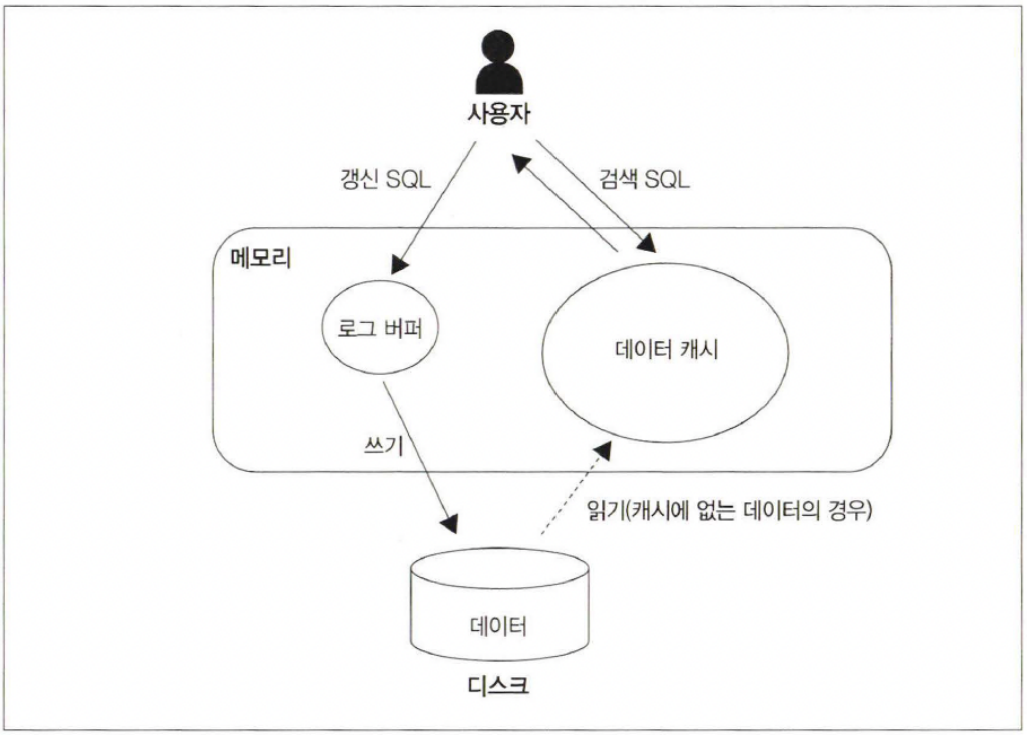
- 가장 큰 문제는 장애로 인해 메모리에 있던 데이터가 모두 사라져 데이터 부정합을 발생시키는 것입니다. **데이터 캐시** 의 경우, 장애로 인해 메모리 위의 데이터가 사라져도 디스크에 원본 데이터가 있으므로 문제가 되지 않습니다. 결과적으로 시간이 더 걸리지만 결과에는 영향을 미치지 않습니다.
- 그러나 로그 버퍼에 있는 데이터가 로그 파일에 반영되기 전에 장애가 발생하면 해당 데이터가 완전히 사라져 복구가 불가능해집니다.
  - 예를 들어, 은행 입출금 또는 카드 인출이 데이터베이스에 반영되지 않을 수 있습니다.
- 이런 문제를 방지하기 위해 DBMS는 **커밋** 시점에 반드시 갱신 정보를 로그 파일(영속적인 저장소에 존재)에 기록함으로써, 장애가 발생해도 정합성을 유지합니다.
- **커밋** 이란 갱신 처리를 '확정'하는 것으로, DBMS는 커밋된 데이터를 영속화합니다.
- 반대로 말하면 커밋 시점에는 반드시 디스크에 **동기** 접근이 발생합니다.
- 이 때 다시 트레이드오프가 발생합니다. 디스크에 동기 처리를 하면 데이터 정합성은 향상되지만 성능은 저하됩니다.

이름	데이터 정합성	성능
동기 처리	○	×
비동기 처리	×	○

시스템 특성에 의한 트레이드 오프

- **데이터 캐시와 로그 버퍼의 크기**

그림 1-5 데이터베이스는 검색을 중시한 메모리 배분이 기본



- DBMS의 버퍼는 보통 다음과 같이 할당됨
  - (데이터 캐시 초깃값) > (로그 버퍼의 초깃값)
  - 이유
    - 데이터베이스가 기본적으로 검색 기능을 메인으로 처리한다고 가정하기 때문
  - 검색 처리 시 레코드가 **수백만~수천만** 건에 달하는 경우 많음
  - 갱신 처리 시 갱신 대상 레코드가 많아봐야 트랜잭션마다 **한 건~ 수만** 건에 달함
- 결론적으로, 자주 검색하는 데이터를 캐시에 올려놓는 것이 효율적
- +) 만약 시스템이 검색에 비해 갱신이 많다면 튜닝하여 사용 가능
- 데이터 베이스가 물리 메모리에 여유가 있을 경우 → 데이터 캐시를 되도록 많이 할당 추천
  - MySQL 메뉴얼에 따르면, 서버가 데이터베이스 전용이라면 물리 메모리의 80%를 버퍼 풀로 할당해도 된다고 함. 이는 다른 애플리케이션이 같은 서버에서 작동할 경우, 그 메모리 사용량을 고려해야 하기 때문.

- **검색과 갱신 중 중요한 것**

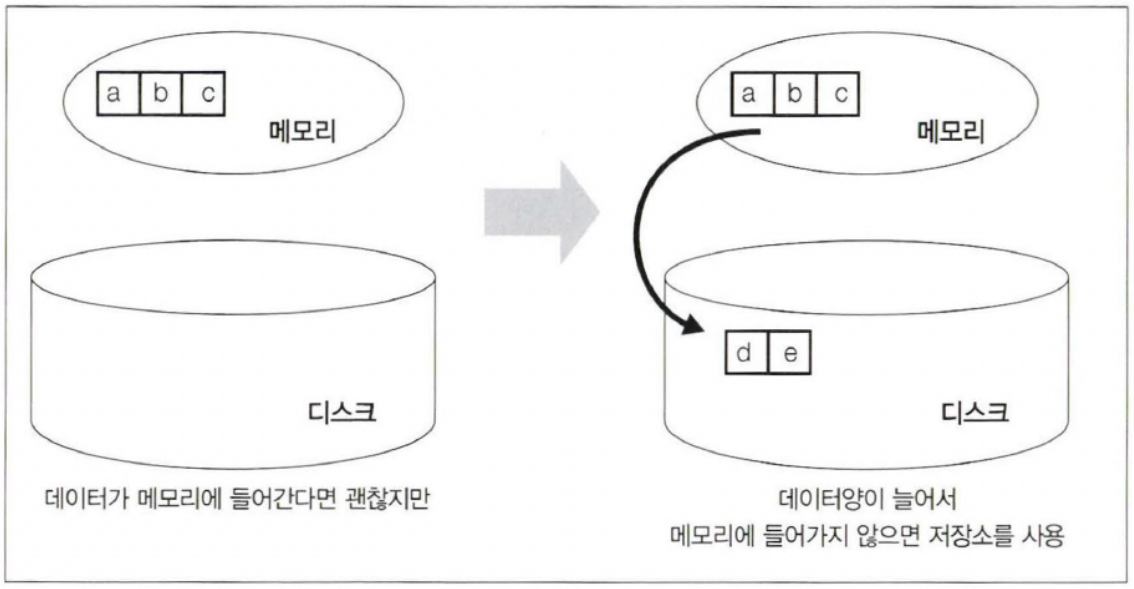
    - 검색, 갱신 중 어떤 것을 우선하여 메모리 관리를 할 것인지 판단해야 함
    - 설계된 데이터베이스의
        - 로그 버퍼 비중이 크다 ⇒ 갱신 처리와 관련해 큰 부하가 걸릴 것을 고려한 설계
        - 데이터 캐시 비중이 크다 ⇒ 검색 처리와 관련된 처리가 중심

추가적인 메모리 영역 '워킹 메모리'

- 언제 사용될까?
    - 정렬 또는 해시 관련 처리에 사용되는 작업용 영역으로 **워킹 메모리** 라고 부른다.
    - 정렬은 ORDER BY 구, 집합 연산 등의 기능을 사용할때 실행된다.
    - 반면 해시는 주로 테이블 등의 결합에서 해시결합이 사용되는 때 실행된다.

> 오라클의 경우 PGA라 명칭하고, MySQL은 `정렬 버퍼` 라고 한다.

- 작업용 메모리 영역은 SQL에서 정렬 또는 해시가 필요한 때 사용되고, 종료되면 해제되는 임시 영역이며, 일반적으로 데이터 캐시와 로그 버퍼와는 다른 영역으로 관리되는 경우가 많다.
- 워킹메모리가 성능적으로 중요한 이유는, 만약 워킹 메모리의 크기가 다루려는 데이터의 크기보다 작다면 DBMS가 저장소를 사용하기 때문이다.
- DBMS는 워킹메모리가 부족할때 사용하는 임시적인 영역을 가지고 있다. 이러한 일시 영역들은 저장소 위에 있으므로 당연히 접근 속도가 느리다. → 디스크 io가 발생하기 때문

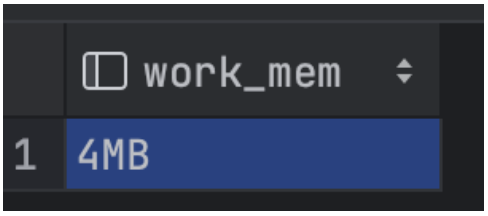


> 오라클의 경우 임시 테이블 스페이스, MsSQL의 경우 TEMPDB

표 1-3 각 DBMS에서 워킹 메모리를 부르는 명칭과 설정

DBMS	명칭	매개변수	기본값
Oracle 11g R2	PGA(Program Global Area)	PGA_AGGREGATE_TARGET	10MB 또는 SGA 크기의 20% 중에 큰 것
PostgreSQL 9.3	워크 버퍼	work_mem	8MB
MySQL 5.7	정렬 버퍼	sort_buffer_size	256KB

- `SHOW work_mem;`



3강 DBMS와 실행 계획

- **권한 이양의 죄악**
  - 자바, C, 루비와 같은 절차가 기초되는 언어는 사용자가 데이터에 접근하기 위한 절차(HOW)를 책임지고 기술하는 것이 전제다.

    - 반면에 비절차적인 RDB는 모든 일을 시스템에게 맡겼다.
    - 그 이유는 '비지니스 전체의 생산성 향상' 때문이다.

- 데이터에 접근 하는 방법은 어떻게 결정할까?
    - RDB에서 데이터 접근 절차를 결정하는 모듈은 쿼리 평가 엔진이라고 부른다.
    - 쿼리 평가 엔진은 입력받은 SQL 구문을 처음 읽어들이는 모듈이기도 하다. 쿼리 평가 모듈은 추가로 파서 또는 옵티마이저와 같은 여러 개의 서브 모듈로 구성된다.

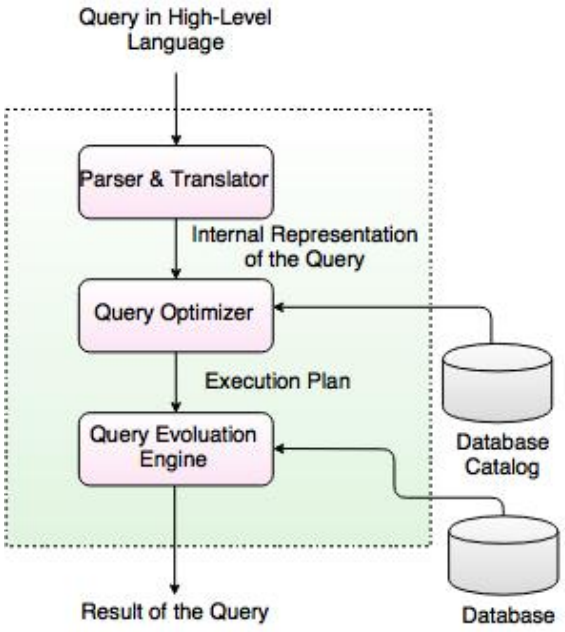


Fig. Query Processing

- 파서
    - 파서의 역할은 입력받은 SQL 구문이 항상 구문적으로 올바른지 검사를 하는 역할이다. 그리고 SQL 구문을 정형적인 형식으로 변환한다. 그래야 DBMS 내부에서 일어나는 후속 처리가 효율화된다.
- 옵티마이저
    - 파서를 통과한 쿼리는 옵티마이저로 전송된다. 이때 최적화(옵티마이저)의 대상은 데이터접근법(실행계획)이다. 옵티마이저가 DBMS 두뇌의 핵심.
    - 옵티마이저는 인덱스 유무, 데이터 분산 또는 편향 정도와 같은 조건을 고려한다. 선택 가능한 많은 실행 계획을 작성하고, 비용을 연산하고, 가장 낮은 비용을 가진 실행 계획을 선택한다.
- 카탈로그 매니저

    - 옵티마이저가 실행 계획을 세울 때 옵티마이저에 중요한 정보를 제공하는 것이 카탈로그 매니저.
        - 카탈로그란?
            - 일종의 통계, 메타 데이터
    - 카탈로그란 DBMS 내부 정보를 모아놓은 테이블로, 테이블 또는 인덱스의 통계 정보가 저장되어 있다.
    - 따라서 카탈로그 정보를 간단하게 '통계 정보'라고 칭하기도 한다.
- 플랜평가
    - 옵티마이저가 SQL 구문에서 여러 개의 실행 계획을 세운 뒤 최적의 실행 결과를 선택하는 것이 플랜 평가다. 실행 계획이라는 것은 DBMS가 바로 실행할 수 있는 코드가 아니다. 오히려 사람이 읽기 쉽게 만들어진 문자 그대로의 '계획서'다. 읽고 성능이 좋지 않다면 수정을 고려할 수 있다.
    - 이렇게 하나의 실행 계획을 선택하면, 이후 DBMS는 실행 계획을 절차적인 코드로 변환하고 데이터 접근을 수행한다.

옵티마이저와 통계 정보

- 옵티마이저는 명령대로 잘처리해주는 만능이 아니다. 특히 카탈로그 매니저가 관리하는 통계 정보에 대해서는 데이터베이스 엔지니어가 항상 신경 써줘야 한다.
- 플랜 선택을 옵티마이저에게 맡기는 경우, 최적의 플랜이 선택되지 않는 경우가 많다. 그 패턴 중 대표 원인으로는 통계 정보가 부족한 경우이다.
- 카탈로그에 포함되어 있는 통계 정보들
    - 각 테이블의 레코드 수
    - 각 테이블의 필드 수와 필드의 크기
    - 필드의 카디널리티(값의 개수)
    - 필드값의 히스토그램(어떤 값이 얼마나 분포되어 있는가)
    - 필드 내부에 있는 NULL 수
    - 인덱스 정보
- 위의 정보들을 활용해 옵티마이저는 실행 계획을 만든다. 위의 카탈로그 정보가 테이블 또는 인덱스의 실제와 일치 하지 않을 경우 문제가 생긴다.

- 즉, 테이블에 데이터 삽입/갱신/제거가 수행 될 때 카탈로그 정보가 갱신되지 않는다면 옵티마이저는 오래된 정보를 바탕으로 실행 계획을 세우게 된다. 그 결과 잘못된 계획을 세울 수 밖에 없게 된다.

최적의 실행 계획이 작성되게 하려면

- 올바른 통계 정보가 모이는 것은 SQL 성능에 굉장히 중요한 요소이다.
    - 따라서 테이블의 데이터가 바뀌면 카탈로그의 통계 정보도 함께 갱신해야만 한다. 수동으로 갱신해야하는 DBMS와 자동으로 갱신되는 DBMS가 존재 한다.
- 통계 정보 갱신은 대상 테이블 또는 인덱스의 크기와 수에 따라 몇십분에서 몇시간이 소요 되는 실행 비용이 큰 작업이다. 하지만 최적의 플랜을 선택하려면 꼭 필요하므로 갱신 시점을 확실하게 검토해야 한다.
- PostgreSQL에서 통계 정보를 갱신하기 위해서는 `ANALYZE` 명령어를 사용

1. **특정 테이블의 통계만 갱신하기**

```
ANALYZE table_name;
```

    - 여기서 `table_name`은 통계를 갱신하려는 테이블의 이름

2. **데이터베이스의 모든 테이블에 대한 통계 갱신하기**

```
ANALYZE;
```

3. `VACUUM` **명령어와 함께 사용하기**

    - `VACUUM`은 불필요한 데이터를 제거하며, `ANALYZE` 옵션과 함께 사용하면 통계 정보도 갱신됨

```
VACUUM ANALYZE table_name;
```

    또한, 전체 데이터베이스에 대해서도 동일한 작업을 수행 가능

```
VACUUM ANALYZE;
```

4강 실행 계획이 SQL 구문의 성능을 결정

- 실행 계획이 만들어지면 DBMS는 그것을 바탕으로 데이터 접근을 수행한다. 통계 정보가 부족하거나, 이미 최적의 방법이 설정되어 있는데도 느린 경우가 있다. 또한 통계 정보가 최신이라도 SQL 구문이 너무 복잡하면 옵티마이저가 최적의 접근 방법을 선택하지 못할 수 있다.

실행 계획 확인 방법 -

- SQL구문의 지연이 발생 했을때 제일 먼저 실행 계획을 살펴봐야 한다.

```
# MySQL의 경우
EXPLAIN EXTENDED SQL 구문

# Oracle의 경우
set autotrace traceonly

# postgresql
EXPLAIN SELECT * FROM table_name WHERE column_name = 'value';
-- 실행 계획과 함께 실제 런타임 통계 확인
EXPLAIN ANALYZE SELECT * FROM table_name WHERE column_name = 'value';
-- 출력 형식 지정
EXPLAIN (FORMAT JSON) SELECT * FROM table_name WHERE column_name = 'value';
```

  기본적으로 3개의 기본적인 SQL 구문의 실행 계획이 있다.

  1. 테이블 풀 스캔의 실행 계획
  2. 인덱스 스캔의 실행 계획
  3. 간단한 테이블 결합의 실행 계획

  샘플 테이블로 기본 키는 점포 ID, 그리고 평가와 주소 지역 데이터를 가지고 있다.
  테이블에는 60개의 레코드를 넣고 통계 정보까지 구현해놨다고 가정.

  - 테이블 풀스캔의 실행 계획

```
SELECT * FROM shops;

# postgreSQL 의 경우
Seq Scan on shops (const=0.00..1.60 rows=60 width=22)
```

- DBMS마다 출력 포맷이 같지는 않지만 공통적으로 나타나는 부분이 있다.

  1. 조작 대상 객체

2. 객체에 대한 조작의 종류
3. 조작 대상이 되는 레코드 수

- 이 3가지 내용은 거의 모든 DBMS의 실행계획에 포함되어 있다.
  - **조작 대상 객체**
    - postgreSQL의 경우 on 이라는 글자뒤에 shops 테이블이 출력된다.
    - 이 부분은 테이블 이외에도 인덱스, 파티션, 시퀀스처럼 SQL 구문으로 조작 가능한 객체라면 무엇이든 올 수 있다.
  - **객체에 대한 조작의 종류**
    - 실행 계획에서 가장 중요한 부분이다.
    - postgreSQL은 문장의 앞부분에 나온다. ‘Seq Scan’은 순차적인 접근(Sequential Scan)의 줄임말로 해당 테이블의 데이터를 전체를 읽어낸다는 뜻이다.
  - **조작 대상이 되는 레코드 수**
    - Rows라는 항목에 출력 된다. 결합 또는 집약이 포함되면 1개의 SQL 구문을 실행해도 여러 개의 조작이 수행된다. 그러면 각 조작에서 얼마만큼의 레코드가 처리되는지가 SQL 구문 전체의 실행 비용을 파악하는데 중요한 지표가 된다.
    - 이 값은 옵티마이저가 실행 계획을 만들 때 나왔던, 카탈로그 매니저로부터 얻은 값이다. 따라서 통계 정보에서 파악한 숫자이므로, 실제 SQL 구문을 실행한 시점의 테이블 레코드 수와 차이가 있을 수 있다.

인덱스 스캔의 실행 계획 -

- 이전에 실행했던 SQL 구문에 조건을 추가한다.

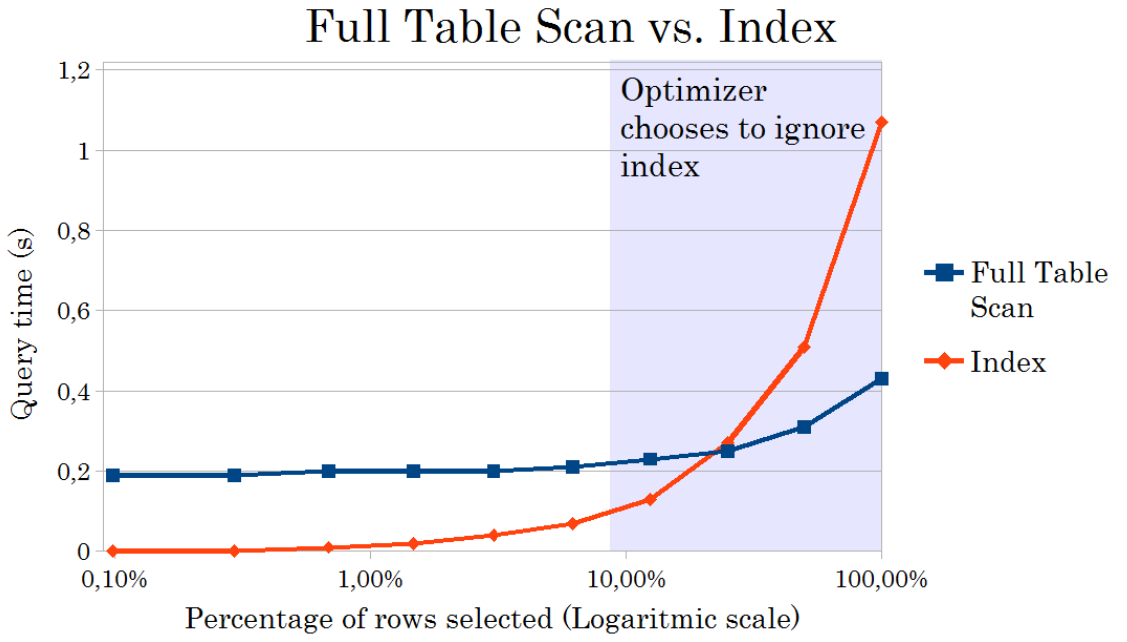
```
SELECT * FROM shows WHERE shop_id = '00050';

`# 결과
Index Scan using pk_shops on shops (cost=0.00..8.27 rows=1 width=320)
Filter (shop_id = '00050'::bpchar)`
```

- 조작 대상이 되는 레코드 수

    - Rows가 1로 변경 되었다. WHERE 구에서 기본 키가 '00050'인 점포를 지정했으므로, 접근 대상은 반드시 레코드 하나이기 때문

- 접근 대상 객체와 조작
    - 이전의 'Seq Scan'에서 'Index Scan'으로 변경 되었다. 이는 인덱스를 사용해 스캔을 수행한다는 뜻이다.
    - 일반적으로 스캔하는 모집합 레코드 수에서 선택되는 레코드 수가 적다면 테이블 풀 스캔보다 빠르게 접근을 수행한다. 이는 풀 스캔이 모집합의 데이터양에 비례하여 처리 비용이 늘어나는것에 반해, 인덱스를 사용할 때 활용되는 B-Tree가 모집합의 데이터양에 따라 대수 함수적으로 처리 비용이 늘어나기 때문이다.
- 쉽게 말해 인덱스의 처리 비용이 완만하게 증가한다는 뜻으로 특정 데이터 양을 손익 분기점으로 인덱스 스캔이 풀 스캔보다 효율적인 접근을 하게 된다는 뜻이다.



간단한 테이블 결합의 실행 계획 -

- SQL에서 지연이 일어나는 경우는 대부분 결합과 관련되어 있다. 결합을 사용하면 실행 계획이 상당히 복잡해진다. 옵티마이저도 최적의 실행 계획을 세우기 어렵다. 따라서 결합 시점의 실행 계획 특성이 굉장히 중요한 의미가 있다.
- 예약에 관한 샘플 데이터를 저장하는 테이블을 추가로 가정한다.
- `SELECT shop_name FROM Shops s INNER JOIN Reservations R ON S.shop_id = R.shop_id;`

일반적으로 DBMS는 결합할 때 세가지의 알고리즘을 사용한다.

1. Nested Loops
    - 가장 간단한 결합으로서 한쪽 테이블을 읽으면서 레코드 하나마다 결합 조건에 맞는 레코드를 다른 쪽에서 찾는 방식. 이중 반복으로 구현되므로 중첩 반복이다.
2. Sort Merge
    - 결합 키(외래 키)로 레코드를 정렬하고, 순차적으로 두 개의 테이블을 결합하는 방법.
    - 결합전에 전처리를 해야하는데 작업용 메모리로 워킹메모리를 사용한다.
3. Hash
    - 이름 그대로 결합 키값을 해시값으로 맵핑하는 방법.
    - 해시 테이블을 만들어야 하므로 작업용 메모리 영역을 필요로 한다.

```
# 결과
Nested Loop (cost=0.14..14.80 rows=10 width=2)
 -> Seq Scan on reservations r (cost=0.00..1.10 rows=10 width6)
 -> Index SCan using pk_shops on shops s (cost=0.14..1.36 rows=1 width=8)
  Index Cond:(shop_id = r.shop_id)
```

- 객체에 대한 조작의 종류

  ‘Nested Loop’라고 나오므로 어떤 알고리즘을 사용하는지 알 수 있다. 일반적으로 실행 계획은 트리 구조이다. 이때 중첩단계가 깊을수록 먼저 실행된다. ‘Nested Loop’보다도 ‘Seq Scan’과 ‘Index Scan’의 단계가 깊으므로, 결합 전에 테이블 접근이 먼저 수행된다. 이때 결합의 경우 어떤 테이블에 먼저 접근하는지가 굉장히 중요하다. 같은 주업 단계에서는 위에서 아래로 실행 된다.

  예를 들어 Reservation 테이블과 Shop 테이블 접근이 같은 중첩 단계에 있지만, Reservation 테이블에 대한 접근이 위에 있으므로, Reservation 테이블에 대한 접근이 먼저 일어난다.