ORDER BY Optimization

목표: ORDER BY 를 통한 정렬을 수행할 때 적용할 수 있는 최적화 기법에 대해서 배워보겠습니다.

포인트: 인덱스의 유무에 따른 ORDER BY 의 내부 동작원리와 이를 이용한 최적화 기법에 대해 이해하는게 중요합니다.

▼ Basic Understanding of ORDER

`ORDER BY` 절에 명시된 칼럼에 인덱스가 있으면, MySQL은 이미 정렬된 데이터를 사용하므로 추가적인 정렬 작업이 필요 없습니다.

인덱스가 없는 경우, `filesort` 방식으로 임시 테이블을 사용해 정렬 작업을 수행합니다.

`ORDER BY` 절이 인덱스를 사용하는지, `filesort` 를 이용하고 있는지 여부를 확인하기 위해 `EXPLAIN` 명령어를 사용할 수 있습니다.

- 실행 계획의 Exta 칼럼에 `Using filesort` 가 있다면 `filesort` 를 사용하고 있는 것입니다.
- 실행 계획의 key 칼럼에 인덱스가 출력되었다면 인덱스를 이용하고 있다는 뜻입니다.

▼ Optimization methods

기본적으로 ORDER BY 절을 사용하는 경우에 인덱스를 사용하도록 하는 것이 좋습니다:

- 이는 특히 `LIMIT` 절과 함께 사용될 때 유의미한데, 인덱스가 있으면 MySQL은 필요한 만큼의 데이터만 읽고나서 처리를 중단합니다. 인덱스가 없으면 전체 데이터를 정렬한 후 원하는 행의 수만큼 추출합니다.

`filesort` 를 이용하고 있는 경우의 최적화 방법:

1. `sort_buffer_size` **튜닝:**
    - `filesort` 는 임시 테이블에 기록하고 `Sort Buffer` 를 통해 정렬을 수행합니다.
    - 정렬해야하는 데이터가 많다면 디스크 수준에서 정렬을 할 것이라서 성능이 잘 나오지 않습니다.
    - 그러므로 `sort_buffer_size` 를 증가시켜서 디스크 정렬을 최소화 하도록 최적화 할 수 있습니다.
    - `sort_merge_passes` 변수를 통해서 디스크 정렬을 수행했는지 알 수 있습니다.

2. **Single-Pass 에서 Two-Pass 로 튜닝:**
    - `filesort` 는 Single-Pass 와 Two-Pass 방식으로 정렬을 수행할 수 있습니다.
    - SIngle-Pass 는 Sort Buffer 에 데이터를 모두 넣어서 정렬을 수행.
    - Two-Pass 는 정렬하는 칼럼과 PK 만 넣어서 정렬을 수행. 이후에 나머지 데이터와 병합.
    - 일반적으로 Single-Pass 가 성능이 좋으나, 데이터의 크기가 큰 경우라면 Two-Pass 가 더 성능적으로 우위
    - 의도적으로 SIngle-Pass 에서 Two-Pass 로 변경하고 싶다면 `max_length_for_sort_data` 변수의 값을 조절해야함. (MySQL 8.0.20 이전의 경우만 해당함.)

3. **문자열 정렬 튜닝:**
    - 문자열 칼럼은 값 전체를 사용해서 정렬하지 않고, `max_sort_lentgh` 값 만큼만 잘라서 정렬을 하도록 수행합니다.
    - 필요하다면 문자열을 정렬하는 경우엔 `max_sort_length` 값을 줄인다면 정렬 결과가 빨라질 수 있습니다.

▼ Considerations for ORDER BY

ORDER BY 절을 사용했을 때 정렬되어서 나오는 결과 순서가 동일해야한다면, `ORDER BY` 절에 고유한 값을 가진 칼럼을 포함시켜야 합니다.

```
mysql> SELECT * FROM ratings ORDER BY category;
+----+----------+--------+
| id | category | rating |
+----+----------+--------+
|  1 |        1 |    4.5 |
|  5 |        1 |    3.2 |
|  3 |        2 |    3.7 |
|  4 |        2 |    3.5 |
|  6 |        2 |    3.5 |
|  2 |        3 |    5.0 |
|  7 |        3 |    2.7 |
+----+----------+--------+

mysql> SELECT * FROM ratings ORDER BY category LIMIT 5;
+----+----------+--------+
| id | category | rating |
+----+----------+--------+
|  1 |        1 |    4.5 |
|  5 |        1 |    3.2 |
|  4 |        2 |    3.5 |
|  3 |        2 |    3.7 |
|  6 |        2 |    3.5 |
+----+----------+--------+
```

```
mysql> SELECT * FROM ratings ORDER BY category, id LIMIT 5;
+----+----------+--------+
| id | category | rating |
+----+----------+--------+
|  1 |        1 |    4.5 |
|  5 |        1 |    3.2 |
|  3 |        2 |    3.7 |
|  4 |        2 |    3.5 |
|  6 |        2 |    3.5 |
+----+----------+--------+
```

▼ Practice

Task 1: Performance Difference Based on the Presence of an Index

목적: 인덱스의 유무에 따라서 ORDER BY 절의 성능 차이를 비교해봅시다.

시나리오: 온라인 상품 관리 시스템에서 최근 추가한 상품 중 가격이 낮은 상위 10개의 상품을 조회

테이블: 상품 목록을 저장하는 `product` 테이블이 있습니다.

- `id` (기본 키)
- `name` (상품명)
- `price` (가격)
- `created_at` (생성 날짜)

1. `product` **테이블 생성**

```
CREATE TABLE product (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100),
    price DECIMAL(10, 2),
    created_at DATETIME
);
```

2. **테스트 데이터 삽입:**

3. **최근 추가한 상품 중 가격이 낮은 상위 10개의 상품 조회 쿼리:**

```
SELECT * FROM product
ORDER BY created_at DESC, price ASC
LIMIT 10;
```

4. **실행 계획과 쿼리 소요 시간 분석:**

```
EXPLAIN SELECT * FROM product
ORDER BY created_at DESC, price ASC
LIMIT 10;
```

```
EXPLAIN ANALYZE SELECT * FROM product
ORDER BY created_at DESC, price ASC
LIMIT 10;
```

5. **정렬 작업에 사용할 인덱스를 생성**

```
CREATE INDEX idx_created_at_price ON product(created_at DESC, price ASC);
```

6. **다시 실행 계획 확인 및 쿼리 소요 시간 분석:**

```
EXPLAIN SELECT * FROM product
ORDER BY created_at DESC, price ASC
LIMIT 10;
```

```
EXPLAIN ANALYZE SELECT * FROM product
ORDER BY created_at DESC, price ASC
LIMIT 10;
```

Task 2: Sort Buffer Size Tuning

목적: Disk 수준의 정렬을 하는 경우 Sort Buffer 사이즈를 올려서 성능 향상

1. **현재 MySQL 의 `sort_buffer_size` 사이즈 확인:**

```
SHOW VARIABLES LIKE 'sort_buffer_size';
```

2. **그리고 디스크 정렬을 했는지 알기 위한 현재 `sort_merge_passes` 값을 확인**

```
SHOW STATUS LIKE 'sort_merge_passes';
```

3. 실행 계획을 통해서 실습 쿼리가 filesort 를 이용하고 있는지 확인.

```
EXPLAIN SELECT * FROM product
ORDER BY name
LIMIT 1000;
```

4. 쿼리 소요 시간 확인:

```
EXPLAIN ANALYZE SELECT * FROM product
ORDER BY name
LIMIT 1000;
```

5. 쿼리 후 디스크 정렬이 발생했는지 `sort_merge_passes` 값이 증가 했는지 확인:

```
SHOW STATUS LIKE 'sort_merge_passes';
```

6. 현재 세션에서 `sort_buffer_size` 를 두배로 올려보고 확인:

```
SET SESSION sort_buffer_size = 2 * 262144;
```

```
SHOW VARIABLES LIKE 'sort_buffer_size';
```

7. 다시 쿼리를 실행하고 소요 시간을 확인:

```
EXPLAIN ANALYZE SELECT * FROM product
ORDER BY name
```

```
LIMIT 1000;
```

8. **다시 디스크 정렬이 얼마나 발생했는지 확인:**

```
SHOW STATUS LIKE 'sort_merge_passes';
```