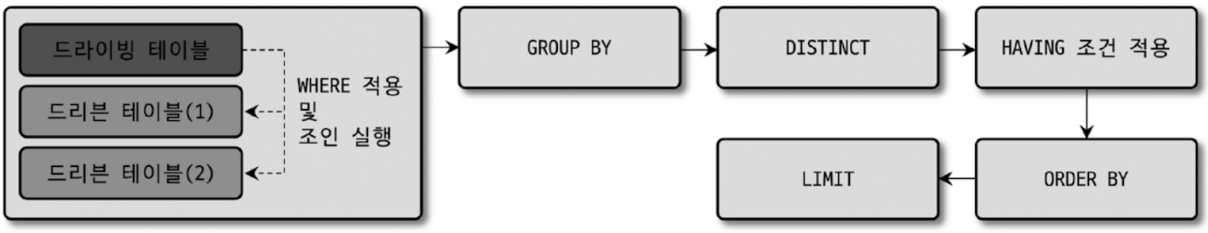
2주차

11.4. SELECT

`INSERT`, `UPDATE`에 비해 `SELECT`는 여러 개의 테이블로부터 데이터를 조합하여 빠르게 가져와야 하기 대문에 더 많은 주의를 기울여야 한다.

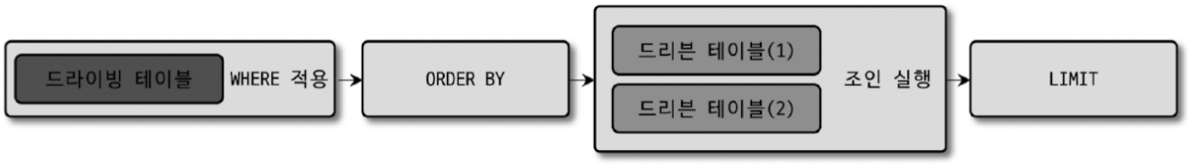
4.1 SELECT 절의 처리 순서

```
SELECT s.emp_no, COUNT(DISTINCT e.first_name) AS cnt
FROM salaries s
         INNER JOIN employees e ON e.emp_no = s.emp_no
WHERE s.emp_no IN (100001, 100002)
GROUP BY s.emp_no
HAVING AVG(s.salary) > 1000
ORDER BY AVG(s.salary)
LIMIT 10;
```



쿼리 절 실행 순서

> ❓❓❓
>
> **SQL에는 `ORDER BY`나 `GROUP BY` 절이 있더라도 인덱스를 이용해 처리할 때는 그 단계 자체가 불필요하므로 생략된다.**



예외적인 쿼리 절 실행 순서

첫 번째 테이블만 읽어 정렬 수행 뒤 나머지 테이블을 읽음 ⇒ `GROUP BY` 절 없이 `ORDER BY`만 사용된 쿼리에서 사용될 수 있는 순서

인라인 뷰를 이용하여 `ORDER BY`와 `LIMIT`의 순서가 바뀌기 때문에 위의 쿼리와 결과가 다를 수 있다.(`GROUP BY`도 마찬가지)

```
SELECT emp_no, cnt
FROM (SELECT s.emp_no, COUNT(DISTINCT e.first_name) AS cnt, MAX(s.salary) AS max
      FROM salaries s
               INNER JOIN employees e ON s.emp_no = e.emp_no
      WHERE s.emp_no IN (100001, 100002)
      GROUP BY s.emp_no
      HAVING MAX(s.salary) > 1000
      LIMIT 10) temp_view
ORDER BY max_salary;
```

☐ 인라인 뷰가 사용되면 10.3.2.8절 'DERIVED'에서 살펴본 것처럼 임시 테이블이 사용되기 때문에 주의해야 한다.

> MySQL 8.0 버전에서는 `FROM` 절에 위치한 서브쿼리(Derived Table)를 외부 쿼리와 병합하면서 쿼리를 최적화할 수 있다. 이 경우도 결국 `FROM` 절의 서브쿼리가 아닌 조인으로 실행되는 형태이다.

4.2 WHERE 절과 GROUP BY 절, ORDER BY 절의 인덱스 사용

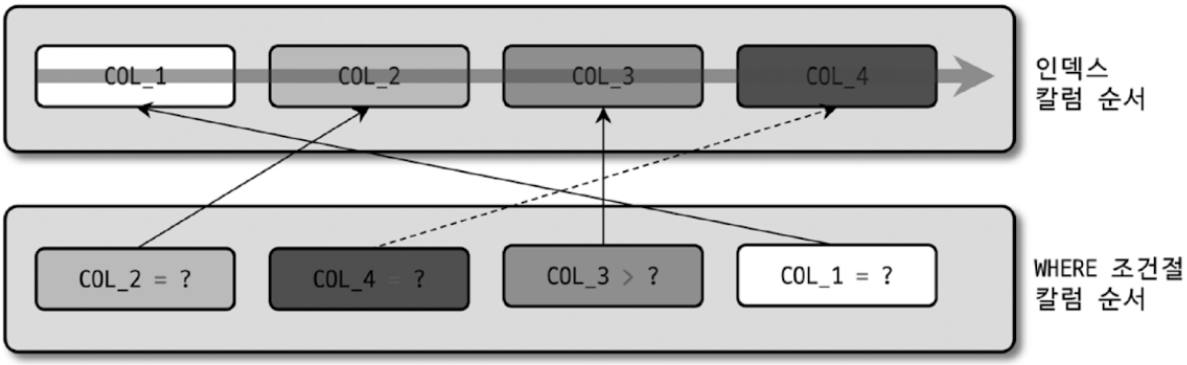
1. 인덱스를 사용하기 위한 기본 규칙

인덱스를 사용하기 위해서는 기본적으로 인덱스된 컬럼의 값 자체를 변환하지 않고 그대로 사용한다는 조건을 만족해야 한다.

저장하고자 하는 값의 타입에 맞춰 컬럼의 타입을 선정하고, SQL을 작성할 때 데이터의 타입에 맞추어 비교 조건을 사용하자.

2. WHERE 절의 인덱스 사용

1. **작업 범위 결정 조건**
   - `WHERE` 절에서 동등 비교 조건이나 `IN`으로 구성된 조건에 사용된 컬럼들이 인덱스의 컬럼 구성과 좌측에서부터 비교하였을 때 얼마나 일치하는가에 따라 달라짐
2. **체크 조건**



결합 인덱스를 사용할 때 `WHERE` 조건절에 나열된 순서가 인덱스와 달라도 MySQL 서버 옵티마이저는 인덱스를 사용할 수 있는 조건들을 뽑아 최적화를 수행할 수 있다.

`COL_3`의 조건이 동등 비교 조건이 아닌 범위 비교 조건이므로 뒤 컬럼인 `COL_4`의 조건은 작업 범위 결정 조건으로 사용되지 못하고 체크 조건으로 사용

```
SELECT *
FROM employees
WHERE first_name = 'Shahab'
   OR last_name = 'Gelosh';
```

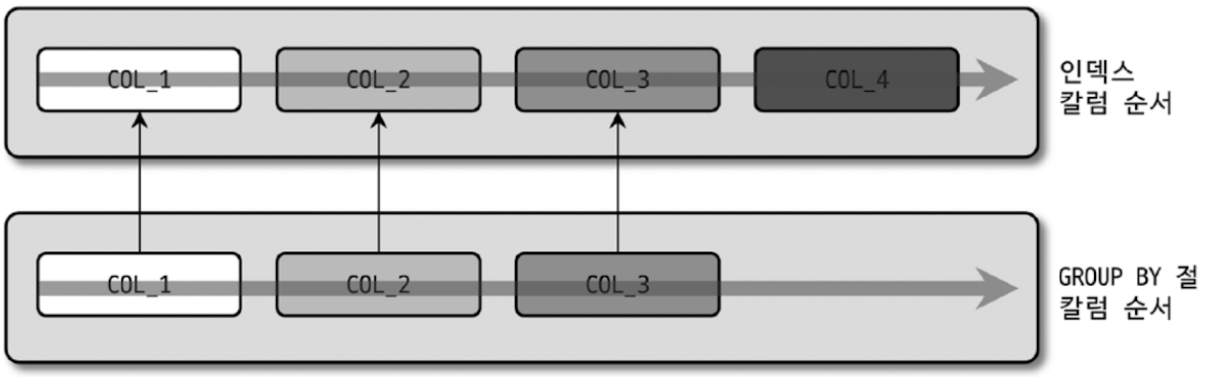
`first_name = ‘ Shahab’` 조건은 인덱스를 이용할 수 있지만 `last_name = ‘Gelosh’`는 인덱스 사용이 불가능하다. `OR`이기 때문에 ‘인덱스 레인지 스캔 + 풀 테이블 스캔’이다. ⇒ 그냥 ‘풀 테이블 스캔’ 한 번으로 실행됨

`OR`가 아닌 `AND` 연산자라면 인덱스를 이용할 수 있다. `WHERE` 조건절에 `OR` 연산자가 있으면 주의!

QUIZ?

3. GROUP BY 절의 인덱스 사용

- `GROUP BY` 절에 명시된 컬럼이 인덱스 컬럼의 순서와 위치가 같아야 한다.
- 인덱스를 구성하는 컬럼 중에서 뒤쪽에 있는 컬럼은 `GROUP BY` 절에 명시되지 않아도 인덱스를 사용할 수 있지만 인덱스의 앞쪽에 있는 컬럼이 `GROUP BY` 절에 명시되지 않으면 인덱스를 사용할 수 없다.
- `WHERE` 조건절과 달리 `GROUP BY` 절에 명시된 컬럼이 하나라도 인덱스에 없으면 `GROUP BY` 절은 전혀 인덱스를 이용하지 못한다.



GROUP BY 절의 인덱스 사용 규칙

But, `WHERE` 조건절에 동등 비교 조건으로 사용된 컬럼은 `GROUP BY` 절에 포함되지 않아도 인덱스를 이용한 `GROUP BY`가 가능한 경우도 있다. ⇒ `WHERE` 조건절에 동등 비교 조건으로 사용된 컬럼을 `GROUP BY` 절로 옮겨도 똑같은 결과가 조회된다면 `WHERE` 절과 `GROUP BY` 절이 모두 인덱스를 사용할 수 있는 쿼리로 판단하면 된다.

QUIZ?

다음과 같이 인덱스가 생성되어 있다.

```
ALTER TABLE ... ADD INDEX ix_test (col_1, col_2, col_3);
```

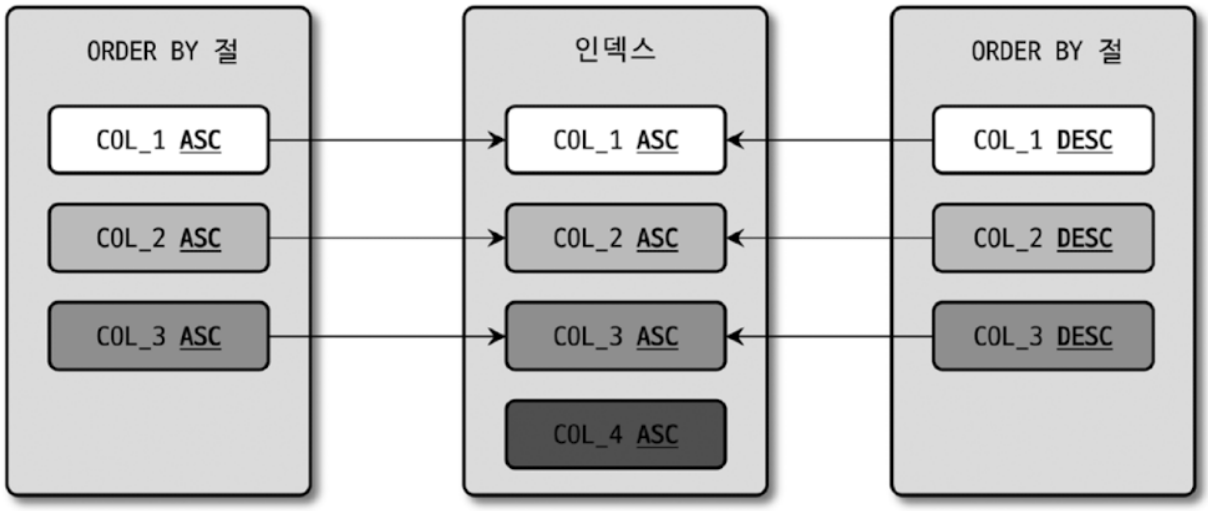
다음 중 인덱스를 이용하지 못하는 경우는?

```
1. ... GR
```

4. ORDER BY 절의 인덱스 사용

`GROUP BY` 절의 인덱스 사용 요건과 거의 흡사하지만, 조건이 하나 더 있다.

- 정렬되는 각 컬럼의 오름차순(ASC) 및 내림차순(DESC) 옵션이 인덱스와 같거나 정반대인 경우에만 사용 가능하다.



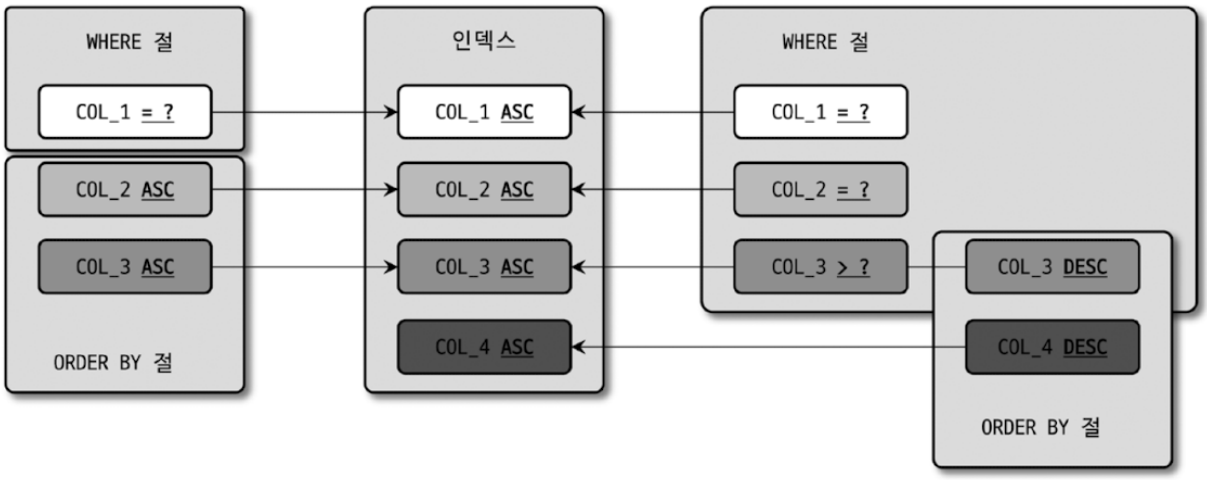
ORDER BY 절의 인덱스 사용 규칙

5. WHERE 조건과 ORDER BY(또는 GROUP BY) 절의 인덱스 사용

`WHERE` 절과 `ORDER BY` 절이 같이 사용된 하나의 쿼리 문장은 다음 3가지 중 한 가지 방법으로만 인덱스를 이용한다.

- `WHERE` 절과 `ORDER BY` 절이 동시에 같은 인덱스를 이용
  - 가장 좋은 방법
  - `WHERE` 절의 비교 조건에 사용되는 컬럼과 `ORDER BY` 절의 정렬 대상 컬럼이 모두 하나의 인덱스에 연속해 포함되어 있을 때
- `WHERE` 절만 인덱스를 이용
  - `WHERE` 절의 조건에 일치하는 레코드 건수가 적을 때 효율적
  - 이 경우, `ORDER BY` 절은 인덱스를 이용한 정렬이 불가능
  - 인덱스를 통해 검색된 결과 레코드를 별도의 정렬 처리 과정(Using Filesort)을 거쳐 정렬 수행
- `ORDER BY` 절만 인덱스를 이용
  - 아주 많은 레코드를 조회해 정렬해야 하는 경우 종종 사용
  - `ORDER BY` 절의 순서대로 인덱스를 읽으면서 레코드 한 건씩 `WHERE` 절의 조건에 일치하는지 비교 (일치하지 않으면 버림)

`WHERE` 절에서 동등 비교 조건으로 비교된 컬럼과 `ORDER BY` 절에 명시된 컬럼이 순서대로 빠짐없이 인덱스 컬럼의 왼쪽부터 일치해야 한다.



WHERE 절과 ORDER BY 절의 인덱스 사용 규칙

`WHERE` 절에 범위 비교 조건이 사용되어도 해당 컬럼이 `ORDER BY`에 사용되므로 인덱스를 사용 가능

> MySQL 8.0에 새롭게 추가된 인덱스 스킵 스캔 최적화는 인덱스에 나열된 컬럼 순서상 선행되는 컬럼의 조건이 없다고 하더라도 인덱스의 후행 컬럼을 이용할 수 있게 해준다.

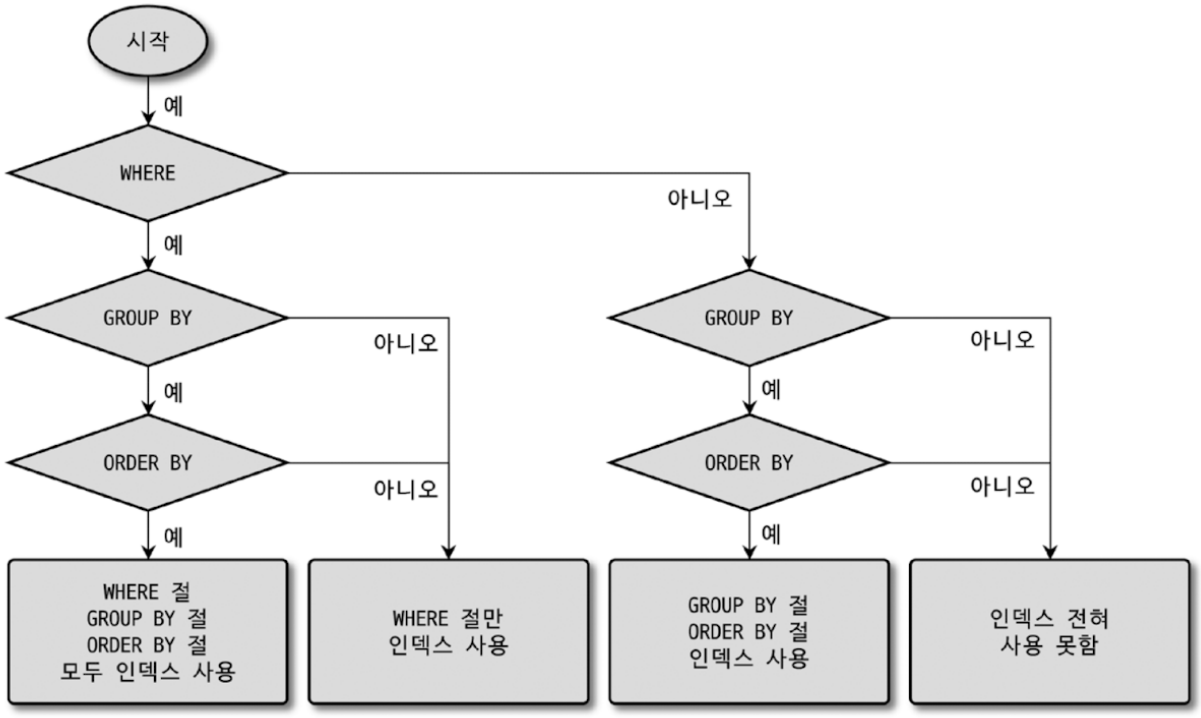
6. GROUP BY 절과 ORDER BY 절의 인덱스 사용

`GROUP BY` 절과 `ORDER BY` 절이 동시에 사용된 쿼리에서 두 절이 모두 하나의 인덱스를 사용해 처리되기 위해서는 `GROUP BY` 절에 명시된 컬럼과 `ORDER BY` 절에 명시된 컬럼의 순서와 내용이 모두 같아야 한다. 둘 중 하나라도 인덱스를 사용할 수 없으면 모두 인덱스를 사용할 수 없다.

MySQL 5.7 이하: `GROUP BY` 컬럼에 대한 정렬도 수행

MySQL 8.0 이상: `GROUP BY` 컬럼 정렬 보장 X

7. WHERE 조건과 ORDER BY 절, GROUP BY 절의 인덱스 사용



인덱스 사용 여부 판단

4.3 WHERE 절의 비교 조건 사용 시 주의사항

1. NULL 비교

MySQL에서는 `NULL` 값이 포함된 레코드도 인덱스로 관리된다.

```
SELECT * FROM titles WHERE to_date IS NULL; // 인덱스 레인지 스캔
SELECT * FROM titles WHERE ISNULL(to_date); // 인덱스 레인지 스캔
SELECT * FROM titles WHERE ISNULL(to_date) = 1; // 풀 스캔
SELECT * FROM titles WHERE ISNULL(to_date) = true; // 풀 스캔
```

`IS NULL` 연산자 권장

2. 문자열이나 숫자 비교

비교 대상 컬럼의 타입에 맞는 리터럴을 사용해 비교하자.

```
SELECT * FROM employees WHERE first_name = 1;
```

옵티마이저는 숫자 타입이 우선순위가 높아 `first_name` 컬럼의 타입 변환이 필요해 인덱스를 사용하지 못한다.

3. 날짜 비교

DATE, DATETIME 문자열 비교

컬럼이 아닌 상수를 변형하는 형태로 쿼리를 작성하자~

DATE와 DATETIME 비교

`DATE` 타입 컬럼과 `DATETIME` 값을 비교하면 MySQL은 `DATE` 타입을 `DATETIME`으로 변환 ⇒ 성능 문제 보다는 결과에 주의해 사용

`DATE()` 함수를 사용해 명시적으로 `DATETIME` 값을 변환하도록 하자.

DATETIME과 TIMESTAMP 비교

컬럼이 `DATETIME`이면 `DATETIME`으로 변환(`FROM_UNIXTIME()`)하여 비교, 컬럼이 `TIMESTAMP`이면 `TIMESTAMP`로 변환(`UNIX_TIMESTAMP()`)하여 비교

Short-Circuit Evalutation

```
boolean inTransaction;
if (inTransaction && hasModified()) {
    commit();
}
```

`if` 문에서 `inTransaction`이 `false`이면 `hasModified()`를 호출(조건 확인)하지 않는다. ⇒ "Short-Circuit Evalutation"

`WHERE` 절에 나열되는 조건의 순서에 따라 쿼리 성능에 영향을 미칠 수 있다. 하지만 `WHERE` 절의 조건 중에서 인덱스를 사용할 수 있는 조건이 있다면 Short-Circuit Evaluation과 무관하게 해당 조건을 최우선으로 사용한다.

쿼리를 작성할 때 가능하면 복잡한 연산 또는 다른 테이블의 레코드를 읽어야 하는 서브쿼리 조건 등은 WHERE 절에서 뒤쪽으로 배치하는 것이 성능상 도움이 될 수 있다.

4.4 DISTINCT

특정 컬럼의 유니크한 값을 조회하려면 `SELECT` 쿼리에 `DISTINCT`를 사용한다.

조인을 할 때 레코드 중복을 막기 위해 `DISTINCT`를 남용하지 않도록 주의하자.

☐ 9.2.5 'DISTINCT 처리' 살펴보기

> 테이블 간 조인 쿼리를 작성하는 경우 각 테이블 간의 조인이 1:1인지, 1:N 조건인지 업무적 특성을 잘 이해하는 것이 중요하다!

4.5 LIMIT n

쿼리 결과에서 지정된 순서에 위치한 레코드만 가져오고자 할 때 사용한다.

`LIMIT {[ offset ,] row_count | row_count OFFSET offset }`

오라클 `ROWNUM`과는 다르니 주의

`LIMIT`의 중요한 특성은 `LIMIT`에서 필요한 레코드 건수만 준비되면 즉시 쿼리를 종료한다. (ex. `SELECT * FROM employees WHERE emp_no BETWEEN 10001 AND 10010 ORDER BY first_name LIMIT 0, 5;` ⇒ 상위 5건만 정렬되면 작업 종료. 그렇다고 성능 향상 효과가 크지는 않을 수 있음)

`LIMIT`의 인자로 표현식이나 서브쿼리를 사용할 수 없다.

offset이 매우 커지는 경우 성능이 저하될 수 있기 때문에 `WHERE` 조건 절로 읽어야 할 위치를 찾고 그 위치에서 10개만 읽는 형태의 쿼리를 사용하는 것이 좋다.

QUIZ?

4.6 COUNT()

레코드의 건수를 반환하는 함수. 컬럼이나 표현식, * 등을 인자로 받는다. *은 레코드 자체를 의미함.

InnoDB와 달리 MyISAM 스토리지 엔진을 사용하는 테이블은 테이블 메타 정보에 전체 레코드 건수를 관리하기 때문에 `WHERE` 절 없이 단순한 `COUNT(*)` 쿼리는 바로 결과를 반환할 수 있어 빠르게 처리된다.

> 대략적인 레코드 건수로 충분하다면 `SHOW TABLE STATUS` 명령으로 통계 정보를 참조하는 것도 좋은 방법이다. `ANALYZE TABLE` 명령으로 통계 정보를 갱신할 수 있다.

`COUNT(*)`와 무관한 `ORDER BY`, `LEFT JOIN` 등의 구문을 함께 사용하지 않도록 하자. 8.0 이후 버전은 `ORDER BY` 절은 옵티마이저가 무시한다.

> Spring Data JPA를 사용하여 `Page<T>` 타입으로 반환을 하는 경우 데이터 조회용 쿼리와 레코드 건 수 반환용 쿼리, 총 두 번의 쿼리를 실행한다. 전체 레코드 개수가 필요 없으면 다음 페이지가 존재하는지 여부를 함께 포함하는 `Slice<T>`를 사용해 쿼리를 최적화할 수 있을 것 같다. 프로젝트에 적용하기

4.7 JOIN

쿼리 패턴별로 `JOIN`이 어떻게 인덱스를 사용하는지 살펴본다.

1. JOIN 순서와 인덱스

인덱스 레인지 스캔

- 인덱스 탐색(index seek): 인덱스에서 조건을 만족하는 값이 저장된 위치를 찾는 과정
- 인덱스 스캔(index scan): 인덱스 탐색으로 탐색된 위치부터 필요한 만큼 인덱스를 차례대로 읽는 과정

보통 인덱스를 이용하는 작업에서는 가져오는 레코드 수가 소량이기 때문에 인덱스 스캔은 부하가 작지만 인덱스 탐색은 상대적으로 부하가 높다.

조인 작업에서 드라이빙 테이블은 인덱스 탐색 1번 이후에는 인덱스 스캔만 실행하면 된다. 하지만 드리븐 테이블에서는 드라이빙 테이블에서 읽은 레코드 수만큼 인덱스 탐색, 인덱스 스캔을 반복해야 한다. ⇒ 옵티마이저는 드리븐 테이블을 최적으로 읽을 수 있게 실행 계획을 수립한다.

옵티마이저는 조인 작업 시, 인덱스가 있는 테이블을 드리븐 테이블로 선택하고 둘 다 있거나 둘 다 없으면 알아서 최적의 방법을 선택한다.(통계 정보에 있는 레코드 수에 따라)

> MySQL 8.0.18 이전에는 인덱스가 모두 없는 테이블 간 조인을 수행할 때 block nested loop join을 이용했지만 8.0.18 부터는 hash join으로 처리한다.

2. JOIN 컬럼의 데이터 타입

`JOIN` 컬럼 간의 비교에서 컬럼 주의

- 데이터 타입이 다른 경우(`VARCHAR`와 `CHAR`, `INT`와 `BIGINT` 등은 상관 무)
- 문자 집합이나 콜레이션이 다른 경우
- 부호 존재 여부 (signed와 unsigned)

3. OUTER 조인의 성능 및 주의사항

MySQL 옵티마이저는 아우터로 조인되는 테이블을 드라이빙 테이블로 절대 선택하지 못 하기 때문에 풀 스캔이 필요한 테이블을 드라이빙 테이블로 선택하는 문제가 발생할 수 있다. ⇒ 필요한 데이터와 조인되는 테이블 간 관계를 정확히 파악해 필요한 경우가 아니라면 아우터 조인보다 이너 조인을 사용하자.

아우터로 조인되는 테이블에 대한 조건을 `WHERE` 절에 명시하면 옵티마이저는 `LEFT JOIN`을 `INNER JOIN`으로 변환하여 실행한다. ⇒ 정상적인 `LEFT JOIN`이 수행되도록 하기 위해 `WHERE` 절이 아닌 `ON` 절에 조건을 명시해야 한다. 단, 안티 조인 효과를 기대하는 경우(아우터 조인 테이블의 조인 컬럼이 NULL인 조건을 검색하는 경우)에는 예외

4. JOIN과 외래키

외래키는 조인과 전혀 연관이 없다. 외래키 생성은 데이터 무결성 보장이 목적이다. ⇒ 참조 무결성

5. 지연된 조인(Delayed Join)

조인의 결과를 `GROUP BY`나 `ORDER BY`를 하는 것은 조인 전에 하는 것보다 더 많은 레코드를 처리해야 한다. 조인이 실행되기 전 `GROUP BY`와 `ORDER BY`를 처리하는 방식이 지연된 조인이다.

지연된 쿼리 변경 가능 조건

- `LEFT (OUTER) JOIN`인 경우 드라이빙 테이블과 드리븐 테이블이 1:1 또는 N:1 관계이어야 한다.
- `INNER JOIN`인 경우 드라이빙 테이블과 드리븐 테이블이 1:1 또는 N:1의 관계임과 동시에 드라이빙 테이블에 있는 레코드는 드리븐 테이블에 모두 존재해야 한다.

```
SELECT e.*
FROM salaries s,
     employees e
WHERE e.emp_no = s.emp_no
  AND s.emp_no BETWEEN 10001 AND 13000
GROUP BY s.emp_no
ORDER BY SUM(s.salary) DESC
LIMIT 10;
```

```
SELECT e.*
FROM (SELECT s.emp_no
      FROM salaries s
      WHERE s.emp_no BETWEEN 10001 AN
      GROUP BY s.emp_no
      ORDER BY SUM(s.salary) DESC
      LIMIT 10) x,
     employees e
WHERE x.emp_no = e.emp_no;
```

> 지연된 조인은 조인 개수를 줄이는 것 뿐 아니라 `GROUP BY`나 `ORDER BY` 처리가 필요한 레코드 수도 줄이는 역할도 한다.

6. 래터럴 조인(Lateral Join)

8.0 버전 이전에는 그룹별로 n 건 씩 가져오는 쿼리를 작성할 수 없었지만 8.0 부터는 래터럴 조인을 이용해 특정 그룹별로 서브쿼리를 실행하여 그 결과와 조인하는 것이 가능하다.

`FROM` 절에 사용된 서브쿼리에서 외부 쿼리의 `FROM` 절에 정의된 테이블의 컬럼을 참조할 수 있다.

```
SELECT *
FROM employees e
         LEFT JOIN LATERAL (SELECT *
                            FROM salaries s
                            WHERE s.emp_no = e.emp_no
                            ORDER BY s.from_date DESC
                            LIMIT 2) s2 ON s2.emp_no = e.emp_no
WHERE e.first_name = 'Matt';
```

> `LATERAL` 키워드를 가진 서브쿼리는 조인 순서상 후순위로 밀리고, 외부 쿼리의 결과 레코드 단위로 임시 테이블이 생성되기 때문에 꼭 필요한 경우에 사용하자.

7. 실행 계획으로 인한 정렬 흐트러짐

8.0 버전 이전에는 네스티드 루프 조인만 가능했지만 8.0 부터는 해시 조인 방식이 도입되었다.

네스티드 루프 조인은 드라이빙 테이블을 읽은 순서대로 레코드가 조회되는 것이 일반적이지만 해시 조인은 그렇지 않다. 옵티마이저에 의해 매번 달라질 수 있으므로 순서가 중요하다면 `ORDER BY` 절을 명시적으로 사용하자.