9️⃣ 옵티마이저와 힌트

9.3 고급 최적화

MySQL 서버의 옵티마이저가 실행 계획을 수립할 때 통계 정보와 옵티마이저 옵션을 결합해서 최적의 실행 계획을 수립하게 된다. 옵티마이저 옵션은 크게 조인 관련된 옵티마이저 옵션과 옵티마이저 스위치로 구분할 수 있다. 조인 관련된 옵티마이저 옵션은 MySQL 서버 초기 버전부터 제공되던 옵션이지만, 많은 사람이 그다지 신경 쓰지 않는 편이다. 하지만 조인이 많이 사용되는 서비스에서는 알아야 하는 부분이기도 하다. 그리고 옵티마이저 스위치는 MySQL 5.5 버전부터 지원되기 시작했는데, 이들은 MySQL 서버의 고급 최적화 기능들을 활성화할지를 제어하는 용도로 사용된다.

9.3.1 옵티마이저 스위치 옵션

옵티마이저 스위치 옵션은 optimizer_switch 시스템 변수를 이용해서 제어하는데, optimizer_switch 시스템 변수에는 여러 개의 옵션을 세트로 묶어서 설정하는 방식을 사용한다. optimizer_switch 시스템 변수에 설정할 수 있는 최적화 옵션을 다음과 같다.

옵티마이저 스위치 이름	기본값	설명
batched_key_access	off	BKA 조인 알고리즘을 사용할지 여부 결정
block_nested_loop	on	Block Nested Loop 조인 알고리즘을 사용할지 여부 설정

등등 ..

각각의 옵티마이저 스위치 옵션은 “default”와 “on”, “off” 중에서 하나를 설정할 수 있는데, “on”으로 설정되면 해당 옵션을 활성화하고, “off”를 설정하면 해당 옵션을 비활성화한다. 그리고 “default” 를 설정하면 기본값이 적용된다. 옵티마이저 스위치 옵션은 글로벌과 세션별 모두 설정할 수 있는 시스템 변수이므로 MySQL 서버 전체적으로 또는 현재 커넥션에 대해서만 다음과 같이 설정할 수 있다.

```
-- // MySQL 서버 전체적으로 옵티마이저 스위치 설정
mysql> SET GLOBAL optimizer_switch='index_merge=on, index_merge_union=on,...';

-- // 현재 커넥션의 옵티마이저 스위치만 설정
mysql> SET SESSION optimizer_switch='index_merge=on, index_merge_union=on,...';
```

또한 다음과 같이 “SET_VAR” 옵티마이저 힌트를 이용해 현재 쿼리에만 설정할 수도 있다.

```
mysql> SELECT /*+ SET_VAR(optimizer_switch='condition_fanout_filter=off') */
      ...
      FROM ...
```

9.3.1.1 MRR과 배치 키 엑세스 (mrr & batched_key_access)

MRR은 “Multi-Range Read'를 줄여서 부르는 이름인데, 매뉴얼에서는 DS -MRR(Disk Sweep Multi-Range Renge Read)이라고도 한다. MySQL 서버에서 지금까지 지원하던 조인 방식은 드라이빙 테이블 (조인에서 제일 먼저 읽는 테이블)의 레코드를 한 건 읽어서 드리블 테이블(조인되는 테이블에서 드라이빙이 아닌 테이블들)의 일치하는 레코드를 찾아서 조인을 수행하는 것이었다. 이를 네스티드 루프 조인(Nested Loop Jomn)이라고 한다.
MySOL 서버의 내부 구조상 조인 처리는 MySQL 엔진이 처리 하지만, 실제 레코드를 검색하고 읽는 부분은 스토리지 엔진이 담당한다. 이때 드라이빙 테이블의 레코드 건별로 드리븐 테이블의 레코드를 찾으면 레코드를 찾고 읽는 스토리지 엔진에서는 아무런 최적화를 수행할 수가 없다.
이 같은 단점을 보완하기 위해 MSQL 서버는 조인 대상 테이블 중 하나로부터 레코드를 읽어서 조인 버퍼에 버퍼링한다. 즉, 드라이빙 테이블의 레코드를 읽어서 드리븐 테이블과의 조인을 즉시 실행하지 않고 조인 대상을 버퍼링하는 것이다. 조인 버퍼에 레코드가 가득 차면 비로소 MySQL 엔진은 버퍼링 된 레코드를 스토리지 엔진으로 한 번에 요청한다. 이렇게 함으로써 스토리지 엔진은 읽어야 할 레코드 들을 데이터 페이지에 정렬된 순서로 접근해서 디스크의 데이터 페이지 읽기를 최소화할 수 있는 것이 다. 물론 데이터 페이지가 메모리(InnoDB 버퍼 풀)에 있다고 하더라도 버퍼 풀의 접근을 최소화할 수 있는 것이다.
이러한 읽기 방식을 NRR(Multi-Range Read)이라고 하며, MIRR을 응용해서 실행되는 조인 방식을 BKA(Batched Key Access) 조인이라고 한다. BKA 조인 최적화는 기본적으로 비활성화돼 있는데.
이는 BKA 조인의 단점이 있기 때문이다. 쿼리의 특성에 따라 BKA 조인이 큰 도움이 되는 경우도 있지만, BKA 조인을 사용하게 되면 부가적인 정렬 작업이 필요해지면서 오히려 성능에 안 좋은 영향을 미치는 경우도 있다.

9.3.1.2 블록 네스티드 루프 조인 (block_nested_loop)

MySQL 서버에서 사용되는 대부분의 조인은 네스티드 루프 조인(Nested Loop Join)인데, 조인의 연 결 조건이 되는 칼럼에 모두 인덱스가 있는 경우 사용되는 조인 방식이다. 다음 예제 쿼리는 employees 테이블에서 first_name 조건에 일치하는 레코드 1건을 찾아서 salaries 테이블의 일치하는 레코드를 찾 는 형태의 조인을 실

행한다.

```
mysql> EXPLAIN
        SELECT *
        FROM employees e
          INNER JOIN salaries s ON s.emp_no=e.emp_no
                    AND s.from_date<=NOW()
                    AND s.to_date>=NOW()
        WHERE e.first_name='Amor';
+----+-------------+-------+------+--------------+------+-------------+
| id | select_type | table | type | key          | rows | Extra       |
+----+-------------+-------+------+--------------+------+-------------+
|  1 | SIMPLE      | e     | ref  | ix_firstname |    1 | NULL        |
|  1 | SIMPLE      | s     | ref  | PRIMARY      |   10 | Using where |
+----+-------------+-------+------+--------------+------+-------------+
```

이러한 형태의 조인은 다음과 같이 프로그래밍 언어에서 마치 중첩된 반복 명령을 사용하는 것처럼 작동한다고 해서 네스티드 루프 조인(Nested Loop Join)이라고 한다. 다음 의사 코드(Psuedo Code)에서도 알 수 있듯이 레코드를 읽어서 다른 버퍼 공간에 저장하지 않고 즉시 드리븐 테이블의 레코드를 찾아서 반환한다는 것을 알 수 있다.

```
for(row1 IN employees){
  for(row2 IN salaries){
    if(condition_matched) return (row1, row2);
  }
}
```

네스티드 루프 조인과 블록 네스티드 루프 조인(Block Nested Loop Join)의 가장 큰 차이는 조인 버퍼(join buffer_size 시스템 설정으로 조정되는 조인을 위한 버퍼)가 사용되는지 여부와 조인에서 드라 이빙 테이블과 드리븐 테이블이 어떤 순서로 조인되느냐다. 조인 알고리즘에서 "Block"이라는 단어가 사용되면 조인용으로 별도의 버퍼가 사용됐다는 것을 의미

하는데, 조인 쿼리의 실행 계획에서 Extra 칼럼에 Using Join buffer"라는 문구가 표시되면 그 실행 계획은 조인 버퍼를 사용한다는 것을 의미 한다.
조인은 드라이빙 테이블에서 일치하는 레코드의 건수만큼 드리븐 테이블을 검색하면서 처리된다. 즉 드라이빙 테이블은 한 번에 쭉 읽지만, 드리븐 테이블은 여러 번 읽는다는 것을 의미한다. 예를 들어, 드라이빙 테이블에서 일치하는 레코드가 1,000건이었는데, 드리븐 테이블의 조인 조건이 인덱스를 이용할 수 없었다면 드리븐 테이블에서 연결되는 레코드를 찾기 위해 1,000번의 풀 테이블 스캔을 해야한다. 그래서 드리븐 테이블을 검색할 때 인덱스를 사용할 수 없는 쿼리는 상당히 느려지며, 옵티마이저는 최대한 드리븐 테이블의 검색이 인덱스를 사용할 수 있게 실행 계획을 수립한다.

이 쿼리의 실행 계획을 살펴보면 다음과 같이 dept_emp 테이블이 드라이빙 테이블이며, employees 테이블을 읽을 때는 조인 버퍼(Join buffer)를 이용해 블록 네스티드 루프 조인을 한다는 것을 Extra 칼럼의 내용으로 알 수 있다.

```
+----+-------------+-------+-------+-------------+-----------------------------------------+
| id | select_type | table | type  | key         | Extra                                   |
+----+-------------+-------+-------+-------------+-----------------------------------------+
|  1 | SIMPLE      | de    | range | ix_fromdate | Using index condition                   |
|  1 | SIMPLE      | e     | range | PRIMARY     | Using join buffer (block nested loop) |
+----+-------------+-------+-------+-------------+-----------------------------------------+
```

그림 9.9는 이 쿼리의 실행 계획에서 조인 버퍼가 어떻게 사용되는지 보여준다. 단계별로 잘라서 실행 내역을 한 번 살펴보자.

1. dept_emp 테이블의 ix_fromdate 인덱스를 이용해(from_date>'1995-01-01') 조건을 만족하는 레코드를 검색한다.

2. 조인에 필요한 나머지 칼럼을 모두 dept_emp 테이블로부터 읽어서 조인 버퍼에 저장한다.

3. employees 테이블의 프라이머리 키를 이용해 (emp_no<109004) 조건을 만족하는 레코드를 검색한다.

4. 3번에서 검색된 결과(employees)에 2번의 캐시된 조인 버퍼의 레코드(dept_emp)를 결합해서 반환한다.

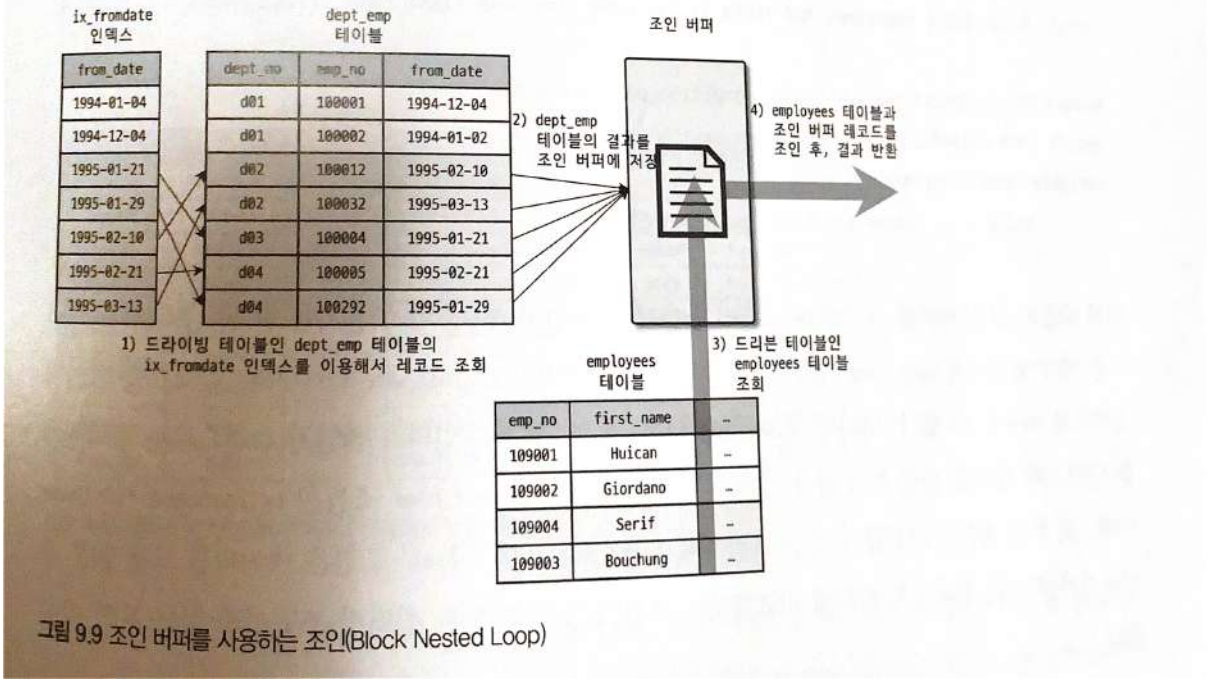


그림 9.9 조인 버퍼를 사용하는 조인(Block Nested Loop)

그림 9.9에서 중요한 점은 조인 버퍼가 사용되는 쿼리에서는 조인의 순서가 거꾸로인 것처럼 실행된다는 것이다. 위에서 설명한 절차의 4번 단계가 employees 테이블의 결과를 기준으로 dept. emp 테이블의 결과를 결합(병합)한다는 것을 의미한다. 실제 이 쿼리의 실행 계획상으로는 dept.emp 테이블이 드라이 빙 테이블이 되고, employees 테이블이 드리븐 테이블이 된다. 하지만 실제 드라이빙 테이블의 결과는 조인 버퍼에 담아두고, 드리븐 테이블을 먼저 읽고 조인 버퍼에서 일치하는 레코드를 찾는 방식으로 처 리된다. 일반적으로 조인이 수행된 후 가져오는 결과는 드라이빙 테이블의 순서에 의해 결정되지만, 조 인 버퍼가 사용되는 조인에서는 결과의 정렬 순서가 흐트러질 수 있음을 기억해야 한다.

9.3.1.3 인덱스 컨디션 푸시다운 (index_condition_pushdown)

MySOL 5.6 버전부터는 인덱스 컨디션 푸시다운(Index Condition Pushdown)이라는 기능이 도 됐다. 사실 인덱스 컨디션 푸시다운은 너무 비효율적이어서 이미 훨씬 오래 전부터 개선됐어야 할 기능인데, 이제서야 보완된 것이다. 우선 간단한 테스트를 위해 다음과 같이 인

덱스를 생성하고, 옵티마이저 스위치를 조정해서 인덱스 컨디션 푸시다운 기능을 비활성화 하자.

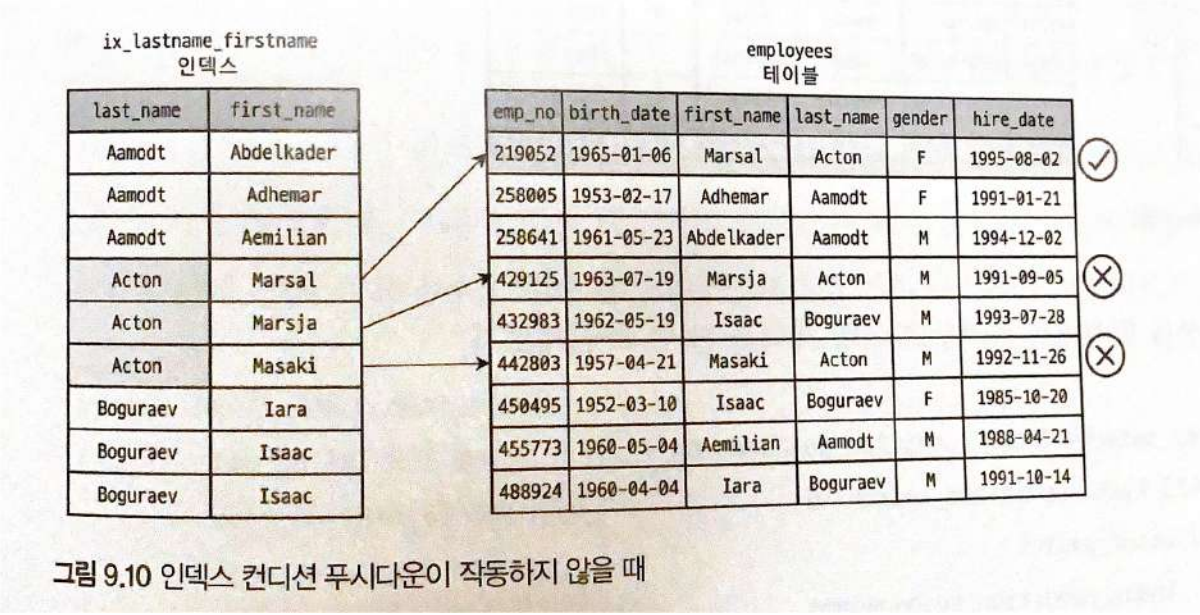


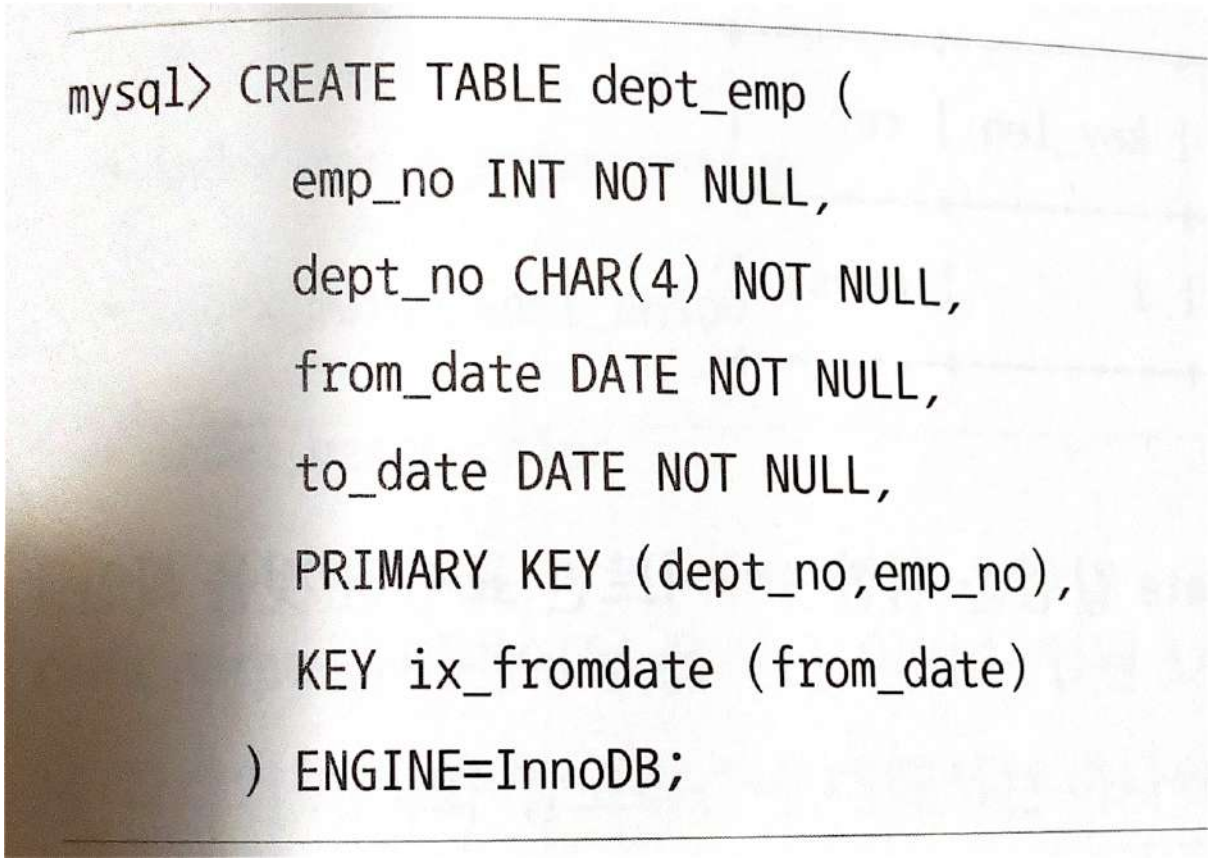
그림 9.10 인덱스 컨디션 푸시다운이 작동하지 않을 때

하지만 여기서 한 번만 더 생각하면 first_name LIKE ‘%sal' 조건을 처리하기 위해 이미 한 번 읽은 ix_lastname_ firstname 인덱스의 first_name 칼럼을 이용하지 않고 왜 다시 테이블의 레코드를 읽어서 처리 했는지 궁금할 것이다. 인덱스의 first name 칼럼을 이용해서 비교했다면 불필요한 2건의 레코드는 테 이블에서 읽지 않아도 됐을 텐데 말이다. 사실 first_name LIKE '%sal’ 조건을 누가 처리하느냐에 따라 인덱스에 포함된 first_name 칼럼을 이용할지 또는 테이블의 first_name 칼럼을 이용할지가 결정된다.
그림 9.10에서 인덱스를 비교하는 작업은 실제 InnODB 스토리지 엔진이 수행하지만 테이블의 레코드 에서 first name 조건을 비교하는 작업은 MySQL 엔진이 수행하는 작업이다. 그런데 MySQL 5.5 버전 까지는 인덱스를 범위 제한 조건으로 사용하지 못하는 first name 조건은 MySQL 엔진이 스토리지 엔 진으로 아예 전달해주지 않았다. 그래서 스토리지 엔진에서는 불필요한 2건의 테이블 읽기를 수행할 수밖에 없었던 것이다.

9.3.1.4 인덱스 확장 (use_index_extensions)

use_index_extensions 옵티마이지 옵션은 InnoDB 스토리지 엔진을 사용하는 데이블에서 세컨더리 인 덱스에 자동으로 추가된 프라이머리 기를 활용할 수 있게 할지를 컬칭하는 유신이다. 우선 "'세컨다리 인덱스에 자동으로 추가된 프라이머리 키"의 의미와 이로 인해 어떤 성능상 장점이 있는지를 살펴보자.

이미 8.9절 ‘클러스터링 인덱스'에서 살펴본 바와 같이 InnoDB 스토리지 엔진은 프라이머리 키를 클러 스터링 키로 생성한다. 그래서 모든 세컨더리 인덱스는 리프 노트에 프라이머리 키 값을 가진다. 예를 들어, 다음과 같이 프라이머리 키와 세컨더리 인덱스를 가진 데이블을 가정해보자.



```
mysql> CREATE TABLE dept_emp (
        emp_no INT NOT NULL,
        dept_no CHAR(4) NOT NULL,
        from_date DATE NOT NULL,
        to_date DATE NOT NULL,
        PRIMARY KEY (dept_no,emp_no),
        KEY ix_fromdate (from_date)
      ) ENGINE=InnoDB;
```

dept emp 테이블에서 프라이머리 키는 (dept_no, emp_no)이며, 세컨더리 인덱스 ix_fromdate는 from.
date 칼럼만 포함한다. 그런데 세컨더리 인덱스는 데이터 레코드를 찾아가기 위해 프라이머리 키인 dept_no와 emp_no 칼럼을 순서대로(프라이머리 키에 명시된 순서) 포함한다. 그래서 최종적으로 ix_fromdate 인덱스는 (from_date, dept_no, emp_no) 조합으로 인덱스를 생성한 것과 흡사하게 작동할 수 있게 된다.

9.3.1.5 인덱스 머지 (index_merge)

인덱스를 이용해 쿼리를 실행하는 경우, 대부분 옵티마이저는 테이블별로 하나의 인텍스만 사용하도록 실행 계획을 수립한다. 하지만 인덱스 머지 실행 계획을 사용하면 하나의 테이블

에 대해 2개 이상의 일 덱스를 이용해 쿼리를 처리한다. 쿼리에서 한 테이블에 대한 WHERE 조건이 여러 개 있더라도 하나의 인
'데스에 포함된 칼럼에 대한 조건만으로 인덱스를 검색하고 나머지 조건은 읽어온 레코드에 대해서 채 크하는 형태로만 사용되는 것이 일반적이다. 이처럼 하나의 인덱스만 사용해서 작업 범위를 충분히 줄 일 수 있는 경우라면 테이블별로 하나의 인덱스만 활용하는 것이 효율적이다. 하지만 쿼리에 사용된 각 각의 조건이 서로 다른 인덱스를 사용할 수 있고 그 조건을 만족하는 레코드 건수가 많을 것으로 예상 될 때 MySQL 서버는 인덱스 머지 실행 계획을 선택한다.
인덱스 머지 실행 계획은 다음과 같이 3개의 세부 실행 계획으로 나누어 볼 수 있다. 3가지 최적화 모두 여러 개의 인덱스를 통해 결과를 가져온다는 것은 동일하지만 각각의 결과를 어떤 방식으로 병합할지에 따라 구분된다.

- index_merge_intersection
- index_merge_sort_union
- index_merge_union

index_merge 옵티마이저 옵션은 위의 나열된 3개의 최적화 옵션을 한 번에 모두 제어할 수 있는 옵션이며, 최적의 최적화 알고리즘에 대해서는 하나씩 예제 쿼리로 살펴보겠다.

9.3.1.6 인덱스 머지 - 교집합 (index_merge_intersection)

다음 쿼리는 2개의 WHERE 조건을 가지고 있는데, employees 테이블의 first_name 칼럼과 emp_no칼럼 모 두 각각의 인덱스(ix_firstname, PRIMARY)를 가지고 있다. 즉, 2개 중에서 어떤 조건을 사용하더라도 인 덱스를 사용할 수 있다. 그에 따라 옵티마이저는 ix_firstname과 PRMARY 키를 모두 사용해서 쿼리를 처 리하기로 결정한다. 실행 계획의 Extra 칼럼에 Using intersect"라고 표시된 것은 이 쿼리가 여러 개 의 이데스를 각각 검색해서 그 결과의 교집합만 반환했다는 것을 의미한다.

9.3.1.7 인덱스 머지 - 합집합 (index_merge_union)

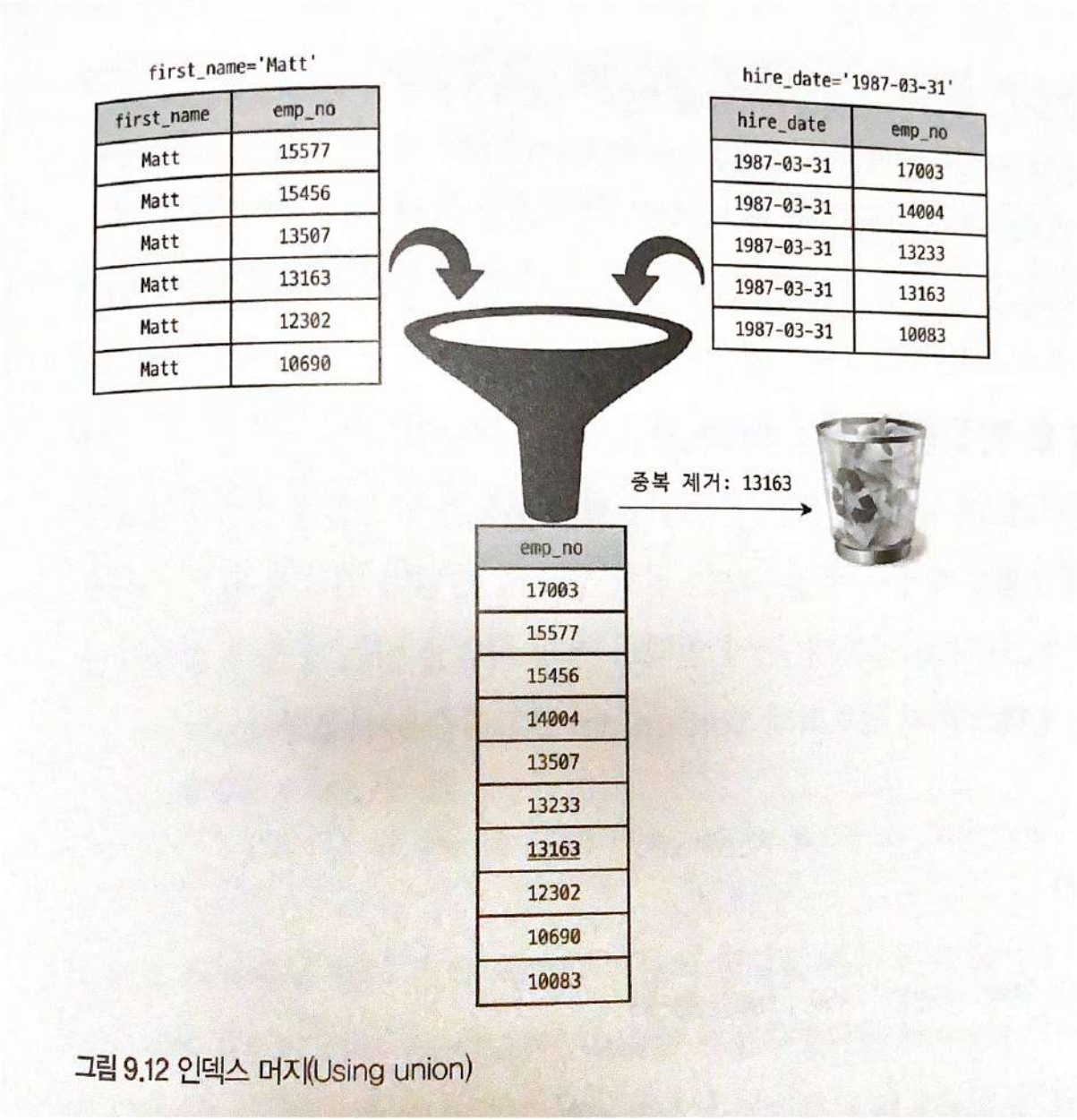


그림 9.12 인덱스 머지(Using union)

그림 9.12는 인덱스 머지 최적화의 'Union' 알고리즘의 작동 방식을 그림으로 표현한 것이다. 이 그림에서도 알 수 있듯이. first_ name 칼럼의 검색 결과와 hire_date 칼럼의 검색 결과에서 사원 번호 가"13163"인 사원은 양쪽 집함에 모두 포함돼 있어서 반드시 제거해야 한다. 하지만 MySL. 서버는 first_name 조건을 검색한 결과와 hire_date 칼럼을 검색한 결과가 프라이머리 키로 이미 각각 정렬돼 있다는 것을 알고 있다. 예제 쿼리를 다음과 같이 각각의 쿼리로 분리해서 실행 계획이나 쿼리 결과를 살펴보면 인덱스 검색을 통한 두 결과 집합이 모두 프라이머리 키로 정렬돼 있다는 것을 쉽게 확인할 수 있다.

9.3.1.8 인덱스 머지 - 정렬 후 합집합 (index_merge_sort_union)

인덱스 머지 최적화의 Union' 알고리즘은 두 결과 집합의 중복을 제거하기 위해 정렬된 결과를 필요로 하는데도 MySQL 서버는 별도의 정렬을 수행하지 않는다는 것과 별도의 정렬이 필요치 않은 이유에 대해 살펴봤다. 하지만 모든 경우가 그렇지는 않은데, 만약 인덱스 머지 작업을 하는 도중에 결과의 정 렬이 필요한 경우 MySQL 서버는 인덱스 머지 최적화의 'Sort union' 알고리즘을 사용한다.

9.3.1.9 세미 조인 (semijoin)

MySQL 서버에서 세미 조인 최적화 기능이 없었을 때는 위의 세미 조인 쿼리의 실행 계획은 다음과 같았다. 일반적으로 다른 RIDBMS에 익숙한 사용자였다면 dept.emp 테이블을 조회하는 서브쿼리 부분이 먼저 실행되고 그다음 employees 테이블에서 일치하는 레코드만 검색할 것으로 기대했을 것이다. 하지 만 MsSQL. 서버는 employees 테이블을 풀 스캔하면서 한 건 한 건 서브쿼리의 조건에 일치하는지 비교했다. 다음 실행 계획만 봐도 대략 57건만 읽으면 될 쿼리를 30만 건 넘게 읽어서 처리된다는 것을 알 수 있다.

9.3.1.10 테이블 풀-아웃 (Table Pull-out)

Table pullout 최적화는 세미 조인의 서브쿼리에 사용된 테이블을 아우터 쿼리로 끄집어낸 후에 쿼리를 조인 쿼리로 재작성하는 형태의 최적화다. 이는 서브쿼리 최적화가 도입되기 이전에 수동으로 쿼리를 튜닝하던 대표적인 방법이었다. 다음 예제 쿼리는 부서 번호가 4009'인 부서에 소속된 모든 사원을 조회하는 쿼리다. 아마도 IN(subquery) 형태의 세미 조인이 가장 빈번하게 사용되는 형태의 쿼리일 것 이다.

9.3.1.11 퍼스트 매치 (firstmatch)

First Match 최적화 전략은 IN(subquery) 형태의 세미 조인을 EXISTS(subquery) 형태로 튜닝한 것과 비슷한 방법으로 실행된다. 다음 예제 쿼리는 이름이 Matt'인 사원 중에서 1995년 1월 1일부터 30일 사이에 직급이 변경된 적이 있는 사원을 조회하는 용도의 쿼리다.

9.3.1.12 루스 스캔 (loosescan)

세미 조인 서브쿼리 최적화의 LooseScan은 인덱스를 사용하는 GROUP BY 최적화 방법에서 살펴본
”Using index for group-by”의 루스 인덱스 스캔(Loose Index Scan)과 비슷한 읽기 방식을 사용한다. 다음 쿼리는 dept_emp 테이블에 존재하는 모든 부서 번호에 대해 부서 정보를 읽어 오기 위한 쿼리다.

9.3.1.13 구체화 (Materialization)

Materialization 최적화는 세미 조인에 사용된 서브쿼리를 통째로 구체화해서 쿼리를 최적화한다는 의미다. 여기서 구체화(Materialization)는 쉽게 표현하면 내부 임시 테이블을 생성한다는 것을 의미한다. 다음의 1995년 1월 1일 조직이 변경된 사원들의 목록을 조회하는 쿼리는 IN (subquery) 포맷의 세미 조인을 사용하는 예제다.

9.3.1.14 중복 제거 (Duplicated Weed-out)

Duplicate Weedout은 세미 조인 서브쿼리를 일반적인 INNER JOIN 쿼리로 바꿔서 실행하고 마지막에 중복된 레코드를 제거하는 방법으로 처리되는 최적화 알고리즘이다. 다음 예제 쿼리는 급여가 150000 이상인 사원들의 정보를 조회하는 쿼리다.

9.3.1.15 컨디션 팬아웃 (condition_fanout_filter)

조인을 상상할 때 테이블의 순서는 쿼리의 성능에 매우 큰 영향을 미친다. 예를 들어, A 테이블과 B테이블을 조인할 때 A테이블에는 조건에 일치하는 레코드가 1만 건이고 B 테이블에는 일치하는 레코드 전수가 10건이라고 가정해보자. 아떼 A 테이블을 조인의 드라이빙 테이블로 결정하면 B 테이블을 1만 번 읽어야 한다. 이때 B 테이블의 인덱스를 이용해 조인을 실행한다고 하더라도 레코드를 읽을 때마다 B 테이블의 인덱스를 구성하는 B-Tree의 루트 노드부터 검색을 실행해야 한다. 그래서 MySQL 옵티 마이저는 여러 테이블이 조인되는 경우 가능하다면 일치하는 레코드 건수가 적은 순서대로 조인을 실행한다.

9.3.1.16 파생 테이블 머지 (derived_merge)

쿼리의 실행 계획을 보면 employees 테이블을 읽는 라인의 select_type 칼럼의 값이 DERIVED라고 표시돼 있다. 이는 emplyees 테이블에서 first_name 칼럼의 값이 Matt'인 레코드들만 읽어서 임시 테이블을 생성하고, 이 임시 테이블을 다시 읽어서 hire._date 칼럼의 값이 ‘1986-04-03’인 레코드만 걸러내어 반환한 것이다. 그래서 MySQL 서버에서는 이렇게 FROM 절에 사용된 서브쿼리를 파생 테이블 Derived Table)이라고 부른다.

9.3.1.17 인비저블 인덱스 (use_invisible_indexes)

NySQL 8.0 버전부터는 인덱스의 가용 상태를 제어할 수 있는 기능이 추가됐다. MySQL 8.0 이전 버전까지는 인덱스가 존재하면 항상 옵티마이저가 실행 계획을 수립할 때 해당 인덱스를 검토하고 사용 했다. 하지만 MySQL 8.0 버전부터는 인덱스를 삭제하지 않고, 해당

인덱스를 사용하지 못하게 제어하는 기능을 제공한다. 다음 예제와 같이 ALTER TABLE ...ALTER INDEX ... [VISIBLE I INVISIBLE] 명령 으로 인덱스의 가용 상태를 변경할 수 있다.

9.3.1.18 스킵 스캔 (skip_scan)

인덱스의 핵심은 값이 정렬돼 있다는 것이며, 이로 인해 인덱스를 구성하는 칼럼의 순서가 매우 중요하다. 예를 들어, (A, B, C) 칼럼으로 구성된 인덱스가 있을 때 쿼리의 WHERE 절에 A와 B 칼럼에 대한 조건이 있다면 이 쿼리는 A 칼럼과 B 칼럼까지만 인덱스를 활용할 수 있고, WHERE 절에 A 칼럼에 대한 조건만 가지고 있다면 A 칼럼까지만 인덱스를 활용할 수 있다. 그런데 WHERE 절에 B와 C 칼럼에 대한 조건을 가지고 있다면 이 쿼리는 인덱스를 활용할 수 없다. 인덱스 스킵 스캔은 제한적이긴 하지만 인덱스의 이런 제약 사항을 뛰어넘을 수 있는 최적화 기법이다.

9.3.1.19 해시 조인 (hash_join)

MySQL 8.0.18 버전부터는 해시 조인이 추가로 지원되기 시작했는데, MySQL 서버에 해시 조인이 없어서 아쉬워했던 많은 사용자에게는 환영할 만한 부분인 것이다. 해시 조인의 진실에 대해 먼저 살펴보고, 그 다음으로 MySQL 서버의 해서 조인이 내부적으로 어떻게 처리되는지 살펴보자.
많은 사용자가 해시 조인 기능을 기대하는 이유가 기존의 네스티드 루프 조인(Nested Loop Join)보다 해시 조인이 빠르다고 생각하기 때문이다. 하지만 이는 항상 옳은 이야기는 아니다. 다음 그림 9,16은 네스티드 루프 조인과 해시 조인의 처리 성능을 비교해 보여주는 것으로, 화살표의 길이는 전제 쿼리의 실행 시간을 의미한다.

9.3.1.20 인덱스 정렬 선호 (prefer_ordering_index)

MySQL 옵티마이저는 ORDER BY 또는 GROUP BY 를 인덱스를 사용해 처리 가능한 경우 쿼리의 실행 계획에서 이 인덱스의 가중치를 높이 설정해서 실행된다.

9.3.2 조인 최적화 알고리즘

MySQL에는 조인 쿼리의 실행 계획 최적화를 위한 알고리즘이 2개 있다. 사실 이 알고리즘들은 MySQL 5.0 버전부터 있던 기능인데. 그 중요성에 비해 모르는 경우가 많아서 함께 설명하고자 한다.
MySQL의 조인 최적화는 나름 많이 개선됐다고 이야기한다. 하지만 사실 테이블의 개수가

많아지면 최적화된 실행 계획을 찾는 것이 상당히 어려워지고, 하나의 쿼리에서 조인되는 테이블의 개수가 많아 지면 실행 계획을 수립하는 데만 몇 분이 걸릴 수도 있다. 테이블의 개수가 특정 한계를 넘어서면 그때 부터는 실행 계획 수립에 소요되는 시간만 몇 시간이나 며칠로 늘어날 수도 있다. 여기서는 왜 그런 현 상이 생기고, 어떻게 그런 현상을 피할 수 있는지 살펴보겠다.
MySQL에는 최적화된 조인 실행 계획 수립을 위한 2가지 알고리즘이 있는데, 적절한 한글 명칭이 없어서 영어를 그대로 표기하겠다. 다음과 같이 간단히 4개의 테이블을 조인하는 쿼리 문장이 조인 옵티 마이저 알고리즘에 따라 어떻게 처리되는지 간단히 살펴보자.

9.3.2.1 Exhaustive 검색 알고리즘

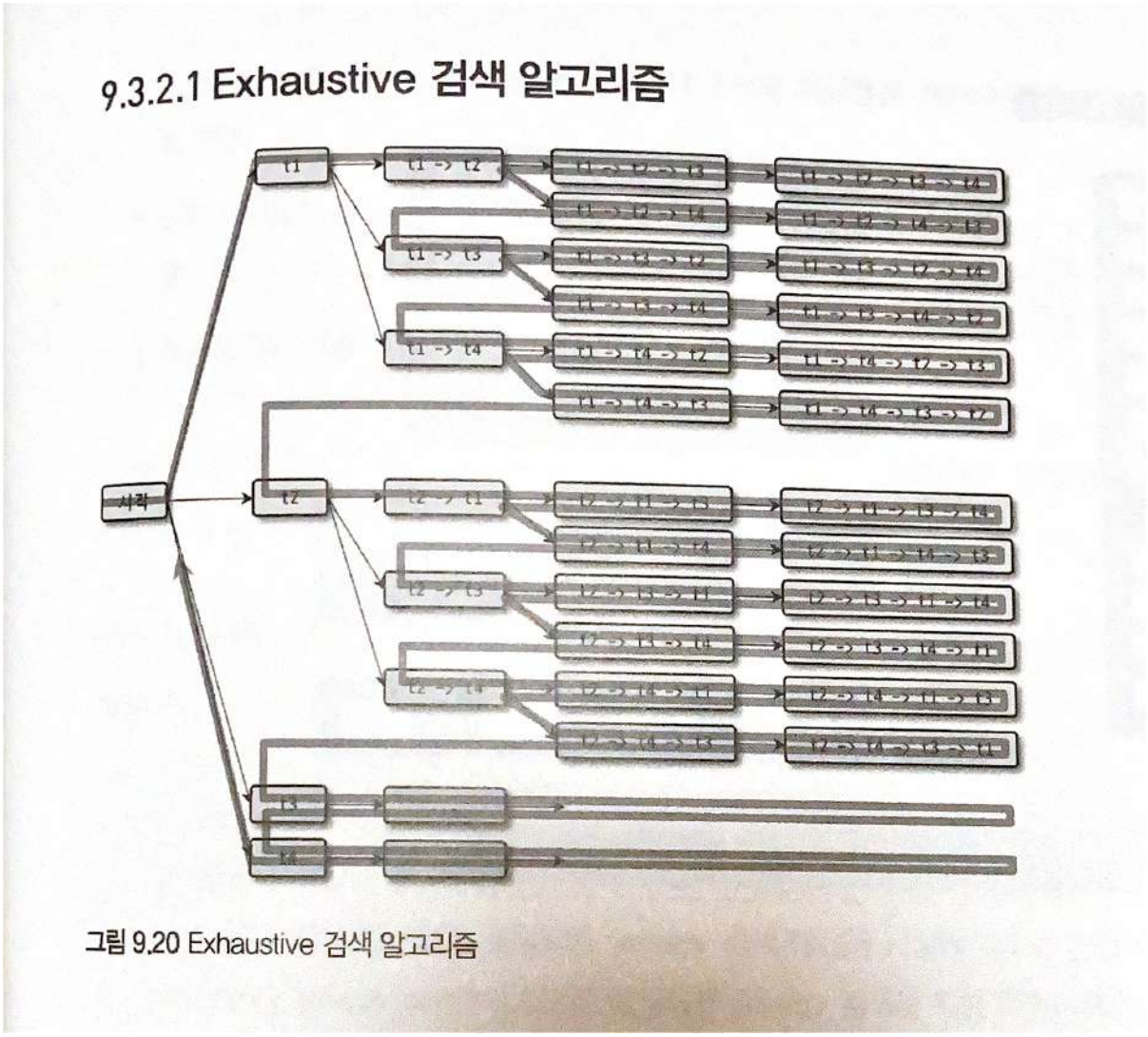


그림 9.20 Exhaustive 검색 알고리즘

Exhaustive 검색 알고리즘은 MySQL 5.0과 그 이전 버전에서 사용되던 조인 최적화 기법으로, FRON 걸이 명시된 모든 테이블의 조함에 대해 실행 계획의 비용을 계산해서 최적의 조합 1개를 찾는 방법이 다. 그림 9,20은 4개의 테이블(11-4)이 Exhaustive 검색 알고리즘으로 처리될 때 최적의 조인 순서를 찾는 방법을 표현한 것이다. 테이블이 20개라면 이 방법으로 처리했을 때 가능한 조인 조합은 모두 20! (Factorial, 3628800)개가 된다. 이건 버전에서 사용되던 Exhaustive 검색 알고리즘에서는 사실 테이블이 10개만 넘어도 실행 계획을 수립하는 데 몇 분이 걸린다. 그리고 테이블이 10개에서 1개만 더 늘어나도 11배의 시간이 더 걸린다.

9.3.2.2 Greedy 검색 알고리즘

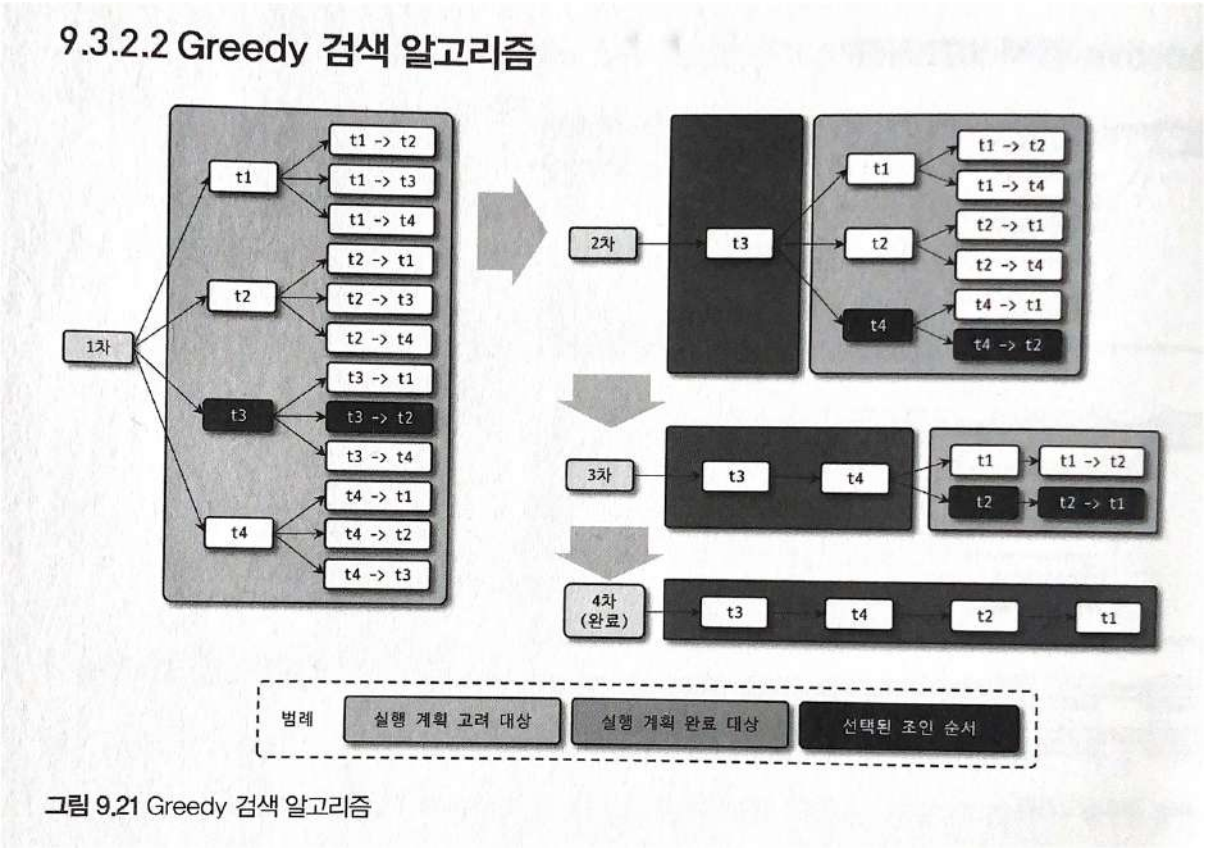


그림 9.21 Greedy 검색 알고리즘

Greedy 검색 알고리즘은 Exhaustive 검색 알고리즘의 시간 소모적인 문제점을 해결하기 위해
MySQL 5.0부터 도입된 조인 최적화 기법이다. 그림 9.21은 4개의 테이블(11r14)이 Greedy 검색 알 고리즘으로 처리될 때(optimizer_ search_depth 시스템 변수의 값은 2.로 가정) 최적의 조인 순서를 검색 하는 방법을 보여준다. Greedy는 Exhaustive 검색 알고리

즘보다는 조금 복잡한 형태로 최적의 조인 순서를 결정한다. 그림 9.21의 내용을 간단히 순서대로 살펴보자.

1. 전체 N개의 테이블 중에서 optimizer_ search_depth 시스템 설정 변수에 정의된 개수의 테이블로 가능한 조인 조
   합을 생성
2. 1번에서 생성된 조인 조합 중에서 최소 비용의 실행 계획 하나를 선정
3. 2번에서 선정된 실행 계획의 첫 번째 테이블을 "부분 실행 계획(그림 9.20에서는 실행 계획 완료 대상으로 표현됨
   의 첫 번째 테이블로 선정
4. 전체 N-1개의 테이블 중(3번에서 선택된 테이블 제외)에서 optimizer_search_depth 시스템 설정 변수에 정의된 개수의 테이블로 가능한 조인 조합을 생성
5. 4번에서 생성된 조인 조합들을 하나씩 3번에서 생성된 "부분 실행 계획'에 대입해 실행 비용을 계산
6. 5번의 비용 계산 결과, 최적의 실행 계획에서 두 번째 테이블을 3번에서 생성된 '부분 실행 계획'의 두 번째 테이블로 선정
7. 남은 테이블이 모두 없어질 때까지 4~6번까지의 과정을 반복 실행하면서 부분 실형 계획'에 테이블의 조인 순서를 기록
8. 최종적으로 '부분 실행 계획'이 테이블의 조인 순서로 결정됨

9.4 쿼리 힌트

MySOL의 버전이 업그레이드되고 통계 정보나 옵티마이저의 최적화 방법들이 더 다양해지면서 쿼리 의 실행 계획 최적화가 많이 성숙하고 있다. 하지만 여전히 MySOL 서버는 우리가 서비스하는 비즈니 스를 100% 이해하지는 못한다. 그래서 서비스 개발자나 DIBA보다 MysQL 서버가 부족한 실행 계획을 수립할 때가 있을 수 있다. 이런 경우에는 옵티마이저에게 쿼리의 실행 계획을 어떻게 수립해야 할지 알려줄 수 있는 방법이 필요하다. 일반적인 RIDBMS에서는 이런 목적으로 힌트가 제공되며, MySQL에 서도 다양한 옵티마이저 힌트를 제공한다.

MySQL 서버에서 사용 가능한 쿼리 힌트는 다음과 같이 2가지로 구분할 수 있다.
• 인덱스 힌트
• 옵티마이저 힌트

인덱스 힌트는 예전 버전의 MySOL. 서버에서 사용되어 오던 "USE INDEX" 같은 힌트를 의미하며, 옵티 마이저 힌트는 MySOL 5.6 버전부터 새롭게 추가되기 시작한 힌트들을 지칭한다. 그런데 여기에 포함 되지 않은 STRAIGHT_JOIN과 같은 힌트들도 있다. 여기서는 옵티마이저 힌트가 아닌 것들은 모두 모아서 인덱스 힌트 절로 분류해서 살펴보겠다.

9.4.1 인덱스 힌트

"STRAIGHT_JOIN”과 "USE INDEX" 등을 포함한 인덱스 힌트들은 모두 MySOL 서버에 옵티마이저 현트가 도입되기 전에 사용되던 기능들이다. 이들은 모두 SQL의 문법에 맞게 사용해야 하기 때문에 사용하게 되면 ANSI-SOL 표준 문법을 준수하지 못하게 되는 단점이 있다". MySQL 5.6 버전부터 추가되기 시 작한 옵티마이저 힌트들은 모두 MySQL 서버를 제외한 다른 RDBMS에서는 주석으로 해석하기 때문 에 ANSI-SOL 표준을 준수한다고 볼 수 있다. 그래서 가능하다면 인덱스 힌트보다는 옵티마이저 현트 를 사용할 것을 추천한다. 또한 인덱스 힌트는 SELECT 명령과 UPDATE 명령에서만 사용할 수 있다.

9.4.1.1 STRAIGHT_JOIN

STRAIGHE JOIN은 옵티마이지 힌트인 동시에(11.6.2절 'JOIN UPDATE의 예제와 같이) 조인 키워드이기도 하다. STRAIGHT_JOIN은 SELECT, UPDATE, DELETE 쿼리에서 여러 개의 테이블이 조인되는 경우 조인 순서를 고정하는 역할을 한다. 다음 쿼리는 3개의 테이블을 조인하지만 어느 테이블이 드라이빙 테이블이 되고 어느 테이블이 드리븐 테이블이 될지 알 수 없다. 옵티마이저가 그때그때 각 테이블의 통계 정보와 취리의 조건을 기반으로 가장 최적이라고 판단되는 순서로 조인한다.

9.4.1.2 USE INDEX / FORCE INDEX / IGNORE INDEX

조인의 순서를 변경하는 것 다음으로 자주 사용되는 것이 인덱스 힌트인데, STRAIGHT JOEN 힌트와는 달 리 인텍스 힌트는 사용하려는 인덱스를 가지는 테이블 뒤에 힌트를 명시해야 한다. 대체로 MySQL 옵티마이저는 어떤 인덱스를 사용해야 할지를 무난하게 잘 선택하는 편이다. 하지만 3~4개 이상의 칼럼 을 포함하는 비슷한 인덱스가 여러 개 존재하는 경우에는 가끔 옵티마이저가 실수를 하는데, 이런 경우 에는 강제로 특정 인덱스를 사용하도록 힌트를 추가한다.

9.4.1.3 SQL_CALC_FOUND_ROWS

MySQL의 LIMIT을 사용하는 경우, 조건을 만족하는 레코드가 LIMIT에 명시된 수보다 더 많다고 하더라 도 LIMIT에 명시된 수만큼 만족하는 레코드를 찾으면 즉시 검색 작업을 멈춘다. 하지만 SQL_CALC_ FOUND.
ROWS 힌트가 포함된 쿼리의 경우에는 LIMIT을 만족하는 수만큼의 레코드를 찾았다고 하더라도 끝까지 점색을 수행한다. 최종적으로 사용자에게는 LIMIT에 제한된 수만큼의 결과 레코드만 반환됨에도 불구 하고 말이다. SQL_ CALC_ FOUND._ ROWS 힌트가 사용된 쿼리가 실행된 경우에는 FOUND_ROWS() 라는 함수를 이 용해 LIMIT을 제외한 조건을 만족하는 레코드가 전체 몇 건이었는지를 알아낼 수 있다.

9.4.2 옵티마이저 힌트

MySQL 8.0 버전에서 사용 가능한 힌트는 종류가 매우 다양하며, 옵티마이저 힌트가 미치는 영향 범위도 매우 다양하다. 우선 옵티마이저 힌트들을 영향 범위별로 구분해서 살펴보고, 그중에서 자주 사용되 는 것들 위주로 예제 쿼리와 함께 살펴보자.

9.4.2.1 옵티마이저 힌트 종류

옵티마이저 힌트는 영향 범위에 따라 다음 4개 그룹으로 나누어 볼 수 있다.
• 인덱스: 특정 인덱스의 이름을 사용할 수 있는 옵티마이저 힌트
• 테이블: 특정 테이블의 이름을 사용할 수 있는 옵티마이저 힌트
• 쿼리 블록: 특정 쿼리 블록에 사용할 수 있는 옵티마이저 힌트로서, 특정 쿼리 블록의 이름을 명시하는 것이 아니라 힌트가 명시된 쿼리 블록에 대해서만 영향을 미치는 옵티마이저 힌트
• 글로벌(쿼리 전체): 전체 쿼리에 대해서 영향을 미치는 힌트

9.4.2.2 MAX_EXECUTION_TIME

옵티마이저 힌트 중에서 유일하게 쿼리의 실행 계획에 영향을 미치지 않는 힌트이며, 단순히 쿼리의 최 대 실행 시간을 설정하는 힌트다. MAX EXECUTION_ TIME 힌트에는 밀리초 단위의 시간을 설정하는데, 쿼리가 지정된 시간을 초과하면 다음과 같이 쿼리는 실패하게 된다.

9.4.2.3 SET_VAR

옵티마이저 힌트뿐만 아니라 MySQL 서버의 시스템 변수들 또한 쿼리의 실행 계획에 상당한 영향을 미친다. 대표적으로 조인 버퍼의 크기를 설정하는 join_buffer_size 시스템 변수의 경우 쿼리에 아무런 영향을 미치지 않을 것처럼 보인다. 하지만 MySQL 서버의 옵티마이

저는 조인 버퍼의 공간이 충분하면 조인 버퍼를 활용하는 형태의 실행 계획을 선택할 수도 있다. 그뿐만 아니라 옵티마이저 힌트로 부 족한 경우 optimizer_switch 시스템 변수를 제어해야 할 수도 있다. 이런 경우에는 다음과 같이 SET_VAR 힌트를 이용하면 된다.

9.4.2.4 SEMIJOIN & NO_SEMJIJOIN

세미 조인의 최적화는 여러 가지 세부 전략이 있다는 것을 이미 살펴봤다. 9.3.1.9절 '세미 조인 (semijoin)'을 참조하자. SEMIJOIN 힌트는 어떤 세부 전략을 사용할지를 제어하는데 사용할 수 있다.

9.4.2.5 SUBQUERY

서브쿼리 최적화는 세미 조인 최적화가 사용되지 못할 때 사용하는 최적화 방법으로, 서브쿼리는 다음 2가지 형태로 최적화 할 수 있다.

최적화 방법	힌트
IN-to-EXISTS	SUBQUERY(INTOEXISTS)
Materialization	SUBQUERY(MATERIALIZATION)

9.4.2.6 BNL & NO_BNL & HASHJOIN & NO_HASHJOIN

MySQL 8.0.19 버전까지는 블록 네스티드 루프(Block Nested L.00p) 조인 알고리즘을 사용했지만 MrSOL 8.0.18 버전부터 도입된 해시 조인 알고리즘이 MySQL 8.0.20 버전부터는 블록 네스티드 루프 조인까지 대체하도록 개선됐다. 그래서 MySQL 8.0.20 버전부터는 블록 네스티드 루프 조인은 MySQL 서버에서 더 이상 사용되지 않는다. 하지만 BNL 힌트와 NO_ BNL 힌트는 MySQL 3.0.20과 그 이 후의 버전에서도 여전히 사용 가능한데, MySQL 3.0.20 버전과 그 이후 버전에서는 BNL 힌트를 사용 하면 해시 조인을 사용하도록 유도하는 힌트로 용도가 변경됐다. 대신 HASHOOTN과 NO_HASHOIN 힌트는 MySQL 8.0.18 버전에서만 유효하며, 그 이후 버전에서는 효력이 없다. 그래서 MySQL 8.0.20과 그 이후 버전에서는 해시 조인을 유도하거나 해시 조인을 사용하지 않게 하고자 한다면 다음 예제 쿼리와 같이 BNL과 NO_BNL 힌트를 사용해야 한다.

9.4.2.7 JOIN_FIXED_ORDER & JOIN_ORDER & JOIN_PREFIX & JOIN_SUFFIX

MySQL 서버에서는 조인의 순서를 결정하기 위해 전통적으로 STRAIGHT_JOIN 힌트를 사용해왔다. 하지 만 STRAIGHT_ JOIN 힌트는 우선 쿼리의 FROM 절에 사용된 테이블의 순서를 조인 순서에 맞게 변경해야 하 는 번거로움이 있었다. 또한 STRAIGHT JOIN은 한 번 사용되면 FROM 절에 명시된 모든 테이블의 조인 순 서가 결정되기 때문에 일부는 조인

순서를 강제하고 나머지는 옵티마이저에게 순서를 결정하게 맞기는 것이 불가능했다. 이 같은 단점을 보완하기 위해 옵티마이저 힌트에서는 STRATGHT JOIN과 동일한 힌트 까지 포함해서 다음과 같이 4개의 힌트를 제공한다.JOIN_ FIXED_ORDER: STRATGHT_JOIN 힌트와 동일하게 FROM 절의 테이블 순서대로 조인을 실행하게 하는 힌트를 제공한다.

- JOIN_ FIXED_ORDER: STRATGHT_JOIN 힌트와 동일하게 FROM 절의 테이블 순서대로 조인을 실행하게 하는 힌트
- JOIN_ ORDER: FROW 절에 사용된 테이블의 순서가 아니라 힌트에 명시된 테이블의 순서대로 조인을 실행하는 힌트
- JOIN_PREFIX: 조인에서 드라이빙 테이블만 강제하는 힌트
- JOIN_SUFFIX: 조인에서 드리븐 테이블(가장 마지막에 조인돼야 할 테이블들)만 강제하는 힌트
-

9.4.2.8 MERGE & NO_MERGE

예전 버전의 MySQL 서버에서는 FROM 절에 사용된 서브쿼리를 항상 내부 임시 테이블로 생성했다. 이렇게 생성된 내부 임시 테이블을 파생 테이블(Derived table)이라고 하는데, 이는 불필요한 자원 소모 를 유발한다. 그래서 MySQL 5.7과 8.0 버전에서는 가능하면 임시 테이블을 사용하지 않게 FROM 절의 서브쿼리를 외부 쿼리와 병합하는 최적화를 도입했다. 때로는 MySQL 옵티마이저가 내부 쿼리를 외부 쿼리와 병합하는 것이 나을 수도 있고, 때로는 내부 임시 테이블을 생성하는 것이 더 나은 선택일 수 도 있다. 하지만 MysQL 옵티마이저는 최적의 방법을 선택하지 못할 수도 있는데, 이때는 다음과 같이 KERGE 또는 NO_MERGE 옵티마이저 힌트를 사용하면 된다.

9.4.2.9 INDEX_MERGE & NO_INDEX_MERGE

MySQL 서버는 가능하다면 테이블당 하나의 인덱스만을 이용해 쿼리를 처리하려고 한다. 하지만 하나 의 인덱스만으로 검색 대상 범위를 충분히 좁힐 수 없다면 MySQL 옵티마이저는 사용 가능한 다른 인 덱스를 이용하기도 한다. 여러 인덱스를 통해 검색된 레코드로부터 교집합 또는 합집합만을 구해서 그 결과를 반환한다. 이처럼 하나의 테이블에 대해 여러 개의 인덱스를 동시에 사용하는 것을 인덱스 머지 (Index Merge)라고 한다. 인덱스 머지 실행 계획은 때로는 성능 향상에 도움이 되지만 항상 그렇지는 않을 수도 있다. 인덱스 머지 실행 계획의 사용 여부를 제어하고자 할 때, 다음 예제와 같이 INDEX VERE 와 NO_INDEX MERGE 옵티마이저 힌트를 이용하면 된다.

9.4.2.10 NO_ICP

인덱스 컨디션 푸시다운(ICP, Index Condition Pushdown) 최적화는 사용 가능하다면 항상 성능 향 상이 도움이 되므로 MySQL 옵티마이저는 최대한 인덱스 컨디션 푸시다운 기능을 사용하는 방향으로 실행 계획을 수립한다. 그래서 MySQL 옵티마이저에서는 ICP 힌트(인덱스 컨디션 푸시다운을 사용하도록 하는 힌트)는 제공되지 않는다. 그런데 인덱스 컨디션 푸시다운으로 인해 여러 실행 계획의 비용 계산이 잘못된다면 결과적으로 잘못된 실행 계획을 수립하게 될 수도 있다.

9.4.2.11 SKIP_SCAN & NO_SKIP_SCAN

인덱스 스킵 스캔은 인덱스의 선행 칼럼에 대한 조건이 없어도 옵티마이저가 해당 인덱스를 사용할 수 있게 해주는 매우 훌륭한 최적화 기능이다. 하지만 조건이 누락된 선행 칼럼이 가지는 유니크한 값의 개수가 많아진다면 인덱스 스킵 스캔의 성능은 오히려 더 떨어진다. MysQL 옵티마이저가 유니크한 값의 개수를 제대로 분석하지 못하거나 잘못된 경로로 인해 비효율적인 인덱스 스킵 스캔을 선택하면 다음 예제와 같이 NO_SKIP_SCAN 옵티마이저 힌트를 이용해 인덱스 스킵 스캔을 사용하지 않게 할 수 있다.

9.4.2.12 INDEX & NO_INDEX

INDEX와 NO_INDEX 옵티마이저 힌트는 예전 MySQL 서버에서 사용되던 인덱스 힌트를 대체하는 용도로 제공된다. 인덱스 힌트를 대체하는 옵티마이저 힌트는 다음과 같다.

인덱스 힌트	옵티마이저 힌트
USE INDEX	INDEX
USE INDEX FOR GROUP BY	GROUP_INDEX
USE INDEX FOR ORDER BY	ORDER_INDEX
IGNORE INDEX	NO_INDEX
IGNORE INDEX FOR GROUP BY	NO_GROUP_INDEX
IGNORE INDEX FOR ORDER BY	NO_ORDER_INDEX