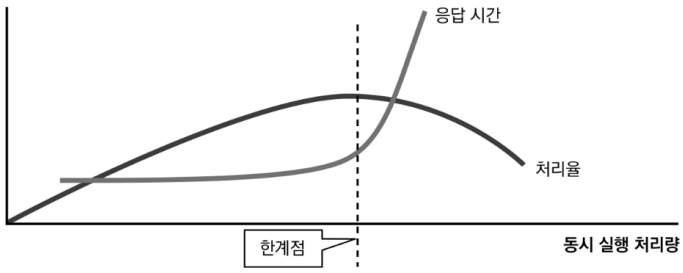
10장 성능을 생각하자

데이터베이스의 성능

성능을 측정하는 2가지 지표

1. 처리시간, 응답시간: 특정 처리의 시작부터 종료까지 걸린 시간
2. 처리율: 시간당 처리하는 처리양, 처리율이 커질수록 많은 자원이 필요하다. (동시에 실행하는 사용자 수가 많아질수록 필요한 물리 자원이 증가한다)

그림 10-2 처리율과 응답 시간의 한계점



시스템은 동시에 실행되는 처리가 가장 많은 순간을 상정해서 자원을 미리 준비해야 극단적인 지연을 피할 수 있다. 이는 동시에 실행되는 처리가 많아질 때 갑자기 응답시간과 처리율이 떨어지는 버틀넥이 존재하기 때문이다.

정점을 미리 예측해서 자원을 확보하는 것을 사이징, 캐퍼시티 플랜이라고 한다.

ex) 3월, 9월의 결산 월에 액세스가 집중된다. → 이 때 자원을 얼마나 확보해두고 있어야 할 것인가?

액세스 집중 유형

주기형: 주기적으로 액세스가 집중되는 경우 (주기적으로 일이 많아지는 경향이 있는 업무 시스템)

돌발형: 언제 액세스가 집중되는지 모름. (온라인 게임)

→ 자원을 가변형으로 유연하게 변동할 수 있는 클라우드 방식이 도움이 될 수 있다.

데이터베이스가 병목이 되기 쉬운 포인트인 이유

1. 취급하는 데이터양이 많다.
2. 자원 증가를 통한 해결이 어렵다.

데이터가 저장되는 곳은 저장소인 하드디스크인데, 스케일 아웃이 어려운 부품이다.

Active Standby, Active Active 구성은 저장소를 포함해 스케일 아웃이 어렵다.

Shared Nothing 만 저장소를 포함해 스케일 아웃이 가능하다 (→ 무슨 뜻이지??)

→ 하드디스크 대신에 메모리를 저장소로 사용하는 방식인 인메모리 데이터베이스를 사용함으로써 성능을 개선할 수도 있다.

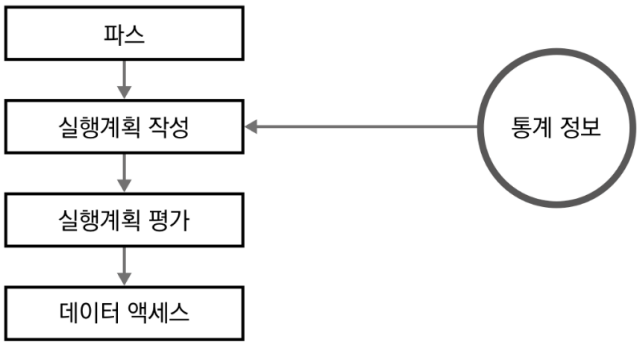
데이터베이스 튜닝

- 데이터베이스가 벙목 포인트가 될 가능성이 크기 때문에 발달하게 된 기술
- 애플리케이션을 효율화해서 같은 양의 자원이라도 데이터베이스의 성능을 향상시키는 기술

DBMS가 SQL 문을 받아서 실행하는 과정

성능을 결정하는 요인을 확인하려면 데이터베이스가 SQL을 어떻게 처리하는지 알아야 한다.

그림 10-4 SQL 문을 받아서 실행하는 과정



파스

- 파서가 SQL 문이 문법적으로 잘못된 부분이 없는지 점검하는 단계

```
mysql> SELECT COUNT(*) FLOM City;

ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for
 the right syntax to use near 'City' at line 1
```

실행계획 작성

- SQL 문에 필요한 데이터를 접근 하는 방법은 여러가지가 있는데 어떤 경로로 접근할지 계획하는 단계
- 옵티마이저가 통계정보를 이용해서 실행계획, 액세스 플랜을 세우는 단계
- SQL은 절차적 언어와 다르게 내부 동작 방식을 사용자가 일일히 지정하지 않는다는 점에서 선언형 언어
- SQL 은 그저 어떤 테이블의 어떤 데이터가 필요하다고 알려줄 뿐 (+ 특정 조건과 일치하는 데이터를 찾아줘! 등) → 이 방법이 구체적으로 드러나지 않는 선언형 언어, 사람이 아니라 데이터베이스가 효율적인 방법을 더 잘 찾을 수 있다고 생각하기 때문에 이런 방식으로 만들어졌다.
- 옵티마이저 대신 엔드 유저가 지정할 수도 있지만 기본적으로 추천하지 않는다 (hint 라는 명령어를 사용)

통계 정보?

1. 테이블의 대강의 행수, 열수
2. 각 열의 길이와 데이터 형
3. 테이블의 크기
4. 열에 대한 기보키나 NOT NULL 제약의 정보
5. 열 값의 분산과 편향

→ 효율성을 따지기 위해 테이블의 데이터를 샘플링 추출해서 계산한 것으로 완전히 정확하진 않다.

```
mysql> show table status;
+-----------------+--------+------+----------------+---------------------+---------------------+
| Name            | Engine | Rows | Avg_row_length | Create_time         | Update_time         |
+-----------------+--------+------+----------------+---------------------+---------------------+
| city            | MyISAM | 4079 |             67 | 2014-10-30 17:16:12 | 2014-10-30 17:16:15 |
| country         | MyISAM | 239  |            261 | 2014-10-30 17:16:13 | 2014-10-30 17:16:17 |
| countrylanguage | MyISAM | 984  |             39 | 2014-10-30 17:16:13 | 2014-10-30 17:16:19 |
+-----------------+--------+------+----------------+---------------------+---------------------+
3 rows in set (0.00 sec)
```

Rows: 행수, Avg_row_length: 평균 레코드의 크기, 단위: 바이트

```
mysql> show index from city;
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+
| city  | 0          | PRIMARY  | 1            | ID          | A         | 4084        | NULL     |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+
1 row in set (0.00 sec)
```

Cardinality: 인덱스 대상의 분산이나 편향

통계정보의 갱신?

- 기본적으로 자동으로 수집되어 구현됨.
- 타이밍이나 조건은 구현에 따라 다르지만 대체로 대량의 데이터가 변경될 때 수집되거나
- 주기적으로 갱신 시간을 정해서 갱신하거나
- 명시적으로 갱신할 수 있다. `ANALYZE TABLE`

실행계획을 정하는 기준 보는 법

SQL문 앞에 EXPLAIN 키워드를 붙여서 확인해 볼 수 있다.

스캔 방식 (풀 스캔 vs 레인지 스캔)

1. 풀 스캔

```
mysql> EXPLAIN SELECT * FROM City;
+----+-------------+-------+------+---------------+------+---------+------+------+-------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra |
+----+-------------+-------+------+---------------+------+---------+------+------+-------+
| 1  | SIMPLE      | City  | ALL  | NULL          | NULL | NULL    | NULL | 4079 | NULL  |
+----+-------------+-------+------+---------------+------+---------+------+------+-------+
1 row in set (0.00 sec)
```

type: ALL

테이블에 포함된 레코드를 처음부터 끝까지 전부 읽어들이는 방식

앞장부터 끝까지 보는 방식, 4079개 (row 값) 레코드를 전부 수집하여 찾는다

2. 레인지 스캔

```
mysql> EXPLAIN SELECT * FROM City WHERE id BETWEEN 1700 and 1710;
+----+-------------+-------+-------+---------------+---------+---------+------+------+-----------------------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref  | rows | Extra                 |
+----+-------------+-------+-------+---------------+---------+---------+------+------+-----------------------+
| 1  | SIMPLE      | City  | range | PRIMARY       | PRIMARY | 4       | NULL | 10   | Using index condition |
+----+-------------+-------+-------+---------------+---------+---------+------+------+-----------------------+
1 row in set (0.00 sec)
```

type: range

테이블의 일부 레코드에만 액세스 하는 방법

책 끝의 색인을 보고 페이지를 찾는 방식

1700번부터 1710번까지의 레코드 10개만 수집한다.

인덱스

레인지 스캔을 하려면 꼭 인덱스가 필요하다.

possible_keys, key: PRIMARY → 기본키를 인덱스로 사용한다는 뜻

```
mysql> SHOW INDEX FROM City;
+-------+------------+----------+ ··· +---------+------+------------+---------+---------------+
| Table | Non_unique | Key_name | ··· | Packed | Null | Index_type | Comment | Index_comment |
+-------+------------+----------+ ··· +---------+------+------------+---------+---------------+
| city  | 0          | PRIMARY  | ··· | NULL    |      | BTREE      |         |               |
+-------+------------+----------+ ··· +---------+------+------------+---------+---------------+
1 row in set (0.00 sec)
```

인덱스 확인하는 명령어, PRIMARY 라는 이름의 인덱스 (id column) 가 하나 있다

```
mysql> EXPLAIN SELECT * FROM City WHERE district = 'Chollabuk';
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra       |
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
| 1  | SIMPLE      | City  | ALL  | NULL          | NULL | NULL    | NULL | 4079 | Using where |
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+1 row
in set, 1 warning (0.01 sec)
```

id 컬럼이 인덱스인 상황, 인덱스가 아닌 district 열을 기준으로 검색을 하는 경우 레인지 스캔이 되지 않아, 전체 행인 4079행을 스캔하게 된다.

인덱스 만드는 법

```
mysql> CREATE INDEX ind_district ON City(district);
```

```
mysql> EXPLAIN SELECT * FROM City WHERE district = 'Chollabuk';
+----+-------------+------+------+---------------+-------------+---------+-------+------+-----------------------+
| id | select_type | table | type | possible_keys | key         | key_len | ref   | rows | Extra                 |
+----+-------------+------+------+---------------+-------------+---------+-------+------+-----------------------+
| 1  | SIMPLE      | City | ref  | ind_district  | ind_district | 20      | const | 4    | Using index condition |
+----+-------------+------+------+---------------+-------------+---------+-------+------+-----------------------+
1 row in set (0.03 sec)
```

인덱스 생성 후 인덱스를 기준으로 검색하면 레인지 검색을 하게 된다. type 값으로는 기본키 인덱스로 검색 하는 경우는 range, 우리가 생성한 인덱스로 조회하는 경우 ref 값이 된다.

사용자는 단지 인덱스를 만들었을 뿐 동일한 SQL 을 날리지만 내부적으로 옵티마이저가 인덱스를 사용하는 레인지 스캔이 빠르다고 판단하여 레인지 스캔으로 실행계획을 바꿨다.

인덱스를 사용하면

1. SQL 문을 변경하지 않아도 성능이 개선된다
2. 테이블의 데이터에 영향을 주지 않는다

인덱스의 구조

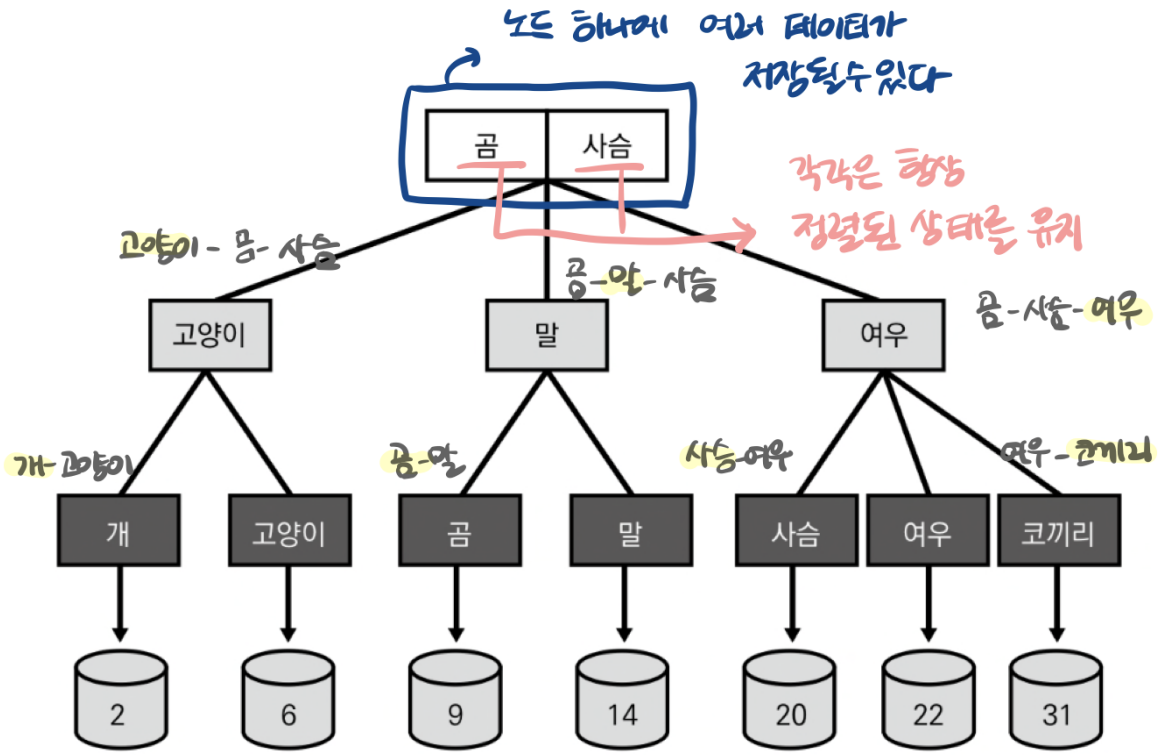
- 논리적인 구조: (단어, 페이지 수) 쌍
- 물리적인 구조: BTREE, ...

B-Tree

B 트리 구조는 검색이 효율적이다. 항상 균형트리를 유지한다.

> 💡 **밸런스 트리(Balanced Tree)란?**
> 트리의 노드가 한 방향으로 쏠리지 않도록, 노드 삽입 및 삭제 시 특정 규칙에 맞게 재 정렬되어 왼쪽과 오른쪽 자식 양쪽 수의 밸런스를 유지하는 트리이다. 항상 양쪽 자식의 밸런스를 유지하므로, 무조건 O(logN)의 시간 복잡도를 가지게 된다.
>
> 다만 재정렬되는 작업으로 인해 노드 삽입 및 삭제 시 일반적인 트리보다 성능이 떨어지게 된다. 그러므로 밸런스 트리는 삽입/삭제의 성능을 희생하고 탐색에 대한 성능을 높였다고 볼 수 있다.

Ex) 고양이, 여우, 사슴, 곰, 코끼리, 말 이라는 데이터가 있을 때 이를 가나다 순서로 B-Tree 를 만들어보면



루트에서 탐색을 시작해서 데이터의 대소를 비교하여 1단계씩 아래로 진행해간다.

그리고 리프 노드는 데이터가 포함된 테이블의 페이지 수를 가리키고 있다.

ex) 말이라는 데이터를 조회 시, 14 번 페이지를 확인한다.

B tree는 균일트리라서 어떤 데이터를 찾더라도 동일한 시간안에 결과를 얻을 수 있다. → O(log n)

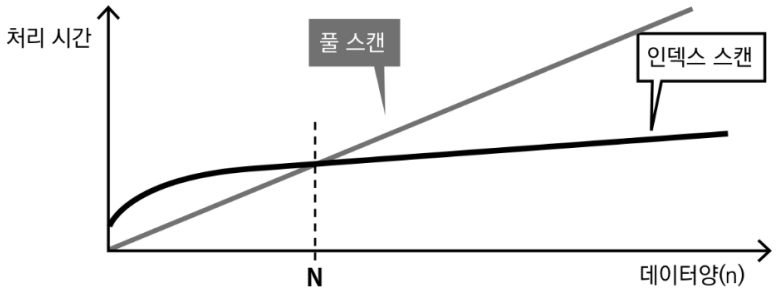
선형검색을 하는 배열의 경우 → 최악의 시간 복잡도 O(n)

다만 데이터가 갱신을 반복하면서 균형이 깨져가게 도고 성능도 악화되어간다.

→ 정기적으로 인덱스 재구성을 하여 트리의 균형을 되찾는 작업이 필요하다.

데이터 양에 비례해 효과가 오른다.

그림 10-9 인덱스 스캔의 진가



데이터 양이 많을 수록 인덱스 스캔의 효과가 점점 두드러진다.

→ 따라서 소규모 데이터의 경우는 크게 효과가 없을 수 있다 (약 100만행 이하의 레코드를 가진 테이블)

```
302p 인덱스를 만들 때 성능적인 이저으로 정렬을 건너뛰는게 가능하다는 예가 있는데 SQL문을 실행할 때 백그라운드로 정렬을 수행하는 경우가 있지만 정렬을 끝낸 인덱스를
 사용하면서 이를 건너뛰는 경우가 있다.

생성했던 ind_district 를 지우고 다시 EXPLAIN Select ... GROUP BY district 명령어 수행시 Using temporary; Using file sort 를 사용하고 있다.
→ 정렬을 위해 임시 영역에 파일을 작성하고 이를 사용한다는 것!
```

내부적으로 정렬을 발생하게 하는 SQL

- GROUP BY
- 집약 함수 (COUNT, SUM, AVG)
- 집합 연산 (UNION, INTERSECT, EXCEPT)

해당 키 열에 인덱스가 존재하면 옵티마이저가 이를 이용해서 정렬을 건너뛰는 효율화 작업을 수행하기에 성능이 빨라진다.

인덱스 사용시 주의점

인덱스 갯수가 많다고 해서 성능이 빨라지는게 아니다.

1. 인덱스 갱신의 오버헤드로 갱신 성능이 떨어진다.

테이블에 데이터가 갱신, 제거되면 인덱스도 갱신해야 한다.

조회 성능 vs 갱신 성능의 트레이드 오프를 주의해야 한다.

2. 의도한 것과 다른 인덱스가 사용된다.

   인덱스의 갯수가 많으면 옵티마이저는 인덱스 중 어떤 인덱스를 사용해서 실행계획을 세울지 헤매게 되고

   최적의 인덱스를 사용하지 못할 수 있다.

인덱스를 만드는 기준

- 크기가 큰 테이블만 만든다
  - 크기가 작은 테이블은 풀 스캔이나 레인지 스캔이나 성능차이가 크지 않다.
- 기본키 제약이나 유일성 제약이 부여된 열에는 불필요하다
  - 기본키나, unique key는 값의 중복 체크를 하기 위해 데이터를 정렬해야 하고, 인덱스를 이용해 정렬하는 것이 편리하기에 기본적으로 인덱스가 붙어있다.
- Cardinality 가 높은 열에 만든다
  - 인덱스 트리를 따라가는 조작이 증가할수록 오버헤드가 증가하기 때문 (????)
  - 예시) 인덱스 효과: `각 사람들의 고유 번호 >>>>>>>>>> 행정구역 > 성별`

옵티마이저가 제 역할을 못하는 경우 (안티 패턴)

1. 결과 정보의 갱신이 OFF 로 설정되어 있는 경우
2. 정기 갱신을 설정하고 데이터 양이 급격히 변화한 경우