8.2.1 SELECT 문 최적화

SELECT 문 형태의 쿼리는 데이터베이스에 대한 모든 조회 작업을 수행합니다. 동적 웹 페이지의 응답 시간을 1초 미만으로 단축하거나, 밤새 대량의 보고서를 생성하는데 소요되는 시간을 몇 시간 단축하기 위해서도 SELECT 문을 튜닝하는 것이 최우선 과제입니다.

SELECT 문 외에도, CREATE TABLE...AS SELECT, INSERT INTO...SELECT, DELETE 문에서의 WHERE 절과 같은 구성 요소에도 이 쿼리 튜닝 방법이 적용됩니다. 이러한 문은 쓰기 작업과 읽기 지향 쿼리 작업을 결합하기 때문에 성능에 대한 추가적인 고려 사항이 있습니다.

NDB 클러스터는 조인 푸시다운 최적화를 지원하므로, 적격한 조인이 NDB Cluster 데이터 노드로 완전히 전송되며, 이 노드 간에 분산되어 병렬로 실행될 수 있습니다. 이 최적화에 대한 자세한 내용은 NDB 푸시 다운 조인 조건을 참고합니다.

쿼리 최적화를 위한 주요 고려 사항은 다음과 같습니다:

- 느린 SELECT ... WHERE 쿼리의 속도를 높이기 위해 가장 먼저 확인해야 할 것은인덱스를 추가할 수 있는지 여부입니다. WHERE 절에서 사용하는 컬럼에 인덱스를 설정하여 평가, 필터링 및 최종 결과 검색 속도를 높일 수 있습니다. 낭비되는 디스크 공간을 피하기 위해, 소수의 인덱스 세트만 구성하여, 애플리케이션에서 사용되는 많은 관련 쿼리의 속도를 높이도록 합니다.

  인덱스는 조인 및 외래 키와 같은 기능을 사용하여 다양한 테이블을 참조하는 쿼리에 특히 중요합니다EXPLAIN 문을 사용하여 SELECT에 사용할 인덱스를 결정할 수 있다.섹션 8.3.1, 'MySQL이 인덱스를 사용하는 방법'및 섹션 8.8.1, 'EXPLAIN을 이용한 쿼리 최적화'를 참고합니다.
- 과도한 시간이 걸리는 함수 호출과 같은 쿼리 부분을 식별하고 조정합니다. 쿼리를 작성하는 방법에 따라 함수가 결과 집합의 모든 행에 대해 한 번씩, 심지어 테이블의 모든 행에 대해 한 번씩 호출되는 등 비효율성이 크게 늘어나는 경우가 있습니다.
- 특히 큰 테이블의 경우, 쿼리에서풀 테이블 스캔 횟수를 최소화합니다.
- ANALYZE TABLE 명령문을 정기적으로 사용하여 테이블 통계를 최신 상태로 유지하여 옵티마이저가 효율적인 실행 계획을 세우는데 필요한 정보를 얻을 수 있도록 합니다.
- 튜닝 기법, 인덱싱 기법 및 각 테이블의 스토리지 엔진에 고유한 구성 매개변수에 대해 학습합니다. InnoDB, MyISAM 어느 쪽이든 쿼리의 높은 성능을 가능하게하고 유지하기 위한 일련의 지침이 있습니다. 자세한 내용은 섹션 8.5.6, "InnoDB 쿼리 최적화" 및 섹션 8.6.1, "MyISAM 쿼리 최적화" 를 참조하십시오.
- 섹션 8.5.3, "InnoDB 읽기 전용 트랜잭션 최적화"기술을 사용하여 InnoDB 테이블의 단일 쿼리 트랜잭션을 최적화할 수 있습니다.
- 특히 옵티마이저가 동일한 변환의 일부를 자동으로 실행하는 경우, 이해하기 어려워지는 쿼리 변환을 피하십시오.
- 기본 지침 중 무엇 하나로도 성능 문제를 쉽게 해결할 수 없는 경우EXPLAIN 플랜을 읽어 특정 쿼리의 내부 세부 정보를 조사하고, 인덱스, WHERE절, 조인 절 등을 조정합니다. (어느 정도 전문 기술에 도달했다면 EXPLAIN 플랜을 읽는 것이 모든 쿼리에 대해 첫 번째 하는 일이 될 수 있습니다.)
- MySQL이 캐쉬에 사용하는 메모리 영역의 크기와 속성을 조정합니다. InnoDB버퍼 풀, MyISAM 키 캐쉬 및 MySQL 쿼리 캐쉬를 효율적으로 사용하면, 두 번째 실행부터는 메모리로부터 결과가 취득되기 때문에, 쿼리의 반복 실행이 고속화됩니다.
- 캐시 메모리 영역을 사용하여 빠르게 실행하는 쿼리역시도, 수행에 필요한 캐시 메모리를 줄이는 등 더욱 최적화 할 수 있으며, 애플리케이션의 확장성을 높일 수 있습니다. 확장성이란 애플리케이션이 더 많은 동시 사용자, 더 많은 요청 등을 처리할 수 있다는 것을 의미하며, 이는 성능 저하 없이 더 많은 동시 사용자를 처리할 수 있다는 것을 의미합니다.
- 테이블에 동시에 액세스하는 다른 세션에 의해 쿼리 속도가 영향을 받을 수 있는 잠금 문제를 처리합니다.

8.2.1.1 WHERE 구문 최적화

이 절에서는 WHERE절을 처리할 때 수행할 수 있는 최적화에 대해 설명합니다. 예제에서는SELECT 문을 사용하지만DELETE 문 UPDATE 문의 WHERE절에도 동일한 최적화를 적용합니다.

> **참고 사항**
> MySQL 옵티마이저에 대한 작업은 계속 진행 되고 있는 중이므로, 여기서 MySQL이 수행하는 모든 최적화를 설명하지는 않습니다.

가독성을 희생하더라도 산술 연산 속도를 높이기 위해 쿼리를 다시 작성하고 싶을 때가 있습니다. MySQL에서는 이러한 최적화를 자동적으로 실행하기 때문에, 많은 경우에 이러한 작업을 회피할 수 있으며, 쿼리를 이해하기 쉽고 유지보수하기 쉬운 형태 그대로 남겨둘 수 있습니다. 다음은 MySQL에 의해 수행되는 최적화 작업 중 일부 입니다:

- 불필요한 괄호 제거:

```
   ((a AND b) AND c OR (((a AND b) AND (c AND d))))
-> (a AND b AND c) OR (a AND b AND c AND d)
```

- 상수 폴딩:

```
   (a<b AND b=c) AND a=5
-> b>5 AND b=c AND a=5
```

- 상수 조건 제거:

```
   (b>=5 AND b=5) OR (b=6 AND 5=5) OR (b=7 AND 5=6)
-> b=5 OR b=6
```

  MySQL 8.0.14 이상에서는, 이 작업이 최적화 단계가 아닌 준비 단계에 수행되기 때문에, 조인의 간략화에 도움이 됩니다. 자세한

내용과 예시는 섹션 8.2.1.9, "아우터 조인 최적화"를 참조하십시오.

- 인덱스에서 사용되는 상수식은 1회만 평가됩니다.
- MySQL 8.0.16 이상에서는, 상수 값을 가진 숫자형 컬럼의 비교가 유효하지 않거나 범위를 벗어난 값이 없는지 확인하여 폴딩되거나 제거됩니다:

```
# CREATE TABLE t (c TINYINT UNSIGNED NOT NULL);
  SELECT * FROM t WHERE c ≪ 256;
->≫ SELECT * FROM t WHERE 1;
```

상세한 정보는 섹션 8.2.1.14, "상수-폴딩 최적화"를 참고합니다.

- WHERE를 사용하지 않는 단일 테이블의COUNT(*)는 MyISAM 테이블과 MEMORY 테이블의 테이블 정보에서 직접 가져 옵니다. NOT NULL식 또한 하나의 테이블에서만 사용되는 경우라면 이와 같이 수행됩니다.
- 유효하지 않은 상수식 조기 감지. MySQL은 일부SELECT 문이 실행될 수 없는 다거나, 아무 결과도 반환 되지 않을 것이란 점을 신속하게 감지합니다.
- GROUP BY 또는 집계 함수(COUNT(), MIN() 등)가 사용하지 않는 경우, HAVING은 WHERE와 병합됩니다.
- 조인된 각 테이블에 대해 더 간단한 WHERE가 구성되어, 더 빨리 테이블의 WHERE 조건 평가를 얻고, 가능한 한 빨리 행을 건너뛰게끔 합니다.
- 쿼리 내의 다른 모든 테이블보다 앞서, 상수 테이블들을 먼저 읽습니다. 상수 테이블은 다음 중 하나입니다:
  - 빈 테이블 또는 1행만 있는 테이블.
  - PRIMARY KEY 또는 UNIQUE 인덱스의 WHERE 절에서 사용되는 테이블. 여기서는 모든 인덱스 부분이 상수 표현식과 비교되어 NOT NULL로 정의됩니다.

다음 테이블들은 모두 상수 테이블로 사용됩니다:

```
SELECT * FROM t WHERE primary_key=1;
SELECT * FROM t1,t2
  WHERE t1.primary_key=1 AND t2.primary_key=t1.id;
```

- 테이블을 조인하기 위한 최적의 조인 조합은 모든 가능성을 시도해 볼 수 있습니다. ORDER BY와 GROUP BY절의 모든 컬럼이 같은 테이블에 있다면, 조인 할 때 해당 테이블이 먼저 선택됩니다.
- ORDER BY절과 다른 GROUP BY절이 있거나, ORDER BY와 GROUP BY에 조인 큐의 첫 번째 테이블이과 다른 테이블의 컬럼이 포함되어 있으면 임시 테이블이 만들어집니다.
- SQL_SMALL_RESULT 한정자를 사용하는 경우 MySQL은 메모리 내 임시 테이블을 사용합니다.
- 옵티마이저가 테이블 스캔을 사용하는 것이 더 효율적이라고 판단하지 않는 한, 각 테이블 인덱스가 쿼리되고 최적의 인덱스가 사용됩니다. 한 때, 최적의 인덱스가 테이블의 30% 이상에 걸쳐 있는지 여부에 따라 스캔을 했지만, 더이상 고정된 비율에 따라 인덱스를 사용할지 풀테이블 스캔을 할지 결정하지 않습니다. 현재의 옵티마이저는 더욱 복잡해지고 테이블 크기, 행 수, I/O 블록 크기와 같은 추가 요소를 기반으로 추정합니다.
- 경우에 따라 MySQL은 데이터 파일을 참조하지 않고도 인덱스에서 행을 읽을 수 있습니다. 인덱스에서 사용되는 모든 컬럼이 숫자라면, 쿼리를 해결하기 위해 인덱스 트리만을 사용합니다.
- 각 행이 출력되기 전에, HAVING 절에 일치하지 않는 것은 스킵됩니다.

매우 빠른 쿼리의 몇 가지 예시:

```
SELECT COUNT(*) FROM tbl_name;

SELECT MIN(key_part1),MAX(key_part1) FROM tbl_name;

SELECT MAX(key_part2) FROM tbl_name
  WHERE key_part1=constant;

SELECT ... FROM tbl_name
  ORDER BY key_part1,key_part2,... LIMIT 10;

SELECT ... FROM tbl_name
  ORDER BY key_part1 DESC, key_part2 DESC, ... LIMIT 10;
```

MySQL은, 인덱스 설정된 컬럼이 숫자라고 가정함으로써, 다음의 쿼리들을 인덱스 트리만을 사용해 해결합니다:

```
SELECT key_part1,key_part2 FROM tbl_name WHERE key_part1=val;

SELECT COUNT(*) FROM tbl_name
  WHERE key_part1=val1 AND key_part2=val2;

SELECT MAX(key_part2) FROM tbl_name GROUP BY key_part1;
```

다음 쿼리는 개별로 정렬 단계를 거치지 않고, 인덱스를 사용하여 정렬 순서대로 행을 검색합니다:

```
SELECT ... FROM tbl_name
  ORDER BY key_part1,key_part2,... ;
```

```
SELECT ... FROM tbl_name
  ORDER BY key_part1 DESC, key_part2 DESC, ... ;
```

8.2.1.2 범위 최적화

The `range` access method uses a single index to retrieve a subset of table rows that are contained within one or several index value intervals. It can be used for a single-part or multiple-part index. The following sections describe conditions under which the optimizer uses range access.

- Range Access Method for Single-Part Indexes
- Range Access Method for Multiple-Part Indexes
- Equality Range Optimization of Many-Valued Comparisons
- Skip Scan Range Access Method
- Range Optimization of Row Constructor Expressions
- Limiting Memory Use for Range Optimization

Range Access Method for Single-Part Indexes

For a single-part index, index value intervals can be conveniently represented by corresponding conditions in the `WHERE` clause, denoted as range conditions rather than “intervals.”

The definition of a range condition for a single-part index is as follows:

- For both `BTREE` and `HASH` indexes, comparison of a key part with a constant value is a range condition when using the `=`, `<=>`, `IN()`, `IS NULL`, or `IS NOT NULL` operators.
- Additionally, for `BTREE` indexes, comparison of a key part with a constant value is a range condition when using the `>`, `<`, `>=`, `<=`, `BETWEEN`, `!=`, or `<>` operators, or `LIKE` comparisons if the argument to `LIKE` is a constant string that does not start with a wildcard character.
- For all index types, multiple range conditions combined with `OR` or `AND` form a range condition.

“Constant value” in the preceding descriptions means one of the following:

- A constant from the query string
- A column of a `const` or `system` table from the same join
- The result of an uncorrelated subquery
- Any expression composed entirely from subexpressions of the preceding types

Here are some examples of queries with range conditions in the `WHERE` clause:

```
SELECT * FROM t1
  WHERE key_col > 1
  AND key_col < 10;

SELECT * FROM t1
  WHERE key_col = 1
  OR key_col IN (15,18,20);

SELECT * FROM t1
  WHERE key_col LIKE 'ab%'
  OR key_col BETWEEN 'bar' AND 'foo';
```

Some nonconstant values may be converted to constants during the optimizer constant propagation phase.

MySQL tries to extract range conditions from the `WHERE` clause for each of the possible indexes. During the extraction process, conditions that cannot be used for constructing the range condition are dropped, conditions that produce overlapping ranges are combined, and conditions that produce empty ranges are removed.

Consider the following statement, where `key1` is an indexed column and `nonkey` is not indexed:

```
SELECT * FROM t1 WHERE
  (key1 < 'abc' AND (key1 LIKE 'abcde%' OR key1 LIKE '%b')) OR
  (key1 < 'bar' AND nonkey = 4) OR
  (key1 < 'uux' AND key1 > 'z');
```

The extraction process for key `key1` is as follows:

1. Start with original `WHERE` clause:

   ```
   (key1 < 'abc' AND (key1 LIKE 'abcde%' OR key1 LIKE '%b')) OR
   (key1 < 'bar' AND nonkey = 4) OR
   (key1 < 'uux' AND key1 > 'z')
   ```

2. Remove `nonkey = 4` and `key1 LIKE '%b'` because they cannot be used for a range scan. The correct way to remove them is to replace them with `TRUE`, so that we do not miss any matching rows when doing the range scan. Replacing them with `TRUE` yields:

   ```
   (key1 < 'abc' AND (key1 LIKE 'abcde%' OR TRUE)) OR
   (key1 < 'bar' AND TRUE) OR
   (key1 < 'uux' AND key1 > 'z')
   ```

3. Collapse conditions that are always true or false:
   - `(key1 LIKE 'abcde%' OR TRUE)` is always true
   - `(key1 < 'uux' AND key1 > 'z')` is always false

   Replacing these conditions with constants yields:

   ```
   (key1 < 'abc' AND TRUE) OR (key1 < 'bar' AND TRUE) OR (FALSE)
   ```

   Removing unnecessary `TRUE` and `FALSE` constants yields:

   ```
   (key1 < 'abc') OR (key1 < 'bar')
   ```

4. Combining overlapping intervals into one yields the final condition to be used for the range scan:

   ```
   (key1 < 'bar')
   ```

In general (and as demonstrated by the preceding example), the condition used for a range scan is less restrictive than the `WHERE` clause. MySQL performs an additional check to filter out rows that satisfy the range condition but not the full `WHERE` clause.

The range condition extraction algorithm can handle nested `AND`/`OR` constructs of arbitrary depth, and its output does not depend on the order in which conditions appear in `WHERE` clause.

MySQL does not support merging multiple ranges for the `range` access method for spatial indexes. To work around this limitation, you can use a `UNION` with identical `SELECT` statements, except that you put each spatial predicate in a different `SELECT`.

Range Access Method for Multiple-Part Indexes

Range conditions on a multiple-part index are an extension of range conditions for a single-part index. A range condition on a multiple-part index restricts index rows to lie within one or several key tuple intervals. Key tuple intervals are defined over a set of key tuples, using ordering from the index.

For example, consider a multiple-part index defined as `key1(`*`key_part1`*`,`*`key_part2`*`,`*`key_part3`*`)`, and the following set of key tuples listed in key order:

```
key_part1  key_part2  key_part3
  NULL       1        'abc'
  NULL       1        'xyz'
  NULL       2        'foo'
   1         1        'abc'
   1         1        'xyz'
   1         2        'abc'
   2         1        'aaa'
```

The condition `key_part1 = 1` defines this interval:

```
(1,-inf,-inf) <= (key_part1,key_part2,key_part3) < (1,+inf,+inf)
```

The interval covers the 4th, 5th, and 6th tuples in the preceding data set and can be used by the range access method.

By contrast, the condition `key_part3 = 'abc'` does not define a single interval and cannot be used by the range access method.

The following descriptions indicate how range conditions work for multiple-part indexes in greater detail.

- For `HASH` indexes, each interval containing identical values can be used. This means that the interval can be produced only for conditions in the following form:

  ```
  key_part1 cmp const1
  ```

```
AND key_part2 cmp const2
AND ...
AND key_partN cmp constN;
```

Here, `const1`, `const2`, ... are constants, `cmp` is one of the `=`, `<=>`, or `IS NULL` comparison operators, and the conditions cover all index parts. (That is, there are `N` conditions, one for each part of an `N`-part index.) For example, the following is a range condition for a three-part `HASH` index:

```
key_part1 = 1 AND key_part2 IS NULL AND key_part3 = 'foo'
```

For the definition of what is considered to be a constant, see Range Access Method for Single-Part Indexes.

- For a `BTREE` index, an interval might be usable for conditions combined with `AND`, where each condition compares a key part with a constant value using `=`, `<=>`, `IS NULL`, `>`, `<`, `>=`, `<=`, `!=`, `<>`, `BETWEEN`, or `LIKE 'pattern'` (where `'pattern'` does not start with a wildcard). An interval can be used as long as it is possible to determine a single key tuple containing all rows that match the condition (or two intervals if `<>` or `!=` is used).
  The optimizer attempts to use additional key parts to determine the interval as long as the comparison operator is `=`, `<=>`, or `IS NULL`. If the operator is `>`, `<`, `>=`, `<=`, `!=`, `<>`, `BETWEEN`, or `LIKE`, the optimizer uses it but considers no more key parts. For the following expression, the optimizer uses `=` from the first comparison. It also uses `>=` from the second comparison but considers no further key parts and does not use the third comparison for interval construction:

  ```
  key_part1 = 'foo' AND key_part2 >= 10 AND key_part3 > 10
  ```

  The single interval is:

  ```
  ('foo',10,-inf) < (key_part1,key_part2,key_part3) < ('foo',+inf,+inf)
  ```

  It is possible that the created interval contains more rows than the initial condition. For example, the preceding interval includes the value `('foo', 11, 0)`, which does not satisfy the original condition.
- If conditions that cover sets of rows contained within intervals are combined with `OR`, they form a condition that covers a set of rows contained within the union of their intervals. If the conditions are combined with `AND`, they form a condition that covers a set of rows contained within the intersection of their intervals. For example, for this condition on a two-part index:

  ```
  (key_part1 = 1 AND key_part2 < 2) OR (key_part1 > 5)
  ```

  The intervals are:

  ```
  (1,-inf) < (key_part1,key_part2) < (1,2)
  (5,-inf) < (key_part1,key_part2)
  ```

  In this example, the interval on the first line uses one key part for the left bound and two key parts for the right bound. The interval on the second line uses only one key part. The `key_len` column in the `EXPLAIN` output indicates the maximum length of the key prefix used.
  In some cases, `key_len` may indicate that a key part was used, but that might be not what you would expect. Suppose that `key_part1` and `key_part2` can be `NULL`. Then the `key_len` column displays two key part lengths for the following condition:

  ```
  key_part1 >= 1 AND key_part2 < 2
  ```

  But, in fact, the condition is converted to this:

  ```
  key_part1 >= 1 AND key_part2 IS NOT NULL
  ```

For a description of how optimizations are performed to combine or eliminate intervals for range conditions on a single-part index, see Range Access Method for Single-Part Indexes. Analogous steps are performed for range conditions on multiple-part indexes.

Equality Range Optimization of Many-Valued Comparisons

Consider these expressions, where `col_name` is an indexed column:

```
col_name IN(val1, ..., valN)
col_name = val1 OR ... OR col_name = valN
```

Each expression is true if `col_name` is equal to any of several values. These comparisons are equality range comparisons (where the “range” is a single value). The optimizer estimates the cost of reading qualifying rows for equality range comparisons as follows:

- If there is a unique index on `col_name`, the row estimate for each range is 1 because at most one row can have the given value.
- Otherwise, any index on `col_name` is nonunique and the optimizer can estimate the row count for each range using dives into the index or index statistics.

With index dives, the optimizer makes a dive at each end of a range and uses the number of rows in the range as the estimate. For example, the expression `col_name IN (10, 20, 30)` has three equality ranges and the optimizer makes two

dives per range to generate a row estimate. Each pair of dives yields an estimate of the number of rows that have the given value.

Index dives provide accurate row estimates, but as the number of comparison values in the expression increases, the optimizer takes longer to generate a row estimate. Use of index statistics is less accurate than index dives but permits faster row estimation for large value lists.

The `eq_range_index_dive_limit` system variable enables you to configure the number of values at which the optimizer switches from one row estimation strategy to the other. To permit use of index dives for comparisons of up to *`N`* equality ranges, set `eq_range_index_dive_limit` to *`N`* + 1. To disable use of statistics and always use index dives regardless of *`N`*, set `eq_range_index_dive_limit` to 0.

To update table index statistics for best estimates, use `ANALYZE TABLE`.

Prior to MySQL 8.0, there is no way of skipping the use of index dives to estimate index usefulness, except by using the `eq_range_index_dive_limit` system variable. In MySQL 8.0, index dive skipping is possible for queries that satisfy all these conditions:

- The query is for a single table, not a join on multiple tables.
- A single-index `FORCE INDEX` index hint is present. The idea is that if index use is forced, there is nothing to be gained from the additional overhead of performing dives into the index.
- The index is nonunique and not a `FULLTEXT` index.
- No subquery is present.
- No `DISTINCT`, `GROUP BY`, or `ORDER BY` clause is present.

For `EXPLAIN FOR CONNECTION`, the output changes as follows if index dives are skipped:

- For traditional output, the `rows` and `filtered` values are `NULL`.
- For JSON output, `rows_examined_per_scan` and `rows_produced_per_join` do not appear, `skip_index_dive_due_to_force` is `true`, and cost calculations are not accurate.

Without `FOR CONNECTION`, `EXPLAIN` output does not change when index dives are skipped.

After execution of a query for which index dives are skipped, the corresponding row in the Information Schema `OPTIMIZER_TRACE` table contains an `index_dives_for_range_access` value of `skipped_due_to_force_index`.

Skip Scan Range Access Method

Consider the following scenario:

```
CREATE TABLE t1 (f1 INT NOT NULL, f2 INT NOT NULL, PRIMARY KEY(f1, f2));
INSERT INTO t1 VALUES
 (1,1), (1,2), (1,3), (1,4), (1,5),
 (2,1), (2,2), (2,3), (2,4), (2,5);
INSERT INTO t1 SELECT f1, f2 + 5 FROM t1;
INSERT INTO t1 SELECT f1, f2 + 10 FROM t1;
INSERT INTO t1 SELECT f1, f2 + 20 FROM t1;
INSERT INTO t1 SELECT f1, f2 + 40 FROM t1;
ANALYZE TABLE t1;

EXPLAIN SELECT f1, f2 FROM t1 WHERE f2 > 40;
```

To execute this query, MySQL can choose an index scan to fetch all rows (the index includes all columns to be selected), then apply the `f2 > 40` condition from the `WHERE` clause to produce the final result set.

A range scan is more efficient than a full index scan, but cannot be used in this case because there is no condition on `f1`, the first index column. However, as of MySQL 8.0.13, the optimizer can perform multiple range scans, one for each value of `f1`, using a method called Skip Scan that is similar to Loose Index Scan (see Section 8.2.1.17, “GROUP BY Optimization”):

1. Skip between distinct values of the first index part, `f1` (the index prefix).
2. Perform a subrange scan on each distinct prefix value for the `f2 > 40` condition on the remaining index part.

For the data set shown earlier, the algorithm operates like this:

1. Get the first distinct value of the first key part (`f1 = 1`).
2. Construct the range based on the first and second key parts (`f1 = 1 AND f2 > 40`).
3. Perform a range scan.
4. Get the next distinct value of the first key part (`f1 = 2`).
5. Construct the range based on the first and second key parts (`f1 = 2 AND f2 > 40`).
6. Perform a range scan.

Using this strategy decreases the number of accessed rows because MySQL skips the rows that do not qualify for each constructed range. This Skip Scan access method is applicable under the following conditions:

- Table T has at least one compound index with key parts of the form ([A_1, ..., A_*k*,] B_1, ..., B_*m*, C [, D_1, ..., D_*n*]). Key parts A and D may be empty, but B and C must be nonempty.
- The query references only one table.
- The query does not use `GROUP BY` or `DISTINCT`.
- The query references only columns in the index.
- The predicates on A_1, ..., A_*k* must be equality predicates and they must be constants. This includes the `IN()` operator.
- The query must be a conjunctive query; that is, an `AND` of `OR` conditions: `(`*cond1*`(`*key_part1*`) OR`*cond2*`(`*key_part1*`))` `AND (`*cond1*`(`*key_part2*`) OR ...) AND ...`
- There must be a range condition on C.
- Conditions on D columns are permitted. Conditions on D must be in conjunction with the range condition on C.

Use of Skip Scan is indicated in `EXPLAIN` output as follows:

- `Using index for skip scan` in the `Extra` column indicates that the loose index Skip Scan access method is used.
- If the index can be used for Skip Scan, the index should be visible in the `possible_keys` column.

Use of Skip Scan is indicated in optimizer trace output by a `"skip scan"` element of this form:

```
"skip_scan_range": {
  "type": "skip_scan",
  "index": index_used_for_skip_scan,
  "key_parts_used_for_access": [key_parts_used_for_access],
  "range": [range]
}
```

You may also see a `"best_skip_scan_summary"` element. If Skip Scan is chosen as the best range access variant, a `"chosen_range_access_summary"` is written. If Skip Scan is chosen as the overall best access method, a `"best_access_path"` element is present.

Use of Skip Scan is subject to the value of the `skip_scan` flag of the `optimizer_switch` system variable. See Section 8.9.2, “Switchable Optimizations”. By default, this flag is `on`. To disable it, set `skip_scan` to `off`.

In addition to using the `optimizer_switch` system variable to control optimizer use of Skip Scan session-wide, MySQL supports optimizer hints to influence the optimizer on a per-statement basis. See Section 8.9.3, “Optimizer Hints”.

Range Optimization of Row Constructor Expressions

The optimizer is able to apply the range scan access method to queries of this form:

```
SELECT ... FROM t1 WHERE ( col_1, col_2 ) IN (( 'a', 'b' ), ( 'c', 'd' ));
```

Previously, for range scans to be used, it was necessary to write the query as:

```
SELECT ... FROM t1 WHERE ( col_1 = 'a' AND col_2 = 'b' )
OR ( col_1 = 'c' AND col_2 = 'd' );
```

For the optimizer to use a range scan, queries must satisfy these conditions:

- Only `IN()` predicates are used, not `NOT IN()`.
- On the left side of the `IN()` predicate, the row constructor contains only column references.
- On the right side of the `IN()` predicate, row constructors contain only runtime constants, which are either literals or local column references that are bound to constants during execution.
- On the right side of the `IN()` predicate, there is more than one row constructor.

For more information about the optimizer and row constructors, see Section 8.2.1.22, “Row Constructor Expression Optimization”

Limiting Memory Use for Range Optimization

To control the memory available to the range optimizer, use the `range_optimizer_max_mem_size` system variable:

- A value of 0 means “no limit.”
- With a value greater than 0, the optimizer tracks the memory consumed when considering the range access method. If the specified limit is about to be exceeded, the range access method is abandoned and other methods, including a full table scan, are considered instead. This could be less optimal. If this happens, the following warning occurs (where `N` is the current `range_optimizer_max_mem_size` value):

  ```
  Warning  3170  Memory capacity of N bytes for
                 'range_optimizer_max_mem_size' exceeded. Range
                 optimization was not done for this query.
  ```

- For `UPDATE` and `DELETE` statements, if the optimizer falls back to a full table scan and

the sql_safe_updates system variable is enabled, an error occurs rather than a warning because, in effect, no key is used to determine which rows to modify. For more information, see Using Safe-Updates Mode (--safe-updates).

For individual queries that exceed the available range optimization memory and for which the optimizer falls back to less optimal plans, increasing the range_optimizer_max_mem_size value may improve performance.

To estimate the amount of memory needed to process a range expression, use these guidelines:

- For a simple query such as the following, where there is one candidate key for the range access method, each predicate combined with OR uses approximately 230 bytes:

```
SELECT COUNT(*) FROM t
WHERE a=1 OR a=2 OR a=3 OR .. . a=N;
```

- Similarly for a query such as the following, each predicate combined with AND uses approximately 125 bytes:

```
SELECT COUNT(*) FROM t
WHERE a=1 AND b=1 AND c=1 ... N;
```

- For a query with IN() predicates:

```
SELECT COUNT(*) FROM t
WHERE a IN (1,2, ..., M) AND b IN (1,2, ..., N);
```

  Each literal value in an IN() list counts as a predicate combined with OR. If there are two IN() lists, the number of predicates combined with OR is the product of the number of literal values in each list. Thus, the number of predicates combined with OR in the preceding case is M × N.

8.2.1.3 인덱스 머지 최적화

The Index Merge access method retrieves rows with multiple range scans and merges their results into one. This access method merges index scans from a single table only, not scans across multiple tables. The merge can produce unions, intersections, or unions-of-intersections of its underlying scans.

Example queries for which Index Merge may be used:

```
SELECT * FROM tbl_name WHERE key1 = 10 OR key2 = 20;

SELECT * FROM tbl_name
  WHERE (key1 = 10 OR key2 = 20) AND non_key = 30;

SELECT * FROM t1, t2
  WHERE (t1.key1 IN (1,2) OR t1.key2 LIKE 'value%')
  AND t2.key1 = t1.some_col;

SELECT * FROM t1, t2
  WHERE t1.key1 = 1
  AND (t2.key1 = t1.some_col OR t2.key2 = t1.some_col2);
```

Note
The Index Merge optimization algorithm has the following known limitations:

- If your query has a complex WHERE clause with deep AND/OR nesting and MySQL does not choose the optimal plan, try distributing terms using the following identity transformations:

```
(x AND y) OR z => (x OR z) AND (y OR z)
(x OR y) AND z => (x AND z) OR (y AND z)
```

- Index Merge is not applicable to full-text indexes.

In EXPLAIN output, the Index Merge method appears as index_merge in the type column. In this case, the key column contains a list of indexes used, and key_len contains a list of the longest key parts for those indexes.

The Index Merge access method has several algorithms, which are displayed in the Extra field of EXPLAIN output:

- Using intersect(...)
- Using union(...)
- Using sort_union(...)

The following sections describe these algorithms in greater detail. The optimizer chooses between different possible Index Merge algorithms and other access methods based on cost estimates of the various available options.

Index Merge Intersection Access Algorithm

This access algorithm is applicable when a `WHERE` clause is converted to several range conditions on different keys combined with `AND`, and each condition is one of the following:

- An `N`-part expression of this form, where the index has exactly `N` parts (that is, all index parts are covered):

  ```
  key_part1 = const1 AND key_part2 = const2 ... AND key_partN = constN
  ```

- Any range condition over the primary key of an `InnoDB` table.

Examples:

```
SELECT * FROM innodb_table
  WHERE primary_key < 10 AND key_col1 = 20;

SELECT * FROM tbl_name
  WHERE key1_part1 = 1 AND key1_part2 = 2 AND key2 = 2;
```

The Index Merge intersection algorithm performs simultaneous scans on all used indexes and produces the intersection of row sequences that it receives from the merged index scans.

If all columns used in the query are covered by the used indexes, full table rows are not retrieved `EXPLAIN` output contains `Using index` in `Extra` field in this case). Here is an example of such a query:

```
SELECT COUNT(*) FROM t1 WHERE key1 = 1 AND key2 = 1;
```

If the used indexes do not cover all columns used in the query, full rows are retrieved only when the range conditions for all used keys are satisfied.

If one of the merged conditions is a condition over the primary key of an `InnoDB` table, it is not used for row retrieval, but is used to filter out rows retrieved using other conditions.

Index Merge Union Access Algorithm

The criteria for this algorithm are similar to those for the Index Merge intersection algorithm. The algorithm is applicable when the table's `WHERE` clause is converted to several range conditions on different keys combined with `OR`, and each condition is one of the following:

- An `N`-part expression of this form, where the index has exactly `N` parts (that is, all index parts are covered):

  ```
  key_part1 = const1 OR key_part2 = const2 ... OR key_partN = constN
  ```

- Any range condition over a primary key of an `InnoDB` table.
- A condition for which the Index Merge intersection algorithm is applicable.

Examples:

```
SELECT * FROM t1
  WHERE key1 = 1 OR key2 = 2 OR key3 = 3;

SELECT * FROM innodb_table
  WHERE (key1 = 1 AND key2 = 2)
    OR (key3 = 'foo' AND key4 = 'bar') AND key5 = 5;
```

Index Merge Sort-Union Access Algorithm

This access algorithm is applicable when the `WHERE` clause is converted to several range conditions combined by `OR`, but the Index Merge union algorithm is not applicable.

Examples:

```
SELECT * FROM tbl_name
  WHERE key_col1 < 10 OR key_col2 < 20;

SELECT * FROM tbl_name
  WHERE (key_col1 > 10 OR key_col2 = 20) AND nonkey_col = 30;
```

The difference between the sort-union algorithm and the union algorithm is that the sort-union algorithm must first fetch row IDs for all rows and sort them before returning any rows.

Influencing Index Merge Optimization

Use of Index Merge is subject to the value of the `index_merge`, `index_merge_intersection`, `index_merge_union`, and `index_merge_sort_union` flags of the `optimizer_switch` system variable. See Section 8.9.2, “Switchable Optimizations”. By default, all those flags are `on`. To enable only certain algorithms, set `index_merge` to `off`, and enable only such of the others as should be permitted.

In addition to using the `optimizer_switch` system variable to control optimizer use of the Index Merge algorithms session-wide, MySQL supports optimizer hints to influence the optimizer on a per-statement basis. See Section 8.9.3, “Optimizer Hints”.

8.2.1.4 해쉬 조인 최적화

By default, MySQL (8.0.18 and later) employs hash joins whenever possible. It is possible to control whether hash joins are employed using one of the `BNL` and `NO_BNL` optimizer hints, or by setting `block_nested_loop=on` or `block_nested_loop=off` as part of the setting for the optimizer_switch server system variable.

Note
MySQL 8.0.18 supported setting a `hash_join` flag in `optimizer_switch`, as well as the optimizer hints `HASH_JOIN` and `NO_HASH_JOIN`. In MySQL 8.0.19 and later, none of these have any effect any longer.

Beginning with MySQL 8.0.18, MySQL employs a hash join for any query for which each join has an equi-join condition, and in which there are no indexes that can be applied to any join conditions, such as this one:

```
SELECT *
   FROM t1
   JOIN t2
      ON t1.c1=t2.c1;
```

A hash join can also be used when there are one or more indexes that can be used for single-table predicates.

A hash join is usually faster than and is intended to be used in such cases instead of the block nested loop algorithm (see Block Nested-Loop Join Algorithm) employed in previous versions of MySQL. Beginning with MySQL 8.0.20, support for block nested loop is removed, and the server employs a hash join wherever a block nested loop would have been used previously.

In the example just shown and the remaining examples in this section, we assume that the three tables `t1`, `t2`, and `t3` have been created using the following statements:

```
CREATE TABLE t1 (c1 INT, c2 INT);
CREATE TABLE t2 (c1 INT, c2 INT);
CREATE TABLE t3 (c1 INT, c2 INT);
```

You can see that a hash join is being employed by using `EXPLAIN`, like this:

```
mysql> EXPLAIN
    -> SELECT * FROM t1
    ->     JOIN t2 ON t1.c1=t2.c1\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1
     filtered: 100.00
        Extra: NULL
```

```
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: t2
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1
     filtered: 100.00
        Extra: Using where; Using join buffer (hash join)
```

(Prior to MySQL 8.0.20, it was necessary to include the `FORMAT=TREE` option to see whether hash joins were being used for a given join.)

`EXPLAIN ANALYZE` also displays information about hash joins used.

The hash join is used for queries involving multiple joins as well, as long as at least one join condition for each pair of tables is an equi-join, like the query shown here:

```
SELECT * FROM t1
    JOIN t2 ON (t1.c1 = t2.c1 AND t1.c2 < t2.c2)
    JOIN t3 ON (t2.c1 = t3.c1);
```

In cases like the one just shown, which makes use of an inner join, any extra conditions which are not equi-joins are applied as filters after the join is executed. (For outer joins, such as left joins, semijoins, and antijoins, they are printed as part of the join.) This can be seen here in the output of `EXPLAIN`:

```
mysql> EXPLAIN FORMAT=TREE
    -> SELECT *
    ->     FROM t1
    ->     JOIN t2
    ->         ON (t1.c1 = t2.c1 AND t1.c2 < t2.c2)
    ->     JOIN t3
    ->         ON (t2.c1 = t3.c1)\G
*************************** 1. row ***************************
EXPLAIN: -> Inner hash join (t3.c1 = t1.c1)  (cost=1.05 rows=1)
    -> Table scan on t3  (cost=0.35 rows=1)
    -> Hash
        -> Filter: (t1.c2 < t2.c2)  (cost=0.70 rows=1)
            -> Inner hash join (t2.c1 = t1.c1)  (cost=0.70 rows=1)
                -> Table scan on t2  (cost=0.35 rows=1)
                -> Hash
                    -> Table scan on t1  (cost=0.35 rows=1)
```

As also can be seen from the output just shown, multiple hash joins can be (and are) used for joins having multiple equi-join conditions.

Prior to MySQL 8.0.20, a hash join could not be used if any pair of joined tables did not have at least one equi-join condition, and the slower block nested loop algorithm was employed. In MySQL 8.0.20 and later, the hash join is used in such cases, as shown here:

```
mysql> EXPLAIN FORMAT=TREE
    -> SELECT * FROM t1
    ->     JOIN t2 ON (t1.c1 = t2.c1)
    ->     JOIN t3 ON (t2.c1 < t3.c1)\G
*************************** 1. row ***************************
EXPLAIN: -> Filter: (t1.c1 < t3.c1)  (cost=1.05 rows=1)
    -> Inner hash join (no condition)  (cost=1.05 rows=1)
        -> Table scan on t3  (cost=0.35 rows=1)
        -> Hash
            -> Inner hash join (t2.c1 = t1.c1)  (cost=0.70 rows=1)
                -> Table scan on t2  (cost=0.35 rows=1)
                -> Hash
                    -> Table scan on t1  (cost=0.35 rows=1)
```

(Additional examples are provided later in this section.)

A hash join is also applied for a Cartesian product—that is, when no join condition is specified, as shown here:

```
mysql> EXPLAIN FORMAT=TREE
    -> SELECT *
    ->     FROM t1
    ->     JOIN t2
    ->     WHERE t1.c2 > 50\G
*************************** 1. row ***************************
EXPLAIN: -> Inner hash join  (cost=0.70 rows=1)
    -> Table scan on t2  (cost=0.35 rows=1)
    -> Hash
        -> Filter: (t1.c2 > 50)  (cost=0.35 rows=1)
            -> Table scan on t1  (cost=0.35 rows=1)
```

In MySQL 8.0.20 and later, it is no longer necessary for the join to contain at least one equi-join condition in order for a hash join to be used. This means that the types of queries which can be optimized using hash joins include those in the following list (with examples):

- *Inner non-equi-join*:

  ```
  mysql> EXPLAIN FORMAT=TREE SELECT * FROM t1 JOIN t2 ON t1.c1 < t2.c1\G
  *************************** 1. row ***************************
  EXPLAIN: -> Filter: (t1.c1 < t2.c1)  (cost=4.70 rows=12)
      -> Inner hash join (no condition)  (cost=4.70 rows=12)
          -> Table scan on t2  (cost=0.08 rows=6)
          -> Hash
              -> Table scan on t1  (cost=0.85 rows=6)
  ```

- *Semijoin*:

  ```
  mysql> EXPLAIN FORMAT=TREE SELECT * FROM t1
      ->     WHERE t1.c1 IN (SELECT t2.c2 FROM t2)\G
  *************************** 1. row ***************************
  EXPLAIN: -> Hash semijoin (t2.c2 = t1.c1)  (cost=0.70 rows=1)
      -> Table scan on t1  (cost=0.35 rows=1)
      -> Hash
          -> Table scan on t2  (cost=0.35 rows=1)
  ```

- *Antijoin*:

  ```
  mysql> EXPLAIN FORMAT=TREE SELECT * FROM t2
      ->     WHERE NOT EXISTS (SELECT * FROM t1 WHERE t1.c1 = t2.c1)\G
  *************************** 1. row ***************************
  EXPLAIN: -> Hash antijoin (t1.c1 = t2.c1)  (cost=0.70 rows=1)
      -> Table scan on t2  (cost=0.35 rows=1)
      -> Hash
          -> Table scan on t1  (cost=0.35 rows=1)

  1 row in set, 1 warning (0.00 sec)

  mysql> SHOW WARNINGS\G
  *************************** 1. row ***************************
    Level: Note
     Code: 1276
  Message: Field or reference 't3.t2.c1' of SELECT #2 was resolved in SELECT #1
  ```

- *Left outer join*:

  ```
  mysql> EXPLAIN FORMAT=TREE SELECT * FROM t1 LEFT JOIN t2 ON t1.c1 = t2.c1\G
  *************************** 1. row ***************************
  EXPLAIN: -> Left hash join (t2.c1 = t1.c1)  (cost=0.70 rows=1)
      -> Table scan on t1  (cost=0.35 rows=1)
      -> Hash
          -> Table scan on t2  (cost=0.35 rows=1)
  ```

- *Right outer join* (observe that MySQL rewrites all right outer joins as left outer joins):

  ```
  mysql> EXPLAIN FORMAT=TREE SELECT * FROM t1 RIGHT JOIN t2 ON t1.c1 = t2.c1\G
  *************************** 1. row ***************************
  EXPLAIN: -> Left hash join (t1.c1 = t2.c1)  (cost=0.70 rows=1)
      -> Table scan on t2  (cost=0.35 rows=1)
      -> Hash
  ```

```
    -> Table scan on t1  (cost=0.35 rows=1)
```

By default, MySQL 8.0.18 and later employs hash joins whenever possible. It is possible to control whether hash joins are employed using one of the `BNL` and `NO_BNL` optimizer hints.

(MySQL 8.0.18 supported `hash_join=on` or `hash_join=off` as part of the setting for the `optimizer_switch` server system variable as well as the optimizer hints `HASH_JOIN` or `NO_HASH_JOIN`. In MySQL 8.0.19 and later, these no longer have any effect.)

Memory usage by hash joins can be controlled using the `join_buffer_size` system variable; a hash join cannot use more memory than this amount. When the memory required for a hash join exceeds the amount available, MySQL handles this by using files on disk. If this happens, you should be aware that the join may not succeed if a hash join cannot fit into memory and it creates more files than set for `open_files_limit`. To avoid such problems, make either of the following changes:

- Increase `join_buffer_size` so that the hash join does not spill over to disk.
- Increase `open_files_limit`.

Beginning with MySQL 8.0.18, join buffers for hash joins are allocated incrementally; thus, you can set `join_buffer_size` higher without small queries allocating very large amounts of RAM, but outer joins allocate the entire buffer. In MySQL 8.0.20 and later, hash joins are used for outer joins (including antijoins and semijoins) as well, so this is no longer an issue.

8.2.1.5 엔진 조건 푸시 다운 최적화

This optimization improves the efficiency of direct comparisons between a nonindexed column and a constant. In such cases, the condition is “pushed down” to the storage engine for evaluation. This optimization can be used only by the `NDB` storage engine.

For NDB Cluster, this optimization can eliminate the need to send nonmatching rows over the network between the cluster's data nodes and the MySQL server that issued the query, and can speed up queries where it is used by a factor of 5 to 10 times over cases where condition pushdown could be but is not used.

Suppose that an NDB Cluster table is defined as follows:

```
CREATE TABLE t1 (
    a INT,
    b INT,
    KEY(a)
) ENGINE=NDB;
```

Engine condition pushdown can be used with queries such as the one shown here, which includes a comparison between a nonindexed column and a constant:

```
SELECT a, b FROM t1 WHERE b = 10;
```

The use of engine condition pushdown can be seen in the output of `EXPLAIN`:

```
mysql> EXPLAIN SELECT a, b FROM t1 WHERE b = 10\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 10
        Extra: Using where with pushed condition
```

However, engine condition pushdown *cannot* be used with the following query:

```
SELECT a,b FROM t1 WHERE a = 10;
```

Engine condition pushdown is not applicable here because an index exists on column `a`. (An index access method would be more efficient and so would be chosen in preference to condition pushdown.)

Engine condition pushdown may also be employed when an indexed column is compared with a constant using a `>` or `<` operator:

```
mysql> EXPLAIN SELECT a, b FROM t1 WHERE a < 2\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
         type: range
possible_keys: a
          key: a
      key_len: 5
          ref: NULL
         rows: 2
        Extra: Using where with pushed condition
```

Other supported comparisons for engine condition pushdown include the following:

- `column` [NOT] LIKE `pattern`
  `pattern` must be a string literal containing the pattern to be matched; for syntax, see Section 12.8.1, “String Comparison Functions and Operators”.
- `column` IS [NOT] NULL
- `column` IN (`value_list`)
  Each item in the `value_list` must be a constant, literal value.
- `column` BETWEEN `constant1` AND `constant2`
  `constant1` and `constant2` must each be a constant, literal value.

In all of the cases in the preceding list, it is possible for the condition to be converted into the form of one or more direct comparisons between a column and a constant.

Engine condition pushdown is enabled by default. To disable it at server startup, set the `optimizer_switch` system variable's `engine_condition_pushdown` flag to `off`. For example, in a `my.cnf` file, use these lines:

```
[mysqld]
optimizer_switch=engine_condition_pushdown=off
```

At runtime, disable condition pushdown like this:

```
SET optimizer_switch='engine_condition_pushdown=off';
```

Limitations. Engine condition pushdown is subject to the following limitations:

- Engine condition pushdown is supported only by the `NDB` storage engine.
- Prior to NDB 8.0.18, columns could be compared with constants or expressions which evaluate to constant values only. In NDB 8.0.18 and later, columns can be compared with one another as long as they are of exactly the same type, including the same signedness, length, character set, precision, and scale, where these are applicable.
- Columns used in comparisons cannot be of any of the `BLOB` or `TEXT` types. This exclusion extends to `JSON`, `BIT`, and `ENUM` columns as well.
- A string value to be compared with a column must use the same collation as the column.
- Joins are not directly supported; conditions involving multiple tables are pushed separately where possible. Use extended `EXPLAIN` output to determine which conditions are actually pushed down. See Section 8.8.3, “Extended EXPLAIN Output Format”.

Previously, engine condition pushdown was limited to terms referring to column values from the same table to which the condition was being pushed. Beginning with NDB 8.0.16, column values from tables earlier in the query plan can also be referred to from pushed conditions. This reduces the number of rows which must be handled by the SQL node during join processing. Filtering can be also performed in parallel in the LDM threads, rather than in a single **mysqld** process. This has the potential to improve performance of queries by a significant margin.

Beginning with NDB 8.0.20, an outer join using a scan can be pushed if there are no unpushable conditions on any table used in the same join nest, or on any table in join nests above it on which it depends. This is also true for a semijoin, provided the optimization strategy employed is `firstMatch` (see Section 8.2.2.1, “Optimizing IN and EXISTS Subquery Predicates with Semijoin Transformations”).

Join algorithms cannot be combined with referring columns from previous tables in the following two situations:

1. When any of the referred previous tables are in a join buffer. In this case, each row retrieved from the scan-filtered table is matched against every row in the buffer. This means that there is no single specific row from which column values can be fetched from when generating the scan filter.
2. When the column originates from a child operation in a pushed join. This is because rows referenced from ancestor operations in the join have not yet been retrieved when the scan filter is generated.

Beginning with NDB 8.0.27, columns from ancestor tables in a join can be pushed down, provided that they meet the requirements listed previously. An example of such a query, using the table `t1` created previously, is shown here:

```
mysql> EXPLAIN
    -> SELECT * FROM t1 AS x
    -> LEFT JOIN t1 AS y
    -> ON x.a=0 AND y.b>=3\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: x
   partitions: p0,p1
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 4
     filtered: 100.00
        Extra: NULL
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: y
   partitions: p0,p1
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 4
     filtered: 100.00
        Extra: Using where; Using pushed condition (`test`.`y`.`b` >= 3); Using join buffer (hash join)
2 rows in set, 2 warnings (0.00 sec)
```

8.2.1.6 인덱스 조건 푸시 다운 최적화

Index Condition Pushdown (ICP) is an optimization for the case where MySQL retrieves rows from a table using an index. Without ICP, the storage engine traverses the index to locate rows in the base table and returns them to the MySQL server which evaluates the `WHERE` condition for the rows. With ICP enabled, and if parts of the `WHERE` condition can be evaluated by using only columns from the index, the MySQL server pushes this part of the `WHERE` condition down to the storage engine. The storage engine then evaluates the pushed index condition by using the index entry and only if this is satisfied is the row read from the table. ICP can reduce the number of times the storage engine must access the base table and the number of times the MySQL server must access the storage engine.

Applicability of the Index Condition Pushdown optimization is subject to these conditions:

- ICP is used for the `range`, `ref`, `eq_ref`, and `ref_or_null` access methods when there is a need to access full table rows.
- ICP can be used for `InnoDB` and `MyISAM` tables, including partitioned `InnoDB` and `MyISAM` tables.
- For `InnoDB` tables, ICP is used only for secondary indexes. The goal of ICP is to reduce the number of full-row reads and thereby reduce I/O operations. For `InnoDB` clustered indexes, the complete record is already read into the `InnoDB` buffer. Using ICP in this case does not reduce I/O.
- ICP is not supported with secondary indexes created on virtual generated columns. `InnoDB` supports secondary indexes on virtual generated columns.
- Conditions that refer to subqueries cannot be pushed down.
- Conditions that refer to stored functions cannot be pushed down. Storage engines cannot invoke stored functions.
- Triggered conditions cannot be pushed down. (For information about triggered conditions, see Section 8.2.2.3, “Optimizing Subqueries with the EXISTS Strategy”.)
- (*MySQL 8.0.30 and later*:) Conditions cannot be pushed down to derived tables containing references to system variables.

To understand how this optimization works, first consider how an index scan proceeds when Index Condition Pushdown is not used:

1. Get the next row, first by reading the index tuple, and then by using the index tuple to locate and read the full table row.
2. Test the part of the `WHERE` condition that applies to this table. Accept or reject the row based on the test result.

Using Index Condition Pushdown, the scan proceeds like this instead:

1. Get the next row's index tuple (but not the full table row).

2. Test the part of the `WHERE` condition that applies to this table and can be checked using only index columns. If the condition is not satisfied, proceed to the index tuple for the next row.
3. If the condition is satisfied, use the index tuple to locate and read the full table row.
4. Test the remaining part of the `WHERE` condition that applies to this table. Accept or reject the row based on the test result.

`EXPLAIN` output shows `Using index condition` in the `Extra` column when Index Condition Pushdown is used. It does not show `Using index` because that does not apply when full table rows must be read.

Suppose that a table contains information about people and their addresses and that the table has an index defined as `INDEX (zipcode, lastname, firstname)`. If we know a person's `zipcode` value but are not sure about the last name, we can search like this:

```
SELECT * FROM people
  WHERE zipcode='95054'
  AND lastname LIKE '%etrunia%'
  AND address LIKE '%Main Street%';
```

MySQL can use the index to scan through people with `zipcode='95054'`. The second part (`lastname LIKE '%etrunia%'`) cannot be used to limit the number of rows that must be scanned, so without Index Condition Pushdown, this query must retrieve full table rows for all people who have `zipcode='95054'`.

With Index Condition Pushdown, MySQL checks the `lastname LIKE '%etrunia%'` part before reading the full table row. This avoids reading full rows corresponding to index tuples that match the `zipcode` condition but not the `lastname` condition.

Index Condition Pushdown is enabled by default. It can be controlled with the `optimizer_switch` system variable by setting the `index_condition_pushdown` flag:

```
SET optimizer_switch = 'index_condition_pushdown=off';
SET optimizer_switch = 'index_condition_pushdown=on';
```

See Section 8.9.2, “Switchable Optimizations”.

8.2.1.7 네스티드-루프 조인 알고리즘

MySQL executes joins between tables using a nested-loop algorithm or variations on it.

- Nested-Loop Join Algorithm
- Block Nested-Loop Join Algorithm

Nested-Loop Join Algorithm

A simple nested-loop join (NLJ) algorithm reads rows from the first table in a loop one at a time, passing each row to a nested loop that processes the next table in the join. This process is repeated as many times as there remain tables to be joined.

Assume that a join between three tables `t1`, `t2`, and `t3` is to be executed using the following join types:

```
Table   Join Type
t1      range
t2      ref
t3      ALL
```

If a simple NLJ algorithm is used, the join is processed like this:

```
for each row in t1 matching range {
  for each row in t2 matching reference key {
    for each row in t3 {
      if row satisfies join conditions, send to client
    }
  }
}
```

Because the NLJ algorithm passes rows one at a time from outer loops to inner loops, it typically reads tables processed in the inner loops many times.

Block Nested-Loop Join Algorithm

A Block Nested-Loop (BNL) join algorithm uses buffering of rows read in outer loops to reduce the number of times that

tables in inner loops must be read. For example, if 10 rows are read into a buffer and the buffer is passed to the next inner loop, each row read in the inner loop can be compared against all 10 rows in the buffer. This reduces by an order of magnitude the number of times the inner table must be read.

Prior to MySQL 8.0.18, this algorithm was applied for equi-joins when no indexes could be used; in MySQL 8.0.18 and later, the hash join optimization is employed in such cases. Starting with MySQL 8.0.20, the block nested loop is no longer used by MySQL, and a hash join is employed for in all cases where the block nested loop was used previously. See Section 8.2.1.4, “Hash Join Optimization”.

MySQL join buffering has these characteristics:

- Join buffering can be used when the join is of type `ALL` or `index` (in other words, when no possible keys can be used, and a full scan is done, of either the data or index rows, respectively), or `range`. Use of buffering is also applicable to outer joins, as described in Section 8.2.1.12, “Block Nested-Loop and Batched Key Access Joins”.
- A join buffer is never allocated for the first nonconstant table, even if it would be of type `ALL` or `index`.
- Only columns of interest to a join are stored in its join buffer, not whole rows.
- The `join_buffer_size` system variable determines the size of each join buffer used to process a query.
- One buffer is allocated for each join that can be buffered, so a given query might be processed using multiple join buffers.
- A join buffer is allocated prior to executing the join and freed after the query is done.

For the example join described previously for the NLJ algorithm (without buffering), the join is done as follows using join buffering:

```
for each row in t1 matching range {
 for each row in t2 matching reference key {
  store used columns from t1, t2 in join buffer
  if buffer is full {
   for each row in t3 {
    for each t1, t2 combination in join buffer {
     if row satisfies join conditions, send to client
    }
   }
   empty join buffer
  }
 }
}

if buffer is not empty {
 for each row in t3 {
  for each t1, t2 combination in join buffer {
   if row satisfies join conditions, send to client
  }
 }
}
```

If `S` is the size of each stored `t1`, `t2` combination in the join buffer and `C` is the number of combinations in the buffer, the number of times table `t3` is scanned is:

```
(S * C)/join_buffer_size + 1
```

The number of `t3` scans decreases as the value of `join_buffer_size` increases, up to the point when `join_buffer_size` is large enough to hold all previous row combinations. At that point, no speed is gained by making it larger.

8.2.1.8 네스티드 조인 최적화

The syntax for expressing joins permits nested joins. The following discussion refers to the join syntax described in Section 13.2.13.2, “JOIN Clause”.

The syntax of *table_factor* is extended in comparison with the SQL Standard. The latter accepts only *table_reference*, not a list of them inside a pair of parentheses. This is a conservative extension if we consider each comma in a list of *table_reference* items as equivalent to an inner join. For example:

```
SELECT * FROM t1 LEFT JOIN (t2, t3, t4)
                ON (t2.a=t1.a AND t3.b=t1.b AND t4.c=t1.c)
```

Is equivalent to:

```
SELECT * FROM t1 LEFT JOIN (t2 CROSS JOIN t3 CROSS JOIN t4)
```

```
        ON (t2.a=t1.a AND t3.b=t1.b AND t4.c=t1.c)
```

In MySQL, `CROSS JOIN` is syntactically equivalent to `INNER JOIN`; they can replace each other. In standard SQL, they are not equivalent. `INNER JOIN` is used with an `ON` clause; `CROSS JOIN` is used otherwise.

In general, parentheses can be ignored in join expressions containing only inner join operations. Consider this join expression:

```
t1 LEFT JOIN (t2 LEFT JOIN t3 ON t2.b=t3.b OR t2.b IS NULL)
  ON t1.a=t2.a
```

After removing parentheses and grouping operations to the left, that join expression transforms into this expression:

```
(t1 LEFT JOIN t2 ON t1.a=t2.a) LEFT JOIN t3
   ON t2.b=t3.b OR t2.b IS NULL
```

Yet, the two expressions are not equivalent. To see this, suppose that the tables `t1`, `t2`, and `t3` have the following state:

- Table `t1` contains rows `(1)`, `(2)`
- Table `t2` contains row `(1,101)`
- Table `t3` contains row `(101)`

In this case, the first expression returns a result set including the rows `(1,1,101,101)`, `(2,NULL,NULL,NULL)`, whereas the second expression returns the rows `(1,1,101,101)`, `(2,NULL,NULL,101)`:

```
mysql> SELECT *
     FROM t1
        LEFT JOIN
        (t2 LEFT JOIN t3 ON t2.b=t3.b OR t2.b IS NULL)
        ON t1.a=t2.a;
+------+------+------+------+
| a    | a    | b    | b    |
+------+------+------+------+
|    1 |    1 |  101 |  101 |
|    2 | NULL | NULL | NULL |
+------+------+------+------+

mysql> SELECT *
     FROM (t1 LEFT JOIN t2 ON t1.a=t2.a)
        LEFT JOIN t3
        ON t2.b=t3.b OR t2.b IS NULL;
+------+------+------+------+
| a    | a    | b    | b    |
+------+------+------+------+
|    1 |    1 |  101 |  101 |
|    2 | NULL | NULL |  101 |
+------+------+------+------+
```

In the following example, an outer join operation is used together with an inner join operation:

```
t1 LEFT JOIN (t2, t3) ON t1.a=t2.a
```

That expression cannot be transformed into the following expression:

```
t1 LEFT JOIN t2 ON t1.a=t2.a, t3
```

For the given table states, the two expressions return different sets of rows:

```
mysql> SELECT *
     FROM t1 LEFT JOIN (t2, t3) ON t1.a=t2.a;
+------+------+------+------+
| a    | a    | b    | b    |
+------+------+------+------+
|    1 |    1 |  101 |  101 |
|    2 | NULL | NULL | NULL |
+------+------+------+------+

mysql> SELECT *
```

```
      FROM t1 LEFT JOIN t2 ON t1.a=t2.a, t3;
+------+------+------+------+
| a    | a    | b    | b    |
+------+------+------+------+
|    1 |    1 |  101 |  101 |
|    2 | NULL | NULL |  101 |
+------+------+------+------+
```

Therefore, if we omit parentheses in a join expression with outer join operators, we might change the result set for the original expression.

More exactly, we cannot ignore parentheses in the right operand of the left outer join operation and in the left operand of a right join operation. In other words, we cannot ignore parentheses for the inner table expressions of outer join operations. Parentheses for the other operand (operand for the outer table) can be ignored.

The following expression:

```
(t1,t2) LEFT JOIN t3 ON P(t2.b,t3.b)
```

Is equivalent to this expression for any tables `t1,t2,t3` and any condition `P` over attributes `t2.b` and `t3.b`:

```
t1, t2 LEFT JOIN t3 ON P(t2.b,t3.b)
```

Whenever the order of execution of join operations in a join expression *joined_table*) is not from left to right, we talk about nested joins. Consider the following queries:

```
SELECT * FROM t1 LEFT JOIN (t2 LEFT JOIN t3 ON t2.b=t3.b) ON t1.a=t2.a
  WHERE t1.a > 1

SELECT * FROM t1 LEFT JOIN (t2, t3) ON t1.a=t2.a
  WHERE (t2.b=t3.b OR t2.b IS NULL) AND t1.a > 1
```

Those queries are considered to contain these nested joins:

```
t2 LEFT JOIN t3 ON t2.b=t3.b
t2, t3
```

In the first query, the nested join is formed with a left join operation. In the second query, it is formed with an inner join operation.

In the first query, the parentheses can be omitted: The grammatical structure of the join expression dictates the same order of execution for join operations. For the second query, the parentheses cannot be omitted, although the join expression here can be interpreted unambiguously without them. In our extended syntax, the parentheses in `(t2, t3)` of the second query are required, although theoretically the query could be parsed without them: We still would have unambiguous syntactical structure for the query because `LEFT JOIN` and `ON` play the role of the left and right delimiters for the expression `(t2,t3)`.

The preceding examples demonstrate these points:

- For join expressions involving only inner joins (and not outer joins), parentheses can be removed and joins evaluated left to right. In fact, tables can be evaluated in any order.
- The same is not true, in general, for outer joins or for outer joins mixed with inner joins. Removal of parentheses may change the result.

Queries with nested outer joins are executed in the same pipeline manner as queries with inner joins. More exactly, a variation of the nested-loop join algorithm is exploited. Recall the algorithm by which the nested-loop join executes a query (see Section 8.2.1.7, “Nested-Loop Join Algorithms”). Suppose that a join query over 3 tables `T1,T2,T3` has this form:

```
SELECT * FROM T1 INNER JOIN T2 ON P1(T1,T2)
            INNER JOIN T3 ON P2(T2,T3)
  WHERE P(T1,T2,T3)
```

Here, `P1(T1,T2)` and `P2(T3,T3)` are some join conditions (on expressions), whereas `P(T1,T2,T3)` is a condition over columns of tables `T1,T2,T3`.

The nested-loop join algorithm would execute this query in the following manner:

```
FOR each row t1 in T1 {
```

```
  FOR each row t2 in T2 such that P1(t1,t2) {
    FOR each row t3 in T3 such that P2(t2,t3) {
      IF P(t1,t2,t3) {
        t:=t1||t2||t3; OUTPUT t;
      }
    }
  }
}
```

The notation `t1||t2||t3` indicates a row constructed by concatenating the columns of rows `t1`, `t2`, and `t3`. In some of the following examples, `NULL` where a table name appears means a row in which `NULL` is used for each column of that table. For example, `t1||t2||NULL` indicates a row constructed by concatenating the columns of rows `t1` and `t2`, and `NULL` for each column of `t3`. Such a row is said to be `NULL`-complemented.

Now consider a query with nested outer joins:

```
SELECT * FROM T1 LEFT JOIN
          (T2 LEFT JOIN T3 ON P2(T2,T3))
          ON P1(T1,T2)
 WHERE P(T1,T2,T3)
```

For this query, modify the nested-loop pattern to obtain:

```
FOR each row t1 in T1 {
  BOOL f1:=FALSE;
  FOR each row t2 in T2 such that P1(t1,t2) {
    BOOL f2:=FALSE;
    FOR each row t3 in T3 such that P2(t2,t3) {
      IF P(t1,t2,t3) {
        t:=t1||t2||t3; OUTPUT t;
      }
      f2=TRUE;
      f1=TRUE;
    }
    IF (!f2) {
      IF P(t1,t2,NULL) {
        t:=t1||t2||NULL; OUTPUT t;
      }
      f1=TRUE;
    }
  }
  IF (!f1) {
    IF P(t1,NULL,NULL) {
      t:=t1||NULL||NULL; OUTPUT t;
    }
  }
}
```

In general, for any nested loop for the first inner table in an outer join operation, a flag is introduced that is turned off before the loop and is checked after the loop. The flag is turned on when for the current row from the outer table a match from the table representing the inner operand is found. If at the end of the loop cycle the flag is still off, no match has been found for the current row of the outer table. In this case, the row is complemented by `NULL` values for the columns of the inner tables. The result row is passed to the final check for the output or into the next nested loop, but only if the row satisfies the join condition of all embedded outer joins.

In the example, the outer join table expressed by the following expression is embedded:

```
(T2 LEFT JOIN T3 ON P2(T2,T3))
```

For the query with inner joins, the optimizer could choose a different order of nested loops, such as this one:

```
FOR each row t3 in T3 {
  FOR each row t2 in T2 such that P2(t2,t3) {
    FOR each row t1 in T1 such that P1(t1,t2) {
      IF P(t1,t2,t3) {
        t:=t1||t2||t3; OUTPUT t;
      }
```

```
    }
  }
}
```

For queries with outer joins, the optimizer can choose only such an order where loops for outer tables precede loops for inner tables. Thus, for our query with outer joins, only one nesting order is possible. For the following query, the optimizer evaluates two different nestings. In both nestings, `T1` must be processed in the outer loop because it is used in an outer join. `T2` and `T3` are used in an inner join, so that join must be processed in the inner loop. However, because the join is an inner join, `T2` and `T3` can be processed in either order.

```
SELECT * T1 LEFT JOIN (T2,T3) ON P1(T1,T2) AND P2(T1,T3)
  WHERE P(T1,T2,T3)
```

One nesting evaluates `T2`, then `T3`:

```
FOR each row t1 in T1 {
  BOOL f1:=FALSE;
  FOR each row t2 in T2 such that P1(t1,t2) {
    FOR each row t3 in T3 such that P2(t1,t3) {
      IF P(t1,t2,t3) {
        t:=t1||t2||t3; OUTPUT t;
      }
      f1:=TRUE
    }
  }
  IF (!f1) {
    IF P(t1,NULL,NULL) {
      t:=t1||NULL||NULL; OUTPUT t;
    }
  }
}
```

The other nesting evaluates `T3`, then `T2`:

```
FOR each row t1 in T1 {
  BOOL f1:=FALSE;
  FOR each row t3 in T3 such that P2(t1,t3) {
    FOR each row t2 in T2 such that P1(t1,t2) {
      IF P(t1,t2,t3) {
        t:=t1||t2||t3; OUTPUT t;
      }
      f1:=TRUE
    }
  }
  IF (!f1) {
    IF P(t1,NULL,NULL) {
      t:=t1||NULL||NULL; OUTPUT t;
    }
  }
}
```

When discussing the nested-loop algorithm for inner joins, we omitted some details whose impact on the performance of query execution may be huge. We did not mention so-called “pushed-down” conditions. Suppose that our `WHERE` condition `P(T1,T2,T3)` can be represented by a conjunctive formula:

```
P(T1,T2,T2) = C1(T1) AND C2(T2) AND C3(T3).
```

In this case, MySQL actually uses the following nested-loop algorithm for the execution of the query with inner joins:

```
FOR each row t1 in T1 such that C1(t1) {
  FOR each row t2 in T2 such that P1(t1,t2) AND C2(t2)  {
    FOR each row t3 in T3 such that P2(t2,t3) AND C3(t3) {
      IF P(t1,t2,t3) {
        t:=t1||t2||t3; OUTPUT t;
      }
    }
  }
}
```

You see that each of the conjuncts C1(T1), C2(T2), C3(T3) are pushed out of the most inner loop to the most outer loop where it can be evaluated. If C1(T1) is a very restrictive condition, this condition pushdown may greatly reduce the number of rows from table T1 passed to the inner loops. As a result, the execution time for the query may improve immensely.

For a query with outer joins, the WHERE condition is to be checked only after it has been found that the current row from the outer table has a match in the inner tables. Thus, the optimization of pushing conditions out of the inner nested loops cannot be applied directly to queries with outer joins. Here we must introduce conditional pushed-down predicates guarded by the flags that are turned on when a match has been encountered.

Recall this example with outer joins:

```
P(T1,T2,T3)=C1(T1) AND C(T2) AND C3(T3)
```

For that example, the nested-loop algorithm using guarded pushed-down conditions looks like this:

```
FOR each row t1 in T1 such that C1(t1) {
  BOOL f1:=FALSE;
  FOR each row t2 in T2
      such that P1(t1,t2) AND (f1?C2(t2):TRUE) {
    BOOL f2:=FALSE;
    FOR each row t3 in T3
        such that P2(t2,t3) AND (f1&&f2?C3(t3):TRUE) {
      IF (f1&&f2?TRUE:(C2(t2) AND C3(t3))) {
        t:=t1||t2||t3; OUTPUT t;
      }
      f2=TRUE;
      f1=TRUE;
    }
    IF (!f2) {
      IF (f1?TRUE:C2(t2) && P(t1,t2,NULL)) {
        t:=t1||t2||NULL; OUTPUT t;
      }
      f1=TRUE;
    }
  }
  IF (!f1 && P(t1,NULL,NULL)) {
      t:=t1||NULL||NULL; OUTPUT t;
  }
}
```

In general, pushed-down predicates can be extracted from join conditions such as P1(T1,T2) and P(T2,T3). In this case, a pushed-down predicate is guarded also by a flag that prevents checking the predicate for the NULL-complemented row generated by the corresponding outer join operation.

Access by key from one inner table to another in the same nested join is prohibited if it is induced by a predicate from the WHERE condition.

8.2.1.9 아우터 조인 최적화

Outer joins include LEFT JOIN and RIGHT JOIN.

MySQL implements an *A* LEFT JOIN *B* *join_specification* as follows:

- Table *B* is set to depend on table *A* and all tables on which *A* depends.
- Table *A* is set to depend on all tables (except *B*) that are used in the LEFT JOIN condition.
- The LEFT JOIN condition is used to decide how to retrieve rows from table *B*. (In other words, any condition in the WHERE clause is not used.)
- All standard join optimizations are performed, with the exception that a table is always read after all tables on which it depends. If there is a circular dependency, an error occurs.
- All standard WHERE optimizations are performed.
- If there is a row in *A* that matches the WHERE clause, but there is no row in *B* that matches the ON condition, an extra *B* row is generated with all columns set to NULL.
- If you use LEFT JOIN to find rows that do not exist in some table and you have the following test: *col_name* IS NULL in the WHERE part, where *col_name* is a column that is declared as NOT NULL, MySQL stops searching for more rows (for a particular key combination) after it has found one row that matches the LEFT JOIN condition.

The RIGHT JOIN implementation is analogous to that of LEFT JOIN with the table roles reversed. Right joins are converted to equivalent left joins, as described in Section 8.2.1.10, “Outer Join Simplification”.

For a LEFT JOIN, if the WHERE condition is always false for the generated NULL row, the LEFT JOIN is changed to an

inner join. For example, the `WHERE` clause would be false in the following query if `t2.column1` were `NULL`:

```
SELECT * FROM t1 LEFT JOIN t2 ON (column1) WHERE t2.column2=5;
```

Therefore, it is safe to convert the query to an inner join:

```
SELECT * FROM t1, t2 WHERE t2.column2=5 AND t1.column1=t2.column1;
```

In MySQL 8.0.14 and later, trivial `WHERE` conditions arising from constant literal expressions are removed during preparation, rather than at a later stage in optimization, by which time joins have already been simplified. Earlier removal of trivial conditions allows the optimizer to convert outer joins to inner joins; this can result in improved plans for queries with outer joins containing trivial conditions in the `WHERE` clause, such as this one:

```
SELECT * FROM t1 LEFT JOIN t2 ON condition_1 WHERE condition_2 OR 0 = 1
```

The optimizer now sees during preparation that 0 = 1 is always false, making `OR 0 = 1` redundant, and removes it, leaving this:

```
SELECT * FROM t1 LEFT JOIN t2 ON condition_1 where condition_2
```

Now the optimizer can rewrite the query as an inner join, like this:

```
SELECT * FROM t1 JOIN t2 WHERE condition_1 AND condition_2
```

Now the optimizer can use table `t2` before table `t1` if doing so would result in a better query plan. To provide a hint about the table join order, use optimizer hints; see Section 8.9.3, “Optimizer Hints”. Alternatively, use `STRAIGHT_JOIN`; see Section 13.2.13, “SELECT Statement”. However, `STRAIGHT_JOIN` may prevent indexes from being used because it disables semijoin transformations; see Section 8.2.2.1, “Optimizing IN and EXISTS Subquery Predicates with Semijoin Transformations”.

8.2.1.10 아우터 조인 단순화

Table expressions in the `FROM` clause of a query are simplified in many cases.

At the parser stage, queries with right outer join operations are converted to equivalent queries containing only left join operations. In the general case, the conversion is performed such that this right join:

```
(T1, ...) RIGHT JOIN (T2, ...) ON P(T1, ..., T2, ...)
```

Becomes this equivalent left join:

```
(T2, ...) LEFT JOIN (T1, ...) ON P(T1, ..., T2, ...)
```

All inner join expressions of the form `T1 INNER JOIN T2 ON P(T1,T2)` are replaced by the list `T1,T2`, `P(T1,T2)` being joined as a conjunct to the `WHERE` condition (or to the join condition of the embedding join, if there is any).

When the optimizer evaluates plans for outer join operations, it takes into consideration only plans where, for each such operation, the outer tables are accessed before the inner tables. The optimizer choices are limited because only such plans enable outer joins to be executed using the nested-loop algorithm.

Consider a query of this form, where `R(T2)` greatly narrows the number of matching rows from table `T2`:

```
SELECT * T1 FROM T1
  LEFT JOIN T2 ON P1(T1,T2)
  WHERE P(T1,T2) AND R(T2)
```

If the query is executed as written, the optimizer has no choice but to access the less-restricted table `T1` before the more-restricted table `T2`, which may produce a very inefficient execution plan.

Instead, MySQL converts the query to a query with no outer join operation if the `WHERE` condition is null-rejected. (That is, it converts the outer join to an inner join.) A condition is said to be null-rejected for an outer join operation if it evaluates to `FALSE` or `UNKNOWN` for any `NULL`-complemented row generated for the operation.

Thus, for this outer join:

```
T1 LEFT JOIN T2 ON T1.A=T2.A
```

Conditions such as these are null-rejected because they cannot be true for any `NULL`-complemented row

(with `T2` columns set to `NULL`):

```
T2.B IS NOT NULL
T2.B > 3
T2.C <= T1.C
T2.B < 2 OR T2.C > 1
```

Conditions such as these are not null-rejected because they might be true for a `NULL`-complemented row:

```
T2.B IS NULL
T1.B < 3 OR T2.B IS NOT NULL
T1.B < 3 OR T2.B > 3
```

The general rules for checking whether a condition is null-rejected for an outer join operation are simple:

- It is of the form `A IS NOT NULL`, where `A` is an attribute of any of the inner tables
- It is a predicate containing a reference to an inner table that evaluates to `UNKNOWN` when one of its arguments is `NULL`
- It is a conjunction containing a null-rejected condition as a conjunct
- It is a disjunction of null-rejected conditions

A condition can be null-rejected for one outer join operation in a query and not null-rejected for another. In this query, the `WHERE` condition is null-rejected for the second outer join operation but is not null-rejected for the first one:

```
SELECT * FROM T1 LEFT JOIN T2 ON T2.A=T1.A
              LEFT JOIN T3 ON T3.B=T1.B
  WHERE T3.C > 0
```

If the `WHERE` condition is null-rejected for an outer join operation in a query, the outer join operation is replaced by an inner join operation.

For example, in the preceding query, the second outer join is null-rejected and can be replaced by an inner join:

```
SELECT * FROM T1 LEFT JOIN T2 ON T2.A=T1.A
              INNER JOIN T3 ON T3.B=T1.B
  WHERE T3.C > 0
```

For the original query, the optimizer evaluates only plans compatible with the single table-access order `T1,T2,T3`. For the rewritten query, it additionally considers the access order `T3,T1,T2`.

A conversion of one outer join operation may trigger a conversion of another. Thus, the query:

```
SELECT * FROM T1 LEFT JOIN T2 ON T2.A=T1.A
              LEFT JOIN T3 ON T3.B=T2.B
  WHERE T3.C > 0
```

Is first converted to the query:

```
SELECT * FROM T1 LEFT JOIN T2 ON T2.A=T1.A
              INNER JOIN T3 ON T3.B=T2.B
  WHERE T3.C > 0
```

Which is equivalent to the query:

```
SELECT * FROM (T1 LEFT JOIN T2 ON T2.A=T1.A), T3
  WHERE T3.C > 0 AND T3.B=T2.B
```

The remaining outer join operation can also be replaced by an inner join because the condition `T3.B=T2.B` is null-rejected. This results in a query with no outer joins at all:

```
SELECT * FROM (T1 INNER JOIN T2 ON T2.A=T1.A), T3
  WHERE T3.C > 0 AND T3.B=T2.B
```

Sometimes the optimizer succeeds in replacing an embedded outer join operation, but cannot convert the embedding outer join. The following query:

```
SELECT * FROM T1 LEFT JOIN
       (T2 LEFT JOIN T3 ON T3.B=T2.B)
       ON T2.A=T1.A
```

```
WHERE T3.C > 0
```

Is converted to:

```
SELECT * FROM T1 LEFT JOIN
        (T2 INNER JOIN T3 ON T3.B=T2.B)
        ON T2.A=T1.A
 WHERE T3.C > 0
```

That can be rewritten only to the form still containing the embedding outer join operation:

```
SELECT * FROM T1 LEFT JOIN
        (T2,T3)
        ON (T2.A=T1.A AND T3.B=T2.B)
 WHERE T3.C > 0
```

Any attempt to convert an embedded outer join operation in a query must take into account the join condition for the embedding outer join together with the `WHERE` condition. In this query, the `WHERE` condition is not null-rejected for the embedded outer join, but the join condition of the embedding outer join `T2.A=T1.A AND T3.C=T1.C` is null-rejected:

```
SELECT * FROM T1 LEFT JOIN
        (T2 LEFT JOIN T3 ON T3.B=T2.B)
        ON T2.A=T1.A AND T3.C=T1.C
 WHERE T3.D > 0 OR T1.D > 0
```

Consequently, the query can be converted to:

```
SELECT * FROM T1 LEFT JOIN
        (T2, T3)
        ON T2.A=T1.A AND T3.C=T1.C AND T3.B=T2.B
 WHERE T3.D > 0 OR T1.D > 0
```

8.2.1.11 멀티-레인지 읽기 최적화

Reading rows using a range scan on a secondary index can result in many random disk accesses to the base table when the table is large and not stored in the storage engine's cache. With the Disk-Sweep Multi-Range Read (MRR) optimization, MySQL tries to reduce the number of random disk access for range scans by first scanning the index only and collecting the keys for the relevant rows. Then the keys are sorted and finally the rows are retrieved from the base table using the order of the primary key. The motivation for Disk-sweep MRR is to reduce the number of random disk accesses and instead achieve a more sequential scan of the base table data.

The Multi-Range Read optimization provides these benefits:

- MRR enables data rows to be accessed sequentially rather than in random order, based on index tuples. The server obtains a set of index tuples that satisfy the query conditions, sorts them according to data row ID order, and uses the sorted tuples to retrieve data rows in order. This makes data access more efficient and less expensive.
- MRR enables batch processing of requests for key access for operations that require access to data rows through index tuples, such as range index scans and equi-joins that use an index for the join attribute. MRR iterates over a sequence of index ranges to obtain qualifying index tuples. As these results accumulate, they are used to access the corresponding data rows. It is not necessary to acquire all index tuples before starting to read data rows.

The MRR optimization is not supported with secondary indexes created on virtual generated columns. `InnoDB` supports secondary indexes on virtual generated columns.

The following scenarios illustrate when MRR optimization can be advantageous:

Scenario A: MRR can be used for `InnoDB` and `MyISAM` tables for index range scans and equi-join operations.

1. A portion of the index tuples are accumulated in a buffer.
2. The tuples in the buffer are sorted by their data row ID.
3. Data rows are accessed according to the sorted index tuple sequence.

Scenario B: MRR can be used for `NDB` tables for multiple-range index scans or when performing an equi-join by an attribute.

1. A portion of ranges, possibly single-key ranges, is accumulated in a buffer on the central node where the query is submitted.
2. The ranges are sent to the execution nodes that access data rows.
3. The accessed rows are packed into packages and sent back to the central node.

4. The received packages with data rows are placed in a buffer.
5. Data rows are read from the buffer.

When MRR is used, the `Extra` column in `EXPLAIN` output shows `Using MRR`.

`InnoDB` and `MyISAM` do not use MRR if full table rows need not be accessed to produce the query result. This is the case if results can be produced entirely on the basis on information in the index tuples (through a covering index); MRR provides no benefit.

Two `optimizer_switch` system variable flags provide an interface to the use of MRR optimization. The `mrr` flag controls whether MRR is enabled. If `mrr` is enabled (`on`), the `mrr_cost_based` flag controls whether the optimizer attempts to make a cost-based choice between using and not using MRR (`on`) or uses MRR whenever possible (`off`). By default, `mrr` is `on` and `mrr_cost_based` is `on`. See Section 8.9.2, “Switchable Optimizations”.

For MRR, a storage engine uses the value of the `read_rnd_buffer_size` system variable as a guideline for how much memory it can allocate for its buffer. The engine uses up to `read_rnd_buffer_size` bytes and determines the number of ranges to process in a single pass.

8.2.1.12 블록 네스티드-루프와 배치 키 액세스 조인

In MySQL, a Batched Key Access (BKA) Join algorithm is available that uses both index access to the joined table and a join buffer. The BKA algorithm supports inner join, outer join, and semijoin operations, including nested outer joins. Benefits of BKA include improved join performance due to more efficient table scanning. Also, the Block Nested-Loop (BNL) Join algorithm previously used only for inner joins is extended and can be employed for outer join and semijoin operations, including nested outer joins.

The following sections discuss the join buffer management that underlies the extension of the original BNL algorithm, the extended BNL algorithm, and the BKA algorithm. For information about semijoin strategies, see Section 8.2.2.1, “Optimizing IN and EXISTS Subquery Predicates with Semijoin Transformations”

- Join Buffer Management for Block Nested-Loop and Batched Key Access Algorithms
- Block Nested-Loop Algorithm for Outer Joins and Semijoins
- Batched Key Access Joins
- Optimizer Hints for Block Nested-Loop and Batched Key Access Algorithms

Join Buffer Management for Block Nested-Loop and Batched Key Access Algorithms

MySQL can employ join buffers to execute not only inner joins without index access to the inner table, but also outer joins and semijoins that appear after subquery flattening. Moreover, a join buffer can be effectively used when there is an index access to the inner table.

The join buffer management code slightly more efficiently utilizes join buffer space when storing the values of the interesting row columns: No additional bytes are allocated in buffers for a row column if its value is `NULL`, and the minimum number of bytes is allocated for any value of the `VARCHAR` type.

The code supports two types of buffers, regular and incremental. Suppose that join buffer `B1` is employed to join tables `t1` and `t2` and the result of this operation is joined with table `t3` using join buffer `B2`:

- A regular join buffer contains columns from each join operand. If `B2` is a regular join buffer, each row `r` put into `B2` is composed of the columns of a row `r1` from `B1` and the interesting columns of a matching row `r2` from table `t3`.
- An incremental join buffer contains only columns from rows of the table produced by the second join operand. That is, it is incremental to a row from the first operand buffer. If `B2` is an incremental join buffer, it contains the interesting columns of the row `r2` together with a link to the row `r1` from `B1`.

Incremental join buffers are always incremental relative to a join buffer from an earlier join operation, so the buffer from the first join operation is always a regular buffer. In the example just given, the buffer `B1` used to join tables `t1` and `t2` must be a regular buffer.

Each row of the incremental buffer used for a join operation contains only the interesting columns of a row from the table to be joined. These columns are augmented with a reference to the interesting columns of the matched row from the table produced by the first join operand. Several rows in the incremental buffer can refer to the same row `r` whose columns are stored in the previous join buffers insofar as all these rows match row `r`.

Incremental buffers enable less frequent copying of columns from buffers used for previous join operations. This provides a savings in buffer space because in the general case a row produced by the first join operand can be matched by several rows produced by the second join operand. It is unnecessary to make several copies of a row from the first operand. Incremental buffers also provide a savings in processing time due to the reduction in copying time.

In MySQL 8.0, the `block_nested_loop` flag of the `optimizer_switch` system variable works as follows:

- Prior to MySQL 8.0.20, it controls how the optimizer uses the Block Nested Loop join algorithm.
- In MySQL 8.0.18 and later, it also controls the use of hash joins (see Section 8.2.1.4, “Hash Join Optimization”).
- Beginning with MySQL 8.0.20, the flag controls hash joins only, and the block nested loop algorithm is no longer supported.

The `batched_key_access` flag controls how the optimizer uses the Batched Key Access join algorithms.

By default, `block_nested_loop` is `on` and `batched_key_access` is `off`. See Section 8.9.2, “Switchable Optimizations”. Optimizer hints may also be applied; see Optimizer Hints for Block Nested-Loop and Batched Key Access Algorithms

For information about semijoin strategies, see Section 8.2.2.1, “Optimizing IN and EXISTS Subquery Predicates with Semijoin Transformations”

Block Nested-Loop Algorithm for Outer Joins and Semijoins

The original implementation of the MySQL BNL algorithm was extended to support outer join and semijoin operations (and was later superseded by the hash join algorithm; see Section 8.2.1.4, “Hash Join Optimization”).

When these operations are executed with a join buffer, each row put into the buffer is supplied with a match flag.

If an outer join operation is executed using a join buffer, each row of the table produced by the second operand is checked for a match against each row in the join buffer. When a match is found, a new extended row is formed (the original row plus columns from the second operand) and sent for further extensions by the remaining join operations. In addition, the match flag of the matched row in the buffer is enabled. After all rows of the table to be joined have been examined, the join buffer is scanned. Each row from the buffer that does not have its match flag enabled is extended by `NULL` complements (`NULL` values for each column in the second operand) and sent for further extensions by the remaining join operations.

In MySQL 8.0, the `block_nested_loop` flag of the `optimizer_switch` system variable works as follows:

- Prior to MySQL 8.0.20, it controls how the optimizer uses the Block Nested Loop join algorithm.
- In MySQL 8.0.18 and later, it also controls the use of hash joins (see Section 8.2.1.4, “Hash Join Optimization”).
- Beginning with MySQL 8.0.20, the flag controls hash joins only, and the block nested loop algorithm is no longer supported.

See Section 8.9.2, “Switchable Optimizations”, for more information. Optimizer hints may also be applied; see Optimizer Hints for Block Nested-Loop and Batched Key Access Algorithms

In `EXPLAIN` output, use of BNL for a table is signified when the `Extra` value contains `Using join buffer (Block Nested Loop)` and the `type` value is `ALL`, `index`, or `range`.

For information about semijoin strategies, see Section 8.2.2.1, “Optimizing IN and EXISTS Subquery Predicates with Semijoin Transformations”

Batched Key Access Joins

MySQL implements a method of joining tables called the Batched Key Access (BKA) join algorithm. BKA can be applied when there is an index access to the table produced by the second join operand. Like the BNL join algorithm, the BKA join algorithm employs a join buffer to accumulate the interesting columns of the rows produced by the first operand of the join operation. Then the BKA algorithm builds keys to access the table to be joined for all rows in the buffer and submits these keys in a batch to the database engine for index lookups. The keys are submitted to the engine through the Multi-Range Read (MRR) interface (see Section 8.2.1.11, “Multi-Range Read Optimization”). After submission of the keys, the MRR engine functions perform lookups in the index in an optimal way, fetching the rows of the joined table found by these keys, and starts feeding the BKA join algorithm with matching rows. Each matching row is coupled with a reference to a row in the join buffer.

When BKA is used, the value of `join_buffer_size` defines how large the batch of keys is in each request to the storage engine. The larger the buffer, the more sequential access is made to the right hand table of a join operation, which can significantly improve performance.

For BKA to be used, the `batched_key_access` flag of the `optimizer_switch` system variable must be set to `on`. BKA uses MRR, so the `mrr` flag must also be `on`. Currently, the cost estimation for MRR is too pessimistic. Hence, it is also necessary for `mrr_cost_based` to be `off` for BKA to be used. The following setting enables BKA:

```
mysql> SET optimizer_switch='mrr=on,mrr_cost_based=off,batched_key_access=on';
```

There are two scenarios by which MRR functions execute:

- The first scenario is used for conventional disk-based storage engines such as InnoDB and MyISAM. For these engines, usually the keys for all rows from the join buffer are submitted to the MRR interface at once. Engine-specific MRR functions perform index lookups for the submitted keys, get row IDs (or primary keys) from them, and then fetch rows for all these selected row IDs one by one by request from BKA algorithm. Every row is returned with an association reference that enables access to the matched row in the join buffer. The rows are fetched by the MRR functions in an optimal way: They are fetched in the row ID (primary key) order. This improves performance because reads are in disk order rather than random order.
- The second scenario is used for remote storage engines such as NDB. A package of keys for a portion of rows from the join buffer, together with their associations, is sent by a MySQL Server (SQL node) to MySQL Cluster data nodes. In return, the SQL node receives a package (or several packages) of matching rows coupled with corresponding associations. The BKA join algorithm takes these rows and builds new joined rows. Then a new set of keys is sent to the data nodes and the rows from the returned packages are used to build new joined rows. The process continues until the last keys from the join buffer are sent to the data nodes, and the SQL node has received and joined all rows matching these keys. This improves performance because fewer key-bearing packages sent by the SQL node to the data nodes means fewer round trips between it and the data nodes to perform the join operation.

With the first scenario, a portion of the join buffer is reserved to store row IDs (primary keys) selected by index lookups and passed as a parameter to the MRR functions.

There is no special buffer to store keys built for rows from the join buffer. Instead, a function that builds the key for the next row in the buffer is passed as a parameter to the MRR functions.

In EXPLAIN output, use of BKA for a table is signified when the Extra value contains Using join buffer (Batched Key Access) and the type value is ref or eq_ref.

Optimizer Hints for Block Nested-Loop and Batched Key Access Algorithms

In addition to using the optimizer_switch system variable to control optimizer use of the BNL and BKA algorithms session-wide, MySQL supports optimizer hints to influence the optimizer on a per-statement basis. See Section 8.9.3, “Optimizer Hints”.

To use a BNL or BKA hint to enable join buffering for any inner table of an outer join, join buffering must be enabled for all inner tables of the outer join.

8.2.1.13 조건 필터링

In join processing, prefix rows are those rows passed from one table in a join to the next. In general, the optimizer attempts to put tables with low prefix counts early in the join order to keep the number of row combinations from increasing rapidly. To the extent that the optimizer can use information about conditions on rows selected from one table and passed to the next, the more accurately it can compute row estimates and choose the best execution plan.

Without condition filtering, the prefix row count for a table is based on the estimated number of rows selected by the WHERE clause according to whichever access method the optimizer chooses. Condition filtering enables the optimizer to use other relevant conditions in the WHERE clause not taken into account by the access method, and thus improve its prefix row count estimates. For example, even though there might be an index-based access method that can be used to select rows from the current table in a join, there might also be additional conditions for the table in the WHERE clause that can filter (further restrict) the estimate for qualifying rows passed to the next table.

A condition contributes to the filtering estimate only if:

- It refers to the current table.
- It depends on a constant value or values from earlier tables in the join sequence.
- It was not already taken into account by the access method.

In EXPLAIN output, the rows column indicates the row estimate for the chosen access method, and the filtered column reflects the effect of condition filtering. filtered values are expressed as percentages. The maximum value is 100, which means no filtering of rows occurred. Values decreasing from 100 indicate increasing amounts of filtering.

The prefix row count (the number of rows estimated to be passed from the current table in a join to the next) is the product of the rows and filtered values. That is, the prefix row count is the estimated row count, reduced by the estimated filtering effect. For example, if rows is 1000 and filtered is 20%, condition filtering reduces the estimated row count of 1000 to a prefix row count of 1000 × 20% = 1000 × .2 = 200.

Consider the following query:

```
SELECT *
  FROM employee JOIN department ON employee.dept_no = department.dept_no
  WHERE employee.first_name = 'John'
  AND employee.hire_date BETWEEN '2018-01-01' AND '2018-06-01';
```

Suppose that the data set has these characteristics:

- The `employee` table has 1024 rows.
- The `department` table has 12 rows.
- Both tables have an index on `dept_no`.
- The `employee` table has an index on `first_name`.
- 8 rows satisfy this condition on `employee.first_name`:

  ```
  employee.first_name = 'John'
  ```

- 150 rows satisfy this condition on `employee.hire_date`:

  ```
  employee.hire_date BETWEEN '2018-01-01' AND '2018-06-01'
  ```

- 1 row satisfies both conditions:

  ```
  employee.first_name = 'John'
  AND employee.hire_date BETWEEN '2018-01-01' AND '2018-06-01'
  ```

Without condition filtering, `EXPLAIN` produces output like this:

```
+----+------------+--------+------------------+---------+---------+------+----------+
| id | table      | type   | possible_keys    | key     | ref     | rows | filtered |
+----+------------+--------+------------------+---------+---------+------+----------+
| 1  | employee   | ref    | name,h_date,dept | name    | const   | 8    | 100.00   |
| 1  | department | eq_ref | PRIMARY          | PRIMARY | dept_no | 1    | 100.00   |
+----+------------+--------+------------------+---------+---------+------+----------+
```

For `employee`, the access method on the `name` index picks up the 8 rows that match a name of `'John'`. No filtering is done (`filtered` is 100%), so all rows are prefix rows for the next table: The prefix row count is `rows` × `filtered` = 8 × 100% = 8.

With condition filtering, the optimizer additionally takes into account conditions from the `WHERE` clause not taken into account by the access method. In this case, the optimizer uses heuristics to estimate a filtering effect of 16.31% for the `BETWEEN` condition on `employee.hire_date`. As a result, `EXPLAIN` produces output like this:

```
+----+------------+--------+------------------+---------+---------+------+----------+
| id | table      | type   | possible_keys    | key     | ref     | rows | filtered |
+----+------------+--------+------------------+---------+---------+------+----------+
| 1  | employee   | ref    | name,h_date,dept | name    | const   | 8    | 16.31    |
| 1  | department | eq_ref | PRIMARY          | PRIMARY | dept_no | 1    | 100.00   |
+----+------------+--------+------------------+---------+---------+------+----------+
```

Now the prefix row count is `rows` × `filtered` = 8 × 16.31% = 1.3, which more closely reflects actual data set.

Normally, the optimizer does not calculate the condition filtering effect (prefix row count reduction) for the last joined table because there is no next table to pass rows to. An exception occurs for `EXPLAIN`: To provide more information, the filtering effect is calculated for all joined tables, including the last one.

To control whether the optimizer considers additional filtering conditions, use the `condition_fanout_filter` flag of the `optimizer_switch` system variable (see Section 8.9.2, “Switchable Optimizations”). This flag is enabled by default but can be disabled to suppress condition filtering (for example, if a particular query is found to yield better performance without it).

If the optimizer overestimates the effect of condition filtering, performance may be worse than if condition filtering is not used. In such cases, these techniques may help:

- If a column is not indexed, index it so that the optimizer has some information about the distribution of column values and can improve its row estimates.
- Similarly, if no column histogram information is available, generate a histogram (see Section 8.9.6, “Optimizer Statistics”).
- Change the join order. Ways to accomplish this include join-order optimizer hints (see Section 8.9.3, “Optimizer Hints”), `STRAIGHT_JOIN` immediately following the `SELECT`, and the `STRAIGHT_JOIN` join operator.
- Disable condition filtering for the session:

  ```
  SET optimizer_switch = 'condition_fanout_filter=off';
  ```

  Or, for a given query, using an optimizer hint:

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

8.2.1.14 상수-폴딩 최적화

Comparisons between constants and column values in which the constant value is out of range or of the wrong type with respect to the column type are now handled once during query optimization rather row-by-row than during execution. The comparisons that can be treated in this manner are `>`, `>=`, `<`, `<=`, `<>` / `!=`, `=`, and `<=>`.

Consider the table created by the following statement:

```
CREATE TABLE t (c TINYINT UNSIGNED NOT NULL);
```

The `WHERE` condition in the query `SELECT * FROM t WHERE c < 256` contains the integral constant 256 which is out of range for a `TINYINT UNSIGNED` column. Previously, this was handled by treating both operands as the larger type, but now, since any allowed value for `c` is less than the constant, the `WHERE` expression can instead be folded as `WHERE 1`, so that the query is rewritten as `SELECT * FROM t WHERE 1`.

This makes it possible for the optimizer to remove the `WHERE` expression altogether. If the column `c` were nullable (that is, defined only as `TINYINT UNSIGNED`) the query would be rewritten like this:

```
SELECT * FROM t WHERE ti IS NOT NULL
```

Folding is performed for constants compared to supported MySQL column types as follows:

- **Integer column type.** Integer types are compared with constants of the following types as described here:
  - **Integer value.** If the constant is out of range for the column type, the comparison is folded to `1` or `IS NOT NULL`, as already shown.
    If the constant is a range boundary, the comparison is folded to `=`. For example (using the same table as already defined):

```
mysql> EXPLAIN SELECT * FROM t WHERE c >= 255;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 5
     filtered: 20.00
        Extra: Using where
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS;
*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: /* select#1 */ select `test`.`t`.`ti` AS `ti` from `test`.`t` where (`test`.`t`.`ti` = 255)
1 row in set (0.00 sec)
```

  - **Floating- or fixed-point value.** If the constant is one of the decimal types (such as `DECIMAL`, `REAL`, `DOUBLE`, or `FLOAT`) and has a nonzero decimal portion, it cannot be equal; fold accordingly. For other comparisons, round up or down to an integer value according to the sign, then perform a range check and handle as already described for integer-integer comparisons.
    A `REAL` value that is too small to be represented as `DECIMAL` is rounded to .01 or -.01 depending on the sign, then handled as a `DECIMAL`.
  - **String types.** Try to interpret the string value as an integer type, then handle the comparison as between integer values. If this fails, attempt to handle the value as a `REAL`.

- **DECIMAL or REAL column.** Decimal types are compared with constants of the following types as described here:
  - **Integer value.** Perform a range check against the column value's integer part. If no folding results, convert the constant to `DECIMAL` with the same number of decimal places as the column value, then check it as a `DECIMAL` (see next).
  - **DECIMAL or REAL value.** Check for overflow (that is, whether the constant has more digits in its integer part than allowed for the column's decimal type). If so, fold.
    If the constant has more significant fractional digits than column's type, truncate the constant. If the comparison operator is `=` or `<>`, fold. If the operator is `>=` or `<=`, adjust the operator due to truncation. For example, if column's type is `DECIMAL(3,1)`, `SELECT * FROM t WHERE f >= 10.13` becomes `SELECT * FROM t WHERE f > 10.1`.
    If the constant has fewer decimal digits than the column's type, convert it to a constant with same number of digits. For underflow of a `REAL` value (that is, too few fractional digits to represent it), convert the constant to decimal 0.
  - **String value.** If the value can be interpreted as an integer type, handle it as such. Otherwise, try to handle it as `REAL`.

- **FLOAT or DOUBLE column.** `FLOAT(m,n)` or `DOUBLE(m,n)` values compared with constants are handled as follows:
  If the value overflows the range of the column, fold.

If the value has more than *n* decimals, truncate, compensating during folding. For = and <> comparisons, fold to TRUE, FALSE, or IS [NOT] NULL as described previously; for other operators, adjust the operator. If the value has more than m integer digits, fold.

Limitations. This optimization cannot be used in the following cases:

1. With comparisons using BETWEEN or IN.
2. With BIT columns or columns using date or time types.
3. During the preparation phase for a prepared statement, although it can be applied during the optimization phase when the prepared statement is actually executed. This due to the fact that, during statement preparation, the value of the constant is not yet known.

8.2.1.15 IS NULL 최적화

MySQL can perform the same optimization on *col_name* IS NULL that it can use for *col_name* = *constant_value*. For example, MySQL can use indexes and ranges to search for NULL with IS NULL.

Examples:

```
SELECT * FROM tbl_name WHERE key_col IS NULL;

SELECT * FROM tbl_name WHERE key_col <=> NULL;

SELECT * FROM tbl_name
  WHERE key_col=const1 OR key_col=const2 OR key_col IS NULL;
```

If a WHERE clause includes a *col_name* IS NULL condition for a column that is declared as NOT NULL, that expression is optimized away. This optimization does not occur in cases when the column might produce NULL anyway (for example, if it comes from a table on the right side of a LEFT JOIN).

MySQL can also optimize the combination *col_name* = *expr* OR *col_name* IS NULL, a form that is common in resolved subqueries. EXPLAIN shows ref_or_null when this optimization is used.

This optimization can handle one IS NULL for any key part.

Some examples of queries that are optimized, assuming that there is an index on columns a and b of table t2:

```
SELECT * FROM t1 WHERE t1.a=expr OR t1.a IS NULL;

SELECT * FROM t1, t2 WHERE t1.a=t2.a OR t2.a IS NULL;

SELECT * FROM t1, t2
  WHERE (t1.a=t2.a OR t2.a IS NULL) AND t2.b=t1.b;

SELECT * FROM t1, t2
  WHERE t1.a=t2.a AND (t2.b=t1.b OR t2.b IS NULL);

SELECT * FROM t1, t2
  WHERE (t1.a=t2.a AND t2.a IS NULL AND ...)
  OR (t1.a=t2.a AND t2.a IS NULL AND ...);
```

ref_or_null works by first doing a read on the reference key, and then a separate search for rows with a NULL key value.

The optimization can handle only one IS NULL level. In the following query, MySQL uses key lookups only on the expression (t1.a=t2.a AND t2.a IS NULL) and is not able to use the key part on b:

```
SELECT * FROM t1, t2
  WHERE (t1.a=t2.a AND t2.a IS NULL)
  OR (t1.b=t2.b AND t2.b IS NULL);
```

8.2.1.16 ORDER BY 최적화

This section describes when MySQL can use an index to satisfy an ORDER BY clause, the filesort operation used when an index cannot be used, and execution plan information available from the optimizer about ORDER BY.

An ORDER BY with and without LIMIT may return rows in different orders, as discussed in Section 8.2.1.19, “LIMIT Query Optimization”.

- Use of Indexes to Satisfy ORDER BY

- Use of filesort to Satisfy ORDER BY
- Influencing ORDER BY Optimization
- ORDER BY Execution Plan Information Available

Use of Indexes to Satisfy ORDER BY

In some cases, MySQL may use an index to satisfy an `ORDER BY` clause and avoid the extra sorting involved in performing a `filesort` operation.

The index may also be used even if the `ORDER BY` does not match the index exactly, as long as all unused portions of the index and all extra `ORDER BY` columns are constants in the `WHERE` clause. If the index does not contain all columns accessed by the query, the index is used only if index access is cheaper than other access methods.

Assuming that there is an index on `(key_part1, key_part2)`, the following queries may use the index to resolve the `ORDER BY` part. Whether the optimizer actually does so depends on whether reading the index is more efficient than a table scan if columns not in the index must also be read.

- In this query, the index on `(key_part1, key_part2)` enables the optimizer to avoid sorting:

  ```
  SELECT * FROM t1
    ORDER BY key_part1, key_part2;
  ```

  However, the query uses `SELECT *`, which may select more columns than `key_part1` and `key_part2`. In that case, scanning an entire index and looking up table rows to find columns not in the index may be more expensive than scanning the table and sorting the results. If so, the optimizer probably does not use the index. If `SELECT *` selects only the index columns, the index is used and sorting avoided.
  If `t1` is an `InnoDB` table, the table primary key is implicitly part of the index, and the index can be used to resolve the `ORDER BY` for this query:

  ```
  SELECT pk, key_part1, key_part2 FROM t1
    ORDER BY key_part1, key_part2;
  ```

- In this query, `key_part1` is constant, so all rows accessed through the index are in `key_part2` order, and an index on `(key_part1, key_part2)` avoids sorting if the `WHERE` clause is selective enough to make an index range scan cheaper than a table scan:

  ```
  SELECT * FROM t1
    WHERE key_part1 = constant
    ORDER BY key_part2;
  ```

- In the next two queries, whether the index is used is similar to the same queries without `DESC` shown previously:

  ```
  SELECT * FROM t1
    ORDER BY key_part1 DESC, key_part2 DESC;

  SELECT * FROM t1
    WHERE key_part1 = constant
    ORDER BY key_part2 DESC;
  ```

- Two columns in an `ORDER BY` can sort in the same direction (both `ASC`, or both `DESC`) or in opposite directions (one `ASC`, one `DESC`). A condition for index use is that the index must have the same homogeneity, but need not have the same actual direction.
  If a query mixes `ASC` and `DESC`, the optimizer can use an index on the columns if the index also uses corresponding mixed ascending and descending columns:

  ```
  SELECT * FROM t1
    ORDER BY key_part1 DESC, key_part2 ASC;
  ```

  The optimizer can use an index on `(key_part1, key_part2)` if `key_part1` is descending and `key_part2` is ascending. It can also use an index on those columns (with a backward scan) if `key_part1` is ascending and `key_part2` is descending. See Section 8.3.13, “Descending Indexes”.
- In the next two queries, `key_part1` is compared to a constant. The index is used if the `WHERE` clause is selective enough to make an index range scan cheaper than a table scan:

  ```
  SELECT * FROM t1
    WHERE key_part1 > constant
    ORDER BY key_part1 ASC;

  SELECT * FROM t1
    WHERE key_part1 < constant
    ORDER BY key_part1 DESC;
  ```

- In the next query, the `ORDER BY` does not name `key_part1`, but all rows selected have a

constant *key_part1* value, so the index can still be used:

```
SELECT * FROM t1
  WHERE key_part1 = constant1 AND key_part2 > constant2
  ORDER BY key_part2;
```

In some cases, MySQL *cannot* use indexes to resolve the ORDER BY, although it may still use indexes to find the rows that match the WHERE clause. Examples:

- The query uses ORDER BY on different indexes:

```
SELECT * FROM t1 ORDER BY key1, key2;
```

- The query uses ORDER BY on nonconsecutive parts of an index:

```
SELECT * FROM t1 WHERE key2=constant ORDER BY key1_part1, key1_part3;
```

- The index used to fetch the rows differs from the one used in the ORDER BY:

```
SELECT * FROM t1 WHERE key2=constant ORDER BY key1;
```

- The query uses ORDER BY with an expression that includes terms other than the index column name:

```
SELECT * FROM t1 ORDER BY ABS(key);
SELECT * FROM t1 ORDER BY -key;
```

- The query joins many tables, and the columns in the ORDER BY are not all from the first nonconstant table that is used to retrieve rows. (This is the first table in the EXPLAIN output that does not have a const join type.)
- The query has different ORDER BY and GROUP BY expressions.
- There is an index on only a prefix of a column named in the ORDER BY clause. In this case, the index cannot be used to fully resolve the sort order. For example, if only the first 10 bytes of a CHAR(20) column are indexed, the index cannot distinguish values past the 10th byte and a filesort is needed.
- The index does not store rows in order. For example, this is true for a HASH index in a MEMORY table.

Availability of an index for sorting may be affected by the use of column aliases. Suppose that the column t1.a is indexed. In this statement, the name of the column in the select list is a. It refers to t1.a, as does the reference to a in the ORDER BY, so the index on t1.a can be used:

```
SELECT a FROM t1 ORDER BY a;
```

In this statement, the name of the column in the select list is also a, but it is the alias name. It refers to ABS(a), as does the reference to a in the ORDER BY, so the index on t1.a cannot be used:

```
SELECT ABS(a) AS a FROM t1 ORDER BY a;
```

In the following statement, the ORDER BY refers to a name that is not the name of a column in the select list. But there is a column in t1 named a, so the ORDER BY refers to t1.a and the index on t1.a can be used. (The resulting sort order may be completely different from the order for ABS(a), of course.)

```
SELECT ABS(a) AS b FROM t1 ORDER BY a;
```

Previously (MySQL 5.7 and lower), GROUP BY sorted implicitly under certain conditions. In MySQL 8.0, that no longer occurs, so specifying ORDER BY NULL at the end to suppress implicit sorting (as was done previously) is no longer necessary. However, query results may differ from previous MySQL versions. To produce a given sort order, provide an ORDER BY clause.

Use of filesort to Satisfy ORDER BY

If an index cannot be used to satisfy an ORDER BY clause, MySQL performs a filesort operation that reads table rows and sorts them. A filesort constitutes an extra sorting phase in query execution.

To obtain memory for filesort operations, as of MySQL 8.0.12, the optimizer allocates memory buffers incrementally as needed, up to the size indicated by the sort_buffer_size system variable, rather than allocating a fixed amount of sort_buffer_size bytes up front, as was done prior to MySQL 8.0.12. This enables users to set sort_buffer_size to larger values to speed up larger sorts, without concern for excessive memory use for small sorts. (This benefit may not occur for multiple concurrent sorts on Windows, which has a weak multithreaded malloc.)

A filesort operation uses temporary disk files as necessary if the result set is too large to fit in memory. Some types of queries are particularly suited to completely in-memory filesort operations. For example, the optimizer can use filesort to efficiently handle in memory, without temporary files, the ORDER BY operation for queries (and subqueries) of the following form:

```
SELECT ... FROM single_table ... ORDER BY non_index_column [DESC] LIMIT [M,]N;
```

Such queries are common in web applications that display only a few rows from a larger result set. Examples:

```
SELECT col1, ... FROM t1 ... ORDER BY name LIMIT 10;
SELECT col1, ... FROM t1 ... ORDER BY RAND() LIMIT 15;
```

Influencing ORDER BY Optimization

For slow `ORDER BY` queries for which `filesort` is not used, try lowering the `max_length_for_sort_data` system variable to a value that is appropriate to trigger a `filesort`. (A symptom of setting the value of this variable too high is a combination of high disk activity and low CPU activity.) This technique applies only before MySQL 8.0.20. As of 8.0.20, `max_length_for_sort_data` is deprecated due to optimizer changes that make it obsolete and of no effect.

To increase `ORDER BY` speed, check whether you can get MySQL to use indexes rather than an extra sorting phase. If this is not possible, try the following strategies:

- Increase the `sort_buffer_size` variable value. Ideally, the value should be large enough for the entire result set to fit in the sort buffer (to avoid writes to disk and merge passes).
  Take into account that the size of column values stored in the sort buffer is affected by the `max_sort_length` system variable value. For example, if tuples store values of long string columns and you increase the value of `max_sort_length`, the size of sort buffer tuples increases as well and may require you to increase `sort_buffer_size`.
  To monitor the number of merge passes (to merge temporary files), check the `Sort_merge_passes` status variable.
- Increase the `read_rnd_buffer_size` variable value so that more rows are read at a time.
- Change the `tmpdir` system variable to point to a dedicated file system with large amounts of free space. The variable value can list several paths that are used in round-robin fashion; you can use this feature to spread the load across several directories. Separate the paths by colon characters (`:`) on Unix and semicolon characters (`;`) on Windows. The paths should name directories in file systems located on different *physical* disks, not different partitions on the same disk.

ORDER BY Execution Plan Information Available

With `EXPLAIN` (see Section 8.8.1, “Optimizing Queries with EXPLAIN”), you can check whether MySQL can use indexes to resolve an `ORDER BY` clause:

- If the `Extra` column of `EXPLAIN` output does not contain `Using filesort`, the index is used and a `filesort` is not performed.
- If the `Extra` column of `EXPLAIN` output contains `Using filesort`, the index is not used and a `filesort` is performed.

In addition, if a `filesort` is performed, optimizer trace output includes a `filesort_summary` block. For example:

```
"filesort_summary": {
  "rows": 100,
  "examined_rows": 100,
  "number_of_tmp_files": 0,
  "peak_memory_used": 25192,
  "sort_mode": "<sort_key, packed_additional_fields>"
}
```

`peak_memory_used` indicates the maximum memory used at any one time during the sort. This is a value up to but not necessarily as large as the value of the `sort_buffer_size` system variable. Prior to MySQL 8.0.12, the output shows `sort_buffer_size` instead, indicating the value of `sort_buffer_size`. (Prior to MySQL 8.0.12, the optimizer always allocates `sort_buffer_size` bytes for the sort buffer. As of 8.0.12, the optimizer allocates sort-buffer memory incrementally, beginning with a small amount and adding more as necessary, up to `sort_buffer_size` bytes.)

The `sort_mode` value provides information about the contents of tuples in the sort buffer:

- `<sort_key, rowid>`: This indicates that sort buffer tuples are pairs that contain the sort key value and row ID of the original table row. Tuples are sorted by sort key value and the row ID is used to read the row from the table.
- `<sort_key, additional_fields>`: This indicates that sort buffer tuples contain the sort key value and columns referenced by the query. Tuples are sorted by sort key value and column values are read directly from the tuple.
- `<sort_key, packed_additional_fields>`: Like the previous variant, but the additional columns are packed tightly together instead of using a fixed-length encoding.

`EXPLAIN` does not distinguish whether the optimizer does or does not perform a `filesort` in memory. Use of an in-memory `filesort` can be seen in optimizer trace output. Look for `filesort_priority_queue_optimization`. For information about the

optimizer trace, see MySQL Internals: Tracing the Optimizer.

8.2.1.17 GROUP BY 최적화

The most general way to satisfy a GROUP BY clause is to scan the whole table and create a new temporary table where all rows from each group are consecutive, and then use this temporary table to discover groups and apply aggregate functions (if any). In some cases, MySQL is able to do much better than that and avoid creation of temporary tables by using index access.

The most important preconditions for using indexes for GROUP BY are that all GROUP BY columns reference attributes from the same index, and that the index stores its keys in order (as is true, for example, for a BTREE index, but not for a HASH index). Whether use of temporary tables can be replaced by index access also depends on which parts of an index are used in a query, the conditions specified for these parts, and the selected aggregate functions.

There are two ways to execute a GROUP BY query through index access, as detailed in the following sections. The first method applies the grouping operation together with all range predicates (if any). The second method first performs a range scan, and then groups the resulting tuples.

- Loose Index Scan
- Tight Index Scan

Loose Index Scan can also be used in the absence of GROUP BY under some conditions. See Skip Scan Range Access Method.

Loose Index Scan

The most efficient way to process GROUP BY is when an index is used to directly retrieve the grouping columns. With this access method, MySQL uses the property of some index types that the keys are ordered (for example, BTREE). This property enables use of lookup groups in an index without having to consider all keys in the index that satisfy all WHERE conditions. This access method considers only a fraction of the keys in an index, so it is called a Loose Index Scan. When there is no WHERE clause, a Loose Index Scan reads as many keys as the number of groups, which may be a much smaller number than that of all keys. If the WHERE clause contains range predicates (see the discussion of the range join type in Section 8.8.1, “Optimizing Queries with EXPLAIN”), a Loose Index Scan looks up the first key of each group that satisfies the range conditions, and again reads the smallest possible number of keys. This is possible under the following conditions:

- The query is over a single table.
- The GROUP BY names only columns that form a leftmost prefix of the index and no other columns. (If, instead of GROUP BY, the query has a DISTINCT clause, all distinct attributes refer to columns that form a leftmost prefix of the index.) For example, if a table t1 has an index on (c1,c2,c3), Loose Index Scan is applicable if the query has GROUP BY c1, c2. It is not applicable if the query has GROUP BY c2, c3 (the columns are not a leftmost prefix) or GROUP BY c1, c2, c4 (c4 is not in the index).
- The only aggregate functions used in the select list (if any) are MIN() and MAX(), and all of them refer to the same column. The column must be in the index and must immediately follow the columns in the GROUP BY.
- Any other parts of the index than those from the GROUP BY referenced in the query must be constants (that is, they must be referenced in equalities with constants), except for the argument of MIN() or MAX() functions.
- For columns in the index, full column values must be indexed, not just a prefix. For example, with c1 VARCHAR(20), INDEX (c1(10)), the index uses only a prefix of c1 values and cannot be used for Loose Index Scan.

If Loose Index Scan is applicable to a query, the EXPLAIN output shows Using index for group-by in the Extra column.

Assume that there is an index idx(c1,c2,c3) on table t1(c1,c2,c3,c4). The Loose Index Scan access method can be used for the following queries:

```
SELECT c1, c2 FROM t1 GROUP BY c1, c2;
SELECT DISTINCT c1, c2 FROM t1;
SELECT c1, MIN(c2) FROM t1 GROUP BY c1;
SELECT c1, c2 FROM t1 WHERE c1 < const GROUP BY c1, c2;
SELECT MAX(c3), MIN(c3), c1, c2 FROM t1 WHERE c2 > const GROUP BY c1, c2;
SELECT c2 FROM t1 WHERE c1 < const GROUP BY c1, c2;
SELECT c1, c2 FROM t1 WHERE c3 = const GROUP BY c1, c2;
```

The following queries cannot be executed with this quick select method, for the reasons given:

- There are aggregate functions other than MIN() or MAX():
  ```
  SELECT c1, SUM(c2) FROM t1 GROUP BY c1;
  ```
- The columns in the GROUP BY clause do not form a leftmost prefix of the index:

```
SELECT c1, c2 FROM t1 GROUP BY c2, c3;
```

- The query refers to a part of a key that comes after the `GROUP BY` part, and for which there is no equality with a constant:

  ```
  SELECT c1, c3 FROM t1 GROUP BY c1, c2;
  ```

  Were the query to include `WHERE c3 = const`, Loose Index Scan could be used.

The Loose Index Scan access method can be applied to other forms of aggregate function references in the select list, in addition to the `MIN()` and `MAX()` references already supported:

- `AVG(DISTINCT)`, `SUM(DISTINCT)`, and `COUNT(DISTINCT)` are supported. `AVG(DISTINCT)` and `SUM(DISTINCT)` take a single argument. `COUNT(DISTINCT)` can have more than one column argument.
- There must be no `GROUP BY` or `DISTINCT` clause in the query.
- The Loose Index Scan limitations described previously still apply.

Assume that there is an index `idx(c1,c2,c3)` on table `t1(c1,c2,c3,c4)`. The Loose Index Scan access method can be used for the following queries:

```
SELECT COUNT(DISTINCT c1), SUM(DISTINCT c1) FROM t1;

SELECT COUNT(DISTINCT c1, c2), COUNT(DISTINCT c2, c1) FROM t1;
```

Tight Index Scan

A Tight Index Scan may be either a full index scan or a range index scan, depending on the query conditions.

When the conditions for a Loose Index Scan are not met, it still may be possible to avoid creation of temporary tables for `GROUP BY` queries. If there are range conditions in the `WHERE` clause, this method reads only the keys that satisfy these conditions. Otherwise, it performs an index scan. Because this method reads all keys in each range defined by the `WHERE` clause, or scans the whole index if there are no range conditions, it is called aTight Index Scan. With a Tight Index Scan, the grouping operation is performed only after all keys that satisfy the range conditions have been found.

For this method to work, it is sufficient that there be a constant equality condition for all columns in a query referring to parts of the key coming before or in between parts of the `GROUP BY` key. The constants from the equality conditions fill in any “gaps” in the search keys so that it is possible to form complete prefixes of the index. These index prefixes then can be used for index lookups. If the `GROUP BY` result requires sorting, and it is possible to form search keys that are prefixes of the index, MySQL also avoids extra sorting operations because searching with prefixes in an ordered index already retrieves all the keys in order.

Assume that there is an index `idx(c1,c2,c3)` on table `t1(c1,c2,c3,c4)`. The following queries do not work with the Loose Index Scan access method described previously, but still work with the Tight Index Scan access method.

- There is a gap in the `GROUP BY`, but it is covered by the condition `c2 = 'a'`:

  ```
  SELECT c1, c2, c3 FROM t1 WHERE c2 = 'a' GROUP BY c1, c3;
  ```

- The `GROUP BY` does not begin with the first part of the key, but there is a condition that provides a constant for that part:

  ```
  SELECT c1, c2, c3 FROM t1 WHERE c1 = 'a' GROUP BY c2, c3;
  ```

8.2.1.18 DISTINCT 최적화

`DISTINCT` combined with `ORDER BY` needs a temporary table in many cases.

Because `DISTINCT` may use `GROUP BY`, learn how MySQL works with columns in `ORDER BY` or `HAVING` clauses that are not part of the selected columns. See Section 12.19.3, “MySQL Handling of GROUP BY”.

In most cases, a `DISTINCT` clause can be considered as a special case of `GROUP BY`. For example, the following two queries are equivalent:

```
SELECT DISTINCT c1, c2, c3 FROM t1
WHERE c1 > const;

SELECT c1, c2, c3 FROM t1
WHERE c1 > const GROUP BY c1, c2, c3;
```

Due to this equivalence, the optimizations applicable to `GROUP BY` queries can be also applied to queries with a `DISTINCT` clause. Thus, for more details on the optimization possibilities for `DISTINCT` queries, see Section 8.2.1.17, "GROUP BY Optimization".

When combining `LIMIT row_count` with `DISTINCT`, MySQL stops as soon as it finds `row_count` unique rows.

If you do not use columns from all tables named in a query, MySQL stops scanning any unused tables as soon as it finds the first match. In the following case, assuming that `t1` is used before `t2` (which you can check with `EXPLAIN`), MySQL stops reading from `t2` (for any particular row in `t1`) when it finds the first row in `t2`:

```
SELECT DISTINCT t1.a FROM t1, t2 where t1.a=t2.a;
```

8.2.1.19 LIMIT 쿼리 최적화

If you need only a specified number of rows from a result set, use a `LIMIT` clause in the query, rather than fetching the whole result set and throwing away the extra data.

MySQL sometimes optimizes a query that has a `LIMIT row_count` clause and no `HAVING` clause:

- If you select only a few rows with `LIMIT`, MySQL uses indexes in some cases when normally it would prefer to do a full table scan.
- If you combine `LIMIT row_count` with `ORDER BY`, MySQL stops sorting as soon as it has found the first `row_count` rows of the sorted result, rather than sorting the entire result. If ordering is done by using an index, this is very fast. If a filesort must be done, all rows that match the query without the `LIMIT` clause are selected, and most or all of them are sorted, before the first `row_count` are found. After the initial rows have been found, MySQL does not sort any remainder of the result set.
  One manifestation of this behavior is that an `ORDER BY` query with and without `LIMIT` may return rows in different order, as described later in this section.
- If you combine `LIMIT row_count` with `DISTINCT`, MySQL stops as soon as it finds `row_count` unique rows.
- In some cases, a `GROUP BY` can be resolved by reading the index in order (or doing a sort on the index), then calculating summaries until the index value changes. In this case, `LIMIT row_count` does not calculate any unnecessary `GROUP BY` values.
- As soon as MySQL has sent the required number of rows to the client, it aborts the query unless you are using `SQL_CALC_FOUND_ROWS`. In that case, the number of rows can be retrieved with `SELECT FOUND_ROWS()`. See Section 12.15, "Information Functions".
- `LIMIT 0` quickly returns an empty set. This can be useful for checking the validity of a query. It can also be employed to obtain the types of the result columns within applications that use a MySQL API that makes result set metadata available. With the **mysql** client program, you can use the `--column-type-info` option to display result column types.
- If the server uses temporary tables to resolve a query, it uses the `LIMIT row_count` clause to calculate how much space is required.
- If an index is not used for `ORDER BY` but a `LIMIT` clause is also present, the optimizer may be able to avoid using a merge file and sort the rows in memory using an in-memory `filesort` operation.

If multiple rows have identical values in the `ORDER BY` columns, the server is free to return those rows in any order, and may do so differently depending on the overall execution plan. In other words, the sort order of those rows is nondeterministic with respect to the nonordered columns.

One factor that affects the execution plan is `LIMIT`, so an `ORDER BY` query with and without `LIMIT` may return rows in different orders. Consider this query, which is sorted by the `category` column but nondeterministic with respect to the `id` and `rating` columns:

```
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 |
+----+----------+--------+
```

Including `LIMIT` may affect order of rows within each `category` value. For example, this is a valid query result:

```
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 |
+----+----------+--------+
```

In each case, the rows are sorted by the `ORDER BY` column, which is all that is required by the SQL standard.

If it is important to ensure the same row order with and without `LIMIT`, include additional columns in the `ORDER BY` clause to make the order deterministic. For example, if `id` values are unique, you can make rows for a given `category` value appear in `id` order by sorting like this:

```
mysql> SELECT * FROM ratings ORDER BY category, id;
+----+----------+--------+
| 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, 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 |
+----+----------+--------+
```

For a query with an `ORDER BY` or `GROUP BY` and a `LIMIT` clause, the optimizer tries to choose an ordered index by default when it appears doing so would speed up query execution. Prior to MySQL 8.0.21, there was no way to override this behavior, even in cases where using some other optimization might be faster. Beginning with MySQL 8.0.21, it is possible to turn off this optimization by setting the `optimizer_switch` system variable's `prefer_ordering_index` flag to `off`.

Example: First we create and populate a table `t` as shown here:

```
# Create and populate a table t:

mysql> CREATE TABLE t (
    ->     id1 BIGINT NOT NULL,
    ->     id2 BIGINT NOT NULL,
    ->     c1 VARCHAR(50) NOT NULL,
    ->     c2 VARCHAR(50) NOT NULL,
    ->  PRIMARY KEY (id1),
    ->  INDEX i (id2, c1)
    -> );

# [Insert some rows into table t - not shown]
```

Verify that the `prefer_ordering_index` flag is enabled:

```
mysql> SELECT @@optimizer_switch LIKE '%prefer_ordering_index=on%';
+-----------------------------------------------------+
| @@optimizer_switch LIKE '%prefer_ordering_index=on%' |
+-----------------------------------------------------+
|                                                   1 |
+-----------------------------------------------------+
```

Since the following query has a `LIMIT` clause, we expect it to use an ordered index, if possible. In this case, as we can see from the `EXPLAIN` output, it uses the table's primary key.

```
mysql> EXPLAIN SELECT c2 FROM t
    ->     WHERE id2 > 3
    ->     ORDER BY id1 ASC LIMIT 2\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t
   partitions: NULL
         type: index
possible_keys: i
          key: PRIMARY
      key_len: 8
          ref: NULL
         rows: 2
     filtered: 70.00
        Extra: Using where
```

Now we disable the `prefer_ordering_index` flag, and re-run the same query; this time it uses the index `i` (which includes the `id2` column used in the `WHERE` clause), and a filesort:

```
mysql> SET optimizer_switch = "prefer_ordering_index=off";

mysql> EXPLAIN SELECT c2 FROM t
    ->     WHERE id2 > 3
    ->     ORDER BY id1 ASC LIMIT 2\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t
   partitions: NULL
         type: range
possible_keys: i
          key: i
      key_len: 8
          ref: NULL
         rows: 14
     filtered: 100.00
        Extra: Using index condition; Using filesort
```

See also Section 8.9.2, “Switchable Optimizations”.

8.2.1.20 함수 호출 최적화

MySQL functions are tagged internally as deterministic or nondeterministic. A function is nondeterministic if, given fixed values for its arguments, it can return different results for different invocations. Examples of nondeterministic functions: `RAND()`, `UUID()`.

If a function is tagged nondeterministic, a reference to it in a `WHERE` clause is evaluated for every row (when selecting from one table) or combination of rows (when selecting from a multiple-table join).

MySQL also determines when to evaluate functions based on types of arguments, whether the arguments are table columns or constant values. A deterministic function that takes a table column as argument must be evaluated whenever that column changes value.

Nondeterministic functions may affect query performance. For example, some optimizations may not be available, or more locking might be required. The following discussion uses `RAND()` but applies to other nondeterministic functions as well.

Suppose that a table `t` has this definition:

```
CREATE TABLE t (id INT NOT NULL PRIMARY KEY, col_a VARCHAR(100));
```

Consider these two queries:

```
SELECT * FROM t WHERE id = POW(1,2);
SELECT * FROM t WHERE id = FLOOR(1 + RAND() * 49);
```

Both queries appear to use a primary key lookup because of the equality comparison against the primary key, but that

is true only for the first of them:

- The first query always produces a maximum of one row because `POW()` with constant arguments is a constant value and is used for index lookup.
- The second query contains an expression that uses the nondeterministic function `RAND()`, which is not constant in the query but in fact has a new value for every row of table `t`. Consequently, the query reads every row of the table, evaluates the predicate for each row, and outputs all rows for which the primary key matches the random value. This might be zero, one, or multiple rows, depending on the `id` column values and the values in the `RAND()` sequence.

The effects of nondeterminism are not limited to `SELECT` statements. This `UPDATE` statement uses a nondeterministic function to select rows to be modified:

```
UPDATE t SET col_a = some_expr WHERE id = FLOOR(1 + RAND() * 49);
```

Presumably the intent is to update at most a single row for which the primary key matches the expression. However, it might update zero, one, or multiple rows, depending on the `id` column values and the values in the `RAND()` sequence.

The behavior just described has implications for performance and replication:

- Because a nondeterministic function does not produce a constant value, the optimizer cannot use strategies that might otherwise be applicable, such as index lookups. The result may be a table scan.
- `InnoDB` might escalate to a range-key lock rather than taking a single row lock for one matching row.
- Updates that do not execute deterministically are unsafe for replication.

The difficulties stem from the fact that the `RAND()` function is evaluated once for every row of the table. To avoid multiple function evaluations, use one of these techniques:

- Move the expression containing the nondeterministic function to a separate statement, saving the value in a variable. In the original statement, replace the expression with a reference to the variable, which the optimizer can treat as a constant value:

  ```
  SET @keyval = FLOOR(1 + RAND() * 49);
  UPDATE t SET col_a = some_expr WHERE id = @keyval;
  ```

- Assign the random value to a variable in a derived table. This technique causes the variable to be assigned a value, once, prior to its use in the comparison in the `WHERE` clause:

  ```
  UPDATE /*+ NO_MERGE(dt) */ t, (SELECT FLOOR(1 + RAND() * 49) AS r) AS dt
  SET col_a = some_expr WHERE id = dt.r;
  ```

As mentioned previously, a nondeterministic expression in the `WHERE` clause might prevent optimizations and result in a table scan. However, it may be possible to partially optimize the `WHERE` clause if other expressions are deterministic. For example:

```
SELECT * FROM t WHERE partial_key=5 AND some_column=RAND();
```

If the optimizer can use `partial_key` to reduce the set of rows selected, `RAND()` is executed fewer times, which diminishes the effect of nondeterminism on optimization.

8.2.1.21 윈도우 함수 최적화

Window functions affect the strategies the optimizer considers:

- Derived table merging for a subquery is disabled if the subquery has window functions. The subquery is always materialized.
- Semijoins are not applicable to window function optimization because semijoins apply to subqueries in `WHERE` and `JOIN ... ON`, which cannot contain window functions.
- The optimizer processes multiple windows that have the same ordering requirements in sequence, so sorting can be skipped for windows following the first one.
- The optimizer makes no attempt to merge windows that could be evaluated in a single step (for example, when multiple `OVER` clauses contain identical window definitions). The workaround is to define the window in a `WINDOW` clause and refer to the window name in the `OVER` clauses.

An aggregate function not used as a window function is aggregated in the outermost possible query. For example, in this query, MySQL sees that `COUNT(t1.b)` is something that cannot exist in the outer query because of its placement in the `WHERE` clause:

```
SELECT * FROM t1 WHERE t1.a = (SELECT COUNT(t1.b) FROM t2);
```

Consequently, MySQL aggregates inside the subquery, treating `t1.b` as a constant and returning the count of rows of `t2`.

Replacing `WHERE` with `HAVING` results in an error:

```
mysql> SELECT * FROM t1 HAVING t1.a = (SELECT COUNT(t1.b) FROM t2);
ERROR 1140 (42000): In aggregated query without GROUP BY, expression #1
of SELECT list contains nonaggregated column 'test.t1.a'; this is
incompatible with sql_mode=only_full_group_by
```

The error occurs because `COUNT(t1.b)` can exist in the `HAVING`, and so makes the outer query aggregated.

Window functions (including aggregate functions used as window functions) do not have the preceding complexity. They always aggregate in the subquery where they are written, never in the outer query.

Window function evaluation may be affected by the value of the `windowing_use_high_precision` system variable, which determines whether to compute window operations without loss of precision. By default, `windowing_use_high_precision` is enabled.

For some moving frame aggregates, the inverse aggregate function can be applied to remove values from the aggregate. This can improve performance but possibly with a loss of precision. For example, adding a very small floating-point value to a very large value causes the very small value to be “hidden” by the large value. When inverting the large value later, the effect of the small value is lost.

Loss of precision due to inverse aggregation is a factor only for operations on floating-point (approximate-value) data types. For other types, inverse aggregation is safe; this includes `DECIMAL`, which permits a fractional part but is an exact-value type.

For faster execution, MySQL always uses inverse aggregation when it is safe:

- For floating-point values, inverse aggregation is not always safe and might result in loss of precision. The default is to avoid inverse aggregation, which is slower but preserves precision. If it is permissible to sacrifice safety for speed, `windowing_use_high_precision` can be disabled to permit inverse aggregation.
- For nonfloating-point data types, inverse aggregation is always safe and is used regardless of the `windowing_use_high_precision` value.
- `windowing_use_high_precision` has no effect on `MIN()` and `MAX()`, which do not use inverse aggregation in any case.

For evaluation of the variance functions `STDDEV_POP()`, `STDDEV_SAMP()`, `VAR_POP()`, `VAR_SAMP()`, and their synonyms, evaluation can occur in optimized mode or default mode. Optimized mode may produce slightly different results in the last significant digits. If such differences are permissible, `windowing_use_high_precision` can be disabled to permit optimized mode.

For `EXPLAIN`, windowing execution plan information is too extensive to display in traditional output format. To see windowing information, use `EXPLAIN FORMAT=JSON` and look for the `windowing` element.

8.2.1.22 행 생성자 표현 최적화

Row constructors permit simultaneous comparisons of multiple values. For example, these two statements are semantically equivalent:

```
SELECT * FROM t1 WHERE (column1,column2) = (1,1);
SELECT * FROM t1 WHERE column1 = 1 AND column2 = 1;
```

In addition, the optimizer handles both expressions the same way.

The optimizer is less likely to use available indexes if the row constructor columns do not cover the prefix of an index. Consider the following table, which has a primary key on `(c1, c2, c3)`:

```
CREATE TABLE t1 (
  c1 INT, c2 INT, c3 INT, c4 CHAR(100),
  PRIMARY KEY(c1,c2,c3)
);
```

In this query, the `WHERE` clause uses all columns in the index. However, the row constructor itself does not cover an index prefix, with the result that the optimizer uses only `c1` (`key_len=4`, the size of `c1`):

```
mysql> EXPLAIN SELECT * FROM t1
    WHERE c1=1 AND (c2,c3) > (1,1)\G
*************************** 1. row ***************************
```

```
           id: 1
  select_type: SIMPLE
        table: t1
   partitions: NULL
         type: ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 3
     filtered: 100.00
        Extra: Using where
```

In such cases, rewriting the row constructor expression using an equivalent nonconstructor expression may result in more complete index use. For the given query, the row constructor and equivalent nonconstructor expressions are:

```
(c2,c3) > (1,1)
c2 > 1 OR ((c2 = 1) AND (c3 > 1))
```

Rewriting the query to use the nonconstructor expression results in the optimizer using all three columns in the index (`key_len=12`):

```
mysql> EXPLAIN SELECT * FROM t1
       WHERE c1 = 1 AND (c2 > 1 OR ((c2 = 1) AND (c3 > 1)))\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
   partitions: NULL
         type: range
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 12
          ref: NULL
         rows: 3
     filtered: 100.00
        Extra: Using where
```

Thus, for better results, avoid mixing row constructors with `AND`/`OR` expressions. Use one or the other.

Under certain conditions, the optimizer can apply the range access method to `IN()` expressions that have row constructor arguments. See Range Optimization of Row Constructor Expressions.

8.2.1.23 풀 테이블 스캔 피하기

The output from `EXPLAIN` shows `ALL` in the `type` column when MySQL uses a full table scan to resolve a query. This usually happens under the following conditions:

- The table is so small that it is faster to perform a table scan than to bother with a key lookup. This is common for tables with fewer than 10 rows and a short row length.
- There are no usable restrictions in the `ON` or `WHERE` clause for indexed columns.
- You are comparing indexed columns with constant values and MySQL has calculated (based on the index tree) that the constants cover too large a part of the table and that a table scan would be faster. See Section 8.2.1.1, “WHERE Clause Optimization”.
- You are using a key with low cardinality (many rows match the key value) through another column. In this case, MySQL assumes that by using the key probably requires many key lookups and that a table scan would be faster.

For small tables, a table scan often is appropriate and the performance impact is negligible. For large tables, try the following techniques to avoid having the optimizer incorrectly choose a table scan:

- Use `ANALYZE TABLE tbl_name` to update the key distributions for the scanned table. SeeSection 13.7.3.1, “ANALYZE TABLE Statement”.
- Use `FORCE INDEX` for the scanned table to tell MySQL that table scans are very expensive compared to using the given index:

  ```
  SELECT * FROM t1, t2 FORCE INDEX (index_for_column)
    WHERE t1.col_name=t2.col_name;
  ```

See Section 8.9.4, “Index Hints”.

- Start **mysqld** with the `--max-seeks-for-key=1000` option or use `SET max_seeks_for_key=1000` to tell the optimizer to assume that no key scan causes more than 1,000 key seeks. See Section 5.1.8, “Server System Variables”.

Revision #9
★Created 8 September 2023 05:42:15 by 신민항
✎Updated 9 September 2023 19:28:53 by 신민항