8장 최적화

목차

이 장에서는 MySQL 성능을 최적화하는 방법을 설명하고 예제를 제공합니다. 최적화에는 여러 수준에서 성능을 구성, 튜닝 및 측정하는 작업이 포함됩니다. 직무 역할(개발자, DBA 또는 이 둘의 조합)에 따라 개별 SQL 문, 전체 애플리케이션, 단일 데이터베이스 서버 또는 여러 네트워크에 연결된 데이터베이스 서버 수준에서 최적화할 수 있습니다. 때로는 사전 예방적으로 성능에 대해 미리 계획할 수도 있고, 문제가 발생한 후에 구성 또는 코드 문제를 해결할 수도 있습니다. CPU 및 메모리 사용량을 최적화하면 확장성도 향상되어 데이터베이스가 속도 저하 없이 더 많은 부하를 처리할 수 있습니다.

8.1 최적화 개요

데이터베이스 성능은 테이블, 쿼리 및 구성 설정과 같은 데이터베이스 수준의 여러 요소에 따라 달라집니다. 이러한 소프트웨어 구성으로 인해 하드웨어 수준에서 CPU 및 I/O 작업이 발생하며, 이를 최소화하고 최대한 효율적으로 만들어야 합니다. 데이터베이스 성능에 대해 작업할 때는 소프트웨어 측면에 대한 높은 수준의 규칙과 지침을 배우고, 월 클럭 시간을 사용하여 성능을 측정하는 것부터 시작합니다. 전문가가 되면 내부에서 일어나는 일에 대해 더 많이 배우고 CPU 주기 및 I/O 작업과 같은 것을 측정하기 시작합니다.

일반적인 사용자는 기존 소프트웨어 및 하드웨어 구성에서 최상의 데이터베이스 성능을 얻는 것을 목표로 합니다. 고급 사용자는 MySQL 소프트웨어 자체를 개선하거나 자체 스토리지 엔진 및 하드웨어 어플라이언스를 개발하여 MySQL 에코시스템을 확장할 수 있는 기회를 모색합니다.

데이터베이스 수준에서 최적화

데이터베이스 애플리케이션을 빠르게 만드는 데 가장 중요한 요소는 기본 설계입니다:

- 테이블이 제대로 구조화되어 있나요? 특히 열에 올바른 데이터 유형이 있고 각 테이블에 작업 유형에 적합한 열이 있나요? 예를 들어, 업데이트를 자주 수행하는 애플리케이션에는 열이 적은 테이블이 많은 반면, 대량의 데이터를 분석하는 애플리케이션에는 열이 많은 테이블이 적은 경우가 많습니다.
- 쿼리를 효율적으로 처리할 수 있는 올바른 인덱스가 마련되어 있나요?
- 각 테이블에 적합한 스토리지 엔진을 사용하고 있으며, 사용하는 각 스토리지 엔진의 강점과 기능을 잘 활용하고 계신가요? 특히, `InnoDB`와 같은 트랜잭션 스토리지 엔진 또는 `MyISAM`과 같은 비트랜잭션 스토리지 엔진의 선택은 성능과 확장성 측면에서 매우 중요할 수 있습니다.

참고

`InnoDB`는 새 테이블의 기본 스토리지 엔진입니다. 실제로는 고급 `InnoDB` 성능 기능으로 인해 특히 사용량이 많은 데이터베이스의 경우 `InnoDB` 테이블이 더 단순한 `MyISAM` 테이블보다 성능이 뛰어난 경우가 많습니다.

- 각 테이블이 적절한 행 형식을 사용합니까? 이 선택은 테이블에 사용되는 스토리지 엔진에 따라 달라집니다. 특히 압축 테이블은 디스크 공간을 덜 사용하므로 데이터를 읽고 쓰는 데 필요한 디스크 I/O가 더 적습니다. 압축은 `InnoDB` 테이블을 사용하는 모든 종류의 워크로드 및 읽기 전용 `MyISAM` 테이블에 사용할 수 있습니다.
- 애플리케이션이 적절한 잠금 전략을 사용하나요? 예를 들어, 데이터베이스 작업을 동시에 실행할 수 있도록 가능한 경우 공유 액세스를 허용하고, 중요한 작업이 최우선 순위를 차지할 수 있도록 적절한 경우 독점 액세스를 요청하는 것입니다. 다시 말하지만, 스토리지 엔진의 선택은 매우 중요합니다. `InnoDB` 스토리지 엔진은 사용자의 개입 없이 대부분의 잠금 문제를 처리하므로 데이터베이스의 동시성을 개선하고 코드에 대한 실험과 튜닝의 양을 줄일 수 있습니다.
- 캐싱에 사용되는 모든 메모리 영역의 크기가 올바르게 설정되어 있나요? 즉, 자주 액세스하는 데이터를

저장할 수 있을 만큼 충분히 크되 물리적 메모리에 과부하가 걸려 페이징을 유발할 정도로 크지 않아야 합니다. 구성할 주요 메모리 영역은 `InnoDB` 버퍼 풀과 `MyISAM` 키 캐시입니다.

하드웨어 수준에서 최적화

데이터베이스가 점점 더 바빠지면 모든 데이터베이스 애플리케이션은 결국 하드웨어 한계에 도달하게 됩니다. DBA는 이러한 병목 현상을 피하기 위해 애플리케이션을 조정하거나 서버를 재구성할 수 있는지 또는 더 많은 하드웨어 리소스가 필요한지 여부를 평가해야 합니다. 시스템 병목 현상은 일반적으로 이러한 원인으로 인해 발생합니다:

- 디스크 검색. 디스크가 데이터 조각을 찾는 데는 시간이 걸립니다. 최신 디스크의 경우 평균 시간은 일반적으로 10ms 미만이므로 이론적으로는 초당 100회 정도 검색할 수 있습니다. 이번에는
  는 새 디스크를 사용하면 느리게 개선되며 단일 테이블에 최적화하기가 매우 어렵습니다. 검색 시간을 최적화하는 방법은 데이터를 둘 이상의 디스크에 분산하는 것입니다.

- 디스크 읽기 및 쓰기. 디스크가 올바른 위치에 있으면 데이터를 읽거나 써야 합니다. 최신 디스크의 경우, 하나의 디스크는 최소 10~20MB/s의 처리량을 제공합니다. 여러 디스크에서 병렬로 읽을 수 있으므로 탐색보다 최적화하기가 더 쉽습니다.
- CPU 주기. 데이터가 주 메모리에 있으면 결과를 얻기 위해 데이터를 처리해야 합니다. 메모리 양에 비해 테이블이 큰 것이 가장 일반적인 제한 요소입니다. 그러나 테이블이 작은 경우에는 일반적으로 속도가 문제가 되지 않습니다.
- 메모리 대역폭. CPU 캐시에 들어갈 수 있는 것보다 더 많은 데이터가 CPU에 필요하면 주 메모리 대역폭이 병목 현상이 발생합니다. 이는 대부분의 시스템에서 흔하지 않은 병목 현상이지만 주의해야 할 사항입니다.

휴대성과 성능의 균형

이식 가능한 MySQL 프로그램에서 성능 지향 SQL 확장을 사용하려면 `/*! */` 주석 구분 기호로 감싸면 됩니다. 다른 SQL 서버는 주석 처리된 키워드를 무시합니다. 주석 작성에 대한 자세한 내용은 섹션 9.7, "주석"을 참조하세요.

8.2 SQL 문 최적화

데이터베이스 애플리케이션의 핵심 로직은 인터프리터를 통해 직접 실행되거나 API를 통해 백그라운드로 제출되는 SQL 문을 통해 수행됩니다. 이 섹션의 튜닝 가이드라인은 모든 종류의 MySQL 애플리케이션의 속도를 높이는 데 도움이 됩니다. 이 가이드라인은 데이터를 읽고 쓰는 SQL 작업, 일반적인 SQL 작업의 백그라운드 오버헤드, 데이터베이스 모니터링과 같은 특정 시나리오에서 사용되는 작업을 다룹니다.

8.2.1 SELECT 문 최적화

`SELECT` 문 형태의 쿼리는 데이터베이스에서 모든 조회 작업을 수행합니다. 동적 웹 페이지에 대해 1초 미만의 응답 시간을 달성하거나 밤새 대량의 보고서를 생성하는 데 걸리는 시간을 단축하려면 이러한 문을 조정하는 것이 최우선 과제입니다.

`SELECT` 문 외에도 쿼리 튜닝 기법은 다음과 같은 구문에도 적용됩니다.
`CREATE TABLE...AS SELECT`, `INSERT INTO...SELECT` 및 `DELETE`의 `WHERE` 절
문을 사용할 수 있습니다. 이러한 문은 쓰기 작업과 읽기 지향 쿼리 작업을 결합하기 때문에 추가적인 성능 고려 사항이 있습니다.

NDB 클러스터는 조인 푸시다운 최적화를 지원하여, 적격 조인 전체가 NDB 클러스터 데이터 노드로 전송되어 노드 간에 분산되어 병렬로 실행될 수 있습니다. 이 최적화에 대한 자세한 내용은 NDB 푸시다운 조인의 조건을 참조하십시오.

쿼리 최적화를 위한 주요 고려 사항은 다음과 같습니다:

- 느린 `SELECT ... WHERE` 쿼리를 더 빠르게 만들려면 먼저 인덱스를 추가할 수 있는지 확인해야 합니다.

`WHERE` 절에 사용되는 열에 인덱스를 설정하면 평가, 필터링 및 최종 결과 검색 속도를 높일 수 있습니다. 디스크 공간 낭비를 방지하려면 애플리케이션에서 사용되는 많은 관련 쿼리의 속도를 높여주는 작은 인덱스 집합을 구성하세요.

인덱스는 조인 및 외래 키와 같은 기능을 사용하여 서로 다른 테이블을 참조하는 쿼리에서 특히 중요합니다. `EXPLAIN` 문을 사용하여 `SELECT`에 사용되는 인덱스를 결정할 수 있습니다. 섹션 8.3.1, 'MySQL에서 인덱스를 사용하는 방법' 및 섹션 8.8.1, 'EXPLAIN을 사용하여 쿼리 최적화'를 참조하십시오.

- 함수 호출과 같이 쿼리에서 시간이 과도하게 걸리는 부분을 분리하고 조정합니다. 쿼리 구조에 따라 결과 집합의 모든 행에 대해 함수를 한 번씩 호출하거나 테이블의 모든 행에 대해 함수를 한 번씩 호출할 수 있어 비효율성이 크게 증가합니다.
- 특히 큰 테이블의 경우 쿼리에서 전체 테이블 스캔 횟수를 최소화하세요.
- 최적화 프로그램이 효율적인 실행 계획을 수립하는 데 필요한 정보를 얻을 수 있도록 주기적으로 테이블 통계를 최신 상태로 유지합니다.

- 각 테이블에 대한 스토리지 엔진에 특정한 튜닝 기술, 인덱싱 기술 및 구성 매개변수에 대해 알아보세요. InnoDB와 `MyISAM` 모두 쿼리에서 고성능을 활성화하고 유지하기 위한 일련의 지침이 있습니다. 자세한 내용은 8.5.6절, 'InnoDB 쿼리 최적화' 및 8.6.1절, 'MyISAM 쿼리 최적화'를 참조하세요.
- 섹션 8.5.3, 'InnoDB 읽기 전용 트랜잭션 최적화'에 나와 있는 기술을 사용하여 `InnoDB` 테이블에 대한 단일 쿼리 트랜잭션을 최적화할 수 있습니다.
- 특히 최적화 도구가 동일한 변환을 자동으로 수행하는 경우 이해하기 어려운 방식으로 쿼리를 변환하지 마세요.
- 기본 지침 중 하나로 성능 문제를 쉽게 해결할 수 없는 경우, `EXPLAIN` 플랜을 읽고 인덱스, `WHERE` 절, 조인 절 등을 조정하여 특정 쿼리의 내부 세부 사항을 조사합니다. (특정 수준의 전문 지식에 도달하면 모든 쿼리에 대해 `EXPLAIN` 플랜을 읽는 것이 첫 번째 단계가 될 수 있습니다.)
- MySQL이 캐싱에 사용하는 메모리 영역의 크기와 속성을 조정합니다. `InnoDB` 버퍼 풀, `MyISAM` 키 캐시 및 MySQL 쿼리 캐시를 효율적으로 사용하면 두 번째 및 그 이후의 메모리에서 결과를 검색하기 때문에 반복 쿼리가 더 빠르게 실행됩니다.
- 캐시 메모리 영역을 사용하여 빠르게 실행되는 쿼리의 경우에도 캐시 메모리를 덜 필요로 하도록 추가로 최적화하여 애플리케이션의 확장성을 높일 수 있습니다. 확장성이란 애플리케이션이 성능 저하 없이 더 많은 동시 사용자, 더 큰 요청 등을 처리할 수 있음을 의미합니다.
- 동시에 테이블에 액세스하는 다른 세션으로 인해 쿼리 속도가 영향을 받을 수 있는 잠금 문제를 처리합니다.

8.2.1.1 WHERE 절 최적화

이 섹션에서는 `WHERE` 절을 처리하기 위해 수행할 수 있는 최적화에 대해 설명합니다. 예제에서는 `SELECT` 문을 사용하지만 `DELETE` 및 `UPDATE` 문의 `WHERE` 절에도 동일한 최적화가 적용됩니다.

참고

MySQL 옵티마이저에 대한 작업이 계속 진행 중이므로 MySQL이 수행하는 모든 최적화가 여기에 문서화되어 있지는 않습니다.

가독성을 희생하면서 산술 연산을 더 빠르게 하기 위해 쿼리를 다시 작성하고 싶을 수 있습니다. MySQL은 유사한 최적화를 자동으로 수행하므로 이러한 작업을 피하고 쿼리를 더 이해하기 쉽고 유지 관리하기 쉬운 형태로 남겨둘 수 있습니다. MySQL에서 수행하는 최적화 중 일부는 다음과 같습니다:

- 불필요한 괄호 제거:

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

- 일정한 접기:

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

- 지속적인 조건 제거:

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

이 작업은 최적화 단계가 아닌 준비 단계에서 수행되므로 조인을 간소화하는 데 도움이 됩니다. 자세한 내용과 예제는 섹션 8.2.1.9, "외부 조인 최적화"를 참조하세요.

- 인덱스에 사용되는 상수 표현식은 한 번만 평가됩니다.
- 상수 값이 있는 숫자 유형의 열을 비교하여 유효하지 않거나 범위를 벗어난 값이 있는지 확인하고 접히거나 제거합니다:

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

  자세한 내용은 섹션 8.2.1.14, "상수 접기 최적화"를 참조하세요.

- COUNT(*)는 WHERE가 없는 단일 테이블의 경우 MyISAM 및 MEMORY 테이블의 테이블 정보에서 직접 검색됩니다. 이 작업은 하나의 테이블에만 사용되는 모든 NOT NULL 표현식에 대해서도 수행됩니다.
- 유효하지 않은 상수 표현식의 조기 감지. MySQL은 일부 SELECT 문은 불가능하며 행을 반환하지 않습니다.
- GROUP BY 또는 집계 함수(COUNT()를 사용하지 않는 경우 HAVING이 WHERE와 병합됩니다, MIN() 등).
- 조인의 각 테이블에 대해 테이블에 대한 빠른 WHERE 평가를 얻고 가능한 한 빨리 행을 건너뛸 수 있도록 더 간단한 WHERE가 구성됩니다.
- 모든 상수 테이블은 쿼리에서 다른 테이블보다 먼저 읽습니다. 상수 테이블은 다음 중 하나입니다:
  - 빈 테이블 또는 행이 하나 있는 테이블입니다.
  - 모든 인덱스 부분이 상수 표현식과 비교되고 NOT NULL로 정의되는 PRIMARY KEY 또는 UNIQUE 인덱스의 WHERE 절과 함께 사용되는 테이블입니다.

  다음 표는 모두 상수 표로 사용됩니다:

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

- 테이블을 조인하는 데 가장 적합한 조인 조합은 모든 가능성을 시도하여 찾을 수 있습니다. ORDER BY 및 GROUP BY 절의 모든 열이 동일한 테이블에서 가져온 경우 조인할 때 해당 테이블이 우선적으로 사용됩니다.
- ORDER BY 절과 다른 GROUP BY 절이 있는 경우, 또는 ORDER BY 또는 GROUP BY가 에 조인 대기열의 첫 번째 테이블이 아닌 다른 테이블의 열이 포함되어 있으면 임시 테이블이 만들어집니다.
- SQL_SMALL_RESULT 수정자를 사용하는 경우 MySQL은 인메모리 임시 테이블을 사용합니다.
- 각 테이블 인덱스가 쿼리되며, 최적화 도구가 테이블 스캔을 사용하는 것이 더 효율적이라고 판단하지 않는 한 최상의 인덱스가 사용됩니다. 한때는 최상의 인덱스가 테이블의 30% 이상을 차지하는지 여부에 따라 스캔이 사용되었지만, 더 이상 고정된 비율로 인덱스 사용과 스캔 중 하나를 결정하지 않습니다. 이제 최적화 도구는 더 복잡해졌으며 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 열이 숫자인 경우 인덱스 트리만 쿼리 해결에 사용됩니다.

- 각 행이 출력되기 전에 `HAVING` 절과 일치하지 않는 행은 건너뜁니다. 매우 빠른 쿼리의

```
SELECT COUNT(*) FROM tbl_name;

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

몇 가지 예입니다:

```
SELECT MAX(key_part2) FROM tbl_name
  WHERE key_part1=상수;

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 인덱싱된 열이 숫자라고 가정하여 인덱스 트리만 사용하여 다음 쿼리를 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 인덱싱을 사용하여 별도의 정렬 패스 없이 정렬된 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

범위 액세스 방법은 단일 인덱스를 사용하여 하나 또는 여러 인덱스 값 간격 내에 포함된 테이블 행의 하위 집합을 검색합니다. 단일 부분 또는 다중 부분 인덱스에 사용할 수 있습니다. 다음 섹션에서는 옵티마이저가 범위 액세스를 사용하는 조건에 대해 설명합니다.

- 단일 부분 인덱스의 범위 액세스 방법
- 여러 부분으로 구성된 인덱스의 범위 액세스 방법
- 다값 비교의 동등 범위 최적화
- 스캔 범위 액세스 방법 건너뛰기
- 행 생성자 표현식의 범위 최적화
- 범위 최적화를 위한 메모리 사용 제한

단일 부분 인덱스의 범위 액세스 방법

단일 부분 인덱스의 경우 인덱스 값 간격은 "간격"이 아닌 *범위 조건으로* 표시되는 `WHERE` 절의 해당 조건으로 편리하게 나타낼 수 있습니다.

단일 부분 인덱스에 대한 범위 조건의 정의는 다음과 같습니다:

- `BTREE` 및 `HASH` 인덱스 모두에서 `=`, `<=>`, `IN()`, `IS NULL` 또는 `IS NOT NULL` 연산자를 사용할 때 키 부분과 상수 값을 비교하는 것은 범위 조건입니다.
- 또한 `BTREE` 인덱스의 경우 >, <, `>=`, `<=`, `BETWEEN`, `!=` 또는 `<>` 연산자를 사용할 때는 키 부분과 상수

값의 비교가 범위 조건이며, `LIKE` 인수가 와일드카드 문자로 시작하지 않는 상수 문자열인 경우 `LIKE` 비교가 범위 조건이 됩니다.

- 모든 인덱스 유형에서 여러 범위 조건은 `OR` 또는 `AND`와 결합하여 범위 조건을 구성합니다. 앞의 설명에서 "상수 값"은 다음 중 하나를 의미합니다:
  - 쿼리 문자열의 상수
  - 동일한 조인에서 생성 또는 시스템 테이블의 열

- 연관성이 없는 하위 쿼리의 결과
- 이전 유형의 하위 표현식으로만 구성된 모든 표현식 다음은 `WHERE` 절에 범위 조건

이 있는 쿼리의 몇 가지 예입니다:

```
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%'
  또는 'bar'와 'foo' 사이의 key_col;
```

일부 상수가 아닌 값은 옵티마이저 상수 전파 단계에서 상수로 변환될 수 있습니다.

MySQL은 가능한 각 인덱스에 대해 `WHERE` 절에서 범위 조건을 추출하려고 시도합니다. 추출 과정에서 범위 조건을 구성하는 데 사용할 수 없는 조건은 삭제되고, 범위가 겹치는 조건은 결합되며, 빈 범위를 생성하는 조건은 제거됩니다.

여기서 key1은인덱싱된 열이고 nonkey는 인덱싱되지 않은 다음 문을 고려합니다:

```
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');
```

키 `키1`의 추출 프로세스는 다음과 같습니다:

1. 원래 `WHERE` 절로 시작합니다:

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

2. 범위 스캔에 사용할 수 없으므로 `nonkey = 4` 및 `key1 LIKE '%b'`를 제거합니다. 이를 제거하는 올바른 방법은 범위 스캔을 수행할 때 일치하는 행을 놓치지 않도록 `TRUE`로 대체하는 것입니다. `TRUE`로 바꾸면 결과가 나옵니다:

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

3. 항상 참 또는 거짓인 축소 조건:

   - `(key1 LIKE 'abcde%' OR TRUE)`는 항상 참입니다.
   - `(key1 < 'uux' AND key1 > 'z')`는 항상 거짓 이

   러한 조건을 상수로 대체하면 결과가 산출됩니다:

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

불필요한 참과 거짓 상수를 제거하면 결과를 얻을 수 있습니다:

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

4. 겹치는 간격을 하나로 합치면 범위 스캔에 사용할 최종 조건이 산출됩니다:

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

일반적으로(그리고 앞의 예제에서 알 수 있듯이) 범위 스캔에 사용되는 조건은 WHERE 절보다 덜 제한적입니다. MySQL은 추가 검사를 수행하여 범위 조건은 충족하지만 전체 WHERE 절은 충족하지 않는 행을 필터링합니다.

범위 조건 추출 알고리즘은 임의의 깊이의 중첩된 AND/OR 구문을 처리할 수 있으며, 출력은 WHERE 절에 조건이 나타나는 순서에 의존하지 않습니다.

MySQL은 공간 인덱스의 범위 액세스 방법에 대해 여러 범위 병합을 지원하지 않습니다. 이 제한을 해결하려면 각 공간 술어를 다른 SELECT에 넣는다는 점을 제외하고 동일한 SELECT 문이 있는 UNION을 사용할 수 있습니다.

여러 부분으로 구성된 인덱스의 범위 액세스 방법

여러 부분으로 구성된 인덱스의 범위 조건은 단일 부분 인덱스의 범위 조건을 확장한 것입니다. 여러 부분으로 구성된 인덱스의 범위 조건은 인덱스 행이 하나 또는 여러 키 튜플 간격 내에 위치하도록 제한합니다. 키 튜플 간격은 인덱스의 순서를 사용하여 키 튜플 집합에 대해 정의됩니다.

예를 들어, 키 순서대로 나열된 다음과 같은 키 튜플 집합을 키1(*key_part1, key_part2, key_part3*)로 정의된 여러 부분으로 구성된 인덱스를 생각해 보겠습니다:

```
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'
```

키_파트1 = 1 조건은 이 간격을 정의합니다:

```
(1,-INF,-INF) <= (KEY_PART1,KEY_PART2,KEY_PART3) <= (1,+INF,+INF)
```

간격은 이전 데이터 집합의 4번째, 5번째, 6번째 튜플을 포함하며 범위 액세스 메서드에서 사용할 수 있습니다.

반면, 조건 *key_part3 = 'abc'*는 단일 간격을 정의하지 않으므로 범위 액세스 메서드에서 사용할 수 없습니다.

다음 설명에서는 여러 부분으로 구성된 인덱스에서 범위 조건이 작동하는 방식을 자세히 설명합니다.

- 해시 인덱스의 경우 동일한 값을 포함하는 각 간격을 사용할 수 있습니다. 즉, 다음과 같은 형식의 조건에 대해서만 간격을 생성할 수 있습니다:

  ```
      key_part1 cmp const1
  AND key_part2 cmp const2
  AND ...
  AND key_partN cmp constN;
  ```

  여기서 *const1*, *const2*, ...는 상수이고, *cmp*는 =, <=> 또는 IS NULL 비교 연산자 중 하나이며, 조건은 모든 인덱스 부분을 포괄합니다. (즉, *N개 부분으로 구성된* 인덱스의 각 부분마다 하나씩 *N개의* 조건이 있습니다.) 예를 들어 다음은 세 부분으로 구성된 HASH 인덱스에 대한 범위 조건입니다:

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

상수로 간주되는 것에 대한 정의는 단일 부분 인덱스의 범위 액세스 방법을 참조하세요.

- BTREE 인덱스의 경우, 각 조건이 =, <=>, IS NULL, >, <, >=, <=, !=, <>, BETWEEN 또는 LIKE '*패턴*'(여기서 '*패턴*'은 와일드카드로 시작하지 않음)을 사용하여 키 부분을 상수 값과 비교하는 AND와 결합된 조건에 간격이 사용 가능할 수 있습니다. 간격은 조건과 일치하는 모든 행을 포함하는 단일 키 튜플을 확인할 수 있는 한 사용할 수 있습니다(<> 또는 !=가 사용된 경우 두 개의 간격).

비교 연산자가 =, <=> 또는 IS NULL인 경우 옵티마이저는 간격을 결정하기 위해 추가 키 부분을 사용하려고 시도합니다. 연산자가 >, <, >=, <=, !=, <>, BETWEEN 또는 LIKE인 경우 옵티마이저는 해당 연산자를 사용하지만 더 이상 핵심 부분을 고려하지 않습니다. 다음 표현식의 경우 옵티마이저는 첫 번째 비교에서 =를 사용합니다. 또한 두 번째 비교에서 >=를 사용하지만 더 이상 핵심 부분을 고려하지 않으며 간격 구성을 위해 세 번째 비교를 사용하지 않습니다:

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

단일 간격은 다음과 같습니다:

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

생성된 간격에 초기 조건보다 많은 행이 포함될 수 있습니다. 예를 들어, 앞의 간격에는 원래 조건을 충족하지 않는 값 ('foo', 11, 0)이 포함되어 있습니다.

- 간격 내에 포함된 행 집합을 포함하는 조건을 OR과 결합하면 해당 간격의 합집합 내에 포함된 행 집합을 포함하는 조건이 됩니다. 조건이 AND와 결합되면 해당 간격의 교집합 내에 포함된 행 집합을 포함하는 조건이 됩니다. 예를 들어, 두 부분으로 구성된 인덱스에 대한 이 조건을 예로 들어 보겠습니다:

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

간격은 다음과 같습니다:

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

이 예에서 첫 번째 줄의 간격은 왼쪽 바운드에 하나의 키 부분을 사용하고 오른쪽 바운드에 두 개의 키 부분을 사용합니다. 두 번째 줄의 간격은 하나의 키 부분만 사용합니다. EXPLAIN 출력의 key_len 열은 사용된 키 접두사의 최대 길이를 나타냅니다.

경우에 따라 key_len은 키 부분이 사용되었음을 나타낼 수 있지만 이는 예상과 다를 수 있습니다. key_part1과 key_part2가 NULL일 수 있다고 가정해 보겠습니다. 그러면 key_len 열에 다음 조건에 대한 두 개의 키 부분 길이가 표시됩니다:

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

그러나 실제로 조건은 다음과 같이 변환됩니다:

```
key_part1 >= 1 AND key_part2가 NULL이 아닌 경우
```

단일 부분 인덱스에서 범위 조건의 간격을 결합하거나 제거하기 위해 최적화가 수행되는 방법에 대한 설명은 단일 부분 인덱스의 범위 액세스 방법을 참조하세요. 여러 부분으로 구성된 인덱스의 범위 조건에 대해서도 유사한 단계가 수행됩니다.

다값 비교의 동등 범위 최적화

다음 표현식을 고려해 보겠습니다. 여기서 col_name은 인덱싱된 열입니다:

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

col_name이 여러 값 중 하나와 같으면 각 표현식은 참입니다. 이러한 비교는 동일 범위 비교입니다('범위'가 단일 값인 경우). 옵티마이저는 다음과 같이 동일 범위 비교를 위해 적격 행을 읽는 데 드는 비용을 추정합니다:

- *col_name에* 고유 인덱스가 있는 경우, 최대 하나의 행만 주어진 값을 가질 수 있으므로 각 범위에 대한 행 추정치는 1입니다.
- 그렇지 않으면 *col_name의* 모든 인덱스는 고유하지 않으며 옵티마이저는 인덱스 또는 인덱스 통계에 대한 심층 분석을 사용하여 각 범위에 대한 행 수를 추정할 수 있습니다.

인덱스 다이브를 사용하면 옵티마이저가 범위의 각 끝에서 다이브를 수행하고 범위의 행 수를 추정치로 사용합니다. 예를 들어, *col_name* IN (10, 20, 30) 표현식에는 세 개의 동일 범위가 있으며 옵티마이저는 범위당 두 번의 다이빙을 수행하여 행 추정치를 생성합니다. 각 다이빙 쌍은 주어진 값을 갖는 행의 수에 대한 추정치를 산출합니다.

인덱스 분석은 정확한 행 추정치를 제공하지만 표현식의 비교 값 수가 증가함에 따라 최적화 도구가 행 추정치를 생성하는 데 시간이 더 오래 걸립니다. 인덱스 통계를 사용하면 인덱스 다이브보다 정확도는 떨어지지만 큰 값 목록에 대해 더 빠르게 행을 추정할 수 있습니다.

eq_range_index_dive_limit 시스템 변수를 사용하면 옵티마이저가 한 행 추정 전략에서 다른 행 추정 전략으로 전환하는 값의 수를 구성할 수 있습니다. 최대 *N개의* 동일 범위 비교에 인덱스 다이브를 사용할 수 있도록 하려면 eq_range_index_dive_limit을 설정합니다.
를 N + 1로 설정합니다. 통계 사용을 비활성화하고 *N에* 관계없이 항상 인덱스 다이빙을 사용하려면 다음을 설정합니다.
EQ_RANGE_INDEX_DIVE_LIMIT를 0으로 설정합니다.

최상의 추정치를 위해 테이블 인덱스 통계를 업데이트하려면 테이블 분석을 사용합니다.

MySQL 8.2 이전에는 인덱스 유용성을 추정하기 위해 인덱스 다이브 사용을 건너뛰는 방법이 없었으며, 시스템 변수인 eq_range_index_dive_limit을 사용하는 것 외에는 방법이 없었습니다. MySQL 8.2에서는 이러한 조건을 모두 충족하는 쿼리에 대해 인덱스 다이브 생략이 가능합니다:

- 쿼리는 여러 테이블에 대한 조인이 아니라 단일 테이블에 대한 쿼리입니다.
- 단일 인덱스 강제 인덱스 힌트가 있습니다. 인덱스 사용이 강제되면 인덱스에 대한 다이빙을 수행하는 추가 오버헤드로 인해 얻을 수 있는 이득이 없기 때문입니다.
- 인덱스는 전체 텍스트 인덱스가 아닌 고유하지 않은 인덱스입니다.
- 하위 쿼리가 없습니다.
- DISTINCT, GROUP BY 또는 ORDER BY 절이 없습니다.

연결에 대한 설명의 경우 인덱스 다이빙을 건너뛰면 출력이 다음과 같이 변경됩니다:

- 기존 출력의 경우 행과 필터링된 값은 NULL입니다.
- JSON 출력의경우, 행_검토_당_스캔 및 행_생산_당_조인은 표시되지 않습니다,
  SKIP_INDEX_DIVE_DUE_TO_FORCE가 참이면 비용 계산이 정확하지 않습니다. FOR

CONNECTION이 없으면 인덱스 다이빙을 건너뛰어도 EXPLAIN 출력은 변경되지 않습니다.

인덱스 다이브가 건너뛰는 쿼리를 실행한 후, 정보 스키마 OPTIMIZER_TRACE 테이블의 해당 행에는 skipped_due_to_force_index의 index_dives_for_range_access 값이 포함되어 있습니다.

스캔 범위 액세스 방법 건너뛰기

다음 시나리오를 고려하세요:

```
CREATE TABLE t1 (f1 INT NOT NULL, f2 INT NOT NULL, PRIMARY KEY(f1, f2));
t1 값에 삽입
  (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;
```

이 쿼리를 실행하기 위해 MySQL은 인덱스 스캔을 선택하여 모든 행(인덱스에는 선택할 모든 열이 포함됨)을 가져온 다음 `WHERE` 절에서 `f2 > 40` 조건을 적용하여 최종 결과 집합을 생성할 수 있습니다.

범위 스캔은 전체 인덱스 스캔보다 더 효율적이지만 첫 번째 인덱스 열인 `f1`에 조건이 없기 때문에 이 경우에는 사용할 수 없습니다. 옵티마이저는 느슨한 인덱스 스캔과 유사한 건너뛰기 스캔이라는 방법을 사용하여 `f1`의 각 값에 대해 하나씩 여러 범위 스캔을 수행할 수 있습니다(섹션 8.2.1.17, "GROUP BY 최적화" 참조):

1. 첫 번째 인덱스 부분인 `f1`(인덱스 접두사)의 고유 값 사이를 건너뜁니다.
2. 나머지 인덱스 부분의 `f2 > 40` 조건에 대해 각각의 고유 접두사 값에 대해 하위 범위 스캔을 수행합니다.

앞서 표시된 데이터 집합의 경우 알고리즘은 다음과 같이 작동합니다:

1. 첫 번째 키 부분의 첫 번째 고유값을 가져옵니다(`f1 = 1`).
2. 첫 번째 및 두 번째 키 부분을 기준으로 범위를 구성합니다(`f1 = 1 AND f2 > 40`).
3. 범위 스캔을 수행합니다.
4. 첫 번째 키 부분의 다음 고유값(`F1 = 2`)을 가져옵니다.
5. 첫 번째와 두 번째 키 부분을 기준으로 범위를 구성합니다(`f1 = 2 AND f2 > 40`).
6. 범위 스캔을 수행합니다.

이 전략을 사용하면 MySQL이 구성된 각 범위에 맞지 않는 행을 건너뛰기 때문에 액세스되는 행 수가 줄어듭니다. 이 건너뛰기 스캔 액세스 방법은 다음 조건에서 적용할 수 있습니다:

- 테이블 T에는 [A_1, ..., *`A_k`*,] B_1, ..., *`B_m`*, C [, D_1, ..., *`D_n`*] 형식의 키 부분이 있는 복합 인덱스가 하나 이상 있습니다. 키 부분 A와 D는 비어 있어도 되지만 B와 C는 비어 있지 않아야 합니다.
- 쿼리는 하나의 테이블만 참조합니다.
- 쿼리에서 `GROUP BY` 또는 `DISTINCT`를 사용하지 않습니다.
- 쿼리는 인덱스의 열만 참조합니다.
- A_1, ..., *`A_k`*의 술어는 같음 술어여야 하며 상수여야 합니다. 여기에는 `IN()` 연산자가 포함됩니다.
- 쿼리는 접속사 쿼리, 즉 `OR` 조건의 `AND`여야 합니다: *`(cond1(key_part1) OR cond2(key_part1)) AND (cond1(key_part2) OR ...) AND ...`*
- C에 범위 조건이 있어야 합니다.
- D 열의 조건은 허용됩니다. D의 조건은 C의 범위 조건과 함께 사용해야 합니다.

스캔 건너뛰기 사용은 다음과 같이 `설명` 출력에 표시됩니다:

- 추가 열에서 건너뛰기 스캔에 인덱스를 사용하면 느슨한 인덱스 건너뛰기 스캔 액세스 방법이 사용됨을 나타냅니다.
- 인덱스가 스캔 건너뛰기에 사용될 수 있는 경우, 가능한_키 열에 인덱스가 표시되어야 합니다. 건너뛰기 스캔의 사용은 옵티마이저 추적 출력에서 이 형식의 "건너뛰기 스캔" 요소로 표시됩니다:

```
"SKIP_SCAN_RANGE": {
  "type": "SKIP_SCAN",
  "index": index_used_for_skip_scan,
  "key_parts_used_for_access": [key_parts_used_for_access],
  "range": [범위] : [범위
}
```

"BEST_SKIP_SCAN_요약" 요소도 표시될 수 있습니다. 건너뛰기 스캔이 최상의 범위 액세스 방식으로 선택된 경우, "선택한_범위_액세스_요약"이 기록됩니다. 건너뛰기 스캔이 전반적인 최상의 액세스 방법으로 선택된 경우 "BEST_ACCESS_PATH" 요소가 표시됩니다.

건너뛰기 스캔의 사용은 optimizer_switch 시스템 변수의 skip_scan 플래그 값에 따라 달라집니다. 섹션 8.9.2, "전환 가능한 최적화"를 참조하십시오. 기본적으로 이 플래그는 켜져 있습니다. 비활성화하려면 skip_scan을 off로 설정합니다.

optimizer_switch 시스템 변수를 사용하여 건너뛰기 스캔의 옵티마이저 사용을 세션 전체에서 제어하는 것 외에도 MySQL은 옵티마이저 힌트를 지원하여 문별로 옵티마이저에 영향을 줄 수 있습니다. 섹션 8.9.3, "옵티마이저 힌트"를 참조하십시오.

행 생성자 표현식의 범위 최적화

옵티마이저는 이 형식의 쿼리에 범위 스캔 액세스 방법을 적용할 수 있습니다:

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

이전에는 범위 스캔을 사용하려면 쿼리를 다음과 같이 작성해야 했습니다:

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

옵티마이저가 범위 스캔을 사용하려면 쿼리가 이러한 조건을 충족해야 합니다:

- IN() 술어만 사용되며 NOT IN()은 사용되지 않습니다.
- IN() 술어의 왼쪽에 있는 행 생성자에는 열 참조만 포함됩니다.
- IN() 술어의 오른쪽에 있는 행 생성자에는 실행 중에 상수에 바인딩되는 리터럴 또는 로컬 열 참조인 런타임 상수만 포함됩니다.
- IN() 술어의 오른쪽에는 행 생성자가 두 개 이상 있습니다.

옵티마이저 및 행 생성자에 대한 자세한 내용은 섹션 8.2.1.22, "행 생성자 표현식 최적화"를 참조하십시오.

범위 최적화를 위한 메모리 사용 제한

범위 옵티마이저에서 사용할 수 있는 메모리를 제어하려면 범위_옵티마이저_최대_메모리_크기 시스템 변수입니다:

- 값이 0이면 "제한 없음"을 의미합니다.
- 값이 0보다 크면 옵티마이저는 범위 액세스 방법을 고려할 때 소비되는 메모리를 추적합니다. 지정된 제한을 초과하려고 하면 범위 액세스 방법이 중단되고 대신 전체 테이블 스캔을 비롯한 다른 방법이 고려됩니다. 이 방법은 최적이 아닐 수 있습니다. 이 경우 다음과 같은 경고가 발생합니다(여기서 *N*은 현재 range_optimizer_max_mem_size 값입니다):

```
경고    3170 메모리 용량은 N바이트입니다.
        'range_optimizer_max_mem_size'를 초과했습니다. 이 쿼
        리에 대해 범위 최적화가 수행되지 않았습니다.
```

- `UPDATE` 및 `DELETE` 문의 경우 옵티마이저가 전체 테이블 스캔으로 되돌아가고 `sql_safe_updates` 시스템 변수가 활성화된 경우 사실상 수정할 행을 결정하는 데 키를 사용하지 않으므로 경고 대신 오류가 발생합니다. 자세한 내용은 안전 업데이트 모드 사용(--safe-updates)을 참조하십시오.

사용 가능한 범위 최적화 메모리를 초과하고 옵티마이저가 덜 최적화된 계획으로 되돌아가는 개별 쿼리의 경우 `range_optimizer_max_mem_size` 값을 늘리면 성능이 향상될 수 있습니다.

범위 표현식을 처리하는 데 필요한 메모리 양을 예측하려면 다음 지침을 따르세요:

- 다음과 같은 간단한 쿼리의 경우, 범위 액세스 방법에 대한 후보 키가 하나인 경우 OR과 결합된 각 술어는 약 230바이트를 사용합니다:

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

- 다음과 같은 쿼리도 마찬가지로, AND와 결합된 각 술어는 약 125바이트를 사용합니다:

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

- IN() 술어가있는 쿼리의 경우:

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

  IN() 목록의 각 리터럴 값은 OR과 결합된 술어로 계산됩니다. 두 개의 IN() 목록이 있는 경우 OR과 결합된 술어의 수는 각 목록에 있는 리터럴 값 수의 곱입니다. 따라서 앞의 경우 OR과 결합된 술어의 수는 *M* × *N입니다*.

8.2.1.3 인덱스 병합 최적화

인덱스 병합 액세스 메서드는 여러 범위 스캔으로 행을 검색하고 그 결과를 하나로 병합합니다. 이 액세스 방법은 단일 테이블의 인덱스 스캔만 병합하며 여러 테이블에 걸친 스캔은 병합하지 않습니다. 병합은 기본 스캔의 유니온, 교집합 또는 교집합의 유니온을 생성할 수 있습니다.

인덱스 병합을 사용할 수 있는 쿼리 예제입니다:

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

SELECT * FROM tbl_name
  WHERE (key1 = 10 또는 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);
```

참고

인덱스 병합 최적화 알고리즘에는 다음과 같은 알려진 제한 사항이 있습니다:

- 쿼리에 깊은 AND/또는 중첩이 있는 복잡한 WHERE 절이 있고 MySQL이 최적의 계획을 선택하지 않는 경우 다음 ID 변환을 사용하여 용어를 배포해 보세요:

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

- 인덱스 병합은 전체 텍스트 인덱스에는 적용되지 않습니다.

EXPLAIN 출력에서 인덱스 병합 메서드는 유형 열에 index_merge로 나타납니다. 이 경우 key 열에는 사용된 인덱스 목록이 포함되고 key_len에는 해당 인덱스에 대한 가장 긴 키 부분의 목록이 포함됩니다.

인덱스 병합 액세스 방법에는 몇 가지 알고리즘이 있으며, 이 알고리즘은
설명 출력:

- 교차(...) 사용

- union(...) 사용
- sort_union(...) 사용

다음 섹션에서는 이러한 알고리즘에 대해 자세히 설명합니다. 옵티마이저는 사용 가능한 다양한 옵션의 비용 추정치를 기반으로 여러 가지 가능한 인덱스 병합 알고리즘과 기타 액세스 방법 중에서 선택합니다.

- 인덱스 병합 교차점 액세스 알고리즘
- 인덱스 병합 유니온 액세스 알고리즘
- 인덱스 병합 정렬-조합 액세스 알고리즘
- 인덱스 병합 최적화에 미치는 영향

인덱스 병합 교차점 액세스 알고리즘

이 액세스 알고리즘은 WHERE 절이 AND와 결합된 여러 키의 여러 범위 조건으로 변환되고 각 조건이 다음 중 하나일 때 적용할 수 있습니다:

- 인덱스에 정확히 *N개의* 부분(즉, 모든 인덱스 부분이 포함됨)이 있는 이 형식의 N-부분 표현식입니다:

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

- InnoDB 테이블의 기본 키에 대한 모든 범위 조건입니다. 예시:

```
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;
```

인덱스 병합 교차 알고리즘은 사용된 모든 인덱스에 대해 동시 스캔을 수행하고 병합된 인덱스 스캔에서 수신한 행 시퀀스의 교차를 생성합니다.

쿼리에 사용된 모든 열이 사용된 인덱스에 포함되는 경우 전체 테이블 행이 검색되지 않습니다(이 경우 EXPLAIN 출력에 추가 필드에 인덱스 사용됨이 포함됨). 다음은 이러한 쿼리의 예입니다:

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

사용된 인덱스가 쿼리에 사용된 모든 열을 포함하지 않는 경우, 사용된 모든 키의 범위 조건이 충족되는 경우에만 전체 행이 검색됩니다.

병합된 조건 중 하나가 InnoDB 테이블의 기본 키에 대한 조건인 경우 행 검색에는 사용되지 않지만 다른 조건을 사용하여 검색된 행을 필터링하는 데 사용됩니다.

인덱스 병합 유니온 액세스 알고리즘

이 알고리즘의 기준은 인덱스 병합 교차 알고리즘의 기준과 유사합니다. 이 알고리즘은 테이블의 WHERE 절이 OR과 결합된 서로 다른 키의 여러 범위 조건으로 변환되고 각 조건이 다음 중 하나인 경우에 적용할

수 있습니다:

- 인덱스에 정확히 *N개의* 부분(즉, 모든 인덱스 부분이 포함됨)이 있는 이 형식의 N-부분 표현식입니다:

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

- `InnoDB` 테이블의 기본 키에 대한 모든 범위 조건입니다.

- 인덱스 병합 교집합 알고리즘을 적용할 수 있는 조건입니다.

예시:

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

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

인덱스 병합 정렬-조합 액세스 알고리즘

이 액세스 알고리즘은 `WHERE` 절이 `OR`로 결합된 여러 범위 조건으로 변환되는 경우 적용 가능하지만 인덱스 병합 유니온 알고리즘은 적용되지 않습니다.

예시:

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

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

정렬-합합 알고리즘과 유니온 알고리즘의 차이점은 정렬-합합 알고리즘은 먼저 모든 행의 행 ID를 가져와 정렬한 후 행을 반환해야 한다는 점입니다.

인덱스 병합 최적화에 미치는 영향

인덱스 병합의 사용은 `optimizer_switch` 시스템 변수의 `index_merge`, `index_merge_intersection`, index_merge_union 및 `index_merge_sort_union` 플래그의 값에 따라 달라집니다. 섹션 8.9.2, "전환 가능한 최적화"를 참조하세요. 기본적으로 이 플래그는 모두 켜져 있습니다. 특정 알고리즘만 활성화하려면 `index_merge`를 `off`로 설정하고 다른 알고리즘 중 허용해야 하는 알고리즘만 활성화합니다.

`optimizer_switch` 시스템 변수를 사용하여 인덱스 병합 알고리즘의 옵티마이저 사용을 세션 전체에서 제어하는 것 외에도 MySQL은 옵티마이저 힌트를 지원하여 문 단위로 옵티마이저에 영향을 줄 수 있습니다. 섹션 8.9.3, "옵티마이저 힌트"를 참조하십시오.

8.2.1.4 해시 조인 최적화

기본적으로 MySQL은 가능한 경우 항상 해시 조인을 사용합니다. 해시 조인 사용 여부는 `BNL` 및 `NO_BNL` 옵티마이저 힌트 중 하나를 사용하거나 optimizer_switch 서버 시스템 변수에 대한 설정의 일부로 `block_nested_loop=on` 또는 `block_nested_loop=off`를 설정하여 제어할 수 있습니다.

MySQL은 각 조인에 동일 조인 조건이 있고 이 쿼리와 같이 조인 조건에 적용할 수 있는 인덱스가 없는 모든 쿼리에 대해 해시 조인을 사용합니다:

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

단일 테이블 술어에 사용할 수 있는 인덱스가 하나 이상 있는 경우에도 해시 조인을 사용할 수 있습니다.

해시 조인은 일반적으로 이전 버전의 MySQL에서 사용되는 블록 중첩 루프 알고리즘(블록 중첩 루프 조인 알고리즘 참조)보다 빠르며, 이러한 경우에 사용하도록 고안되었습니다. 블록 중첩 루프에 대한 지원은 MySQL 8.0에서 제거되었으며, MySQL 8.2 서버는 이전에 블록 중첩 루프가 사용되었던 모든 곳에서 해시 조인을 사용합니다.

방금 표시된 예와 이 섹션의 나머지 예에서는 다음 세 테이블이 있다고 가정합니다.
`T1`, `T2` 및 T3은 다음 문을 사용하여 생성되었습니다:

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

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

다음과 같이 EXPLAIN을 사용하면 해시 조인이 사용되고 있음을 확인할 수 있습니다:

```
mysql> 설명
    -> SELECT * FROM t1
    ->     JOIN t2 ON t1.c1=t2.c1\G
*************************** 1. 행 ***************************
           ID: 1
  select_type: SIMPLE 테이
        블: t1
   파티션: NULL
         유형: 모든
가능한_키: NULL
          키: NULL
      key_len: NULL
          참조:
         NULL 행: 1
     필터링됨: 100.00 추가:
        NULL
*************************** 2. 행 ***************************
           ID: 1
  select_type: SIMPLE 테이
        블: t2
   파티션: NULL
         유형: 모든
가능한_키: NULL
          키: NULL
      key_len: NULL
          참조:
         NULL 행: 1
     필터링됨: 100.00
        추가: 사용 위치; 조인 버퍼 사용(해시 조인)
```

분석 설명에는 사용된 해시 조인에 대한 정보도 표시됩니다.

해시 조인은 여기에 표시된 쿼리와 같이 각 테이블 쌍에 대한 조인 조건이 하나 이상 동일 조인인 경우 다중 조인을 포함하는 쿼리에도 사용됩니다:

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

방금 표시된 경우와 같이 내부 조인을 사용하는 경우, 조인이 실행된 후 동일 조인이 아닌 모든 추가 조건이 필터로 적용됩니다. (왼쪽 조인, 세미조인 및 안티조인과 같은 외부 조인의 경우 조인의 일부로 인쇄됩니다.) 이는 EXPLAIN의 출력에서 확인할 수 있습니다:

```
mysql> 설명 형식=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 ***************************
설명: -> 내부 해시 조인 (t3.c1 = t1.c1) (cost=1.05 rows=1)
    -> t3에서 테이블 스캔(비용=0.35 행=1)
    -> 해시
        -> 필터: (t1.c2 < t2.c2) (cost=0.70 rows=1)
            -> Inner 해시 조인(t2.c1 = t1.c1) (비용=0.70 rows=1)
                -> t2에서 테이블 스캔(비용=0.35 행=1)
                -> 해시
                    -> t1에서 테이블 스캔(비용=0.35 행=1)
```

방금 표시된 출력에서도 알 수 있듯이 다중 해시 조인은 여러 개의 동일 조인 조건이 있는 조인에 사용할 수 있으며 실제로도 사용되고 있습니다.

여기에 표시된 것처럼 조인된 테이블 쌍에 하나 이상의 동조인 조건이 없는 경우에도 해시 조인이 사용됩니다:

```
mysql> 설명 형식=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)
    -> 내부 해시 조인(조건 없음)  (비용=1.05 rows=1)
        -> t3에서 테이블 스캔(비용=0.35 행=1)
        -> 해시
            -> Inner 해시 조인(t2.c1 = t1.c1)  (비용=0.70 rows=1)
                -> t2에서 테이블 스캔(비용=0.35행=1)
                -> 해시
                    -> t1에서 테이블 스캔(비용=0.35 행=1)
```

(이 섹션의 뒷부분에 추가 예제가 제공됩니다.)

해시 조인은 다음과 같이 조인 조건이 지정되지 않은 경우, 즉 직교 곱에 대해서도 적용됩니다:

```
mysql> 설명 형식=TREE
    -> SELECT *
    ->     FROM t1
    ->     JOIN t2
    ->     WHERE t1.c2 > 50\G
*************************** 1. row ***************************
설명: -> 내부 해시 조인  (비용=0.70 rows=1)
    -> t2에서 테이블 스캔(비용=0.35 행=1)
    -> 해시
            -> 필터: (t1.c2 > 50)  (cost=0.35 rows=1)
                -> t1에서 테이블 스캔(비용=0.35 행=1)
```

해시 조인을 사용하기 위해 조인에 하나 이상의 동등 조인 조건이 포함될 필요는 없습니다. 즉, 해시 조인을 사용하여 최적화할 수 있는 쿼리 유형에는 다음 목록(예제 포함)에 있는 쿼리 유형이 포함됩니다:

- *내부 비동일 조인*:

```
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)
    -> 내부 해시 조인(조건 없음)  (비용=4.70 행=12)
        -> t2에서 테이블 스캔(비용=0.08행=6)
        -> 해시
            -> t1에서 테이블 스캔(비용=0.85 행=6)
```

- *세미조인*:

```
mysql> EXPLAIN FORMAT=TREE SELECT * FROM t1
    ->     WHERE t1.c1 IN (SELECT t2.c2 FROM t2)\G
*************************** 1. row ***************************
EXPLAIN: -> 해시 세미조인  (t2.c2 = t1.c1)  (cost=0.70 rows=1)
    -> t1에서 테이블 스캔(비용=0.35 행=1)
    -> 해시
        -> t2에서 테이블 스캔(비용=0.35행=1)
```

- *안티조인*:

```
mysql> EXPLAIN FORMAT=TREE SELECT * FROM t2
    ->     WHERE NOT EXISTS (SELECT * FROM t1 WHERE t1.c1 = t2.c1)\G
*************************** 1. row ***************************
EXPLAIN: -> 해시 안티조인 (t1.c1 = t2.c1) (cost=0.70 rows=1)
    -> t2에서 테이블 스캔(비용=0.35 행=1)
    -> 해시
        -> t1에서 테이블 스캔(비용=0.35 행=1)

세트 내 1행, 경고 1건 (0.00초) mysql> SHOW

WARNINGS\G
*************************** 1. 행 *************************** 레벨:
```

참고

```
   코드: 1276
메시지: SELECT #2의 필드 또는 참조 't3.t2.c1'이 SELECT #1에서 해결되었습니다.
```

- *왼쪽 외부 조인*:

```
mysql> EXPLAIN FORMAT=TREE SELECT * FROM t1 LEFT JOIN t2 ON t1.c1 = t2.c1\G
*************************** 1. row ***************************
EXPLAIN: -> 왼쪽 해시 조인 (t2.c1 = t1.c1) (cost=0.70 rows=1)
    -> t1에서 테이블 스캔(비용=0.35 행=1)
    -> 해시
        -> t2에서 테이블 스캔(비용=0.35 행=1)
```

- *오른쪽 외부 조인*(MySQL이 모든 오른쪽 외부 조인을 왼쪽 외부 조인으로 다시 작성하는 것을 관찰합니다):

```
mysql> EXPLAIN FORMAT=TREE SELECT * FROM t1 RIGHT JOIN t2 ON t1.c1 = t2.c1\G
*************************** 1. row ***************************
EXPLAIN: -> 왼쪽 해시 조인 (t1.c1 = t2.c1) (cost=0.70 rows=1)
    -> t2에서 테이블 스캔(비용=0.35행=1)
    -> 해시
        -> t1에서 테이블 스캔(비용=0.35 행=1)
```

기본적으로 MySQL은 가능한 경우 항상 해시 조인을 사용합니다. 해시 조인 사용 여부는 `BNL` 및 `NO_BNL` 옵티마이저 힌트 중 하나를 사용하여 제어할 수 있습니다.

해시 조인에 의한 메모리 사용량은 `join_buffer_size` 시스템 변수를 사용하여 제어할 수 있으며, 해시 조인은 이 양보다 더 많은 메모리를 사용할 수 없습니다. 해시 조인에 필요한 메모리가 사용 가능한 양을 초과하는 경우 MySQL은 디스크의 파일을 사용하여 이 문제를 처리합니다. 이 경우 해시 조인이 메모리에 맞지 않아 `open_files_limit`에 설정된 것보다 더 많은 파일을 생성하는 경우 조인이 성공하지 못할 수 있다는 점에 유의해야 합니다. 이러한 문제를 방지하려면 다음 중 하나를 변경합니다:

- 해시 조인이 디스크로 넘어가지 않도록 `join_buffer_size`를 늘립니다.
- `open_files_limit`을 늘립니다.

해시 조인을 위한 조인 버퍼는 점진적으로 할당되므로 매우 많은 양의 RAM을 할당하는 작은 쿼리 없이 `join_buffer_size`를 더 높게 설정할 수 있지만, 외부 조인은 전체 버퍼를 할당합니다. 해시 조인은 외부 조인(안티조인 및 세미조인 포함)에도 사용되므로 더 이상 문제가 되지 않습니다.

8.2.1.5 엔진 상태 푸시다운 최적화

이 최적화를 통해 인덱싱되지 않은 열과 상수를 직접 비교할 때 효율성이 향상됩니다. 이러한 경우 조건은 평가를 위해 스토리지 엔진으로 "푸시 다운"됩니다. 이 최적화는 `NDB` 저장소 엔진에서만 사용할 수 있습니다.

NDB 클러스터의 경우, 이 최적화를 통해 클러스터의 데이터 노드와 쿼리를 실행한 MySQL 서버 간에 네트워크를 통해 일치하지 않는 행을 전송할 필요가 없으며, 조건 푸시다운이 사용될 수 있지만 사용되지 않는 경우보다 5배에서 10배까지 쿼리 속도를 높일 수 있습니다.

NDB 클러스터 테이블이 다음과 같이 정의되어 있다고 가정합니다:

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

엔진 조건 푸시다운은 여기에 표시된 것과 같은 쿼리와 함께 사용할 수 있으며, 여기에는 인덱싱되지 않은 열과 상수 간의 비교가 포함됩니다:

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

엔진 상태 푸시다운의 사용은 EXPLAIN의 출력에서 확인할 수 있습니다:

```
mysql> EXPLAIN SELECT a, b FROM t1 WHERE b = 10\G
*************************** 1. 행 ***************************
           ID: 1
  select_type: SIMPLE 테이
        블: t1
         유형: 모든
가능한_키: NULL
          키: NULL
      key_len: NULL
          참조:
         NULL 행:
         10
        추가: 푸시 조건으로 where 사용
```

그러나 엔진 조건 *푸시다운은* 다음 쿼리와 함께 사용할 *수 없습니다:*

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

인덱스가 열 a에 존재하므로 엔진 조건 푸시다운은 여기에 적용되지 않습니다. (인덱스 액세스 방법이 더 효율적이므로 조건 푸시다운보다 우선적으로 선택됩니다.)

인덱싱된 열을 > 또는 < 연산자를 사용하여 상수와 비교할 때도 엔진 조건 푸시다운을 사용할 수 있습니다:

```
mysql> EXPLAIN SELECT a, b FROM t1 WHERE a < 2\G
*************************** 1. 행 ***************************
           ID: 1
  select_type: SIMPLE 테이
        블: t1
         유형: 범위 가능_키
: A
          키: 키
      _len: 5
          참조:
         NULL 행: 2
        추가: 푸시 조건으로 where 사용
```

엔진 상태 푸시다운에 대해 지원되는 다른 비교는 다음과 같습니다:

- *열* [NOT] LIKE *패턴*

  패턴은 일치시킬 패턴을 포함하는 문자열 리터럴이어야 합니다(구문은 12.8.1절. "문자열 비교 함수 및 연산자"를 참조하세요).

- *열* IS [NOT] NULL

- *열* IN (*value_list*)

  value_list의 각 항목은 상수 값이어야 하며 리터럴 값이어야 합니다.

- 상수1과 *상수2* 사이의 *열*

constant1과 *constant2는* 각각 상수 리터럴 값이어야 합니다.

이전 목록의 모든 경우에서 조건을 열과 상수 간의 하나 이상의 직접 비교 형식으로 변환할 수 있습니다.

엔진 조건 푸시다운은 기본적으로 활성화되어 있습니다. 서버 시작 시 이 기능을 비활성화하려면 `optimizer_switch` 시스템 변수의 `engine_condition_pushdown` 플래그를 `off`로 설정하세요. 예를 들어, `my.cnf` 파일에서 다음 줄을 사용합니다:

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

런타임에 다음과 같이 조건 푸시다운을 비활성화합니다:

```
SET optimizer_switch='엔진_조건_푸시다운=오프';
```

제한 사항. 엔진 상태 푸시다운에는 다음과 같은 제한 사항이 적용됩니다:

- 엔진 조건 푸시다운은 `NDB` 스토리지 엔진에서만 지원됩니다.
- NDB 8.2에서는 열이 동일한 부호, 길이, 문자 집합, 정밀도 및 배율(해당되는 경우)을 포함하여 정확히 동일한 유형인 경우 열을 서로 비교할 수 있습니다.
- 비교에 사용되는 열은 `BLOB` 또는 `TEXT` 유형일 수 없습니다. 이 예외는 `JSON`, `BIT` 및 `ENUM` 열에도 적용됩니다.
- 열과 비교할 문자열 값은 열과 동일한 데이터 정렬을 사용해야 합니다.
- 조인은 직접 지원되지 않으며, 여러 테이블이 관련된 조건은 가능한 경우 개별적으로 푸시됩니다. 실제로 어떤 조건이 푸시되는지 확인하려면 확장 `EXPLAIN` 출력을 사용하십시오. 섹션 8.8.3, "확장 EXPLAIN 출력 형식"을 참조하십시오.

이전에는 엔진 조건 푸시다운이 조건이 푸시되는 동일한 테이블의 열 값을 참조하는 조건으로 제한되었습니다. NDB 8.2에서는 쿼리 계획의 앞부분에 있는 테이블의 열 값도 푸시된 조건에서 참조할 수 있습니다. 따라서 조인 처리 중에 SQL 노드가 처리해야 하는 행의 수가 줄어듭니다. 필터링은 단일 `mysqld` 프로세스가 아닌 LDM 스레드에서 병렬로 수행할 수도 있습니다. 이렇게 하면 쿼리 성능이 크게 향상될 수 있습니다.

동일한 조인 중첩에서 사용되는 테이블이나 그 위에 있는 조인 중첩의 테이블에 푸시할 수 없는 조건이 없는 경우 `NDB`는 스캔을 사용하여 외부 조인을 푸시할 수 있습니다. 이는 세미조인의 경우에도 마찬가지이며, 사용되는 최적화 전략이 `firstMatch`인 경우입니다(세미조인 변환을 사용하여 IN 및 EXISTS 하위 쿼리 조건 최적화 참조).

다음 두 가지 상황에서는 조인 알고리즘을 이전 테이블의 참조 열과 결합할 수 없습니다:

1. 참조된 이전 테이블 중 하나라도 조인 버퍼에 있는 경우. 이 경우 스캔 필터링된 테이블에서 검색된 각 행은 버퍼의 모든 행과 일치합니다. 즉, 스캔 필터를 생성할 때 열 값을 가져올 수 있는 특정 행이 하나도 없습니다.
2. 열이 푸시된 조인의 하위 작업에서 시작된 경우. 이는 스캔 필터가 생성될 때 조인의 상위 작업에서 참조된 행이 아직 검색되지 않았기 때문입니다.

조인의 조상 테이블에 있는 열은 이전에 나열된 요구 사항을 충족하는 경우 아래로 밀어낼 수 있습니다. 이전에 만든 테이블 `t1`을 사용하는 이러한 쿼리의 예가 여기에 나와 있습니다:

```
mysql> 설명
    ->SELECT    * FROM t1 AS x
    ->LEFT      JOIN t1 AS y
    ->ON  x.a=0 AND y.b>=3\G
*************************** 1. 행 ***************************
           ID: 1
  select_type: SIMPLE 테이
        블: x
   파티션: P0,P1
         유형: 모든
가능한_키: NULL
          키: NULL
      key_len: NULL
         참조:
         NULL 행: 4
     필터링됨: 100.00 추가:
        NULL
*************************** 2. 행 ***************************
           ID: 1
  select_type: SIMPLE 테이
        블: y
```

```
   파티션: p0,p1
          유형: 모든 가능한
_키: NULL
           키: NULL
      key_len: NULL
           참조: NULL 행:
          4
      필터링됨: 100.00
         추가: 어디 사용; 푸시된 조건 사용(`test`.`y`.`b` >= 3); 조인 버퍼 사용(해시 조인)
세트 2줄, 경고 2번(0.00초)
```

8.2.1.6 인덱스 조건 푸시다운 최적화

인덱스 조건 푸시다운(ICP)은 MySQL이 인덱스를 사용하여 테이블에서 행을 검색하는 경우를 위한 최적화 기능입니다. ICP를 사용하지 않으면 스토리지 엔진이 인덱스를 탐색하여 기본 테이블에서 행을 찾은 다음 행에 대한 `WHERE` 조건을 평가하는 MySQL 서버로 반환합니다. ICP가 활성화되어 있고 인덱스의 열만 사용하여 `WHERE` 조건의 일부를 평가할 수 있는 경우, MySQL 서버는 이 부분을 스토리지 엔진으로 푸시합니다. 그러면 스토리지 엔진은 인덱스 항목을 사용하여 푸시된 인덱스 조건을 평가하고 이 조건이 충족되는 경우에만 테이블에서 행을 읽습니다. ICP는 스토리지 엔진이 기본 테이블에 액세스해야 하는 횟수와 MySQL 서버가 스토리지 엔진에 액세스해야 하는 횟수를 줄일 수 있습니다.

인덱스 조건 푸시다운 최적화의 적용 여부는 이러한 조건에 따라 달라집니다:

- ICP는 전체 테이블 행에 액세스해야 하는 경우 범위, `ref`, `eq_ref` 및 `ref_or_null` 액세스 메서드에 사용됩니다.
- ICP는 파티션된 `InnoDB` 및 `MyISAM` 테이블을 포함하여 `InnoDB` 및 `MyISAM` 테이블에 사용할 수 있습니다.
- `InnoDB` 테이블의 경우, ICP는 보조 인덱스에만 사용됩니다. ICP의 목표는 전체 행 읽기 횟수를 줄여 I/O 작업을 줄이는 것입니다. `InnoDB` 클러스터 인덱스의 경우, 전체 레코드가 이미 `InnoDB` 버퍼에 읽혀 있습니다. 이 경우 ICP를 사용해도 I/O가 줄어들지 않습니다.
- 가상 생성 열에 생성된 보조 인덱스에는 ICP가 지원되지 않습니다. `InnoDB` 는 가상으로 생성된 열에 대한 보조 인덱스를 지원합니다.
- 하위 쿼리를 참조하는 조건은 푸시다운할 수 없습니다.
- 저장된 함수를 참조하는 조건은 푸시다운할 수 없습니다. 저장소 엔진은 저장된 함수를 호출할 수 없습니다.
- 트리거된 조건은 푸시다운할 수 없습니다. (트리거된 조건에 대한 자세한 내용은 8.2.2.3절. "EXISTS 전략으로 하위 쿼리 최적화"를 참조하세요.)
- 조건은 시스템 변수에 대한 참조가 포함된 파생 테이블로 푸시다운할 수 없습니다.

이 최적화가 어떻게 작동하는지 이해하려면 먼저 인덱스 조건 푸시다운을 사용하지 않을 때 인덱스 스캔이 어

떻게 진행되는지 살펴보세요:

1. 먼저 인덱스 튜플을 읽어서 다음 행을 가져온 다음, 인덱스 튜플을 사용하여 전체 테이블 행을 찾아서 읽습니다.
2. 이 테이블에 적용되는 `WHERE` 조건의 일부를 테스트합니다. 테스트 결과에 따라 행을 수락하거나 거부합니다.

대신 인덱스 조건 푸시다운을 사용하면 다음과 같이 스캔이 진행됩니다:

1. 다음 행의 인덱스 튜플을 가져옵니다(전체 테이블 행은 아님).
2. 이 테이블에 적용되고 인덱스 열만 사용하여 확인할 수 있는 `WHERE` 조건의 일부를 테스트합니다. 조건이 충족되지 않으면 다음 행의 인덱스 튜플로 진행합니다.
3. 조건이 충족되면 인덱스 튜플을 사용하여 전체 테이블 행을 찾아서 읽습니다.

4. 이 테이블에 적용되는 `WHERE` 조건의 나머지 부분을 테스트합니다. 테스트 결과에 따라 행을 수락하거나 거부합니다.

인덱스 조건 푸시다운이 사용된 경우 `추가` 열에 인덱스 `조건 사용이 설명` 출력에 표시됩니다. 전체 테이블 행을 읽어야 하는 경우에는 적용되지 않으므로 `인덱스` 사용은 표시되지 않습니다.

테이블에 사람과 주소에 대한 정보가 포함되어 있고 테이블에 `INDEX(우편번호, 성, 이름)`로 정의된 인덱스가 있다고 가정해 보겠습니다. 어떤 사람의 `우편번호` 값은 알고 있지만 성을 모르는 경우 다음과 같이 검색할 수 있습니다:

```
SELECT * FROM people
  WHERE zipcode='95054'
  그리고 성이 '%에트루니아%' 같은 경우
  AND 주소 '%메인스트리트%'와 같은 주소를 입력합니다;
```

MySQL은 이 인덱스를 사용하여 `우편번호가 '95054'인` 사람을 검색할 수 있습니다. 두 번째 부분(`성 '%etrunia%'처럼`)은 스캔해야 하는 행 수를 제한하는 데 사용할 수 없으므로 인덱스 조건 푸시다운이 없으면 이 쿼리는 `zipcode='95054'인` 모든 사람에 대한 전체 테이블 행을 검색해야 합니다.

인덱스 조건 푸시다운을 사용하면 MySQL은 전체 테이블 행을 읽기 전에 성이 `'%etrunia%'와 같은` 부분을 확인합니다. 이렇게 하면 `우편 번호 조건은` 일치하지만 `성` 조건은 일치하지 않는 인덱스 튜플에 해당하는 전체 행을 읽는 것을 방지할 수 있습니다.

인덱스 조건 푸시다운은 기본적으로 활성화되어 있습니다. `최적화_스위치로` 제어할 수 있습니다.
시스템 변수에 `인덱스_조건_푸시다운` 플래그를 설정합니다:

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

섹션 8.9.2, '전환 가능한 최적화'를 참조하세요.

8.2.1.7 중첩 루프 조인 알고리즘

MySQL은 중첩 루프 알고리즘 또는 이를 변형한 알고리즘을 사용하여 테이블 간 조인을 실행합니다.

- 중첩 루프 조인 알고리즘
- 블록 중첩 루프 조인 알고리즘 차단

중첩 루프 조인 알고리즘

단순 중첩 루프 조인(NLJ) 알고리즘은 루프의 첫 번째 테이블에서 한 번에 하나씩 행을 읽고 각 행을 조인의 다음 테이블을 처리하는 중첩 루프에 전달합니다. 이 프로세스는 조인할 테이블이 남아 있는 동안 여러 번 반복됩니다.

다음 조인 유형을 사용하여 세 테이블 `t1`, `t2` 및 `t3` 간의 조인을 실행한다고 가정합니다:

```
테이블    조인 유형
t1        범위
t2        ref
t3        ALL
```

간단한 NLJ 알고리즘을 사용하는 경우 조인은 다음과 같이 처리됩니다:

```
T1 일치 범위의 각 행에 대해 {
  T2의 각 행이 참조 키와 일치하는 경우 { T3의 각 행이 일
    치하는 경우 {
      행이 조인 조건을 만족하면 클라이언트로 전송합니다.
    }
  }
}
```

NLJ 알고리즘은 외부 루프에서 내부 루프로 한 번에 하나씩 행을 전달하기 때문에 일반적으로 내부 루프에서 처리된 테이블을 여러 번 읽습니다.

블록 중첩 루프 조인 알고리즘 차단

블록 중첩 루프(BNL) 조인 알고리즘은 외부 루프에서 읽은 행의 버퍼링을 사용하여 내부 루프에서 테이블을 읽어야 하는 횟수를 줄입니다. 예를 들어, 10개의 행을 버퍼로 읽고 버퍼를 다음 내부 루프로 전달하면 내부 루프에서 읽은 각 행을 버퍼의 모든 10개의 행과 비교할 수 있습니다. 이렇게 하면 내부 테이블을 읽어야 하는 횟수가 크게 줄어듭니다.

MySQL은 인덱스를 사용할 수 없는 경우 이퀄 조인을 위해 해시 조인 최적화를 사용합니다. MySQL

조인 버퍼링에는 이러한 특성이 있습니다:

- 조인 버퍼링은 조인이 `ALL` 또는 인덱스 유형(즉, 사용 가능한 키를 사용할 수 없고 데이터 또는 인덱스 행에 대해 각각 전체 스캔이 수행되는 경우) 또는 범위인 경우에 사용할 수 있습니다. 버퍼링 사용은 8.2.1.12절. "중첩-루프 및 배치 키 액세스 조인 차단"에 설명된 대로 외부 조인에도 적용할 수 있습니다.
- 조인 버퍼는 첫 번째 비상수 테이블에 대해 할당되지 않으며, 이 테이블이 `ALL` 유형이거나 색인.
- 조인 버퍼에는 전체 행이 아닌 조인에 관심 있는 열만 저장됩니다.
- `join_buffer_size` 시스템 변수는 쿼리를 처리하는 데 사용되는 각 조인 버퍼의 크기를 결정합니다.
- 버퍼링할 수 있는 각 조인마다 하나의 버퍼가 할당되므로 주어진 쿼리는 여러 조인 버퍼를 사용하여 처리될 수 있습니다.
- 조인 버퍼는 조인을 실행하기 전에 할당되고 쿼리가 완료된 후에 해제됩니다.

앞서 설명한 NLJ 알고리즘에 대한 조인 예제(버퍼링 없음)의 경우 조인 버퍼링을 사용하여 다음과 같이 조인이 수행됩니다:

```
T1 일치 범위의 각 행에 대해 {
  참조 키와 일치하는 T2의 각 행에 대해 {버퍼가 가득 차면 조
    인 버퍼에 T1, T2에서 사용한 열을 저장합니다.
      T3의 각 행에 대해 {
        조인 버퍼의 각 t1, t2 조합에 대해 {
          행이 조인 조건을 만족하면 클라이언트로 전송합니다.
        }
      }
      빈 조인 버퍼
    }
  }
}

버퍼가 비어있지 않은 경우 {
  T3의 각 행에 대해 {
    조인 버퍼의 각 t1, t2 조합에 대해 {
      행이 조인 조건을 만족하면 클라이언트로 전송합니다.
    }
  }
}
```

S는 조인 버퍼에 저장된 각 `t1`, `t2` 조합의 크기이고 C는 버퍼에 있는 조합의 수이면 테이블 t3이 스캔되는 횟수는 다음과 같습니다:

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

조인 버퍼 크기가 이전의 모든 행 조합을 담을 수 있을 정도로 커질 때까지 조인 버퍼 크기가 커질수록 `t3` 스캔 횟수가 감소합니다. 이 시점에서는 더 크게 설정해도 속도가 향상되지 않습니다.

8.2.1.8 중첩된 조인 최적화

조인을 표현하는 구문은 중첩 조인을 허용합니다. 다음 설명은 섹션 13.2.13.2, "JOIN 절"에 설명된 조인 구문을 참조합니다.

table_factor의 구문은 SQL 표준과 비교하여 확장되었습니다. 후자는 한 쌍의 괄호 안에 있는 *테이블 참조* 목록이 아닌 *테이블 참조만* 허용합니다. 테이블_참조 항목 목록의 각 쉼표를 내부 조인과 동등한 것으로 간주하는 경우 이는 보수적인 확장입니다. 예를 들어

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

와 동일합니다:

```
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)
```

MySQL에서 교차 조인은 구문적으로 내부 조인과 동일하며 서로를 대체할 수 있습니다. 표준 SQL에서는 서로 동등하지 않습니다. INNER JOIN은 ON 절과 함께 사용되며, CROSS JOIN은 그렇지 않은 경우에 사용됩니다.

일반적으로 내부 조인 연산만 포함된 조인 식에서는 괄호를 무시할 수 있습니다. 이 조인 식을 살펴보겠습니다:

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

괄호를 제거하고 왼쪽으로 그룹화 작업을 수행하면 해당 조인 식이 이 식으로 변환됩니다:

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

하지만 두 표현식은 동일하지 않습니다. 이를 확인하기 위해 테이블 t1, t2 및 t3의 상태가 다음과 같다고 가정합니다:

- 테이블 t1에는 행 (1), (2)가 포함됩니다.
- 테이블 t2에 행 (1,101)이 포함됩니다.
- 표 t3에 행 (101)이 포함됩니다.

이 경우 첫 번째 표현식은 (1,1,101,101), (2,NULL,NULL,NULL 행을 포함한 결과 집합을 반환하는 반면, 두 번째 표현식은 (1,1,101,101), (2,NULL,NULL,101) 행을 반환합니다:)

```
mysql> SELECT *
       FROM t1
            왼쪽 조인
            (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 | 널 | 널 | 널 | 널 | 널
+------+------+------+--------+

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

```
+------+------+------+--------+
|    1 |    1 | 101  | 101    |
|    2 | 널   | 널   | 101    |
+------+------+------+--------+
```

다음 예에서는 외부 조인 작업이 내부 조인 작업과 함께 사용됩니다:

```
t1 좌측 조인 (t2, t3) ON t1.a=t2.a
```

해당 표현식은 다음 표현식으로 변환할 수 없습니다:

```
t1 좌 조인 t2 ON t1.a=t2.a, t3
```

주어진 테이블 상태에 대해 두 표현식은 서로 다른 행 집합을 반환합니다:

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

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

따라서 외부 조인 연산자를 사용하는 조인 식에서 괄호를 생략하면 원래 식의 결과 집합이 변경될 수 있습니다.

보다 정확하게는 왼쪽 외부 조인 작업의 오른쪽 피연산자와 오른쪽 조인 작업의 왼쪽 피연산자에서 괄호를 무시할 수 없습니다. 즉, 외부 조인 작업의 내부 테이블 식에 괄호를 무시할 수 없습니다. 다른 피연산자(외부 테이블의 피연산자)에 대한 괄호는 무시할 수 있습니다.

다음 표현식입니다:

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

모든 테이블 `t1,t2,t3` 및 특성 `t2.b` 및 `t3.b`:

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

조인 식(*`joined_table`*)에서 조인 연산의 실행 순서가 왼쪽에서 오른쪽이 아닌 경우 중첩 조인에 대해 이야기합니다. 다음 쿼리를 살펴보겠습니다:

```
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
```

이러한 쿼리는 이러한 중첩 조인을 포함하는 것으로 간주됩니다:

```
t2 좌 조인 t3 ON t2.b=t3.b
T2, T3
```

첫 번째 쿼리에서 중첩 조인은 왼쪽 조인 연산으로 형성됩니다. 두 번째 쿼리에서는 내부 조인 연산으로 중첩 조인이 형성됩니다.

첫 번째 쿼리에서는 괄호를 생략할 수 있습니다. 조인 식의 문법 구조는 조인 연산에 대해 동일한 실행 순서를 지정합니다. 두 번째 쿼리의 경우 괄호를 생략할 수 없지만 여기서의 조인 식은 괄호 없이도 명확하게 해석할 수 있습니다. 확장 구문에서는 이론적으로는 괄호 없이도 쿼리를 구문 분석할 수 있지만 두 번째 쿼리의 `(t2, t3)`에 괄호가 필요합니다: `LEFT JOIN`과 `ON`이 표현식 `(t2,t3)`의 왼쪽 및 오른쪽 구분자 역할을 하기 때문에 쿼리의 구문 구조는 여전히 명확합니다.

앞의 예는 이러한 점을 잘 보여줍니다:

- 외부 조인이 아닌 내부 조인만 포함하는 조인 식의 경우 괄호를 제거하고 조인을 왼쪽에서 오른쪽으로 평가할 수 있습니다. 실제로 테이블은 어떤 순서로든 평가할 수 있습니다.
- 일반적으로 외부 조인 또는 외부 조인과 내부 조인이 혼합된 외부 조인의 경우에는 그렇지 않습니다. 괄호를 제거하면 결과가 변경될 수 있습니다.

중첩된 외부 조인이 있는 쿼리는 내부 조인이 있는 쿼리와 동일한 파이프라인 방식으로 실행됩니다. 더 정확하게는 중첩 루프 조인 알고리즘의 변형이 활용됩니다. 중첩 루프 조인이 쿼리를 실행하는 알고리즘을 기억해 보십시오(섹션 8.2.1.7, "중첩 루프 조인 알고리즘" 참조). 3개의 테이블 `T1,T2,T3`에 대한 조인 쿼리의 형식이 다음과 같다고 가정합니다:

```
SELECT * FROM T1 INNER JOIN T2 ON A1(T1,T2)
                 INNER JOIN T3 ON P2(T2,T3)
  WHERE P(T1,T2,T3)
```

여기서 `P1(T1,T2)` 및 `P2(T3,T3)`는 (표현식에 대한) 일부 조인 조건이며, `P(T1,T2,T3)`는 다음과 같습니다. 는 테이블 `T1,T2,T3`의 열에 대한 조건입니다.

중첩 루프 조인 알고리즘은 다음과 같은 방식으로 이 쿼리를 실행합니다:

```
T1의 각 행 t1에 대해 {
  T2의 각 행 t2에 대해 P1(t1,t2) { T3의 각 행 t3에
    대해 P2(t2,t3) {
      IF P(t1,t2,t3) {
         t:=t1||t2||t3; OUTPUT t;
      }
    }
  }
}
```

`t1||t2||t3` 표기법은 `t1`, `t2`, `t3` 행의 열을 연결하여 구성된 행을 나타냅니다. 다음 예제 중 일부에서 테이블 이름이 표시된 곳의 `NULL`은 해당 테이블의 각 열에 NULL이 사용된 행을 의미합니다. 예를 들어, `t1||t2||NULL`은 행 t1과 `t2`의 열을 연결하여 구성된 행을 나타내며, `t3`의 각 열에 대해 `NULL`을 사용합니다. 이러한 행을 `NULL`로 보완된 행이라고 합니다.

이제 중첩된 외부 조인이 있는 쿼리를 살펴보겠습니다:

```
T1 좌측 조인에서 * 선택
               (T2 LEFT JOIN T3 ON P2(T2,T3))
               ON A1(T1,T2)
  여기서 p(t1,t2,t3)
```

이 쿼리의 경우 중첩 루프 패턴을 수정하여 얻습니다:

```
T1의 각 행 t1에 대해 { BOOL
  f1:=FALSE;
  T2의 각 행 t2에 대해 P1(t1,t2) { BOOL f2:=FALSE;
    T3의 각 행 t3에 대해 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;
    }
  }
}
```

일반적으로 외부 조인 작업에서 첫 번째 내부 테이블에 대한 중첩된 루프에 대해 루프 전에 꺼져 있고 루프 후에 확인되는 플래그가 도입됩니다. 플래그는 외부 테이블의 현재 행에 대해 내부 피연산자를 나타내는 테이블에서 일치하는 항목이 발견되면 켜집니다. 루프 사이클이 끝날 때 플래그가 여전히 꺼져 있으면 외부 테이블의 현재 행에 대한 일치 항목을 찾지 못한 것입니다. 이 경우 행은 내부 테이블의 열에 대해 `NULL` 값으로 보완됩니다. 결과 행은 출력에 대한 최종 검사 또는 다음 중첩 루프로 전달되지만 행이 포함된 모든 외부 조인의 조인 조건을 충족하는 경우에만 전달됩니다.

이 예제에서는 다음 표현식으로 표현되는 외부 조인 테이블이 내장되어 있습니다:

```
(T2 좌측 조인 T3 ON P2(T2,T3))
```

내부 조인이 있는 쿼리의 경우, 옵티마이저는 다음과 같이 중첩된 루프의 다른 순서를 선택할 수 있습니다:

```
T3의 각 행 t3에 대해 {
  T2의 각 행 t2에 대해 P2(t2,t3) { T1의 각 행 t1에
    대해 P1(t1,t2) {
      IF P(t1,t2,t3) {
         t:=t1||t2||t3; OUTPUT t;
      }
    }
  }
}
```

외부 조인이 있는 쿼리의 경우 옵티마이저는 외부 테이블에 대한 루프가 내부 테이블에 대한 루프보다 앞서는 순서만 선택할 수 있습니다. 따라서 외부 조인이 있는 쿼리의 경우 중첩 순서는 하나만 가능합니다. 다음 쿼리의 경우 옵티마이저는 두 개의 서로 다른 중첩을 평가합니다. 두 중첩 모두에서 T1은 외부 조인에 사용되므로 외부 루프에서 처리되어야 합니다. `T2` 및 T3은 내부 조인에 사용되므로 해당 조인은 내부 루프에서 처리되어야 합니다. 그러나 조인이 내부 조인이므로 `T2` 및 `T3`은 어느 순서로든 처리할 수 있습니다.

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

하나의 중첩은 `T2`를 평가한 다음 `T3`를 평가합니다:

```
T1의 각 행 t1에 대해 { BOOL
  f1:=FALSE;
  T2의 각 행 t2에 대해 P1(t1,t2) { T3의 각 행 t3에
    대해 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;
    }
  }
```

```
}
```

다른 중첩은 T3을 평가한 다음 T2를 평가합니다:

```
T1의 각 행 t1에 대해 { BOOL
  f1:=FALSE;
  T3의 각 행 t3에 대해 P2(t1,t3) { T2의 각 행 t2에
    대해 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;
    }
  }
}
```

내부 조인을 위한 중첩 루프 알고리즘에 대해 설명할 때 쿼리 실행 성능에 큰 영향을 미칠 수 있는 몇 가지 세부 사항을 생략했습니다. 소위 "푸시다운" 조건에 대해서는 언급하지 않았습니다. WHERE 조건 P(T1,T2,T3)를 접속식으로 표현할 수 있다고 가정해 보겠습니다:

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

이 경우 MySQL은 실제로 다음과 같은 중첩 루프 알고리즘을 사용하여 내부 조인이 있는 쿼리를 실행합니다:

```
T1의 각 행 t1에 대해 C1(t1) {
  T2의 각 행 t2에 대해 P1(t1,t2) AND C2(t2) { T3의 각 행 t3에
    대해 P2(t2,t3) AND C3(t3) {
      IF P(t1,t2,t3) {
         t:=t1||t2||t3; OUTPUT t;
      }
    }
  }
}
```

각 접속사 C1(T1), C2(T2), C3(T3)이 가장 안쪽 루프에서 평가할 수 있는 가장 바깥쪽 루프로 밀려난 것을 볼 수 있습니다. C1(T1)이 매우 제한적인 조건인 경우, 이 조건 푸시다운은 테이블 T1에서 내부 루프로 전달되는 행의 수를 크게 줄일 수 있습니다. 결과적으로 쿼리 실행 시간이 크게 향상될 수 있습니다.

외부 조인이 있는 쿼리의 경우 외부 테이블의 현재 행이 내부 테이블에 일치하는 항목이 있는 것으로 확인된 후에만 WHERE 조건을 확인해야 합니다. 따라서 내부 중첩 루프에서 조건을 푸시하는 최적화를 외부 조인이 있는 쿼리에 직접 적용할 수 없습니다. 여기에서는 일치하는 항목이 발견되었을 때 켜지는 플래그로 보호되는 조건 푸시 다운 술어를 도입해야 합니다.

외부 조인이 있는 이 예제를 기억해 보십시오:

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

이 예제에서 가드 푸시다운 조건을 사용하는 중첩 루프 알고리즘은 다음과 같습니다:

```
T1의 각 행 t1에 대해 C1(t1) { BOOL
  f1:=FALSE;
  T2의 각 행 t2에 대해
      P1(t1,t2) AND (f1?C2(t2):TRUE) { BOOL
    f2:=FALSE;
    T3의 각 행 t3에 대해
        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;
  }
}
```

일반적으로 푸시다운 조건은 `P1(T1,T2)` 및 `P(T2,T3)`와 같은 조인 조건에서 추출할 수 있습니다. 이 경우 푸시다운된 조건은 해당 외부 조인 연산에 의해 생성된 `NULL`로 보완된 행에 대한 조건을 확인하지 못하도록 하는 플래그로도 보호됩니다.

동일한 중첩 조인에서 한 내부 테이블에서 다른 테이블로 키에 의한 액세스가 `WHERE` 조건의 술어에 의해 유도되는 경우 금지됩니다.

8.2.1.9 외부 조인 최적화

외부 조인에는 왼쪽 조인 및 오른쪽 조인이 포함됩니다.

MySQL은 다음과 같이 *A* 좌측 조인 *B* *조인_명세를* 구현합니다:

- 테이블 B는 테이블 *A* 및 *A가 종속된* 모든 테이블에 종속되도록 설정됩니다.
- 테이블 A는 왼쪽 조인 조건에 사용되는 모든 테이블(*B* 제외)에 종속되도록 설정됩니다.
- `LEFT JOIN` 조건은 테이블 *B에서* 행을 검색하는 방법을 결정하는 데 사용됩니다(즉, `WHERE` 절의 모든 조건은 사용되지 않습니다).
- 모든 표준 조인 최적화가 수행되지만 테이블이 종속된 모든 테이블을 항상 읽은 후에 테이블을 읽는다는 점을 제외하면 모든 표준 조인 최적화가 수행됩니다. 순환 종속성이 있는 경우 오류가 발생합니다.
- 모든 표준 `WHERE` 최적화가 수행됩니다.
- *A에* `WHERE` 절과 일치하는 행이 있지만 *B에* `ON`과 일치하는 행이 없는 경우 조건으로 설정하면 모든 열이 `NULL`로 설정된 추가 *B* 행이 생성됩니다.
- 일부 테이블에 존재하지 않는 행을 찾기 위해 `LEFT JOIN`을 사용하는 경우 다음과 같은 테스트가 있습니다. `WHERE` 부분의 *col_name* `IS NULL`(여기서 *col_name*은 `NOT NULL`로 선언된 열입니다.) MySQL은 `LEFT JOIN` 조건과 일치하는 행 하나를 찾은 후 더 이상의 행 검색(특정 키 조합에 대한)을 중지합니다.

오른쪽 조인 구현은 테이블 역할이 반전된 왼쪽 조인의 구현과 유사합니다. 오른쪽 조인은 섹션 8.2.1.10, "외부 조인 단순화"에 설명된 대로 동등한 왼쪽 조인으로 변환됩니다.

왼쪽 조인의 경우 생성된 `NULL` 행에 대해 `WHERE` 조건이 항상 false인 경우 왼쪽 조인은 내부 조인으로 변경됩니다. 예를 들어 `t2.column1`이 `NULL`인 경우 다음 쿼리에서 `WHERE` 절은 거짓이 됩니다:

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

따라서 쿼리를 Inner 조인으로 변환하는 것이 안전합니다:

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

상수 리터럴 표현식에서 발생하는 사소한 `WHERE` 조건은 최적화의 후반 단계가 아니라 준비 중에 제거되며, 이때 조인이 이미 간소화됩니다. 이전

사소한 조건을 제거하면 옵티마이저가 외부 조인을 내부 조인으로 변환할 수 있으므로, 이 쿼리와 같이 `WHERE` 절에 사소한 조건이 포함된 외부 조인이 있는 쿼리에 대한 계획이 개선될 수 있습니다:

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

이제 옵티마이저는 준비 중에 0 = 1이 항상 거짓이므로 `OR 0 = 1`이 중복된다는 것을 확인하고 이를 제거하여 이 값을 남깁니다:

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

이제 옵티마이저는 다음과 같이 내부 조인으로 쿼리를 다시 작성할 수 있습니다:

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

이제 옵티마이저는 더 나은 쿼리 계획을 생성하는 경우 테이블 `t1`보다 테이블 `t2`를 먼저 사용할 수 있습니다. 테이블 조인 순서에 대한 힌트를 제공하려면 옵티마이저 힌트를 사용하십시오(섹션 8.9.3, "옵티마이저 힌트" 참조). 또는 `STRAIGHT_JOIN`을 사용할 수 있습니다(섹션 13.2.13, "SELECT 문"을 참조). 그러나,
STRAIGHT_JOIN은 세미조인 변환을 비활성화하므로 인덱스가 사용되지 않을 수 있습니다(세미조인 변환으로 IN 및 EXISTS 하위 쿼리 조건 최적화를 참조하세요).

8.2.1.10 외부 조인 간소화

쿼리의 `FROM` 절에 있는 테이블 표현식은 많은 경우 간소화됩니다.

구문 분석기 단계에서 오른쪽 외부 조인 연산이 포함된 쿼리는 왼쪽 조인 연산만 포함된 동등한 쿼리로 변환됩니다. 일반적인 경우 오른쪽 조인이 되도록 변환이 수행됩니다:

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

이와 동등한 왼쪽 조인이 됩니다:

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

`T1 INNER JOIN T2 ON P(T1,T2)` 형식의 모든 내부 조인 식은 `WHERE` 조건(또는 포함 조인의 조인 조건이 있는 경우 해당 조건)에 대한 연결로 조인되는 목록 `T1,T2`, `P(T1,T2)`로 대체됩니다.

옵티마이저는 외부 조인 작업에 대한 계획을 평가할 때 이러한 각 작업에 대해 외부 테이블이 내부 테이블보다 먼저 액세스되는 계획만 고려합니다. 이러한 계획만 중첩 루프 알고리즘을 사용하여 외부 조인을 실행할 수 있으므로 옵티마이저 선택이 제한됩니다.

`R(T2)`가 테이블에서 일치하는 행의 수를 크게 좁히는 이 형식의 쿼리를 예로 들어 보겠습니다.
`T2`:

```
T1에서 * T1 선택
  좌 조인 T2 ON A1(T1,T2) WHERE
  P(T1,T2) AND R(T2)
```

쿼리가 작성된 대로 실행되면 옵티마이저는 제한이 덜한 테이블에 액세스할 수밖에 없습니다.

T1보다 더 제한적인 테이블 `T2`를 사용하면 매우 비효율적인 실행 계획이 생성될 수 있습니다.

대신 MySQL은 `WHERE` 조건이 null 거부되는 경우 쿼리를 외부 조인 연산이 없는 쿼리로 변환합니다. (즉, 외부 조인을 내부 조인으로 변환합니다.) 조건이 외부 조인 연산에 대해 생성된 모든 `NULL`로 보완된 행에 대해 `FALSE` 또는 `UNKNOWN`으로 평가되는 경우 조건이 외부 조인 연산에서 Null 거부되었다고 합니다.

따라서 이 외부 조인의 경우:

```
T1 좌측 조인 T2 ON T1.A=T2.A
```

이러한 조건은 NULL로 보완된 행(T2 열이 NULL로 설정된 경우)에 대해 참이 될 수 없으므로 Null 거부됩니다:

```
T2.B가 널이 아님 T2.B
> 3
T2.C <= T1.C
T2.B < 2 또는 T2.C > 1
```

이와 같은 조건은 NULL로 보완된 행에 해당할 수 있으므로 Null 거부되지 않습니다:

```
T2.B는 NULL입니다.
T1.B < 3 또는 T2.B가 널이 아님
T1.B < 3 또는 T2.B > 3
```

외부 조인 작업에 대해 조건이 null 거부되는지 확인하는 일반적인 규칙은 간단합니다:

- 여기서 A는 내부 테이블의 속성 중 하나이며, A IS NOT NULL 형식입니다.
- 인자 중 하나가 NULL일 때 UNKNOWN으로 평가되는 내부 테이블에 대한 참조가 포함된 술어입니다.
- 접속사로 널 거부 조건을 포함하는 접속사입니다.
- 무효로 거부된 조건의 분리입니다.

쿼리에서 한 외부 조인 연산에 대해 조건이 거부될 수 있고 다른 연산에 대해 거부되지 않을 수 있습니다. 이 쿼리에서 WHERE 조건은 두 번째 외부 조인 연산에 대해 Null 거부되지만 첫 번째 연산에 대해서는 Null 거부되지 않습니다:

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

쿼리에서 외부 조인 작업에 대해 WHERE 조건이 null 거부된 경우 외부 조인 작업은 내부 조인 작업으로 대체됩니다.

예를 들어, 앞의 쿼리에서 두 번째 외부 조인은 null 거부되며 내부 조인으로 대체할 수 있습니다:

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

원래 쿼리의 경우 옵티마이저는 단일 테이블 액세스 순서와 호환되는 플랜만 평가합니다.
T1,T2,T3. 재작성된 쿼리의 경우 액세스 순서 T3,T1,T2를 추가로 고려합니다.

하나의 외부 조인 작업의 변환이 다른 작업의 변환을 트리거할 수 있습니다. 따라서 쿼리:

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

가 먼저 쿼리로 변환됩니다:

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

쿼리와 동일합니다:

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

나머지 외부 조인 작업은 다음과 같은 조건 때문에 내부 조인으로 대체할 수도 있습니다.

T3.B=T2.B는 널 거부됩니다. 이렇게 하면 외부 조인이 전혀 없는 쿼리가 생성됩니다:

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

옵티마이저가 내장된 외부 조인 작업을 대체하는 데는 성공하지만 내장된 외부 조인을 변환할 수 없는 경우가 있습니다. 다음 쿼리입니다:

```
T1 좌측 조인에서 * 선택
               (T2 좌측 조인 T3 ON T3.B=T2.B) ON
               T2.A=T1.A
  여기서 T3.C > 0
```

로 변환됩니다:

```
T1 좌측 조인에서 * 선택
               (T2 내부 조인 T3 ON T3.B=T2.B) ON
               T2.A=T1.A
  여기서 T3.C > 0
```

이는 여전히 임베딩 외부 조인 연산을 포함하는 양식으로만 다시 작성할 수 있습니다:

```
T1 좌측 조인에서 * 선택
               (T2,T3)
               ON (T2.A=T1.A 및 T3.B=T2.B) 여
  기서 T3.C > 0
```

쿼리에서 내장된 외부 조인 연산을 변환하려는 모든 시도는 WHERE 조건과 함께 내장된 외부 조인에 대한 조인 조건을 고려해야 합니다. 이 쿼리에서 WHERE 조건은 내장된 외부 조인에 대해 null 거부되지 않지만, 내장된 외부 조인 T2.A=T1.A 및 T3.C=T1.C의 조인 조건은 null 거부됩니다:

```
T1 좌측 조인에서 * 선택
               (T2 LEFT JOIN T3 ON T3.B=T2.B)
               ON T2.A=T1.A AND T3.C=T1.C
  여기서 t3.d > 0 또는 t1.d > 0
```

따라서 쿼리를 다음과 같이 변환할 수 있습니다:

```
T1 좌측 조인에서 * 선택
               (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 다중 범위 읽기 최적화

테이블이 크고 스토리지 엔진의 캐시에 저장되지 않은 경우 보조 인덱스에서 범위 스캔을 사용하여 행을 읽으면 기본 테이블에 대한 많은 임의 디스크 액세스가 발생할 수 있습니다. 디스크 스윕 다중 범위 읽기(MRR) 최적화를 사용하면 MySQL은 먼저 인덱스만 스캔하고 관련 행의 키를 수집하여 범위 스캔을 위한 임의 디스크 액세스 횟수를 줄이려고 시도합니다. 그런 다음 키를 정렬하고 마지막으로 기본 키의 순서를 사용하여 기본 테이블에서 행을 검색합니다. 디스크 스윕 MRR의 동기는 무작위 디스크 액세스 횟수를 줄이고 대신 기본 테이

블 데이터를 보다 순차적으로 스캔하기 위한 것입니다.

다중 범위 읽기 최적화는 이러한 이점을 제공합니다:

- MRR을 사용하면 인덱스 튜플을 기반으로 데이터 행을 임의의 순서가 아닌 순차적으로 액세스할 수 있습니다. 서버는 쿼리 조건을 만족하는 인덱스 튜플 집합을 가져와서 정렬합니다.
  데이터 행 ID 순서에 따라 정렬된 튜플을 사용하여 데이터 행을 순서대로 검색합니다. 따라서 데이터 액세스가 더 효율적이고 비용이 적게 듭니다.

- MRR을 사용하면 조인 속성에 인덱스를 사용하는 범위 인덱스 스캔 및 이퀴 조인과 같이 인덱스 튜플을 통해 데이터 행에 액세스해야 하는 작업에 대한 키 액세스 요청을 일괄 처리할 수 있습니다. MRR은 인덱스 범위 시퀀스를 반복하여 적격 인덱스 튜플을 얻습니다. 다음과 같이

결과가 누적되면 해당 데이터 행에 액세스하는 데 사용됩니다. 데이터 행 읽기를 시작하기 전에 모든 인덱스 튜플을 획득할 필요는 없습니다.

가상으로 생성된 열에 생성된 보조 인덱스에서는 MRR 최적화가 지원되지 않습니다. `InnoDB`는 가상으로 생성된 열에 대한 보조 인덱스를 지원합니다.

다음 시나리오는 MRR 최적화가 유리할 수 있는 경우를 보여줍니다:

시나리오 A: MRR은 인덱스 범위 스캔 및 이퀄 조인 작업을 위해 `InnoDB` 및 `MyISAM` 테이블에 사용할 수 있습니다.

1. 인덱스 튜플의 일부가 버퍼에 누적됩니다.
2. 버퍼의 튜플은 데이터 행 ID를 기준으로 정렬됩니다.
3. 데이터 행은 정렬된 인덱스 튜플 시퀀스에 따라 액세스됩니다.

시나리오 B: MRR은 다중 범위 인덱스 스캔을 위해 또는 속성에 의한 동조인을 수행할 때 `NDB` 테이블에 사용할 수 있습니다.

1. 쿼리가 제출되는 중앙 노드의 버퍼에 범위의 일부(단일 키 범위일 수 있음)가 누적됩니다.
2. 범위는 데이터 행에 액세스하는 실행 노드로 전송됩니다.
3. 액세스한 행은 패키지로 패킹되어 중앙 노드로 다시 전송됩니다.
4. 데이터 행이 포함된 수신된 패키지는 버퍼에 배치됩니다.
5. 버퍼에서 데이터 행을 읽습니다.

MRR을 사용하는 경우 설명 출력의 추가 열에 `MRR` 사용 중이 표시됩니다.

쿼리 결과를 생성하기 위해 전체 테이블 행에 액세스할 필요가 없는 경우 `InnoDB` 및 MyISAM은 MRR을 사용하지 않습니다. 이는 전적으로 인덱스 튜플의 정보를 기반으로 결과를 생성할 수 있는 경우(커버링 인덱스를 통해)에 해당하며, MRR은 아무런 이점을 제공하지 않습니다.

두 개의 `optimizer_switch` 시스템 변수 플래그는 MRR 최적화 사용에 대한 인터페이스를 제공합니다. `mrr` 플래그는 MRR 사용 여부를 제어합니다. mrr이활성화된 경우(켜짐), `mrr_cost_based` 플래그는 옵티마이저가 MRR 사용과 미사용 사이에서 비용 기반 선택을 시도할지(켜짐), 아니면 가능한 경우 항상 MRR을 사용할지(꺼짐)를 제어합니다. 기본적으로 MRR은 켜져 있고 MRR_COST_BASED는 켜져 있습니다. 섹션 8.9.2, "전환 가능한 최적화"를 참조하세요.

MRR의 경우 스토리지 엔진은 버퍼에 할당할 수 있는 메모리 양에 대한 지침으로 `read_rnd_buffer_size` 시스템 변수 값을 사용합니다. 엔진은 최대 `READ_RND_BUFFER_SIZE` 바이트를 설정하고 단일 패스에서 처리할 범위의 수를 결정합니다.

8.2.1.12 중첩 루프 및 배치 키 액세스 조인 차단

MySQL에서는 조인된 테이블에 대한 인덱스 액세스와 조인 버퍼를 모두 사용하는 BKA(배치 키 액세스) 조인 알고리즘을 사용할 수 있습니다. BKA 알고리즘은 중첩된 외부 조인을 포함하여 Inner 조인, Outer 조인 및 Semi-조인 작업을 지원합니다. BKA의 장점으로는 보다 효율적인 테이블 스캔으로 인한 조인 성능 향상 등이 있습니다. 또한 이전에는 블록 중첩 루프(BNL) 조인 알고리즘만 사용되었습니다.
이 확장되어 중첩된 외부 조인을 포함한 외부 조인 및 세미조인 작업에도 사용할 수 있습니다.

다음 섹션에서는 원래 BNL 알고리즘, 확장된 BNL 알고리즘 및 BKA 알고리즘의 확장에 기반이 되는 조인 버퍼 관리에 대해 설명합니다. 세미조인 전략에 대한 자세한 내용은 세미조인 변환을 사용하여 IN 및 EXISTS 하위 쿼리 조건 최적화하기를 참조하세요.

- 블록 중첩 루프 및 배치 키 액세스 알고리즘을 위한 조인 버퍼 관리

- 외부 조인 및 세미조인을 위한 블록 중첩 루프 알고리즘
- 일괄 키 액세스 조인
- 블록 중첩 루프 및 배치 키 액세스 알고리즘을 위한 옵티마이저 힌트

블록 중첩 루프 및 배치 키 액세스 알고리즘을 위한 조인 버퍼 관리

MySQL은 조인 버퍼를 사용하여 내부 테이블에 대한 인덱스 액세스 없이 내부 조인뿐만 아니라 하위 쿼리 평활화 후에 나타나는 외부 조인 및 세미조인도 실행할 수 있습니다. 또한, 조인 버퍼는 내부 테이블에 대한 인덱스 액세스가 있을 때 효과적으로 사용할 수 있습니다.

조인 버퍼 관리 코드는 흥미로운 행 열의 값을 저장할 때 조인 버퍼 공간을 약간 더 효율적으로 활용합니다: 행 열의 값이 `NULL`인 경우 버퍼에 추가 바이트가 할당되지 않으며, `VARCHAR` 유형의 모든 값에 대해 최소 바이트 수가 할당됩니다.

이 코드는 일반 버퍼와 증분 버퍼의 두 가지 유형을 지원합니다. 조인 버퍼 B1을 사용하여 테이블 `t1` 및 `t2`를 조인하고 이 작업의 결과를 조인 버퍼 `B2`를 사용하여 테이블 `t3`과 조인한다고 가정합니다:

- 일반 조인 버퍼에는 각 조인 피연산자의 열이 포함됩니다. `B2`가 일반 조인 버퍼인 경우 `B2`에 입력되는 각 행 r은 `B1`의 행 *`r1`*의 열과 테이블 `t3`의 일치하는 행 *`r2`*의 흥미로운 열로 구성됩니다.
- 증분 조인 버퍼에는 두 번째 조인 피연산자에 의해 생성된 테이블 행의 열만 포함됩니다. 즉, 첫 번째 피연산자 버퍼에서 행으로 증분됩니다. `B2`가 증분 조인 버퍼인 경우 `B1`에서 *`r1`* 행으로 연결되는 링크와 함께 *`r2`* 행의 흥미로운 열이 포함됩니다.

증분 조인 버퍼는 항상 이전 조인 작업의 조인 버퍼를 기준으로 증분되므로 첫 번째 조인 작업의 버퍼는 항상 일반 버퍼입니다. 방금 주어진 예에서 테이블 `t1` 및 `t2`를 조인하는 데 사용된 버퍼 `B1`은 일반 버퍼여야 합니다.

조인 연산에 사용되는 증분 버퍼의 각 행에는 조인할 테이블의 행에서 흥미로운 열만 포함됩니다. 이러한 열은 첫 번째 조인 피연산자에 의해 생성된 테이블에서 일치하는 행의 흥미로운 열에 대한 참조로 보강됩니다. 증분 버퍼의 여러 행이 이전 조인 버퍼에 저장된 열이 모두 행 *`r`*과 일치하는 한 증분 버퍼의 여러 행이 동일한 행 *`r`*을 참조할 수 있습니다.

증분 버퍼를 사용하면 이전 조인 연산에 사용된 버퍼에서 열을 복사하는 빈도를 줄일 수 있습니다. 일반적인 경우 첫 번째 조인 피연산자가 생성한 행은 두 번째 조인 피연산자가 생성한 여러 행과 일치할 수 있으므로 버퍼 공간을 절약할 수 있습니다. 첫 번째 피연산자로부터 행의 복사본을 여러 개 만들 필요가 없습니다. 증분 버퍼는 또한 복사 시간 단축으로 인해 처리 시간을 절약할 수 있습니다.

`optimizer_switch` 시스템 변수의 `block_nested_loop` 플래그는 해시 조인을 제어합니다.

`batched_key_access` 플래그는 옵티마이저가 일괄 키 액세스 조인 알고리즘을 사용하는 방식을 제어합니다.

기본적으로 block_nested_loop는 켜져 있고 batched_key_access는 꺼져 있습니다. 섹션 8.9.2, "전환 가능한 최적화"를 참조하세요. 최적화 힌트도 적용할 수 있습니다. 블록 중첩 루프 및 배치 키 액세스 알고리즘에 대한 최적화 힌트를 참조하세요.

세미조인 전략에 대한 자세한 내용은 세미조인 변환을 사용하여 IN 및 EXISTS 하위 쿼리 조건 최적화를 참조하십시오.

외부 조인 및 세미조인을 위한 블록 중첩 루프 알고리즘

MySQL BNL 알고리즘의 원래 구현은 외부 조인 및 세미 조인 작업을 지원하도록 확장되었습니다(나중에 해시 조인 알고리즘으로 대체되었습니다. 8.2.1.4절. "해시 조인 최적화" 참조).

이러한 작업이 조인 버퍼를 사용하여 실행되면 버퍼에 입력된 각 행에 일치 플래그가 제공됩니다.

조인 버퍼를 사용하여 외부 조인 연산을 실행하는 경우 두 번째 피연산자에 의해 생성된 테이블의 각 행이 조인 버퍼의 각 행과 일치하는지 확인합니다. 일치하는 항목이 발견되면 새 확장 행(원래 행에 두 번째 피연산자의 열을 더한 행)이 형성되고 나머지 조인 작업에 의해 추가 확장을 위해 전송됩니다. 또한 버퍼에서 일치하는 행의 일치 플래그가 활성화됩니다. 조인할 테이블의 모든 행을 검사한 후 조인 버퍼가 스캔됩니다. 버퍼에서 일치 플래그가 활성화되지 않은 각 행은 `NULL` 보간(두 번째 피연산자의 각 열에 대한 `NULL` 값)으로 확장되고 나머지 조인 연산에 의해 추가 확장을 위해 전송됩니다.

`optimizer_switch` 시스템 변수의 `block_nested_loop` 플래그는 해시 조인을 제어합니다.

자세한 내용은 섹션 8.9.2, "전환 가능한 최적화"를 참조하세요. 최적화 힌트도 적용될 수 있습니다. 블록 중첩 루프 및 배치 키 액세스 알고리즘에 대한 최적화 힌트를 참조하세요.

설명 출력에서 테이블에 BNL을 사용하는 것은 추가 값에 조인 버퍼 사용(블록 중첩 루프)이 포함되고 유형 값이 `ALL`, 인덱스 또는 범위인 경우를 의미합니다.

세미조인 전략에 대한 자세한 내용은 세미조인 변환을 사용하여 IN 및 EXISTS 하위 쿼리 조건 최적화를 참조하십시오.

일괄 키 액세스 조인

MySQL은 일괄 키 액세스(BKA) 조인 알고리즘이라는 테이블 조인 방법을 구현합니다. 두 번째 조인 피연산자가 생성한 테이블에 대한 인덱스 액세스가 있는 경우 BKA를 적용할 수 있습니다. BNL 조인 알고리즘과 마찬가지로 BKA 조인 알고리즘은 조인 버퍼를 사용하여 조인 작업의 첫 번째 피연산자가 생성한 행의 관심 열을 누적합니다. 그런 다음 BKA 알고리즘은 버퍼의 모든 행에 대해 조인할 테이블에 액세스하기 위한 키를 생성하고 인덱스 조회를 위해 데이터베이스 엔진에 이러한 키를 일괄적으로 제출합니다. 키는 MRR(다중 범위 읽기) 인터페이스를 통해 엔진에 제출됩니다(섹션 8.2.1.11, "다중 범위 읽기 최적화" 참조). 키를 제출한 후 MRR 엔진 함수는 인덱스에서 최적의 방식으로 조회를 수행하여 이러한 키로 찾은 조인된 테이블의 행을 가져오고 일치하는 행을 BKA 조인 알고리즘에 공급하기 시작합니다. 일치하는 각 행은 조인 버퍼의 행에 대한 참조와 연결됩니다.

BKA를 사용하는 경우, `join_buffer_size`의 값은 스토리지 엔진에 대한 각 요청에서 키 배치의 크기를 정의합니다. 버퍼가 클수록 조인 작업의 오른쪽 테이블에 대한 순차 액세스가 더 많이 이루어지므로 성능이 크게 향상될 수 있습니다.

BKA를 사용하려면 `optimizer_switch` 시스템 변수의 `batched_key_access` 플래그가 `on`으로 설정되어 있어야 합니다. BKA는 `MRR`을 사용하므로 `mrr` 플래그도 켜져 있어야 합니다. 현재 MRR에 대한 비용 추정이 너무 비관적입니다. 따라서 BKA를 사용하려면 `mrr_cost_based`도 꺼져 있어야 합니다. 다음 설정은 BKA를 활성화합니다:

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

MRR 함수가 실행되는 시나리오에는 두 가지가 있습니다:

- 첫 번째 시나리오는 InnoDB 및 MyISAM과 같은 기존 디스크 기반 스토리지 엔진에 사용됩니다. 이러한 엔진의 경우, 일반적으로 조인 버퍼의 모든 행에 대한 키가 한 번에 MRR 인터페이스에 제출됩니다. 엔진별 MRR 함수는 제출된 키에 대한 인덱스 조회를 수행하고, 이로부터 행 ID(또는 기본 키)를 가져온 다음, BKA 알고리즘의 요청에 따라 선택한 모든 행 ID에 대한 행을 하나씩 가져옵니다. 모든 행은 조인 버퍼에서 일치하는 행에 액세스할 수 있는 연결 참조와 함께 반환됩니다. 행은 MRR 함수에 의해 최적의 방식으로 가져옵니다: 행 ID(기본 키) 순서로 가져옵니다. 이렇게 하면 읽기가 임의의 순서가 아닌 디스크 순서로 이루어지므로 성능이 향상됩니다.
- 두 번째 시나리오는 NDB와 같은 원격 저장소 엔진에 사용됩니다. 조인 버퍼의 일부 행에 대한 키 패키지가 해당 연결과 함께 MySQL 서버에 의해 전송됩니다.

(SQL 노드)를 MySQL 클러스터 데이터 노드로 전송합니다. 그 대가로 SQL 노드는 해당 연결과 결합된 일치하는 행의 패키지(또는 여러 패키지)를 받습니다. BKA 조인 알고리즘은 이러한 행을 가져와서 새로운 조인 행을 만듭니다. 그런 다음 새 키 세트가 데이터 노드로 전송되고 반환된 패키지의 행이 새 조인 행을 작성하는 데 사용됩니다. 이 프로세스는 조인 버퍼의 마지막 키가 데이터 노드로 전송되고 SQL 노드가 이러한 키와 일치하는 모든 행을 수신하여 조인할 때까지 계속됩니다. 이렇게 하면 SQL 노드가 데이터 노드로 전송하는 키가 포함된 패키지의 수가 적기 때문에 조인 작업을 수행하기 위해 SQL 노드와 데이터 노드 간에 왕복하는 횟수가 줄어들어 성능이 향상됩니다.

첫 번째 시나리오에서는 조인 버퍼의 일부가 인덱스 조회에서 선택한 행 ID(기본 키)를 저장하기 위해 예약되고 MRR 함수에 매개 변수로 전달됩니다.

조인 버퍼의 행에 대해 작성된 키를 저장하는 특별한 버퍼는 없습니다. 대신 버퍼의 다음 행에 대한 키를 작성하는 함수가 MRR 함수에 매개 변수로 전달됩니다.

`EXPLAIN` 출력에서 테이블에 대한 BKA 사용은 `추가` 값에 `조인 버퍼 사용(일괄 키 액세스)`이 포함되고 `유형` 값이 `ref` 또는 `eq_ref`인 경우 표시됩니다.

블록 중첩 루프 및 배치 키 액세스 알고리즘을 위한 옵티마이저 힌트

`optimizer_switch` 시스템 변수를 사용하여 세션 전체에서 BNL 및 BKA 알고리즘의 옵티마이저 사용을 제어하는 것 외에도 MySQL은 문별로 옵티마이저에 영향을 줄 수 있는 옵티마이저 힌트를 지원합니다. 섹션 8.9.3, "옵티마이저 힌트"를 참조하십시오.

외부 조인의 모든 내부 테이블에 대해 조인 버퍼링을 사용하도록 설정하려면 외부 조인의 모든 내부 테이블에 대해 조인 버퍼링을 사용하도록 설정해야 합니다.

8.2.1.13 조건 필터링

조인 처리에서 접두사 행은 조인의 한 테이블에서 다음 테이블로 전달되는 행입니다. 일반적으로 옵티마이저는 조인 순서에서 접두사 행 수가 적은 테이블을 조인 초기에 배치하여 행 조합의 수가 급격히 증가하지 않도록 합니다. 최적화 프로그램이 한 테이블에서 선택되어 다음 테이블로 전달되는 행의 조건에 대한 정보를 사용할 수 있는 범위 내에서 행 추정치를 더 정확하게 계산하고 최상의 실행 계획을 선택할 수 있습니다.

조건 필터링을 사용하지 않으면 테이블의 접두사 행 수는 옵티마이저가 선택한 액세스 방법에 따라 WHERE 절에서 선택한 예상 행 수를 기반으로 합니다. 조건 필터링을 사용하면 옵티마이저가 액세스 방법에서 고려하지 않은 다른 관련 조건을 `WHERE` 절에 사용할 수 있으므로 접두사 행 수 추정치를 개선할 수 있습니다. 예를 들어, 조인의 현재 테이블에서 행을 선택하는 데 사용할 수 있는 인덱스 기반 액세스 방법이 있더라도 다음 테이블로 전달되는 적격 행에 대한 추정치를 필터링(더 제한)할 수 있는 `WHERE` 절의 테이블에 대한 추가 조건이 있을 수 있습니다.

조건은 다음과 같은 경우에만 필터링 예상치에 기여합니다:

- 현재 테이블을 참조합니다.
- 이 값은 조인 시퀀스의 이전 테이블에 있는 상수 값에 따라 달라집니다.
- 액세스 방법에서 이미 고려되지 않았습니다.

설명 출력에서 행 열은 선택한 액세스 방법에 대한 행 예상치를 나타내며, 필터링된 열은 조건 필터링의 효과를 반영합니다. 필터링된 값은 백분율로 표시됩니다. 최대값은 100이며, 이는 행 필터링이 발생하지 않았음을 의미합니다. 100에서 감소하는 값은 필터링의 양이 증가함을 나타냅니다.

접두사 행 수(조인의 현재 테이블에서 다음 테이블로 전달될 것으로 예상되는 행 수)는 행과 필터링된 값의 곱입니다. 즉, 접두사 행 수는 예상 행 수에서 예상 필터링 효과로 줄어든 값입니다. 예를 들어 행이 1000이고 필터링된 값이

20%인 경우 조건 필터링은 예상 행 수 1000을 접두사 행 수 1000 × 20% = 1000 × .2 = 200으로 줄입니다.

다음 쿼리를 생각해 보세요:

```
SELECT *
  FROM employee JOIN department ON employee.dept_no = department.dept_no
  WHERE employee.first_name = 'John'
  AND employee.hire date가 '2018-01-01'과 '2018-06-01' 사이여야 합니다;
```

데이터 집합에 이러한 특성이 있다고 가정합니다:

- 직원 테이블에는 1024개의 행이 있습니다.
- 부서 테이블에는 12개의 행이 있습니다.
- 두 테이블 모두 부서 번호에 인덱스가 있습니다.
- 직원 테이블에는 first_name에 대한 인덱스가 있습니다.
- employee.first_name에서 이 조건을 충족하는 행이 8개입니다:

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

- employee.hire_date에서 이 조건을 충족하는 행은 150개입니다:

  ```
  employee.hire_date '2018-01-01'과 '2018-06-01' 사이
  ```

- 1행은 두 조건을 모두 충족합니다:

  ```
  employee.first_name = 'John'
  AND employee.hire_date가 '2018-01-01'과 '2018-06-01' 사이인 경우
  ```

조건 필터링이 없으면 EXPLAIN은 다음과 같은 출력을 생성합니다:

```
+----+------------+--------+-----------------+---------+---------+------+ ------------------+
| ID | 테이블 | 유형 | 가능한_키 | 키 | 참조 |       행 | 필터링됨 |
+----+------------+--------+-----------------+---------+---------+------+ ------------------+
| 1 | 직원 | eq_ref | PRIMARY | 이름,hire_date,부서 | 이름 | const | 8 | 부서_번호 | 1 | 100.00 | 100.00 |
+----+------------+--------+-----------------+---------+---------+------+ ------------------+
```

employee의 경우, 이름 인덱스의 액세스 메서드는 'John'이라는 이름과 일치하는 8개의 행을 선택합니다. 필터링이 수행되지 않으므로(필터링은 100%) 모든 행이 다음 테이블의 접두사 행이 됩니다: 접두사 행 수는 행 × 필터링됨 = 8 × 100% = 8입니다.

조건 필터링을 사용하면 옵티마이저는 액세스 메서드에서 고려하지 않은 WHERE 절의 조건을 추가로 고려합니다. 이 경우 옵티마이저는 휴리스틱을 사용하여 employee.hire_date의 BETWEEN 조건에 대해 16.31%의 필터링 효과를 추정합니다. 결과적으로 EXPLAIN은 다음과 같은 출력을 생성합니다:

```
+----+------------+--------+-----------------+---------+---------+------+ ------------------+
| ID | 테이블 | 유형 | 가능한_키 | 키 | 참조 |       행 | 필터링됨 |
+----+------------+--------+-----------------+---------+---------+------+ ------------------+
| 1 | 직원 | eq_ref | PRIMARY | 이름,hire_date,부서 | 이름 | const | 8 | 부서_번호 | 1 | 16.31 | 100.00 |
+----+------------+--------+-----------------+---------+---------+------+ ------------------+
```

이제 접두사 행 수는 행 × 필터링됨 = 8 × 16.31% = 1.3으로, 실제 데이터 집합을 더 가깝게 반영합니다.

일반적으로 옵티마이저는 행을 전달할 다음 테이블이 없기 때문에 마지막으로 조인된 테이블에 대한 조건 필터링 효과(접두사 행 수 감소)를 계산하지 않습니다. 설명: 자세한 정보를 제공하기 위해 마지막 테이블을 포함하여 조인된 모든 테이블에 대해 필터링 효과가 계산되는 경우는 예외입니다.

옵티마이저가 추가 필터링 조건을 고려할지 여부를 제어하려면
조건_팬아웃_필터 플래그를 최적화_스위치 시스템 변수에 설정합니다(섹션 8.9.2 참조),

"전환 가능한 최적화"). 이 플래그는 기본적으로 활성화되어 있지만 조건 필터링을 억제하기 위해 비활성화할 수 있습니다(예: 특정 쿼리가 조건 필터링 없이도 더 나은 성능을 내는 것으로 확인된 경우).

옵티마이저가 조건 필터링의 효과를 과대평가하는 경우 조건 필터링을 사용하지 않을 때보다 성능이 저하될 수 있습니다. 이러한 경우 이러한 기법이 도움이 될 수 있습니다:

- 열이 인덱싱되지 않은 경우 열을 인덱싱하여 옵티마이저가 열 값의 분포에 대한 정보를 확보하고 행 추정치를 개선할 수 있도록 합니다.
- 마찬가지로 열 히스토그램 정보를 사용할 수 없는 경우 히스토그램을 생성합니다(섹션 8.9.6, '최적화 도구 통계' 참조).
- 조인 순서를 변경한다. 이를 수행하는 방법에는 조인 순서 최적화 프로그램 힌트(섹션 8.9.3, "최적화 프로그램 힌트" 참조), `SELECT` 바로 뒤의 `STRAIGHT_JOIN` 및 `STRAIGHT_JOIN` 조인 연산자가 있습니다.
- 세션에 대한 조건 필터링을 비활성화합니다:

```
SET optimizer_switch = '조건_팬아웃_필터=오프';
```

또는 주어진 쿼리에 대해 최적화 도구 힌트를 사용합니다:

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

8.2.1.14 상시 접기 최적화

상수 값이 범위를 벗어나거나 열 유형에 대해 잘못된 유형인 상수와 열 값 간의 비교는 이제 쿼리 최적화 중에 실행 중이 아니라 행 단위로 한 번만 처리됩니다. 이러한 방식으로 처리할 수 있는 비교는 >, >=, <입니다, <=, <>/!=, = 및 <=>.

다음 문으로 생성된 테이블을 살펴보겠습니다:

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

`SELECT * FROM t WHERE c < 256` 쿼리의 `WHERE` 조건에는 적분 상수 256이 포함되며, 이는 `TINYINT UNSIGNED` 열의 범위를 벗어납니다. 이전에는 두 피연산자 모두 더 큰 유형으로 처리하여 이 문제를 처리했지만, 이제 `c`에 허용되는 모든 값이 상수보다 작으므로 `WHERE` 식을 `WHERE 1`로 접을 수 있으므로 쿼리가 `SELECT * FROM t WHERE 1`로 재작성됩니다.

이렇게 하면 옵티마이저가 `WHERE` 식을 완전히 제거할 수 있습니다. 열 `c`가 무효화 가능(즉, `TINYINT UNSIGNED`로만 정의됨)한 경우 쿼리는 다음과 같이 재작성됩니다:

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

폴딩은 다음과 같이 지원되는 MySQL 열 유형과 비교하여 상수에 대해 수행됩니다:

- **정수 열 유형:** 정수 유형은 여기에 설명된 대로 다음 유형의 상수와 비교됩니다:
  - **정수 값입니다.** 상수가 열 유형에 대한 범위를 벗어나는 경우 비교가 `1`로 접혀집니다. 또는 이미 표시된 것처럼 `IS NOT NULL`입니다.

상수가 범위 경계인 경우 비교는 =로 접힙니다. 예를 들어 (이미 정의된 것과 동일한 테이블 사용):

```
mysql> EXPLAIN SELECT * FROM t WHERE c >= 255;
*************************** 1. 행 ***************************
           ID: 1
  select_type: SIMPLE 테이
        블: t
   파티션: NULL
         유형: 모든
가능한_키: NULL
```

```
          키: NULL
      key_len: NULL
          참조: NULL
          행: 5
     필터링됨: 20.00 추가: 사
        용 위치
세트 1행, 경고 1회(0.00초)

mysql> SHOW WARNINGS;
*************************** 1. 행 *************************** 레벨:
  참고
   코드: 1003
Message: /* select#1 */ select `test`.`t`.`ti` AS `ti` from `test`.`t` where (`test`.`t`.`ti` = 255)
한 세트에 1행(0.00초)
```

- **부동 소수점 또는 고정** 소수점 **값. 상수가 소수점** 유형(예: `DECIMAL`, `REAL`, `DOUBLE` 또는 `FLOAT`) 중 하나이고 소수점 부분이 0이 아닌 경우 동일할 수 없으므로 적절히 배분합니다. 다른 비교의 경우 부호에 따라 정수 값으로 반올림 또는 반내림한 다음 범위 검사를 수행하고 정수-정수 비교에 대해 이미 설명한 대로 처리합니다.

  너무 작아서 십진수로 표현할 수 없는 `REAL` 값은 부호에 따라 .01 또는 -.01로 반올림한 다음 십진수로 처리됩니다.

- **문자열 유형.** 문자열 값을 정수 유형으로 해석한 다음 정수 값 간의 비교를 처리합니다. 실패하면 값을 실수로 처리하려고 시도합니다.

- **DECIMAL 또는 REAL** 열. Decimal 유형은 여기에 설명된 대로 다음 유형의 상수와 비교됩니다:

  - **정수 값.** 열 값의 정수 부분에 대해 범위 검사를 수행합니다. 폴딩 결과가 없으면 상수를 열 값과 소수점 이하 자릿수가 같은 `DECIMAL`로 변환한 다음 `DECIMAL`로 확인합니다(다음 참조).

  - **DECIMAL 또는 REAL 값.** 오버플로 여부(즉, 상수의 정수 부분에 열의 소수 유형에 허용되는 것보다 많은 자릿수가 있는지 여부)를 확인합니다. 그렇다면 접습니다.

    상수에 열 유형보다 더 큰 분수 자릿수가 있는 경우 상수를 잘라냅니다. 비교 연산자가 `=` 또는 `<>`인 경우 접습니다. 연산자가 `>=` 또는 `<=`인 경우 잘림으로 인해 연산자를 조정합니다. 예를 들어 열의 유형이 `DECIMAL(3,1)`인 경우 `SELECT * FROM t WHERE f >=`는 다음과 같습니다. `10.13`은 `SELECT * FROM t WHERE f > 10.1`이 됩니다.

    상수의 소수 자릿수가 열 유형보다 적은 경우 동일한 자릿수를 가진 상수로 변환합니다. `REAL` 값의 언더플로우(즉, 소수 자릿수가 너무 적어 표현할 수 없는 경우)의 경우 상수를 소수점 0으로 변환합니다.

  - **문자열 값** . 값이 정수 유형으로 해석될 수 있으면 정수 유형으로 처리합니다. 그렇지 않으면 `REAL`로 처리하세요.

- **FLOAT 또는 DOUBLE column.** `FLOAT(m,n)` 또는 `DOUBLE(m,n)` 값은 상수와 비교하여 다음과

같이 처리됩니다:

값이 열의 범위를 초과하는 경우 접습니다.

값이 *소수점*이하인 경우 잘라내어 접는 동안 보정합니다. 및 `<>` 비교의 경우 앞서 설명한 대로 `TRUE`, `FALSE` 또는 `IS [NOT] NULL`로 폴딩하고, 다른 연산자의 경우 연산자를 조정합니다.

값이 정수를 초과하는 자릿수인 경우 접습니다.

제한 사항. 다음과 같은 경우에는 이 최적화를 사용할 수 없습니다:

1. `BETWEEN` 또는 `IN`을 사용하여 비교합니다.
2. 비트 열 또는 날짜 또는 시간 유형을 사용하는 열을 사용합니다.

3. 준비된 문을 준비하는 단계에서는 준비된 문을 실제로 실행할 때 최적화 단계에서 적용될 수 있습니다. 이는 문 준비 중에 상수의 값을 아직 알 수 없기 때문입니다.

8.2.1.15 IS NULL 최적화

MySQL은 *col_name = constant_value*에 사용할 수 있는 것과 동일한 최적화를 *col_name* IS NULL에 수행할 수 있습니다. 예를 들어, MySQL은 인덱스와 범위를 사용하여 IS NULL로 NULL을 검색할 수 있습니다.

예시:

```
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;
```

NOT NULL로 선언된 열에 대해 WHERE 절에 *col_name* IS NULL 조건이 포함된 경우 해당 식은 최적화되지 않습니다. 이 최적화는 열이 어쨌든 NULL을 생성할 수 있는 경우(예: LEFT JOIN의 오른쪽에 있는 테이블에서 오는 경우)에는 발생하지 않습니다.

MySQL은 또한 확인된 하위 쿼리에서 일반적으로 사용되는 형식인 *col_name = expr* 또는 *col_name* IS NULL 조합을 최적화할 수 있습니다. 이 최적화가 사용되는 경우 EXPLAIN은 ref_or_null을 표시합니다.

이 최적화는 모든 주요 부분에 대해 하나의 IS NULL을 처리할 수 있습니다.

테이블 t2의 열 a와 b에 인덱스가 있다고 가정하여 최적화된 쿼리의 몇 가지 예입니다:

```
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 ...)
  또는 (t1.a=t2.a AND t2.a IS NULL AND ...);
```

ref_or_null은 먼저 참조 키에 대한 읽기를 수행한 다음, NULL 키 값을 가진 행을 별도로 검색하는 방식으로 작동합니다.

최적화는 하나의 IS NULL 수준만 처리할 수 있습니다. 다음 쿼리에서 MySQL은 표현식 (t1.a=t2.a AND t2.a IS NULL)에 대해서만 키 조회를 사용하며 b의 키 부분은 사용할 수 없습니다:

```
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 최적화별 주문

이 섹션에서는 MySQL이 인덱스를 사용하여 `ORDER BY` 절을 만족시킬 수 있는 경우, 인덱스를 사용할 수 없는 경우 사용되는 파일 정렬 작업 및 최적화 프로그램에서 사용할 수 있는 `ORDER BY`에 대한 실행 계획 정보를 설명합니다.

8.2.1.19절 "LIMIT 쿼리 최적화"에서 설명한 대로 `LIMIT`가 있든 없든 `ORDER BY`는 서로 다른 순서로 행을 반환할 수 있습니다.

- 주문 기준 충족을 위한 인덱스 사용

- 주문 기준 충족을 위한 파일 정렬 사용
- 최적화를 통한 주문에 미치는 영향
- 실행 계획별 주문 정보 제공

주문 기준 충족을 위한 인덱스 사용

경우에 따라 MySQL은 인덱스를 사용하여 `ORDER BY` 절을 충족하고 파일 정렬 작업을 수행하는 데 관련된 추가 정렬을 피할 수 있습니다.

인덱스의 사용되지 않은 부분과 여분의 모든 `ORDER BY` 열이 `WHERE` 절에 상수인 경우 `ORDER BY`가 인덱스와 정확히 일치하지 않더라도 인덱스를 사용할 수 있습니다. 인덱스에 쿼리에서 액세스하는 모든 열이 포함되어 있지 않은 경우 인덱스 액세스가 다른 액세스 방법보다 저렴한 경우에만 인덱스가 사용됩니다.

`(key_part1, key_part2)`에 인덱스가 있다고 가정하면 다음 쿼리에서 인덱스를 사용하여 `ORDER BY` 부분을 확인할 수 있습니다. 옵티마이저가 실제로 그렇게 할지 여부는 인덱스에 없는 열도 읽어야 하는 경우 인덱스를 읽는 것이 테이블 스캔보다 더 효율적인지 여부에 따라 달라집니다.

- 이 쿼리에서 인덱스 온 `(key_part1, key_part2)`을 사용하면 옵티마이저가 정렬을 피할 수 있습니다:

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

그러나 쿼리에서는 `SELECT *`를 사용하므로 `key_part1` 및 `key_part2`보다 더 많은 열을 선택할 수 있습니다. 이 경우 전체 인덱스를 스캔하고 테이블 행을 조회하여 인덱스에 없는 열을 찾는 것이 테이블을 스캔하고 결과를 정렬하는 것보다 비용이 더 많이 들 수 있습니다. 그렇다면 옵티마이저가 인덱스를 사용하지 않는 것일 수 있습니다. `SELECT *`가 인덱스 열만 선택하면 인덱스가 사용되며 정렬이 수행되지 않습니다.

`t1`이 `InnoDB` 테이블인 경우 테이블 기본 키는 암시적으로 인덱스의 일부이며 인덱스를 사용하여 이 쿼리에 대한 `ORDER BY`를 확인할 수 있습니다:

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

- 이 쿼리에서 `key_part1`은 상수이므로 인덱스를 통해 액세스하는 모든 행은 `key_part2` 순서이며, 인덱스 온 `(key_part1, key_part2)`은 `WHERE` 절이 충분히 선택적이어서 인덱스 범위 스캔이 테이블 스캔보다 저렴할 경우 정렬을 피합니다:

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

- 다음 두 쿼리에서 인덱스 사용 여부는 이전에 표시된 `DESC`가 없는 동일한 쿼리와 유사합니다:

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

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

- `ORDER BY`의 두 열은 같은 방향(둘 다 `ASC` 또는 둘 다 `DESC`)으로 정렬하거나 반대 방향(하나의 `ASC`, 하나의 `DESC`)으로 정렬할 수 있습니다. 인덱스 사용 조건은 인덱스가 동일한 동질성을 가져야 하지만 실제 방향이 같을 필요는 없다는 것입니다.

  쿼리가 ASC와 `DESC`를 혼합하는 경우, 인덱스가 해당 혼합 오름차순 및 내림차순 열을 사용하는 경우 옵티마이저는 열에 인덱스를 사용할 수 있습니다:

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

옵티마이저는 key_part1이 내림차순인 경우 인덱스 온(*key_part1*, *key_part2*)*을* 사용할 수 있습니다. *키_파트2는* 오름차순입니다. 또한 key_part1이 오름차순이고 *key_part2가* 내림차순인 경우 해당 열에 인덱스를 사용할 수도 있습니다(역방향 스캔 포함). 섹션 8.3.13, "내림차순 인덱스"를 참조하세요.

- 다음 두 쿼리에서 *key_part1은 상수와* 비교됩니다. 인덱스는 WHERE 절은 인덱스 범위 스캔을 테이블 스캔보다 저렴하게 만들 수 있을 만큼 선택적입니다:

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

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

- 다음 쿼리에서 ORDER BY의 이름은 *key_part1이* 아니지만 선택된 모든 행에는 상수가 있습니다. *키_파트1* 값으로 변경하여 인덱스를 계속 사용할 수 있습니다:

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

경우에 따라 MySQL은 인덱스를 사용하여 ORDER BY를 확인할 수 없지만 인덱스를 사용하여 WHERE 절과 일치하는 행을 찾을 수 있습니다. 예시:

- 쿼리는 서로 다른 인덱스에서 ORDER BY를 사용합니다:

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

- 이 쿼리는 인덱스의 연속되지 않은 부분에 ORDER B를사용합니다:

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

- 행을 가져오는 데 사용되는 인덱스가 ORDER BY에 사용된 인덱스와 다릅니다:

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

- 쿼리는 인덱스 열 이름 이외의 용어가 포함된 표현식과 함께 ORDER BY를 사용합니다:

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

- 쿼리가 여러 테이블을 조인하며 ORDER BY의 열이 모두 행을 검색하는 데 사용되는 첫 번째 상수가 아닌 테이블의 열이 아닙니다. (이 테이블은 EXPLAIN 출력에서 상수 조인 유형이 없는 첫 번째 테이블입니다.)

- 쿼리에는 다른 ORDER BY 및 GROUP BY 표현식이 있습니다.

- ORDER BY 절에 이름이 지정된 열의 접두사에만 인덱스가 있습니다. 이 경우 인덱스를 사용하여 정렬 순서를 완전히 확인할 수 없습니다. 예를 들어 CHAR(20) 열의 처음 10바이트만 인덱싱된 경우 인덱스는 10번째 바이트 이후의 값을 구분할 수 없으므로 파일 정렬이 필요합니다.

- 인덱스는 행을 순서대로 저장하지 않습니다. 예를 들어, MEMORY 테이블의 HASH 인덱스가 이에 해당합니다.

정렬을 위한 인덱스의 가용성은 열 별칭의 사용으로 인해 영향을 받을 수 있습니다. t1.a 열이 인덱싱되었다고 가정합니다. 이 문에서 선택 목록에 있는 열의 이름은 a이며, ORDER BY의 a에 대한 참조와 마찬가지로

t1.a를 참조하므로 t1.a의 인덱스를 사용할 수 있습니다:

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

이 문에서 선택 목록의 열 이름도 a이지만 별칭 이름입니다. 이는 다음을 나타냅니다.
ABS(a)와 마찬가지로 ORDER BY에서 a에 대한 참조가 있으므로 t1.a의 인덱스는 사용할 수 없습니다:

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

다음 문에서 ORDER BY는 선택 목록의 열 이름이 아닌 이름을 참조합니다. 그러나 t1에 a라는 이름의 열이 있으므로 ORDER BY는 t1.a를 참조하고 t1.a의 인덱스를 사용할 수 있습니다. (물론 결과 정렬 순서는 ABS(a)의 순서와 완전히 다를 수 있습니다.)

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

이전 버전(MySQL 8.1 이하)에서는 특정 조건에서 GROUP BY가 암시적으로 정렬되었습니다. MySQL 8.2에서는 더 이상 이러한 현상이 발생하지 않으므로, 암시적 정렬을 억제하기 위해 마지막에 ORDER BY NULL을 지정할 필요가 없습니다(이전과 같이). 그러나 쿼리 결과는 이전 MySQL 버전과 다를 수 있습니다. 지정된 정렬 순서를 생성하려면 ORDER BY 절을 제공합니다.

주문 기준 충족을 위한 파일 정렬 사용

인덱스를 사용하여 ORDER BY 절을 충족할 수 없는 경우 MySQL은 테이블 행을 읽고 정렬하는 파일 정렬 작업을 수행합니다. 파일 정렬은 쿼리 실행에서 추가적인 정렬 단계를 구성합니다.

파일 정렬 연산을 위한 메모리를 확보하기 위해 옵티마이저는 필요에 따라 sort_buffer_size 시스템 변수에 지정된 크기까지 메모리 버퍼를 점진적으로 할당합니다. 따라서 사용자는 작은 정렬에 과도한 메모리를 사용할 염려 없이 큰 정렬의 속도를 높이기 위해 sort_buffer_size를 더 큰 값으로 설정할 수 있습니다. (멀티스레드 할당 능력이 약한 Windows에서 여러 개의 동시 정렬을 수행할 경우 이 이점이 발생하지 않을 수 있습니다.)

파일 정렬 작업은 결과 집합이 너무 커서 메모리에 넣을 수 없는 경우 필요에 따라 임시 디스크 파일을 사용합니다. 일부 쿼리 유형은 특히 완전 인메모리 파일 정렬 작업에 적합합니다. 예를 들어, 옵티마이저는 파일을 정렬하여 다음과 같은 형식의 쿼리(및 하위 쿼리)에 대해 임시 파일 없이 메모리에서 ORDER BY 연산을 효율적으로 처리할 수 있습니다:

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

이러한 쿼리는 더 큰 결과 집합에서 몇 개의 행만 표시하는 웹 애플리케이션에서 일반적입니다. 예시:

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

최적화를 통한 주문에 미치는 영향

ORDER BY 속도를 높이려면 MySQL이 추가 정렬 단계 대신 인덱스를 사용하도록 할 수 있는지 확인하세요. 이것이 가능하지 않은 경우 다음 전략을 시도해 보세요:

- sort_buffer_size 변수 값을 늘립니다. 이 값은 전체 결과 집합이 정렬 버퍼에 들어갈 수 있을 만큼 충분히 커야 합니다(디스크 쓰기 및 병합 패스를 피하기 위해).

  정렬 버퍼에 저장되는 열 값의 크기는 max_sort_length 시스템 변수 값의 영향을 받는다는 점을 고려하세요. 예를 들어, 튜플이 긴 문자열 열의 값을 저장하는 경우 max_sort_length 값을 늘리면 정렬 버퍼 튜플의 크기도 함께 증가하므로 sort_buffer_size를 늘려야 할 수 있습니다.

임시 파일을 병합하기 위한 병합 패스 횟수를 모니터링하려면 `Sort_merge_pass` 상태 변수를 전달합니다.

- 한 번에 더 많은 행을 읽을 수 있도록 `read_rnd_buffer_size` 변수 값을 늘리세요.
- 여유 공간이 많은 전용 파일 시스템을 가리키도록 `tmpdir` 시스템 변수를 변경합니다. 이 변수 값은 라운드 로빈 방식으로 사용되는 여러 경로를 나열할 수 있으므로 이 기능을 사용하여 여러 디렉터리로 부하를 분산할 수 있습니다. Unix에서는 콜론 문자(`:`)로, Windows에서는 세미콜론 문자(`;`)로 경로를 구분합니다. 경로는 동일한 디스크의 다른 파티션이 아닌 다른 *물리적* 디스크에 있는 파일 시스템의 디렉터리 이름을 지정해야 합니다.

실행 계획별 주문 정보 제공

EXPLAIN(8.8.1절. "EXPLAIN으로 쿼리 최적화하기" 참조)을 사용하면 MySQL이 인덱스를 사용하여 ORDER BY 절을 해결할 수 있는지 확인할 수 있습니다:

- 설명 출력의 추가 열에 파일 정렬 사용이 포함되지 않은 경우 인덱스가 사용되며 파일 정렬이 수행되지 않습니다.

  - 설명 출력의 추가 열에 파일 정렬 사용이 포함된 경우 인덱스가 사용되지 않고 파일 정렬이 수행됩니다.

또한 파일 정렬이 수행되는 경우 옵티마이저 추적 출력에는 filesort_summary 블록이 포함됩니다. 예를 들어

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

peak_memory_used는 정렬 중에 한 번에 사용되는 최대 메모리를 나타냅니다. 이 값은 sort_buffer_size 시스템 변수의 값만큼 크지만 반드시 이 값만큼 크지는 않습니다. 옵티마이저는 정렬 버퍼 메모리를 소량부터 시작하여 필요에 따라 정렬 버퍼 메모리를 최대 정렬_버퍼_size 바이트까지 증분하여 할당합니다.)

sort_mode 값은 정렬 버퍼에 있는 튜플의 내용에 대한 정보를 제공합니다:

- <sort_key, rowid>: 이는 정렬 버퍼 튜플이 원래 테이블 행의 정렬 키 값과 행 ID를 포함하는 쌍임을 나타냅니다. 튜플은 정렬 키 값으로 정렬되며 행 ID는 테이블에서 행을 읽는 데 사용됩니다.

- <sort_key, additional_fields>: 이것은 정렬 버퍼 튜플에 정렬 키 값과 쿼리에서 참조하는 열이 포함되어 있음을 나타냅니다. 튜플은 정렬 키 값으로 정렬되며 열 값은 튜플에서 직접 읽습니다.

- <sort_key, packed_additional_fields>: 이전 변형과 비슷하지만 고정 길이 인코딩을 사용하는 대신 추가 열을 촘촘하게 패킹합니다.

설명은 옵티마이저가 파일 정렬을 수행하는지 수행하지 않는지 구분하지 않습니다.
를 메모리에 저장합니다. 인메모리 파일 정렬의 사용은 옵티마이저 추적 출력에서 확인할 수 있습니다.
filesort_priority_queue_optimization을 찾아보세요. 옵티마이저 추적에 대한 자세한 내용은 MySQL 내부를 참조하세요: 옵티마이저 추적을 참조하세요.

8.2.1.17 최적화별 그룹화

GROUP BY 절을 충족하는 가장 일반적인 방법은 전체 테이블을 스캔하여 각 그룹의 모든 행이 연속된 새 임시 테이블을 만든 다음 이 임시 테이블을 사용하여 그룹을 검색하고 집계 함수(있는 경우)를 적용하는 것입니

다. 경우에 따라 MySQL은 인덱스 액세스를 사용하여 임시 테이블 생성을 피하고 이보다 훨씬 더 나은 작업을 수행할 수 있습니다.

`GROUP BY`에 인덱스를 사용하기 위한 가장 중요한 전제 조건은 모든 `GROUP BY` 열이 동일한 인덱스의 특성을 참조하고 인덱스가 키를 순서대로 저장한다는 것입니다(예를 들어 `BTREE` 인덱스의 경우와 같지만 `HASH` 인덱스의 경우와 같지 않음). 임시 테이블 사용을 인덱스 액세스로 대체할 수 있는지 여부는 쿼리에서 인덱스의 어떤 부분이 사용되는지, 이러한 부분에 대해 지정된 조건 및 선택한 집계 함수에 따라 달라집니다.

다음 섹션에 자세히 설명된 대로 인덱스 액세스를 통해 `GROUP BY` 쿼리를 실행하는 방법에는 두 가지가 있습니다. 첫 번째 방법은 모든 범위 술어(있는 경우)와 함께 그룹화 연산을 적용합니다. 두 번째 방법은 먼저 범위 스캔을 수행한 다음 결과 튜플을 그룹화합니다.

- 느슨한 인덱스 스캔
- 타이트한 인덱스 스캔

느슨한 인덱스 스캔은 일부 조건에서 그룹 기준이 없는 경우에도 사용할 수 있습니다. 스캔 범위 액세스 방법 건너뛰기를 참조하십시오.

느슨한 인덱스 스캔

`GROUP BY`를 처리하는 가장 효율적인 방법은 인덱스를 사용하여 그룹화 열을 직접 검색하는 경우입니다. 이 액세스 방법을 사용하면 MySQL은 키가 정렬되는 일부 인덱스 유형(예: `BTREE`)의 속성을 사용합니다. 이 속성을 사용하면 인덱스에서 모든 `WHERE` 조건을 만족하는 모든 키를 고려할 필요 없이 인덱스에서 조회 그룹을 사용할 수 있습니다. 이 액세스 방법은 인덱스에 있는 키의 일부만 고려하므로 느슨한 *인덱스 스캔이라고 합니다*. `WHERE` 절이 없는 경우 느슨한 인덱스 스캔은 그룹 수만큼의 키를 읽으며, 이는 모든 키의 수보다 훨씬 적을 수 있습니다. `WHERE` 절에 범위 술어가 포함된 경우(8.8.1절 "EXPLAIN으로 쿼리 최적화"의 범위 조인 유형에 대한 설명 참조) 느슨한 인덱스 스캔은 범위 조건을 충족하는 각 그룹의 첫 번째 키를 조회하고 다시 가능한 가장 작은 수의 키를 읽습니다. 이는 다음 조건에서 가능합니다:

- 쿼리는 단일 테이블에 대한 것입니다.
- `GROUP BY`는 인덱스의 가장 왼쪽 접두사를 형성하는 열만 이름을 지정하고 다른 열은 지정하지 않습니다. (쿼리에 `GROUP BY` 대신 `DISTINCT` 절이 있는 경우 모든 고유 특성은 인덱스의 가장 왼쪽 접두사를 형성하는 열을 참조합니다.) 예를 들어, 테이블 t1에 `(c1,c2,c3)`에 대한 인덱스가 있는 경우 쿼리에 `GROUP BY c1, c2`가 있는 경우 느슨한 인덱스 스캔을 적용할 수 있습니다. 쿼리에 `GROUP BY c2, c3`(열이 가장 왼쪽 접두사가 아님) 또는 `GROUP BY c1, c2, c4`(c4가 인덱스에 없음)가 있는 경우에는 적용되지 않습니다.
- 선택 목록에 사용되는 유일한 집계 함수는 `MIN()` 및 `MAX()`이며, 모두 동일한 열을 참조합니다. 열은 인덱스에 있어야 하며 `GROUP BY`의 열 바로 뒤에 와야 합니다.
- 쿼리에서 참조된 `GROUP BY`에서 참조된 인덱스 이외의 다른 부분은 상수여야 합니다(즉, 상수와 동일하게 참조되어야 함). 단, `MIN()` 또는 `MAX()` 함수의 인수는 예외입니다.
- 인덱스의 열의 경우 접두사만 인덱싱하는 것이 아니라 전체 열 값을 인덱싱해야 합니다. 예를 들어, `c1 VARCHAR(20), INDEX (c1(10))`의 경우 인덱스는 접두사 `c1` 값만 사용하며 느슨한 인덱스 스캔에 사용할 수 없습니다.

느슨한 인덱스 스캔이 쿼리에 적용 가능한 경우 설명 출력에 추가 열에 그룹에 인덱스 사용-으로 표시됩니다.

테이블 `t1(c1,c2,c3,c4)`에 `idx(c1,c2,c3,c4)` 인덱스가 있다고 가정합니다. 느슨한 인덱스 스캔 액세스 메서드는 다음 쿼리에 사용할 수 있습니다:

```
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;
```

다음 쿼리는 주어진 이유로 인해 이 빠른 선택 방법으로 실행할 수 없습니다:

- `MIN()` 또는 `MAX()` 이외의 집계 함수가 있습니다:

  ```
  SELECT c1, SUM(c2) FROM t1 GROUP BY c1;
  ```

- `GROUP BY` 절의 열은 인덱스의 가장 왼쪽 접두사를 형성하지 않습니다:

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

- 쿼리는 `GROUP BY` 부분 뒤에 오는 키의 일부이며 상수와 같지 않은 부분을 나타냅니다:

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

  쿼리에 `WHERE c3 = const가` 포함되는 경우, 느슨한 인덱스 스캔을 사용할 수 있습니다.

느슨한 인덱스 스캔 액세스 방법은 이미 지원되는 `MIN()` 및 `MAX()` 참조 외에도 선택 목록의 다른 형태의 집계 함수 참조에 적용할 수 있습니다:

- `AVG(DISTINCT)`, `SUM(DISTINCT)`, `COUNT(DISTINCT)`가지원됩니다. `AVG(DISTINCT)` 및 `SUM(DISTINCT)`는 단일 인수를 받습니다. `COUNT(DISTINCT)`는 둘 이상의 열 인수를 가질 수 있습니다.
- 쿼리에는 `GROUP BY` 또는 `DISTINCT` 절이 없어야 합니다.
- 앞서 설명한 느슨한 인덱스 스캔 제한은 여전히 적용됩니다.

테이블 `t1(c1,c2,c3,c4)`에 `idx(c1,c2,c3,c4)` 인덱스가 있다고 가정합니다. 느슨한 인덱스 스캔 액세스 메서드는 다음 쿼리에 사용할 수 있습니다:

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

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

타이트한 인덱스 스캔

정밀 인덱스 스캔은 쿼리 조건에 따라 전체 인덱스 스캔 또는 범위 인덱스 스캔이 될 수 있습니다.

느슨한 인덱스 스캔의 조건이 충족되지 않는 경우에도 `GROUP BY` 쿼리에 대한 임시 테이블 생성을 피할 수 있습니다. `WHERE` 절에 범위 조건이 있는 경우 이 메서드는 해당 조건을 만족하는 키만 읽습니다. 그렇지 않으면 인덱스 스캔을 수행합니다. 이 메서드는 `WHERE` 절에 정의된 각 범위의 모든 키를 읽거나 범위 조건이 없는 경우 전체 인덱스를 스캔하기 때문에 이를 정밀 *인덱스 스캔*이라고 합니다. 정밀 인덱스 스캔을 사용하면 범위 조건을 만족하는 모든 키를 찾은 후에만 그룹화 작업이 수행됩니다.

이 방법이 작동하려면 `GROUP BY` 키의 앞이나 사이에 오는 키의 일부를 참조하는 쿼리의 모든 열에 대해 일정한 동일성 조건이 있으면 충분합니다. 이 경우
동일성 조건의 상수는 검색 키의 "틈새"를 채워서 다음을 형성할 수 있습니다.
인덱스의 접두사를 완성합니다. 그러면 이러한 인덱스 접두사를 인덱스 조회에 사용할 수 있습니다. `GROUP BY` 결과에 정렬이 필요하고 인덱스의 접두사인 검색 키를 구성할 수 있는 경우, 정렬된 인덱스에서 접두사로 검색하면 이미 모든 키가 순서대로 검색되므로 MySQL에서는 추가 정렬 작업을 피할 수 있습니다.

테이블 `t1(c1,c2,c3,c4)`에 `idx(c1,c2,c3,c4)` 인덱스가 있다고 가정합니다. 다음 쿼리는 앞에서 설명한 느슨한 인덱스 스캔 액세스 방법에서는 작동하지 않지만 긴밀한 인덱스 스캔 액세스 방법에서는 여전히 작동합니다.

- `GROUP BY`에 간격이 있지만 `c2 = 'a'` 조건에 의해 커버됩니다:

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

- GROUP BY는 키의 첫 번째 부분으로 시작하지 않지만 해당 부분에 대한 상수를 제공하는 조건이 있습니다:

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

8.2.1.18 차별화된 최적화

DISTINCT를 ORDER BY와 결합하면 많은 경우에 임시 테이블이 필요합니다.

DISTINCT는 GROUP BY를 사용할 수 있으므로 선택한 열의 일부가 아닌 ORDER BY 또는 HAVING 절의 열에 대해 MySQL이 어떻게 작동하는지 알아보십시오. 섹션 12.19.3, "MySQL의 GROUP BY 처리"를 참조하십시오.

대부분의 경우 DISTINCT 절은 GROUP BY의 특수한 경우로 간주할 수 있습니다. 예를 들어 다음 두 쿼리는 동일합니다:

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

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

이러한 동등성으로 인해 GROUP BY 쿼리에 적용되는 최적화는 DISTINCT 절이 있는 쿼리에도 적용될 수 있습니다. 따라서 DISTINCT 쿼리에 대한 최적화 가능성에 대한 자세한 내용은 섹션 8.2.1.17, "GROUP BY 최적화"를 참조하십시오.

LIMIT row_count와 DISTINCT를 결합하면 MySQL은 *row_count*를 발견하는 즉시 중지합니다. 고유 행입니다.

쿼리에 이름이 지정된 모든 테이블의 열을 사용하지 않는 경우 MySQL은 첫 번째 일치 항목을 발견하는 즉시 사용하지 않는 테이블 검색을 중지합니다. 다음 사례에서 t1이 t2보다 먼저 사용되었다고 가정하면(EXPLAIN으로 확인할 수 있음), MySQL은 t2에서 첫 번째 행을 찾으면 t1의 특정 행에 대해 t2에서 읽기를 중지합니다:

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

8.2.1.19 제한 쿼리 최적화

결과 집합에서 지정된 수의 행만 필요한 경우, 전체 결과 집합을 가져와 여분의 데이터를 버리는 대신 쿼리에 LIMIT 절을 사용합니다.

MySQL은 때때로 LIMIT *row_count* 절이 있고 HAVING 절이 없는 쿼리를 최적화합니다:

- LIMIT를 사용하여 몇 개의 행만 선택하면 일반적으로 전체 테이블 스캔을 선호할 때 MySQL이 인덱스를 사용하는 경우가 있습니다.

- LIMIT row_count와 ORDER BY를 결합하면 MySQL은 전체 결과를 정렬하는 대신 정렬된 결과의 첫 번째 *row_count* 행을 찾자마자 정렬을 중지합니다. 인덱스를 사용하여 정렬을 수행하는 경우 매우 빠릅니다. 파일 정렬을 수행해야 하는 경우, LIMIT 절 없이 쿼리와 일치하는 모든 행이 선택되고 첫 번째 *row_count*가 발견되기 전에 대부분 또는 전체가 정렬됩니다. 초기 행을 찾은 후에는 MySQL은 결과 집합의 나머지 부분을 정렬하지 않습니다.

  이 동작의 한 가지 예는 이 섹션의 뒷부분에 설명된 대로 LIMIT가 있거나없는 ORDER BY 쿼리가 다른 순서로 행을 반환할 수 있다는 것입니다.

- LIMIT row_count와 DISTINCT를 결합하면 MySQL은 *row_count*를 발견하는 즉시 중지합니다. 고유 행입니다.

- 경우에 따라 인덱스를 순서대로 읽거나 인덱스에 대해 정렬을 수행한 다음 인덱스 값이 변경될 때까지 요약을 계산하여 `GROUP BY`를 해결할 수 있습니다. 이 경우 `LIMIT` *`row_count`*는 불필요한 `GROUP BY` 값을 계산하지 않습니다.

- MySQL이 필요한 수의 행을 클라이언트에 전송하자마자 `SQL_CALC_FOUND_ROWS`를 사용하지 않는 한 쿼리를 중단합니다. 이 경우 `SELECT FOUND_ROWS()`를 사용하여 행 수를 검색할 수 있습니다. 섹션 12.15, "정보 함수"를 참조하십시오.

- `LIMIT 0`은 빈 집합을 빠르게 반환합니다. 이는 쿼리의 유효성을 확인하는 데 유용할 수 있습니다. 또한 결과 집합 메타데이터를 사용할 수 있도록 하는 MySQL API를 사용하는 애플리케이션 내에서 결과 열의 유형을 가져오는 데 사용할 수도 있습니다. `mysql` 클라이언트 프로그램에서 `-- column-type-info` 옵션을 사용하여 결과 열 유형을 표시할 수 있습니다.

- 서버가 임시 테이블을 사용하여 쿼리를 해결하는 경우, 서버는 `LIMIT row_count` 절을 사용하여 필요한 공간을 계산합니다.
- 인덱스가 `ORDER BY`에 사용되지 않고 `LIMIT` 절도 있는 경우 옵티마이저는 병합 파일 사용을 피하고 메모리 내 파일 정렬 작업을 사용하여 메모리에서 행을 정렬할 수 있습니다.

여러 행이 `ORDER BY` 열에 동일한 값을 갖는 경우 서버는 해당 행을 어떤 순서로든 자유롭게 반환할 수 있으며 전체 실행 계획에 따라 다르게 반환할 수 있습니다. 즉, 이러한 행의 정렬 순서는 정렬되지 않은 열에 대해 비결정적입니다.

실행 계획에 영향을 미치는 한 가지 요소는 LIMIT이므로 `LIMIT`가 있든 없든 `ORDER BY` 쿼리는 서로 다른 순서로 행을 반환할 수 있습니다. 카테고리 열을 기준으로 정렬되지만 `ID` 및 등급 열에 대해서는 비결정적인 이 쿼리를 예로 들어 보겠습니다:

```
mysql> SELECT * FROM ratings ORDER BY category;
+----+----------+ ---------+
| 아이디 | 카테고리 | 등급
+----+----------+ ---------+

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

`LIMIT`를 포함하면 각 카테고리 값 내의 행 순서에 영향을 줄 수 있습니다. 예를 들어, 이것은 유효한 쿼리 결과입니다:

```
mysql> SELECT * FROM ratings ORDER BY category LIMIT 5;
+----+----------+ ---------+
| 아이디 | 카테고리 | 등급
+----+----------+ ---------+

|  1 |         1 |    4.5 |
|  5 |         1 |    3.2 |
|  4 |         2 |    3.5 |
|  3 |         2 |    3.7 |
| 6 |2 |3.     5 |
+----+----------+ ---------+
```

각각의 경우 행은 SQL 표준에서 요구하는 `ORDER BY` 열을 기준으로 정렬됩니다.

`LIMIT`를 사용하거나 사용하지 않고 동일한 행 순서를 보장하는 것이 중요한 경우 `ORDER BY` 절에 추가 열을 포함시켜 순서를 결정적으로 만들 수 있습니다. 예를 들어 `ID` 값이 고유한 경우 다음과 같이 정렬하여 특정 카테고리 값에 대한 행이 `ID` 순서대로 표시되도록 할 수 있습니다:

```
mysql> SELECT * FROM ratings ORDER BY category, id;
+----+----------+ ---------+
| 아이디 | 카테고리 | 등급
+----+----------+ ---------+
| 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;
+----+----------+ ---------+
| 아이디 | 카테고리 | 등급
+----+----------+ ---------+
| 1 |1 |4      .5 |
| 5 |1 |3.     2 |
| 3 |2 |3.     7 |
```

```
| 4 |          2 |3.   5 |
| 6 |          2 |3.   5 |
+----+----------+ ---------+
```

ORDER BY 또는 GROUP BY와 LIMIT 절이 있는 쿼리의 경우, 옵티마이저는 기본적으로 정렬된 인덱스를 선택하려고 시도하며, 그렇게 하는 것이 쿼리 실행 속도를 높일 수 있는 것으로 보입니다. 다른 최적화를 사용하는 것이 더 빠를 수 있는 경우, optimizer_switch 시스템 변수의 prefer_ordering_index 플래그를 off로 설정하여 이 최적화를 해제할 수 있습니다.

예시: 먼저 다음과 같이 테이블 t를 생성하고 채웁니다:

```
# 테이블을 생성하고 채웁니다:

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

# [테이블 t에 일부 행 삽입 - 표시되지 않음]
```

prefer_ordering_index 플래그가 활성화되어 있는지 확인합니다:

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

다음 쿼리에는 LIMIT 절이 있으므로 가능한 경우 정렬된 인덱스를 사용할 것으로 예상합니다. 이 경우 EXPLAIN 출력에서 볼 수 있듯이 테이블의 기본 키를 사용합니다.

```
mysql> EXPLAIN SELECT c2 FROM t
    ->     WHERE id2 > 3
    ->     ORDER BY id1 ASC LIMIT 2\G
*************************** 1. 행 ***************************
           ID: 1
  select_type: SIMPLE 테이
        블: t
   파티션: NULL
         유형: 인덱스 가능_
키: i
          키입니다: PRIMARY
      key_len: 8
          참조:
         NULL 행: 2
     필터링됨: 70.00 추가: 사
        용 위치
```

이제 prefer_ordering_index 플래그를 비활성화하고 동일한 쿼리를 다시 실행합니다. 이번에는 인덱스 i(WHERE 절에 사용된 id2 열을 포함)와 파일 정렬을 사용합니다:

```
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. 행 ***************************
           ID: 1
  select_type: SIMPLE 테이
        블: t
   파티션: NULL
         유형: 범위 가능_키
: i
          KEY: I
      KEY_LEN: 8
```

```
          참조:
         NULL 행:
         14
     필터링됨: 100.00
        추가: 인덱스 조건 사용; 파일 정렬 사용
```

섹션 8.9.2, '전환 가능한 최적화'도 참조하세요.

8.2.1.20 함수 호출 최적화

MySQL 함수는 내부적으로 결정론적 또는 비결정론적이라는 태그가 붙습니다. 인수의 고정된 값이 주어졌을 때 함수가 다른 호출에 대해 다른 결과를 반환할 수 있는 경우 비결정적 함수입니다. 비결정적 함수의 예는 다음과 같습니다: `rand()`, `uuid()`.

함수에 비결정적 태그가 지정된 경우, `WHERE` 절에서 함수에 대한 참조는 모든 행(하나의 테이블에서 선택하는 경우) 또는 행 조합(다중 테이블 조인에서 선택하는 경우)에 대해 평가됩니다.

또한 MySQL은 인수가 테이블 열인지 상수 값인지에 따라 인수의 유형에 따라 함수를 평가할 시기를 결정합니다. 테이블 열을 인수로 사용하는 결정론적 함수는 해당 열의 값이 변경될 때마다 평가되어야 합니다.

비결정적 함수는 쿼리 성능에 영향을 미칠 수 있습니다. 예를 들어, 일부 최적화를 사용할 수 없거나 더 많은 잠금이 필요할 수 있습니다. 다음 설명에서는 `RAND()`를 사용하지만 다른 비결정적 함수에도 적용됩니다.

테이블 `t`에 이 정의가 있다고 가정합니다:

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

다음 두 쿼리를 고려해 보세요:

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

두 쿼리 모두 기본 키와의 동일성 비교 때문에 기본 키 조회를 사용하는 것처럼 보이지만 이는 첫 번째 쿼리에만 해당됩니다:

- 첫 번째 쿼리는 항상 최대 하나의 행을 생성하는데, 이는 상수 인수가 있는 `POW()`가 상수 값이며 인덱스 조회에 사용되기 때문입니다.
- 두 번째 쿼리에는 쿼리에서 일정하지는 않지만 실제로 테이블 `t`의 모든 행에 대해 새로운 값을 갖는 비결정론적 함수 `RAND()`를 사용하는 표현식이 포함되어 있습니다. 따라서 쿼리는 테이블의 모든 행을 읽고 각 행의 술어를 평가한 다음 기본 키가 임의의 값과 일치하는 모든 행을 출력합니다. 이 값은 `ID` 열 값과 `RAND()` 시퀀스의 값에 따라 0, 1 또는 여러 행이 될 수 있습니다.

비결정성의 효과는 `SELECT` 문에만 국한되지 않습니다. 이 `UPDATE` 문은 비결정 함수를 사용하여 수정할 행을 선택합니다:

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

아마도 의도는 기본 키가 표현식과 일치하는 행을 하나만 업데이트하는 것일 것입니다. 그러나 `ID` 열 값과 `RAND()` 시퀀스의 값에 따라 0개, 1개 또는 여러 개의 행을 업데이트할 수 있습니다.

방금 설명한 동작은 성능과 복제에 영향을 미칩니다:

- 비결정적 함수는 일정한 값을 생성하지 않으므로 최적화 도구는 인덱스 조회와 같이 다른 방식으로 적용할 수 있는 전략을 사용할 수 없습니다. 그 결과 테이블 스캔이 발생할 수 있습니다.
- `InnoDB는` 하나의 일치하는 행에 대해 단일 행 잠금을 사용하는 대신 범위 키 잠금으로 에스컬레이션할 수 있습니다.
- 결정론적으로 실행되지 않는 업데이트는 복제에 안전하지 않습니다.

어려움은 RAND() 함수가 테이블의 모든 행에 대해 한 번씩 평가된다는 사실에서 비롯됩니다. 여러 번의 함수 평가를 피하려면 다음 기술 중 하나를 사용하십시오:

- 비결정적 함수가 포함된 표현식을 별도의 문으로 이동하여 값을 변수에 저장합니다. 원래 문에서 표현식을 변수에 대한 참조로 바꾸면 옵티마이저가 상수 값으로 처리할 수 있습니다:

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

- 파생 테이블의 변수에 임의의 값을 할당합니다. 이 기법을 사용하면 변수가 WHERE 절의 비교에 사용되기 전에 변수에 값이 한 번 할당됩니다:

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

앞서 언급했듯이, WHERE 절에 비결정론적 표현식이 있으면 최적화가 방지되고 테이블 스캔이 발생할 수 있습니다. 그러나 다른 표현식이 결정론적일 경우 WHERE 절을 부분적으로 최적화할 수 있습니다. 예를 들어

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

옵티마이저가 partial_key를 사용하여 선택한 행 집합을 줄일 수 있는 경우 RAND()가 실행되는 횟수가 줄어들어 최적화에 대한 비결정성의 영향이 줄어듭니다.

8.2.1.21 창 기능 최적화

창 함수는 옵티마이저가 고려하는 전략에 영향을 줍니다:

- 하위 쿼리에 창 함수가 있는 경우 하위 쿼리에 대한 파생 테이블 병합이 비활성화됩니다. 하위 쿼리는 항상 구체화됩니다.
- 세미조인은 창 함수를 포함할 수 없는 WHERE 및 JOIN ... ON의 하위 쿼리에 적용되므로 창 함수 최적화에는 세미조인을 적용할 수 없습니다. ON의 하위 쿼리에 적용되므로 창 함수를 포함할 수 없습니다.
- 옵티마이저는 동일한 순서 요구 사항을 가진 여러 창을 순서대로 처리하므로 첫 번째 창 다음에 오는 창은 정렬을 건너뛸 수 있습니다.
- 옵티마이저는 단일 단계로 평가할 수 있는 창을 병합하려고 시도하지 않습니다(예: 여러 OVER 절에 동일한 창 정의가 포함된 경우). 해결 방법은 WINDOW 절에서 창을 정의하고 OVER 절에서 창 이름을 참조하는 것입니다.

창 함수로 사용되지 않는 집계 함수는 가능한 가장 바깥쪽 쿼리에서 집계됩니다. 예를 들어, 이 쿼리에서 MySQL은 COUNT(t1.b)가 WHERE 절에 배치되어 있기 때문에 외부 쿼리에는 존재할 수 없는 것으로 인식합니다:

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

결과적으로 MySQL은 하위 쿼리 내부에서 집계하여 t1.b를 상수로 취급하고 t2의 행 수를 반환합니다.

WHERE를 HAVING으로 바꾸면 오류가 발생합니다:

```
mysql> SELECT * FROM t1 HAVING t1.a = (SELECT COUNT(t1.b) FROM t2);
오류 1140 (42000): GROUP BY가 없는 집계 쿼리에서 SELECT 목록의 식 #1에 집계되지
않은 열 'test.t1.a'가 포함되어 있으며, 이는 sql_mode=only_full_group_by와 호환
되지 않습니다.
```

이 오류는 COUNT(t1.b)가 HAVING에 존재할 수 있으므로 외부 쿼리가 집계되기 때문에 발생합니다.

윈도우 함수(윈도우 함수로 사용되는 집계 함수 포함)에는 앞의 복잡성이 없습니다. 창 함수는 항상 작성된 하위 쿼리에서 집계되며, 외부 쿼리에서는 집계되지 않습니다.

창 함수 평가는 정밀도 손실 없이 창 연산을 계산할지 여부를 결정하는 windowing_use_high_precision 시스템 변수의 값에 영향을 받을 수 있습니다. 기본적으로 windowing_use_high_precision은 활성화되어 있습니다.

일부 움직이는 프레임 집계의 경우 역집계 함수를 적용하여 집계에서 값을 제거할 수 있습니다. 이렇게 하면 성능이 향상될 수 있지만 정밀도가 떨어질 수 있습니다. 예를 들어, 매우 큰 값에 매우 작은 부동 소수점 값을 추가하면 매우 작은 값이 큰 값에 의해 "숨겨지게" 됩니다. 나중에 큰 값을 반전하면 작은 값의 효과가 사라집니다.

역집계로 인한 정밀도 손실은 부동 소수점(근사값) 데이터 유형에 대한 연산에만 해당됩니다. 다른 유형의 경우 역집계는 안전하며, 여기에는 분수 부분을 허용하지만 정확한 값 유형인 DECIMAL이 포함됩니다.

빠른 실행을 위해 MySQL은 안전할 때 항상 역집계를 사용합니다:

- 부동 소수점 값의 경우 역집계가 항상 안전한 것은 아니며 정밀도가 손실될 수 있습니다. 기본값은 속도가 느리지만 정밀도가 유지되는 역 집계를 피하는 것입니다. 속도를 위해 안전성을 희생할 수 있는 경우, windowing_use_high_precision을 비활성화하여 역집계를 허용할 수 있습니다.
- 부동 소수점이 아닌 데이터 유형의 경우 역집계는 항상 안전하며, 부동 소수점이 아닌 데이터 유형의 창문_사용_높은_정확도 값입니다.
- windowing_use_high_precision은 어떤 경우에도 역집계를 사용하지 않는 MIN() 및 MAX()에는 영향을 미치지 않습니다.

분산 함수 STDDEV_POP(), STDDEV_SAMP(), VAR_POP(), VAR_SAMP() 및 그 동의어에 대한 평가는 최적화 모드 또는 기본 모드에서 수행될 수 있습니다. 최적화 모드는 마지막 유효 자릿수에서 약간 다른 결과를 생성할 수 있습니다. 이러한 차이가 허용되는 경우 최적화된 모드를 허용하기 위해 windowing_use_high_precision을 비활성화할 수 있습니다.

EXPLAIN의 경우 윈도우잉 실행 계획 정보가 너무 방대하여 기존 출력 형식으로 표시할 수 없습니다. 윈도우잉 정보를 보려면 EXPLAIN FORMAT=JSON을 사용하여 윈도우잉 요소를 찾습니다.

8.2.1.22 행 생성자 표현식 최적화

행 생성자를 사용하면 여러 값을 동시에 비교할 수 있습니다. 예를 들어, 이 두 문은 의미적으로 동일합니다:

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

또한 옵티마이저는 두 표현식을 동일한 방식으로 처리합니다.

행 생성자 열이 인덱스의 접두사를 포함하지 않는 경우 옵티마이저는 사용 가능한 인덱스를 사용할 가능성이 적습니다. 기본 키가 (c1, c2, c3)인 다음 테이블을 생각해 보겠습니다:

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

이 쿼리에서 WHERE 절은 인덱스의 모든 열을 사용합니다. 그러나 행 생성자 자체는 인덱스 접두사를 포함하지 않으므로 옵티마이저는 c1(key_len=4, c1의 크기)만 사용하게 됩니다:

```
mysql> EXPLAIN SELECT * FROM t1
       WHERE c1=1 AND (c2,c3) > (1,1)\G
*************************** 1. 행 ***************************
           ID: 1
  select_type: SIMPLE 테이
       블: t1
        파티션: NULL
          유형: ref
```

```
      가능한_키: PRIMARY
       키입니다: PRIMARY
      key_len: 4
          참조: const 행:
         3
     필터링됨: 100.00 추가: 사
        용 위치
```

이러한 경우 동등한 비구축자 식을 사용하여 행 생성자 식을 다시 작성하면 보다 완전한 인덱스 사용이 가능할 수 있습니다. 주어진 쿼리에 대해 행 생성자 및 동등한 비생성자 표현식은 다음과 같습니다:

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

비구축자 식을 사용하도록 쿼리를 다시 작성하면 인덱스의 세 열을 모두 사용하는 옵티마이저가 생성됩니다 (`key_len=12`):

```
mysql> EXPLAIN SELECT * FROM t1
       WHERE c1 = 1 AND (c2 > 1 OR ((c2 = 1) AND (c3 > 1)))\G
*************************** 1. 행 ***************************
           ID: 1
  select_type: SIMPLE 테이
        블: t1
   파티션: NULL
         유형: 범위 가능
한_키: PRIMARY
          키입니다: PRIMARY
      key_len: 12
          참조:
         NULL 행: 3
     필터링됨: 100.00 추가: 사
        용 위치
```

따라서 더 나은 결과를 얻으려면 행 생성자를 `AND/OR` 표현식과 혼합하지 마세요. 둘 중 하나만 사용하십시오.

특정 조건에서 옵티마이저는 행 생성자 인수가 있는 `IN()` 식에 범위 액세스 메서드를 적용할 수 있습니다. 행 생성자 식의 범위 최적화를 참조하십시오.

8.2.1.23 전체 테이블 스캔 피하기

MySQL이 쿼리를 해결하기 위해 전체 테이블 스캔을 사용하는 경우 `EXPLAIN`의 출력에는 유형 열에 `ALL`이 표시됩니다. 이는 일반적으로 다음과 같은 조건에서 발생합니다:

- 테이블이 너무 작아서 키 조회를 번거롭게 하는 것보다 테이블 스캔을 수행하는 것이 더 빠릅니다. 이는 행 수가 10개 미만이고 행 길이가 짧은 테이블에 일반적으로 사용됩니다.
- 인덱싱된 열에 대한 `ON` 또는 `WHERE` 절에는 사용 가능한 제한이 없습니다.
- 인덱싱된 열과 상수 값을 비교하고 있는데 MySQL에서 상수가 테이블의 너무 많은 부분을 차지하므로 테이블 스캔이 더 빠르다고 계산했습니다(인덱스 트리에 기반). 섹션 8.2.1.1, "WHERE 절 최적화"를 참조하

십시오.

- 카디널리티가 낮은 키(키 값과 일치하는 행이 많은 키)를 다른 열을 통해 사용하고 있습니다. 이 경우 MySQL은 키를 사용하면 많은 키 조회가 필요할 수 있으며 테이블 스캔이 더 빠를 것이라고 가정합니다.

작은 테이블의 경우 테이블 스캔이 적절하며 성능에 미치는 영향은 무시할 수 있습니다. 큰 테이블의 경우 최적화 도구가 테이블 스캔을 잘못 선택하지 않도록 다음 기술을 시도해 보십시오:

- 스캔한 테이블의 키 분포를 업데이트하려면 `ANALYZE TABLE tbl_name`을 사용합니다. 섹션 13.7.3.1, "테이블 분석 문"을 참조하세요.
- 스캔한 테이블에 강제 인덱스를 사용하여 MySQL에 테이블 스캔이 주어진 인덱스를 사용하는 것보다 매우 비싸다는 것을 알립니다:

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

섹션 8.9.4, '색인 힌트'를 참조하세요.

- `max-seeks-for-key=1000` 옵션으로 `mysqld`를 시작하거나 `SET` max_seeks_for_key=1000을 사용하여 최적화 프로그램에 키 스캔이 1,000개 이상의 키 찾기를 유발하지 않는 것으로 가정하도록 지시합니다. 섹션 5.1.8, "서버 시스템 변수"를 참조하십시오.

8.2.2 하위 쿼리, 파생 테이블, 뷰 참조 및 일반 테이블 식 최적화하기

MySQL 쿼리 최적화 도구에는 하위 쿼리를 평가하는 데 사용할 수 있는 다양한 전략이 있습니다:

- `IN`, `= ANY` 또는 `EXISTS` 술어와 함께 사용되는 하위 쿼리의 경우 옵티마이저에 다음과 같은 선택 사항이 있습니다:
  - 세미조인
  - 구체화
  - 기존 전략
- `NOT IN`, `<> ALL` 또는 `NOT EXISTS` 술어와 함께 사용되는 하위 쿼리의 경우 옵티마이저에 다음과 같은 선택 사항이 있습니다:
  - 구체화
  - 기존 전략

파생된 테이블의 경우 최적화 프로그램에는 이러한 선택 사항이 있습니다(뷰 참조 및 일반 테이블 식에도 적용됨):

- 파생된 테이블을 외부 쿼리 블록에 병합합니다.
- 파생된 테이블을 내부 임시 테이블로 구체화합니다.

다음 논의에서는 앞의 최적화 전략에 대한 자세한 정보를 제공합니다.

참고

하위 쿼리를 사용하여 단일 테이블을 수정하는 `UPDATE` 및 `DELETE` 문에 대한 제한 사항은 옵티마이저가 준조인 또는 구체화 하위 쿼리 최적화를 사용하지 않는다는 것입니다. 해결 방법으로, 하위 쿼리가 아닌 조인을 사용하는 다중 테이블 `UPDATE` 및 `DELETE` 문으로 다시 작성해 보십시오.

8.2.2.1 세미조인 및 반조인 변환으로 IN 및 EXISTS 하위 쿼리 조건 최적화하기

세미조인은 테이블 풀아웃, 중복 위드아웃, 퍼스트 매치, 루스 스캔, 구체화 등 여러 실행 전략을 가능하게 하는

준비 시간 변환입니다. 옵티마이저는 이 섹션에 설명된 대로 세미조인 전략을 사용하여 하위 쿼리 실행을 개선합니다.

두 테이블 간의 Inner 조인의 경우 조인은 다른 테이블에 일치하는 항목이 있는 횟수만큼 한 테이블에서 행을 반환합니다. 그러나 일부 질문의 경우 중요한 정보는 일치하는 항목의 수가 아니라 일치하는 항목이 있는지 여부입니다. 코스 커리큘럼의 수업과 수업 명단(각 수업에 등록한 학생)을 각각 나열하는 `class` 및 `roster`라는 이름의 테이블이 있다고 가정합니다. 실제로 학생이 등록한 수업을 나열하려면 이 조인을 사용할 수 있습니다:

```
SELECT class.class_num, class.class_name
    FROM class
    내부 조인 명단
```

```
    WHERE class.class_num = roster.class_num;
```

그러나 결과에는 등록된 각 학생에 대해 각 수업이 한 번씩 나열됩니다. 질문의 경우 이는 불필요한 정보 중복입니다.

`class_num`이 `클래스` 테이블의 기본 키라고 가정하면 `SELECT` DISTINCT를 사용하여 중복을 억제할 수 있지만, 일치하는 모든 행을 먼저 생성한 후 나중에 중복을 제거하는 것은 비효율적입니다.

하위 쿼리를 사용하여 동일한 중복 없는 결과를 얻을 수 있습니다:

```
SELECT class_num, class_name
    FROM class
    WHERE class_num IN
        (SELECT class_num FROM roster);
```

여기서 옵티마이저는 `IN` 절이 하위 쿼리가 `명단` 테이블에서 각 클래스 번호의 인스턴스를 하나만 반환하도록 요구한다는 것을 인식할 수 있습니다. 이 경우 쿼리는 *세미조인*, 즉 `로스터의` 행과 일치하는 `클래스` 내 각 행의 인스턴스 하나만 반환하는 연산을 사용할 수 있습니다.

`EXISTS` 하위 쿼리 술어가 포함된 다음 문은 `IN` 하위 쿼리 술어가 포함된 이전 문과 동일합니다:

```
SELECT class_num, class_name
    FROM class
    어디에 존재
        (SELECT * FROM roster WHERE class.class_num = roster.class_num);
```

`EXISTS` 하위 쿼리 조건이 있는 모든 문은 동등한 `IN` 하위 쿼리 조건이 있는 문과 동일한 세미조인 변환이 적용됩니다.

다음 하위 쿼리는 안티조인으로 변환됩니다:

- `없음 (...에서 ... 선택)`
- `존재하지 않음(...에서 ... 선택)`.
- `에서 (...에서 ...를 선택)이 참이 아닙니다.`
- `존재(...에서 ...를 선택)가 참이 아닙니다`.
- `에서 (...에서 ... 선택)이 거짓입니다.`
- `존재(...에서 ... 선택)가 거짓입니다`.

즉, `IN(SELECT ... FROM ...)` 또는 `EXISTS(SELECT ... FROM ...)` 형식의 하위 쿼리의 모든 부정은 안티조인으로 변환됩니다.

조인 반대는 일치하는 항목이 없는 행만 반환하는 연산입니다. 여기에 표시된 쿼리를 살펴보겠습니다:

```
SELECT class_num, class_name
    FROM class
    WHERE class_num NOT IN
        (SELECT class_num FROM roster);
```

이 쿼리는 내부적으로 다음과 같이 재작성됩니다. `SELECT class_num, class_name FROM class ANTIJOIN 로스터 ON class_num` 이 쿼리는 로스터의 어떤 행과도 일치하지 *않는* 클래스 내 각 행의 인스턴스를 하나씩 반환합니다. 즉, 클래스의 각 행에 대해 로스터에서 일치하는 행이 발견되는 즉시 클래스의 행을 버릴 수 있습니다.

비교 대상 표현식이 널 가능하면 대부분의 경우 반조인 변환을 적용할 수 없습니다. 이 규칙의 예외는 `(... NOT IN (SELECT ...)) IS NOT FALSE` 및 이에 상응하는 `(... IN (SELECT ...)) IS NOT TRUE`는 반조인으로 변환될 수 있습니다.

외부 쿼리 사양에서는 외부 조인 및 내부 조인 구문이 허용되며 테이블 참조는 기본 테이블, 파생 테이블, 뷰 참조 또는 일반 테이블 식일 수 있습니다.

MySQL에서 하위 쿼리는 이러한 기준을 충족해야 세미조인(또는 그렇지 않은 경우 안티조인)으로 처리할 수 있습니다.
하위 쿼리를 수정합니다):

- `WHERE` 또는
  `ON` 절을 `AND` 표현식의 용어로 사용할 수 있습니다. 예를 들어

  ```
  선택 ...
      FROM ot1, ...
      WHERE (oe1, ...) IN
          (SELECT ie1, ... FROM it1, ... WHERE ...);
  ```

  여기서 *`ot_i`* 및 *`it_i`*는 쿼리의 외부 및 내부에 있는 테이블을 나타내고, *`oe_i`* 및 *`ie_i`*는 쿼리의 외부 및 내부에 있는 테이블을 나타냅니다.
  는 외부 테이블과 내부 테이블의 열을 참조하는 표현식을 나타냅니다.

  하위 쿼리는 `NOT`, `IS [NOT] TRUE` 또는 `IS [NOT] FALSE`로 수정된 표현식에 대한 인수가 될 수도 있습니다.

- `UNION` 구문이 없는 단일 `SELECT`여야 합니다.

- `HAVING` 절을 포함하지 않아야 합니다.

- 명시적으로 또는 암시적으로 그룹화되어 있든 상관없이 집계 함수를 포함해서는 안 됩니다.

- `LIMIT` 절이 없어야 합니다.

- 문은 외부 쿼리에서 `STRAIGHT_JOIN` 조인 유형을 사용해서는 안 됩니다.

- `STRAIGHT_JOIN` 수정자가 없어야 합니다.

- 외부 테이블과 내부 테이블을 합한 수는 조인에 허용되는 최대 테이블 수보다 작아야 합니다.

- 하위 쿼리는 상관관계가 있거나 상관관계가 없을 수 있습니다. 상관관계는 `EXISTS`의 인수로 사용된 하위 쿼리의 `WHERE` 절에서 사소한 상관관계가 있는 술어를 살펴보고, 마치 `IN(SELECT b FROM ...)` 내에서 사용된 것처럼 최적화할 수 있게 해줍니다. *사소한 상관 관계라는* 용어는 해당 술어가 동등 술어이고, `WHERE` 절의 유일한 술어이며(또는 `AND`와 결합된 경우), 한 피연산자가 서브쿼리에서 참조된 테이블의 피연산자이고 다른 피연산자가 외부 쿼리 블록의 피연산자인 경우를 의미합니다.

- `DISTINCT` 키워드는 허용되지만 무시됩니다. 세미조인 전략은 중복 제거를 자동으로 처리합니다.

- 하위 쿼리에 하나 이상의 집계 함수가 포함되어 있지 않는 한 `GROUP BY` 절은 허용되지만 무시됩니다.

- 주문은 세미조인 전략의 평가와 관련이 없으므로 `ORDER BY` 절은 허용되지만 무시됩니다.

하위 쿼리가 앞의 기준을 충족하는 경우 MySQL은 이를 세미조인(또는 해당되는 경우 안티조인)으로 변환하고 이러한 전략 중에서 비용 기반 선택을 합니다:

- 하위 쿼리를 조인으로 변환하거나 테이블 풀아웃을 사용하여 하위 쿼리 테이블과 외부 테이블 간에 내부 조인으로 쿼리를 실행합니다. 테이블 풀아웃은 하위 쿼리에서 외부 쿼리로 테이블을 끌어옵니다.
- *중복 위드아웃*: 조인처럼 세미조인을 실행하고 임시 테이블을 사용하여 중복 레코드를 제거합니다.
- *FirstMatch*: 내부 테이블에서 행 조합을 스캔할 때 주어진 값 그룹의 인스턴스가 여러 개 있는 경우 모두 반환하지 않고 하나를 선택합니다. 이렇게 하면 스캔이 "단축"되어 불필요한 행이 생성되지 않습니다.

- *LooseScan*: 각 하위 쿼리의 값 그룹에서 단일 값을 선택할 수 있는 인덱스를 사용하여 하위 쿼리 테이블을 스캔합니다.

- 조인을 수행하는 데 사용되는 인덱싱된 임시 테이블로 하위 쿼리를 구체화하며, 여기서 인덱스는 중복을 제거하는 데 사용됩니다. 인덱스는 나중에 임시 테이블을 외부 테이블과 조인할 때 조회에 사용될 수도 있으며, 그렇지 않은 경우 테이블이 스캔됩니다. 구체화에 대한 자세한 내용은 섹션 8.2.2.2, "구체화를 사용하여 하위 쿼리 최적화"를 참조하십시오.

이러한 각 전략은 다음 `optimizer_switch` 시스템 변수 플래그를 사용하여 활성화 또는 비활성화할 수 있습니다:

- `세미조인` 플래그는 세미조인 및 안티조인 사용 여부를 제어합니다.
- `세미조인을` 사용하도록 설정하면 `첫 번째 일치`, `루즈 스캔`, `중복 제거` 및 `구체화` 플래그를 사용하면 허용된 세미조인 전략을 보다 세밀하게 제어할 수 있습니다.
- `중복 위드아웃` 세미조인 전략이 비활성화되어 있으면 다른 모든 해당 전략도 비활성화하지 않는 한 사용되지 않습니다.
- `중복` 제거가 비활성화되어 있으면 최적화 도구가 최적이 아닌 쿼리 계획을 생성하는 경우가 있습니다. 이는 탐욕스러운 검색 중 휴리스틱 프루닝으로 인해 발생하며, `optimizer_prune_level=0을` 설정하면 이를 방지할 수 있습니다.

이 플래그는 기본적으로 활성화되어 있습니다. 섹션 8.9.2, "전환 가능한 최적화"를 참조하세요.

옵티마이저는 뷰 및 파생 테이블 처리의 차이를 최소화합니다. 이는 `STRAIGHT_JOIN` 수정자를 사용하는 쿼리와 준조인으로 변환할 수 있는 `IN` 하위 쿼리가 있는 뷰에 영향을 미칩니다. 다음 쿼리는 처리 변경으로 인해 변환이 변경되고 따라서 실행 전략이 달라지기 때문에 이를 설명합니다:

```
CREATE VIEW v AS
SELECT *
FROM t1
WHERE a IN (SELECT b
            FROM t2);

SELECT STRAIGHT_JOIN *
FROM t3 JOIN v ON t3.x = v.a;
```

옵티마이저는 먼저 뷰를 살펴보고 `IN` 하위 쿼리를 준조인으로 변환한 다음 뷰를 외부 쿼리에 병합할 수 있는지 여부를 확인합니다. 외부 쿼리의 `STRAIGHT_JOIN` 수정자가 세미조인을 방지하므로 옵티마이저는 병합을 거부하고 구체화된 테이블을 사용하여 파생된 테이블을 평가합니다.

`EXPLAIN` 출력은 다음과 같이 세미조인 전략의 사용을 나타냅니다:

- 확장 `EXPLAIN` 출력의 경우 다음 `SHOW WARNINGS에` 표시되는 텍스트에 다시 작성된 쿼리가 표시되며, 이 쿼리는 세미조인 구조를 표시합니다. (섹션 8.8.3, "확장 EXPLAIN 출력 형식" 참조) 이를 통해 세미조인에서 어떤 테이블이 추출되었는지를 알 수 있습니다. 서브쿼리가 세미조인으로 변환된 경우, 서브쿼리 술어가 사라지고 해당 테이블과 `WHERE` 절이 외부 쿼리 조인 목록과 `WHERE` 절에 병합된 것을 볼 수 있습니다.

- 중복 제거를 위한 임시 테이블 사용은 추가 열에서 임시 시작 및 임시 종료로 표시됩니다. 뽑아내지 않고 임시 시작 및 임시 종료가 적용되는 설명 출력 행 범위에 있는 테이블은 임시 테이블에 해당 행이 표시됩니다.
- 추가 열의 `FirstMatch(tbl_name)`는 조인 바로 가기를 나타냅니다.
- 추가 열의 LooseScan `(m..n)`은 LooseScan 전략을 사용했음을 나타냅니다. m과 `n`은 주요 부품 번호입니다.
- 구체화를 위한 임시 테이블 사용은 `select_type` 값이 `MATERIALIZED` 및 테이블 값이 `<subqueryN>`인 행입니다.

문이 ORDER BY 또는 LIMIT를 사용하지 않고 옵티마이저 힌트 또는 옵티마이저_switch 설정에 의해 세미조인 변환이 허용되는 경우, [NOT] IN 또는 [NOT] EXISTS 서브쿼리 조건을 사용하는 단일 테이블 UPDATE 또는 DELETE 문에도 세미조인 변환을 적용할 수 있습니다.

8.2.2.2 구체화를 통한 하위 쿼리 최적화

옵티마이저는 구체화를 사용하여 보다 효율적인 하위 쿼리 처리를 가능하게 합니다. 구체화는 일반적으로 메모리에 임시 테이블로 하위 쿼리 결과를 생성하여 쿼리 실행 속도를 높입니다. MySQL은 처음 하위 쿼리 결과를 필요로 할 때 해당 결과를 임시 테이블로 구체화합니다. 이후에도 결과가 필요할 때마다 MySQL은 임시 테이블을 다시 참조합니다. 옵티마이저는 조회를 빠르고 저렴하게 하기 위해 해시 인덱스로 테이블을 인덱싱할 수 있습니다. 인덱스에는 중복을 제거하고 테이블을 더 작게 만들기 위한 고유 값이 포함됩니다.

하위 쿼리 구체화는 가능한 경우 메모리 내 임시 테이블을 사용하며, 테이블이 너무 커지면 온디스크 저장소로 다시 넘어갑니다. 섹션 8.4.4, "MySQL에서 내부 임시 테이블 사용"을 참조하세요.

구체화를 사용하지 않으면 옵티마이저가 상관 관계가 없는 하위 쿼리를 상관 관계가 있는 하위 쿼리로 재작성하는 경우가 있습니다. 예를 들어, 다음의 IN 하위 쿼리는 상관 관계가 없습니다(*여기서* *where_condition은* t1이 아닌 t2의 열만 포함):

```
SELECT * FROM t1
WHERE t1.a IN (SELECT t2.b FROM t2 WHERE where_condition);
```

옵티마이저는 이를 EXISTS 상관관계 하위 쿼리로 다시 작성할 수 있습니다:

```
SELECT * FROM t1
WHERE EXISTS (SELECT t2.b FROM t2 WHERE where_condition AND t1.a=t2.b);
```

임시 테이블을 사용한 하위 쿼리 구체화는 이러한 재작성을 방지하고 하위 쿼리를 외부 쿼리의 행당 한 번이 아닌 한 번만 실행할 수 있게 해줍니다.

MySQL에서 하위 쿼리 구체화를 사용하려면 optimizer_switch 시스템 변수 구체화 플래그를 활성화해야 합니다. (섹션 8.9.2, "전환 가능한 최적화" 참조) 구체화 플래그를 활성화하면 이러한 사용 사례에 해당하는 술어에 대해 선택 목록, WHERE, ON, GROUP BY, HAVING 또는 ORDER BY에 표시되는 하위 쿼리 술어에 구체화가 적용됩니다:

- 술어는 외부 표현식 *oe_i* 또는 내부 표현식 ie_i가 널 가능하지 않을 때 이 형식을 갖습니다. *N* 가 1 이상입니다.

  ```
  (oe_1, oe_2, ..., oe_N) [NOT] IN (SELECT ie_1, i_2, ..., ie_N ...)
  ```

- 술어는 하나의 외부 표현식 oe와 내부 표현식 *즉,* 표현식이 무효화될 수 있는 경우 이 형식을 갖습니다.

  ```
  oe [NOT] IN (SELECT 즉 ...)
  ```

- 술어는 IN 또는 NOT IN이며UNKNOWN(NULL)의 결과는 다음과 같은 의미를 갖습니다. FALSE.

다음 예는 UNKNOWN 및 FALSE 술어 평가의 동등성 요구 사항이 하위 쿼리 구체화 사용 가능 여부에 어떤 영

향을 미치는지 설명합니다. *어디_조건이* `t1`이 아닌 `t2`의 열만 포함하므로 하위 쿼리가 상관관계가 없다고 가정합니다.

이 쿼리는 구체화될 수 있습니다:

```
SELECT * FROM t1
WHERE t1.a IN (SELECT t2.b FROM t2 WHERE where_condition);
```

여기서 `IN` 술어가 `UNKNOWN`을 반환하는지 `FALSE`를 반환하는지는 중요하지 않습니다. 어느 쪽이든 T1은 쿼리 결과에 포함되지 않습니다.

하위 쿼리 구체화가 사용되지 않는 예는 다음 쿼리이며, 여기서 t2.b는 null 가능한 열입니다:

```
SELECT * FROM t1
WHERE (t1.a,t1.b) NOT IN (SELECT t2.a,t2.b FROM t2)
                          WHERE where_condition);
```

하위 쿼리 구체화 사용에는 다음과 같은 제한 사항이 적용됩니다:

- 내부 표현식과 외부 표현식의 유형이 일치해야 합니다. 예를 들어, 두 표현식이 모두 정수이거나 둘 다 소수인 경우 옵티마이저는 구체화를 사용할 수 있지만, 한 표현식이 정수이고 다른 표현식이 소수인 경우에는 사용할 수 없습니다.
- 내부 표현식은 BLOB일 수 없습니다.

쿼리와 함께 EXPLAIN을 사용하면 옵티마이저가 하위 쿼리 구체화를 사용하는지 여부를 어느 정도 알 수 있습니다:

- 구체화를 사용하지 않는 쿼리 실행과 비교하여 select_type이 DEPENDENT SUBQUERY에서 SUBQUERY로 변경될 수 있습니다. 이는 외부 행당 한 번 실행되는 하위 쿼리의 경우 구체화를 사용하면 하위 쿼리를 한 번만 실행할 수 있음을 나타냅니다.
- 확장된 설명 출력의 경우 다음 경고 표시로 표시되는 텍스트에는 다음이 포함됩니다.
  구체화 및 구체화-서브쿼리.

MySQL은 또한 [NOT] IN 또는 [NOT] EXISTS 하위 쿼리 술어를 사용하는 단일 테이블 UPDATE 또는 DELETE 문에 하위 쿼리 구체화를 적용할 수 있으며, 이 문이 ORDER BY 또는 LIMIT를 사용하지 않고 최적화 힌트 또는 optimizer_switch 설정에 의해 하위 쿼리 구체화가 허용되는 경우에 한합니다.

8.2.2.3 EXISTS 전략으로 하위 쿼리 최적화하기

IN(또는 =ANY) 연산자를 사용하여 하위 쿼리 결과를 테스트하는 비교에 특정 최적화를 적용할 수 있습니다. 이 섹션에서는 이러한 최적화, 특히 NULL 값으로 인해 발생하는 문제와 관련하여 이러한 최적화에 대해 설명합니다. 논의의 마지막 부분에서는 옵티마이저를 도울 수 있는 방법을 제안합니다.

다음 하위 쿼리 비교를 고려해 보세요:

```
outer_expr IN (SELECT inner_expr FROM ... WHERE subquery_where)
```

MySQL은 "외부에서 내부로" 쿼리를 평가합니다. 즉, 먼저 외부 표현식 *outer_expr의* 값을 가져온 다음 하위 쿼리를 실행하여 생성되는 행을 캡처합니다.

매우 유용한 최적화는 하위 쿼리에 관심 있는 행은 내부 표현식 inner_expr이 *outer_expr과* 같은 행뿐임을 "알려주는" 것입니다. 이 작업은 하위 쿼리의 WHERE 절에 적절한 등식을 밀어넣어 보다 제한적으로 만들면 됩니다. 변환된 비교는 다음과 같습니다:

```
EXISTS (SELECT 1 FROM ... WHERE subquery_where AND outer_expr=inner_expr)
```

변환 후 MySQL은 푸시다운 등식을 사용하여 하위 쿼리를 평가하기 위해 검사해야 하는 행의 수를 제한할 수 있습니다.

보다 일반적으로, *N값* 행을 반환하는 하위 쿼리에 대한 *N값* 비교는 동일한 변환을 적용받습니다. *oe_i* 및 *ie_i가* 해당하는 외부 및 내부 표현식 값을 나타내는 경우, 이 하위 쿼리 비교는 다음과 같습니다:

```
(oe_1, ..., oe_N) IN
  (SELECT ie_1, ..., ie_N FROM ... WHERE subquery_where)
```

됩니다:

```
EXISTS (SELECT 1 FROM ... WHERE subquery_where
                           AND oe_1 = ie_1
```

```
                          AND ...
                          AND oe_N = ie_N)
```

간단하게 설명하기 위해 다음 설명에서는 외부 및 내부 표현식 값의 단일 쌍을 가정합니다. 방금 설명한 '푸

시다운' 전략은 이 조건 중 하나라도 해당되는 경우 작동합니다:

- *outer_expr* 및 *inner_expr*은 NULL일 수 없습니다.
- 하위 쿼리 결과에서 NULL과 FALSE를 구분할 필요는 없습니다. 하위 쿼리가 OR의 일부인 경우 또는 AND 표현식을 사용하면 MySQL은 사용자가 신경 쓰지 않는다고 가정합니다. 옵티마이저가 NULL과 FALSE 하위 쿼리 결과를 구분할 필요가 없다는 것을 알아차리는 또 다른 사례는 이 구문입니다:

  ```
  ... WHERE outer_expr IN (서브쿼리)
  ```

  이 경우 WHERE 절은 IN(*하위 쿼리*)이 NULL 또는 FALSE를 반환하는지 여부에 관계없이 행을 거부합니다.

*outer_expr*이 NULL이 아닌 값으로 알려져 있지만 하위 쿼리가 *outer_expr* = *inner_expr*과 같은 행을 생성하지 않는다고 가정합니다. 그러면 *outer_expr* IN (SELECT ...)은 다음과 같이 평가됩니다:

- SELECT가 inner_expr이 NULL인 행을 생성하는 경우 NULL
- SELECT가 NULL이 아닌 값만 생성하거나 아무 것도 생성하지 않는 경우 FALSE

이 상황에서는 *outer_expr = inner_expr*인 행을 찾는 접근 방식이 더 이상 유효하지 않습니다. 이러한 행을 찾아야 하지만, 찾을 수 없는 경우 inner_expr이 NULL인 행도 찾아야 합니다. 대략적으로 말하면, 하위 쿼리는 다음과 같이 변환할 수 있습니다:

```
EXISTS (SELECT 1 FROM ... WHERE subquery_where AND
        (outer_expr=inner_expr OR inner_expr IS NULL))
```

추가 IS NULL 조건을 평가할 필요가 있기 때문에 MySQL에 ref_or_null 액세스 메서드가 있습니다:

```
mysql> 설명
       SELECT outer_expr IN (SELECT t2.maybe_null_key
                             FROM t2, t3 WHERE ...)
       FROM t1;
*************************** 1. 행 ***************************
           ID: 1
  select_type: PRIMARY 테이
        블: t1
...
*************************** 2. 행 ***************************
           ID: 2
  SELECT_TYPE: 종속 하위 쿼리 테이블:
        t2
         유형: REF_OR_NULL 가능_키:
어쩌면_NULL_키
          키: 어쩌면_널키 키_len: 5
          참조: 함수
         행: 2
        추가: 어디 사용; 인덱스 사용
...
```

`unique_subquery` 및 `index_subquery` 하위 쿼리별 액세스 메서드에도 "또는 `NULL`" 변형.

추가 `OR ... IS NULL` 조건이 추가되면 쿼리 실행이 약간 더 복잡해지지만(그리고 하위 쿼리 내에서 일부 최적화를 적용할 수 없게 되지만) 일반적으로 견딜 수 있는 수준입니다.

*`outer_expr`*이 `NULL`일 수 있는 경우 상황은 훨씬 더 나빠집니다. SQL 해석에 따르면 `NULL`을 "알 수 없는 값"으로, `NULL IN (SELECT` *`inner_expr`* `...)`은 다음과 같이 평가해야 합니다:

- `SELECT`가 행을 생성하는 경우 `NULL`

- SELECT가 행을 생성하지 않는 경우 FALSE

적절한 평가를 위해서는 SELECT가 행을 생성했는지 여부를 확인할 수 있어야 하므로 *outer_expr = inner_expr을* 하위 쿼리로 푸시 다운할 수 없습니다. 이는 동등성을 푸시다운할 수 없는 경우 많은 실제 하위 쿼리가 매우 느려지기 때문에 문제가 됩니다.

기본적으로 하위 쿼리를 실행하는 방식은
outer_expr.

옵티마이저는 속도보다 SQL 준수를 선택하므로 *outer_expr*
가 NULL일 수 있습니다:

- outer_expr이 NULL인 경우, 다음 식을 평가하기 위해서는 SELECT
  를 사용하여 행이 생성되는지 여부를 확인합니다:

  ```
  NULL IN (SELECT inner_expr FROM ... WHERE subquery_where)
  ```

  여기서 앞서 언급한 종류의 푸시다운 평등 없이 원래의 SELECT를 실행해야 합니다.

- 반면에 outer_expr이 NULL이 아닌 경우 이 비교는 절대적으로 필수적입니다:

  ```
  outer_expr IN (SELECT inner_expr FROM ... WHERE subquery_where)
  ```

  푸시다운 조건을 사용하는 이 표현식으로 변환합니다:

  ```
  EXISTS (SELECT 1 FROM ... WHERE subquery_where AND outer_expr=inner_expr)
  ```

  이 변환이 없으면 하위 쿼리 속도가 느려집니다.

조건을 하위 쿼리로 푸시다운할지 여부에 대한 딜레마를 해결하기 위해 조건을 "트리거" 함수 안에 래핑합니다. 따라서 다음과 같은 형식의 표현식을 사용할 수 있습니다:

```
outer_expr IN (SELECT inner_expr FROM ... WHERE subquery_where)
```

로 변환됩니다:

```
EXISTS (SELECT 1 FROM ... WHERE subquery_where
                          AND trigcond(outer_expr=inner_expr)))
```

보다 일반적으로 하위 쿼리 비교가 여러 쌍의 외부 및 내부 표현식을 기반으로 하는 경우 변환은 이 비교를 수행합니다:

```
(oe_1, ..., oe_N) IN (SELECT ie_1, ..., ie_N FROM ... WHERE subquery_where)
```

그리고 이 표현식으로 변환합니다:

```
EXISTS (SELECT 1 FROM ... WHERE subquery_where
                          AND trigcond(oe_1=ie_1)
                          AND ...
                          AND trigcond(oe_N=ie_N)
       )
```

각 trigcond(*X*)는 다음 값으로 평가되는 특수 함수입니다:

- "연결된" 외부 표현식 oe_i가 NULL이 아닌 경우 *X를 반환합니다.*
- "연결된" 외부 표현식 oe_i가 NULL인 경우 TRUE

참고

트리거 함수는 CREATE TRIGGER로 생성하는 종류의 트리거가 *아닙니다*.

trigcond() 함수 안에 래핑된 주식은 쿼리 최적화 프로그램에 대한 일급 술어가 아닙니다. 대부분의 최적화는 쿼리에서 켜고 끌 수 있는 술어를 처리할 수 없습니다.

실행 시간이 길어지므로 트리그콘드(X)를 알 수 없는 함수로 가정하고 무시합니다. 트리거된 이퀄리티는 이러한 최적화에서 사용할 수 있습니다:

- 참조 최적화: trigcond(X=Y [또는 Y IS NULL])를 사용하여 ref, eq_ref 또는 ref_or_null 테이블 액세스를 구성할 수 있습니다.
- 인덱스 조회 기반 하위 쿼리 실행 엔진: trigcond(X=Y)를 사용하여 다음과 같이 구성할 수 있습니다. 고유_서브쿼리 또는 인덱스_서브쿼리 액세스.
- 테이블 조건 생성기: 하위 쿼리가 여러 테이블의 조인인 경우 트리거된 조건이 가능한 한 빨리 확인됩니다.

옵티마이저가 트리거된 조건을 사용하여 일종의 인덱스 조회 기반 액세스를 생성하는 경우(앞 목록의 처음 두 항목과 마찬가지로), 조건이 꺼진 경우에 대비한 대체 전략이 있어야 합니다. 이 폴백 전략은 항상 동일합니다. 전체 테이블 스캔을 수행합니다. 설명 출력에서 폴백은 추가 열의 NULL 키에서 전체 스캔으로 표시됩니다:

```
mysql> EXPLAIN SELECT t1.col1,
       t1.col1 IN (SELECT t2.key1 FROM t2 WHERE t2.col2=t1.col2) FROM t1\G
*************************** 1. 행 ***************************
           ID: 1
  select_type: PRIMARY 테이
        블: t1
        ...
*************************** 2. 행 ***************************
           ID: 2
  SELECT_TYPE: 종속 하위 쿼리 테이블:
        t2
         유형: index_subquery
possible_keys: key1
          키: 키1 키
      _len: 5
          참조: 함수
         행: 2
        추가: 사용 위치; NULL 키에서 전체 스캔
```

설명에 이어 경고 표시를 실행하면 트리거된 상태를 확인할 수 있습니다:

```
*************************** 1. 행 *************************** 레
  벨: 참고
   코드: 1003
메시지: `test`.`t1`.`col1`을 `col1`으로 선택합니다,
         <in_optimizer>(`test`.`t1`.`col1`,
         <exists>(<index_lookup>(<cache>(`test`.`t1`.`col1`)에서 키1
         의 t2가 NULL을 확인합니다.
         where (`test`.`t2`.`col2` = `test`.`t1`.`col2`) having
         trigcond(<is_not_null_test>(`test`.`t2`.`key1`))))) AS
         test`.`t1`에서 `t1.col1 IN (t2에서 t2.key1 선택 여기서
         t2.col2=t1.col2)`를 선택합니다.
```

트리거된 조건을 사용하면 성능에 몇 가지 영향이 있습니다. 이전에는 그렇지 않던 NULL IN (SELECT ...) 식이 이제 전체 테이블 스캔(느린)을 유발할 수 있습니다. 이는 정확한 결과를 얻기 위해 지불하는 대가

입니다(트리거 조건 전략의 목표는 속도가 아니라 규정 준수를 개선하는 것입니다).

다중 테이블 하위 쿼리의 경우, 조인 옵티마이저가 외부 식이 `NULL`인 경우에 대해 최적화하지 않기 때문에 `NULL IN (SELECT ...)`의 실행이 특히 느립니다. 이는 왼쪽에 `NULL`이 있는 하위 쿼리 평가가 매우 드물다고 가정하며, 그렇지 않음을 나타내는 통계가 있더라도 마찬가지입니다. 반면에, 외부 식이 `NULL`일 수 있지만 실제로는 NULL이 아닌 경우에는 성능 저하가 없습니다.

쿼리 최적화 도구가 쿼리를 더 잘 실행할 수 있도록 다음 제안 사항을 사용하세요:

- 열이 실제로 존재하는 경우 `NOT NULL`로 선언합니다. 이렇게 하면 열에 대한 조건 테스트를 간소화하여 옵티마이저의 다른 측면에도 도움이 됩니다.
- 하위 쿼리 결과에서 NULL과 `FALSE`를 구분할 필요가 없는 경우 실행 경로가 느려지는 것을 쉽게 피할 수 있습니다. 다음과 같은 비교를 대체합니다:

```
outer_expr [NOT] IN (SELECT inner_expr FROM ...)
```

이 표현식을 사용합니다:

```
(outer_expr IS NOT NULL) AND (outer_expr [NOT] IN (SELECT inner_expr FROM ...))
```

그러면 표현식 결과가 명확해지면 MySQL이 `AND` 부분의 평가를 중지하기 때문에 `NULL IN (SELECT ...)`은 평가되지 않습니다.

또 다른 재작성 가능성:

```
[NOT] EXISTS (SELECT inner_expr FROM ...
        WHERE inner_expr=외부_expr)
```

`optimizer_switch` 시스템 변수의 `subquery_materialization_cost_based` 플래그를 사용하면 서브쿼리 구체화와 `IN-to-EXISTS` 서브쿼리 변환 간의 선택에 대한 제어가 가능합니다. 섹션 8.9.2, "전환 가능한 최적화"를 참조하세요.

8.2.2.4 병합 또는 구체화를 통한 파생 테이블, 뷰 참조 및 일반 테이블 표현식 최적화

옵티마이저는 두 가지 전략을 사용하여 파생된 테이블 참조를 처리할 수 있습니다(뷰 참조 및 일반 테이블 표현식에도 적용됨):

- 파생된 테이블을 외부 쿼리 블록에 병합합니다.
- 파생된 테이블을 내부 임시 테이블로 구체화 예제 1:

```
SELECT * FROM (SELECT * FROM t1) AS derived_t1;
```

파생 테이블 `derived_t1`을 병합하면 해당 쿼리는 다음과 유사하게 실행됩니다:

```
SELECT * FROM t1;
```

예 2:

```
SELECT *
  FROM t1 JOIN (SELECT t2.f1 FROM t2) AS derived_t2 ON t1.f2=derived_t2.f1
  WHERE t1.f1 > 0;
```

파생 테이블 `derived_t2`를 병합하면 해당 쿼리는 다음과 유사하게 실행됩니다:

```
SELECT t1.*, t2.f1
  FROM t1 JOIN t2 ON t1.f2=t2.f1
  WHERE t1.f1 > 0;
```

구체화를 사용하면 파생된_t1과 파생된_t2는 각각 해당 쿼리 내에서 별도의 테이블로 취급됩니다.

옵티마이저는 파생 테이블, 뷰 참조 및 공통 테이블 표현식을 동일한 방식으로 처리합니다: 가능한 경우 불필요한 구체화를 피하여 외부 쿼리에서 파생 테이블로 조건을 푸시다운하고 보다 효율적인 실행 계획을 생성할 수 있습니다. (예는 섹션 8.2.2.2, '구체화를 사용하여 하위 쿼리 최적화'를 참조하십시오.)

병합으로 인해 61개 이상의 기본 테이블을 참조하는 외부 쿼리 블록이 생성되는 경우, 옵티마이저는 대신 구체화를

선택합니다.

옵티마이저는 이러한 조건이 모두 참이면 파생 테이블 또는 뷰 참조의 `ORDER BY` 절을 외부 쿼리 블록으로 전파합니다:

- 외부 쿼리는 그룹화되거나 집계되지 않습니다.

- 외부 쿼리에는 DISTINCT, HAVING 또는 ORDER BY가 지정되어 있지 않습니다.
- 외부 쿼리에는 이 파생된 테이블 또는 뷰 참조가 FROM 절의 유일한 원본으로 사용됩니다. 그렇지 않으

면 옵티마이저가 ORDER BY 절을 무시합니다.

옵티마이저가 파생 테이블, 뷰 참조 및 공통 테이블 표현식을 외부 쿼리 블록에 병합하려고 시도할지 여부에 영향을 줄 수 있는 방법은 다음과 같습니다:

- MERGE 및 NO_MERGE 옵티마이저 힌트를 사용할 수 있습니다. 병합을 방지하는 다른 규칙이 없다는 가정 하에 적용됩니다. 섹션 8.9.3, "최적화 프로그램 힌트"를 참조하세요.
- 마찬가지로 optimizer_switch 시스템 변수의 derived_merge 플래그를 사용할 수 있습니다. 섹션 8.9.2, "전환 가능한 최적화"를 참조하세요. 기본적으로 이 플래그는 병합을 허용하도록 활성화되어 있습니다. 이 플래그를 비활성화하면 병합을 방지하고 ER_UPDATE_TABLE_USED 오류를 방지할 수 있습니다.

  파생_병합 플래그는 ALGORITHM 절이 포함되지 않은 뷰에도 적용됩니다. 따라서 하위 쿼리에 해당하는 식을 사용하는 뷰 참조에 대해 ER_UPDATE_TABLE_USED 오류가 발생하는 경우 뷰 정의에 ALGORITHM=TEMPTABLE을 추가하면 병합이 방지되고 derived_merge 값보다 우선합니다.
- 병합을 방지하는 구문을 하위 쿼리에 사용하여 병합을 비활성화할 수 있지만, 구체화에 미치는 영향이 명시적이지는 않습니다. 병합을 방지하는 구조는 파생 테이블, 공통 테이블 식 및 뷰 참조에 대해 동일하게 적용됩니다:
  - 집계 함수 또는 윈도우 함수(합계(), 최소(), 최대(), 카운트() 등)
  - 구별
  - 그룹 기준
  - 보유
  - LIMIT
  - 유니온 또는 유니온 모두
  - 선택 목록의 하위 쿼리
  - 사용자 변수에 대한 할당
  - 리터럴 값만 참조합니다(이 경우 기본 테이블이 없음).

옵티마이저가 파생 테이블에 대해 병합 대신 구체화 전략을 선택하면 다음과 같이 쿼리를 처리합니다:

- 옵티마이저는 쿼리 실행 중에 해당 내용이 필요할 때까지 파생된 테이블 구체화를 연기합니다. 이렇게 하면 구체화를 지연시키면 구체화를 전혀 하지 않아도 되므로 성능이 향상됩니다. 파생된 테이블의 결과를 다른

테이블에 조인하는 쿼리를 생각해 보겠습니다: 옵티마이저가 다른 테이블을 먼저 처리한 후 행이 반환되지 않는다는 것을 발견하면 조인을 더 이상 수행할 필요가 없으며 옵티마이저는 파생 테이블의 구체화를 완전히 건너뛸 수 있습니다.

- 쿼리 실행 중에 옵티마이저는 파생된 테이블에 인덱스를 추가하여 테이블에서 행 검색 속도를 높일 수 있습니다.

파생 테이블이 포함된 `SELECT` 쿼리에 대해 다음 `EXPLAIN` 문을 고려합니다:

```
EXPLAIN SELECT * FROM (SELECT * FROM t1) AS derived_t1;
```

옵티마이저는 `SELECT` 실행 중에 결과가 필요할 때까지 지연시켜 파생된 테이블을 구체화하지 않습니다. 이 경우 쿼리가 실행되지 않으므로(`EXPLAIN` 문에서 발생하므로) 결과가 필요하지 않습니다.

실행된 쿼리의 경우에도 파생된 테이블 구체화가 지연되면 옵티마이저가 구체화를 완전히 피할 수 있습니다. 이 경우, 구체화를 수행하는 데 필요한 시간만큼 쿼리 실행이 더 빨라집니다. 파생된 테이블의 결과를 다른 테이블에 조인하는 다음 쿼리를 생각해 보겠습니다:

```
SELECT *
  FROM t1 JOIN (SELECT t2.f1 FROM t2) AS derived_t2
          ON t1.f2=derived_t2.f1
  WHERE t1.f1 > 0;
```

최적화가 `t1`을 먼저 처리하고 `WHERE` 절이 빈 결과를 생성하는 경우 조인은 반드시 비어 있어야 하며 파생된 테이블을 구체화할 필요가 없습니다.

파생된 테이블에 구체화가 필요한 경우, 옵티마이저는 구체화된 테이블에 인덱스를 추가하여 테이블에 대한 액세스 속도를 높일 수 있습니다. 이러한 인덱스를 통해 테이블에 대한 참조 액세스를 활성화하면 쿼리 실행 중에 읽는 데이터의 양을 크게 줄일 수 있습니다. 다음 쿼리를 살펴보겠습니다:

```
SELECT *
 FROM t1 JOIN (SELECT DISTINCT f1 FROM t2) AS derived_t2
         ON t1.f1=derived_t2.f1;
```

옵티마이저는 가장 낮은 비용의 실행 계획을 위해 참조 액세스를 사용할 수 있는 경우 `derived_t2`에서 `f1` 열에 대한 인덱스를 구성합니다. 인덱스를 추가한 후 옵티마이저는 구체화된 파생 테이블을 인덱스가 있는 일반 테이블과 동일하게 취급할 수 있으며, 이 테이블은
생성된 인덱스입니다. 인덱스 생성에 따른 오버헤드는 인덱스가 없는 쿼리 실행 비용에 비해 무시할 수 있는 수준입니다. 참조 액세스가 다른 액세스 방법보다 더 높은 비용을 초래하는 경우, 옵티마이저는 인덱스를 생성하지 않고 아무 것도 잃지 않습니다.

옵티마이저 추적 출력의 경우 병합된 파생 테이블 또는 뷰 참조는 노드로 표시되지 않습니다. 상위 쿼리의 계획에는 해당 테이블의 기초 테이블만 표시됩니다.

파생 테이블의 구체화에 해당하는 내용은 공통 테이블 표현식(CTE)에도 적용됩니다. 또한 다음 고려 사항은 특히 CTE와 관련이 있습니다.

쿼리에 의해 CTE가 구체화되면 쿼리가 여러 번 참조하더라도 쿼리에 대해 한 번만 구체화됩니다.

재귀적 CTE는 항상 구체화됩니다.

CTE가 구체화되면 옵티마이저는 인덱싱을 통해 CTE에 대한 최상위 문에 의한 액세스 속도를 높일 수 있다고 판단되는 경우 관련 인덱스를 자동으로 추가합니다. 이는 파생 테이블의 자동 인덱싱과 유사하지만, CTE가 여러 번 참조되는 경우 옵티마이저가 여러 인덱스를 생성하여 각 참조에 의한 액세스 속도를 가장 적절한 방식으로 높일 수 있다는 점을 제외하면 다릅니다.

`MERGE` 및 `NO_MERGE` 옵티마이저 힌트를 CTE에 적용할 수 있습니다. 최상위 문에 있는 각 CTE 참조는 고유한 힌트를 가질 수 있으며, 이를 통해 CTE 참조를 선택적으로 병합하거나
구체화되었습니다. 다음 문은 힌트를 사용하여 cte1과 `cte2`를 병합해야 함을 나타냅니다.
를 구체화해야 합니다:

```
WITH
  cte1 AS (SELECT a, b FROM table1),
  cte2 AS (SELECT c, d FROM table2)
SELECT /*+ MERGE(cte1) NO_MERGE(cte2) */ cte1.b, cte2.d
FROM cte1 JOIN cte2
WHERE cte1.a = cte2.c;
```

CREATE VIEW의 ALGORITHM 절은 뷰 정의에서 SELECT 문 앞에 오는 WITH 절에 대한 구체화에 영향을 주지 않습니다. 이 문을 고려하십시오:

```
create algorithm={temptable|merge} VIEW v1 AS WITH ... SELECT ...
```

ALGORITHM 값은 WITH 절이 아닌 SELECT의 구체화에만 영향을 줍니다.

앞서 언급했듯이 CTE는 여러 번 참조되더라도 한 번만 구체화됩니다. 일회성 구체화를 나타내기 위해 옵티마이저 추적 출력에는 생성_tmp_table의 발생과 하나 이상의 재사용_tmp_table의 발생이 포함됩니다.

CTE는 파생 테이블과 유사하며, 이 경우 materialized_from_subquery 노드가 참조를 따릅니다. 이는 여러 번 참조되는 CTE의 경우에 해당하므로, (하위 쿼리가 여러 번 실행되어 불필요하게 장황한 출력을 생성하는) materialized_from_subquery 노드가 중복되지 않습니다. CTE에 대한 참조에는 서브쿼리 계획에 대한 설명이 포함된 완전한 materialized_from_subquery 노드가 하나만 있습니다. 다른 참조에는 reduced_from_subquery 노드가 있습니다. 동일한 아이디어가 EXPLAIN 출력을 TRADITIONAL 형식으로 출력합니다: 다른 참조에 대한 하위 쿼리는 표시되지 않습니다.

8.2.2.5 파생 조건 푸시다운 최적화

MySQL은 적격 하위 쿼리에 대해 파생 조건 푸시다운을 지원합니다. SELECT와 같은 쿼리의 경우 * FROM (SELECT i, j FROM t1) AS dt WHERE i > *상수인 경우,* 많은 경우 외부 WHERE 조건을 파생 테이블로 푸시할 수 있으며, 이 경우 SELECT * FROM (SELECT i, j FROM t1 WHERE i > *상수*) AS dt가 됩니다. 파생 테이블을 외부 쿼리에 병합할 수 없는 경우(예: 파생 테이블이 집계를 사용하는 경우) 외부 WHERE 조건을 파생 테이블로 푸시하면 처리해야 하는 행 수가 감소하여 쿼리 실행 속도가 빨라집니다.

다음과 같은 상황에서 외부 WHERE 조건을 파생된 구체화된 테이블로 푸시다운할 수 있습니다:

- 파생 테이블에 집계 또는 창 함수를 사용하지 않는 경우 외부 WHERE 조건을 직접 아래로 푸시할 수 있습니다. 여기에는 여러 개의 술어가 AND, OR 또는 둘 다로 결합된 WHERE 조건이 포함됩니다.

  예를 들어 SELECT * FROM (SELECT f1, f2 FROM t1) AS dt WHERE f1 < 3 AND f2 > 11 쿼리는 SELECT f1, f2 FROM (SELECT f1, f2 FROM t1 WHERE f1 < 3 AND f2 > 11) AS dt로 재작성됩니다.

- 파생 테이블에 GROUP BY가 있고 창 함수를 사용하지 않는 경우 GROUP BY에 속하지 않는 하나 이상의 열을 참조하는 외부 WHERE 조건을 HAVING 조건으로 파생 테이블로 푸시다운할 수 있습니다.

  예를 들어, SELECT * FROM (SELECT i, j, SUM(k) AS 합계 FROM t1 GROUP BY i, j) AS dt WHERE sum > 100은 파생 조건 푸시다운에 따라 SELECT * FROM (SELECT i, j, SUM(k) AS 합계 FROM t1 GROUP BY i, j HAVING 합계 > 100) AS dt로 재작성됩니다.

- 파생 테이블이 GROUP BY를 사용하고 외부 WHERE 조건의 열이 GROUP BY 열인 경우, 해당 열을 참조하는 WHERE 조건을 파생 테이블로 직접 푸시다운할 수 있습니다.

  예를 들어 SELECT * FROM (SELECT i,j, SUM(k) AS sum FROM t1 GROUP BY i,j) AS dt WHERE i > 10 쿼리는 SELECT * FROM (SELECT i,j, SUM(k) AS sum FROM t1 WHERE i > 10 GROUP BY i,j) AS dt로 재작성됩니다.

  외부 WHERE 조건에 GROUP BY에 속하는 열을 참조하는 술어와 그렇지 않은 열을 참조하는 술어가 있는

경우 전자의 술어는 `WHERE` 조건으로, 후자의 술어는 `HAVING` 조건으로 푸시 다 운 됩 니 다 . 예를 들어 `SELECT * FROM (SELECT i, j, SUM(k) AS sum FROM t1 GROUP BY i,j) AS dt WHERE i > 10 AND sum > 100` 쿼리에서 외부 `WHERE` 절의 술어 `i > 10`은 `GROUP BY` 칼럼을 참조하는 반면, 술어 `sum > 100`은 `GROUP BY` 칼럼을 참조하지 않습니다. 따라서 파생된 테이블 푸시다운 최적화는 여기에 표시된 것과 유사한 방식으로 쿼리를 다시 작성하게 합니다:

```
에서 * 선택 (
    SELECT i, j, SUM(k) AS sum FROM t1
        WHERE i > 10
```

```
        합계가 100을 초과하는
        i, j로 그룹화
    ) AS dt;
```

파생 조건 푸시다운을 활성화하려면 `optimizer_switch` 시스템 변수의 `derived_condition_pushdown` 플래그(이번 릴리스에 추가됨)를 기본 설정인 on으로 설정해야 합니다. `optimizer_switch`에 의해 이 최적화가 비활성화되어 있는 경우, 특정 쿼리에 대해 `DERIVED_CONDITION_PUSHDOWN` 최적화 힌트를 사용하여 이를 활성화할 수 있습니다. 특정 쿼리에 대한 최적화를 비활성화하려면 `NO_DERIVED_CONDITION_PUSHDOWN` 최적화 힌트를 사용합니다.

파생된 테이블 조건 푸시다운 최적화에는 다음과 같은 제한 사항 및 제한 사항이 적용됩니다:

- 파생된 테이블 조건 푸시다운 최적화는 다음과 같은 예외를 제외하고 `UNION` 쿼리에 사용할 수 있습니다:
  - `UNION`의 일부인 구체화된 파생 테이블이 재귀적 공통 테이블 표현식인 경우 조건 푸시다운을 `UNION` 쿼리와 함께 사용할 수 없습니다(재귀적 공통 테이블 표현식 참조).
  - 비결정적 표현식이 포함된 조건은 파생 테이블로 푸시다운할 수 없습니다.
- 파생 테이블은 `LIMIT` 절을 사용할 수 없습니다.
- 하위 쿼리가 포함된 조건은 푸시다운할 수 없습니다.
- 파생된 테이블이 외부 조인의 내부 테이블인 경우 최적화를 사용할 수 없습니다.
- 구체화된 파생 테이블이 공통 테이블 표현식인 경우 여러 번 참조되는 경우 조건이 해당 테이블로 푸시되지 않습니다.
- 매개 변수를 사용하는 조건은 조건이 *파생된_열* 형식인 경우 푸시다운할 수 있습니다. `> ?.` 외부 `WHERE` 조건의 파생 열이 기초 파생 테이블에 `?`이 있는 식인 경우 이 조건은 푸시다운할 수 없습니다.
- 조건이 뷰 자체에 있는 것이 아니라 `ALGORITHM=TEMPTABLE`을 사용하여 만든 뷰의 테이블에 있는 쿼리의 경우, 다중 동등성이 해결 단계에서 인식되지 않으므로 조건이 푸시다운되지 않을 수 있습니다. 이는 쿼리를 최적화할 때 조건 푸시다운이 해결 단계에서 발생하는 반면 다중 등식 전파는 최적화 중에 발생하기 때문입니다.

  이 경우 ALGORITHM=MERGE를 사용하는 뷰의 경우 동등성이 전파되고 조건이 아래로 밀려날 수 있으므로 문제가 되지 않습니다.
- 파생 테이블의 `SELECT` 목록에 사용자 변수에 대한 할당이 포함된 경우 조건을 푸시다운할 수 없습니다.

8.2.3 INFORMATION_SCHEMA 최적화 쿼리

데이터베이스를 모니터링하는 애플리케이션은 `INFORMATION_SCHEMA` 테이블을 자주 사용할 수 있습니다. 이러한 테이블에 대한 쿼리를 가장 효율적으로 작성하려면 다음과 같은 일반적인 지침을 따르세요:

- 데이터 사전 테이블에 대한 뷰인 `INFORMATION_SCHEMA` 테이블만 쿼리해 보십시오.
- 정적 메타데이터에 대해서만 쿼리하세요. 열을 선택하거나 정적 메타데이터와 함께 동적 메타데이터에 대한 검색 조건을 사용하면 동적 메타데이터를 처리하는 데 오버헤드가 추가됩니다.

참고

`INFORMATION_SCHEMA` 쿼리에서 데이터베이스 및 테이블 이름에 대한 비교 동작이 예상과 다를 수 있습니다. 자세한 내용은 섹션 10.8.7, "INFORMATION_SCHEMA 검색에서 데이터 정렬 사용"을 참조하세요.

이러한 `INFORMATION_SCHEMA` 테이블은 데이터 사전 테이블에 대한 뷰로 구현되므로 해당 테이블을 쿼리하면 데이터 사전에서 정보를 검색합니다:

```
문자_세트 체크_제약 조
건 콜레이션
콜레이션_문자_설정_적용 가능성 열
이벤트
파일
innodb_columns
innodb_datafiles
innodb_fields
innodb_foreign
innodb_foreign_cols
innodb_indexes
innodb_tables
innodb_tablespaces
innodb_tablespaces_brief
innodb_tablestats
key_column_사용 매개변수
파티션 참조_제약 조건 자원_
그룹 루틴
스키마 통계 테
이블
테이블_제약 조건 트리거
뷰 뷰_루틴_사용량 뷰 테
이블_사용량
```

일부 유형의 값은 보기가 아닌 `INFORMATION_SCHEMA` 테이블의 경우에도 데이터 사전에서 조회하여 검색할 수 있습니다. 여기에는 데이터베이스 및 테이블 이름, 테이블 유형, 스토리지 엔진과 같은 값이 포함됩니다.

일부 `INFORMATION_SCHEMA` 테이블에는 테이블 통계를 제공하는 열이 포함되어 있습니다:

```
통계.카디널리티 테이블.자동 증
가 테이블.평균_행_길이 테이블.
체크섬 테이블.체크타임 테이블.
생성 시간 테이블.데이터_자유
테이블.데이터_길이 테이블.인덱
스 길이 테이블.최대_데이터_길
이 테이블_행 테이블.업데이트
시간
```

이러한 열은 동적 테이블 메타데이터, 즉 테이블 콘텐츠가 변경됨에 따라 변경되는 정보를 나타냅니다.

기본적으로 MySQL은 열이 쿼리될 때 `mysql.index_stats` 및 `mysql.innodb_table_stats` 사전 테이블에서 해당 열의 캐시된 값을 검색하므로 스토리지 엔진에서 직접 통계를 검색하는 것보다 더 효율적입니다. 캐시된 통계를 사용할 수 없거나 만료된 경우, MySQL은 스토리지 엔진에서 최신 통계를 검색하여 `mysql.index_stats` 및 `mysql.innodb_table_stats` 딕셔너리 테이블에 캐시합니다. 후속 쿼리는 캐시된 통계가 만료될 때까지 캐시된 통계를 검색합니다. 서버를 다시 시작하거나 `mysql.index_stats` 및 `mysql.innodb_table_stats` 테이블을 처음 열면 캐시된 통계가 자동으로 업데이트되지 않습니다.

`information_schema_stats_expiry` 세션 변수는 캐시된 통계가 만료되기 전 기간을 정의합니다. 기

본값은 86400초(24시간)이지만 최대 1년까지 기간을 연장할 수 있습니다.

특정 테이블에 대해 언제든지 캐시된 값을 업데이트하려면 테이블 분석을 사용합니다.

통계 열을 쿼리해도 mysql.index_stats 및
이러한 상황에서는 mysql.innodb_table_stats 사전 테이블을 사용합니다:

- 캐시된 통계가 만료되지 않은 경우.
- 정보_스키마_통계_만료가0으로 설정된 경우.
- 서버가 읽기 전용, super_read_only, 트랜잭션 읽기 전용 또는 innodb_read_only 모드.
- 쿼리가 성능 스키마 데이터도 가져오는 경우.

information_schema_stats_expiry는 세션 변수이며, 각 클라이언트 세션은 자체 만료 값을 정의할 수 있습니다. 스토리지 엔진에서 검색되어 한 세션에서 캐시된 통계는 다른 세션에서 사용할 수 있습니다.

참고

innodb_read_only 시스템 변수가 활성화된 경우 InnoDB를 사용하는 데이터 사전의 통계 테이블을 업데이트할 수 없기 때문에 ANALYZE TABLE이 실패할 수 있습니다. 키 배포를 업데이트하는 ANALYZE TABLE 작업의 경우 테이블 자체를 업데이트하는 경우에도 실패가 발생할 수 있습니다(
예를 들어, MyISAM 테이블인 경우). 업데이트된 분포 통계를 얻으려면 정보_스키마_통계_만료=0.

데이터 사전 테이블에 대한 뷰로 구현된 INFORMATION_SCHEMA 테이블의 경우, 기본 데이터 사전 테이블의 인덱스를 통해 옵티마이저가 효율적인 쿼리 실행 계획을 구성할 수 있습니다. 옵티마이저의 선택 사항을 보려면 EXPLAIN을 사용합니다. 또한 서버가 INFORMATION_SCHEMA 쿼리를 실행하는 데 사용하는 쿼리를 보려면 EXPLAIN 바로 뒤에 있는 SHOW WARNINGS를 사용합니다.

utf8mb4 문자 집합에 대한 콜레이션을 식별하는 이 문을 고려해 보겠습니다:

```
mysql> SELECT COLLATION_NAME
       FROM INFORMATION_SCHEMA.COLLATION_CHARACTER_SET_APPLICABILITY
       WHERE CHARACTER_SET_NAME = 'utf8mb4';
+----------------------------+
| 콜레이션_이름                |
+----------------------------+
| UTF8MB4_일반_CI             |
| utf8mb4_bin                |
| UTF8MB4_UNICODE_CI         |
| UTF8MB4_ICELANDIC_CI       |
| UTF8MB4_LATVIAN_CI         |
| UTF8MB4_ROMANIAN_CI        |
| UTF8MB4_Slovenian_CI       |
...
```

서버는 해당 문을 어떻게 처리하나요? 알아보려면 EXPLAIN을 사용하세요:

```
mysql> EXPLAIN SELECT COLLATION_NAME
       FROM INFORMATION_SCHEMA.COLLATION_CHARACTER_SET_APPLICABILITY
       WHERE CHARACTER_SET_NAME = 'utf8mb4'\G
*************************** 1. 행 ***************************
           ID: 1
  select_type: SIMPLE 테이
        블: cs
   파티션: NULL
         유형: const 가능한_키:
PRIMARY,name
          키: 이름
      키_len: 194
          참조: const 행:
         1
     필터링됨: 100.00 추가: 인
        덱스 사용
*************************** 2. 행 ***************************
           ID: 1
```

```
  선택 유형: SIMPLE
        테이블: COL
   파티션: NULL
         유형: 참조 가능_키: 문자_
세트_ID
          키: 문자_설정_ID 키_len:
      8
          참조: const 행:
         68
     필터링됨: 100.00 추가:
        NULL
세트 2행, 경고 1회(0.01초)
```

해당 문을 충족하는 데 사용된 쿼리를 보려면 경고 표시를 사용합니다:

```
mysql> SHOW WARNINGS\G
*************************** 1. 행 *************************** 레
  벨: 참고
   코드: 1003
Message: /* select#1 */ select `mysql`.`col`.`name` AS `COLLATION_NAME`
         from `mysql`.`character_sets` `cs`
         조인 `mysql`.`콜레이션` `콜`
         여기서 ((`mysql`.`col`.`character_set_id` = '45')
         및 ('utf8mb4' = 'utf8mb4'))
```

경고 표시에서 알 수 있듯이 서버는 COLLATION_CHARACTER_SET_APPLICABILITY에 대한 쿼리를 mysql 시스템 데이터베이스의 문자 집합 및 콜레이션 데이터 사전 테이블에 대한 쿼리로 처리합니다.

8.2.4 성능 스키마 최적화 쿼리

데이터베이스를 모니터링하는 애플리케이션은 성능 스키마 테이블을 자주 사용할 수 있습니다. 이러한 테이블에 대한 쿼리를 가장 효율적으로 작성하려면 해당 테이블의 인덱스를 활용하세요. 예를 들어 인덱싱된 열의 특정 값과의 비교를 기반으로 검색된 행을 제한하는 WHERE 절을 포함할 수 있습니다.

대부분의 성능 스키마 테이블에는 인덱스가 있습니다. 인덱스가 없는 테이블은 일반적으로 행 수가 적거나 자주 쿼리되지 않을 가능성이 높은 테이블입니다. 성능 스키마 인덱스는 옵티마이저가 전체 테이블 스캔 이외의 실행 계획에 액세스할 수 있도록 합니다. 이러한 인덱스는 해당 테이블을 사용하는 시스템 스키마 뷰와 같은 관련 개체의 성능도 개선합니다.

주어진 성능 스키마 테이블에 인덱스가 있는지 여부와 인덱스가 무엇인지 확인하려면 SHOW INDEX를 사용합니다. 또는 테이블 만들기 표시를 클릭합니다:

```
mysql> SHOW INDEX FROM performance_schema.accounts\G
*************************** 1. 행 *************************** 테
        이블: 계정
   비고유: 0 키_이름: 계정
 Seq_in_index: 1 열
  _이름: USER
    콜레이션: NULL 카
       디널리티: NULL
  Sub_part: NULL 패
         킹됨: NULL
        Null: YES
   인덱스_유형: HASH
      댓글:
인덱스_댓글:
      표시됨: 예
*************************** 2. 행 *************************** 테
        이블: 계정
   비고유: 0 키_이름: 계정
 Seq_in_index: 2 열
  _이름: HOST
    콜레이션: NULL 카
       디널리티: NULL
    Sub_part: NULL
```

```
       Packed: NULL
         Null: YES
   Index_type: 해시
      코멘트:
인덱스_댓글:
      표시됨: 예

mysql> SHOW CREATE TABLE performance_schema.rwlock_instances\G
*************************** 1. 행 *************************** 테

       이블: rwlock_instances
테이블을 생성합니다: CREATE TABLE `rwlock_instances` (
  `NAME` varchar(128) NOT NULL,
  `OBJECT_INSTANCE_BEGIN` bigint(20) 부호 없는 NOT NULL,
  `WRITE_LOCKED_BY_THREAD_ID` bigint(20) 부호 없는 기본값 NULL,
  `READ_LOCKED_BY_COUNT` int(10) 부호 없는 NOT NULL,
  PRIMARY KEY (`OBJECT_INSTANCE_BEGIN`),
  키 `이름` (`이름`),
  키 `쓰기_잠금_바이_스레드_ID` (`쓰기_잠금_바이_스레드_ID`)
) 엔진=성능_화학 기본 문자 집합=utf8mb4 콜레이트=utf8mb4_0900_ai_ci
```

성능 스키마 쿼리의 실행 계획과 인덱스 사용 여부를 확인하려면 다음을 사용하세요.

설명:

```
mysql> EXPLAIN SELECT * FROM performance_schema.accounts
       WHERE (USER,HOST) = ('root','localhost')\G
*************************** 1. 행 ***************************
           ID: 1
  선택 유형: SIMPLE
        테이블: 계정 파티션:
   NULL
         유형: const 가
능한_키: 계정
          키를 입력합니다: 계정
      key_len: 278
          참조: const,const
         행: 1
     필터링됨: 100.00 추가:
        NULL
```

EXPLAIN 출력은 옵티마이저가 USER 및 HOST 열로 구성된 계정 테이블 ACCOUNT 인덱스를 사용함을 나타냅니다.

성능 스키마 인덱스는 가상의 것으로, 성능 스키마 스토리지 엔진의 구성 요소이며 메모리나 디스크 스토리지를 사용하지 않습니다. 성능 스키마는 인덱스 정보를 옵티마이저에 보고하여 옵티마이저가 효율적인 실행 계획을 구성할 수 있도록 합니다. 그러면 성능 스키마는 무엇을 찾아야 하는지에 대한 옵티마이저 정보(예: 특정 키 값)를 사용하므로 실제 인덱스 구조를 구축하지 않고도 효율적인 조회를 수행할 수 있습니다. 이 구현은 두 가지 중요한 이점을 제공합니다:

- 따라서 테이블을 자주 업데이트할 때 일반적으로 발생하는 유지 관리 비용을 완전히 피할 수 있습니다.
- 쿼리 실행의 초기 단계에서 검색되는 데이터의 양을 줄입니다. 인덱싱된 열의 조건에 대해 성능 스키마는 쿼리 조건을 충족하는 테이블 행만 효율적으로 반환합니다. 인덱스가 없는 경우 성능 스키마는 테이블의 모

든 행을 반환하므로 나중에 최적화 프로그램에서 각 행에 대해 조건을 평가하여 최종 결과를 생성해야 합니다.

성능 스키마 인덱스는 미리 정의되어 있으며 삭제, 추가 또는 변경할 수 없습니다. 성능 스키마 인덱스는

해시 인덱스와 유사합니다. 예를 들어

- 연산자는 `=` 또는 `<=>` 연산자를 사용하는 등호 비교에만 사용됩니다.
- 순서가 지정되지 않습니다. 쿼리 결과에 특정 행 순서 지정 특성이 있어야 하는 경우에는 `ORDER BY` 절.

해시 인덱스에 대한 자세한 내용은 섹션 8.3.9, "B-Tree와 해시 인덱스의 비교"를 참조하세요.

8.2.5 데이터 변경 최적화 성명서

이 섹션에서는 데이터 변경 문의 속도를 높이는 방법을 설명합니다: 삽입, 업데이트 및 삭제에 대해 설명합니다. 기존의 OLTP 애플리케이션과 최신 웹 애플리케이션은 일반적으로 동시성이 매우 중요한 소규모 데이터 변경 작업을 많이 수행합니다. 데이터 분석 및 보고 애플리케이션은 일반적으로 한 번에 많은 행에 영향을 미치는 데이터 변경 작업을 실행하며, 여기서 주요 고려 사항은 대량의 데이터를 작성하고 인덱스를 최신 상태로 유지하기 위한 I/O입니다. 대량의 데이터를 삽입하고 업데이트할 때(업계에서는 "추출-변환-로드"의 약자로 ETL이라고 함) `INSERT`, `UPDATE`, `DELETE` 문의 효과를 모방하는 다른 SQL 문이나 외부 명령을 사용하는 경우가 있습니다.

8.2.5.1 INSERT 문 최적화

삽입 속도를 최적화하려면 여러 개의 작은 작업을 하나의 큰 작업으로 결합하세요. 이상적으로는 단일 연결을 만들고, 여러 새 행에 대한 데이터를 한 번에 전송하고, 모든 인덱스 업데이트와 일관성 확인을 마지막까지 지연하는 것이 좋습니다.

행을 삽입하는 데 필요한 시간은 다음 요소에 의해 결정되며, 숫자는 대략적인 비율을 나타냅니다:

- 연결: (3)
- 서버로 쿼리 보내기: (2)
- 구문 분석 쿼리: (2)
- 행 삽입: (행의 1×크기)
- 인덱스 삽입: (1 × 인덱스 수)
- 마감: (1)

여기에는 동시에 실행 중인 각 쿼리에 대해 한 번씩 수행되는 테이블을 여는 초기 오버헤드는 고려되지 않습니다.

테이블의 크기는 B-트리 인덱스를 가정할 때 인덱스 삽입 속도를 로그 *N만큼* 느리게 합니다. 다음 방

법을 사용하여 삽입 속도를 높일 수 있습니다:

- 동일한 클라이언트의 여러 행을 동시에 삽입하는 경우 여러 `VALUES` 목록이 포함된 `INSERT` 문을 사용하여 한 번에 여러 행을 삽입할 수 있습니다. 이렇게 하면 별도의 단일 행 `INSERT` 문을 사용하는 것보다 훨씬 빠릅니다(경우에 따라 몇 배 더 빠릅니다). 비어 있지 않은 테이블에 데이터를 추가하는 경우 `bulk_insert_buffer_size` 변수를 조정하여 데이터 삽입 속도를 더욱 높일 수 있습니다. 섹션 5.1.8, "서버 시스템 변수"를 참조하십시오.
- 텍스트 파일에서 표를 로드할 때는 데이터 로드를 사용합니다. 이 방법은 일반적으로

`INSERT` 문. 섹션 13.2.9, "데이터 로드 문"을 참조하십시오.

- 열에 기본값이 있다는 사실을 활용하세요. 삽입할 값이 기본값과 다른 경우에만 명시적으로 값을 삽입하세요. 이렇게 하면 MySQL이 수행해야 하는 구문 분석이 줄어들고 삽입 속도가 향상됩니다.
- `InnoDB` 테이블과 관련된 팁은 섹션 8.5.5, 'InnoDB 테이블의 대량 데이터 로드'를 참조하십시오.
- `MyISAM` 테이블과 관련된 팁은 섹션 8.6.2, 'MyISAM 테이블의 대량 데이터 로드'를 참조하십시오.

8.2.5.2 업데이트 문 최적화

업데이트 문은 쓰기 오버헤드가 추가되는 `SELECT` 쿼리처럼 최적화되어 있습니다. 쓰기 속도는 업데이트되는 데이터의 양과 업데이트되는 인덱스의 수에 따라 달라집니다. 변경되지 않은 인덱스는 업데이트되지 않습니다.

빠른 업데이트를 얻는 또 다른 방법은 업데이트를 지연한 다음 나중에 여러 업데이트를 연속으로 수행하는 것입니다. 테이블을 잠그면 여러 업데이트를 함께 수행하는 것이 한 번에 하나씩 수행하는 것보다 훨씬 빠릅니다.

동적 행 형식을 사용하는 `MyISAM` 테이블의 경우 행을 더 긴 총 길이로 업데이트하면 행이 분할될 수 있습니다. 이 작업을 자주 수행하는 경우 `OPTIMIZE TABLE`을 가끔 사용하는 것이 매우 중요합니다. 섹션 13.7.3.4, "테이블 최적화 문"을 참조하십시오.

8.2.5.3 삭제 문 최적화

`MyISAM` 테이블에서 개별 행을 삭제하는 데 필요한 시간은 인덱스 수에 정확히 비례합니다. 행을 더 빨리 삭제하려면 `key_buffer_size` 시스템 변수를 늘려 키 캐시 크기를 늘릴 수 있습니다. 5.1.1절. "서버 구성"을 참조하세요.

`MyISAM` 테이블에서 모든 행을 삭제하려면 *`tbl_name`* 테이블을 잘라내는 것이 *`tbl_name`* *테이블을* 삭제하는 `DELETE FROM`보다 빠릅니다. 트랜잭션이 활성 상태이거나 테이블 잠금이 활성 상태일 때 트랜잭션에 안전하지 않으며, 트랜잭션이 활성 상태인 동안 트랜잭션을 시 도 하 면 오류가 발생합니다. 섹션 13.1.37, "테이블 잘라내기 문"을 참조하십시오.

8.2.6 데이터베이스 최적화 권한

권한 설정이 복잡할수록 모든 SQL 문에 더 많은 오버헤드가 적용됩니다. `GRANT` 문으로 설정된 권한을 단순화하면 MySQL은 클라이언트가 문을 실행할 때 권한 확인 오버헤드를 줄일 수 있습니다. 예를 들어 테이블 수준 또는 컬럼 수준 권한을 부여하지 않으면 서버가 `tables_priv` 및 `columns_priv` 테이블의 내용을 확인할 필요가 없습니다. 마찬가지로 계정에 리소스 제한을 두지 않으면 서버가 리소스 카운팅을 수행할 필요가 없습니다. 문 처리 부하가 매우 높은 경우 권한 확인 오버헤드를 줄이기 위해 간소화된 부여 구조를 사용하는 것이 좋습니다.

8.2.7 기타 최적화 팁

이 섹션에는 쿼리 처리 속도를 개선하기 위한 여러 가지 기타 팁이 나열되어 있습니다:

- 애플리케이션에서 관련 업데이트를 수행하기 위해 여러 데이터베이스 요청을 하는 경우 해당 문을 저장된 루틴으로 결합하면 성능에 도움이 될 수 있습니다. 마찬가지로 애플리케이션에서 여러 열 값 또는 대량의 데이터를 기반으로 단일 결과를 계산하는 경우, 계산을 로드 가능한 함수로 결합하면 성능에 도움이 될 수 있습니다. 이렇게 하면 결과적으로 빠른 데이터베이스 작업을 다른 쿼리, 애플리케이션, 심지어 다른 프로그래밍 언어로 작성된 코드에서도 재사용할 수 있습니다. 자세한 내용은 섹션 25.2, '저장된 루틴 사용' 및 MySQL에 함수 추가하기를 참조하세요.

- 아카이브 테이블에서 발생하는 압축 문제를 해결하려면 테이블 최적화를 사용합니다. 섹션 16.5, "아카이브 스토리지 엔진"을 참조하세요.

- 가능하면 보고서를 '실시간' 또는 통계 보고서에 필요한 데이터는 실시간 데이터에서 주기적으로 생성되

는 요약 테이블에서만 생성되는 '통계'로 분류합니다.

- 행과 열로 구성된 테이블 구조에 잘 맞지 않는 데이터가 있는 경우, 데이터를 `BLOB` 열에 패킹하여 저장할 수 있습니다. 이 경우 애플리케이션에 정보를 패킹 및 언패킹하는 코드를 제공해야 하지만 관련 값 집합을 읽고 쓰기 위한 I/O 작업이 절약될 수 있습니다.

- 웹 서버에서는 이미지 및 기타 바이너리 자산을 파일로 저장하고 파일 자체보다는 경로 이름을 데이터베이스에 저장합니다. 대부분의 웹 서버는 데이터베이스 콘텐츠보다 파일 캐싱에 더 능숙하므로 일반적으로 파일을 사용하는 것이 더 빠릅니다. (이 경우 백업 및 저장소 문제를 직접 처리해야 합니다.)

- 정말 빠른 속도가 필요하다면 낮은 수준의 MySQL 인터페이스를 살펴보세요. 예를 들어, MySQL `InnoDB` 또는 `MyISAM` 스토리지 엔진에 직접 액세스하면 SQL 인터페이스를 사용할 때보다 상당한 속도 향상을 얻을 수 있습니다.

  마찬가지로 `NDBCLUSTER` 스토리지 엔진을 사용하는 데이터베이스의 경우, NDB API의 사용 가능성을 조사할 수 있습니다(MySQL NDB 클러스터 API 개발자 가이드 참조).

- 복제는 일부 작업에서 성능 이점을 제공할 수 있습니다. 클라이언트 검색을 복제본 간에 분산하여 부하를 분산할 수 있습니다. 백업하는 동안 소스 속도가 느려지는 것을 방지하려면 복제본을 사용하여 백업을 만들 수 있습니다. 17장, *복제를* 참조하세요.

8.3 최적화 및 색인

`SELECT` 작업의 성능을 개선하는 가장 좋은 방법은 쿼리에서 테스트되는 하나 이상의 열에 인덱스를 생성하는 것입니다. 인덱스 항목은 테이블 행에 대한 포인터처럼 작동하므로 쿼리에서 `WHERE` 절의 조건과 일치하는 행을 빠르게 결정하고 해당 행에 대한 다른 열 값을 검색할 수 있습니다. 모든 MySQL 데이터 유형을 인덱싱할 수 있습니다.

쿼리에 사용되는 모든 가능한 열에 대해 인덱스를 만들고 싶을 수 있지만, 불필요한 인덱스는 공간을 낭비하고 MySQL이 사용할 인덱스를 결정하는 데 시간을 낭비하게 됩니다. 또한 인덱스는 각 인덱스를 업데이트해야 하므로 삽입, 업데이트 및 삭제 비용이 추가됩니다. 최적의 인덱스 세트를 사용하여 빠른 쿼리를 수행하려면 적절한 균형을 찾아야 합니다.

8.3.1 MySQL이 인덱스를 사용하는 방법

인덱스는 특정 열 값을 가진 행을 빠르게 찾는 데 사용됩니다. 인덱스가 없으면 MySQL은 첫 번째 행부터 시작한 다음 전체 테이블을 읽어서 관련 행을 찾아야 합니다. 테이블이 클수록 이 작업은 더 많은 비용이 듭니다. 테이블에 해당 열에 대한 인덱스가 있는 경우 MySQL은 모든 데이터를 볼 필요 없이 데이터 파일 중간에서 찾을 위치를 빠르게 결정할 수 있습니다. 이는 모든 행을 순차적으로 읽는 것보다 훨씬 빠릅니다.

대부분의 MySQL 인덱스(기본 키, 고유, 인덱스 및 전체 텍스트)는 B-트리에 저장됩니다. 예외가 있습니다: 공간 데이터 유형의 인덱스는 R-트리를 사용하고, 메모리 테이블은 해시 인덱스도 지원하며, `InnoDB`는 전체 텍스트 인덱스에 반전된 목록을 사용합니다.

일반적으로 인덱스는 다음 설명에 설명된 대로 사용됩니다. (메모리 테이블에서 사용되는) 해시 인덱스와 관련된 특성은 섹션 8.3.9, "B-Tree와 해시 인덱스 비교"에 설명되어 있습니다.

MySQL은 이러한 작업에 인덱스를 사용합니다:

- `WHERE` 절과 일치하는 행을 빠르게 찾으려면.
- 고려 대상에서 행을 제거합니다. 여러 인덱스 중에서 선택할 수 있는 경우 MySQL은 일반적으로 가장 적은 수의 행을 찾는 인덱스(가장 선택적인 인덱스)를 사용합니다.
- 테이블에 여러 열 인덱스가 있는 경우 인덱스의 가장 왼쪽 접두사는 최적화 프로그램에서 행을 조회하는 데 사용할 수 있습니다. 예를 들어 `(col1, col2, col3)` 에 3개의 열 인덱스가 있는 경우 `(col1)`, (col1, `col2`) 및 `(col1, col2, col3)`에 인덱싱된 검색 기능이 있습니다. 자세한 내용은 섹션 8.3.6, "다중 열 인덱스"를 참조하세요.

- 조인을 수행할 때 다른 테이블에서 행을 검색합니다. MySQL은 열의 인덱스가 동일한 유형과 크기로 선언된 경우 보다 효율적으로 사용할 수 있습니다. 이 맥락에서 VARCHAR와 `CHAR`는 같은 크기로 선언되면 같은 것으로 간주됩니다. 예를 들어, `VARCHAR(10)`과 `CHAR(10)`은 크기가 같지만 `VARCHAR(10)`과 `CHAR(15)`는 그렇지 않습니다.

  이진 문자열이 아닌 열을 비교하려면 두 열 모두 동일한 문자 집합을 사용해야 합니다. 예를 들어, `utf8mb4` 열을 `latin1` 열과비교하면 인덱스를 사용할 수 없습니다.

  서로 다른 열을 비교(예: 문자열 열을 임시 또는 숫자 열과 비교)할 때 값을 변환하지 않고 직접 비교할 수 없는 경우 인덱스를 사용하지 못할 수 있습니다. 숫자 열의 `1`과 같은 주어진 값의 경우 문자열 열의 `'1'`, `'1'`, `'00001'` 또는 `'01.e1'` 등의 값과 동일하게 비교될 수 있습니다. 이렇게 하면 문자열 열에 대한 인덱스 사용이 배제됩니다.

- 특정 인덱싱된 열 *`key_col`*에 대한 `MIN()` 또는 `MAX()` 값을 찾으려면. 이는 모든 키에서 `WHERE` *`key_part_N`* `=` *상수를* 사용하고 있는지 확인하는 전처리기에 의해 최적화됩니다.

인덱스에서 *key_col* 앞에 나오는 부분을 제거합니다. 이 경우 MySQL은 각 MIN() 또는 MAX() 식에 대해 단일 키 조회를 수행한 후 상수로 대체합니다. 모든 표현식이 상수로 대체되면 쿼리가 한 번에 반환됩니다. 예를 들어

```
SELECT MIN(key_part2),MAX(key_part2)
  FROM tbl_name WHERE key_part1=10;
```

- 사용 가능한 인덱스의 가장 왼쪽 접두사에서 정렬 또는 그룹화가 수행되는 경우 테이블을 정렬하거나 그룹화합니다(예: ORDER BY *key_part1, key_part2)*. 모든 키 부분 뒤에 DESC가 오는 경우, 키는 역순으로 읽습니다. (또는 인덱스가 내림차순 인덱스인 경우 키를 정방향 순서로 읽습니다.) 8.2.1.16절, '최적화 기준 주문', 8.2.1.17절, '최적화 기준 그룹화' 및 8.3.13절, '내림차순 색인'을 참조하세요.

- 경우에 따라 데이터 행을 참조하지 않고 값을 검색하도록 쿼리를 최적화할 수 있습니다. (쿼리에 필요한 모든 결과를 제공하는 인덱스를 커버링 인덱스라고 합니다.) 쿼리가 테이블에서 일부 인덱스에 포함된 열만 사용하는 경우, 선택한 값을 인덱스 트리에서 검색하여 속도를 높일 수 있습니다:

```
SELECT key_part3 FROM tbl_name
  WHERE key_part1=1
```

인덱스는 작은 테이블에 대한 쿼리 또는 보고서 쿼리가 대부분의 행 또는 모든 행을 처리하는 큰 테이블에 대한 쿼리에서는 덜 중요합니다. 쿼리에서 대부분의 행에 액세스해야 하는 경우, 인덱스를 통해 작업하는 것보다 순차적으로 읽는 것이 더 빠릅니다. 순차 읽기는 쿼리에 모든 행이 필요하지 않더라도 디스크 검색을 최소화합니다. 자세한 내용은 섹션 8.2.1.23, "전체 테이블 스캔 피하기"를 참조하십시오.

8.3.2 기본 키 최적화

테이블의 기본 키는 가장 중요한 쿼리에 사용하는 열 또는 열 집합을 나타냅니다. 빠른 쿼리 성능을 위해 연관 인덱스가 있습니다. 쿼리 성능은 NULL 값을 포함할 수 없기 때문에 NOT NULL 최적화의 이점을 누릴 수 있습니다. InnoDB 스토리지 엔진을 사용하면 테이블 데이터가 물리적으로 구성되어 기본 키 열을 기반으로 초고속 조회 및 정렬을 수행할 수 있습니다.

테이블이 크고 중요하지만 기본 키로 사용할 명확한 열 또는 열 집합이 없는 경우, 기본 키로 사용할 자동 증가 값을 사용하여 별도의 열을 만들 수 있습니다. 이러한 고유 ID는 외래 키를 사용하여 테이블을 조인할 때 다른 테이블의 해당 행에 대한 포인터 역할을 할 수 있습니다.

8.3.3 공간 인덱스 최적화

MySQL에서는 NOT NULL 지오메트리 값 열에 공간 인덱스를 생성할 수 있습니다(11.4.10절. "공간 인덱스 생성" 참조). 옵티마이저는 인덱싱된 열에 대한 SRID 속성을 확인하여 비교에 사용할 공간 참조 시스템(SRS)을 결정하고 SRS에 적합한 계산을 사용합니다. (MySQL 8.2 이전에는 옵티마이저가 다음 비교를 수행합니다.
데카르트 계산을 사용하는 공간 인덱스 값, 열에 데카르트가 아닌 SRID를 가진 값이 포함된 경우 이러한 작업 결과

는 정의되지 않음).

비교가 제대로 작동하려면 `SPATIAL` 인덱스의 각 열이 SRID가 제한되어야 합니다. 즉, 열 정의에 명시적인 `SRID` 특성이 포함되어야 하며 모든 열 값의 SRID가 동일해야 합니다.

옵티마이저는 SRID가 제한된 열에 대해서만 `SPATIAL` 인덱스를 고려합니다:

- 데카르트 SRID로 제한된 열의 인덱스를 사용하면 데카르트 경계 상자 계산이 가능합니다.
- 지리적 SRID로 제한된 열의 인덱스를 사용하면 지리적 경계 상자 계산이 가능합니다.

옵티마이저는 `SRID` 특성이 없는(따라서 SRID로 제한되지 않는) 열의 `SPATIAL` 인덱스를 무시합니다. MySQL은 다음과 같이 여전히 이러한 인덱스를 유지합니다:

- 테이블 수정(INSERT, UPDATE, DELETE 등)을 위해 업데이트됩니다. 열에 데카르트 및 지리적 값이 혼합되어 있어도 인덱스가 데카르트인 것처럼 업데이트가 수행됩니다.
- 이러한 인덱스는 이전 버전과의 호환성(예: MySQL 8.0에서 덤프를 수행하고 MySQL 8.1에서 복원하는 기능)을 위해서만 존재합니다. SRID가 제한되지 않은 열의 공간 인덱스는 최적화 프로그램에서 사용할 수 없으므로 이러한 각 열을 수정해야 합니다:
  - 열 내의 모든 값의 SRID가 동일한지 확인합니다. 지오메트리 열 *col_name*에 포함된 SRID를 확인하려면 다음 쿼리를 사용합니다:

    ```
    SELECT DISTINCT ST_SRID(col_name) FROM tbl_name;
    ```

    쿼리가 두 개 이상의 행을 반환하는 경우 열에 여러 SRID가 혼합되어 있는 것입니다. 이 경우 모든 값이 동일한 SRID를 갖도록 내용을 수정합니다.
  - 명시적인 SRID 속성을 갖도록 열을 재정의합니다.
  - 공간 인덱스를 다시 생성합니다.

8.3.4 외래 키 최적화

테이블에 많은 열이 있고 다양한 열 조합을 쿼리하는 경우, 사용 빈도가 낮은 데이터를 각각 몇 개의 열이 있는 별도의 테이블로 분할하고 주 테이블에서 숫자 ID 열을 복제하여 주 테이블에 다시 연결하는 것이 효율적일 수 있습니다. 이렇게 하면 각 작은 테이블에 데이터를 빠르게 조회할 수 있는 기본 키를 가질 수 있으며, 조인 작업을 사용하여 필요한 열 집합만 쿼리할 수 있습니다. 데이터가 분산되는 방식에 따라, 관련 열이 디스크에 함께 패킹되므로 쿼리에서 더 적은 I/O를 수행하고 캐시 메모리를 덜 차지할 수 있습니다. (성능을 극대화하기 위해 쿼리는 디스크에서 가능한 한 적은 수의 데이터 블록을 읽으려고 시도하며, 열이 몇 개만 있는 테이블은 각 데이터 블록에 더 많은 행을 넣을 수 있습니다.)

8.3.5 열 색인

가장 일반적인 인덱스 유형은 단일 열을 포함하며, 해당 열의 값 복사본을 데이터 구조에 저장하여 해당 열 값이 있는 행을 빠르게 조회할 수 있도록 합니다. B-트리 데이터 구조를 사용하면 인덱스에서 특정 값, 값 집합 또는 범위의
값에 해당하는 연산자(=, >, ≤, BETWEEN, IN 등)를 WHERE 절에서 사용할 수 있습니다.

테이블당 최대 인덱스 수와 최대 인덱스 길이는 스토리지 엔진별로 정의됩니다. 15장, *InnoDB 스토리지 엔진* 및 16장, *대체 스토리지 엔진을* 참조하십시오. 모든 스토리지 엔진은 테이블당 최소 16개의 인덱스와 최소 256바이트의 총 인덱스 길이를 지원합니다. 대부분의 스토리지 엔진은 더 높은 제한을 가지고 있습니다.

열 인덱스에 대한 자세한 내용은 섹션 13.1.15, "인덱스 생성 문"을 참조하십시오.

- 색인 접두사

- 전체 텍스트 색인
- 공간 인덱스
- 메모리 스토리지 엔진의 인덱스

색인 접두사

문자열 열에 대한 인덱스 사양에 *col_name(N)* 구문을 사용하면 열의 처음 *N* 문자만 사용하는 인덱스를 만들 수 있습니다. 이러한 방식으로 열 값의 접두사만 인덱싱하면 인덱스 파일을 훨씬 더 작게 만들 수 있습니다. BLOB 또는 TEXT 열을 인덱싱하는 경우 인덱스의 접두사 길이를 지정해야 합니다. 예를 들어

```
CREATE TABLE test (blob_col BLOB, INDEX(blob_col(10));
```

접두사는 중복 또는 COMPACT 행 형식을 사용하는 InnoDB 테이블의 경우 최대 767바이트까지 사용할 수 있습니다. 접두사 길이 제한은 DYNAMIC 또는 COMPRESSED 행 형식을 사용하는 InnoDB 테이블의 경우 3072바이트입니다. MyISAM 테이블의 경우 접두사 길이 제한은 1000바이트입니다.

참고

접두사 제한은 바이트 단위로 측정되는 반면, CREATE TABLE, ALTER TABLE 및 CREATE INDEX 문의 접두사 길이는 비바이너리 문자열 유형(CHAR, VARCHAR, TEXT)의 경우 문자 수로, 바이너리 문자열 유형(BINARY, VARBINARY, BLOB)의 경우 바이트 수로 해석됩니다. 다중 바이트 문자 집합을 사용하는 비 이진 문자열 열의 접두사 길이를 지정할 때는 이 점을 고려해야 합니다.

검색어가 인덱스 접두사 길이를 초과하는 경우 인덱스를 사용하여 일치하지 않는 행을 제외하고 나머지 행에 대해 일치 가능성이 있는지 검사합니다.

인덱스 접두사에 대한 자세한 내용은 섹션 13.1.15, "인덱스 생성 문"을 참조하세요.

전체 텍스트 색인

전체 텍스트 인덱스는 전체 텍스트 검색에 사용됩니다. InnoDB 및 MyISAM 저장소 엔진만 전체 텍스트 인덱스를 지원하며 CHAR, VARCHAR 및 TEXT 열에 대해서만 지원합니다. 인덱싱은 항상 전체 열에 대해 수행되며 열 접두사 인덱싱은 지원되지 않습니다. 자세한 내용은 섹션 12.9, "전체 텍스트 검색 함수"를 참조하세요.

최적화는 단일 InnoDB 테이블에 대한 특정 종류의 전체 텍스트 쿼리에 적용됩니다. 이러한 특성을 가진 쿼리는 특히 효율적입니다:

- 문서 ID만 반환하는 전체 텍스트 쿼리 또는 문서 ID와 검색 순위만 반환하는 쿼리입니다.
- 일치하는 행을 점수 내림차순으로 정렬하고 상위 N개의 일치하는 행을 가져오는 LIMIT 절을 적용하는 전체 텍스트 쿼리입니다. 이 최적화가 적용되려면 내림차순으로 WHERE 절이 없어야 하고 ORDER BY 절이 하나만 있어야 합니다.
- 검색어와 일치하는 행의 COUNT(*) 값만 검색하는 전체 텍스트 쿼리로, 추가 WHERE 절이 없습니다. WHERE 절을 > 0 비교 연산자 없이 WHERE MATCH(*text*) AGAINST('*other_text*')로 코딩합니다.

전체 텍스트 표현식이 포함된 쿼리의 경우 MySQL은 쿼리 실행의 최적화 단계에서 해당 표현식을 평가합니다. 옵티마이저는 단순히 전체 텍스트 표현식을 보고 추정하는 것이 아니라 실행 계획을 개발하는 과정에서 실제로 평가합니다.

이 동작의 의미는 전체 텍스트 쿼리에 대한 설명이 일반적으로 최적화 단계에서 표현식 평가가 발생하지 않는 전체 텍스트가 아닌 쿼리에 비해 느리다는 것입니다.

전체 텍스트 쿼리에 대한 설명은 최적화 중에 발생하는 일치로 인해 추가 열에 최적화된 테이블 선택이 표

시될 수 있으며, 이 경우 나중에 실행하는 동안 테이블 액세스가 필요하지 않습니다.

공간 인덱스

공간 데이터 유형에 인덱스를 만들 수 있습니다. `MyISAM` 및 `InnoDB`는 공간 유형에 대한 R-트리 인덱스를 지원합니다. 다른 스토리지 엔진은 공간 유형 인덱싱에 B-트리를 사용합니다(공간 유형 인덱싱을 지원하지 않는 `ARCHIVE`는 제외).

메모리 스토리지 엔진의 인덱스

메모리 저장소 엔진은 기본적으로 `HASH` 인덱스를 사용하지만 `BTREE` 인덱스도 지원합니다.

8.3.6 다중 열 인덱스

MySQL은 복합 인덱스(즉, 여러 열에 대한 인덱스)를 만들 수 있습니다. 인덱스는 최대 16개의 열로 구성될 수 있습니다. 특정 데이터 유형의 경우 열의 접두사를 인덱싱할 수 있습니다(섹션 8.3.5, "열 인덱스" 참조).

MySQL은 인덱스의 모든 열을 테스트하는 쿼리 또는 첫 번째 열, 처음 두 열, 처음 세 열만 테스트하는 쿼리 등에 다중 열 인덱스를 사용할 수 있습니다. 인덱스 정의에서 올바른 순서로 열을 지정하면 단일 복합 인덱스로 동일한 테이블에 대한 여러 종류의 쿼리 속도를 높일 수 있습니다.

여러 열로 구성된 인덱스는 정렬된 배열로 간주할 수 있으며, 이 배열의 행에는 인덱스된 열의 값을 연결하여 생성된 값이 포함됩니다.

참고

복합 인덱스의 대안으로 다른 열의 정보를 기반으로 "해시"된 열을 도입할 수 있습니다. 이 열이 짧고, 상당히 고유하며, 인덱싱된 경우, 많은 열에 대한 "넓은" 인덱스보다 빠를 수 있습니다. MySQL에서는 이 추가 열을 매우 쉽게 사용할 수 있습니다:

```
SELECT * FROM tbl_name
  WHERE hash_col=MD5(CONCAT(val1,val2))
  AND col1=val1 AND col2=val2;
```

테이블에 다음과 같은 사양이 있다고 가정합니다:

```
CREATE TABLE test (
    id INT NOT NULL,
    last_name CHAR(30) NOT NULL,
    first_name CHAR(30) NOT NULL,
    PRIMARY KEY (id),
    인덱스 이름(성, 이름)
);
```

이름 인덱스는 last_name 및 first_name 열에 대한 인덱스입니다. 이 인덱스는 마지막 이름과 이름 값의 조합에 대해 알려진 범위의 값을 지정하는 쿼리에서 조회에 사용할 수 있습니다. 이 열은 인덱스의 가장 왼쪽 접두사이기 때문에 last_name 값만 지정하는 쿼리에도 사용할 수 있습니다(이 섹션의 뒷부분에 설명됨). 따라서 이름 인덱스는 다음 쿼리에서 조회에 사용됩니다:

```
SELECT * FROM test WHERE last_name='Jones';

SELECT * FROM test
  WHERE last_name='Jones' AND first_name='John';

SELECT * FROM test
  WHERE last_name='Jones'
  AND (first_name='John' OR first_name='Jon');

SELECT * FROM test
  WHERE last_name='Jones'
  AND first_name >='M' AND first_name < 'N';
```

그러나 다음 쿼리에서는 이름 인덱스가 조회에 사용되지 *않습니다:*

```
SELECT * FROM test WHERE first_name='John';

SELECT * FROM test
  WHERE last_name='Jones' OR first_name='John';
```

다음과 같은 SELECT 문을 실행한다고 가정합니다:

```
SELECT * FROM tbl_name
  WHERE col1=val1 AND col2=val2;
```

col1과 col2에 다중 열 인덱스가 있는 경우 적절한 행을 직접 가져올 수 있습니다. col1과 col2에 별도의 단일 열 인덱스가 존재하는 경우, 옵티마이저는 인덱스를 사용하려고 시도합니다.

병합 최적화(섹션 8.2.1.3, "인덱스 병합 최적화" 참조) 또는 더 많은 행을 제외하는 인덱스를 결정하고 해당 인덱스를 사용하여 행을 가져오는 데 가장 제한적인 인덱스를 찾으려고 시도합니다.

테이블에 여러 열 인덱스가 있는 경우 인덱스의 가장 왼쪽 접두사는 최적화 프로그램에서 행을 조회하는 데 사용할 수 있습니다. 예를 들어 `(col1, col2, col3)`에 세 개의 열 인덱스가 있는 경우 `(col1)`, (col1, `col2)`, `(col1, col2, col3)`에 인덱스 검색 기능이 있습니다.

열이 인덱스의 가장 왼쪽 접두사를 형성하지 않는 경우 MySQL은 인덱스를 사용하여 조회를 수행할 수 없습니다. 여기에 표시된 `SELECT` 문이 있다고 가정해 보겠습니다:

```
SELECT * FROM tbl_name WHERE col1=val1;
SELECT * FROM tbl_name WHERE col1=val1 AND col2=val2;

SELECT * FROM tbl_name WHERE col2=val2;
SELECT * FROM tbl_name WHERE col2=val2 AND col3=val3;
```

`(col1, col2, col3)`에 인덱스가 있는 경우 처음 두 쿼리만 인덱스를 사용합니다. 세 번째 및 네 번째 쿼리는 인덱싱된 열을 포함하지만 `(col2)` 및 `(col2, col3)`이 `(col1, col2, col3)`의 가장 왼쪽 접두사가 아니므로 조회를 수행하기 위해 인덱스를 사용하지 않습니다.

8.3.7 색인 확인 사용량

모든 쿼리가 테이블에서 생성한 인덱스를 실제로 사용하는지 항상 확인합니다. 섹션 8.8.1, "EXPLAIN을 사용하여 쿼리 최적화"에 설명된 대로 `EXPLAIN` 문을 사용합니다.

8.3.8 InnoDB 및 MyISAM 인덱스 통계 수집

스토리지 엔진은 옵티마이저에서 사용할 테이블에 대한 통계를 수집합니다. 테이블 통계는 값 그룹을 기반으로 하며, 여기서 값 그룹은 동일한 키 접두사 값을 가진 행의 집합입니다. 옵티마이저의 목적상 중요한 통계는 평균 값 그룹 크기입니다.

MySQL은 다음과 같은 방식으로 평균값 그룹 크기를 사용합니다:

- 각 참조 액세스에 대해 읽어야 하는 행 수를 예측하려면 다음과 같이 하세요.
- 부분 조인이 생성하는 행 수, 즉 다음과 같은 형식의 연산으로 생성되는 행 수를 예측하려면 다음을 수행합니다.

  ```
  (...) JOIN tbl_name ON tbl_name.key = expr
  ```

인덱스의 평균 값 그룹 크기가 커지면 조회당 평균 행 수가 증가하기 때문에 인덱스는 이 두 가지 목적에 덜 유용합니다: 인덱스가 최적화 목적에 적합하려면 각 인덱스 값이 테이블의 적은 수의 행을 대상으로 하는 것이 가장 좋습니다. 주어진 인덱스 값이 많은 수의 행을 생성하는 경우 인덱스의 유용성이 떨어지고 MySQL이 인덱스를 사용할 가능성이 낮아집니다.

평균 값 그룹 크기는 값 그룹의 수인 테이블 카디널리티와 관련이 있습니다. 테이블의

SHOW INDEX 문은 N/S를 기준으로 카디널리티 값을 표시합니다. 여기서 N은 테이블의 행 수이고 S는 평균 값 그룹 크기입니다. 이 비율은 테이블에 있는 대략적인 값 그룹의 수를 산출합니다.

<=> 비교 연산자를 기반으로 하는 조인의 경우 NULL은 다른 값과 다르게 취급되지 않습니다:
다른 N의 경우 N <=> N과 마찬가지로 NULL <=> NULL입니다.

그러나 = 연산자를 기반으로 하는 조인의 경우 NULL은 NULL이 아닌 값과 다릅니다. 즉, expr1 또는 expr2(또는 둘 다)가 NULL인 경우 expr1 = expr2는 참이 아닙니다. 이는 tbl_name.key = expr 형식의 비교를 위한 참조 액세스에 영향을 줍니다: 비교가 참일 수 없기 때문에 MySQL은 expr의 현재 값이 NULL인 경우 테이블에 액세스하지 않습니다.

비교의 경우 테이블에 NULL 값이 몇 개 있는지는 중요하지 않습니다. 최적화를 위해 관련 값은 NULL이 아닌 값 그룹의 평균 크기입니다. 그러나 MySQL은 현재 이 평균 크기를 수집하거나 사용할 수 없습니다.

InnoDB 및 MyISAM 테이블의 경우 각각 innodb_stats_method 및 myisam_stats_method 시스템 변수를 사용하여 테이블 통계 수집을 일부 제어할 수 있습니다. 이 변수에는 다음과 같이 세 가지 가능한 값이 있습니다:

- 변수가 nulls_equal로 설정되면 모든 NULL 값이 동일한 것으로 취급됩니다(즉, 모두 단일 값 그룹을 형성합니다).

  NULL 값 그룹 크기가 평균 NULL이 아닌 값 그룹 크기보다 훨씬 큰 경우, 이 방법은 평균 값 그룹 크기를 상향 조정합니다. 이로 인해 인덱스가 최적화 프로그램에 실제보다 덜 유용한 것처럼 보이게 되어 NULL이 아닌 값을 찾는 조인에서 유용성이 떨어집니다. 결과적으로, nulls_equal 메서드는 옵티마이저가 인덱스를 참조 액세스에 사용해야 할 때 사용하지 않도록 만들 수 있습니다.

- 변수가 nulls_unequal로 설정되면 NULL 값은 동일하게 간주되지 않습니다. 대신 각 NULL 값은 크기 1의 별도 값 그룹을 형성합니다.

  NULL 값이 많은 경우 이 방법을 사용하면 평균 값 그룹 크기가 아래로 치우치게 됩니다. NULL이 아닌 값의 평균 그룹 크기가 큰 경우 NULL 값을 각각 크기 1의 그룹으로 계산합니다.
  를 사용하면 옵티마이저가 NULL이 아닌 값을 찾는 조인에 대해 인덱스 값을 과대평가하게 됩니다. 결과적으로, 다른 방법이 더 나을 수 있는데도 최적화 프로그램이 이 인덱스를 참조 조회에 사용하도록 만들 수 있습니다.

- 변수가 nulls_ignored로 설정되면 NULL 값은 무시됩니다.

대신 <=>을 사용하는 조인을 많이 사용하는 경우 비교에서 NULL 값은 특별하지 않으며 하나의 NULL은 다른 NULL과 동일합니다. 이 경우 nulls_equal이 적절한 통계 메서드입니다.

innodb_stats_method 시스템 변수에는 전역 값이 있고, myisam_stats_method 시스템 변수에는 전역 값과 세션 값이 모두 있습니다. 전역 값을 설정하면 해당 스토리지 엔진의 테이블에 대한 통계 수집에 영향을 줍니다. 세션 값을 설정하면 현재 클라이언트 연결에 대한 통계 수집에만 영향을 줍니다. 즉, myisam_stats_method의 세션 값을 설정하여 다른 클라이언트에 영향을 주지 않고 특정 메서드로 테이블의 통계를 강제로 다시 생성할 수 있습니다.

MyISAM 테이블 통계를 다시 생성하려면 다음 방법 중 하나를 사용할 수 있습니다:

- myisamchk *--stats_method=method_name* --analyze 실행
- 테이블을 변경하여 통계를 최신 상태로 유지한 다음(예: 행을 삽입한 다음 삭제), myisam_stats_method를 설정하고 ANALYZE TABLE 문을 실행합니다.

innodb_stats_method 및 myisam_stats_method 사용과 관련된 몇 가지 주의 사항:

- 방금 설명한 대로 테이블 통계를 명시적으로 수집하도록 강제할 수 있습니다. 그러나 MySQL이 자동으로 통계를 수집할 수도 있습니다. 예를 들어, 테이블에 대한 문을 실행하는 과정에서 일부 문이 테이블을 수정하는

경우 MySQL이 통계를 수집할 수 있습니다. (예를 들어 대량 삽입 또는 삭제 또는 일부 `ALTER TABLE` 문에서 발생할 수 있습니다.) 이 경우 통계는 당시 `innodb_stats_method` 또는 `myisam_stats_method`에 있는 값을 사용하여 수집됩니다. 따라서 한 가지 방법을 사용하여 통계를 수집하지만 나중에 테이블의 통계가 자동으로 수집될 때 시스템 변수가 다른 방법으로 설정되어 있으면 다른 방법이 사용됩니다.

- 특정 테이블에 대한 통계를 생성하는 데 어떤 방법이 사용되었는지 알 수 있는 방법이 없습니다.
- 이러한 변수는 `InnoDB` 및 `MyISAM` 테이블에만 적용됩니다. 다른 스토리지 엔진에는 테이블 통계를 수집하는 메서드가 하나만 있습니다. 일반적으로 `nulls_equal` 메서드에 더 가깝습니다.

8.3.9 B-Tree와 해시 인덱스 비교

B-트리 및 해시 데이터 구조를 이해하면 인덱스에 이러한 데이터 구조를 사용하는 여러 스토리지 엔진에서 다양한 쿼리가 어떻게 수행되는지 예측하는 데 도움이 될 수 있으며, 특히 B-트리 또는 해시 인덱스를 선택할 수 있는 `MEMORY` 스토리지 엔진의 경우 더욱 그렇습니다.

- B-Tree 인덱스 특성
- 해시 인덱스 특성

B-Tree 인덱스 특성

B-트리 인덱스는 =, >, >=, <, <= 또는 BETWEEN 연산자를 사용하는 표현식에서 열 비교에 사용할 수 있습니다. LIKE 인수가 와일드카드 문자로 시작하지 않는 상수 문자열인 경우 이 인덱스는 LIKE 비교에도 사용할 수 있습니다. 예를 들어 다음 SELECT 문은 인덱스를 사용합니다:

```
SELECT * FROM tbl_name WHERE key_col LIKE 'Patrick%';
SELECT * FROM tbl_name WHERE key_col LIKE 'Pat%_ck%';
```

첫 번째 문에서는 'Patrick' <= key_col < 'Patricl'인 행만 고려됩니다. 두 번째 문에서는 'Pat' <= key_col < 'Pau'인 행만 고려됩니다.

다음 SELECT 문은 인덱스를 사용하지 않습니다:

```
SELECT * FROM tbl_name WHERE key_col LIKE '%Patrick%';
SELECT * FROM tbl_name WHERE key_col LIKE other_col;
```

첫 번째 문에서 LIKE 값은 와일드카드 문자로 시작합니다. 두 번째 문에서는 LIKE 값은 상수가 아닙니다.

사용하는 경우 ... '%string%'과 같이 *문자열*이세 글자보다 길면 MySQL은 *터보 보이어-무어* 알고리즘을 사용하여 문자열의 패턴을 초기화한 다음 이 패턴을 사용하여 검색을 더 빠르게 수행합니다.

col_name IS NULL을 사용한 검색은 col_name이 인덱싱된 경우 인덱스를 사용합니다.

WHERE 절의 모든 AND 수준에 걸쳐 있지 않은 인덱스는 쿼리를 최적화하는 데 사용되지 않습니다. 즉, 인덱스를 사용하려면 모든 AND 그룹에서 인덱스의 접두사를 사용해야 합니다.

다음 WHERE 절은 인덱스를 사용합니다:

```
... WHERE index_part1=1 AND index_part2=2 AND other_column=3

    /* index = 1 또는 index = 2 */
... WHERE index=1 OR A=10 AND index=2

         /* "index_part1='hello'"처럼 최적화됨 */
... WHERE index_part1='hello' AND index_part3=5

    /* 인덱스1에는 인덱스를 사용할 수 있지만 인덱스2나 인덱스3에는 사용할 수 없습니다 */.
... WHERE index1=1 AND index2=2 OR index1=3 AND index3=3;
```

이러한 WHERE 절은 인덱스를 사용하지 *않습니다*:

```
    /* index_part1은 사용되지 않음 */
... WHERE index_part2=1 AND index_part3=2

    /* WHERE 절의 두 부분 모두에 인덱스가 사용되지 않습니다 */.
... WHERE index=1 또는 A=10

    /* 모든 행에 걸쳐 인덱스가 없습니다 */.
... WHERE index_part1=1 또는 index_part2=10
```

인덱스를 사용할 수 있는 경우에도 MySQL이 인덱스를 사용하지 않는 경우가 있습니다. 이러한 상황이 발생하는 한 가지 상황은 최적화 프로그램이 인덱스를 사용하면 MySQL이 테이블의 행 중 매우 많은 비율에 액세스해야 한다고 추정하는 경우입니다. (이 경우 테이블 스캔은 검색 횟수가 적기 때문에 훨씬 빠를 가능성이 높습니다.) 그러나 이러한 쿼리가 `LIMIT`를 사용하여 일부 행만 검색하는 경우에는 결과에서 반환할 몇 개의 행을 훨씬 더 빨리 찾을 수 있으므로 MySQL은 어쨌든 인덱스를 사용합니다.

해시 인덱스 특성

해시 인덱스는 방금 설명한 것과는 다소 다른 특성을 가지고 있습니다:

- 연산자는 `=` 또는 `<=>` 연산자를 사용하는 등호 비교에만 사용됩니다(하지만 *매우* 빠릅니다). 값의 범위를 찾는 `<`와 같은 비교 연산자에는 사용되지 않습니다. 이러한 유형의 단일 값 조회에 의존하는 시스템을 "키-값 저장소"라고 하며, 이러한 애플리케이션에 MySQL을 사용하려면 가능한 경우 해시 인덱스를 사용하세요.
- 옵티마이저는 해시 인덱스를 사용하여 `ORDER BY` 연산 속도를 높일 수 없습니다. (이 유형의 인덱스는 다음 항목을 순서대로 검색하는 데 사용할 수 없습니다.)
- MySQL은 두 값 사이에 대략 몇 개의 행이 있는지 확인할 수 없습니다(이 값은 범위 최적화 프로그램에서 사용할 인덱스를 결정하는 데 사용됨). `MyISAM` 또는 `InnoDB` 테이블을 해시 인덱싱된 메모리 테이블로 변경하는 경우 일부 쿼리에 영향을 미칠 수 있습니다.
- 전체 키만 행을 검색하는 데 사용할 수 있습니다. (B-트리 인덱스의 경우 키의 가장 왼쪽 접두사를 사용하여 행을 찾을 수 있습니다.)

8.3.10 색인 사용 확장

`InnoDB`는 각 보조 인덱스에 기본 키 열을 추가하여 각 보조 인덱스를 자동으로 확장합니다. 이 테이블 정의를 살펴보겠습니다:

```
CREATE TABLE t1 (
  i1 INT NOT NULL DEFAULT 0,
  i2 INT NOT NULL DEFAULT 0,
  d DATE DEFAULT NULL,
  PRIMARY KEY (i1, i2),
  INDEX k_d (d)
 ) 엔진 = InnoDB;
```

이 테이블은 열 `(i1, i2)`의 기본 키를 정의합니다. 또한 열 `(d)`에 보조 인덱스 `k_d`를 정의하지만 내부적으로 `InnoDB`는 이 인덱스를 확장하여 열 `(d, i1, i2)`로 취급합니다.

옵티마이저는 확장된 보조 인덱스의 기본 키 열을 고려하여 해당 인덱스의 사용 방법과 사용 여부를 결정합니다. 이를 통해 보다 효율적인 쿼리 실행 계획과 더 나은 성능을 얻을 수 있습니다.

옵티마이저는 참조, 범위 및 `index_merge` 인덱스 액세스, 느슨한 인덱스 스캔 액세스, 조인 및 정렬 최적화, `MIN()`/`MAX()` 최적화를 위해 확장된 보조 인덱스를 사용할 수 있습니다.

다음 예는 옵티마이저가 확장 보조 인덱스를 사용하는지 여부에 따라 실행 계획이 어떻게 영향을 받는지 보여줍니다. ㅠ이 이러한 행으로 채워져 있다고 가정합니다:

```
삽입     INTO t1 VALUES
        '1998-01-01'), (1, 2, '1999-01-01'),
(1, 1,
(1, 3, '2000-01-01'), (1, 4, '2001-01-01'),
(1, 5, '2002-01-01'), (2, 1, '1998-01-01'),
(2, 2, '1999-01-01'), (2, 3, '2000-01-01'),
(2, 4, '2001-01-01'), (2, 5, '2002-01-01'),
(3, 1, '1998-01-01'), (3, 2, '1999-01-01'),
```

```
(3, 3, '2000-01-01'), (3, 4, '2001-01-01'),
(3, 5, '2002-01-01'), (4, 1, '1998-01-01'),
(4, 2, '1999-01-01'), (4, 3, '2000-01-01'),
(4, 4, '2001-01-01'), (4, 5, '2002-01-01'),
(5, 1, '1998-01-01'), (5, 2, '1999-01-01'),
(5, 3, '2000-01-01'), (5, 4, '2001-01-01'),
(5, 5, '2002-01-01');
```

이제 이 쿼리를 생각해 보세요:

```
EXPLAIN SELECT COUNT(*) FROM t1 WHERE i1 = 3 AND d = '2000-01-01'
```

실행 계획은 확장 인덱스 사용 여부에 따라 달라집니다.

옵티마이저가 인덱스 확장을 고려하지 않는 경우 인덱스 k_d를 (d)로만 처리합니다. 설명를 쿼리에 입력하면 이 결과가 생성됩니다:

```
mysql> EXPLAIN SELECT COUNT(*) FROM t1 WHERE i1 = 3 AND d = '2000-01-01'\G
*************************** 1. 행 ***************************
           ID: 1
  select_type: SIMPLE 테이
        블: t1
         유형: 참조 가능한_키:
PRIMARY,k_d
          키:
      K_D_KEY_LEN:
      4
          참조: const 행:
         5
        추가: 어디 사용; 인덱스 사용
```

옵티마이저가 인덱스 확장을 고려할 때, k_d는 (d, i1, i2)로 처리됩니다. 이 경우, 가장 왼쪽에 있는 인덱스 접두사(d, i1)를사용하여 더 나은 실행 계획을 생성할 수 있습니다:

```
mysql> EXPLAIN SELECT COUNT(*) FROM t1 WHERE i1 = 3 AND d = '2000-01-01'\G
*************************** 1. 행 ***************************
           ID: 1
  select_type: SIMPLE 테이
        블: t1
         유형: 참조 가능한_키:
PRIMARY,k_d
          키:
      K_D_KEY_LEN:
      8
          참조: const,const
         행: 1
        추가: 인덱스 사용
```

두 경우 모두 key는 옵티마이저가 보조 인덱스 k_d를 사용함을 나타내지만 EXPLAIN 출력은 확장 인덱스 사용으로 인한 이러한 개선 사항을 보여줍니다:

- key_len은 4바이트에서 8바이트로 증가하며, 이는 키 조회가 d뿐만 아니라 열 d와 i1도 사용함을 나타냅니다.
- 키 조회가 하나가 아닌 두 개의 키 부분을 사용하기 때문에 참조 값이 const에서 const,const로 변경됩니다.
- 행 수가 5개에서 1개로 감소하여 InnoDB가 결과를 생성하기 위해 더 적은 수의 행을 검사해야 함을 나타냅니다.
- 추가 값이 사용 위치; 인덱스 사용에서 인덱스 사용으로 변경됩니다. 즉, 데이터 행의 열을 참조하지 않고 인덱스만 사용하여 행을 읽을 수 있습니다.

확장 인덱스 사용에 대한 옵티마이저 동작의 차이도 SHOW STATUS를 통해 확인할 수 있습니다:

```
플러시 테이블 t1;
플러시 상태;
SELECT COUNT(*) FROM t1 WHERE i1 = 3 AND d = '2000-01-01';
'handler_read%' 같은 상태 표시
```

앞의 명령문에는 테이블 캐시를 플러시하고 상태 카운터를 지우는 `FLUSH TABLES` 및 `FLUSH` STATUS가 포함되어 있습니다.

인덱스 확장을 사용하지 않으면 `SHOW` STATUS가 이 결과를 생성합니다:

```
+-----------------------+ ---------+
|                        변수_이름| 값 |
+-----------------------+ ---------+
| 핸들러_읽기_우선          | 0      |
| 핸들러_읽기_키            | 1      |
| 핸들러_읽기_마지막         | 0      |
| 핸들러_읽기_다음          | 5      |
| 핸들러_읽기_이전          | 0      |
| 핸들러_read_rnd          | 0      |
| 핸들러_read_rnd_next     | 0      |
```

```
+----------------------+ ---------+
```

인덱스 확장을 사용하면 SHOW STATUS가 이 결과를 생성합니다. Handler_read_next 값이 5에서 1로 감소하여 인덱스가 더 효율적으로 사용됨을 나타냅니다:

```
+----------------------+ ---------+
|                        변수_이름|
값 |
+----------------------+ ---------+
| 핸들러_읽기_우선           | 0      |
| 핸들러_읽기_키             | 1      |
| 핸들러_읽기_마지막         | 0      |
| 핸들러_읽기_다음           | 1      |
| 핸들러_읽기_이전           | 0      |
| 핸들러_read_rnd          | 0      |
| 핸들러_read_rnd_next | 0        |
+----------------------+ ---------+
```

optimizer_switch 시스템 변수의 use_index_extensions 플래그를 사용하면 InnoDB 테이블의 보조 인덱스 사용 방법을 결정할 때 옵티마이저가 기본 키 열을 고려할지 여부를 제어할 수 있습니다. 기본적으로 use_index_extensions는 활성화되어 있습니다. 인덱스 확장 사용을 비활성화하면 성능이 향상되는지 확인하려면 이 문을 사용하세요:

```
SET optimizer_switch = '사용_인덱스_확장=오프';
```

옵티마이저에 의한 인덱스 확장 사용에는 인덱스의 키 부분 수(16개)와 최대 키 길이(3072바이트)에 대한 일반적인 제한이 적용됩니다.

8.3.11 생성된 열의 옵티마이저 사용 인덱스

MySQL은 생성된 열에 대한 인덱스를 지원합니다. 예를 들어

```
CREATE TABLE t1 (f1 INT, gc INT AS (f1 + 1) STORED, INDEX (gc));
```

생성된 열인 gc는 f1 + 1 표현식으로 정의됩니다. 또한 이 열은 인덱싱되며 옵티마이저는 실행 계획을 구성하는 동안 해당 인덱스를 고려할 수 있습니다. 다음 쿼리에서 WHERE 절은 gc를 참조하며옵티마이저는 해당 열의 인덱스가 더 효율적인 계획을 생성하는지 여부를 고려합니다:

```
SELECT * FROM t1 WHERE gc > 9;
```

옵티마이저는 쿼리에서 이름으로 해당 열을 직접 참조하지 않더 라 도 생성된 열의 인덱스를 사용하여 실행 계획을 생성할 수 있습니다. 이는 WHERE, ORDER BY 또는 GROUP BY 절이 인덱싱된 일부 생성된 열의 정의와 일치하는 표현식을 참조하는 경우에 발생합니다. 다음 쿼리는 gc를 직접 참조하지는 않지만 gc의 정의와 일치하는 표현식을 사용합니다:

```
SELECT * FROM t1 WHERE f1 + 1 > 9;
```

옵티마이저는 표현식 F1 + 1이 GC의 정의와 일치하고 GC가 인덱싱되어 있다는 것을 인식하므로 실행 계획 구성 중에 해당 인덱스를 고려합니다. EXPLAIN을 사용하면 이를 확인할 수 있습니다:

```
mysql> EXPLAIN SELECT * FROM t1 WHERE f1 + 1 > 9\G
*************************** 1. 행 ***************************
           ID: 1
  select_type: SIMPLE 테이
        블: t1
   파티션: NULL
         유형: 범위 가능_키
: GC
          키: GC
      KEY_LEN: 5
          참조:
         NULL 행: 1
     필터링됨: 100.00
```

추가: 인덱스 조건 사용

실제로 최적화 프로그램은 표현식 `F1 + 1`을 해당 표현식과 일치하는 생성된 열의 이름으로 대체했습니다. 이는 경고 표시로 표시되는 확장된 설명 정보에서 사용할 수 있는 재작성된 쿼리에서도 알 수 있습니다:

```
mysql> SHOW WARNINGS\G
*************************** 1. 행 ***************************
  벨: 참고
   코드: 1003
Message: /* select#1 */ select `test`.`t1`.`f1` AS `f1`,`test`.`t1`.`gc`
         AS `gc` from `test`.`t1` where (`test`.`t1`.`gc` > 9)
```

옵티마이저가 생성된 열 인덱스를 사용하는 데는 다음과 같은 제한 및 조건이 적용됩니다:

- 쿼리 표현식이 생성된 열 정의와 일치하려면 표현식이 동일해야 하고 결과 유형이 동일해야 합니다. 예를 들어 , 생성된 열 표현식이 `f1 + 1`인 경우 쿼리에서 `1 + f1`을 사용하거나 `f1 + 1`(정수 표현식)을 문자열과 비교하면 최적화 도구에서 일치하는 것으로 인식하지 않습니다.

- 최적화는 다음 연산자에 적용됩니다: `=`, `<`, `<=`, `>`, `>=`, `BETWEEN`, `IN()`.

  `BETWEEN` 및 `IN()` 이외의 연산자의 경우 두 피연산자를 일치하는 생성 열로 대체할 수 있습니다. `BETWEEN` 및 `IN()`의 경우 첫 번째 인수만 일치하는 생성 열로 대체할 수 있으며 다른 인수는 결과 유형이 동일해야 합니다. `BETWEEN` 및 `IN()`은 아직 JSON 값과 관련된 비교에는 지원되지 않습니다.

- 생성된 열은 최소한 함수 호출 또는 이전 항목에 언급된 연산자 중 하나를 포함하는 표현식으로 정의되어야 합니다. 표현식은 다른 열에 대한 단순한 참조로 구성될 수 없습니다. 예를 들어 `gc INT AS (f1) STORED`는 열 참조로만 구성되므로 `gc`의 인덱스는 고려되지 않습니다.

- 따옴표로 묶인 문자열을 반환하는 JSON 함수에서 값을 계산하는 인덱싱된 생성 열과 문자열을 비교하려면 열 정의에 `JSON_UNQUOTE()`를 사용하여 함수 값에서 여분의 따옴표를 제거해야 합니다. (문자열을 함수 결과와 직접 비교하는 경우 JSON 비교기가 따옴표 제거를 처리하지만 인덱스 조회에서는 이 작업이 수행되지 않습니다.) 예를 들어 다음과 같이 열 정의를 작성하는 대신:

  ```
  doc_name TEXT AS (JSON_EXTRACT(jdoc, '$.name')) STORED
  ```

  다음과 같이 작성합니다:

  ```
  doc_name TEXT AS (JSON_UNQUOTE(JSON_EXTRACT(jdoc, '$.name'))) STORED
  ```

  후자의 정의를 사용하면 옵티마이저가 이 두 가지 비교에 대해 일치하는 항목을 감지할 수 있습니다:

  ```
  ... WHERE JSON_EXTRACT(jdoc, '$.name') = 'some_string' ...
  ... WHERE JSON_UNQUOTE(JSON_EXTRACT(jdoc, '$.name')) = 'some_string' ...
  ```

  열 정의에 JSON_UNQUOTE()가 없으면 옵티마이저는 해당 비교 중 첫 번째 비교에 대해서만 일치 항목을 감지합니다.

- 옵티마이저가 잘못된 인덱스를 선택하면 인덱스 힌트를 사용하여 인덱스를 비활성화하고 옵티마이저가 다른 선택을 하도록 강제할 수 있습니다.

8.3.12 보이지 않음 색인

MySQL은 보이지 않는 인덱스, 즉 옵티마이저에서 사용하지 않는 인덱스를 지원합니다. 이 기능은 기본 키(명시적 또는 암시적) 이외의 인덱스에 적용됩니다.

인덱스는 기본적으로 표시됩니다. 새 인덱스의 표시 여부를 명시적으로 제어하려면 `CREATE TABLE`, `CREATE INDEX` 또는 `ALTER TABLE`에 대한 인덱스 정의의 일부로 `VISIBLE` 또는 `INVISIBLE` 키워드를 사용하세요:

```
CREATE TABLE t1 (
  i INT,
  j INT,
  k INT,
  INDEX i_idx (i) INVISIBLE
) 엔진 = InnoDB;
CREATE INDEX j_idx ON t1 (j) INVISIBLE;
ALTER TABLE t1 ADD INDEX k_idx (k) INVISIBLE;
```

기존 인덱스의 표시 여부를 변경하려면 `ALTER TABLE ...`과 함께 `VISIBLE` 또는 `INVISIBLE` 키워드를 사용합니다. ALTER `INDEX` 작업을 수행합니다:

```
ALTER TABLE t1 ALTER INDEX i_idx INVISIBLE;
ALTER TABLE t1 ALTER INDEX i_idx VISIBLE;
```

인덱스가 표시되는지 또는 보이지 않는지에 대한 정보는 정보 스키마에서 확인할 수 있습니다.

통계 테이블 또는 인덱스 표시 출력. 예를 들어

```
mysql> SELECT INDEX_NAME, IS_VISIBLE
       정보_schema.statistics에서
       WHERE TABLE_SCHEMA = 'db1' AND TABLE_NAME = 't1';
+------------+-------------+
| index_name | is_visible |
+------------+-------------+
| i_idx| 예   |
| j_idx| 아니요             |
| k_idx| 아니요             |
+------------+-------------+
```

보이지 않는 인덱스를 사용하면 인덱스가 필요한 것으로 판명될 경우 실행 취소해야 하는 파괴적인 변경을 하지 않고도 인덱스 제거가 쿼리 성능에 미치는 영향을 테스트할 수 있습니다. 인덱스를 삭제하고 다시 추가하는 작업은 큰 테이블의 경우 비용이 많이 들 수 있는 반면, 인덱스를 보이지 않게 하거나 보이게 하는 작업은 제자리에서 빠르게 수행할 수 있습니다.

보이지 않게 설정된 인덱스가 실제로 필요하거나 최적화 프로그램에서 사용하는 경우, 테이블에 대한 쿼리에 대한 인덱스 부재의 영향을 알아차릴 수 있는 몇 가지 방법이 있습니다:

- 보이지 않는 인덱스를 참조하는 인덱스 힌트가 포함된 쿼리에서 오류가 발생합니다.
- 성능 스키마 데이터는 영향을 받는 쿼리의 워크로드 증가를 보여줍니다.
- 쿼리에는 서로 다른 `EXPLAIN` 실행 계획이 있습니다.
- 이전에는 나타나지 않던 쿼리가 느린 쿼리 로그에 나타납니다.

`optimizer_switch` 시스템 변수의 `use_invisible_indexes` 플래그는 옵티마이저가 쿼리 실행 계획 구성에 보이지 않는 인덱스를 사용할지 여부를 제어합니다. 플래그가 꺼져 있으면 (기본값) 옵티마이저는 보이지 않는 인덱스를 무시합니다(이 플래그가 도입되기 전과 동일한 동작). 만약
플래그가 켜져 있으면 보이지 않는 인덱스는 보이지 않는 상태로 유지되지만 옵티마이저는 실행 계획 구성에 이를 고려합니다.

SET_VAR 옵티마이저 힌트를 사용하여 optimizer_switch의 값을 일시적으로 업데이트하면 다음과 같이 단일 쿼리 기간 동안만 보이지 않는 인덱스를 활성화할 수 있습니다:

```
mysql> EXPLAIN SELECT /*+ SET_VAR(optimizer_switch = 'use_invisible_indexes=on') */
    >     i, j FROM t1 WHERE j >= 50\G
*************************** 1. 행 ***************************
           ID: 1
  select_type: SIMPLE 테이
        블: t1
   파티션: NULL
         유형: 범위 가
능_키: J_IDX
          키: J_IDX
      KEY_LEN: 5
```

```
          참조:
         NULL 행: 2
     필터링됨: 100.00
        추가: 인덱스 조건 사용

mysql> EXPLAIN SELECT i, j FROM t1 WHERE j >= 50\G
*************************** 1. 행 ***************************
           ID: 1
  select_type: SIMPLE 테이
        블: t1
   파티션: NULL
         유형: 모든
가능한_키: NULL
          키: NULL
      key_len: NULL
          참조:
         NULL 행: 5
     필터링되었습니다: 33.33
        추가: 어디 사용
```

인덱스 가시성은 인덱스 유지 관리에 영향을 주지 않습니다. 예를 들어 인덱스는 테이블 행이 변경될 때마다 계속 업데이트되며, 고유 인덱스는 인덱스의 표시 여부와 관계없이 열에 중복이 삽입되는 것을 방지합니다.

명시적 기본 키가 없는 테이블에 `NOT NULL` 열에 `UNIQUE` 인덱스가 있는 경우 여전히 유효한 암시적 기본 키를 가질 수 있습니다. 이 경우 첫 번째 인덱스는 명시적 기본 키와 동일한 제약 조건을 테이블 행에 적용하며 해당 인덱스는 보이지 않게 만들 수 없습니다. 다음 테이블 정의를 고려하십시오:

```
CREATE TABLE t2 (
  i INT NOT NULL,
  j INT NOT NULL,
  UNIQUE j_idx (j)
   ) 엔진 = InnoDB;
```

이 정의에는 명시적인 기본 키가 포함되어 있지 않지만 `NOT NULL` 열 `j의` 인덱스는 기본 키와 동일한 제약 조건을 행에 적용하며 보이지 않게 만들 수 없습니다:

```
mysql> ALTER TABLE t2 ALTER INDEX j_idx INVISIBLE;
오류 3522 (HY000): 기본 키 인덱스는 보이지 않을 수 없습니다.
```

이제 테이블에 명시적 기본 키가 추가되었다고 가정합니다:

```
ALTER TABLE t2 ADD PRIMARY KEY (i);
```

명시적 기본 키는 보이지 않게 만들 수 없습니다. 또한 `j의` 고유 인덱스는 더 이상 암시적 기본 키로 작동하지 않으므로 보이지 않게 설정할 수 있습니다:

```
mysql> ALTER TABLE t2 ALTER INDEX j_idx INVISIBLE;
쿼리 확인, 영향을 받는 행 0개(0.03초)
```

8.3.13 내림차순 색인

MySQL은 내림차순 인덱스를 지원합니다: 인덱스 정의의 `DESC`는 더 이상 무시되지 않지만 키 값을 내림차순으로 저장합니다. 이전에는 인덱스를 역순으로 스캔할 수 있었지만 성능 저하가 발생했습니다. 내림차순 인덱스는 정방향 순서로 스캔할 수 있으므로 더 효율적입니다. 또한 내림차순 인덱스를 사용하면 일부 열에는 오름차순, 다른 열에는 내림차순을 혼합하는 것이 가장 효율적인 스캔 순서일 때 옵티마이저가 여러 열 인덱스를 사용할 수 있게 해줍니다.

열의 오름차순 및 내림차순 인덱스의 다양한 조합에 대한 두 개의 열과 네 개의 2열 인덱스 정의가 포함된 다음 테이블 정의를 고려합니다:

```
CREATE TABLE t (
  c1 INT, c2 INT,
  INDEX idx1(c1 ASC, c2 ASC),
```

```
  INDEX idx2(c1 ASC, c2 DESC),
  INDEX idx3(c1 DESC, c2 ASC),
  INDEX idx4(c1 DESC, c2 DESC)
);
```

테이블 정의는 4개의 개별 인덱스를 생성합니다. 옵티마이저는 각 `ORDER BY` 절에 대해 정방향 인덱스 스캔을 수행할 수 있으며 파일 정렬 작업을 사용할 필요가 없습니다:

```
ORDER BY c1 ASC, c2 ASC -- 옵티마이저가 idx1을 사용할 수
있음 ORDER BY c1 DESC, c2 DESC -- 옵티마이저가 idx4를 사
용할 수 있음 ORDER BY c1 ASC, c2 DESC -- 옵티마이저가
idx2를 사용할 수 있음 ORDER BY c1 DESC, c2 ASC -- 옵티마
이저가 idx3을 사용할 수 있음
```

내림차순 인덱스 사용에는 이러한 조건이 적용됩니다:

- 내림차순 인덱스는 `InnoDB` 스토리지 엔진에서만 지원되며 이러한 제한이 있습니다:
  - 인덱스에 내림차순 인덱스 키 열이 포함되어 있거나 기본 키에 내림차순 인덱스 열이 포함되어 있는 경우 보조 인덱스에는 변경 버퍼링이 지원되지 않습니다.
  - `InnoDB` SQL 구문 분석기는 내림차순 인덱스를 사용하지 않습니다. `InnoDB` 전체 텍스트 검색의 경우, 이는 인덱싱된 테이블의 `FTS_DOC_ID` 열에 필요한 인덱스를 내림차순 인덱스로 정의할 수 없음을 의미합니다. 자세한 내용은 섹션 15.6.2.4, "InnoDB 전체 텍스트 인덱스"를 참조하세요.
- 내림차순 인덱스는 오름차순 인덱스를 사용할 수 있는 모든 데이터 유형에 대해 지원됩니다.
- 내림차순 인덱스는 일반(비생성) 및 생성 열에 대해 지원됩니다(둘 다 가상 및 저장).
- 내림차순 키 부분을 포함하여 일치하는 열이 포함된 모든 인덱스를 사용할 수 있습니다.
- 내림차순 키 부분이 있는 인덱스는 집계 함수를 호출하지만 `GROUP BY` 절이 없는 쿼리의 `MIN()`/MAX`()` 최적화에는 사용되지 않습니다.
- 내림차순 인덱스는 `BTREE` 인덱스에 지원되지만 `HASH` 인덱스에는 지원되지 않습니다. 내림차순 인덱스는 전체 텍스트 또는 공간 인덱스에는 지원되지 않습니다.

  해시, 풀텍스트, 공간 인덱스에 명시적으로 `ASC` 및 `DESC` 지정자를 지정하면 오류가 발생합니다.

다음과 같이 최적화 도구가 내림차순 인덱스를 사용할 수 있음을 `EXPLAIN` 출력의 **`Extra`** 열에서 확인할 수 있습니다:

```
mysql> CREATE TABLE t1 (
    -> INT,
    -> b INT,
    -> INDEX a_desc_b_asc (a DESC, b ASC)
    -> );

mysql> EXPLAIN SELECT * FROM t1 ORDER BY a ASC\G
*************************** 1. 행 ***************************
           ID: 1
  select_type: SIMPLE 테이
        블: t1
   파티션: NULL
         유형: 인덱스 가능한
_키: NULL
          키: A_DESC_B_ASC 키_LEN:
      10
          참조:
         NULL 행: 1
     필터링됨: 100.00
        추가: 뒤로 인덱스 스캔; 인덱스 사용
```

내림차순 인덱스 사용은 `EXPLAIN FORMAT=TREE` 출력에서 다음과 같이 추가하여 표시됩니다. `(역방향)을` 인덱스 이름 뒤에 붙입니다(예: 다음과 같이):

```
mysql> EXPLAIN FORMAT=TREE SELECT * FROM t1 ORDER BY a ASC\G
*************************** 1. 행 ***************************
설명: -> a_desc_b_asc(역방향)를 사용하여 t1에서 인덱스 스캔 (비용=0.35 rows=1)
```

추가 정보 설명도 참조하세요.

8.3.14 타임스탬프 열에서 인덱싱된 조회

시간 값은 `TIMESTAMP` 열에 UTC 값으로 저장되며, `TIMESTAMP` 열에 삽입 및 검색된 값은 세션 표준 시간대와 UTC 간에 변환됩니다. (이 변환은 `CONVERT_TZ()` 함수에 의해 수행되는 것과 동일한 유형의 변환입니다. 세션 표준 시간대가 UTC인 경우 사실상 표준 시간대 변환이 수행되지 않습니다.)

일광 절약 시간제(DST)와 같은 현지 시간대 변경 규칙으로 인해 UTC와 UTC가 아닌 시간대 간의 변환은 양방향으로 일대일 변환이 되지 않습니다. 고유한 UTC 값이 다른 시간대에서는 고유하지 않을 수 있습니다. 다음 예는 UTC가 아닌 표준 시간대에서 동일하게 되는 고유한 UTC 값을 보여줍니다:

```
mysql> CREATE TABLE tstable (ts TIMESTAMP);
mysql> SET time_zone = 'UTC'; -- UTC 값 삽입
mysql> INSERT INTO tstable VALUES
       ('2018-10-28 00:30:00'),
       ('2018-10-28 01:30:00');
mysql> SELECT ts FROM tstable;
+---------------------+
| ts                  |
+---------------------+
| 2018-10-28 00:30:00 |
| 2018-10-28 01:30:00 |
+---------------------+
mysql> SET time_zone = 'MET'; -- UTC가 아닌 값 검색
mysql> SELECT ts FROM tstable;
+---------------------+
| ts                  |
+---------------------+
| 2018-10-28 02:30:00 |
| 2018-10-28 02:30:00 |
+---------------------+
```

참고

`'MET'` 또는 `'유럽/암스테르담'`과 같은 명명된 표준 시간대를 사용하려면 표준 시간대 테이블을 올바르게 설정해야 합니다. 자세한 내용은 5.1.15절. "MySQL 서버 표준 시간대 지원"을 참조하세요.

`'MET'` 표준 시간대로 변환할 때 두 개의 서로 다른 UTC 값이 동일하다는 것을 알 수 있습니다. 이 현상은 옵티마이저가 인덱스를 사용하여 쿼리를 실행하는지 여부에 따라 주어진 `TIMESTAMP` 열 쿼리에 대해 다른 결과를 초래할 수 있습니다.

쿼리가 앞서 표시된 테이블에서 `WHERE` 절을 사용하여 값을 선택하여 검색한다고 가정합니다.
열을 사용자가 제공한 타임스탬프 리터럴과 같은 단일 특정 값에 대해 사용할 수 있습니다:

```
SELECT ts FROM tstable
WHERE ts = '리터럴';
```

이 조건에서 쿼리가 실행된다고 가정해 보겠습니다:

- 세션 표준 시간대는 UTC가 아니며 DST 시프트가 있습니다. 예를 들어

```
SET time_zone = 'MET';
```

- `TIMESTAMP` 열에 저장된 고유 UTC 값은 서머타임 이동으로 인해 세션 표준 시간대에서 고유하지 않습니다. (앞서 표시된 예는 이러한 현상이 어떻게 발생하는지 보여줍니다.)

- 쿼리는 세션 시간대에서 DST로 진입한 후 1시간 이내의 검색 값을 지정합니다.

이러한 조건에서 `WHERE` 절의 비교는 인덱싱되지 않은 조회와 인덱싱된 조회에 대해 서로 다른 방식으로 수행되며 서로 다른 결과로 이어집니다:

- 인덱스가 없거나 옵티마이저가 인덱스를 사용할 수 없는 경우, 비교는 세션 시간대에서 이루어집니다. 옵티마이저는 테이블 스캔을 수행하여 각 `ts` 열 값을 검색하고 UTC에서 세션 표준 시간대로 변환한 다음 검색 값과 비교합니다(세션 표준 시간대에서도 해석됨):

```
mysql> SELECT ts FROM tstable
       WHERE ts = '2018-10-28 02:30:00';
+---------------------+
| ts                  |
+---------------------+
| 2018-10-28 02:30:00 |
| 2018-10-28 02:30:00 |
+---------------------+
```

  저장된 `ts` 값은 세션 시간대로 변환되므로 쿼리에서 UTC 값으로는 다르지만 세션 시간대에서는 동일한 두 개의 타임스탬프 값을 반환할 수 있습니다: 하나는 시계가 변경될 때 DST 변경 이전에 발생한 값이고 다른 하나는 DST 변경 이후에 발생한 값입니다.

- 사용 가능한 인덱스가 있는 경우 비교는 UTC로 이루어집니다. 옵티마이저는 인덱스 스캔을 수행하여 먼저 세션 시간대의 검색 값을 UTC로 변환한 다음 그 결과를 UTC 인덱스 항목과 비교합니다:

```
mysql> ALTER TABLE tstable ADD INDEX (ts);
mysql> SELECT ts FROM tstable
       WHERE ts = '2018-10-28 02:30:00';
+---------------------+
| ts                  |
+---------------------+
| 2018-10-28 02:30:00 |
+---------------------+
```

  이 경우 (변환된) 검색 값은 인덱스 항목에만 일치하며, 저장된 고유한 UTC 값에 대한 인덱스 항목도 고유하므로 검색 값은 이 중 하나만 일치할 수 있습니다.

인덱싱되지 않은 조회와 인덱싱된 조회에 대한 옵티마이저 작업이 다르기 때문에 쿼리는 각 경우에 다른 결과를 생성합니다. 인덱싱되지 않은 조회 결과는 세션 시간대에서 일치하는 모든 값을 반환합니다. 인덱싱된 조회는 그렇게 할 수 없습니다:

- 이 작업은 UTC 값만 알고 있는 스토리지 엔진 내에서 수행됩니다.
- 동일한 UTC 값에 매핑되는 두 개의 서로 다른 세션 표준 시간대 값의 경우, 인덱싱된 조회는 해당 UTC 인덱스 항목과만 일치하고 단일 행만 반환합니다.

앞의 설명에서, `tstable`에 저장된 데이터 세트는 서로 다른 UTC 값으로 구성되어 있습니다. 이러한 경우, 표시된 형식의 모든 인덱스 사용 쿼리는 최대 하나의 인덱스 항목과 일치합니다.

인덱스가 고유하지 않은 경우 테이블(및 인덱스)에 주어진 UTC 값의 여러 인스턴스가 저장될 수 있습니다. 예를 들어, `ts` 열에는 UTC 값의 여러 인스턴스가 포함될 수 있습니다. `'2018-10-28 00:30:00'`. 이 경우 인덱스를 사용하는 쿼리는 각각을 반환합니다(결과 집합에서 `'2018-10-28 02:30:00'` MET 값으로 변환됨). 인덱스 사용 쿼리는 세션 시간대의 검색 값으로 변환되는 여러

UTC 값을 일치시키는 것이 아니라 변환된 검색 값을 UTC 인덱스 항목의 단일 값과 일치시킵니다.

세션 시간대에 일치하는 모든 `ts` 값을 반환하는 것이 중요한 경우 해결 방법은 `IGNORE INDEX` 힌트를 사용하여 인덱스 사용을 억제하는 것입니다:

```
mysql> SELECT ts FROM tstable
       IGNORE INDEX (ts)
       WHERE ts = '2018-10-28 02:30:00';
+---------------------+
```

```
| ts                  |
+---------------------+
| 2018-10-28 02:30:00 |
| 2018-10-28 02:30:00 |
+---------------------+
```

양방향 시간대 변환에 대한 일대일 매핑 부족은 `FROM_UNIXTIME()` 및 `UNIX_TIMESTAMP()` 함수로 수행되는 변환과 같은 다른 컨텍스트에서도 동일하게 발생합니다. 섹션 12.7, "날짜 및 시간 함수"를 참조하세요.

8.4 데이터베이스 최적화 구조

데이터베이스 디자이너로서 스키마, 테이블, 열을 가장 효율적으로 구성할 수 있는 방법을 찾아야 합니다. 애플리케이션 코드를 튜닝할 때와 마찬가지로 I/O를 최소화하고, 관련 항목을 함께 유지하며, 데이터 볼륨이 증가해도 성능이 높게 유지되도록 미리 계획해야 합니다. 효율적인 데이터베이스 설계로 시작하면 팀원들이 고성능 애플리케이션 코드를 더 쉽게 작성할 수 있으며, 애플리케이션이 발전하고 재작성될 때에도 데이터베이스가 지속될 가능성이 높아집니다.

8.4.1 데이터 최적화 크기

디스크 공간을 최소화하도록 테이블을 설계하세요. 이렇게 하면 디스크에 쓰거나 디스크에서 읽는 데이터의 양이 줄어들어 성능이 크게 향상될 수 있습니다. 테이블 크기가 작을수록 일반적으로 쿼리 실행 중에 콘텐츠가 활발하게 처리되는 동안 메인 메모리를 덜 필요로 합니다. 테이블 데이터의 공간이 줄어들면 인덱스도 작아져 더 빠르게 처리할 수 있습니다.

MySQL은 다양한 저장소 엔진(테이블 유형)과 행 형식을 지원합니다. 각 테이블에 대해 사용할 스토리지 및 인덱싱 방법을 결정할 수 있습니다. 애플리케이션에 적합한 테이블 형식을 선택하면 성능을 크게 향상시킬 수 있습니다. 15장, *InnoDB 스토리지 엔진* 및 16장, *대체 스토리지 엔진을* 참조하세요.

여기에 나열된 기술을 사용하면 테이블의 성능을 개선하고 저장 공간을 최소화할 수 있습니다:

- 테이블 열
- 행 형식
- 색인
- 조인
- 정규화

테이블 열

- 가능한 한 가장 효율적인(가장 작은) 데이터 유형을 사용하세요. MySQL에는 디스크 공간과 메모리를 절약하는 특수한 유형이 많이 있습니다. 예를 들어, 가능하면 더 작은 정수 유형을 사용하여 더 작은 테이블을 만

드세요. MEDIUMINT 열은 공간을 25% 적게 사용하므로 INT보다 MEDIUMINT가 더 나은 선택인 경우가 많습니다.

- 가능하면 열을 NOT NULL로 선언하세요. 이렇게 하면 인덱스를 더 잘 사용할 수 있고 각 값이 NULL인지 테스트하는 데 드는 오버헤드를 제거하여 SQL 작업이 더 빨라집니다. 또한 열당 1비트씩 저장 공간을 절약할 수 있습니다. 테이블에 NULL 값이 꼭 필요한 경우 이를 사용하세요. 모든 열에 NULL 값을 허용하는 기본 설정은 피하세요.

행 형식

- InnoDB 테이블은 기본적으로 DYNAMIC 행 형식을 사용하여 생성됩니다. DYNAMIC이 아닌 다른 행 형식을 사용하려면 innodb_default_row_format을 구성하거나 CREATE TABLE 또는 ALTER TABLE 문에 ROW_FORMAT 옵션을 명시적으로 지정합니다.

  컴팩트, 다이나믹 및 압축을 포함하는 컴팩트 행 형식 제품군은 일부 작업에서 CPU 사용량을 증가시키는 대신 행 저장 공간을 줄입니다. 다음과 같은 경우

워크로드가 캐시 적중률과 디스크 속도에 의해 제한되는 일반적인 워크로드라면 더 빠를 가능성이 높습니다. 드물게 CPU 속도에 의해 제한되는 경우라면 속도가 느려질 수 있습니다.

컴팩트한 행 형식 제품군은 utf8mb3 또는 utf8mb4와 같은 가변 길이 문자 집합을 사용할 때 CHAR 열 저장소도 최적화합니다. ROW_FORMAT=REDUNDANT를 사용하면 CHAR(*N*)이 다음을 차지합니다. *N* × 문자 집합의 최대 바이트 길이. 많은 언어가 주로 단일 바이트 utf8mb3 또는 utf8mb4 문자를 사용하여 작성될 수 있으므로 고정된 저장소 길이로 인해 공간이 낭비되는 경우가 많습니다. 컴팩트한 행 형식 제품군을 사용하는 InnoDB는 *N*의 범위에서 가변적인 양의 스토리지를 할당합니다.
를 후행 공백을 제거하여 해당 열에 대한 문자 집합의 최대 바이트 길이인 *N* ×로 설정합니다. 일반적인 경우 제자리 업데이트를 용이하게 하기 위해 최소 저장소 길이는 N바이트입니다. 자세한 내용은 섹션 15.10, "InnoDB 행 형식"을 참조하십시오.

- 테이블 데이터를 압축된 형태로 저장하여 공간을 더욱 최소화하려면 InnoDB 테이블을 생성할 때 ROW_FORMAT=COMPRESSED를 지정하거나 기존 MyISAM 테이블에서 myisampack 명령을 실행합니다. (InnoDB 압축 테이블은 읽기 및 쓰기가 가능하지만 MyISAM 압축 테이블은 읽기 전용입니다.)

색인

- MyISAM 테이블의 경우 가변 길이 열(VARCHAR, TEXT 또는 BLOB
열)의 경우 고정 크기 행 형식이 사용됩니다. 이 방식은 더 빠르지만 공간을 다소 낭비할 수 있습니다. 참조 섹션 16.2.3, "MyISAM 테이블 저장소 형식". CREATE TABLE 옵션 ROW_FORMAT=FIXED를 사용하여 VARCHAR 열이 있더라도 고정 길이 행을 원한다는 것을 암시할 수 있습니다.

- 테이블의 기본 인덱스는 가능한 한 짧아야 합니다. 이렇게 하면 각 행을 쉽고 효율적으로 식별할 수 있습니다. InnoDB 테이블의 경우 기본 키 열은 각 보조 인덱스 항목에 중복되므로 보조 인덱스가 많은 경우 기본 키가 짧으면 상당한 공간을 절약할 수 있습니다.

조인

- 쿼리 성능을 개선하는 데 필요한 인덱스만 생성하세요. 인덱스는 검색에는 좋지만 삽입 및 업데이트 작업 속도가 느려집니다. 주로 열의 조합을 검색하여 테이블에 액세스하는 경우 각 열에 대해 별도의 인덱스가 아닌 단일 복합 인덱스를 만드세요. 인덱스의 첫 번째 부분은 가장 많이 사용되는 열이어야 합니다. 테이블에서 선택할 때 *항상* 많은 열을 사용하는 경우 인덱스의 첫 번째 열은 중복이 가장 많은 열이어야 인덱스를 더 잘 압축할 수 있습니다.

- 긴 문자열 열에 첫 번째 문자 수에 고유 접두사가 있을 가능성이 매우 높은 경우 열의 가장 왼쪽 부분에 인덱스를 생성하는 MySQL의 지원을 사용하여 이 접두사만 인덱싱하는 것이 좋습니다(섹션 13.1.15, "CREATE INDEX 문" 참조). 인덱스가 짧을수록 디스크 공간이 적게 필요할 뿐만 아니라 인덱스 캐시에서 더 많은 히트를 얻을 수 있으므로 디스크 검색 횟수가 줄어들기 때문에 더 빠릅니다. 5.1.1절, "서버 구성"을 참조하십시오.

- 어떤 상황에서는 자주 스캔하는 테이블을 두 개로 분할하는 것이 유리할 수 있습니다. 동적 형식의 테이블이고 테이블을 스캔할 때 관련 행을 찾는 데 사용할 수 있는 더 작은 정적 형식의 테이블을 사용할 수 있는 경우 특히 그렇습니다.

- 동일한 데이터 유형의 다른 테이블에 동일한 정보를 가진 열을 선언하여 해당 열을 기반으로 조인 속도를 높입니다.
- 열 이름을 단순하게 유지하면 여러 테이블에서 동일한 이름을 사용할 수 있고 조인 쿼리를 간소화할 수 있습니다. 예를 들어, 고객이라는 이름의 테이블에서 고객_이름 대신 이름이라는 열 이름을 사용합니다. 다른 SQL 서버로 이름을 이식할 수 있게 하려면 이름을 18자 미만으로 짧게 유지하는 것이 좋습니다.

정규화

- 일반적으로 모든 데이터를 중복되지 않게 유지하세요(데이터베이스 이론에서 *제3의 정규형이라고 하는* 것을 준수하세요). 이름이나 주소와 같은 긴 값을 반복하는 대신 고유한 ID를 할당하고, 필요에 따라 여러 개의 작은 테이블에 걸쳐 이러한 ID를 반복하고, 조인 절에서 ID를 참조하여 쿼리에서 테이블을 조인합니다.

- 예를 들어 큰 테이블의 모든 데이터를 분석하는 비즈니스 인텔리전스 시나리오에서 디스크 공간과 여러 데이터 복사본을 유지하는 데 드는 유지 관리 비용보다 속도가 더 중요한 경우, 정규화 규칙을 완화하여 정보를 복제하거나 요약 테이블을 만들어 속도를 높일 수 있습니다.

8.4.2 MySQL 데이터 최적화 유형

8.4.2.1 숫자 데이터 최적화

- 고유 ID 또는 문자열이나 숫자로 표현할 수 있는 기타 값의 경우 문자열 열보다 숫자 열을 선호합니다. 큰 숫자 값은 해당 문자열보다 적은 바이트에 저장할 수 있으므로 전송 및 비교에 더 빠르고 메모리를 덜 차지합니다.
- 숫자 데이터를 사용하는 경우, 대부분의 경우 텍스트 파일에 액세스하는 것보다 데이터베이스에서 (라이브 연결을 사용하여) 정보에 액세스하는 것이 더 빠릅니다. 데이터베이스의 정보는 텍스트 파일보다 더 압축된 형식으로 저장될 가능성이 높으므로 데이터베이스에 액세스할 때 디스크 액세스 횟수가 줄어듭니다. 또한 텍스트 파일을 파싱하여 줄과 열 경계를 찾을 필요가 없으므로 애플리케이션에 코드를 저장할 수 있습니다.

8.4.2.2 문자 및 문자열 유형에 최적화

문자 및 문자열 열의 경우 다음 지침을 따르세요:

- 언어별 데이터 정렬 기능이 필요하지 않은 경우 빠른 비교 및 정렬 작업을 위해 이진 데이터 정렬 순서를 사용하세요. `BINARY` 연산자를 사용하여 특정 쿼리 내에서 이진 콜레이션을 사용할 수 있습니다.
- 서로 다른 열의 값을 비교할 때는 쿼리를 실행하는 동안 문자열 변환을 피하기 위해 가능한 한 동일한 문자 집합과 데이터 정렬을 사용하여 해당 열을 선언합니다.
- 크기가 8KB 미만인 열 값의 경우 `BLOB` 대신 이진 `VARCHAR`를 사용합니다. `GROUP BY` 및 `ORDER BY` 절은 임시 테이블을 생성할 수 있으며, 이러한 임시 테이블은 원본 테이블에 `BLOB` 열이 포함되지 않은 경우 `MEMORY` 스토리지 엔진을 사용할 수 있습니다.
- 테이블에 이름, 주소와 같은 문자열 열이 포함되어 있지만 많은 쿼리에서 해당 열을 검색하지 않는 경우 문자열 열을 별도의 테이블로 분할하고 필요한 경우 외래 키가 포함된 조인 쿼리를 사용하는 것이 좋습니다. MySQL은 행에서 값을 검색할 때 해당 행의 모든 열(및 인접한 다른 행)이 포함된 데이터 블록을 읽습니다. 가장 자주 사용되는 열만 사용하여 각 행을 작게 유지하면 각 데이터 블록에 더 많은 행을 넣을 수 있습니다. 이러한 컴팩트한 테이블은 일반적인 쿼리에 대한 디스크 I/O 및 메모리 사용량을 줄여줍니다.
- 무작위로 생성된 값을 `InnoDB` 테이블의 기본 키로 사용하는 경우, 가능하면 현재 날짜 및 시간과 같은 오름차순 값을 접두사로 붙이세요. 연속된 기본값이 물리적으로 서로 가까이 저장되어 있으면 `InnoDB`에서 더 빠르게 삽입하고 검색할 수 있습니다.

- 일반적으로 숫자 열이 동등한 문자열 열보다 선호되는 이유는 섹션 8.4.2.1, '숫자 데이터에 맞게 최적화'를 참조하십시오.

8.4.2.3 BLOB 유형에 맞게 최적화

- 텍스트 데이터가 포함된 큰 블롭을 저장할 때는 먼저 압축하는 것이 좋습니다. 전체 테이블이 `InnoDB` 또는 `MyISAM`에 의해 압축된 경우에는 이 기술을 사용하지 마세요.
- 여러 열이 있는 테이블의 경우 BLOB 열을 사용하지 않는 쿼리에 대한 메모리 요구 사항을 줄이려면 BLOB 열을 별도의 테이블로 분할하고 필요할 때 조인 쿼리로 참조하는 것을 고려하십시오.
- BLOB 값을 검색하고 표시하는 데 필요한 성능 요구 사항이 다른 데이터 유형과 매우 다를 수 있으므로 BLOB 관련 테이블을 다른 저장 장치 또는

별도의 데이터베이스 인스턴스입니다. 예를 들어, BLOB을 검색하려면 SSD 장치보다 기존 하드 드라이브에 더 적합한 대용량 순차 디스크 읽기가 필요할 수 있습니다.

- 바이너리 VARCHAR가 필요한 이유는 섹션 8.4.2.2, "문자 및 문자열 유형에 최적화"를 참조하십시오. 열이 동등한 BLOB 열보다 선호되는 경우가 있습니다.
- 매우 긴 텍스트 문자열에 대해 동일성을 테스트하는 대신 열 값의 해시를 별도의 열에 저장하고 해당 열을 인덱싱한 다음 쿼리에서 해시값을 테스트할 수 있습니다. (MD5() 또는 CRC32() 함수를 사용하여 해시 값을 생성할 수 있습니다.) 해시 함수는 서로 다른 입력에 대해 중복된 결과를 생성할 수 있으므로 여전히 AND *blob_column* = long_string_value에
  를 쿼리에 추가하여 잘못된 일치를 방지하고, 해시값에 대한 인덱스가 더 작고 쉽게 스캔할 수 있어 성능상의 이점을 얻을 수 있습니다.

8.4.3 많은 테이블에 최적화

개별 쿼리를 빠르게 처리하기 위한 몇 가지 기술에는 데이터를 여러 테이블로 분할하는 것이 포함됩니다. 테이블 수가 수천 개 또는 수백만 개에 달하면 이러한 모든 테이블을 처리하는 데 따른 오버헤드가 새로운 성능 고려 사항이 됩니다.

8.4.3.1 MySQL이 테이블을 열고 닫는 방법

mysqladmin 상태 명령을 실행하면 다음과 같은 내용이 표시됩니다:

```
가동 시간: 426 실행 중인 스레드: 1 질문 11082
다시 로드합니다: 1 오픈 테이블: 12
```

테이블 수가 12개 미만인 경우 열려 있는 테이블 값 12는 다소 당황스러울 수 있습니다.

MySQL은 멀티스레드이므로 특정 테이블에 대해 동시에 많은 클라이언트가 쿼리를 실행할 수 있습니다. 여러 클라이언트 세션이 동일한 테이블에 대해 서로 다른 상태를 갖는 문제를 최소화하기 위해 각 동시 세션에서 테이블을 독립적으로 엽니다. 이렇게 하면 메모리가 추가로 사용되지만 일반적으로 성능이 향상됩니다. MyISAM 테이블을 사용하면 테이블이 열려 있는 각 클라이언트의 데이터 파일에 대해 하나의 추가 파일 설명자가 필요합니다. (반면 인덱스 파일 설명자는 모든 세션 간에 공유됩니다.)

table_open_cache 및 max_connections 시스템 변수는 서버가 열어 두는 최대 파일 수에 영향을 줍니다. 이 값 중 하나 또는 둘 모두를 늘리면 운영 체제에서 프로세스당 열린 파일 설명자 수에 대해 부과한 제한에 부딪힐 수 있습니다. 많은 운영 체제에서 열린 파일 수 제한을 늘릴 수 있지만, 그 방법은 시스템마다 크게 다릅니다. 운영 체제 설명서를 참조하여 제한을 늘릴 수 있는지 여부와 그 방법을 확인하세요.

TABLE_OPEN_CACHE는 MAX_CONNECTIONS와 관련이 있습니다. 예를 들어, 동시 실행 연결이 200개인 경우 테이블 캐시 크기를 *200*N* 이상으로 지정합니다. 여기서 *N*은 실행하는 쿼리에서 조인당 최대 테이블 수입니다. 또한 임시 테이블 및 파일을 위한 추가 파일 설명자를 예약해야 합니다.

운영 체제가 `table_open_cache` 설정에 암시된 열린 파일 설명자 수를 처리할 수 있는지 확인하세요. `table_open_cache`를 너무 높게 설정하면 MySQL에서 파일 설명자가 부족하여 연결이 거부되거나 쿼리 수행에 실패하는 등의 증상이 나타날 수 있습니다.

또한 `MyISAM` 스토리지 엔진은 각 고유한 열린 테이블에 대해 두 개의 파일 설명자가 필요하다는 점을 고려하세요. MySQL에서 사용할 수 있는 파일 설명자 수를 늘리려면 `open_files_limit` 시스템 변수를 설정합니다. 섹션 B.3.2.16, "파일을 찾을 수 없음 및 유사 오류"를 참조하세요.

열려 있는 테이블의 캐시는 `table_open_cache` 항목 수준으로 유지됩니다. 서버는 시작할 때 캐시 크기를 자동으로 조정합니다. 명시적으로 크기를 설정하려면 시작할 때 `table_open_cache` 시스템 변수를 설정합니다. 이 섹션의 뒷부분에 설명된 대로 MySQL은 쿼리를 실행하기 위해 이보다 더 많은 테이블을 일시적으로 열 수 있습니다.

MySQL은 다음과 같은 상황에서 사용하지 않는 테이블을 닫고 테이블 캐시에서 제거합니다:

- 캐시가 가득 차서 스레드가 캐시에 없는 테이블을 열려고 시도하는 경우.
- 캐시에 table_open_cache 항목이 2개 이상 포함되어 있고 캐시에 있는 테이블이 더 이상 스레드에서 사용되지 않는 경우.
- 테이블 플러시 작업이 발생하는 경우. 누군가 테이블 플러시 작업을 발행할 때 발생합니다. 문을 실행하거나 mysqladmin flush-tables 또는 mysqladmin refresh 명령을 실행합니다.

테이블 캐시가 가득 차면 서버는 다음 절차를 사용하여 사용할 캐시 항목을 찾습니다:

- 현재 사용되지 않는 테이블은 가장 최근에 사용된 테이블부터 해제됩니다.
- 새 테이블을 열어야 하지만 캐시가 꽉 차서 테이블을 해제할 수 없는 경우 필요에 따라 캐시가 일시적으로 확장됩니다. 캐시가 일시적으로 확장된 상태에서 테이블이 사용 상태에서 미사용 상태로 전환되면 테이블이 닫히고 캐시에서 해제됩니다.

각 동시 액세스에 대해 MyISAM 테이블이 열립니다. 즉, 두 개의 스레드가 동일한 테이블에 액세스하거나 한 스레드가 동일한 쿼리에서 테이블에 두 번 액세스하는 경우(예: 테이블을 자신에 조인하여 액세스하는 경우) 테이블을 두 번 열어야 합니다. 각 동시 열기에는 테이블 캐시에 항목이 필요합니다. MyISAM 테이블을 처음 열면 데이터 파일과 인덱스 파일에 대한 두 개의 파일 설명자가 필요합니다. 테이블을 추가로 사용할 때마다 데이터 파일에 대한 파일 설명자는 하나만 필요합니다. 인덱스 파일 설명자는 모든 스레드에서 공유됩니다.

HANDLER *tbl_name* OPEN 문을 사용하여 테이블을 여는 경우 스레드에 전용 테이블 개체가 할당됩니다. 이 테이블 개체는 다른 스레드에서 공유되지 않으며 스레드가 핸들러 *tbl_name* CLOSE를 호출하거나 스레드가 종료됩니다. 이 경우 테이블은 테이블 캐시에 다시 저장됩니다(캐시가 가득 차지 않은 경우). 섹션 13.2.5, "HANDLER 문"을 참고한다.

테이블 캐시가 너무 작은지 확인하려면 서버 시작 이후 테이블 열기 작업 횟수를 나타내는 Opened_tables 상태 변수를 확인합니다:

```
mysql> SHOW GLOBAL STATUS LIKE 'Opened_tables';
+---------------+---------+
| 변수_이름 | 값 |
+---------------+---------+
| 열린_테이블 | 2741 |
+---------------+---------+
```

플러시 테이블을 많이 발행하지 않은 경우에도 값이 매우 크거나 빠르게 증가하는 경우 문을 사용하여 서버를 시작할 때 table_open_cache 값을 늘리세요.

8.4.3.2 동일한 데이터베이스에 많은 테이블을 만들 때의 단점

동일한 데이터베이스 디렉터리에 MyISAM 테이블이 여러 개 있는 경우 열기, 닫기 및 만들기 작업이 느려집니다. 여러 테이블에서 SELECT 문을 실행하는 경우 테이블 캐시가 가득 차면 열어야 하는 모든 테이블에 대해 다

른 테이블을 닫아야 하므로 약간의 오버헤드가 발생합니다. 테이블 캐시에 허용되는 항목 수를 늘리면 이 오버헤드를 줄일 수 있습니다.

8.4.4 MySQL에서 내부 임시 테이블 사용

경우에 따라 서버는 문을 처리하는 동안 내부 임시 테이블을 생성합니다. 사용자는 이러한 상황이 발생하는 시점을 직접 제어할 수 없습니다.

서버는 다음과 같은 조건에서 임시 테이블을 생성합니다:

- 나중에 설명하는 몇 가지 예외를 제외하고, 유니온 진술의 평가.
- 임시 알고리즘, `UNION` 또는 집계를 사용하는 뷰와 같은 일부 뷰에 대한 평가입니다.
- 파생 테이블의 평가(섹션 13.2.15.8, "파생 테이블" 참조).

- 일반 테이블 표현식 평가(섹션 13.2.20, "WITH(일반 테이블 표현식)" 참조).
- 하위 쿼리 또는 준조인 구체화를 위해 만들어진 테이블(섹션 8.2.2, "하위 쿼리, 파생 테이블, 뷰 참조 및 일반 테이블 식 최적화하기" 참조).
- `ORDER BY` 절과 다른 `GROUP BY` 절을 포함하거나 `ORDER BY` 또는 `GROUP BY`에 조인 큐의 첫 번째 테이블이 아닌 다른 테이블의 열이 포함된 문 평가.
- `ORDER BY`와 결합된 `DISTINCT`를 평가하려면 임시 테이블이 필요할 수 있습니다.
- `SQL_SMALL_RESULT` 수정자를 사용하는 쿼리의 경우 쿼리에 디스크 저장소가 필요한 요소(나중에 설명)도 포함되어 있지 않는 한 MySQL은 인메모리 임시 테이블을 사용합니다.
- 동일한 테이블에서 선택하고 동일한 테이블에 삽입하는 `INSERT ... SELECT` 문을 평가하기 위해 MySQL은 `SELECT`의 행을 보관할 내부 임시 테이블을 생성한 다음 해당 행을 대상 테이블에 삽입합니다. 섹션 13.2.7.1, "INSERT ... SELECT 문"을 참조하십시오.
- 다중 테이블 `UPDATE` 문 평가.
- `GROUP_CONCAT()` 또는 `COUNT(DISTINCT)` 식의 평가.
- 창 함수 평가(섹션 12.20, "창 함수" 참조)는 필요에 따라 임시 테이블을 사용합니다.

문에 임시 테이블이 필요한지 확인하려면 `EXPLAIN`을 사용하고 추가 열에 `Using temporary`가 있는지 확인합니다(섹션 8.8.1, "EXPLAIN으로 쿼리 최적화하기" 참조). 파생되거나 구체화된 임시 테이블에 대해 반드시 `Using temporary`가 표시되는 것은 아닙니다. 윈도우 함수를 사용하는 문의 경우 `FORMAT=JSON`과 함께 `EXPLAIN`은 항상 윈도우 단계에 대한 정보를 제공합니다. 윈도우 함수가 임시 테이블을 사용하는 경우 각 단계에 대해 표시됩니다.

일부 쿼리 조건에서는 인메모리 임시 테이블을 사용할 수 없으며, 이 경우 서버는 대신 온디스크 테이블을 사용합니다:

- 테이블에 `BLOB` 또는 `TEXT` 열이 있는지 여부. MySQL 8.2의 인메모리 내부 임시 테이블을 위한 기본 스토리지 엔진인 `TempTable` 스토리지 엔진은 이진 대용량 개체 유형을 지원합니다. 내부 임시 테이블 스토리지 엔진을 참조하십시오.
- `UNION` 또는 `UNION ALL`을 사용하는 경우 `SELECT` 목록에 최대 길이가 512(이진 문자열의 경우 바이트, 이진 문자열이 아닌 경우 문자)보다 큰 문자열 열이 있는지 여부입니다.
- `SHOW COLUMNS` 및 `DESCRIBE` 문은 일부 열의 유형으로 BLOB을 사용하므로 결과에 사용되는 임시 테이블은 디스크에 있는 테이블입니다.

서버는 특정 자격을 충족하는 `UNION` 문에 임시 테이블을 사용하지 않습니다. 대신 결과 열 타입 캐스팅을 수행하는 데 필요한 데이터 구조만 임시 테이블 생성에서 유지합니다. 테이블은 완전히 인스턴스화되지 않으며 행을 쓰거나 읽지 않고 행을 클라이언트로 직접 전송합니다. 그 결과 메모리 및 디스크 요구 사항이 줄어들고, 서

버가 마지막 쿼리 블록이 실행될 때까지 기다릴 필요가 없기 때문에 첫 번째 행이 클라이언트로 전송되기까지의 지연이 줄어듭니다. `EXPLAIN` 및 옵티마이저 추적 출력은 이 실행 전략을 반영합니다: `UNION RESULT` 쿼리 블록은 임시 테이블에서 읽는 부분에 해당하므로 존재하지 않습니다.

이러한 조건은 임시 테이블 없이 평가할 수 있는 자격을 부여합니다:

- 유니온은 유니온 또는 유니온 디스팅트가 아닌 유니온 올입니다.
- 글로벌 `ORDER BY` 절이 없습니다.
- 유니온이 `{INSERT | REPLACE}`의 최상위 쿼리 블록이 아닙니다. `... SELECT ...` 문을 사용합니다.

내부 임시 테이블 스토리지 엔진

내부 임시 테이블은 메모리에 보관하고 TempTable 또는 MEMORY에 의해 처리될 수 있습니다.
스토리지 엔진에 의해 디스크에 저장되거나 InnoDB 스토리지 엔진에 의해 디스크에 저장됩니다.

인메모리 내부 임시 테이블용 스토리지 엔진

internal_tmp_mem_storage_engine 변수는 인메모리 내부 임시 테이블에 사용되는 스토리지 엔진을 정의합니다. 허용되는 값은 TempTable(기본값) 및 MEMORY입니다.

참고

internal_tmp_mem_storage_engine에 대한 세션 설정 구성하기 SESSION_VARIABLES_ADMIN 또는 SYSTEM_VARIABLES_ADMIN이 필요합니다. 권한이 있습니다.

TempTable 스토리지 엔진은 VARCHAR 및 VARBINARY 열과 기타 이진 대용량 객체 유형을 위한 효율적인 스토리지를 제공합니다.

다음 변수는 TempTable 스토리지 엔진 제한 및 동작을 제어합니다:

- tmp_table_size: TempTable 스토리지 엔진에 의해 생성된 개별 인메모리 내부 임시 테이블의 최대 크기를 정의합니다. tmp_table_size 제한에 도달하면,
  MySQL은 인메모리 내부 임시 테이블을 InnoDB 온디스크 내부 임시 테이블로 자동 변환합니다. 기본 tmp_table_size 설정은 16777216 바이트(16 MiB)입니다.

  tmp_table_size 제한은 개별 쿼리가 과도한 양의 글로벌 TempTable 리소스를 소비하지 못하도록 하기 위한 것으로, 이는 TempTable 리소스를 필요로 하는 동시 쿼리의 성능에 영향을 미칠 수 있습니다. 글로벌 TempTable 리소스는 temptable_max_ram 및 temptable_max_mmap 설정에 의해 제어됩니다.

  tmp_table_size 제한이 temptable_max_ram 제한보다 작으면 인메모리 임시 테이블에 tmp_table_size 제한에서 허용하는 것보다 많은 데이터를 포함할 수 없습니다.
  tmp_table_size 제한이 temptable_max_ram 및
  한도를초과하는 경우, 인메모리 임시 테이블에 temptable_max_ram과 temptable_max_map 한도를 합한 것보다 더 많은 테이블을 포함할 수 없습니다.

- TEMPTABLE_MAX_RAM: 임시 테이블 스토리지 엔진이 메모리 매핑된 파일에서 공간 할당을 시작하기 전 또는 MySQL이 구성에 따라 InnoDB 온디스크 내부 임시 테이블 사용을 시작하기 전에 사용할 수 있는 최대 RAM 양을 정의합니다. 기본 temptable_max_ram 설정은 1073741824 바이트(1GiB)입니다.

참고

temptable_max_ram 설정은 임시 테이블 스토리지 엔진을 사용하는 각 스레드에 할당된 스레드-로컬 메모리 블록을 고려하지 않습니다. 스레드-로컬 메모리 블록의 크기는 스레드의 첫 번째 메모리 할당 요청의 크기에 따라 달라집니다. 요청이 1MB 미만인 경우(대부분의 경우) 스레드-로컬 메모리 블록 크기는 1MB입니다. 요

청이 1MB보다 크면 스레드-로컬 메모리 블록은 초기 메모리 요청과 거의 같은 크기입니다. 스레드-로컬 메모리 블록은 스레드가 종료될 때까지 스레드-로컬 스토리지에 유지됩니다.

- `TEMPTABLE_USE_MMAP`: `temptable_max_ram` 제한을 초과할 때 `TempTable` 스토리지 엔진이 메모리 매핑된 파일에서 공간을 할당할지, 아니면 MySQL이 `InnoDB` 온디스크 내부 임시 테이블을 사용할지 제어합니다. 기본 설정은 `temptable_use_mmap=ON`입니다.

참고

`temptable_use_mmap` 변수는 더 이상 사용되지 않으며, 향후 MySQL 버전에서는 이 변수에 대한 지원이 제거될 것으로 예상됩니다.

`temptable_max_mmap=0`으로 설정하는 것은 `temptable_use_mmap=OFF`로 설정하는 것과 동일합니다.

- TEMPTABLE_MAX_MMAP: MySQL이 InnoDB 온디스크 내부 임시 테이블을 사용하기 전에 임시 테이블 스토리지 엔진이 메모리 매핑된 파일에서 할당할 수 있는 최대 메모리 양을 설정합니다. 기본 설정은 1073741824 바이트(1GiB)입니다. 이 제한의 목적은 다음과 같습니다.
  메모리 매핑된 파일이 임시 디렉터리(tmpdir)에서 너무 많은 공간을 사용하는 위험을 해결합니다.
  temptable_max_mmap = 0은 메모리 매핑된 파일의 할당을 비활성화하여 temptable_use_mmap 설정에 관계없이 해당 파일의 사용을 효과적으로 비활성화합니다.

TempTable 스토리지 엔진의 메모리 매핑 파일 사용에는 이러한 규칙이 적용됩니다:

- 임시 파일은 tmpdir 변수로 정의된 디렉터리에 생성됩니다.
- 임시 파일은 만들어서 열면 즉시 삭제되므로 tmpdir 디렉터리에 표시되지 않습니다. 임시 파일이 차지하는 공간은
  임시 파일이 열려 있는 동안에는 운영 체제를 사용할 수 없습니다. 임시 파일이 다음과 같은 경우 공간이 회수됩니다.
  를 닫거나 mysqld 프로세스가 종료될 때입니다.
- 데이터는 RAM과 임시 파일 간, RAM 내 또는 임시 파일 간에 이동되지 않습니다.
- 다음에 정의된 한도 내에서 공간을 사용할 수 있게 되면 새 데이터가 RAM에 저장됩니다.
  TEMPTABLE_MAX_RAM. 그렇지 않으면 새 데이터가 임시 파일에 저장됩니다.
- 테이블 데이터의 일부가 임시 파일에 기록된 후 RAM에 여유 공간이 생기면 나머지 테이블 데이터도 RAM에 저장할 수 있습니다.

인메모리 임시 테이블에 MEMORY 스토리지 엔진을 사용하는 경우 (internal_tmp_mem_storage_engine=MEMORY), MySQL은 인메모리 임시 테이블이 너무 커지면 자동으로 온디스크 테이블로 변환합니다. 인메모리 임시 테이블의 최대 크기는 tmp_table_size 또는 max_heap_table_size 값 중 더 작은 값으로 정의됩니다. 이는 CREATE TABLE로 명시적으로 생성된 메모리 테이블과는 다릅니다. 이러한 테이블의 경우 max_heap_table_size 변수만 테이블이 커질 수 있는 크기를 결정하며, 온디스크 형식으로 변환하지 않습니다.

온디스크 내부 임시 테이블용 스토리지 엔진

MySQL 8.2는 온디스크 내부 임시 테이블에 InnoDB 스토리지 엔진만 사용합니다. (MYISAM 스토리지 엔진은 더 이상 이 용도로 지원되지 않습니다).

InnoDB 온디스크 내부 임시 테이블은 기본적으로 데이터 디렉터리에 있는 세션 임시 테이블 공간에 생성됩니다. 자세한 내용은 섹션 15.6.3.5, "임시 테이블 공간"을 참조하십시오.

내부 임시 테이블 저장 형식

메모리 내 내부 임시 테이블이 TempTable 스토리지 엔진에 의해 관리되는 경우, VARCHAR 열, VARBINARY 열 및 기타 이진 대형 객체 유형 열을 포함하는 행은 메모리에서 셀 배열로 표시되며, 각 셀에는 NULL 플래그, 데이터 길이 및 데이터 포인터가 포함됩니다. 열 값은 배열 뒤의 연속된 순서로 메모리의 단일

영역에 패딩 없이 배치됩니다. 배열의 각 셀은 16바이트의 저장 공간을 사용합니다. `TempTable` 스토리지 엔진이 메모리 매핑된 파일에서 공간을 할당할 때도 동일한 스토리지 형식이 적용됩니다.

메모리 내부 임시 테이블이 메모리 스토리지 엔진에 의해 관리되는 경우 고정 길이 행 형식이 사용됩니다. `VARCHAR` 및 `VARBINARY` 열 값은 최대 열 길이까지 덧대어져 사실상 `CHAR` 및 `BINARY` 열로 저장됩니다.

디스크의 내부 임시 테이블은 항상 `InnoDB`에서 관리합니다.

메모리 스토리지 엔진을 사용할 때 문은 처음에 메모리 내부 임시 테이블을 생성한 다음 테이블이 너무 커지면 디스크에 있는 테이블로 변환할 수 있습니다. 이러한 경우 변환을 건너뛰고 처음부터 디스크에 내부 임시 테이블을 생성하면 더 나은 성능을 얻을 수 있습니다. `big_tables` 변수를 사용하여 내부 임시 테이블을 디스크에 강제로 저장할 수 있습니다.

내부 임시 테이블 생성 모니터링

내부 임시 테이블이 메모리 또는 디스크에 생성되면 서버는 `Created_tmp_tables` 값을 증가시킵니다. 내부 임시 테이블이 디스크에 생성되면 서버는 `Created_tmp_disk_tables` 값을 증가시킵니다. 디스크에 너무 많은 내부 임시 테이블이 생성되는 경우 내부 임시 테이블 스토리지 엔진에 설명된 엔진별 제한을 조정하는 것이 좋습니다.

참고

알려진 제한 사항으로 인해 `Created_tmp_disk_tables`는 메모리 매핑 파일에 생성된 온디스크 임시 테이블을 계산하지 않습니다. 기본적으로 TempTable 스토리지 엔진 오버플로 메커니즘은 내부 임시 테이블을 생성합니다.
테이블을 메모리 매핑된 파일에 저장합니다. 내부 임시 테이블 스토리지 엔진을 참조하세요.

`memory/temptable/physical_ram` 및 `memory/temptable/physical_disk` 성능 스키마 계측기를 사용하여 메모리 및 디스크에서 `TempTable` 공간 할당을 모니터링할 수 있습니다. `memory/temptable/physical_ram`은 할당된 RAM의 양을 보고하고, `memory/temptable/physical_disk`는 메모리 매핑 파일이 TempTable 오버플로 메커니즘으로 사용될 때 디스크에서 할당된 공간의 양을 보고합니다. 물리적_디스크 계측기가 0 이외의 값을 보고하고 메모리 매핑 파일이 TempTable 오버플로 메커니즘으로 사용되는 경우, 어느 시점에 TempTable 메모리 제한에 도달한 것입니다. `memory_summary_global_by_event_name`과 같은 성능 스키마 메모리 요약 테이블에서 데이터를 쿼리할 수 있습니다. 섹션 27.12.20.10, "메모리 요약 테이블"을 참조하세요.

8.4.5 데이터베이스 및 테이블 수 제한

MySQL은 데이터베이스 수에 제한이 없습니다. 기본 파일 시스템에는 디렉터리 수에 제한이 있을 수 있습니다.

MySQL에는 테이블 수에 제한이 없습니다. 기본 파일 시스템에는 테이블을 나타내는 파일 수에 제한이 있을 수 있습니다. 개별 스토리지 엔진은 엔진별 제약 조건을 부과할 수 있습니다. `InnoDB`는 최대 40억 개의 테이블을 허용합니다.

8.4.6 테이블 제한 크기

MySQL 데이터베이스의 유효 최대 테이블 크기는 일반적으로 MySQL 내부 제한이 아니라 파일 크기에 대한 운영 체제 제약 조건에 따라 결정됩니다. 운영 체제 파일 크기 제한에 대한 최신 정보는 사용 중인 운영 체제별 설명서를 참조하세요.

Windows 사용자의 경우, FAT 및 VFAT(FAT32)는 MySQL과 함께 프로덕션 환경에서 사용하기에 적합하지 *않다는 점에* 유의하세요. 대신 NTFS를 사용하세요.

전체 테이블 오류가 발생하는 경우 몇 가지 이유가 있을 수 있습니다:

- 디스크가 꽉 찼을 수 있습니다.
- `InnoDB` 테이블을 사용 중이며 `InnoDB` 테이블 스페이스 파일에 공간이 부족합니다. 최대 테이블 스페이스 크기는 테이블의 최대 크기이기도 합니다. 테이블 공간 크기 제한에 대해서는 섹션 15.22, "InnoDB 제한"을 참조하십시오.

  일반적으로 테이블을 여러 개의 테이블 스페이스 파일로 분할하는 것은 크기가 1TB보다 큰 테이블에 권장됩니다.
- 운영 체제 파일 크기 제한에 도달했습니다. 예를 들어 최대 2GB 크기의 파일만 지원하는 운영 체제에서 `MyISAM` 테이블을 사용 중이며 데이터 파일 또는 인덱스 파일에 대해 이 제한에 도달한 경우입니다.
- `MyISAM` 테이블을 사용 중이며 테이블에 필요한 공간이 내부 포인터 크기에서 허용하는 공간을 초과합니다. `MyISAM`에서는 기본적으로 데이터 및 인덱스 파일을 최대 256TB까지 늘릴 수 있지만, 이 제한은 최대 허용 크기인 65,536TB(256^7 - 1바이트)까지 변경할 수 있습니다.

기본 제한보다 큰 MyISAM 테이블이 필요하고 운영 체제가 대용량 파일을 지원하는 경우 CREATE TABLE 문은 AVG_ROW_LENGTH 및 MAX_ROWS 옵션을 지원합니다. 섹션 13.1.20, "CREATE TABLE 문"을 참조하십시오. 서버는 이러한 옵션을 사용하여 허용할 테이블의 크기를 결정합니다.

포인터 크기가 기존 테이블에 비해 너무 작은 경우 ALTER TABLE로 옵션을 변경하여 테이블의 최대 허용 크기를 늘릴 수 있습니다. 섹션 13.1.9, "ALTER TABLE 문"을 참조하십시오.

```
ALTER TABLE tbl_name MAX_ROWS=1000000000 AVG_ROW_LENGTH=nnn;
```

이 경우 MySQL은 행 수만을 기준으로 필요한 공간을 최적화할 수 없으므로 BLOB 또는 TEXT 열이 있는 테이블에 대해서만 AVG_ROW_LENGTH를 지정해야 합니다.

MyISAM 테이블의 기본 크기 제한을 변경하려면 내부 행 포인터에 사용되는 바이트 수를 설정하는 myisam_data_pointer_size를 설정합니다. 이 값은 MAX_ROWS 옵션을 지정하지 않은 경우 새 테이블의 포인터 크기를 설정하는 데 사용됩니다. myisam_data_pointer_size의 값은 2에서 7 사이일 수 있습니다. 예를 들어 동적 저장소 형식을 사용하는 테이블의 경우 값이 4이면 최대 4GB까지, 값이 6이면 최대 256TB까지 테이블을 허용합니다. 고정 저장소 형식을 사용하는 테이블은 최대 데이터 길이가 더 큽니다. 저장소 형식 특성에 대한 자세한 내용은 16.2.3절. "MyISAM 테이블 저장소 형식"을 참조하십시오.

이 문을 사용하여 최대 데이터 및 인덱스 크기를 확인할 수 있습니다:

```
SHOW TABLE STATUS FROM db_name LIKE 'tbl_name';
```

myisamchk -dv /path/to/table-index-file을 사용할 수도 있습니다. 섹션 13.7.7, "SHOW 문" 또는 섹션 4.6.4, "myisamchk - MyISAM 테이블 유지 관리 유틸리티"를 참조하세요.

MyISAM 테이블의 파일 크기 제한을 해결하는 다른 방법은 다음과 같습니다:

- 큰 테이블이 읽기 전용인 경우 myisampack을 사용하여 압축할 수 있습니다. myisampack은 일반적으로 테이블을 50% 이상 압축하므로 사실상 훨씬 더 큰 테이블을 가질 수 있습니다. 또한 여러 테이블을 단일 테이블로 병합할 수도 있습니다. 섹션 4.6.6, "myisampack - 압축된 읽기 전용 MyISAM 테이블 생성"을 참조하세요.
- MySQL에는 단일 MERGE 테이블과 동일한 구조를 가진 MyISAM 테이블 컬렉션을 처리할 수 있는 MERGE 라이브러리가 포함되어 있습니다. 섹션 16.7, "MERGE 스토리지 엔진"을 참조하십시오.

- 메모리(힙) 스토리지 엔진을 사용 중이므로 이 경우 MAX_HEAP_TABLE_SIZE 시스템 변수. 섹션 5.1.8, "서버 시스템 변수"를 참조하세요.

8.4.7 테이블 열 수 및 행 제한 크기

이 섹션에서는 테이블의 열 수와 개별 행의 크기에 대한 제한에 대해 설명합니다.

- 열 개수 제한
- 행 크기 제한

열 개수 제한

MySQL에는 테이블당 4096개의 열이 하드 제한되어 있지만, 특정 테이블의 경우 실제 최대 열 수는 이보다 적을 수 있습니다. 정확한 열 제한은 여러 요인에 따라 달라집니다:

- 테이블의 최대 행 크기는 모든 열의 총 길이가 이 크기를 초과할 수 없으므로 열의 수(및 크기)를 제한합니다. 행 크기 제한을 참조하십시오.
- 개별 열의 저장소 요구 사항에 따라 주어진 최대 행 크기에 맞는 열의 수가 제한됩니다. 일부 데이터 유형의 저장소 요구 사항은 저장소 엔진, 저장소 형식 및 문자 집합과 같은 요인에 따라 달라집니다. 섹션 11.7, "데이터 유형 저장소 요구 사항"을 참조하세요.

- 스토리지 엔진은 테이블 열 수를 제한하는 추가 제한을 부과할 수 있습니다. 예를 들어 `InnoDB`는 테이블 당 1017개의 열로 제한됩니다. 15.22절, "InnoDB 제한"을 참조하세요. 다른 저장소 엔진에 대한 자세한 내용은 16장, *대체 저장소 엔진을* 참조하십시오.

- 함수 키 부분(섹션 13.1.15, "인덱스 생성 문" 참조)은 숨겨진 가상 생성 저장 칼럼으로 구현되므로 테이블 인덱스의 각 함수 키 부분은 테이블 총 칼럼 제한에 포함됩니다.

행 크기 제한

주어진 테이블의 최대 행 크기는 여러 요인에 의해 결정됩니다:

- 스토리지 엔진이 더 큰 행을 지원할 수 있는 경우에도 MySQL 테이블의 내부 표현에는 최대 행 크기 제한이 65,535바이트로 설정되어 있습니다. `BLOB` 및 `TEXT` 열은 콘텐츠가 나머지 행과 별도로 저장되므로 행 크기 제한에 9~12바이트만 기여합니다.

- 데이터베이스 페이지 내에 로컬로 저장된 데이터에 적용되는 `InnoDB` 테이블의 최대 행 크기는 4KB, 8KB, 16KB 및 32KB `innodb_page_size` 설정의 경우 페이지의 절반보다 약간 작습니다. 예를 들어, 기본 16KB `InnoDB` 페이지 크기의 경우 최대 행 크기는 8KB보다 약간 작습니다. 64KB 페이지의 경우 최대 행 크기는 16KB보다 약간 작습니다. 섹션 15.22, "InnoDB 제한"을 참조하십시오.

  가변 길이 열이 포함된 행이 `InnoDB` 최대 행 크기를 초과하는 경우 `InnoDB`는 행이 `InnoDB` 행 크기 제한에 맞을 때까지 가변 길이 열을 선택하여 외부 페이지 외부에 저장합니다. 페이지 외부에 저장되는 가변 길이 열에 대해 로컬로 저장되는 데이터의 양은 행 형식에 따라 다릅니다. 자세한 내용은 섹션 15.10, "InnoDB 행 형식"을 참조하세요.

- 저장소 형식에 따라 페이지 헤더 및 트레일러 데이터의 양이 다르므로 행에 사용할 수 있는 저장소의 양에 영향을 줍니다.

  - `InnoDB` 행 형식에 대한 자세한 내용은 섹션 15.10, "InnoDB 행 형식"을 참조하십시오.

  - `MyISAM` 저장소 형식에 대한 자세한 내용은 섹션 16.2.3, "MyISAM 테이블 저장소 형식"을 참조하세요.

행 크기 제한 예제

- MySQL 최대 행 크기 제한인 65,535바이트는 다음 `InnoDB` 및 `MyISAM` 예제에서 확인할 수 있습니다. 이 제한은 스토리지 엔진이 더 큰 행을 지원할 수 있더라도 스토리지 엔진에 관계없이 적용됩니다.

```
mysql> CREATE TABLE t (a VARCHAR(10000), b VARCHAR(10000),
       c VARCHAR(10000), d VARCHAR(10000), e VARCHAR(10000),
       f VARCHAR(10000), g VARCHAR(6000)) ENGINE=InnoDB CHARACTER SET latin1;
오류 1118 (42000): 행 크기가 너무 큽니다. 사용된 테이블 유형에 대한 최대 행 크기(BLOB
제외)는 65535입니다. 여기에는 스토리지 오버헤드가 포함되므로 설명서를 확인하세요. 일부 열
mysql> CREATE TABLE t (a VARCHAR(10000), b VARCHAR(10000),
       c VARCHAR(10000), d VARCHAR(10000), e VARCHAR(10000),
       f VARCHAR(10000), g VARCHAR(6000)) ENGINE=MyISAM CHARACTER SET latin1;
오류 1118 (42000): 행 크기가 너무 큽니다. 사용된 테이블 유형에 대한 최대 행 크기(BLOB
제외)는 65535입니다. 여기에는 스토리지 오버헤드가 포함되므로 설명서를 확인하세요. 일부 열
을 텍스트 또는 BLOB으로 변경해야 합니다.
```

다음 MyISAM 예제에서 열을 TEXT로 변경하면 65,535바이트 행 크기 제한을 피할 수 있으며, BLOB 및 TEXT 열이 행 크기에 9~12바이트만 기여하기 때문에 작업이 성공할 수 있습니다.

```
mysql> CREATE TABLE t (a VARCHAR(10000), b VARCHAR(10000),
       c VARCHAR(10000), d VARCHAR(10000), e VARCHAR(10000),
       f VARCHAR(10000), g TEXT(6000)) ENGINE=MyISAM CHARACTER SET latin1;
쿼리 확인, 영향을 받는 행 0개(0.02초)
```

열을 TEXT로 변경하면 MySQL 65,535바이트 행 크기 제한을 피할 수 있고, 가변 길이 열을 페이지 외부에 저장하면 InnoDB 행 크기 제한을 피할 수 있으므로 InnoDB 테이블에 대한 작업이 성공합니다.

```
mysql> CREATE TABLE t (a VARCHAR(10000), b VARCHAR(10000),
       c VARCHAR(10000), d VARCHAR(10000), e VARCHAR(10000),
       f VARCHAR(10000), g TEXT(6000)) ENGINE=InnoDB CHARACTER SET latin1;
쿼리 확인, 영향을 받는 행 0개(0.02초)
```

- 가변 길이 열의 저장소에는 길이 바이트가 포함되며, 이 바이트는 행 크기로 계산됩니다. 예를 들어, VARCHAR(255) 문자 집합 utf8mb3 열은 값의 길이를 저장하는 데 2바이트가 필요하므로 각 값은 최대 767바이트를 차지할 수 있습니다.

  테이블 t1을 생성하는 문은 열에 32,765 + 2바이트와 32,766 + 2바이트가 필요하므로 성공하며, 이는 최대 행 크기인 65,535바이트 내에 해당합니다:

  ```
  mysql> CREATE TABLE t1
         (c1 VARCHAR(32765) NOT NULL, c2 VARCHAR(32766) NOT NULL)
         ENGINE = InnoDB CHARACTER SET latin1;
  쿼리 확인, 영향을 받는 행 0개(0.02초)
  ```

  열 길이가 최대 길이인 65,535바이트 이내이지만 길이를 기록하는 데 2바이트가 추가로 필요하여 행 크기가 65,535바이트를 초과하므로 테이블 t2를 생성하는 문이 실패합니다:

  ```
  mysql> CREATE TABLE t2
         (c1 VARCHAR(65535) NOT NULL)
         엔진 = InnoDB 문자 집합 latin1;
  오류 1118 (42000): 행 크기가 너무 큽니다. 사용된 테이블 유형에 대한 최대 행 크기(BLOB
  제외)는 65535입니다. 여기에는 스토리지 오버헤드가 포함되므로 설명서를 확인하세요. 일부 열
  을 텍스트 또는 BLOB으로 변경해야 합니다.
  ```

  열 길이를 65,533 이하로 줄이면 문이 성공할 수 있습니다.

  ```
  mysql> CREATE TABLE t2
         (c1 VARCHAR(65533) NOT NULL)
         엔진 = InnoDB 문자 집합 latin1;
  쿼리 확인, 영향을 받는 행 0개(0.01초)
  ```

- MyISAM 테이블의 경우 NULL 열은 행에 추가 공간이 있어야 해당 값이 NULL인지 여부를 기록할 수 있습니다. 각 NULL 열은 가장 가까운 바이트 단위로 반올림하여 1비트를 추가로 사용합니다.

  MyISAM에서 가변 길이 열 길이 바이트에 필요한 공간 외에 NULL 열을 위한 공간이 필요하여 행 크기가 65,535바이트를 초과하므로 테이블 t3을 만드는 문이 실패합니다:

  ```
  mysql> CREATE TABLE t3
         (c1 VARCHAR(32765) NULL, c2 VARCHAR(32766) NULL)
         ENGINE = MyISAM CHARACTER SET latin1;
  오류 1118 (42000): 행 크기가 너무 큽니다. 사용된 테이블 유형에 대한 최대 행 크기(BLOB
  제외)는 65535입니다. 여기에는 스토리지 오버헤드가 포함되므로 설명서를 확인하세요. 일부 열
  을 텍스트 또는 BLOB으로 변경해야 합니다.
  ```

  InnoDB NULL 열 저장소에 대한 자세한 내용은 섹션 15.10, "InnoDB 행 형식"을 참조하십시오.

- InnoDB는 행 크기(데이터베이스 페이지 내에 로컬로 저장된 데이터의 경우)를 4KB, 8KB, 16KB 및 32KB innodb_page_size 설정의 경우 데이터베이스 페이지의 절반 미만으로, 64KB 페이지의 경우 16KB

미만으로 제한합니다.

정의된 열이 16KB InnoDB 페이지의 행 크기 제한을 초과하므로 테이블 t4를 만드는 문이 실패합니다.

```
mysql> CREATE TABLE t4 (
       c1       CHAR(255),c2       CHAR(255),c3
       CHAR(255),c4  CHAR(255),c5  CHAR(255),c6
       CHAR(255),c7  CHAR(255),c8  CHAR(255),c9
       CHAR(255),
```

```
        c10       CHAR(255),c11       CHAR(255),c12
        CHAR(255),c13  CHAR(255),c14  CHAR(255),c15
        CHAR(255),c16  CHAR(255),c17  CHAR(255),c18
        CHAR(255), c19 CHAR(255),c20 CHAR(255),c21
        CHAR(255), c22 CHAR(255),c23 CHAR(255),c24
        CHAR(255),c25  CHAR(255),c26  CHAR(255),c27
        CHAR(255), c28 CHAR(255),c29 CHAR(255),c30
        CHAR(255), c31 CHAR(255),c32 CHAR(255),c33
        CHAR(255)
        ) 엔진=InnoDB 행_형식=동적 기본 문자 집합 latin1;
```

오류 1118(42000): 행 크기가 너무 큽니다(> 8126). 일부 열을 TEXT 또는 BLOB로 변경하면 도움이 될 수 있습니다. 현재 행 형식에서는 접두사 0바이트의 BLOB가 인라인으로 저장됩니다.

8.5 InnoDB에 최적화 테이블

InnoDB는 안정성과 동시성이 중요한 프로덕션 데이터베이스에서 MySQL 고객이 일반적으로 사용하는 스토리지 엔진입니다. InnoDB는 MySQL의 기본 스토리지 엔진입니다. 이 섹션에서는 InnoDB 테이블에 대한 데이터베이스 작업을 최적화하는 방법을 설명합니다.

8.5.1 InnoDB용 스토리지 레이아웃 최적화 테이블

- 데이터가 안정된 크기에 도달하거나 테이블이 수십 또는 수백 메가바이트 증가하면 OPTIMIZE TABLE 문을 사용하여 테이블을 재구성하고 낭비되는 공간을 압축하는 것이 좋습니다. 재구성된 테이블은 전체 테이블 스캔을 수행하는 데 필요한 디스크 I/O가 줄어듭니다. 이는 인덱스 사용량 개선 또는 응용 프로그램 코드 조정과 같은 다른 기술이 실용적이지 않은 경우 성능을 개선할 수 있는 간단한 기술입니다.

  테이블 최적화는 테이블의 데이터 부분을 복사하고 인덱스를 다시 작성합니다. 인덱스 내 데이터 패킹이 개선되고 테이블 공간과 디스크 내 조각화가 줄어드는 이점이 있습니다. 이점은 각 테이블의 데이터에 따라 다릅니다. 일부 데이터에는 상당한 이점이 있지만 다른 데이터에는 이점이 없거나 다음에 테이블을 최적화할 때까지 시간이 지남에 따라 이점이 감소할 수 있습니다. 테이블이 크거나 재구축 중인 인덱스가 버퍼 풀에 맞지 않는 경우 이 작업이 느려질 수 있습니다. 테이블에 많은 데이터를 추가한 후 처음 실행하는 것이 나중에 실행하는 것보다 훨씬 느린 경우가 많습니다.

- InnoDB에서 긴 기본 키(긴 값을 가진 단일 열 또는 긴 복합 값을 형성하는 여러 열)를 사용하면 디스크 공간을 많이 낭비하게 됩니다. 기본 키 값
  은 동일한 행을 가리키는 모든 보조 인덱스 레코드에 중복됩니다. (참조
  섹션 15.6.2.1, "클러스터 및 보조 인덱스" 참조). 기본 키가 긴 경우 AUTO_INCREMENT 열을 기본 키로 생성하거나 전체 열 대신 긴 VARCHAR 열의 접두사를 인덱싱합니다.

- 가변 길이 문자열을 저장하거나 NULL 값이 많은 열에는 CHAR 대신 VARCHAR 데이터 유형을 사용합니다. CHAR(*N*) 열은 문자열이 더 짧거나 값이 NULL인 경우에도 데이터를 저장하는 데 항상 N자를 사용합니다. 테이블 크기가 작을수록 버퍼 풀에 더 잘 맞고 디스크 I/O를 줄일 수 있습니다.

  COMPACT 행 형식(기본 InnoDB 형식) 및 가변 길이 문자 집합(예: utf8mb4 또는 sjis)을 사용하는 경우 CHAR(*N*) 열은 가변 공간을 차지하지만 여전히 최소 N바이트를 차지합니다.

- 테이블이 크거나 반복적인 텍스트 또는 숫자 데이터가 많이 포함된 경우 압축 행 형식을 사용하는 것이 좋습니다. 데이터를 버퍼 풀로 가져오거나 전체 테이블 스캔을 수행하는 데 필요한 디스크 I/O가 줄어듭니다. 영구적인 결정을 내리기 전에 압축 행 형식과 압축 행 형식을 사용하여 달성할 수 있는 압축량을 측정해 보십시오.

8.5.2 InnoDB 트랜잭션 최적화 관리

InnoDB 트랜잭션 처리를 최적화하려면 트랜잭션 기능의 성능 오버헤드와 서버의 워크로드 간에 이상적인 균형을 찾아야 합니다. 예를 들어, 애플리케이션이 초당 수천 번 커밋하는 경우 성능 문제가 발생할 수 있고, 2~3 시간마다만 커밋하는 경우 다른 성능 문제가 발생할 수 있습니다.

- 기본 MySQL 설정인 `AUTOCOMMIT=1`은 사용량이 많은 데이터베이스 서버에 성능 제한을 가할 수 있습니다. 가능한 경우 여러 관련 데이터 변경 작업을 단일 트랜잭션으로 묶어두세요, `SET AUTOCOMMIT=0` 또는 `START TRANSCTION` 문을 실행한 후 `COMMIT`을 실행합니다. 명령문을 사용하여 모든 변경을 수행합니다.

  `InnoDB`는 트랜잭션이 데이터베이스를 수정한 경우 각 트랜잭션 커밋 시 로그를 디스크에 플러시해야 합니다. 각 변경 후에 커밋이 수행되면(기본 자동 커밋 설정과 마찬가지로) 저장 장치의 I/O 처리량이 초당 잠재적 작업 수에 제한을 두게 됩니다.

- 또는 단일 `SELECT` 문으로만 구성된 트랜잭션의 경우 자동 커밋을 켜면 InnoDB가 읽기 전용 트랜잭션을 인식하고 최적화하는 데 도움이 됩니다. 요구 사항은 섹션 8.5.3, "InnoDB 읽기 전용 트랜잭션 최적화"를 참조하십시오.

- 대량의 행을 삽입, 업데이트 또는 삭제한 후에는 롤백을 수행하지 마세요. 대규모 트랜잭션으로 인해 서버 성능이 저하되는 경우 롤백을 수행하면 문제가 악화되어 원래 데이터 변경 작업보다 몇 배 더 오래 걸릴 수 있습니다. 서버를 시작할 때 롤백이 다시 시작되므로 데이터베이스 프로세스를 종료해도 도움이 되지 않습니다.

  이 문제가 발생할 가능성을 최소화합니다:

  - 버퍼 풀의 크기를 늘려 모든 데이터 변경 사항이 디스크에 즉시 기록되지 않고 캐시될 수 있도록 합니다.
  - 삽입 작업 외에 업데이트 및 삭제 작업도 버퍼링되도록 `innodb_change_buffering=all`로 설정합니다.
  - 빅 데이터 변경 작업 중에 주기적으로 `COMMIT` 문을 실행하여 단일 삭제 또는 업데이트를 더 적은 수의 행에 대해 작동하는 여러 문으로 분할하는 것을 고려하세요.

  런어웨이 롤백이 발생하면 이를 제거하려면 롤백이 CPU에 바인딩되어 빠르게 실행되도록 버퍼 풀을 늘리거나, 15.18.2절 "InnoDB 복구"에 설명된 대로 서버를 종료하고 `innodb_force_recovery=3`으로 재시작합니다.

  이 문제는 업데이트 및 삭제 작업이 메모리에 캐시되어 처음에 더 빠르게 수행되고 필요한 경우 롤백도 더 빠르게 수행할 수 있도록 하는 기본 설정 `innodb_change_buffering=all`을 사용하면 드물게 발생할 것으로 예상됩니다. 삽입, 업데이트 또는 삭제가 많은 장기 실행 트랜잭션을 처리하는 서버에서는 이 매개변수 설정을 사용해야 합니다.

- 예기치 않은 종료가 발생하는 경우 최신 커밋된 트랜잭션 중 일부가 손실되는 것을 감당할 수 있는 경우 `innodb_flush_log_at_trx_commit` 매개 변수를 0으로 설정할 수 있습니다. InnoDB는 어쨌든 초당 한 번씩 로그 플러시를 시도하지만 플러시가 보장되는 것은 아닙니다.

- 행이 수정되거나 삭제되면 행과 관련 실행 취소 로그는 트랜잭션이 커밋된 직후 또는 트랜잭션이 완료된 직후에도 물리적으로 제거되지 않습니다. 이전 데이터는 더 일찍 또는 동시에 시작된 트랜잭션이 완료될 때까지

보존되므로 해당 트랜잭션은 수정 또는 삭제된 행의 이전 상태에 액세스할 수 있습니다. 따라서 장기간 실행되는 트랜잭션은 다른 트랜잭션에 의해 변경된 데이터를 InnoDB가 제거하지 못하게 할 수 있습니다.

- 장기 실행 트랜잭션 내에서 행이 수정되거나 삭제되면, 읽기 커밋 및 반복 읽기 격리 수준을 사용하는 다른 트랜잭션이 동일한 행을 읽을 경우 이전 데이터를 재구성하기 위해 더 많은 작업을 수행해야 합니다.
- 장기 실행 트랜잭션이 테이블을 수정하는 경우, 다른 트랜잭션에서 해당 테이블에 대한 쿼리는 커버링 인덱스 기법을 사용하지 않습니다. 일반적으로 보조 인덱스에서 모든 결과 열을 검색할 수 있는 쿼리는 대신 테이블 데이터에서 적절한 값을 조회합니다.

  보조 인덱스 페이지에 너무 새로운 PAGE_MAX_TRX_ID가 있거나 보조 인덱스의 레코드가 삭제 표시되어 있는 경우, InnoDB는 클러스터된 인덱스를 사용하여 레코드를 조회해야 할 수 있습니다.

8.5.3 InnoDB 읽기 전용 트랜잭션 최적화

InnoDB는 읽기 전용으로 알려진 트랜잭션에 대해 트랜잭션 ID(TRX_ID 필드)를 설정하는 것과 관련된 오버헤드를 피할 수 있습니다. 트랜잭션 ID는 쓰기 작업을 수행할 수 있는 트랜잭션이나 SELECT ... FOR UPDATE와 같이 읽기 잠금을 수행할 수 있는 트랜잭션에만 필요합니다. 불필요한 트랜잭션 ID를 제거하면 쿼리 또는 데이터 변경 문이 읽기 뷰를 구성할 때마다 참조되는 내부 데이터 구조의 크기를 줄일 수 있습니다.

InnoDB는 다음과 같은 경우 읽기 전용 트랜잭션을 감지합니다:

- 트랜잭션은 START TRANSACTION READ ONLY 문으로 시작됩니다. 이 경우 데이터베이스(InnoDB, MyISAM 또는 기타 유형의 테이블)를 변경하려고 시도하면 오류가 발생하고 트랜잭션은 읽기 전용 상태로 계속됩니다:

  ```
  오류 1792 (25006): 읽기 전용 트랜잭션에서 문을 실행할 수 없습니다.
  ```

  읽기 전용 트랜잭션에서 세션별 임시 테이블을 변경하거나 해당 변경 및 잠금이 다른 트랜잭션에 표시되지 않으므로 해당 테이블에 대한 잠금 쿼리를 계속 실행할 수 있습니다.

- 자동 커밋 설정이 켜져 있으므로 트랜잭션이 단일 문으로 보장되고 트랜잭션을 구성하는 단일 문은 "비잠금" SELECT 문입니다. 즉, FOR UPDATE 또는 LOCK IN 공유 모드 절을 사용하지 않는 SELECT입니다.

- 트랜잭션이 읽기 전용 옵션 없이 시작되었지만 아직 행을 명시적으로 잠그는 업데이트나 문이 실행되지 않았습니다. 업데이트 또는 명시적 잠금이 필요할 때까지 트랜잭션은 읽기 전용 모드로 유지됩니다.

따라서 보고서 생성기와 같이 읽기 집약적인 애플리케이션의 경우, 일련의 InnoDB 쿼리를 START TRANSACTION 읽기 전용 및 커밋 내에서 그룹화하거나, SELECT 문을 실행하기 전에 자동 커밋 설정을 켜거나, 쿼리에 산재되어 있는 데이터 변경 문을 피하는 방식으로 쿼리 시퀀스를 조정할 수 있습니다.

트랜잭션 시작 및 자동 커밋에 대한 자세한 내용은 섹션 13.3.1, "트랜잭션 시작, 커밋 및 롤백 문"을 참조하세요.

참고

자동 커밋, 비잠금, 읽기 전용(AC-NL- RO)에 해당하는 트랜잭션은 특정 내부 InnoDB 데이터 구조에서 제외되므로 엔진 INNODB 상태 표시 출력에 나열되지 않습니다.

8.5.4 InnoDB 재실행 최적화 로깅

재실행 로깅을 최적화하려면 다음 가이드라인을 고려하세요:

- 재실행 로그 파일의 크기를 늘립니다. InnoDB가 재실행 로그 파일을 가득 채우면 버퍼 풀의 수정된 내용을 체크포인트에 있는 디스크에 기록해야 합니다. 재실행 로그 파일이 작으면 불필요한 디스크 쓰기가 많이 발생합니다.

재실행 로그 파일 크기는 innodb_redo_log_capacity에 의해 결정됩니다. InnoDB는 동일한 크기의 재실행 로그 파일을 32개 유지하려고 시도하며, 각 파일은 1/32 * innodb_redo_log_capacity와 같습니다. 따라서 innodb_redo_log_capacity 설정을 변경하면 재실행 로그 파일의 크기가 변경됩니다.

재실행 로그 파일 구성 수정에 대한 자세한 내용은 15.6.5절. "재실행 로그"를 참조하세요.

- 로그 버퍼의 크기를 늘리는 것을 고려하세요. 로그 버퍼가 크면 트랜잭션이 커밋되기 전에 로그를 디스크에 쓸 필요 없이 대용량 트랜잭션을 실행할 수 있습니다. 따라서 많은 행을 업데이트, 삽입 또는 삭제하는 트랜잭션이 있는 경우 로그 버퍼를 크게 만들면 디스크 I/O를 절약할 수 있습니다. 로그 버퍼 크기는 MySQL 8.2에서 동적으로 구성할 수 있는 innodb_log_buffer_size 구성 옵션을 사용하여 구성합니다.

- "읽기-온-쓰기"를 방지하려면 `innodb_log_write_ahead_size` 구성 옵션을 구성합니다. 이 옵션은 재실행 로그의 쓰기 미리 쓰기 블록 크기를 정의합니다. 운영 체제 또는 파일 시스템 캐시 블록 크기와 일치하도록 `innodb_log_write_ahead_size`를 설정합니다. 읽기-온-쓰기는 재실행 로그의 쓰기-앞서 블록 크기와 운영 체제 또는 파일 시스템 캐시 블록 크기가 일치하지 않아 재실행 로그 블록이 운영 체제 또는 파일 시스템에 완전히 캐시되지 않을 때 발생합니다.

  `innodb_log_write_ahead_size`의 유효한 값은 `InnoDB` 로그 파일 블록 크기(2^n)의 배수입니다. 최소값은 `InnoDB` 로그 파일 블록 크기(512)입니다. 최소값을 지정하면 미리 쓰기가 수행되지 않습니다. 최대값은 `innodb_page_size` 값과 같습니다. `innodb_log_write_ahead_size` 값을 `innodb_page_size` 값보다 큰 값으로지정하면 `innodb_log_write_ahead_size` 설정이 `innodb_page_size` 값으로 잘립니다.

  운영 체제 또는 파일 시스템 캐시 블록 크기에 비해 `innodb_log_write_ahead_size` 값을 너무 낮게 설정하면 읽기-온-쓰기가 발생합니다. 값을 너무 높게 설정하면 한 번에 여러 블록이 쓰여지기 때문에 로그 파일 쓰기에 대한 `fsync` 성능에 약간의 영향을 미칠 수 있습니다.

- MySQL은 로그 버퍼에서 시스템 버퍼로 재실행 로그 레코드를 쓰고 시스템 버퍼를 재실행 로그 파일로 플러시하기 위한 전용 로그 라이터 스레드를 제공합니다. `innodb_log_writer_threads` 변수를 사용하여 로그 작성자 스레드를 활성화 또는 비활성화할 수 있습니다. 전용 로그 작성자 스레드는 동시성이 높은 시스템에서 성능을 향상시킬 수 있지만, 동시성이 낮은 시스템에서는 전용 로그 작성자 스레드를 비활성화하면 더 나은 성능을 제공합니다.

- 플러시 재실행을 기다리는 사용자 스레드의 스핀 지연 사용을 최적화합니다. 스핀 지연은 지연 시간을 줄이는 데 도움이 됩니다. 동시성이 낮은 기간에는 지연 시간을 줄이는 것이 우선 순위가 낮을 수 있으며, 이러한 기간에는 스핀 지연을 사용하지 않으면 에너지 소비를 줄일 수 있습니다. 동시성이 높은 기간에는 스핀 지연에 처리 능력을 소모하지 않도록 하여 다른 작업에 사용할 수 있도록 하는 것이 좋습니다. 다음 시스템 변수를 사용하면 스핀 지연 사용의 경계를 정의하는 워터마크 값의 높고 낮은 값을 설정할 수 있습니다.

  - `innodb_log_wait_for_flush_spin_hwm`: 플러시 재실행을 기다리는 동안 사용자 스레드가 더 이상 회전하지 않는 최대 평균 로그 플러시 시간을 정의합니다. 기본값은 400마이크로초입니다.

  - `innodb_log_spin_cpu_abs_lwm`: 플러시 재실행을 기다리는 동안 사용자 스레드가 더 이상 회전하지 않는 최소 CPU 사용량을 정의합니다. 이 값은 CPU 코어 사용량의 합으로 표현됩니다. 예를 들어 기본값인 80은 단일 CPU 코어의 80%입니다. 멀티 코어 프로세서가 있는 시스템에서 값 150은 하나의 CPU 코어 사용량 100%에 두 번째 CPU 코어 사용량 50%를 더한 값입니다.

  - `innodb_log_spin_cpu_pct_hwm`: 플러시 재실행을 기다리는 동안 사용자 스레드가 더 이상 회전하지 않는 최대 CPU 사용량을 정의합니다. 이 값은 모든 CPU 코어의 총 처리 능력을 합한 값의 백분율로 표시됩니다. 기본값은 50%입니다. 예를 들어 CPU 코어 2개 사용 100%는 CPU 코어가 4개인 서버에서 CPU 코어를 합친 처리 능력의 50%입니다.

innodb_log_spin_cpu_pct_hwm 구성 옵션은 프로세서 선호도를 고려합니다. 예를 들어 서버에 48개의 코어가 있지만 mysqld 프로세스가 4개의 CPU 코어에만 고정되어 있는 경우, 나머지 44개의 CPU 코어는 무시됩니다.

8.5.5 InnoDB용 대량 데이터 로드 테이블

이러한 성능 팁은 섹션 8.2.5.1, '삽입 문 최적화'의 빠른 삽입에 대한 일반 지침을 보완합니다.

- InnoDB로 데이터를 가져올 때 자동 커밋 모드는 모든 삽입에 대해 디스크에 로그 플러시를 수행하므로 자동 커밋 모드를 끄십시오. 가져오기 작업 중에 자동 커밋을 비활성화하려면 SET 자동 커밋 및 COMMIT 문으로 둘러싸세요:

```
SET 자동 커밋=0;
```

```
... SQL 가져오기 문 ...
커밋;
```

mysqldump 옵션 --opt는 덤프 파일을 설정 자동 커밋 및 커밋 문으로 래핑하지 않고도 InnoDB 테이블로 빠르게 가져올 수 있는 덤프 파일을 생성합니다.

- 보조 키에 고유 제약 조건이 있는 경우 가져오기 세션 중에 일시적으로 고유성 검사를 해제하여 테이블 가져오기 속도를 높일 수 있습니다:

```
SET 고유_체크=0;
... SQL 가져오기 문 ...
고유_체크=1로 설정합니다;
```

큰 테이블의 경우, InnoDB가 변경 버퍼를 사용하여 보조 인덱스 레코드를 일괄적으로 쓸 수 있으므로 디스크 I/O를 많이 절약할 수 있습니다. 데이터에 중복 키가 포함되어 있지 않은지 확인하세요.

- 테이블에 외래 키 제약 조건이 있는 경우 가져오기 세션 기간 동안 외래 키 검사를 해제하여 테이블 가져오기 속도를 높일 수 있습니다:

```
SET foreign_key_checks=0;
... SQL 가져오기 문 ...
외국어_키_체크=1로 설정합니다;
```

큰 테이블의 경우 이렇게 하면 디스크 I/O를 많이 절약할 수 있습니다.

- 많은 행을 삽입해야 하는 경우 다중 행 INSERT 구문을 사용하면 클라이언트와 서버 간의 통신 오버헤드를 줄일 수 있습니다:

```
INSERT INTO yourtable VALUES (1,2), (5,5), ...;
```

이 팁은 InnoDB 테이블뿐만 아니라 모든 테이블에 삽입할 때 유효합니다.

- 자동 증가 열이 있는 테이블에 대량 삽입을 수행할 때는 innodb_autoinc_lock_mode를 1(연속) 대신 2(인터리브)로 설정합니다. 자세한 내용은 15.6.1.6절, "InnoDB에서 자동 증가 처리"를 참조하십시오.

- 대량 삽입을 수행할 때는 PRIMARY KEY 순서로 행을 삽입하는 것이 더 빠릅니다. InnoDB 테이블은 클러스터된 인덱스를 사용하므로 PRIMARY KEY 순서대로 데이터를 사용하는 것이 상대적으로 빠릅니다. 버퍼 풀에 완전히 들어가지 않는 테이블의 경우 PRIMARY KEY 순서로 대량 삽입을 수행하는 것이 특히 중요합니다.

- InnoDB 전체 텍스트 인덱스에 데이터를 로드할 때 최적의 성능을 얻으려면 다음 단계를 따르세요:

  1. 테이블 생성 시 FTS_DOC_ID_INDEX라는 고유 인덱스가 있는 BIGINT UNSIGNED NOT NULL 유형의 열 FTS_DOC_ID를 정의합니다. 예를 들어

```
CREATE TABLE t1 (
FTS_DOC_ID BIGINT 부호 없는 NOT NULL AUTO_INCREMENT,
title varchar(255) NOT NULL DEFAULT '',
text mediumtext NOT NULL,
기본 키 (`FTS_DOC_ID`)
) 엔진=InnoDB 기본 문자 집합=utf8mb4;
t1(FTS_DOC_ID)에 고유 인덱스 FTS_DOC_ID_INDEX를 생성합니다;
```

2. 테이블에 데이터를 로드합니다.

3. 데이터가 로드된 후 전체 텍스트 인덱스를 생성합니다.

참고

테이블 생성 시 `FTS_DOC_ID` 칼럼을 추가할 때는 `INSERT` 또는 `UPDATE`를 수행할 때마다 `FTS_DOC_ID`가 단조롭게 증가해야 하므로 `FULLTEXT` 인덱싱된 칼럼이 업데이트될 때 `FTS_DOC_ID` 칼럼이 업데이트되도록 합니다. 테이블 생성 시 `FTS_DOC_ID`를 추가하지 않도록 선택한 경우

로 설정하고 InnoDB가 DOC ID를 관리하도록 하면, InnoDB는 다음 CREATE FULLTEXT INDEX 호출 시 FTS_DOC_ID를 숨겨진 열로 추가합니다. 그러나 이 접근 방식은 테이블을 다시 빌드해야 하므로 성능에 영향을 줄 수 있습니다.

- *새* MySQL 인스턴스에 데이터를 로드하는 경우 ALTER INSTANCE를 사용하여 재실행 로깅을 비활성화하는 것이 좋습니다.
  {활성화|비활성화} INNODB REDO_LOG 구문. 재실행 로깅을 비활성화하면 재실행 로그 쓰기를 방지하여 데이터 로딩 속도를 높일 수 있습니다. 자세한 내용은 재실행 로깅 비활성화를 참조하십시오.

경고

이 기능은 새 MySQL 인스턴스에 데이터를 로드할 때만 사용할 수 있습니다. *프로덕션 시스템에서 재실행 로깅을 비활성화하지 마세요.* 재실행 로깅이 비활성화된 상태에서 서버를 종료했다가 다시 시작할 수는 있지만, 재실행 로깅이 비활성화된 상태에서 예기치 않게 서버가 중단되면 데이터 손실 및 인스턴스 손상이 발생할 수 있습니다.

- MySQL Shell을 사용하여 데이터 가져오기. MySQL Shell의 병렬 테이블 가져오기 유틸리티 util.importTable()은 대용량 데이터 파일에 대해 MySQL 관계형 테이블로 데이터를 빠르게 가져올 수 있습니다. MySQL Shell의 덤프 로드 유틸리티 util.loadDump()는 병렬 로드 기능도 제공합니다. MySQL 셸 유틸리티를 참조하세요.

8.5.6 InnoDB 최적화 쿼리

InnoDB 테이블에 대한 쿼리를 조정하려면 각 테이블에 적절한 인덱스 집합을 만듭니다. 자세한 내용은 섹션 8.3.1, 'MySQL에서 인덱스를 사용하는 방법'을 참조하세요. InnoDB 인덱스에 대한 다음 지침을 따르세요:

- 각 InnoDB 테이블에는 기본 키가 있으므로(요청 여부와 관계없이) 각 테이블에 대해 가장 중요하고 시간이 중요한 쿼리에 사용되는 기본 키 열 집합을 지정합니다.
- 기본 키에 너무 많거나 너무 긴 열을 지정하지 마세요. 이러한 열 값은 각 보조 인덱스에 중복되므로 기본 키에 너무 많은 열을 지정하지 마세요. 인덱스에 불필요한 데이터가 포함되어 있으면 이 데이터를 읽기 위한 I/O와 이를 캐시하기 위한 메모리가 서버의 성능과 확장성을 저하시킵니다.
- 각 쿼리는 하나의 인덱스만 사용할 수 있으므로 각 열에 대해 별도의 보조 인덱스를 만들지 마세요. 거의 테스트하지 않는 열이나 서로 다른 값이 몇 개만 있는 열에 대한 인덱스는 쿼리에 도움이 되지 않을 수 있습니다. 동일한 테이블에 대한 쿼리가 많고 열의 다양한 조합을 테스트하는 경우 많은 수의 단일 열 인덱스보다는 적은 수의 연결된 인덱스를 만드세요. 인덱스에 결과 집합에 필요한 모든 열이 포함된 경우(커버링 인덱스라고 함) 쿼리에서 테이블 데이터를 전혀 읽지 않을 수 있습니다.
- 인덱싱된 열에 NULL 값을 포함할 수 없는 경우 테이블을 만들 때 NOT NULL로 선언합니다. 옵티마이저는 각 열에 NULL 값이 포함되어 있는지 여부를 알면 쿼리에 가장 효과적으로 사용할 인덱스를 더 잘 결정할 수

있습니다.

- 섹션 8.5.3, 'InnoDB 읽기 전용 트랜잭션 최적화'에 나와 있는 기술을 사용하여 `InnoDB` 테이블에 대한 단일 쿼리 트랜잭션을 최적화할 수 있습니다.

8.5.7 InnoDB DDL 최적화 운영

- 테이블 및 인덱스에 대한 많은 DDL 작업(`CREATE`, `ALTER` 및 `DROP` 문)을 온라인으로 수행할 수 있습니다. 자세한 내용은 섹션 15.12, "InnoDB 및 온라인 DDL"을 참조하십시오.
- 보조 인덱스 추가를 위한 온라인 DDL 지원은 일반적으로 보조 인덱스 없이 테이블을 생성한 다음 데이터가 로드된 후 보조 인덱스를 추가하여 테이블 및 관련 인덱스를 생성하고 로드하는 프로세스의 속도를 높일 수 있다는 의미입니다.
- 테이블을 비우려면 `DELETE FROM tbl_name`이 아니라 `TRUNCATE TABLE`을 사용하십시오. 외래 키 제약 조건으로 인해 `TRUNCATE` 문이 일반 `DELETE` 문처럼 작동할 수 있으며, 이 경우 `DROP TABLE` 및 `CREATE TABLE`과 같은 명령 시퀀스가 가장 빠를 수 있습니다.
- 기본 키는 각 `InnoDB` 테이블의 스토리지 레이아웃에 필수적이며 기본 키의 정의를 변경하면 전체 테이블을 재구성해야 하므로 항상 기본 키를 다음과 같이 설정합니다.

부분으로 변경하고 미리 계획하여 ALTER 또는 DROP
를 기본 키로 설정합니다.

8.5.8 InnoDB 디스크 최적화 I/O

데이터베이스 설계 모범 사례와 SQL 작업 튜닝 기술을 따르고 있지만 디스크 I/O 활동이 많아 데이터베이스 속도가 여전히 느린 경우 다음과 같은 디스크 I/O 최적화를 고려하세요. Unix 최상위 도구 또는 Windows 작업 관리자에서 워크로드의 CPU 사용률이 70% 미만으로 표시되는 경우 워크로드가 디스크에 묶여 있을 가능성이 높습니다.

- 버퍼 풀 크기 늘리기

  테이블 데이터가 InnoDB 버퍼 풀에 캐시되어 있으면 디스크 I/O 없이 쿼리에서 반복적으로 액세스할 수 있습니다. 버퍼 풀의 크기를 지정하려면 innodb_buffer_pool_size 옵션. 이 메모리 영역은 일반적으로 시스템 메모리의 50~75%로 구성하는 것이 권장될 정도로 중요합니다. 자세한 내용은 섹션 8.12.3.1, "MySQL이 메모리를 사용하는 방법"을 참조하십시오.

- 플러시 방법 조정

  일부 버전의 GNU/Linux 및 Unix에서는 (InnoDB가 기본적으로 사용하는) Unix fsync() 호출 및 유사한 메서드를 사용하여 파일을 디스크로 플러시하는 속도가 놀라울 정도로 느립니다. 데이터베이스 쓰기 성능이 문제가 되는 경우, innodb_flush_method 매개변수를 O_DSYNC로 설정하여 벤치마크를 수행하세요.

- 운영 체제 플러시에 대한 임계값 구성

  기본적으로 InnoDB가 새 로그 파일 또는 테이블 스페이스 파일과 같은 새 데이터 파일을 만들면 파일이 디스크에 플러시되기 전에 운영 체제 캐시에 완전히 쓰여지므로 한 번에 많은 양의 디스크 쓰기 작업이 발생할 수 있습니다. 운영 체제 캐시에서 데이터를 더 작고 주기적으로 플러시하도록 하려면 innodb_fsync_threshold 변수를 사용하여
  임계값(바이트 단위)을 입력합니다. 바이트 임계값에 도달하면 운영 체제의 콘텐츠가
  캐시가 디스크에 플러시됩니다. 기본값 0을 사용하면 파일이 캐시에 완전히 쓰여진 후에만 데이터를 디스크에 플러시하는 기본 동작이 강제로 적용됩니다.

  임계값을 지정하여 더 작은 규모의 주기적 플러시를 강제하는 것은 여러 MySQL 인스턴스가 동일한 저장 장치를 사용하는 경우에 유용할 수 있습니다. 예를 들어, 새 MySQL 인스턴스와 관련 데이터 파일을 생성하면 디스크 쓰기 활동이 급증하여 동일한 저장 장치를 사용하는 다른 MySQL 인스턴스의 성능이 저하될 수 있습니다. 임계값을 구성하면 이러한 쓰기 활동의 급증을 방지하는 데 도움이 됩니다.

- fsync() 대신 fdatasync() 사용

  fdatasync() 시스템 호출을 지원하는 플랫폼에서 innodb_use_fdatasync 변수를 사용하면 운영

체제 플러시에 `fsync()` 대신 `fdatasync()` 사용을 허용할 수 있습니다. 후속 데이터 검색에 필요한 경우가 아니라면 `fdatasync()` 시스템 호출은 파일 메타데이터의 변경 사항을 플러시하지 않으므로 잠재적인 성능 이점을 제공합니다.

`fsync`, `O_DSYNC` 및 `O_DIRECT`와 같은 `innodb_flush_method` 설정의 하위 집합은 `fsync()` 시스템 호출을 사용합니다. 이러한 설정을 사용할 때 `innodb_use_fdatasync` 변수를 사용할 수 있습니다.

- Linux에서 네이티브 AIO와 함께 noop 또는 데드라인 I/O 스케줄러 사용

  `InnoDB`는 Linux의 비동기 I/O 하위 시스템(기본 AIO)을 사용하여 데이터 파일 페이지에 대한 읽기 및 쓰기 요청을 수행합니다. 이 동작은 기본적으로 활성화되어 있는 `innodb_use_native_aio` 구성 옵션에 의해 제어됩니다. 네이티브 AIO를 사용하면 I/O 스케줄러의 유형이
  가 I/O 성능에 더 큰 영향을 미칩니다. 일반적으로 noop 및 데드라인 I/O 스케줄러가 권장됩니다. 벤치마크를 수행하여 워크로드 및 환경에 가장 적합한 결과를 제공하는 I/O 스케줄러를 결정하세요. 자세한 내용은 섹션 15.8.6, "Linux에서 비동기 I/O 사용"을 참조하세요.

- x86_64 아키텍처용 Solaris 10에서 직접 I/O 사용

  x86_64 아키텍처용 Solaris 10(AMD 옵테론)에서 `InnoDB` 스토리지 엔진을 사용하는 경우, `InnoDB` 성능 저하를 방지하려면 InnoDB 관련 파일에 직접 I/O를 사용하세요. `InnoDB` 관련 파일을 저장하는 데 사용되는 전체 UFS 파일 시스템에 직접 I/O를 사용하려면, 강제 디렉터리 옵션을 사용하여 마운트 합니다(`mount_ufs(1M)` 참조). (Solaris 10/x86_64의 기본값은 사용하지 않는 것입니다.
  이 옵션) 전체 파일 시스템이 아닌 `InnoDB` 파일 작업에만 직접 I/O를 적용하려면 `innodb_flush_method = O_DIRECT`를 설정합니다. 이 설정을 사용하면 `InnoDB`는 데이터 파일에 대한 I/O(로그 파일에 대한 I/O가 아님)에 대해 `fcntl()` 대신 `directio()`를 호출합니다.

- Solaris 2.6 이상에서 데이터 및 로그 파일에 원시 스토리지 사용

  Solaris 2.6 이상 릴리스 및 모든 플랫폼(sparc/x86/x64/amd64)에서 `innodb_buffer_pool_size` 값이 큰 `InnoDB` 스토리지 엔진을 사용하는 경우, 앞서 설명한 대로 `forcedirectio` 마운트 옵션을 사용하여 원시 장치 또는 별도의 직접 I/O UFS 파일 시스템에서 `InnoDB` 데이터 파일 및 로그 파일로 벤치마크를 수행합니다. (로그 파일에 대한 직접 I/O를 원하는 경우 `innodb_flush_method`를 설정하지 않고 마운트 옵션을 사용해야 합니다.) 베리타스 파일 시스템 VxFS 사용자는 `convosync=direct` mount 옵션을 사용해야 합니다.

  `MyISAM` 테이블용 파일과 같은 다른 MySQL 데이터 파일을 직접 I/O 파일 시스템에 배치하지 마세요. 실행 파일이나 라이브러리는 직접 I/O 파일 시스템에 배치해서는 *안 됩니다*.

- 추가 저장 장치 사용

  추가 저장 장치를 사용하여 RAID 구성을 설정할 수 있습니다. 관련 정보는 섹션 8.12.1, "디스크 I/O 최적화하기"를 참조하세요.

  또는 `InnoDB` 테이블스페이스 데이터 파일과 로그 파일을 다른 물리적 디스크에 배치할 수도 있습니다. 자세한 내용은 다음 섹션을 참조하세요:

  - 섹션 15.8.1, "InnoDB 시동 구성"
  - 섹션 15.6.1.2, "외부에서 테이블 생성"
  - 일반 테이블 스페이스 만들기
  - 섹션 15.6.1.4, "InnoDB 테이블 이동 또는 복사"

- 비회전식 스토리지 고려

  일반적으로 비회전식 스토리지는 무작위 I/O 작업에 더 나은 성능을 제공하며, 회전식 스토리지는 순차적 I/O 작업에 더 적합합니다. 데이터 및 로그 파일을 회전형 스토리지에 분산할 경우

및 비회전식 저장 장치에서 각 파일에서 주로 수행되는 I/O 작업 유형을 고려하세요.

랜덤 I/O 지향 파일에는 일반적으로 테이블별 파일 및 일반 테이블 스페이스 데이터 파일, 실행 취소 테이블 스페이스 파일, 임시 테이블 스페이스 파일이 포함됩니다. 순차적 I/O 지향 파일에는 `InnoDB` 시스템 테이블스페이스 파일, 이중 쓰기 파일, 바이너리 로그 파일 및 재실행 로그 파일과 같은 로그 파일이 포함됩니다.

비회전식 스토리지를 사용하는 경우 다음 구성 옵션에 대한 설정을 검토합니다:

- `innodb_checksum_algorithm`

  `crc32` 옵션은 더 빠른 체크섬 알고리즘을 사용하며 빠른 스토리지 시스템에 권장됩니다.

- `innodb_flush_neighbors`

  회전식 스토리지 장치에 대해 I/O를 최적화합니다. 비회전식 스토리지 또는 회전식 스토리지와 비회전식 스토리지가 혼합된 경우 비활성화합니다. 기본적으로 비활성화되어 있습니다.

- `INNODB_IDLE_FLUSH_PCT`

  유휴 기간 동안 페이지 플러시를 제한할 수 있으므로 비회전식 저장 장치의 수명을 연장하는 데 도움이 됩니다.

- `innodb_io_capacity`

  일반적으로 저사양 비회전식 스토리지 장치에는 기본 설정인 200으로 충분합니다. 고급형 버스 연결 장치의 경우 1000과 같이 더 높은 설정을 고려하세요.

- `INNODB_IO_CAPACITY_MAX`

  기본값인 2000은 비회전식 스토리지를 사용하는 워크로드를 위한 것입니다. 고급 버스 연결형 비회전식 스토리지 장치의 경우 2500과 같이 더 높은 설정을 고려하세요.

- `innodb_log_compressed_pages`

  재실행 로그가 회전되지 않는 저장소에 있는 경우 이 옵션을 비활성화하여 로깅을 줄이는 것이 좋습니다. 압축 페이지의 로깅 비활성화를 참조하세요.

- `innodb_log_file_size`(더 이상 사용되지 않음)

  재실행 로그가 비회전식 스토리지에 있는 경우 이 옵션을 구성하여 캐싱 및 쓰기 결합을 최대화합니다.

- `innodb_redo_log_capacity`

  재실행 로그가 비회전식 스토리지에 있는 경우 이 옵션을 구성하여 캐싱 및 쓰기 결합을 최대화합니다.

- `innodb_page_size`

  디스크의 내부 섹터 크기와 일치하는 페이지 크기를 사용하는 것이 좋습니다. 초기 세대 SSD 장치의 섹

터 크기는 4KB인 경우가 많습니다. 일부 최신 장치에는 16KB 섹터 크기가 있습니다. 섹터 크기는

기본 InnoDB 페이지 크기는 16KB입니다. 페이지 크기를 저장 장치 블록 크기에 가깝게 유지하면 디스크에 다시 쓰여지는 변경되지 않은 데이터의 양을 최소화할 수 있습니다.

- 빈로그_행_이미지

  이진 로그가 비회전 스토리지에 있고 모든 테이블에 기본 키가 있는 경우 로깅을 줄이려면 이 옵션을 최소로 설정하는 것이 좋습니다.

운영 체제에서 TRIM 지원이 활성화되어 있는지 확인합니다. 일반적으로 기본적으로 활성화되어 있습니다.

- 백로그를 방지하기 위해 I/O 용량 증가

  InnoDB 체크포인트 작업으로 인해 처리량이 주기적으로 감소하는 경우, innodb_io_capacity 구성 옵션의 값을 늘리는 것을 고려하세요. 값이 높을수록 플러싱이 더 자주 수행되어 처리량 저하를 유발할 수 있는 작업 백로그를 방지할 수 있습니다.

- 플러싱이 뒤처지지 않는 경우 I/O 용량 감소

  InnoDB 플러싱 작업으로 인해 시스템이 뒤처지지 않는다면, innodb_io_capacity 구성 옵션의 값을 낮추는 것을 고려하세요. 일반적으로 이 옵션 값은 가능한 한 낮게 유지하되, 이전 단락에서 언급한 것처럼 주기적으로 처리량이 떨어질 정도로 낮게 설정하지 않습니다. 옵션 값을 낮출 수 있는 일반적인 시나리오에서는 엔진 INNODB 상태 표시의 출력에서 다음과 같은 조합을 볼 수 있습니다:

  - 히스토리 목록 길이가 수천 개 미만으로 짧습니다.
  - 삽입된 행에 가깝게 버퍼 병합을 삽입합니다.
  - 버퍼 풀의 수정된 페이지가 지속적으로 버퍼 풀의 innodb_max_dirty_pages_pct보다 훨씬 낮습니다. (서버가 대량 삽입을 수행하지 않을 때 측정합니다. 대량 삽입 중에는 수정된 페이지 비율이 크게 상승하는 것이 정상입니다.)
  - 로그 시퀀스 번호 - 마지막 체크포인트는 7/8 미만 또는 이상적으로는 InnoDB 로그 파일 총 크기의 6/8 미만입니다.

- 시스템 테이블스페이스 파일을 Fusion-io 장치에 저장하기

  원자 쓰기를 지원하는 Fusion-io 장치에 이중 쓰기 저장 영역이 포함된 파일을 저장하면 이중 쓰기 버퍼 관련 I/O 최적화의 이점을 활용할 수 있습니다. (이중 쓰기 버퍼 저장 영역은 이중 쓰기 파일에 있습니다. 섹션 15.6.4, "이중 쓰기 버퍼") 이중 쓰기 저장 영역 파일을 원자 쓰기를 지원하는 Fusion-io 장치에 배치하면 이중 쓰기 버퍼가 자동으로 비활성화되고 모든 데이터 파일에 Fusion-io 원자 쓰기가 사용됩니다. 이 기능은 Fusion-io 하드웨어에서만 지원되며 Linux의 Fusion-io NVMFS에서만 활성화됩니다. 이 기능을 최대한 활용하려면 innodb_flush_method를 O_DIRECT로 설정하는 것이 좋습니다.

참고

이중 쓰기 버퍼 설정은 전역 설정이므로 Fusion-io 하드웨어에 상주하지 않는 데이터 파일에 대해서도 이중 쓰기 버퍼가 비활성화됩니다.

- 압축 페이지 로깅 비활성화

  `InnoDB` 테이블 압축 기능을 사용하는 경우 압축된 데이터가 변경되면 재압축된 페이지의 이미지가 재실행 로그에 기록됩니다. 이 동작은 복구 중에 다른 버전의 `zlib` 압축 알고리즘을 사용할 경우 발생할 수 있는 손상을 방지하기 위해 기본적으로 활성화되어 있는 `innodb_log_compressed_pages`에 의해 제어됩니다. 다음과 같은 경우 `zlib` 버전이 변경되지 않았는지 확인하려면 `innodb_log_compressed_pages`를 비활성화하십시오. 를 사용하여 압축 데이터를 수정하는 워크로드에 대한 재실행 로그 생성을 줄일 수 있습니다.

8.5.9 InnoDB 구성 최적화 변수

부하가 가볍고 예측 가능한 서버와 항상 최대 용량에 가깝게 실행되거나 활동이 급증하는 서버에는 서로 다른 설정이 가장 적합합니다.

InnoDB 스토리지 엔진은 많은 최적화를 자동으로 수행하기 때문에 많은 성능 튜닝 작업에는 데이터베이스가 제대로 작동하는지 모니터링하고 성능이 저하될 경우 구성 옵션을 변경하는 작업이 포함됩니다. 자세한 InnoDB 성능 모니터링에 대한 자세한 내용은 15.16절, "MySQL 성능 스키마와 InnoDB 통합"을 참조하세요.

수행할 수 있는 주요 구성 단계는 다음과 같습니다:

- 빈번한 소규모 디스크 쓰기를 방지하기 위해 InnoDB가 변경된 데이터를 버퍼링하는 데이터 변경 작업의 유형을 제어합니다. 변경 버퍼링 구성을 참조하십시오. 기본값은 모든 유형의 데이터 변경 작업을 버퍼링하는 것이므로 버퍼링 양을 줄여야 하는 경우에만 이 설정을 변경하세요.
- innodb_adaptive_hash_index 옵션을 사용하여 적응형 해시 인덱싱 기능을 켜고 끕니다. 자세한 내용은 섹션 15.5.3, "적응형 해시 인덱스"를 참조하세요. 비정상적인 활동 기간 동안 이 설정을 변경한 다음 원래 설정으로 복원할 수 있습니다.
- 컨텍스트 전환이 병목 현상인 경우 InnoDB가 처리하는 동시 스레드 수에 제한을 설정합니다. 15.8.4절, "InnoDB의 스레드 동시성 구성"을 참조하세요.
- InnoDB가 읽기 전 작업으로 수행하는 프리페칭의 양을 제어합니다. 시스템에 사용되지 않는 I/O 용량이 있는 경우, 읽기 전 작업이 많으면 쿼리 성능이 향상될 수 있습니다. 너무 많은 읽기를 수행하면 부하가 많은 시스템에서 주기적으로 성능이 저하될 수 있습니다. 15.8.3.4절, "InnoDB 버퍼 풀 프리페칭 구성(읽기 앞서)"을 참조하세요.
- 기본값으로 완전히 활용되지 않는 고급 I/O 하위 시스템이 있는 경우 읽기 또는 쓰기 작업에 대한 백그라운드 스레드 수를 늘립니다. 15.8.5절, "백그라운드 InnoDB I/O 스레드 수 구성"을 참조하세요.
- 백그라운드에서 InnoDB가 수행하는 I/O의 양을 제어합니다. 섹션 15.8.7, "InnoDB I/O 용량 구성"을 참조하세요. 주기적인 성능 저하가 관찰되는 경우 이 설정을 축소할 수 있습니다.
- InnoDB가 특정 유형의 백그라운드 쓰기를 수행하는 시기를 결정하는 알고리즘을 제어합니다. 섹션 15.8.3.5, "버퍼 풀 플러싱 구성"을 참조하십시오. 이 알고리즘은 일부 유형의 워크로드에서는 작동하지만 다른 워크로드에서는 작동하지 않으므로 주기적인 성능 저하가 관찰되는 경우 이 기능을 비활성화할 수 있습니다.
- 멀티코어 프로세서와 캐시 메모리 구성을 활용하여 컨텍스트 전환의 지연을 최소화합니다. 섹션 15.8.8, "스핀 잠금 폴링 구성"을 참조하십시오.
- 테이블 스캔과 같은 일회성 작업이 InnoDB 버퍼 캐시에 저장된 자주 액세스하는 데이터를 방해하지 않

도록 방지합니다. 섹션 15.8.3.3, "버퍼 풀 스캔 방지 기능 만들기"를 참조하십시오.

- 로그 파일을 안정성과 충돌 복구에 적합한 크기로 조정합니다. `InnoDB` 로그 파일은 충돌 후 시작 시간이 길어지는 것을 방지하기 위해 작게 유지되는 경우가 많습니다. MySQL에 도입된 최적화 5.5는 충돌 복구 프로세스의 특정 단계 속도를 높입니다. 특히 메모리 관리 알고리즘이 개선되어 재실행 로그 스캔과 재실행 로그 적용이 더 빨라졌습니다. 긴 시작 시간을 피하기 위해 로그 파일을 인위적으로 작게 유지했다면 이제 로그 파일 크기를 늘려 재실행 로그 레코드의 재활용으로 인해 발생하는 I/O를 줄이는 것을 고려할 수 있습니다.

- 특히 멀티 기가바이트 버퍼 풀이 있는 시스템에서는 `InnoDB` 버퍼 풀의 크기와 인스턴스 수를 구성하는 것이 중요합니다. 15.8.3.2절, "다중 버퍼 풀 인스턴스 구성"을 참조하세요.

- 최대 동시 트랜잭션 수를 늘리면 가장 바쁜 데이터베이스의 확장성이 크게 향상됩니다. 섹션 15.6.6, "로그 실행 취소"를 참조하세요.

- 퍼지 작업(가비지 컬렉션의 일종)을 백그라운드 스레드로 이동합니다. 참조 섹션 15.8.9, "퍼지 구성". 이 설정의 결과를 효과적으로 측정하려면 먼저 다른 I/O 관련 및 스레드 관련 구성 설정을 조정하세요.

- `InnoDB가` 동시 스레드 간에 수행하는 스위칭 양을 줄여 사용량이 많은 서버에서 SQL 작업이 대기 열에 올라 '트래픽 체증'을 형성하지 않도록 합니다. 값을 설정합니다.
  옵션의 값을 최대 32(고성능 최신 시스템의 경우 약 32까지)로 설정합니다.
  `innodb_concurrency_tickets` 옵션의 값을 일반적으로 5000 정도로 늘립니다. 이 옵션 조합은 `InnoDB가` 한 번에 처리하는 스레드 수에 상한선을 설정하고 각 스레드가 교체되기 전에 상당한 작업을 수행할 수 있도록 하여 대기 중인 스레드 수를 낮게 유지하고 과도한 컨텍스트 전환 없이 작업을 완료할 수 있도록 합니다.

8.5.10 테이블이 많은 시스템을 위한 InnoDB 최적화

- 비영구 옵티마이저 통계(기본 구성이 아닌 구성)를 구성한 경우 `InnoDB는` 테이블에 해당 값을 저장하는 대신 시작 후 해당 테이블에 처음 액세스할 때 테이블의 인덱스 카디널리티 값을 계산합니다. 이 단계는 데이터를 여러 테이블로 분할하는 시스템에서 상당한 시간이 걸릴 수 있습니다. 이 오버헤드는 초기 테이블 열기 작업에만 적용되므로 나중에 사용할 수 있도록 테이블을 '워밍업'하려면 시작 직후에 `SELECT와` 같은 문을 실행하여 테이블에 액세스합니다.
  `1 FROM tbl_name LIMIT 1`.

  최적화 프로그램 통계는 기본적으로 디스크에 지속되며, `innodb_stats_persistent` 구성 옵션으로 활성화할 수 있습니다. 영구 옵티마이저 통계에 대한 자세한 내용은 15.8.10.1절. "영구 옵티마이저 통계 매개변수 구성"을 참조하십시오.

8.6 MyISAM에 최적화 테이블

테이블 잠금은 동시 업데이트를 수행하는 기능을 제한하기 때문에 `MyISAM` 스토리지 엔진은 읽기 위주의 데이터 또는 동시성이 낮은 작업에서 가장 잘 작동합니다. MySQL에서는 `MyISAM이` 아닌 `InnoDB가` 기본 스토리지 엔진입니다.

8.6.1 MyISAM 최적화 쿼리

`MyISAM` 테이블의 쿼리 속도를 높이기 위한 몇 가지 일반적인 팁입니다:

- MySQL이 쿼리를 더 잘 최적화하도록 하려면 테이블에 데이터가 로드된 후 테이블을 분석하거나 `myisamchk --analyze를` 실행합니다. 이렇게 하면 동일한 값을 가진 행의 평균 수를 나타내는 각 인덱스 부분의 값이 업데이트됩니다. (고유 인덱스의 경우 이 값은 항상 1입니다.) MySQL은 이 값을 사용하여 비상수 표현식을 기반으로 두 테이블을 조인할 때 어떤 인덱스를 선택할지 결정합니다. 테이블 분석 결과는 `SHOW INDEX FROM tbl_name을` 사용하여 카디널리티 값을 검사하여 확인할 수 있습니다. `myisamchk --`

`description --verbose`는 인덱스 분포 정보를 표시합니다.

- 인덱스에 따라 인덱스와 데이터를 정렬하려면 `myisamchk --sort-index --sort- records=1`을 사용합니다(인덱스 1을 기준으로 정렬한다고 가정). 이 방법은 인덱스에 따라 모든 행을 순서대로 읽으려는 고유 인덱스가 있는 경우 쿼리를 더 빠르게 만드는 좋은 방법입니다. 이 방법으로 큰 테이블을 처음 정렬할 때는 시간이 오래 걸릴 수 있습니다.

- 자주 업데이트되는 `MyISAM` 테이블에서는 복잡한 `SELECT` 쿼리를 피하여 읽기와 쓰기 간의 경합으로 인해 발생하는 테이블 잠금 문제를 방지하세요.

- `MyISAM`은 동시 삽입을 지원합니다: 테이블에 데이터 파일 중간에 빈 블록이 없는 경우 다른 스레드가 테이블에서 읽는 동시에 테이블에 새 행을 삽입할 수 있습니다. 이 작업을 수행하는 것이 중요하다면 행 삭제를 피하는 방식으로 테이블을 사용하는 것이 좋습니다. 또 다른 가능성은 테이블에서 많은 행을 삭제한 후 테이블 최적화를 실행하여 테이블 조각 모음을 수행하는 것입니다. 이 동작은 `concurrent_insert` 변수를 설정하여 변경할 수 있습니다. 행이 삭제된 테이블에서도 새 행을 강제로 추가(따라서 동시 삽입 허용)할 수 있습니다. 섹션 8.11.3, "동시 삽입"을 참조하십시오.

- 자주 변경되는 MyISAM 테이블의 경우 모든 가변 길이 열(VARCHAR, BLOB 및 TEXT)을 사용하지 않도록 하십시오. 테이블에 가변 길이 열이 하나라도 포함된 경우 테이블은 동적 행 형식을 사용합니다. 16장, *대체 저장소 엔진을* 참조하십시오.

- 일반적으로 행이 커진다고 해서 테이블을 다른 테이블로 분할하는 것은 유용하지 않습니다. 행에 액세스할 때 성능에 가장 큰 타격을 주는 것은 행의 첫 바이트를 찾는 데 필요한 디스크 검색입니다. 데이터를 찾은 후 대부분의 최신 디스크는 대부분의 애플리케이션에서 전체 행을 충분히 빠르게 읽을 수 있습니다. 테이블을 분할하여 상당한 차이를 만드는 유일한 경우는 고정 행 크기로 변경할 수 있는 동적 행 형식을 사용하는 MyISAM 테이블이거나 테이블을 자주 스캔해야 하지만 대부분의 열은 필요하지 않은 경우입니다. 16장, *대체 저장소 엔진을* 참조하십시오.

- 일반적으로 expr1의 행을 검색하는 경우 ALTER TABLE ... ORDER BY *expr1*, *expr2*, ... 를 사용하면 일반적으로 *expr1*, *expr2*, ...의 순서로 행을 검색합니다. 테이블을 광범위하게 변경한 후 이 옵션을 사용하면 더 높은 성능을 얻을 수 있습니다.

- 많은 행의 정보를 기반으로 카운트와 같은 결과를 자주 계산해야 하는 경우 새 테이블을 도입하고 카운터를 실시간으로 업데이트하는 것이 더 좋을 수 있습니다. 다음 형식의 업데이트는 매우 빠릅니다:

  ```
  UPDATE tbl_name SET count_col=count_col+1 WHERE key_col=상수;
  ```

  이는 테이블 수준 잠금(단일 작성자와 다중 독자)만 있는 MyISAM과 같은 MySQL 스토리지 엔진을 사용할 때 매우 중요합니다. 이 경우 행 잠금 관리자가 해야 할 일이 적기 때문에 대부분의 데이터베이스 시스템에서 더 나은 성능을 제공합니다.

- 동적 형식의 MyISAM 테이블로 조각화를 방지하려면 주기적으로 테이블 최적화를 사용하세요. 섹션 16.2.3, "MyISAM 테이블 저장소 형식"을 참조하십시오.

- DELAY_KEY_WRITE=1 테이블 옵션으로 MyISAM 테이블을 선언하면 테이블이 닫힐 때까지 인덱스 업데이트가 디스크에 플러시되지 않으므로 인덱스 업데이트가 더 빨라집니다. 단점은 이러한 테이블이 열려 있는 동안 무언가에 의해 서버가 종료되는 경우, 서버를 다시 시작하기 전에 myisam_recover_options 시스템 변수를 설정한 상태로 서버를 실행하거나 myisamchk를 실행하여 테이블이 정상인지 확인해야 한다는 것입니다. (단, 이 경우에도 데이터 행에서 키 정보가 항상 생성될 수 있으므로 DELAY_KEY_WRITE를 사용하여 아무 것도 잃지 않아야 합니다.)

- 문자열은 MyISAM 인덱스에서 자동으로 접두사 및 끝 공백으로 압축됩니다. 섹션 13.1.15, "인덱스 생성 문"을 참조하십시오.

- 애플리케이션에서 쿼리 또는 답변을 캐싱한 다음 많은 삽입 또는 업데이트를 함께 실행하여 성능을 향상시킬 수 있습니다. 이 작업 중에 테이블을 잠그면 모든 업데이트 후에 인덱스 캐시가 한 번만 플러시됩니다.

8.6.2 MyISAM용 대량 데이터 로드 테이블

이러한 성능 팁은 섹션 8.2.5.1, '삽입 문 최적화'의 빠른 삽입에 대한 일반 지침을 보완합니다.

- `MyISAM` 테이블의 경우 데이터 파일 중간에 삭제된 행이 없는 경우 동시 삽입을 사용하여 `SELECT` 문이 실행되는 동안 동시에 행을 추가할 수 있습니다. 섹션 8.11.3, "동시 삽입"을 참조하십시오.

- 몇 가지 추가 작업을 수행하면 테이블에 인덱스가 많은 경우 `MyISAM` 테이블에 대해 데이터 로드 작업을 더 빠르게 실행할 수 있습니다. 다음 절차를 따르세요:

  1. `FLUSH TABLES` 문 또는 `mysqladmin flush-tables` 명령을 실행합니다.

  2. `myisamchk --keys-used=0 -rq` /path/to/db/tbl_name을 사용하여 테이블에 대한 모든 인덱스 사용을 제거합니다.

  3. 데이터 로드를 사용하여 테이블에 데이터를 삽입합니다. 이 방법은 인덱스를 업데이트하지 않으므로 매우 빠릅니다.

4. 나중에 테이블에서 읽기만 하려는 경우 myisampack을 사용하여 압축하세요. 섹션 16.2.3.3, "압축된 테이블 특성"을 참조하세요.

5. `myisamchk -rq /path/to/db/tbl_name`으로 인덱스를 다시 생성합니다. 이렇게 하면 디스크에 쓰기 전에 인덱스 트리가 메모리에 생성되므로, 많은 디스크 찾기를 피할 수 있으므로 데이터를 로드하는 동안 인덱스를 업데이트하는 것보다 훨씬 빠릅니다. 결과 인덱스 트리도 완벽하게 균형을 이룹니다.

6. `FLUSH TABLES` 문 또는 `mysqladmin flush-tables` 명령을 실행합니다.

데이터를 삽입할 `MyISAM` 테이블이 비어 있는 경우 `LOAD DATA`는 앞의 최적화를 자동으로 수행합니다. 자동 최적화와 프로시저를 명시적으로 사용하는 것의 주요 차이점은 서버가 `LOAD DATA` 문을 실행할 때 인덱스 재생성을 위해 할당할 수 있는 것보다 훨씬 더 많은 임시 메모리를 인덱스 생성에 할당할수 있다는 점입니다.

`myisamchk` 대신 다음 문을 사용하여 `MyISAM` 테이블의 고유하지 않은 인덱스를 사용하지 않도록 설정하거나 활성화할 수도 있습니다. 이러한 문을 사용하는 경우 `FLUSH TABLES` 작업을 건너뛸 수 있습니다:

```
ALTER TABLE tbl_name DISABLE KEYS;
ALTER TABLE tbl_name ENABLE KEYS;
```

- 비트랜잭션 테이블에 대해 여러 문을 사용하여 수행되는 `INSERT` 작업의 속도를 높이려면 테이블을 잠급니다:

```
잠금 테이블을 쓰기로 설정합니다;
INSERT INTO a VALUES (1,23),(2,34),(4,33);
INSERT INTO a VALUES (8,26),(6,29);
...
테이블 잠금 해제;
```

이렇게 하면 모든 `INSERT` 문이 완료된 후 인덱스 버퍼가 디스크에 한 번만 플러시되므로 성능이 향상됩니다. 일반적으로 인덱스 버퍼 플러시는 `INSERT` 문 수만큼 많이 수행됩니다. 단일 `INSERT` 문으로 모든 행을 삽입할 수 있는 경우 명시적 잠금 문이 필요하지 않습니다.

잠금을 사용하면 다중 연결 테스트의 총 시간이 단축되지만 개별 연결의 최대 대기 시간은 잠금을 기다리기 때문에 늘어날 수 있습니다. 다음과 같이 5개의 클라이언트가 동시에 삽입을 시도한다고 가정해 보겠습니다:

- 연결 1은 1000개의 삽입을 수행합니다.
- 연결 2, 3, 4는 삽입 1회를 수행합니다.
- 연결 5는 1000개의 삽입을 수행합니다.

잠금을 사용하지 않으면 연결 2, 3, 4가 연결 1, 5보다 먼저 완료됩니다. 잠금을 사용하는 경우 연결 2, 3, 4가 연결 1 또는 5보다 먼저 완료되지 않을 수 있지만 총 시간은 약 40% 더 빨라질 것입니다.

MySQL에서 `INSERT`, `UPDATE`, `DELETE` 작업은 매우 빠르지만 약 5회 이상 연속적으로 삽입 또는 업데이

트하는 모든 작업에 잠금을 추가하면 전반적인 성능을 향상시킬 수 있습니다. 연속 삽입을 매우 많이 수행하는 경우 다른 스레드가 테이블에 액세스할 수 있도록 가끔씩(각각 1,000행 정도) 테이블 잠금을 수행한 다음 테이블 잠금 해제를 수행할 수 있습니다.
이렇게 해도 성능은 여전히 향상됩니다.

방금 설명한 전략을 사용하더라도 INSERT는 여전히 데이터를 로드하는 데 `LOAD DATA`보다 훨씬 느립니다.

- MyISAM 테이블의 성능을 높이려면 데이터 로드 및 삽입 모두에 대해 key_buffer_size 시스템 변수를 늘려 키 캐시를 확대합니다. 섹션 5.1.1, "서버 구성"을 참조하십시오.

8.6.3 수리 테이블 최적화 문

MyISAM 테이블에 대한 복구 테이블은 복구 작업에 myisamchk를 사용하는 것과 유사하며 일부 동일한 성능 최적화가 적용됩니다:

- myisamchk에는 메모리 할당을 제어하는 변수가 있습니다. 섹션 4.6.4.6, "myisamchk 메모리 사용량"에 설명된 대로 이러한 변수를 설정하여 성능을 개선할 수 있습니다.

- REPAIR TABLE의 경우 동일한 원칙이 적용되지만 서버에서 복구를 수행하므로 myisamchk 변수 대신 서버 시스템 변수를 설정합니다. 또한 메모리 할당 변수를 설정하는 것 외에도 myisam_max_sort_file_size 시스템 변수를 늘리면 복구가 더 빠른 파일 정렬 방법을 사용할 가능성이 높아지고 키 캐시 방법으로 복구하는 느린 방법을 피할 수 있습니다. 테이블 파일 사본을 저장할 수 있는 충분한 여유 공간이 있는지 확인한 후 이 변수를 시스템의 최대 파일 크기로 설정합니다. 사용 가능한 공간은 원본 테이블 파일이 포함된 파일 시스템에서 사용할 수 있어야 합니다.

다음 옵션을 사용하여 메모리 할당 변수를 설정하는 myisamchk 테이블 복구 작업을 수행한다고 가정합니다:

```
--key_buffer_size=128M --myisam_sort_buffer_size=256M
--read_buffer_size=64M --write_buffer_size=64M
```

이러한 myisamchk 변수 중 일부는 서버 시스템 변수에 해당합니다:

myisamchk 변수	시스템 변수
키_버퍼_크기	키_버퍼_크기
MYISAM_SORT_BUFFER_SIZE	MYISAM_SORT_BUFFER_SIZE
READ_BUFFER_SIZE	READ_BUFFER_SIZE
WRITE_BUFFER_SIZE	없음

각 서버 시스템 변수는 런타임에 설정할 수 있으며, 일부 서버 시스템 변수(myisam_sort_buffer_size, read_buffer_size)에는 전역 값 외에 세션 값도 있습니다. 세션 값을 설정하면 현재 세션에만 변경 효과가 제한되며 다른 사용자에게는 영향을 주지 않습니다. 전역 전용 변수(key_buffer_size, myisam_max_sort_file_size)를 변경하면 다른 사용자에게도 영향을 줍니다. key_buffer_size의 경우 버퍼가
는 해당 사용자와 공유됩니다. 예를 들어 myisamchk key_buffer_size 변수를 128MB로 설정한 경우, 다른 세션의 활동에서 키 버퍼를 사용할 수 있도록 해당 key_buffer_size 시스템 변수를 이보다 크게 설정할 수 있습니다(아직 더 크게 설정되지 않은 경우). 그러나,
전역 키 버퍼 크기를 변경하면 버퍼가 무효화되어 디스크 I/O가 증가하고 다른 세션의 속도가 느려집니다.
이 문제를 방지하는 대안은 별도의 키 캐시를 사용하고, 별도의 키 캐시를 할당하는 것입니다.

에 복구할 테이블의 인덱스를 할당하고 복구가 완료되면 할당을 해제합니다. 섹션 8.10.2.2, "다중 키 캐시"를 참조하세요.

앞의 설명에 따라 다음과 같이 `REPAIR TABLE` 작업을 수행하여 `myisamchk` 명령과 유사한 설정을 사용할 수 있습니다. 여기서는 별도의 128MB 키 버퍼가 할당되고 파일 시스템이 최소 100GB의 파일 크기를 허용하는 것으로 가정합니다.

```
SET SESSION myisam_sort_buffer_size = 256*1024*1024;
SET SESSION read_buffer_size = 64*1024*1024;
SET GLOBAL myisam_max_sort_file_size = 100*1024*1024*1024;
SET GLOBAL repair_cache.key_buffer_size = 128*1024*1024;
CACHE INDEX tbl_name IN repair_cache;
인덱스를 캐시에 로드합니다;
```

```
REPAIR TABLE tbl_name ;
GLOBAL repair_cache.key_buffer_size = 0으로 설정합니다;
```

전역 변수를 변경하려고 하지만 다른 사용자에게 미치는 영향을 최소화하기 위해 테이블 복구 작업 기간 동안만 변경하려는 경우 해당 값을 사용자 변수에 저장하고 나중에 복원합니다. 예를 들어

```
SET @old_myisam_sort_buffer_size = @@GLOBAL.myisam_max_sort_파일_size;
SET GLOBAL myisam_max_sort_파일_size = 100*1024*1024*1024;
REPAIR TABLE tbl_name ;
SET GLOBAL myisam_max_sort_파일_size = @old_myisam_max_sort_파일_size;
```

기본적으로 값을 적용하려면 서버를 시작할 때 REPAIR TABLE에 영향을 주는 시스템 변수를 전역적으로 설정할 수 있습니다. 예를 들어 서버 my.cnf 파일에 다음 줄을 추가합니다:

```
[mysqld]
myisam_sort_buffer_size=256M
key_buffer_size=1G
myisam_max_sort_file_size=100G
```

이 설정에는 read_buffer_size가 포함되지 않습니다. 전역적으로 read_buffer_size를 큰 값으로 설정하면 모든 세션에 적용되므로 동시 세션이 많은 서버의 경우 과도한 메모리 할당으로 인해 성능이 저하될 수 있습니다.

8.7 메모리 최적화 표

자주 액세스하고 읽기 전용이거나 거의 업데이트되지 않는 중요하지 않은 데이터에는 메모리 테이블을 사용하는 것을 고려하세요. 실제 워크로드에서 애플리케이션을 동급의 InnoDB 또는 MyISAM 테이블과 벤치마크하여 추가 성능이 데이터 손실 위험이나 애플리케이션 시작 시 디스크 기반 테이블에서 데이터를 복사하는 오버헤드를 감수할 만한 가치가 있는지 확인합니다.

메모리 테이블에서 최상의 성능을 얻으려면 각 테이블에 대한 쿼리 종류를 검토하고 각 관련 인덱스에 사용할 유형(B 트리 인덱스 또는 해시 인덱스)을 지정합니다. CREATE INDEX 문에서 USING BTREE 또는 USING HASH 절을 사용합니다. B-트리 인덱스는 > 또는 BETWEEN과 같은 연산자를 통해 보다 크거나 보다 작은 비교를 수행하는 쿼리에 대해 빠릅니다. 해시 인덱스는 = 연산자를통해 단일 값을 조회하거나 IN 연산자를 통해 제한된 값 집합을 조회하는 쿼리에만 빠릅니다. BTREE를 사용하는 것이 기본값인 HASH를 사용하는 것보다 더 나은 선택인 이유에 대해서는 8.2.1.23절 "전체 테이블 스캔 피하기"를 참조하세요. 다양한 유형의 메모리 인덱스 구현에 대한 자세한 내용은 8.3.9절, "B-Tree와 해시 인덱스 비교"를 참조하세요.

8.8 쿼리 실행 이해 플랜

테이블, 열, 인덱스의 세부 사항 및 WHERE 절의 조건에 따라 MySQL 최적화 도구는 SQL 쿼리와 관련된 조회를 효율적으로 수행하기 위해 여러 가지 기술을 고려합니다. 거대한 테이블에 대한 쿼리는 모든 행을 읽지 않고도 수행할 수 있으며, 여러 테이블을 포함하는 조인은 모든 행의 조합을 비교하지 않고도 수행할 수 있습니다. 최적화 프로그램이 가장 효율적인 쿼리를 수행하기 위해 선택하는 작업 집합을 "쿼리 실행 계획"이라고 하며,

EXPLAIN 계획이라고도 합니다. 여러분의 목표는 쿼리가 잘 최적화되었음을 나타내는 EXPLAIN 계획의 측면을 인식하고, 비효율적인 연산이 발견되는 경우 계획을 개선하기 위한 SQL 구문 및 인덱싱 기술을 학습하는 것입니다.

8.8.1 설명으로 쿼리 최적화하기

EXPLAIN 문은 MySQL이 문을 실행하는 방식에 대한 정보를 제공합니다:

- EXPLAIN은 SELECT, DELETE, INSERT, REPLACE 및 UPDATE 문과 함께 작동합니다.
- 설명 가능한 문과 함께 EXPLAIN을 사용하는 경우 MySQL은 최적화 프로그램에서 문 실행 계획에 대한 정보를 표시합니다. 즉, 테이블이 조인되는 방법 및 순서에 대한 정보를 포함하여 MySQL이 문을 처리하는 방법을 설명합니다. 정보

EXPLAIN을 사용하여 실행 계획 정보를 얻는 방법에 대한 자세한 내용은 8.8.2절. "EXPLAIN 출력 형식"을 참조하세요.

- 설명 가능한 문이 아닌 `FOR CONNECTION connection_id`와 함께 EXPLAIN을 사용하면 명명된 연결에서 실행되는 문의 실행 계획을 표시합니다. 섹션 8.8.4, "명명된 연결에 대한 실행 계획 정보 얻기"를 참조하십시오.
- `SELECT` 문의 경우 `EXPLAIN`은 `SHOW` 경고를 사용하여 표시할 수 있는 추가 실행 계획 정보를 생성합니다. 섹션 8.8.3, "확장 EXPLAIN 출력 형식"을 참조한다.
- 설명은 분할된 테이블과 관련된 쿼리를 검사하는 데 유용합니다. 섹션 24.3.5, "파티션에 대한 정보 얻기"를 참조하십시오.
- `형식` 옵션을 사용하여 출력 형식을 선택할 수 있습니다. `TRADITIONAL`은 출력을 표 형식으로 표시합니다. `FORMAT` 옵션이 없는 경우 기본값입니다. `JSON` 형식은 정보를 JSON 형식으로 표시합니다.

`EXPLAIN`을 사용하면 테이블에 인덱스를 추가해야 하는 위치를 확인할 수 있으므로 인덱스를 사용하여 행을 찾음으로써 문이 더 빠르게 실행되도록 할 수 있습니다. 또한 EXPLAIN을 사용하여 옵티마이저가 테이블을 최적의 순서로 조인하는지 확인할 수 있습니다. 최적화 프로그램에 `SELECT` 문에서 테이블 이름이 지정된 순서와 일치하는 조인 순서를 사용하도록 힌트를 제공하려면 `SELECT`가 아닌 `SELECT STRAIGHT_JOIN`으로 문을 시작하십시오. (섹션 13.2.13, "SELECT 문" 참조) 그러나 `STRAIGHT_JOIN`은 준조인 변환을 비활성화하므로 인덱스가 사용되지 않을 수 있습니다. 세미조인 변환을 사용하여 IN 및 EXISTS 서브쿼리 조건 최적화를 참조하십시오.

옵티마이저 트레이스는 때때로 `EXPLAIN`의 정보를 보완하는 정보를 제공할 수 있습니다. 그러나 옵티마이저 추적 형식과 내용은 버전 간에 변경될 수 있습니다. 자세한 내용은 MySQL 내부를 참조하세요: 옵티마이저 추적을 참조하세요.

인덱스가 사용되어야 할 때 사용되지 않는 문제가 있는 경우, 테이블 분석을 실행하여 키의 카디널리티와 같은 테이블 통계를 업데이트하여 옵티마이저의 선택에 영향을 줄 수 있는 정보를 업데이트하세요. 섹션 13.7.3.1, "테이블 분석 문"을 참조하십시오.

참고

테이블의 열에 대한 정보를 얻기 위해 EXPLAIN을 사용할 수도 있습니다. `EXPLAIN tbl_name`은 `DESCRIBE tbl_name`과 동의어이며 `tbl_name`에서 칼럼을 보여준다. 자세한 내용은 섹션 13.8.1, "DESCRIBE 문" 및 섹션 13.7.7.6, "SHOW COLUMNS 문"을 참조한다.

8.8.2 설명 출력 형식

`EXPLAIN` 문은 MySQL이 문을 실행하는 방법에 대한 정보를 제공합니다. `EXPLAIN`는 `SELECT`, `DELETE`, `INSERT`, `REPLACE` 및 `UPDATE` 문과 함께 작동합니다.

EXPLAIN은 SELECT 문에 사용된 각 테이블에 대한 정보 행을 반환합니다. 출력에 테이블이 나열되는 순서는 MySQL이 문을 처리하는 동안 테이블을 읽는 순서입니다. 즉, MySQL은 첫 번째 테이블에서 행을 읽은 다음 두 번째 테이블에서 일치하는 행을 찾은 다음 세 번째 테이블에서 일치하는 행을 찾는 식으로 처리합니다. 모든 테이블이 처리되면 MySQL은 선택한 열을 출력하고
은 일치하는 행이 더 많은 테이블을 찾을 때까지 테이블 목록을 역추적합니다. 이 테이블에서 다음 행을 읽고 다음 테이블에서 프로세스를 계속 진행합니다.

참고

MySQL 워크벤치에는 설명 출력의 시각적 표현을 제공하는 시각적 설명 기능이 있습니다. 자습서를 참조하십시오: 설명 기능을 사용하여 쿼리 성능 개선하기를 참조하세요.

- 출력 열 설명
- 조인 유형 설명

- 추가 정보 설명
- 출력 해석 설명

출력 열 설명

이 섹션에서는 설명에 의해 생성되는 출력 열에 대해 설명합니다. 이후 섹션에서는 유형 및 추가 열에 대한 추가 정보를 제공합니다.

EXPLAIN의 각 출력 행은 하나의 테이블에 대한 정보를 제공합니다. 각 행에는 표 8.1, "EXPLAIN 출력 열"에 요약된 값이 포함되어 있으며 표 다음에 더 자세히 설명되어 있습니다. 열 이름은 테이블의 첫 번째 열에 표시되며, 두 번째 열은 FORMAT=JSON을 사용할 때 출력에 표시되는 것과 동일한 속성 이름을 제공합니다.

표 8.1 출력 열 설명

열	JSON 이름	의미
id	select_id	SELECT 식별자
선택 유형	없음	SELECT 유형
테이블	테이블_이름	출력 행의 테이블
파티션	파티션	일치하는 파티션
유형	액세스 유형	조인 유형
가능한_키	가능한_키	선택할 수 있는 인덱스
키	키	실제로 선택한 인덱스
key_len	키_길이	선택한 키의 길이
ref	ref	인덱스와 비교한 열
행	행	검사할 행의 추정치
필터링됨	필터링됨	테이블 조건에 따라 필터링된 행의 백분율
추가	없음	추가 정보

참고

NULL인 JSON 속성은JSON 형식에 표시되지 않습니다.
설명 출력.

- id (JSON 이름: select_id)

  SELECT 식별자입니다. 쿼리 내 SELECT의 일련 번호입니다. 행이 다른 행의 유니온 결과를 참조하는 경우 이 값은 NULL일 수 있습니다. 이 경우 테이블 열에 <unionM,N>과 같은 값이 표시되어 해당 행이 ID 값이 M과 N인 행의 유니온을 참조한다는 것을 나타냅니다.

- `select_type` (JSON 이름: 없음)

  `SELECT` 유형은 다음 표에 표시된 유형 중 하나 일 수 있습니다. JSON 형식의 `EXPLAIN`은 `SELECT` 유형이 `SIMPLE` 또는 `PRIMARY`가 아닌 경우 쿼리 블록의 속성으로 `SELECT` 유형을 노출합니다. 해당되는 경우 JSON 이름도 표에 나와 있습니다.

`select_type` 값	**JSON 이름**	**의미**
간단	없음	단순 `SELECT`(`UNION` 또는 하위 쿼리)
기본	없음	가장 바깥쪽 선택

select_type 값	JSON 이름	의미
UNION	없음	두 번째 이상 SELECT 유니온의 문
종속 결합	의존적 (참)	외부 쿼리에 의존하는 UNION의 두 번째 이상 SELECT 문
연합 결과	union_결과	유니온의 결과.
검색	없음	하위 쿼리에서 첫 번째 SELECT
종속 하위 쿼리	의존적 (참)	외부 쿼리에 종속된 하위 쿼리의 첫 번째 SELECT
파생	없음	파생 테이블
종속 파생	의존적 (참)	다른 테이블에 종속된 파생 테이블
소재	MATERIALIZED_From_SUBQUE	내 애터럴라이즈드 하위 쿼리
캐시할 수 없는 하위 쿼리	캐시 가능 (거짓)	결과를 캐시할 수 없고 외부 쿼리의 각 행에 대해 다시 평가해야 하는 하위 쿼리입니다.
캐시할 수 없는 유니온	캐시 가능 (거짓)	캐시할 수 없는 하위 쿼리에 속하는 UNION에서 두 번째 이상의 선택 항목(캐시할 수 없는 하위 쿼리 참조)

DEPENDENT는 일반적으로 상호 연관된 하위 쿼리를 사용함을 나타냅니다. 섹션 13.2.15.7, "상관된 하위 쿼리"를 참조하세요.

DEPENDENT SUBQUERY 평가는 UNCACHEABLE SUBQUERY 평가와 다릅니다. DEPENDENT SUBQUERY의 경우 하위 쿼리는 외부 컨텍스트에서 변수 값이 다른 각 집합에 대해 한 번만 다시 평가됩니다. UNCACHEABLE SUBQUERY의 경우 외부 컨텍스트의 각 행에 대해 하위 쿼리가 다시 평가됩니다.

EXPLAIN과 함께 FORMAT=JSON을 지정하면 출력에 select_type에 직접적으로 해당하는 속성이 하나도 없으며, 쿼리 블록 속성이 주어진 SELECT에 해당합니다. 방금 표시된 대부분의 SELECT 하위 쿼리 유형에 해당하는 속성을 사용할 수 있으며(예: MATERIALIZED의 경우 materialized_from_subquery), 적절한 경우 표시됩니다. SIMPLE 또는 PRIMARY에 해당하는 JSON은 없습니다.

SELECT 문이 아닌 경우 select_type 값은 영향을 받는 테이블의 문 유형을 표시합니다. 예를 들어 DELETE 문의 경우 select_type은 DELETE입니다.

- 테이블 (JSON 이름: table_name)

  출력 행이 참조하는 테이블의 이름입니다. 다음 값 중 하나 일 수도 있습니다:

- *<유니온M,N>*: 행은 `ID` 값이 M과 *N인* 행의 유니온을 나타냅니다.
- *<derivedN>*: 행은 `id` 값이 *N인* 행에 대한 파생 테이블 결과를 참조합니다. 예를 들어 `FROM` 절의 하위 쿼리에서 파생 테이블이 생성될 수 있습니다.
- *<subqueryN>*: 행은 `id` 값이 *N인* 행에 대한 구체화된 하위 쿼리의 결과를 나타냅니다. 8.2.2.2절. "구체화를 사용하여 하위 쿼리 최적화하기"를 참조하십시오.

- 파티션(JSON 이름: 파티션)

  쿼리에서 레코드가 일치할 파티션입니다. 파티션이 지정되지 않은 테이블의 경우 값은 `NULL`입니다. 섹션 24.3.5, "파티션에 대한 정보 얻기"를 참조하십시오.

- 유형(JSON 이름: access_type)

  조인 유형입니다. 다양한 유형에 대한 설명은 조인 유형 설명을 참조하십시오.

- 가능한_키(JSON 이름: 가능한_키)

  possible_keys 열은 MySQL이 이 테이블에서 행을 찾기 위해 선택할 수 있는 인덱스를 나타냅니다. 이 열은 EXPLAIN의 출력에 표시되는 테이블의 순서와는 완전히 독립적입니다. 즉, 가능한_키의 일부 키는 생성된 테이블 순서로 실제로 사용할 수 없을 수도 있습니다.

  이 열이 NULL(또는 JSON 형식의 출력에서 정의되지 않음)인 경우 관련 인덱스가 없습니다. 이 경우 WHERE 절을 검토하여 인덱싱에 적합한 열을 참조하는지 확인하여 쿼리 성능을 개선할 수 있습니다. 그렇다면 적절한 인덱스를 생성하고 EXPLAIN으로 쿼리를 다시 확인합니다. 섹션 13.1.9, "테이블 문 변경"을 참조하세요.

  테이블에 어떤 인덱스가 있는지 확인하려면 *tbl_name*에서 SHOW INDEX를 사용합니다.

- 키(JSON 이름: 키)

  키 열은 MySQL이 실제로 사용하기로 결정한 키(인덱스)를 나타냅니다. MySQL이 행을 조회하는 데 possible_keys 인덱스 중 하나를 사용하기로 결정한 경우 해당 인덱스가 키 값으로 나열됩니다.

  가능한_keys 값에 없는 인덱스의 이름을 key로 지정할 수 있습니다. 가능한_키 인덱스 중 행을 조회하는 데 적합한 인덱스가 없지만 쿼리에서 선택한 모든 열이 다른 인덱스의 열인 경우 이러한 문제가 발생할 수 있습니다. 즉, 명명된 인덱스가 선택한 열을 포함하므로 검색할 행을 결정하는 데는 사용되지 않지만 인덱스 스캔이 데이터 행 스캔보다 더 효율적입니다.

  InnoDB의 경우, InnoDB는 각 보조 인덱스에 기본 키 값을 저장하기 때문에 쿼리에서 기본 키도 선택하더라도 보조 인덱스가 선택한 열을 포함할 수 있습니다. 키가 NULL인 경우 MySQL은 쿼리를 보다 효율적으로 실행하는 데 사용할 인덱스를 찾지 못합니다.

  MySQL에서 가능한_키 열에 나열된 인덱스를 사용하거나 무시하도록 하려면 쿼리에서 FORCE INDEX, USE INDEX 또는 IGNORE INDEX를 사용합니다. 섹션 8.9.4, "인덱스 힌트"를 참조하십시오.

  MyISAM 테이블의 경우 테이블 분석을 실행하면 옵티마이저가 더 나은 인덱스를 선택하는 데 도움이 됩니다. MyISAM 테이블의 경우 myisamchk --analyze도 동일한 작업을 수행합니다. 13.7.3.1절, "테이블 분석 문" 및 7.6절, "MyISAM 테이블 유지 관리 및 크래시 복구"를 참조하세요.

- 키_len(JSON 이름: 키_길이)

  key_len 열은 MySQL이 사용하기로 결정한 키의 길이를 나타냅니다. key_len 값을 통해 MySQL이 실제로 여러 부분으로 구성된 키의 몇 부분을 사용하는지 확인할 수 있습니다. key 열이 NULL이면 key_len 열도 NULL입니다.

키 저장 형식으로 인해 NULL이 될 수 있는 열의 키 길이는
NOT NULL 열입니다.

- ref (JSON 이름: ref)

  참조 열은 키에 이름이 지정된 인덱스와 비교되는 열 또는 상수를 표시합니다.
  열을 클릭하여 테이블에서 행을 선택합니다.

  값이 함수인 경우 사용된 값은 특정 함수의 결과입니다. 어떤 함수인지 확인하려면 설명 뒤에 나오는 경고 표시를 사용하여 확장된 설명 출력을 확인합니다. 함수는 실제로 산술 연산자와 같은 연산자일 수 있습니다.

- 행(JSON 이름: 행)

  행 열은 MySQL이 쿼리를 실행하기 위해 검사해야 한다고 생각하는 행의 수를 나타냅니다.

  InnoDB 테이블의 경우 이 숫자는 추정치이며 항상 정확하지 않을 수 있습니다.

- 필터링됨 (JSON 이름: 필터링됨)

  필터링된 열은 테이블 조건에 의해 필터링된 테이블 행의 예상 백분율을 나타냅니다. 최대값은 100이며, 이는 행 필터링이 발생하지 않았음을 의미합니다. 100에서 감소하는 값은 필터링이 증가했음을 나타냅니다. 행은 검사된 행의 예상 수를 표시하고 행 × 필터링됨은 다음 테이블과 조인된 행의 수를 표시합니다. 예를 들어 행이 1000이고 필터링이 50.00(50%)인 경우 다음 테이블과 조인할 행의 수는 1000 × 50% = 500입니다.

- 추가(JSON 이름: 없음)

  이 열에는 MySQL이 쿼리를 해결하는 방법에 대한 추가 정보가 포함되어 있습니다. 다양한 값에 대한 설명은 추가 정보 설명을 참조하세요.

  Extra 열에 해당하는 단일 JSON 속성은 없지만, 이 열에서 발생할 수 있는 값은 JSON 속성 또는 메시지 속성의 텍스트로 노출됩니다.

조인 유형 설명

설명 출력의 유형 열은 테이블이 조인되는 방식을 설명합니다. JSON 형식의 출력에서는 access_type 속성의 값으로 표시됩니다. 다음 목록은 가장 좋은 유형부터 가장 나쁜 유형까지 순서대로 조인 유형을 설명합니다:

- 시스템

  테이블에 행이 하나만 있습니다(=시스템 테이블). 이것은 const 조인 유형의 특수한 경우입니다.

- const

  테이블에는 쿼리 시작 시 읽혀지는 일치하는 행이 최대 하나뿐입니다. 행이 하나뿐이므로 이 행의 열에 있는 값은 나머지 최적화 프로그램에서 상수로 간주할 수 있습니다. 상수 테이블은 한 번만 읽히기 때문에 매우 빠릅니다.

  const는 PRIMARY KEY 또는 UNIQUE 인덱스의 모든 부분을 상수 값과 비교할 때 사용됩니다. 다음 쿼리에서는 *tbl_name*을 const 테이블로 사용할 수 있습니다:

```
SELECT * FROM tbl_name WHERE primary_key=1;

SELECT * FROM tbl_name
  WHERE primary_key_part1=1 AND primary_key_part2=2;
```

- eq_ref

  이전 테이블의 행을 조합할 때마다 이 테이블에서 하나의 행을 읽습니다. 시스템 및 const 유형을 제외하고는 이것이 가장 적합한 조인 유형입니다. 인덱스의 모든 부분이 조인에서 사용되고 인덱스가 PRIMARY

KEY 또는 UNIQUE NOT NULL 인덱스인 경우에 사용됩니다.

연산자를 사용하여 비교되는 인덱싱된 열에 eq_ref를 사용할 수 있습니다. 비교 값은 상수이거나 이 테이블보다 먼저 읽은 테이블의 열을 사용하는 표현식일 수 있습니다. 다음 예제에서 MySQL은 eq_ref 조인을 사용하여 *ref_table을* 처리할 수 있습니다:

```
SELECT * FROM ref_table,other_table
  WHERE ref_table.key_column=다른_table.column;

SELECT * FROM ref_table,other_table
  WHERE ref_table.key_column_part1=다른_table.column
  AND ref_table.key_column_part2=1;
```

- ref

  이전 테이블의 각 행 조합에 대해 이 테이블에서 인덱스 값이 일치하는 모든 행을 읽습니다. ref는 조인이 키의 가장 왼쪽 접두사만 사용하거나 키가 PRIMARY KEY 또는 UNIQUE 인덱스가 아닌 경우(즉, 조인이 키 값에 따라 단일 행을 선택할 수 없는 경우)에 사용됩니다. 사용되는 키가 몇 개의 행만 일치하는 경우 이 조인 유형이 적합한 조인 유형입니다.

  또는 <=> 연산자를 사용하여 비교되는 인덱싱된 열에 ref를 사용할 수 있습니다. 다음 예제에서 MySQL은 ref 조인을 사용하여 *ref_table을* 처리할 수 있습니다:

```
SELECT * FROM ref_table WHERE key_column=expr;

SELECT * FROM ref_table,other_table
  WHERE ref_table.key_column=다른_table.column;

SELECT * FROM ref_table,other_table
  WHERE ref_table.key_column_part1=다른_table.column
  AND ref_table.key_column_part2=1;
```

- 전체 텍스트

  조인은 전체 텍스트 인덱스를 사용하여 수행됩니다.

- ref_or_null

  이 조인 유형은 ref와 비슷하지만 MySQL이 NULL 값이 포함된 행을 추가로 검색한다는 점이 다릅니다. 이 조인 유형 최적화는 하위 쿼리 해결에 가장 자주 사용됩니다. 다음 예제에서 MySQL은 ref_or_null 조인을 사용하여 *ref_table을* 처리할 수 있습니다:

```
SELECT * FROM ref_table
  WHERE key_column=expr 또는 key_column이 NULL입니다;
```

  섹션 8.2.1.15, "IS NULL 최적화"를 참조하세요.

- index_merge

  이 조인 유형은 인덱스 병합 최적화가 사용되었음을 나타냅니다. 이 경우 출력 행의 key 열에는 사용된 인덱스 목록이 포함되고 key_len에는 사용된 인덱스의 가장 긴 키 부분 목록이 포함됩니다. 자세한 내용은 섹션 8.2.1.3, "인덱스 병합 최적화"를 참조하세요.

- 고유_서브쿼리

  이 유형은 다음 형식의 일부 IN 하위 쿼리에 대해 eq_ref를 대체합니다:

```
value IN (SELECT primary_key FROM single_table WHERE some_expr)
```

  고유_서브쿼리는 효율성을 높이기 위해 서브쿼리를 완전히 대체하는 인덱스 조회 함수일 뿐입니다.

- index_subquery

  이 조인 유형은 unique_subquery와 유사합니다. IN 서브쿼리를 대체하지만 다음 형식의 서브쿼리에서

고유하지 않은 인덱스에 대해 작동합니다:

```
value IN (SELECT key_column FROM single_table WHERE some_expr)
```

- 범위

  인덱스를 사용하여 행을 선택하면 지정된 범위에 있는 행만 검색됩니다. 출력 행의 키 열은 사용된 인덱스를 나타냅니다. key_len에는 사용된 가장 긴 키 부분이 포함됩니다. 이 유형의 경우 참조 열은 NULL입니다.

  범위는 =, <>, >, >=, < 중 하나를 사용하여 키 열을 상수와 비교할 때 사용할 수 있습니다, <=, IS NULL, <=>, BETWEEN, LIKE 또는 IN() 연산자를 사용할 수 있습니다:

```
SELECT * FROM tbl_name
  WHERE key_column = 10;

SELECT * FROM tbl_name
  WHERE 키_열 10과 20 사이;

SELECT * FROM tbl_name
  WHERE key_column IN (10,20,30);

SELECT * FROM tbl_name
  WHERE key_part1 = 10 AND key_part2 IN (10,20,30);
```

- index

  인덱스 조인 유형은 인덱스 트리가 스캔된다는 점을 제외하면 ALL과 동일합니다. 이는 두 가지 방식으로 발생합니다:

  - 인덱스가 쿼리에 대한 커버링 인덱스이고 테이블에서 필요한 모든 데이터를 충족하는 데 사용할 수 있는 경우 인덱스 트리만 스캔됩니다. 이 경우 추가 열에 인덱스 사용이라고 표시됩니다. An 인덱스 전용 스캔은 일반적으로 인덱스의 크기가 테이블 데이터보다 작기 때문에 일반적으로 ALL보다 빠릅니다.
  - 인덱스에서 읽기를 사용하여 인덱스 순서대로 데이터 행을 조회하는 전체 테이블 스캔이 수행됩니다. 사용 인덱스는 추가 열에 표시되지 않습니다.

  쿼리가 단일 인덱스의 일부인 열만 사용하는 경우 MySQL에서 이 조인 유형을 사용할 수 있습니다.

- 모두

  이전 테이블의 각 행 조합에 대해 전체 테이블 스캔이 수행됩니다. 테이블이 상수로 표시되지 않은 첫 번째 테이블인 경우 일반적으로 좋지 않으며, 다른 모든 경우에는 *매우 좋지 않습니다.* 일반적으로 이전 테이블의 상수 값 또는 열 값을 기반으로 테이블에서 행을 검색할 수 있는 인덱스를 추가하여 ALL을 피할 수 있습니다.

추가 정보 설명

설명 출력의 추가 열에는 MySQL이 쿼리를 해결하는 방법에 대한 추가 정보가 포함되어 있습니다. 다음 목록은 이 열에 표시될 수 있는 값에 대해 설명합니다. 또한 각 항목은 JSON 형식의 출력에 대해 어떤 속성이 Extra 값을 표시하는지를 나타냅니다. 이 중 일부에는 특정 속성이 있습니다. 나머지는 메시지 속성의 텍스트로 표시됩니다.

쿼리를 최대한 빠르게 수행하려면 Using filesort 및 Using temporary의 추가 열 값 또는 JSON 형식의 EXPLAIN 출력에서 using_filesort 및 using_temporary_table 속성이 true와 같은지

확인합니다.

- 역방향 인덱스 스캔(JSON: backward_index_scan)

  옵티마이저는 `InnoDB` 테이블에서 내림차순 인덱스를 사용할 수 있습니다. 인덱스 사용과 함께 표시됩니다. 자세한 내용은 섹션 8.3.13, "내림차순 인덱스"를 참조하십시오.

- *'table'*의 자식이 `join@1`을 푸시했습니다(JSON: 메시지 텍스트).

  이 테이블은 NDB 커널로 푸시다운할 수 있는 조인에서 테이블의 자식으로 참조됩니다. 푸시다운 조인이 활성화된 NDB 클러스터에서만 적용됩니다. 자세한 내용과 예제는 `ndb_join_pushdown` 서버 시스템 변수에 대한 설명을 참조하세요.

- const 행을 찾을 수 없음(JSON 속성: const_row_not_found)

  SELECT ... FROM *tbl_name*과 같은 쿼리의 경우 테이블이 비어 있습니다.

- 모든 행 삭제(JSON 속성: 메시지)

  삭제의 경우 일부 저장소 엔진(예: MyISAM)은 간단하고 빠른 방법으로 모든 테이블 행을 제거하는 핸들러 메서드를 지원합니다. 엔진이 이 최적화를 사용하는 경우 이 추가 값이 표시됩니다.

- 고유(JSON 속성: 고유)

  MySQL은 고유 값을 찾고 있으므로 일치하는 첫 번째 행을 찾은 후에는 현재 행 조합에 대한 추가 행 검색을 중지합니다.

- FirstMatch(*tbl_name*) (JSON 속성: first_match)

  세미조인 FirstMatch 조인 바로 가기 전략은 *tbl_name*에 사용됩니다.

- NULL 키에 대한 전체 검사(JSON 속성: 메시지)

  이는 옵티마이저가 인덱스 조회 액세스 방법을 사용할 수 없는 경우 대체 전략으로 하위 쿼리 최적화를 위해 발생합니다.

- 보유 불가(JSON 속성: 메시지)

  HAVING 절은 항상 거짓이며 어떤 행도 선택할 수 없습니다.

- 불가능한 WHERE(JSON 속성: 메시지)

  WHERE 절은 항상 거짓이며 어떤 행도 선택할 수 없습니다.

- 생성 테이블을 읽은 후 발견한 불가능한 WHERE(JSON 속성: 메시지) MySQL이 모든 생성(및 시스템)

  테이블을 읽었으며 WHERE 절이 항상 거짓임을 확인했습니다.

- LooseScan(*m..n*) (JSON 속성: 메시지)

  세미조인 루즈스캔 전략이 사용됩니다. m과 n은 주요 부품 번호입니다.

- 일치하는 최소/최대 행이 없음(JSON 속성: 메시지)

  SELECT MIN(...) FROM ... WHERE와 같은 쿼리의 조건을 충족하는 행이 없습니다.
  조건입니다.

- const 테이블에 일치하는 행이 없습니다(JSON 속성: 메시지).

  조인이 있는 쿼리의 경우 빈 테이블이 있거나 고유 인덱스 조건을 충족하는 행이 없는 테이블이 있습니다.

- 파티션 가지치기 후 일치하는 행이 없음(JSON 속성: 메시지)

DELETE 또는 UPDATE의 경우 옵티마이저가 파티션 정리 후 삭제하거나 업데이트할 항목을 찾지 못했습니다. 이는 SELECT 문에 대한 Impossible WHERE와 비슷한 의미입니다.

- 사용된 테이블 없음(JSON 속성: 메시지)

  쿼리에 FROM 절이 없거나 FROM DUAL 절이 있습니다.

  INSERT 또는 REPLACE 문의 경우 SELECT 부분이 없을 때 EXPLAIN은 이 값을 표시합니다. 예를 들어, EXPLAIN INSERT INTO t VALUES(10)의 경우 이 값이 표시되는데, 이는 EXPLAIN INSERT INTO t SELECT 10 FROM DUAL과 동일하기 때문입니다.

- 존재하지 않음(JSON 속성: 메시지)

MySQL은 쿼리에 대해 왼쪽 조인 최적화를 수행할 수 있었고 왼쪽 조인 조건과 일치하는 행 하나를 찾은 후에는 이 테이블에서 이전 행 조합에 대해 더 많은 행을 검사하지 않습니다. 다음은 이러한 방식으로 최적화할 수 있는 쿼리 유형의 예입니다:

```
SELECT * FROM t1 LEFT JOIN t2 ON t1.id=t2.id
  WHERE t2.id IS NULL;
```

t2.id가NOT NULL로 정의되어 있다고 가정합니다. 이 경우 MySQL은 t1을 스캔하고 t1.id의 값을 사용하여 t2의 행을 조회합니다. MySQL은 t2에서 일치하는 행을 찾으면 t2.id가 절대 NULL이 될 수 없다는 것을 알고 t2의 나머지 행 중 동일한 id 값을 가진 행을 스캔하지 않습니다. 즉, t1의 각 행에 대해 MySQL은 t2에서 실제로 일치하는 행의 수에 관계없이 t2에서 단 한 번의 조회만 수행하면 됩니다.

이는 또한 NOT IN(*하위 쿼리*) 또는 NOT EXISTS(*하위 쿼리*) 형식의 WHERE 조건이 내부적으로 안티조인으로 변환되었음을 나타낼 수도 있습니다. 이렇게 하면 하위 쿼리가 제거되고 해당 테이블이 최상위 쿼리에 대한 계획에 포함되므로 비용 계획이 개선됩니다. 세미조인과 안티조인을 병합함으로써 옵티마이저는 실행 계획에서 테이블의 순서를 더 자유롭게 재조정할 수 있으며, 경우에 따라 더 빠른 계획으로 이어질 수 있습니다.

특정 쿼리에 대해 안티조인 변환이 수행되는 시기는 EXPLAIN 실행 후 SHOW WARNINGS의 메시지 열을 확인하거나 EXPLAIN FORMAT=TREE의 출력에서 확인할 수 있습니다.

참고

안티조인은 세미조인 *table_a JOIN table_b ON 조건의* 보완입니다. 반조인은 *table_b에 조건과* 일치하는 행이 *없는 table_a의* 모든 행을 반환합니다.

- 계획이 아직 준비되지 않았습니다(JSON 속성: 없음).

  이 값은 최적화 프로그램이 명명된 연결에서 실행되는 문에 대한 실행 계획 생성을 완료하지 않은 경우 연결에 대한 설명과 함께 발생합니다. 실행 계획 출력이 여러 줄로 구성된 경우 전체 실행 계획을 결정하는 옵티마이저의 진행 상황에 따라 일부 또는 전부가 이 추가 값을 가질 수 있습니다.

- 각 레코드에 대해 확인된 범위(인덱스 맵: *N*) (JSON 속성: 메시지)

  MySQL에서 사용할 수 있는 적절한 인덱스를 찾지 못했지만 일부 인덱스는 이전 테이블의 열 값을 알고 난 후에 사용할 수 있다는 것을 발견했습니다. 이전 테이블의 각 행 조합에 대해 MySQL은 행을 검색하는 데 범위 또는 index_merge 액세스 방법을 사용할 수 있는지 여부를 확인합니다. 이 방법은 매우 빠르지는 않지만 인덱스가 전혀 없는 조인을 수행하는 것보다 빠릅니다. 적용 가능 기준은 섹션 8.2.1.2, "범위 최적화" 및 섹션 8.2.1.3, "인덱스 병합 최적화"에 설명된 것과 같지만, 이전 테이블의 모든 열 값이 알려져 있고 상수로 간주된다는 점을 제외하면 다릅니다.

  인덱스는 테이블의 SHOW INDEX에 표시된 것과 같은 순서로 1부터 번호가 매겨집니다. 인덱스 맵 값 N은 어떤 인덱스가 후보인지를 나타내는 비트마스크 값입니다. 예를 들어 0x19(이진 11001) 값은 인덱스 1, 4, 5가 고려된다는 의미입니다.

- 재귀적(JSON 속성: 재귀적)

  이는 행이 재귀적 공통 테이블 표현식의 재귀적 `SELECT` 부분에 적용됨을 나타냅니다. 섹션 13.2.20, "WITH(공통 테이블 표현식)"를 참조하십시오.

- 리머터리얼라이즈(JSON 속성: 리머터리얼라이즈)

  테이블 `T`에 대한 설명 행에 리재구성 `(X,...)`이 표시되며, 여기서 `X`는 `T`의 새 행을 읽을 때 리재구성이 트리거되는 측면 파생 테이블입니다. 예를 들어

```
SELECT
```

```
  ...
FROM
  t,
  LATERAL(t를 참조하는 파생 테이블) AS dt
...
```

파생된 테이블의 콘텐츠는 최상위 쿼리에 의해 새로운 t 행이 처리될 때마다 최신 상태로 가져오기 위해 다시 구체화됩니다.

- 스캔한 N개의 데이터베이스(JSON 속성: 메시지)

  이 값은 8.2.3절. "정보 스키마 쿼리 최적화"에 설명된 대로 서버가 정보 스키마 테이블에 대한 쿼리를 처리할 때 수행하는 디렉터리 검색 횟수를 나타냅니다. N의 값은 0, 1 또는 모두일 수 있습니다.

- 최적화된 테이블 선택(JSON 속성: 메시지)

  옵티마이저는 1) 최대 하나의 행만 반환해야 하고, 2) 이 행을 생성하려면 결정론적 행 집합을 읽어야 한다고 결정합니다. 최적화 단계에서 읽을 행을 읽을 수 있는 경우(예: 인덱스 행 읽기), 쿼리 실행 중에 테이블을 읽을 필요가 없습니다.

  첫 번째 조건은 쿼리가 암시적으로 그룹화될 때 충족됩니다(집계 함수를 포함하지만 GROUP BY 절이 없음). 두 번째 조건은 사용된 인덱스당 하나의 행 조회가 수행될 때 충족됩니다. 읽은 인덱스의 수에 따라 읽을 행의 수가 결정됩니다.

  다음과 같이 암시적으로 그룹화된 쿼리를 생각해 보겠습니다:

  ```
  SELECT MIN(c1), MIN(c2) FROM t1;
  ```

  하나의 인덱스 행을 읽어 MIN(c1)을 검색할 수 있고, 다른 인덱스에서 하나의 행을 읽어 MIN(c2)를 검색할 수 있다고 가정합니다. 즉, 각 열 c1과 c2에 대해 해당 열이 인덱스의 첫 번째 열인 인덱스가 존재합니다. 이 경우 두 개의 결정론적 행을 읽어서 생성된 하나의 행이 반환됩니다.

  읽을 행이 결정론적이지 않은 경우 이 추가 값이 발생하지 않습니다. 이 쿼리를 살펴보겠습니다:

  ```
  SELECT MIN(c2) FROM t1 WHERE c1 <= 10;
  ```

  (c1, c2)가 커버링 인덱스라고 가정합니다. 이 인덱스를 사용하면 최소 c2 값을 찾기 위해 c1 <= 10인 모든 행을 스캔해야 합니다. 반대로 이 쿼리를 생각해 보겠습니다:

  ```
  SELECT MIN(c2) FROM t1 WHERE c1 = 10;
  ```

  이 경우 c1 = 10인 첫 번째 인덱스 행에는 최소 c2 값이 포함됩니다. 반환된 행을 생성하려면 하나의 행만 읽어야 합니다.

  테이블당 정확한 행 수를 유지하는 저장소 엔진(예: MyISAM)의 경우(InnoDB는 아님), 이 추가 값은 WHERE 절이 누락되었거나 항상 참이고 GROUP BY 절이 없는 COUNT(*) 쿼리에 대해 발생할 수 있습니

다. (이는 스토리지 엔진이 결정적인 수의 행을 읽을 수 있는지 여부에 영향을 미치는 암시적으로 그룹화된 쿼리의 인스턴스입니다.)

- `Skip_open_table`, `Open_frm_only`, `Open_full_table` (JSON 속성: 메시지)

  이 값은 `INFORMATION_SCHEMA`에 대한 쿼리에 적용되는 파일 열기 최적화를 나타냅니다.
  테이블.

  - `Skip_open_table`: 테이블 파일을 열 필요가 없습니다. 정보는 이미 데이터 사전에서 사용할 수 있습니다.
  - `Open_frm_only`: 테이블 정보를 위해 데이터 사전만 읽으면 됩니다.
  - `Open_full_table`: 최적화되지 않은 정보 조회. 테이블 정보는 데이터 사전과 테이블 파일을 읽어서 읽어야 합니다.

- 임시 시작, 임시 종료(JSON 속성: 메시지)

  세미조인 중복 위드아웃 전략에 임시 테이블을 사용함을 나타냅니다.

- 고유 행을 찾을 수 없음(JSON 속성: 메시지)

  `SELECT ... FROM tbl_name`과 같은 쿼리의 경우 `UNIQUE` 조건을 충족하는 행이 없습니다.
  인덱스 또는 기본 키를 입력합니다.

- 파일 정렬 사용(JSON 속성: `using_filesort`)

  MySQL은 정렬된 순서로 행을 검색하는 방법을 찾기 위해 추가 패스를 수행해야 합니다. 정렬은 조인 유형에 따라 모든 행을 살펴보고 `WHERE` 절과 일치하는 모든 행에 대해 정렬 키와 행에 대한 포인터를 저장하는 방식으로 수행됩니다. 그런 다음 키가 정렬되고 정렬된 순서대로 행이 검색됩니다. 섹션 8.2.1.16, "최적화 기준 정렬"을 참조하십시오.

- 인덱스 사용(JSON 속성: `using_index`)

  실제 행을 읽기 위해 추가 검색을 수행할 필요 없이 인덱스 트리의 정보만 사용하여 테이블에서 열 정보를 검색합니다. 이 전략은 쿼리가 단일 인덱스의 일부인 열만 사용하는 경우에 사용할 수 있습니다.

  사용자 정의 클러스터 인덱스가 있는 `InnoDB` 테이블의 경우 추가 열에 인덱스 사용이 없는 경우에도 해당 인덱스를 사용할 수 있습니다. 유형이 인덱스이고 키가 `PRIMARY`인 경우에 해당됩니다.

  사용된 커버리지 인덱스에 대한 정보는 `EXPLAIN FORMAT=TRADITIONAL` 및
  설명 형식=`json`. `EXPLAIN FORMAT=TREE`에도 표시됩니다.

- 인덱스 조건 사용(JSON 속성: `using_index_condition`)

  테이블은 인덱스 튜플에 액세스하고 먼저 테스트하여 전체 테이블 행을 읽을지 여부를 결정하여 읽습니다. 이러한 방식으로 인덱스 정보는 필요한 경우가 아니면 전체 테이블 행 읽기를 연기("푸시 다운")하는 데 사용됩니다. 섹션 8.2.1.6, "인덱스 조건 푸시다운 최적화"를 참조하십시오.

- 그룹별 인덱스 사용(JSON 속성: `using_index_for_group_by`)

  인덱스 테이블 액세스 사용 방법과 유사하게, 그룹별로 인덱스 사용은 MySQL이 실제 테이블에 대한 추

가 디스크 액세스 없이 `GROUP BY` 또는 `DISTINCT` 쿼리의 모든 열을 검색하는 데 사용할 수 있는 인덱스를 찾았음을 나타냅니다. 또한 인덱스가 가장 효율적인 방식으로 사용되므로 각 그룹에 대해 몇 개의 인덱스 항목만 읽습니다. 자세한 내용은 섹션 8.2.1.17, "GROUP BY 최적화"를 참조하십시오.

- 건너뛰기 스캔에 인덱스 사용(JSON 속성: `using_index_for_skip_scan`) 건너뛰기 스캔 액세스 메서드가 사용됨을 나타냅니다. 스킵 스캔 범위 액세스 방법을 참조하십시오.

- 조인 버퍼 사용(중첩 루프 차단), 조인 버퍼 사용(일괄 키 액세스), 조인 버퍼 사용(해시 조인) (JSON 속성: `using_join_buffer`)

이전 조인의 테이블을 조인 버퍼로 부분적으로 읽은 다음 버퍼에서 해당 행을 사용하여 현재 테이블과의 조인을 수행합니다. (블록 중첩 루프)는 블록 중첩 루프 알고리즘을 사용하고, (배치 키 액세스)는 배치 키 액세스 알고리즘을 사용하며, (해시 조인)은 해시 조인을 사용함을 나타냅니다. 즉, 설명 출력의 앞줄에 있는 테이블의 키가 버퍼링되고, 조인 버퍼 사용으로 표시된줄로 표시된 테이블에서 일치하는 행을 일괄적으로 가져옵니다.

JSON 형식의 출력에서 using_join_buffer의 값은 항상 블록 중첩 루프, 일괄 키 액세스 또는 해시 조인 중 하나입니다.

해시 조인에 대한 자세한 내용은 섹션 8.2.1.4, "해시 조인 최적화" 및 블록 중첩 루프 조인 알고리즘을 참조하세요.

일괄 키 액세스 알고리즘에 대한 자세한 내용은 일괄 키 액세스 조인을 참조하세요.

- MRR 사용(JSON 속성: 메시지)

  테이블은 다중 범위 읽기 최적화 전략을 사용하여 읽습니다. 섹션 8.2.1.11, "다중 범위 읽기 최적화"를 참조하십시오.

- Using sort_union(...), Using union(...), Using intersect(...) (JSON 속성: 메시지)

  인덱스 스캔이 index_merge에 대해 병합되는 방법을 보여주는 특정 알고리즘을 나타냅니다. 조인 유형. 섹션 8.2.1.3, "인덱스 병합 최적화"를 참조하세요.

- 임시 사용(JSON 속성: using_temporary_table)

  쿼리를 해결하려면 MySQL에서 결과를 저장할 임시 테이블을 만들어야 합니다. 이는 일반적으로 쿼리에 열을 다르게 나열하는 GROUP BY 및 ORDER BY 절이 포함된 경우에 발생합니다.

- where(JSON 속성: 첨부된_조건) 사용

  WHERE 절은 다음 테이블과 일치시킬 행을 제한하거나 클라이언트로 전송하는 데 사용됩니다. 특별히 테이블에서 모든 행을 가져오거나 검사하려는 것이 아니라면, 추가 값이 사용 위치가 아니고 테이블 조인 유형이 ALL 또는 인덱스인 경우 쿼리에 문제가 있을 수 있습니다.

  JSON 형식의 출력에 직접 대응하는 항목이 없는 경우, 첨부된_조건을 사용합니다. 프로퍼티에 사용된 WHERE 조건이 포함되어 있습니다.

- 푸시 조건과 함께 where 사용(JSON 속성: 메시지)

  이 항목은 NDB *테이블에만* 적용됩니다. 즉, NDB 클러스터가 인덱싱되지 않은 열과 상수 간의 직접 비교의 효율성을 개선하기 위해 조건 푸시다운 최적화를 사용하고 있음을 의미합니다. 이러한 경우, 조건은 클러스터의 데이터 노드로 "푸시다운"되어 모든 데이터 노드에서 동시에 평가됩니다. 이렇게 하면 네트워크를 통해 일치하지 않는 행을 전송할 필요가 없으며, 조건 푸시다운을 사용할 수 있지만 사용하지 않는 경우보다 5배에서 10배까지 쿼리 속도를 높일 수 있습니다. 자세한 내용은 섹션 8.2.1.5, "엔진 조건 푸시다운 최적화"를 참조하세요.

- 제로 제한(JSON 속성: 메시지)

  쿼리에 `LIMIT 0` 절이 있어 행을 선택할 수 없습니다.

출력 해석 설명

`EXPLAIN` 출력의 행 열에 있는 값의 곱을 구하면 조인이 얼마나 좋은지 잘 알 수 있습니다. 이 값은 쿼리를 실행하기 위해 MySQL이 얼마나 많은 행을 검사해야 하는지 대략적으로 알려줍니다. `max_join_size` 시스템 변수를 사용하여 쿼리를 제한하는 경우 이 행 곱은 실행할 다중 테이블 `SELECT` 문과 중단할 문을 결정하는 데도 사용됩니다. 5.1.1절. "서버 구성"을 참조하십시오.

다음 예는 EXPLAIN에서 제공하는 정보를 기반으로 다중 테이블 조인을 점진적으로 최적화하는 방법을 보여 줍니다.

여기에 표시된 SELECT 문이 있고 다음을 사용하여 이 문을 검사한다고 가정해 보겠습니다.
설명:

```
EXPLAIN SELECT tt.TicketNumber, tt.TimeIn,
               tt.ProjectReference, tt.EstimatedShipDate,
               tt.ActualShipDate, tt.ClientID,
               tt.ServiceCodes, tt.RepetitiveID,
               tt.CurrentProcess, tt.CurrentDPPerson,
               tt.RecordVolume, tt.DPPrinted, et.COUNTRY,
               et_1.COUNTRY, do.CUSTNAME
        FROM tt, et, et AS et_1, do
        WHERE tt.SubmitTime IS NULL
          AND tt.ActualPC = et.EMPLOYID
          AND tt.AssignedPC = et_1.EMPLOYID
          AND tt.ClientID = do.CUSTNMBR;
```

이 예제에서는 다음과 같은 가정을 합니다:

- 비교 대상 열은 다음과 같이 선언되었습니다.

표	열	데이터 유형
tt	실제 PC	CHAR(10)
tt	할당된 PC	CHAR(10)
tt	ClientID	CHAR(10)
et	EMPLOYID	CHAR(15)
do	CUSTNMBR	CHAR(15)

- 테이블에는 다음과 같은 인덱스가 있습니다.

표	색인
tt	실제 PC
tt	할당된 PC
tt	ClientID
et	EMPLOYID(기본 키)
do	CUSTNMBR(기본 키)

- tt.ActualPC 값이 균등하게 분포되어 있지 않습니다.

처음에 최적화가 수행되기 전에 EXPLAIN 문은 다음과 같은 정보를 생성합니다:

```
표 외  모두  possible_keys key key_len ref rows
             PRIMARY       NULL       74
       입력                                      추가
do     모두  기본          NULL NULL  NULL 2135
et_1   모두  기본          NULL NULL  NULL 74
tt     모두  할당된 PC,     NULL NULL  NULL 3872
             ClientID,
               실제 PC                          0x23)
             확인된 범위    각 레코드    (인덱스 맵:
```

각 테이블의 유형이 ALL이므로 이 출력은 MySQL이 모든 테이블, 즉 행의 모든 조합의 데카르트 곱을 생성하

고 있음을 나타냅니다. 각 테이블에 있는 행 수의 곱을 검사해야 하므로 시간이 꽤 오래 걸립니다. 현재 사례의 경우 이 곱은 74입니다.

× 2135 × 74 × 3872 = 45,268,558,720행. 테이블이 더 컸다면 시간이 얼마나 걸릴지 상상만 해도 알 수 있습니다.

여기서 한 가지 문제는 열의 인덱스가 동일한 유형과 크기로 선언된 경우 MySQL에서 보다 효율적으로 사용할 수 있다는 것입니다. 이 컨텍스트에서 VARCHAR와 CHAR는 다음과 같은 경우 동일한 것으로 간주됩니다.

는 같은 크기로 선언되어 있고, tt.ActualPC는 CHAR(10)으로 선언되어 있고 et.EMPLOYID는 CHAR(15)로 선언되어 있어 길이가 불일치합니다.

열 길이 간의 이러한 불일치를 해결하려면 ALTER TABLE을 사용하여 ActualPC를 10자에서 15자로 늘립니다:

```
mysql> ALTER TABLE tt MODIFY ActualPC VARCHAR(15);
```

이제 tt.ActualPC 및 et.EMPLOYID는 모두 VARCHAR(15)입니다. EXPLAIN 문을 다시 실행하면 이 결과가 생성됩니다:

```
테이블 유형      possible_keys 키TT      key_len ref          행       추가
                                         NULL    NULL
       ALL       할당된 PC,    NULL                         3872     사용
                 ClientID,                                           어디
                   실제 PC
do     ALL       기본          NULL      NULL    NULL       2135
            각 레코드에 대해 확인된 범위(인덱스 맵: 0x1)
et_1 ALL         PRIMARY       NULL      NULL    NULL       74
            각 레코드에 대해 확인된 범위(인덱스 맵: 0x1)
             eteq_ref                PRIMARYPRIMARY 15tt
                          .ActualPC 1
```

완벽하지는 않지만 훨씬 낫습니다: 행 값의 곱이 74배나 작아졌습니다. 이 버전은 몇 초 안에 실행됩니다.

두 번째 변경을 통해 tt.AssignedPC의 열 길이 불일치를 제거할 수 있습니다.
= et_1.EMPLOYID 및 tt.ClientID = do.CUSTNMBR을 비교합니다:

```
mysql> ALTER TABLE tt MODIFY AssignedPC VARCHAR(15),
                      클라이언트 ID 수정    VARCHAR(15);
```

수정 후 EXPLAIN은 여기에 표시된 출력을 생성합니다:

```
테이블 유형      가능한_키 키              키_렌 참조                행 추가
et     ALL       PRIMARY       NULL      NULL    NULL              74
tt     ref       할당PC, 클라이 실제PC 15                 et.EMPLOYID       52   어디에
                 언트ID, 실제PC                                                  사용
ET_1 EQ_REF      기본                    초등 15                   tt.AssignedPC 1
do     eq_ref    기본                    초등 15                   tt.ClientID   1
```

이 시점에서 쿼리는 가능한 한 거의 최적화되었습니다. 남은 문제는 기본적으로 MySQL은 tt.ActualPC 열의 값이 균등하게 분포되어 있다고 가정하는데, tt 테이블의 경우 그렇지 않다는 것입니다. 다행히도 MySQL에 키 분포를 분석하도록 지시하는 것은 쉽습니다:

```
mysql> ANALYZE TABLE tt;
```

추가 인덱스 정보를 사용하면 조인이 완벽해지며 EXPLAIN은 이 결과를 생성합니다:

```
테이블 유형      가능한_키 키       key_len ref           행 추가
tt     ALL       할당된 PC NULL     NULL    NULL          3872 사용 중
                 ClientID,                                     어디
                 실제 PC
et     eq_ref    기본      기본     15      tt.ActualPC   1
ET_1 EQ_REF      기본      기본     15      tt.AssignedPC 1
do     eq_ref    기본      기본     15      tt.ClientID   1
```

EXPLAIN 출력의 행 열은 MySQL 조인 최적화 프로그램에서 추측한 값입니다. 행 곱을 쿼리가 반환하는 실

제 행 수와 비교하여 숫자가 진실에 가까운지 확인합니다. 숫자가 상당히 다른 경우, `SELECT` 문에 `STRAIGHT_JOIN`을 사용하고 `FROM` 절에서 테이블을 다른 순서로 나열하면 더 나은 성능을 얻을 수 있습니다. (단, `STRAIGHT_JOIN`으로 인해 인덱스가 사용되지 않을 수 있습니다.
를 사용하지 않는 것이 좋습니다. 세미조인 변환을 사용하여 IN 및 EXISTS 하위 쿼리 조건 최적화하기 참조).

경우에 따라 하위 쿼리와 함께 `EXPLAIN` SELECT를 사용할 때 데이터를 수정하는 문을 실행할 수 있습니다(자세한 내용은 13.2.15.8절. "파생 테이블"을 참조).

8.8.3 확장된 EXPLAIN 출력 형식

`EXPLAIN` 문은 `EXPLAIN` 출력의 일부가 아닌 추가("확장") 정보를 생성하지만, `EXPLAIN` 뒤에 `SHOW WARNINGS` 문을 실행하면 볼 수 있습니다. 확장 정보는 `SELECT`, `DELETE`, `INSERT`, `REPLACE` 및 `UPDATE` 문에서 사용할 수 있습니다.

`SHOW WARNINGS` 출력의 메시지 값에는 최적화 프로그램이 `SELECT` 문에서 테이블 및 열 이름을 한정하는 방법, 재작성 및 최적화 규칙을 적용한 후 `SELECT`의 모습, 최적화 프로세스에 대한 기타 참고 사항 등이 표시됩니다.

`EXPLAIN` 다음에 `SHOW WARNINGS` 문으로 표시할 수 있는 확장 정보는 `SELECT` 문에 대해서만 생성됩니다. 다른 설명 가능한 문(삭제, 삽입, 대체 및 업데이트)에 대해서는 `SHOW WARNINGS`가 빈 결과를 표시합니다.

다음은 확장된 `EXPLAIN` 출력의 예입니다:

```
mysql> 설명
       SELECT t1.a, t1.a IN (SELECT t2.a FROM t2) FROM t1\G
*************************** 1. 행 ***************************
           ID: 1
  select_type: PRIMARY 테이
       블: t1
         유형: 인덱스 가능한
_키: NULL
          키입니다: PRIMARY
      key_len: 4
          참조:
         NULL 행: 4
     필터링됨: 100.00 추가: 인
        덱스 사용
*************************** 2. 행 ***************************
           ID: 2
  select_type: SUBQUERY 테
        이블: t2
         유형: 인덱스 가능_
키: A
          키: 키
      _len: 5
          참조:
         NULL 행: 3
     필터링됨: 100.00 추가: 인
        덱스 사용
세트 2행, 경고 1회(0.00초)

mysql> SHOW WARNINGS\G
*************************** 1. 행 *************************** 레
  벨: 참고
   코드: 1003
메시지: /* select#1 */ select `test`.`t1`.`a` AS `a`,
         <in_optimizer>(`test`.`t1`.`a`,`test`.`t1`.`a` in
         ( <materialize> (/* select#2 */ select `test`.`t2`.`a`
         from `test`.`t2` where 1 has 1 ),
         <기본_인덱스_조회>(`test`.`t1`.`a` in
         <자동 키>의 <임시 테이블>
         where ((`test`.`t1`.`a` = `materialized-subquery`.`a`)))))
         test`.`t1`에서 `t1.a IN (SELECT t2.a FROM t2)` AS `t1.a
1행 1세트(0.00초)
```

경고 표시로 표시되는 문에는 쿼리 재작성 또는 최적화 프로그램 작업에 대한 정보를 제공하는 특수 마커가 포함될 수 있으므로 이 문은 반드시 유효한 SQL이 아니며 실행을 위한 것이 아닙니다. 출력에는 최적화 프로그램에서 수행한 작업에 대한 추가적인 비SQL 설명 메모를 제공하는 `Message` 값이 있는 행도 포함될 수 있습니다.

다음 목록에서는 경고 표시로 표시되는 확장 출력에 나타날 수 있는 특수 마커에 대해 설명합니다:

- `<auto_key>`

임시 테이블에 대해 자동으로 생성된 키입니다.

- <캐시>(EXPR)

  표현식(예: 스칼라 하위 쿼리)은 한 번 실행되고 결과 값은 나중에 사용할 수 있도록 메모리에 저장됩니다. 여러 값으로 구성된 결과의 경우 임시 테이블이 생성되고
  <임시 테이블>이 대신 표시됩니다.

- <존재>(*쿼리 조각*)

  하위 쿼리 술어가 EXISTS 술어로 변환되고 하위 쿼리가 EXISTS 술어와 함께 사용할 수 있도록 변환됩니다.

- <in_optimizer>(*쿼리 조각*)

  사용자가 중요하게 생각하지 않는 내부 옵티마이저 객체입니다.

- <index_lookup>(*쿼리 조각*)

  쿼리 조각은 인덱스 조회를 사용하여 적격 행을 찾기 위해 처리됩니다.

- <if>(*condition, expr1, expr2*)

  조건이 참이면 *expr1로* 평가하고, 그렇지 않으면 *expr2로* 평가합니다.

- <IS_NOT_NULL_TEST>(*EXPR*)

  표현식이 NULL로 평가되지 않는지 확인하는 테스트입니다.

- <materialize>(*쿼리 조각*)

  하위 쿼리 구체화가 사용됩니다.

- 구체화된-서브쿼리`.*col_name*

  하위 쿼리 평가 결과를 보유하기 위해 구체화된 내부 임시 테이블의 *col_name* 열에 대한 참조입니다.

- <기본_인덱스_조회>(*쿼리 조각*)

  쿼리 조각은 기본 키 조회를 사용하여 적격 행을 찾기 위해 처리됩니다.

- <ref_null_helper>(*expr*)

  사용자가 중요하게 생각하지 않는 내부 옵티마이저 객체입니다.

- */* *select#N* **/* *select_stmt*

  SELECT는 확장되지 않은 EXPLAIN 출력의 행에 연결되며, 이 행의 ID 값은 *N입니다*.

- *외부_테이블* 세미 조인(*내부_테이블*)

세미조인 작업. *inner_tables*는 끌어오지 않은 테이블을 표시합니다. 세미조인 변환을 사용하여 IN 및 EXISTS 하위 쿼리 조건 최적화를 참조하십시오.

- <임시 테이블>

  이는 중간 결과를 캐시하기 위해 생성된 내부 임시 테이블을 나타냅니다.

일부 테이블이 `const` 또는 시스템 유형인 경우 이러한 테이블의 열을 포함하는 표현식은 최적화 프로그램에 의해 조기에 평가되며 표시되는 문의 일부가 아닙니다. 그러나 FORMAT=JSON을 사용하면 일부 `const` 테이블 액세스가 const 값을 사용하는 참조 액세스로 표시됩니다.

8.8.4 명명된 연결에 대한 실행 계획 정보 얻기

명명된 연결에서 실행되는 설명 가능한 문에 대한 실행 계획을 얻으려면 이 문을 사용합니다:

```
연결 연결_id에 대한 [옵션] 설명;
```

연결에 대한 설명은 주어진 연결에서 쿼리를 실행하는 데 현재 사용되고 있는 설명 정보를 반환합니다. 데이터(및 지원 통계)의 변경으로 인해 동일한 쿼리 텍스트에 대해 EXPLAIN을 실행하는 것과 다른 결과가 나올 수 있습니다. 이러한 동작 차이는 일시적인 성능 문제를 진단하는 데 유용할 수 있습니다. 예를 들어 한 세션에서 완료하는 데 시간이 오래 걸리는 문을 실행하는 경우 다른 세션에서 연결에 대한 설명을 사용하면 지연의 원인에 대한 유용한 정보를 얻을 수 있습니다.

connection_id는 정보_스켐마 프로세스 목록 테이블 또는 SHOW 프로세스 목록 문에서 얻은 연결 식별자입니다. PROCESS 권한이 있는 경우 모든 연결에 대해 식별자를 지정할 수 있습니다. 그렇지 않으면 자신의 연결에 대해서만 식별자를 지정할 수 있습니다. 모든 경우에 지정된 연결에 대한 쿼리를 설명할 수 있는 충분한 권한이 있어야 합니다.

명명된 연결에서 문을 실행하지 않는 경우 결과는 비어 있습니다. 그렇지 않으면 명명된 연결에서 실행 중인 문이 설명 가능한 경우에만 연결에 대한 설명이 적용됩니다. 여기에는 SELECT, DELETE, INSERT, REPLACE 및 UPDATE가 포함됩니다. (단, 연결에 대한 EXPLAIN
는 준비된 명령문, 심지어 이러한 유형의 준비된 명령문에도 작동하지 않습니다.)

명명된 연결이 설명 가능한 문을 실행하는 경우, 출력은 문 자체에 EXPLAIN을 사용하면 얻을 수 있는 결과입니다.

명명된 연결이 설명할 수 없는 문을 실행하는 경우 오류가 발생합니다. 예를 들어 EXPLAIN은 설명할 수 없으므로 현재 세션의 연결 식별자 이름을 지정할 수 없습니다:

```
mysql> SELECT CONNECTION_ID();
+-----------------+
| connection_id() |
+-----------------+
              |373 |
+-----------------+
1행 1세트(0.00초)

mysql> 연결에 대한 설명 373;
오류 1889 (HY000): 연결에 대한 설명 명령은 SELECT/업데이트/삽입/삭제/교체
에 대해서만 지원됩니다.
```

Com_explain_other 상태 변수는 연결에 대한 설명의 수를 나타냅니다.
문을 실행합니다.

8.8.5 쿼리 성능 추정

대부분의 경우, 디스크 탐색 횟수를 계산하여 쿼리 성능을 추정할 수 있습니다. 작은 테이블의 경우, 일반적으로 인덱스가 캐시되어 있을 가능성이 높기 때문에 한 번의 디스크 탐색으로 행을 찾을 수 있습니다. 더 큰 테이

블의 경우, B-트리 인덱스를 사용하면 한 행을 찾는 데 다음과 같은 횟수의 탐색이 필요하다고 추정할 수 있습니다: `log(row_count) / log(index_block_length / 3 * 2 / (index_length + data_pointer_length)) + 1`.

MySQL에서 인덱스 블록은 일반적으로 1,024바이트이고 데이터 포인터는 일반적으로 4바이트입니다. 키 값 길이가 3바이트(`MEDIUMINT` 크기)인 500,000행 테이블의 경우, 수식은 `log(500,000)/log(1024/3*2/(3+4))`를 나타냅니다. `+ 1` = 4입니다.

이 인덱스에는 약 500,000 * 7 * 3/2 = 5.2MB의 저장 공간이 필요하므로(일반적인 인덱스 버퍼 채우기 비율이 2/3이라고 가정), 메모리에 인덱스의 대부분이 있으므로 행을 찾기 위해 데이터를 읽는 데 한두 번만 호출하면 될 것입니다.

그러나 쓰기의 경우 새 인덱스 값을 배치할 위치를 찾기 위해 네 번의 탐색 요청이 필요하며, 일반적으로 인덱스를 업데이트하고 행을 쓰기 위해 두 번의 탐색 요청이 필요합니다.

앞의 논의는 애플리케이션 성능이 로그 *N만큼* 서서히 저하된다는 것을 의미하지 않습니다. 모든 것이 OS 또는 MySQL 서버에 의해 캐시되는 한, 상황은 약간만 악화됩니다.
테이블이 커질수록 속도가 느려집니다. 데이터가 너무 커져서 캐시할 수 없게 되면 속도가 훨씬 느려지기 시작합니다. 를 사용하여 애플리케이션이 디스크 검색(로그 *N만큼* 증가)에만 바인딩될 때까지 기다리세요. 이를 방지하려면 데이터가 증가함에 따라 키 캐시 크기를 늘리세요. `MyISAM` 테이블의 경우 키 캐시 크기는 `key_buffer_size` 시스템 변수에 의해 제어됩니다. 5.1.1절, "서버 구성"을 참조하세요.

8.9 쿼리 최적화 도구 제어

MySQL은 쿼리 계획이 평가되는 방식에 영향을 미치는 시스템 변수, 전환 가능한 최적화, 최적화 및 인덱스 힌트, 최적화 비용 모델을 통해 최적화 도구를 제어할 수 있습니다.

서버는 `column_statistics` 데이터 사전 테이블에서 열 값에 대한 히스토그램 통계를 유지 관리합니다(섹션 8.9.6, "최적화 도구 통계" 참조). 다른 데이터 사전 테이블과 마찬가지로 이 테이블은 사용자가 직접 액세스할 수 없습니다. 대신 데이터에 대한 보기로 구현된 `INFORMATION_SCHEMA.COLUMN_STATISTICS`를 쿼리하여 히스토그램 정보를 얻을 수 있습니다.
사전 테이블을 사용합니다. `ANALYZE TABLE` 문을 사용하여 히스토그램 관리를 수행할 수도 있습니다.

8.9.1 쿼리 계획 평가 제어

쿼리 최적화 도구의 임무는 SQL 쿼리를 실행하기 위한 최적의 계획을 찾는 것입니다. "좋은" 계획과 "나쁜" 계획의 성능 차이는 몇 초 또는 몇 시간 또는 며칠에 불과할 수 있기 때문에 MySQL을 포함한 대부분의 쿼리 최적화 도구는 가능한 모든 쿼리 평가 계획 중에서 최적의 계획을 찾기 위해 어느 정도 철저한 검색을 수행합니다. 조인 쿼리의 경우, MySQL 최적화 도구가 조사하는 가능한 계획의 수는 쿼리에서 참조되는 테이블의 수에 따라 기하급수적으로 증가합니다. 테이블 수가 적은 경우(일반적으로 7~10개 미만) 이는 문제가 되지 않습니다. 그러나 더 큰 쿼리가 제출되면 쿼리 최적화에 소요되는 시간이 서버 성능의 주요 병목 현상이 될 수 있습니다.

쿼리 최적화를 위한 보다 유연한 방법을 사용하면 사용자가 최적화 도구가 최적의 쿼리 평가 계획을 얼마나 철저하게 검색할지 제어할 수 있습니다. 일반적으로 옵티마이저가 조사하는 계획이 적을수록 쿼리를 컴파일하는 데 소요되는 시간이 줄어듭니다. 반면에 최적화 도구가 일부 계획을 건너뛰기 때문에 최적의 계획을 찾지 못할 수도 있습니다.

평가하는 플랜 수에 대한 옵티마이저의 동작은 두 가지 시스템 변수를 사용하여 제어할 수 있습니다:

- `optimizer_prune_level` 변수는 각 테이블에 액세스하는 행 수에 대한 추정치를 기반으로 특정 계획을 건너뛰도록 옵티마이저에 지시합니다. 경험에 따르면 이러한 종류의 '교육적 추측'은 최적의 계획을 놓치는 경우가 거의 없으며 쿼리 컴파일 시간을 크게 줄일 수 있습니다. 이것이 바로 이 옵션이 기본적

으로 켜져 있는 이유입니다(optimizer_prune_level=1). 하지만,
옵티마이저가 더 나은 쿼리 계획을 놓쳤다고 생각되면 이 옵션을 끌 수 있습니다.
(optimizer_prune_level=0)을 사용하면 쿼리 컴파일 시간이 훨씬 더 오래 걸릴 수 있습니다. 이 휴리스틱을 사용하더라도 옵티마이저는 여전히 대략 기하급수적인 수의 계획을 탐색한다는 점에 유의하세요.

- optimizer_search_depth 변수는 옵티마이저가 각 불완전한 계획의 '미래'를 얼마나 더 확장해야 하는지 평가해야 하는지를 알려줍니다. optimizer_search_depth 값이 작을수록 쿼리 컴파일 시간이 훨씬 더 짧아질 수 있습니다. 예를 들어, 테이블이 12개, 13개 또는 그 이상인 쿼리의 경우 optimizer_search_depth가 쿼리의 테이블 수에 가까울 경우 컴파일하는 데 몇 시간, 심지어 며칠이 걸릴 수도 있습니다. 동시에 optimizer_search_depth를 3 또는 4로 컴파일하면 옵티마이저는 다음과 같이 컴파일할 수 있습니다.
  로 설정하면 동일한 쿼리에 대해 1분 이내에 검색 결과를 얻을 수 있습니다.
  optimizer_search_depth의 적정 값이 무엇인지 확실하지 않은 경우 이 변수를 0으로 설정하여 최적화 도구가 자동으로 값을 결정하도록 할 수 있습니다.

8.9.2 전환 가능한 최적화

optimizer_switch 시스템 변수를 사용하면 옵티마이저 동작을 제어할 수 있습니다. 이 변수의 값은 플래그 집합으로, 각 플래그는 해당 옵티마이저 동작의 활성화 또는 비활성화 여부를 나타내는 켜짐 또는 꺼짐 값을 갖습니다. 이 변수에는 전역 및 세션 값이 있으며 런타임에 변경할 수 있습니다.

글로벌 기본값은 서버를 시작할 때 설정할 수 있습니다.

현재 옵티마이저 플래그 세트를 확인하려면 변수 값을 선택합니다:

```
mysql> SELECT @@optimizer_switch\G
*************************** 1. row ***************************
@@optimizer_switch: index_merge=on,index_merge_union=on,
                    index_merge_sort_union=on,index_merge_intersection=on,
                    engine_condition_pushdown=on,index_condition_pushdown=on,
                    mrr=on,mrr_cost_based=on,block_nested_loop=on,
                    batched_key_access=off,materialization=on,semijoin=on,
                    looscan=on,firstmatch=on,duplicateweedout=on,
                    subquery_materialization_cost_based=on, use_index_extensions=on,
                    condition_fanout_filter=on, derived_merge=on, use_invisible_indexes=off,
                    skip_scan=on, hash_join=on,subquery_to_derived=off,
                    prefer_ordering_index=on,hypergraph_optimizer=off,
                    derived_condition_pushdown=on,hash_set_operations=on
한 세트에 1행(0.00초)
```

optimizer_switch의 값을 변경하려면 쉼표로 구분된 하나 이상의 명령어 목록으로 구성된 값을 지정합니다:

```
SET [GLOBAL|SESSION] optimizer_switch='command[,command]...';
```

각 *명령* 값은 다음 표에 표시된 형식 중 하나를 사용해야 합니다.

명령 구문	의미
기본값	모든 최적화를 기본값으로 재설정
opt_name=기본값	명명된 최적화를 기본값으로 설정합니다.
opt_name=off	명명된 최적화 비활성화
opt_name=on	명명된 최적화 사용

값에 포함된 명령의 순서는 중요하지 않지만 기본 명령이 있는 경우 기본 명령이 먼저 실행됩니다. *opt_name* 플래그를 기본값으로 설정하면 켜짐 또는 꺼짐 중 기본값이 되는값으로 설정됩니다. 값에 지정된 *opt_name* *을 두* 번 이상 지정하는 것은 허용되지 않으며 다음과 같은 오류가 발생합니다.
오류를 반환합니다. 값에 오류가 있으면 할당이 오류와 함께 실패하고 다음과 같은 값이 남습니다.
optimizer_switch 변경되지 않았습니다.

다음 목록은 허용되는 *opt_name* 플래그 이름을 최적화 전략별로 그룹화하여 설명합니다:

- 배치 키 액세스 플래그
  - batched_key_access(기본값 꺼짐)

    BKA 조인 알고리즘의 사용을 제어합니

다.

batched_key_access가 켜짐으로 설정되었을 때 효과를 발휘하려면 mrr 플래그도 켜져 있어야 합니다. 현재 MRR에 대한 비용 추정이 너무 비관적입니다. 따라서 BKA를 사용하려면 mrr_cost_based도 꺼져 있어야 합니다.

자세한 내용은 섹션 8.2.1.12, "중첩 루프 및 일괄 키 액세스 조인 차단"을 참조하세요.

- 중첩 루프 플래그 차단
  - 블록 네스티드 루프(기본값 켜짐)

    해시 조인 사용을 제어하며, BNL 및 NO_BNL 옵티마이저 힌트와 마찬가지로 해시 조인 사용을 제어합니다.

자세한 내용은 섹션 8.2.1.12, "중첩 루프 및 일괄 키 액세스 조인 차단"을 참조하세요.

- 조건 필터링 플래그

  - 조건_팬아웃_필터(기본값 켜짐) 조건 필터링

    사용을 제어합니다.

  자세한 내용은 섹션 8.2.1.13, '조건 필터링'을 참조하세요.

- 파생 조건 푸시다운 플래그

  - 파생 조건 푸시다운(기본값 켜짐) 파생 조건 푸시다

    운을 제어합니다.

  자세한 내용은 섹션 8.2.2.5, "파생된 조건 푸시다운 최적화"를 참조하세요.

- 파생 테이블 병합 플래그

  - 파생_병합(기본값 켜짐)

    파생된 테이블 및 뷰를 외부 쿼리 블록으로 병합하는 것을 제어합니다.

  파생 테이블, 뷰 참조 및 공통 테이블 식을 병합을 방지하는 다른 규칙이 없다고 가정할 때, 파생 테이블, 뷰 참조 및 공통 테이블 식을 외부 쿼리 블록에 병합할지 여부를 옵티마이저가 제어합니다(예: 뷰에 대한 `ALGORITHM` 지시문이 파생된_merge 설정보다 우선함). 기본적으로 이 플래그는 병합을 사용하도록 설정되어 있습니다.

  자세한 내용은 섹션 8.2.2.4, '병합 또는 구체화를 사용하여 파생 테이블, 뷰 참조 및 일반 테이블 표현식 최적화'를 참조하십시오.

- 엔진 상태 푸시다운 플래그

  - 엔진 컨디션 푸시다운(기본값 켜짐) 엔진 컨디션

    푸시다운을 제어합니다.

  자세한 내용은 섹션 8.2.1.5, "엔진 상태 푸시다운 최적화"를 참조하세요.

- 해시 조인 플래그

  - 해시 조인(기본값 켜짐)

    MySQL 8.2에서는 효과가 없습니다. 대신 `block_nested_loop` 플래그를 사

  용하세요. 자세한 내용은 섹션 8.2.1.4, "해시 조인 최적화"를 참조하세요.

- 인덱스 조건 푸시다운 플래그

  - `index_condition_pushdown`(기본값 켜짐

) 인덱스 조건 푸시다운을 제어합니다.

  자세한 내용은 섹션 8.2.1.6, "인덱스 조건 푸시다운 최적화"를 참조하세요.

- 색인 확장 플래그

  - `use_index_extensions`(기본값 켜짐

) 인덱스 확장 사용을 제어합니다.

  자세한 내용은 섹션 8.3.10, '인덱스 확장 사용'을 참조하세요.

- 인덱스 병합 플래그

  - `index_merge`(기본값 켜짐)

    모든 인덱스 병합 최적화를 제어합니다.

  - `index_merge_intersection`(기본값 켜짐)

    인덱스 병합 교차점 액세스 최적화를 제어합니다.

  - `index_merge_sort_union`(기본값 켜짐)

    인덱스 병합 정렬-연합 액세스 최적화를 제어합니다.

  - `index_merge_union`(기본값 켜짐)

    인덱스 병합 유니온 액세스 최적화를 제어합니다.

  자세한 내용은 섹션 8.2.1.3, "인덱스 병합 최적화"를 참조하세요.

- 인덱스 가시성 플래그

  - `use_invisible_indexes`(기본값 꺼짐)

    보이지 않는 인덱스의 사용을 제어합니다.

  자세한 내용은 섹션 8.3.12, "보이지 않는 색인"을 참조하세요.

- 최적화 플래그 제한

  - 선호 주문 인덱스(기본값 켜짐)

    쿼리에 `LIMIT` 절이 있는 `ORDER BY` 또는 `GROUP BY`가 있는 경우 옵티마이저가 정렬되지 않은 인덱스, 파일 정렬 또는 다른 최적화 대신 정렬된 인덱스를 사용할지 여부를 제어합니다. 이 최적화는 기본적으로 옵티마이저가 이 최적화를 사용하면 쿼리를 더 빠르게 실행할 수 있다고 판단할 때마다 수행됩니다.

    이 결정을 내리는 알고리즘은 데이터 분포가 항상 어느 정도 균일하다고 가정하기 때문에 생각할 수 있는 모든 경우를 처리할 수 없기 때문에 이 최적화가 바람직하지 않은 경우가 있을 수 있습니다. 선호 주문 인덱스 플래그를 `off`로 설정하면 이 최적화를 비활성화할 수 있습니다.

  자세한 내용과 예제는 섹션 8.2.1.19, "쿼리 최적화 제한"을 참조하세요.

- 다중 범위 읽기 플래그

  - `MRR`(기본값 켜짐)

    다중 범위 읽기 전략을 제어합니다.

- `MRR_COST_BASED`(기본값 켜짐)

  `mrr=온인` 경우 비용 기반 MRR 사용을 제어합니다.

자세한 내용은 섹션 8.2.1.11, "다중 범위 읽기 최적화"를 참조하세요.

- 세미 조인 플래그
  - 중복 제거(기본값 켜짐)

    세미조인 중복 위드아웃 전략을 제어합니다.

  - 첫 번째 일치(기본값 켜짐)

세미조인 퍼스트매치 전략을 제어합니다.

- 루즈 스캔(기본값 켜짐)

  세미조인 루즈스캔 전략을 제어합니다(그룹별 루즈 인덱스 스캔과 혼동하지 마세요).

- 세미조인(기본값 켜짐) 모든 세미

  조인 전략을 제어합니다.

  이는 안티조인 최적화에도 적용됩니다.

세미조인, 퍼스트매치, 루즈스캔, 중복 위드아웃 플래그를 사용하면 세미조인 전략을 제어할 수 있습니다. 세미조인 플래그는 세미조인 사용 여부를 제어합니다. 이 플래그를 켜짐으로 설정하면 첫 번째 일치 및 루즈스캔 플래그를 통해 허용된 세미조인 전략을 보다 세밀하게 제어할 수 있습니다.

중복 위드아웃 세미조인 전략이 비활성화되어 있으면 다른 모든 해당 전략도 비활성화하지 않는 한 사용되지 않습니다.

세미조인과 구체화가 모두 켜져 있으면 세미조인은 해당되는 경우 구체화도 사용합니다. 이 플래그는 기본적으로 켜져 있습니다.

자세한 내용은 준조인 변환을 사용하여 IN 및 EXISTS 하위 쿼리 조건 최적화를 참조하십시오.

- 작업 플래그 설정

  - 해시_설정_작업(기본값 켜짐)

    `EXCEPT` 및 `INTERSECT`를 포함하는 집합 연산에 대해 해시 테이블 최적화를 활성화합니다(기본적으로 활성화됨). 그렇지 않으면 이전 버전의 MySQL에서와 같이 임시 테이블 기반 중복 제거가 사용됩니다.

    이 최적화에 의해 해싱에 사용되는 메모리 양은 `set_operations_buffer_size` 시스템 변수를 사용하여 제어할 수 있으며, 일반적으로 이 값을 높이면 이러한 연산을 사용하는 문의 실행 시간이 빨라집니다.

    MySQL 8.2.0에 추가되었습니다.

- 스캔 플래그 건너뛰기

  - `SKIP_SCAN`(기본값 켜짐)

    스캔 건너뛰기 액세스 방법 사용을 제어합니다.

자세한 내용은 스캔 범위 액세스 방법 건너뛰기를 참조하세요.

- 서브쿼리 구체화 플래그

  - 구체화(기본값 켜짐)

    머티리얼라이제이션(세미조인 머티리얼라이제이션 포함)을 제어합니다.

  - subquery_materialization_cost_based (기본값 켜짐

) 비용 기반 구체화 선택 사항 사용.

  구체화 플래그는 하위 쿼리 구체화 사용 여부를 제어합니다. 세미조인과 구체화가 모두 켜져 있으면 세미조인도 해당되는 경우 구체화를 사용합니다. 이 플래그는 기본적으로 켜져 있습니다.

  subquery_materialization_cost_based 플래그를 사용하면 서브쿼리 구체화와 IN-to-EXISTS 서브쿼리 변환 간의 선택에 대한 제어가 가능합니다. 플래그가 켜져 있으면(기본값), 옵티마이저는 두 방법 중 하나를 사용할 수 있는 경우 서브쿼리 구체화와 IN-to-EXISTS 서브쿼리 변환 중 비용 기반 선택을 수행합니다. 플래그가 꺼져 있으면 옵티마이저는 IN-to-EXISTS 하위 쿼리 변환 대신 하위 쿼리 구체화를 선택합니다.

  자세한 내용은 섹션 8.2.2, "하위 쿼리, 파생 테이블, 뷰 참조 및 일반 테이블 식 최적화"를 참조하십시오.

- 하위 쿼리 변환 플래그

  - 서브쿼리_투_파생(기본값 꺼짐)

    옵티마이저는 많은 경우 SELECT, WHERE, JOIN 또는 HAVING 절의 스칼라 하위 쿼리를 파생된 테이블의 왼쪽 외부 조인으로 변환할 수 있습니다. (파생 테이블의 null 가능성에 따라 이 작업을 내부 조인으로 더 단순화할 수도 있습니다.) 이 작업은 다음 조건을 충족하는 하위 쿼리에 대해 수행할 수 있습니다:

    - 하위 쿼리는 RAND() 와 같은 비결정론적 함수를 사용하지 않습니다.
    - 이 하위 쿼리는 MIN() 또는 MAX()를 사용하도록 다시 작성할 수 있는 ANY 또는 ALL 하위 쿼리가 아닙니다.
    - 사용자 변수를 다시 작성하면 실행 순서에 영향을 미칠 수 있으므로 동일한 쿼리에서 변수에 두 번 이상 액세스하는 경우 예기치 않은 결과가 발생할 수 있으므로 상위 쿼리에서는 사용자 변수를 설정하지 않습니다.
    - 하위 쿼리는 상호 연관되어서는 안 됩니다. 즉, 외부 쿼리에 있는 테이블의 열을 참조하거나 외부 쿼리에서 평가되는 집계를 포함하지 않아야 합니다.

    이 최적화는 GROUP BY를 포함하지 않는 IN, NOT IN, EXISTS 또는 NOT EXISTS의 인수가 되는 테이블 하위 쿼리에도 적용할 수 있습니다.

    대부분의 경우 이 최적화를 활성화해도 성능이 눈에 띄게 개선되지 않고 쿼리가 더 느리게 실행될 수 있기

때문에 이 플래그의 기본값은 꺼져 있지만, `subquery_to_derived` 플래그를 켜기로 설정하여 최적화를 활성화할 수 있습니다. 이 플래그는 주로 테스트에 사용하기 위한 것입니다.

스칼라 하위 쿼리를 사용하는 예입니다:

```
d
mysql> CREATE TABLE t1(a INT);

mysql> CREATE TABLE t2(a INT);

mysql> INSERT INTO t1 VALUES ROW(1), ROW(2), ROW(3), ROW(4;)

mysql> INSERT INTO t2 VALUES ROW(1), ROW(2;)

mysql> SELECT * FROM t1
```

```
    ->     WHERE t1.a > (SELECT COUNT(a) FROM t2);
+------+
| a    |
+------+
|3 |
|4 |
+------+

mysql> SELECT @@optimizer_switch LIKE '%subquery_to_derived=off%';
+-----------------------------------------------------+
| @@optimizer_switch LIKE '%subquery_to_derived=off%' | @@옵티마이저_스위치
+-----------------------------------------------------+
|1 |
+-----------------------------------------------------+

mysql> EXPLAIN SELECT * FROM t1 WHERE t1.a > (SELECT COUNT(a) FROM t2)\G
*************************** 1. 행 ***************************
           ID: 1
  select_type: PRIMARY
        테이블: t1
   파티션: NULL
         유형: 모든 가능한
_키: NULL
          키: NULL
      key_len: NULL
          참조: NULL 행:
         4
     필터링되었습니다: 33.33
        추가: 어디 사용
*************************** 2. 행 ***************************
           ID: 2
  select_type: SUBQUERY
        테이블: t2
   파티션: NULL
         유형: 모든 가능한
_키: NULL
          키: NULL
      key_len: NULL
          참조: NULL 행:
         2
     필터링됨: 100.00 추가
        : NULL

mysql> SET @@optimizer_switch='subquery_to_derived=on';


mysql> SELECT @@optimizer_switch LIKE '%subquery_to_derived=off%';
+-----------------------------------------------------+
| @@optimizer_switch LIKE '%subquery_to_derived=off%' | @@옵티마이저_스위치
+-----------------------------------------------------+
|0 |
+-----------------------------------------------------+

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

mysql> EXPLAIN SELECT * FROM t1 WHERE t1.a > (SELECT COUNT(a) FROM t2)\G
*************************** 1. 행 ***************************
```

```
           ID: 1
  select_type: PRIMARY 테이
        블: <derived2>
   파티션: NULL
         유형: 모든 가능한
_키: NULL
          키입니다: NULL
      key_len: NULL
          참조: NULL 행:
         1
```

```
     필터링됨: 100.00 추가:
        NULL
*************************** 2. 행 ***************************
           ID: 1
  select_type: PRIMARY 테이
         블: t1
   파티션: NULL
         유형: 모든
가능한_키: NULL
          키: NULL
      key_len: NULL
          참조:
         NULL 행: 4
     필터링되었습니다: 33.33
        추가: 사용 위치; 조인 버퍼 사용(해시 조인)
*************************** 3. 행 ***************************
           ID: 2
  select_type: 파생 테이블:
        t2
   파티션: NULL
         유형: 모든
가능한_키: NULL
          키: NULL
      key_len: NULL
          참조:
         NULL 행: 2
     필터링됨: 100.00 추가:
        NULL
```

두 번째 `EXPLAIN` 문 바로 뒤에 `SHOW` WARNINGS를 실행하면 알 수 있듯이 최적화가 활성화된 경우 `SELECT * FROM t1 WHERE t1.a > (SELECT COUNT(a) FROM t2)` 쿼리가 여기에 표시된 것과 유사한 형태로 재작성됩니다:

```
SELECT t1.a FROM t1
    JOIN ( SELECT COUNT(t2.a) AS c FROM t2 ) AS d
            WHERE t1.a > dc;
```

예: `IN(`*하위 쿼리*`)이` 포함된 쿼리 사용:

```
mysql> DROP TABLE IF EXISTS t1, t2;

mysql> CREATE TABLE t1 (a INT, b INT);
mysql> CREATE TABLE t2 (a INT, b INT);

mysql> INSERT INTO t1 VALUES ROW(1,10), ROW(2,20), ROW(3,30);
mysql> INSERT INTO t2
    ->    값 행(1,10), 행(2,20), 행(3,30), 행(1,110), 행(2,120), 행(3,130);

mysql> SELECT * FROM t1
    ->     WHERE t1 .b < 0
    ->             OR
    ->             t1.a IN (SELECT t2.a + 1 FROM t2);
+------+ ------+
| a| b  |
+------+ ------+
|2    |20 |
|3    |30 |
+------+ ------+

mysql> SET @@optimizer_switch="subquery_to_derived=off";

mysql> EXPLAIN SELECT * FROM t1
    ->             WHERE t1 .b < 0
    ->                     OR
    ->                     t1.a IN (SELECT t2.a + 1 FROM t2)\G
*************************** 1. 행 ***************************
           ID: 1
  select_type: PRIMARY 테이
        블: t1
   파티션: NULL
         유형: ALL
```

```
가능한_키: NULL
          키: NULL
      key_len: NULL
          참조: NULL 행:
         3
     필터링됨: 100.00 추가: 사
        용 위치
*************************** 2. 행 ***************************
           ID: 2
  SELECT_TYPE: 종속 하위 쿼리 테이블: t2
   파티션: NULL
         유형: 모든 가능한
_키: NULL
          키: NULL
      key_len: NULL
          참조: NULL 행:
         6
     필터링됨: 100.00 추가: 사
        용 위치

mysql> SET @@optimizer_switch="subquery_to_derived=on";

mysql> EXPLAIN SELECT * FROM t1
    ->              WHERE t1 .b < 0
    ->                     OR
    ->                     t1.a IN (SELECT t2.a + 1 FROM t2)\G
*************************** 1. 행 ***************************
           ID: 1
  select_type: PRIMARY
        테이블: t1
   파티션: NULL
         유형: 모든 가능한
_키: NULL
          키: NULL
      key_len: NULL
          참조: NULL 행:
         3
     필터링됨: 100.00 추가
        : NULL
*************************** 2. 행 ***************************
           ID: 1
  select_type: PRIMARY 테이
        블: <derived2>
   파티션: NULL
         유형: 참조 가능한_키:
<auto_key0>
          키: <auto_key0>
      키_len: 9
          참조: STD2.T1.A 행
         : 2
     필터링됨: 100.00
        추가: 어디 사용; 인덱스 사용
*************************** 3. 행 ***************************
           ID: 2
  select_type: 파생 테이
```

```
        블: t2
   파티션: NULL
        유형: 모든 가능한
_키: NULL
         키: NULL
     key_len: NULL
         참조: NULL 행:
        6
    필터링됨: 100.00
       추가: 임시 사용
```

이 쿼리에 대해 `EXPLAIN`을 실행한 후 `SHOW WARNINGS`의 결과를 확인하고 단순화하면 `subquery_to_derived` 플래그가 활성화된 경우 `SELECT * FROM t1 WHERE t1.b`가 표시됩니다.

`< 0 OR t1.a IN (SELECT t2.a + 1 FROM t2)`는 여기에 표시된 것과 유사한 형태로 재작성됩니다:

```
SELECT a, b FROM t1
    좌측 조인 (SELECT DISTINCT a + 1 AS e FROM t2) d
    ON t1.a = d.e
    WHERE t1.b < 0
            또는
            d.e는 NULL이 아닙니다;
```

예를 들어, 이전 예와 동일한 테이블 및 데이터에 `EXISTS(`*하위 쿼리*`)`가 있는 쿼리를 사용합니다:

```
mysql> SELECT * FROM t1
    ->     WHERE t1 .b < 0
    ->             OR
    ->             EXISTS(SELECT * FROM t2 WHERE t2.a = t1.a + 1);
+------+ ------+
| a| b  |
+------+ ------+
|1    |10 |
|2    |20 |
+------+ ------+

mysql> SET @@optimizer_switch="subquery_to_derived=off";

mysql> EXPLAIN SELECT * FROM t1
    ->             WHERE t1 .b < 0
    ->                     OR
    ->                     EXISTS(SELECT * FROM t2 WHERE t2.a = t1.a + 1)\G
*************************** 1. 행 ***************************
           ID: 1
  select_type: PRIMARY 테이
        블: t1
   파티션: NULL
         유형: 모든
가능한_키: NULL
          키: NULL
      key_len: NULL
          참조:
         NULL 행: 3
     필터링됨: 100.00 추가:
        사용 위치
*************************** 2. 행 ***************************
           ID: 2
  SELECT_TYPE: 종속 하위 쿼리 테이블: t2
   파티션: NULL
         유형: 모든
가능한_키: NULL
          키입니다:
      NULL key_len:
      NULL
          참조:
         NULL 행: 6
     필터링됨: 16.67 추가: 어
        디 사용

mysql> SET @@optimizer_switch="subquery_to_derived=on";

mysql> EXPLAIN SELECT * FROM t1
    ->             WHERE t1 .b < 0
    ->                     OR
    ->                     EXISTS(SELECT * FROM t2 WHERE t2.a = t1.a + 1)\G
*************************** 1. 행 ***************************
           ID: 1
  select_type: PRIMARY 테이
        블: t1
   파티션: NULL
         유형: 모든
가능한_키: NULL
          키입니다: NULL
```

```
      key_len: NULL
          참조:
         NULL 행: 3
     필터링됨: 100.00 추가:
        NULL
*************************** 2. 행 ***************************
           ID: 1
  select_type: PRIMARY 테이
        블: <derived2>
   파티션: NULL
         유형: 모든
가능한_키: NULL
          키: NULL
      key_len: NULL
          참조:
         NULL 행: 6
     필터링됨: 100.00
        추가: 사용 위치; 조인 버퍼 사용(해시 조인)
*************************** 3. 행 ***************************
           ID: 2
  select_type: 파생 테이블:
        t2
   파티션: NULL
         유형: 모든
가능한_키: NULL
          키: NULL
      key_len: NULL
          참조:
         NULL 행: 6
     필터링됨: 100.00
        추가: 임시 사용
```

SELECT * FROM t1 WHERE t1.b < 0 OR EXISTS(SELECT * FROM t2 WHERE t2.a = t1.a + 1) 쿼리에서 EXPLAIN을 실행한 후 SHOW WARNINGS를 실행하는 경우 다음과 같은 경우

subquery_to_derived를 활성화하고 결과의 두 번째 행을 단순화하면 이와 유사한 형태로 재작성된 것을 확인할 수 있습니다:

```
SELECT a, b FROM t1
좌측 조인 (SELECT DISTINCT 1 AS e1, t2.a AS e2 FROM t2) d
ON t1.a + 1 = d.e2
WHERET1.b < 0
        또는
        d.e1은 NULL이 아닙니다;
```

자세한 내용은 섹션 8.2.2.4, '병합 또는 구체화를 사용하여 파생 테이블, 뷰 참조 및 일반 테이블 식 최적화' 및 섹션 8.2.1.19, '제한 쿼리 최적화', '준조인 변환을 사용하여 IN 및 EXISTS 하위 쿼리 예측 조건 최적화'를 참조하세요.

optimizer_switch에 값을 할당하면 언급되지 않은 플래그는 현재 값을 유지합니다. 따라서 다른 동작에 영

향을 주지 않고 단일 문에서 특정 옵티마이저 동작을 활성화 또는 비활성화할 수 있습니다. 이 문은 다른 옵티마이저 플래그의 존재 여부와 그 값에 의존하지 않습니다. 모든 인덱스 병합 최적화가 활성화되어 있다고 가정합니다:

```
mysql> SELECT @@optimizer_switch\G
*************************** 1. row ***************************
@@optimizer_switch: index_merge=on,index_merge_union=on,
                    index_merge_sort_union=on,index_merge_intersection=on,engi
                    ne_condition_pushdown=on,index_condition_pushdown=on,mrr=o
                    n,mrr_cost_based=on,block_nested_loop=on,
                    batched_key_access=off,materialization=on,semijoin=on,
                    loosescan=on, firstmatch=on,
                    subquery_materialization_cost_based=on,
                    use_index_extensions=on, condition_fanout_filter=on,
                    derived_merge=on, use_invisible_indexes=off, skip_scan=on,
                    hash_join=on,subquery_to_derived=off,
                    prefer_ordering_index=on
```

서버가 특정 쿼리에 인덱스 병합 유니온 또는 인덱스 병합 정렬 유니온 액세스 메서드를 사용 중이고 옵티마이저가 이 메서드 없이 더 나은 성능을 낼 수 있는지 확인하려면 다음과 같이 변수 값을 설정하세요:

```
mysql> SET optimizer_switch='index_merge_union=off,index_merge_sort_union=off';

mysql> SELECT @@optimizer_switch\G
*************************** 1. row ***************************
@@optimizer_switch: index_merge=on,index_merge_union=off,
                    index_merge_sort_union=off,index_merge_intersection=on,eng
                    ine_condition_pushdown=on,index_condition_pushdown=on,mrr=
                    on,mrr_cost_based=on,block_nested_loop=on,
                    batched_key_access=off,materialization=on,semijoin=on,
                    loosescan=on, firstmatch=on,
                    subquery_materialization_cost_based=on,
                    use_index_extensions=on, condition_fanout_filter=on,
                    derived_merge=on, use_invisible_indexes=off, skip_scan=on,
                    hash_join=on,subquery_to_derived=off,
                    prefer_ordering_index=on
```

8.9.3 최적화 도구 힌트

최적화 전략을 제어하는 한 가지 방법은 `optimizer_switch` 시스템 변수를 설정하는 것입니다(섹션 8.9.2, "전환 가능한 최적화" 참조). 이 변수를 변경하면 이후의 모든 쿼리 실행에 영향을 미치므로, 한 쿼리와 다른 쿼리에 다르게 영향을 미치려면 각 쿼리 전에 `optimizer_switch`를 변경해야 합니다.

옵티마이저를 제어하는 또 다른 방법은 개별 문 내에서 지정할 수 있는 옵티마이저 힌트를 사용하는 것입니다. 옵티마이저 힌트는 문 단위로 적용되므로 `optimizer_switch`를 사용할 때보다 문 실행 계획을 더 세밀하게 제어할 수 있습니다. 예를 들어, 한 문에서 한 테이블에 대한 최적화를 활성화하고 다른 테이블에 대한 최적화를 비활성화할 수 있습니다.

문 내의 힌트는 `optimizer_switch` 플래그보다 우선합니다. 예제:

```
SELECT /*+ NO_RANGE_OPTIMIZATION(t3 PRIMARY, f2_idx) */ f1
  FROM t3 WHERE f1 > 30 AND f1 < 33;
SELECT /*+ BKA(t1) NO_BKA(t2) */ * FROM t1 INNER JOIN t2 WHERE ...;
SELECT /*+ NO_ICP(t1, t2) */ * FROM t1 INNER JOIN t2 WHERE ....;
SELECT /*+ SEMIJOIN(FIRSTMATCH, LOOSESCAN) */ * FROM t1 ...;
EXPLAIN SELECT /*+ NO_ICP(t1) */ * FROM t1 WHERE ...;
SELECT /*+ MERGE(dt) */ * FROM (SELECT * FROM t1) AS dt;
INSERT /*+ SET_VAR(foreign_key_checks=OFF) */ INTO t2 VALUES(2);
```

여기에 설명된 최적화 도구 힌트는 섹션 8.9.4, "인덱스 힌트"에 설명된 인덱스 힌트와 다릅니다. 옵티마이저와 인덱스 힌트는 따로 또는 함께 사용할 수 있습니다.

- 옵티마이저 힌트 개요
- 옵티마이저 힌트 구문
- 가입 주문 최적화 도구 힌트
- 테이블 수준 최적화 도구 힌트
- 인덱스 수준 최적화 도구 힌트

- 하위 쿼리 최적화 도구 힌트
- 명령문 실행 시간 최적화 도구 힌트
- 변수 설정 힌트 구문
- 리소스 그룹 힌트 구문

- 쿼리 블록 이름 지정을 위한 옵티마이저 힌트

옵티마이저 힌트 개요

옵티마이저 힌트는 다양한 범위 수준에서 적용됩니다:

- 전역: 힌트가 전체 문에 영향을 미칩니다.
- 쿼리 블록: 힌트는 문 내의 특정 쿼리 블록에 영향을 줍니다.
- 테이블 수준: 힌트는 쿼리 블록 내의 특정 테이블에 영향을 줍니다.
- 인덱스 수준: 힌트는 테이블 내의 특정 인덱스에 영향을 줍니다.

다음 표에는 사용 가능한 최적화 도구 힌트, 해당 힌트가 영향을 미치는 최적화 도구 전략, 적용되는 범위가 요약되어 있습니다. 자세한 내용은 나중에 설명합니다.

표 8.2 사용 가능한 옵티마이저 힌트

힌트 이름	설명	적용 범위
BKA, NO_BKA	일괄 키 액세스 조인 처리에 영향	쿼리 블록, 테이블
BNL, NO_BNL	해시 조인 최적화에 영향을 줍니다.	쿼리 블록, 테이블
파생_조건_푸시다운 NO_DERIVED_CONDITION_PUSH	구체화된 파생 테이블에 대해 파생된 DcOoWnNdition 푸시다운 최적화를 NU, se 또는 무시합니다.	쿼리 블록, 테이블
그룹_인덱스, NO_그룹_인덱스	GROUP BY 작업에서 인덱스 스캔에 지정된 인덱스를 사용하거나 무시합니다.	색인
해시 조인, NO_HASH_JOIN	해시 조인 최적화에 영향을 줍니다(MySQL 8.2에서는 영향 없음).	쿼리 블록, 테이블
index, no_index	JOIN_INDEX, GROUP_INDEX 및 ORDER_INDEX의 조합으로 작동합니다, 또는 NO_JOIN_INDEX, NO_GROUP_INDEX 및 NO_ORDER_INDEX의 조합으로 사용됩니다.	색인
INDEX_MERGE, NO_INDEX_MERGE	인덱스 병합 최적화에 영향을 줍니다.	표, 색인
조인_고정 주문	에 지정된 테이블 순서를 사용합니다	쿼리 블록

	. 트 조인 주문에 대한 FROM 절	
JOIN_INDEX, NO_JOIN_INDEX	모든 액세스 방법에 대해 지정된 인덱스를 사용하거나 무시합니다.	색인
JOIN_ORDER	조인 순서에 대한 힌트에 지정된 테이블 순서 사용	쿼리 블록
JOIN_PREFIX	조인 순서의 첫 번째 테이블에 힌트에 지정된 테이블 순서 사용	쿼리 블록
JOIN_SUFFIX	조인 순서의 마지막 테이블에 힌트에 지정된 테이블 순서 사용	쿼리 블록
최대 실행 시간	문 실행 시간 제한	글로벌
merge, no_merge	파생된 테이블/뷰를 외부 쿼리 블록에 병합하는 데 영향을 줍니다.	표

힌트 이름	설명	적용 범위
MRR, NO_MRR	다중 범위 읽기 최적화에 영향을 미칩니다.	표, 색인
NO_ICP	인덱스 조건 푸시다운 최적화에 영향을 줍니다.	표, 색인
NO_RANGE_OPTIMIZATION	범위 최적화에 영향을 미칩니다.	표, 색인
주문_인덱스, 아니_주문_인덱스	행 정렬에 지정된 인덱스를 사용하거나 무시합니다.	색인
QB_NAME	쿼리 블록에 이름 할당	쿼리 블록
RESOURCE_GROUP	문 실행 중 리소스 그룹 설정	글로벌
세미조인, 아니_세미조인	세미조인 및 안티조인 전략에 영향을 미칩니다.	쿼리 블록
SKIP_스캔, NO_SKIP_스캔	스캔 건너뛰기 최적화에 영향을 미칩니다.	표, 색인
SET_VAR	문 실행 중 변수 설정	글로벌
검색	구체화, IN-에 영향을 줍니다. to-EXISTS 하위 쿼리 전략	쿼리 블록

최적화를 비활성화하면 옵티마이저가 해당 최적화를 사용할 수 없습니다. 최적화를 사용하도록 설정하면 최적화 도구가 해당 전략이 문 실행에 적용되는 경우 자유롭게 사용할 수 있다는 의미이지, 최적화 도구가 반드시 해당 전략을 사용해야 한다는 의미는 아닙니다.

옵티마이저 힌트 구문

MySQL은 섹션 9.7, "주석"에 설명된 대로 SQL 문에서 주석을 지원합니다. 옵티마이저 힌트는 `/**+ ... */` 주석 안에 지정해야 합니다. 즉, 옵티마이저 힌트는 `/*역주: /*역주: */의` 변형을 사용합니다. `* ... */` C 스타일 주석 구문, `/*` 주석 열기 시퀀스 뒤에 `+` 문자가 있습니다. 예시:

```
/*+ BKA(t1) */
/*+ BNL(t1, t2) */
/*+ NO_RANGE_OPTIMIZATION(t4 PRIMARY) */
/*+ QB_NAME(qb2) */
```

문자 뒤에는 공백을 사용할 수 있습니다.

구문 분석기는 `SELECT`, `UPDATE`, `INSERT`, `REPLACE` 및 `DELETE` 문의 초기 키워드 뒤에 옵티마이저 힌트 주석을 인식합니다. 이러한 컨텍스트에서는 힌트가 허용됩니다:

- 쿼리 및 데이터 변경 문의 시작 부분에 있습니다:

```
SELECT /*+ ... */ ...
INSERT /*+ ... */ ...
대체 /*+ ... */ ...
업데이트 /*+ ... */ ...
DELETE /*+ ... */ ...
```

- 쿼리 블록의 시작 부분에 있습니다:

```
(선택 /*+ ... */ ... )
(선택 ... ) 유니온 (선택 /*+ ... */ ... )
(SELECT /*+ ... */ ... ) UNION (SELECT /*+ ... */ ... )
UPDATE ... WHERE x IN (SELECT /*+ ... */ ...)
삽입 ... 선택 /**+ ... */ ...
```

- 설명이 앞에 오는 힌트형 문에서. 예를 들어

```
설명 선택 /**+ ... */ ...
```

```
설명 업데이트 ... WHERE x IN (SELECT /*+ ... */ ...)
```

즉, 최적화 힌트가 실행 계획에 어떤 영향을 미치는지 확인하기 위해 EXPLAIN을 사용할 수 있다는 의미입니다. 힌트가 어떻게 사용되는지 확인하려면 EXPLAIN 바로 뒤에 SHOW WARNINGS를 사용하십시오. 다음 SHOW WARNINGS에 의해 표시되는 확장된 EXPLAIN 출력은 사용된 힌트를 나타냅니다. 무시된 힌트는 표시되지 않습니다.

힌트 댓글에는 여러 개의 힌트가 포함될 수 있지만 쿼리 블록에는 여러 개의 힌트 댓글을 포함할 수 없습니다. 이것은 유효합니다:

```
SELECT /*+ BNL(t1) BKA(t2) */ ...
```

그러나 이것은 유효하지 않습니다:

```
SELECT /*+ BNL(t1) */ /* BKA(t2) */ ...
```

힌트 댓글에 여러 개의 힌트가 포함되어 있는 경우 중복 및 충돌의 가능성이 존재합니다. 다음과 같은 일반적인 가이드라인이 적용됩니다. 특정 힌트 유형의 경우 힌트 설명에 명시된 대로 추가 규칙이 적용될 수 있습니다.

- 중복 힌트: MRR(idx1) MRR(idx1) */와 같은 힌트의 경우 MySQL은 첫 번째 힌트를 사용하며 중복 힌트에 대한 경고를 발행합니다.
- 충돌하는 힌트: MRR(idx1) NO_MRR(idx1) */와 같은 힌트의 경우 MySQL은 첫 번째 힌트를 사용하고 두 번째 충돌하는 힌트에 대해 경고를 발행합니다.

쿼리 블록 이름은 식별자이며 유효한 이름과 인용하는 방법에 대한 일반적인 규칙을 따릅니다(9.2절, "스키마 객체 이름" 참조).

힌트 이름, 쿼리 블록 이름 및 전략 이름은 대소문자를 구분하지 않습니다. 테이블 및 인덱스 이름에 대한 참조는 일반적인 식별자 대소문자 구분 규칙을 따릅니다(9.2.3절, '식별자 대소문자 구분' 참조).

가입 주문 최적화 도구 힌트

조인 순서 힌트는 옵티마이저가 테이블을 조인하는 순서에 영향을 줍니다.

JOIN_FIXED_ORDER 힌트 구문:

```
힌트 이름([@쿼리_블록_이름])
```

기타 조인 순서 힌트 구문:

```
hint_name([@query_block_name] tbl_name [, tbl_name] ...)
힌트_이름(tbl_name[@쿼리_블록_이름] [, tbl_name[@쿼리_블록_이름]] ...)
```

구문은 이러한 용어를 나타냅니다:

- 힌트_이름: 허용되는 힌트 이름입니다:
  - 조인_고정된_순서: 옵티마이저가 테이블이 테이블에 표시되는 순서를 사용하여 조인하도록 합니다.

`FROM` 절입니다. 이는 `SELECT STRAIGHT_JOIN`을 지정하는 것과 동일합니다.

- `JOIN_ORDER`: 옵티마이저에 지정된 테이블 순서를 사용하여 테이블을 조인하도록 지시합니다. 힌트는 명명된 테이블에 적용됩니다. 최적화 프로그램은 지정된 테이블 사이를 포함하여 조인 순서의 아무 곳에나 명명되지 않은 테이블을 배치할 수 있습니다.

- `JOIN_PREFIX`: 조인 실행 계획의 첫 번째 테이블에 대해 지정된 테이블 순서를 사용하여 테이블을 조인하도록 옵티마이저에 지시합니다. 이 힌트는 명명된 테이블에 적용됩니다. 최적화 프로그램은 명명된 테이블 뒤에 다른 모든 테이블을 배치합니다.

- `JOIN_SUFFIX`: 조인 실행 계획의 마지막 테이블에 대해 지정된 테이블 순서를 사용하여 테이블을 조인하도록 옵티마이저에 지시합니다. 이 힌트는 명명된 테이블에 적용됩니다. 최적화 프로그램은 다른 모든 테이블을 명명된 테이블 앞에 배치합니다.

- *tbl_name*: 문에 사용된 테이블의 이름입니다. 테이블 이름을 지정하는 힌트는 이름을 지정하는 모든 테이블에 적용됩니다. JOIN_FIXED_ORDER 힌트는 테이블 이름을 지정하지 않으며 해당 힌트가 발생하는 쿼리 블록의 FROM 절에 있는 모든 테이블에 적용됩니다.

  테이블에 별칭이 있는 경우 힌트는 테이블 이름이 아닌 별칭을 참조해야 합니다.

  힌트의 테이블 이름은 스키마 이름으로 한정할 수 없습니다.

- *쿼리 블록 이름*: 힌트가 적용되는 쿼리 블록입니다. 힌트에 선행 *@query_block_name이* 포함되지 않은 경우 힌트는 해당 힌트가 발생하는 쿼리 블록에 적용됩니다. For *tbl_name@query_block_name* 구문을 사용하는 경우, 힌트는 명명된 쿼리 블록의 명명된 테이블에 적용됩니다. 쿼리 블록에 이름을 지정하려면 쿼리 블록 이름 지정을 위한 최적화 도구 힌트를 참조하십시오.

예시:

```
SELECT
/*+ JOIN_PREFIX(t2, t5@subq2, t4@subq1)
    JOIN_ORDER(t4@subq1, t3)
    JOIN_SUFFIX(t1) */
COUNT(*) FROM t1 JOIN t2 JOIN t3
           WHERE t1.f1 IN (SELECT /*+ QB_NAME(subq1) */ f1 FROM t4)
             AND t2.f1 IN (SELECT /*+ QB_NAME(subq2) */ f1 FROM t5);
```

힌트는 외부 쿼리 블록에 병합되는 세미조인 테이블의 동작을 제어합니다. 하위 쿼리 subq1 및 subq2가 세미조인으로 변환되면 테이블 t4@subq1 및 t5@subq2가 외부 쿼리 블록에 병합됩니다. 이 경우 외부 쿼리 블록에 지정된 힌트가 t4@subq1, t5@subq2 테이블의 동작을 제어합니다.

옵티마이저는 이러한 원칙에 따라 조인 순서 힌트를 해결합니다:

- 여러 힌트 인스턴스

  각 유형에 대해 하나의 JOIN_PREFIX 및 JOIN_SUFFIX 힌트만 적용됩니다. 이후 동일한 유형의 힌트는 경고와 함께 무시됩니다. JOIN_ORDER는 여러 번 지정할 수 있습니다.

  예시:

  ```
  /*+ JOIN_PREFIX(t1) JOIN_PREFIX(t2) */
  ```

  두 번째 JOIN_PREFIX 힌트는 경고와 함께 무시됩니다.

  ```
  /*+ JOIN_PREFIX(t1) JOIN_SUFFIX(t2) */
  ```

  두 가지 힌트 모두 적용 가능합니다. 경고가 발생하지 않습니다.

  ```
  /*+ JOIN_ORDER(t1, t2) JOIN_ORDER(t2, t3) */
  ```

  두 가지 힌트 모두 적용 가능합니다. 경고가 발생하지 않습니다.

- 상충되는 힌트

JOIN_ORDER와 JOIN_PREFIX에 동시에 적용할 수 없는 테이블 순서가 있는 경우와 같이 힌트가 충돌하는 경우도 있습니다:

```
SELECT /*+ JOIN_ORDER(t1, t2) JOIN_PREFIX(t2, t1) */ ... FROM t1, t2;
```

이 경우 처음 지정된 힌트가 적용되고 이후 상충되는 힌트는 경고 없이 무시됩니다. 적용이 불가능한 유효한 힌트는 경고 없이 자동으로 무시됩니다.

- 무시된 힌트

  힌트에 지정된 테이블에 순환 종속성이 있는 경우 힌트는 무시됩니다. 예제:

```
/*+ JOIN_ORDER(t1, t2) JOIN_PREFIX(t2, t1) */
```

`JOIN_ORDER` 힌트는 테이블 `t2`를 `t1`에 종속되도록 설정합니다. 테이블 `t1`은 `t2`에 종속될 수 없으므로 `JOIN_PREFIX` 힌트는 무시됩니다. 무시된 힌트는 확장 `EXPLAIN` 출력에 표시되지 않습니다.

- `const` 테이블과의 상호작용

  MySQL 옵티마이저는 조인 순서에서 `const` 테이블을 먼저 배치하며, `const` 테이블의 위치는 힌트의 영향을 받지 않습니다. 조인 순서 힌트에서 `const` 테이블에 대한 참조는 무시되지만 힌트는 여전히 적용됩니다. 예를 들어 다음과 같습니다:

  ```
  JOIN_ORDER(t1, const_tbl, t2)
  JOIN_ORDER(t1, t2)
  ```

  확장된 설명 출력에 표시되는 허용되는 힌트에는 지정된 대로 `const` 테이블이 포함됩니다.

- 조인 작업 유형과의 상호 작용

  MySQL은 여러 유형의 조인을 지원합니다: 왼쪽, 오른쪽, 내부, 교차, `straight_join`. 지정된 조인 유형과 충돌하는 힌트는 경고 없이 무시됩니다.

  예시:

  ```
  SELECT /*+ JOIN_PREFIX(t1, t2) */FROM t2 LEFT JOIN t1;
  ```

  여기에서 힌트에서 요청한 조인 순서와
  왼쪽 조인. 힌트는 경고 없이 무시됩니다.

테이블 수준 최적화 도구 힌트

테이블 수준 힌트가 영향을 줍니다:

- 블록 중첩 루프(BNL) 및 일괄 키 액세스(BKA) 조인 처리 알고리즘 사용(섹션 8.2.1.12, "블록 중첩 루프 및 일괄 키 액세스 조인" 참조).
- 파생 테이블, 뷰 참조 또는 공통 테이블 식을 외부 쿼리 블록에 병합할지, 아니면 내부 임시 테이블을 사용하여 구체화할지 여부를 결정합니다.
- 파생된 테이블 조건 푸시다운 최적화 사용. 섹션 8.2.2.5, "파생된 조건 푸시다운 최적화"를 참조하십시오.

이러한 힌트 유형은 특정 테이블 또는 쿼리 블록의 모든 테이블에 적용됩니다.

```
힌트_이름([@쿼리_블록_이름] [tbl_이름 [, tbl_이름] ...])
힌트_이름([tbl_name@쿼리_블록_이름 [, tbl_name@쿼리_블록_이름] ...])
```

구문은 이러한 용어를 나타냅니다:

- 힌트_이름: 허용되는 힌트 이름입니다: 트

  - BKA, NO_BKA: 지정된 테이블에 대한 일괄 키 액세스를 사용하거나 사용하지 않도록 설정합니다.

  - BNL, NO_BNL: 해시 조인 최적화를 활성화 및 비활성화합니다.

  - DERIVED_CONDITION_PUSHDOWN, NO_DERIVED_CONDITION_PUSHDOWN: 사용 또는 사용 안 함 지정된 테이블에 대해 파생된 테이블 조건 푸시다운을 사용합니다. 자세한 내용은 섹션 8.2.2.5, "파생된 조건 푸시다운 최적화"를 참조하십시오.

  - HASH_JOIN, NO_HASH_JOIN: 이 힌트는 MySQL 8.2에서는 효과가 없으며, BNL 또는 NO_BNL을 사용합니다.
    대신

- MERGE, NO_MERGE: 지정된 테이블, 뷰 참조 또는 공통 테이블 표현식에 대해 병합을 활성화하거나 병합을 비활성화하고 대신 구체화를 사용합니다.

참고

블록 중첩 루프 또는 일괄 키 액세스 힌트를 사용하여 외부 조인의 내부 테이블에 대해 조인 버퍼링을 사용하도록 설정하려면 외부 조인의 모든 내부 테이블에 대해 조인 버퍼링을 사용하도록 설정해야 합니다.

- *tbl_name*: 문에 사용된 테이블의 이름입니다. 힌트는 이름이 지정된 모든 테이블에 적용됩니다. 힌트에 테이블 이름이 지정되지 않은 경우 힌트가 발생하는 쿼리 블록의 모든 테이블에 적용됩니다.

  테이블에 별칭이 있는 경우 힌트는 테이블 이름이 아닌 별칭을 참조해야 합니다.

  힌트의 테이블 이름은 스키마 이름으로 한정할 수 없습니다.

- *쿼리 블록 이름*: 힌트가 적용되는 쿼리 블록입니다. 힌트에 쿼리 블록 *이름* 앞에 오는 경우 힌트는 해당 힌트가 발생하는 쿼리 블록에 적용됩니다. *tbl_name@query_block_name* 구문의 경우, 힌트는 명명된 쿼리 블록의 명명된 테이블에 적용됩니다. 쿼리 블록에 이름을 지정하려면 쿼리 블록 이름 지정을 위한 최적화 도구 힌트를 참조하십시오.

예시:

```
SELECT /*+ NO_BKA(t1, t2) */ t1.* FROM t1 INNER JOIN t2 INNER JOIN t3;
SELECT /*+ NO_BNL() BKA(t1) */ t1.* FROM t1 INNER JOIN t2 INNER JOIN t3;
SELECT /*+ NO_MERGE(dt) */ * FROM (SELECT * FROM t1) AS dt;
```

테이블 수준 힌트는 발신자 테이블이 아닌 이전 테이블에서 레코드를 수신하는 테이블에 적용됩니다. 이 문을 고려하십시오:

```
SELECT /*+ BNL(t2) */ FROM t1, t2;
```

옵티마이저가 t1을 먼저 처리하기로 선택한 경우 옵티마이저는 t2에서 읽기를 시작하기 전에 t1의 행을 버퍼링하여 t2에 블록 중첩 루프 조인을 적용합니다. 대신 옵티마이저가 t2를 먼저 처리하도록 선택하면 t2가 발신자 테이블이므로 힌트가 적용되지 않습니다.

MERGE 및 NO_MERGE 힌트의 경우 이러한 우선 순위 규칙이 적용됩니다:

- 힌트는 기술적 제약이 아닌 모든 최적화 도구 휴리스틱보다 우선합니다. (힌트를 제안으로 제공해도 효과가 없는 경우 옵티마이저가 이를 무시할 이유가 있습니다.)
- 힌트는 optimizer_switch 시스템 변수의 derived_merge 플래그보다 우선합니다.
- 뷰 참조의 경우 뷰 정의의 ALGORITHM={MERGE|TEMPTABLE} 절이 뷰를 참조하는 쿼리에 지정된 힌트보다 우선합니다.

인덱스 수준 최적화 도구 힌트

인덱스 수준 힌트는 옵티마이저가 특정 테이블 또는 인덱스에 사용하는 인덱스 처리 전략에 영향을 줍니다. 이러한 힌트 유형은 ICP(인덱스 조건 푸시다운), MRR(다중 범위 읽기), 인덱스 병합 및 범위 최적화의 사용에 영향을 줍니다(8.2.1절, "SELECT 문 최적화" 참조).

인덱스 수준 힌트 구문:

```
힌트_이름([@쿼리_블록_이름] tbl_이름 [인덱스_이름 [, 인덱스_이름] ...])
힌트_이름(tbl_이름@쿼리_블록_이름 [인덱스_이름 [, 인덱스_이름] ...])
```

구문은 이러한 용어를 나타냅니다:

- 힌트_이름: 허용되는 힌트 이름입니다:

- `GROUP_INDEX`, `NO_GROUP_INDEX`: `GROUP BY` 연산에 대한 인덱스 스캔에 지정된 인덱스를 사용하거나 사용하지 않도록 설정합니다. 인덱스 힌트 `FORCE INDEX FOR GROUP BY`, `IGNORE INDEX FOR GROUP BY`와 동일합니다.

- `INDEX`, `NO_INDEX`: 서버가 모든 범위에 대해 지정된 인덱스를 사용하도록 강제하는 `JOIN_INDEX`, `GROUP_INDEX`, `ORDER_INDEX`의 조합으로 작동하거나, 서버가 모든 범위에 대해 지정된 인덱스를 무시하도록 하는 `NO_JOIN_INDEX`, `NO_GROUP_INDEX`, `NO_ORDER_INDEX`의 조합으로 작동합니다. `FORCE INDEX`, `IGNORE INDEX`와 동일합니다.

- `INDEX_MERGE`, `NO_INDEX_MERGE`: 지정된 테이블 또는 인덱스에 대한 인덱스 병합 액세스 방법을 사용하거나 사용하지 않도록 설정합니다. 이 액세스 방법에 대한 자세한 내용은 섹션 8.2.1.3, "인덱스 병합 최적화"를 참조하십시오. 이 힌트는 세 가지 인덱스 병합 알고리즘 모두에 적용됩니다.

  `인덱스 병합` 힌트는 옵티마이저가 지정된 인덱스 집합을 사용하여 지정된 테이블에 인덱스 병합을 사용하도록 강제합니다. 인덱스가 지정되지 않은 경우 옵티마이저는 가능한 모든 인덱스 조합을 고려하고 가장 비용이 적게 드는 인덱스 조합을 선택합니다. 인덱스 조합이 주어진 문에 적용될 수 없는 경우 힌트는 무시될 수 있습니다.

  `NO_INDEX_MERGE` 힌트는 지정된 인덱스가 포함된 인덱스 병합 조합을 비활성화합니다. 힌트에 인덱스가 지정되지 않은 경우 테이블에 대해 인덱스 병합이 허용되지 않습니다.

- `JOIN_INDEX`, `NO_JOIN_INDEX`: MySQL이 `ref`, `range`, `index_merge` 등과 같은 액세스 메서드에 대해 지정된 인덱스를 사용하거나 무시하도록 강제합니다. `FORCE INDEX FOR JOIN`, `IGNORE INDEX FOR JOIN`과 동일합니다.

- `MRR`, `NO_MRR`: 지정된 테이블 또는 인덱스에 대해 MRR을 사용 또는 사용하지 않도록 설정합니다. MRR 힌트는 `InnoDB` 및 `MyISAM` 테이블에만 적용됩니다. 이 액세스 방법에 대한 자세한 내용은 섹션 8.2.1.11, "다중 범위 읽기 최적화"를 참조하십시오.

- `NO_ICP`: 지정된 테이블 또는 인덱스에 대해 ICP를 비활성화합니다. 기본적으로 ICP는 후보 최적화 전략이므로 활성화에 대한 힌트가 없습니다. 이 액세스 방법에 대한 자세한 내용은 다음을 참조하십시오. 섹션 8.2.1.6, '인덱스 조건 푸시다운 최적화'.

- `NO_RANGE_OPTIMIZATION`: 지정된 테이블 또는 인덱스에 대한 인덱스 범위 액세스를 비활성화합니다. 이 힌트는 테이블 또는 인덱스에 대한 인덱스 병합 및 느슨한 인덱스 스캔도 비활성화합니다. 기본적으로 범위 액세스는 후보 최적화 전략이므로 이를 활성화하기 위한 힌트는 없습니다.

  이 힌트는 범위의 수가 많고 범위 최적화에 많은 리소스가 필요할 때 유용할 수 있습니다.

- `ORDER_INDEX`, `NO_ORDER_INDEX`: MySQL이 행을 정렬할 때 지정된 인덱스를 사용하거나 무시하도록 합니다. `주문 기준 인덱스 강제 적용`, `주문 기준 인덱스 무시와` 동일합니다.

- `SKIP_SCAN`, `NO_SKIP_SCAN`: 지정된 테이블 또는 인덱스에 대한 스캔 건너뛰기 액세스 방법을 사용하거나 사용하지 않도록 설정합니다. 이 접근 방법에 대한 자세한 내용은 스캔 범위 접근 방법 건너뛰기를 참

조하십시오.

`SKIP_SCAN` 힌트는 옵티마이저가 지정된 인덱스 집합을 사용하여 지정된 테이블에 대해 건너뛰기 스캔을 사용하도록 강제합니다. 인덱스가 지정되지 않은 경우 옵티마이저는 가능한 모든 인덱스를 고려하고 가장 비용이 적게 드는 인덱스를 선택합니다. 인덱스가 주어진 문에 적용될 수 없는 경우 힌트는 무시될 수 있습니다.

`NO_SKIP_SCAN` 힌트는 지정된 인덱스에 대해 건너뛰기 스캔을 비활성화합니다. 힌트에 인덱스가 지정되지 않은 경우 테이블에 대해 건너뛰기 스캔이 허용되지 않습니다.

- *`tbl_name`*: 힌트가 적용되는 테이블입니다.
- *`index_name`*: 명명된 테이블에 있는 인덱스의 이름입니다. 힌트는 이름이 지정된 모든 인덱스에 적용됩니다. 힌트에서 인덱스 이름을 지정하지 않으면 테이블의 모든 인덱스에 적용됩니다.

기본 키를 참조하려면 PRIMARY라는 이름을 사용합니다. 테이블의 인덱스 이름을 보려면 SHOW INDEX를 사용합니다.

- *쿼리 블록 이름*: 힌트가 적용되는 쿼리 블록입니다. 힌트에 선행 *@query_block_name*이 포함되지 않은 경우 힌트는 해당 힌트가 발생하는 쿼리 블록에 적용됩니다. For *tbl_name@query_block_name* 구문을 사용하는 경우, 힌트는 명명된 쿼리 블록의 명명된 테이블에 적용됩니다. 쿼리 블록에 이름을 지정하려면 쿼리 블록 이름 지정을 위한 최적화 도구 힌트를 참조하십시오.

예시:

```
SELECT /*+ INDEX_MERGE(t1 f3, PRIMARY) */ f2 FROM t1
  WHERE f1 = 'o' AND f2 = f3 AND f3 <= 4;
SELECT /*+ MRR(t1) */ * FROM t1 WHERE f2 <= 3 AND 3 <= f3;
SELECT /*+ NO_RANGE_OPTIMIZATION(t3 PRIMARY, f2_idx) */ f1
  FROM t3 WHERE f1 > 30 AND f1 < 33;
INSERT INTO t3(f1, f2, f3)
  (SELECT /*+ NO_ICP(t2) */ t2.f1, t2.f2, t2.f3 FROM t1,t2
   WHERE t1.f1=t2.f1 AND t2.f2 BETWEEN t1.f1
   AND t1.f2 AND t2.f2 + 1 >= t1.f1 + 1);
SELECT /*+ SKIP_SCAN(t1 PRIMARY) */ f1, f2
  FROM t1 WHERE f2 > 40;
```

다음 예제에서는 인덱스 병합 힌트를 사용하지만 다른 인덱스 수준 힌트도 optimizer_switch 시스템 변수 또는 인덱스 힌트와 관련하여 힌트 무시 및 옵티마이저 힌트의 우선순위에 관한 동일한 원칙을 따릅니다.

테이블 t1에 열 a, b, c, d가 있고 인덱스 이름이 i_a, i_b, i_c인 인덱스가 a, b, c를 각각 입력합니다:

```
SELECT /*+ INDEX_MERGE(t1 i_a, i_b, i_c)*/ * FROM t1
  WHERE a = 1 AND b = 2 AND c = 3 AND d = 4;
```

이 경우 인덱스 병합은 (i_a, i_b, i_c)에 사용됩니다.

```
SELECT /*+ INDEX_MERGE(t1 i_a, i_b, i_c)*/ * FROM t1
  WHERE b = 1 AND c = 2 AND d = 3;
```

이 경우 인덱스 병합은 (i_b, i_c)에 사용됩니다.

```
/*+ INDEX_MERGE(t1 i_a, i_b) NO_INDEX_MERGE(t1 i_b) */
```

동일한 테이블에 대한 선행 힌트가 있기 때문에 NO_INDEX_MERGE는 무시됩니다.

```
/*+ NO_INDEX_MERGE(t1 i_a, i_b) INDEX_MERGE(t1 i_b) */
```

동일한 테이블에 대한 선행 힌트가 있기 때문에 INDEX_MERGE는 무시됩니다.

인덱스 병합 및 노 인덱스 병합 옵티마이저 힌트의 경우 이러한 우선 순위 규칙이 적용됩니다:

- 옵티마이저 힌트가 지정되어 있고 적용 가능한 경우, 옵티마이저 힌트는 optimizer_switch 시스템 변수의 인덱스 병합 관련 플래그보다 우선합니다.

  ```
  SET optimizer_switch='index_merge_intersection=off';
  SELECT /*+ INDEX_MERGE(t1 i_b, i_c) */ * FROM t1
  WHERE b = 1 AND c = 2 AND d = 3;
  ```

  힌트는 optimizer_switch보다 우선합니다. 이 경우 인덱스 병합은 (i_b, i_c)에 사용됩니다.

  ```
  SET optimizer_switch='index_merge_intersection=on';
  SELECT /*+ INDEX_MERGE(t1 i_b) */ * FROM t1
  WHERE b = 1 AND c = 2 AND d = 3;
  ```

힌트는 하나의 인덱스만 지정하므로 적용할 순 없으며 optimizer_switch 플래그(on)가 적용됩니다.

인덱스 병합은 옵티마이저가 비용 효율적이라고 평가하는 경우에 사용됩니다.

```
SET optimizer_switch='index_merge_intersection=off';
```

```
SELECT /*+ INDEX_MERGE(t1 i_b) */ * FROM t1
WHERE b = 1 AND c = 2 AND d = 3;
```

힌트는 하나의 인덱스만 지정하므로 적용할 수 없으며 `optimizer_switch` 플래그(`off`)가 적용됩니다. 인덱스 병합은 사용되지 않습니다.

- 인덱스 수준 옵티마이저 힌트 `GROUP_INDEX`, `INDEX`, `JOIN_INDEX` 및 `ORDER_INDEX`는 모두 동등한 `FORCE INDEX` 힌트보다우선하며, 즉 `FORCE INDEX` 힌트가 무시됩니다. 마찬가지로 `NO_GROUP_INDEX`, `NO_INDEX`, `NO_JOIN_INDEX` 및 `NO_ORDER_INDEX` 힌트도 모두 `IGNORE` INDEX와 동등한 힌트보다 우선하며, 이 역시 무시됩니다.

  인덱스 수준 옵티마이저 힌트인 `GROUP_INDEX`, `NO_GROUP_INDEX`, `INDEX`,`NO_INDEX`, `JOIN_INDEX`, `NO_JOIN_INDEX`, `ORDER_INDEX` 및 `NO_ORDER_INDEX` 힌트는 다른 인덱스 수준 옵티마이저 힌트를 포함한 모든 다른 옵티마이저 힌트보다 우선합니다. 다른 모든 옵티마이저 힌트는 이 힌트가 허용하는 인덱스에만 적용됩니다.

  `GROUP_INDEX`, `INDEX`, `JOIN_INDEX` 및 `ORDER_INDEX` 힌트는 모두 `FORCE INDEX`와 동일하며`USE INDEX`와 동일하지 않습니다. 이러한 힌트 중 하나 이상을 사용하면 테이블에서 행을 찾기 위해 명명된 인덱스 중 하나를 사용할 방법이 없는 경우에만 테이블 스캔이 사용되기 때문입니다. MySQL이 지정된 `USE INDEX` 인스턴스와 동일한 인덱스 또는 인덱스 집합을 사용하도록 하려면 `NO_INDEX`, `NO_JOIN_INDEX`, `NO_GROUP_INDEX`, `NO_ORDER_INDEX` 또는 이들의 조합을 사용할 수 있습니다.

  `SELECT a,c FROM t1 USE INDEX FOR ORDER BY (i_a) ORDER BY a` 쿼리에서 `USE INDEX`의 효과를 복제하려면 `NO_ORDER_INDEX` 옵티마이저 힌트를 사용하여 다음과 같이 원하는 인덱스를 제외한 테이블의 모든 인덱스를 포함하도록 할 수 있습니다:

```
SELECT /*+ NO_ORDER_INDEX(t1 i_b,i_c) */ a,c
    FROM t1
    주문 기준 a;
```

  여기에 표시된 것처럼 테이블 전체에 대한 `NO_ORDER_INDEX`와 주문 기준에 대한 `USE INDEX`를 결합하려고 시도하면 `NO_ORDER_BY`로 인해 `USE INDEX`가 무시되므로 이 작업을 수행할 수 없습니다:

```
mysql> EXPLAIN SELECT /*+ NO_ORDER_INDEX(t1) */ a,c FROM t1
    ->     USE INDEX FOR ORDER BY (i_a) ORDER BY a\G
*************************** 1. 행 ***************************
           ID: 1
  select_type: SIMPLE
        테이블: t1
   파티션: NULL
         유형: 모든
가능한_키: NULL
          키: NULL
      key_len: NULL
          참조:
         NULL 행:
         256
     필터링됨: 100.00
        추가: 파일 정렬 사용
```

  - `USE 인덱스`, `FORCE 인덱스`, `IGNORE` 인덱스 힌트에는 다음과 같은 우선순위가 있습니다.

인덱스 병합 및 노 인덱스 병합 옵티마이저 힌트.

```
/*+ INDEX_MERGE(t1 i_a, i_b, i_c) */ ... IGNORE INDEX i_a
```

인덱스 무시는 인덱스 병합보다 우선하므로 인덱스 i_a는 인덱스 병합의 가능한 범위에서 제외됩니다.

```
/*+ NO_INDEX_MERGE(t1 i_a, i_b) */ ... FORCE INDEX i_a, i_b
```

인덱스 병합은 강제 인덱스로 인해 `i_a, i_b`에 대해 허용되지 않지만 옵티마이저는 범위 또는 참조 액세스에 대해 `i_a` 또는 `i_b` 중 하나를 사용하도록 강제합니다. 충돌은 없으며 두 힌트 모두 적용 가능합니다.

- 인덱스 무시 힌트가 여러 인덱스를 지정하는 경우 해당 인덱스는 인덱스 병합에 사용할 수 없습니다.

- `FORCE INDEX` 및 `USE INDEX` 힌트는 명명된 인덱스만 인덱스 병합에 사용할 수 있도록 합니다.

```
SELECT /*+ INDEX_MERGE(t1 i_a, i_b, i_c) */ a FROM t1
FORCE INDEX (i_a, i_b) WHERE c = 'h' AND a = 2 AND b = 'b';
```

  인덱스 병합 교차점 액세스 알고리즘은 `(i_a, i_b)`에 사용됩니다. `FORCE INDEX`가 `USE INDEX`로 변경된 경우에도 마찬가지입니다.

하위 쿼리 최적화 도구 힌트

서브쿼리 힌트는 세미조인 변환을 사용할지 여부와 허용할 세미조인 전략에 영향을 미치며, 세미조인을 사용하지 않는 경우 서브쿼리 구체화 또는 `IN-to-EXISTS` 변환을 사용할지 여부에 영향을 줍니다. 이러한 최적화에 대한 자세한 내용은 섹션 8.2.2, "하위 쿼리, 파생 테이블, 뷰 참조 및 일반 테이블 식 최적화"를 참조하십시오.

세미조인 전략에 영향을 미치는 힌트 구문입니다:

```
힌트_이름([@쿼리_블록_이름] [전략 [, 전략] ...])
```

구문은 이러한 용어를 나타냅니다:

- 힌트_이름: 허용되는 힌트 이름입니다:
  - 세미조인, `NO_SEMIJOIN`: 명명된 세미조인 전략을 활성화 또는 비활성화합니다.
- *전략*: 전략: 활성화 또는 비활성화할 세미조인 전략입니다. 이러한 전략 이름은 허용됩니다: 덥스위드아웃, 퍼스트매치, 루즈스캔, 구체화.

  세미조인 힌트의 경우, 명명된 전략이 없는 경우 `optimizer_switch` 시스템 변수에 따라 활성화된 전략에 따라 가능한 경우 세미조인이 사용됩니다. 전략의 이름이 지정되어 있지만 문에 적용할 수 없는 경우 `DUPSWEEDOUT`이 사용됩니다.

  `NO_SEMIJOIN` 힌트의 경우, 전략 이름이 지정되지 않으면 세미조인이 사용되지 않습니다. 문에 적용 가능한 모든 전략을 배제하는 전략이 명명된 경우 `DUPSWEEDOUT`이 사용됩니다.

한 하위 쿼리가 다른 하위 쿼리 내에 중첩되어 있고 두 쿼리 모두 외부 쿼리의 세미조인으로 병합되는 경우 가장 안쪽 쿼리에 대한 세미조인 전략의 지정은 무시됩니다. 세미조인 및 `NO_SEMIJOIN` 힌트를 사용하여 이러한 중첩된 하위 쿼리에 대한 세미조인 변환을 활성화 또는 비활성화할 수 있습니다.

중복 제거가 비활성화되어 있으면 최적화 도구가 최적이 아닌 쿼리 계획을 생성하는 경우가 있습니다. 이는 탐욕스러운 검색 중 휴리스틱 프루닝으로 인해 발생하며, `optimizer_prune_level=0`을 설정하면 이를 방지할 수 있습니다.

예시:

```
SELECT /*+ NO_SEMIJOIN(@subq1 FIRSTMATCH, LOOSESCAN) */ * FROM t2
  WHERE t2.a IN (SELECT /*+ QB_NAME(subq1) */ a FROM t3);
SELECT /*+ SEMIJOIN(@subq1 MATERIALIZATION, DUPSWEEDOUT) */ * FROM t2
  WHERE t2.a IN (SELECT /*+ QB_NAME(subq1) */ a FROM t3);
```

하위 쿼리 구체화 또는 IN에서 EXISTS로의 변환을 사용할지 여부에 영향을 주는 힌트 구문입니다:

```
SUBQUERY([@query_block_name] 전략)
```

힌트 이름은 항상 SUBQUERY입니다.

SUBQUERY 힌트의 경우 다음과 같은 *전략* 값이 허용됩니다: INTOEXISTS, MATERIALIZATION. 예제:

```
SELECT id, a IN (SELECT /*+ SUBQUERY(MATERIALIZATION) */ a FROM t1) FROM t2;
SELECT * FROM t2 WHERE t2.a IN (SELECT /*+ SUBQUERY(INTOEXISTS) */ a FROM t1);
```

세미조인 및 SUBQUERY 힌트의 경우 앞에 오는 *@query_block_name은* 힌트가 적용되는 쿼리 블록을 지정합니다. 힌트에 선행 *@query_block_name이* 포함되지 않은 경우 힌트는 해당 힌트가 발생하는 쿼리 블록에 적용됩니다. 쿼리 블록에 이름을 지정하려면 쿼리 블록 이름 지정을 위한 옵티마이저 힌트를 참조하세요.

힌트 댓글에 여러 개의 하위 쿼리 힌트가 포함되어 있는 경우 첫 번째 힌트가 사용됩니다. 해당 유형의 다른 후속 힌트가 있는 경우 경고를 생성합니다. 다른 유형의 다음 힌트는 자동으로 무시됩니다.

명령문 실행 시간 최적화 도구 힌트

MAX_EXECUTION_TIME 힌트는 SELECT 문에만 허용됩니다. 이 힌트는 서버가 문을 종료하기 전에 문이 실행될 수 있는 시간(밀리초 단위의 시간 초과 값)에 제한 *N을* 설정합니다:

```
최대 실행 시간(N)
```

타임아웃이 1초(1000밀리초)인 예제입니다:

```
SELECT /*+ MAX_EXECUTION_TIME(1000) */ * FROM t1 INNER JOIN t2 WHERE ...
```

MAX_EXECUTION_TIME(*N*) 힌트는 문 실행 시간 제한을 N밀리초로 설정합니다. 이 옵션이 없거나 N이 0이면 max_execution_time 시스템 변수에 의해 설정된 문 시간 제한이 적용됩니다.

MAX_EXECUTION_TIME 힌트는 다음과 같이 적용할 수 있습니다:

- 유니온 또는 하위 쿼리가 있는 문과 같이 여러 SELECT 키워드가 있는 문입니다, MAX_EXECUTION_TIME은 전체 문에 적용되며 첫 번째 SELECT 뒤에 나타나야 합니다.
- 읽기 전용 SELECT 문에 적용됩니다. 읽기 전용이 아닌 문은 부작용으로 데이터를 수정하는 저장 함수를 호출하는 문입니다.
- 저장된 프로그램의 SELECT 문에는 적용되지 않으며 무시됩니다.

변수 설정 힌트 구문

SET_VAR 힌트는 시스템 변수의 세션 값을 일시적으로(단일 문이 실행되는 동안) 설정합니다. 예시:

```
SELECT /*+ SET_VAR(sort_buffer_size = 16M) */ name FROM people ORDER BY name;
INSERT /*+ SET_VAR(foreign_key_checks=OFF) */ INTO t2 VALUES(2);
SELECT /*+ SET VAR(optimizer switch = 'mrr cost 기반=off') */ 1;
```

SET_VAR 힌트 구문:

```
SET_VAR(var_name = value)
```

var_name은 세션 값이 있는 시스템 변수의 이름을 지정합니다(나중에 설명하겠지만 모든 변수에 이름을 지정할 수 있는 것은 아닙니다). *value는* 변수에 할당할 값으로, 값은 스칼라여야 합니다.

SET_VAR은 다음 문에서 볼 수 있듯이 임시 변수 변경을 수행합니다:

```
mysql> SELECT @@unique_checks;
+------------------+
| @@유니크체크 | @@유니크체크
+------------------+
|1 |
+------------------+
mysql> SELECT /*+ SET_VAR(unique_checks=OFF) */ @@unique_checks;
+------------------+
| @@유니크체크 | @@유니크체크
+------------------+
|0 |
+------------------+
mysql> SELECT @@unique_checks;
+------------------+
| @@유니크체크 | @@유니크체크
```

```
+-----------------+
|1 |
+-----------------+
```

SET_VAR을 사용하면 변수 값을 저장하고 복원할 필요가 없습니다. 따라서 여러 문을 하나의 문으로 대체할 수 있습니다. 다음 문 시퀀스를 살펴보겠습니다:

```
SET @saved_val = @@SESSION.var_name;
SET @@SESSION.var_name = value;
SELECT ...
SET @@SESSION.var_name = @saved_val;
```

시퀀스는 이 단일 문으로 대체할 수 있습니다:

```
SELECT /*+ SET_VAR(var_name = value) ...
```

독립형 `SET` 문을 사용하면 세션 변수의 이름을 지정하는 데 이러한 구문 중 하나를 사용할 수 있습니다:

```
SET SESSION var_name = 값; SET
@@SESSION.var_name = 값; SET
@@.var_name = 값;
```

`SET_VAR` 힌트는 세션 변수에만 적용되므로 세션 범위는 암시적이며 `SESSION`, `@@SESSION.`, `@@`는 필요하지도 않고 허용되지도 않습니다. 명시적인 세션 표시기 구문을 포함하면 `SET_VAR` 힌트가 경고와 함께 무시됩니다.

모든 세션 변수를 `SET_VAR`과 함께 사용할 수 있는 것은 아닙니다. 개별 시스템 변수 설명에 각 변수의 힌트 가능 여부가 나와 있습니다(5.1.8절, "서버 시스템 변수"를 참조하세요). 런타임에 시스템 변수를 `SET_VAR`과 함께 사용하려고 시도하여 확인할 수도 있습니다. 변수가 힌트 가능하지 않으면 경고가 발생합니다:

```
mysql> SELECT /*+ SET_VAR(collation_server = 'utf8mb4') */ 1;
+---+
| 1 |
+---+
| 1 |
+---+
세트 1행, 경고 1회(0.00초)

mysql> SHOW WARNINGS\G
*************************** 1. 행 ***************************
  벨: 경고
   코드: 4537
메시지: SET_VAR 힌트를 사용하여 'collation_server' 변수를 설정할 수 없습니다.
```

`SET_VAR` 구문을 사용하면 하나의 변수만 설정할 수 있지만 여러 개의 변수를 설정하기 위해 여러 개의 힌트를 제공할 수 있습니다:

```
SELECT /*+ SET_VAR(optimizer_switch = 'mrr_cost_기반=off')
           SET_VAR(max_heap_table_size = 1G) */ 1;
```

동일한 변수 이름을 가진 힌트가 동일한 문에 여러 개 나타나면 첫 번째 힌트가 적용되고 나머지는 경고와 함께 무시됩니다:

```
SELECT /*+ SET_VAR(max_heap_table_size = 1G)
           SET_VAR(max_heap_table_size = 3G) */ 1;
```

이 경우 두 번째 힌트는 상충된다는 경고와 함께 무시됩니다.

지정된 이름을 가진 시스템 변수가 없거나 변수 값이 올바르지 않은 경우 `SET_VAR` 힌트는 경고와 함께 무시됩니다:

```
SELECT /*+ SET_VAR(max_size = 1G) */ 1;
SELECT /*+ SET_VAR(optimizer_switch = 'mrr_cost_based=yes') */ 1;
```

첫 번째 문에는 `max_size` 변수가 없습니다. 두 번째 문에서 `mrr_cost_based`는 `on` 또는 `off` 값을 사용하므로 `yes`로 설정하려고 시도하면 올바르지 않습니다. 각각의 경우 힌트는 경고와 함께 무시됩니다.

SET_VAR 힌트는 문 수준에서만 허용됩니다. 하위 쿼리에서 사용되는 경우 힌트는 경고와 함께 무시됩니다.

복제본은 보안 문제가 발생할 가능성을 피하기 위해 복제된 문에서 SET_VAR 힌트를 무시합니다.

리소스 그룹 힌트 구문

RESOURCE_GROUP 옵티마이저 힌트는 리소스 그룹 관리에 사용됩니다(섹션 5.1.16, "리소스 그룹" 참조). 이 힌트는 문을 실행하는 스레드를 명명된 리소스 그룹에 일시적으로(문이 실행되는 기간 동안) 할당합니다. 이 힌트를 사용하려면 RESOURCE_GROUP_ADMIN 또는 RESOURCE_GROUP_USER 권한이 필요합니다.

예시:

```
SELECT /*+ RESOURCE_GROUP(USR_default) */ name FROM people ORDER BY name;
INSERT /*+ RESOURCE_GROUP(Batch) */ INTO t2 VALUES(2);
```

RESOURCE_GROUP 힌트 구문:

```
RESOURCE_GROUP(group_name)
```

*group_name*은 문 실행 기간 동안 스레드가 할당되어야 하는 리소스 그룹을 나타냅니다. 그룹이 존재하지 않으면 경고가 발생하고 힌트가 무시됩니다.

RESOURCE_GROUP 힌트는 초기 문 키워드(SELECT, INSERT, REPLACE, UPDATE 또는 DELETE) 뒤에 나타나야 합니다.

RESOURCE_GROUP의 대체 문으로, 리소스 그룹에 스레드를 일시적으로 할당하는 SET RESOURCE GROUP 문을 사용할 수 있습니다. 섹션 13.7.2.4, "SET RESOURCE GROUP 문"을 참고한다.

쿼리 블록 이름 지정을 위한 옵티마이저 힌트

테이블 수준, 인덱스 수준 및 하위 쿼리 최적화 도구 힌트를 사용하면 특정 쿼리 블록에 인수 구문의 일부로 이름을 지정할 수 있습니다. 이러한 이름을 만들려면 해당 이름이 발생하는 쿼리 블록에 이름을 할당하는 QB_NAME 힌트를 사용합니다:

```
QB_NAME(name)
```

QB_NAME 힌트를 사용하면 다른 힌트가 적용되는 쿼리 블록을 명확하게 지정할 수 있습니다. 또한 복잡한 문을 더 쉽게 이해할 수 있도록 단일 힌트 주석 내에 쿼리 블록 이름이 아닌 모든 힌트를 지정할 수 있습니다. 다음 문을 예로 들어 보겠습니다:

```
선택 ...
  에서 (선택 ...
  에서 (선택 ...에서 ...)) ...
```

QB_NAME 힌트는 문에서 쿼리 블록에 이름을 할당합니다:

```
SELECT /*+ QB_NAME(qb1) */ ...
  FROM (SELECT /*+ QB_NAME(qb2) */ ...
  FROM (SELECT /*+ QB_NAME(qb3) */ ... FROM ...)) ...
```

그러면 다른 힌트에서 해당 이름을 사용하여 적절한 쿼리 블록을 참조할 수 있습니다:

```
SELECT /*+ QB_NAME(qb1) MRR(@qb1 t1) BKA(@qb2) NO_MRR(@qb3t1 idx1, id2) */ ...
  FROM (SELECT /*+ QB_NAME(qb2) */ ...
  FROM (SELECT /*+ QB_NAME(qb3) */ ... FROM ...)) ...
```

결과 효과는 다음과 같습니다:

- `MRR(@qb1 t1)`은 쿼리 블록 `qb1`의 테이블 `t1`에 적용됩니다.
- `BKA(@qb2)`는 쿼리 블록 `qb2`에 적용됩니다.
- `NO_MRR(@qb3 t1 idx1, id2)`은 쿼리 블록 `qb3`의 테이블 `t1`에 있는 인덱스 `idx1` 및 `idx2`에 적용됩니다.

쿼리 블록 이름은 식별자이며 유효한 이름과 인용하는 방법에 대한 일반적인 규칙을 따릅니다(9.2절. "스키마 객체 이름" 참조). 예를 들어 공백이 포함된 쿼리 블록 이름은 반드시 따옴표로 묶어야 하며, 백틱을 사용하여 따옴표로 묶을 수 있습니다:

```
SELECT /*+ BKA(@`내 힌트 이름`) */ ...
  FROM (SELECT /*+ QB_NAME(`내 힌트 이름`) */ ...) ...
```

ANSI_QUOTES SQL 모드가 활성화된 경우 쿼리 블록 이름을 큰따옴표 안에 따옴표로 묶을 수도 있습니다:

```
SELECT /*+ BKA(@"내 힌트 이름") */ ...
  FROM (SELECT /*+ QB_NAME("내 힌트 이름") */ ...) ...
```

8.9.4 색인 힌트

인덱스 힌트는 쿼리 처리 중에 인덱스를 선택하는 방법에 대한 정보를 옵티마이저에 제공합니다. 여기에 설명된 인덱스 힌트는 8.9.3절. "최적화 도구 힌트"에 설명된 최적화 도구 힌트와 다릅니다. 인덱스 힌트와 옵티마이저 힌트는 별도로 또는 함께 사용할 수 있습니다.

인덱스 힌트는 SELECT 및 UPDATE 문에 적용됩니다. 또한 다중 테이블 DELETE 문을 사용할 수 있지만, 이 섹션의 뒷부분에서 설명하는 것처럼 단일 테이블 DELETE 문은 사용할 수 없습니다.

인덱스 힌트는 테이블 이름 뒤에 지정됩니다. (SELECT 문에서 테이블을 지정하는 일반적인 구문에 대해서는 섹션 13.2.13.2, "JOIN 절"을 참조하십시오.) 인덱스 힌트를 포함하여 개별 테이블을 참조하는 구문은 다음과 같습니다:

```
tbl_name [[AS] 별칭] [index_hint_list]

index_hint_list:
    index_hint [index_hint] ...

index_hint:
    인덱스|키} 사용
      [FOR {JOIN|주문 기준|그룹 기준}] ([index_list])
  | {무시|강제} {인덱스|키}
      [FOR {JOIN|주문 기준|그룹 기준}] (index_list)

index_list:
    index_name [, index_name] ...
```

USE INDEX(*index_list*) 힌트는 테이블에서 행을 찾을 때 명명된 인덱스 중 하나만 사용하도록 MySQL에 지시합니다. 대체 구문인 IGNORE INDEX(*index_list*)는 특정 인덱스 또는 인덱스를 사용하지 말라고 MySQL에 지시합니다. 이러한 힌트는 EXPLAIN에서 MySQL이 사용 가능한 인덱스 목록에서 잘못된 인덱스를 사용하고 있는 것으로 표시되는 경우에 유용합니다.

FORCE INDEX 힌트는 USE INDEX(*index_list*)와 같은 역할을 하지만 테이블 스캔이 *매우* 비싸다고 가정한다는 점이 추가됩니다. 즉, 테이블 스캔은 테이블에서 행을 찾기 위해 명명된 인덱스 중 하나를 사용할 방법이 없는 경우에만 사용됩니다.

참고

MySQL 8.2는 인덱스 수준 옵티마이저 힌트 `JOIN_INDEX`, `GROUP_INDEX`, `ORDER_INDEX` 및 INDEX를 지원하며, 이는 `FORCE INDEX` 인덱스 힌트와 동일하고 이를 대체하기 위한 것입니다. NO_JOIN_INDEX, `NO_GROUP_INDEX`, `NO_ORDER_INDEX` 및 `NO_INDEX` 옵티마이저 힌트를 추가했는데, 이는 `IGNORE INDEX` 인덱스 힌트와 동일하며 이를 대체하기 위한 것입니다. 따라서 `USE INDEX`, `FORCE INDEX` 및 `IGNORE INDEX`는 향후 MySQL 릴리스에서 더 이상 사용되지 않으며, 이후 언젠가는 완전히 제거될 것으로 예상해야 합니다.

이러한 인덱스 수준 옵티마이저 힌트는 단일 테이블 및 다중 테이블 `DELETE` 문 모두에서 지원됩니다.

자세한 내용은 인덱스 수준 최적화 도구 힌트를 참조하세요.

각 힌트에는 열 이름이 아닌 인덱스 이름이 필요합니다. 기본 키를 참조하려면 PRIMARY라는 이름을 사용합니다. 테이블의 인덱스 이름을 보려면 SHOW INDEX 문 또는 정보 스키마 통계 테이블을 사용합니다.

index_name 값은 전체 인덱스 이름일 필요는 없습니다. 인덱스 이름의 명확한 접두사일 수 있습니다. 접두사가 모호하면 오류가 발생합니다.

예시:

```
SELECT * FROM table1 USE INDEX (col1_index,col2_index)
  WHERE col1=1 AND col2=2 AND col3=3;

SELECT * FROM table1 IGNORE INDEX (col3_index)
  WHERE col1=1 AND col2=2 AND col3=3;
```

인덱스 힌트 구문에는 다음과 같은 특징이 있습니다:

- "인덱스를 사용하지 않음"을 의미하는 USE INDEX에서 *index_list*를 생략하는 것은 구문상 유효합니다. 생략 FORCE INDEX 또는 IGNORE INDEX에 대한 *index_list*는 구문 오류입니다.
- 힌트에 FOR 절을 추가하여 인덱스 힌트의 범위를 지정할 수 있습니다. 이렇게 하면 쿼리 처리의 다양한 단계에 대한 실행 계획의 옵티마이저 선택을 보다 세밀하게 제어할 수 있습니다. MySQL이 테이블에서 행을 찾는 방법과 조인을 처리하는 방법을 결정할 때 사용되는 인덱스에만 영향을 미치려면 FOR JOIN을 사용합니다. 행을 정렬하거나 그룹화할 때 인덱스 사용에 영향을 미치려면 FOR ORDER BY 또는 FOR GROUP BY를 사용합니다.
- 여러 인덱스 힌트를 지정할 수 있습니다:

  ```
  SELECT * FROM t1 USE INDEX (i1) IGNORE INDEX FOR ORDER BY (i2) ORDER BY a;
  ```

  동일한 힌트 내에서도 여러 힌트에서 동일한 인덱스의 이름을 지정하는 것은 오류가 아닙니다:

  ```
  SELECT * FROM t1 USE INDEX (i1) USE INDEX (i1,i1);
  ```

  그러나 동일한 테이블에 대해 USE INDEX와 FORCE INDEX를 혼용하는 것은 오류입니다:

  ```
  SELECT * FROM t1 USE INDEX FOR JOIN (i1) FORCE INDEX FOR JOIN (i2);
  ```

인덱스 힌트에 FOR 절이 포함되지 않은 경우 힌트의 범위는 문의 모든 부분에 적용됩니다. 예를 들어, 이 힌트입니다:

```
인덱스 무시(i1)
```

는 이 힌트 조합과 동일합니다:

```
IGNORE INDEX FOR JOIN (i1)
IGNORE INDEX FOR ORDER BY (i1)
IGNORE INDEX FOR GROUP BY (i1)
```

MySQL 5.0에서는 FOR 절이 없는 힌트 범위가 행 검색에만 적용되었습니다. FOR 절이없을 때 서버가 이 이전 동작을 사용하도록 하려면 서버를 시작할 때 이전 시스템 변수를 활성화합니다. 복제 설정에서 이 변수를 활성화할 때는 주의하세요. 문 기반 바이너리 로깅을 사용하면 원본과 복제본에 대해 서로 다른 모드를 사용하면 복제 오류가 발생할 수 있습니다.

인덱스 힌트가 처리될 때 유형별(USE, FORCE, IGNORE) 및 범위별(FOR JOIN, FOR ORDER BY, FOR GROUP BY)로 단일 목록으로 수집됩니다. 예를 들어

```
SELECT * FROM t1
  USE INDEX () IGNORE INDEX (i2) USE INDEX (i1) USE INDEX (i2);
```

와 동일합니다:

```
SELECT * FROM t1
   USE INDEX (i1,i2) IGNORE INDEX (i2);
```

그러면 인덱스 힌트가 각 범위에 대해 다음 순서로 적용됩니다:

1. {사용|힘} INDEX가 있으면 적용됩니다. (그렇지 않은 경우 옵티마이저가 결정한 인덱스 집합이 사용됩니다.)

2. 이전 단계의 결과 위에 인덱스 무시가 적용됩니다. 예를 들어 다음 두 쿼리는 동일합니다:

```
SELECT * FROM t1 USE INDEX (i1) IGNORE INDEX (i2) USE INDEX (i2);

SELECT * FROM t1 USE INDEX (i1);
```

전체 텍스트 검색의 경우 색인 힌트는 다음과 같이 작동합니다:

- 자연어 모드 검색의 경우, 인덱스 힌트는 자동으로 무시됩니다. 예를 들어, IGNORE INDEX(i1)은 경고 없이 무시되고 인덱스는 계속 사용됩니다.

- 부울 모드 검색의 경우 FOR ORDER BY 또는 FOR GROUP BY가 포함된 인덱스 힌트는 자동으로 무시됩니다. FOR JOIN이 있거나 FOR 수정자가 없는 인덱스 힌트는 무시됩니다. 전체 텍스트가 아닌 검색에 힌트가 적용되는 방식과 달리, 힌트는 쿼리 실행의 모든 단계(행 찾기 및 검색, 그룹화 및 순서 지정)에 사용됩니다. 이는 전체 텍스트가 아닌 인덱스에 대한 힌트가 제공된 경우에도 마찬가지입니다.

  예를 들어 다음 두 쿼리는 동일한 쿼리입니다:

```
SELECT * FROM t
  사용 인덱스(index1)
  IGNORE INDEX FOR ORDER BY (index1)
  IGNORE INDEX FOR GROUP BY (index1)
  WHERE ... 부울 모드에서 ... ;

SELECT * FROM t
  사용 인덱스(index1)
  WHERE ... 부울 모드에서 ... ;
```

인덱스 힌트는 DELETE 문과 함께 작동하지만, 여기에 표시된 것처럼 다중 테이블 DELETE 구문을 사용하는 경우에만 작동합니다:

```
mysql> EXPLAIN DELETE FROM t1 USE INDEX(col2)
    -> WHERE col1은 1과 100 사이이고 COL2는 1과 100\G 사이입니다.
오류 1064 (42000): SQL 구문에 오류가 있습니다. 1줄의 'use index(col2) where col1은 1에서 100 사이
, col2는 1에서 100 사이' 근처에 사용할 올바른 구문이 있는지 MySQL 서버 버전에 해당하는 매뉴얼을 확인하세요
.

mysql> EXPLAIN DELETE t1.* FROM t1 USE INDEX(col2)
    -> WHERE col1은 1과 100 사이이고 COL2는 1과 100\G 사이입니다.
*************************** 1. 행 ***************************
           ID: 1
  select_type: DELETE 테이
        블: t1
   파티션: NULL
         유형: 범위 가능_키
: COL2
          키: COL2
      KEY_LEN: 5
          참조:
         NULL 행:
         72
     필터링됨: 11.11 추가: 어
        디 사용
세트 1행, 경고 1회(0.00초)
```

8.9.5 옵티마이저 비용 모델

실행 계획을 생성하기 위해 옵티마이저는 쿼리 실행 중에 발생하는 다양한 작업의 비용 추정치를 기반으로 하는 비용 모델을 사용합니다. 옵티마이저에는 실행 계획에 관한 결정을 내리는 데 사용할 수 있는 컴파일된 기본 '비용 상수' 집합이 있습니다.

옵티마이저에는 실행 계획 구성 중에 사용할 비용 추정 데이터베이스도 있습니다. 이러한 추정치는 `mysql` 시스템 데이터베이스의 `server_cost` 및 `engine_cost` 테이블에 저장됩니다.

테이블은 언제든지 구성할 수 있습니다. 이 테이블의 목적은 옵티마이저가 쿼리 실행 계획에 도달하려고 할 때 사용하는 비용 추정치를 쉽게 조정할 수 있도록 하는 것입니다.

- 비용 모델 일반 운영
- 비용 모델 데이터베이스
- 비용 모델 데이터베이스 변경하기

비용 모델 일반 운영

구성 가능한 옵티마이저 비용 모델은 다음과 같이 작동합니다:

- 서버는 시작할 때 비용 모델 테이블을 메모리로 읽고 런타임에 메모리 내 값을 사용합니다. 테이블에 지정된 `NULL`이 아닌 모든 비용 추정치가 해당 컴파일된 기본 비용 상수입니다. `NULL` 추정값은 컴파일된 기본값을 사용하도록 최적화 프로그램에 지시합니다.
- 런타임에 서버는 비용 테이블을 다시 읽을 수 있습니다. 이는 스토리지 엔진이 동적으로 로드되거나 `FLUSH OPTIMIZER_COSTS` 문이 실행될 때 발생합니다.
- 비용 테이블을 사용하면 서버 관리자가 테이블의 항목을 변경하여 비용 추정치를 쉽게 조정할 수 있습니다. 또한 항목의 비용을 `NULL`로 설정하여 기본값으로 쉽게 되돌릴 수 있습니다. 옵티마이저는 인메모리 비용 값을 사용하므로 테이블을 변경한 후 `FLUSH OPTIMIZER_COSTS`를 수행해야 적용됩니다.
- 클라이언트 세션이 시작될 때 현재 메모리 내 비용 추정치는 세션이 종료될 때까지 해당 세션 전체에 적용됩니다. 특히 서버가 비용 테이블을 다시 읽는 경우 변경된 추정치는 이후에 시작된 세션에만 적용됩니다. 기존 세션은 영향을 받지 않습니다.
- 비용 테이블은 특정 서버 인스턴스에만 적용됩니다. 서버는 비용 테이블 변경 사항을 복제본에 복제하지 않습니다.

비용 모델 데이터베이스

옵티마이저 비용 모델 데이터베이스는 쿼리 실행 중에 발생하는 작업에 대한 비용 추정 정보를 포함하는 두 개의 테이블로 구성된 `mysql` 시스템 데이터베이스로 구성됩니다:

- `server_cost`: 일반 서버 운영에 대한 옵티마이저 비용 추정치
- `engine_cost`: 특정 스토리지 엔진과 관련된 작업에 대한 옵티마이저 비용 추정치

`server_cost` 테이블에는 이러한 열이 포함되어 있습니다:

- 비용_이름

  비용 모델에서 사용되는 비용 견적의 이름입니다. 이 이름은 대소문자를 구분하지 않습니다. 서버가 이 테이블을 읽을 때 비용 이름을 인식하지 못하면 오류 로그에 경고를 기록합니다.

- 비용_값

  비용 예상 값입니다. 값이 NULL이 아닌 경우 서버는 이 값을 비용으로 사용합니다. 그렇지 않으면 기본 견적(컴파일된 값)을 사용합니다. DBA는 이 열을 업데이트하여 비용 견적을 변경할 수 있습니다. 서버가 이 테이블을 읽을 때 비용 값이 유효하지 않은(양수가 아닌) 것을 발견하면 오류 로그에 경고를 기록합니다.

  기본 비용 견적을 재정의하려면(NULL을 지정하는 항목의 경우) 비용을 NULL이 아닌 값으로 설정합니다. 기본값으로 되돌리려면 값을 NULL로 설정합니다. 그런 다음 FLUSH OPTIMIZER_COSTS를 실행하여 서버에 비용 테이블을 다시 읽으라고 지시합니다.

- last_update

마지막 행 업데이트 시간입니다.

- 댓글

  비용 견적과 관련된 설명 댓글입니다. DBA는 이 열을 사용하여 비용 견적 행에 특정 값이 저장되는 이유에 대한 정보를 제공할 수 있습니다.

- 기본값

  비용 견적의 기본(컴파일된) 값입니다. 이 열은 읽기 전용으로 생성된 열로, 관련 비용 견적이 변경되더라도 해당 값이 유지됩니다. 런타임에 테이블에 추가된 행의 경우 이 열의 값은 `NULL`입니다.

`server_cost` 테이블의 기본 키는 `cost_name` 열이므로 비용 견적에 대해 여러 항목을 만들 수 없습니다.

서버는 `server_cost` 테이블에 대해 이러한 `cost_name` 값을 인식합니다:

- `DISK_TEMPTABLE_CREATE_COST`, `DISK_TEMPTABLE_ROW_COST`

  디스크 기반 스토리지 엔진(`InnoDB` 또는 `MyISAM`)에 저장된 내부적으로 생성된 임시 테이블에 대한 비용 추정치입니다. 이 값을 높이면 내부 임시 테이블 사용에 대한 비용 추정치가 증가하고 옵티마이저가 임시 테이블을 덜 사용하는 쿼리 계획을 선호하게 됩니다. 이러한 테이블에 대한 자세한 내용은 섹션 8.4.4, "MySQL에서 내부 임시 테이블 사용"을 참조하세요.

  이러한 디스크 매개 변수의 기본값이 해당 메모리 매개 변수의 기본값(`memory_temptable_create_cost`)에 비해 클수록 더 큰 기본값입니다, `memory_temptable_row_cost`)는 디스크 기반 테이블을 처리하는 데 드는 더 큰 비용을 반영합니다.

- 키_비교_비용

  레코드 키를 비교하는 데 드는 비용입니다. 이 값을 늘리면 많은 키를 비교하는 쿼리 계획이 더 비싸집니다. 예를 들어, 파일 정렬을 수행하는 쿼리 계획은 인덱스를 사용하여 정렬을 피하는 쿼리 계획에 비해 상대적으로 더 비쌉니다.

- `memory_temptable_create_cost`, `memory_temptable_row_cost`

  메모리 스토리지 엔진에 저장된 내부적으로 생성된 임시 테이블에 대한 비용 추정치입니다. 이 값을 높이면 내부 임시 테이블 사용에 대한 비용 추정치가 증가하고 옵티마이저가 임시 테이블을 덜 사용하는 쿼리 계획을 선호하게 됩니다. 이러한 테이블에 대한 자세한 내용은 섹션 8.4.4, "MySQL의 내부 임시 테이블 사용"을 참조하세요.

  이러한 메모리 매개변수의 기본값이 해당 디스크 매개변수(`disk_temptable_create_cost`, `disk_temptable_row_cost`)의 기본값에 비해 작다는 것은 메모리 기반 테이블을 처리하는 데 드는 비용이 더 적다는 것을 반영합니다.

- 행_평가_비용

레코드 조건을 평가하는 데 드는 비용입니다. 이 값을 늘리면 많은 행을 검사하는 쿼리 계획이 적은 행을 검사하는 쿼리 계획에 비해 비용이 더 많이 듭니다. 예를 들어, 테이블 스캔은 더 적은 행을 읽는 범위 스캔에 비해 상대적으로 더 비쌉니다.

엔진_비용 테이블에는 이러한 열이 포함되어 있습니다:

- 엔진_이름

  이 비용 견적이 적용되는 스토리지 엔진의 이름입니다. 이름은 대소문자를 구분하지 않습니다. 이 값이 기본값인 경우 자체 명명된 항목이 없는 모든 스토리지 엔진에 적용됩니다. 서버가 이 테이블을 읽을 때 엔진 이름을 인식하지 못하면 오류 로그에 경고를 기록합니다.

- 장치 유형

  이 비용 견적이 적용되는 장치 유형입니다. 이 열은 하드 디스크 드라이브와 솔리드 스테이트 드라이브와 같은 다양한 저장 장치 유형에 대해 서로 다른 비용 견적을 지정하기 위한 것입니다. 현재 이 정보는 사용되지 않으며 0만이 허용되는 유일한 값입니다.

- 비용_이름

  server_cost 테이블과 동일합니다.

- 비용_값

  server_cost 테이블과 동일합니다.

- last_update

  server_cost 테이블과 동일합니다.

- 댓글

  server_cost 테이블과 동일합니다.

- 기본값

  비용 견적의 기본(컴파일된) 값입니다. 이 열은 읽기 전용으로 생성된 열로, 관련 비용 견적이 변경되더라도 해당 값이 유지됩니다. 비용 견적에 추가된 행의 경우
  테이블에서 이 열의 값은 NULL이지만, 행이 원래 행 중 하나와 동일한 cost_name 값을 갖는 경우 default_value 열의 값은 해당 행과 동일하다는 예외가 있습니다.

engine_cost 테이블의 기본 키는 (cost_name, engine_name, device_type) 열로 구성된 튜플이므로 해당 열의 값 조합에 대해 여러 항목을 만들 수 없습니다.

서버는 엔진 비용 테이블에 대해 이러한 비용_이름 값을 인식합니다:

- io_block_read_cost

  디스크에서 인덱스 또는 데이터 블록을 읽는 데 드는 비용입니다. 이 값을 늘리면 디스크 블록을 많이 읽는 쿼리 계획이 디스크 블록을 적게 읽는 쿼리 계획에 비해 비용이 더 많이 듭니다. 예를 들어, 테이블 스캔은 더 적은 블록을 읽는 범위 스캔에 비해 상대적으로 비용이 더 많이 듭니다.

- 메모리 블록 읽기 비용

  io_block_read_cost와 유사하지만, 인메모리 데이터베이스 버퍼에서 인덱스 또는 데이터 블록을 읽는 데 드는 비용을 나타냅니다.

io_block_read_cost와 memory_block_read_cost 값이 다른 경우, 동일한 쿼리를 두 번 실행할 때마다 실행 계획이 변경될 수 있습니다. 메모리 액세스 비용이 디스크 액세스 비용보다 작다고 가정해 보겠습니다. 이

경우, 데이터가 버퍼 풀로 읽혀지기 전에 서버를 시작할 때 데이터가 메모리에 있기 때문에 쿼리가 실행된 후와 다른 계획이 나올 수 있습니다.

비용 모델 데이터베이스 변경하기

비용 모델 매개 변수를 기본값에서 변경하려는 DBA의 경우 값을 두 배 또는 절반으로 늘리고 그 효과를 측정해 보세요.

`io_block_read_cost` 및 `memory_block_read_cost` 매개변수를 변경하면 가치 있는 결과를 얻을 가능성이 가장 높습니다. 이러한 매개변수 값을 사용하면 데이터 액세스 방법의 비용 모델에서 다양한 소스에서 정보를 읽는 비용, 즉 디스크에서 정보를 읽는 비용과 이미 메모리 버퍼에 있는 정보를 읽는 비용을 고려할 수 있습니다.

예를 들어, 다른 모든 조건이 동일하다면 io_block_read_cost를 memory_block_read_cost보다 큰 값으로 설정하면 옵티마이저가 디스크에서 읽어야 하는 계획보다 메모리에 이미 저장된 정보를 읽는 쿼리 계획을 선호하게 됩니다.

이 예는 io_block_read_cost의 기본값을 변경하는 방법을 보여줍니다:

```
UPDATE mysql.engine_cost
  SET cost_value = 2.0
  WHERE cost_name = 'io_block_read_cost';
FLUSH OPTIMIZER_COSTS;
```

이 예제는 InnoDB 스토리지 엔진에 대해서만 io_block_read_cost의 값을 변경하는 방법을 보여줍니다:

```
mysql.engine_cost에 삽입
  VALUES ('InnoDB', 0, 'io_block_read_cost', 3.0,
  CURRENT_TIMESTAMP, 'InnoDB에 느린 디스크 사용');
flush optimizer costs;
```

8.9.6 옵티마이저 통계

column_statistics 데이터 사전 테이블은 옵티마이저가 쿼리 실행 계획을 구성할 때 사용할 수 있도록 열 값에 대한 히스토그램 통계를 저장합니다. 히스토그램 관리를 수행하려면 ANALYZE TABLE 문을 사용합니다.

column_statistics 테이블에는 이러한 특성이 있습니다:

- 이 테이블에는 기하 도형 유형(공간 데이터)을 제외한 모든 데이터 유형의 열에 대한 통계가 포함되어 있습니다. JSON.
- 테이블은 영구적이므로 서버를 시작할 때마다 열 통계를 만들 필요가 없습니다.
- 서버는 테이블에 대한 업데이트를 수행하지만 사용자는 수행하지 않습니다.

column_statistics 테이블은 데이터 사전의 일부이므로 사용자가 직접 액세스할 수 없습니다. 히스토그램 정보는 데이터 사전 테이블에 뷰로 구현된 INFORMATION_SCHEMA.COLUMN_STATISTICS를 사용하여 사용할 수 있습니다. COLUMN_STATISTICS에는 이러한 열이 있습니다:

- 스키마_이름, 테이블_이름, 칼럼_이름: 통계가 적용되는 스키마, 테이블 및 열의 이름입니다.
- 히스토그램: 히스토그램으로 저장된 열 통계를 설명하는 JSON 값입니다.

열 히스토그램에는 열에 저장된 값 범위의 일부에 대한 버킷이 포함되어 있습니다. 히스토그램은 열 통계를 유연하게 표현할 수 있는 JSON 객체입니다. 다음은 히스토그램 객체 샘플입니다:

```
{
  "버킷": [ [
      1,
      0.3333333333333333
    ],
    [
      2,
      0.6666666666666666
    ],
    [
      3,
      1
    ]
  ],
  "널 값": 0,
  "마지막 업데이트": "2017-03-24 13:32:40.000000",
```

```
  "샘플링 속도": 1,
  "히스토그램 유형": "싱글톤", "지정된 버킷
  수": 128, "데이터 유형": "int",
  "collation-id": 8
}
```

히스토그램 개체에는 이러한 키가 있습니다:

- 버킷: 히스토그램 버킷입니다. 버킷 구조는 히스토그램 유형에 따라 다릅니다. 싱글톤 히

  스토그램의 경우 버킷에는 두 개의 값이 포함됩니다:

  - 값 1: 버킷의 값입니다. 유형은 열 데이터 유형에 따라 다릅니다.
  - 값 2: 값의 누적 빈도를 나타내는 배입니다. 예를 들어 .25 및 .75는 열에 있는 값의 25% 및 75%가 버킷 값보다 작거나 같음을 나타냅니다.

  등고 히스토그램의 경우 버킷에는 네 가지 값이 포함됩니다:

  - 값 1, 2: 버킷의 하위 및 상위 포함 값입니다. 유형은 열 데이터 유형에 따라 다릅니다.
  - 값 3: 값의 누적 빈도를 나타내는 배수입니다. 예를 들어 .25 및 .75는 열에 있는 값의 25% 및 75%가 버킷 상한 값보다 작거나 같음을 나타냅니다.
  - 값 4: 버킷 하위 값에서 상위 값까지의 범위에 있는 고유 값의 개수입니다.
- 널 값: 0.0에서 1.0 사이의 숫자로, SQL인 열 값의 비율을 나타냅니다.
  NULL 값. 0이면 열에 NULL 값이 없는 것입니다.
- 마지막 업데이트: 히스토그램이 생성된 시점으로, UTC 값(*YYYY-MM-DD hh:mm:ss.uuuuu* 형식)입니다.
- 샘플링 비율: 히스토그램을 생성하기 위해 샘플링된 데이터의 비율을 나타내는 0.0에서 1.0 사이의 숫자입니다. 값이 1이면 모든 데이터를 읽었음을 의미합니다(샘플링 없음).
- 히스토그램 유형: 히스토그램 유형입니다:
  - 싱글톤: 하나의 버킷은 열에서 하나의 단일 값을 나타냅니다. 이 히스토그램 유형은 열의 고유 값 수가 히스토그램을 생성한 분석 테이블 문에 지정된 버킷 수보다 작거나 같을 때 만들어집니다.
  - 등고선: 하나의 버킷은 값의 범위를 나타냅니다. 이 히스토그램 유형은 열의 고유 값 수가 히스토그램을 생성한 분석 테이블 문에 지정된 버킷 수보다 많을 때 생성됩니다.
- 지정된 버킷 수입니다: 분석 테이블에 지정된 버킷 수입니다.
  문으로 히스토그램을 생성했습니다.
- 데이터 유형: 이 히스토그램에 포함된 데이터 유형입니다. 이는 영구 저장소에서 메모리로 히스토그램을 읽고 구문 분석할 때 필요합니다. 값은 int, uint(부호 없는 정수), double, 10진수, 날짜/시간 또는 문자

열(문자 및 이진 문자열 포함) 중 하나입니다.

- `collation-id`: 히스토그램 데이터의 데이터 정렬 ID입니다. 데이터 유형 값이 문자열일 때 주로 의미가 있습니다. 값은 정보 스키마 `COLLATIONS` 테이블의 `ID` 열 값에 해당합니다.

히스토그램 개체에서 특정 값을 추출하려면 `JSON` 연산을 사용할 수 있습니다. 예를 들면 다음과 같습니다:

```
mysql> SELECT
         테이블_이름, 열_이름,
```

```
        HISTOGRAM->>'$."data-type"' AS '데이터 유형',
        JSON_LENGTH(HISTOGRAM->>'$."buckets"') AS 'bucket-count'
      정보_schema.column_statistics에서;
+-----------------+-------------+-----------+ -------------------+
|                 TABLE_NAME| COLUMN_NAME | 데이터 유형 | 버킷 수 |
+-----------------+-------------+-----------+ ---|-----------226-|
                                                  |          1024 |
| 국가|            인구 | int                    |           457 |
| 국가 언어 | 언어 | 문자열
+-----------------+-------------+-----------+ -------------------+
| 도시 |           인구 | int
```

최적화 도구는 통계가 수집되는 모든 데이터 유형의 열에 대해 해당되는 경우 히스토그램 통계를 사용합니다. 최적화 도구는 히스토그램 통계를 적용하여 상수 값에 대한 열 값 비교의 선택성(필터링 효과)을 기반으로 행 추정치를 결정합니다. 이러한 형식의 술어는 히스토그램 사용에 적합합니다:

```
col_name = 상수
col_name <> 상수
col_name != 상수
col_name >= 상수
col_name <= 상수
col_name >= 상수
col_name <= 상수
col_name IS NULL
col_name IS NOT NULL
col_name 상수와 상수 사이 col_name 상수와 상수 사이
가 아님 col_name IN (상수[, 상수] ...) col_name
NOT IN (상수[, 상수] ...)
```

예를 들어 이러한 문에는 히스토그램 사용에 적합한 술어가 포함되어 있습니다:

```
SELECT * FROM orders WHERE 금액이 100.0 AND 300.0 사이인 경우;
SELECT * FROM tbl WHERE col1 = 15 AND col2 > 100입니다;
```

상수 값에 대한 비교 요구 사항에는 다음과 같이 상수인 함수가 포함됩니다.

ABS() 및 FLOOR():

```
SELECT * FROM tbl WHERE col1 < ABS(-34);
```

히스토그램 통계는 주로 인덱싱되지 않은 열에 유용합니다. 히스토그램 통계를 적용할 수 있는 열에 인덱스를 추가하면 옵티마이저가 행을 추정하는 데 도움이 될 수도 있습니다. 단점이 있습니다:

- 테이블 데이터가 수정되면 인덱스를 업데이트해야 합니다.
- 히스토그램은 필요할 때만 생성되거나 업데이트되므로 테이블 데이터가 수정될 때 오버헤드가 추가되지 않습니다. 반면에 테이블 수정이 발생하면 통계는 다음에 업데이트될 때까지 점점 더 오래된 통계가 됩니다.

옵티마이저는 히스토그램 통계에서 얻은 것보다 범위 옵티마이저 행 추정치를 선호합니다. 옵티마이저가 범위 옵티마이저가 적용된다고 판단하면 히스토그램 통계를 사용하지 않습니다.

인덱싱된 열의 경우 인덱스 다이빙을 사용하여 동일성 비교를 위한 행 추정치를 얻을 수 있습니다(섹션 8.2.1.2,

"범위 최적화" 참조). 이 경우 인덱스 다이브를 사용하면 더 나은 추정치를 얻을 수 있으므로 히스토그램 통계가 반드시 유용하지는 않습니다.

경우에 따라 히스토그램 통계를 사용해도 쿼리 실행이 개선되지 않을 수 있습니다(예: 통계가 오래된 경우). 이 경우인지 확인하려면 테이블 분석을 사용하여 히스토그램 통계를 다시 생성한 다음 쿼리를 다시 실행합니다.

또는 히스토그램 통계를 비활성화하려면 분석 테이블을 사용하여 삭제합니다. 히스토그램 통계를 비활성화하는 다른 방법은 `optimizer_switch` 시스템 변수의 `condition_fanout_filter` 플래그를 해제하는 것입니다(이 경우 다른 최적화도 비활성화될 수 있음):

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

히스토그램 통계가 사용되는 경우 결과 효과는 EXPLAIN을 사용하여 볼 수 있습니다. 열 col1에 사용할 수 있는 인덱스가 없는 다음 쿼리를 예로 들어 보겠습니다:

```
SELECT * FROM t1 WHERE col1 < 24;
```

히스토그램 통계에 따르면 t1의 행 중 57%가 col1 < 24 술어를 만족하는 것으로 나타나면 인덱스가 없는 경우에도 필터링이 수행될 수 있으며, 필터링된 열에 57.00이 표시됩니다.

8.10 버퍼링 및 캐싱

MySQL은 메모리 버퍼에 정보를 캐시하여 성능을 향상시키는 몇 가지 전략을 사용합니다.

8.10.1 InnoDB 버퍼 풀 최적화

InnoDB는 메모리에서 데이터와 인덱스를 캐싱하기 위해 버퍼 풀이라는 저장 영역을 유지합니다. InnoDB 버퍼 풀의 작동 방식을 파악하고 이를 활용하여 자주 액세스하는 데이터를 메모리에 보관하는 것은 MySQL 튜닝의 중요한 측면입니다.

InnoDB 버퍼 풀의 내부 작동에 대한 설명, LRU 교체 알고리즘에 대한 개요 및 일반 구성 정보는 15.5.1 절. "버퍼 풀"을 참조하세요.

추가적인 InnoDB 버퍼 풀 구성 및 튜닝 정보는 다음 섹션을 참조하세요:

- 섹션 15.8.3.4, "InnoDB 버퍼 풀 프리페칭 구성(읽기 앞서)"
- 섹션 15.8.3.5, "버퍼 풀 플러싱 구성하기"
- 섹션 15.8.3.3, "버퍼 풀 스캔 저항성 만들기"
- 섹션 15.8.3.2, "여러 버퍼 풀 인스턴스 구성"
- 섹션 15.8.3.6, "버퍼 풀 상태 저장 및 복원"
- 섹션 15.8.3.1, "InnoDB 버퍼 풀 크기 구성"

8.10.2 MyISAM 키 캐시

디스크 I/O를 최소화하기 위해 MyISAM 스토리지 엔진은 많은 데이터베이스 관리 시스템에서 사용하는 전략을 활용합니다. 캐시 메커니즘을 사용하여 가장 자주 액세스하는 테이블 블록을 메모리에 보관합니다:

- 인덱스 블록의 경우 *키 캐시*(또는 *키 버퍼*)라는 특수 구조가 유지됩니다. 이 구조에는 가장 많이 사용되는 인덱스 블록이 배치되는 여러 블록 버퍼가 포함되어 있습니다.
- 데이터 블록의 경우 MySQL은 특별한 캐시를 사용하지 않습니다. 대신 기본 운영 체제 파일 시스템 캐시에 의존합니다.

이 섹션에서는 먼저 `MyISAM` 키 캐시의 기본 작동에 대해 설명합니다. 그런 다음 키 캐시 성능을 개선하고 캐시 작동을 더 잘 제어할 수 있는 기능에 대해 설명합니다:

- 여러 세션이 동시에 캐시에 액세스할 수 있습니다.
- 여러 개의 키 캐시를 설정하고 특정 캐시에 테이블 인덱스를 할당할 수 있습니다.

키 캐시의 크기를 제어하려면 `key_buffer_size` 시스템 변수를 사용합니다. 이 변수를 0으로 설정하면 키 캐시가 사용되지 않습니다. 키 캐시 값이 너무 작아서 최소 블록 버퍼 수(8)를 할당할 수 없는 경우에도 키 캐시는 사용되지 않습니다.

키 캐시가 작동하지 않을 때는 운영 체제에서 제공하는 기본 파일 시스템 버퍼링만을 사용하여 인덱스 파일에 액세스합니다. (즉, 테이블 인덱스 블록은 테이블 데이터 블록에 사용되는 것과 동일한 전략을 사용하여 액세스됩니다.)

인덱스 블록은 MyISAM 인덱스 파일에 대한 연속적인 액세스 단위입니다. 일반적으로 인덱스 블록의 크기는 인덱스 B-트리의 노드 크기와 같습니다. (인덱스는 B-트리 데이터 구조를 사용하여 디스크에 표시됩니다. 트리의 맨 아래에 있는 노드가 리프 노드입니다. 리프 노드 위의 노드는 비리프 노드입니다.)

키 캐시 구조의 모든 블록 버퍼는 크기가 동일합니다. 이 크기는 테이블 인덱스 블록의 크기와 같거나, 크거나, 작을 수 있습니다. 일반적으로 이 두 값 중 하나는 다른 값의 배수입니다.

테이블 인덱스 블록의 데이터에 액세스해야 하는 경우 서버는 먼저 키 캐시의 일부 블록 버퍼에서 사용할 수 있는지 확인합니다. 있는 경우 서버는 키 캐시에 있는 데이터에 액세스합니다.
를 사용합니다. 즉, 디스크에서 읽거나 디스크에 쓰는 대신 캐시에서 읽거나 캐시에 씁니다. 그렇지 않으면 서버는 다른 테이블 인덱스 블록(또는 블록)이 포함된 캐시 블록 버퍼를 선택하고 필요한 테이블 인덱스 블록의 복사본으로 데이터를 대체합니다. 새 인덱스 블록이 캐시에 들어오는 즉시 인덱스 데이터에 액세스할 수 있습니다.

교체하도록 선택한 블록이 수정된 경우 해당 블록은 "더티"로 간주됩니다. 이 경우 교체되기 전에 해당 블록의 내용이 해당 블록을 가져온 테이블 인덱스로 플러시됩니다.

일반적으로 서버는 *LRU(최근 사용량 최소화)* 전략을 따릅니다: 교체할 블록을 선택할 때 가장 최근에 사용된 인덱스 블록을 선택합니다. 이 선택을 더 쉽게 하기 위해 키 캐시 모듈은 사용된 모든 블록을 사용 시간별로 정렬된 특수 목록(*LRU 체인*)에 유지합니다. 블 록 에 액세스하면 가장 최근에 사용된 블록이 목록의 맨 끝에 배치됩니다. 블록이 필요한 경우
교체되면 목록 맨 앞에 있는 블록이 가장 최근에 사용되지 않은 블록이며 퇴거 대상의 첫 번째 후보가 됩니다.

InnoDB 스토리지 엔진은 버퍼 풀을 관리하기 위해 LRU 알고리즘도 사용합니다. 섹션 15.5.1, "버퍼 풀"을 참조하십시오.

8.10.2.1 공유 키 캐시 액세스

스레드는 다음 조건에 따라 키 캐시 버퍼에 동시에 액세스할 수 있습니다:

- 업데이트되지 않는 버퍼는 여러 세션에서 액세스할 수 있습니다.
- 업데이트 중인 버퍼는 버퍼를 사용해야 하는 세션이 업데이트가 완료될 때까지 대기하도록 합니다.
- 여러 세션이 서로 간섭하지 않는 한(즉, 서로 다른 인덱스 블록이 필요하여 서로 다른 캐시 블록이 교체되는 경우) 캐시 블록 교체를 초래하는 요청을 시작할 수 있습니다.

키 캐시에 대한 공유 액세스를 통해 서버의 처리량을 크게 개선할 수 있습니다.

8.10.2.2 다중 키 캐시

참고

MySQL 8.2부터 여러 MyISAM 키 캐시를 참조하기 위해 여기에 설명된 복합 부분 구

조화된 변수 구문은 더 이상 사용되지 않습니다.

키 캐시에 대한 공유 액세스는 성능을 향상시키지만 세션 간의 경합을 완전히 제거하지는 못합니다. 세션들은 여전히 키 캐시 버퍼에 대한 액세스를 관리하는 제어 구조를 두고 경쟁합니다. 키 캐시 액세스 경합을 더욱 줄이기 위해 MySQL은 다중 키 캐시도 제공합니다. 이 기능을 사용하면 서로 다른 키 캐시에 서로 다른 테이블 인덱스를 할당할 수 있습니다.

키 캐시가 여러 개 있는 경우 서버는 특정 `MyISAM` 테이블에 대한 쿼리를 처리할 때 어떤 캐시를 사용할지 알아야 합니다. 기본적으로 모든 `MyISAM` 테이블 인덱스는 기본 키 캐시에 캐시됩니다. 특정 키 캐시에 테이블 인덱스를 할당하려면 `CACHE INDEX` 문을 사용합니다(
섹션 13.7.8.2, "캐시 인덱스 문"). 예를 들어, 다음 문은 테이블 `t1`, `t2` 및 `t3`의 인덱스를 `hot_cache`라는 키 캐시에 할당합니다:

```
mysql> CACHE INDEX t1, t2, t3 IN hot_cache;
+---------+--------------------+----------+----------+
| 표 | 연산                        | 메시지 유형 | 메시지 텍스트 |
+---------+--------------------+----------+----------+
| test.t1 | assign_to_keycache | status| 확인          |
| test.t2 | assign_to_keycache | status| 확인          |
| test.t3 | assign_to_keycache | status| 확인          |
+---------+--------------------+----------+----------+
```

CACHE INDEX 문에서 참조하는 키 캐시는 SET GLOBAL 매개변수 설정 문으로 크기를 설정하거나 서버 시작 옵션을 사용하여 생성할 수 있습니다. 예를 들면 다음과 같습니다:

```
mysql> SET GLOBAL keycache1.key_buffer_size=128*1024;
```

키 캐시를 삭제하려면 크기를 0으로 설정합니다:

```
mysql> SET GLOBAL keycache1.key_buffer_size=0;
```

기본 키 캐시는 삭제할 수 없습니다. 이 시도는 모두 무시됩니다:

```
mysql> SET GLOBAL key_buffer_size = 0;

mysql> SHOW 변수에 'key_buffer_size' 같은 변수를 입력합니다;
+-----------------+ ----------+
| 변수_이름| 값      |
+-----------------+ ----------+
| 키_버퍼_크기 | 8384512 |
+-----------------+ ----------+
```

키 캐시 변수는 이름과 구성 요소가 있는 구조화된 시스템 변수입니다. keycache1.key_buffer_size의 경우 keycache1은 캐시 변수 이름이고 key_buffer_size는 캐시 구성 요소입니다. 구조화된 키 캐시 시스템 변수를 참조하는 데 사용되는 구문에 대한 설명은 섹션 5.1.9.5, "구조화된 시스템 변수"를 참조하세요.

기본적으로 테이블 인덱스는 서버 시작 시 생성된 기본(기본) 키 캐시에 할당됩니다. 키 캐시가 삭제되면 해당 키 캐시에 할당된 모든 인덱스가 기본 키 캐시에 다시 할당됩니다.

사용량이 많은 서버의 경우 세 개의 키 캐시를 사용하는 전략을 사용할 수 있습니다:

- 모든 키 캐시에 할당된 공간의 20%를 차지하는 '핫' 키 캐시입니다. 검색에 많이 사용되지만 업데이트되지 않는 테이블에 사용합니다.
- 모든 키 캐시에 할당된 공간의 20%를 차지하는 "콜드" 키 캐시입니다. 임시 테이블과 같이 중간 크기의 집중적으로 수정되는 테이블에 이 캐시를 사용합니다.
- 키 캐시 공간의 60%를 차지하는 "따뜻한" 키 캐시입니다. 다른 모든 테이블에 기본적으로 사용되는 기본 키 캐시로 사용합니다.

3개의 키 캐시를 사용하는 것이 유리한 이유 중 하나는 하나의 키 캐시 구조에 대한 액세스가 다른 키 캐시에 대한 액세스를 차단하지 않기 때문입니다. 한 캐시에 할당된 테이블에 액세스하는 문은 다른 캐시에 할당된 테이블

에 액세스하는 문과 경쟁하지 않습니다. 성능 향상은 다른 이유에서도 발생합니다:

- 핫 캐시는 검색 쿼리에만 사용되므로 그 내용은 절대 수정되지 않습니다. 따라서 인덱스 블록을 디스크에서 가져와야 할 때마다 교체용으로 선택한 캐시 블록의 내용을 먼저 플러시할 필요가 없습니다.
- 핫 캐시에 할당된 인덱스의 경우, 인덱스 스캔이 필요한 쿼리가 없는 경우 인덱스 B-트리의 비리프 노드에 해당하는 인덱스 블록이 캐시에 남아 있을 확률이 높습니다.
- 임시 테이블에 대해 가장 자주 실행되는 업데이트 작업은 업데이트된 노드가 캐시에 있고 디스크에서 먼저 읽을 필요가 없는 경우 훨씬 빠르게 수행됩니다. 임시 테이블의 인덱스 크기가 콜드 키 캐시의 크기와 비슷하다면 업데이트된 노드가 캐시에 있을 확률이 매우 높습니다.

CACHE INDEX 문은 테이블과 키 캐시 간의 연결을 설정하지만 서버가 다시 시작될 때마다 연결이 손실됩니다. 서버가 시작될 때마다 연결이 적용되도록 하려면 옵션 파일을 사용하는 것이 한 가지 방법입니다: 키 캐시를 구성하는 변수 설정과 실행할 CACHE INDEX 문이 포함된 파일의 이름을 지정하는 init_file 시스템 변수를 포함합니다. 예를 들어

```
key_buffer_size = 4G
hot_cache.key_buffer_size = 2G
cold_cache.key_buffer_size = 2G
init_file=/path/to/data-directory/mysqld_init.sql
```

mysqld_init.sql의 문은 서버가 시작될 때마다 실행됩니다. 파일에는 한 줄당 하나의 SQL 문이 포함되어야 합니다. 다음 예제에서는 각각 여러 테이블을 hot_cache와 cold_cache에 할당합니다:

```
CACHE INDEX db1.t1, db1.t2, db2.t3 IN hot_cache
CACHE INDEX db1.t4, db2.t5, db2.t6 IN cold_cache
```

8.10.2.3 중간 지점 삽입 전략

기본적으로 키 캐시 관리 시스템은 퇴거할 키 캐시 블록을 선택하기 위해 간단한 LRU 전략을 사용하지만, *중간 지점 삽입 전략이라는* 보다 정교한 방법도 지원합니다.

중간점 삽입 전략을 사용할 때, LRU 체인은 핫 서브리스트와 웜 서브리스트의 두 부분으로 나뉩니다.
두 부분 사이의 분할 지점은 고정되어 있지 않지만, 키 캐시 관리 시스템은 웜 부분이 "너무 짧지 않도록
" 항상 최소한의
키 캐시 블록의 퍼센트 키 캐시 블록은 구조화된 키 캐시 변수의 구성 요소이므로 해당 값은 캐시별로 설정할 수 있는 매개변수입니다.

인덱스 블록이 테이블에서 키 캐시로 읽혀지면 웜 하위 목록의 끝에 배치됩니다. 일정 횟수의 히트(블록에 대한 액세스)가 발생하면 핫 하위 목록으로 승격됩니다. 현재 블록을 승격하는 데 필요한 히트 횟수(3)는 모든 인덱스 블록에 대해 동일합니다.

인기 하위 목록으로 승격된 블록은 목록 끝에 배치됩니다. 그런 다음 블록은 이 하위 목록 내에서 순환합니다. 블록이 하위 목록의 시작 부분에 충분히 오랫동안 머무르면 따뜻한 하위 목록으로 강등됩니다. 이 시간은 키 캐시의 key_cache_age_threshold 구성 요소의 값에 의해 결정됩니다.

임계값은 *N개의* 블록을 포함하는 키 캐시의 경우, 마지막 *N* * key_cache_age_threshold / 100 히트 이내에 액세스하지 않은 핫 하위 목록의 시작 부분에 있는 블록을 따뜻한 하위 목록의 시작 부분으로 이동하도록 규정하고 있습니다. 그러면 교체할 블록은 항상 따뜻한 하위 목록의 시작 부분에서 가져오기 때문에 첫 번째 퇴거 후보가 됩니다.

중간점 삽입 전략을 사용하면 더 가치가 높은 블록을 항상 캐시에 보관할 수 있습니다. 일반 LRU 전략을 사용하려면 key_cache_division_limit 값을 기본값인 100으로 설정하세요.

중간 지점 삽입 전략은 인덱스 스캔이 필요한 쿼리를 실행할 때 중요한 상위 레벨 B-트리 노드에 해당하는 모든 인덱스 블록을 캐시 밖으로 효과적으로 밀어낼 때 성능을 개선하는 데 도움이 됩니다. 이를 방지하려면

key_cache_division_limit을 100보다 훨씬 작게 설정한 중간 지점 삽입 전략을 사용해야 합니다. 그러면 인덱스 스캔 작업 중에도 자주 조회되는 중요한 노드가 핫 하위 목록에 보존됩니다.

8.10.2.4 인덱스 사전 로드

키 캐시에 전체 인덱스의 블록 또는 적어도 비리프 노드에 해당하는 블록을 저장하기에 충분한 블록이 있는 경우, 사용을 시작하기 전에 인덱스 블록으로 키 캐시에 미리 로드하는 것이 좋습니다. 사전 로딩을 사용하면 디스크에서 인덱스 블록을 순차적으로 읽는 가장 효율적인 방법으로 테이블 인덱스 블록을 키 캐시 버퍼에 넣을 수 있습니다.

사전 로드하지 않아도 쿼리에서 필요에 따라 블록이 키 캐시에 계속 배치됩니다. 블록이 캐시에 남아 있더라도 모든 블록에 대한 버퍼가 충분하기 때문에 디스크에서 순차적으로 가져오지 않고 임의의 순서로 가져옵니다.

인덱스를 캐시에 미리 로드하려면 `LOAD INDEX INTO CACHE` 문을 사용합니다. 예를 들어, 다음 문은 테이블 `t1` 및 `t2`의 인덱스 노드(인덱스 블록)를 미리 로드합니다:

```
mysql> LOAD INDEX INTO CACHE t1, t2 IGNORE LEAVES;
+---------+--------------+----------+ --------------+
| 표 | 연산                 | 메시지 유형 | 메시지 텍스트 |
+---------+--------------+----------+ --------------+
| test.t1 | preload_keys | status| 확인          |
| test.t2 | preload_keys | status| 확인          |
+---------+--------------+----------+ --------------+
```

`IGNORE LEAVES` 수정자는 인덱스의 비리프 노드에 대한 블록만 미리 로드하도록 합니다. 따라서 표시된 문은 `t1`의 모든 인덱스 블록을 미리 로드하지만 `t2`의 비리프 노드에 대한 블록만 미리 로드합니다.

인덱스가 `CACHE INDEX` 문을 사용하여 키 캐시에 할당된 경우 미리 로드하면 인덱스 블록이 해당 캐시에 배치됩니다. 그렇지 않으면 인덱스가 기본 키 캐시에 로드됩니다.

8.10.2.5 키 캐시 블록 크기

`key_cache_block_size` 변수를 사용하여 개별 키 캐시에 대한 블록 버퍼의 크기를 지정할 수 있습니다. 이를 통해 인덱스 파일에 대한 I/O 작업의 성능을 조정할 수 있습니다.

읽기 버퍼의 크기가 기본 운영 체제 I/O 버퍼의 크기와 같을 때 I/O 작업에서 최상의 성능을 얻을 수 있습니다. 그러나 키 노드의 크기를 I/O 버퍼의 크기와 동일하게 설정한다고 해서 항상 최상의 전체 성능이 보장되는 것은 아닙니다. 큰 리프 노드를 읽을 때 서버는 불필요한 데이터를 많이 가져와서 다른 리프 노드의 읽기를 효과적으로 방해합니다.

`MyISAM` 테이블의 `.MYI` 인덱스 파일에서 블록 크기를 제어하려면 `--myisam-block-size`를 사용합니다. 옵션을 선택합니다.

8.10.2.6 키 캐시 재구성

키 캐시는 언제든지 매개변수 값을 업데이트하여 재구성할 수 있습니다. 예를 들어

```
mysql> SET GLOBAL cold_cache.key_buffer_size=4*1024*1024;
```

키 캐시 컴포넌트의 현재 값과 다른 값을 키 버퍼 크기 또는 키캐시블록 크기에 할당하면 서버는 캐시의 이전 구조를 파괴하고 새 값에 따라 새 구조를 만듭니다. 캐시에 더티 블록이 포함되어 있는 경우 서버는 캐시를 파괴하고 다시 생성하기 전에 해당 블록을 디스크에 저장합니다. 다른 주요 캐시 매개변수를 변경하면 구조 재구성이 수행되지 않습니다.

키 캐시를 재구성할 때 서버는 먼저 더티 버퍼의 콘텐츠를 디스크로 플러시합니다. 그 후에는 캐시 콘텐츠를 사용할 수 없게 됩니다. 그러나 재구성을 해도 캐시에 할당된 인덱스를 사용해야 하는 쿼리는 차단되지 않습니다.

대신 서버는 기본 파일 시스템 캐싱을 사용하여 테이블 인덱스에 직접 액세스합니다. 파일 시스템 캐싱은 키 캐시를 사용하는 것만큼 효율적이지 않으므로 쿼리가 실행되더라도 속도 저하가 예상될 수 있습니다. 캐시가 재구성되면 캐시에 할당된 인덱스를 캐싱하는 데 다시 사용할 수 있게 되고 인덱스에 대한 파일 시스템 캐싱 사용이 중단됩니다.

8.10.3 준비된 명령문 및 저장된 프로그램 캐싱

클라이언트가 세션 중에 여러 번 실행할 수 있는 특정 문에 대해 서버는 해당 문을 내부 구조로 변환하고 실행 중에 사용할 수 있도록 해당 구조를 캐시합니다. 캐싱

를 사용하면 세션 중에 다시 필요한 경우 문을 다시 변환하는 오버헤드를 피할 수 있으므로 서버가 더 효율적으로 작동할 수 있습니다. 이러한 문에 대한 변환 및 캐싱이 발생합니다:

- 준비된 문은 SQL 수준에서 처리되는 문(`PREPARE` 문 사용)과 바이너리 클라이언트/서버 프로토콜을 사용하여 처리되는 문(`mysql_stmt_prepare()` C API 함수 사용) 모두에 해당합니다.
  `max_prepared_stmt_count` 시스템 변수는 총 개수를 제어합니다.
  문을 캐시합니다. (모든 세션에 걸쳐 준비된 문 수의 합계입니다.)
- 저장된 프로그램(저장 프로시저 및 함수, 트리거, 이벤트). 이 경우 서버는 전체 프로그램 본문을 변환하여 캐시합니다. `stored_program_cache` 시스템 변수는 서버가 세션당 캐시하는 대략적인 저장 프로그램 수를 나타냅니다.

서버는 준비된 문과 저장된 프로그램에 대한 캐시를 세션별로 유지 관리합니다. 한 세션에 대해 캐시된 문은 다른 세션에서 액세스할 수 없습니다. 세션이 종료되면 서버는 해당 세션에 대해 캐시된 모든 문을 삭제합니다.

서버가 캐시된 내부 문 구조를 사용하는 경우 구조가 오래되지 않도록 주의해야 합니다. 문에서 사용되는 개체에 대한 메타데이터 변경이 발생하여 현재 개체 정의와 내부 문 구조에 표시된 정의가 불일치할 수 있습니다. 메타데이터 변경은 create, drop과 같은 DDL 문에 대해 발생합니다,
테이블을 변경, 이름 바꾸기 또는 잘라내거나 테이블을 분석, 최적화 또는 복구하는 문입니다. 테이블 콘텐츠 변경(예: `INSERT` 또는 `UPDATE)은` 메타데이터를 변경하지 않으며 `SELECT` 문도 변경하지 않습니다.

다음은 문제를 설명하는 예시입니다. 클라이언트가 이 문을 준비한다고 가정해 보겠습니다:

```
'SELECT * FROM t1'에서 s1을 준비합니다;
```

`SELECT *는` 내부 구조에서 테이블의 열 목록으로 확장됩니다. 테이블의 열 집합이 `ALTER TABLE로` 수정되면 준비된 문이 최신 상태가 되지 않습니다. 클라이언트가 다음에 `s1을` 실행할 때 서버가 이 변경 사항을 감지하지 못하면 준비된 문이 잘못된 결과를 반환합니다.

준비된 문에서 참조하는 테이블 또는 뷰의 메타데이터 변경으로 인해 발생하는 문제를 방지하기 위해 서버는 이러한 변경 사항을 감지하고 다음에 문이 실행될 때 자동으로 다시 준비합니다. 즉, 서버가 문을 다시 작성하고 내부 구조를 다시 빌드합니다. 참조된 테이블 또는 뷰가 테이블 정의 캐시에서 플러시된 후에도 재구문은 캐시에 새 항목을 위한 공간을 확보하기 위해 암시적으로 또는 명시적으로 테이블을 플러시한 후에 발생합니다.

마찬가지로 저장된 프로그램에서 사용하는 객체에 변경 사항이 발생하면 서버는 프로그램 내에서 영향을 받는 문을 다시 작성합니다.

서버는 표현식 내 객체의 메타데이터 변경도 감지합니다. 이러한 메타데이터는 저장된 프로그램과 관련된 문(예: `DECLARE CURSOR`)이나 흐름 제어 문(예: `IF`, `CASE`, `RETURN)`에 사용될 수 있습니다.

저장된 프로그램 전체를 복구하지 않기 위해 서버는 필요한 경우에만 프로그램 내의 영향을 받는 문이나 표현식을 복구합니다. 예시:

- 테이블 또는 뷰의 메타데이터가 변경되었다고 가정합니다. 테이블 또는 뷰에 액세스하는 프로그램 내의 `SELECT *`에 대해서는 재구문이 수행되지만 테이블 또는 뷰에 액세스하지 않는 `SELECT *`에 대해서는 재구문이 수행되지 않습니다.

- 문이 영향을 받으면 서버는 가능한 경우 부분적으로만 복구합니다. 이 경우를 고려해 보세요.
  문을 사용합니다:

```
CASE case_expr
  WHEN when_expr1 ...
  WHEN when_expr2 ...
  WHEN when_expr3 ...
  ...
사례 종료
```

메타데이터 변경이 `WHEN` *때_expr3에만* 영향을 미치는 경우 해당 표현식은 구문 분석됩니다. *`case_expr`* 및 다른 `WHEN` 표현식은 구문 분석되지 않습니다.

재구문은 원래 내부 형식으로 변환할 때 적용되었던 기본 데이터베이스 및 SQL 모드를 사용합니다.

서버는 최대 세 번까지 복구를 시도합니다. 모든 시도가 실패하면 오류가 발생합니다.

복구는 자동으로 수행되지만, 그 정도에 따라 준비된 문과 저장된 프로그램 성능이 저하됩니다.

준비된 문의 경우 `Com_stmt_reprepare` 상태 변수가 준비 횟수를 추적합니다.

8.11 잠금 작업 최적화

MySQL은 잠금을 사용하여 테이블 콘텐츠에 대한 경합을 관리합니다:

- 내부 잠금은 여러 스레드에서 테이블 콘텐츠에 대한 경합을 관리하기 위해 MySQL 서버 자체 내에서 수행됩니다. 이 유형의 잠금은 전적으로 서버에 의해 수행되고 다른 프로그램과 관련이 없기 때문에 내부 잠금입니다. 섹션 8.11.1, "내부 잠금 방법"을 참조하세요.
- 외부 잠금은 서버와 다른 프로그램이 `MyISAM` 테이블 파일을 잠가서 어떤 프로그램이 어느 시점에 테이블에 액세스할 수 있는지 서로 조정할 때 발생합니다. 섹션 8.11.5, "외부 잠금"을 참조하십시오.

8.11.1 내부 잠금 방법

이 섹션에서는 내부 잠금, 즉 여러 세션의 테이블 콘텐츠 경합을 관리하기 위해 MySQL 서버 자체 내에서 수행되는 잠금에 대해 설명합니다. 이러한 유형의 잠금은 전적으로 서버에 의해 수행되고 다른 프로그램과 관련이 없기 때문에 내부 잠금입니다. 다른 프로그램에 의해 MySQL 파일에서 수행되는 잠금에 대해서는 섹션 8.11.5, "외부 잠금"을 참조하십시오.

- 행 수준 잠금
- 테이블 수준 잠금
- 잠금 유형 선택하기

행 수준 잠금

MySQL은 `InnoDB` 테이블에 행 수준 잠금을 사용하여 여러 세션의 동시 쓰기 액세스를 지원하므로 다중 사용자, 고도의 동시성 및 OLTP 애플리케이션에 적합합니다.

단일 `InnoDB` 테이블에서 여러 개의 동시 쓰기 작업을 수행할 때 교착 상태를 방지하려면 데이터 변경 문이 트랜잭션 후반에 발생하더라도 수정될 것으로 예상되는 각 행 그룹에 대해 `SELECT ... FOR UPDATE` 문을 실행하여 트랜잭션 시작 시 필요한 잠금을 획득합니다. 트랜잭션이 둘 이상의 테이블을 수정하거나 잠그는 경우 각 트랜잭션 내에서 동일한 순서로 해당 문을 실행합니다. 교착 상태는 심각한 오류를 나타내는 것

이 아니라 성능에 영향을 미치지만, InnoDB는 기본적으로 교착 상태를 자동으로 감지하고 영향을 받는 트랜잭션 중 하나를 롤백하기 때문입니다.

동시성이 높은 시스템에서 교착 상태 감지는 수많은 스레드가 동일한 잠금을 기다릴 때 속도 저하를 유발할 수 있습니다. 때로는 교착 상태 감지를 비활성화하고 교착 상태 발생 시 트랜잭션 롤백을 위해 innodb_lock_wait_timeout 설정에 의존하는 것이 더 효율적일 수 있습니다. 교착 상태 감지는 innodb_deadlock_detect 구성 옵션을 사용하여 비활성화할 수 있습니다.

행 수준 잠금의 장점:

- 서로 다른 세션이 서로 다른 행에 액세스할 때 잠금 충돌이 줄어듭니다.

- 롤백 시 변경 사항이 줄어듭니다.
- 한 행을 장시간 잠글 수 있습니다.

테이블 수준 잠금

MySQL은 `MyISAM`, `MEMORY` 및 `MERGE` 테이블에 테이블 수준 잠금을 사용하여 한 번에 하나의 세션만 해당 테이블을 업데이트할 수 있도록 허용합니다. 이 잠금 수준은 이러한 스토리지 엔진을 읽기 전용, 읽기 중심 또는 단일 사용자 애플리케이션에 더 적합하게 만듭니다.

이러한 저장소 엔진은 쿼리 시작 시 항상 필요한 모든 잠금을 한 번에 요청하고 항상 동일한 순서로 테이블을 잠금으로써 교착 상태를 방지합니다. 단, 이 전략은 동시성을 감소시키므로 테이블을 수정하려는 다른 세션은 현재 데이터 변경 문이 완료될 때까지 기다려야 한다는 단점이 있습니다.

테이블 수준 잠금의 장점:

- 상대적으로 적은 메모리 필요(행 잠금을 사용하려면 잠긴 행 또는 행 그룹당 메모리 필요)
- 하나의 잠금 장치만 사용하므로 테이블의 넓은 부분에 사용할 때 빠릅니다.
- 많은 양의 데이터에 대해 GROUP `BY` 작업을 자주 수행하거나 전체 테이블을 자주 스캔해야 하는 경우 빠릅니다.

MySQL은 다음과 같이 테이블 쓰기 잠금을 부여합니다:

1. 테이블에 잠금 장치가 없는 경우 쓰기 잠금을 설정합니다.
2. 그렇지 않으면 잠금 요청을 쓰기 잠금 대기열에 넣습니다.

MySQL은 다음과 같이 테이블 읽기 잠금을 부여합니다:

1. 테이블에 쓰기 잠금이 없는 경우 읽기 잠금을 설정합니다.
2. 그렇지 않으면 잠금 요청을 읽기 잠금 대기열에 넣습니다.

테이블 업데이트는 테이블 검색보다 우선순위가 높습니다. 따라서 잠금이 해제되면 쓰기 잠금 대기열의 요청에 잠금을 사용할 수 있게 된 다음 읽기 잠금 대기열의 요청에 잠금을 사용할 수 있게 됩니다. 이렇게 하면 테이블에 대한 `SELECT` 활동이 많은 경우에도 테이블에 대한 업데이트가 "고갈"되지 않습니다. 그러나 테이블에 대한 업데이트가 많은 경우 `SELECT` 문은 더 이상 업데이트가 없을 때까지 대기합니다.

읽기 및 쓰기 우선순위 변경에 대한 자세한 내용은 섹션 8.11.2, "테이블 잠금 문제"를 참조하세요. 시스템의 테이블 잠금 경합을 분석하려면 `Table_locks_immediate`를 확인하면 됩니다.
및 `Table_locks_waited` 상태 변수에 대한 요청 횟수를 나타내는
테이블 잠금을 즉시 부여할 수 있는 경우와 기다려야 하는 경우로 각각 나뉩니다:

```
mysql> SHOW STATUS LIKE 'Table%';
+-----------------------+-----------+
|                       변수_이름| 값   |
+-----------------------+-----------+
| Table_locks_immediate | 1151552 |
|                       Table_locks_waited| 15324    |
+-----------------------+-----------+
```

성능 스키마 잠금 테이블은 잠금 정보도 제공합니다. 섹션 27.12.13, "성능 스키마 잠금 테이블"을 참조하십시오.

`MyISAM` 스토리지 엔진은 동시 삽입을 지원하여 특정 테이블에 대한 읽기와 쓰기 간의 경합을 줄입니다: `MyISAM` 테이블에 데이터 파일 중간에 빈 블록이 없는 경우, 행은 항상 데이터 파일 끝에 삽입됩니다. 이 경우 동시 `INSERT` 및

SELECT 문을 잠금 없이 사용할 수 있습니다. 즉, 다른 클라이언트가 MyISAM 테이블에서 읽는 동시에 행을 삽입할 수 있습니다. 테이블 중간에 행이 삭제되거나 업데이트되어 구멍이 발생할 수 있습니다. 구멍이 있는 경우 동시 삽입은 비활성화되지만 모든 구멍이 새 데이터로 채워지면 자동으로 다시 활성화됩니다. 이 동작을 제어하려면 concurrent_insert 시스템 변수를 사용합니다. 섹션 8.11.3, "동시 삽입"을 참조하십시오.

테이블 잠금으로 명시적으로 테이블 잠금을 획득한 경우 테이블이 잠겨 있는 동안 다른 세션이 동시 삽입을 수행할 수 있도록 읽기 잠금이아닌 읽기 로컬 잠금을 요청할 수 있습니다.

동시 삽입이 불가능한 경우 테이블 t1에서 많은 INSERT 및 SELECT 작업을 수행하려면 임시 테이블 temp_t1에 행을 삽입하고 임시 테이블의 행으로 실제 테이블을 업데이트할 수 있습니다:

```
mysql> LOCK TABLES t1 WRITE, temp_t1 WRITE;
mysql> INSERT INTO t1 SELECT * FROM temp_t1;
mysql> DELETE FROM temp_t1;
mysql> 테이블 잠금 해제;
```

잠금 유형 선택하기

일반적으로 다음과 같은 경우 테이블 잠금 기능이 행 수준 잠금 기능보다 우수합니다:

- 테이블에 대한 대부분의 문은 읽습니다.
- 테이블에 대한 문은 읽기와 쓰기가 혼합되어 있으며, 여기서 쓰기는 하나의 키 읽기로 가져올 수 있는 단일 행에 대한 업데이트 또는 삭제입니다:

  ```
  UPDATE tbl_name SET column=value WHERE unique_key_col=key_value;
  DELETE FROM tbl_name WHERE unique_key_col=key_value;
  ```

- SELECT는 동시 INSERT 문과 결합되며, UPDATE 또는 DELETE는 거의 없습니다.
  문을 사용합니다.
- 작성자가 없는 전체 테이블에 대한 많은 스캔 또는 그룹별 작업.

상위 수준 잠금을 사용하면 행 수준 잠금에 비해 잠금 오버헤드가 적기 때문에 다양한 유형의 잠금을 지원하여 애플리케이션을 더 쉽게 조정할 수 있습니다.

행 수준 잠금 이외의 옵션:

- 한 명의 작성자가 동시에 여러 명의 독자를 가질 수 있는 버전 관리(예: 동시 삽입을 위해 MySQL에서 사용되는 버전 관리). 즉, 데이터베이스 또는 테이블이 액세스가 시작되는 시점에 따라 데이터에 대해 서로 다른 보기를 지원합니다. 이에 대한 다른 일반적인 용어로는 "시간 여행", "쓰기 시 복사" 또는 "주문형 복사"가 있습니다.
- 주문형 복사는 많은 경우 행 수준 잠금보다 우수합니다. 하지만 최악의 경우 일반 잠금을 사용하는 것보다 훨씬 더 많은 메모리를 사용할 수 있습니다.
- 행 수준 잠금을 사용하는 대신 MySQL의 GET_LOCK() 및 RELEASE_LOCK()에서 제공하는 것과 같은 애플리케이션 수준 잠금을 사용할 수 있습니다. 이러한 잠금은 권고 잠금이므로 서로 협력하는 애플리케이

션에서만 작동합니다. 섹션 12.14, "잠금 함수"를 참조하십시오.

8.11.2 테이블 잠금 문제

`InnoDB` 테이블은 행 수준 잠금을 사용하므로 여러 세션과 애플리케이션이 서로 기다리게 하거나 일관성 없는 결과를 생성하지 않고 동일한 테이블에서 동시에 읽고 쓸 수 있습니다.

이 저장소 엔진의 경우, 추가 보호 기능을 제공하지 않고 대신 동시성을 감소시키므로 `LOCK TABLES` 문을 사용하지 마세요. 자동 행 수준 잠금 기능은 가장 중요한 데이터가 있는 가장 바쁜 데이터베이스에 적합하며, 테이블을 잠그고 잠금 해제할 필요가 없으므로 애플리케이션 로직도 간소화됩니다. 따라서 `InnoDB` 스토리지 엔진은 MySQL의 기본값입니다.

MySQL은 InnoDB를 제외한 모든 스토리지 엔진에 대해 페이지, 행 또는 열 잠금 대신 테이블 잠금을 사용합니다. 잠금 작업 자체는 오버헤드가 크지 않습니다. 그러나 한 번에 하나의 세션만 테이블에 쓸 수 있으므로 이러한 다른 스토리지 엔진에서 최상의 성능을 얻으려면 주로 자주 쿼리되고 삽입 또는 업데이트가 거의 이루어지지 않는 테이블에 사용하세요.

- InnoDB를 선호하는 성능 고려 사항
- 잠금 성능 문제에 대한 해결 방법

InnoDB를 선호하는 성능 고려 사항

InnoDB를 사용하여 테이블을 만들지 다른 스토리지 엔진을 사용하여 테이블을 만들지 선택할 때는 테이블 잠금의 다음과 같은 단점을 염두에 두어야 합니다:

- 테이블 잠금을 사용하면 여러 세션이 동시에 테이블에서 읽을 수 있지만, 한 세션이 테이블에 쓰려면 먼저 독점 액세스 권한을 얻어야 하므로 다른 세션이 먼저 테이블 작업을 완료할 때까지 기다려야 할 수 있습니다. 업데이트가 진행되는 동안 이 특정 테이블에 액세스하려는 다른 모든 세션은 업데이트가 완료될 때까지 기다려야 합니다.
- 디스크가 꽉 차서 세션을 계속 진행하려면 여유 공간을 확보해야 하므로 세션이 대기 중일 때 테이블 잠금이 문제를 일으킵니다. 이 경우 문제가 있는 테이블에 액세스하려는 모든 세션도 디스크 공간이 더 확보될 때까지 대기 상태에 놓이게 됩니다.
- 실행하는 데 시간이 오래 걸리는 SELECT 문은 그 동안 다른 세션이 테이블을 업데이트하지 못하게 하여 다른 세션이 느리거나 응답하지 않는 것처럼 보이게 합니다. 한 세션이 업데이트를 위해 테이블에 대한 독점 액세스 권한을 얻으려고 대기하는 동안 SELECT 문을 실행하는 다른 세션은 그 뒤에 대기열에 대기하므로 읽기 전용 세션의 경우에도 동시성이 감소합니다.

성능 잠금 문제에 대한 해결 방법

다음 항목에서는 테이블 잠금으로 인한 경합을 피하거나 줄일 수 있는 몇 가지 방법을 설명합니다:

- 테이블을 InnoDB 스토리지 엔진으로 전환하려면 설정 중에 CREATE TABLE ... ENGINE=INNODB를 사용하거나 기존 테이블에 대해 ALTER TABLE ... ENGINE=INNODB를 사용합니다. 이 스토리지 엔진에 대한 자세한 내용은 15장, *InnoDB 스토리지 엔진을* 참조하십시오.
- SELECT 문이 더 빠르게 실행되도록 최적화하여 테이블을 더 짧은 시간 동안 잠그도록 합니다. 이를 위해 몇 가지 요약 테이블을 만들어야 할 수도 있습니다.
- 낮은 우선순위 업데이트로 mysqld를 시작합니다. 테이블 수준 잠금만 사용하는 스토리지 엔진(예: MyISAM, MEMORY 및 MERGE)의 경우 테이블을 업데이트(수정)하는 모든 문에 SELECT 문보다 낮은 우선순위를 부여합니다. 이 경우 앞의 시나리오에서 두 번째 SELECT 문은UPDATE 문보다 먼저 실행되며 첫 번째 SELECT가 완료될 때까지 기다리지 않습니다.

- 특정 연결에서 발행된 모든 업데이트가 낮은 우선순위로 수행되도록 지정하려면 `low_priority_updates` 서버 시스템 변수를 1로 설정합니다.
- 특정 `INSERT`, `UPDATE` 또는 `DELETE` 문에 낮은 우선순위를 부여하려면 `LOW_PRIORITY`를 사용합니다. 속성입니다.
- 특정 `SELECT` 문에 더 높은 우선순위를 부여하려면 `HIGH_PRIORITY` 속성을 사용합니다. 섹션 13.2.13, "SELECT 문"을 참조하십시오.
- `max_write_lock_count` 시스템 변수의 값을 낮게 설정하여 MySQL을 시작하면 테이블에 대한 특정 수의 쓰기 잠금이 발생한 후(예: 삽입 작업) 테이블을 대기 중인 모든 `SELECT` 문의 우선순위가 일시적으로 상승합니다. 이렇게 하면 특정 수의 쓰기 잠금이 발생한 후에 읽기 잠금이 허용됩니다.
- `SELECT` 문과 `DELETE` 문이 혼합되어 문제가 있는 경우 `DELETE`에 `LIMIT` 옵션을 사용하면 도움이 될 수 있습니다. 섹션 13.2.2, "DELETE 문"을 참조하십시오.

- SELECT 문과 함께 SQL_BUFFER_RESULT를 사용하면 테이블 잠금 기간을 단축하는 데 도움이 될 수 있습니다. 섹션 13.2.13, "SELECT 문"을 참조하십시오.
- 테이블 내용을 별도의 테이블로 분할하면 한 테이블의 열에 대해 쿼리를 실행하고 업데이트는 다른 테이블의 열에만 한정하여 쿼리를 실행할 수 있으므로 도움이 될 수 있습니다.
- 단일 큐를 사용하도록 mysys/thr_lock.c의 잠금 코드를 변경할 수 있습니다. 이 경우 쓰기 잠금과 읽기 잠금의 우선 순위가 동일해져 일부 애플리케이션에 도움이 될 수 있습니다.

8.11.3 동시 삽입

MyISAM 스토리지 엔진은 특정 테이블에 대한 읽기 및 쓰기 간의 경합을 줄이기 위해 동시 삽입을 지원합니다: MyISAM 테이블에 데이터 파일에 구멍(중간에 삭제된 행)이 없는 경우, SELECT 문이 테이블에서 행을 읽는 것과 동시에 INSERT 문을 실행하여 테이블 끝에 행을 추가할 수 있습니다. INSERT 문이 여러 개 있는 경우 대기열에 추가되고 SELECT 문과 동시에 순서대로 수행됩니다. 동시 INSERT의 결과는 즉시 표시되지 않을 수 있습니다.

concurrent_insert 시스템 변수를 설정하여 동시 삽입 처리를 수정할 수 있습니다. 기본적으로 이 변수는 자동 (또는 1)으로 설정되어 있으며 동시 삽입은 방금 설명한 대로 처리됩니다. concurrent_insert가 NEVER(또는 0)로 설정되어 있으면 동시 삽입이 비활성화됩니다. 변수가 다음과 같이 설정된 경우 항상(또는 2)로 설정하면 행이 삭제된 테이블에서도 테이블 끝에 동시 삽입이 허용됩니다. concurrent_insert 시스템 변수에 대한 설명도 참조하십시오.

바이너리 로그를 사용하는 경우 동시 삽입은 CREATE ... 또는 INSERT ... 문에 대한 일반 삽입으로 변환됩니다. SELECT 또는 INSERT ... SELECT 문에 대해 일반 삽입으로 변환됩니다. 이는 백업 작업 중에 로그를 적용하여 테이블의 정확한 복사본을 다시 생성할 수 있도록 하기 위해 수행됩니다. 섹션 5.4.4, "바이너리 로그"를 참조하십시오. 또한 이러한 문에 대해 선택한 테이블에 읽기 잠금이 설정되어 해당 테이블에 대한 삽입이 차단됩니다. 그 결과 해당 테이블에 대한 동시 삽입도 대기해야 합니다.

데이터 로드에서 동시 삽입 조건을 충족하는(즉, 중간에 빈 블록이 없는) MyISAM 테이블에 CONCURRENT를 지정하면 다른 세션이 데이터 로드 실행 중에 테이블에서 데이터를 검색할 수 있습니다. 다른 세션이 동시에 테이블을 사용하고 있지 않더라도 CONCURRENT 옵션을 사용하면 데이터 로드 성능에 약간의 영향을 미칩니다.

높은 우선순위를 지정하면 서버가 해당 옵션으로 시작된 경우 --low-priority-updates 옵션의 효과를 재정의합니다. 또한 동시 삽입이 사용되지 않도록 합니다.

LOCK TABLE의 경우, READ LOCAL과 READ의 차이점은 READ LOCAL은 잠금이 유지되는 동안 충돌하지 않는 INSERT 문(동시 삽입)을 실행할 수있다는 것입니다. 그러나 잠금이 유지되는 동안 서버 외부의 프로세스를 사용하여 데이터베이스를 조작하려는 경우에는 이 기능을 사용할 수 없습니다.

8.11.4 메타데이터 잠금

MySQL은 메타데이터 잠금을 사용하여 데이터베이스 개체에 대한 동시 액세스를 관리하고 데이터 일관성을 보장합니다. 메타데이터 잠금은 테이블뿐만 아니라 스키마, 저장 프로그램(프로시저, 함수, 트리거, 예약된 이벤트), 테이블 스페이스, `GET_LOCK()` 함수로 획득한 사용자 잠금(12.14절. "잠금 함수" 참조), 5.6.9.1절. "잠금 서비스"에 설명된 잠금 서비스를 통해 획득한 잠금에도 적용됩니다.

성능 스키마 메타데이터_잠금 테이블은 메타데이터 잠금 정보를 노출하므로 어떤 세션이 잠금을 보유하고 있는지, 잠금을 기다리다가 차단되었는지 등을 확인하는 데 유용할 수 있습니다. 자세한 내용은 섹션 27.12.13.3, "메타데이터_잠금 테이블"을 참조하세요.

메타데이터 잠금은 약간의 오버헤드를 수반하며, 이는 쿼리 볼륨이 증가함에 따라 증가합니다. 메타데이터 경합은 여러 쿼리가 동일한 개체에 액세스하려고 시도할수록 증가합니다.

메타데이터 잠금은 테이블 정의 캐시를 대체하는 것이 아니며, 해당 뮤텍스와 잠금은 `LOCK_open` 뮤텍스와 다릅니다. 다음 설명에서는 메타데이터 잠금이 어떻게 작동하는지에 대한 몇 가지 정보를 제공합니다.

- 메타데이터 잠금 획득
- 메타데이터 잠금 해제

메타데이터 잠금 획득

특정 잠금에 대한 대기자가 여러 명 있는 경우, `max_write_lock_count` 시스템 변수와 관련된 예외를 제외하고 우선순위가 가장 높은 잠금 요청이 먼저 충족됩니다. 쓰기 잠금 요청은 읽기 잠금 요청보다 우선순위가 높습니다. 그러나 max_write_lock_count가 낮은 값(예: 10)으로 설정된 경우, 읽기 잠금 요청이 이미 10개의 쓰기 잠금 요청에 우선하여 전달된 경우 읽기 잠금 요청이 보류 중인 쓰기 잠금 요청보다 우선할 수 있습니다. 기본적으로 `max_write_lock_count`는 매우 큰 값을 갖기 때문에 일반적으로 이러한 동작은 발생하지 않습니다.

문은 메타데이터 잠금을 동시에 획득하는 것이 아니라 하나씩 획득하고 그 과정에서 교착 상태 감지를 수행합니다.

DML 문은 일반적으로 문에 테이블이 언급된 순서대로 잠금을 획득합니다.

DDL 문, `LOCK TABLES` 및 기타 유사한 문은 명시적으로 명명된 테이블에 대해 이름 순서대로 잠금을 획득하여 동시 DDL 문 간에 발생할 수 있는 교착 상태의 수를 줄이려고 합니다. 암시적으로 사용되는 테이블(예: 잠겨 있어야 하는 외래 키 관계의 테이블)의 경우 잠금이 다른 순서로 획득될 수 있습니다.

예를 들어 `RENAME TABLE`은 이름 순서대로 잠금을 가져오는 DDL 문입니다:

- 이 `RENAME TABLE` 문은 `tbla`의 이름을 다른 이름으로 바꾸고 `tblc`의 이름을 `tbla`로 바꿉니다:

  ```
  테이블 이름을 tbla에서 tbld로, tblc에서 tbla로 변경합니다;
  ```

  이 문은 `tbla`, `tblc` 및 `tbld`에 대해 순서대로 메타데이터 잠금을 획득합니다(`tbld`는 다음과 같으므로). `tblc` 순서로):

- 이 약간 다른 명령문은 `tbla`의 이름을 다른 이름으로 바꾸고 `tblc`의 이름도 `tbla`로 바꿉니다:

  ```
  테이블 이름 변경 tbla에서 tblb로, tblc에서 tbla로;
  ```

  이 경우 문은 순서대로 `tbla`, `tblb` 및 `tblc`에서 메타데이터 잠금을 획득합니다(왜냐하면 `tblb`는 이름 순서대로 `tblc` 앞에옵니다):

두 문 모두 `tbla` 및 `tblc`에 대한 잠금을 순서대로 획득하지만 나머지 테이블 이름에 대한 잠금이 `tblc` 이전 또는 이후에 획득되는지 여부가 다릅니다.

다음 예시와 같이 여러 트랜잭션이 동시에 실행될 때 메타데이터 잠금 획득 순서에 따라 작업 결과가 달라질 수 있습니다.

동일한 구조를 가진 두 개의 테이블 x와 `x_new`로 시작합니다. 세 클라이언트가 이 테이블과 관련된 문을 발행합니다:

클라이언트 1:

```
테이블 잠금 x 쓰기, x_새 쓰기;
```

이 문은 x와 x_new에 대해 이름 순서대로 쓰기 잠금을 요청하고 획득합니다. 클라이언트 2:

```
INSERT INTO x VALUES(1);
```

이 문은 x에 대한 쓰기 잠금을 요청하고 차단합니다. 클라이언트 3:

```
테이블 x의 이름을 x_old로, x_new를 x로 변경합니다;
```

이 문은 x, x_new, x_old에 대해 이름 순서대로 독점 잠금을 요청하지만 x에 대한 잠금을 기다리는 것을 차단합니다.

클라이언트 1:

```
테이블 잠금 해제;
```

이 문은 x와 x_new에 대한 쓰기 잠금을 해제합니다. 클라이언트 3의 x에 대한 독점 잠금 요청은 클라이언트 2의 쓰기 잠금 요청보다 우선순위가 높으므로 클라이언트 3은 x에 대한 잠금을 획득한 다음 x_new 및 x_old에 대한 잠금을 획득하고 이름 변경을 수행한 후 잠금을 해제합니다. 그런 다음 클라이언트 2는 x에 대한 잠금을 획득하고 삽입을 수행한 후 잠금을 해제합니다.

잠금 획득 순서에 따라 테이블 이름 바꾸기가 INSERT보다 먼저 실행됩니다. 삽입이 수행되는 x는 클라이언트 2가 삽입을 수행했을 때 이름이 x_new였던 테이블이 클라이언트 3에 의해 x로 이름이 변경된 테이블입니다:

```
mysql> SELECT * FROM x;
+-------+
| i     |
+-------+
|1 |
+-------+

mysql> SELECT * FROM x_old;
빈 세트 (0.01초)
```

이제 대신 동일한 구조를 가진 x 및 new_x라는 테이블로 시작합니다. 다시 세 명의 클라이언트가 이 테이블과 관련된 문을 발행합니다:

클라이언트 1:

```
테이블 잠금 x 쓰기, new_x 쓰기;
```

이 문은 new_x와 x에 대해 이름 순서대로 쓰기 잠금을 요청하고 획득합니다. 클라이언트

2:

```
INSERT INTO x VALUES(1);
```

이 문은 x에 대한 쓰기 잠금을 요청하고 차단합니다. 클라이언트 3:

```
테이블 x의 이름을 old_x로, new_x를 x로 변경합니다;
```

이 문은 new_x, old_x, x에 대해 이름 순서대로 독점 잠금을 요청하지만 new_x에 대한 잠금을 기다리는 것을 차단합니다.

클라이언트 1:

```
테이블 잠금 해제;
```

이 문은 x와 new_x에 대한 쓰기 잠금을 해제합니다. x의 경우, 보류 중인 유일한 요청은 클라이언트 2의 요청

이므로 클라이언트 2가 해당 잠금을 획득하고 삽입을 수행한 후 잠금을 해제합니다. `new_x`의 경우 유일하게 보류 중인 요청은 클라이언트 3의 요청이며, 클라이언트 3은 해당 잠금을 획득할 수 있습니다(또한 `old_x`의 잠금도 획득할 수 있습니다). 이름 바꾸기 작업은 클라이언트 2 삽입이 완료되어 잠금을 해제할 때까지 `x`에 대한 잠금을 계속 차단합니다. 그런 다음 클라이언트 3이 `x`에 대한 잠금을 획득하고 이름 변경을 수행한 후 잠금을 해제합니다.

이 경우 잠금 획득 순서로 인해 `INSERT`가 `RENAME TABLE`보다 먼저 실행됩니다. `x` 삽입이 발생하는 곳은 원래의 `x`이며, 이름 바꾸기 작업을 통해 `old_x`로 이름이 변경되었습니다:

```
mysql> SELECT * FROM x;
빈 세트 (0.01초)

mysql> SELECT * FROM old_x;
```

```
+------+
| i    |
+------+
|1 |
+------+
```

앞의 예에서와 같이 동시 문에서 잠금 획득 순서가 애플리케이션의 연산 결과에 영향을 미치는 경우 테이블 이름을 조정하여 잠금 획득 순서에 영향을 줄 수 있습니다.

메타데이터 잠금은 필요에 따라 외래 키 제약 조건으로 연결된 테이블로 확장되어 관련 테이블에서 충돌하는 DML 및 DDL 작업이 동시에 실행되는 것을 방지합니다. 상위 테이블을 업데이트할 때 외래 키 메타데이터를 업데이트하는 동안 하위 테이블에서 메타데이터 잠금이 수행됩니다. 외래 키 메타데이터는 자식 테이블이 소유합니다.

메타데이터 잠금 해제

트랜잭션 직렬화를 보장하기 위해 서버는 한 세션이 다른 세션에서 완료되지 않은 명시적 또는 암시적으로 시작된 트랜잭션에서 사용되는 테이블에 대해 데이터 정의 언어(DDL) 문을 수행하도록 허용하지 않아야 합니다. 서버는 트랜잭션 내에서 사용되는 테이블에 대한 메타데이터 잠금을 획득하고 트랜잭션이 종료될 때까지 해당 잠금의 해제를 연기함으로써 이를 달성합니다. 메타데이터 잠금
을 설정하면 테이블의 구조가 변경되지 않습니다. 이 잠금 방식은 한 세션 내의 트랜잭션에서 사용 중인 테이블은 트랜잭션이 종료될 때까지 다른 세션에서 DDL 문에 사용할 수 없다는 의미를 갖습니다.

이 원칙은 트랜잭션 테이블뿐만 아니라 비트랜잭션 테이블에도 적용됩니다. 세션에서 다음과 같이 트랜잭션 테이블 t와 비트랜잭션 테이블 `nt`를 사용하는 트랜잭션을 시작한다고 가정해 보겠습니다:

```
거래를 시작합니다;
SELECT * FROM t;
SELECT * FROM nt;
```

서버는 트랜잭션이 종료될 때까지 t와 `nt` 모두에 메타데이터 잠금을 유지합니다. 다른 세션이 두 테이블에 대해 DDL 또는 쓰기 잠금 작업을 시도하면 트랜잭션 종료 시 메타데이터 잠금이 해제될 때까지 차단됩니다. 예를 들어 두 번째 세션이 이러한 작업을 시도하면 차단됩니다:

```
테이블 삭제 t;
테이블 변경 t ...;
테이블 삭제 nt;
ALTER TABLE nt ...;
LOCK TABLE t ... WRITE;
```

잠금 테이블에도 동일한 동작이 적용됩니다 `... READ`. 즉, 테이블(트랜잭션 또는 비트랜잭션) 블록을 업데이트하는 명시적 또는 암시적으로 시작된 트랜잭션은 해당 테이블에 대해 `LOCK TABLES ... READ`에 의해 차단된 트랜잭션입니다.

서버가 구문적으로는 유효하지만 실행 중에 실패한 문에 대한 메타데이터 잠금을 획득하는 경우, 잠금을 조기에 해제하지 않습니다. 실패한 문이 바이너리 로그에 기록되고 잠금이 로그 일관성을 보호하기 때문에 잠금 해제는 여전히 트랜잭션이 끝날 때까지 연기됩니다.

자동 커밋 모드에서 각 명령문은 사실상 완전한 트랜잭션이므로 명령문에 대해 획득한 메타데이터 잠금은 명령문 끝까지만 유지됩니다.

준비 문 중에 획득한 메타데이터 잠금은 다중 명세서 트랜잭션 내에서 준비가 수행되는 경우에도 명세서가 준비되면 해제됩니다.

준비된 상태의 XA 트랜잭션의 경우, 메타데이터 잠금은 클라이언트 연결 해제와 서버 재시작 시에도 `XA 커밋` 또는 `XA 롤백`이 실행될 때까지 유지됩니다.

8.11.5 외부 잠금

외부 잠금은 파일 시스템 잠금을 사용하여 여러 프로세스의 `MyISAM` 데이터베이스 테이블에 대한 경합을 관리하는 것입니다. 외부 잠금은 MySQL과 같은 단일 프로세스(예

서버가 테이블에 액세스해야 하는 유일한 프로세스라고 가정할 수 없습니다. 다음은 몇 가지 예입니다:

- 동일한 데이터베이스 디렉터리를 사용하는 여러 서버를 실행하는 경우(권장하지 않음) 각 서버는 외부 잠금을 사용하도록 설정해야 합니다.
- MyISAM 테이블에 대한 테이블 유지 관리 작업을 수행하기 위해 myisamchk를 사용하는 경우 서버가 실행 중이 아니거나 서버가 외부 잠금을 사용하도록 설정되어 있는지 확인하여 테이블에 대한 액세스를 위해 필요에 따라 테이블 파일을 잠그도록 해야 합니다. MyISAM 테이블을 패킹하기 위해 myisampack을 사용하는 경우에도 마찬가지입니다.

  외부 잠금을 사용하도록 설정한 상태에서 서버를 실행하는 경우 테이블 확인과 같은 읽기 작업에 언제든지 myisamchk를 사용할 수 있습니다. 이 경우 서버가 myisamchk가 사용 중인 테이블을 업데이트하려고 하면 서버는 myisamchk가 완료될 때까지 기다렸다가 계속 진행합니다.

  테이블 복구 또는 최적화와 같은 쓰기 작업에 myisamchk를 사용하거나 테이블을 패킹하는 데 myisampack을 사용하는 경우 항상 mysqld 서버가 테이블을 사용하고 있지 않은지 확인해야 합니다. mysqld를 중지하지 않는 경우 myisamchk를 실행하기 전에 최소한 mysqladmin 테이블 플러시 작업을 수행해야 합니다. 서버와 myisamchk가 동시에 테이블에 액세스하면 테이블이 *손상될 수 있습니다.*

외부 잠금이 적용되면 테이블에 액세스해야 하는 각 프로세스는 테이블에 액세스하기 전에 테이블 파일에 대한 파일 시스템 잠금을 획득합니다. 필요한 모든 잠금을 획득할 수 없는 경우,
로 설정하면 현재 잠금을 보유하고 있는 프로세스가 잠금을 해제할 때까지 프로세스가 테이블에 액세스하지 못하도록 차단됩니다(현재 잠금을 보유하고 있는 프로세스가 잠금을 해제하면).

외부 잠금은 서버가 테이블에 액세스하기 전에 다른 프로세스를 기다려야 하는 경우가 있기 때문에 서버 성능에 영향을 미칩니다.

단일 서버를 실행하여 지정된 데이터 디렉터리에 액세스하는 경우(일반적인 경우)와 서버가 실행되는 동안 myisamchk와 같은 다른 프로그램이 테이블을 수정할 필요가 없는 경우 외부 잠금은 필요하지 않습니다. 다른 프로그램에서 테이블만 *읽는 경우에는* 외부 잠금이 필요하지 않지만, myisamchk가 테이블을 읽는 동안 서버가 테이블을 변경하면 myisamchk가 경고를 보고할 수 있습니다.

외부 잠금을 사용하지 않도록 설정한 상태에서 myisamchk를 사용하려면 myisamchk가 실행되는 동안 서버를 중지하거나 myisamchk를 실행하기 전에 테이블을 잠그고 플러시해야 합니다. 이 요구 사항을 피하려면 CHECK TABLE 및 REPAIR TABLE 문을 사용하여 MyISAM 테이블을 확인하고 복구합니다.

mysqld의 경우 외부 잠금은 skip_external_locking 시스템 변수의 값에 의해 제어됩니다. 이 변수가 활성화되면 외부 잠금이 비활성화되고, 그 반대의 경우도 마찬가지입니다. 외부 잠금은 기본적으로 비활성화되어 있습니다.

외부 잠금 사용은 서버 시작 시 --external-locking 또는 --skip-external-locking 옵션을 사용하여 제어할 수 있습니다.

외부 잠금 옵션을 사용하여 여러 MySQL 프로세스에서 MyISAM 테이블을 업데이트할 수 있도록 하는 경우,

지연 키 쓰기 시스템 변수를 ALL로 설정한 상태에서 서버를 시작하거나 공유 테이블에 대해 지연 키 쓰기 =1 테이블 옵션을 사용하지 마세요. 그렇지 않으면 인덱스 손상이 발생할 수 있습니다.

이 조건을 충족하는 가장 쉬운 방법은 항상 --external-locking과 -- delay-key-write=OFF를 함께 사용하는 것입니다. (많은 설정에서 앞의 옵션을 혼합하여 사용하는 것이 유용하기 때문에 기본적으로 이 옵션을 사용하지 않습니다.)

8.12 MySQL 서버 최적화

이 섹션에서는 데이터베이스 서버의 최적화 기법에 대해 설명하며, 주로 SQL 문 튜닝보다는 시스템 구성을 다룹니다. 이 섹션의 정보는 관리하는 서버 전반의 성능과 확장성을 보장하려는 DBA, 데이터베이스 설정이 포함된 설치 스크립트를 작성하는 개발자, 개발, 테스트 등을 위해 MySQL을 직접 실행하는 사용자 중 생산성을 극대화하려는 사용자에게 적합합니다.

8.12.1 디스크 I/O 최적화

이 섹션에서는 데이터베이스 서버에 더 많은 스토리지 하드웨어를 더 빠르게 할당할 수 있는 경우 스토리지 장치를 구성하는 방법에 대해 설명합니다. I/O 성능을 개선하기 위해 `InnoDB` 구성을 최적화하는 방법에 대한 자세한 내용은 섹션 8.5.8, "InnoDB 디스크 I/O 최적화"를 참조하십시오.

- 디스크 검색은 성능의 큰 병목 현상입니다. 이 문제는 데이터의 양이 너무 많아져 효과적인 캐싱이 불가능해지기 시작하면 더욱 분명해집니다. 대규모의 경우
  데이터에 무작위로 액세스하는 데이터베이스의 경우, 읽기를 위해서는 적어도 한 번의 디스크 탐색이 필요하고 쓰기를 위해서는 두 번의 디스크 탐색이 필요할 수 있습니다. 이 문제를 최소화하려면 탐색 시간이 짧은 디스크를 사용하세요.

- 파일을 다른 디스크에 심볼릭 링크하거나 디스크를 스트라이핑하여 사용 가능한 디스크 스핀들 수를 늘리고(따라서 검색 오버헤드를 줄이세요):

  - 심볼릭 링크 사용

    즉, `MyISAM` 테이블의 경우 인덱스 파일과 데이터 파일을 데이터 디렉터리의 일반적인 위치에서 다른 디스크(스트라이프 처리될 수도 있음)로 심볼릭 링크합니다. 이렇게 하면 디스크가 다른 용도로 사용되지 않는다고 가정할 때 검색 및 읽기 시간이 모두 향상됩니다. 섹션 8.12.2, "심볼릭 링크 사용"을 참조하세요.

    `InnoDB` 테이블에는 심볼릭 링크가 지원되지 않습니다. 그러나 `InnoDB` 데이터 및 로그 파일을 다른 물리적 디스크에 배치할 수 있습니다. 자세한 내용은 섹션 8.5.8, "InnoDB 디스크 I/O 최적화"를 참조하십시오.

  - 스트라이핑

    스트라이핑은 디스크가 여러 개인 경우 첫 번째 블록을 첫 번째 디스크에, 두 번째 블록을 두 번째 디스크에, N번째 블록을 (*`N MOD number_of_disks`*) 디스크에 넣는 방식으로 데이터를 배치하는 것을 의미합니다. 즉, 일반 데이터 크기가 스트라이프 크기보다 작거나 완벽하게 정렬된 경우 훨씬 더 나은 성능을 얻을 수 있습니다. 스트라이핑은 운영 체제 및 스트라이프 크기에 따라 크게 달라지므로 다양한 스트라이프 크기로 애플리케이션을 벤치마킹하세요. 섹션 8.13.2, "자체 벤치마크 사용"을 참조하세요.

    스트라이핑의 속도 차이는 매개변수에 따라 *크게 달라집니다.* 스트라이핑 매개변수와 디스크 수를 어떻게 설정하느냐에 따라 차이가 몇 배로 측정될 수 있습니다. 무작위 또는 순차 액세스에 맞게 최적화할지 선택해야 합니다.

- 안정성을 위해 RAID 0+1(스트라이핑+미러링)을 사용할 수도 있지만, 이 경우 *`N`* 드라이브의 데이터를 보관하려면 2 × *`N`* 드라이브가 필요합니다. 예산이 충분하다면 이 방법이 가장 좋습니다.
  그러나 이를 효율적으로 처리하기 위해 볼륨 관리 소프트웨어에 투자해야 할 수도 있습니다.

- 데이터의 중요도에 따라 RAID 레벨을 변경하는 것이 좋습니다. 예를 들어, 재생성할 수 있는 덜 중요한 데이터는 RAID 0 디스크에 저장하고, 호스트 정보 및 로그와 같은 매우 중요한 데이터는 RAID 0+1 또는 RAID

N 디스크에 저장합니다. RAID *N은* 패리티 비트를 업데이트하는 데 시간이 걸리기 때문에 쓰기 횟수가 많은 경우 문제가 될 수 있습니다.

- 데이터베이스에서 사용하는 파일 시스템에 대한 매개 변수를 설정할 수도 있습니다:

  파일에 마지막으로 액세스한 시간을 알 필요가 없는 경우(데이터베이스 서버에서는 별로 유용하지 않음), `-o noatime` 옵션을 사용하여 파일 시스템을 마운트할 수 있습니다. 이렇게 하면 파일 시스템의 아이노드에서 마지막 액세스 시간에 대한 업데이트를 건너뛰므로 일부 디스크 검색을 피할 수 있습니다.

  많은 운영 체제에서 `-o async` 옵션으로 파일 시스템을 마운트하여 비동기식으로 업데이트하도록 설정할 수 있습니다. 컴퓨터가 상당히 안정적이라면 이 옵션을 사용하면 안정성을 크게 저하시키지 않으면서도 더 나은 성능을 얻을 수 있습니다. (이 플래그는 Linux에서 기본적으로 켜져 있습니다.)

MySQL과 함께 NFS 사용

MySQL과 함께 NFS를 사용할지 여부를 고려할 때는 신중해야 합니다. 운영 체제 및 NFS 버전에 따라 달라질 수 있는 잠재적 문제는 다음과 같습니다:

- NFS 볼륨에 저장된 MySQL 데이터 및 로그 파일이 잠겨서 사용할 수 없게 됩니다. 잠금 문제는 여러 MySQL 인스턴스가 동일한 데이터 디렉터리에 액세스하는 경우에 발생할 수 있습니다.
  또는 정전 등으로 인해 MySQL이 부적절하게 종료된 경우 등입니다. NFS 버전 4는 권고 및 임대 기반 잠금을 도입하여 근본적인 잠금 문제를 해결합니다. 그러나 MySQL 인스턴스 간에 데이터 디렉터리를 공유하는 것은 권장되지 않습니다.
- 메시지가 제대로 수신되지 않거나 네트워크 트래픽이 손실되어 데이터 불일치가 발생할 수 있습니다. 이 문제를 방지하려면 하드 및 인트라 마운트 옵션과 함께 TCP를 사용하세요.
- 최대 파일 크기 제한. NFS 버전 2 클라이언트는 파일의 최저 2GB(서명된 32비트 오프셋)까지만 액세스할 수 있습니다. NFS 버전 3 클라이언트는 더 큰 파일(최대 64비트 오프셋)을 지원합니다. 지원되는 최대 파일 크기는 NFS 서버의 로컬 파일 시스템에 따라 달라집니다.

전문 SAN 환경이나 기타 스토리지 시스템 내에서 NFS를 사용하는 것이 이러한 환경 외부에서 NFS를 사용하는 것보다 안정성이 더 높은 경향이 있습니다. 그러나 SAN 환경 내의 NFS는 직접 연결되거나 버스에 연결된 비회전식 스토리지보다 속도가 느릴 수 있습니다.

NFS를 사용하기로 선택한 경우, 프로덕션 환경에 배포하기 전에 NFS 설정을 철저히 테스트하는 것과 마찬가지로 NFS 버전 4 이상을 사용하는 것이 좋습니다.

8.12.2 심볼릭 링크 사용

데이터베이스 디렉터리에서 다른 위치로 데이터베이스 또는 테이블을 이동하고 새 위치에 대한 심볼릭 링크로 바꿀 수 있습니다. 예를 들어 데이터베이스를 여유 공간이 더 많은 파일 시스템으로 이동하거나 테이블을 다른 디스크로 분산하여 시스템 속도를 높이려는 경우 이 작업을 수행할 수 있습니다.

`InnoDB` 테이블의 경우 15.6.1.2절 "외부에서 테이블 만들기"에 설명된 대로 심볼릭 링크 대신 `CREATE TABLE` 문의 `DATA DIRECTORY` 절을 사용합니다. 이 새로운 기능은 지원되는 크로스 플랫폼 기술입니다.

이 작업을 수행하는 권장 방법은 전체 데이터베이스 디렉터리를 다른 디스크에 심볼릭 링크하는 것입니다. 심볼릭 링크 `MyISAM` 테이블은 최후의 수단으로만 사용합니다.

데이터 디렉터리의 위치를 확인하려면 이 문을 사용합니다:

```
'datadir'과 같은 변수를 표시합니다;
```

8.12.2.1 유닉스에서 데이터베이스에 심볼릭 링크 사용

Unix에서는 이 절차를 사용하여 데이터베이스를 심볼릭 링크합니다:

1. 데이터베이스 만들기를 사용하여 데이터베이스를 만듭니다:

   ```
   mysql> CREATE DATABASE mydb1;
   ```

`CREATE DATABASE`를 사용하면 MySQL 데이터 디렉터리에 데이터베이스가 생성되고 서버가 데이터베이스 디렉터리에 대한 정보로 데이터 사전을 업데이트할 수 있습니다.

2. 서버를 중지하여 이동하는 동안 새 데이터베이스에서 활동이 발생하지 않도록 합니다.

3. 데이터베이스 디렉터리를 여유 공간이 있는 디스크로 옮깁니다. 예를 들어 `tar` 또는 `mv`를 사용합니다. 데이터베이스 디렉터리를 이동하지 않고 복사하는 방법을 사용하는 경우 복사한 후 원본 데이터베이스 디렉터리를 제거합니다.

4. 데이터 디렉터리에서 이동된 데이터베이스 디렉터리로 연결되는 소프트 링크를 만듭니다:

```
$> ln -s /path/to/mydb1 /path/to/datadir
```

   이 명령은 데이터 디렉터리에 `mydb1`이라는 심볼릭 링크를 만듭니다.

5. 서버를 다시 시작합니다.

8.12.2.2 유닉스에서 MyISAM 테이블에 심볼릭 링크 사용

참고

여기에 설명된 심볼릭 링크 지원과 이를 제어하는 `--symbolic-links` 옵션은 더 이상 사용되지 않으며, 향후 MySQL 버전에서 제거될 예정입니다. 또한 이 옵션은 기본적으로 비활성화되어 있습니다.

심볼릭 링크는 `MyISAM` 테이블에 대해서만 완벽하게 지원됩니다. 다른 스토리지 엔진용 테이블에서 사용하는 파일의 경우 심볼릭 링크를 사용하려고 하면 이상한 문제가 발생할 수 있습니다. `InnoDB` 테이블의 경우 15.6.1.2절. '외부에서 테이블 만들기'에 설명된 대체 기술을 대신 사용하십시오.

완전히 작동하는 `realpath()` 호출이 없는 시스템에서는 테이블을 심볼릭 링크하지 마십시오. (Linux 및 Solaris는 `realpath()`를 지원합니다). 시스템에서 심볼릭 링크를 지원하는지 확인하려면 이 문을 사용하여 `have_symlink` 시스템 변수의 값을 확인합니다:

```
'have_symlink'와 같은 변수를 표시합니다;
```

`MyISAM` 테이블의 심볼릭 링크 처리는 다음과 같이 작동합니다:

- 데이터 디렉토리에는 항상 데이터(`.MYD`) 파일과 인덱스(`.MYI`) 파일이 있습니다. 데이터 파일과 인덱스 파일은 다른 곳으로 이동하여 심볼릭 링크를 통해 데이터 디렉터리에서 바꿀 수 있습니다.
- 데이터 파일과 인덱스 파일을 서로 다른 디렉터리에 독립적으로 심볼릭 링크할 수 있습니다.
- 실행 중인 MySQL 서버에 심볼릭 링크를 수행하도록 지시하려면 `데이터 디렉터리` 및 `인덱스 디렉터리` 옵션을 사용하여 `테이블을` 생성합니다. 섹션 13.1.20, "CREATE TABLE 문"을 참조한다. 또는 `mysqld가` 실행되고 있지 않은 경우 명령줄에서 `ln -s를` 사용하여 수동으로 심볼릭 링크를 수행할 수 있습니다.

  

  참고

  `데이터 디렉토리` 및 `인덱스 디렉토리` 옵션 중 하나 또는 둘 다와 함께 사용되는 경로에 MySQL `데이터` 디렉토리가 포함되지 않을 수 있습니다. (버그 #32167)

- `myisamchk`는 심볼릭 링크를 데이터 파일 또는 인덱스 파일로 바꾸지 않습니다. 심볼릭 링크가 가리키는 파일에서 직접 작동합니다. 모든 임시 파일은 데이터 파일 또는 인덱스 파일이 있는 디렉터리에 생성됩니다. `ALTER TABLE`, `OPTIMIZE TABLE` 및 `REPAIR TABLE` 문도 마찬가지입니다.
- 

  참고

  심볼릭 링크를 사용하는 테이블을 삭제하면 심볼릭 *링크와 심볼릭 링크가 가리키는 파일이 모두 삭제됩니다*. 따라서 `루트` 운영 체제 사용자로 `mysqld를` 실행하거나 운영 체제 사용자에게 MySQL 데이터베이스 디렉터리에 대한 쓰기 권한을 허용하지 *않는 것이* 매우 좋습니다.

- `ALTER TABLE ... RENAME` 또는 `RENAME TABLE`을 사용하여 테이블 이름을 변경하고 테이블을 다른 데이터베이스로 이동하지 않으면 데이터베이스 디렉터리의 심볼 링크가 새 이름으로 변경되고 데이터 파일 및 인덱스 파일의 이름도 그에 따라 변경됩니다.

- `ALTER TABLE ... RENAME` 또는 `RENAME TABLE`을 사용하여 테이블을 다른 데이터베이스로 이동하면 테이블이 다른 데이터베이스 디렉터리로 이동됩니다. 테이블 이름이 변경되면 새 데이터베이스 디렉터리의 심볼 링크가 새 이름으로 변경되고 데이터 파일 및 인덱스 파일의 이름도 그에 따라 변경됩니다.

- 심볼릭 링크를 사용하지 않는 경우, `--skip-symbolic-links` 옵션을 사용하여 `mysqld`를 시작하여 아무도 `mysqld`를 사용하여 데이터 디렉터리 외부에 파일을 드롭하거나 이름을 바꿀 수 없도록 합니다.

이러한 테이블 심볼릭 링크 작업은 지원되지 않습니다:

- `ALTER TABLE`은 `데이터 디렉터리` 및 `인덱스 디렉터리` 테이블 옵션을 무시합니다.

8.12.2.3 Windows에서 데이터베이스에 심볼릭 링크 사용

Windows에서는 데이터베이스 디렉터리에 심볼릭 링크를 사용할 수 있습니다. 이렇게 하면 데이터베이스 디렉터리에 대한 심볼릭 링크를 설정하여 다른 위치(예: 다른 디스크)에 데이터베이스 디렉터리를 배치할 수 있습니다. Windows에서 데이터베이스 심볼릭 링크를 사용하는 방법은 유닉스에서 사용하는 방법과 유사하지만 링크 설정 절차가 다릅니다.

`mydb`라는 데이터베이스의 데이터베이스 디렉터리를 `D:\data\mydb`에 배치한다고 가정합니다. 이렇게 하려면 MySQL 데이터 디렉터리에 `D:\data\mydb`를 가리키는 심볼릭 링크를 생성합니다. 그러나 심볼릭 링크를 만들기 전에 필요한 경우 `D:\데이터\mydb` 디렉터리를 생성하여 `D:\데이터\mydb` 디렉터리가 존재하는지 확인합니다. 데이터 디렉터리에 `mydb`라는 이름의 데이터베이스 디렉터리가 이미 있는 경우 이 디렉터리를 `D:\데이터`로 이동합니다. 그렇지 않으면 심볼릭 링크는 아무런 효과가 없습니다. 문제를 방지하려면 데이터베이스 디렉터리를 이동할 때 서버가 실행 중이 아닌지 확인하세요.

Windows에서는 `mklink` 명령을 사용하여 심볼릭 링크를 만들 수 있습니다. 이 명령에는 관리자 권한이 필요합니다.

1. 원하는 데이터베이스 경로가 존재하는지 확인합니다. 이 예제에서는 `D:\data\mydb`와 `mydb`라는 데이터베이스를 사용합니다.
2. 데이터베이스가 아직 존재하지 않는 경우 `mysql` 클라이언트에서 `CREATE DATABASE` mydb를 실행하여 데이터베이스를 생성합니다.
3. MySQL 서비스를 중지합니다.
4. Windows 탐색기 또는 명령줄을 사용하여 데이터 디렉터리에서 `mydb` 디렉터리를 `D:\데이터로` 이동하여 같은 이름의 디렉터리로 바꿉니다.
5. 아직 명령 프롬프트를 사용하고 있지 않다면 명령 프롬프트를 열고 다음과 같이 데이터 디렉터리로 위치를 변경합니다:

   ```
   C:\> cd \path\to\datadir
   ```

   MySQL 설치 위치가 기본 위치인 경우 이 위치를 사용할 수 있습니다:

   ```
   C:\> cd C:\ProgramData\MySQL\MySQL Server 8.2\Data
   ```

6. 데이터 디렉터리에서 데이터베이스 디렉터리의 위치를 가리키는 `mydb`라는 심볼릭 링크를 만듭니다:

   ```
   C:\> mklink /d mydb D:\data\mydb
   ```

7. MySQL 서비스를 시작합니다.

그 후 데이터베이스 `mydb`에서 생성된 모든 테이블은 `D:\data\mydb`에 생성됩니다.

또는 MySQL이 지원되는 모든 Windows 버전에서 데이터 디렉터리에 대상 디렉터리의 경로가 포함된 `.sym` 파일을 생성하여 MySQL 데이터베이스에 대한 심볼릭 링크를 만들 수 있습니다. 파일 이름은 *`db_name`*`.sym` 이어야 하며, 여기서 *`db_name`*은 데이터베이스 이름입니다.

`.sym` 파일을 사용하는 Windows에서 데이터베이스 심볼릭 링크에 대한 지원은 기본적으로 활성화되어 있습니다. `.sym` 파일 심볼릭 링크가 필요하지 않은 경우 `--skip- 심볼릭 링크` 옵션을 사용하여 `mysqld`를 시작하여 지원을 비활성화할 수 있습니다. 시스템에서 `.sym` 파일 심볼릭 링크를 지원하는지 확인하려면 이 문을 사용하여 `have_symlink` 시스템 변수 값을 확인합니다:

```
'have_symlink'와 같은 변수를 표시합니다;
```

`.sym` 파일 심볼릭 링크를 만들려면 이 절차를 따르세요:

1. 데이터 디렉터리로 위치를 변경합니다:

```
C:\> cd \path\to\datadir
```

2. 데이터 디렉터리에서 다음 경로 이름이 포함된 mydb.sym이라는 텍스트 파일을 만듭니다: D:\data\mydb\

참고

새 데이터베이스 및 테이블의 경로 이름은 절대 경로여야 합니다. 상대 경로를 지정하는 경우 위치는 mydb.sym 파일에 상대적입니다.

그 후 데이터베이스 mydb에서 생성된 모든 테이블은 D:\data\mydb에 생성됩니다.

8.12.3 메모리 사용 최적화

8.12.3.1 MySQL이 메모리를 사용하는 방법

MySQL은 데이터베이스 작업의 성능을 향상시키기 위해 버퍼와 캐시를 할당합니다. 기본 구성은 약 512MB의 RAM이 있는 가상 머신에서 MySQL 서버를 시작할 수 있도록 설계되어 있습니다. 특정 캐시 값을 늘려서 MySQL 성능을 향상시킬 수 있습니다.
및 버퍼 관련 시스템 변수를 사용할 수 있습니다. 메모리가 제한된 시스템에서 MySQL을 실행하도록 기본 구성을 수정할 수도 있습니다.

다음 목록은 MySQL이 메모리를 사용하는 몇 가지 방법을 설명합니다. 해당되는 경우 관련 시스템 변수가 참조됩니다. 일부 항목은 스토리지 엔진 또는 기능별로 다릅니다.

- InnoDB 버퍼 풀은 테이블, 인덱스 및 기타 보조 버퍼를 위해 캐시된 InnoDB 데이터를 보관하는 메모리 영역입니다. 대용량 읽기 작업의 효율성을 위해 버퍼 풀은 잠재적으로 여러 행을 보유할 수 있는 페이지로 나뉩니다. 캐시 관리의 효율성을 위해 버퍼 풀은 페이지의 링크된 목록으로 구현되며, 거의 사용되지 않는 데이터는 LRU 알고리즘의 변형을 사용하여 캐시에서 에이징됩니다. 자세한 내용은 섹션 15.5.1, "버퍼 풀"을 참조하세요.

  버퍼 풀의 크기는 시스템 성능에 중요합니다:

  - InnoDB는 서버 시작 시 malloc() 연산을 사용하여 전체 버퍼 풀에 대한 메모리를 할당합니다. innodb_buffer_pool_size 시스템 변수는 버퍼 풀 크기를 정의합니다. 일반적으로 권장되는 innodb_buffer_pool_size 값은 시스템 메모리의 50~75%이며, 서버가 실행되는 동안 동적으로 구성할 수 있습니다. 자세한 내용은 섹션 15.8.3.1, "InnoDB 버퍼 풀 크기 구성"을 참조하세요.
  - 메모리가 많은 시스템에서는 버퍼 풀을 여러 버퍼 풀 인스턴스로 분할하여 동시성을 향상시킬 수 있습니다. innodb_buffer_pool_instances 시스템 변수는 버퍼 풀 인스턴스 수를 정의합니다.
  - 버퍼 풀이 너무 작으면 페이지가 버퍼 풀에서 플러시되었다가 잠시 후 다시 필요하므로 과도한 이탈이 발생할 수 있습니다.
  - 버퍼 풀이 너무 크면 메모리 경쟁으로 인해 스왑이 발생할 수 있습니다.

- 스토리지 엔진 인터페이스를 사용하면 옵티마이저가 여러 행을 읽을 것으로 예상되는 스캔에 사용할 레코드 버퍼의 크기에 대한 정보를 제공할 수 있습니다. 버퍼 크기는 예상 크기에 따라 달라질 수 있습니다. `InnoDB`는 이 가변 크기 버퍼링 기능을 사용하여 행 프리페칭의 이점을 활용하고 래칭 및 B-트리 탐색의 오버헤드를 줄입니다.

- 모든 스레드는 `MyISAM` 키 버퍼를 공유합니다. `키 버퍼` 크기는 `키 버퍼 크기` 시스템 변수에 의해 결정됩니다.

  서버가 `MyISAM` 테이블을 열 때마다 인덱스 파일은 한 번 열리고, 데이터 파일은 테이블에 액세스하는 동시에 실행 중인 각 스레드에 대해 한 번씩 열립니다. 각 동시 실행 스레드마다 테이블 구조, 각 열에 대한 열 구조, `3 * N` 크기의 *버퍼*가 할당됩니다(여기서 *`N`*은 최대 행 길이이며, `BLOB` 열은 포함되지 않음). `BLOB` 열에는 5~8바이트가 필요합니다.

에 BLOB 데이터의 길이를 더한 값입니다. MyISAM 스토리지 엔진은 내부 사용을 위해 여분의 행 버퍼를 하나 더 유지합니다.

- myisam_use_mmap 시스템 변수를 1로 설정하여 모든 MyISAM에 대한 메모리 매핑을 활성화할 수 있습니다. 테이블.
- 내부 인메모리 임시 테이블이 너무 커지면(tmp_table_size 및 max_heap_table_size 시스템 변수를 사용하여 결정됨) MySQL은 테이블을 인메모리에서 InnoDB 스토리지 엔진을 사용하는 온디스크 형식으로 자동 변환합니다. 허용되는 임시 테이블 크기는 섹션 8.4.4, 'MySQL의 내부 임시 테이블 사용'에 설명된 대로 늘릴 수 있습니다.

  CREATE TABLE로 명시적으로 생성된 메모리 테이블의 경우 max_heap_table_size 시스템 변수만 테이블이 커질 수 있는 크기를 결정하며, 온디스크 형식으로 변환되지 않습니다.
- MySQL 성능 스키마는 낮은 수준에서 MySQL 서버 실행을 모니터링하기 위한 기능입니다. 성능 스키마는 서버 시작 시 필요한 메모리를 할당하는 대신 실제 서버 부하에 따라 메모리 사용량을 확장하여 메모리를 점진적으로 동적으로 할당합니다. 메모리가 할당되면 서버가 다시 시작될 때까지 메모리가 해제되지 않습니다. 자세한 내용은 섹션 27.17, "성능 스키마 메모리 할당 모델"을 참조하세요.
- 서버가 클라이언트 연결을 관리하는 데 사용하는 각 스레드에는 스레드별 공간이 필요합니다. 다음 목록은 이러한 공간과 그 크기를 제어하는 시스템 변수를 나타냅니다:
  - 스택(스레드_스택)
  - 연결 버퍼(net_buffer_length)
  - 결과 버퍼(net_buffer_length)

  연결 버퍼와 결과 버퍼는 각각 net_buffer_length 바이트와 같은 크기로 시작하지만 필요에 따라 최대 max_allowed_packet 바이트까지 동적으로 확대됩니다. 결과 버퍼는 각 SQL 문이 끝날 때마다 net_buffer_length 바이트까지 줄어듭니다. 문이 실행되는 동안 현재 문 문자열의 복사본도 할당됩니다.

  각 연결 스레드는 문 다이제스트를 계산하기 위해 메모리를 사용합니다. 서버는 세션당 최대_다이제스트_길이 바이트를 할당합니다. 섹션 27.10, "성능 스키마 문 다이제스트 및 샘플링"을 참조하십시오.
- 모든 스레드는 동일한 기본 메모리를 공유합니다.
- 스레드가 더 이상 필요하지 않으면 스레드가 스레드 캐시로 돌아가지 않는 한 해당 스레드에 할당된 메모리가 해제되어 시스템으로 반환됩니다. 이 경우 메모리는 할당된 상태로 유지됩니다.
- 테이블을 순차적으로 스캔하는 각 요청은 *읽기 버퍼*를 할당합니다. 테이블의 READ_BUFFER_SIZE 시스템 변수는 버퍼 크기를 결정합니다.
- 임의의 순서로 행을 읽을 때(예: 정렬 후) 디스크 찾기를 피하기 위해 *무작위 읽기* 버퍼가 할당될 수 있습니

다. `read_rnd_buffer_size` 시스템 변수는 버퍼 크기를 결정합니다.

- 모든 조인은 단일 패스로 실행되며, 대부분의 조인은 임시 테이블을 사용하지 않고도 수행할 수 있습니다. 대부분의 임시 테이블은 메모리 기반 해시 테이블입니다. 행 길이가 크거나(모든 열 길이의 합으로 계산됨) `BLOB` 열이 포함된 임시 테이블은 디스크에 저장됩니다.
- 정렬을 수행하는 대부분의 요청은 결과 집합 크기에 따라 정렬 버퍼와 0~2개의 임시 파일을 할당합니다. 섹션 B.3.3.5, "MySQL이 임시 파일을 저장하는 위치"를 참조하세요.
- 거의 모든 구문 분석과 계산은 스레드 로컬 및 재사용 가능한 메모리 풀에서 수행됩니다. 작은 항목에는 메모리 오버헤드가 필요하지 않으므로 일반적인 느린 메모리 할당 및 해제 문제를 피할 수 있습니다. 메모리는 예기치 않게 큰 문자열에 대해서만 할당됩니다.

- BLOB 열이 있는 각 테이블에 대해 버퍼가 동적으로 확대되어 더 큰 BLOB 값을 읽을 수 있습니다. 테이블을 스캔하면 버퍼는 가장 큰 BLOB 값만큼 커집니다.

- MySQL에는 테이블 캐시를 위한 메모리와 설명자가 필요합니다. 사용 중인 모든 테이블에 대한 핸들러 구조는 테이블 캐시에 저장되며 "선입선출"(FIFO)로 관리됩니다. table_open_cache 시스템 변수는 초기 테이블 캐시 크기를 정의합니다(8.4.3.1절. 'MySQL이 테이블을 열고 닫는 방법'을 참조하세요).

  MySQL에는 테이블 정의 캐시를 위한 메모리도 필요합니다. table_definition_cache 시스템 변수는 테이블 정의 캐시에 저장할 수 있는 테이블 정의의 수를 정의합니다. 많은 수의 테이블을 사용하는 경우 큰 테이블 정의 캐시를 생성하여 테이블을 여는 속도를 높일 수 있습니다. 테이블 정의 캐시는 테이블 캐시와 달리 공간을 덜 차지하며 파일 설명자를 사용하지 않습니다.

- FLUSH TABLES 문 또는 mysqladmin flush-tables 명령은 사용하지 않는 모든 테이블을 한 번에 닫고 현재 실행 중인 스레드가 완료되면 사용 중인 모든 테이블이 닫히도록 표시합니다. 이렇게 하면 사용 중인 대부분의 메모리를 효과적으로 확보할 수 있습니다. 모든 테이블이 닫힐 때까지 flush tables는 반환되지 않습니다.

- 서버는 GRANT, CREATE USER, CREATE SERVER, INSTALL PLUGIN 문의 결과로 정보를 메모리에 캐시합니다. 이 메모리는 해당 REVOKE, DROP USER, DROP SERVER, UNINSTALL PLUGIN 문에 의해 해제되지 않으므로 캐싱을 유발하는 문들의 인스턴스를 많이 실행하는 서버의 경우 FLUSH PRIVILEGES로 해제하지 않으면 캐시된 메모리 사용량이 증가합니다.

- 복제 토폴로지에서 다음 설정은 메모리 사용량에 영향을 미치며 필요에 따라 조정할 수 있습니다:

  - 복제 소스의 max_allowed_packet 시스템 변수는 소스가 처리를 위해 복제본으로 전송하는 최대 메시지 크기를 제한합니다. 이 설정의 기본값은 64M입니다.

  - 멀티스레드 복제본의 시스템 변수 replica_pending_jobs_size_max는 처리 대기 중인 메시지를 보관하는 데 사용할 수 있는 최대 메모리 양을 설정합니다. 이 설정의 기본값은 128M입니다. 메모리는 필요할 때만 할당되지만 복제 토폴로지가 때때로 대규모 트랜잭션을 처리하는 경우 이 메모리가 사용될 수 있습니다. 이는 소프트 제한이며 더 큰 트랜잭션도 처리할 수 있습니다.

  - 복제 소스 또는 복제본의 rpl_read_size 시스템 변수는 바이너리 로그 파일 및 릴레이 로그 파일에서 읽혀지는 최소 데이터 양(바이트)을 제어합니다. 기본값은 8192바이트입니다. 소스의 덤프 스레드와 복제본의 코디네이터 스레드를 포함하여 바이너리 로그 및 릴레이 로그 파일에서 읽는 각 스레드에 대해 이 값 크기의 버퍼가 할당됩니다.

  - binlog_transaction_dependency_history_size 시스템 변수는 인메모리 기록으로 보관되는 행 해시 수를 제한합니다.

  - max_binlog_cache_size 시스템 변수는 개별 트랜잭션의 메모리 사용량 상한을 지정합니다.

  - max_binlog_stmt_cache_size 시스템 변수는 문 캐시에 의한 메모리 사용량의 상한을 지정합니다.

`ps` 및 기타 시스템 상태 프로그램에서 mysqld가 많은 메모리를 사용한다고 보고할 수 있습니다. 이는 서로 다른 메모리 주소에 스레드 스택이 있기 때문일 수 있습니다. 예를 들어, Solaris 버전의 `ps`는 스택 간에 사용되지 않는 메모리를 사용한 메모리로 계산합니다. 이를 확인하려면 스왑으로 사용 가능한 스왑을 확인하십시오.

`-s`. 여러 메모리 누수 감지기(상용 및 오픈 소스 모두)로 `mysqld`를 테스트하므로 메모리 누수가 없어야 합니다.

8.12.3.2 MySQL 메모리 사용량 모니터링

다음 예제에서는 성능 스키마 및 시스템 스키마를 사용하여 MySQL 메모리 사용량을 모니터링하는 방법을 보여 줍니다.

대부분의 성능 스키마 메모리 계측 기능은 기본적으로 비활성화되어 있습니다. 계측기는 성능 스키마 설정_인스트루먼트 테이블의 ENABLED 열을 업데이트하여 활성화할 수 있습니다. 메모리 계측기의 이름은 *memory/code_영역/계측기_이름* 형식이며, 여기서 *code_영역은* sql 또는 innodb와 같은 값이고 *instrument_name은* 계측기 세부 정보입니다.

1. 사용 가능한 MySQL 메모리 계측기를 보려면 성능 스키마 설정_인스트루먼트 테이블을 쿼리하세요. 다음 쿼리는 모든 코드 영역에 대해 수백 개의 메모리 계측기를 반환합니다.

```
mysql> SELECT * FROM performance_schema.setup_instruments
       WHERE NAME LIKE '%memory%';
```

코드 영역을 지정하여 결과 범위를 좁힐 수 있습니다. 예를 들어 InnoDB로 결과를 제한할 수 있습니다. 메모리 인스트루먼트를 코드 영역으로 지정합니다.

```
mysql> SELECT * FROM performance_schema.setup_instruments
       WHERE NAME LIKE '%memory/innodb%';
+-----------------------------------------+---------+ ------------+
| 이름                                    | 활성화  | 시간        |
+-----------------------------------------+---------+ ------------+
| 메모리/인노드/적응형 해시 인덱스        | 아니요  |             | 아니요  |
| memory/innodb/buf_buf_pool              | NO      |             | 아니요  |
| memory/innodb/dict_stats_index_map_t    | 아니요  |             | 아니요  |
| memory/innodb/dict_stats_n_diff_on_level|         |             | 아니요  |
| 메모리/이노드/기타                      |         |             | 아니요  |
| memory/innodb/row_log_buf               | 아니요  |             | 아니요  |
| memory/innodb/row_merge_sort            | 아니요  |             | 아니요  |
| memory/innodb/std                       | 아니요  |             | 아니요  |
| memory/innodb/trx_sys_t::rw_trx_ids     | 아니요  |             | 아니요  |
...
```

MySQL 설치에 따라 코드 영역에는 performance_schema, sql, 클라이언트, innodb, myisam, csv, 메모리, 블랙홀, 아카이브, 파티션 등이 포함될 수 있습니다.

2. 메모리 계측기를 사용하도록 설정하려면 MySQL 구성 파일에 성능 스키마 계측기 규칙을 추가합니다. 예를 들어 모든 메모리 계측기를 사용하려면 이 규칙을 구성 파일에 추가하고 서버를 다시 시작합니다:

```
performance-schema-instrument='memory/%=COUNTED'
```

참고

시작 시 메모리 계측을 활성화하면 시작 시 발생하는 메모리 할당을 계산할 수 있습니다.

서버를 다시 시작한 후, 사용하도록 설정한 메모리 계측기에 대해 성능 스키마 setup_instruments 테

이블의 ENABLED 열이 YES로 보고되어야 합니다. 메모리 작업에는 시간이 지정되지 않으므로 메모리 계측기에 대해 setup_instruments 테이블의 TIMED 열은 무시됩니다.

```
mysql> SELECT * FROM performance_schema.setup_instruments
       WHERE NAME LIKE '%memory/innodb%';
+---------------------------------------------+---------+ ------------+
| 이름| 활성화 | 시간 |
+---------------------------------------------+---------+ ------------+
| 메모리/인노드/적응형 해시 인덱스| 아니요 | 아니요 |
| memory/innodb/buf_buf_pool| NO | 아니요 |
| memory/innodb/dict_stats_index_map_t | 아니요 | 아니요 |
| memory/innodb/dict_stats_bg_recalc_pool_t | 아니요 | 아니요 |
| memory/innodb/dict_stats_n_diff_on_level | 아니요 | 아니요 |
| 메모리/이노드/기타 | 아니요 | 아니요 |
| memory/innodb/row_log_buf | 아니요 | 아니요 |
| memory/innodb/row_merge_sort | 아니요 | 아니요 |
| memory/innodb/std | 아니요 | 아니요 |
| memory/innodb/trx_sys_t::rw_trx_ids | 아니요 | 아니요 |
...
```

3. 메모리 계측기 데이터를 쿼리합니다. 이 예에서 메모리 계측기 데이터는 EVENT_NAME별로 데이터를 요약하는 성능 스키마 memory_summary_global_by_event_name 테이블에서 쿼리됩니다. EVENT_NAME은 계측기의 이름입니다.

   다음 쿼리는 InnoDB 버퍼 풀에 대한 메모리 데이터를 반환합니다. 열 설명은 섹션 27.12.20.10, "메모리 요약 테이블"을 참조하십시오.

```
mysql> SELECT * FROM performance_schema.memory_summary_global_by_vent_name
       WHERE EVENT_NAME LIKE 'memory/innodb/buf_buf_pool'\G
                    EVENT_NAME: memory/innodb/buf_buf_pool
                   COUNT_ALLOC: 1
                    COUNT_FREE: 0
    SUM_NUMBER_OF_BYTE_ALLOC: 137428992
              합계_바이트_수_자유: 0
               LOW_COUNT_USED: 0
           current_count_used: 1
              high_count_used: 1
     LOW_NUMBER_OF_BYTE_USED: 0
  CURRENT_NUMBER_OF_BYTE_USED: 137428992
     HIGH_NUMBER_OF_BYTE_USED: 137428992
```

   동일한 기본 데이터는 서버 내의 현재 메모리 사용량을 할당 유형별로 분류하여 전 세계적으로 표시하는 sys 스키마 memory_global_by_current_bytes 테이블을 사용하여 쿼리할 수 있습니다.

```
mysql> SELECT * FROM sys.memory_global_by_current_bytes
       WHERE event_name LIKE 'memory/innodb/buf_buf_pool'\G
*************************** 1. 행 *************************** 이벤트_이름:
       memory/innodb/buf_buf_pool
    current_count: 1
    current_alloc: 131.06 MiB
      현재_평균_할당: 131.06 MiB
       high_count: 1
       high_alloc: 131.06 MiB
   high_avg_alloc: 131.06 MiB
```

   이 시스템 스키마 쿼리는 코드 영역별로 현재 할당된 메모리(current_alloc)를 집계합니다:

```
mysql> SELECT SUBSTRING_INDEX(event_name,'/',2) AS
       code_area, FORMAT_BYTES(SUM(current_alloc))
       AS current_alloc
       FROM sys.x$memory_global_by_current_bytes
       GROUP BY SUBSTRING_INDEX(event_name,'/',2)
       ORDER BY SUM(current_alloc) DESC;
+--------------------------+------------------+
| 코드 영역 |                  현재 할당 |
| 메모리/INODB                | 843.24 MiB     |
| 메모리/성능 스키마             | 81.29 MiB      |
| 메모리/마이시스                | 8.20 MiB       |
| 메모리/sql                  | 2.47 MiB       |
| 메모리/메모리                 | 174.01 KiB     |
| 메모리/마이삼                 | 46.53 KiB      |
| 메모리/블랙홀                 | 512바이트        |
| 메모리/연합                  | 512바이트        |
| 메모리/CSV                  | 512바이트        |
| 메모리/바이오| 496바이트       |
+--------------------------+------------------+
```

   sys 스키마에 대한 자세한 내용은 28장, *MySQL sys 스키마*를 참조하세요.

8.12.3.3 큰 페이지 지원 활성화

일부 하드웨어 및 운영 체제 아키텍처는 기본값(일반적으로 4KB)보다 큰 메모리 페이지를 지원합니다. 이 지원의 실제 구현은 기본 하드웨어 및 운영 체제에 따라 다릅니다. 메모리 액세스를 많이 수행하는 애플리케이션의 경우 큰 페이지를 사용하면 번역 룩어사이드 버퍼(TLB) 누락이 줄어들어 성능이 향상될 수 있습니다.

MySQL에서는 InnoDB에서 대용량 페이지를 사용하여 버퍼 풀과 추가 메모리 풀에 메모리를 할당할 수 있습니다.

MySQL에서 대용량 페이지의 표준 사용은 지원되는 최대 크기인 최대 4MB를 사용하려고 시도합니다. Solaris에서는 "초대형 페이지" 기능을 통해 최대 256MB의 페이지를 사용할 수 있습니다. 이 기능은 최신 SPARC 플랫폼에서 사용할 수 있습니다. 이 기능은 --super-large-pages 또는 -- skip-super-large-pages 옵션을 사용하여 사용하거나 사용하지 않도록 설정할 수 있습니다.

MySQL은 Linux의 대용량 페이지 지원(Linux에서는 HugeTLB라고 함) 구현도 지원합니다.

Linux에서 대용량 페이지를 사용하려면 커널이 이를 지원하도록 설정해야 하며 HugeTLB 메모리 풀을 구성해야 합니다. 참고로 HugeTBL API는 Linux 소스의 Documentation/vm/hugetlbpage.txt 파일에 문서화되어 있습니다.

Red Hat Enterprise Linux와 같은 일부 최신 시스템의 커널에는 기본적으로 대용량 페이지 기능이 활성화되어 있을 수 있습니다. 사용 중인 커널에 해당되는지 확인하려면 다음 명령을 사용하여 "huge"가 포함된 출력 줄을 찾아보세요:

```
$> grep -i 거대한 /proc/meminfo
AnonHugePages:   2658304 kB
ShmemHugePages:        0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
거대한 페이지 크기:     2048 kB
Hugetlb:               0 kB
```

비어 있지 않은 명령 출력은 대용량 페이지 지원이 있음을 나타내지만 0 값은 사용하도록 구성된 페이지가 없음을 나타냅니다.

대용량 페이지를 지원하기 위해 커널을 재구성해야 하는 경우 hugetlbpage.txt 파일에서 지침을 참조하세요.

Linux 커널에 대형 페이지 지원이 활성화되어 있다고 가정하고 다음 단계에 따라 MySQL에서 사용하도록 구성합니다:

1. 필요한 대용량 페이지의 수를 결정합니다. 이 값은 InnoDB 버퍼 풀의 크기를 큰 페이지 크기로 나눈 값으로, innodb_buffer_pool_size / Hugepagesize로 계산할 수 있습니다. 버퍼 풀 크기를 기본값으로 가정하고 /proc/meminfo에서 얻은 Hugepagesize 값을 사용하면 134217728 / 2048 또는 65536(64K)이 됩니다. 이 값을 *P라고 부릅니다*.

2. 시스템 루트로 텍스트 편집기에서 /etc/sysctl.conf 파일을 열고 여기에 표시된 줄을 추가합니다. 여기서 *P는* 이전 단계에서 얻은 큰 페이지의 수입니다:

   ```
   vm.nr_hugepages=P
   ```

   이전에 얻은 실제 값을 사용하여 추가 줄은 다음과 같이 표시되어야 합니다:

   ```
   vm.nr_huge_pages=65536
   ```

업데이트된 파일을 저장합니다.

3. 시스템 루트로 다음 명령을 실행합니다:

```
sudo sysctl -p
```

참고

일부 시스템에서는 대용량 페이지 파일의 이름이 약간 다를 수 있습니다(예: 일부 배포판에서는 `nr_hugepages`라고 부릅니다). `sysctl`이 파일 이름과 관련된 오류를 반환하는 경우 `/proc/sys/vm`에서 해당 파일의 이름을 확인하고 그 이름을 대신 사용하세요.

큰 페이지 구성을 확인하려면 앞서 설명한 대로 /proc/meminfo를 다시 확인합니다. 이제 출력에 다음과 같이 0이 아닌 값이 추가로 표시될 것입니다:

```
$> grep -i 거대한 /proc/meminfo
AnonHugePages:    2686976 kB
ShmemHugePages:         0 kB
HugePages_Total:      233
HugePages_Free:       233
HugePages_Rsvd:         0
HugePages_Surp:         0
거대한 페이지 크기:     2048 kB
Hugetlb:           477184 kB
```

4. 선택적으로 Linux 가상 머신을 압축할 수도 있습니다. 여기에 표시된 것과 유사한 스크립트 파일에 있는 일련의 명령을 사용하여 이 작업을 수행할 수 있습니다:

```
동기
화 동
기화
동기
화
에코 3 > /proc/sys/vm/drop_caches 에
코 1 > /proc/sys/vm/com팩트_메모리
```

이 작업을 수행하는 방법에 대한 자세한 내용은 운영 플랫폼 설명서를 참조하세요.

5. 서버에서 사용하는 my.cnf와 같은 구성 파일을 확인하고, innodb_buffer_pool_chunk_size가 대용량 페이지 크기보다 크게 설정되어 있는지 확인합니다. 이 변수의 기본값은 128M입니다.

6. MySQL 서버의 대형 페이지 지원은 기본적으로 비활성화되어 있습니다. 이 기능을 사용하려면 서버를 시작하여 --large-pages. 서버 my.cnf 파일의 [mysqld] 섹션에 다음 줄을 추가하여 이 작업을 수행할 수도 있습니다:

```
큰 페이지=ON
```

이 옵션을 활성화하면 InnoDB는 버퍼 풀과 추가 메모리 풀에 대용량 페이지를 자동으로 사용합니다. InnoDB가 이 작업을 수행할 수 없는 경우 기존 메모리 사용으로 돌아가 오류 로그에 경고를 기록합니다: Warning: 기존 메모리 풀 사용.

다음과 같이 mysqld를 재시작한 후 /proc/meminfo를 다시 확인하여 MySQL이 이제 대용량 페이지를 사용하고 있는지 확인할 수 있습니다:

```
$> grep -i 거대한 /proc/meminfo
AnonHugePages:    2516992 kB
ShmemHugePages:         0 kB
HugePages_Total:      233
HugePages_Free:       222
HugePages_Rsvd:        55
HugePages_Surp:         0
거대한 페이지 크기:     2048 kB
Hugetlb:           477184 kB
```

8.13 성능 측정(벤치마킹)

성능을 측정하려면 다음 요소를 고려하세요:

- 조용한 시스템에서 단일 작업의 속도를 측정하든, 일련의 작업('워크로드')이 일정 기간 동안 어떻게 작동하는지 측정하든 상관없습니다. 간단한 테스트에서는 일반적으로 한 가지 측면(구성 설정, 테이블의 인덱스 세트, 쿼리의 SQL 절)을 변경하는 것이 성능에 어떤 영향을 미치는지 테스트합니다. 벤치마크는 일반적으로 장기간에 걸친 정교한 성능 테스트이며, 그 결과에 따라 하드웨어 및 스토리지 구성과 같은 높은 수준의 선택이나 새 MySQL 버전으로 업그레이드할 시기를 결정할 수 있습니다.
- 벤치마킹을 위해서는 정확한 상황을 파악하기 위해 데이터베이스 워크로드를 시뮬레이션해야 하는 경우가 있습니다.

- 성능은 매우 다양한 요인에 따라 달라질 수 있으므로 몇 퍼센트 포인트의 차이가 결정적인 승리가 아닐 수도 있습니다. 다른 환경에서 테스트하면 결과가 정반대로 바뀔 수도 있습니다.
- 특정 MySQL 기능은 워크로드에 따라 성능에 도움이 되거나 도움이 되지 않을 수 있습니다. 완전성을 위해 항상 해당 기능을 켜고 끈 상태에서 성능을 테스트하세요. 각 워크로드에서 시도해야 할 가장 중요한 기능은 `InnoDB` 테이블에 대한 적응형 해시 인덱스입니다.

이 섹션에서는 개발자 한 명이 수행할 수 있는 간단하고 직접적인 측정 기법부터 수행 및 결과 해석에 추가적인 전문 지식이 필요한 복잡한 측정 기법까지 살펴봅니다.

8.13.1 표현식 및 함수 속도 측정하기

특정 MySQL 표현식 또는 함수의 속도를 측정하려면 `mysql` 클라이언트 프로그램을 사용하여 `BENCHMARK()` 함수를 호출합니다. 구문은 `BENCHMARK(loop_count,expr)` 입니다. 반환 값은 항상 0이지만, `mysql`은 문이 실행되는 데 걸린 대략적인 시간을 표시하는 줄을 인쇄합니다. 예를 들어

```
mysql> SELECT BENCHMARK(1000000,1+1);
+------------------------+
| 벤치마크(1000000,1+1) |
+------------------------+
|0 |
+------------------------+
한 세트에 1행(0.32초)
```

이 결과는 펜티엄 II 400MHz 시스템에서 얻은 결과입니다. 이 시스템에서 MySQL이 0.32초 만에 1,000,000개의 단순 덧셈식을 실행할 수 있음을 보여줍니다.

기본 제공 MySQL 함수는 일반적으로 고도로 최적화되어 있지만 일부 예외가 있을 수 있습니다. 벤치마크() 는 쿼리에 어떤 함수가 문제가 있는지 알아내는 데 유용한 도구입니다.

8.13.2 자체 벤치마크 사용

애플리케이션과 데이터베이스를 벤치마킹하여 병목 지점을 찾아보세요. 하나의 병목 현상을 해결한 후(또는 "더미" 모듈로 대체하여) 다음 병목 현상을 파악할 수 있습니다. 현재 애플리케이션의 전반적인 성능이 만족스럽더라도 언젠가 추가 성능이 필요할 때를 대비해 각 병목 현상에 대한 계획을 세우고 해결 방법을 결정해야 합니다.

무료 벤치마크 제품군은 오픈 소스 데이터베이스 벤치마크(http:// osdb.sourceforge.net/)입니다.

시스템 부하가 매우 높을 때만 문제가 발생하는 경우가 매우 흔합니다. 실제 운영 중인 시스템을 테스트한 후 부하 문제가 발생했을 때 문의하는 고객들이 많습니다. 대부분의 경우 성능 문제는 다음과 같은 문제로 인한 것으로 밝혀졌습니다.
기본 데이터베이스 설계(예: 테이블 스캔이 부하가 높을 때 좋지 않음) 또는 운영 체제 또는 라이브러리 문제. 대부분의 경우 이러한 문제는 시스템이 아직 운영 중이 아니라면 훨씬 쉽게 해결할 수 있습니다.

이와 같은 문제를 방지하려면 최악의 부하 상태에서 전체 애플리케이션을 벤치마킹하세요:

- 여러 클라이언트가 동시에 쿼리를 실행하여 발생하는 높은 부하를 시뮬레이션하는 데 `mysqlslap` 프로그램이 유용할 수 있습니다. 4.5.8절, "mysqlslap - 로드 에뮬레이션 클라이언트"를 참조하세요.

- https:// launchpad.net/sysbench 및 http://osdldbt.sourceforge.net/#dbt2 에서 제공되는 SysBench 및 DBT2와 같은 벤치마킹 패키지를 사용해 볼 수도 있습니다.

이러한 프로그램이나 패키지는 시스템을 중단시킬 수 있으므로 개발 시스템에서만 사용해야 합니다.

8.13.3 performance_schema로 성능 측정

`performance_schema` 데이터베이스의 테이블을 쿼리하여 서버 및 서버가 실행 중인 애플리케이션의 성능 특성에 대한 실시간 정보를 확인할 수 있습니다. 자세한 내용은 27장, *MySQL 성능 스키마를* 참조하십시오.

8.14 서버 스레드(프로세스) 정보 살펴보기

MySQL 서버가 수행 중인 작업을 확인하려면 서버 내에서 실행 중인 스레드 집합이 현재 수행 중인 작업을 나타내는 프로세스 목록을 검토하는 것이 도움이 될 수 있습니다. 예를 들어

```
mysql> SHOW PROCESSLIST\G
*************************** 1. 행 ***************************
     Id: 5
   사용자:
   event_scheduler 호스트
   : localhost
     db: NULL
명령: 데몬 시간:
   2756681
  상태입니다: 빈 대기열 대기 중 정보: NULL
*************************** 2. 행 ***************************
     Id: 20
   사용자: 나
   호스트:
     localhost:52943 db:
     test
명령: 쿼리 시간: 0
  상태: 시작
      정보: 프로세스 목록 표시
```

스레드는 `KILL` 문으로 종료할 수 있습니다. 섹션 13.7.8.4, "KILL 문"을 참조하세요.

8.14.1 프로세스 목록에 액세스하기

다음 설명에서는 프로세스 정보의 출처와 프로세스 정보를 보는 데 필요한 권한을 열거하고 프로세스 목록 항목의 내용을 설명합니다.

- 프로세스 정보 출처
- 프로세스 목록에 액세스하는 데 필요한 권한
- 프로세스 목록 항목의 내용

프로세스 정보 출처

이러한 소스에서 프로세스 정보를 사용할 수 있습니다:

- `프로세스 목록 표시` 문: 섹션 13.7.7.31, "프로세스 목록 표시 문"
- `mysqladmin 프로세스 목록` 명령: 섹션 4.5.2, "mysqladmin - MySQL 서버 관리 프로그램"

- INFORMATION_SCHEMA PROCESSLIST 테이블: 섹션 26.3.23, "정보_화학물질 처리 목록 테이블"
- 성능 스키마 프로세스 목록 테이블: 섹션 27.12.22.7, "프로세스 목록 테이블"
- 성능 스키마는 접두사가 PROCESSLIST_인 이름을 가진 테이블 열을 스레드합니다: 섹션 27.12.22.8, "스레드 테이블"
- 시스템 스키마 프로세스 목록 및 세션 보기: 섹션 28.4.3.22, "프로세스 목록 및 x 프로세스 목록 보기" 및 섹션 28.4.3.33, "세션 및 x$세션 보기"

스레드 테이블은 프로세스 목록 표시, 정보_케미컬 프로세스 목록 및 mysqladmin 프로세스 목록은 다음과 같습니다:

- 스레드 테이블에 대한 액세스는 뮤텍스가 필요하지 않으며 서버 성능에 미치는 영향이 최소화됩니다. 다른 소스는 뮤텍스가 필요하기 때문에 성능에 부정적인 영향을 미칩니다.

참고

스레드 테이블과 마찬가지로 뮤텍스가 필요하지 않고 더 나은 성능 특성을 가진 성능 스키마 프로세스 목록 테이블을 기반으로 SHOW PROCESSLIST의 대체 구현을 사용할 수 있습니다. 자세한 내용은 섹션 27.12.22.7, "프로세스 리스트 테이블"을 참조한다.

- 스레드 테이블에는 다른 소스에는 표시되지 않는 백그라운드 스레드가 표시됩니다. 또한 스레드가 포그라운드 스레드인지 백그라운드 스레드인지 여부, 스레드와 연결된 서버 내 위치 등 다른 소스에는 없는 각 스레드에 대한 추가 정보도 제공합니다. 즉, 스레드 테이블을 사용하여 다른 소스가 할 수 없는 스레드 활동을 모니터링할 수 있습니다.

- 27.12.22.8절. "스레드 테이블"에 설명된 대로 성능 스키마 스레드 모니터링을 사용하거나 사용하지 않도록 설정할 수 있습니다.

이러한 이유로 다른 스레드 정보 소스 중 하나를 사용하여 서버 모니터링을 수행하는 DBA는 대신 스레드 테이블을 사용하여 모니터링할 수 있습니다.

시스템 스키마 프로세스 목록 보기는 성능 스키마 스레드 테이블의 정보를 보다 접근하기 쉬운 형식으로 표시합니다. 시스템 스키마 세션 보기는 시스템 스키마 프로세스 목록 보기와 마찬가지로 사용자 세션에 대한 정보를 표시하지만 백그라운드 프로세스는 필터링된 상태로 표시됩니다.

프로세스 목록에 액세스하는 데 필요한 권한

대부분의 프로세스 정보 소스에서 PROCESS 권한이 있는 경우 다른 사용자의 토론글을 포함한 모든 토론글을 볼 수 있습니다. 그렇지 않은 경우(PROCESS 권한이 없는 경우) 익명이 아닌 사용자는 자신의 스레드에 대한 정보에는 액세스할 수 있지만 다른 사용자의 스레드에는 액세스할 수 없으며, 익명 사용자는 스레드 정보에 액세스할 수 없습니다.

성능 스키마 스레드 테이블도 스레드 정보를 제공하지만 테이블 액세스에는 다른 권한 모델을 사용합니다. 섹션 27.12.22.8, "스레드 테이블"을 참조하세요.

프로세스 목록 항목의 내용

각 프로세스 목록 항목에는 여러 가지 정보가 포함되어 있습니다. 다음 목록에서는 SHOW PROCESSLIST 출력의 레이블을 사용하여 이러한 정보를 설명합니다. 다른 프로세스 정보 소스에서도 유사한 레이블을 사용합니다.

- Id는 스레드와 연결된 클라이언트의 연결 식별자입니다.

- 사용자 및 호스트는 스레드와 연결된 계정을 나타냅니다.

- db는 스레드의 기본 데이터베이스이며, 아무것도 선택하지 않은 경우 NULL입니다.
- 명령 및 상태는 스레드가 수행 중인 작업을 나타냅니다.

  대부분의 상태는 매우 빠른 작업에 해당합니다. 스레드가 특정 상태에 몇 초 동안 머무른다면 조사해야 할 문제가 있을 수 있습니다.

  다음 섹션에는 사용 가능한 명령 값과 상태 값이 범주별로 그룹화되어 나열되어 있습니다. 이러한 값 중 일부는 그 의미가 자명합니다. 다른 값에 대해서는 추가 설명이 제공됩니다.

  

  참고

  프로세스 목록 정보를 검사하는 애플리케이션은 명령과 상태가 변경될 수 있음을 알고 있어야 합니다.

- 시간은 스레드가 현재 상태에 얼마나 오래 있었는지를 나타냅니다. 경우에 따라 스레드의 현재 시간 개념이 변경될 수 있습니다: 스레드에서 SET TIMESTAMP =

값입니다. 복제 SQL 스레드의 경우 이 값은 마지막으로 복제된 이벤트의 타임스탬프와 복제 호스트의 실제 시간 사이의 시간(초)입니다. 섹션 17.2.3, "복제 스레드"를 참조하십시오.

- `Info`는 스레드가 실행 중인 문을 나타내며, 실행 중인 문이 없는 경우 `NULL`입니다. `SHOW PROCESSLIST`의 경우 이 값에는 문의 처음 100자만 포함됩니다. 전체 문을 보려면 `SHOW FULL PROCESSLIST`를 사용하거나 다른 프로세스 정보 소스를 쿼리합니다.

8.14.2 스레드 명령 값

스레드에는 다음 명령 값 중 하나를 사용할 수 있습니다:

- 빈로그 덤프

  바이너리 로그 콘텐츠를 복제본으로 보내기 위한 복제 소스의 스레드입니다.

- 사용자 변경

  스레드가 사용자 변경 작업을 실행 중입니다.

- 스템트 닫기

  스레드가 준비된 문을 닫고 있습니다.

- 연결

  소스에 연결된 복제 수신기 스레드와 복제 작업자 스레드에서 사용합니다.

- 연결 아웃

  복제본이 소스에 연결 중입니다.

- `DB` 생성

  스레드가 데이터베이스 생성 작업을 실행 중입니다.

- 데몬

  이 스레드는 클라이언트 연결을 서비스하는 스레드가 아니라 서버 내부의 스레드입니다.

- `Debug`

  스레드가 디버깅 정보를 생성하고 있습니다.

- 지연된 삽입

  스레드는 지연된 삽입 핸들러입니다.

- `DB` 삭제

  스레드가 데이터베이스 삭제 작업을 실행 중입니다.

- 오류
- 실행

  스레드가 준비된 문을 실행하고 있습니다.

- 가져오기

  스레드는 준비된 문을 실행하여 결과를 가져옵니다.

- 필드 목록

  스레드가 테이블 열에 대한 정보를 검색하고 있습니다.

- 초기화 DB

  스레드가 기본 데이터베이스를 선택하고 있습니다.

- Kill

  스레드가 다른 스레드를 죽이고 있습니다.

- 긴 데이터

  스레드는 준비된 문을 실행한 결과에서 긴 데이터를 검색하고 있습니다.

- 핑

  스레드가 서버 핑 요청을 처리하고 있습니다.

- 준비

  스레드가 준비된 문장을 준비하고 있습니다.

- 프로세스 목록

  스레드가 서버 스레드에 대한 정보를 생성하고 있습니다.

- 쿼리

  단일 스레드 복제 적용 스레드 및 복제 코디네이터 스레드에서 쿼리를 실행하는 동안 사용자 클라이언트에 사용됩니다.

- 종료

  스레드가 종료됩니다.

- 새로 고침

  스레드가 테이블, 로그 또는 캐시를 플러시하거나 상태 변수 또는 복제 서버 정보를 재설정하는 중입니다.

- 슬레이브 등록

  스레드가 복제 서버를 등록하고 있습니다.

- 스템트 재설정

  스레드가 준비된 문을 재설정하고 있습니다.

- 옵션 설정

  스레드가 클라이언트 문 실행 옵션을 설정하거나 재설정하고 있습니다.

- 종료

스레드가 서버를 종료하고 있습니다.

- 수면

  스레드는 클라이언트가 새 문을 보내기를 기다리고 있습니다.

- 통계

  스레드가 서버 상태 정보를 생성하고 있습니다.

- 시간

  미사용.

8.14.3 일반 스레드 상태

다음 목록은 복제와 같은 보다 전문적인 작업이 아닌 일반적인 쿼리 처리와 관련된 스레드 상태 값에 대해 설명합니다. 이 중 대부분은 서버에서 버그를 찾는 데만 유용합니다.

- 생성 후

  이 상태는 스레드가 테이블(내부 임시 테이블 포함)을 생성할 때 테이블을 생성하는 함수가 끝날 때 발생합니다. 이 상태는 오류로 인해 테이블을 만들 수 없는 경우에도 사용됩니다.

- 테이블 변경

  서버가 제자리 변경 테이블을 실행하는 중입니다.

- 분석

  스레드가 MyISAM 테이블 키 분포를 계산하고 있습니다(예: 분석 테이블).

- 권한 확인

  스레드는 서버가 문을 실행하는 데 필요한 권한을 가지고 있는지 확인합니다.

- 표 확인

  스레드가 테이블 확인 작업을 수행 중입니다.

- 정리

  스레드가 하나의 명령을 처리하고 메모리를 확보하고 특정 상태 변수를 재설정할 준비를 하고 있습니다.

- 마감 테이블

  스레드는 변경된 테이블 데이터를 디스크에 플러시하고 사용한 테이블을 닫습니다. 이 작업은 빠르게 진행되어야 합니다. 그렇지 않은 경우 디스크가 꽉 차 있지 않은지, 디스크 사용량이 많지 않은지 확인하세요.

- 스토리지 엔진에 변경 테이블 커밋

  서버가 제자리 테이블 변경을 완료하고 결과를 커밋하고 있습니다.

- HEAP를 온디스크로 변환

  스레드가 내부 임시 테이블을 메모리 테이블에서 온디스크 테이블로 변환하고 있습니다.

- TMP 테이블로 복사

  스레드가 ALTER TABLE 문을 처리 중입니다. 이 상태는 새 구조의 테이블이 생성된 후 행이 테이블에 복사되기

전에 발생합니다.

이 상태의 스레드에 대해 성능 스키마를 사용하여 복사 작업의 진행 상황을 확인할 수 있습니다. 섹션 27.12.5, "성능 스키마 스테이지 이벤트 테이블"을 참조하십시오.

- `그룹 테이블로 복사`

  문에 서로 다른 `ORDER BY` 및 `GROUP BY` 조건이 있는 경우 행이 그룹별로 정렬되어 임시 테이블에 복사됩니다.

- `tmp 테이블에 복사`

  서버가 메모리의 임시 테이블에 복사하고 있습니다.

- 디스크의 tmp 테이블에 복사

  서버가 디스크의 임시 테이블로 복사하는 중입니다. 임시 결과 집합이 너무 커졌습니다(섹션 8.4.4, "MySQL에서 내부 임시 테이블 사용" 참조). 따라서 스레드가 메모리를 절약하기 위해 임시 테이블을 인메모리에서 디스크 기반 형식으로 변경하고 있습니다.

- 인덱스 만들기

  스레드가 ALTER TABLE ... MyISAM 테이블에 대해 키를 활성화합니다.

- 정렬 인덱스 만들기

  스레드가 내부 임시 테이블을 사용하여 확인된 SELECT를 처리하고 있습니다.

- 테이블 만들기

  스레드가 테이블을 생성하고 있습니다. 여기에는 임시 테이블 생성이 포함됩니다.

- tmp 테이블 만들기

  스레드가 메모리 또는 디스크에 임시 테이블을 만들고 있습니다. 테이블이 메모리에 만들어졌지만 나중에 디스크에 있는 테이블로 변환되는 경우 해당 작업 중 상태는 디스크의 tmp 테이블에 복사 중입니다.

- 기본 테이블에서 삭제

  서버가 다중 테이블 삭제의 첫 번째 부분을 실행하고 있습니다. 첫 번째 테이블에서만 삭제하고 다른 (참조) 테이블에서 삭제하는 데 사용할 열과 오프셋을 저장하고 있습니다.

- 참조 테이블에서 삭제

  서버는 다중 테이블 삭제의 두 번째 부분을 실행하고 다른 테이블에서 일치하는 행을 삭제합니다.

- DISCARD_OR_IMPORT_TABLE스페이스

  스레드가 ALTER TABLE ... DISCARD TABLESPACE 또는 ALTER TABLE ... IMPORT 테이블스페이스 문을 처리하고 있습니다.

- 끝

  이 작업은 ALTER TABLE, CREATE VIEW, DELETE, INSERT, SELECT 또는 UPDATE 문의 마지막에 수행되지만 정리하기 전에 수행됩니다.

  최종 상태의 경우 다음과 같은 작업이 발생할 수 있습니다:

  - 바이너리 로그에 이벤트 쓰기
  - 블롭을 포함한 메모리 버퍼 확보하기

- 실행

  스레드가 문을 실행하기 시작했습니다.

- `init_command` 실행

  스레드가 `init_command` 시스템 변수 값에 있는 문을 실행하고 있습니다.

- 아이템 해제

  스레드가 명령을 실행했습니다. 이 상태는 일반적으로 정리가 뒤따릅니다.

- 전체 텍스트 초기화

서버가 자연어 전체 텍스트 검색을 수행할 준비를 하고 있습니다.

- `init`

  이 상태는 `ALTER TABLE`, `DELETE`, `INSERT`, `SELECT` 또는 `UPDATE` 문이 초기화되기 전에 발생합니다. 이 상태에서 서버가 수행하는 작업에는 바이너리 로그 및 `InnoDB` 로그 플러시가 포함됩니다.

- `Killed`

  누군가 스레드에 `KILL` 문을 보냈으므로 다음 번에 킬 플래그를 확인할 때 스레드가 중단되어야 합니다. 플래그는 MySQL의 각 주요 루프에서 확인되지만 경우에 따라 스레드가 종료되기까지 짧은 시간이 걸릴 수 있습니다. 스레드가 다른 스레드에 의해 잠겨 있는 경우 다른 스레드가 잠금을 해제하는 즉시 종료가 적용됩니다.

- 시스템 테이블 잠금

  스레드가 시스템 테이블(예: 시간대 또는 로그 테이블)을 잠그려고 합니다.

- 느린 쿼리 로깅

  스레드가 느린 쿼리 로그에 문을 작성하고 있습니다.

- 로그인

  클라이언트가 성공적으로 인증될 때까지의 연결 스레드의 초기 상태입니다.

- 키 관리

  서버가 테이블 인덱스를 활성화 또는 비활성화하고 있습니다.

- 시스템 테이블 열기

  스레드가 시스템 테이블(예: 시간대 또는 로그 테이블)을 열려고 시도하고 있습니다.

- 테이블 열기

  스레드가 테이블을 열려고 합니다. 이 절차는 열기를 방해하는 요소가 없는 한 매우 빠르게 진행되어야 합니다. 예를 들어 `ALTER TABLE` 또는 `LOCK TABLE` 문은 문이 완료될 때까지 테이블을 열지 못하게 할 수 있습니다. 또한 `table_open_cache` 값이 충분히 큰지 확인하는 것도 좋습니다.

  시스템 테이블의 경우 시스템 테이블 열기 상태가 대신 사용됩니다.

- 최적화

  서버가 쿼리에 대한 초기 최적화를 수행 중입니다.

- 준비

  이 상태는 쿼리 최적화 중에 발생합니다.

- 변경 테이블 준비

  서버가 제자리 테이블 변경을 실행할 준비를 하고 있습니다.

- 오래된 릴레이 로그 삭제

  스레드가 불필요한 릴레이 로그 파일을 제거하고 있습니다.

- 쿼리 종료

  이 상태는 쿼리를 처리한 후 항목 해제 상태가 되기 전에 발생합니다.

- 클라이언트로부터 수신

  서버가 클라이언트로부터 패킷을 읽고 있습니다.

- 중복 제거

  이 쿼리는 `SELECT DISTINCT`를 사용하고 있었기 때문에 MySQL이 초기 단계에서 고유 연산을 최적화할 수 없었습니다. 이 때문에 MySQL은 결과를 클라이언트에 보내기 전에 중복된 행을 모두 제거하기 위한 추가 단계를 필요로 합니다.

- `TMP` 테이블 제거

  `SELECT` 문을 처리한 후 스레드가 내부 임시 테이블을 제거하고 있습니다. 임시 테이블이 생성되지 않은 경우 이 상태는 사용되지 않습니다.

- 이름 바꾸기

  스레드가 테이블 이름을 변경하고 있습니다.

- 결과 테이블 이름 바꾸기

  스레드가 `ALTER TABLE` 문을 처리 중이며 새 테이블을 생성하고 원래 테이블을 대체하기 위해 이름을 변경하고 있습니다.

- 테이블 다시 열기

  스레드가 테이블에 대한 잠금을 얻었지만 잠금을 얻은 후 기본 테이블 구조가 변경된 것을 발견했습니다. 잠금을 해제하고 테이블을 닫은 후 다시 열려고 시도합니다.

- 정렬하여 복구

  복구 코드는 정렬을 사용하여 인덱스를 생성합니다.

- 수리 완료

  스레드가 `MyISAM` 테이블에 대한 다중 스레드 복구를 완료했습니다.

- 키 캐시로 복구

  복구 코드는 키 캐시를 통해 키를 하나씩 생성하는 방식을 사용합니다. 이는 정렬하여 복구합니다.

- 롤백

  스레드가 트랜잭션을 롤백하고 있습니다.

- 저장 상태

  복구 또는 분석과 같은 `MyISAM` 테이블 작업의 경우, 스레드는 새 테이블 상태를 `.MYI` 파일 헤더에 저장합

니다. 상태에는 행 수, 자동 증가 카운터 및 키 분포와 같은 정보가 포함됩니다.

- 업데이트할 행 검색

  스레드는 업데이트하기 전에 일치하는 모든 행을 찾기 위해 첫 번째 단계를 수행합니다. 업데이트가 관련 행을 찾는 데 사용되는 인덱스를 변경하는 경우 이 작업을 수행해야 합니다.

- 데이터 전송

  이 상태는 이제 실행 중 상태에 포함됩니다.

- 고객에게 보내기

  서버가 클라이언트에 패킷을 쓰고 있습니다.

- 설정

  스레드가 테이블 변경 작업을 시작하고 있습니다.

- 그룹 정렬

  스레드가 GROUP BY를 만족하기 위해 정렬을 수행하고 있습니다.

- 주문 정렬

  스레드는 ORDER BY를 만족하기 위해 정렬을 수행합니다.

- 정렬 색인

  스레드는 MyISAM 테이블 최적화 작업 중에 보다 효율적인 액세스를 위해 인덱스 페이지를 정렬하고 있습니다.

- 정렬 결과

  SELECT 문의 경우 이는 정렬 인덱스 만들기와 유사하지만 임시 테이블이 아닌 경우입니다.

- 시작

  문 실행이 시작되는 첫 번째 단계입니다.

- 통계

  서버가 쿼리 실행 계획을 개발하기 위해 통계를 계산하고 있습니다. 스레드가 오랫동안 이 상태에 있으면 서버가 다른 작업을 수행하느라 디스크에 묶여 있는 것일 수 있습니다.

- 시스템 잠금

  스레드가 mysql_lock_tables()를 호출했는데 그 이후로 스레드 상태가 업데이트되지 않았습니다. 이는 여러 가지 이유로 발생할 수 있는 매우 일반적인 상태입니다.

  예를 들어, 스레드가 테이블에 대한 내부 또는 외부 시스템 잠금을 요청하거나 대기 중인 경우입니다. 이 문제는 InnoDB가 테이블 잠금 실행 중에 테이블 수준 잠금을 기다릴 때 발생할 수 있습니다.
  이 상태가 외부 잠금 요청으로 인해 발생하고 다중 mysqld를 사용하지 않는 경우
  서버에 액세스하는 경우 외부 시스템 잠금을 비활성화할 수 있습니다.
  --외부 잠금 건너뛰기 옵션. 그러나 외부 잠금은 기본적으로 비활성화되어 있으므로 이 옵션은 효과가 없을 수 있습니다. SHOW PROFILE의 경우 이 상태는 스레드가 잠금을 요청하고 있음을 의미합니다(잠금을 기다리는 것이 아님).

  시스템 테이블의 경우 시스템 테이블 잠금 상태가 대신 사용됩니다.

- 업데이트

  스레드가 테이블 업데이트를 시작할 준비를 하고 있습니다.

- 업데이트

스레드가 업데이트할 행을 검색하여 업데이트하고 있습니다.

- 기본 테이블 업데이트

  서버가 다중 테이블 업데이트의 첫 번째 부분을 실행하고 있습니다. 첫 번째 테이블만 업데이트하고 다른 (참조) 테이블을 업데이트하는 데 사용할 열과 오프셋을 저장하고 있습니다.

- 참조 테이블 업데이트

  서버는 다중 테이블 업데이트의 두 번째 부분을 실행하고 다른 테이블에서 일치하는 행을 업데이트합니다.

- 사용자 잠금

스레드가 `GET_LOCK()` 호출로 요청된 권고 잠금을 요청하거나 대기 중입니다. `SHOW PROFILE`의 경우, 이 상태는 스레드가 잠금을 요청 중(잠금을 기다리는 중이 아님)임을 의미합니다.

- 사용자 수면

  스레드가 `SLEEP()` 호출을 호출했습니다.

- 커밋 잠금을 기다리는 중

  읽기 잠금으로 테이블을 플러시하면 커밋 잠금을 기다립니다.

- 핸들러 커밋 대기 중

  스레드는 트랜잭션이 커밋되기를 기다리는 동안 쿼리 처리의 다른 부분과 대기 중입니다.

- 테이블 대기

  스레드가 테이블의 기본 구조가 변경되었다는 알림을 받았으며 새 구조를 가져오려면 테이블을 다시 열어야 합니다. 그러나 테이블을 다시 열려면 다른 모든 스레드가 해당 테이블을 닫을 때까지 기다려야 합니다.

  이 알림은 다른 스레드에서 해당 테이블에 대해 `FLUSH TABLES` 또는 다음 문 중 하나를 사용한 경우 발생합니다: `FLUSH TABLES tbl_name`, `ALTER TABLE`, `RENAME TABLE`, `REPAIR TABLE`, `ANALYZE TABLE` 또는 `OPTIMIZE TABLE`.

- 테이블 플러시 대기

  스레드가 테이블 플러시를 실행 중이고 모든 스레드가 테이블을 닫을 때까지 기다리거나, 테이블의 기본 구조가 변경되어 새 구조를 가져오기 위해 테이블을 다시 열어야 한다는 알림을 받은 경우입니다. 그러나 테이블을 다시 열려면 다른 모든 스레드가 해당 테이블을 닫을 때까지 기다려야 합니다.

  이 알림은 다른 스레드에서 해당 테이블에 대해 `FLUSH TABLES` 또는 다음 문 중 하나를 사용한 경우 발생합니다: `FLUSH TABLES tbl_name`, `ALTER TABLE`, `RENAME TABLE`, `REPAIR TABLE`, `ANALYZE TABLE` 또는 `OPTIMIZE TABLE`.

- *잠금 유형* 잠금을 기다리는 중

  서버는 메타데이터 잠금 하위 시스템에서 THR_LOCK 잠금 또는 잠금을 획득하기 위해 대기 중이며, 여기서 *lock_type은* 잠금 유형을 나타냅니다.

  이 상태는 THR_LOCK을 기다리는 중임을 나타냅니다:

  - 테이블 레벨 잠금 대기 중

  이러한 상태는 메타데이터 잠금을 기다리는 중임을 나타냅니다:

  - 이벤트 메타데이터 잠금을 기다리는 중
  - 글로벌 읽기 잠금을 기다리는 중
  - 스키마 메타데이터 잠금을 기다리는 중
  - 저장된 함수 메타데이터 잠금을 기다리는 중
  - 저장 프로시저 메타데이터 잠금을 기다리는 중
  - 테이블 메타데이터 잠금을 기다리는 중
  - 트리거 메타데이터 잠금을 기다리는 중

  테이블 잠금 표시기에 대한 자세한 내용은 섹션 8.11.1, '내부 잠금 방법'을 참조하세요. 메타데이터 잠금에 대한 자세한 내용은 섹션 8.11.4, "메타데이터 잠금"을 참조하십시오. 잠금 요청을 차단하는 잠금을 확인하려면 27.12.13절. "성능 스키마 잠금 테이블"에 설명된 성능 스키마 잠금 테이블을 사용하세요.

- 컨디션 대기 중

  스레드가 조건이 참이 되기를 기다리는 일반적인 상태입니다. 특정 상태 정보를 사용할 수 없습니다.

- 인터넷에 쓰기

  서버가 네트워크에 패킷을 쓰고 있습니다.

8.14.4 복제 소스 스레드 상태

다음 목록은 복제 소스의 Binlog 덤프 스레드에 대한 상태 열에 표시될 수 있는 가장 일반적인 상태를 보여줍니다. 소스에 Binlog 덤프 스레드가 표시되지 않으면 복제가 실행 중이 아니며, 즉 현재 연결된 복제본이 없다는 뜻입니다.

MySQL 8.0에서는 계측 이름에 호환되지 않는 변경 사항이 적용되었습니다. 이러한 계측 이름과 함께 작동하는 모니터링 도구가 영향을 받을 수 있습니다. 호환되지 않는 변경 사항으로 인해 영향을 받는 경우, 용어 사용 이전 시스템 변수를 BEFORE_8_0_26으로 설정하여 이전 목록에 지정된 개체에 대한 이전 버전의 이름을 MySQL Server에서 사용하도록 하세요. 이렇게 하면 이전 이름을 사용하는 모니터링 도구가 새 이름을 사용하도록 업데이트될 때까지 계속 작동할 수 있습니다.

개별 함수를 지원하려면 세션 범위로 `termination_use_previous` 시스템 변수를 설정하거나 모든 새 세션에 대해 전역 범위를 기본값으로 설정합니다. 전역 범위를 사용하면 느린 쿼리 로그에 이전 버전의 이름이 포함됩니다.

- `하나의 빈로그 읽기 완료, 다음 빈로그로 전환`

  스레드가 바이너리 로그 파일 읽기를 완료하고 복제본으로 보낼 다음 파일을 여는 중입니다.

- `마스터가 모든 빈로그를 슬레이브로 보냈습니다. 추가 업데이트를 기다리는 중입니다`

  `. 원본이 모든 빈로그를 복제본으로 보냈습니다. 추가 업데이트를 기다리는 중입니`

  `다.`

스레드가 바이너리 로그에서 나머지 모든 업데이트를 읽고 이를 복제본으로 보냈습니다. 이제 스레드는 유휴 상태가 되어 소스에서 발생하는 새 업데이트로 인해 바이너리 로그에 새 이벤트가 나타날 때까지 기다립니다.

- 슬레이브에 빈로그 이벤트 보내기 복제본

  에 빈로그 이벤트 보내기

  바이너리 로그는 이벤트로 구성되며, 이벤트는 일반적으로 업데이트와 몇 가지 다른 정보로 구성됩니다. 스레드가 바이너리 로그에서 이벤트를 읽었으며 이제 이를 복제본으로 전송하고 있습니다.

- 해지 완료 대기 중

  스레드가 중지될 때 발생하는 매우 짧은 상태입니다.

8.14.5 복제 I/O(수신기) 스레드 상태

다음 목록은 복제 서버의 복제 I/O(수신자) 스레드에 대한 상태 열에 표시되는 가장 일반적인 상태를 보여 줍니다. 이 상태는 `Replica_IO_State` 열에도 표시됩니다.
를 사용하여 복제본 상태를 표시하므로 이 문을 사용하면 어떤 일이 일어나고 있는지 잘 파악할 수 있습니다.

MySQL 8.0에서는 계측 이름에 호환되지 않는 변경 사항이 적용되었습니다. 이러한 계측 이름과 함께 작동하는 모니터링 도구가 영향을 받을 수 있습니다. 호환되지 않는 변경 사항으로 인해 영향을 받는 경우, 용어 사용 이전 시스템 변수를 `BEFORE_8_0_26`으로 설정하여 이전 목록에 지정된 개체에 대한 이전 버전의 이름을 MySQL Server에서 사용하도록 하세요. 이렇게 하면 이전 이름을 사용하는 모니터링 도구가 새 이름을 사용하도록 업데이트될 때까지 계속 작동할 수 있습니다.

개별 함수를 지원하려면 세션 범위로 `termination_use_previous` 시스템 변수를 설정하거나 모든 새 세션에 대해 전역 범위를 기본값으로 설정합니다. 전역 범위를 사용하면 느린 쿼리 로그에 이전 버전의 이름이 포함됩니다.

- 마스터 버전 확인 소스 버전

  확인

  소스에 대한 연결이 설정된 후 매우 짧게 발생하는 상태입니다.

- 마스터에 연결하기 소스에

  연결하기

  스레드가 소스에 연결을 시도하고 있습니다.

- 마스터 이벤트를 릴레이 로그에 큐 큐잉 소스 이벤

트를 릴레이 로그에 큐잉

스레드가 이벤트를 읽고 SQL 스레드가 처리할 수 있도록 릴레이 로그에 복사하고 있습니다.

- 빈로그 덤프 요청 실패 후 다시 연결하기

  스레드가 소스에 다시 연결하려고 합니다.

- 마스터 이벤트 읽기에 실패한 후 다시 연결하기 소스 이벤트

  읽기에 실패한 후 다시 연결하기

  스레드가 소스에 다시 연결하려고 합니다. 연결이 다시 설정되면 상태는 마스터가 이벤트를 보낼 때까지 대기 중이 됩니다.

- 마스터에 슬레이브 등록하기 소스에 복

  제본 등록하기

  소스에 대한 연결이 설정된 후 매우 짧게 발생하는 상태입니다.

- 빈로그 덤프 요청하기

  소스에 대한 연결이 설정된 후 매우 짧게 발생하는 상태입니다. 스레드는 요청된 바이너리 로그 파일 이름과 위치부터 시작하여 바이너리 로그의 내용에 대한 요청을 소스에 보냅니다.

- 커밋 차례를 기다리는 중

  다음과 같은 경우 복제 스레드가 이전 작업자 스레드의 커밋을 기다릴 때 발생하는 상태입니다. REPLICA_PRESERVE_COMMIT_ORDER가 활성화되어 있습니다.

- 마스터가 이벤트를 전송할 때까지 기다리는

  중 소스에서 이벤트를 전송할 때까지 기다

  리는 중

  스레드가 소스에 연결되었고 바이너리 로그 이벤트가 도착하기를 기다리고 있습니다. 소스가 유휴 상태인 경우 이 대기 시간이 오래 지속될 수 있습니다. 대기 시간이 `replica_net_timeout` 초 동안 지속되면 시간 초과가 발생합니다. 이 시점에서 스레드는 연결이 끊어진 것으로 간주하고 재연결을 시도합니다.

- 마스터 업데이트 대기 중 소스 업

  데이트 대기 중

  마스터에 연결하거나 소스에 연결하기 전의 초기 상태입니다.

- 종료 시 슬레이브 뮤텍스 대기 중 종료 시

  복제본 뮤텍스 대기 중

  스레드가 중지되는 동안 잠시 발생하는 상태입니다.

- 슬레이브 SQL 스레드가 충분한 릴레이 로그 공간을 확보할 때까지 기다리는 중 복제본

  SQL 스레드가 충분한 릴레이 로그 공간을 확보할 때까지 기다리는 중

0이 아닌 릴레이 로그 공간 제한 값을 사용하고 있으며 릴레이 로그의 크기가 이 값을 초과할 정도로 커졌습니다. I/O(수신자) 스레드는 일부 릴레이 로그 파일을 삭제할 수 있도록 SQL(적용자) 스레드가 릴레이 로그 내용을 처리하여 충분한 공간을 확보할 때까지 대기 중입니다.

- 빈로그 덤프 요청 실패 후 재연결 대기 중

  바이너리 로그 덤프 요청이 실패한 경우(연결 끊김으로 인해) 스레드는 절전 상태로 전환된 후 주기적으로 재연결을 시도합니다. 재시도 간격은 복제 소스 변경을 사용하여 지정할 수 있습니다.

- 마스터 이벤트 읽기에 실패한 후 다시 연결 대기 중 소스 이벤트 읽기

  에 실패한 후 다시 연결 대기 중

  읽는 동안 오류가 발생했습니다(연결 끊김으로 인해). 다시 연결을 시도하기 전에 `CHANGE REPLICATION SOURCE TO` 문에서 설정한 시간(초) 동안 스레드가 절전 중입니다.

8.14.6 복제 SQL 스레드 상태

다음 목록은 복제 서버의 복제 SQL 스레드에 대한 상태 열에 표시될 수 있는 가장 일반적인 상태를 보여줍니다.

MySQL 8.0에서는 계측 이름에 호환되지 않는 변경 사항이 적용되었습니다. 이러한 계측 이름과 함께 작동하는 모니터링 도구가 영향을 받을 수 있습니다. 호환되지 않는 변경 사항으로 인해 영향을 받는 경우, 용어 사용 이전 시스템 변수를 BEFORE_8_0_26으로 설정하여 이전 목록에 지정된 개체에 대한 이전 버전의 이름을 MySQL Server에서 사용하도록 하세요. 이렇게 하면 이전 이름을 사용하는 모니터링 도구가 새 이름을 사용하도록 업데이트될 때까지 계속 작동할 수 있습니다.

개별 함수를 지원하려면 세션 범위로 termination_use_previous 시스템 변수를 설정하거나 모든 새 세션에 대해 전역 범위를 기본값으로 설정합니다. 전역 범위를 사용하면 느린 쿼리 로그에 이전 버전의 이름이 포함됩니다.

- 데이터 파일 불러오기를 재생하기 전에 임시 파일 만들기(추가)

  스레드는 LOAD DATA 문을 실행 중이며 복제본이 행을 읽는 데이터가 포함된 임시 파일에 데이터를 추가하고 있습니다.

- 데이터 파일 불러오기를 재생하기 전에 임시 파일 만들기(생성)

  스레드가 LOAD DATA 문을 실행 중이며 복제본이 행을 읽는 데이터가 포함된 임시 파일을 생성하고 있습니다. 이 상태는 MySQL 5.0.3보다 낮은 버전의 MySQL을 실행하는 소스에서 원본 LOAD DATA 문을 로깅한 경우에만 발생할 수 있습니다.

- 릴레이 로그에서 이벤트 읽기

  스레드가 릴레이 로그에서 이벤트를 읽었으므로 이벤트를 처리할 수 있습니다.

- 슬레이브가 모든 릴레이 로그를 읽었습니다. 추가 업데이트를 기다리는 중

  입니다. 복제본이 모든 릴레이 로그를 읽었습니다. 추가 업데이트를 기다리

  는 중입니다.

  스레드는 릴레이 로그 파일에 있는 모든 이벤트를 처리했으며, 이제 I/O(수신자) 스레드가 릴레이 로그에 새 이벤트를 기록할 때까지 기다리고 있습니다.

- 코디네이터의 이벤트를 기다리는 중

  멀티스레드 복제본(replica_parallel_workers가 1보다 큼)을 사용하면 복제본 작업자 스레드 중 하나가 코디네이터 스레드에서 이벤트를 대기합니다.

- 종료 시 슬레이브 뮤텍스 대기 중 종료 시

  복제 뮤텍스 대기 중

스레드가 중지될 때 발생하는 매우 짧은 상태입니다.

- 슬레이브 워커가 보류 중인 이벤트를 해제하기를 기다리는 중 복제

  워커가 보류 중인 이벤트를 해제하기를 기다리는 중

  이 대기 작업은 워커가 처리 중인 이벤트의 총 크기가 `replica_pending_jobs_size_max` 시스템 변수의 크기를 초과할 때 발생합니다. 크기가 이 제한 아래로 떨어지면 코디네이터가 스케줄링을 재개합니다. 이 상태는 `replica_parallel_workers`가 0보다 크게 설정된 경우에만 발생합니다.

- 릴레이 로그에서 다음 이벤트 대기 중

  릴레이 로그에서 이벤트를 읽기 전 초기 상태입니다.

- 마스터 실행 이벤트 후 `MASTER_DELAY` 초까지 대기 중 마스터 실행 이벤트 후

  `SOURCE_DELAY` 초까지 대기 중

SQL 스레드가 이벤트를 읽었지만 복제 지연이 경과되기를 기다리는 중입니다. 이 지연은 복제 원본 변경 대상의 SOURCE_DELAY | MASTER_DELAY 옵션으로 설정됩니다.

SQL 스레드의 정보 열에 문 텍스트가 표시될 수도 있습니다. 이는 스레드가 릴레이 로그에서 이벤트를 읽고 이로부터 문을 추출하여 실행 중일 수 있음을 나타냅니다.

8.14.7 복제 연결 스레드 상태

이러한 스레드 상태는 복제본 서버에서 발생하지만 I/O 또는 SQL 스레드가 아닌 연결 스레드와 연관되어 있습니다.

- 마스터 변경

  복제 소스 변경

  스레드가 CHANGE REPLICATION SOURCE TO 문을 처리하고 있습니다.

- 노예 죽이기

  스레드가 STOP REPLICA 문을 처리 중입니다.

- 마스터 덤프 테이블 열기

  이 상태는 마스터 덤프에서 테이블을 생성한 후에 발생합니다.

- 마스터 덤프 테이블 데이터 읽기

  이 상태는 마스터 덤프 테이블을 연 후에 발생합니다.

- 마스터 덤프 테이블의 인덱스 재구축

  이 상태는 마스터 덤프 테이블 데이터를 읽은 후에 발생합니다.

8.14.8 NDB 클러스터 스레드 상태

- 빈로그에 이벤트 커밋하기
- mysql.ndb_apply_status 열기
- 이벤트 처리

  스레드가 바이너리 로깅을 위한 이벤트를 처리하고 있습니다.

- 스키마 테이블에서 이벤트 처리

  스레드가 스키마 복제 작업을 수행합니다.

- 종료
- 인덱스 테이블 스키마 작업 및 빈로그 동기화

  이것은 NDB에 대한 스키마 작업의 올바른 바이너리 로그를 갖는 데 사용됩니다.

- `ndbcluster` 전역 스키마 잠금이 허용되기를 기다리는 중입니다.

  스레드가 전역 스키마 잠금을 받을 수 있는 권한을 기다리고 있습니다.

- 데이터베이스 클러스터에서 이벤트를 기다리는 중입니다.

  서버는 NDB 클러스터에서 SQL 노드 역할을 하며 클러스터 관리 노드에 연결되어 있습니다.

- 데이터베이스 클러스터에서 첫 번째 이벤트를 기다리는 중입니다.
- 현재 위치에 도달할 때까지 ndbcluster 빈로그 업데이트를 기다리는 중입니다.
- dbcluster 글로벌 스키마 잠금을 기다리는 중입니다.

  스레드가 다른 스레드가 보유한 전역 스키마 잠금이 해제되기를 기다리고 있습니다.

- dbcluster가 시작되기를 기다리는 중입니다.
- 스키마 에포크 대기 중

  스레드가 스키마 에포크(즉, 전역 체크포인트)를 기다리는 중입니다.

8.14.9 이벤트 스케줄러 스레드 상태

이러한 상태는 이벤트 스케줄러 스레드, 예약된 이벤트를 실행하기 위해 생성된 스레드 또는 스케줄러를 종료하는 스레드에 대해 발생합니다.

- 지우기

  스케줄러 스레드 또는 이벤트를 실행하던 스레드가 종료 중이며 곧 종료됩니다.

- 초기화

  스케줄러 스레드 또는 이벤트를 실행하는 스레드가 초기화되었습니다.

- 다음 활성화 대기 중

  스케줄러에 비어 있지 않은 이벤트 대기열이 있지만 다음 활성화가 향후에 있습니다.

- 스케줄러가 중지되기를 기다리는 중

  스레드가 SET GLOBAL event_scheduler=OFF를 실행하고 스케줄러가 중지되기를 기다리는 중입니다.

- 빈 대기열에서 대기 중

  스케줄러의 이벤트 큐가 비어 있고 절전 중입니다.