15장 InnoDB 스토리지 엔진

목차

15.1 InnoDB 소개

InnoDB는 높은 안정성과 고성능의 균형을 이루는 범용 스토리지 엔진입니다. MySQL 8.2에서 `InnoDB`는 기본 MySQL 스토리지 엔진입니다. 다른 기본 스토리지 엔진을 구성하지 않은 경우 `ENGINE` 절 없이 `CREATE TABLE` 문을 실행하면 `InnoDB` 테이블이 생성됩니다.

InnoDB의 주요 이점

- DML 작업은 사용자 데이터를 보호하기 위한 커밋, 롤백 및 충돌 복구 기능을 갖춘 트랜잭션과 함께 ACID 모델을 따릅니다. 섹션 15.2, "InnoDB와 ACID 모델"을 참조하세요.
- 행 수준 잠금과 Oracle 스타일의 일관된 읽기는 다중 사용자 동시성 및 성능을 향상시킵니다. 섹션 15.7, "InnoDB 잠금 및 트랜잭션 모델"을 참조하세요.
- `InnoDB` 테이블은 기본 키를 기반으로 쿼리를 최적화하기 위해 디스크에 데이터를 정렬합니다. 각 `InnoDB` 테이블에는 클러스터된 인덱스라고 하는 기본 키 인덱스가 있어 기본 키 조회를 위한 I/O를 최소화하도록 데이터를 구성합니다. 섹션 15.6.2.1, "클러스터된 인덱스 및 보조 인덱스"를 참조하십시오.
- 데이터 무결성을 유지하기 위해 `InnoDB`는 외래 키 제약 조건을 지원합니다. 외래 키를 사용하면 삽입, 업데이트 및 삭제가 관련 테이블 간에 불일치를 초래하지 않는지 확인합니다. 섹션 13.1.20.5, "외래 키 제약 조건"을 참조하십시오.

표 15.1 InnoDB 스토리지 엔진 기능

기능	지원
B-트리 인덱스	예
백업/시점 복구(스토리지 엔진이 아닌 서버에서 구현됨)	예
클러스터 데이터베이스 지원	아니요
클러스터된 인덱스	예
압축 데이터	예
데이터 캐시	예
암호화된 데이터	예(암호화 기능을 통해 서버에서 구현되며, MySQL 5.7 이상에서는 미사용 데이터 암호화가 지원됩니다.)
외래 키 지원	예
전체 텍스트 검색 색인	예(전체 텍스트 인덱스에 대한 지원은 MySQL 5.6 이상에서 제공됩니다.)
지리공간 데이터 유형 지원	예
지리공간 인덱싱 지원	예(지리공간 인덱싱 지원은 MySQL 5.7 이상에서 사용할 수 있습니다.)
해시 인덱스	아니요(InnoDB는 적응형 해시 인덱스 기능을 위해 내부적으로 해시 인덱스를 활용합니다.)
인덱스 캐시	예
잠금 세부 수준	행

MVCC	예
복제 지원(스토리지 엔진이 아닌 서버에서 구현됩니다.)	예
저장 용량 제한	64TB
T-트리 인덱스	아니요
거래	예
데이터 사전 통계 업데이트	예

MySQL과 함께 제공되는 다른 스토리지 엔진과 `InnoDB`의 기능을 비교하려면 16장, *대체 스토리지 엔진의* 스토리지 엔진 *기능* 표를 참조하세요.

InnoDB 개선 사항 및 새로운 기능

`InnoDB` 개선 사항 및 새로운 기능에 대한 자세한 내용은 다음을 참조하세요:

- 1.3절 'MySQL 8.2의 새로운 기능'의 `InnoDB` 개선 사항 목록에 나와 있습니다.
- 릴리스 정보.

추가 InnoDB 정보 및 리소스

- `InnoDB` 관련 용어 및 정의는 MySQL 용어집을 참조하세요.
- `InnoDB` 스토리지 엔진 전용 포럼은 MySQL 포럼::InnoDB를 참조하세요.
- InnoDB는 MySQL과 동일한 GNU GPL 라이선스 버전 2(1991년 6월)에 따라 게시됩니다. MySQL 라이선스에 대한 자세한 내용은 http://www.mysql.com/company/legal/licensing/ 을 참조하세요.

15.1.1 InnoDB 테이블 사용의 이점

`InnoDB` 테이블에는 다음과 같은 이점이 있습니다:

- 하드웨어 또는 소프트웨어 문제로 인해 서버가 예기치 않게 종료된 경우, 당시 데이터베이스에서 어떤 일이 발생했는지와 관계없이 데이터베이스를 다시 시작한 후 특별한 조치를 취할 필요가 없습니다. `InnoDB` 충돌 복구는 충돌이 발생하기 전에 커밋된 변경 사항을 자동으로 마무리하고, 진행 중이지만 커밋되지 않은 변경 사항을 실행 취소하여 다시 시작하여 중단한 지점부터 계속할 수 있도록 합니다. 섹션 15.18.2, "InnoDB 복구"를 참조하세요.
- `InnoDB` 스토리지 엔진은 데이터에 액세스할 때 테이블 및 인덱스 데이터를 주 메모리에 캐싱하는 자체 버퍼 풀을 유지합니다. 자주 사용되는 데이터는 메모리에서 직접 처리됩니다. 이 캐시는 다양한 유형의 정보에 적용되며 처리 속도를 높여줍니다. 전용 데이터베이스 서버에서는 물리적 메모리의 최대 80%가 버퍼 풀에 할당되는 경우가 많습니다. 섹션 15.5.1, "버퍼 풀"을 참조하세요.
- 관련 데이터를 다른 테이블로 분할하는 경우 참조 무결성을 적용하는 외래 키를 설정할 수 있습니다. 섹션 13.1.20.5, "외래 키 제약 조건"을 참조하십시오.
- 디스크나 메모리에서 데이터가 손상된 경우, 체크섬 메커니즘은 데이터를 사용하기 전에 가짜 데이터를 경고합니다. `innodb_checksum_algorithm` 변수는 `InnoDB`에서 사용하는 체크섬 알고리즘을 정의합니다.
- 각 테이블에 적절한 기본 키 열을 사용하여 데이터베이스를 설계하면 해당 열과 관련된 작업이 자동으로 최적화됩니다. `WHERE` 절, `ORDER BY` 절, `GROUP BY` 절 및 조인 작업에서 기본 키 열을 참조하는 것이 매우 빠릅니다. 섹션 15.6.2.1, "클러스터 및 보조 인덱스"를 참조하세요.
- 삽입, 업데이트, 삭제는 변경 버퍼링이라는 자동 메커니즘에 의해 최적화됩니다. `InnoDB`는 동일한 테이블

에 대한 읽기 및 쓰기 액세스를 동시에 허용할 뿐만 아니라 변경된 데이터를 캐시하여 디스크 I/O를 간소화합니다. 섹션 15.5.2, "변경 버퍼"를 참조하십시오.

- 성능 이점은 장기 실행 쿼리가 있는 대규모 테이블에만 국한되지 않습니다. 테이블에서 동일한 행에 반복해서 액세스하는 경우, 적응형 해시 인덱스가 이러한 조회를 마치 해시 테이블에서 나온 것처럼 더욱 빠르게 수행할 수 있도록 대신합니다. 섹션 15.5.3, "적응형 해시 인덱스"를 참조하세요.
- 테이블 및 관련 인덱스를 압축할 수 있습니다. 섹션 15.9, "InnoDB 테이블 및 페이지 압축"을 참조하세요.
- 데이터를 암호화할 수 있습니다. 섹션 15.13, "InnoDB 저장 데이터 암호화"를 참조하세요.
- 인덱스를 생성 및 삭제하고 성능과 가용성에 훨씬 적은 영향을 미치면서 다른 DDL 작업을 수행할 수 있습니다. 섹션 15.12.1, "온라인 DDL 작업"을 참조하세요.
- 테이블별 파일 테이블스페이스를 잘라내는 것은 매우 빠르며 운영 체제에서 `InnoDB`만이 아니라 재사용할 수 있는 디스크 공간을 확보할 수 있습니다. 15.6.3.2절. "테이블별 파일 테이블 스페이스"를 참조하십시오.

  - 테이블 데이터의 저장소 레이아웃은 `BLOB` 및 긴 텍스트 필드에 더 효율적인 동적 행 형식을 사용합니다. 섹션 15.10, "InnoDB 행 형식"을 참조하세요.
  - `INFORMATION_SCHEMA`를 쿼리하여 스토리지 엔진의 내부 작동을 모니터링할 수 있습니다. 테이블. 섹션 15.15, "InnoDB 정보_화학 테이블"을 참조하십시오.
- 성능 스키마 테이블을 쿼리하여 스토리지 엔진의 성능 세부 정보를 모니터링할 수 있습니다. 섹션 15.16, "MySQL 성능 스키마와 InnoDB 통합"을 참조하세요.
- 동일한 문 내에서도 `InnoDB` 테이블을 다른 MySQL 스토리지 엔진의 테이블과 혼합할 수 있습니다. 예를 들어, 조인 작업을 사용하여 단일 쿼리에서 `InnoDB` 테이블과 `MEMORY` 테이블의 데이터를 결합할 수 있습니다.
- InnoDB는 대용량 데이터를 처리할 때 CPU 효율성과 성능을 극대화하도록 설계되었습니다.
- `InnoDB` 테이블은 파일 크기가 2GB로 제한되는 운영 체제에서도 대량의 데이터를 처리할 수 있습니다.

MySQL 서버 및 애플리케이션 코드에 적용할 수 있는 `InnoDB` 관련 튜닝 기술에 대해서는 8.5절. 'InnoDB 테이블에 맞게 최적화'를 참조하세요.

15.1.2 InnoDB 테이블 모범 사례

이 섹션에서는 `InnoDB` 테이블을 사용할 때의 모범 사례를 설명합니다.

- 가장 자주 쿼리되는 열을 사용하여 모든 테이블에 기본 키를 지정하거나, 명확한 기본 키가 없는 경우 자동 증가 값을 지정합니다.
- 여러 테이블에서 데이터를 가져올 때마다 해당 테이블의 동일한 ID 값을 기반으로 조인을 사용합니다. 빠른 조인 성능을 위해 조인 열에 외래 키를 정의하고 각 테이블에서 동일한 데이터 유형으로 해당 열을 선언합니다. 외래 키를 추가하면 참조된 열이 인덱싱되므로 성능이 향상될 수 있습니다. 또한 외래 키는 영향을 받는 모든 테이블에 삭제 및 업데이트를 전파하고, 상위 테이블에 해당 ID가 없는 경우 하위 테이블에 데이터를 삽입하지 못하도록 합니다.
- 자동 커밋을 끕니다. 초당 수백 번 커밋하면 저장 장치의 쓰기 속도에 따라 성능에 제한이 생깁니다.
- 관련 DML 작업 집합을 `START TRANSACTION` 및 `COMMIT` 문으로 괄호로 묶어 트랜잭션으로 그룹화하세요. 커밋을 너무 자주 실행하고 싶지는 않지만, 커밋 없이 몇 시간 동안 실행되는 `INSERT`, `UPDATE` 또는 `DELETE` 문을 대량으로 실행하고 싶지도 않습니다.
- `LOCK TABLES` 문을 사용하지 마세요. `InnoDB`는 안정성이나 고성능을 희생하지 않고도 동일한 테이블을 읽고 쓰는 여러 세션을 한 번에 처리할 수 있습니다. 행 집합에 대한 쓰기 전용 액세스 권한을 얻으려면 `SELECT ... FOR UPDATE` 구문을 사용하여 업데이트하려는 행만 잠그면 됩니다.
- 테이블의 데이터와 인덱스를 시스템 테이블 스페이스 대신 별도의 파일에 넣으려면 `innodb_file_per_table` 변수를 활성화하거나 일반 테이블 스페이스를 사용합니다.

`innodb_file_per_table` 변수는 기본적으로 활성화되어 있습니다.

- 데이터 및 액세스 패턴이 `InnoDB` 테이블 또는 페이지 압축 기능의 이점을 누릴 수 있는지 평가하세요. 읽기/쓰기 기능의 저하 없이 `InnoDB` 테이블을 압축할 수 있습니다.
- 사용하지 않으려는 스토리지 엔진으로 테이블이 생성되지 않도록 하려면 `--sql_mode=NO_ENGINE_SUBSTITUTION` 옵션으로 서버를 실행합니다.

15.1.3 InnoDB가 기본 스토리지 엔진인지 확인

`SHOW ENGINES` 문을 실행하여 사용 가능한 MySQL 스토리지 엔진을 확인합니다. 기본값을 찾습니다. 지원 열에 입력합니다.

```
mysql> SHOW ENGINES;
```

또는 정보 스키마 엔진 테이블을 쿼리합니다.

```
mysql> SELECT * FROM INFORMATION_SCHEMA.ENGINES;
```

15.1.4 InnoDB를 사용한 테스트 및 벤치마킹

InnoDB가 기본 스토리지 엔진이 아닌 경우, 명령줄에 `--default-storage-engine=InnoDB`를 정의하거나 MySQL 서버 옵션 파일의 `[mysqld]` 섹션에 `default-storage-engine=innodb`를 정의하여 서버를 다시 시작하면 데이터베이스 서버 및 애플리케이션이 `InnoDB`에서 올바르게 작동하는지 확인할 수 있습니다.

기본 저장소 엔진을 변경하면 새로 만든 테이블에만 영향을 미치므로 애플리케이션 설치 및 설정 단계를 실행하여 모든 것이 제대로 설치되었는지 확인한 다음 애플리케이션 기능을 실행하여 데이터 로드, 편집 및 쿼리 기능이 작동하는지 확인합니다. 테이블이 다음에 의존하는 경우
를 다른 스토리지 엔진에만 해당하는 기능으로 설정하면 오류가 발생합니다. 이 경우
엔진=다른_엔진_이름 절을추가하여 오류를 방지할 수 있습니다.

스토리지 엔진에 대해 신중하게 결정하지 않았고 `InnoDB`를 사용하여 생성한 특정 테이블이 어떻게 작동하는지 미리 보려면 각 테이블에 대해 `ALTER TABLE table_name ENGINE=InnoDB;` 명령을 실행하세요. 또는 원본 테이블을 방해하지 않고 테스트 쿼리 및 기타 문을 실행하려면 복사본을 만드세요:

```
테이블 생성 ... ENGINE=InnoDB AS SELECT * FROM other_engine_table;
```

실제 워크로드에서 전체 애플리케이션의 성능을 평가하려면 최신 MySQL 서버를 설치하고 벤치마크를 실행하세요.

설치부터 사용량이 많은 경우, 서버 재시작까지 전체 애플리케이션 수명 주기를 테스트합니다. 데이터베이스가 사용 중일 때 서버 프로세스를 종료하여 정전을 시뮬레이션하고 서버를 다시 시작할 때 데이터가 성공적으로 복구되는지 확인합니다.

특히 소스 서버와 복제본에서 서로 다른 MySQL 버전 및 옵션을 사용하는 경우 복제 구성을 테스트합니다.

15.2 InnoDB와 ACID 모델

ACID 모델은 비즈니스 데이터와 미션 크리티컬 애플리케이션에 중요한 신뢰성 측면을 강조하는 일련의 데이터베이스 설계 원칙입니다. MySQL에는 소프트웨어 충돌 및 하드웨어 오작동과 같은 예외적인 조건으로 인해 데이터가 손상되지 않고 결과가 왜곡되지 않도록 ACID 모델을 밀접하게 준수하는 `InnoDB` 스토리지 엔진과 같은 구성 요소가 포함되어 있습니다. ACID 호환 기능을 사용하면 일관성 검사 및 충돌 복구 메커니즘을 다시 개발할 필요가 없습니다. 추가적인 소프트웨어 보호 장치, 매우 안정적인 하드웨어 또는 소량의 데이터 손실이나 불일치를 견딜 수 있는 애플리케이션이 있는 경우, MySQL 설정을 조정하여 성능 또는 처리량 향상을 위해 일부 ACID 안정성과 맞바꿀 수 있습니다.

다음 섹션에서는 MySQL 기능, 특히 `InnoDB` 스토리지 엔진이 ACID 모델의 카테고리와 상호 작용하는 방식에 대해 설명합니다:

- **A**: 원자성.
- **C**: 일관성.
- **I:**: 격리.
- **D**: 내구성.

원자성

ACID 모델의 **원자성** 측면은 주로 `InnoDB` 트랜잭션과 관련이 있습니다. 관련 MySQL 기능은 다음과 같습니다:

- 자동 커밋 설정입니다.
- COMMIT 문입니다.
- 롤백 문입니다.

일관성

ACID 모델의 **일관성** 측면은 주로 데이터를 충돌로부터 보호하기 위한 내부 InnoDB 처리와 관련이 있습니다. 관련 MySQL 기능은 다음과 같습니다:

- InnoDB 이중 쓰기 버퍼. 섹션 15.6.4, "이중 쓰기 버퍼"를 참조하십시오.
- InnoDB 충돌 복구. InnoDB 충돌 복구를 참조하십시오.

격리

ACID 모델의 **격리** 측면은 주로 InnoDB 트랜잭션, 특히 각 트랜잭션에 적용되는 격리 수준과 관련이 있습니다. 관련 MySQL 기능은 다음과 같습니다:

- 자동 커밋 설정입니다.
- 트랜잭션 격리 수준 및 SET TRANSACTION 문. 섹션 15.7.2.1, "트랜잭션 격리 수준"을 참조하십시오.
- InnoDB 잠금에 대한 로우 레벨 세부 정보입니다. 자세한 내용은 INFORMATION_SCHEMA 테이블 (15.15.2절, "InnoDB INFORMATION_SCHEMA 트랜잭션 및 잠금 정보" 참조) 및 성능 스키마 data_locks 및 data_lock_waits 테이블에서 볼 수 있습니다.

내구성

ACID 모델의 **내구성** 측면은 특정 하드웨어 구성과 상호 작용하는 MySQL 소프트웨어 기능과 관련이 있습니다. CPU, 네트워크, 저장 장치의 성능에 따라 많은 가능성이 있기 때문에 이 측면은 구체적인 가이드라인을 제공하기 가장 복잡합니다. (그리고 이러한 가이드라인은 "새 하드웨어 구입"의 형태를 취할 수도 있습니다.) 관련 MySQL 기능은 다음과 같습니다:

- InnoDB 이중 쓰기 버퍼. 섹션 15.6.4, "이중 쓰기 버퍼"를 참조하십시오.
- innodb_flush_log_at_trx_commit 변수입니다.
- sync_binlog 변수입니다.
- innodb_file_per_table 변수입니다.
- 디스크 드라이브, SSD 또는 RAID 어레이와 같은 저장 장치의 쓰기 버퍼입니다.

- 저장 장치의 배터리 지원 캐시입니다.
- MySQL을 실행하는 데 사용되는 운영 체제, 특히 `fsync()` 시스템 호출에 대한 지원.
- MySQL 서버를 실행하고 MySQL 데이터를 저장하는 모든 컴퓨터 서버 및 저장 장치의 전력을 보호하는 무정전 전원 공급 장치(UPS)입니다.
- 백업 빈도 및 유형, 백업 보존 기간과 같은 백업 전략.
- 분산 또는 호스팅된 데이터 애플리케이션의 경우, MySQL 서버용 하드웨어가 위치한 데이터 센터의 특정 특성 및 데이터 센터 간의 네트워크 연결이 중요합니다.

15.3 InnoDB 다중 버전 관리

InnoDB는 다중 버전 스토리지 엔진입니다. 동시성 및 롤백과 같은 트랜잭션 기능을 지원하기 위해 변경된 행의 이전 버전에 대한 정보를 유지합니다. 이 정보는 실행 취소에 저장됩니다.

테이블스페이스를 롤백 세그먼트라는 데이터 구조에 저장합니다. 섹션 15.6.3.4, "테이블 스페이스 실행 취소"를 참조하십시오. InnoDB는 롤백 세그먼트의 정보를 사용하여 트랜잭션 롤백에 필요한 실행 취소 작업을 수행합니다. 또한 이 정보를 사용하여 일관된 읽기를 위해 행의 이전 버전을 빌드합니다. 섹션 15.7.2.3, "일관된 비잠금 읽기"를 참조하십시오.

내부적으로 InnoDB는 데이터베이스에 저장된 각 행에 세 개의 필드를 추가합니다:

- 6바이트 DB_TRX_ID 필드는 행을 삽입하거나 업데이트한 마지막 트랜잭션의 트랜잭션 식별자를 나타냅니다. 또한 삭제는 내부적으로 업데이트로 처리되며, 행의 특수 비트가 삭제된 것으로 표시되도록 설정됩니다.

- 롤 포인터라고 하는 7바이트 DB_ROLL_PTR 필드입니다. 롤 포인터는 롤백 세그먼트에 기록된 실행 취소 로그 레코드를 가리킵니다. 행이 업데이트된 경우 실행 취소 로그 레코드에는 행이 업데이트되기 전의 내용을 다시 작성하는 데 필요한 정보가 포함됩니다.

- 6바이트 DB_ROW_ID 필드에는 새 행이 삽입될 때 단조롭게 증가하는 행 ID가 포함됩니다. InnoDB가 클러스터된 인덱스를 자동으로 생성하는 경우 인덱스에 행 ID 값이 포함됩니다. 그렇지 않으면 DB_ROW_ID 열은 인덱스에 나타나지 않습니다.

롤백 세그먼트의 실행 취소 로그는 삽입 실행 취소 로그와 업데이트 실행 취소 로그로 나뉩니다. 삽입 실행 취소 로그는 트랜잭션 롤백에만 필요하며 트랜잭션이 커밋되는 즉시 삭제할 수 있습니다.
업데이트 실행 취소 로그는 일관된 읽기에도 사용되지만, 일관된 읽기에서 데이터베이스 행의 이전 버전을 빌드하기 위해 업데이트 실행 취소 로그의 정보가 필요할 수 있는 스냅샷을 InnoDB에 할당된 트랜잭션이 없는 경우에만 삭제할 수 있습니다. 실행 취소 로그에 대한 자세한 내용은 섹션 15.6.6, "실행 취소 로그"를 참조하십시오.

일관된 읽기만 발행하는 트랜잭션을 포함하여 정기적으로 트랜잭션을 커밋하는 것이 좋습니다. 그렇지 않으면 InnoDB가 업데이트 실행 취소 로그의 데이터를 삭제할 수 없으며 롤백 세그먼트가 너무 커져서 해당 세그먼트가 있는 실행 취소 테이블스페이스를 가득 채울 수 있습니다. 실행 취소 테이블스페이스 관리에 대한 자세한 내용은 15.6.3.4절. "테이블스페이스 실행 취소"를 참조하십시오.

롤백 세그먼트에 있는 실행 취소 로그 레코드의 물리적 크기는 일반적으로 해당 삽입 또는 업데이트된 행보다 작습니다. 이 정보를 사용하여 롤백 세그먼트에 필요한 공간을 계산할 수 있습니다.

InnoDB 다중 버전 관리 체계에서는 SQL 문으로 행을 삭제할 때 데이터베이스에서 행이 물리적으로 즉시 제거되지 않습니다. InnoDB는 삭제를 위해 기록된 업데이트 실행 취소 로그 레코드를 폐기할 때만 해당 행과 해당 인덱스 레코드를 물리적으로 제거합니다. 이 제거 작업을 퍼지라고 하며, 일반적으로 삭제를 수행한 SQL 문과 동일한 순서로 시간이 걸리는 매우 빠른 작업입니다.

테이블에서 거의 동일한 속도로 작은 단위로 행을 삽입하고 삭제하는 경우, 제거 스레드가 지연되기 시작하고 모든 "죽은" 행으로 인해 테이블이 점점 더 커져 모든 것이 디스크에 묶여 매우 느려질 수 있습니다. 이러한 경우 새 행 작업을 스로틀하고 innodb_max_purge_lag 시스템 변수를 조정하여 제거 스레드에 더 많은 리

소스를 할당합니다. 자세한 내용은 섹션 15.8.9, "퍼지 구성"을 참조하세요.

다중 버전 및 보조 인덱스

`InnoDB` 다중 버전 동시성 제어(MVCC)는 보조 인덱스를 클러스터된 인덱스와 다르게 처리합니다. 클러스터된 인덱스의 레코드는 제자리에서 업데이트되며, 숨겨진 시스템 열은 이전 버전의 레코드를 재구성할 수 있는 실행 취소된 로그 항목을 가리킵니다. 클러스터된 인덱스 레코드와 달리 보조 인덱스 레코드에는 숨겨진 시스템 열이 포함되지 않으며 제자리에서 업데이트되지도 않습니다.

보조 인덱스 열이 업데이트되면 이전 보조 인덱스 레코드가 삭제 표시되고, 새 레코드가 삽입되며, 삭제 표시된 레코드는 결국 제거됩니다. 보조 인덱스 레코드가 삭제 표시되거나 보조 인덱스 페이지가 최신 트랜잭션에 의해 업데이트되면 `InnoDB`는 클러스터된 인덱스에서 데이터베이스 레코드를 조회합니다. 클러스터된 인덱스에서 레코드의 `DB_TRX_ID`를 확인하고, 읽기 트랜잭션이 시작된 후 레코드가 수정된 경우 실행 취소 로그에서 레코드의 올바른 버전을 검색합니다.

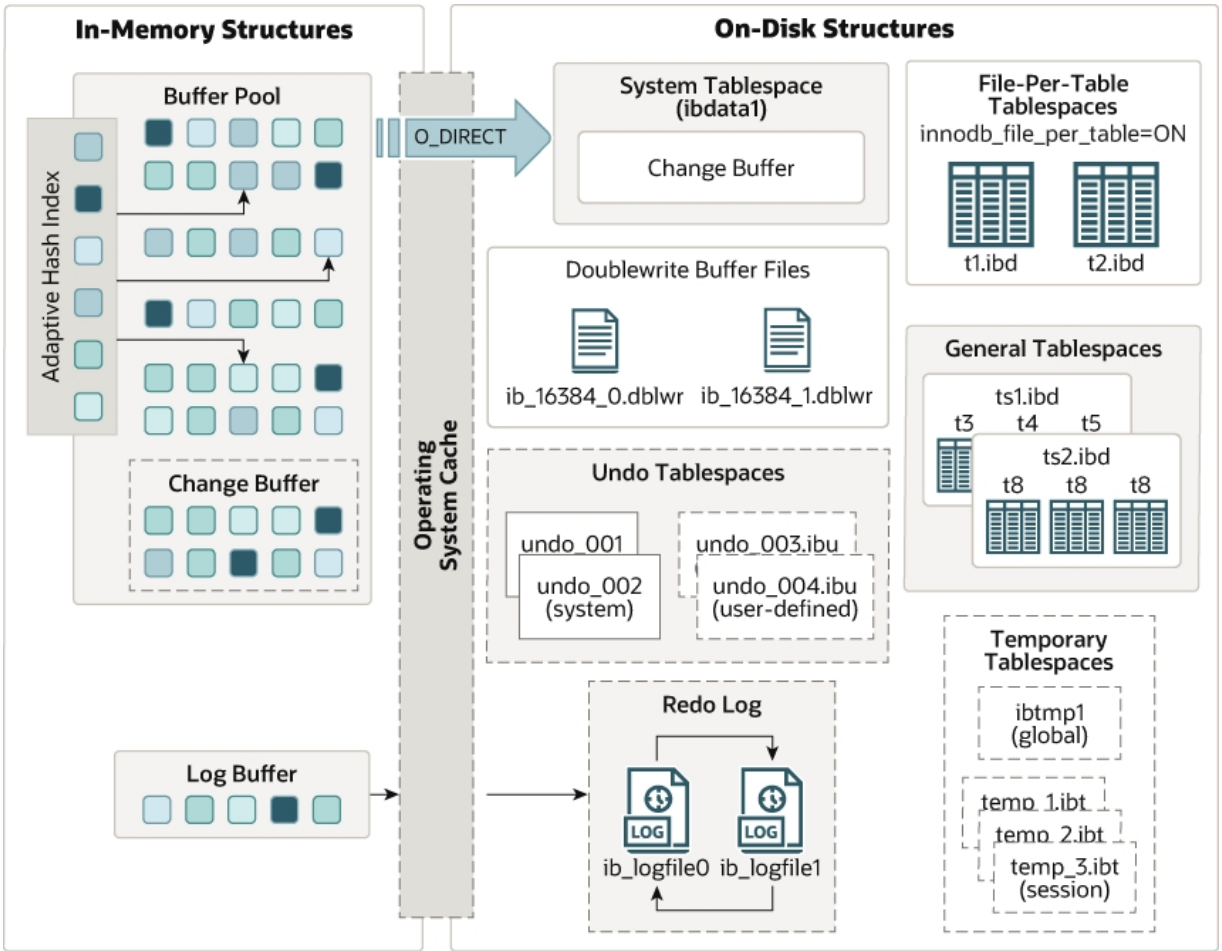
보조 인덱스 레코드가 삭제되도록 표시되거나 보조 인덱스 페이지가 최신 트랜잭션에 의해 업데이트되는 경우, 커버링 인덱스 기법은 사용되지 않습니다. 인덱스 구조에서 값을 반환하는 대신 `InnoDB`는 클러스터된 인덱스에서 레코드를 조회합니다.

그러나 인덱스 조건 푸시다운(ICP) 최적화가 활성화되어 있고 인덱스의 필드만 사용하여 `WHERE` 조건의 일부를 평가할 수 있는 경우 MySQL 서버는 여전히 이 부분을 인덱스를 사용하여 평가되는 스토리지 엔진으로 푸시다운합니다. 일치하는 레코드가 발견되지 않으면 클러스터된 인덱스 조회가 피됩니다. 삭제 표시된 레코드 중에도 일치하는 레코드가 발견되면 `InnoDB`는 클러스터된 인덱스에서 해당 레코드를 조회합니다.

15.4 InnoDB 아키텍처

다음 다이어그램은 `InnoDB` 스토리지 엔진 아키텍처를 구성하는 인메모리 및 온디스크 구조를 보여줍니다. 각 구조에 대한 자세한 내용은 섹션 15.5, "InnoDB 인메모리 구조" 및 섹션 15.6, "InnoDB 온디스크 구조"를 참조하십시오.

그림 15.1 InnoDB 아키텍처



15.5 InnoDB 인메모리 구조

이 섹션에서는 `InnoDB` 인메모리 구조 및 관련 주제에 대해 설명합니다.

15.5.1 버퍼 풀

버퍼 풀은 테이블 및 인덱스 데이터가 액세스될 때 InnoDB가 캐시하는 주 메모리 영역입니다. 버퍼 풀을 사용하면 자주 사용하는 데이터를 메모리에서 직접 액세스할 수 있으므로 처리 속도가 빨라집니다. 전용 서버에서는 물리적 메모리의 최대 80%가 버퍼 풀에 할당되는 경우가 많습니다.

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 가장 최근에 사용된 데이터(LRU) 알고리즘의 변형을 사용하여 캐시에서 에이징됩니다.

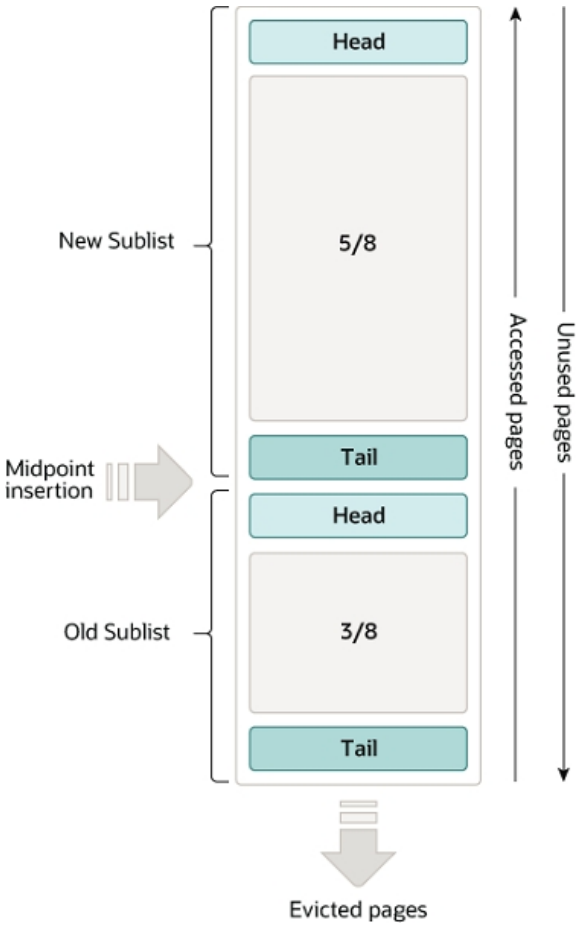
버퍼 풀을 활용하여 자주 액세스하는 데이터를 메모리에 보관하는 방법을 아는 것은 MySQL 튜닝의 중요한 측면입니다.

버퍼 풀 LRU 알고리즘

버퍼 풀은 LRU 알고리즘의 변형을 사용하여 목록으로 관리됩니다. 버퍼 풀에 새 페이지를 추가할 공간이 필요하면 최근에 가장 적게 사용된 페이지가 제거되고 새 페이지가 목록의 중간에 추가됩니다. 이 중간 지점 삽입 전략은 목록을 두 개의 하위 목록으로 취급합니다:

- 머리글에는 최근에 액세스한 새('젊은') 페이지의 하위 목록이 표시됩니다.
- 꼬리에는 최근에 액세스한 횟수가 적은 오래된 페이지의 하위 목록이 표시됩니다.

그림 15.2 버퍼 풀 목록



알고리즘은 자주 사용하는 페이지를 새 하위 목록에 보관합니다. 이전 하위 목록에는 자주 사용하지 않는 페이지가 포함되며, 이러한 페이지는 퇴출 후보가 됩니다.

기본적으로 알고리즘은 다음과 같이 작동합니다:

- 버퍼 풀의 3/8은 이전 하위 목록에 할당됩니다.
- 목록의 중간 지점은 새 하위 목록의 꼬리가 이전 하위 목록의 머리와 만나는 경계입니다.

- `InnoDB`는 페이지를 버퍼 풀로 읽을 때 처음에는 중간 지점(이전 하위 목록의 헤드)에 페이지를 삽입합니다. 페이지가 읽혀질 수 있는 이유는 SQL 쿼리와 같은 사용자 시작 작업에 필요하기 때문이거나 `InnoDB`에서 자동으로 수행되는 미리 읽기 작업의 일부이기 때문입니다.
- 이전 하위 목록에 있는 페이지에 액세스하면 해당 페이지가 '젊은' 상태가 되어 새 하위 목록의 맨 위로 이동합니다. 사용자가 시작한 작업으로 인해 페이지를 읽은 경우 첫 번째 액세스가 즉시 발생하고 해당 페이지가 젊은 상태가 됩니다. 미리 읽기 작업으로 인해 페이지를 읽은 경우 첫 번째 액세스가 즉시 발생하지 않으며 페이지가 제거되기 전에 전혀 발생하지 않을 수 있습니다.
- 데이터베이스가 작동함에 따라 액세스되지 않는 버퍼 풀의 페이지는 목록의 맨 끝으로 이동하여 '노화'됩니다. 새 하위 목록과 이전 하위 목록의 페이지는 다른 페이지가 새로 만들어질 때 모두 노화됩니다. 이전 하위 목록의 페이지도 중간 지점에 페이지가 삽입됨에 따라 노화됩니다. 결국 사용되지 않는 페이지는 이전 하위 목록의 맨 끝에 도달하여 퇴출됩니다.

기본적으로 쿼리에서 읽은 페이지는 즉시 새 하위 목록으로 이동되므로 버퍼 풀에 더 오래 유지됩니다. 예를 들어, `mysqldump` 작업 또는 `WHERE` 절이 없는 `SELECT` 문에 대해 수행되는 테이블 스캔은 새 데이터를 다시 사용하지 않더라도 대량의 데이터를 버퍼 풀로 가져와서 같은 양의 이전 데이터를 제거할 수 있습니다. 마찬가지로, 미리 읽기 백그라운드 스레드에 의해 로드되고 한 번만 액세스되는 페이지는 새 목록의 맨 위로 이동됩니다. 이러한 상황에서는 자주 사용하는 페이지가 이전 하위 목록으로 밀려나 퇴출 대상이 될 수 있습니다. 이 동작을 최적화하는 방법에 대한 자세한 내용은 15.8.3.3절.
버퍼 풀 스캔 저항성" 및 섹션 15.8.3.4, "InnoDB 버퍼 풀 프리페칭 구성(읽기 앞서)"을 참조하십시오.

`InnoDB` 표준 모니터 출력에는 버퍼 풀 및 메모리 섹션에 버퍼 풀 LRU 알고리즘 작동에 관한 여러 필드가 포함되어 있습니다. 자세한 내용은 InnoDB 표준 모니터를 사용하여 버퍼 풀 모니터링하기를 참조하세요.

버퍼 풀 구성

버퍼 풀의 다양한 측면을 구성하여 성능을 개선할 수 있습니다.

- 버퍼 풀의 크기를 가능한 한 큰 값으로 설정하여 서버의 다른 프로세스가 과도한 페이징 없이 실행될 수 있도록 충분한 메모리를 남겨두는 것이 이상적입니다. 버퍼 풀이 클수록 `InnoDB`는 인메모리 데이터베이스처럼 작동하여 디스크에서 데이터를 한 번 읽은 다음 후속 읽기 중에 메모리에서 데이터에 액세스합니다. 섹션 15.8.3.1, "InnoDB 버퍼 풀 크기 구성"을 참조하세요.
- 메모리가 충분한 64비트 시스템에서는 버퍼 풀을 여러 부분으로 분할하여 동시 작업 간에 메모리 구조에 대한 경합을 최소화할 수 있습니다. 자세한 내용은 섹션 15.8.3.2, "여러 버퍼 풀 인스턴스 구성"을 참조하십시오.
- 자주 액세스하지 않는 대량의 데이터를 버퍼 풀로 가져오는 작업으로 인해 활동이 갑자기 급증하더라도 자주 액세스하는 데이터를 메모리에 유지할 수 있습니다. 자세한 내용은 섹션 15.8.3.3, "버퍼 풀 스캔 방지 기능 만들기"를 참조하십시오.

- 임박한 필요를 예상하여 페이지를 버퍼 풀에 비동기적으로 미리 가져오기 위해 읽기 미리 가져오기 요청을 수행하는 방법과 시기를 제어할 수 있습니다. 자세한 내용은 섹션 15.8.3.4, "InnoDB 버퍼 풀 프리페칭 구성(읽기 앞서)"을 참조하세요.
- 백그라운드 플러시가 발생하는 시기와 플러시 속도를 워크로드에 따라 동적으로 조정할지 여부를 제어할 수 있습니다. 자세한 내용은 섹션 15.8.3.5, "버퍼 풀 플러시 구성"을 참조하세요.
- 서버 재시작 후 긴 워밍업 기간을 피하기 위해 InnoDB가 현재 버퍼 풀 상태를 보존하는 방법을 구성할 수 있습니다. 자세한 내용은 섹션 15.8.3.6, "버퍼 풀 상태 저장 및 복원"을 참조하세요.

InnoDB 표준 모니터를 사용하여 버퍼 풀 모니터링하기

엔진 엔진 상태 표시를 사용하여 액세스할 수 있는 InnoDB 표준 모니터 출력은 버퍼 풀 작동에 관한 메트릭을 제공합니다. 버퍼 풀 메트릭은 InnoDB 표준 모니터 출력의 버퍼 풀 및 메모리 섹션에 있습니다:

```
----------------------
버퍼 풀 및 메모리
----------------------
할당된 총 대용량 메모리 2198863872 할당된 사전
메모리 776332 버퍼 풀     크기131072
여유 버퍼             124908
데이터베이스           페이
지5720 이전 데이터베이스 페
이지 2071 수정된 데이터베이
스 페이지 910 보류 중인 읽
기 0
쓰기 보류 중입니다: LRU 0, 플러시 목록 0, 단일 페이지 0 페이지가
영 4, 영이 아님 0
0.10 영/초, 0.00 비영/초
페이지 읽기 197, 생성 5523, 쓰기 5060
0.00 읽기/초, 190.89 생성/초, 244.94 쓰기/초
버퍼 풀 적중률 1000 / 1000, 영 만들기 비율 0 / 1000 없음
0 / 1000
페이지 앞 읽기 0.00초, 액세스 권한 없이 퇴거 0.00초, 무작위 앞 읽기 0.00초
LRU len: 5720, unzip_LRU len: 0
I/O 합계[0]:cur[0], 압축 해제 합계[0]:cur[0]
```

다음 표에서는 InnoDB 표준 모니터에서 보고하는 버퍼 풀 메트릭에 대해 설명합니다.

InnoDB 표준 모니터 출력에 제공된 초당 평균은 InnoDB 표준 모니터 출력이 마지막으로 인쇄된 후 경과된 시간을 기준으로 합니다.

표 15.2 InnoDB 버퍼 풀 메트릭

이름	설명
할당된 총 메모리	버퍼 풀에 할당된 총 메모리(바이트)입니다.
할당된 사전 메모리	InnoDB 데이터 사전에 할당된 총 메모리(바이트)입니다.
버퍼 풀 크기	버퍼 풀에 할당된 페이지 단위의 총 크기입니다.
여유 버퍼	버퍼 풀 사용 가능 목록의 총 크기(페이지)입니다.
데이터베이스 페이지	버퍼 풀 LRU 목록의 총 크기(페이지)입니다.
이전 데이터베이스 페이지	버퍼 풀 이전 LRU 하위 목록의 페이지 단위 총 크기입니다.
수정된 DB 페이지	버퍼 풀에서 수정된 현재 페이지 수입니다.
보류 중인 읽기	버퍼 풀에서 읽기를 기다리는 버퍼 풀 페이지 수입니다.
보류 중인 쓰기 LRU	버퍼 풀 내의 오래된 더티 페이지 수로, LRU 목록의 맨 아래에서 기록합니다.

보류 중인 쓰기 플러시 목록	체크포인트하는 동안 플러시할 버퍼 풀 페이지 수입니다.
보류 중 단일 페이지 쓰기	버퍼 풀 내에서 보류 중인 독립 페이지 쓰기 수입니다.
젊게 만든 페이지	버퍼 풀 LRU 목록에서 영 페이지가 된 총 페이지 수('새' 페이지의 하위 목록 머리로 이동됨)입니다.
젊지 않은 페이지	버퍼 풀 LRU 목록에서 젊어지지 않은 페이지의 총 수(젊어지지 않고 "오래된" 하위 목록에 남아 있는 페이지)입니다.
youngs/s	버퍼 풀 LRU 목록의 이전 페이지에 대한 초당 평균 액세스로 인해 발생한 결과입니다.

이름	설명
	페이지를 젊게 만들기. 다음 참고 사항을 참조하세요. 자세한 내용은 이 표를 참조하세요.
비청소년/S	페이지를 젊게 만들지 않은 버퍼 풀 LRU 목록의 오래된 페이지에 대한 초당 평균 액세스 횟수입니다. 자세한 내용은 이 표 뒤에 나오는 참고 사항을 참조하세요.
읽은 페이지	버퍼 풀에서 읽은 총 페이지 수입니다.
생성된 페이지	버퍼 풀 내에서 생성된 총 페이지 수입니다.
작성된 페이지	버퍼 풀에서 기록된 총 페이지 수입니다.
읽기/초	초당 평균 버퍼 풀 페이지 읽기 횟수입니다.
생성/s	초당 생성되는 버퍼 풀 페이지의 평균 개수입니다.
쓰기/초	초당 평균 버퍼 풀 페이지 쓰기 횟수입니다.
버퍼 풀 적중률	버퍼 풀에서 읽은 페이지와 디스크 스토리지에서 읽은 페이지의 버퍼 풀 페이지 적중률입니다.
영 메이킹 비율	페이지 접속으로 인해 페이지가 젊어지는 평균 히트율입니다. 자세한 내용은 이 표 뒤에 나오는 참고 사항을 참조하세요.
아니요(영 메이킹 비율)	페이지 액세스가 페이지를 젊게 만들지 않은 평균 히트율입니다. 자세한 내용은 이 표 뒤에 나오는 참고 사항을 참조하세요.
미리 읽은 페이지	미리 읽기 작업의 초당 평균입니다.
액세스 권한 없이 퇴거된 페이지	버퍼 풀에서 액세스하지 않고 퇴거된 페이지의 초당 평균입니다.
무작위로 미리 읽기	무작위 미리 읽기 작업의 초당 평균입니다.
LRU len	버퍼 풀 LRU 목록의 총 크기(페이지)입니다.
unzip_LRU len	버퍼 풀 unzip_LRU 목록의 길이(페이지 단위)입니다.
I/O 합계	액세스한 버퍼 풀 LRU 목록 페이지의 총 개수입니다.
I/O cur	현재 간격으로 액세스한 버퍼 풀 LRU 목록 페이지의 총 개수입니다.
I/O 압축 해제 합계	압축 해제된 버퍼 풀 unzip_LRU 목록 페이지의 총 개수입니다.

I/O 압축 해제 cur	현재 간격으로 압축 해제된 버퍼 풀 unzip_LRU 목록 페이지의 총 개수입니다.

참고:

- 영스/초 지표는 오래된 페이지에만 적용됩니다. 페이지 액세스 횟수를 기반으로 합니다. 특정 페이지에 대해 여러 번의 액세스가 있을 수 있으며, 이 모든 액세스가 계산됩니다. 대규모 스캔이 발생하지 않을 때 영스/초 값이 매우 낮게 표시되면 지연 시간을 줄이거나 이전 하위 목록에 사용되는 버퍼 풀의 비율을 높이는 것이 좋습니다. 비율 높이기

를 사용하면 이전 하위 목록을 더 크게 만들어 해당 하위 목록의 페이지가 꼬리 부분으로 이동하는 데 시간이 더 오래 걸리므로 해당 페이지가 다시 액세스되어 젊어질 가능성이 높아집니다. 15.8.3.3절. "버퍼 풀 스캔에 저항하도록 만들기"를 참조하세요.

- 비영/초 지표는 오래된 페이지에만 적용됩니다. 페이지 액세스 횟수를 기반으로 합니다. 특정 페이지에 대해 여러 번의 액세스가 있을 수 있으며, 이 모든 액세스가 계산됩니다. 대용량 테이블 스캔을 수행할 때 비영값이 더 높게 나타나지 않고 영값이 더 높게 나타나면 지연 값을 늘리세요. 섹션 15.8.3.3, "버퍼 풀 스캔 내성 만들기"를 참조하십시오.

- 영 메이킹 비율은 이전 하위 목록에 있는 페이지에 대한 액세스뿐만 아니라 모든 버퍼 풀 페이지 액세스를 설명합니다. 영 메이킹 비율과 비율은 일반적으로 전체 버퍼 풀 적중률에 합산되지 않습니다. 이전 하위 목록의 페이지 히트는 페이지를 새 하위 목록으로 이동시키지만, 새 하위 목록의 페이지 히트는 페이지가 헤드에서 일정 거리에 있는 경우에만 페이지를 목록의 헤드로 이동시킵니다.

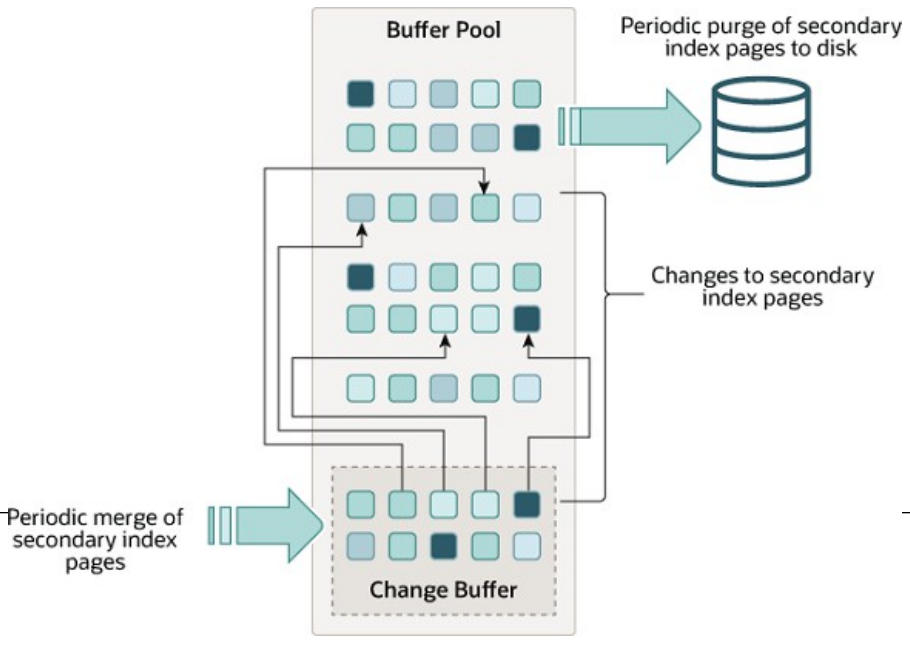
- not(젊은 만들기 비율)은 innodb_old_blocks_time에 정의된 지연이 충족되지 않거나 새 하위 목록의 페이지 히트로 인해 페이지가 헤드로 이동되지 않아 페이지 액세스가 페이지를 젊은 상태로 만들지 못한 평균 히트율입니다. 이 비율은 이전 하위 목록에 있는 페이지에 대한 액세스뿐만 아니라 모든 버퍼 풀 페이지 액세스를 설명합니다.

버퍼 풀 서버 상태 변수와 INNODB_BUFFER_POOL_STATS 테이블은 InnoDB 표준 모니터 출력에 있는 것과 동일한 많은 버퍼 풀 메트릭을 제공합니다. 자세한 내용은 예제 15.10, "INNODB_BUFFER_POOL_STATS 테이블 쿼리"를 참조하십시오.

15.5.2 버퍼 변경

변경 버퍼는 해당 페이지가 버퍼 풀에 없을 때 보조 인덱스 페이지의 변경 내용을 캐시하는 특수 데이터 구조입니다. 삽입, 업데이트 또는 삭제 작업(DML)으로 인해 발생할 수 있는 버퍼링된 변경 사항은 나중에 다른 읽기 작업에 의해 페이지가 버퍼 풀에 로드될 때 병합됩니다.

그림 15.3 버퍼 변경



클러스터된 인덱스와 달리 보조 인덱스는 일반적으로 고유하지 않으며, 보조 인덱스에 대한 삽입은 비교적 무작위적인 순서로 이루어집니다. 마찬가지로 삭제 및 업데이트는 인덱스 트리에 인접하지 않은 보조 인덱스 페이지에 영향을 미칠 수 있습니다. 나중에 다른 작업을 통해 영향을 받는 페이지를 버퍼 풀로 읽을 때 캐시된 변경 사항을 병합하면 디스크에서 보조 인덱스 페이지를 버퍼 풀로 읽는 데 필요한 상당한 랜덤 액세스 I/O를 피할 수 있습니다.

시스템이 대부분 유휴 상태이거나 느리게 종료될 때 주기적으로 실행되는 퍼지 작업은 업데이트된 인덱스 페이지를 디스크에 씁니다. 퍼지 작업은 각 값을 디스크에 즉시 기록하는 것보다 일련의 인덱스 값에 대한 디스크 블록을 더 효율적으로 기록할 수 있습니다.

영향을 받는 행이 많고 업데이트할 보조 인덱스가 많은 경우 변경 버퍼 병합에 몇 시간이 걸릴 수 있습니다. 이 시간 동안 디스크 I/O가 증가하여 디스크에 바인딩된 쿼리의 속도가 크게 느려질 수 있습니다. 트랜잭션이 커밋된 후 서버를 종료하고 다시 시작한 후에도 변경 버퍼 병합이 계속 발생할 수 있습니다(자세한 내용은 15.21.3 절. 'InnoDB 복구 강제 수행' 참조).

메모리에서 변경 버퍼는 버퍼 풀의 일부를 차지합니다. 디스크에서 변경 버퍼는 데이터베이스 서버가 종료될 때 인덱스 변경이 버퍼링되는 시스템 테이블 스페이스의 일부입니다.

변경 버퍼에 캐시되는 데이터 유형은 `innodb_change_buffering` 변수에 의해 관리됩니다. 자세한 내용은 변경 버퍼링 구성을 참조하세요. 최대 변경 버퍼 크기를 구성할 수도 있습니다. 자세한 내용은 변경 버퍼 최대 크기 구성을 참조하세요.

인덱스에 내림차순 인덱스 열이 포함되어 있거나 기본 키에 내림차순 인덱스 열이 포함되어 있는 경우 보조 인덱스에는 변경 버퍼링이 지원되지 않습니다.

변경 버퍼에 대해 자주 묻는 질문에 대한 답변은 섹션 A.16, 'MySQL 8.2 FAQ'를 참조하세요: InnoDB 변경 버퍼"를 참조하세요.

변경 버퍼링 구성

테이블에서 `INSERT`, `UPDATE`, `DELETE` 작업을 수행할 때 인덱싱된 열의 값(특히 보조 키의 값)은 정렬되지 않은 순서인 경우가 많으므로 보조 인덱스를 최신 상태로 유지하는 데 상당한 I/O가 필요합니다. 변경 버퍼는 관련 페이지가 버퍼 풀에 없을 때 보조 인덱스 항목에 대한 변경 사항을 캐시하므로 디스크에서 페이지를 즉시 읽지 않음으로써 비용이 많이 드는 I/O 작업을 피할 수 있습니다. 버퍼링된 변경 사항은 페이지가 버퍼 풀에 로드될 때 병합되며, 업데이트된 페이지는 나중에 디스크에 플러시됩니다. `InnoDB` 메인 스레드는 서버가 거의 유휴 상태일 때와 느리게 종료되는 동안 버퍼링된 변경 사항을 병합합니다.

변경 버퍼링은 디스크 읽기 및 쓰기 횟수를 줄일 수 있기 때문에 대량 삽입과 같이 대량의 DML 작업을 수행하는 애플리케이션과 같이 I/O에 바인딩된 워크로드에 가장 유용합니다.

그러나 변경 버퍼는 버퍼 풀의 일부를 차지하므로 데이터 페이지를 캐시하는 데 사용할 수 있는 메모리가 줄어듭니다. 작업 집합이 버퍼 풀에 거의 들어맞거나 테이블에 보조 인덱스가 상대적으로 적은 경우 변경 버퍼링을 비활성화하는 것이 유용할 수 있습니다. 작업 데이터 집합이 버퍼 풀에 완전히 들어맞는 경우, 변경 버퍼링은 버퍼 풀에 없는 페이지에만 적용되므로 추가 오버헤드가 발생하지 않습니다.

`innodb_change_buffering` 변수는 `InnoDB`가 변경 버퍼링을 수행하는 정도를 제어합니다. 삽입, 삭제 작업(인덱스 레코드가 처음에 삭제되도록 표시된 경우) 및 제거 작업(인덱스 레코드가 물리적으로 삭제된 경우)에 대해 버퍼링을 활성화 또는 비활성화할 수 있습니다. 업데이트 작업은 삽입과 삭제의 조합입니다. 기본

`innodb_change_buffering` 값은 모두입니다.

허용되는 `innodb_change_버퍼링` 값은 다음과 같습니다:

- **모두**

  기본값: 버퍼 삽입, 삭제 표시 작업 및 제거.

- **없음**

  어떤 작업도 버퍼링하지 마세요.

- **인서트**

  버퍼 삽입 작업.

- **삭제**

  버퍼 삭제 표시 작업.

- **변경 사항**

  삽입 및 삭제 표시 작업을 모두 버퍼링합니다.

- **퍼지**

  백그라운드에서 발생하는 물리적 삭제 작업을 버퍼링합니다.

MySQL 옵션 파일(`my.cnf` 또는 `my.ini`)에서 `innodb_change_buffering` 변수를 설정하거나 전역 시스템 변수를 설정할 수 있는 충분한 권한이 필요한 `SET GLOBAL` 문을 사용하여 동적으로 변경할 수 있습니다. 섹션 5.1.9.1, "시스템 변수 권한"을 참조하세요. 설정을 변경하면 새 작업의 버퍼링에 영향을 미치며, 기존 버퍼링된 항목의 병합에는 영향을 미치지 않습니다.

변경 버퍼 최대 크기 구성

`innodb_change_buffer_max_size` 변수를 사용하면 변경 버퍼의 최대 크기를 버퍼 풀의 총 크기 대비 백분율로 구성할 수 있습니다. 기본적으로 `innodb_change_buffer_max_size`는 25로 설정되어 있습니다. 최대 설정은 50입니다.

삽입, 업데이트 및 삭제 활동이 많은 MySQL 서버에서 변경 버퍼 병합이 새 변경 버퍼 항목에 보조를 맞추지 못하여 변경 버퍼가 최대 크기 제한에 도달하는 경우 `innodb_change_buffer_max_size`를 늘리는 것을 고려하세요.

보고에 사용되는 정적 데이터가 있거나 변경 버퍼가 버퍼 풀과 공유되는 메모리 공간을 너무 많이 사용하여 버퍼 풀에서 페이지가 원하는 것보다 빨리 노후화되는 경우 MySQL 서버에서 `innodb_change_buffer_max_size`를 줄이는 것을 고려하세요.

대표 워크로드로 다양한 설정을 테스트하여 최적의 구성을 결정합니다. `innodb_change_buffer_max_size` 변수는 동적이므로 서버를 다시 시작하지 않고도 설정을 수정할 수 있습니다.

변경 버퍼 모니터링

변경 버퍼 모니터링에 사용할 수 있는 옵션은 다음과 같습니다:

- `InnoDB` 표준 모니터 출력에는 변경 버퍼 상태 정보가 포함됩니다. 모니터 데이터를 보려면 `SHOW ENGINE INNODB STATUS` 문을 실행합니다.

  ```
  mysql> SHOW ENGINE INNODB STATUS\G
  ```

  버퍼 상태 변경 정보는 버퍼 및 적응형 해시 인덱스 삽입 제목 아래에 있으며 다음과 유사하게 표시됩니

다:

```
-------------------------------------
삽입 버퍼 및 적응형 해시 인덱스
-------------------------------------
Ibuf: 크기 1, 자유 목록 길이 0, 세그 크기 2, 0 병합된 작업을 병합
합니다:
 삽입 0, 마크 0 삭제, 삭제 0 폐기된 작업:
 삽입 0, 마크 삭제 0, 삭제 0
해시 테이블 크기 4425293, 사용 셀 32, 노드 힙 버퍼 1개 해시 검색 횟수
13577.57회/초, 비해시 검색 횟수 202.47회/초
```

자세한 내용은 섹션 15.17.3, "InnoDB 표준 모니터 및 잠금 모니터 출력"을 참조하세요.

- 정보 스키마 `INNODB_METRICS` 테이블은 `InnoDB` 표준 모니터 출력에 있는 대부분의 데이터 포인트와 기타 데이터 포인트를 제공합니다. 변경 버퍼 메트릭과 각 메트릭에 대한 설명을 보려면 다음 쿼리를 실행하세요:

```
mysql> SELECT NAME, COMMENT FROM INFORMATION_SCHEMA.INNODB_METRICS WHERE NAME LIKE '%ibuf%'\G
```

섹션 15.15.6, "InnoDB 정보_SCHEMA 메트릭 테이블"을 참조하세요.

- 정보 스키마 INNODB_BUFFER_PAGE 테이블은 버퍼 인덱스 변경 및 버퍼 비트맵 페이지 변경을 포함하여 버퍼 풀의 각 페이지에 대한 메타데이터를 제공합니다. 변경 버퍼 페이지는 PAGE_TYPE으로 식별됩니다. IBUF_INDEX는 변경 버퍼 인덱스 페이지의 페이지 유형이고, IBUF_BITMAP은 변경 버퍼 비트맵 페이지의 페이지 유형입니다.

경고

INNODB_BUFFER_PAGE 테이블을 쿼리하면 상당한 성능 오버헤드가 발생할 수 있습니다. 성능에 영향을 미치지 않도록 하려면 테스트 인스턴스에서 조사하려는 문제를 재현하고 테스트 인스턴스에서 쿼리를 실행하세요.

예를 들어, INNODB_BUFFER_PAGE 테이블을 쿼리하여 총 버퍼 풀 페이지의 백분율로 IBUF_INDEX 및 IBUF_BITMAP 페이지의 대략적인 수를 확인할 수 있습니다.

```
mysql> SELECT (SELECT COUNT(*) FROM INFORMATION_SCHEMA.INNODB_BUFFER_PAGE
       WHERE PAGE_TYPE LIKE 'IBUF%') AS change_buffer_pages,
       (information_schema.innodb_buffer_page에서 count(*) 선택) AS total_pages,
       (SELECT ((변경_버퍼_페이지/총_페이지)*100)) AS change_
       버퍼_페이지_퍼센티지;
+--------------------+-------------+ ----------------------------------+
| 변경_버퍼_페이지 | 총_페이지 | 변경_버퍼_페이지_퍼센티지 |
+--------------------+-------------+ ----------------------------------+
                    |25          |8192 |0.                          3052 |
+--------------------+-------------+ ----------------------------------+
```

INNODB_BUFFER_PAGE 테이블이 제공하는 기타 데이터에 대한 자세한 내용은 26.4.2절, "정보_스켐마 INNODB_BUFFER_PAGE 테이블"을 참조하세요. 관련 사용 정보는 15.15.5절, "InnoDB INFORMATION_SCHEMA 버퍼 풀 테이블"을 참조하세요.

- 성능 스키마는 고급 성능 모니터링을 위한 변경 버퍼 뮤텍스 대기 계측 기능을 제공합니다. 변경 버퍼 계측을 보려면 다음 쿼리를 실행하세요:

```
mysql> SELECT * FROM performance_schema.setup_instruments
       WHERE NAME LIKE '%wait/synch/mutex/innodb/ibuf%';
+-----------------------------------------------------------+---------+ -------------+
| 이름|                                                      활성화 | 시간 |
+-----------------------------------------------------------+---------+ -------------+
| 대기/동기화/뮤텍스/innodb/ibuf_bitmap_mutex                | 예      | 예    |
| 대기/동기화/뮤텍스/innodb/ibuf_mutex                       | 예      | 예    |
| 대기/동기화/뮤텍스/innodb/ibuf_pessimistic_insert_mutex | 예| 예     |
+-----------------------------------------------------------+---------+ -------------+
```

InnoDB 뮤텍스 대기 모니터링에 대한 자세한 내용은 15.16.2절, '성능 스키마를 사용하여 InnoDB 뮤텍스 대기 모니터링'을 참조하세요.

15.5.3 적응형 해시 인덱스

적응형 해시 인덱스를 사용하면 트랜잭션 기능이나 안정성을 저하시키지 않으면서도 워크로드와 버퍼 풀

에 충분한 메모리가 적절히 조합된 시스템에서 InnoDB가 인메모리 데이터베이스와 같은 성능을 발휘할 수 있습니다. 적응형 해시 인덱스는 `innodb_adaptive_hash_index` 변수로 활성화하거나 서버 시작 시 `--skip-innodb- adaptive-hash-index`로 해제할 수 있습니다.

관찰된 검색 패턴을 기반으로 인덱스 키의 접두사를 사용하여 해시 인덱스가 구축됩니다. 접두사는 길이에 상관없이 사용할 수 있으며, B-트리의 일부 값만 해시 인덱스에 표시될 수 있습니다. 해시 인덱스는 자주 액세스하는 인덱스 페이지에 대한 요청에 따라 구축됩니다.

테이블이 거의 전적으로 메인 메모리에 들어맞는 경우, 해시 인덱스는 인덱스 값을 일종의 포인터로 전환하여 모든 요소를 직접 조회할 수 있게 함으로써 쿼리 속도를 높여줍니다. `InnoDB`에는 다음을 모니터링하는 메커니즘이 있습니다.

인덱스 검색. InnoDB가 쿼리가 해시 인덱스를 구축하면 이점을 얻을 수 있다고 판단하면 자동으로 해시 인덱스를 구축합니다.

일부 워크로드의 경우, 해시 인덱스 조회로 인한 속도 향상이 인덱스 조회를 모니터링하고 해시 인덱스 구조를 유지하기 위한 추가 작업보다 훨씬 더 큰 효과를 발휘합니다. 적응형 해시 인덱스에 대한 액세스는 때때로 여러 개의 동시 조인과 같은 과중한 워크로드에서 경합의 원인이 될 수 있습니다. `LIKE` 연산자 및 `%` 와일드카드를 사용하는 쿼리도 이점을 얻지 못하는 경향이 있습니다. 적응형 해시 인덱스의 이점을 누리지 못하는 워크로드의 경우, 이 인덱스를 해제하면 불필요한 성능 오버헤드를 줄일 수 있습니다. 적응형 해시 인덱스가 특정 시스템 및 워크로드에 적합한지 여부를 미리 예측하기는 어렵기 때문에, 적응형 해시 인덱스를 사용 및 사용하지 않도록 설정한 상태에서 벤치마크를 실행하는 것을 고려하세요.

적응형 해시 인덱스 기능은 파티션으로 나뉩니다. 각 인덱스는 특정 파티션에 바인딩되며, 각 파티션은 별도의 래치로 보호됩니다. 파티셔닝은 `innodb_adaptive_hash_index_parts` 변수를 설정합니다. `innodb_adaptive_hash_index_parts` 변수는 기본적으로 8로 설정됩니다. 최대 설정은 512입니다.

엔진 `INNODB` 상태 표시 출력의 `SEMAPHORES` 섹션에서 적응형 해시 인덱스 사용 및 경합을 모니터링할 수 있습니다. `btr0sea.c`에서 생성된 rw-latches를 대기 중인 스레드가 많은 경우 적응형 해시 인덱스 파티션 수를 늘리거나 적응형 해시 인덱스를 비활성화하는 것을 고려하세요.

해시 인덱스의 성능 특성에 대한 자세한 내용은 섹션 8.3.9, "B-Tree와 해시 인덱스 비교"를 참조하세요.

15.5.4 로그 버퍼

로그 버퍼는 디스크의 로그 파일에 기록할 데이터를 보관하는 메모리 영역입니다. 로그 버퍼 크기는 `innodb_log_buffer_size` 변수에 의해 정의됩니다. 기본 크기는 16MB입니다. 로그 버퍼의 내용은 주기적으로 디스크에 플러시됩니다. 로그 버퍼가 크면 트랜잭션이 커밋되기 전에 재실행 로그 데이터를 디스크에 쓸 필요 없이 대용량 트랜잭션을 실행할 수 있습니다. 따라서 많은 행을 업데이트, 삽입 또는 삭제하는 트랜잭션이 있는 경우 로그 버퍼의 크기를 늘리면 디스크 I/O를 절약할 수 있습니다.

`innodb_flush_log_at_trx_commit` 변수는 로그 버퍼의 내용이 디스크에 기록되고 플러시되는 방식을 제어합니다. `innodb_flush_log_at_timeout` 변수는 로그 플러시 빈도를 제어합니다.

관련 정보는 메모리 구성 및 섹션 8.5.4, "InnoDB 재실행 로깅 최적화"를 참조하세요.

15.6 InnoDB 온디스크 구조

이 섹션에서는 `InnoDB` 온디스크 구조 및 관련 주제에 대해 설명합니다.

15.6.1 테이블

이 섹션에서는 `InnoDB` 테이블과 관련된 주제를 다룹니다.

15.6.1.1 InnoDB 테이블 만들기

`InnoDB` 테이블은 `CREATE TABLE` 문을 사용하여 생성됩니다:

```
CREATE TABLE t1 (a INT, b CHAR (20), PRIMARY KEY (a)) ENGINE=InnoDB;
```

기본 저장소 엔진으로 `InnoDB`가 정의되어 있는 경우 `ENGINE=InnoDB` 절은 필요하지 않습니다. 그러나 기본 저장소 엔진이 `InnoDB`가 아니거나 알 수 없는 다른 MySQL Server 인스턴스에서 `CREATE TABLE` 문을 재생해야 하는 경우 `ENGINE` 절이 유용합니다. 다음 문을 실행하여 MySQL 서버 인스턴스의 기본 저장소 엔진을 확인할 수 있습니다:

```
mysql> SELECT @@default_storage_engine;
+--------------------------+
| @@디폴트_스토리지_엔진 | @@디폴트_스토리지_엔진
```

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

InnoDB 테이블은 기본적으로 테이블별 파일 테이블스페이스에 생성됩니다. InnoDB 시스템 테이블 스페이스에서 InnoDB 테이블을 생성하려면 테이블을 생성하기 전에 innodb_file_per_table 변수를 비활성화합니다. 일반 테이블 스페이스에서 InnoDB 테이블을 만들려면 CREATE TABLE ... 테이블 스페이스 구문을 사용합니다. 자세한 내용은 섹션 15.6.3, "테이블 스페이스"를 참조하십시오.

행 형식

InnoDB 테이블의 행 형식에 따라 행이 디스크에 물리적으로 저장되는 방식이 결정됩니다. InnoDB는 각각 다른 저장 특성을 가진 네 가지 행 형식을 지원합니다. 지원되는 행 형식에는 중복, 콤팩트, 동적 및 압축이 포함됩니다. 기본값은 DYNAMIC 행 형식입니다. 행 형식 특성에 대한 자세한 내용은 섹션 15.10, "InnoDB 행 형식"을 참조하십시오.

innodb_default_row_format 변수는 기본 행 형식을 정의합니다. 테이블의 행 형식은 CREATE TABLE 또는 ALTER TABLE 문에서 ROW_FORMAT 테이블 옵션을 사용하여 명시적으로 정의할 수도 있습니다. 테이블의 행 형식 정의를 참조하십시오.

기본 키

생성하는 각 테이블에 대해 기본 키를 정의하는 것이 좋습니다. 기본 키 열을 선택할 때는 다음과 같은 특성을 가진 열을 선택합니다:

- 가장 중요한 쿼리에서 참조하는 열입니다.
- 절대 비워두지 않는 열입니다.
- 값이 중복되지 않는 열입니다.
- 한번 삽입하면 값이 거의 변경되지 않는 열입니다.

예를 들어 사람에 대한 정보가 포함된 테이블에서 이름(이름, 성)에 기본 키를 만들지 않는 이유는 한 명 이상의 사람이 같은 이름을 가질 수 있고, 이름 열이 비어 있을 수 있으며, 사람들이 이름을 변경하는 경우도 있기 때문입니다. 제약 조건이 너무 많아서 기본 키로 사용할 열 집합이 분명하지 않은 경우가 많으므로 기본 키의 전부 또는 일부로 사용할 숫자 ID를 가진 새 열을 만듭니다. 자동 증가 열을 선언하여 행이 삽입될 때 오름차순 값이 자동으로 채워지도록 할 수 있습니다:

```
# ID 값은 다른 테이블의 관련 항목 간에 포인터처럼 작동할 수 있습니다. CREATE TABLE t5 (id INT
AUTO_INCREMENT, b CHAR (20), PRIMARY KEY (id));

# 기본 키는 둘 이상의 열로 구성될 수 있습니다. 모든 자동 인코딩 열이 먼저 와야 합니다. CREATE TABLE t6 (id
INT AUTO_INCREMENT, a INT, b CHAR (20), PRIMARY KEY (id,a));
```

자동 증가 열에 대한 자세한 내용은 섹션 15.6.1.6, 'InnoDB에서 AUTO_INCREMENT 처리'를 참조하세요
.

기본 키를 정의하지 않아도 테이블이 올바르게 작동하지만 기본 키는 성능의 여러 측면과 관련이 있으며 규모가 크거나 자주 사용되는 테이블의 경우 중요한 설계 요소입니다. 항상 `CREATE TABLE` 문에 기본 키를 지정하는 것이 좋습니다. 테이블을 생성하고 데이터를 로드한 다음 나중에 `ALTER TABLE`을 실행하여 기본 키를 추가하면 테이블을 생성할 때 기본 키를 정의하는 것보다 훨씬 느린 작업이 수행됩니다. 기본 키에 대한 자세한 내용은 15.6.2.1절. "클러스터된 인덱스 및 보조 인덱스"를 참조하십시오.

InnoDB 테이블 속성 보기

`InnoDB` 테이블의 속성을 보려면 `SHOW TABLE STATUS` 문을 실행합니다:

```
mysql> SHOW TABLE STATUS FROM test LIKE 't%' \G;
*************************** 1. 행 *************************** 이
           름: t1
         엔진: InnoDB
```

```
        버전: 10 행_형식:
     Dynamic
             행: 0
     평균_행_길이: 0
    데이터_길이: 16384
  최대_데이터_길이: 0
      Index_길이: 0
      데이터_무료: 0 자동 증
가: NULL
    생성_시간: 2021-02-18 12:18:28 업데이트_
    시간: NULL
     Check_time: NULL
      콜레이션: utf8mb4_0900_ai_ci 체크섬:
       NULL
     Create_옵션:
            댓글:
```

테이블 상태 표시 출력에 대한 자세한 내용은 섹션 13.7.7.40, "테이블 상태 표시 문"을 참조하십시오.

InnoDB 정보 스키마 시스템 테이블을 쿼리하여 InnoDB 테이블 속성에 액세스할 수도 있습니다:

```
mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_TABLES WHERE NAME='test/t1' \G
*************************** 1. 행 ***************************
     TABLE_ID: 1144
         이름: test/t1 플래
         그: 33
       N_COLS: 5
        공간: 30
   ROW_FORMAT: 동적
ZIP_PAGE_SIZE: 0
   SPACE_TYPE: 단일
 INSTANT_COLS: 0
```

자세한 내용은 섹션 15.15.3, "InnoDB 정보_스키마 스키마 객체 테이블"을 참조하세요.

15.6.1.2 외부에서 테이블 만들기

데이터 디렉터리 외부에 InnoDB 테이블을 생성하는 데는 여러 가지 이유가 있습니다. 예를 들어 공간 관리, I/O 최적화, 특정 성능 또는 용량 특성을 가진 저장 장치에 테이블을 배치하는 등의 이유가 있을 수 있습니다.

InnoDB는 외부에서 테이블을 생성하기 위해 다음과 같은 방법을 지원합니다:

- 데이터 디렉터리 절 사용
- 테이블 생성 사용 ... 테이블 스페이스 구문
- 외부 일반 테이블 스페이스에서 테이블 만들기

데이터 디렉터리 절 사용

CREATE TABLE 문에 DATA DIRECTORY 절을 지정하여 외부 디렉터리에 InnoDB 테이블을 생성할 수 있

습니다.

```
CREATE TABLE t1 (c1 INT PRIMARY KEY) DATA DIRECTORY = '/external/directory';
```

DATA DIRECTORY 절은 파일 단위 테이블 테이블 스페이스에서 생성된 테이블에 대해 지원됩니다. 테이블은 innodb_file_per_table 변수가 활성화된 경우 암시적으로 파일 단위 테이블 테이블 공간에 생성되며, 기본적으로 활성화되어 있습니다.

```
mysql> SELECT @@innodb_file_per_table;
+-------------------------+
| @@innodb_file_per_table | @@innodb_file_per_table
+-------------------------+
                          |1 |
```

```
+-------------------------+
```

테이블별 파일 테이블 스페이스에 대한 자세한 내용은 섹션 15.6.3.2, "테이블별 파일 테이블 스페이스"를 참조하십시오.

CREATE TABLE 문에 DATA DIRECTORY 절을 지정하면 지정된 디렉터리 아래의 스키마 디렉터리에 테이블의 데이터 파일(*table_name*.ibd)이 생성됩니다.

데이터 디렉터리 절을 사용하여 데이터 디렉터리 외부에서 생성된 테이블 및 테이블 파티션은 InnoDB에 알려진 디렉터리로 제한됩니다. 이 요구 사항을 통해 데이터베이스 관리자는
를 사용하여 테이블스페이스 데이터 파일이 생성되는 위치를 제어하고 복구 중에 데이터 파일을 찾을 수 있도록 합니다(충돌 복구 중 테이블스페이스 검색 참조). 알려진 디렉터리는 datadir, innodb_data_home_dir 및 innodb_directories 변수에 의해 정의된 디렉터리입니다. 다음 구문을 사용하여 이러한 설정을 확인할 수 있습니다:

```
mysql> SELECT @@datadir,@@innodb_data_home_dir,@@innodb_directories;
```

사용하려는 디렉터리를 알 수 없는 경우 테이블을 생성하기 전에 innodb_directories 설정에 추가합니다. innodb_directories 변수는 읽기 전용입니다. 이 변수를 구성하려면 서버를 다시 시작해야 합니다. 시스템 변수 설정에 대한 일반적인 정보는 5.1.9절. "시스템 변수 사용"을 참조하세요.

다음 예제에서는 데이터 디렉터리 절을 사용하여 외부 디렉터리에 테이블을 생성하는 방법을 보여줍니다. innodb_file_per_table 변수가 활성화되어 있고 해당 디렉터리가 InnoDB에 알려져 있다고 가정합니다.

```
mysql> USE 테스트;
데이터베이스 변경

mysql> CREATE TABLE t1 (c1 INT PRIMARY KEY) DATA DIRECTORY = '/external/directory';

# MySQL은 외부 디렉터리 아래의 스키마 디렉터리에 # 테이블의 데이터 파일
을 생성합니다.

cd /외부/디렉토리/테스트
$> ls
t1.ibd
```

사용 참고 사항:

- MySQL은 처음에는 테이블스페이스 데이터 파일을 열어 두어 장치를 분리할 수 없도록 하지만 서버가 사용 중일 경우 결국 파일을 닫을 수 있습니다. 실수로 마운트 해제하지 않도록 주의하세요. 외부 장치에 연결하거나 장치 연결이 끊어진 상태에서 MySQL을 시작하지 마세요. 연결된 데이터 파일이 누락된 상태에서 테이블에 액세스를 시도하면 서버를 다시 시작해야 하는 심각한 오류가 발생합니다.

  예상 경로에서 데이터 파일을 찾을 수 없는 경우 서버 재시작이 실패할 수 있습니다. 이 경우 백업에서 테이블스페이스 데이터 파일을 복원하거나 테이블을 끌어 놓아 데이터 사전에서 해당 테이블에 대한 정보를 제거할 수 있습니다.

- NFS 마운트 볼륨에 테이블을 배치하기 전에 MySQL에서 NFS 사용하기에 설명된 잠재적인 문제를 검토하세요.
- LVM 스냅샷, 파일 복사 또는 기타 파일 기반 메커니즘을 사용하여 테이블의 데이터 파일을 백업하는 경우 항상 `FLUSH TABLES ... FOR EXPORT` 문을 먼저 사용하여 백업이 수행되기 전에 메모리에 버퍼링된 모든 변경 내용이 디스크로 플러시되도록 합니다.
- 데이터 디렉터리 절을 사용하여 외부 디렉터리에 테이블을 생성하는 것은 InnoDB가 지원하지 않는 심볼릭 링크를 사용하는 대신 사용할 수 있습니다.
- 원본과 복제본이 동일한 호스트에 있는 복제 환경에서는 `DATA DIRECTORY` 절이 지원되지 않습니다. `DATA DIRECTORY` 절에는 전체 디렉터리 경로가 필요합니다. 이 경우 경로를 복제하면 원본과 복제본이 같은 위치에 테이블을 생성하게 됩니다.

- 파일 단위 테이블 스페이스에서 생성된 테이블은 InnoDB에 직접 알려진 디렉터리가 아니면 실행 취소 테이블 스페이스 디렉터리(innodb_undo_directory)에 생성할 수 없습니다. 알려진 디렉터리는 datadir, innodb_data_home_dir 및 innodb_directories 변수에 의해 정의된 디렉터리입니다.

테이블 생성 사용 ... 테이블 스페이스 구문

테이블 생성 ... TABLESPACE 구문을 데이터 디렉터리 절과 함께 사용하여 외부 디렉터리에 테이블을 생성할 수 있습니다. 이렇게 하려면 테이블 스페이스 이름으로 innodb_file_per_table을 지정합니다.

```
mysql> CREATE TABLE t2 (c1 INT PRIMARY KEY) TABLESPACE = innodb_file_per_table 데
       이터 디렉토리 = '/external/directory';
```

이 방법은 파일 단위 테이블 테이블 공간에서 생성된 테이블에 대해서만 지원되지만 innodb_file_per_table 변수를 활성화할 필요는 없습니다. 다른 모든 측면에서 이 메서드는 위에서 설명한 CREATE TABLE ... 데이터 디렉터리 메서드와 동일합니다. 동일한 사용 참고 사항이 적용됩니다.

외부 일반 테이블 스페이스에서 테이블 만들기

외부 디렉터리에 있는 일반 테이블 스페이스에서 테이블을 만들 수 있습니다.

- 외부 디렉터리에 일반 테이블 스페이스를 만드는 방법에 대한 자세한 내용은 일반 테이블 스페이스 만들기를 참조하세요.
- 일반 테이블 스페이스에서 테이블을 만드는 방법에 대한 자세한 내용은 일반 테이블 스페이스에 테이블 추가를 참조하십시오.

15.6.1.3 InnoDB 테이블 가져오기

이 섹션에서는 테이블, 분할된 테이블 또는 파일 단위 테이블 공간에 있는 개별 테이블 파티션을 가져올 수 있는 *전송 가능한 테이블 공간* 기능을 사용하여 테이블을 가져오는 방법에 대해 설명합니다. 테이블을 가져오려는 이유는 여러 가지가 있습니다:

- 프로덕션 서버에 추가 부하가 걸리지 않도록 비프로덕션 MySQL 서버 인스턴스에서 보고서를 실행하려면 다음과 같이 하세요.
- 새 복제본 서버로 데이터를 복사합니다.
- 백업된 테이블스페이스 파일에서 테이블을 복원합니다.
- 데이터를 다시 삽입하고 인덱스를 다시 작성해야 하는 덤프 파일 가져오기보다 데이터를 더 빠르게 이동할 수 있는 방법입니다.
- 스토리지 요구 사항에 더 적합한 저장 매체가 있는 서버로 데이터를 이동합니다. 예를 들어 사용량이 많은 테이블을 SSD 장치로 옮기거나 대용량 테이블을 고용량 HDD 장치로 옮길 수 있습니다.

이동 가능한 테이블 공간 기능은 이 섹션의 다음 항목에서 설명합니다:

- 전제 조건
- 테이블 가져오기
- 분할된 테이블 가져오기
- 테이블 파티션 가져오기
- 제한 사항
- 사용 참고 사항
- 내부

전제 조건

- `innodb_file_per_table` 변수를 활성화해야 하며, 기본적으로 활성화되어 있습니다.

- 테이블스페이스의 페이지 크기는 대상 MySQL 서버 인스턴스의 페이지 크기와 일치해야 합니다. InnoDB 페이지 크기는 MySQL 서버 인스턴스를 초기화할 때 구성되는 innodb_page_size 변수에 의해 정의됩니다.

- 테이블에 외래 키 관계가 있는 경우 DISCARD TABLESPACE를 실행하기 전에 foreign_key_checks를 비활성화해야 합니다. 또한 모든 외래 키 관련 테이블을 동일한 논리적 시점에 내보내야 하며, ALTER TABLE ... IMPORT TABLESPACE는 가져온 데이터에 외래 키 제약 조건을 적용하지 않습니다. 이렇게 하려면 관련 테이블 업데이트를 중지하고, 모든 트랜잭션을 커밋하고, 테이블에 대한 공유 잠금을 획득한 다음 내보내기 작업을 수행합니다.

- 다른 MySQL 서버 인스턴스에서 테이블을 가져오는 경우 두 MySQL 서버 인스턴스 모두 GA(일반 사용 가능성) 상태여야 하며 동일한 버전이어야 합니다. 그렇지 않으면 테이블을 가져올 동일한 MySQL 서버 인스턴스에서 테이블을 만들어야 합니다.

- CREATE TABLE 문에 DATA DIRECTORY 절을 지정하여 외부 디렉터리에 테이블을 생성한 경우 대상 인스턴스에서 대체하는 테이블도 동일한 DATA DIRECTORY 절로 정의해야 합니다. 절이 일치하지 않으면 스키마 불일치 오류가 보고됩니다. 소스 테이블이 DATA DIRECTORY 절로 정의되었는지 확인하려면 SHOW CREATE TABLE을 사용하여 테이블 정의를 확인합니다. DATA DIRECTORY 절 사용에 대한 자세한 내용은 섹션 15.6.1.2, "외부에서 테이블 생성"을 참조하십시오.

- 테이블 정의에 ROW_FORMAT 옵션이 명시적으로 정의되어 있지 않거나 ROW_FORMAT=DEFAULT가 사용되는 경우, 원본 인스턴스와 대상 인스턴스에서 innodb_default_row_format 설정이 동일해야 합니다. 그렇지 않으면 가져오기 작업을 시도할 때 스키마 불일치 오류가 보고됩니다. 테이블 정의를 확인하려면 SHOW CREATE TABLE을 사용합니다. SHOW VARIABLES를 사용하여 innodb_default_row_format 설정을 확인합니다. 관련 정보는 테이블의 행 형식 정의하기를 참조하세요.

테이블 가져오기

이 예에서는 파일 단위 테이블 공간에 있는 파티션되지 않은 일반 테이블을 가져오는 방법을 보여 줍니다.

1. 대상 인스턴스에서 가져오려는 테이블과 동일한 정의의 테이블을 만듭니다. (테이블 정의는 SHOW CREATE TABLE 구문을 사용하여 얻을 수 있습니다.) 테이블 정의가 일치하지 않으면 가져오기 작업을 시도할 때 스키마 불일치 오류가 보고됩니다.

```
mysql> USE 테스트;
mysql> CREATE TABLE t1 (c1 INT) ENGINE=INNODB;
```

2. 대상 인스턴스에서 방금 만든 테이블의 테이블 스페이스를 삭제합니다. (가져오기 전에 수신 테이블의 테이블 스페이스를 삭제해야 합니다.)

```
mysql> ALTER TABLE t1 DISCARD TABLESPACE;
```

3. 소스 인스턴스에서 FLUSH TABLES ... FOR EXPORT를 실행하여 가져오려는 테이블을 쿼리합니다. 테이블이 큐에 들어가면 테이블에 읽기 전용 트랜잭션만 허용됩니다.

```
mysql> USE 테스트;
mysql> 내보내기를 위해 테이블 t1을 플러시합니다;
```

테이블 플러시 ... FOR EXPORT는 명명된 테이블의 변경 내용을 디스크에 플러시하여 서버가 실행되는 동안 이진 테이블 복사본을 만들 수 있도록합니다. FLUSH TABLES ... FOR EXPORT를 실행하면 InnoDB는 테이블의 스키마 디렉터리에 .cfg 메타데이터 파일을 생성합니다. .cfg 파일에는 가져오기 작업 중에 스키마 확인에 사용되는 메타데이터가 포함되어 있습니다.

참고

FLUSH TABLES ... FOR EXPORT를 실행하는 연결은 열려 있어야 합니다. FOR EXPORT가 실행되는 동안 열려 있어야 하며, 그렇지 않으면 연결이 닫힐 때 잠금이 해제되면서 .cfg 파일이 제거됩니다.

4. 소스 인스턴스에서 대상 인스턴스로 .ibd 파일 및 .cfg 메타데이터 파일을 복사합니다. 예를 들어

```
$> scp /path/to/datadir/test/t1.{ibd,cfg} 대상-서버:/path/to/datadir/test
```

다음 단계에 설명된 대로 공유 잠금을 해제하기 전에 .ibd 파일과 .cfg 파일을 복사해야 합니다.

참고

암호화된 테이블 스페이스에서 테이블을 가져오는 경우 InnoDB는 .cfg 메타데이터 파일과 함께 .cfp 파일을 생성합니다. .cfp 파일은 .cfg 파일과 함께 대상 인스턴스에 복사해야 합니다. 대상 인스턴스에 .cfp 파일에는 전송 키와 암호화된 테이블스페이스 키가 포함되어 있습니다. 가져오기 시 InnoDB는 전송 키를 사용하여 테이블스페이스 키의 암호를 해독합니다. 관련 정보는 섹션 15.13, "InnoDB 저장 데이터 암호화"를 참조하십시오.

5. 소스 인스턴스에서 테이블 잠금 해제를 사용하여 테이블 플러시 ...에서 획득한 잠금을 해제합니다. FOR EXPORT 문으로 획득한 잠금을 해제합니다:

```
mysql> USE test;
mysql> UNLOCK TABLES;
```

테이블 잠금 해제 작업은 .cfg 파일도 제거합니다.

6. 대상 인스턴스에서 테이블스페이스를 가져옵니다:

```
mysql> USE 테스트;
mysql> ALTER TABLE t1 IMPORT TABLESPACE;
```

분할된 테이블 가져오기

이 예에서는 각 테이블 파티션이 파일 단위 테이블 공간에 있는 파티션된 테이블을 가져오는 방법을 보여 줍니다.

1. 대상 인스턴스에서 가져오려는 파티션 테이블과 동일한 정의로 파티션된 테이블을 만듭니다. (테이블 정의는 SHOW CREATE TABLE 구문을 사용하여 얻을 수 있습니다.) 테이블 정의가 일치하지 않으면 가져오기 작업을 시도할 때 스키마 불일치 오류가 보고됩니다.

```
mysql> USE 테스트;
mysql> CREATE TABLE t1 (i int) ENGINE = InnoDB PARTITION BY KEY (i) PARTITIONS 3;
```

datadir/test 디렉터리에는 세 파티션 각각에 대한 테이블스페이스 .ibd 파일이 있습니다.

```
mysql> \! ls /path/to/datadir/test/
T1#P#P0.IBD T1#P#P1.IBD T1#P#P2.IBD
```

2. 대상 인스턴스에서 분할된 테이블의 테이블 스페이스를 삭제합니다. (가져오기 작업을 수행하기 전에 수신 테이블의 테이블 스페이스를 삭제해야 합니다.)

```
mysql> ALTER TABLE t1 DISCARD TABLESPACE;
```

파티션된 테이블의 세 개의 테이블스페이스 .ibd 파일은 /datadir/test에서 삭제됩니다.
디렉터리로 이동합니다.

3. 소스 인스턴스에서 FLUSH TABLES ... FOR EXPORT를 실행하여 가져오려는 분할된 테이블을 쿼리합니다. 테이블이 큐에 들어가면 해당 테이블에서는 읽기 전용 트랜잭션만 허용됩니다.

```
mysql> USE 테스트;
mysql> 내보내기를 위해 테이블 t1을 플러시합니다;
```

테이블 플러시 ... FOR EXPORT는 명명된 테이블의 변경 사항을 디스크에 플러시하여 서버가 실행되는 동안 이진 테이블 복사본을 만들 수 있도록 합니다. 테이블을 플러시할 때 ...

EXPORT를 실행하면 InnoDB는 테이블의 스키마 디렉터리에 테이블의 각 테이블스페이스 파일에 대한 .cfg 메타데이터 파일을 생성합니다.

```
mysql> \! ls /path/to/datadir/test/
t1#p#p0.ibd t1#p#p0.ibd t1#p#p1.ibd
t1#p#p2.ibd t1#p#p0.cfg t1#p#p1.cfg
t1#p#p2.cfg
```

.cfg 파일에는 테이블 스페이스를 가져올 때 스키마 확인에 사용되는 메타데이터가 포함되어 있습니다. 테이블 플러시 ... FOR EXPORT는 개별 테이블 파티션이 아닌 테이블에서만 실행할 수 있습니다.

4. 소스 인스턴스 스키마 디렉터리에서 대상 인스턴스 스키마 디렉터리로 .ibd 및 .cfg 파일을 복사합니다. 예를 들어

```
$>scp /path/to/datadir/test/t1*.{ibd,cfg} 대상-서버:/path/to/datadir/test
```

다음 단계에 설명된 대로 공유 잠금을 해제하기 전에 .ibd 및 .cfg 파일을 복사해야 합니다.

참고

암호화된 테이블 스페이스에서 테이블을 가져오는 경우 InnoDB는 .cfg 메타데이터 파일과 함께 .cfp 파일을 생성합니다. .cfp 파일은 .cfg 파일과 함께 대상 인스턴스에 복사해야 합니다.

.cfp 파일에는 전송 키와 암호화된 테이블스페이스 키가 포함되어 있습니다. 가져올 때 InnoDB는 전송 키를 사용하여 테이블스페이스 키를 해독합니다. 관련 정보는 섹션 15.13, "InnoDB 저장 데이터 암호화"를 참조하십시오.

5. 소스 인스턴스에서 테이블 잠금 해제를 사용하여 테이블 플러시 ...로 획득한 잠금을 해제합니다. FOR EXPORT:

```
mysql> USE 테스트;
mysql> 테이블 잠금 해제;
```

6. 대상 인스턴스에서 파티션된 테이블의 테이블 스페이스를 가져옵니다:

```
mysql> USE 테스트;
mysql> ALTER TABLE t1 IMPORT TABLESPACE;
```

테이블 파티션 가져오기

이 예에서는 각 파티션이 테이블별 테이블 공간 파일에 있는 개별 테이블 파티션을 가져오는 방법을 보여 줍니다.

다음 예에서는 4개의 파티션으로 구성된 테이블 중 2개의 파티션(p2 및 p3)을 가져옵니다.

1. 대상 인스턴스에서 파티션을 가져오려는 파티션 테이블과 동일한 정의로 파티션된 테이블을 만듭니다. (테이블 정의는 SHOW CREATE TABLE 구문을 사용하여 얻을 수 있습니다.) 테이블 정의가 일치하지

않으면 가져오기 작업을 시도할 때 스키마 불일치 오류가 보고됩니다.

```
mysql> USE 테스트;
mysql> CREATE TABLE t1 (i int) ENGINE = InnoDB PARTITION BY KEY (i) PARTITIONS 4;
```

`datadir/test` 디렉터리에는 4개의 파티션 각각에 대한 테이블스페이스 `.ibd` 파일이 있습니다.

```
mysql> \! ls /path/to/datadir/test/
T1#P#P0.IBD T1#P#P1.IBD T1#P#P2.IBD T1#P#P3.IBD
```

2. 대상 인스턴스에서 소스 인스턴스에서 가져오려는 파티션을 삭제합니다. (파티션을 가져오기 전에 수신 파티션 테이블에서 해당 파티션을 삭제해야 합니다.)

```
mysql> ALTER TABLE t1 DISCARD PARTITION p2, p3 TABLESPACE;
```

삭제된 두 파티션의 테이블스페이스 .ibd 파일은 /datadir/test에서 제거됩니다.
디렉터리에 다음 파일을 남깁니다:

```
mysql> \! ls /path/to/datadir/test/
T1#P#P0.IBD T1#P#P1.IBD
```

참고

테이블 변경 ... 파티션 삭제 ... 테이블 공간은
를 하위 파티션 테이블에서 실행하는 경우 파티션 및 하위 파티션 테이블 이름이 모두 허용됩니다. 파티션 이름을 지정하면 해당 파티션의 하위 파티션이 작업에 포함됩니다.

3. 소스 인스턴스에서 테이블을 플러시 ... FOR EXPORT를 실행하여 분할된 테이블을 큐에 넣습니다. 테이블이 큐에 들어가면 해당 테이블에는 읽기 전용 트랜잭션만 허용됩니다.

```
mysql> USE 테스트;
mysql> 내보내기를 위해 테이블 t1을 플러시합니다;
```

테이블 플러시 ... FOR EXPORT는 명명된 테이블의 변경 사항을 디스크로 플러시하여 인스턴스가 실행되는 동안 이진 테이블 복사본을 만들 수 있도록합니다. FLUSH TABLES ... FOR EXPORT를 실행하면 InnoDB는 테이블의 스키마 디렉터리에 있는 각 테이블의 테이블스페이스 파일에 대한 .cfg 메타데이터 파일을 생성합니다.

```
mysql> \! ls /path/to/datadir/test/
T1#P#P0.IBD T1#P#P1.IBD T1#P#P2.IBD T1#P#P3.IBD
T1#P#P0.CFG T1#P#P1.CFG T1#P#P2.CFG T1#P#P3.CFG
```

.cfg 파일에는 가져오기 작업 중 스키마 확인에 사용되는 메타데이터가 포함되어 있습니다.
테이블 플러시 ... FOR EXPORT는 개별 테이블 파티션이 아닌 테이블에서만 실행할 수 있습니다.

4. 소스 인스턴스 스키마 디렉터리에서 대상 인스턴스 스키마 디렉터리로 파티션 p2 및 파티션 p3의 .ibd 및 .cfg 파일을 복사합니다.

```
$> scp t1#p#p2.ibd t1#p#p2.cfg t1#p#p3.ibd t1#p#p3.cfg 대상-서버:/path/to/datadir/test
```

다음 단계에 설명된 대로 공유 잠금을 해제하기 전에 .ibd 및 .cfg 파일을 복사해야 합니다.

참고

암호화된 테이블스페이스에서 파티션을 가져오는 경우, InnoDB는 .cfg 메타데이터 파일과 함께 .cfp 파일을 생성합니다. .cfp 파일은 .cfg 파일과 함께 대상 인스턴스에 복사해야 합니다.
.cfp 파일에는 전송 키와 암호화된 테이블스페이스 키가 포함되어 있습니다. 가져올 때 InnoDB는 전송 키를 사용하여 테이블스페이스 키를 해독합니다. 관련 정보는 섹션 15.13, "InnoDB 저장 데이터 암호화"를 참조하십시오.

5. 소스 인스턴스에서 테이블 잠금 해제를 사용하여 테이블 플러시 ...로 획득한 잠금을 해제합니다. FOR EXPORT:

```
mysql> USE 테스트;
mysql> 테이블 잠금 해제;
```

6. 대상 인스턴스에서 테이블 파티션 p2 및 p3을 가져옵니다:

```
mysql> USE 테스트;
mysql> ALTER TABLE t1 IMPORT PARTITION p2, p3 TABLESPACE;
```

참고

테이블 변경 ... 파티션 가져오기 ... 테이블 공간은
하위 파티션 테이블에서 실행되는 경우 파티션 및 하위 파티션 테이블 이름은 모두

허용됩니다. 파티션 이름을 지정하면 해당 파티션의 하위 파티션이 작업에 포함됩니다.

제한 사항

- *전송 가능한 테이블 공간* 기능은 파일 단위 테이블 공간에 있는 테이블에만 지원됩니다. 시스템 테이블 공간 또는 일반 테이블 공간에 있는 테이블에는 지원되지 않습니다. 공유 테이블 스페이스의 테이블은 쿼리할 수 없습니다.
- 테이블 플러시 ... 전체 텍스트 검색 보조 테이블은 플러시할 수 없으므로 전체 텍스트 인덱스가 있는 테이블에서는 FOR EXPORT가지원되지 않습니다. 전체 텍스트 인덱스가있는 테이블을 가져온 후 테이블 최적화를 실행하여 전체 텍스트 인덱스를 다시 작성합니다. 또는 내보내기 작업 전에 전체 텍스트 인덱스를 삭제하고 대상 인스턴스에서 테이블을 가져온 후 인덱스를 다시 생성합니다.
- .cfg 메타데이터 파일 제한으로 인해 분할된 테이블을 가져올 때 파티션 유형 또는 파티션 정의 차이에 대한 스키마 불일치는 보고되지 않습니다. 열 차이점은 보고됩니다.

사용 참고 사항

- 즉시 추가되거나 삭제된 열이 포함된 테이블을 제외하고 ALTER TABLE ... 테이블 공간 가져오기에는 테이블을 가져오는 데 .cfg 메타데이터 파일이 필요하지 않습니다. 그러나 .cfg 파일 없이 가져올 때는 메타데이터 검사가 수행되지 않으며 다음과 유사한 경고가 표시됩니다:

```
메시지: InnoDB: IO 읽기 오류: (2, 해당 파일 또는 디렉터리 없음) '.\ test\t.cfg'를 여는 동안
오류가 발생하여 스키마 확인 없이 가져오기를 시도합니다.
1행 1세트(0.00초)
```

  .cfg 메타데이터 파일 없이 테이블을 가져오는 것은 스키마 불일치가 예상되지 않고 테이블에 즉시 추가되거나 삭제된 열이 포함되지 않은 경우에만 고려해야 합니다. 기능
  를 사용하면 메타데이터에 액세스할 수 없는 크래시 복구 시나리오에서 .cfg 파일 없이 가져올 수 있습니다.

  ALGORITHM=INSTANT를 사용하여 추가 또는 삭제된 열이 있는 테이블을 가져오려고 시도하는 중입니다. .cfg 파일을 사용하지 않으면 정의되지 않은 동작이 발생할 수 있습니다.

- Windows에서 InnoDB는 데이터베이스, 테이블 스페이스 및 테이블 이름을 내부적으로 소문자로 저장합니다. Linux 및 Unix와 같이 대소문자를 구분하는 운영 체제에서 가져오기 문제를 방지하려면 모든 데이터베이스, 테이블 공간 및 테이블을 소문자 이름을 사용하여 생성하세요. 이름을 소문자로 생성하는 편리한 방법은 서버를 초기화하기 전에 lower_case_table_names를 1로 설정하는 것입니다. (서버를 초기화할 때 사용한 설정과 다른 lower_case_table_names 설정으로 서버를 시작하는 것은 금지되어 있습니다.)

```
[mysqld]
lower_case_table_names=1
```

내부

- 테이블 변경을 실행할 때 ... 파티션 삭제 ... 테이블 스페이스 및 ALTER 테이블 ... 파티션 가져오기 ... 테이블스페이스에서 파티션 및 하위 파티션 테이블 이름을 모두 사용할 수 있습니다. 파티션 이름을 지정하면 해당 파티션의 하위 파티션이 작업에 포함됩니다.

다음 정보에서는 테이블 가져오기 절차 중 오류 로그에 기록되는 내부 및 메시지에 대해 설명합니다.

ALTER TABLE ... 대상 인스턴스에서 DISCARD TABLESPACE가 실행됩니다:

- 테이블이 X 모드로 잠겨 있습니다.
- 테이블 스페이스가 테이블에서 분리됩니다.

테이블을 플러시할 때 ... FOR EXPORT가 소스 인스턴스에서 실행됩니다:

- 내보내기를 위해 플러시 중인 테이블은 공유 모드에서 잠겨 있습니다.
- 퍼지 코디네이터 스레드가 중지됩니다.
- 더티 페이지는 디스크에 동기화됩니다.
- 테이블 메타데이터는 바이너리 `.cfg` 파일에 기록

```
[참고] InnoDB: '"test"."t1"'의 디스크에 동기화 시작. [참고]
InnoDB: 퍼지 중지 중
[참고] InnoDB: 테이블 메타데이터를 './test/t1.cfg'에 쓰기 [참고]
InnoDB: 테이블 '"test"."t1"'이 디스크에 플러시됨
```

됩니다. 이 작업에 대한 예상 오류 로그 메시지입니다:

소스 인스턴스에서 테이블 잠금 해제가실행되는 경우:

- 바이너리 `.cfg` 파일이 삭제됩니다.
- 가져오기 중인 테이블의 공유 잠금이 해제되고 제거 코디네이터 스레드가 다시 시작됩니다.

이 작업에 대한 예상 오류 로그 메시지입니다:

```
[참고] InnoDB: 메타데이터 파일 './test/t1.cfg' 삭제 [참고] InnoDB: 퍼
지 재시작
```

`ALTER TABLE ...` 대상 인스턴스에서 `IMPORT` TABLESPACE를 실행하면 가져오기 알고리즘은 가져오는 각 테이블 스페이스에 대해 다음 작업을 수행합니다:

- 각 테이블스페이스 페이지의 손상 여부를 확인합니다.
- 각 페이지의 스페이스 ID 및 로그 시퀀스 번호(LSN)가 업데이트됩니다.
- 플래그의 유효성이 검사되고 헤더 페이지에 대한 LSN이 업데이트됩니다.
- Btree 페이지가 업데이트됩니다.
- 페이지 상태가 더티로 설정되어 디스크에 기록됩니다. 이 작

업에 대해 예상되는 오류 로그 메시지입니다:

```
[참고] InnoDB: 호스트 'host_name'에서 내보낸 테이블 'test/t1'에 대한 테이블 스페이스 가
져오기 중
[참고] InnoDB: 1단계 - 모든 페이지 업데이트 [참
고] InnoDB: 디스크에 동기화
[참고] InnoDB: 디스크에 동기화 - 완료!
[참고] InnoDB: 3단계 - 디스크에 변경 사항 플러시 [참고]
InnoDB: 4단계 - 플러시 완료
```

참고

또한 테이블 공간이 삭제되었다는 경고(대상 테이블의 테이블 공간을 삭제한 경우)와 누락된 `.ibd` 파일로 인해 통계를 계산할 수 없다는 메시지가 표시될 수도 있습니다:

```
[경고] InnoDB: 테이블 "test"."t1" 테이블 스페이스가 폐기된 것으로 설정되었습니다.
7f34d9a37700 InnoDB: 테이블에 대한 통계를 계산할 수 없습니다.
"test"."t1" 파일이 누락되었기 때문입니다. 도움이 필요하면
http://dev.mysql.com/doc/refman/8.1/en/innodb-troubleshooting.html
```

15.6.1.4 InnoDB 테이블 이동 또는 복사

이 섹션에서는 `InnoDB` 테이블의 일부 또는 전부를 다른 서버 또는 인스턴스로 이동하거나 복사하는 기술에 대해 설명합니다. 예를 들어, 전체 MySQL 인스턴스를 더 크고 빠른 서버로 이동하거나, 전체 MySQL 인스턴스를 새 복제본 서버로 복제하거나, 개별 테이블을 다른 인스턴스로 복사하여 애플리케이션을 개발 및 테스트하거나, 데이터 웨어하우스 서버로 복사하여 보고서를 생성할 수 있습니다.

Windows에서 InnoDB는 내부적으로 데이터베이스 및 테이블 이름을 항상 소문자로 저장합니다. 바이너리 형식의 데이터베이스를 유닉스에서 윈도우로 또는 윈도우에서 유닉스로 이동하려면 모든 데이터베이스와 테이블 이름을 소문자로 생성하세요. 이 작업을 수행하는 편리한 방법은 데이터베이스 또는 테이블을 생성하기 전에 my.cnf 또는 my.ini 파일의 [mysqld] 섹션에 다음 줄을 추가하는 것입니다:

```
[mysqld]
lower_case_table_names=1
```

참고

서버를 초기화할 때 사용한 설정과 다른 소문자_테이블_이름 설정으로 서버를 시작하는 것은 금지되어 있습니다.

InnoDB 테이블을 이동하거나 복사하는 기술에는 다음이 포함됩니다:

- 테이블 가져오기
- MySQL 엔터프라이즈 백업
- 데이터 파일 복사(콜드 백업 방법)
- 논리적 백업에서 복원

테이블 가져오기

파일 단위 테이블 스페이스에 있는 테이블은 이동 *가능한 테이블 스페이스* 기능을 사용하여 다른 MySQL 서버 인스턴스 또는 백업에서 가져올 수 있습니다. 섹션 15.6.1.3, 'InnoDB 테이블 가져오기'를 참조하세요.

MySQL 엔터프라이즈 백업

MySQL Enterprise 백업 제품을 사용하면 데이터베이스의 일관된 스냅샷을 생성하면서 운영 중단을 최소화하여 실행 중인 MySQL 데이터베이스를 백업할 수 있습니다. MySQL Enterprise 백업이 테이블을 복사하는 동안에도 읽기 및 쓰기는 계속할 수 있습니다. 또한 MySQL 엔터프라이즈 백업은 압축된 백업 파일을 생성하고 테이블의 하위 집합을 백업할 수 있습니다. MySQL 바이너리 로그와 함께 특정 시점 복구를 수행할 수 있습니다. MySQL Enterprise 백업은 MySQL Enterprise 구독의 일부로 포함되어 있습니다.

MySQL 엔터프라이즈 백업에 대한 자세한 내용은 섹션 30.2, 'MySQL 엔터프라이즈 백업 개요'를 참조하세요.

데이터 파일 복사(콜드 백업 방법)

15.18.1절, 'InnoDB 백업'의 '콜드 백업'에 나열된 모든 관련 파일을 복사하여 InnoDB 데이터베이스를 이동할 수 있습니다.

InnoDB 데이터 및 로그 파일은 동일한 부동 소수점 숫자 형식을 사용하는 모든 플랫폼에서 바이너리 호환이 가능합니다. 부동 소수점 형식은 다르지만 테이블에 FLOAT 또는 DOUBLE 데이터 유형을 사용하지 않은 경우,

절차는 동일합니다. 관련 파일을 복사하기만 하면 됩니다.

테이블별 `.ibd` 파일을 이동하거나 복사할 때 데이터베이스 디렉터리 이름은 소스 시스템과 대상 시스템에서 동일해야 합니다. `InnoDB` 공유 테이블 공간에 저장된 테이블 정의에는 데이터베이스 이름이 포함됩니다. 테이블스페이스 파일에 저장된 트랜잭션 ID와 로그 시퀀스 번호도 데이터베이스마다 다릅니다.

한 데이터베이스에서 다른 데이터베이스로 `.ibd` 파일 및 관련 테이블을 이동하려면 테이블 이름 바꾸기를 사용합니다.
문을 사용합니다:

```
테이블 이름을 db1.tbl_name에서 db2.tbl_name으로 변경합니다;
```

`.ibd` 파일의 "깨끗한" 백업이 있는 경우 다음과 같이 해당 파일이 생성된 MySQL 설치로 복원할 수 있습니다:

1. 테이블을 삭제하거나 잘라내면 테이블 공간에 저장된 테이블 ID가 변경되므로 `.ibd` 파일을 복사한 이후 테이블을 삭제하거나 잘라내지 않아야 합니다.
2. 이 `ALTER TABLE` 문을 실행하여 현재 `.ibd` 파일을 삭제합니다:

```
ALTER TABLE tbl_name DISCARD TABLESPACE;
```

3. 백업 `.ibd` 파일을 적절한 데이터베이스 디렉터리에 복사합니다.
4. 이 `ALTER TABLE` 문을 실행하여 `InnoDB`에 테이블에 새 `.ibd` 파일을 사용하도록 지시합니다:

```
ALTER TABLE tbl_name IMPORT TABLESPACE;
```

참고

`ALTER TABLE ... 테이블 공간` 가져오기 기능은 가져온 데이터에 외래 키 제약 조건을 적용하지 않습니다.

여기서 '깨끗한' `.ibd` 파일 백업이란 다음 요구 사항을 충족하는 백업을 말합니다:

- `.ibd` 파일에는 트랜잭션에 의한 커밋되지 않은 수정 사항이 없습니다.
- `.ibd` 파일에 병합되지 않은 삽입 버퍼 항목이 없습니다.
- 퍼지는 `.ibd` 파일에서 삭제 표시된 모든 인덱스 레코드를 제거합니다.
- mysqld가 `.ibd` 파일의 수정된 모든 페이지를 버퍼 풀에서 파일로 플러시했습니다. 다음 방

법을 사용하여 깨끗한 백업 `.ibd` 파일을 만들 수 있습니다:

1. `mysqld` 서버에서 모든 활동을 중지하고 모든 트랜잭션을 커밋합니다.
2. `엔진 상태` 표시에서 데이터베이스에 활성 트랜잭션이 없고 `InnoDB`의 메인 스레드 상태가 `서버 활동 대기 중이라고` 표시될 때까지 기다립니다. 그런 다음 `.ibd` 파일의 복사본을 만들 수 있습니다.

`.ibd` 파일의 깨끗한 복사본을 만드는 또 다른 방법은 MySQL 엔터프라이즈 백업 제품을 사용하는 것입니다:

1. MySQL 엔터프라이즈 백업을 사용하여 `InnoDB` 설치를 백업합니다.
2. 백업에서 두 번째 `mysqld` 서버를 시작하고 백업에서 `.ibd` 파일을 정리하도록 합니다.

논리적 백업에서 복원

`mysqldump`와 같은 유틸리티를사용하여 논리적 백업을 수행하면 다른 SQL 서버로 전송할 원본 데이터베이스 객체 정의 및 테이블 데이터를 재현하기 위해 실행할 수 있는 SQL 문 집합을 생성할 수 있습니다. 이 방법을 사용하면 형식이 다른지 또는 테이블에 부동 소수점 데이터가 포함되어 있는지 여부는 중요하지 않습니다.

이 방법의 성능을 개선하려면 데이터를 가져올 때 `자동 커밋을` 비활성화하세요. 전체 테이블 또는 테이블의 세그먼트를 가져온 후에만 커밋을 수행합니다.

15.6.1.5 MyISAM에서 InnoDB로 테이블 변환하기

안정성과 확장성을 높이기 위해 `InnoDB`로 변환하려는 `MyISAM` 테이블이 있는 경우, 변환하기 전에 다음 지침과 팁을 검토하세요.

참고

이전 버전의 MySQL에서 생성된 파티션된 `MyISAM` 테이블은 MySQL 8.2와 호환되지 않습니다. 이러한 테이블은 업그레이드하기 전에 파티셔닝을 제거하거나 `InnoDB`로 변환하여 준비해야 합니다. 자세한 내용은 섹션 24.6.2, '스토리지 엔진과 관련된 파티셔닝 제한 사항'을 참조하세요.

- MyISAM 및 InnoDB의 메모리 사용량 조정
- 너무 길거나 짧은 거래 처리하기
- 교착 상태 처리하기
- 스토리지 레이아웃
- 기존 테이블 변환
- 테이블 구조 복제
- 데이터 전송
- 스토리지 요구 사항
- 기본 키 정의
- 애플리케이션 성능 고려 사항
- InnoDB 테이블과 연결된 파일 이해

MyISAM 및 InnoDB의 메모리 사용량 조정

`MyISAM` 테이블에서 전환할 때, 더 이상 결과 캐싱에 필요하지 않은 메모리를 확보하기 위해 `key_buffer_size` 구성 옵션의 값을 낮추세요. 캐시를 할당하는 유사한 역할을 수행하는 `innodb_buffer_pool_size` 구성 옵션의 값을 높입니다.
메모리를 사용합니다. `InnoDB` 버퍼 풀은 테이블 데이터와 인덱스 데이터를 모두 캐시하여 쿼리 조회 속도를 높이고 쿼리 결과를 메모리에 보관하여 재사용할 수 있도록 합니다. 버퍼 풀 크기 구성에 대한 지침은 섹션 8.12.3.1, 'MySQL이 메모리를 사용하는 방법'을 참조하세요.

너무 길거나 짧은 거래 처리하기

`MyISAM` 테이블은 트랜잭션을 지원하지 않기 때문에 자동 커밋 구성 옵션과 `COMMIT` 및 `ROLLBACK` 문에 대해 크게 신경 쓰지 않았을 수도 있습니다. 이러한 키워드는 여러 세션이 동시에 `InnoDB` 테이블을 읽고 쓸 수 있도록 하는 데 중요하며, 쓰기 작업이 많은 워크로드에서 상당한 확장성 이점을 제공합니다.

트랜잭션이 열려 있는 동안 시스템은 트랜잭션 시작 시점의 데이터 스냅샷을 유지하므로, 스트레이 트랜잭션이 계속 실행되는 동안 시스템이 수백만 개의 행을 삽입, 업데이트, 삭제하는 경우 상당한 오버헤드가 발생할 수 있습니다. 따라서 트랜잭션이 너무 오래 실행되지 않도록 주의하세요:

- 대화형 실험을 위해 `mysql` 세션을 사용하는 경우, 완료되면 항상 커밋(변경 사항을 마무리하기 위해) 또는 롤백(변경 사항을 취소하기 위해)을 수행하세요. 대화형 세션을 장시간 열어 두지 말고 종료하여 실수로 트랜잭션이 장시간 열려 있는 것을 방지하세요.
- 애플리케이션의 오류 처리기가 불완전한 변경 사항을 롤백하거나
  변경 사항을 커밋합니다.

- 롤백은 상대적으로 비용이 많이 드는 작업으로, 대부분의 변경 사항이 성공적으로 커밋되고 롤백이 거의 발생하지 않을 것으로 예상하여 `INSERT`, `UPDATE`, `DELETE` 작업이 커밋 전에 `InnoDB` 테이블에 쓰여지기 때문입니다. 대량의 데이터를 실험할 때는 많은 수의 행을 변경한 다음 해당 변경 사항을 롤백하지 않도록 하세요.

- `INSERT` 문 시퀀스를 사용하여 대량의 데이터를 로드할 때는 주기적으로 결과를 커밋하여 트랜잭션이 몇 시간 동안 지속되는 것을 방지하세요. 데이터 웨어하우징을 위한 일반적인 로드 작업에서는 문제가 발생하면 롤백을 수행하지 않고 테이블을 잘라내고(`TRUNCATE TABLE` 사용) 처음부터 다시 시작합니다.

앞의 팁은 너무 긴 트랜잭션 중에 낭비될 수 있는 메모리와 디스크 공간을 절약합니다. 트랜잭션이 예상보다 짧으면 과도한 I/O가 문제입니다. MySQL은 커밋할 때마다 각 변경 사항이 디스크에 안전하게 기록되도록 하는데, 여기에는 약간의 I/O가 포함됩니다.

- InnoDB 테이블에 대한 대부분의 작업에서는 자동 커밋 설정을 0으로 사용해야 합니다. 효율성 측면에서 보면, 이렇게 하면 많은 수의 연속적인 INSERT, UPDATE 또는 DELETE 문을 실행할 때 불필요한 I/O를 피할 수 있습니다. 안전성의 관점에서 보면, mysql 명령줄 또는 애플리케이션의 예외 처리기에서 실수를 한 경우 손실되거나 왜곡된 데이터를 복구하기 위해 ROLLBACK 문을 실행할 수 있습니다.
- 보고서 생성 또는 통계 분석을 위한 일련의 쿼리를 실행할 때 InnoDB 테이블에는 autocommit=1이 적합합니다. 이 경우 커밋 또는 롤백과 관련된 I/O 페널티가 없으며, InnoDB가 읽기 전용 워크로드를 자동으로 최적화할 수 있습니다.
- 일련의 관련 변경을 수행하는 경우 마지막에 한 번의 커밋으로 모든 변경을 한 번에 마무리하세요. 예를 들어 여러 테이블에 관련 정보를 삽입하는 경우 모든 변경을 수행한 후 한 번의 커밋을 수행합니다. 또는 여러 개의 연속된 INSERT 문을 실행하는 경우 모든 데이터가 로드된 후 한 번의 커밋을 수행하고, 수백만 개의 INSERT 문을 실행하는 경우 다음을 수행합니다.
  만 개 또는 십만 개의 레코드마다 커밋을 발행하여 거대한 트랜잭션을 분할하여 트랜잭션이 너무 커지지 않도록 합니다.
- SELECT 문도 트랜잭션을 열 수 있으므로 대화형 mysql 세션에서 일부 보고서를 실행하거나 쿼리를 디버깅한 후에는 COMMIT을 실행하거나 mysql 세션을 닫아야 한다는 점을 기억하세요.

관련 정보는 섹션 15.7.2.2, "자동 커밋, 커밋 및 롤백"을 참조하세요.

교착 상태 처리하기

MySQL 오류 로그에 "교착 상태"를 언급하는 경고 메시지가 표시되거나 엔진 INNODB 상태 표시의 출력이 표시될 수 있습니다. 교착 상태는 InnoDB 테이블에 심각한 문제가 아니며, 수정 조치가 필요하지 않은 경우가 많습니다. 두 개의 트랜잭션이 여러 테이블을 수정하기 시작하면
테이블을 서로 다른 순서로 처리하면 각 트랜잭션이 다른 트랜잭션을 기다리는 상태에 도달하여 둘 다 진행할 수 없게 됩니다. 교착 상태 감지가 활성화된 경우(기본값), MySQL은 이 상태를 즉시 감지하고 "작은" 트랜잭션을 취소(롤백)하여 다른 트랜잭션이 계속 진행될 수 있도록 합니다. 교착 상태 감지가 innodb_deadlock_detect 구성 옵션을 사용하여 비활성화되어 있는 경우, InnoDB는 교착 상태 발생 시 트랜잭션을 롤백하기 위해 innodb_lock_wait_timeout 설정에 의존합니다.

어느 쪽이든, 교착 상태로 인해 강제로 취소된 트랜잭션을 다시 시작하려면 애플리케이션에 오류 처리 로직이 필요합니다. 이전과 동일한 SQL 문을 다시 발행하면 원래의 타이밍 문제가 더 이상 적용되지 않습니다. 다른 트랜잭션이 이미 완료되었으므로 트랜잭션을 계속 진행할 수 있거나, 다른 트랜잭션이 아직 진행 중이므로 트랜잭션이 완료될 때까지 기다립니다.

교착 상태 경고가 계속 발생하는 경우 애플리케이션 코드를 검토하여 일관된 방식으로 SQL 작업의 순서를 바꾸거나 트랜잭션을 단축할 수 있습니다. 테스트는 innodb_print_all_deadlocks 옵션을 활성화하면 엔진 INNODB 상태 표시 출력의 마지막 경고만 표시되는 것이 아니라 MySQL 오류 로그에서 모든 교착 상태 경고를 볼 수 있습니다.

자세한 내용은 섹션 15.7.5, "InnoDB의 교착 상태"를 참조하세요.

스토리지 레이아웃

`InnoDB` 테이블에서 최상의 성능을 얻으려면 스토리지 레이아웃과 관련된 여러 매개변수를 조정할 수 있습니다.

크기가 크고 자주 액세스되며 중요한 데이터를 포함하는 `MyISAM` 테이블을 변환하는 경우, `CREATE TABLE` 문의 `innodb_file_per_table` 및 `innodb_page_size` 변수와 `ROW_FORMAT` 및 `KEY_BLOCK_SIZE` 절을 조사하고 고려해야 합니다.

초기 실험에서 가장 중요한 설정은 `innodb_file_per_table`입니다. 기본값인 이 설정을 활성화하면 새 `InnoDB` 테이블이 암시적으로 테이블별 파일 테이블 공간에 생성됩니다. `InnoDB` 시스템 테이블 공간과 달리, 테이블별 파일 테이블 공간을 사용하면 테이블이 잘리거나 삭제될 때 운영 체제에서 디스크 공간을 회수할 수 있습니다. 테이블별 파일
테이블 스페이스는 동적 및 압축 행 형식과 테이블 압축, 긴 가변 길이 열을 위한 효율적인 오프페이지 저장소, 큰 인덱스 접두사 등의 관련 기능도 지원합니다. 자세한 내용은 섹션 15.6.3.2, "테이블별 파일 테이블 스페이스"를 참조하십시오.

여러 테이블과 모든 행 형식을 지원하는 공유 일반 테이블 스페이스에 InnoDB 테이블을 저장할 수도 있습니다. 자세한 내용은 섹션 15.6.3.3, "일반 테이블 스페이스"를 참조하십시오.

기존 테이블 변환

InnoDB가 아닌 테이블을 InnoDB를 사용하도록 변환하려면 ALTER TABLE을 사용합니다:

```
ALTER TABLE table_name ENGINE=InnoDB;
```

테이블 구조 복제

ALTER TABLE을 사용하는 대신 MyISAM 테이블의 복제본인 InnoDB 테이블을 만들 수 있습니다.
를 사용하여 변환을 수행하고 전환하기 전에 이전 테이블과 새 테이블을 나란히 테스트합니다.

동일한 열 및 인덱스 정의로 빈 InnoDB 테이블을 생성합니다. SHOW CREATE TABLE table_name\G를 사용하여 사용할 전체 CREATE TABLE 문을 확인합니다. ENGINE 절을 ENGINE=INNODB로 변경합니다.

데이터 전송

이전 섹션에서와 같이 생성한 빈 InnoDB 테이블에 대량의 데이터를 전송하려면 INSERT INTO *innodb_table* SELECT * FROM *myisam_table* ORDER BY *primary_key_columns*로 행을 삽입합니다.

데이터를 삽입한 후 InnoDB 테이블에 대한 인덱스를 생성할 수도 있습니다. 이전에는 InnoDB에서 새로운 보조 인덱스를 생성하는 것이 느린 작업이었지만, 이제 인덱스 생성 단계에서 상대적으로 적은 오버헤드로 데이터를 로드한 후에 인덱스를 생성할 수 있습니다.

보조 키에 고유 제약 조건이 있는 경우 가져오기 작업 중에 일시적으로 고유성 검사를 해제하여 테이블 가져오기 속도를 높일 수 있습니다:

```
SET 고유_체크=0;
... 가져오기 작업 ...
고유_체크=1로 설정합니다;
```

큰 테이블의 경우, InnoDB가 변경 버퍼를 사용하여 보조 인덱스 레코드를 일괄적으로 쓸 수 있으므로 디스크 I/O를 절약할 수 있습니다. 고유_체크는 스토리지 엔진이 중복 키를 무시하도록 허용하지만 요구하지는 않습니다.

삽입 프로세스를 더 잘 제어하기 위해 큰 테이블을 조각으로 삽입할 수 있습니다:

```
INSERT INTO newtable SELECT * FROM oldtable
   WHERE yourkey > something AND yourkey <= somethingelse;
```

모든 레코드가 삽입된 후 테이블 이름을 변경할 수 있습니다.

큰 테이블을 변환하는 동안 InnoDB 버퍼 풀의 크기를 늘려 디스크 I/O를 줄이세요. 일반적으로 권장되는 버퍼 풀 크기는 시스템 메모리의 50~75%입니다. InnoDB 로그 파일의 크기를 늘릴 수도 있습니다.

스토리지 요구 사항

변환 프로세스 중에 `InnoDB` 테이블에 데이터의 임시 복사본을 여러 개 만들려는 경우 테이블을 삭제할 때 디스크 공간을 확보할 수 있도록 테이블별 파일 테이블 공간에 테이블을 만드는 것이 좋습니다. `innodb_file_per_table` 구성 옵션을 사용하도록 설정하면(기본값) 새로 생성된 `InnoDB` 테이블은 암시적으로 테이블별 파일 테이블 공간에 생성됩니다.

`MyISAM` 테이블을 직접 변환하든 복제된 `InnoDB` 테이블을 생성하든, 프로세스 중에 이전 테이블과 새 테이블을 모두 저장할 수 있는 충분한 디스크 공간이 있는지 확인하세요. **`InnoDB` 테이블은 `MyISAM` 테이블보다 더 많은 디스크 공간이 필요합니다.** `ALTER TABLE` 작업에 공간이 부족하면 롤백이 시작되며, 디스크에 바인딩된 경우 몇 시간이 걸릴 수 있습니다. 삽입의 경우, `InnoDB는` 삽입 버퍼를 사용하여 보조 인덱스 레코드를 일괄적으로 인덱스에 병합합니다. 이렇게 하면 디스크 I/O를 많이 절약할 수 있습니다. 롤백의 경우 이러한 메커니즘이 사용되지 않으며, 롤백은 삽입보다 30배 더 오래 걸릴 수 있습니다.

런어웨이 롤백의 경우 데이터베이스에 중요한 데이터가 없는 경우 수백만 개의 디스크 I/O 작업이 완료될 때까지 기다리지 말고 데이터베이스 프로세스를 종료하는 것이 좋습니다. 전체 절차는 섹션 15.21.3, 'InnoDB 복구 강제 실행'을 참조하세요.

기본 키 정의

기본 키 절은 MySQL 쿼리의 성능과 테이블 및 인덱스의 공간 사용량에 영향을 미치는 중요한 요소입니다. 기본 키는 테이블의 행을 고유하게 식별합니다. 테이블의 모든 행에는 기본 키 값이 있어야 하며, 두 행이 동일한 기본 키 값을 가질 수 없습니다.

다음은 기본 키에 대한 가이드라인이며, 자세한 설명이 이어집니다.

- 각 테이블에 대해 기본 키를 선언합니다. 일반적으로 가장 중요한 컬럼은 `WHERE` 절을 사용하여 단일 행을 조회합니다.
- 나중에 `ALTER TABLE` 문을 통해 추가하는 대신 원래 `CREATE TABLE` 문에서 `PRIMARY KEY` 절을 선언합니다.
- 열과 데이터 유형을 신중하게 선택합니다. 문자나 문자열 열보다 숫자 열을 선호합니다.
- 사용할 다른 안정적이고 고유하며 널이 아닌 숫자 열이 없는 경우 자동 증가 열을 사용하는 것이 좋습니다.
- 기본 키 열의 값이 변경될 수 있는지 의심스러운 경우 자동 증가 열을 사용하는 것도 좋은 선택입니다. 기본 키 열의 값을 변경하는 작업은 테이블 내 및 각 보조 인덱스 내의 데이터를 재정렬해야 하는 등 비용이 많이 드는 작업입니다.

아직 기본 키가 없는 테이블에 기본 키를 추가하는 것이 좋습니다. 테이블의 최대 예상 크기를 기준으로 가장 작은 실제 숫자 유형을 사용합니다. 이렇게 하면 각 행을 약간 더 콤팩트하게 만들 수 있으므로 큰 테이블의 경우 상당한 공간을 절약할 수 있습니다. 테이블에 보조 인덱스가 있는 경우 기본 키 값이 각 보조 인덱스 항목에서 반복되므로 공간 절약 효과는 배가됩니다. 디스크의 데이터 크기를 줄이는 것 외에도, 기본 키가 작으면 버퍼 풀에 더 많은 데이터를 넣을 수 있어 모든 종류의 작업 속도가 빨라지고 동시성이 향상됩니다.

테이블에 이미 `VARCHAR`와 같은 더 긴 열에 기본 키가 있는 경우, 해당 열이 쿼리에서 참조되지 않더라도 부호 없는 새 `AUTO_INCREMENT` 열을 추가하고 기본 키를 해당 열로 전환하는 것을 고려하세요. 이 설계 변경으로 보조 인덱스에서 상당한 공간을 절약할 수 있습니다. 이전 기본 키 열을 `UNIQUE NOT NULL`로 지정하여 모든 열에서 중복 또는 null 값을 방지하는 등 `PRIMARY KEY` 절과 동일한 제약 조건을 적용할 수 있습니다.

여러 테이블에 관련 정보를 분산하는 경우 일반적으로 각 테이블은 기본 키에 동일한 열을 사용합니다. 예를 들어 인사 데이터베이스에는 각각 직원 번호가 기본 키인 여러 테이블이 있을 수 있습니다. 영업 데이터베이스에는 고객 번호가 기본 키인 일부 테이블과 주문 번호가 기본 키인 다른 테이블이 있을 수 있습니다. 기본 키를 사용하는 조회는 매우 빠르기 때문에 이러한 테이블에 대한 효율적인 조인 쿼리를 구성할 수 있습니다.

`PRIMARY KEY` 절을 완전히 생략하면 MySQL이 보이지 않는 절을 생성합니다. 6바이트 값으로 필요한 것

보다 길 수 있으므로 공간을 낭비할 수 있습니다. 이 값은 숨겨져 있으므로 쿼리에서 참조할 수 없습니다.

애플리케이션 성능 고려 사항

InnoDB의 안정성과 확장성 기능을 사용하려면 동급의 MyISAM 테이블보다 더 많은 디스크 스토리지가 필요합니다. 공간 활용도를 높이고, 결과 집합을 처리할 때 I/O 및 메모리 소비를 줄이며, 인덱스 조회를 효율적으로 사용하는 쿼리 최적화 계획을 개선하기 위해 열 및 인덱스 정의를 약간 변경할 수 있습니다.

기본 키에 숫자 ID 열을 설정한 경우, 특히 조인 쿼리에서 해당 값을 사용하여 다른 테이블의 관련 값과 상호 참조할 수 있습니다. 예를 들어, 국가 이름을 입력으로 받아 같은 이름을 검색하는 쿼리를 수행하는 대신 한 번 조회하여 국가 ID를 확인한 다음 다른 쿼리(또는 단일 조인 쿼리)를 수행하여 여러 테이블에서 관련 정보를 조회합니다.

고객 또는 카탈로그 품목 번호를 숫자 문자열로 저장하여 잠재적으로 몇 바이트를 사용하는 대신 숫자 ID로 변환하여 저장하고 쿼리할 수 있습니다. 4바이트 부호 없는 `INT` 열은 40억 개 이상의 항목을 인덱싱할 수 있습니다(미국식 10억의 의미: 1000억). 다양한 정수 타입의 범위는 11.1.2절, "정수 타입(정확한 값) - INTEGER, INT, SMALLINT, TINYINT, MEDIUMINT, BIGINT"를 참조하세요.

InnoDB 테이블과 연결된 파일 이해

`InnoDB` 파일은 `MyISAM` 파일보다 더 많은 주의와 계획이 필요합니다.

- `InnoDB` 시스템 테이블 스페이스를 나타내는 ibdata 파일을 삭제해서는 안 됩니다.
- `InnoDB` 테이블을 다른 서버로 이동하거나 복사하는 방법은 섹션 15.6.1.4, "InnoDB 테이블 이동 또는 복사"에 설명되어 있습니다.

15.6.1.6 InnoDB의 AUTO_INCREMENT 처리

`InnoDB`는 구성 가능한 잠금 메커니즘을 제공하여 `AUTO_INCREMENT` 열이 있는 테이블에 행을 추가하는 SQL 문의 확장성 및 성능을 크게 향상시킬 수 있습니다. `InnoDB` 테이블에서 `AUTO_INCREMENT` 메커니즘을 사용하려면 테이블에서 인덱싱된 `SELECT MAX(ai_col)` 조회와 동등한 작업을 수행하여 최대 열 값을 구할 수 있도록 일부 인덱스의 첫 번째 또는 유일한 열로 `AUTO_INCREMENT` 열을 정의해야 합니다. 인덱스가 `PRIMARY KEY` 또는 `UNIQUE`일 필요는 없지만 `AUTO_INCREMENT` 열에 중복되는 값을 방지하려면 이러한 인덱스 유형을 사용하는 것이 좋습니다.

이 섹션에서는 `AUTO_INCREMENT` 잠금 모드와 각 모드의 사용 의미에 대해 설명합니다.
`AUTO_INCREMENT` 잠금 모드 설정과 `InnoDB`가 `AUTO_INCREMENT` 카운터를 초기화하는 방법을 설명합니다.

- InnoDB AUTO_INCREMENT 잠금 모드
- InnoDB AUTO_INCREMENT 잠금 모드 사용 시사점
- InnoDB AUTO_INCREMENT 카운터 초기화
- 참고

InnoDB AUTO_INCREMENT 잠금 모드

이 섹션에서는 자동 증가 값을 생성하는 데 사용되는 자동 증가 잠금 모드와 각 잠금 모드가 복제에 미치는 영향에 대해 설명합니다. 자동 증가 잠금 모드는 시작 시 `innodb_autoinc_lock_mode` 변수를 사용하여 구성됩니다.

다음 용어는 `innodb_autoinc_lock_mode` 설정을 설명하는 데 사용됩니다:

- "삽입과 유사한" 문

  테이블에 새 행을 생성하는 모든 문(`INSERT`, `INSERT ... SELECT`, `REPLACE`, `REPLACE ...` 포함)을 포함합니다. `SELECT` 및 `LOAD DATA`. "단순 삽입", "대량 삽입" 및 "혼합 모드" 삽입을 포함합니다.

- "간단한 삽입물"

  삽입할 행 수를 미리 결정할 수 있는 문(문이 처음 처리될 때). 여기에는 중첩된 하위 쿼리가 없는 단일 행 및 다중 행 `INSERT` 및 `REPLACE` 문이 포함되지만 `INSERT ... ON DUPLICATE KEY UPDATE`는 포함되지 않습니다.

- "대량 삽입"

  삽입할 행 수(및 필요한 자동 증분 값의 수)를 미리 알 수 없는 문입니다. 여기에는 `INSERT ... SELECT`, `REPLACE ... SELECT` 및 `LOAD DATA` 문이 포함되지만 일반 `INSERT` 문은 포함되지 않습니다. `InnoDB`는 각 행이 처리될 때 한 번에 하나씩 `AUTO_INCREMENT` 열에 새 값을 할당합니다.

- "혼합 모드 삽입"

  이는 새 행의 일부(전부는 아님)에 대한 자동 증가 값을 지정하는 "단순 삽입" 문입니다. 다음은 예제이며, 여기서 c1은 테이블 t1의 AUTO_INCREMENT 열입니다:

  ```
  INSERT INTO t1 (c1,c2) VALUES (1,'a', (NULL,'b', (5,'c', (NULL,'d');
  ```

  또 다른 유형의 "혼합 모드 삽입"은 INSERT ... ON DUPLICATE KEY UPDATE이며, 최악의 경우 INSERT 다음에 UPDATE가 뒤따르는데, 이 경우 업데이트 단계에서 AUTO_INCREMENT 열에 할당된 값이 사용되거나 사용되지 않을 수 있습니다.

innodb_autoinc_lock_mode 변수에는 세 가지 가능한 설정이 있습니다. 설정은 각각 "기존", "연속" 또는 "인터리브" 잠금 모드에 대해 0, 1 또는 2입니다. 인터리브 잠금 모드(innodb_autoinc_lock_mode=2)가 기본값입니다.

MySQL 8.2의 인터리브 잠금 모드의 기본 설정은 기본 복제 유형이 문 기반 복제에서 행 기반 복제로 변경된 것을 반영합니다. 문 기반 복제에서는 자동 증가 값이 할당되도록 연속 자동 증가 잠금 모드가 필요합니다. 주어진 SQL 문 시퀀스에 대해 예측 가능하고 반복 가능한 순서인 반면, 행 기반 복제는 SQL 문의 실행 순서에 민감하지 않습니다.

- innodb_autoinc_lock_mode = 0("기존" 잠금 모드)

  기존 잠금 모드는 innodb_autoinc_lock_mode 변수가 도입되기 전에 존재했던 것과 동일한 동작을 제공합니다. 기존 잠금 모드 옵션은 이전 버전과의 호환성, 성능 테스트 및 '혼합 모드 삽입' 관련 문제 해결을 위해 제공되며, 이는 의미상의 차이로 인해 발생할 수 있습니다.

  이 잠금 모드에서 모든 "INSERT 유사" 문은 자동 증가 열이 있는 테이블에 삽입할 때 특수한 테이블 수준 AUTO-INC 잠금을 얻습니다. 이 잠금은 일반적으로 트랜잭션의 끝이 아닌 문 끝까지 유지되어 주어진 INSERT 문 시퀀스에 대해 예측 가능하고 반복 가능한 순서로 자동 증가 값이 할당되고 주어진 문에서 할당된 자동 증가 값이 연속되도록 보장합니다.

  문 기반 복제의 경우, 이는 SQL 문이 복제본 서버에서 복제될 때 소스 서버에서와 동일한 값이 자동 증가 열에 사용됨을 의미합니다. 여러 INSERT 문의 실행 결과는 결정론적이며 복제본은 원본과 동일한 데이터를 복제합니다. 여러 INSERT 문에 의해 생성된 자동 증가 값이 인터리빙되는 경우, 두 개의 동시 INSERT 문의 결과는 비결정적이며, 문 기반 복제를 사용하여 복제본 서버로 안정적으로 전파될 수 없습니다.

  이를 명확히 하기 위해 이 표를 사용하는 예제를 살펴보겠습니다:

  ```
  CREATE TABLE t1 (
    c1 INT(11) NOT NULL AUTO_INCREMENT,
    c2 VARCHAR(10) DEFAULT NULL,
    PRIMARY KEY (c1)
  ) 엔진=InnoDB;
  ```

  두 개의 트랜잭션이 실행 중이며, 각 트랜잭션은 AUTO_INCREMENT 열이 있는 테이블에 행을 삽입한다고 가정합니다. 한 트랜잭션은 1000개의 행을 삽입하는 INSERT ... SELECT 문을 사용하고 있고, 다른 트

랜잭션은 하나의 행을 삽입하는 간단한 INSERT 문을 사용하고 있습니다:

```
Tx1: INSERT INTO t1 (c2) SELECT 1000 행을 다른 테이블에서 ... Tx2:
INSERT INTO t1 (c2) VALUES ('xxx');
```

InnoDB는 INSERT의 SELECT에서 얼마나 많은 행이 검색되는지 미리 알 수 없습니다.
문이 진행됨에 따라 자동 증가 값을 한 번에 하나씩 할당합니다.

테이블 수준 잠금이 문 끝에 유지되면 테이블 t1을 참조하는 INSERT 문은 한 번에 하나만 실행할 수 있으며 다른 문에 의한 자동 증가 수 생성은 인터리빙되지 않습니다. Tx1 INSERT ... SELECT 문에 의해 생성된 자동 증가 값은 연속적이며, INSERT 문에 의해 사용된 (단일) 자동 증가 값은

문은 어떤 문이 먼저 실행되느냐에 따라 Tx2에 사용된 모든 문보다 작거나 큽니다.

바이너리 로그에서 재생할 때 SQL 문이 동일한 순서로 실행되는 한(문 기반 복제를 사용하는 경우 또는 복구 시나리오에서), 결과는 Tx1 및 Tx2가 처음 실행되었을 때와 동일합니다. 따라서 문이 끝날 때까지 유지되는 테이블 수준 잠금은 자동 증가를 사용하는 `INSERT` 문을 문 기반 복제와 함께 사용하기에 안전합니다. 그러나 이러한 테이블 수준 잠금은 여러 트랜잭션이 동시에 삽입 문을 실행하는 경우 동시성 및 확장성을 제한합니다.

앞의 예에서 테이블 수준 잠금이 없는 경우 Tx2의 `INSERT`에 사용되는 자동 증가 열의 값은 문이 실행되는 정확한 시점에 따라 달라집니다. Tx2의 `INSERT`가 Tx1의 INSERT가 실행되는 동안(시작 전이나 완료 후가 아닌) 실행되는 경우 두 `INSERT` 문에 할당된 특정 자동 증가 값은 비결정적이며 실행에 따라 달라질 수 있습니다.

연속 잠금 모드에서 `InnoDB`는 행 수를 미리 알고 있는 '단순 삽입' 문에 테이블 수준의 `AUTO-INC` 잠금을 사용하지 않으면서도 문 기반 복제에 대한 결정론적 실행 및 안전성을 유지할 수 있습니다.

복구 또는 복제의 일부로 SQL 문을 재생하기 위해 바이너리 로그를 사용하지 않는 경우, 인터리브 잠금 모드를 사용하면 테이블 수준 `AUTO-INC` 잠금의 모든 사용을 제거하여 동시성 및 성능을 더욱 향상시킬 수 있지만, 문에 할당된 자동 증가 번호의 공백이 허용되고 동시에 실행되는 문에 할당된 번호가 인터리브 될 가능성이 있습니다.

- `innodb_autoinc_lock_mode = 1`("연속" 잠금 모드)

  이 모드에서 "대량 삽입"은 특수 `AUTO-INC` 테이블 수준 잠금을 사용하며 문이 끝날 때까지 잠금을 유지합니다. 이는 모든 `INSERT ... SELECT`, `REPLACE ... SELECT` 및 `LOAD DATA` 문에 적용됩니다. `AUTO-INC` 잠금이 유지되는 문은 한 번에 하나만 실행할 수 있습니다. 대량 삽입 작업의 소스 테이블이 대상 테이블과 다른 경우 대상 테이블의 `AUTO-INC` 잠금이 해제됩니다.
  은 소스 테이블에서 선택한 첫 번째 행에 공유 잠금이 수행된 후에 수행됩니다. 대량 삽입 작업의 원본과 대상이 동일한 테이블인 경우 선택한 모든 행에서 공유 잠금이 수행된 후 자동-`INC` 잠금이 수행됩니다.

  "단순 삽입"(삽입할 행 수를 미리 알고 있는 경우)은 문이 완료될 때까지가 *아니라* 할당 프로세스가 진행되는 동안에만 유지되는 뮤텍스(경량 잠금)의 제어 하에 필요한 수의 자동 증가 값을 가져와 테이블 수준의 `AUTO-INC` 잠금을 피할 수 있습니다. 다른 트랜잭션이 자동 증가 잠금을 보유하지 않는 한 테이블 수준의 자동 증가 잠금은 사용되지 않습니다. 다른 트랜잭션이 자동 삽입 잠금을 보유하고 있는 경우, "단순 삽입"은 마치 "대량 삽입"처럼 자동 삽입 잠금을 기다립니다.

  이 잠금 모드는 행 수를 미리 알 수 없고 문이 진행됨에 따라 자동 증가 번호가 할당되는 `INSERT` 문이 있는 경우 "`INSERT`와 유사한" 문에 할당된 모든 자동 증가 값이 연속적이며 문 기반 복제에 대해 작업이 안전하도록 보장합니다.

  간단히 말해, 이 잠금 모드는 확장성을 크게 개선하는 동시에 문 기반 복제와 함께 사용하기에 안전합니다. 또

한, "기존" 잠금 모드와 마찬가지로 특정 문에 할당된 자동 증가 번호는 *연속적입니다*. 한 가지 중요한 예외를 제외하고는 자동 증가를 사용하는 모든 문에 대해 "기존" 모드와 비교했을 때 의미론에 *변화가 없습니다.*

예외는 사용자가 여러 행으로 구성된 '단순 삽입'에서 일부 행(전부는 아님)에 대해 자동 증가 열에 명시적인 값을 제공하는 '혼합 모드 삽입'의 경우입니다. 이러한 삽입의 경우 `InnoDB`는 삽입할 행 수보다 더 많은 자동 증가 값을 할당합니다. 그러나 자동으로 할당된 모든 값은 가장 최근에 실행된 이전 문에서 생성된 자동 증가 값보다 연속적으로 생성되며, 따라서 이보다 높습니다. "초과" 숫자는 손실됩니다.

- `innodb_autoinc_lock_mode = 2`("인터리브" 잠금 모드)

  이 잠금 모드에서는 테이블 수준의 `AUTO-INC` 잠금을 사용하는 "`INSERT`와 유사한" 문이 없으며, 여러 문이 동시에 실행될 수 있습니다. 이 모드는 가장 빠르고 확장성이 뛰어난 잠금 모드이지만, SQL 문이 바이너리 로그에서 재생되는 문 기반 복제 또는 복구 시나리오를 사용할 때는 *안전하지 않습니다.*

  이 잠금 모드에서 자동 증가 값은 동시에 실행되는 모든 "`INSERT` 유사" 문에서 고유하고 단조롭게 증가하도록 보장됩니다. 그러나 여러 문이 동시에 숫자를 생성할 수 있기 때문에(즉, 숫자 할당이 문 간에 *인터리빙되므로*) 특정 문에 의해 삽입된 행에 대해 생성된 값이 연속적이지 않을 수 있습니다.

  실행되는 문이 삽입할 행 수를 미리 알고 있는 "단순 삽입"만인 경우 "혼합 모드 삽입"을 제외하고는 단일 문에 대해 생성되는 숫자에 간격이 발생하지 않습니다. 그러나 "대량 삽입"이 실행되는 경우 특정 문에 할당된 자동 증가 값에 간격이 있을 수 있습니다.

InnoDB AUTO_INCREMENT 잠금 모드 사용 시사점

- 복제와 함께 자동 증가 사용

  문 기반 복제를 사용하는 경우 `innodb_autoinc_lock_mode`를 0 또는 1로 설정하고 원본과 복제본에 동일한 값을 사용합니다. `innodb_autoinc_lock_mode` = 2("인터리브")를 사용하거나 원본과 복제본이 동일한 잠금 모드를 사용하지 않는 구성의 경우 자동 증가 값이 원본과 복제본에서 동일하게 유지되지 않습니다.

  행 기반 또는 혼합 형식 복제를 사용하는 경우 행 기반 복제는 SQL 문의 실행 순서에 민감하지 않으므로 모든 자동 증가 잠금 모드가 안전합니다(혼합 형식은 문 기반 복제에 안전하지 않은 모든 문에 대해 행 기반 복제를 사용함).

- "손실된" 자동 증가 값 및 시퀀스 간격

  모든 잠금 모드(0, 1 및 2)에서 자동 증가 값을 생성한 트랜잭션이 롤백되면 해당 자동 증가 값은 "손실"됩니다. 자동 증가 열에 대해 값이 생성되면 "`INSERT` 같은" 문이 완료되었는지 여부와 포함된 트랜잭션이 롤백되었는지 여부에 관계없이 롤백할 수 없습니다. 이렇게 손실된 값은 재사용되지 않습니다. 따라서 테이블의 자동 증가 열에 저장된 값에 공백이 있을 수 있습니다.

- `AUTO_INCREMENT` 열에 NULL 또는 0 지정하기

  모든 잠금 모드(0, 1, 2)에서 사용자가 `AUTO_INCREMENT` 열에 NULL 또는 0을 지정하면 `INSERT`를 사용하면 `InnoDB`는 값이 지정되지 않은 것처럼 행을 처리하고 해당 행에 대한 새 값을 생성합니다.

- `AUTO_INCREMENT` 열에 음수 값 할당하기

  모든 잠금 모드(0, 1, 2)에서 자동 증가 열에 음수 값을 할당하면 자동 증가 메커니즘의 동작이 정의되지 않습니다.

- `AUTO_INCREMENT` 값이 지정된 정수 유형에 대한 최대 정수보다 커지는 경우

모든 잠금 모드(0, 1, 2)에서 값이 지정된 정수 유형에 저장할 수 있는 최대 정수보다 커지는 경우 자동 증가 메커니즘의 동작이 정의되지 않습니다.

- '대량 삽입'에 대한 자동 증가 값의 차이

  `innodb_autoinc_lock_mode`가 0("기존") 또는 1("연속")로 설정된 경우 테이블 수준 `AUTO-INC` 잠금이 문이 끝날 때까지 유지되고 한 번에 하나의 문만 실행할 수 있으므로 특정 문에서 생성되는 자동 증가 값은 간격 없이 연속적입니다.

innodb_autoinc_lock_mode를 2("인터리브")로 설정하면 "대량 삽입"에 의해 생성된 자동 증분 값에 간격이 있을 수 있지만, 이는 "INSERT- like" 문을 동시에 실행하는 경우에만 해당합니다.

잠금 모드 1 또는 2의 경우 대량 삽입의 경우 각 문에 필요한 자동 증가 값의 정확한 수를 알 수 없고 과대 추정이 가능하므로 연속된 문 사이에 간격이 발생할 수 있습니다.

- "혼합 모드 삽입"에 의해 할당된 자동 증가 값

"단순 삽입"이 일부(전부는 아님) 결과 행에 대한 자동 증가 값을 지정하는 "혼합 모드 삽입"을 생각해 보십시오. 이러한 문은 잠금 모드 0, 1 및 2에서 다르게 동작합니다. 예를 들어, c1이 테이블 t1의 AUTO_INCREMENT 열이고 가장 최근에 자동 생성된 시퀀스 번호가 100이라고 가정합니다.

```
mysql> CREATE TABLE t1 (
    -> c1 INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    -> c2 CHAR(1)
    -> ) 엔진 = innodb;
```

이제 다음 "혼합 모드 삽입" 문을 고려해 보겠습니다:

```
mysql> INSERT INTO t1 (c1,c2) VALUES (1,'a', (NULL,'b', (5,'c', (NULL,'d');
```

innodb_autoinc_lock_mode가 0("기존")으로 설정된 경우, 4개의 새 행이 생성됩니다:

```
mysql> SELECT c1, c2 FROM t1 ORDER BY c2;
+-----+ ------+
| c1 | c2    |
+-----+ ------+
|1   | a     |
| 101 | b    |
|5   | c     |
| 102 | d    |
+-----+ ------+
```

자동 증가 값은 문 실행 시작 시 한 번에 모두 할당되는 것이 아니라 한 번에 하나씩 할당되므로 다음 사용 가능한 자동 증가 값은 103입니다. 이 결과는 동시에 실행 중인 "INSERT와 유사한" 문(모든 유형)이 있는지 여부에 관계없이 해당됩니다.

innodb_autoinc_lock_mode가 1("연속")로 설정된 경우 4개의 새 행도 마찬가지입니다:

```
mysql> SELECT c1, c2 FROM t1 ORDER BY c2;
+-----+ ------+
| c1 | c2    |
+-----+ ------+
|1   | a     |
| 101 | b    |
|5   | c     |
| 102 | d    |
+-----+ ------+
```

그러나 이 경우 문이 처리될 때 4개의 자동 증가 값이 할당되지만 2개만 사용되기 때문에 다음에 사용 가능한 자동 증가 값은 103이 아니라 105입니다. 이 결과는 동시에 실행 중인 "INSERT와 유사한" 문(모든 유형)이 있는지 여부에 관계없이 적용됩니다.

innodb_autoinc_lock_mode를 2("인터리브")로 설정하면 4개의 새 행이 생성됩니다:

```
mysql> SELECT c1, c2 FROM t1 ORDER BY c2;
+------+------+
| c1   | c2   |
+------+------+
|    1 | a    |
| NULL | b    |
|    5 | c    |
| NULL | d    |
```

```
+-----+ ------+
```

x와 *y의 값은*고유하며 이전에 생성된 어떤 행보다 큽니다. 그러나 *x* 및 *y의* 특정 값은 동시에 실행되는 문에서 생성된 자동 증가 값의 수에 따라 달라집니다.

마지막으로 가장 최근에 생성된 시퀀스 번호가 100일 때 발행되는 다음 문을 고려합니다:

```
mysql> INSERT INTO t1 (c1,c2) VALUES (1,'a', (NULL,'b', (101,'c', (NULL,'d');
```

innodb_autoinc_lock_mode 설정에서 이 문은 101이 행(NULL, 'b')에 할당되고 행(101, 'c')의 삽입이 실패하기 때문에 중복 키 오류 23000(쓸 수 없음, 테이블에 중복 키 있음)을 생성합니다.

- INSERT 문 시퀀스 중간에 AUTO_INCREMENT 열 값 수정하기

  현재 최대 자동 증가 값보다 큰 값으로 AUTO_INCREMENT 열 값을 수정하면 새 값이 유지되고 후속 INSERT 작업은 새롭고 더 큰 값부터 자동 증가 값을 할당합니다. 이 동작은 다음 예제에서 설명합니다:

```
mysql> CREATE TABLE t1 (
    -> c1 INT NOT NULL AUTO_INCREMENT,
    -> 기본 키(c1)
    -> ) 엔진 = InnoDB;

mysql> INSERT INTO t1 VALUES(0), (0), (3);

mysql> SELECT c1 FROM t1;
+----+
| c1 |
+----+
| 1 |
| 2 |
| 3 |
+----+

mysql> UPDATE t1 SET c1 = 4 WHERE c1 = 1;

mysql> SELECT c1 FROM t1;
+----+
| c1 |
+----+
| 2 |
| 3 |
| 4 |
+----+

mysql> INSERT INTO t1 VALUES(0);

mysql> SELECT c1 FROM t1;
+----+
| c1 |
+----+
| 2 |
| 3 |
| 4 |
| 5 |
+----+
```

InnoDB AUTO_INCREMENT 카운터 초기화

이 섹션에서는 InnoDB가 AUTO_INCREMENT 카운터를 초기화하는 방법을 설명합니다.

InnoDB 테이블에 AUTO_INCREMENT 열을 지정하는 경우 인메모리 테이블 객체에는 열에 새 값을 할당할

때 사용되는 자동 증가 카운터라는 특수 카운터가 포함되어 있습니다.

현재 최대 자동 증가 카운터 값은 변경될 때마다 다시 실행 로그에 기록되고 각 체크포인트의 데이터 사전에 저장되므로 서버 재시작 시에도 현재 최대 자동 증가 카운터 값이 영구적으로 유지됩니다.

정상 종료 후 서버를 재시작하면 InnoDB는 데이터 사전에 저장된 현재 최대 자동 증가 값을 사용하여 인메모리 자동 증가 카운터를 초기화합니다.

크래시 복구 중 서버가 다시 시작되면 InnoDB는 데이터 사전에 저장된 현재 최대 자동 증가 값을 사용하여 인메모리 자동 증가 카운터를 초기화하고 재실행 로그에서 마지막 체크포인트 이후 기록된 자동 증가 카운터 값을 검색합니다. 재실행 로그에 기록된 값이 인메모리 카운터 값보다 크면 재실행 로그에 기록된 값이 적용됩니다. 그러나 다음과 같은 경우에는
예기치 않은 서버 종료가 발생하면 이전에 할당된 자동 증가 값의 재사용을 보장할 수 없습니다. INSERT 또는 UPDATE 작업으로 인해 현재 최대 자동 증가 값이 변경될 때마다 새 값이 재실행 로그에 기록되지만, 재실행 로그가 디스크에 플러시되기 전에 예기치 않은 종료가 발생하면 서버가 다시 시작된 후 자동 증가 카운터가 초기화될 때 이전에 할당된 값이 재사용될 수 있습니다.

InnoDB가 자동 증가 카운터를 초기화하기 위해 SELECT MAX(ai_col) FROM *table_name* FOR UPDATE 문과 동등한 문을 사용하는 유일한 상황은 .cfg 메타데이터 파일 없이 테이블을 가져올 때입니다. 그렇지 않으면 현재 최대 자동 증가 카운터 값이 .cfg 메타데이터 파일(있는 경우)에서 읽혀집니다. 카운터 값 초기화 외에도, 테이블의 현재 최대 자동 증가 카운터 값을 결정하기 위해 테이블의 현재 최대 자동 증가 카운터 값을 결정하기 위해 SELECT MAX(ai_col) FROM *table_name* 문과 동일한 문이 사용됩니다. AUTO_INCREMENT
= *N* 문을 사용합니다. 예를 들어 일부 레코드를 삭제한 후 카운터 값을 더 작은 값으로 설정하려고 할 수 있습니다. 이 경우 테이블을 검색하여 새 카운터 값이 실제 현재 최대 카운터 값보다 작거나 같지 않은지 확인해야 합니다.

MySQL 5.7 이하에서는 서버를 다시 시작하면 초기 카운터 값을 설정하거나 기존 카운터 값을 변경하기 위해 각각 CREATE TABLE 또는 ALTER TABLE 문에서 사용할 수 있는 AUTO_INCREMENT = N 테이블 옵션의 효과가 취소됩니다. 서버를 다시 시작해도 AUTO_INCREMENT = N 테이블 옵션의 효과는 취소되지 않습니다. 자동 증가 카운터를 특정 값으로 초기화하거나 자동 증가 카운터 값을 더 큰 값으로 변경하면 서버를 다시 시작할 때 새 값이 유지됩니다.

참고

테이블 변경 ... AUTO_INCREMENT = N은 자동 증가 카운터 값을 현재 최대값보다 큰 값으로만 변경할 수 있습니다.

현재 최대 자동 증가 값이 유지되므로 이전에 할당된 값을 재사용할 수 없습니다.

자동 증가 카운터가 초기화되기 전에 SHOW TABLE STATUS 문이 테이블을 검사하는 경우 InnoDB는 테이블을 열고 데이터 사전에 저장된 현재 최대 자동 증가 값을 사용하여 카운터 값을 초기화합니다. 그런 다음 이 값은 나중에 삽입할 때 사용할 수 있도록 메모리에 저장됩니다.
또는 업데이트. 카운터 값의 초기화는 트랜잭션이 끝날 때까지 지속되는 테이블에 대한 일반 독점 잠금 읽기를

사용합니다. InnoDB는 사용자가 지정한 자동 증가 값이 0보다 큰 새로 생성된 테이블에 대해 자동 증가 카운터를 초기화할 때도 동일한 절차를 따릅니다.

자동 증가 카운터가 초기화된 후 행을 삽입할 때 자동 증가 값을 명시적으로 지정하지 않으면 InnoDB는 암시적으로 카운터를 증가시키고 새 값을 열에 할당합니다. 자동 증가 열 값을 명시적으로 지정하는 행을 삽입하고 그 값이 현재 최대 카운터 값보다 크면 카운터가 지정된 값으로 설정됩니다.

InnoDB는 서버가 실행되는 동안 인메모리 자동 증가 카운터를 사용합니다. 서버가 중지되었다가 다시 시작되면 앞서 설명한 대로 InnoDB는 자동 증가 카운터를 다시 초기화합니다.

자동 증가 오프셋 변수는 자동 증가의 시작점을 결정합니다.
열 값입니다. 기본 설정은 1입니다.

자동 증가_증가 변수는 연속되는 열 값 사이의 간격을 제어합니다. 기본 설정은 1입니다.

참고

AUTO_INCREMENT 정수 열의 값이 부족하면 후속 INSERT 작업에서 중복 키 오류가 반환됩니다. 이는 일반적인 MySQL 동작입니다.

15.6.2 색인

이 섹션에서는 InnoDB 인덱스와 관련된 주제를 다룹니다.

15.6.2.1 클러스터 및 보조 인덱스

각 InnoDB 테이블에는 행 데이터를 저장하는 클러스터된 인덱스라는 특수 인덱스가 있습니다. 일반적으로 클러스터된 인덱스는 기본 키와 동의어입니다. 쿼리, 삽입 및 기타 데이터베이스 작업에서 최상의 성능을 얻으려면 InnoDB가 클러스터된 인덱스를 사용하여 공통 조회 및 DML 작업을 최적화하는 방법을 이해하는 것이 중요합니다.

- 테이블에 기본 키를 정의하면 InnoDB는 이를 클러스터된 인덱스로 사용합니다. 각 테이블에 대해 기본 키를 정의해야 합니다. 기본 키를 사용할 논리적으로 고유하고 널이 아닌 열 또는 열 집합이 없는 경우 자동 증가 열을 추가합니다. 자동 증가 열 값은 고유하며 새 행이 삽입될 때 자동으로 추가됩니다.
- 테이블에 대해 PRIMARY KEY를 정의하지 않으면 InnoDB는 모든 키 열이 NOT NULL로 정의된 첫 번째 UNIQUE 인덱스를 클러스터링된 인덱스로 사용합니다.
- 테이블에 PRIMARY KEY 또는 적절한 고유 인덱스가 없는 경우 InnoDB는 행 ID 값을 포함하는 합성 열에 GEN_CLUST_INDEX라는 숨겨진 클러스터된 인덱스를 생성합니다. 행은 InnoDB가 할당하는 행 ID에 따라 정렬됩니다. 행 ID는 새 행이 삽입될 때마다 단조롭게 증가하는 6바이트 필드입니다. 따라서 행 ID에 의해 정렬된 행은 물리적으로 삽입된 순서대로 정렬됩니다.

클러스터된 인덱스로 쿼리 속도를 높이는 방법

클러스터된 인덱스를 통해 행에 액세스하면 인덱스 검색이 해당 행 데이터가 포함된 페이지로 바로 연결되므로 속도가 빠릅니다. 테이블이 큰 경우, 클러스터된 인덱스 아키텍처는 인덱스 레코드와 다른 페이지를 사용하여 행 데이터를 저장하는 스토리지 조직과 비교할 때 디스크 I/O 작업을 절약하는 경우가 많습니다.

보조 인덱스가 클러스터된 인덱스와 관련되는 방식

클러스터된 인덱스 이외의 인덱스를 보조 인덱스라고 합니다. InnoDB에서 보조 인덱스의 각 레코드에는 행에 대한 기본 키 열과 보조 인덱스에 지정된 열이 포함됩니다. InnoDB는 이 기본 키 값을 사용하여 클러스터된 인덱스에서 행을 검색합니다.

기본 키가 길면 보조 인덱스가 더 많은 공간을 사용하므로 기본 키가 짧은 것이 유리합니다.

InnoDB 클러스터 및 보조 인덱스를 활용하기 위한 지침은 8.3절. "최적화 및 인덱스"를 참조하세요.

15.6.2.2 InnoDB 인덱스의 물리적 구조

공간 인덱스를 제외한 `InnoDB` 인덱스는 B-트리 데이터 구조입니다. 공간 인덱스는 다차원 데이터 인덱싱을 위한 특수 데이터 구조인 R-트리를 사용합니다. 인덱스 레코드는 해당 B-트리 또는 R-트리 데이터 구조의 리프 페이지에 저장됩니다. 인덱스 페이지의 기본 크기는 16KB입니다. 페이지 크기는 MySQL 인스턴스가 초기화될 때 `innodb_page_size` 설정에 의해 결정됩니다. 섹션 15.8.1, "InnoDB 시작 구성"을 참조하세요.

`InnoDB` 클러스터 인덱스에 새 레코드가 삽입되면 `InnoDB`는 향후 인덱스 레코드의 삽입 및 업데이트를 위해 페이지의 1/16을 여유 공간으로 남겨두려고 시도합니다. 인덱스 레코드가 순차적(오름차순 또는 내림차순)으로 삽입되는 경우, 결과 인덱스 페이지는 약 15/16이 가득 차게 됩니다. 레코드가 임의의 순서로 삽입되면 페이지가 1/2에서 15/16까지 가득 차게 됩니다.

InnoDB는 B-트리 인덱스를 생성하거나 재구축할 때 대량 로드를 수행합니다. 이 인덱스 생성 방법을 정렬 인덱스 빌드라고 합니다. innodb_fill_factor 변수는 정렬된 인덱스 빌드 중에 채워지는 각 B-트리 페이지의 공간 비율을 정의하며, 나머지 공간은 향후 인덱스 성장을 위해 예약됩니다. 정렬 인덱스 빌드는 공간 인덱스에는 지원되지 않습니다. 자세한 내용은 섹션 15.6.2.3, "정렬된 인덱스 빌드"를 참조하십시오.
innodb_fill_factor를 100으로 설정하면 클러스터된 인덱스 페이지에서 공간의 1/16이 향후 인덱스 증가를 위해 비워집니다.

InnoDB 인덱스 페이지의 채우기 계수가 지정하지 않은 경우 기본적으로 50%인 MERGE_THRESHOLD 아래로 떨어지면 InnoDB는 인덱스 트리를 축소하여 페이지를 비우려고 시도합니다. MERGE_THRESHOLD 설정은 B-트리 및 R-트리 인덱스 모두에 적용됩니다. 자세한 내용은 섹션 15.8.11, "인덱스 페이지에 대한 병합 임계값 구성"을 참조하십시오.

15.6.2.3 정렬된 인덱스 빌드

InnoDB는 인덱스를 생성하거나 재구축할 때 인덱스 레코드를 한 번에 하나씩 삽입하는 대신 일괄 로드를 수행합니다. 이 인덱스 생성 방법을 정렬 인덱스 빌드라고도 합니다. 정렬된 인덱스 빌드는 공간 인덱스에는 지원되지 않습니다.

인덱스 빌드에는 세 단계가 있습니다. 첫 번째 단계에서는 클러스터된 인덱스가 스캔되고 인덱스 항목이 생성되어 정렬 버퍼에 추가됩니다. 정렬 버퍼가 가득 차면 항목이 정렬되고 임시 중간 파일에 기록됩니다. 이 프로세스를 "실행"이라고도 합니다. 두 번째 단계에서는 임시 중간 파일에 하나 이상의 실행이 기록된 상태에서 파일의 모든 항목에 대해 병합 정렬이 수행됩니다. 세 번째이자 마지막 단계에서는 정렬된 항목이 B-트리에 삽입되며, 이 마지막 단계는 멀티스레드로 진행됩니다.

정렬된 인덱스 빌드가 도입되기 전에는 인덱스 항목이 삽입 API를 사용하여 한 번에 한 레코드씩 B-트리에 삽입되었습니다. 이 방법에는 B-트리 커서를 열어 삽입 위치를 찾은 다음, 최적 삽입을 사용하여 B-트리 페이지에 항목을 삽입하는 작업이 포함되었습니다. 페이지로 인해 삽입이 실패한 경우
가 가득 차면 비관적 삽입이 수행되는데, 이 삽입은 B-트리 커서를 열고 필요에 따라 B-트리 노드를 분할 및 병합하여 항목을 넣을 공간을 찾습니다. 이러한 "하향식" 인덱스 구축 방법의 단점은 삽입 위치를 검색하는 데 드는 비용과 B-트리 노드를 계속 분할하고 병합해야 한다는 점입니다.

정렬된 인덱스 빌드는 인덱스 빌드에 "상향식" 접근 방식을 사용합니다. 이 접근 방식을 사용하면 B-트리의 모든 수준에서 가장 오른쪽에 있는 리프 페이지에 대한 참조가 유지됩니다. 필요한 B-트리 깊이에 있는 가장 오른쪽 리프 페이지가 할당되고 항목이 정렬된 순서에 따라 삽입됩니다. 리프 페이지가 가득 차면 노드 포인터가 부모 페이지에 추가되고 다음 삽입을 위해 형제 리프 페이지가 할당됩니다.
이 프로세스는 모든 항목이 삽입될 때까지 계속되며, 이로 인해 루트 수준까지 삽입될 수 있습니다. 형제 페이지가 할당되면 이전에 고정된 리프 페이지에 대한 참조가 해제되고 새로 할당된 리프 페이지가 가장 오른쪽 리프 페이지이자 새로운 기본 삽입 위치가 됩니다.

향후 인덱스 성장을 위한 B-트리 페이지 공간 확보

향후 인덱스 증가에 대비하여 공간을 확보하려면 `innodb_fill_factor` 변수를 사용하여 B-tree 페이지 공간의 일정 비율을 예약할 수 있습니다. 예를 들어, `innodb_fill_factor`를 80으로 설정하면 정렬된 인덱스 빌드 중에 B-트리 페이지 공간의 20%를 예약합니다. 이 설정는 B-트리 리프 페이지와 비리프 페이지 모두에 적용됩니다. 텍스트 또는 블롭 항목에 사용되는 외부 페이지에는 적용되지 않습니다. `innodb_fill_factor` 값은 엄격한 제한이 아닌 힌트로 해석되므로 예약된 공간의 양이 구성한 것과 정확히 일치하지 않을 수 있습니다.

정렬된 인덱스 빌드 및 전체 텍스트 인덱스 지원

전체 텍스트 인덱스에 대해 정렬된 인덱스 빌드가 지원됩니다. 이전에는 전체 텍스트 인덱스에 항목을 삽입하는 데 SQL을 사용했습니다.

정렬된 인덱스 빌드 및 압축 테이블

압축 테이블의 경우 이전 인덱스 생성 방법에서는 압축 페이지와 압축되지 않은 페이지 모두에 항목을 추가했습니다. 수정 로그(압축된 페이지의 여유 공간을 나타냄)의 경우

가 가득 차면 압축된 페이지가 다시 압축됩니다. 공간 부족으로 인해 압축에 실패하면 페이지가 분할됩니다. 정렬된 인덱스 빌드에서는 항목이 압축되지 않은 페이지에만 추가됩니다. 압축되지 않은 페이지가 가득 차면 압축됩니다. 적응형 패딩 사용
를 사용하여 대부분의 경우 압축이 성공하는지 확인하지만 압축에 실패하면 페이지가 분할되고 압축이 다시 시도됩니다. 이 프로세스는 압축이 성공할 때까지 계속됩니다. B-Tree 페이지 압축에 대한 자세한 내용은 섹션 15.9.1.5, "InnoDB 테이블의 압축 작동 방식"을 참조하십시오.

정렬된 인덱스 빌드 및 로깅 재실행

정렬된 인덱스 빌드 중에는 재실행 로깅이 비활성화됩니다. 대신, 인덱스 빌드가 예기치 않은 종료 또는 실패를 견딜 수 있도록 보장하는 체크포인트가 있습니다. 이 체크포인트는 모든 더티 페이지를 디스크에 강제로 씁니다. 정렬된 인덱스 빌드 중에 페이지 클리너 스레드는 체크포인트 작업을 빠르게 처리할 수 있도록 주기적으로 더러운 페이지를 플러시하도록 신호를 보냅니다. 일반적으로 페이지 클리너 스레드는 깨끗한 페이지의 수가 설정된 임계값 아래로 떨어지면 더러운 페이지를 플러시합니다. 정렬된 인덱스 빌드의 경우, 더티 페이지는 즉시 플러시되어 체크포인트 오버헤드를 줄이고 I/O 및 CPU 활동을 병렬화합니다.

정렬된 인덱스 빌드 및 옵티마이저 통계

정렬된 인덱스 빌드는 이전 인덱스 생성 방법에서 생성된 것과 다른 최적화 도구 통계를 생성할 수 있습니다. 워크로드 성능에는 영향을 미치지 않을 것으로 예상되는 통계의 차이는 인덱스를 채우는 데 사용되는 알고리즘이 다르기 때문입니다.

15.6.2.4 InnoDB 전체 텍스트 인덱스

전체 텍스트 인덱스는 텍스트 기반 열(`CHAR`, `VARCHAR` 또는 `TEXT` 열)에 생성되어 해당 열에 포함된 데이터에 대한 쿼리 및 DML 작업의 속도를 높입니다.

전체 텍스트 인덱스는 `CREATE TABLE` 문의 일부로 정의되거나 다음을 사용하여 기존 테이블에 추가됩니다. `테이블 변경` 또는 `인덱스 생성`.

전체 텍스트 검색은 `MATCH() ... AGAINST` 구문을 사용하여 수행됩니다. 사용법에 대한 자세한 내용은 섹션 12.9, "전체 텍스트 검색 함수"를 참조하십시오.

이 섹션의 다음 항목에서는 `InnoDB` 전체 텍스트 인덱스에 대해 설명합니다:

- InnoDB 전체 텍스트 인덱스 디자인
- InnoDB 전체 텍스트 인덱스 테이블
  - InnoDB 전체 텍스트 인덱스 캐시
  - InnoDB 전체 텍스트 인덱스 DOC_ID 및 FTS_DOC_ID 열
  - InnoDB 전체 텍스트 인덱스 삭제 처리

- InnoDB 전체 텍스트 인덱스 트랜잭션 처리
- InnoDB 전체 텍스트 인덱스 모니터링

InnoDB 전체 텍스트 인덱스 디자인

`InnoDB` 전체 텍스트 인덱스는 반전 인덱스 디자인으로 되어 있습니다. 반전 인덱스는 단어 목록과 각 단어에 대해 해당 단어가 나타나는 문서 목록을 저장합니다. 근접 검색을 지원하기 위해 각 단어의 위치 정보도 바이트 오프셋으로 저장됩니다.

InnoDB 전체 텍스트 인덱스 테이블

`InnoDB` 전체 텍스트 인덱스가 생성되면 다음 예제와 같이 인덱스 테이블 집합이 생성됩니다:

```
mysql> CREATE TABLE opening_lines (
       id INT 서명되지 않은 자동_인크립션이 아닌 기본 키,
       opening_line TEXT(500),
       author VARCHAR(200),
       title VARCHAR(200),
       전체 텍스트 idx (여는_줄)
       ) 엔진=InnoDB;

mysql> SELECT table_id, name, space from INFORMATION_SCHEMA.INNODB_TABLES
       WHERE name LIKE 'test/%';
+----------+--------------------------------------------------------+ -------------+
| 테이블_ID | 이름 |                                                          공간 |
+----------+--------------------------------------------------------+ -------------+
|       333 | test/fts_0000000000000147_00000000000001c9_index_1   |    289 |
|       334 | test/fts_0000000000000147_00000000000001c9_index_2   |    290 |
|       335 | test/fts_0000000000000147_00000000000001c9_index_3   |    291 |
|       336 | test/fts_0000000000000147_00000000000001c9_index_4   |    292 |
|       337 | test/fts_0000000000000147_00000000000001c9_index_5   |    293 |
|       338 | test/fts_0000000000000147_00000000000001c9_index_6   |    294 |
|       330 | test/fts_0000000000000147_being_deleted              |    286 |
|       331 | test/fts_0000000000000147_being_deleted_cache        |    287 |
|       332 | test/fts_0000000000000147_config                     |    288 |
|       328 | test/fts_0000000000000147_deleted                    |    284 |
|       329 | test/fts_0000000000000147_deleted_cache              |    285 |
|327    | 테스트/개방_라인                                                 |283
|
+----------+--------------------------------------------------------+ -------------+
```

처음 여섯 개의 인덱스 테이블은 반전된 인덱스를 구성하며 보조 인덱스 테이블이라고 합니다. 들어오는 문서가 토큰화되면 개별 단어("토큰"이라고도 함)가 위치 정보 및 연결된 DOC_ID와 함께 인덱스 테이블에 삽입됩니다. 단어는 단어의 첫 번째 문자의 문자 집합 정렬 가중치에 따라 6개의 인덱스 테이블 간에 완전히 정렬되고 분할됩니다.

반전된 인덱스는 병렬 인덱스 생성을 지원하기 위해 6개의 보조 인덱스 테이블로 분할됩니다. 기본적으로 두 개의 스레드가 단어 및 관련 데이터를 토큰화, 정렬 및 인덱스 테이블에 삽입합니다. 이 작업을 수행하는 스레드 수는 innodb_ft_sort_pll_degree 변수를 사용하여 구성할 수 있습니다. 큰 테이블에 전체 텍스트 인덱스를 만들 때는 스레드 수를 늘리는 것을 고려하세요.

보조 인덱스 테이블 이름 앞에는 fts_가 붙고 뒤에는 *index_#가 붙습니다*. 각 보조 인덱스 테이블은 인덱싱된 테이블의 table_id와 일치하는 보조 인덱스 테이블 이름의 16진수 값으로 인덱싱된 테이블과 연결됩니다. 예를 들어, test/opening_lines 테이블의 table_id는 327이며, 이 테이블의 16진수 값은 0x147입니다. 앞의 예에서와 같이, "147" 16진수 값은 test/opening_lines 테이블과 연관된 보조 인덱스 테이블의 이름에 나타납니다.

전체 텍스트 인덱스의 index_id를 나타내는 16진수 값은 보조 인덱스 테이블 이름에도 나타납니다. 예를 들어, 보조 테이블 이름 test/에서 fts_0000000000000147_00000000000001c9_index_1에서 16진수 값 1c9의 소수점 값은 457입니다. 이 값(457)에 대한 정보 스키마 INNODB_INDEXES 테이블을 쿼리하여 opening_lines 테이블(idx)에 정의된 인덱스를 식별할 수 있습니다.

```
mysql> SELECT index_id, name, table_id, space FROM INFORMATION_SCHEMA.INNODB_INDEXES
       WHERE index_id=457;
+----------+------+----------+-----------+
| 인덱스_ID | 이름 | 테이블_ID | 공간 |
+----------+------+----------+-----------+
|457 |  IDX |              327 |    283 |
+----------+------+----------+-----------+
```

기본 테이블이 파일 단위 테이블 스페이스에 생성된 경우 인덱스 테이블은 자체 테이블 스페이스에 저장됩니다. 그렇지 않으면 인덱스 테이블은 인덱싱된 테이블이 있는 테이블 공간에 저장됩니다.

앞의 예에 표시된 다른 인덱스 테이블은 공통 인덱스 테이블이라고 하며 삭제 처리 및 전체 텍스트 인덱스의 내부 상태 저장에 사용됩니다. 각 전체 텍스트 인덱스에 대해 생성되는 반전 인덱스 테이블과 달리 이 테이블 집합은 특정 테이블에 생성된 모든 전체 텍스트 인덱스에 공통입니다.

전체 텍스트 인덱스가 삭제되더라도 일반 인덱스 테이블은 유지됩니다. 전체 텍스트 인덱스가 삭제되면 인덱스에 대해 생성된 `FTS_DOC_ID` 열은 유지되며, 인덱스에서 `FTS_DOC_ID` 열을 사용하려면 이전에 인덱싱된 테이블을 다시 빌드해야 합니다. `FTS_DOC_ID` 열을 관리하려면 공통 인덱스 테이블이 필요합니다.

- `FTS_*_삭제` 및 `FTS_*_삭제된_캐시`

  삭제되었지만 아직 전체 텍스트 인덱스에서 데이터가 제거되지 않은 문서에 대한 문서 ID(DOC_ID)를 포함합니다. `fts_*_deleted_cache`는 `fts_*_deleted` 테이블의 인메모리 버전입니다.

- `FTS_*_BEING_DELETED` 및 `FTS_*_BEING_DELETED_CACHE`

  삭제된 문서와 현재 전체 텍스트 인덱스에서 데이터가 제거되는 과정에 있는 문서의 문서 ID(DOC_ID)를 포함합니다. `fts_*_being_deleted_cache` 테이블은 `fts_*_being_deleted` 테이블의 인메모리 버전입니다.

- `fts_*_config`

  전체 텍스트 인덱스의 내부 상태에 대한 정보를 저장합니다. 가장 중요한 것은 구문 분석이 완료되어 디스크에 플러시된 문서를 식별하는 `FTS_SYNCED_DOC_ID`를 저장한다는 점입니다. 충돌 복구 시 `FTS_SYNCED_DOC_ID` 값은 디스크에 플러시되지 않은 문서를 식별하는 데 사용되므로 문서를 다시 구문 분석하여 전체 텍스트 인덱스 캐시에 다시 추가할 수 있습니다. 이 테이블의 데이터를 보려면 정보 스키마 `INNODB_FT_CONFIG` 테이블을 쿼리하세요.

InnoDB 전체 텍스트 인덱스 캐시

문서가 삽입되면 문서가 토큰화되고 개별 단어와 관련 데이터가 전체 텍스트 인덱스에 삽입됩니다. 이 프로세스는 작은 문서의 경우에도 보조 인덱스 테이블에 수많은 작은 삽입을 초래할 수 있으므로 이러한 테이블에 대한 동시 액세스가 논쟁의 대상이 될 수 있습니다. To
이 문제를 방지하기 위해 `InnoDB`는 전체 텍스트 인덱스 캐시를 사용하여 최근에 삽입된 행에 대한 인덱스 테이블 삽입을 일시적으로 캐시합니다. 이 인메모리 캐시 구조는 캐시가 가득 찰 때까지 삽입을 유지한 다음 디스크(보조 인덱스 테이블)로 일괄 플러시합니다. 정보 스키마 `INNODB_FT_INDEX_CACHE` 테이블을 쿼리하여 최근에 삽입된 행에 대한 토큰화된 데이터를 볼 수 있습니다.

캐싱 및 일괄 플러싱 동작은 바쁜 삽입 및 업데이트 시간 동안 동시 액세스 문제를 일으킬 수 있는 보조 인덱스 테이블의 빈번한 업데이트를 방지합니다. 일괄 처리 기술은 또한 동일한 단어에 대한 여러 번의 삽입을 방지하고 중복 항목을 최소화합니다. 각 단어를 개별적으로 플러시하는 대신, 동일한 단어에 대한 삽입이 병합되어 단일 항목으로 디스크에 플러시되므로 보조 인덱스 테이블을 최대한 작게 유지하면서 삽입 효율성을 개선할 수 있습니다.

`innodb_ft_cache_size` 변수는 전체 텍스트 인덱스 캐시 크기(테이블 단위로)를 구성하는 데 사용되며, 이는 전체 텍스트 인덱스 캐시가 플러시되는 빈도에 영향을 줍니다. 또한 `innodb_ft_total_cache_size` 변수를 사용하여 지정된 인스턴스의 모든 테이블에 대한 전

체 텍스트 인덱스 캐시 크기 제한을 전역으로 정의할 수도 있습니다.

전체 텍스트 인덱스 캐시는 보조 인덱스 테이블과 동일한 정보를 저장합니다. 그러나 전체 텍스트 인덱스 캐시는 최근에 삽입된 행에 대한 토큰화된 데이터만 캐시합니다. 이미 디스크(보조 인덱스 테이블)로 플러시된 데이터는 쿼리 시 전체 텍스트 인덱스 캐시로 다시 가져오지 않습니다. 보조 인덱스 테이블의 데이터는 직접 쿼리되며, 보조 인덱스 테이블의 결과는 반환되기 전에 전체 텍스트 인덱스 캐시의 결과와 병합됩니다.

InnoDB 전체 텍스트 인덱스 DOC_ID 및 FTS_DOC_ID 열

InnoDB는 DOC_ID라는 고유한 문서 식별자를 사용하여 전체 텍스트 인덱스의 단어를 해당 단어가 나타나는 문서 레코드에 매핑합니다. 매핑에는 인덱싱된 테이블에 FTS_DOC_ID 열이 필요합니다. FTS_DOC_ID 열이 정의되어 있지 않으면 InnoDB는 전체 텍스트 인덱스가 생성될 때 숨겨진 FTS_DOC_ID 열을 자동으로 추가합니다. 다음 예제는 이 동작을 보여줍니다.

다음 테이블 정의에는 FTS_DOC_ID 열이 포함되어 있지 않습니다:

```
mysql> CREATE TABLE opening_lines (
       id INT 서명되지 않은 자동_인크립션이 아닌 기본 키,
       opening_line TEXT(500),
       author VARCHAR(200),
       title VARCHAR(200)
       ) 엔진=InnoDB;
```

CREATE FULLTEXT INDEX 구문을 사용하여 테이블에 전체 텍스트 인덱스를 생성하면 InnoDB가 FTS_DOC_ID 열을 추가하기 위해 테이블을 다시 작성 중임을 알리는 경고가 반환됩니다.

```
mysql> CREATE FULLTEXT INDEX idx ON opening_lines(opening_line);
쿼리 확인, 영향을 받는 행 0개, 경고 1개(0.19초)
레코드: 0 중복: 0 경고: 1

mysql> SHOW WARNINGS;
+---------+------+ ----------------------------------------------------+
| 레벨|     코드 | 메시지                                                |
+---------+------+ ----------------------------------------------------+
| 경고 | 124 | FTS_DOC_ID 열을 추가하기 위해 InnoDB 테이블 재구축 중 | 경고
+---------+------+ ----------------------------------------------------+
```

ALTER TABLE을 사용하여 FTS_DOC_ID 열이 없는 테이블에 전체 텍스트 인덱스를 추가하는 경우에도 동일한 경고가 반환됩니다. CREATE TABLE 시 전체 텍스트 인덱스를 생성할 때 FTS_DOC_ID 열을 지정하지 않으면 InnoDB는 경고 없이 숨겨진 FTS_DOC_ID 열을 추가합니다.

테이블을 생성할 때FTS_DOC_ID 열을 정의하는 것이 이미 데이터가 로드된 테이블에 전체 텍스트 인덱스를 생성하는 것보다 비용이 적게 듭니다. 데이터를 로드하기 전에 테이블에 FTS_DOC_ID 열을 정의하면 새 열을 추가하기 위해 테이블과 해당 인덱스를 다시 빌드할 필요가 없습니다. 전체 텍스트 인덱스 생성 성능에 신경 쓰지 않는 경우 FTS_DOC_ID 열을 생략하여 InnoDB가 자동으로 생성하도록 할 수 있습니다. InnoDB는 FTS_DOC_ID 열에 고유 인덱스(FTS_DOC_ID_INDEX)와 함께 숨겨진 FTS_DOC_ID 열을생성합니다. 고유한 FTS_DOC_ID 열을 생성하려면 다음 예제에서와 같이 열을 BIGINT UNSIGNED NOT NULL로 정의하고 이름을 FTS_DOC_ID(모두 대문자)로 지정해야 합니다:

참고

FTS_DOC_ID 열은 정의할 필요가 없습니다.
열을 사용하지 않아도 되지만, 이렇게 하면 데이터를 더 쉽게 로드할 수 있습니다.

```
mysql> CREATE TABLE opening_lines (
       FTS_DOC_ID BIGINT 부호 없는 자동 증가 NOT NULL 기본 키,
       opening_line TEXT(500),
       author VARCHAR(200),
       title VARCHAR(200)
       ) 엔진=InnoDB;
```

FTS_DOC_ID 열을 직접 정의하기로 선택한 경우 비어 있거나 중복된 값이 발생하지 않도록 열을 관리할 책임이 있습니다. FTS_DOC_ID 값은 재사용할 수 없으므로 FTS_DOC_ID 값을 계속 늘려야 합니다.

선택 사항으로, 필요한 고유 FTS_DOC_ID_INDEX(모두 대문자)를 생성할 수 있습니다.
FTS_DOC_ID 열입니다.

```
mysql> CREATE UNIQUE INDEX FTS_DOC_ID_INDEX on opening_lines(FTS_DOC_ID);
```

FTS_DOC_ID_INDEX를 생성하지 않으면 InnoDB가 자동으로 생성합니다.

참고

FTS_DOC_ID_INDEX는 내림차순 인덱스로 정의할 수 없습니다.
`InnoDB` SQL 구문 분석기는 내림차순 인덱스를 사용하지 않습니다.

가장 많이 사용된 `FTS_DOC_ID` 값과 새 `FTS_DOC_ID` 값 사이에 허용되는 간격은 65535입니다.

테이블 재구성을 피하기 위해 전체 텍스트 인덱스를 삭제할 때 `FTS_DOC_ID` 열이 유지됩니다.

InnoDB 전체 텍스트 인덱스 삭제 처리

전체 텍스트 인덱스 열이 있는 레코드를 삭제하면 보조 인덱스 테이블에서 수많은 작은 삭제가 발생하여 이러한 테이블에 대한 동시 액세스가 논쟁의 대상이 될 수 있습니다. 이 문제를 방지하기 위해 색인 테이블에서 레코드가 삭제될 때마다 삭제된 문서의 DOC_ID가 특수 FTS_*_DELETED 테이블에 기록되며, 색인된 레코드는 전체 텍스트 색인에 남아 있습니다. 이전
쿼리 결과를 반환하는 경우, FTS_*_DELETED 테이블의 정보를 사용하여 삭제된 DOC_ID를 필터링합니다. 이 설계의 장점은 삭제가 빠르고 저렴하다는 것입니다. 단점은 레코드를 삭제한 후 인덱스의 크기가 즉시 줄어들지 않는다는 것입니다. 삭제된 레코드에 대한 전체 텍스트 인덱스 항목을 제거하려면 인덱싱된 테이블에서 innodb_optimize_fulltext_only=ON으로 테이블 최적화를 실행하여 전체 텍스트 인덱스를 재구축합니다. 자세한 내용은 InnoDB 전체 텍스트 인덱스 최적화를 참조하세요.

InnoDB 전체 텍스트 인덱스 트랜잭션 처리

InnoDB 전체 텍스트 인덱스는 캐싱 및 일괄 처리 동작으로 인해 특별한 트랜잭션 처리 특성을 가지고 있습니다. 특히, 전체 텍스트 인덱스의 업데이트 및 삽입은 트랜잭션 커밋 시점에 처리되므로 전체 텍스트 검색에서는 커밋된 데이터만 볼 수 있습니다. 다음 예제는 이 동작을 보여줍니다. 전체 텍스트 검색은 삽입된 줄이 커밋된 후에만 결과를 반환합니다.

```
mysql> CREATE TABLE opening_lines (
       id INT 서명되지 않은 자동_인크립션이 아닌 기본 키,
       opening_line TEXT(500),
       author VARCHAR(200),
       title VARCHAR(200),
       전체 텍스트 idx (여는_줄)
       ) 엔진=InnoDB;

mysql> BEGIN;

mysql> INSERT INTO opening_lines(opening_line,author,title) VALUES
       ('Call me Ishmael.','Herman Melville','Moby-Dick'),
       ('비명이 하늘을 가로지른다.', '토마스 핀천', '중력의 무지개'), ('나는 투명인간이다.', '랄프
       엘리슨', '인비저블맨'),
       ('지금 어디? 지금 누구? 지금 언제?', '사무엘 베케트', '이름 없는 자'), ('첫눈에 반
       했어.', '조셉 헬러', '캐치-22'),
       ('이 모든 일은 어느 정도 일어났습니다.','커트 보네거트','슬러터하우스-파이브'),
       ('댈러웨이 부인이 직접 꽃을 사겠다고 했어요.','버지니아 울프','댈러웨이 부인'), ('태우는 것은 즐거움이었다.','
       레이 브래드버리','화씨 451도');

mysql> SELECT COUNT(*) FROM opening_lines WHERE MATCH(opening_line) AGAINST('Ishmael');
+----------+
| COUNT(*) |
+----------+
|0 |
+----------+

mysql> COMMIT;

mysql> SELECT COUNT(*) FROM opening_lines WHERE MATCH(opening_line) AGAINST('Ishmael');
+----------+
| COUNT(*) |
+----------+
|1 |
```

```
+----------+
```

InnoDB 전체 텍스트 인덱스 모니터링

다음 `INFORMATION_SCHEMA` 테이블을 쿼리하여 `InnoDB` 전체 텍스트 인덱스의 특수 텍스트 처리 측면을 모니터링하고 검사할 수 있습니다:

- `innodb_ft_config`
- `innodb_ft_index_table`
- `innodb_ft_index_cache`

- INNODB_FT_기본_스톱워드
- innodb_ft_deleted
- INNODB_FT_BEING_DELETED

또한 INNODB_INDEXES 및 INNODB_TABLES.

자세한 내용은 섹션 15.15.4, "InnoDB 정보_SCHEMA 전체 텍스트 인덱스 테이블"을 참조하세요.

15.6.3 테이블 공간

이 섹션에서는 InnoDB 테이블 스페이스와 관련된 주제를 다룹니다.

15.6.3.1 시스템 테이블 스페이스

시스템 테이블 스페이스는 변경 버퍼의 저장 영역입니다. 테이블이 파일별 테이블 공간이나 일반 테이블 공간이 아닌 시스템 테이블 공간에 생성된 경우 테이블 및 인덱스 데이터도 포함될 수 있습니다.

시스템 테이블스페이스에는 하나 이상의 데이터 파일이 있을 수 있습니다. 기본적으로 ibdata1이라는 단일 시스템 테이블스페이스 데이터 파일이 데이터 디렉터리에 생성됩니다. 시스템 테이블스페이스 데이터 파일의 크기와 개수는 innodb_data_file_path 시작 옵션으로 정의됩니다. 구성 정보는 시스템 테이블스페이스 데이터 파일 구성을 참조하세요.

시스템 테이블 스페이스에 대한 추가 정보는 섹션의 다음 항목에서 확인할 수 있습니다:

- 시스템 테이블 공간 크기 조정
- 시스템 테이블 공간에 원시 디스크 파티션 사용

시스템 테이블 공간 크기 조정

이 섹션에서는 시스템 테이블 스페이스의 크기를 늘리거나 줄이는 방법에 대해 설명합니다.

시스템 테이블 공간 크기 늘리기

시스템 테이블스페이스의 크기를 늘리는 가장 쉬운 방법은 자동 확장이 되도록 구성하는 것입니다. 이렇게 하려면 innodb_data_file_path 설정에서 마지막 데이터 파일에 자동 확장 속성을 지정하고 서버를 다시 시작하면 됩니다. 예를 들어

```
innodb_data_파일_경로=ibdata1:10M:자동 확장
```

자동 확장 속성을 지정하면 필요한 공간에 따라 데이터 파일의 크기가 8MB 단위로 자동으로 증가합니다. innodb_autoextend_increment 변수는 증분 크기를 제어합니다.

다른 데이터 파일을 추가하여 시스템 테이블 스페이스 크기를 늘릴 수도 있습니다. 이렇게 하려면

1. MySQL 서버를 중지합니다.
2. `innodb_data_file_path` 설정의 마지막 데이터 파일이 `autoextend` 속성으로 정의되어 있는 경우, 이를 제거하고 현재 데이터 파일 크기를 반영하도록 size 속성을 수정합니다. 지정할 적절한 데이터 파일 크기를 확인하려면 파일 시스템에서 파일 크기를 확인한 후 가장 가까운 MB 값으로 반올림합니다(여기서 MB는 1024 x 1024 바이트).
3. 선택적으로 자동 확장 속성을 지정하여 `innodb_data_file_path` 설정에 새 데이터 파일을 추가합니다. 자동 확장 특성은 `innodb_data_file_path` 설정의 마지막 데이터 파일에 대해서만 지정할 수 있습니다.
4. MySQL 서버를 시작합니다.

예를 들어 이 테이블스페이스에는 자동 확장되는 데이터 파일이 하나 있습니다:

```
innodb_data_home_dir =
innodb_data_파일_경로 = /ibdata/ibdata1:10M:자동 확장
```

시간이 지남에 따라 데이터 파일이 988MB로 증가했다고 가정합니다. 다음은 현재 데이터 파일 크기를 반영하도록 size 속성을 수정하고 50MB 자동 확장 데이터 파일을 새로 지정한 후의 innodb_data_file_path 설정입니다:

```
innodb_data_home_dir =
innodb_data_file_path = /ibdata/ibdata1:988M;/disk2/ibdata2:50M:autoextend
```

새 데이터 파일을 추가할 때 기존 파일 이름을 지정하지 마세요. 서버를 시작할 때 InnoDB가 새 데이터 파일을 생성하고 초기화합니다.

참고

기존 시스템 테이블스페이스 데이터 파일의 크기 속성을 변경하여 크기를 늘릴 수는 없습니다. 예를 들어 innodb_data_file_path를 ibdata1:10M:autoextend에서 ibdata1:12M:autoextend로 설정하면 서버를 시작할 때 다음과 같은 오류가 발생합니다:

```
[ERROR] [MY-012263] [InnoDB] 자동 확장 innodb_system
데이터 파일 './ibdata1'의 크기가 .cnf 파일에 지정된 크기(초기 768페이지, 최대 0(0이 아닌 경우 관련) 페이지)와 다른 640페이지(MB로 반내림)입니다!
```

이 오류는 기존 데이터 파일 크기(InnoDB 페이지에 표시됨)가 구성 파일에 지정된 데이터 파일 크기와 다르다는 것을 나타냅니다. 이 오류가 발생하면 이전 innodb_data_file_path 설정으로 복원하고 시스템 테이블 스페이스 크기 조정 지침을 참조하세요.

InnoDB 시스템 테이블 스페이스 크기 줄이기

기존 시스템 테이블스페이스의 크기를 줄이는 것은 지원되지 않습니다. 시스템 테이블스페이스를 더 작게 만들 수 있는 유일한 옵션은 백업에서 원하는 시스템 테이블스페이스 크기 구성으로 생성된 새 MySQL 인스턴스로 데이터를 복원하는 것입니다.

백업 생성에 대한 자세한 내용은 섹션 15.18.1, "InnoDB 백업"을 참조하세요.

새 시스템 테이블 스페이스의 데이터 파일 구성에 대한 자세한 내용은 시스템 테이블 스페이스 데이터 파일 구성을 참조하십시오. 시스템 테이블 스페이스 데이터 파일 구성을 참조하십시오.

큰 시스템 테이블 스페이스를 피하려면 데이터에 파일 단위 테이블 스페이스 또는 일반 테이블 스페이스를 사용하는 것이 좋습니다. 테이블별 파일 테이블스페이스는 기본 테이블스페이스 유형이며 InnoDB 테이블을 만들 때 암시적으로 사용됩니다. 시스템 테이블 스페이스와 달리 테이블별 파일 테이블 스페이스는 잘리거나 삭제될

때 디스크 공간을 운영 체제에 반환합니다. 자세한 내용은 섹션 15.6.3.2, "테이블별 파일 테이블 공간"을 참조하십시오. 일반 테이블 스페이스는 시스템 테이블 스페이스의 대안으로 사용할 수 있는 다중 테이블 테이블 스페이스입니다. 15.6.3.3절. "일반 테이블 스페이스"를 참조하십시오.

시스템 테이블 공간에 원시 디스크 파티션 사용

원시 디스크 파티션을 시스템 테이블스페이스 데이터 파일로 사용할 수 있습니다. 이 기술을 사용하면 파일 시스템 오버헤드 없이 Windows와 일부 Linux 및 Unix 시스템에서 버퍼링되지 않는 I/O를 사용할 수 있습니다. 원시 파티션을 사용하거나 사용하지 않고 테스트를 수행하여 원시 파티션이 시스템 성능을 향상시키는지 확인합니다.

원시 디스크 파티션을 사용하는 경우 MySQL 서버를 실행하는 사용자 ID에 해당 파티션에 대한 읽기 및 쓰기 권한이 있는지 확인합니다. 예를 들어, `mysql` 사용자로 서버를 실행하는 경우, 해당 파티션은 `mysql`에서 읽고 쓸 수 있어야 합니다. `memlock` 옵션으로 서버를 실행하는 경우 서버를 루트로 실행해야 하므로 루트에서 파티션을 읽고 쓸 수 있어야 합니다.

아래에 설명된 절차에는 옵션 파일 수정이 포함됩니다. 자세한 내용은 4.2.2.2절. "옵션 파일 사용하기"를 참조하세요.

Linux 및 유닉스 시스템에서 원시 디스크 파티션 할당하기

1. 새 데이터 파일을 생성할 때는 innodb_data_file_path 옵션의 데이터 파일 크기 바로 뒤에 newraw 키워드를 지정합니다. 파티션은 최소한 지정한 크기만큼 커야 합니다. InnoDB에서 1MB는 1024×1024바이트인 반면, 디스크 사양에서 1MB는 일반적으로 1,000,000바이트를 의미한다는 점에 유의하세요.

```
[mysqld]
innodb_data_home_dir=
innodb_data_파일_경로=/dev/hdd1:3Gnewraw;/dev/hdd2:2Gnewraw
```

2. 서버를 다시 시작합니다. InnoDB가 newraw 키워드를 인식하고 새 파티션을 초기화합니다. 그러나 아직 InnoDB 테이블을 만들거나 변경하지 마세요. 그렇지 않으면 다음에 서버를 다시 시작할 때 InnoDB가 파티션을 다시 초기화하여 변경 내용이 손실됩니다. (안전 조치로 InnoDB는 새로 만들기가 지정된 파티션이 있는 경우 사용자가 데이터를 수정할 수 없도록 합니다.)

3. InnoDB가 새 파티션을 초기화한 후 서버를 중지하고 데이터 파일 사양에서 newraw를 원시 데이터로 변경합니다:

```
[mysqld]
innodb_data_home_dir=
innodb_data_파일_경로=/dev/hdd1:3Graw;/dev/hdd2:2Graw
```

4. 서버를 다시 시작합니다. 이제 InnoDB에서 변경을 허용합니다.

Windows에서 원시 디스크 파티션 할당하기

Windows 시스템에서는 innodb_data_file_path 설정이 Windows에서 약간 다르다는 점을 제외하면 Linux 및 Unix 시스템에 대해 설명된 것과 동일한 단계 및 관련 지침이 적용됩니다.

1. 새 데이터 파일을 생성할 때는 innodb_data_file_path 옵션의 데이터 파일 크기 바로 뒤에 newraw라는 키워드를 지정합니다:

```
[mysqld]
innodb_data_home_dir=
innodb_data_파일_경로=//./D::10Gnewraw
```

   ./는 실제 드라이브에 액세스하기 위한 Windows 구문 \\.\에 해당합니다. 위의 예에서 D: 는 파티션의 드라이브 문자입니다.

2. 서버를 다시 시작합니다. InnoDB가 newraw 키워드를 인식하고 새 파티션을 초기화합니다.

3. InnoDB가 새 파티션을 초기화한 후 서버를 중지하고 데이터 파일 사양에서 newraw를 원시 파일로 변경합니다:

```
[mysqld]
innodb_data_home_dir=
innodb_data_파일_경로=//./D::10Graw
```

4. 서버를 다시 시작합니다. 이제 InnoDB에서 변경을 허용합니다.

15.6.3.2 테이블별 파일 테이블 공간

테이블별 파일 테이블 스페이스에는 단일 InnoDB 테이블에 대한 데이터와 인덱스가 포함되며, 단일 데이터

파일로 파일 시스템에 저장됩니다.

이 섹션의 다음 항목에서는 테이블별 파일 테이블 스페이스 특성에 대해 설명합니다:

- 테이블별 파일 테이블 스페이스 구성
- 테이블별 파일 테이블 스페이스 데이터 파일
- 테이블별 파일 테이블 스페이스의 장점
- 테이블별 파일 테이블 스페이스의 단점

테이블별 파일 테이블 스페이스 구성

InnoDB는 기본적으로 테이블별 파일 테이블 공간에 테이블을 생성합니다. 이 동작은 innodb_file_per_table 변수에 의해 제어됩니다. innodb_file_per_table을 비활성화하면 InnoDB가 시스템 테이블 공간에 테이블을 생성합니다.

innodb_file_per_table 설정은 옵션 파일에 지정하거나 런타임에 SET GLOBAL 문을 사용하여 구성할 수 있습니다. 런타임에 설정을 변경하려면 전역 시스템 변수를 설정할 수 있는 충분한 권한이 필요합니다. 섹션 5.1.9.1, "시스템 변수 권한"을 참조하십시오.

옵션 파일:

```
[mysqld]
innodb_file_per_table=ON
```

런타임에 SET GLOBAL 사용:

```
mysql> SET GLOBAL innodb_file_per_table=ON;
```

테이블별 파일 테이블 스페이스 데이터 파일

테이블별 파일 테이블 스페이스는 MySQL 데이터 디렉터리 아래의 스키마 디렉터리에 있는 .ibd 데이터 파일에 만들어집니다. .ibd 파일은 테이블의 이름을 따서 명명됩니다(*table_name*.ibd). 예를 들어 test.t1 테이블에 대한 데이터 파일은 MySQL 데이터 디렉터리 아래의 테스트 디렉터리에 생성됩니다:

```
mysql> USE 테스트;

mysql> CREATE TABLE t1 (
   id INT PRIMARY KEY AUTO_INCREMENT,
   name VARCHAR(100)
 ) 엔진 = InnoDB;

cd /path/to/mysql/data/test
$> ls
t1.ibd
```

CREATE TABLE 문의 DATA DIRECTORY 절을 사용하여 데이터 디렉터리 외부에 파일 단위 테이블 스페이스 데이터 파일을 암시적으로 생성할 수 있습니다. 자세한 내용은 섹션 15.6.1.2, "외부에서 테이블 생성"을 참조하십시오.

테이블별 파일 테이블 스페이스의 장점

테이블별 파일 테이블 스페이스는 시스템 테이블 스페이스나 일반 테이블 스페이스와 같은 공유 테이블 스페이스에 비해 다음과 같은 이점이 있습니다.

- 파일 단위 테이블 공간에서 생성된 테이블을 잘라내거나 삭제하면 디스크 공간이 운영 체제로 반환됩니다. 공유 테이블 공간에 저장된 테이블을 잘라내거나 삭제하면 공유 테이블 공간 데이터 파일 내에 여유 공간이 생성되며, 이 공간은 InnoDB 데이터에만 사용할 수 있습니다. 즉, 테이블이 잘리거나 삭제된 후에도 공유 테이블 스페이스 데이터 파일의 크기는 줄어들지 않습니다.

- 공유 테이블 공간에 있는 테이블에서 테이블을 복사하는 `ALTER TABLE` 작업을 수행하면 테이블 공간에서 차지하는 디스크 공간이 증가할 수 있습니다. 이러한 작업에는 테이블의 데이터와 인덱스만큼의 추가 공간이 필요할 수 있습니다. 이 공간은 테이블별 테이블 공간의 경우처럼 운영 체제로 다시 해제되지 않습니다.

- 테이블 단위 테이블 공간에 있는 테이블에서 실행하면 테이블 잘라내기 성능이 향상됩니다.

- 테이블별 테이블 공간 데이터 파일은 I/O 최적화, 공간 관리 또는 백업을 위해 별도의 저장 장치에 생성할 수 있습니다. 섹션 15.6.1.2, "외부에서 테이블 생성"을 참조하십시오.

- 다른 MySQL 인스턴스에서 파일 단위 테이블 공간에 있는 테이블을 가져올 수 있습니다. 15.6.1.3절. 'InnoDB 테이블 가져오기'를 참조하세요.

- 파일 단위 테이블 스페이스에서 생성된 테이블은 시스템 테이블 스페이스에서 지원하지 않는 동적 및 압축 행 형식과 관련된 기능을 지원합니다. 섹션 15.10, "InnoDB 행 형식"을 참조하십시오.
- 개별 테이블스페이스 데이터 파일에 저장된 테이블은 데이터 손상이 발생하거나 백업 또는 바이너리 로그를 사용할 수 없는 경우, 또는 MySQL 서버 인스턴스를 다시 시작할 수 없는 경우 시간을 절약하고 성공적인 복구 가능성을 높일 수 있습니다.
- 파일 단위 테이블 공간에서 생성된 테이블은 다른 `InnoDB` 테이블의 사용을 중단하지 않고도 MySQL 엔터프라이즈 백업을 사용하여 신속하게 백업하거나 복원할 수 있습니다. 이 기능은 백업 일정이 다양하거나 백업 빈도가 낮은 테이블에 유용합니다. 자세한 내용은 부분 백업 만들기를 참조하세요.
- 테이블별 파일 테이블스페이스는 테이블스페이스 데이터 파일의 크기를 모니터링하여 파일 시스템의 테이블 크기를 모니터링할 수 있습니다.
- 일반적인 Linux 파일 시스템은 `innodb_flush_method`가 `O_DIRECT`로 설정된 경우 공유 테이블 스페이스 데이터 파일과 같은 단일 파일에 대한 동시 쓰기를 허용하지 않습니다. 따라서 이 설정과 함께 테이블별 파일 테이블스페이스를 사용할 경우 성능이 향상될 수 있습니다.
- 공유 테이블 스페이스의 테이블은 64TB 테이블 스페이스 크기 제한에 의해 크기가 제한됩니다. 이에 비해 테이블별 테이블 공간에는 각 파일당 64TB 크기 제한이 적용되므로 개별 테이블의 크기를 늘릴 수 있는 충분한 공간이 제공됩니다.

테이블별 파일 테이블 스페이스의 단점

테이블별 파일 테이블 스페이스는 시스템 테이블 스페이스나 일반 테이블 스페이스와 같은 공유 테이블 스페이스에 비해 다음과 같은 단점이 있습니다.

- 테이블당 파일 테이블 스페이스를 사용하면 각 테이블에 같은 테이블의 행에서만 사용할 수 있는 미사용 공간이 있을 수 있으므로 제대로 관리하지 않으면 공간이 낭비될 수 있습니다.
- `fsync` 작업은 단일 공유 테이블 스페이스 데이터 파일이 아닌 여러 파일별 테이블 데이터 파일에서 수행됩니다. 파일 단위로 동기화 작업이 수행되므로 여러 테이블에 대한 쓰기 작업을 결합할 수 없으므로 총 동기화 작업 수가 더 많아질 수 있습니다.
- `mysqld`는 각 파일-테이블 테이블 공간에 대해 열린 파일 핸들을 유지해야 하므로 파일-테이블 테이블 공간에 많은 테이블이 있는 경우 성능에 영향을 줄 수 있습니다.
- 각 테이블에 자체 데이터 파일이 있는 경우 더 많은 파일 설명자가 필요합니다.
- 더 많은 조각화가 발생할 가능성이 있으며, 이로 인해 테이블 삭제 및 테이블 스캔 성능이 저하될 수 있습니다. 그러나 조각화를 관리하면 테이블별 파일 테이블스페이스를 통해 이러한 작업의 성능을 개선할 수 있습니다.

- 버퍼 풀은 테이블별 파일 테이블 공간에 있는 테이블을 삭제할 때 검사되며, 큰 버퍼 풀의 경우 몇 초가 걸릴 수 있습니다. 스캔은 광범위한 내부 잠금으로 수행되므로 다른 작업이 지연될 수 있습니다.
- 자동 확장되는 공유 테이블스페이스 파일이 가득 차면 크기를 확장하기 위한 증분 크기를 정의하는 `innodb_autoextend_increment` 변수는 `innodb_autoextend_increment` 설정에 관계없이 자동 확장되는 테이블별 파일 테이블스페이스 파일에는 적용되지 않습니다. 초기 파일 단위 테이블 테이블스페이스 확장은 소량으로 이루어지며, 그 이후에는 4MB 단위로 확장이 이 루 어 집 니 다 .

15.6.3.3 일반 테이블 스페이스

일반 테이블 스페이스는 `CREATE TABLESPACE` 구문을 사용하여 생성되는 공유 `InnoDB` 테이블 스페이스입니다. 이 섹션의 다음 항목에서 일반 테이블스페이스 기능 및 특징에 대해 설명합니다:

- 일반 테이블 스페이스 기능
- 일반 테이블 스페이스 만들기
- 일반 테이블 스페이스에 테이블 추가하기
- 일반 테이블 스페이스 행 형식 지원
- ALTER TABLE을 사용하여 테이블 공간 간에 테이블 이동하기
- 일반 테이블 스페이스 이름 바꾸기
- 일반 테이블 스페이스 삭제
- 일반적인 테이블 공간 제한

일반 테이블 스페이스 기능

일반 테이블 스페이스는 다음과 같은 기능을 제공합니다:

- 시스템 테이블 스페이스와 마찬가지로 일반 테이블 스페이스는 여러 테이블의 데이터를 저장할 수 있는 공유 테이블 스페이스입니다.
- 일반 테이블 스페이스는 파일별 테이블 스페이스에 비해 잠재적인 메모리 이점이 있습니다. 서버는 테이블스페이스 수명 기간 동안 테이블스페이스 메타데이터를 메모리에 유지합니다. 더 적은 수의 일반 테이블 스페이스에 있는 여러 테이블은 별도의 파일별 테이블 스페이스에 있는 동일한 수의 테이블보다 테이블 스페이스 메타데이터에 더 적은 메모리를 사용합니다.
- 일반 테이블 스페이스 데이터 파일은 MySQL 데이터 디렉터리에 상대적이거나 독립적인 디렉터리에 배치할 수 있으며, 이를 통해 테이블별 파일 테이블 스페이스의 많은 데이터 파일 및 스토리지 관리 기능을 사용할 수 있습니다. 테이블별 파일 테이블스페이스와 마찬가지로 데이터 파일을 MySQL 데이터 디렉터리 외부에 배치하는 기능을 사용하면 중요한 테이블의 성능을 별도로 관리하거나, 특정 테이블에 대해 RAID 또는 DRBD를 설정하거나, 특정 디스크에 테이블을 바인딩하는 등의 작업을 수행할 수 있습니다.
- 일반 테이블 스페이스는 모든 테이블 행 형식과 관련 기능을 지원합니다.
- 테이블 공간 옵션은 일반 테이블 공간, 파일별 테이블 공간 또는 시스템 테이블 공간에 테이블을 생성하기 위해 `CREATE TABLE`과 함께 사용할 수 있습니다.
- 테이블 공간 옵션은 `ALTER TABLE`과 함께 사용하여 일반 테이블 공간, 파일별 테이블 공간 및 시스템 테이블 공간 간에 테이블을 이동할 수 있습니다.

일반 테이블 스페이스 만들기

일반 테이블 스페이스는 `CREATE TABLESPACE` 구문을 사용하여 만듭니다.

```
테이블스페이스 테이블스페이스_이름 생성 [데
    이터파일 '파일_이름' 추가] [파일_블록
    _크기 = 값]
        [엔진 [=] 엔진 이름]]
```

일반 테이블스페이스는 데이터 디렉터리 내부 또는 외부에 만들 수 있습니다. 암시적으로 생성된 파일 단위 테이블 테이블 스페이스와의 충돌을 방지하기 위해 데이터 디렉터리 아래의 하위 디렉터리에 일반 테이블 스페이스를 생성하는 것은 지원되지 않습니다. 데이터 디렉터리 외부에 일반 테이블 스페이스를 만들 때는 해당 디렉터리가 존재해야 하며 테이블 스페이스를 만들기 전에 InnoDB에 알려야 합니다. 테이블스페이스를 만들려면
InnoDB에 알려진 알 수 없는 디렉터리가 있는 경우, 해당 디렉터리를 innodb_directories 인수 값에 추가합니다. innodb_directories는 읽기 전용 시작 옵션입니다. 이 옵션을 구성하려면 서버를 다시 시작해야 합니다. 예시:

데이터 디렉터리에 일반 테이블 스페이스 만들기:

```
mysql> CREATE TABLESPACE `ts1` ADD DATAFILE 'ts1.ibd' Engine=InnoDB;
```

또는

```
mysql> CREATE TABLESPACE `ts1` Engine=InnoDB;
```

ADD DATAFILE 절은 선택 사항입니다. 테이블스페이스를 생성할 때 ADD DATAFILE 절을 지정하지 않으면 고유한 파일 이름을 가진 테이블스페이스 데이터 파일이 암시적으로 생성됩니다. 고유 파일 이름은 대시로 구분된 5개의 16진수 그룹으로 형식이 지정된 128비트 UUID입니다(*aaaaaaaa- bbbb-cccc-dddd-eeeeeeeeeeee*). 일반 테이블스페이스 데이터 파일은 파일 확장자가 .ibd입니다. 복제 환경에서는 소스에서 생성된 데이터 파일 이름이 복제본에서 생성된 데이터 파일 이름과 동일하지 않습니다.

데이터 디렉터리 외부의 디렉터리에 일반 테이블 스페이스 만들기:

```
mysql> CREATE TABLESPACE `ts1` ADD DATAFILE '/my/tablespace/directory/ts1.ibd' Engine=InnoDB;
```

테이블스페이스 디렉터리가 데이터 디렉터리 아래에 있지 않은 경우 데이터 디렉터리에 상대적인 경로를 지정할 수 있습니다. 이 예제에서는 my_tablespace 디렉터리가 데이터 디렉터리와 같은 수준에 있습니다:

```
mysql> CREATE TABLESPACE `ts1` ADD DATAFILE '../my_tablespace/ts1.ibd' Engine=InnoDB;
```

참고

ENGINE = InnoDB 절은 CREATE TABLESPACE 문의 일부로 정의하거나 InnoDB를 기본 스토리지 엔진으로 정의해야 합니다 (default_storage_engine=InnoDB).

일반 테이블 스페이스에 테이블 추가하기

일반 테이블 스페이스를 생성한 후 CREATE TABLE *tbl_name* ... 다음 예제와 같이 테이블스페이스에 테이블을 추가하려면 TABLESPACE [=] *tablespace_name* 또는 ALTER TABLE *tbl_name* TABLESPACE [=] *tablespace_name* 문을 사용할 수 있습니다:

테이블 만들기:

```
mysql> CREATE TABLE t1 (c1 INT PRIMARY KEY) TABLESPACE ts1;
```

테이블 변경:

```
mysql> ALTER TABLE t2 TABLESPACE ts1;
```

공유 테이블 공간에 테이블 파티션을 추가하는 것은 지원되지 않습니다. 공유 테이블 공간에는 InnoDB 시스템 테이블 스페이스 및 일반 테이블 스페이스.

자세한 구문 정보는 테이블 생성 및 테이블 변경을 참조하십시오.

일반 테이블 스페이스 행 형식 지원

일반 테이블 스페이스는 모든 테이블 행 형식(중복, 압축, 동적, 압축)을 지원하지만 물리적 페이지 크기가 다르기 때문에 압축된 테이블과 압축되지 않은 테이블이 동일한 일반 테이블 스페이스에 공존할 수 없다는 점에 주의해야 합니다.

일반 테이블스페이스에 압축 테이블(ROW_FORMAT=COMPRESSED)을 포함하려면 FILE_BLOCK_SIZE 옵션을 지정해야 하며, FILE_BLOCK_SIZE 값은 innodb_page_size 값과 관련하여 유효한 압축 페이지 크기여야 합니다. 또한 압축된 테이블의 실제 페이지 크기(KEY_BLOCK_SIZE)는 FILE_BLOCK_SIZE/1024와 같아야 합니다. 예를 들어, innodb_page_size=16KB이고 FILE_BLOCK_SIZE=8K인 경우 테이블의 KEY_BLOCK_SIZE는 8이어야 합니다.

다음 표는 허용되는 innodb_page_size, FILE_BLOCK_SIZE 및 KEY_BLOCK_SIZE 조합을 보여줍니다. FILE_BLOCK_SIZE 값은 바이트 단위로도 지정할 수 있습니다. 주어진 FILE_BLOCK_SIZE에 대해 유효한 KEY_BLOCK_SIZE 값을 확인하려면, 주어진 FILE_BLOCK_SIZE를 FILE_BLOCK_SIZE 값을 1024로 늘립니다. 테이블 압축은 32K 및 64K InnoDB 페이지에는 지원되지 않습니다.

크기. KEY_BLOCK_SIZE에 대한 자세한 내용은 테이블 생성 및 섹션 15.9.1.2, "압축 테이블 생성"을 참조하십시오.

표 15.3 압축 테이블에 허용되는 페이지 크기, FILE_BLOCK_SIZE 및 KEY_BLOCK_SIZE 조합

InnoDB 페이지 크기 (innodb_page_size)	허용된 FILE_BLOCK_SIZE 값	허용된 KEY_BLOCK_SIZE 값
64KB	64K (65536)	압축은 지원되지 않습니다.
32KB	32K (32768)	압축은 지원되지 않습니다.
16KB	16K (16384)	없음. innodb_page_size가 파일 블록 크기와 같으면 테이블 스페이스에 압축된 테이블을 포함할 수 없습니다.
16KB	8K (8192)	8
16KB	4K (4096)	4
16KB	2K (2048)	2
16KB	1K (1024)	1
8KB	8K (8192)	없음. innodb_page_size가 파일 블록 크기와 같으면 테이블 스페이스에 압축된 테이블을 포함할 수 없습니다.
8KB	4K (4096)	4
8KB	2K (2048)	2
8KB	1K (1024)	1
4KB	4K (4096)	없음. innodb_page_size가 파일 블록 크기와 같으면 테이블 스페이스에 압축된 테이블을 포함할 수 없습니다.
4KB	2K (2048)	2
4KB	1K (1024)	1

이 예제에서는 일반 테이블 스페이스를 생성하고 압축 테이블을 추가하는 방법을 보여줍니다. 이 예에서는 기본 innodb_page_size가 16KB라고 가정합니다. FILE_BLOCK_SIZE가 8192이면 압축 테이블의 KEY_BLOCK_SIZE가 8이어야 합니다.

```
mysql> CREATE TABLESPACE `ts2` ADD DATAFILE 'ts2.ibd' FILE_BLOCK_SIZE = 8192 엔진=InnoDB;

mysql> CREATE TABLE t4 (c1 INT PRIMARY KEY) TABLESPACE ts2 ROW_FORMAT=COMPRESSED KEY_BLOCK_SIZE=8;
```

일반 테이블스페이스를 생성할 때 FILE_BLOCK_SIZE를 지정하지 않으면 FILE_BLOCK_SIZE의 기본값은 innodb_page_size이다. FILE_BLOCK_SIZE가 innodb_page_size와 같으면 테이블스페이스에는 압축되지 않은 행 형식(COMPACT, 중복 및 동적 행 형식)의 테이블만 포함될 수 있습니다.

ALTER TABLE을 사용하여 테이블 공간 간에 테이블 이동하기

테이블 공간 옵션을 사용하여 테이블을 기존 일반 테이블 공간, 새 파일별 테이블 공간 또는 시스템 테이블 공간으로 이동하는 데 `ALTER` TABLE을 사용할 수 있습니다.

공유 테이블 공간에 테이블 파티션을 추가하는 것은 지원되지 않습니다. 공유 테이블 공간에는 `InnoDB` 시스템 테이블 스페이스 및 일반 테이블 스페이스.

테이블별 테이블 공간 또는 시스템 테이블 공간에서 일반 테이블 공간으로 테이블을 이동하려면 일반 테이블 공간의 이름을 지정합니다. 일반 테이블스페이스가 존재해야 합니다. 자세한 내용은 테이블 공간 변경을 참조하십시오.

```
ALTER TABLE tbl_name 테이블스페이스 [=] 테이블스페이스_이름;
```

일반 테이블 스페이스 또는 파일 단위 테이블 스페이스에서 시스템 테이블 스페이스로 테이블을 이동하려면 테이블 스페이스 이름으로 innodb_system을 지정합니다.

```
ALTER TABLE tbl_name TABLESPACE [=] innodb_system;
```

테이블을 시스템 테이블 스페이스 또는 일반 테이블 스페이스에서 테이블별 파일 테이블 스페이스로 이동하려면 테이블 스페이스 이름으로 innodb_file_per_table을 지정합니다.

```
ALTER TABLE tbl_name TABLESPACE [=] innodb_file_per_table;
```

변경 테이블 ... 테이블 스페이스 작업이 전체 테이블 재구성을 유발합니다.
속성은 이전 값에서 변경되지 않았습니다.

테이블 변경 ... 테이블 스페이스 구문은 테이블을 임시 테이블 스페이스에서 영구 테이블 스페이스로 이동하는 것을 지원하지 않습니다.

DATA DIRECTORY 절은 CREATE TABLE ... TABLESPACE=innodb_file_per_table과 함께 사용할 수 있지만, 그렇지 않은 경우 TABLESPACE 옵션과 함께 사용하는 것은 지원되지 않습니다. DATA DIRECTORY 절에 지정된 디렉터리는 InnoDB에 알려져 있어야 합니다. 자세한 내용은 DATA DIRECTORY 절 사용을 참조하십시오.

암호화된 테이블 공간에서 테이블을 이동할 때는 제한 사항이 적용됩니다. 암호화 제한을 참조하십시오.

일반 테이블 스페이스 이름 바꾸기

일반 테이블 스페이스의 이름 변경은 ALTER TABLESPACE ... RENAME TO 구문을 사용합니다.

```
테이블 공간 s1 이름을 s2로 변경합니다;
```

일반 테이블 스페이스의 이름을 변경하려면 CREATE TABLESPACE 권한이 필요합니다.

자동 커밋과 관계없이 자동 커밋 모드에서 암시적으로 RENAME TO 작업이 수행됩니다.
설정합니다.

테이블 공간에 있는 테이블에 대해 테이블 잠금 또는 읽기 잠금으로 테이블 플러시가 적용되어 있는 동안에는 RENAME TO 작업을 수행할 수 없습니다.

일반 테이블 스페이스 내의 테이블에 대해 독점 메타데이터 잠금이 적용되며, 테이블 스페이스의 이름이 변경되는 동안에는 동시 DDL이 방지됩니다. 동시 DML이 지원됩니다.

일반 테이블 스페이스 삭제

DROP TABLESPACE 문은 InnoDB 일반 테이블 스페이스를 삭제하는 데 사용됩니다.

DROP TABLESPACE 작업을 수행하기 전에 모든 테이블을 테이블스페이스에서 삭제해야 합니다. 테이블 스페이스가 비어 있지 않으면 DROP TABLESPACE가 오류를 반환합니다.

다음과 유사한 쿼리를 사용하여 일반 테이블 스페이스에서 테이블을 식별합니다.

```
mysql> SELECT a.NAME AS space_name, b.NAME AS table_name FROM INFORMATION_SCHEMA.INNODB_TABLESPACES a,
       INFORMATION_SCHEMA.INNODB_TABLES b WHERE a.SPACE=b.SPACE AND a.NAME LIKE 'ts1';
+------------+-------------+
| 공간_이름 | 테이블_이름 |
+------------+-------------+
| TS1| 테스트/T1            |
| TS1| 테스트/T2            |
| TS1| 테스트/T3            |
+------------+-------------+
```

일반 `InnoDB` 테이블스페이스는 테이블스페이스의 마지막 테이블이 삭제될 때 자동으로 삭제되지 않습니다. 테이블스페이스는 `DROP TABLESPACE` *`tablespace_name`*을 사용하여 명시적으로 삭제해야 합니다.

일반 테이블스페이스는 특정 데이터베이스에 속하지 않습니다. DROP DATABASE 작업은 일반 테이블 스페이스에 속한 테이블을 삭제할 수 있지만 테이블 스페이스에 속한 모든 테이블을 삭제하는 경우에도 테이블 스페이스를 삭제할 수는 없습니다.

시스템 테이블 스페이스와 마찬가지로 일반 테이블 스페이스에 저장된 테이블을 잘라내거나 삭제하면 내부적으로 일반 테이블 스페이스 .ibd 데이터 파일에 새 InnoDB 데이터에만 사용할 수 있는 여유 공간이 생깁니다. 테이블별 파일 테이블스페이스가 DROP TABLE 작업 중에 삭제되는 경우 해당 공간은 운영 체제로 다시 해제되지 않습니다.

이 예는 InnoDB 일반 테이블 스페이스를 삭제하는 방법을 보여줍니다. 일반 테이블 스페이스 ts1 는 단일 테이블로 생성됩니다. 테이블을 삭제하기 전에 테이블 스페이스를 삭제해야 합니다.

```
mysql> CREATE TABLESPACE `ts1` ADD DATAFILE 'ts1.ibd' Engine=InnoDB;

mysql> CREATE TABLE t1 (c1 INT PRIMARY KEY) TABLESPACE ts1 Engine=InnoDB;

mysql> DROP TABLE t1;

mysql> DROP TABLESPACE ts1;
```

참고

테이블스페이스_이름은 MySQL에서 대소문자를 구분하는 식별자입니다.

일반적인 테이블 공간 제한

- 생성된 테이블스페이스 또는 기존 테이블스페이스는 일반 테이블스페이스로 변경할 수 없습니다.
- 임시 일반 테이블 스페이스 생성은 지원되지 않습니다.
- 일반 테이블 스페이스는 임시 테이블을 지원하지 않습니다.
- 시스템 테이블 스페이스와 마찬가지로 일반 테이블 스페이스에 저장된 테이블을 잘라내거나 삭제하면 내부적으로 일반 테이블 스페이스 .ibd 데이터 파일에 새 InnoDB 데이터에만 사용할 수 있는 여유 공간이 생깁니다. 이 공간은 테이블별 테이블 스페이스의 경우처럼 운영 체제로 다시 릴리스되지 않습니다.

  또한 공유 테이블 스페이스(일반 테이블 스페이스 또는 시스템 테이블 스페이스)에 있는 테이블에서 테이블을 복사하는 ALTER TABLE 작업을 수행하면 테이블 스페이스에서 사용하는 공간이 늘어날 수 있습니다. 이러한 작업에는 테이블의 데이터에 인덱스를 더한 만큼의 추가 공간이 필요합니다. 테이블을 복사하는 ALTER TABLE 작업에 필요한 추가 공간은 테이블별 테이블 공간의 경우처럼 운영 체제로 다시 해제되지 않습니다.
- 테이블 변경 ... 테이블 공간 삭제 및 테이블 가져 오기 ...는 다음과 같습니다.
  일반 테이블 스페이스에 속한 테이블에는 지원되지 않습니다.
- 일반 테이블 공간에 테이블 파티션을 배치하는 것은 지원되지 않습니다.
- 원본과 복제본이 동일한 호스트에 상주하는 복제 환경에서는 원본과 복제본이 동일한 위치에 동일한 이름의

테이블스페이스를 생성하게 되므로 `ADD DATAFILE` 절이 지원되지 않습니다. 그러나 `ADD DATAFILE` 절을 생략하면 데이터 디렉터리에 고유한 파일 이름으로 생성된 테이블스페이스가 생성되므로 허용됩니다.

- 일반 테이블스페이스는 `InnoDB`에 직접 알려진 디렉터리가 아니면 실행 취소 테이블스페이스 디렉터리(`innodb_undo_directory`)에 생성할 수 없습니다. 알려진 디렉터리는 `datadir`, `innodb_data_home_dir` 및 `innodb_directories` 변수에 의해 정의된 디렉터리입니다.

15.6.3.4 테이블 스페이스 실행 취소

실행 취소 테이블스페이스에는 클러스터된 인덱스 레코드에 대한 트랜잭션의 최신 변경 사항을 실행 취소하는 방법에 대한 정보가 포함된 레코드 모음인 실행 취소 로그가 포함되어 있습니다.

테이블 스페이스 실행 취소는 이 섹션의 다음 항목에 설명되어 있습니다:

- 기본 테이블 스페이스 실행 취소
- 테이블 공간 크기 실행 취소
- 테이블 스페이스 실행 취소 추가
- 테이블 공간 실행 취소 삭제
- 테이블 공간 실행 취소 이동
- 롤백 세그먼트 수 구성하기
- 테이블 공간 실행 취소 잘라내기
- 테이블 스페이스 상태 변수 실행 취소

기본 테이블 스페이스 실행 취소

MySQL 인스턴스가 초기화될 때 두 개의 기본 실행 취소 테이블스페이스가 생성됩니다. 기본 실행 취소 테이블스페이스는 초기화 시 생성되어 SQL 문을 수락하기 전에 존재해야 하는 롤백 세그먼트의 위치를 제공합니다. 실행 취소 테이블스페이스의 자동 잘림을 지원하려면 최소 두 개의 실행 취소 테이블스페이스가 필요합니다. 실행 취소 테이블 공간 잘라내기를 참조하십시오.

기본 실행 취소 테이블스페이스는 `innodb_undo_directory` 변수에 의해 정의된 위치에 생성됩니다. `innodb_undo_directory` 변수가 정의되지 않은 경우 기본 실행 취소 테이블스페이스는 데이터 디렉터리에 생성됩니다. 기본 실행 취소 테이블스페이스 데이터 파일의 이름은 `undo_001` 및 `undo_002`입니다. 데이터 사전에 정의된 해당 실행 취소 테이블스페이스 이름은 `innodb_undo_001` 및 `innodb_undo_002`입니다.

SQL 문을 사용하여 런타임에 추가 실행 취소 테이블스페이스를 만들 수 있습니다. 실행 취소 테이블 공간 추가를 참조하십시오.

테이블 공간 크기 실행 취소

초기 실행 취소 테이블스페이스 크기는 일반적으로 16MiB입니다. 잘라내기 작업으로 새 실행 취소 테이블스페이스가 생성되면 초기 크기가 달라질 수 있습니다. 이 경우 파일 확장자 크기가 16MB보다 크고 이전 파일 확장자가 최근 1초 이내에 발생한 경우, 새 실행 취소 테이블스페이스는 `innodb_max_undo_log_size` 변수에 정의된 크기의 1/4 크기로 만들어집니다.

실행 취소 테이블 스페이스는 최소 16MB 확장됩니다. 급격한 증가를 처리하기 위해 이전 파일 확장자 크기가 0.1초 미만인 경우 파일 확장자 크기가 두 배가 됩니다. 확장자 크기가 두 배로 늘어나는 것은 최대 256MB까지 여러 번 발생할 수 있습니다. 이전 파일 확장자가 0.1초 이상 이전에 발생한 경우 확장자 크기가 절반으로

줄어들며, 이 역시 최소 16MB까지 여러 번 발생할 수 있습니다. 실행 취소 테이블 스페이스에 대해 AUTOEXTEND_SIZE 옵션이 정의되어 있는 경우 AUTOEXTEND_SIZE 설정과 위에서 설명한 논리에 의해 결정된 확장 크기 중 더 큰 값만큼 확장됩니다. AUTOEXTEND_SIZE 옵션에 대한 자세한 내용은 섹션 15.6.3.9, "테이블스페이스 AUTOEXTEND_SIZE 구성"을 참조하십시오.

테이블 스페이스 실행 취소 추가

실행 취소 로그는 장기간 실행되는 트랜잭션 중에 커질 수 있으므로 추가 실행 취소 테이블스페이스를 만들면 개별 실행 취소 테이블스페이스가 너무 커지는 것을 방지하는 데 도움이 됩니다. 추가 실행 취소 테이블스페이스는 런타임에 CREATE UNDO TABLESPACE 구문을 사용하여 만들 수 있습니다.

```
CREATE UNDO 테이블스페이스 tablespace_name 데이터 파일 추가 'file_name.ibu';
```

실행 취소 테이블스페이스 파일 이름은 확장자가 .ibu여야 합니다. 실행 취소 테이블스페이스 파일 이름을 정의할 때 상대 경로를 지정할 수 없습니다. 정규화된 경로도 허용되지만 이 경로는 InnoDB에 알려진 경로여야 합니다. 알려진 경로는 innodb_directories 변수에 정의된 경로입니다. 고유

테이블 스페이스 파일 이름 실행 취소는 데이터를 이동하거나 복제할 때 발생할 수 있는 파일 이름 충돌을 방지하는 데 권장됩니다.

참고

복제 환경에서는 소스 및 각 복제본에 고유한 실행 취소 테이블스페이스 파일 디렉터리가 있어야 합니다. 실행 취소 테이블스페이스 파일 생성을 공통 디렉터리에 복제하면 파일 이름 충돌이 발생할 수 있습니다.

시작 시 `innodb_directories` 변수로 정의된 디렉터리에서 실행 취소된 테이블스페이스 파일이 있는지 검색합니다. (이 검색은 하위 디렉터리도 탐색합니다.) `innodb_data_home_dir`, `innodb_undo_directory` 및 `datadir` 변수에 의해 정의된 디렉터리는 `innodb_directories` 변수가 명시적으로 정의되었는지 여부와 관계없이 `innodb_directories` 값에 자동으로 추가됩니다. 따라서 실행 취소 테이블스페이스는 이러한 변수에 의해 정의된 경로에 위치할 수 있습니다.

실행 취소 테이블스페이스 파일 이름에 경로가 포함되지 않은 경우 실행 취소 테이블스페이스는 `innodb_undo_directory` 변수에 의해 정의된 디렉터리에 생성됩니다. 해당 변수가 정의되지 않은 경우 실행 취소 테이블스페이스는 데이터 디렉터리에 생성됩니다.

참고

`InnoDB` 복구 프로세스에서는 실행 취소 테이블스페이스 파일이 알려진 디렉터리에 있어야 합니다. 실행 취소 테이블스페이스 파일은 복구를 다시 실행하기 전과 다른 데이터 파일을 열기 전에 찾아서 열어야 커밋되지 않은 트랜잭션과 데이터 사전 변경 사항을 롤백할 수 있습니다. 복구 전에 실행 취소 테이블스페이스를 찾지 못하면 사용할 수 없으므로 데이터베이스 불일치가 발생할 수 있습니다. 데이터 사전에 알려진 실행 취소 테이블스페이스를 찾을 수 없는 경우 시작 시 오류 메시지가 보고됩니다. 알려진 디렉터리 요구 사항은 실행 취소 테이블스페이스 이식성도 지원합니다. 실행 취소 테이블스페이스 이동을 참조하세요.

데이터 디렉터리를 기준으로 한 경로에 실행 취소 테이블스페이스를 만들려면 `innodb_undo_directory`를 설정합니다. 변수를 상대 경로로 설정하고 실행 취소 테이블스페이스를 만들 때만 파일 이름을 지정합니다. 실행

```
정보_schema.files에서 테이블스페이스_이름, 파일_이름을 선택하고 여기서 파일
  유형은 '로그 취소'와 같습니다;
```

MySQL 인스턴스는 MySQL 인스턴스가 초기화될 때 생성되는 2개의 기본 실행 취소 테이블스페이스를 포함하여 최대 127개의 실행 취소 테이블스페이스를 지원합니다.

테이블 스페이스 실행 취소는 `DROP UNDO TABALESPACE` 구문을 사용하여 삭제할 수 있습니다. 실행 취소 테이블 공간 삭제하기를 참조하십시오.

테이블 공간 실행 취소 삭제

CREATE UNDO TABLESPACE 구문을 사용하여 만든 테이블 스페이스 실행 취소는 런타임에 다음을 사용하여 삭제할 수 있습니다.
테이블 스페이스 삭제 실행 취소 구문.

실행 취소 테이블스페이스를 비워야 삭제할 수 있습니다. 실행 취소 테이블스페이스를 비우려면 먼저 실행 취소 테이블스페이스가 더 이상 새 트랜잭션에 롤백 세그먼트를 할당하는 데 사용되지 않도록 ALTER UNDO TABLESPACE 구문을 사용하여 비활성 상태로 표시해야 합니다.

```
ALTER UNDO TABLESPACE 테이블스페이스_이름을 비활성 상태로 설정합니다;
```

실행 취소 테이블스페이스가 비활성 상태로 표시되면 현재 실행 취소 테이블스페이스의 롤백 세그먼트를 사용 중인 트랜잭션과 해당 트랜잭션이 완료되기 전에 시작된 모든 트랜잭션이 완료되도록 허용됩니다. 트랜잭션이 완료되면 제거 시스템이 실행 취소 테이블스페이스의 롤백 세그먼트를 해제하고 실행 취소 테이블스페이스가 초기 크기로 잘립니다. (실행 취소 테이블스페이스를 잘라낼 때도 동일한 프로세스가 사용됩니다. 실행 취소 테이블 공간 잘라내기 참조). 실행 취소 테이블스페이스가 비어 있으면 삭제할 수 있습니다.

```
DROP UNDO 테이블스페이스 테이블스페이스_이름;
```

참고

또는 실행 취소 테이블스페이스를 비어 있는 상태로 두었다가 나중에 필요한 경우 `ALTER UNDO TABLESPACE tablespace_name SET ACTIVE` 문을 실행하여 다시 활성화할 수도 있습니다.

실행 취소 테이블스페이스의 상태는 정보 스키마를 쿼리하여 모니터링할 수 있습니다. `INNODB_TABLESPACES` 테이블.

```
information_schema.innodb_tablespaces에서 이름, 상태 선택
  WHERE 이름이 '테이블스페이스_이름'과 같은 경우;
```

비활성 상태는 실행 취소 테이블스페이스의 롤백 세그먼트가 새 트랜잭션에서 더 이상 사용되지 않음을 나타냅니다. 비어 있음 상태는 실행 취소 테이블스페이스가 비어 있고 삭제할 준비가 되었음을 나타냅니다, 테이블스페이스가 비어 있거나 `ALTER UNDO TABLESPACE 테이블스페이스_이름 SET ACTIVE` 문을 사용하여 다시 활성화할 준비가 된 상태여야 합니다. 비어 있지 않은 테이블스페이스를 실행 취소하려고 하면 오류가 반환됩니다.

MySQL 인스턴스가 초기화될 때 생성된 기본 실행 취소 테이블스페이스(`innodb_undo_001` 및 `innodb_undo_002`)는 삭제할 수 없습니다. 그러나 `ALTER UNDO TABLESPACE tablespace_name SET INACTIVE` 문을 사용하여 비활성화할 수 있습니다. 기본 실행 취소 테이블스페이스를 비활성 상태로 만들려면 먼저 해당 테이블스페이스를 대신할 실행 취소 테이블스페이스가 있어야 합니다. 최소의 활성 실행 취소 테이블 스페이스가 항상 있어야 실행 취소 테이블 스페이스의 자동 잘림을 지원할 수 있습니다.

테이블 공간 실행 취소 이동

서버가 오프라인 상태일 때 `CREATE UNDO TABLESPACE` 구문을 사용하여 만든 실행 취소 테이블스페이스를 알려진 디렉터리로 이동할 수 있습니다. 알려진 디렉터리는 `innodb_directories` 변수에 의해 정의된 디렉터리입니다. `innodb_data_home_dir`, `innodb_undo_directory` 및 `datadir`에 의해 정의된 디렉터리는 `innodb_directories` 변수가 명시적으로 정의되었는지 여부에 관계없이 `innodb_directories` 값에 자동으로 추가됩니다. 이러한 디렉터리와 그 하위 디렉터리는 시작 시 실행 취소 테이블스페이스 파일을 검색합니다. 이러한 디렉터리로 이동된 실행 취소 테이블스페이스 파일은 시작 시 발견되며 이동된 실행 취소 테이블스페이스로 간주됩니다.

MySQL 인스턴스가 초기화될 때 생성되는 기본 실행 취소 테이블스페이스(`innodb_undo_001` 및 `innodb_undo_002`)는 `innodb_undo_directory` 변수에 의해 정의된 디렉터리에 상주해야 합니다. `innodb_undo_directory` 변수가 정의되지 않은 경우 기본 실행 취소 테이블스페이스가 데이터 디렉터리에 있습니다. 서버가 오프라인 상태일 때 기본 실행 취소 테이블스페이스를 이동하는 경우, 서버를 새 디렉터리로 구성된 `innodb_undo_directory` 변수로 시작해야 합니다.

실행 취소 로그의 I/O 패턴으로 인해 실행 취소 테이블스페이스는 SSD 스토리지에 적합한 후보입니다.

롤백 세그먼트 수 구성하기

`innodb_rollback_segments` 변수는 각 실행 취소 테이블스페이스와 글로벌 임시 테이블스페이스에 할당된 롤백 세그먼트 수를 정의합니다. `innodb_rollback_segments` 변수는 서버를 시작할 때 또는 서버가 실행되는 동안 구성할 수 있습니다.

`innodb_rollback_segments`의 기본 설정은 128이며, 이는 최대값이기도 합니다. 롤백 세그먼트가 지원하는 트랜잭션 수에 대한 자세한 내용은 섹션 15.6.6, "로그 실행 취소"를 참조하십시오.

테이블 공간 실행 취소 잘라내기

실행 취소 테이블스페이스를 잘라내는 방법에는 두 가지가 있으며, 개별적으로 또는 조합하여 실행 취소 테이블스페이스 크기를 관리할 수 있습니다. 한 가지 방법은 자동화된 방법으로, 구성 변수를 사용하여 활성화할 수 있습니다. 다른 방법은 수동으로, SQL 문을 사용하여 수행됩니다.

자동화된 방법은 실행 취소 테이블 공간 크기를 모니터링할 필요가 없으며, 일단 활성화되면 수동 개입 없이 실행 취소 테이블 공간의 비활성화, 잘림 및 재활성화를 수행합니다.

실행 취소된 테이블 스페이스가 잘리기 위해 오프라인으로 전환되는 시기를 제어하려는 경우 수동 잘림 방법을 사용하는 것이 더 좋을 수 있습니다. 예를 들어 워크로드가 가장 많은 시간대에는 실행 취소 테이블스페이스가 잘리지 않도록 하고 싶을 수 있습니다.

자동 잘라내기

실행 취소 테이블스페이스를 자동으로 잘라내려면 최소 두 개의 활성 실행 취소 테이블스페이스가 필요하므로 한 실행 취소 테이블스페이스는 활성 상태로 유지되고 다른 테이블스페이스는 오프라인으로 전환되어 잘려야 합니다. 기본적으로 MySQL 인스턴스가 초기화될 때 실행 취소 테이블스페이스가 두 개 생성됩니다.

테이블 스페이스가 자동으로 잘리도록 실행 취소를 수행하려면 `innodb_undo_log_truncate`를 활성화합니다. 변수로 설정합니다. 예를 들어

```
mysql> SET GLOBAL innodb_undo_log_truncate=ON;
```

`innodb_undo_log_truncate` 변수가 활성화되면 `innodb_max_undo_log_size` 변수에 정의된 크기 제한을 초과하는 실행 취소 테이블스페이스는 잘릴 수 있습니다. `innodb_max_undo_log_size` 변수는 동적이며 기본값은 1073741824 바이트(1024 MiB)입니다.

```
mysql> SELECT @@innodb_max_undo_log_size;
+-----------------------------+
| @@innodb_max_undo_log_size | @@innodb_max_undo_log_size
+-----------------------------+
                  |1073741824 |
+-----------------------------+
```

`innodb_undo_log_truncate` 변수가 활성화된 경우:

1. `innodb_max_undo_log_size` 설정을 초과하는 기본 및 사용자 정의 실행 취소 테이블스페이스는 잘릴 수 있도록 표시됩니다. 잘릴 실행 취소 테이블스페이스의 선택은 매번 동일한 실행 취소 테이블스페이스가 잘리지 않도록 순환 방식으로 수행됩니다.

2. 선택한 실행 취소 테이블 스페이스에 있는 롤백 세그먼트는 새 트랜잭션에 할당되지 않도록 비활성 상태가 됩니다. 현재 롤백 세그먼트를 사용 중인 기존 트랜잭션은 완료할 수 있습니다.

3. 퍼지 시스템은 더 이상 사용하지 않는 실행 취소 로그를 해제하여 롤백 세그먼트를 비웁니다.

4. 실행 취소 테이블 스페이스의 모든 롤백 세그먼트가 해제되면 잘라내기 작업이 실행되어 실행 취소 테이블 스페이스가 초기 크기로 잘립니다.

   잘라내기 작업 후 실행 취소 테이블 스페이스의 크기는 작업 완료 후 즉시 사용되기 때문에 초기 크기보다 클 수 있습니다.

   `innodb_undo_directory` 변수는 기본 실행 취소 테이블스페이스 파일의 위치를 정의합니다. `innodb_undo_directory` 변수가 정의되지 않은 경우 기본 실행 취소 테이블스페이스는 데이터 디렉터리에 있습니다. `CREATE UNDO TABLESPACE` 구문을 사용하여 생성된 사용자 정의 실행 취소 테이블스페이스를 포함한 모든 실행 취소 테이블스페이스 파일의 위치는 정보 스키마 `FILES` 테이블을 쿼리하여 확인할 수 있습니다:

```
정보_schema.files에서 테이블스페이스_이름, 파일_이름을 선택하고 여기서 파일 유형은 '로그 실행 취소'입니다;
```

5. 롤백 세그먼트가 다시 활성화되어 새 트랜잭션에 할당할 수 있습니다.

수동 잘라내기

실행 취소 테이블스페이스를 수동으로 잘라내려면 최소 3개의 활성 실행 취소 테이블스페이스가 필요합니다. 자동 잘림이 활성화될 가능성을 지원하려면 항상 2개의 활성 실행 취소 테이블스페이스가 필요합니다. 실행 취소 테이블스페이스가 3개 이상이면 이 요구 사항을 충족하는 동시에 실행 취소 테이블스페이스를 수동으로 오프라인으로 전환할 수 있습니다.

실행 취소 테이블 스페이스의 잘림을 수동으로 시작하려면 다음 문을 실행하여 실행 취소 테이블 스페이스를 비활성화합니다:

```
ALTER UNDO TABLESPACE 테이블스페이스_이름을 비활성 상태로 설정합니다;
```

실행 취소 테이블스페이스가 비활성 상태로 표시되면 현재 실행 취소 테이블스페이스의 롤백 세그먼트를 사용 중인 트랜잭션은 해당 트랜잭션이 완료되기 전에 시작된 모든 트랜잭션과 마찬가지로 완료될 수 있습니다. 트랜잭션이 완료되면 제거 시스템이 실행 취소 테이블스페이스의 롤백 세그먼트를 해제하고 실행 취소 테이블스페이스가 초기 크기로 잘리고 실행 취소 테이블스페이스 상태가 비활성에서 비어 있음으로 변경됩니다.

참고

`ALTER UNDO TABLESPACE tablespace_name SET INACTIVE` 문으로 실행 취소 테이블스페이스가 비활성화되면 제거 스레드는 다음 기회에 해당 실행 취소 테이블스페이스를 찾습니다. 실행 취소 테이블스페이스를 찾아서 잘라내도록 표시하면 제거 스레드는 빈도가 증가하여 실행 취소 테이블스페이스를 빠르게 비우고 잘라냅니다.

실행 취소된 테이블스페이스의 상태를 확인하려면 정보 스키마 `INNODB_TABLESPACES`를 쿼리합니다.
테이블로 이동합니다.

```
information_schema.innodb_tablespaces에서 이름, 상태 선택
  WHERE 이름이 '테이블스페이스_이름'과 같은 경우;
```

실행 취소 테이블 스페이스가 빈 상태가 되면 다음 문을 실행하여 다시 활성화할 수 있습니다:

```
ALTER UNDO TABLESPACE 테이블스페이스_이름 설정 활성;
```

비어 있는 상태의 실행 취소 테이블스페이스를 삭제할 수도 있습니다. 실행 취소 테이블 공간 삭제하기를 참조하십시오.

실행 취소 테이블 공간의 자동 잘라내기 가속화

제거 스레드는 실행 취소 테이블스페이스를 비우고 잘라내는 작업을 담당합니다. 기본적으로 제거 스레드는 제거가 호출되는 128회마다 잘라낼 실행 취소 테이블스페이스를 찾습니다. 제거 스레드가 잘라낼 실행 취소 테이블스페이스를 찾는 빈도는 기본 설정이 128인 `innodb_purge_rseg_truncate_frequency` 변수에 의해 제어됩니다.

```
mysql> SELECT @@innodb_purge_rseg_truncate_frequency;
+-----------------------------------------+
@@innodb_purge_rseg_truncate_frequency | @@innodb_purge_rseg_truncate_frequency |
+-----------------------------------------+
                                        |128 |
+-----------------------------------------+
```

빈도를 늘리려면 `innodb_purge_rseg_truncate_frequency` 설정을 줄이십시오. 예를 들어, 제거 스레드가 제거가 호출되는 32회마다 한 번씩 실행 취소된 테이블스페이스를 찾도록 하려면 `innodb_purge_rseg_truncate_frequency`를 32로 설정합니다.

```
mysql> SET GLOBAL innodb_purge_rseg_truncate_frequency=32;
```

테이블 스페이스 실행 취소 파일 자르기의 성능 영향

실행 취소 테이블스페이스가 잘리면 실행 취소 테이블스페이스의 롤백 세그먼트가 비활성화됩니다. 다른 실행 취소 테이블스페이스의 활성 롤백 세그먼트가 전체 시스템 부하를 담당하므로 약간의 성능 저하가 발생할 수

있습니다. 성능에 영향을 미치는 정도는 여러 요인에 따라 달라집니다:

- 실행 취소 테이블 스페이스 수
- 실행 취소 로그 수
- 테이블 공간 크기 실행 취소
- I/O 서브시스템의 속도
- 기존 장기 실행 트랜잭션

- 시스템 부하

잠재적인 성능 영향을 피하는 가장 쉬운 방법은 실행 취소 테이블 공간의 수를 늘리는 것입니다.

테이블 공간 잘라내기 실행 취소 모니터링

실행 취소 및 제거 하위 시스템 카운터는 실행 취소 로그 잘림과 관련된 백그라운드 활동을 모니터링하기 위해 제공됩니다. 카운터 이름과 설명은 정보 스키마 INNODB_METRICS 테이블을 쿼리하세요.

```
SELECT NAME, SUBSYSTEM, COMMENT FROM INFORMATION_SCHEMA.INNODB_METRICS WHERE NAME LIKE '%truncate%';
```

카운터 활성화 및 카운터 데이터 쿼리에 대한 자세한 내용은 섹션 15.15.6, "InnoDB 정보_SCHEMA 메트릭 테이블"을 참조하십시오.

테이블 공간 잘림 제한 실행 취소

체크포인트 간에 동일한 실행 취소 테이블 스페이스에 대한 잘라내기 작업의 수는 다음과 같이 제한됩니다. 64. 이 제한은 예를 들어 사용량이 많은 시스템에서 innodb_max_undo_log_size가너무 낮게 설정된 경우 발생할 수 있는 과도한 수의 실행 취소 테이블 공간 잘림 작업으로 인한 잠재적 문제를 방지합니다. 이 제한을 초과하는 경우에도 실행 취소 테이블스페이스는 여전히 비활성 상태가 될 수 있지만 다음 체크포인트가 지날 때까지 잘리지 않습니다. MySQL 8.2에서 제한은 50000입니다.

테이블 공간 잘림 복구 실행 취소

테이블스페이스 잘라내기 실행 취소 작업은 서버 로그 디렉터리에 임시 *undo_space_number_trunc*.log 파일을 생성합니다. 이 로그 디렉터리는 innodb_log_group_home_dir에 의해 정의됩니다. 잘라내기 작업 중에 시스템 장애가 발생하는 경우, 임시 로그 파일을 통해 시작 프로세스가 잘라낸 테이블스페이스를 식별하고 작업을 계속할 수 있습니다.

테이블 스페이스 상태 변수 실행 취소

다음 상태 변수를 사용하면 총 실행 취소 테이블 스페이스 수, 암시적(InnoDB에서 생성한) 실행 취소 테이블 스페이스, 명시적(사용자가 생성한) 실행 취소 테이블 스페이스 및 활성 실행 취소 테이블 스페이스 수를 추적할 수 있습니다:

```
mysql> SHOW STATUS LIKE 'Innodb_undo_tablespaces%';
+----------------------------------+-----------+
| 변수_이름                        | 값        |
+----------------------------------+-----------+
| Innodb_undo_tablespaces_total    | 2         |
| Innodb_undo_tablespaces_implicit | 2         |
| Innodb_undo_tablespaces_explicit | 0         |
| Innodb_undo_tablespaces_active   | 2         |
+----------------------------------+-----------+
```

상태 변수에 대한 설명은 5.1.10절. "서버 상태 변수"를 참조하세요.

15.6.3.5 임시 테이블 공간

InnoDB는 세션 임시 테이블스페이스와 글로벌 임시 테이블스페이스를 사용합니다.

세션 임시 테이블 공간

세션 임시 테이블 스페이스에는 사용자가 생성한 임시 테이블과 InnoDB가 온디스크 내부 임시 테이블의 스토리지 엔진으로 구성된 경우 옵티마이저가 생성한 내부 임시 테이블이 저장됩니다. 온디스크 내부 임시 테이블은 InnoDB 스토리지 엔진을 사용합니다.

세션 임시 테이블 공간은 디스크에 임시 테이블을 생성하기 위한 첫 번째 요청 시 임시 테이블 공간 풀에서 세션에 할당됩니다. 세션에는 최대 2개의 테이블스페이스가 할당되는데, 하나는 사용자가 만든 임시 테이블용이고 다른 하나는 내부에서 만든 임시 테이블용입니다.
에 의해 할당됩니다. 세션에 할당된 임시 테이블 공간은 세션에서 만든 모든 온디스크 임시 테이블에 사용됩니다. 세션 연결이 끊어지면 해당 임시 테이블 공간이 잘립니다.

풀로 다시 릴리스됩니다. 서버가 시작되면 10개의 임시 테이블스페이스로 구성된 풀이 생성됩니다. 풀의 크기는 줄어들지 않으며 필요에 따라 테이블스페이스가 풀에 자동으로 추가됩니다. 임시 테이블스페이스 풀은 정상 종료 또는 중단된 초기화 시 제거됩니다. 세션 임시 테이블스페이스 파일은 생성 시 5페이지 크기이며 파일 이름 확장자는 `.ibt`입니다.

세션 임시 테이블스페이스에는 40만 개의 공간 ID 범위가 예약되어 있습니다. 세션 임시 테이블스페이스 풀은 서버가 시작될 때마다 다시 생성되므로 서버가 종료될 때 세션 임시 테이블스페이스의 공간 ID는 유지되지 않으며 재사용될 수 있습니다.

`innodb_temp_tablespaces_dir` 변수는 세션 임시 테이블스페이스가 생성되는 위치를 정의합니다. 기본 위치는 데이터 디렉터리의 `#innodb_temp` 디렉터리입니다. 임시 테이블스페이스 풀을 만들 수 없는 경우 시작이 거부됩니다.

```
cd BASEDIR/data/#innodb_temp
$> ls
TEMP_10.IBT TEMP_2.IBT TEMP_4.IBT TEMP_6.IBT TEMP_8.IBT
TEMP_1.IBTTEMP_3.      IBT TEMP_5.IBT TEMP_7.IBT TEMP_9.IBT
```

SBR(문 기반 복제) 모드에서는 복제본에서 생성된 임시 테이블이 단일 세션 임시 테이블 공간에 상주하며, 이 임시 테이블은 MySQL 서버가 종료될 때만 잘립니다.

`INNODB_SESSION_TEMP_TABLESPACES` 테이블은 세션 임시 테이블 스페이스에 대한 메타데이터를 제공합니다.

정보 스키마 `INNODB_TEMP_TABLE_INFO` 테이블은 `InnoDB` 인스턴스에서 활성화된 사용자가 생성한 임시 테이블에 대한 메타데이터를 제공합니다.

글로벌 임시 테이블 스페이스

글로벌 임시 테이블 스페이스(`ibtmp1`)에는 사용자가 만든 임시 테이블의 변경 사항에 대한 롤백 세그먼트가 저장됩니다.

`innodb_temp_data_file_path` 변수는 글로벌 임시 테이블스페이스 데이터 파일의 상대 경로, 이름, 크기 및 속성을 정의합니다. `innodb_temp_data_file_path`에 값을 지정하지 않으면 기본 동작은 `innodb_data_home_dir` 디렉터리에 `ibtmp1`이라는 이름의 단일 자동 확장 데이터 파일을 생성하는 것입니다. 초기 파일 크기는 12MB보다 약간 큽니다.

전역 임시 테이블스페이스는 정상 종료 또는 중단된 초기화 시 제거되고 서버가 시작될 때마다 다시 생성됩니다. 글로벌 임시 테이블스페이스는 생성될 때 동적으로 생성된 공간 ID를 받습니다. 글로벌 임시 테이블스페이스를 만들 수 없는 경우 시작이 거부됩니다. 서버가 예기치 않게 중단되는 경우 글로벌 임시 테이블스페이스는 제거되지 않습니다. 이 경우 데이터베이스 관리자가 글로벌 임시 테이블스페이스를 수동으로 제거하거나 MySQL 서버를 다시 시작할 수 있습니다. MySQL 서버를 다시 시작하면 글로벌 임시 테이블 스페이스가 자동으로 제거되고 다시 생성됩니다.

글로벌 임시 테이블 스페이스는 원시 장치에 상주할 수 없습니다.

정보 스키마 `FILES` 테이블은 글로벌 임시 테이블 스페이스에 대한 메타데이터를 제공합니다. 이와 유사한 쿼리를 실행하여 글로벌 임시 테이블스페이스 메타데이터를 확인합니다:

```
mysql> SELECT * FROM INFORMATION_SCHEMA.FILES WHERE TABLESPACE_NAME='innodb_temporary'\G
```

기본적으로 글로벌 임시 테이블스페이스 데이터 파일은 자동 확장되며 필요에 따라 크기가 증가합니다.

글로벌 임시 테이블 스페이스 데이터 파일이 자동 확장되는지 확인하려면 `innodb_temp_data_file_path` 설정:

```
mysql> SELECT @@innodb_temp_data_file_path;
+------------------------------+
| @@innodb_temp_data_file_path |
+------------------------------+
| ibtmp1:12M:자동 확장           |
```

```
+-----------------------------+
```

글로벌 임시 테이블스페이스 데이터 파일의 크기를 확인하려면 정보 스키마 FILES를 살펴보십시오. 테이블에 이와 유사한 쿼리를 사용합니다:

```
mysql> SELECT FILE_NAME, TABLESPACE_NAME, ENGINE, INITIAL_SIZE, TOTAL_EXTENTS*EXTENT_SIZE
       AS TotalSizeBytes, DATA_FREE, MAXIMUM_SIZE FROM INFORMATION_SCHEMA.FILES
       WHERE TABLESPACE_NAME = 'innodb_temporary'\G
*************************** 1. 행 ***************************
      FILE_NAME: ./ibtmp1
테이블스페이스_이름: innodb_temporary 엔진:
         InnoDB
   초기 크기: 12582912
 총 크기 바이트: 12582912
      DATA_FREE: 6291456
   maximum_size: null
```

TotalSizeBytes는 글로벌 임시 테이블 스페이스 데이터 파일의 현재 크기를 표시합니다. 다른 필드 값에 대한 자세한 내용은 26.3.15절. "정보 스키마 파일 테이블"을 참조하십시오.

또는 운영 체제에서 글로벌 임시 테이블스페이스 데이터 파일 크기를 확인하세요. 글로벌 임시 테이블스페이스 데이터 파일은 innodb_temp_data_file_path 변수에 의해 정의된 디렉터리에 있습니다.

글로벌 임시 테이블스페이스 데이터 파일이 차지하는 디스크 공간을 회수하려면 MySQL 서버를 다시 시작합니다. 서버를 다시 시작하면 innodb_temp_data_file_path에 정의된 속성에 따라 글로벌 임시 테이블스페이스 데이터 파일이 제거되고 다시 생성됩니다.

글로벌 임시 테이블스페이스 데이터 파일의 크기를 제한하려면 다음을 구성합니다.
를사용하여최대 파일 크기를 지정합니다. 예를 들어

```
[mysqld]
innodb_temp_data_file_path=ibtmp1:12M:autoextend:max:500M
```

innodb_temp_data_file_path를 구성하려면 서버를 다시 시작해야 합니다.

15.6.3.6 서버가 오프라인 상태일 때 테이블스페이스 파일 이동하기

시작 시 테이블스페이스 파일을 검색할 디렉터리를 정의하는 innodb_directories 변수는 서버가 오프라인 상태일 때 테이블스페이스 파일을 새 위치로 이동하거나 복원할 수 있도록 지원합니다. 시작 중에 검색된 테이블스페이스 파일은 데이터 사전에서 참조된 파일 대신 사용되며, 데이터 사전은 재배치된 파일을 참조하도록 업데이트됩니다. 중복 테이블스페이스 파일이 검색된 경우
검사 결과, 동일한 테이블스페이스 ID에 대해 여러 파일이 발견되었다는 오류와 함께 시작이 실패합니다.

innodb_data_home_dir, innodb_undo_directory, datadir 변수에 의해 정의된 디렉터리는 innodb_directories 인자 값에 자동으로 추가됩니다. 이러한 디렉터리는 innodb_directories 설정이 명시적으로 지정되었는지 여부와 관계없이 시작 시 검색됩니다. 이러한 디렉터리가 암시적으로 추가되면 innodb_directories 설정을 구성하지 않고도 시스템 테이블스페이스 파일, 데이터 디렉터리 또는 테이블스페이스 파일 실행 취소를 이동할 수 있습니다. 그러나 디렉터리가 변경되면 설정을 업데이트해야 합니다. 예를 들어, 데이터 디렉터리를 재배치한 후 서버를 다시 시작하기 전에 --datadir 설정을 업데이트해야 합 니 다 .

innodb_directories 변수는 시작 명령 또는 MySQL 옵션 파일에서 지정할 수 있습니다. 일부 명령 해석기에서는 세미콜론(;)이 특수 문자로 해석되므로 인수 값 주위에 따옴표가 사용됩니다. (예를 들어, Unix 셸은 이를 명령 종결자로 취급합니다.)

시작 명령:

```
mysqld --innodb-directories="directory_path_1;directory_path_2"
```

MySQL 옵션 파일입니다:

```
[mysqld]
innodb_directories="directory_path_1;directory_path_2"
```

다음 절차는 개별 테이블별 파일 및 일반 테이블 스페이스 파일, 시스템 테이블 스페이스 파일, 테이블 스페이스 파일 실행 취소 또는 데이터 디렉터리를 이동하는 데 적용됩니다. 파일 또는 디렉터리를 이동하기 전에 다음 사용 참고 사항을 검토하세요.

1. 서버를 중지합니다.
2. 테이블스페이스 파일 또는 디렉터리를 원하는 위치로 이동합니다.
3. 새 디렉터리를 InnoDB에 알립니다.
   - 개별 테이블별 파일 또는 일반 테이블 스페이스 파일을 이동하는 경우 알 수 없는 디렉터리를 innodb_directories 값입니다.
     - innodb_data_home_dir, innodb_undo_directory, datadir 변수에 의해 정의된 디렉터리는 innodb_directories 인수 값에 자동으로 추가되므로 이를 지정할 필요가 없습니다.
     - 테이블별 테이블스페이스 파일은 스키마와 이름이 같은 디렉터리로만 이동할 수 있습니다. 예를 들어 배우 테이블이 sakila 스키마에 속하는 경우 actor.ibd 데이터 파일은 sakila라는 이름의 디렉토리로만 이동할 수 있습니다.
     - 일반 테이블스페이스 파일은 데이터 디렉터리 또는 데이터 디렉터리의 하위 디렉터리로 이동할 수 없습니다.
   - 시스템 테이블 스페이스 파일, 테이블 스페이스 실행 취소 또는 데이터 디렉터리를 이동하는 경우 필요에 따라 innodb_data_home_dir, innodb_undo_directory 및 datadir 설정을 변경합니다.
4. 서버를 다시 시작합니다.

사용 참고 사항

- 와일드카드 표현식은 innodb_directories 인수 값에 사용할 수 없습니다.
- innodb_directories 스캔은 지정된 디렉터리의 하위 디렉터리도 검색합니다. 중복된 디렉터리와 하위 디렉터리는 스캔할 디렉터리 목록에서 삭제됩니다.
- innodb_directories는 InnoDB 테이블스페이스 파일 이동을 지원합니다. InnoDB가 아닌 다른 스토리지 엔진에 속한 파일 이동은 지원되지 않습니다. 이 제한은 전체 데이터 디렉터리를 이동할 때도 적용됩니다.
- innodb_directories는 스캔한 디렉터리로 파일을 이동할 때 테이블스페이스 파일 이름 변경을 지원합니다. 또한 테이블스페이스 파일을 지원되는 다른 운영 체제로 이동하는 기능도 지원합니다.
- 테이블스페이스 파일을 다른 운영 체제로 옮길 때는 테이블스페이스 파일 이름에 대상 시스템에서 금지된 문자나 특별한 의미가 있는 문자가 포함되어 있지 않은지 확인합니다.
- Windows 운영 체제에서 Linux 운영 체제로 데이터 디렉터리를 이동할 때는 바이너리 로그 인덱스 파일의 바

이너리 로그 파일 경로를 슬래시 대신 백슬래시를 사용하도록 수정하세요. 기본적으로 바이너리 로그 인덱스 파일의 기본 이름은 바이너리 로그 파일과 동일하며 확장자는 '.index'입니다. 바이너리 로그 인덱스 파일의 위치는 --log-bin으로 정의합니다. 기본 위치는 데이터 디렉터리입니다.

- 테이블스페이스 파일을 다른 운영 체제로 이동하는 경우 플랫폼 간 복제가 발생하는 경우 플랫폼별 디렉터리가 포함된 DDL 문이 올바르게 복제되도록 하는 것은 데이터베이스 관리자의 책임입니다. 디렉터리를 지정할 수 있는 문에는 CREATE TABLE ... 데이터 디렉터리 및 CREATE TABLESPACE ... ADD 데이터 파일.
- 절대 경로 또는 데이터 디렉터리 외부 위치에 생성된 테이블별 파일 및 일반 테이블 스페이스의 디렉터리를 innodb_directories 설정에 추가합니다. 그렇지 않으면 복구 중에 InnoDB가 파일을 찾을 수 없습니다. 관련 정보는 충돌 복구 중 테이블 공간 검색을 참조하십시오.

  테이블스페이스 파일 위치를 보려면 정보 스키마 FILES 테이블을 쿼리합니다:

```
mysql> SELECT TABLESPACE_NAME, FILE_NAME FROM INFORMATION_SCHEMA.FILES \G
```

15.6.3.7 테이블 공간 경로 유효성 검사 사용 안 함

InnoDB는 시작 시 innodb_directories 변수로 정의된 디렉터리에서 테이블스페이스 파일을 검색합니다. 검색된 테이블스페이스 파일의 경로는 데이터 사전에 기록된 경로와 비교하여 유효성을 검사합니다. 경로가 일치하지 않으면 데이터 사전의 경로가 업데이트됩니다.

innodb_validate_tablespace_paths 변수를 사용하면 테이블스페이스 경로 유효성 검사를 비활성화할 수 있습니다. 이 기능은 테이블스페이스 파일을 이동하지 않는 환경을 위한 것입니다.
경로 유효성 검사를 비활성화하면 테이블스페이스 파일이 많은 시스템에서 시작 시간이 향상됩니다.
log_error_verbosity를 3으로 설정하면 테이블스페이스 경로 유효성 검사를 비활성화할 때 시작 시 다음 메시지가 인쇄됩니다:

```
[InnoDB] InnoDB 테이블 스페이스 경로 유효성 검사 건너뛰기. 수동으로
이동한 테이블스페이스 파일은 감지되지 않습니다!
```

경고

테이블스페이스 파일을 이동한 후 테이블스페이스 경로 유효성 검사를 비활성화한 상태로 서버를 시작하면 정의되지 않은 동작이 발생할 수 있습니다.

15.6.3.8 Linux에서 테이블 공간 할당 최적화하기

Linux에서 InnoDB가 테이블별 파일 및 일반 테이블 공간에 공간을 할당하는 방식을 최적화할 수 있습니다. 기본적으로 추가 공간이 필요한 경우 InnoDB는 테이블 공간에 페이지를 할당하고 해당 페이지에 물리적으로 NULL을 씁니다. 이 동작은 새 페이지가 자주 할당되는 경우 성능에 영향을 미칠 수 있습니다. 새로 할당된 테이블스페이스 페이지에 물리적으로 NULL을 쓰지 않도록 하려면 Linux 시스템에서 innodb_extend_and_initialize를 비활성화하면 됩니다. innodb_extend_and_initialize가 비활성화되면 물리적으로 NULL을 쓰지 않고 공간을 예약하는 posix_fallocate() 호출을 사용하여 테이블스페이스 파일에 공간이 할당됩니다.

posix_fallocate() 호출을 사용하여 페이지를 할당하는 경우 기본적으로 확장자 크기가 작고 페이지가 한 번에 몇 개만 할당되는 경우가 많으므로 조각화가 발생하고 무작위 I/O가 증가할 수 있습니다. 이 문제를 방지하려면 posix_fallocate() 호출을 활성화할 때 테이블스페이스 확장 크기를 늘리세요. 테이블스페이스 확장 크기는 최대 4GB까지 늘릴 수 있습니다.
옵션을 사용하여 설정합니다. 자세한 내용은 섹션 15.6.3.9, "테이블 스페이스 AUTOEXTEND_SIZE 구성"을 참조하십시오.

InnoDB는 새 테이블스페이스 페이지를 할당하기 전에 재실행 로그 레코드를 기록합니다. 페이지 할당 작업이 중단되면 복구 중에 재실행 로그 레코드에서 작업이 다시 재생됩니다.
(재실행 로그 레코드에서 재생된 페이지 할당 작업은 물리적으로 새로 할당된 페이지에 NULL을 씁니다.) 재실행 로그 레코드는 innodb_extend_and_initialize 설정에 관계없이 페이지를 할당하기 전에 기록됩니다.

Linux 이외의 시스템 및 Windows에서 `InnoDB`는 테이블 스페이스에 새 페이지를 할당하고 해당 페이지에 물리적으로 NULL을 쓰는데, 이것이 기본 동작입니다. 이러한 시스템에서 `innodb_extend_and_initialize`를 비활성화하려고 시도하면 다음 오류가 반환됩니다:

```
이 플랫폼에서는 innodb_extend_and_initialize 변경이 지원되지 않습니다. 기본값으로 돌아갑니다.
```

15.6.3.9 테이블스페이스 AUTOEXTEND_SIZE 구성

기본적으로 테이블별 파일 또는 일반 테이블 공간에 추가 공간이 필요한 경우 테이블 공간은 다음 규칙에 따라 점진적으로 확장됩니다:

- 테이블 스페이스의 크기가 한 범위보다 작으면 한 번에 한 페이지씩 확장됩니다.
- 테이블 스페이스의 크기가 1엑세스보다 크지만 32엑세스보다 작은 경우 한 번에 한 엑세스씩 확장됩니다.

- 테이블스페이스의 크기가 32엑스텐트를 초과하는 경우 한 번에 4개씩 확장됩니다. 익스텐

션 크기에 대한 자세한 내용은 섹션 15.11.2, "파일 공간 관리"를 참조하십시오.

테이블별 파일 또는 일반 테이블 스페이스가 확장되는 양은 AUTOEXTEND_SIZE 옵션을 지정하여 구성할 수 있습니다. 확장 크기를 크게 구성하면 조각화를 방지하고 대량의 데이터를 쉽게 수집할 수 있습니다.

테이블별 파일 테이블스페이스에 대한 확장자 크기를 구성하려면, AUTOEXTEND_SIZE 크기를 CREATE TABLE 또는 ALTER TABLE 문을 사용합니다:

```
CREATE TABLE t1 (c1 INT) AUTOEXTEND_SIZE = 4M;

ALTER TABLE t1 AUTOEXTEND_SIZE = 8M;
```

일반 테이블스페이스에 대한 확장 크기를 구성하려면, AUTOEXTEND_SIZE 크기를 테이블스페이스 생성 또는 테이블스페이스 변경 문을 사용합니다:

```
CREATE TABLESPACE ts1 AUTOEXTEND_SIZE = 4M;

ALTER TABLESPACE ts1 AUTOEXTEND_SIZE = 8M;
```

참고

실행 취소 테이블스페이스를 만들 때도 AUTOEXTEND_SIZE 옵션을 사용할 수 있지만 실행 취소 테이블스페이스에 대한 확장 동작이 다릅니다. 자세한 내용은 섹션 15.6.3.4, "테이블 스페이스 실행 취소"를 참조하십시오.

AUTOEXTEND_SIZE 설정은 4M의 배수여야 합니다. 4M의 배수가 아닌 AUTOEXTEND_SIZE 설정을 지정하면 오류가 반환됩니다.

AUTOEXTEND_SIZE 기본 설정은 0으로, 위에서 설명한 기본 동작에 따라 테이블 스페이스가 확장됩니다.

허용되는 최대 자동 확장 크기는 4GB입니다.

다음 표에 표시된 대로 최소 AUTOEXTEND_SIZE 설정은 InnoDB 페이지 크기에 따라 달라집니다:

InnoDB 페이지 크기	최소 자동 확장 크기
4K	4M
8K	4M
16K	4M
32K	8M
64K	16M

기본 InnoDB 페이지 크기는 16K(16384바이트)입니다. MySQL 인스턴스의 InnoDB 페이지 크기를 확인하려면 innodb_page_size 설정을 쿼리하세요:

```
mysql> SELECT @@GLOBAL.innodb_page_size;
+---------------------------+
| @@GLOBAL.innodb_page_size |
+---------------------------+
                     |16384 |
+---------------------------+
```

테이블스페이스에 대한 AUTOEXTEND_SIZE 설정이 변경되면 이후에 발생하는 첫 번째 확장은 테이블스페이스 크기를 AUTOEXTEND_SIZE 설정의 배수만큼 증가시킵니다. 이후의 확장은 구성된 크기입니다.

테이블별 파일 또는 일반 테이블 스페이스가 0이 아닌 AUTOEXTEND_SIZE 설정으로 생성되면 테이블 스페이스는 지정된 AUTOEXTEND_SIZE 크기로 초기화됩니다.

ALTER TABLESPACE를 사용하여 테이블별 파일 테이블 스페이스의 AUTOEXTEND_SIZE를 구성할 수 없습니다. ALTER TABLE을 사용해야 합니다.

파일 단위 테이블 공간에서 생성된 테이블의 경우 테이블 만들기 표시에는 AUTOEXTEND_SIZE 옵션을 0이 아닌 값으로 구성한 경우에만 사용할 수 있습니다.

InnoDB 테이블스페이스에 대한 AUTOEXTEND_SIZE를 확인하려면 정보 스키마를 쿼리합니다. INNODB_TABLESPACES 테이블. 예를 들어

```
mysql> SELECT NAME, AUTOEXTEND_SIZE FROM INFORMATION_SCHEMA.INNODB_TABLESPACES
       WHERE NAME LIKE 'test/t1';
+---------+------------------+
| 이름|     자동 확장 크기 |
+---------+------------------+
test/t1              |4194304 |4194304 |
+---------+------------------+

mysql> SELECT NAME, AUTOEXTEND_SIZE FROM INFORMATION_SCHEMA.INNODB_TABLESPACES
       WHERE NAME LIKE 'ts1';
+------+ -----------------+
| 이름 | 자동 확장 크기 |
+------+ -----------------+
| ts1             |4194304 |
+------+ -----------------+
```

참고

기본 설정인 AUTOEXTEND_SIZE가 0이면 위에서 설명한 기본 테이블 스페이스 확장 동작에 따라 테이블 스페이스가 확장된다는 의미입니다.

15.6.4 이중 쓰기 버퍼

이중 쓰기 버퍼는 InnoDB 데이터 파일의 적절한 위치에 페이지를 쓰기 전에 버퍼 풀에서 플러시된 페이지를 InnoDB가 쓰는저장 영역입니다. 페이지 쓰기 도중에 운영 체제, 스토리지 하위 시스템 또는 예기치 않은 mysqld 프로세스 종료가 발생하는 경우, InnoDB는 충돌 복구 중에 이중 쓰기 버퍼에서 페이지의 올바른 복사본을 찾을 수 있습니다.

데이터가 두 번 쓰여지지만, 이중 쓰기 버퍼는 두 배의 I/O 오버헤드나 두 배의 I/O 작업을 필요로 하지 않습니다. 데이터는 운영 체제에 대한 fsync() 호출 한 번으로 큰 순차 청크로 이중 쓰기 버퍼에 기록됩니다 (innodb_flush_method가 O_DIRECT_NO_FSYNC로 설정된 경우 제외).

이중 쓰기 버퍼 저장 영역은 이중 쓰기 파일에 있습니다.

이중 쓰기 버퍼 구성을 위해 제공되는 변수는 다음과 같습니다:

- innodb_doublewrite

  innodb_doublewrite 변수는 이중 쓰기 버퍼의 사용 여부를 제어합니다. 대부분의 경우 기본적으로 활성화되어 있습니다. 이중 쓰기 버퍼를 비활성화하려면 innodb_doublewrite를 OFF로 설정합니다. 예를 들어 벤치마크를 수행할 때처럼 데이터 무결성보다 성능이 더 중요한 경우에는 이중 쓰기 버퍼를 비활성화하는 것이 좋습니다.

  innodb_doublewrite는 DETECT_AND_RECOVER 및 DETECT_ONLY 설정을 지원합니다.

DETECT_AND_RECOVER 설정은 ON 설정과 동일합니다. 이 설정을 사용하면 이중 쓰기 버퍼가 완전히 활성화되어 데이터베이스 페이지 콘텐츠가 복구 중에 액세스되는 이중 쓰기 버퍼에 기록되어 불완전한 페이지 쓰기를 수정합니다.

DETECT_ONLY 설정을 사용하면 메타데이터만 이중 쓰기 버퍼에 기록됩니다. 데이터베이스 페이지 콘텐츠는 이중 쓰기 버퍼에 기록되지 않으며 복구 시 이중 쓰기 버퍼를 사용하여 불완전한 페이지 쓰기를 수정하지 않습니다. 이 경량 설정은 불완전한 페이지 쓰기만 감지하기 위한 것입니다.

MySQL은 이중 쓰기 버퍼를 활성화하는 innodb_doublewrite 설정의 동적 변경을 ON, DETECT_AND_RECOVER 및 DETECT_ONLY 사이에서 지원합니다. MySQL은 이중 쓰기 버퍼를 활성화하는 설정에서 OFF로 또는 그 반대로의 동적 변경을 지원하지 않습니다.

이중 쓰기 버퍼가 원자 쓰기를 지원하는 Fusion-io 장치에 있는 경우, 이중 쓰기 버퍼는 자동으로 비활성화되고 데이터 파일 쓰기는 대신 Fusion-io 원자 쓰기를 사용하여 수행됩니다. 그러나 innodb_doublewrite 설정은 전역 설정이라는 점에 유의하십시오. 이중 쓰기 버퍼가 비활성화되면 Fusion-io 하드웨어에 상주하지 않는 파일을 포함한 모든 데이터 파일에 대해 이 기능이 비활성화됩니다. 이 기능은 Fusion-io 하드웨어에서만 지원되며 Fusion-
io NVMFS를 지원합니다. 이 기능을 최대한 활용하려면 innodb_flush_method 설정의 O_DIRECT를 권장합니다.

- innodb_doublewrite_dir

  innodb_doublewrite_dir 변수는 InnoDB가 이중 쓰기 파일을 생성하는 디렉터리를 정의합니다. 디렉터리가 지정되지 않은 경우 기본적으로 데이터 디렉터리인 innodb_data_home_dir 디렉터리에 이중 쓰기 파일이 생성됩니다.

  스키마 이름과의 충돌을 피하기 위해 지정된 디렉토리 이름 앞에 해시 기호 '#'이 자동으로 접두사로 붙습니다. 그러나 디렉터리 이름에 '.', '#' 또는 '/' 접두사가 명시적으로 지정된 경우에는 디렉터리 이름에 해시 기호 '#'가 접두사로 추가되지 않습니다.

  이중 쓰기 디렉터리는 사용 가능한 가장 빠른 저장 매체에 배치하는 것이 가장 이상적입니다.

- innodb_doublewrite_files

  innodb_doublewrite_files 변수는 이중 쓰기 파일의 수를 정의합니다. 기본적으로 각 버퍼 풀 인스턴스에 대해 두 개의 이중 쓰기 파일이 생성됩니다: 플러시 목록 이중 쓰기 파일과 LRU 목록 이중 쓰기 파일입니다.

  플러시 목록 이중 쓰기 파일은 버퍼 풀 플러시 목록에서 플러시된 페이지를 위한 파일입니다. 플러시 목록 이중 쓰기 파일의 기본 크기는 InnoDB 페이지 크기 * 이중 쓰기 페이지 바이트입니다.

  LRU 목록 이중 쓰기 파일은 버퍼 풀 LRU 목록에서 플러시된 페이지용입니다. 단일 페이지 플러시를 위한 슬롯도 포함되어 있습니다. LRU 목록 이중 쓰기 파일의 기본 크기는 InnoDB 페이지 크기 *(이중 쓰기 페이지 + (512 / 버퍼 풀 인스턴스 수))이며, 여기서 512는 단일 페이지 플러시를 위해 예약된 슬롯의 총 개수입니다.

  이중 쓰기 파일은 최소 두 개입니다. 최대 이중 쓰기 파일 수는 버퍼 풀 인스턴스 수의 두 배입니다. (버퍼 풀 인스턴스 수는 innodb_buffer_pool_instances 변수에 의해 제어됩니다.)

  이중 쓰기 파일 이름은 다음과 같은 형식을 갖습니다: *#ib_page_size_file_number*.dblwr (또는 .bdblwr로 변경할 수 있습니다(DETECT_ONLY 설정 사용). 예를 들어, InnoDB 페이지 크기가 16KB이고 버퍼 풀이 단일인 MySQL 인스턴스에 대해 다음과 같은 이중 쓰기 파일이 만들어집니다:

```
#ib_16384_0.dblwr
#ib_16384_1.dblwr
```

`innodb_doublewrite_files` 변수는 고급 성능 튜닝을 위한 것입니다. 기본 설정은 대부분의 사용자에게 적합할 것입니다.

- `innodb_doublewrite_pages`

  `innodb_doublewrite_pages` 변수는 스레드당 최대 이중 쓰기 페이지 수를 제어합니다. 값을 지정하지 않으면 `innodb_doublewrite_pages`가 `innodb_write_io_threads` 값으로 설정됩니다. 이 변수는 고급 성능 튜닝을 위한 것입니다. 기본값은 대부분의 사용자에게 적합합니다.

- `INNODB_DOUBLEWRITE_BATCH_SIZE`

innodb_doublewrite_batch_size 변수는 일괄적으로 작성할 이중 쓰기 페이지 수를 제어합니다. 이 변수는 고급 성능 튜닝을 위한 것입니다. 기본값은 대부분의 사용자에게 적합합니다.

InnoDB는 암호화된 테이블 스페이스에 속하는 이중 쓰기 파일 페이지를 자동으로 암호화합니다(섹션 15.13, "InnoDB 저장 데이터 암호화" 참조). 마찬가지로 페이지 압축 테이블스페이스에 속하는 이중 쓰기 파일 페이지도 압축됩니다. 따라서 이중 쓰기 파일에는 암호화되지 않은 페이지와 압축되지 않은 페이지, 암호화된 페이지, 압축된 페이지, 암호화 및 압축된 페이지 등 다양한 페이지 유형이 포함될 수 있습니다.

15.6.5 로그 다시 실행

재실행 로그는 불완전한 트랜잭션에 의해 쓰여진 데이터를 수정하기 위해 충돌 복구 중에 사용되는 디스크 기반 데이터 구조입니다. 정상적인 작업 중에 재실행 로그는 SQL 문 또는 하위 수준 API 호출로 인해 발생하는 테이블 데이터 변경 요청을 인코딩합니다. 예기치 않은 종료 전에 데이터 파일 업데이트를 완료하지 못한 수정 사항은 초기화 중 및 연결이 수락되기 전에 자동으로 재생됩니다. 충돌 복구에서 재실행 로그의 역할에 대한 자세한 내용은 섹션 15.18.2, "InnoDB 복구"를 참조하세요.

재실행 로그는 물리적으로 재실행 로그 파일로 디스크에 표시됩니다. 재실행 로그 파일에 기록되는 데이터는 영향을 받는 레코드의 관점에서 인코딩되며, 이 데이터를 통칭하여 재실행이라고 합니다. 재실행 로그 파일을 통한 데이터의 통과는 계속 증가하는 LSN 값으로 표시됩니다. 재실행 로그 데이터는 데이터 수정이 발생하면 추가되며, 체크포인트가 진행됨에 따라 가장 오래된 데이터는 잘립니다.

재실행 로그와 관련된 정보 및 절차는 섹션의 다음 항목에 설명되어 있습니다:

- 재실행 로그 용량 구성
- 자동 재실행 로그 용량 구성
- 로그 아카이빙 재실행
- 재실행 로깅 비활성화
- 관련 주제

재실행 로그 용량 구성

innodb_redo_log_capacity 시스템 변수는 재실행 로그 파일이 차지하는 디스크 공간의 양을 제어합니다. 이 변수는 시작 시 옵션 파일에서 설정하거나 런타임에 SET GLOBAL 문을 사용하여 설정할 수 있습니다(예: 다음 문은 재실행 로그 용량을 8GB로 설정합니다):

```
SET GLOBAL innodb_redo_log_capacity = 8589934592;
```

런타임에 설정하면 구성 변경이 즉시 적용되지만 새 제한이 완전히 구현되는 데는 시간이 걸릴 수 있습니다. 재실행 로그 파일이 지정된 값보다 적은 공간을 차지하면 더티 페이지가 버퍼 풀에서 테이블 공간 데이터 파일로

덜 공격적으로 플러시되어 결국 재실행 로그 파일이 차지하는 디스크 공간이 증가합니다. 재실행 로그 파일이 지정된 값보다 더 많은 공간을 차지하는 경우 더티 페이지가 더 적극적으로 플러시되어 결국 재실행 로그 파일이 차지하는 디스크 공간이 감소합니다.

`innodb_redo_log_capacity` 변수는 더 이상 사용되지 않는 `innodb_log_files_in_group` 및 `innodb_log_file_size` 변수를 대체합니다. `innodb_redo_log_capacity` 설정이 정의된 경우, `innodb_log_files_in_group` 및 `innodb_log_file_size` 설정은 무시되며, 그렇지 않은 경우 이 설정은 `innodb_redo_log_capacity` 설정(`innodb_log_files_in_group` *)을 계산하는 데 사용됩니다. `innodb_log_file_size` = `innodb_redo_log_capacity`). 이러한 변수가 설정되지 않은 경우, 재실행 로그 용량은 기본값인 104857600 바이트(100MB)인 `innodb_redo_log_capacity`로 설정됩니다. 최대 재실행 로그 용량은 128GB입니다.

재실행 로그 파일은 innodb_log_group_home_dir 변수에 의해 다른 디렉터리가 지정되지 않는 한 데이터 디렉터리의 #innodb_redo 디렉터리에 있습니다. innodb_log_group_home_dir이 정의된 경우, 재실행 로그 파일은 해당 디렉터리의 #innodb_redo 디렉터리에 있습니다. 재실행 로그 파일에는 일반 로그 파일과 예비 로그 파일의 두 가지 유형이 있습니다. 일반 재실행 로그 파일은 사용 중인 로그 파일입니다. 예비 재실행 로그 파일은 사용되기를 기다리는 로그 파일입니다. InnoDB는 총 32개의 재실행 로그 파일을 유지하려고 하며, 각 파일의 크기는 1/32 * innodb_redo_log_capacity와 동일하지만, innodb_redo_log_capacity 설정을 수정한 후 파일 크기가 한동안 달라질 수 있습니다.

재실행 로그 파일은 *#ib_redoN* 명명 규칙을 사용하며, 여기서 *N은* 재실행 로그 파일 번호입니다. 예비 재실행 로그 파일은 _tmp 접미사로 표시됩니다. 다음 예제에서는 21개의 활성 재실행 로그 파일과 11개의 예비 재실행 로그 파일이 순차적으로 번호가 매겨진 #innodb_redo 디렉터리의 재실행 로그 파일을 보여 줍니다.

```
'#ib_redo582'   '#ib_redo590'   '#ib_redo598'       '#ib_redo606_tmp'
'#ib_redo583'   '#ib_redo591'   '#ib_redo599'       '#ib_redo607_tmp'
'#ib_redo584'   '#ib_redo592'   '#ib_redo600'       '#ib_redo608_tmp'
'#ib_redo585'   '#ib_redo593'   '#ib_redo601'       '#ib_redo609_tmp'
'#ib_redo586'   '#ib_redo594'   '#ib_redo602'       '#ib_redo610_tmp'
'#ib_redo587'   '#ib_redo595'   '#ib_redo603_tmp'   '#ib_redo611_tmp'
'#ib_redo588'   '#ib_redo596'   '#ib_redo604_tmp'   '#ib_redo612_tmp'
'#ib_redo589'   '#ib_redo597'   '#ib_redo605_tmp'   '#ib_redo613_tmp'
```

예를 들어, 다음 쿼리는 이전 예제에 나열된 활성 재실행 로그 파일에 대한 START_LSN 및 END_LSN 값을 표시합니다:

```
mysql> SELECT FILE_NAME, START_LSN, END_LSN FROM performance_schema.innodb_redo_log_files;
+----------------------------+--------------+ ------------------+

| 파일_이름| 시작_LSN| 끝_LSN    |
+----------------------------+--------------+ ------------------+
| ./#innodb_redo/#ib_redo582 | 117654982144 | 117658256896 |
| ./#innodb_redo/#ib_redo583 | 117658256896 | 117661531648 |
| ./#innodb_redo/#ib_redo584 | 117661531648 | 117664806400 |
| ./#innodb_redo/#ib_redo585 | 117664806400 | 117668081152 |
| ./#innodb_redo/#ib_redo586 | 117668081152 | 117671355904 |
| ./#innodb_redo/#ib_redo587 | 117671355904 | 117674630656 |
| ./#innodb_redo/#ib_redo588 | 117674630656 | 117677905408 |
| ./#innodb_redo/#ib_redo589 | 117677905408 | 117681180160 |
| ./#innodb_redo/#ib_redo590 | 117681180160 | 117684454912 |
| ./#innodb_redo/#ib_redo591 | 117684454912 | 117687729664 |
| ./#innodb_redo/#ib_redo592 | 117687729664 | 117691004416 |
| ./#innodb_redo/#ib_redo593 | 117691004416 | 117694279168 |
| ./#innodb_redo/#ib_redo594 | 117694279168 | 117697553920 |
| ./#innodb_redo/#ib_redo595 | 117697553920 | 117700828672 |
| ./#innodb_redo/#ib_redo596 | 117700828672 | 117704103424 |
| ./#innodb_redo/#ib_redo597 | 117704103424 | 117707378176 |
| ./#innodb_redo/#ib_redo598 | 117707378176 | 117710652928 |
| ./#innodb_redo/#ib_redo599 | 117710652928 | 117713927680 |
| ./#innodb_redo/#ib_redo600 | 117713927680 | 117717202432 |
| ./#innodb_redo/#ib_redo601 | 117717202432 | 117720477184 |
| ./#innodb_redo/#ib_redo602 | 117720477184 | 117723751936 |
+----------------------------+--------------+ ------------------+
```

체크포인트를 수행할 때 InnoDB는 이 LSN이 포함된 파일의 헤더에 체크포인트 LSN을 저장합니다. 복구하는 동안 모든 재실행 로그 파일이 확인되고 최신 체크포인트 LSN에서 복구가 시작됩니다.

재실행 로그 및 재실행 로그 용량 크기 조정 작업을 모니터링하기 위해 몇 가지 상태 변수가 제공됩니다. 예를 들어, Innodb_redo_log_resize_status를 쿼리하여 크기 조정 작업의 상태를 볼 수 있습니다:

```
mysql> SHOW STATUS LIKE 'Innodb_redo_log_resize_status';
+------------------------------+----------+
| 변수_이름                    | 값       |
+------------------------------+----------+
| Innodb_redo_log_resize_status | 확인     |
+------------------------------+----------+
```

Innodb_redo_log_capacity_resized 상태 변수는 현재 재실행 로그 용량 제한을 표시합니다:

```
mysql> SHOW STATUS LIKE 'Innodb_redo_log_capacity_resized';
 +---------------------------------+ --------------+
| 변수_이름| 값                       |
+---------------------------------+ --------------+
| Innodb_redo_log_capacity_resized | 104857600 |
+---------------------------------+ --------------+
```

기타 적용 가능한 상태 변수는 다음과 같습니다:

- Innodb_redo_log_checkpoint_lsn
- Innodb_redo_log_current_lsn
- Innodb_redo_log_flushed_to_disk_lsn
- Innodb_redo_log_logical_size
- Innodb_redo_log_physical_size
- Innodb_redo_log_read_only
- Innodb_redo_log_uuid

자세한 내용은 상태 변수 설명을 참조하세요.

innodb_redo_log_files를 쿼리하여 활성 재실행 로그 파일에 대한 정보를 볼 수 있습니다.
성능 스키마 테이블입니다. 다음 쿼리는 테이블의 모든 열에서 데이터를 검색합니다:

```
SELECT FILE_ID, START_LSN, END_LSN, SIZE_IN_BYTE, IS_FULL, CUSTOMER_LEVEL
FROM performance_schema.innodb_redo_log_files;
```

자동 재실행 로그 용량 구성

innodb_dedicated_server가 활성화되면 InnoDB는 재실행 로그 용량을 비롯한 특정 InnoDB 매개변수를 자동으로 구성합니다. 자동 구성은 MySQL 서버가 사용 가능한 모든 시스템 리소스를 사용할 수 있는 MySQL 전용 서버에 상주하는 MySQL 인스턴스를 위한 것입니다. 자세한 내용은 섹션 15.8.12, '전용 MySQL 서버에 대한 자동 구성 사용'을 참조하세요.

로그 아카이빙 재실행

재실행 로그 레코드를 복사하는 백업 유틸리티가 백업 작업이 진행되는 동안 재실행 로그 생성을 따라가지 못하여 해당 레코드를 덮어쓰는 바람에 재실행 로그 레코드가 손실될 수 있습니다. 이 문제는 백업 작업 중에 MySQL 서버 활동이 많고 재실행 로그 파일 저장 미디어가 백업 저장 미디어보다 빠른 속도로 작동하는 경우에 가장 자주 발생합니다. 재실행 로그 아카이빙 기능은 재실행 로그 파일 외에 아카이브 파일에 재실행 로그 레코드를 순차적으로 기록하여 이 문제를 해결합니다. 백업 유틸리티는 필요에 따라 아카이브 파일에서 재실행 로그 레코드를 복사할 수 있으므로 잠재적인 데이터 손실을 방지할 수 있습니다.

서버에 재실행 로그 아카이빙이 구성되어 있는 경우, MySQL 엔터프라이즈 에디션에서 사용할 수 있는 MySQL 엔터프라이즈 백업은 MySQL 서버를 백업할 때 재실행 로그 아카이빙 기능을 사용합니다.

서버에서 재실행 로그 보관을 사용하도록 설정하려면 innodb_redo_log_archive_dirs 시스템 변수에

값을 설정해야 합니다. 이 값은 레이블이 지정된 재실행 로그 아카이브 디렉터리의 세미콜론으로 구분된 목록으로 지정됩니다. *레이블:디렉토리* 쌍은 콜론(:)으로 구분됩니다. 예를 들어

```
mysql> SET GLOBAL innodb_redo_log_archive_dirs='label1:디렉토리_경로1[;label2:디렉토리_경로2;...]';
```

레이블은 아카이브 디렉터리에 대한 임의의 식별자입니다. 허용되지 않는 콜론(:)을 제외한 모든 문자 문자열을 사용할 수 있습니다. 빈 레이블도 허용되지만 콜론(:)은 다음과 같습니다.

이 경우에도 여전히 필요합니다. *디렉터리_경로를* 지정해야 합니다. 재실행 로그 아카이브 파일로 선택한 디렉터리는 재실행 로그 아카이브가 활성화될 때 존재해야 하며, 그렇지 않으면 오류가 반환됩니다. 경로에는 콜론(':')을 포함할 수 있지만 세미콜론(;)은 허용되지 않습니다.

재실행 로그 아카이빙을 활성화하려면 먼저 `innodb_redo_log_archive_dirs` 변수를 구성해야 합니다. 기본값은 `NULL`이며, 이 경우 재실행 로그 아카이빙을 활성화할 수 없습니다.

참고

지정하는 아카이브 디렉터리는 다음 요구 사항을 충족해야 합니다. (이 요구 사항은 로그 아카이브 재실행이 활성화된 경우에 적용됩니다.):

- 디렉터리가 존재해야 합니다. 디렉터리는 로그 아카이브 재실행 프로세스에 의해 생성되지 않습니다. 그렇지 않으면 다음 오류가 반환됩니다:

  오류 3844 (HY000): 로그 아카이브 디렉터리 '*directory_path1*'이 존재하지 않거나 디렉터리가 아닙니다.

- 디렉터리는 전 세계에서 액세스할 수 없어야 합니다. 이는 재실행 로그 데이터가 시스템에서 권한이 없는 사용자에게 노출되는 것을 방지하기 위한 것입니다. 그렇지 않으면 다음 오류가 반환됩니다:

  오류 3846(HY000): 로그 아카이브 디렉터리 '*directory_path1*'에 모든 OS 사용자가 액세스할 수 있도록 다시 실행합니다.

- 디렉터리는 `datadir`, `innodb_data_home_dir`, `innodb_directories`, `innodb_log_group_home_dir`, `innodb_temp_tablespaces_dir`, `innodb_tmpdir` `innodb_undo_directory` 또는 `secure_file_priv`에 정의된 디렉터리일 수 없으며 해당 디렉터리의 상위 디렉터리 또는 하위 디렉터리일 수 없습니다. 그렇지 않으면 다음과 유사한 에러가 반환됩니다:

  오류 3845 (HY000): 로그 아카이브 디렉터리 '*디렉터리_경로1*'이 서버 디렉터리 'datadir' - '*/path/to/data_directory*' 안에 있거나, 아래에 있거나, 위에 있는 경우 다시 실행합니다.

재실행 로그 아카이빙을 지원하는 백업 유틸리티가 백업을 시작하면, 백업 유틸리티는 `innodb_redo_log_archive_start()` 함수를 호출하여 재실행 로그 아카이빙을 활성화합니다.

재실행 로그 보관을 지원하는 백업 유틸리티를 사용하지 않는 경우에는 그림과 같이 수동으로 재실행 로그 보관을 활성화할 수도 있습니다:

```
mysql> SELECT innodb_redo_log_archive_start('label', 'subdir');
+-----------------------------------------+
| innodb_redo_log_archive_start('label') |
+-----------------------------------------+
| 0                                       |
+-----------------------------------------+
```

또는:

실행

```
mysql> DO innodb_redo_log_archive_start('label', 'subdir');
쿼리 확인, 영향을 받는 행 0개(0.09초)
```

참고

재실행 로그 아카이빙을 활성화(`innodb_redo_log_archive_start()` 사용)하는 MySQL 세션은 아카이빙이 진행되는 동안 열려 있어야 합니다. 동일한 세션에서 재실행 로그 아카이빙을 비활성화해야 합니다 (`innodb_redo_log_archive_stop()` 사용). 재실행 로그 아카이브가 명시적으로 비활성화되기 전에 세션이 종료되면 서버는 재실행 로그 아카이브를 암시적으로 비활성화하고 재실행 로그 아카이브 파일을 제거합니다.

여기서 label은 innodb_redo_log_archive_dirs에 정의된 레이블이고, subdir은 아카이브 파일을 저장하기 위해 *label로* 식별된 디렉터리의 하위 디렉터리를 지정하는 선택적 인자로, 단순한 디렉토리 이름이어야 합니다(슬래시(/), 백슬래시(\) 또는 콜론(:)은 허용되지 않음). subdir은 비어 있거나 널일 수 있거나 생략할 수 있습니다.

INNODB_REDO_LOG_ARCHIVE 권한이 있는 사용자만 innodb_redo_log_archive_start()를 호출하여 재실행 로그 아카이빙을 활성화하거나 다음을 사용하여 비활성화할 수 있습니다. innodb_redo_log_archive_stop(). 백업 유틸리티를 실행하는 MySQL 사용자 또는 재실행 로그 아카이브를 수동으로 활성화 및 비활성화하는 MySQL 사용자에게는 이 권한이 있어야 합니다.

재실행 로그 아카이브 파일 경로는 *directory_identified_by_label/*[*subdir/*]archive.*serverUUID*.000001.log이며, 여기서 directory_identified_by_label은 innodb_redo_log_archive_start()의 *label* 인수로 식별되는 아카이브 디렉토리이고, subdir은 innodb_redo_log_archive_start()에 사용되는 선택적 인수입니다.

예를 들어, 재실행 로그 아카이브 파일의 전체 경로와 이름은 다음과 유사하게 표시됩니다:

```
/directory_path/subdirectory/archive.e71a47dc-61f8-11e9-a3cb-080027154b4d.000001.log
```

백업 유틸리티는 InnoDB 데이터 파일 복사를 완료한 후 innodb_redo_log_archive_stop() 함수를 호출하여 재실행 로그 아카이빙을 비활성화합니다.

재실행 로그 보관을 지원하는 백업 유틸리티를 사용하지 않는 경우에는 그림과 같이 재실행 로그 보관을 수동으로 비활성화할 수도 있습니다:

```
mysql> SELECT innodb_redo_log_archive_stop();
+--------------------------------+
| innodb_redo_log_archive_stop() |
+--------------------------------+
| 0                              |
+--------------------------------+
```

또는:

```
mysql> DO innodb_redo_log_archive_stop();
쿼리 확인, 영향을 받는 행 0개(0.01초)
```

중지 기능이 성공적으로 완료되면 백업 유틸리티가 아카이브 파일에서 재실행 로그 데이터의 관련 섹션을 찾아 백업에 복사합니다.

백업 유틸리티가 재실행 로그 데이터 복사를 완료하고 재실행 로그 아카이브 파일이 더 이상 필요하지 않게 되면 아카이브 파일을 삭제합니다.

정상적인 상황에서는 아카이브 파일을 제거하는 것은 백업 유틸리티의 책임입니다. 그러나 innodb_redo_log_archive_stop()이호출되기 전에 로그 아카이브 재실행 작업이 예기치 않게 종료되는 경우 MySQL 서버가 파일을 제거합니다.

성능 고려 사항

재실행 로그 아카이빙을 활성화하면 일반적으로 추가 쓰기 작업으로 인해 약간의 성능 비용이 발생합니다.

유닉스 및 유닉스와 유사한 운영 체제에서는 업데이트 속도가 지속적으로 높지 않다고 가정할 때 일반적으로 성능에 미치는 영향은 미미합니다. Windows에서는 일반적으로 동일한 가정 하에 성능에 미치는 영향이 약간 더 높습니다.

업데이트 속도가 지속적으로 빠르고 재실행 로그 아카이브 파일이 재실행 로그 파일과 동일한 저장 매체에 있는 경우, 쓰기 활동이 복합적으로 발생하여 성능에 미치는 영향이 더 클 수 있습니다.

업데이트 속도가 지속적으로 빠르고 재실행 로그 아카이브 파일이 재실행 로그 파일보다 느린 저장 매체에 있는 경우 성능에 임의로 영향을 미칩니다.

재실행 로그 아카이브 파일에 기록해도 재실행 로그 아카이브 파일 저장 매체가 재실행 로그 파일 저장소보다 훨씬 느린 속도로 작동하는 경우를 제외하고는 정상적인 트랜잭션 로깅을 방해하지 않습니다.

미디어에 저장되어 있고 재실행 로그 아카이브 파일에 기록되기를 기다리는 영구 재실행 로그 블록의 백로그가 많은 경우입니다. 이 경우 트랜잭션 로깅 속도는 재실행 로그 아카이브 파일이 있는 느린 저장 미디어에서 관리할 수 있는 수준으로 감소합니다.

재실행 로깅 비활성화

`ALTER INSTANCE DISABLE INNODB REDO_LOG` 문을 사용하여 재실행 로깅을 비활성화할 수 있습니다. 이 기능은 새 MySQL 인스턴스에 데이터를 로드하기 위한 것입니다. 재실행 로깅을 비활성화하면 재실행 로그 쓰기 및 이중 쓰기 버퍼링을 방지하여 데이터 로딩 속도를 높일 수 있습니다.

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.

다시 실행 로깅이 비활성화된 상태에서 예기치 않은 서버 중단 후 서버 재시작을 시도하면 다음 오류가 발생하여 거부됩니다:

```
[오류] [MY-013598] [InnoDB] Innodb 다시 실행 로깅이 비활성화되었을 때 서
버가 종료되었습니다. 데이터 파일이 손상되었을 수 있습니다.
innodb_force_recovery=6으로 데이터베이스를 재시작해 볼 수 있습니다.
```

이 경우 새 MySQL 인스턴스를 초기화하고 데이터 로드 절차를 다시 시작하세요.

재실행 로깅을 활성화 및 비활성화하려면 `INNODB_REDO_LOG_ENABLE` 권한이 필요합니다.

`Innodb_redo_log_enabled` 상태 변수를 사용하면 재실행 로깅 상태를 모니터링할 수 있습니

다.

재실행 로깅이 비활성화되어 있는 동안에는 복제 작업 및 재실행 로그 보관이 허용되지 않으며 그 반대의 경우도 마찬가지입니다.

`ALTER INSTANCE [ENABLE|DISABLE] INNODB REDO_LOG` 작업에는 독점 백업 메타데이터 잠금이 필요하므로 다른 `ALTER INSTANCE` 작업이 동시에 실행되는 것을 방지합니다. 다른 `ALTER INSTANCE` 작업은 실행하기 전에 잠금이 해제될 때까지 기다려야 합니다.

다음 절차는 새 MySQL 인스턴스에 데이터를 로드할 때 재실행 로깅을 비활성화하는 방법을 설명합니다.

1. 새 MySQL 인스턴스에서 재실행 로깅을 비활성화하는 사용자 계정에 `INNODB_REDO_LOG_ENABLE` 권한을 부여합니다.

```
mysql> 'data_load_admin'에 *.*에 INNODB_REDO_LOG_ENABLE ON을 부여합니다;
```

2. `data_load_admin` 사용자로서 다시 실행 로깅을 비활성화합니다:

```
mysql> ALTER INSANCE DISABLE INNODB REDO_LOG;
```

3. `Innodb_redo_log_enabled` 상태 변수를 확인하여 재실행 로깅이 비활성화되어 있는지 확인합니다.

```
mysql> SHOW GLOBAL STATUS LIKE 'Innodb_redo_log_enabled';
+------------------------+ ---------+
| 변수_이름|                 값 |
+------------------------+ ---------+
| Innodb_redo_log_enabled | OFF    |
+------------------------+ ---------+
```

4. 데이터 로드 작업을 실행합니다.

5. 데이터 로드 작업이 완료된 후 `데이터_load_admin` 사용자로서 로깅 다시 실행을 활성화합니다:

취소

```
mysql> ALTER INSANCE ENABLE INNODB REDO_LOG;
```

6. `Innodb_redo_log_enabled` 상태 변수를 확인하여 재실행 로깅이 활성화되어 있는지 확인합니다.

```
mysql> SHOW GLOBAL STATUS LIKE 'Innodb_redo_log_enabled';
+-----------------------+ ---------+
| 변수_이름|                 값 |
+-----------------------+ ---------+
| Innodb_redo_log_enabled | ON     |
+-----------------------+ ---------+
```

관련 주제

- 로그 구성 재실행
- 섹션 8.5.4, "InnoDB 재실행 로깅 최적화"
- 로그 암호화 다시 실행

15.6.6 로그 실행 취소

실행 취소 로그는 단일 읽기-쓰기 트랜잭션과 관련된 실행 취소 로그 레코드의 모음입니다. 실행 취소 로그 레코드에는 클러스터된 인덱스 레코드에 대한 트랜잭션의 최신 변경 사항을 실행 취소하는 방법에 대한 정보가 포함되어 있습니다. 다른 트랜잭션이 일관된 읽기 작업의 일부로 원본 데이터를 확인해야 하는 경우, 수정되지 않은 데이터는 실행 취소 로그 레코드에서 검색됩니다. 실행 취소 로그는 롤백 세그먼트 내에 포함된 실행 취소 로그 세그먼트 내에 존재합니다. 롤백 세그먼트는 실행 취소 테이블 스페이스와 글로벌 임시 테이블 스페이스에 있습니다.

글로벌 임시 테이블 공간에 있는 실행 취소 로그는 사용자 정의 임시 테이블의 데이터를 수정하는 트랜잭션에 사용됩니다. 이러한 실행 취소 로그는 크래시 복구에 필요하지 않으므로 다시 실행 로그를 기록하지 않습니다. 서버가 실행되는 동안 롤백하는 데만 사용됩니다. 이러한 유형의 실행 취소 로그는 재실행 로깅 I/O를 피함으로써 성능에 도움이 됩니다.

로그 실행 취소를 위한 저장 데이터 암호화에 대한 자세한 내용은 로그 암호화 실행 취소를 참조하세요.

각 실행 취소 테이블스페이스와 글로벌 임시 테이블스페이스는 개별적으로 최대 128개의 롤백 세그먼트를 지원합니다. `innodb_rollback_segments` 변수는 롤백 세그먼트의 수를 정의합니다.

롤백 세그먼트가 지원하는 트랜잭션의 수는 롤백 세그먼트의 실행 취소 슬롯 수와 각 트랜잭션에 필요한 실행 취소 로그 수에 따라 달라집니다. 롤백 세그먼트의 실행 취소 슬롯 수는 `InnoDB` 페이지 크기에 따라 다릅니다.

InnoDB 페이지 크기	롤백 세그먼트의 실행 취소 슬롯 수(InnoDB 페이지 크기/16)
4096 (4KB)	256
8192 (8KB)	512

16384 (16KB)	1024
32768 (32KB)	2048
65536 (64KB)	4096

트랜잭션에는 다음 작업 유형별로 하나씩 최대 4개의 실행 취소 로그가 할당됩니다:

1. 사용자 정의 테이블에 대한 `INSERT` 작업
2. 사용자 정의 테이블의 업데이트 및 삭제 작업
3. 사용자 정의 임시 테이블에 대한 `INSERT` 작업
4. 사용자 정의 임시 테이블에 대한 업데이트 및 삭제 작업

취소

실행 취소 로그는 필요에 따라 할당됩니다. 예를 들어, 일반 테이블과 임시 테이블에서 INSERT, UPDATE, DELETE 작업을 수행하는 트랜잭션에는 실행 취소 로그 4개가 완전히 할당되어야 합니다. 일반 테이블에서 INSERT 작업만 수행하는 트랜잭션에는 실행 취소 로그 하나가 필요합니다.

일반 테이블에서 작업을 수행하는 트랜잭션은 할당된 실행 취소 테이블스페이스 롤백 세그먼트에서 실행 취소 로그를 할당받습니다. 임시 테이블에서 작업을 수행하는 트랜잭션에는 할당된 글로벌 임시 테이블 스페이스 롤백 세그먼트에서 실행 취소 로그가 할당됩니다.

트랜잭션에 할당된 실행 취소 로그는 트랜잭션이 실행되는 동안 트랜잭션에 첨부된 상태로 유지됩니다. 예를 들어, 일반 테이블에 대한 INSERT 작업에 대해 트랜잭션에 할당된 실행 취소 로그는 해당 트랜잭션이 수행하는 일반 테이블에 대한 모든 INSERT 작업에 사용됩니다.

위에서 설명한 요소를 고려할 때 다음 공식을 사용하여 InnoDB가 지원할 수 있는 동시 읽기-쓰기 트랜잭션의 수를 추정할 수 있습니다.

참고

InnoDB가 지원할 수 있는 동시 읽기-쓰기 트랜잭션 수에 도달하기 전에 동시 트랜잭션 제한 오류가 발생할 수 있습니다. 이 오류는 트랜잭션에 할당된 롤백 세그먼트의 실행 취소 슬롯이 부족할 때 발생합니다. 이러한 경우 트랜잭션을 다시 실행해 보세요.

트랜잭션이 임시 테이블에서 작업을 수행할 때 InnoDB가 지원할 수 있는 동시 읽기-쓰기 트랜잭션의 수입니다.
는 전역에 할당된 롤백 세그먼트 수에 의해 제약을 받습니다.
임시 테이블 스페이스(기본값은 128)입니다.

- 각 트랜잭션이 INSERT **또는** UPDATE 또는 DELETE 작업을 수행하는 경우, InnoDB가 지원할 수 있는 동시 읽기-쓰기 트랜잭션의 수는 다음과 같습니다:

```
(innodb_page_size / 16) * innodb_rollback_segments * 실행 취소 테이블 공간 수
```

- 각 트랜잭션이 **INSERT와** UPDATE 또는 DELETE 작업을 수행하는 경우, InnoDB가 지원할 수 있는 동시 읽기-쓰기 트랜잭션의 수는 다음과 같습니다:

```
(innodb_page_size / 16 / 2) * innodb_rollback_segments * 실행 취소 테이블 공간 수
```

- 각 트랜잭션이 임시 테이블에서 INSERT 작업을 수행하는 경우, InnoDB가 지원할 수 있는 동시 읽기-쓰기 트랜잭션의 수는 다음과 같습니다:

```
(innodb_page_size / 16) * innodb_rollback_segments
```

- 각 트랜잭션이 임시 테이블에 대해 **INSERT와** UPDATE 또는 DELETE 작업을 수행하는 경우, InnoDB가 지원할 수 있는 동시 읽기-쓰기 트랜잭션의 수는 다음과 같습니다:

```
(innodb_page_size / 16 / 2) * innodb_rollback_segments
```

15.7 InnoDB 잠금 및 트랜잭션 모델

대규모, 사용량이 많거나 안정성이 높은 데이터베이스 애플리케이션을 구현하거나, 다른 데이터베이스 시스템에서 상당한 양의 코드를 포팅하거나, MySQL 성능을 조정하려면 `InnoDB` 잠금 및 `InnoDB` 트랜잭션 모델을 이해하는 것이 중요합니다.

이 섹션에서는 익숙해야 하는 `InnoDB` 잠금 및 `InnoDB` 트랜잭션 모델과 관련된 몇 가지 주제에 대해 설명합니다.

- 섹션 15.7.1, 'InnoDB 잠금'에서는 `InnoDB`에서 사용하는 잠금 유형에 대해 설명합니다.
- 섹션 15.7.2, "InnoDB 트랜잭션 모델"에서는 트랜잭션 격리 수준과 각 수준에서 사용되는 잠금 전략에 대해 설명합니다. 또한 자동 커밋, 일관된 비잠금 읽기 및 잠금 읽기의 사용에 대해서도 설명합니다.

- 15.7.3절, 'InnoDB의 다양한 SQL 문에 의해 설정된 잠금'에서는 다양한 문에 대해 InnoDB에 설정된 특정 유형의 잠금에 대해 설명합니다.
- 15.7.4절, '팬텀 행'에서는 InnoDB가 팬텀 행을 피하기 위해 다음 키 잠금을 사용하는 방법에 대해 설명합니다.
- 15.7.5절, 'InnoDB의 교착 상태'에서는 교착 상태 예제를 제공하고, 교착 상태 감지에 대해 설명하며, InnoDB에서 교착 상태를 최소화하고 처리하기 위한 팁을 제공합니다.

15.7.1 InnoDB 잠금

이 섹션에서는 InnoDB에서 사용하는 잠금 유형에 대해 설명합니다.

- 공유 및 독점 잠금
- 인텐션 잠금
- 레코드 잠금
- 갭 잠금
- 다음 키 잠금
- 인텐션 잠금 삽입
- 자동-INC 잠금
- 공간 인덱스에 대한 술어 잠금

공유 및 독점 잠금

InnoDB는 공유(S) 잠금과 독점(X) 잠금의 두 가지 유형이 있는 표준 행 수준 잠금을 구현합니다.

- 공유(S) 잠금은 잠금을 보유한 트랜잭션이 행을 읽을 수 있도록 허용합니다.
- 독점(X) 잠금은 잠금을 보유한 트랜잭션이 행을 업데이트하거나 삭제할 수 있도록 허용합니다.

트랜잭션 T1이 행 r에 대한 공유(S) 잠금을 보유하고 있는 경우, 행 r에 대한 잠금을 요청하는 일부 개별 트랜잭션 T2의 요청은 다음과 같이 처리됩니다:

- T2의 S 잠금 요청은 즉시 승인될 수 있습니다. 결과적으로 T1과 T2는 모두 r에 대한 S 잠금을 보유하게 됩니다.
- T2의 X 잠금 요청은 즉시 승인될 수 없습니다.

트랜잭션 T1이 행 r에 대해 배타적(X) 잠금을 보유하고 있는 경우, 다른 트랜잭션 T2가 행 r에 대해 두 가지 유형의 잠금을 요청해도 즉시 승인할 수 없습니다. 대신 트랜잭션 T2는 트랜잭션 T1이 행 r에 대한 잠금을

해제할 때까지 기다려야 합니다.

인텐션 잠금

InnoDB는 행 잠금과 테이블 잠금이 공존할 수 있는 *다중 세분성 잠금을* 지원합니다. 예를 들어, LOCK TABLES ... WRITE 문은 지정된 테이블에 배타적 잠금(X 잠금)을 취합니다. 여러 세부 수준에서 잠그는 것을 실용적으로 만들기 위해 InnoDB는 인텐션 잠금을 사용합니다. 인텐션 잠금은 테이블 수준 잠금으로, 트랜잭션이 나중에 테이블의 행에 대해 어떤 유형의 잠금(공유 또는 독점)을 필요로 하는지 나타내는 테이블 수준 잠금입니다. 인텐션 잠금에는 두 가지 유형이 있습니다:

- 의도 공유 잠금(IS)은 트랜잭션이 테이블의 개별 행에 *공유* 잠금을 설정하려는 의도가 있음을 나타냅니다.
- 의도 독점 잠금(IX)은 트랜잭션이 테이블의 개별 행에 독점 잠금을 설정하려는 의도가 있음을 나타냅니다.

예를 들어 SELECT ... FOR SHARE는 IS 잠금을 설정하고 SELECT ... FOR UPDATE는 IX 잠금을 설정합니다.
잠급니다.

인텐션 잠금 프로토콜은 다음과 같습니다:

- 트랜잭션이 테이블의 행에 대한 공유 잠금을 획득하려면 먼저 테이블에서 IS 잠금 또는 그 이상의 강력한 잠금을 획득해야 합니다.
- 트랜잭션이 테이블의 행에 대한 독점 잠금을 획득하려면 먼저 테이블에서 IX 잠금을 획득해야 합니다.

테이블 수준 잠금 유형 호환성은 다음 매트릭스에 요약되어 있습니다.

	X	IX	S	IS
X	충돌	충돌	충돌	충돌
IX	충돌	호환 가능	충돌	호환 가능
S	충돌	충돌	호환 가능	호환 가능
IS	충돌	호환 가능	호환 가능	호환 가능

요청하는 트랜잭션이 기존 잠금과 호환되는 경우 잠금이 부여되지만, 기존 잠금과 충돌하는 경우 잠금이 부여되지 않습니다. 트랜잭션은 충돌하는 기존 잠금이 해제될 때까지 대기합니다. 잠금 요청이 기존 잠금과 충돌하여 교착 상태를 유발하기 때문에 잠금을 부여할 수 없는 경우 오류가 발생합니다.

의도 잠금은 전체 테이블 요청(예: 테이블 잠금 ... 쓰기)을 제외하고는 아무것도 차단하지 않습니다. 의도 잠금의 주요 목적은 누군가 행을 잠그고 있거나 테이블에서 행을 잠그려고 한다는 것을 표시하는 것입니다.

인텐션 잠금에 대한 트랜잭션 데이터는 엔진 INNODB 상태 표시 및 InnoDB 모니터 출력에 다음과 유사하게 표시됩니다:

```
TABLE LOCK 테이블 `test`.`t` trx id 10080 잠금 모드 IX
```

레코드 잠금

레코드 잠금은 인덱스 레코드에 대한 잠금입니다. 예를 들어 SELECT c1 FROM t WHERE c1 = 10 FOR UPDATE; 는 다른 트랜잭션이 t.c1의 값이 10인 행을 삽입, 업데이트 또는 삭제하지 못하도록 합니다.

레코드 잠금은 테이블이 인덱스 없이 정의된 경우에도 항상 인덱스 레코드를 잠급니다. 이러한 경우 InnoDB는 숨겨진 클러스터된 인덱스를 생성하고 이 인덱스를 레코드 잠금에 사용합니다. 섹션 15.6.2.1, "클러스터 및 보조 인덱스"를 참조하십시오.

레코드 잠금에 대한 트랜잭션 데이터는 엔진 INNODB 상태 표시에서 다음과 유사하게 나타납니다.
및 InnoDB 모니터 출력:

```
레코드 록 스페이스 id 58 페이지 번호 3 n 비트 72 테이블 `test`.`t`의 `PRIMARY` 인덱스 10078
lock_mode X 레코드를 잠그지만 갭은 잠그지 않습니다.
레코드 잠금, 힙 번호 2 물리적 레코드: n_fields 3; 컴팩트 포맷; 정보 비트 0 0: 렌 4; 헥스
 8000000A; asc                    ;;
 1: LEN 6; HEX 00000000274F; ASC      'O;;
 2: LEN 7; HEX B60000019D0110; ASC         ;;
```

갭 잠금

간격 잠금은 인덱스 레코드 사이의 간격에 대한 잠금 또는 첫 번째 또는 마지막 인덱스 레코드 앞뒤의 간격에 대한 잠금입니다. 예를 들어, `SELECT c1 FROM t WHERE c1 BETWEEN 10과 20 FOR UPDATE;` 는 다른 트랜잭션이 `t.c1` 열에 `15라는` 값을 삽입하는 것을 방지합니다.
범위의 모든 기존 값 사이의 간격이 잠겨 있기 때문에 열에 이미 해당 값이 있습니다.

갭은 단일 인덱스 값, 여러 인덱스 값에 걸쳐 있을 수도 있고 비어 있을 수도 있습니다.

갭 잠금은 성능과 동시성 사이의 절충점 중 하나로, 일부 트랜잭션 격리 수준에서는 사용되지만 다른 트랜잭션 격리 수준에서는 사용되지 않습니다.

고유 인덱스를 사용하여 고유 행을 검색하기 위해 행을 잠그는 문에는 갭 잠금이 필요하지 않습니다. (검색 조건에 다중 열 고유 인덱스의 일부 열만 포함되는 경우는 포함되지 않습니다. 이 경우 갭 잠금이 발생합니다.) 예를 들어 `id` 열에 고유 인덱스가 있는 경우 다음 문은 `id` 값이 100인 행에 대해서만 인덱스-레코드 잠금을 사용하며 다른 세션이 앞의 간격에 행을 삽입하는지 여부는 중요하지 않습니다:

```
SELECT * FROM child WHERE id = 100;
```

id가 인덱싱되지 않았거나 고유하지 않은 인덱스가 있는 경우, 이 문은 앞의 갭을 잠급니다.

또한 서로 다른 트랜잭션이 갭에 상충하는 잠금을 보유할 수 있다는 점도 주목할 필요가 있습니다. 예를 들어, 트랜잭션 A는 하나의 갭에 공유 갭 잠금(갭 S-락)을, 트랜잭션 B는 동일한 갭에 배타적 갭 잠금(갭 X-락)을 보유할 수 있습니다. 충돌하는 갭 잠금이 허용되는 이유는 인덱스에서 레코드가 제거될 경우, 서로 다른 트랜잭션이 해당 레코드에 보유한 갭 잠금을 병합해야 하기 때문입니다.

`InnoDB`의 갭 잠금은 "순수 억제형"으로, 다른 트랜잭션이 갭에 삽입되는 것을 방지하는 것이 유일한 목적입니다. 갭 잠금은 공존할 수 있습니다. 한 트랜잭션이 갭 잠금을 설정해도 다른 트랜잭션이 같은 갭에 갭 잠금을 설정하는 것을 막지는 못합니다. 공유 갭 잠금과 배타적 갭 잠금 사이에는 차이가 없습니다. 서로 충돌하지 않으며 동일한 기능을 수행합니다.

갭 잠금은 명시적으로 비활성화할 수 있습니다. 트랜잭션 격리 수준을 읽기 커밋으로 변경하면 이런 일이 발생합니다. 이 경우 갭 잠금은 검색 및 인덱스 스캔에 대해 비활성화되며 외래 키 제약 조건 검사 및 중복 키 검사에만 사용됩니다.

`READ COMMITTED` 격리 수준을 사용하면 다른 효과도 있습니다. 일치하지 않는 행에 대한 레코드 잠금은 MySQL이 `WHERE` 조건을 평가한 후에 해제됩니다. `UPDATE` 문의 경우 `InnoDB`는 "준정합성" 읽기를 수행하여 최신 커밋된 버전을 MySQL에 반환하므로 MySQL이 행이 `UPDATE`의 `WHERE` 조건과 일치하는지 여부를 확인할 수 있습니다.

다음 키 잠금

다음 키 잠금은 인덱스 레코드에 대한 레코드 잠금과 인덱스 레코드 앞의 갭에 대한 갭 잠금의 조합입니다.

InnoDB는 테이블 인덱스를 검색하거나 스캔할 때 발견되는 인덱스 레코드에 공유 또는 독점 잠금을 설정하는 방식으로 행 수준 잠금을 수행합니다. 따라서 행 수준 잠금은 실제로는 인덱스 레코드 잠금입니다. 인덱스 레코드에 대한 다음 키 잠금은 해당 인덱스 레코드 앞의 '갭'에도 영향을 줍니다. 즉, 다음 키 잠금은 인덱스 레코드 잠금에 인덱스 레코드 앞의 갭에 대한 갭 잠금을 더한 것입니다. 한 세션이 인덱스의 레코드 `R`에 대해 공유 또는 독점 잠금을 가지고 있는 경우, 다른 세션은 인덱스 순서에서 `R` 바로 앞의 갭에 새 인덱스 레코드를 삽입할 수 없습니다.

인덱스에 값 10, 11, 13, 20이 포함되어 있다고 가정합니다. 이 인덱스에 대해 가능한 다음 키 잠금은 다음 간격을 포함하며, 여기서 둥근 대괄호는 간격 끝점을 제외하고 대괄호는 끝점을 포함함을 나타냅니다:

```
(음의 무한대, 10]
(10, 11]
(11, 13]
(13, 20]
(20, 양의 무한대)
```

마지막 간격 동안, 다음 키 잠금은 인덱스에서 가장 큰 값과 실제 인덱스의 어떤 값보다 높은 값을 갖는 "최상위" 의사 레코드 위의 간격을 잠급니다. 최상위 레코드는 실제 인덱스 레코드가 아니므로, 사실상 이 다음 키 잠금은 가장 큰 인덱스 값 다음의 간격만 잠급니다.

기본적으로 InnoDB는 반복 읽기 트랜잭션 격리 수준에서 작동합니다. 이 경우 InnoDB는 검색 및 인덱스 스캔에 다음 키 잠금을 사용하여 팬텀 행을 방지합니다(섹션 15.7.4, "팬텀 행" 참조).

다음 키 잠금에 대한 트랜잭션 데이터는 엔진 INNODB 상태 표시 및 InnoDB 모니터 출력에 다음과 유사하게 나타납니다:

```
레코드 록 공간 id 58 페이지 번호 3 n 비트 72 테이블 `test`.`t`의 `PRIMARY` 인덱스 10080
lock_mode X
레코드 잠금, 힙 번호 1 물리적 레코드: n_fields 1; 컴팩트 포맷; 정보 비트 0 0: 렌 8; 헥스
 73757072656d756d; asc supremum;;

레코드 잠금, 힙 번호 2 물리적 레코드: n_fields 3; 컴팩트 포맷; 정보 비트 0 0: 렌 4; 헥스
 8000000A; asc                    ;;
 1: LEN 6; HEX 00000000274F; ASC     'O;;
 2: LEN 7; HEX B60000019D0110; ASC         ;;
```

인텐션 잠금 삽입

삽입 의도 잠금은 행 삽입 전에 INSERT 연산에 의해 설정되는 갭 잠금의 한 유형입니다. 이 잠금은 동일한 인덱스 갭에 삽입하는 여러 트랜잭션이 갭 내의 동일한 위치에 삽입하지 않는 경우 서로를 기다릴 필요가 없도록 삽입 의도를 알립니다. 값이 4와 7인 인덱스 레코드가 있다고 가정해 보겠습니다. 각각 5와 6의 값을 삽입하려는 별도의 트랜잭션은 삽입된 행에 대한 독점 잠금을 얻기 전에 각각 삽입 의도 잠금으로 4와 7 사이의 간격을 잠그지만, 행이 충돌하지 않으므로 서로를 차단하지 않습니다.

다음 예시는 삽입된 레코드에 대한 독점 잠금을 얻기 전에 삽입 의도 잠금을 취하는 트랜잭션을 보여줍니다. 이 예시에는 두 클라이언트, 즉 A와 B가 포함됩니다.

클라이언트 A는 두 개의 인덱스 레코드(90과 102)가 포함된 테이블을 생성한 다음 ID가 100보다 큰 인덱스 레코드에 독점 잠금을 설정하는 트랜잭션을 시작합니다. 이 배타적 잠금에는 102번 레코드 앞에 갭 잠금이 포함됩니다:

```
mysql> CREATE TABLE child (id int(11) NOT NULL, PRIMARY KEY(id)) ENGINE=InnoDB;
mysql> INSERT INTO child (id) values (90),(102);

mysql> 트랜잭션 시작;
mysql> SELECT * FROM child WHERE id > 100 FOR UPDATE;
+-----+
| 아이디 |
+-----+
| 102 |
+-----+
```

클라이언트 B는 갭에 레코드를 삽입하는 트랜잭션을 시작합니다. 트랜잭션은 독점 잠금을 얻기 위해 기다리는 동안 삽입 의도 잠금을 취합니다.

```
mysql> 트랜잭션 시작;
mysql> INSERT INTO child (id) VALUES (101);
```

삽입 의도 잠금에 대한 트랜잭션 데이터는 엔진 INNODB 상태 표시 및 InnoDB 모니터 출력에 다음과 유사하게 나타납니다:

```
레코드 록 스페이스 id 31 페이지 번호 3 n 비트 72 인덱스 `test`.`child` 테이블의 `PRIMARY` trx
id 8731 lock_mode X 레코드 삽입 의도 대기 전 간격 잠금
레코드 잠금, 힙 번호 3 물리적 레코드: n_fields 3; 컴팩트 포맷; 정보 비트 0 0: 렌 4; 헥스
 8000000a; asc     ;;
 0: LEN 6; HEX f9000000172011C; ASC      r ;;...
 1: LEN 6; HEX 000000002215; ASC
```

자동-INC 잠금

AUTO-INC 잠금은 AUTO_INCREMENT 열이 있는 테이블에 삽입하는 트랜잭션이 취하는 특수한 테이블 수준 잠금입니다. 가장 간단한 경우, 한 트랜잭션이 테이블에 값을 삽입하는 경우 다른 트랜잭션은 해당 테이블에 자체 삽입을 수행하기 위해 기다려야 하므로 첫 번째 트랜잭션에 의해 삽입된 행은 연속적인 기본 키 값을 받게 됩니다.

innodb_autoinc_lock_mode 변수는 자동 증가 잠금에 사용되는 알고리즘을 제어합니다. 이 변수를 사용하면 예측 가능한 자동 증가 값의 시퀀스와 삽입 작업의 최대 동시성 사이에서 절충하는 방법을 선택할 수 있습니다.

자세한 내용은 15.6.1.6절, 'InnoDB에서 AUTO_INCREMENT 처리'를 참조하세요.

공간 인덱스에 대한 술어 잠금

InnoDB는 공간 데이터가 포함된 열의 공간 인덱싱을 지원합니다(11.4.9절, '공간 분석 최적화' 참조).

공간 인덱스와 관련된 연산에 대한 잠금을 처리하기 위해 다음 키 잠금은 반복 읽기 또는 직렬화 가능 트랜잭션 격리 수준을 지원하기 위해 잘 작동하지 않습니다. 다차원 데이터에는 절대적인 순서 개념이 없으므로 어떤 키가 "다음" 키인지 명확하지 않습니다.

SPATIAL 인덱스가 있는 테이블에 대한 격리 수준을 지원하기 위해 InnoDB는 술어 잠금을 사용합니다. SPATIAL 인덱스에는 최소 바운딩 사각형(MBR) 값이 포함되어 있으므로 InnoDB는 쿼리에 사용되는 MBR 값에 술어 잠금을 설정하여 인덱스에서 일관된 읽기를 적용합니다. 다른 트랜잭션은 쿼리 조건과 일치하는 행을 삽입하거나 수정할 수 없습니다.

15.7.2 InnoDB 트랜잭션 모델

InnoDB 트랜잭션 모델은 다중 버전 데이터베이스의 장점과 기존의 2단계 잠금을 결합하는 것을 목표로 합니다. InnoDB는 행 수준에서 잠금을 수행하고 쿼리를 기본적으로 Oracle 스타일에 따라 비잠금 일관성 읽기로 실행합니다. InnoDB의 잠금 정보는 공간 효율적으로 저장되므로 잠금 에스컬레이션이 필요하지 않습니다. 일반적으로 여러 사용자가 InnoDB 테이블의 모든 행 또는 행의 임의의 하위 집합을 잠글 수 있으며, 이 경우 InnoDB 메모리가 고갈되지 않습니다.

15.7.2.1 트랜잭션 격리 수준

트랜잭션 격리는 데이터베이스 처리의 기초 중 하나입니다. 격리란 약어 ACID의 I를 의미하며, 격리 수준은 여러 트랜잭션이 동시에 변경을 수행하고 쿼리를 수행할 때 성능과 안정성, 일관성 및 결과 재현성 간의 균형을 미세 조정하는 설정입니다.

InnoDB는 SQL:1992 표준에서 설명하는 네 가지 트랜잭션 격리 수준을 모두 제공합니다: 읽기 미커밋, 읽기 커밋, 반복 읽기 및 직렬화 가능입니다. 기본 격리 수준은 반복 읽기입니다.

사용자는 SET TRANSACTION 문을 사용하여 단일 세션 또는 모든 후속 연결에 대한 격리 수준을 변경할 수 있습니다. 모든 연결에 대한 서버의 기본 격리 수준을 설정하려면 명령줄 또는 옵션 파일에서 --transaction-isolation 옵션을 사용합니다. 격리 수준 및 수준 설정 구문에 대한 자세한 내용은 섹션 13.3.7, "SET 트랜잭션 문"을 참조하십시오.

InnoDB는 다양한 잠금 전략을 사용하여 여기에 설명된 각 트랜잭션 격리 수준을 지원합니다. ACID 규정 준수가 중요한 중요한 데이터에 대한 작업의 경우 기본 REPEATABLE READ 수준을 사용하여 높은 수준의 일관

성을 적용할 수 있습니다. 또는 대량 보고와 같이 정확한 일관성과 반복 가능한 결과가 잠금을 위한 오버헤드를 최소화하는 것보다 덜 중요한 상황에서는 `READ COMMITTED` 또는 `READ UNCOMMITTED`로 일관성 규칙을 완화할 수 있습니다. `SERIALIZABLE`은 반복 읽기보다 더 엄격한 규칙을 적용하며, 주로 XA 트랜잭션과 같은 특수한 상황에서 동시성 및 교착 상태와 관련된 문제를 해결하는 데 사용됩니다.

다음 목록은 MySQL이 다양한 트랜잭션 수준을 지원하는 방법을 설명합니다. 이 목록은 가장 일반적으로 사용되는 수준부터 가장 적게 사용되는 수준까지 나열되어 있습니다.

- 반복 읽기

  이것은 `InnoDB`의 기본 격리 수준입니다. 동일한 트랜잭션 내에서 일관된 읽기는 첫 번째 읽기에 의해 설정된 스냅샷을 읽습니다. 즉, 일반(비잠금) 읽기를 여러 번 실행하는 경우
  동일한 트랜잭션 내에서 이러한 `SELECT` 문은 서로에 대해서도 일관성을 유지합니다. 섹션 15.7.2.3, "일관된 비잠금 읽기"를 참조하십시오.

읽기 잠금(SELECT(FOR UPDATE 또는 FOR SHARE 포함), UPDATE 및 DELETE 문)의 경우 잠금은 문이 고유한 검색 조건이 있는 고유 인덱스를 사용하는지 또는 범위 유형 검색 조건을 사용하는지에 따라 달라집니다.

- 고유한 검색 조건이 있는 고유 인덱스의 경우, InnoDB는 발견된 인덱스 레코드만 잠그고 그 앞의 간격은 잠그지 않습니다.
- 다른 검색 조건의 경우, InnoDB는 갭 잠금 또는 다음 키 잠금을 사용하여 스캔된 인덱스 범위를 잠그고 해당 범위가 포함하는 갭에 다른 세션이 삽입하는 것을 차단합니다. 갭 잠금 및 다음 키 잠금에 대한 자세한 내용은 섹션 15.7.1, "InnoDB 잠금"을 참조하십시오.

- 읽기 완료

  동일한 트랜잭션 내에서도 각 일관된 읽기는 고유한 새 스냅샷을 설정하고 읽습니다. 일관된 읽기에 대한 자세한 내용은 섹션 15.7.2.3, "일관된 비잠금 읽기"를 참조하십시오.

  읽기 잠금(SELECT(FOR UPDATE 또는 FOR SHARE 포함), UPDATE 문 및 DELETE 문), 업데이트 문 및 삭제 문의 경우 InnoDB는 인덱스 레코드만 잠그고 그 앞의 갭은 잠그지 않으므로 잠긴 레코드 옆에 새 레코드를 자유롭게 삽입할 수 있습니다. 갭 잠금은 외래 키 제약 조건 검사 및 중복 키 검사에만 사용됩니다.

  간격 잠금이 비활성화되어 있으므로 다른 세션에서 새 행을 간격에 삽입할 수 있으므로 팬텀 행 문제가 발생할 수 있습니다. 팬텀 행에 대한 자세한 내용은 섹션 15.7.4, "팬텀 행"을 참조하십시오.

  읽기 커밋 격리 수준에서는 행 기반 바이너리 로깅만 지원됩니다. 사용하는 경우 binlog_format=MIXED와 함께 READ COMMITTED를 사용하면 서버가 자동으로 행 기반 로깅을 사용합니다. READ COMMITTED를 사용하면 추가적인 효과가 있습니다:

  - 업데이트 또는 삭제 문의 경우 InnoDB는 업데이트 또는 삭제하는 행에 대해서만 잠금을 유지합니다. 일치하지 않는 행에 대한 레코드 잠금은 MySQL이 WHERE 조건을 평가한 후에 해제됩니다. 이렇게 하면 교착 상태가 발생할 확률이 크게 줄어들지만 여전히 교착 상태가 발생할 수 있습니다.
  - UPDATE 문의 경우, 행이 이미 잠겨 있는 경우 InnoDB는 '준정합성' 읽기를 수행하여 최신 커밋된 버전을 MySQL에 반환함으로써 MySQL이 행이 UPDATE의 WHERE 조건과 일치하는지 여부를 확인할 수 있도록 합니다. 행이 일치하는 경우(업데이트해야 함) MySQL은 해당 행을 다시 읽고 이번에는 InnoDB가 해당 행을 잠그거나 잠금을 기다립니다.

이 표부터 시작하여 다음 예제를 살펴보겠습니다:

```
CREATE TABLE t (a INT NOT NULL, b INT) ENGINE = InnoDB;
INSERT INTO t VALUES (1,2,(2,3,(3,2,(4,3,(5,2)); COMMIT;
```

이 경우 테이블에 인덱스가 없으므로 검색 및 인덱스 스캔은 인덱싱된 열이 아닌 숨겨진 클러스터된 인덱스를 사용하여 레코드 잠금을 수행합니다(섹션 15.6.2.1, "클러스터 및 보조 인덱스" 참조).

한 세션이 이러한 문을 사용하여 업데이트를 수행한다고 가정해 보겠습니다:

```
# 세션 A
거래를 시작합니다;
UPDATE t SET b = 5 WHERE b = 3;
```

또한 두 번째 세션이 첫 번째 세션에 이어 이러한 문을 실행하여 업데이트를 수행한다고 가정해 보겠습니다:

```
# 세션 B
UPDATE t SET b = 4 WHERE b = 2;
```

InnoDB는 각 UPDATE를 실행할 때 먼저 각 행에 대한 독점 잠금을 획득한 다음 행을 수정할지 여부를 결정합니다. InnoDB가 행을 수정하지 않으면 잠금을 해제합니다. 그렇지 않으면,

InnoDB는 트랜잭션이 끝날 때까지 잠금을 유지합니다. 이는 다음과 같이 트랜잭션 처리에 영향을 줍니다.

기본 REPEATABLE READ 격리 수준을 사용하는 경우 첫 번째 업데이트는 읽는 각 행에 대해 X-락을 획득하고 그 중 어떤 것도 해제하지 않습니다:

```
x-lock(1,2); retain x-lock
X-LOCK(2,3); UPDATE(2,3) TO (2,5); RETAIN X-LOCK
X-LOCK(3,2); RETAIN X-LOCK
X-LOCK(4,3); UPDATE(4,3) TO (4,5); RETAIN X-LOCK
X-LOCK(5,2); RETAIN X-LOCK
```

두 번째 업데이트는 잠금을 획득하려고 시도하는 즉시 차단되며(첫 번째 업데이트가 모든 행에 잠금을 유지했기 때문에) 첫 번째 업데이트가 커밋되거나 롤백될 때까지 진행되지 않습니다:

```
x-lock(1,2); 차단하고 첫 번째 업데이트가 커밋 또는 롤백될 때까지 기다립니다.
```

대신 READ COMMITTED를 사용하는 경우 첫 번째 UPDATE는 읽는 각 행에 대해 x 잠금을 획득하고 수정하지 않는 행에 대해서는 잠금을 해제합니다:

```
x-lock(1,2); unlock(1,2)
x-lock(2,3); update(2,3) to (2,5); retain x-lock
x-lock(3,2); unlock(3,2)
x-lock(4,3); update(4,3) to (4,5); retain x-lock
x-lock(5,2); unlock(5,2)
```

두 번째 업데이트의 경우 InnoDB는 "반정합성" 읽기를 수행하여 MySQL에서 읽는 각 행의 최신 커밋된 버전을 반환하므로 MySQL은 해당 행이 업데이트의 WHERE 조건과 일치하는지 여부를 확인할 수 있습니다:

```
x-lock(1,2); update(1,2) to (1,4); retain x-lock
x-lock(2,3); unlock(2,3)
x-lock(3,2); update(3,2) to (3,4); retain x-lock
x-lock(4,3); unlock(4,3)
x-lock(5,2); update(5,2) to (5,4); retain x-lock
```

그러나 WHERE 조건에 인덱싱된 열이 포함되어 있고 InnoDB가 인덱스를 사용하는 경우 레코드 잠금을 취하고 유지할 때 인덱싱된 열만 고려됩니다. 다음 예제에서 첫 번째 UPDATE는 b = 2인 각 행에 대해 x 잠금을 취하고 유지합니다. 두 번째 업데이트는 b 열에 정의된 인덱스도 사용하므로 동일한 레코드에 대한 x 잠금을 획득하려고 시도하면 차단됩니다.

```
CREATE TABLE t (a INT NOT NULL, b INT, c INT, INDEX (b)) ENGINE = InnoDB;
INSERT INTO t VALUES (1,2,3),(2,2,4);
커밋;

# 세션 A
거래를 시작합니다;
UPDATE t SET b = 3 WHERE b = 2 AND c = 3;

# 세션 B
UPDATE t SET b = 4 WHERE b = 2 AND c = 4;
```

읽기 커밋 격리 수준은 시작 시 설정하거나 런타임에 변경할 수 있습니다. 런타임에 모든 세션에 대해 전역적으로 설정하거나 세션별로 개별적으로 설정할 수 있습니다.

- 커밋되지 않은 읽기

SELECT 문은 비잠금 방식으로 수행되지만 가능한 행의 이전 버전이 사용될 수 있습니다. 따라서 이 격리

수준을 사용하면 이러한 읽기가 일관되지 않습니다. 이를 더티 읽기라고도 합니다. 그렇지 않으면 이 격리 수준은 `READ COMMITTED`처럼 작동합니다.

- 직렬화 가능

  이 수준은 반복 가능한 읽기와 비슷하지만, 자동 커밋이 비활성화되어 있는 경우 `InnoDB`는 모든 일반 `SELECT` 문을 `SELECT ... FOR SHARE`로 암시적으로 변환합니다. 자동 커밋이 활성화된 경우 `SELECT`는 자체 트랜잭션입니다. 따라서 읽기 전용으로 알려져 있으며 다음과 같이 수행하면 직렬화할 수 있습니다.

일관된(잠금 해제되지 않은) 읽기이며 다른 트랜잭션에 대해 차단할 필요가 없습니다. (다른 트랜잭션이 선택한 행을 수정한 경우 일반 SELECT를 강제로 차단하려면 자동 커밋을 비활성화합니다.)

조인 목록 또는 하위 쿼리를 통해 MySQL 부여 테이블에서 데이터를 읽지만 수정하지 않는 DML 작업은 격리 수준과 관계없이 MySQL 부여 테이블에 대한 읽기 잠금을 획득하지 않습니다. 자세한 내용은 부여 테이블 동시성을 참조하십시오.

15.7.2.2 자동 커밋, 커밋 및 롤백

InnoDB에서는 모든 사용자 활동이 트랜잭션 내에서 발생합니다. 자동 커밋 모드가 활성화된 경우 각 SQL 문은 자체적으로 단일 트랜잭션을 형성합니다. 기본적으로 MySQL은 자동 커밋이 활성화된 상태에서 각 새 연결에 대해 세션을 시작하므로, 해당 문이 오류를 반환하지 않는 경우 각 SQL 문 후에 커밋을 수행합니다. 문이 오류를 반환하는 경우 커밋 또는 롤백 동작은 오류에 따라 달라집니다. 섹션 15.21.5, "InnoDB 오류 처리"를 참조하세요.

자동 커밋이 활성화된 세션은 명시적인 START TRANSACTION 또는 BEGIN 문으로 트랜잭션을 시작하고 COMMIT 또는 ROLLBACK 문으로 트랜잭션을 종료하여 여러 문으로 구성된 트랜잭션을 수행할 수 있습니다. 13.3.1절, "START 트랜잭션, COMMIT 및 ROLLBACK 문"을 참조하세요.

세션 내에서 자동 커밋 모드가 자동 커밋 = 0으로 비활성화되어 있으면 세션에 항상 트랜잭션이 열려 있습니다. 커밋 또는 롤백 문을 사용하면 현재 트랜잭션이 종료되고 새 트랜잭션이 시작됩니다.

자동 커밋이 비활성화된 세션이 최종 트랜잭션을 명시적으로 커밋하지 않고 종료되면 MySQL은 해당 트랜잭션을 롤백합니다.

일부 문은 마치 문을 실행하기 전에 커밋을 수행한 것처럼 트랜잭션을 암시적으로 종료합니다. 자세한 내용은 섹션 13.3.3, "암시적 커밋을 유발하는 문"을 참조하세요.

커밋은 현재 트랜잭션에서 변경한 내용이 영구적으로 적용되어 다른 세션에 표시된다는 의미입니다. 반면에 ROLLBACK 문은 현재 트랜잭션에서 수행한 모든 수정 사항을 취소합니다. 커밋과 롤백 모두 현재 트랜잭션 중에 설정된 모든 InnoDB 잠금을 해제합니다.

트랜잭션으로 DML 작업 그룹화

기본적으로 MySQL 서버에 대한 연결은 자동 커밋 모드가 활성화된 상태로 시작되며, 모든 SQL 문을 실행할 때 자동으로 커밋합니다. 일련의 DML 문을 실행한 후 커밋하거나 모두 함께 롤백하는 것이 표준 관행인 다른 데이터베이스 시스템을 사용해 본 경험이 있다면 이 작동 모드가 익숙하지 않을 수 있습니다.

여러 문으로 구성된 트랜잭션을 사용하려면 SET autocommit = 0 SQL 문으로 자동 커밋을 끄고 각 트랜잭션을 적절히 COMMIT 또는 ROLLBACK으로 종료합니다. 자동 커밋을 켜두려면 각 트랜잭션을 START TRANSACTION으로 시작하고 COMMIT 또는 ROLLBACK으로 종료합니다. 다음 예는 두 개의 트랜잭션을 보여줍니다. 첫 번째 트랜잭션은 커밋되고 두 번째 트랜잭션은 롤백됩니다.

```
mysql test
```

```
mysql> CREATE TABLE customer (a INT, b CHAR (20), INDEX (a));
쿼리 확인, 영향을 받은 행 0개(0.00초)
mysql> -- 자동 커밋을 켠 상태에서 트랜잭션을 수행합니다.
mysql> 트랜잭션 시작;
쿼리 확인, 영향을 받은 행 0개(0.00초)
mysql> INSERT INTO customer VALUES (10, 'Heikki');
쿼리 확인, 영향을 받은 행 1개(0.00초) mysql>
COMMIT;
쿼리 확인, 영향을 받은 행 0개(0.00초)
mysql> -- 자동 커밋이 해제된 상태에서 다른 트랜잭션을 수행합니다.
mysql> SET 자동 커밋=0;
쿼리 확인, 영향을 받은 행 0개(0.00초)
mysql> INSERT INTO customer VALUES (15, 'John');
쿼리 확인, 영향을 받은 행 1개(0.00초)
mysql> INSERT INTO customer VALUES (20, 'Paul');
쿼리 확인, 영향을 받은 행 1개(0.00초)
```

```
mysql> DELETE FROM customer WHERE b = 'Heikki';
쿼리 확인, 영향을 받은 행 1개(0.00초)
mysql> -- 이제 마지막 2개의 삽입과 삭제를 실행 취소합니다.
mysql> ROLLBACK;
쿼리 확인, 영향을 받은 행 0개(0.00초) mysql>
SELECT * FROM customer;
+------+ --------+
| a| b |
+------+ --------+
|10 | 헤이키 |
+------+ --------+
1 행 집합 (0.00초) mysql>
```

클라이언트 측 언어로 된 트랜잭션

PHP, Perl DBI, JDBC, ODBC 또는 MySQL의 표준 C 호출 인터페이스와 같은 API에서는 SELECT 또는 INSERT와 같은 다른 SQL 문과 마찬가지로 COMMIT과 같은 트랜잭션 제어 문을 문자열로 MySQL 서버로 보낼 수 있습니다. 일부 API는 별도의 특수 트랜잭션 커밋 및 롤백 함수나 메서드를 제공하기도 합니다.

15.7.2.3 일관된 비잠금 읽기

일관된 읽기란 InnoDB가 다중 버전 관리를 사용하여 특정 시점의 데이터베이스 스냅샷을 쿼리에 제공한다는 의미입니다. 쿼리는 해당 시점 이전에 커밋된 트랜잭션이 변경한 내용만 보고, 이후 또는 커밋되지 않은 트랜잭션이 변경한 내용은 보지 않습니다. 이 규칙의 예외는 쿼리가 동일한 트랜잭션 내에서 이전 문에 의해 수행된 변경 사항을 볼 수 있다는 것입니다. 이 예외로 인해 다음과 같은 예외가 발생합니다. 테이블의 일부 행을 업데이트하는 경우 SELECT는 업데이트된 행의 최신 버전을 확인하지만 일부 행의 이전 버전도 확인할 수 있습니다. 다른 세션이 동시에 동일한 테이블을 업데이트하는 경우 예외는 데이터베이스에 존재하지 않았던 상태의 테이블이 표시될 수 있음을 의미합니다.

트랜잭션 격리 수준이 반복 읽기(기본 수준)인 경우, 동일한 트랜잭션 내의 모든 일관된 읽기는 해당 트랜잭션에서 첫 번째 읽기에 의해 설정된 스냅샷을 읽습니다. 현재 트랜잭션을 커밋하고 그 후에 새 쿼리를 실행하면 쿼리에 대한 최신 스냅샷을 얻을 수 있습니다.

읽기 커밋 격리 수준을 사용하면 트랜잭션 내에서 일관된 각 읽기가 고유한 새 스냅샷을 설정하고 읽습니다.

일관된 읽기는 InnoDB가 읽기 커밋 및 반복 읽기 격리 수준에서 SELECT 문을 처리하는 기본 모드입니다. 일관된 읽기는 액세스하는 테이블에 잠금을 설정하지 않으므로 다른 세션은 테이블에서 일관된 읽기가 수행되는 동시에 해당 테이블을 자유롭게 수정할 수 있습니다.

기본 REPEATABLE READ 격리 수준에서 실행 중이라고 가정해 보겠습니다. 일관된 읽기(즉, 일반 SELECT 문)를 실행하면 InnoDB는 쿼리가 데이터베이스를 볼 수 있는 시점을 트랜잭션에 제공합니다. 다른 트랜잭션이 행을 삭제하고 타임포인트가 할당된 후에 커밋하는 경우 해당 행이 삭제된 것으로 표시되지 않습니다. 삽입과 업데이트도 비슷하게 처리됩니다.

참고

데이터베이스 상태의 스냅샷은 트랜잭션 내의 `SELECT` 문에 적용되며 반드시 DML 문에는 적용되지 않습니다. 일부 행을 삽입하거나 수정한 다음 해당 트랜잭션을 커밋하는 경우, 세션에서 해당 행을 쿼리할 수 없더라도 동시 진행 중인 다른 반복 가능한 읽기 트랜잭션에서 실행된 `DELETE` 또는 `UPDATE` 문이 방금 커밋된 행에 영향을 미칠 수 있습니다. 트랜잭션이 다른 트랜잭션에 의해 커밋된 행을 업데이트하거나 삭제하는 경우 이러한 변경 사항이 현재 트랜잭션에 표시됩니다. 예를 들어 다음과 같은 상황이 발생할 수 있습니다:

```
SELECT COUNT(c1) FROM t1 WHERE c1 = 'xyz';
```

```
-- 0을 반환합니다: 일치하는 행이 없습니다
. DELETE FROM t1 WHERE c1 =
'xyz';
-- 다른 트랜잭션이 최근에 커밋한 여러 행을 삭제합니다.

SELECT COUNT(c2) FROM t1 WHERE c2 = 'abc';
-- 0을 반환합니다: 일치하는 행이 없습니다.
UPDATE t1 SET c2 = 'cba' WHERE c2 = 'abc';
-- 10개의 행에 영향을 줍니다: 다른 txn이 방금 'abc' 값으로 10개의 행을 커밋했습니다.
SELECT COUNT(c2) FROM t1 WHERE c2 = 'cba';
-- 10을 반환합니다: 이 txn은 이제 방금 업데이트한 행을 볼 수 있습니다.
```

트랜잭션을 커밋한 다음 다른 SELECT 또는
일관된 스냅샷으로 트랜잭션을 시작하세요.

이를 *다중 버전 동시성 제어라고* 합니다.

다음 예제에서 세션 A는 B가 삽입을 커밋하고 A도 커밋한 경우에만 B가 삽입한 행을 볼 수 있으므로 시점도 B의 커밋을 지나서 앞당겨집니다.

```
                세션 A                  세션 B

                SET 자동 커밋=0;        SET 자동 커밋=0;
시간
|               SELECT * FROM t;
|               빈 세트
|                                       INSERT INTO t VALUES (1, 2);
|
v               SELECT * FROM t;
                빈 세트
                                        커밋;
                SELECT * FROM t;
                빈 세트 COMMIT;


                SELECT * FROM t;
                ----------------------
                |    1    |    2    |
                ----------------------
```

데이터베이스의 "가장 최신" 상태를 보려면 읽기 커밋 격리 수준 또는 잠금 읽기를 사용하세요:

```
SELECT * FROM t FOR SHARE;
```

READ COMMITTED 격리 수준을 사용하면 트랜잭션 내의 각 일관된 읽기가 자체의 새 스냅샷을 설정하고 읽습니다. FOR SHARE를 사용하면 대신 잠금 읽기가 발생합니다: 가장 최신 행을 포함하는 트랜잭션이 종료될 때까지 SELECT가 차단됩니다(15.7.2.4절, "읽기 잠금" 참조).

특정 DDL 문에서는 일관된 읽기가 작동하지 않습니다:

- MySQL은 삭제된 테이블을 사용할 수 없고 InnoDB가 테이블을 삭제하기 때문에 일관된 읽기는 테이블 삭제에 대해 작동하지 않습니다.
- 일관된 읽기는 원본 테이블의 임시 복사본을 만들고 임시 복사본이 작성될 때 원본 테이블을 삭제하는 ALTER TABLE 작업에 대해 작동하지 않습니다. 트랜잭션 내에서 일관된 읽기를 다시 발행하면 트랜잭션의 스냅샷을 생성할 때 해당 행이 존재하지 않았기 때문에 새 테이블의 행이 표시되지 않습니다. 이 경우 트랜잭션은 오류를 반환합니다: ER_TABLE_DEF_CHANGED, "테이블 정의가 변경되었습니다. 트랜잭션

을 다시 시도하십시오".

읽기 유형은 `INSERT INTO ...`와 같은 절의 선택에 따라 다릅니다. `SELECT`, `UPDATE ... (SELECT)` 및 `CREATE TABLE ...` SELECT와 같은 절의 읽기 유형은 `FOR UPDATE` 또는 `FOR SHARE`를 지정하지 않습니다:

- 기본적으로 `InnoDB`는 이러한 문에 대해 더 강력한 잠금을 사용하며 `SELECT` 부분은 동일한 트랜잭션 내에서도 일관된 각 읽기가 자체의 새로운 스냅샷을 설정하고 읽는 `READ` COMMITTED와 같이 작동합니다.

- 이러한 경우 잠금 해제 읽기를 수행하려면 트랜잭션의 격리 수준을 읽기 미완료 또는 읽기 커밋으로 설정하여 선택한 테이블에서 읽은 행에 잠금이 설정되지 않도록 합니다.

15.7.2.4 읽기 잠금

데이터를 쿼리한 다음 동일한 트랜잭션 내에서 관련 데이터를 삽입하거나 업데이트하는 경우 일반 SELECT 문으로는 충분한 보호 기능을 제공하지 못합니다. 다른 트랜잭션이 방금 쿼리한 것과 동일한 행을 업데이트하거나 삭제할 수 있기 때문입니다. InnoDB는 추가적인 안전성을 제공하는 두 가지 유형의 잠금 읽기를 지원합니다:

- 공유를 위해 ... 선택

  읽은 모든 행에 공유 모드 잠금을 설정합니다. 다른 세션은 행을 읽을 수 있지만 트랜잭션이 커밋될 때까지 행을 수정할 수 없습니다. 이러한 행 중 하나라도 다른 세션에 의해 변경된 경우 트랜잭션이 아직 커밋되지 않은 경우 쿼리는 해당 트랜잭션이 종료될 때까지 기다린 다음 최신 값을 사용합니다.

  참고

  SELECT ... FOR SHARE는 SELECT ... LOCK IN SHARE MODE를 대체하지만 이전 버전과의 호환성을 위해 공유 모드 잠금을 계속 사용할 수 있습니다. 문은 동일합니다. 그러나 FOR SHARE는 OF *table_name*, NOWAIT 및 SKIP LOCKED 옵션을 지원합니다. NOWAIT 및 SKIP LOCKED로 읽기 동시성 잠그기를 참조하세요.

  SELECT ... FOR SHARE에는 SELECT 권한이 필요합니다.

  SELECT ... FOR SHARE 문은 MySQL 부여 테이블에서 읽기 잠금을 획득하지 않습니다. 자세한 내용은 테이블 동시성 부여를 참조하십시오.

- 업데이트하려면 ... 선택

  인덱스 레코드에 대해 검색이 발생하면 해당 행에 대해 UPDATE 문을 실행한 것과 동일하게 행과 연결된 모든 인덱스 항목을 잠급니다. 다른 트랜잭션은 해당 행을 업데이트하거나 SELECT ... FOR SHARE를 수행하거나 특정 트랜잭션 격리 수준에서 데이터를 읽지 못하도록 차단됩니다. 일관된 읽기는 읽기 보기에 존재하는 레코드에 설정된 모든 잠금을 무시합니다.
  (이전 버전의 레코드는 잠글 수 없으며, 메모리 내 레코드 사본에 실행 취소 로그를 적용하여 재구성됩니다.)

  SELECT ... FOR UPDATE를 수행하려면 SELECT 권한과 DELETE, LOCK TABLES 또는 UPDATE 권한 중 하나 이상이 필요합니다.

이러한 절은 주로 단일 테이블에 있거나 여러 테이블로 분할된 트리 구조 또는 그래프 구조 데이터를 다룰 때 유용합니다. 한 곳에서 다른 곳으로 가장자리나 트리 분기를 이동하면서 다시 돌아와서 이러한 '포인터' 값을 변경할 수 있는 권한을 보유할 수 있습니다.

트랜잭션이 커밋되거나 롤백될 때 FOR 공유 및 FOR 업데이트 쿼리에 의해 설정된 모든 잠금이 해제됩니다.

참고

읽기 잠금은 자동 커밋이 비활성화된 경우에만 가능합니다(트랜잭션 시작을 통해 트랜잭션을 시작하거나 자동 커밋을 0으로 설정하여).

외부 문의 잠금 읽기 절은 하위 쿼리에도 잠금 읽기 절을 지정하지 않는 한 중첩된 하위 쿼리에서 테이블의 행을 잠그지 않습니다. 예를 들어 다음 문은 테이블 t2의 행을 잠그지 않습니다.

```
SELECT * FROM t1 WHERE c1 = (SELECT c1 FROM t2) FOR UPDATE;
```

테이블 t2의 행을 잠그려면 하위 쿼리에 잠금 읽기 절을 추가합니다:

```
SELECT * FROM t1 WHERE c1 = (SELECT c1 FROM t2 FOR UPDATE) FOR UPDATE;
```

읽기 잠금 예제

테이블 자식에 새 행을 삽입하고 자식 행이 테이블 부모에 부모 행을 가지고 있는지 확인한다고 가정해 보겠습니다. 애플리케이션 코드는 이 작업 시퀀스 전체에서 참조 무결성을 보장할 수 있습니다.

먼저 일관된 읽기를 사용하여 PARENT 테이블을 쿼리하고 부모 행이 존재하는지 확인합니다. CHILD 테이블에 자식 행을 안전하게 삽입할 수 있을까요? 아니요, 다른 세션에서 SELECT와 INSERT 사이에 사용자가 인지하지 못하는 사이에 부모 행을 삭제할 수 있기 때문입니다.

이 잠재적인 문제를 방지하려면 FOR SHARE를 사용하여 SELECT를 수행합니다:

```
SELECT * FROM parent WHERE NAME = 'Jones' FOR SHARE;
```

FOR SHARE 쿼리가 부모 'Jones'를 반환하면 자식 레코드를 CHILD 테이블에 안전하게 추가하고 트랜잭션을 커밋할 수 있습니다. PARENT 테이블의 해당 행에서 독점 잠금을 획득하려고 시도하는 모든 트랜잭션은 작업이 완료될 때까지, 즉 모든 테이블의 데이터가 일관된 상태가 될 때까지 대기합니다.

다른 예로, CHILD 테이블에 추가된 각 자식에 고유 식별자를 할당하는 데 사용되는 CHILD_CODES 테이블의 정수 카운터 필드를 생각해 보겠습니다. 데이터베이스의 두 사용자가 카운터에 대해 동일한 값을 볼 수 있고 두 트랜잭션이 동일한 식별자를 가진 행을 CHILD 테이블에 추가하려고 하면 중복 키 오류가 발생하므로 카운터의 현재 값을 읽기 위해 일관된 읽기 또는 공유 모드 읽기를 사용하지 마십시오.

두 명의 사용자가 동시에 카운터를 읽으면 적어도 한 명은 카운터를 업데이트하려고 할 때 교착 상태에 빠지기 때문에 FOR SHARE는 좋은 솔루션이 아닙니다.

카운터 읽기 및 증가를 구현하려면 먼저 다음을 사용하여 카운터 잠금 읽기를 수행합니다.
업데이트를 클릭한 다음 카운터를 증가시킵니다. 예를 들어

```
SELECT counter_field FROM child_codes FOR UPDATE;
UPDATE child_codes SET counter_field = counter_field + 1;
```

SELECT ... FOR UPDATE는 사용 가능한 최신 데이터를 읽고, 읽는 각 행에 독점 잠금을 설정합니다. 따라서 검색된 SQL UPDATE가 행에 설정하는 것과동일한 잠금을 설정합니다.

앞의 설명은 SELECT ... FOR UPDATE의 작동 방식을 보여주는 예일 뿐입니다. MySQL에서 고유 식별자를 생성하는 특정 작업은 실제로 테이블에 대한 단 한 번의 액세스만으로 수행할 수 있습니다:

```
UPDATE child_codes SET counter_field = LAST_INSERT_ID(counter_field + 1);
SELECT LAST_INSERT_ID();
```

SELECT 문은 식별자 정보(현재 연결에만 해당)만 검색합니다. 테이블에는 액세스하지 않습니다.

NOWAIT 및 건너뛰기 잠금을 통한 읽기 동시성 잠금

트랜잭션에 의해 행이 잠긴 경우 동일한 잠긴 행을 요청하는 SELECT ... FOR UPDATE 또는 SELECT ... FOR SHARE 트랜잭션은 차단 트랜잭션이 행 잠금을 해제할 때까지 기다려야 합니다. 이 동작은 다른 트랜잭션에서 업데이트를 위해 쿼리된 행을 트랜잭션이 업데이트하거나 삭제하는 것을 방지합니다. 그러나 요청된 행이 잠겨 있을 때 쿼리가 즉시 반환되도록 하거나 결과 집합에서 잠긴 행을 제외하는 것이 허용되는 경우에

는 행 잠금이 해제될 때까지 기다릴 필요가 없습니다.

다른 트랜잭션이 행 잠금을 해제할 때까지 기다리지 않으려면 `NOWAIT` 및 `SKIP LOCKED` 옵션을 `SELECT ... FOR UPDATE` 또는 `SELECT ... FOR SHARE` 잠금 읽기 문과 함께 사용할 수 있습니다.

- `NOWAIT`

  `NOWAIT`를 사용하는 잠금 읽기는 행 잠금을 획득하기 위해 기다리지 않습니다. 쿼리는 즉시 실행되며, 요청된 행이 잠겨 있으면 오류와 함께 실패합니다.

- 잠금 건너뛰기

  SKIP LOCKED를 사용하는 잠금 읽기는 행 잠금을 획득하기 위해 기다리지 않습니다. 쿼리가 즉시 실행되어 결과 집합에서 잠긴 행을 제거합니다.

참고

잠긴 행을 건너뛰는 쿼리는 데이터의 일관되지 않은 보기를 반환합니다. 따라서 잠금 건너뛰기는 일반적인 트랜잭션 작업에는 적합하지 않습니다. 그러나 여러 세션이 동일한 큐와 같은 테이블에 액세스할 때 잠금 경합을 피하는 데 사용할 수 있습니다.

NOWAIT 및 건너뛰기 잠금은 행 수준 잠금에만 적용됩니다.

NOWAIT 또는 SKIP LOCKED를 사용하는 문은 문 기반 복제에 안전하지 않습니다.

다음 예시는 NOWAIT와 SKIP LOCKED를 보여줍니다. 세션 1은 단일 레코드에 대해 행 잠금을 취하는 트랜잭션을 시작합니다. 세션 2는 NOWAIT 옵션을 사용하여 동일한 레코드에 대한 잠금 읽기를 시도합니다. 요청된 행이 세션 1에 의해 잠겨 있기 때문에 잠금 읽기는 오류와 함께 즉시 반환됩니다. 세션 3에서 SKIP LOCKED를 사용한 잠금 읽기는 세션 1에 의해 잠긴 행을 제외한 요청된 행을 반환합니다.

```
# 세션 1:

mysql> CREATE TABLE t (i INT, PRIMARY KEY (i)) ENGINE = InnoDB;

mysql> INSERT INTO t (i) VALUES(1),(2),(3;)

mysql> 트랜잭션 시작;

mysql> SELECT * FROM t WHERE i = 2 FOR UPDATE;
+---+
| i |
+---+
| 2 |
+---+

# 세션 2:

mysql> 트랜잭션 시작;

mysql> SELECT * FROM t WHERE i = 2 FOR UPDATE NOWAIT;
오류 3572(HY000): 잠금을 기다리지 마십시오. #


세션 3:


mysql> 트랜잭션 시작;

mysql> SELECT * FROM t FOR UPDATE SKIP LOCKED;
+---+
| i |
+---+
| 1 |
| 3 |
+---+
```

15.7.3 InnoDB의 다양한 SQL 문에 의해 설정된 잠금

잠금 읽기, 업데이트 또는 삭제는 일반적으로 SQL 문을 처리할 때 스캔되는 모든 인덱스 레코드에 레코드 잠금을 설정합니다. 문에 행을 제외할 수 있는 `WHERE` 조건이 있는지 여부는 중요하지 않습니다. InnoDB는 정확한 `WHERE` 조건과 동일하지만 어떤 인덱스 범위가 스캔되었는지만 알 수 있습니다. 잠금은 일반적으로 다음 키 잠금으로, 레코드 바로 앞의 '간격'에 삽입되는 것을 차단합니다. 그러나 갭 잠금을 명시적으로 비활성화하면 다음 키 잠금이 사용되지 않을 수 있습니다. 자세한 내용은 섹션 15.7.1, "InnoDB 잠금"을 참조하세요. 트랜잭션 격리 수준도 설정되는 잠금에 영향을 줄 수 있습니다(섹션 15.7.2.1, "트랜잭션 격리 수준"을 참조하세요).

검색에 보조 인덱스가 사용되고 설정할 인덱스 레코드 잠금이 배타적인 경우 InnoDB는 또한 해당 클러스터된 인덱스 레코드를 검색하고 잠금을 설정합니다.

문에 적합한 인덱스가 없고 MySQL이 문 처리를 위해 전체 테이블을 스캔해야 하는 경우 테이블의 모든 행이 잠기게 되어 다른 사용자의 테이블에 대한 모든 삽입이 차단됩니다. 쿼리가 필요 이상으로 많은 행을 스캔하지 않도록 좋은 인덱스를 생성하는 것이 중요합니다.

InnoDB는 다음과 같이 특정 유형의 잠금을 설정합니다.

- SELECT ... FROM은 일관된 읽기로, 트랜잭션 격리 수준을 SERIALIZABLE로 설정하지 않는 한 데이터베이스의 스냅샷을 읽고 잠금을 설정하지 않습니다. SERIALIZABLE 레벨의 경우 검색은 발견되는 인덱스 레코드에 공유 다음 키 잠금을 설정합니다. 그러나 고유 인덱스를 사용하여 고유 행을 검색하기 위해 행을 잠그는 문에는 인덱스 레코드 잠금만 필요합니다.

- 고유 인덱스를 사용하는 SELECT ... FOR UPDATE 및 SELECT ... FOR SHARE 문은 스캔된 행에 대한 잠금을 획득하고 결과 집합에 포함할 자격이 없는 행(예: WHERE 절에 지정된 기준을 충족하지 않는 경우)에 대한 잠금을 해제합니다. 그러나 쿼리 실행 중에 결과 행과 원본 소스 간의 관계가 손실되어 행의 잠금이 즉시 해제되지 않는 경우도 있습니다. 예를 들어, UNION에서는 테이블에서 스캔된(그리고 잠긴) 행이 결과 집합에 적합한지 평가하기 전에 임시 테이블에 삽입될 수 있습니다. 이 경우 임시 테이블의 행과 원본 테이블의 행의 관계가 손실되고 쿼리 실행이 끝날 때까지 후자의 행은 잠금이 해제되지 않습니다.

- 읽기 잠금(SELECT(FOR UPDATE 또는 FOR SHARE 포함), UPDATE 및 DELETE 문)의 경우 수행되는 잠금은 문이 고유 검색 조건이 있는 고유 인덱스를 사용하는지 또는 범위 유형 검색 조건을 사용하는지에 따라 달라집니다.

  - 고유한 검색 조건이 있는 고유 인덱스의 경우, InnoDB는 발견된 인덱스 레코드만 잠그고 그 앞의 간격은 잠그지 않습니다.

  - 다른 검색 조건과 고유하지 않은 인덱스의 경우, InnoDB는 갭 잠금 또는 다음 키 잠금을 사용하여 스캔된 인덱스 범위를 잠그고 다른 세션이 해당 갭에 삽입하는 것을 차단합니다.
    범위로 제한합니다. 갭 잠금 및 다음 키 잠금에 대한 자세한 내용은 섹션 15.7.1, "InnoDB 잠금"을 참조하세요.

- 검색에서 인덱스 레코드가 발견되면 SELECT ... FOR UPDATE는 다른 세션이 SELECT ... FOR SHARE를 수행하거나 특정 트랜잭션 격리 수준에서 읽지 못하도록 차단합니다. 일관된 읽기는 읽기 뷰에 존재하는 레코드에 설정된 모든 잠금을 무시합니다.

- UPDATE ... WHERE ...는 검색에서 발견되는 모든 레코드에 대해 독점적인 다음 키 잠금을 설정합니다. 그러나 고유 인덱스를 사용하여 고유 행을 검색하기 위해 행을 잠그는 문에는 인덱스 레코드 잠금만 필요합니다.

- UPDATE가 클러스터된 인덱스 레코드를 수정할 때 영향을 받는 보조 인덱스 레코드에 암시적 잠금이 수행됩니다. 또한 새 보조 인덱스 레코드를 삽입하기 전에 중복 검사 스캔을 수행할 때와 새 보조 인덱스 레코드

를 삽입할 때 업데이트 작업은 영향을 받는 보조 인덱스 레코드에 대해 공유 잠금을 수행합니다.

- `DELETE FROM ... WHERE ...`는 검색에서 발견되는 모든 레코드에 대해 독점적인 다음 키 잠금을 설정합니다. 그러나 고유 인덱스를 사용하여 고유 행을 검색하기 위해 행을 잠그는 문에는 인덱스 레코드 잠금만 필요합니다.

- `INSERT`는 삽입된 행에 독점 잠금을 설정합니다. 이 잠금은 다음 키 잠금이 아닌 인덱스 레코드 잠금이며(즉, 갭 잠금이 없음) 다른 세션이 삽입된 행 앞의 갭에 삽입되는 것을 방지하지 않습니다.

  행을 삽입하기 전에 삽입 의도 갭 잠금이라고 하는 일종의 갭 잠금이 설정됩니다. 이 잠금은 동일한 인덱스 갭에 삽입하는 여러 트랜잭션이 갭 내의 동일한 위치에 삽입하지 않는 경우 서로를 기다릴 필요가 없도록 삽입 의도를 알립니다. 값이 4와 7인 인덱스 레코드가 있다고 가정해 보겠습니다. 삽입을 시도하는 트랜잭션을 분리합니다.

값이 각각 5와 6이면 삽입된 행에 대한 독점 잠금을 얻기 전에 삽입 의도 잠금으로 4와 7 사이의 간격을 잠그지만 행이 충돌하지 않으므로 서로를 차단하지 않습니다.

중복 키 오류가 발생하면 중복 인덱스 레코드에 공유 잠금이 설정됩니다. 공유 잠금을 사용하면 다른 세션에 이미 독점 잠금이 있는 경우 여러 세션이 동일한 행을 삽입하려고 시도하는 경우 교착 상태가 발생할 수 있습니다. 다른 세션이 행을 삭제하면 이러한 문제가 발생할 수 있습니다. `InnoDB` 테이블 `t1`의 구조가 다음과 같다고 가정해 보겠습니다:

```
CREATE TABLE t1 (i INT, PRIMARY KEY (i)) ENGINE = InnoDB;
```

이제 세 세션이 순서대로 다음 작업을 수행한다고 가정해 보겠습니다: 세션 1:

```
거래를 시작합니다;
INSERT INTO t1 VALUES(1);
```

세션 2:

```
거래를 시작합니다;
INSERT INTO t1 VALUES(1);
```

세션 3:

```
거래를 시작합니다;
INSERT INTO t1 VALUES(1);
```

세션 1:

```
롤백;
```

세션 1의 첫 번째 작업은 행에 대한 독점 잠금을 획득합니다. 세션 2와 세션 3의 작업은 모두 중복 키 오류를 발생시키고 둘 다 행에 대한 공유 잠금을 요청합니다. 세션 1이 롤백하면 행에 대한 독점 잠금이 해제되고 세션 2 및 3에 대한 대기 중인 공유 잠금 요청이 승인됩니다. 이 시점에서 세션 2와 3은 교착 상태가 됩니다: 상대방이 보유한 공유 잠금으로 인해 어느 쪽도 행에 대한 독점 잠금을 획득할 수 없습니다.

테이블에 이미 키 값이 1인 행이 포함되어 있고 세 세션이 순서대로 다음 작업을 수행하는 경우에도 비슷한 상황이 발생합니다:

세션 1:

```
거래를 시작합니다;
DELETE FROM t1 WHERE i = 1;
```

세션 2:

```
거래를 시작합니다;
INSERT INTO t1 VALUES(1);
```

세션 3:

```
거래를 시작합니다;
INSERT INTO t1 VALUES(1);
```

세션 1:

```
커밋;
```

세션 1의 첫 번째 작업은 행에 대한 독점 잠금을 획득합니다. 세션 2와 세션 3의 작업은 모두 중복 키 오류를 발생시키고 둘 다 행에 대한 공유 잠금을 요청합니다. 세션 1이 커밋하면 행에 대한 독점 잠금이 해제되고 세션 2 및 3에 대한 대기 중인 공유 잠금 요청이 허용됩니다. 이 시점에서 세션 2와 3은 교착 상태가 됩니다: 상대방이 보유한 공유 잠금으로 인해 어느 쪽도 행에 대한 독점 잠금을 획득할 수 없습니다.

- `INSERT ... ON DUPLICATE KEY UPDATE`는 중복 키 오류가 발생할 때 업데이트할 행에 공유 잠금이 아닌 독점 잠금이 설정된다는 점에서 단순 `INSERT`와 다릅니다. 중복된 기본 키 값에 대해 독점 인덱스-레코드 잠금이 수행됩니다. 중복된 고유 키 값에 대해 독점적인 다음 키 잠금이 수행됩니다.

- 고유 키에 충돌이 없는 경우 `REPLACE`는 `INSERT`와 같이 수행됩니다. 그렇지 않으면 대체할 행에 독점적인 다음 키 잠금이 설정됩니다.

- `INSERT INTO T SELECT ... FROM S WHERE ...`는 `T`에 삽입된 각 행에 배타적 인덱스 레코드 잠금(갭 잠금 없음)을 설정합니다. 트랜잭션 격리 수준이 `READ COMMITTED`인 경우 `InnoDB`는 `S`에서 일관된 읽기(잠금 없음)로 검색을 수행합니다. 그렇지 않으면 `InnoDB`는 `S`의 행에 대해 공유 다음 키 잠금을 설정합니다: 문 기반 바이너리 로그를 사용하여 롤포워드 복구하는 동안 모든 SQL 문은 원래 수행된 것과 정확히 동일한 방식으로 실행되어야 합니다.

  `CREATE TABLE ... SELECT ...`는 공유된 다음 키 잠금으로 `SELECT`를 수행하거나 `INSERT ... SELECT`와 같이 일관된 읽기로 수행합니다.

  `SELECT`가 `REPLACE INTO t SELECT ... FROM s WHERE ...` 또는 `UPDATE t ...` 구문에 사용되는 경우 `WHERE col IN (SELECT ... FROM s ...)` 구문에서 `InnoDB`는 테이블 `s`의 행에 공유 다음 키 잠금을 설정합니다.

  - `InnoDB`는 `AUTO_INCREMENT`와 연결된 인덱스의 끝에 독점 잠금을 설정합니다.
열을 초기화하는 동안 테이블에서 이전에 지정한 `AUTO_INCREMENT` 열을초기화합니다.

  `innodb_autoinc_lock_mode=0`을 사용하면 `InnoDB`는 자동 증가 카운터에 액세스하는 동안 현재 SQL 문의 끝(전체 트랜잭션의 끝이 아님)까지 잠금을 획득하고 유지하는 특수한 `AUTO-INC` 테이블 잠금 모드를 사용합니다. `AUTO-INC` 테이블 잠금이 유지되는 동안 다른 클라이언트는 테이블에 삽입할 수 없습니다. `innodb_autoinc_lock_mode=1`을 사용하는 "대량 삽입"에 대해서도 동일한 동작이 발생합니다. 테이블 수준 `AUTO-INC` 잠금은 `innodb_autoinc_lock_mode=2`와 함께 사용되지 않습니다. 자세한 내용은 15.6.1.6절, "InnoDB에서 자동 인크립션 처리"를 참고한다.

  `InnoDB`는 잠금을 설정하지 않고 이전에 초기화된 `AUTO_INCREMENT` 열의 값을 가져옵니다.

- 테이블에 외래 키 제약 조건이 정의되어 있는 경우 제약 조건을 확인해야 하는 모든 삽입, 업데이트 또는 삭제는 제약 조건을 확인하기 위해 살펴보는 레코드에 공유 레코드 수준 잠금을 설정합니다. `InnoDB`는 제약 조건이 실패하는 경우에도 이러한 잠금을 설정합니다.

- 테이블 잠금은 테이블 잠금을 설정하지만, 이러한 잠금을 설정하는 것은 `InnoDB` 계층보다상위의 MySQL 계층입니다. `innodb_table_locks = 1`(기본값)이고 `autocommit = 0`이면 InnoDB는 테이블 잠금을 인식하고, `InnoDB` 위의 MySQL 계층은 행 수준 잠금에 대해 인식합니다.

  그렇지 않으면 InnoDB의 자동 교착 상태 감지 기능이 이러한 테이블 잠금이 관련된 교착 상태를 감지할 수 없습니다. 또한 이 경우 상위 MySQL 계층은 행 수준 잠금에 대해 알지 못하므로 다른 세션에 현재 행 수준 잠

금이 있는 테이블에서 테이블 잠금이 발생할 수 있습니다. 그러나 15.7.5.2절 "교착 상태 감지"에서 설명한 것처럼 트랜잭션 무결성을 위태롭게 하지는 않습니다.

- 테이블 잠금은 `innodb_table_locks=1`(기본값)인 경우 각 테이블에 대해 두 개의 잠금을 획득합니다. MySQL 계층의 테이블 잠금 외에도 `InnoDB` 테이블 잠금도 획득합니다. `InnoDB` 테이블 잠금을 획득하지 않으려면 `innodb_table_locks=0`으로 설정합니다. `InnoDB` 테이블 잠금을 획득하지 않으면 테이블의 일부 레코드가 다른 트랜잭션에 의해 잠겨 있더라도 `LOCK TABLES`가 완료됩니다.

  MySQL 8.2에서 `innodb_table_locks=0`은 명시적으로 `LOCK TABLES ... WRITE`. 그러나 `LOCK TABLES ...` WRITE로 읽기 또는 쓰기를 위해 잠긴 테이블에는 효과가 있다. `WRITE` 로 암시적으로 (예: 트리거를 통해) 또는 `LOCK TABLES ... READ`.

- 트랜잭션이 보유한 모든 InnoDB 잠금은 트랜잭션이 커밋되거나 중단될 때 해제됩니다. 따라서 자동 커밋 =1 모드에서는 획득한 InnoDB 테이블 잠금이 즉시 해제되기 때문에 InnoDB 테이블에서 LOCK TABLES 를 호출하는 것은 의미가 없습니다.
- 테이블 잠금은 암시적 커밋 및 테이블 잠금 해제를 수행하므로 트랜잭션 중간에 추가 테이블을 잠글 수 없습니다.

15.7.4 팬텀 행

소위 *팬텀* 문제는 동일한 쿼리가 서로 다른 시간에 서로 다른 행 집합을 생성할 때 트랜잭션 내에서 발생합니다 . 예를 들어 SELECT가 두 번 실행되었지만 두 번째 실행 시 처음에 반환되지 않은 행이 반환되는 경우 해당 행은 "팬텀" 행입니다.

하위 테이블의 id 열에 인덱스가 있고 나중에 선택한 행의 일부 열을 업데이트할 의도로 식별자 값이 100보다 큰 테이블의 모든 행을 읽고 잠그고 싶다고 가정해 보겠습니다:

```
SELECT * FROM child WHERE id > 100 FOR UPDATE;
```

쿼리는 id가100보다 큰 첫 번째 레코드부터 인덱스를 스캔합니다. 테이블에 id 값이 90과 102인 행이 포함되도록 합니다. 스캔 범위의 인덱스 레코드에 설정된 잠금이 그 간격(이 경우 90과 102 사이의 간격)에 삽입되는 것을 잠그지 않으면 다른 세션이 테이블에 101의 ID를 가진 새 행을 삽입할 수 있습니다. 동일한 트랜잭션 내에서 동일한 SELECT를 실행하는 경우 쿼리가 반환하는 결과 집합에 ID가 101인 새 행("팬텀")이 표시됩니다. 행 집합을 데이터 항목으로 간주하면, 새로운 팬텀 자식은 트랜잭션이 트랜잭션 중에 읽은 데이터가 변경되지 않도록 트랜잭션이 실행될 수 있어야 한다는 트랜잭션 격리 원칙을 위반하게 됩니다.

팬텀을 방지하기 위해 InnoDB는 인덱스 행 잠금과 갭 잠금을 결합한 *다음 키 잠금*이라는 알고리즘을 사용합니다. InnoDB는 테이블 인덱스를 검색하거나 스캔할 때 발견되는 인덱스 레코드에 공유 또는 독점 잠금을 설정하는 방식으로 행 수준 잠금을 수행합니다. 따라서 행 수준 잠금은 실제로는 인덱스 레코드 잠금입니다. 또한 인덱스 레코드에 대한 다음 키 잠금은 인덱스 레코드 앞의 "갭"에도 영향을 줍니다. 즉, 다음 키 잠금은 인덱스 레코드 잠금과 인덱스 레코드 앞의 갭에 대한 갭 잠금입니다. 한 세션이 인덱스의 레코드 R에 대해 공유 또는 독점 잠금을 가지고 있는 경우, 다른 세션은 인덱스 순서에서 R 바로 앞의 갭에 새 인덱스 레코드를 삽입할 수 없습니다 .

InnoDB가 인덱스를 스캔할 때 인덱스의 마지막 레코드 이후의 간격을 잠글 수도 있습니다. 앞의 예제에서 바로 그런 일이 발생했습니다: id가 100보다 큰 테이블에 삽입되는 것을 방지하기 위해 InnoDB에서 설정한 잠금에는 id 값 102 다음 간격에 대한 잠금이 포함됩니다.

다음 키 잠금을 사용하여 애플리케이션에서 고유성 검사를 구현할 수 있습니다: 공유 모드에서 데이터를 읽을 때 삽입하려는 행에 대한 중복이 보이지 않으면 안전하게 행을 삽입할 수 있으며, 읽는 동안 행의 후속 키에 설정된 다음 키 잠금이 행에 대한 중복 삽입을 방지한다는 것을 알 수 있습니다. 따라서 다음 키 잠금을 사용하면 테이블에 없는 항목을 '잠금'할 수 있습니다.

갭 잠금은 섹션 15.7.1, "InnoDB 잠금"에 설명된 대로 비활성화할 수 있습니다. 갭 잠금이 비활성화되면 다른 세션이 갭에 새 행을 삽입할 수 있으므로 팬텀 문제가 발생할 수 있습니다.

15.7.5 InnoDB의 교착 상태

교착 상태는 서로 다른 트랜잭션이 상대방이 필요로 하는 잠금을 보유하고 있어 진행할 수 없는 상황입니다. 두 트랜잭션 모두 리소스를 사용할 수 있게 되기를 기다리고 있기 때문에 어느 쪽도 보유하고 있는 잠금을 해제하지 못합니다.

교착 상태는 트랜잭션이 여러 테이블의 행을 잠그는 경우(`UPDATE` 또는 `SELECT ... FOR UPDATE`와 같은 문을 통해), 그러나 반대 순서로 잠글 때 발생할 수 있습니다. 이러한 문이 인덱스 레코드 및 간격의 범위를 잠그는 경우에도 교착 상태가 발생할 수 있으며, 각 트랜잭션이 일부 잠금을 획득하지만

타이밍 문제로 인해 다른 항목이 아닙니다. 교착 상태의 예는 15.7.5.1절, "InnoDB 교착 상태 예"를 참조하십시오.

교착 상태의 가능성을 줄이려면 `LOCK TABLES` 문 대신 트랜잭션을 사용하고, 데이터를 삽입하거나 업데이트하는 트랜잭션은 장시간 열려 있지 않을 정도로 작게 유지하세요.
여러 트랜잭션이 여러 테이블 또는 큰 범위의 행을 업데이트하는 경우 각 트랜잭션에서 동일한 작업 순서(예: `SELECT ... FOR UPDATE`)를 사용하고, `SELECT ... FOR UPDATE` 및 `UPDATE ...` 문에 사용되는 열에 인덱스를 생성합니다. `WHERE` 문에 사용되는 칼럼에 인덱스를 생성합니다. 격리 수준은 읽기 작업의 동작을 변경하는 반면 교착 상태는 쓰기 작업으로 인해 발생하므로 격리 수준은 교착 상태 발생 가능성에 영향을 미치지 않습니다. 자세한 내용은
교착 상태를 피하고 복구하는 방법은 15.7.5.3절 "교착 상태를 최소화하고 처리하는 방법"을 참조하세요.

교착 상태 감지가 활성화되어 있고(기본값) 교착 상태가 발생하면 `InnoDB`는 조건을 감지하고 트랜잭션 중 하나(피해 트랜잭션)를 롤백합니다. `innodb_deadlock_detect` 변수를 사용하여 교착 상태 감지가 비활성화되어 있는 경우, `InnoDB`는 교착 상태 발생 시 `innodb_lock_wait_timeout` 설정에 의존하여 트랜잭션을 롤백합니다. 따라서 애플리케이션 로직이 올바르더라도 트랜잭션을 다시 시도해야 하는 경우를 처리해야 합니다. `InnoDB` 사용자 트랜잭션의 마지막 교착 상태를 보려면 엔진 `INNODB` 상태 표시를 사용합니다. 잦은 교착 상태가 트랜잭션 구조 또는 애플리케이션 오류 처리에 문제가 있음을 나타내는 경우, 모든 교착 상태에 대한 정보를 `mysqld` 오류 로그에 인쇄하도록 `innodb_print_all_deadlocks`를 활성화합니다. 교착 상태가 자동으로 감지되고 처리되는 방법에 대한 자세한 내용은 섹션 15.7.5.2, "교착 상태 감지"를 참조하십시오.

15.7.5.1 InnoDB 교착 상태 예시

다음 예는 잠금 요청으로 인해 교착 상태가 발생할 때 오류가 어떻게 발생할 수 있는지 보여줍니다. 이 예시에는 두 개의 클라이언트인 A와 B가 포함됩니다.

InnoDB 상태에는 마지막 교착 상태에 대한 세부 정보가 포함되어 있습니다. 잦은 교착 상태의 경우 전역 변수 `innodb_print_all_deadlock`. 오류 로그에 교착 상태 정보를 추가합니다.

클라이언트 A는 `innodb_print_all_deadlock`을 활성화하고, 'Animals' 및 'Birds' 두 개의 테이블을 생성한 후 각각에 데이터를 삽입합니다. 클라이언트 A가 트랜잭션을 시작하고 공유 모드에서 Animals의 행을 선택합니다:

```
mysql> SET GLOBAL innodb_print_all_deadlock = ON;
쿼리 확인, 영향을 받은 행 0개(0.00초)

mysql> CREATE TABLE Animals (name VARCHAR(10) PRIMARY KEY, value INT) ENGINE = InnoDB;
쿼리 확인, 영향을 받는 행 0개(0.01초)

mysql> CREATE TABLE Birds (name VARCHAR(10) PRIMARY KEY, value INT) ENGINE = InnoDB;
쿼리 확인, 영향을 받는 행 0개(0.01초)

mysql> INSERT INTO Animals (name,value) VALUES ("Aardvark",10;)
쿼리 확인, 영향을 받은 행 1개(0.00초)

mysql> INSERT INTO Birds (name,value) VALUES ("Buzzard",20);
쿼리 확인, 영향을 받은 행 1개(0.00초)

mysql> 트랜잭션 시작;
쿼리 확인, 영향을 받은 행 0개(0.00초)

mysql> SELECT value FROM Animals WHERE name='Aardvark' FOR SHARE;
+--------+
| 값 |
+--------+
     |10 |
+--------+
1행 1세트(0.00초)
```

다음으로 클라이언트 B가 트랜잭션을 시작하고 공유 모드의 Birds에서 행을 선택합니다:

```
mysql> 트랜잭션 시작;
쿼리 확인, 영향을 받은 행 0개(0.00초)
```

```
mysql> SELECT value FROM Birds WHERE name='Buzzard' FOR SHARE;
+-------+
| 값 |
+-------+
     |20 |
+-------+
한 세트에 1행(0.00초)
```

성능 스키마는 두 개의 select 문 뒤에 잠금을 표시합니다:

```
mysql> SELECT ENGINE_TRANSACTION_ID as Trx_Id,
              OBJECT_NAME은 `Table`,
              INDEX_NAME은 `Index`,
              LOCK_DATA는 데이터,
              LOCK_MODE는 모드,
              LOCK_STATUS는 상태,
              LOCK_TYPE은 유형입니다.
        FROM performance_schema.data_locks;
+----------------+--------+---------+-----------+--------------+--------+----------------+
| Trx_Id | 테이블 | 인덱스 | 데이터 | 모드 | 상태 | 유형 | 부여됨 | 테이블 |
+----------------+--------+---------+-----------+--------------+--------+----------------+
| 421291106148352 | 재 | NULL | NULL | IS | 부여됨 | TABLE |
| 421291106148352 | 동물 | PRIMARY | '땅돼지' | S,REC_NOT_GAP | 부여됨 | 기록 |
| 421291106147544 | 동물 | NULL | NULL | IS | 부여됨 | TABLE |
| 421291106147544 | 동물 | PRIMARY | 'Buzzard' | S,REC_NOT_GAP | GRANTED | RECORD |
+----------------+--------+---------+-----------+--------------+--------+----------------+
4행 세트(0.00초)
```

그런 다음 클라이언트 B가 동물의 행을 업데이트합니다:

```
mysql> UPDATE Animals SET value=30 WHERE name='땅돼지';
```

클라이언트 B는 기다려야 합니다. 성능 스키마에는 잠금 대기가 표시됩니다:

```
mysql> SELECT REQUESTING_ENGINE_LOCK_ID as Req_Lock_Id,
              REQUESTING_ENGINE_TRANSACTION_ID as Req_Trx_Id,
              BLOCKING_ENGINE_LOCK_ID는 Blk_Lock_Id,
              BLOCKING_ENGINE_TRANSACTION_ID는 Blk_Trx_Id로 설정합니다.
        FROM performance_schema.data_lock_waits;
+-------------------------------------+-----------+-------------------------------------+----------
|                                  Req_Lock_Id| Req_Trx_Id | Blk_Lock_Id| Blk_Trx_I
+-------------------------------------+-----------+-------------------------------------+----------
| 139816129437696:27:4:2:139816016601240 |43260  | 139816129436888:27:4:2:139816016594720 | 421291106
+-------------------------------------+-----------+-------------------------------------+----------
1행 1세트(0.00초)

mysql> SELECT ENGINE_LOCK_ID as Lock_Id,
              엔진_트랜잭션_아이디를 Trx_id로, 오브젝
              트 이름을 `테이블`로, 인덱스 이름을 `인
              덱스`로 설정합니다,
              LOCK_DATA는 데이터,
              LOCK_MODE는 모드,
              LOCK_STATUS는 상태,
              LOCK_TYPE은 유형입니다.
        FROM performance_schema.data_locks;
+-------------------------------------+----------------+---------+---------+-----------+-------------
| Lock_Id| Trx_Id| 테이블|                 인덱스|            데이터| 모드
+-------------------------------------+----------------+---------+---------+-----------+-------------
```

```
| 139816129437696:1187:139816016603896                |43260 | 동물 |            NULL| NULL| IX
| 139816129437696:1188:139816016603808    |43260       |                새|        NULL| NULL| IS
| 139816129437696:28:4:2:139816016600896 |43260       | 새             | PRIMARY | 'Buzzard' | S,REC_NOT_GA
| 139816129437696:27:4:2:139816016601240 |43260       | 동물 | PRIMARY | '땅돼지' | X,REC_NOT_GA
| 139816129436888:1187:139816016597712|   421291106147544 | 동물 |               NULL| NULL| IS
| 139816129436888:27:4:2:139816016594720 | 421291106147544 | 동물 | PRIMARY | '땅돼지' | S,REC_NOT_GA
+----------------------------------------+-----------------+---------+---------+------------+-------------
6행 세트(0.00초)
```

InnoDB는 트랜잭션이 데이터베이스를 수정하려고 시도할 때만 순차 트랜잭션 ID를 사용합니다. 따라서 이전 읽기 전용 트랜잭션 ID는 421291106148352 에서 43260으로 변경됩니다.

클라이언트 A가 Birds에서 동시에 행을 업데이트하려고 시도하면 교착 상태가 발생합니다:

```
mysql> UPDATE Birds SET value=40 WHERE name='Buzzard;
오류 1213 (40001): 잠금을 얻으려고 할 때 교착 상태가 발견되었습니다. 트랜잭션을 다시 시작해보십시오.
```

InnoDB가 교착 상태를 일으킨 트랜잭션을 롤백합니다. 이제 클라이언트 B에서 첫 번째 업데이트를 진행할 수 있습니다.

정보 스키마에는 교착 상태의 수가 포함되어 있습니다:

```
mysql> SELECT `count` FROM INFORMATION_SCHEMA.INNODB_METRICS
          WHERE NAME="lock_deadlock";
+--------+
| 카운트 |
+--------+
       |1 |
+--------+
1행 1세트(0.00초)
```

InnoDB 상태에는 교착 상태 및 트랜잭션에 대한 다음 정보가 포함되어 있습니다. 또한 읽기 전용 트랜잭션 ID 421291106147544 가 순차 트랜잭션 ID 43261로 변경되었음을 보여줍니다.

```
mysql> SHOW ENGINE INNODB STATUS;
------------------------------
최근 감지된 교착 상태
------------------------------
2022-11-25 15:58:22 139815661168384
*** (1) 거래:
트랜잭션 43260, 활성 186초 인덱스 시작 인덱스 읽기 사용 중인
mysql 테이블 1, 잠김 1
잠금 대기 4 잠금 구조체, 힙 크기 1128, 2 행 잠금
MySQL 스레드 id 19, OS 스레드 핸들 139815619204864, 쿼리 id 143 localhost u2 업데이트 업데이트 애니멀
SET 값=30 WHERE 이름='땅돼지'                                                    잠금 모

*** (1)은 자물쇠를 고정합니다:
레코드 잠금 공간 id 28 페이지 번호 4 n 비트 72 인덱스 `test`.`Birds` 테이블의 PRIMARY trx id 43260 레코드
잠금, 힙 번호 2 물리적 레코드: n_fields 4; 압축 형식; 정보 비트 0
 0: 렌 7; 헥스 42757A7A617264; ASC 버자드;; 1:
 렌 6; 헥스 00000000A8FB; ASC           ;;                                      0 lock_
 2: LEN 7; HEX 82000000E40110; ASC        ;;
 3: LEN 4; HEX 80000014; ASC     ;;


*** (1)이 잠금이 부여되기를 기다리는 중입니다:
레코드 잠금 공간 id 27 페이지 번호 4 n 비트 72 인덱스 `test`.`Animals` 테이블의 PRIMARY trx id 4326 레코드
잠금, 힙 번호 2 물리적 레코드: n_fields 4; 압축 형식; 정보 비트 0
 0: 렌 8; 헥스 416172647661726B; asc 땅돼지;; 1:
 렌 6; 헥스 00000000A8F9; asc           ;;
 2: LEN 7; HEX 82000000E20110; ASC        ;;
 3: LEN 4; HEX 8000000A; ASC     ;;


                                                                              1 잠금
*** (2) 거래:
트랜잭션 43261, ACTIVE 209초 인덱스 시작 인덱스 읽기 사용 중인
mysql 테이블 1, 잠김 1
잠금 대기 4 잠금 구조체, 힙 크기 1128, 2 행 잠금
MySQL 스레드 id 18, OS 스레드 핸들 139815618148096, 쿼리 id 146 localhost u1 업데이트 업데이트 버즈
SET 값=40 WHERE 이름='버자드'
```

lock_mo

```
 0: 렌 7; 헥스 42757A7A617264; ASC 버자드;; 1:
 렌 6; 헥스 00000000A8FB; ASC           ;;
 2: LEN 7; HEX 82000000E40110; ASC        ;;
 3: LEN 4; HEX 80000014; ASC     ;;

*** 트랜잭션 롤백 (2)
--------------
트랜잭션
--------------
Trx ID 카운터 43262
trx의 n:o < 43256에 대해 퍼지 완료됨 실행 취소 n:o < 0 상태: 실행 중이지만 유휴 상
태 히스토리 목록 길이 0
각 세션의 트랜잭션 목록입니다:
---트랜잭션 421291106147544, 시작되지 않음
0 잠금 구조체, 힙 크기 1128, 0 행 잠금
---트랜잭션 421291106146736, 시작되지 않음
0 잠금 구조체, 힙 크기 1128, 0 행 잠금
---트랜잭션 421291106145928, 시작되지 않음
0 잠금 구조체, 힙 크기 1128, 0 행 잠금
---트랜잭션 43260, 활성 219초
구조체 잠금 4개, 힙 크기 1128, 행 잠금 2개, 로그 항목 실행 취소 1
MySQL 스레드 id 19, OS 스레드 핸들 139815619204864, 쿼리 id 143 로컬 호스트 u2
```

오류 로그에는 트랜잭션 및 잠금에 대한 이 정보가 포함되어 있습니다:

```
mysql> SELECT @@log_error;
+---------------------+
| @@log_error         |
+---------------------+
| /var/log/mysqld.log |
+---------------------+
한 세트에 1행(0.00초)

트랜잭션 43260, 활성 186초 인덱스 시작 인덱스 읽기 사용 중인 mysql
테이블 1, 잠김 1
잠금 대기 4 잠금 구조체, 힙 크기 1128, 2 행 잠금
MySQL 스레드 id 19, OS 스레드 핸들 139815619204864, 쿼리 id 143 localhost u2 업데이트 업데이트 애니
멀 SET 값=30 WHERE 이름='땅돼지'
레코드 잠금 공간 id 28 페이지 번호 4 n 비트 72 인덱스 `test`.`Birds` 테이블의 PRIMARY trx id 43260 잠금 모드 레코드 잠금
, 힙 번호 2 물리적 레코드: n_fields 4; 컴팩트 포맷; 정보 비트 0
 0: 렌 7; 헥스 42757A7A617264; ASC 버자드;; 1:
 렌 6; 헥스 00000000A8FB; ASC           ;;
 2: LEN 7; HEX 82000000E40110; ASC        ;;
 3: LEN 4; HEX 80000014; ASC     ;;

레코드 잠금 공간 id 27 페이지 번호 4 n 비트 72 인덱스 `test`.`Animals` 테이블의 PRIMARY trx id 43260 lock_mod 레코드
잠금, 힙 번호 2 물리적 레코드: n_fields 4; 압축 형식; 정보 비트 0
 0: 렌 8; 헥스 416172647661726B; asc 땅돼지;; 1:
 렌 6; 헥스 00000000A8F9; asc           ;;
 2: LEN 7; HEX 82000000E20110; ASC        ;;
 3: LEN 4; HEX 8000000A; ASC     ;;

트랜잭션 43261, ACTIVE 209초 인덱스 시작 인덱스 읽기 사용 중인
mysql 테이블 1, 잠김 1
```

```
잠금 대기 4 잠금 구조체, 힙 크기 1128, 2 행 잠금
MySQL 스레드 id 18, OS 스레드 핸들 139815618148096, 쿼리 id 146 localhost u1 업데이트 업데이트 버즈
SET 값=40 WHERE 이름='버자드'
레코드 잠금 공간 id 27 페이지 번호 4 n 비트 72 인덱스 `test`.`Animals` 테이블의 PRIMARY trx id 43261 잠금 모드 레코드 잠
금, 힙 번호 2 물리적 레코드: n_fields 4; 압축 형식; 정보 비트 0
 0: 렌 8; 헥스 416172647661726B; asc 땅돼지;; 1:
 렌 6; 헥스 00000000A8F9; asc           ;;
 2: LEN 7; HEX 82000000E20110; ASC        ;;
 3: LEN 4; HEX 8000000A; ASC     ;;

레코드 잠금 공간 id 28 페이지 번호 4 n 비트 72 인덱스 `test`.`Birds` 테이블의 PRIMARY trx id 43261 lock_mode 레코드 잠
금, 힙 번호 2 물리적 레코드: n_fields 4; 압축 형식; 정보 비트 0
 0: 렌 7; 헥스 42757A7A617264; ASC 버자드;; 1:
 렌 6; 헥스 00000000A8FB; ASC           ;;
 2: LEN 7; HEX 82000000E40110; ASC        ;;
 3: LEN 4; HEX 80000014; ASC     ;;
```

15.7.5.2 교착 상태 감지

교착 상태 감지가 활성화된 경우(기본값), InnoDB는 트랜잭션 교착 상태를 자동으로 감지하고 교착 상태를 해제하기 위해 트랜잭션을 롤백합니다. InnoDB는 롤백할 작은 트랜잭션을 선택하려고 하는데, 트랜잭션의 크기는 삽입, 업데이트 또는 삭제된 행의 수에 따라 결정됩니다.

innodb_table_locks = 1(기본값)인 경우 InnoDB는 테이블 잠금을 인식하고 자동 커밋합니다. = 0이고, 그 위에 있는 MySQL 계층이 행 수준 잠금에 대해 알고 있어야 합니다. 그렇지 않으면 MySQL LOCK TABLES 문에 의해 설정된 테이블 잠금 또는 InnoDB가 아닌 다른 스토리지 엔진에 의해 설정된 잠금이 관련된 교착 상태를 InnoDB가 감지할 수 없습니다. 이러한 상황은 innodb_lock_wait_timeout 시스템 변수의 값을 설정하여 해결합니다.

InnoDB 모니터 출력의 최신 감지된 데드락 섹션에 다음과 같은 메시지가 포함된 경우 잠금 테이블에서 너무 깊거나 긴 검색이 그래프를 기다리는 경우 롤백됩니다. 트랜잭션 다음에는 대기 목록에 있는 트랜잭션 수가 200개 제한에 도달했음을 나타냅니다. 200개를 초과하는 대기 목록은 교착 상태로 간주되며 대기 목록을 확인하려는 트랜잭션은 롤백됩니다. 잠금 스레드가 대기 목록에 있는 트랜잭션이 소유한 잠금을 1,000,000개 이상 확인해야 하는 경우에도 동일한 오류가 발생할 수 있습니다.

교착 상태를 피하기 위해 데이터베이스 작업을 구성하는 기법은 15.7.5절. "InnoDB의 교착 상태"를 참조하세요.

교착 상태 감지 비활성화

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 변수를 사용하여 비활성화할 수 있습니다.

15.7.5.3 교착 상태를 최소화하고 처리하는 방법

이 섹션에서는 15.7.5.2절 "교착 상태 감지"의 교착 상태에 대한 개념 정보를 기반으로 합니다. 교착 상태를 최소화하기 위해 데이터베이스 작업을 구성하는 방법과 애플리케이션에 필요한 후속 오류 처리에 대해 설명합니다.

교착 상태는 트랜잭션 데이터베이스의 전형적인 문제이지만, 특정 트랜잭션을 전혀 실행할 수 없을 정도로 빈번하지 않는 한 위험하지는 않습니다. 일반적으로 애플리케이션을 사용하여 교착 상태로 인해 트랜잭션이 롤백되는 경우 항상 트랜잭션을 다시 발급할 수 있도록 준비합니다.

InnoDB는 자동 행 수준 잠금을 사용합니다. 하나의 행만 삽입하거나 삭제하는 트랜잭션의 경우에도 교착 상태가 발생할 수 있습니다. 이러한 작업은 실제로 "원자적"인 것이 아니라 삽입 또는 삭제된 행의 인덱스 레코드(여러 개일 수도 있음)에 자동으로 잠금을 설정하기 때문입니다.

다음 기술을 사용하여 교착 상태에 대처하고 교착 상태의 발생 가능성을 줄일 수 있습니다:

- 언제든지 엔진 `INNODB` 상태 표시를 실행하여 가장 최근 교착 상태의 원인을 파악하세요. 이를 통해 교착 상태를 피하기 위해 애플리케이션을 조정하는 데 도움이 될 수 있습니다.
- 교착 상태 경고가 자주 발생하여 우려되는 경우 `innodb_print_all_deadlocks` 변수를 활성화하여 보다 광범위한 디버깅 정보를 수집하세요. 최신 교착 상태뿐만 아니라 각 교착 상태에 대한 정보가 MySQL 오류 로그에 기록됩니다. 디버깅이 끝나면 이 옵션을 비활성화하세요.
- 교착 상태로 인해 트랜잭션이 실패할 경우 항상 트랜잭션을 다시 발급할 준비를 하시기 바랍니다. 교착 상태는 위험하지 않습니다. 다시 시도하면 됩니다.
- 트랜잭션을 작고 짧은 기간으로 유지하여 충돌이 덜 일어나도록 하세요.

- 트랜잭션이 충돌할 가능성을 줄이려면 관련 변경을 수행한 후 즉시 트랜잭션을 커밋하세요. 특히 커밋되지 않은 트랜잭션이 있는 대화형 `mysql` 세션을 장시간 열어 두지 마세요.
- 읽기 잠금(`SELECT ... FOR UPDATE` 또는 `SELECT ... FOR SHARE`)을 사용하는 경우 읽기 커밋됨과 같이 더 낮은 격리 수준을 사용해보십시오.
- 트랜잭션 내에서 여러 테이블을 수정하거나 동일한 테이블의 다른 행 집합을 수정할 때는 매번 일관된 순서로 해당 작업을 수행하세요. 그러면 트랜잭션이 잘 정의된 대기열을 형성하고 교착 상태가 발생하지 않습니다. 예를 들어, 데이터베이스 작업을 애플리케이션 내에서 함수로 구성하거나 저장된 루틴을 호출하여 서로 다른 위치에서 여러 개의 유사한 `INSERT`, `UPDATE`, `DELETE` 문을 코딩하는 대신 저장된 루틴을 호출하세요.
- 쿼리가 더 적은 수의 인덱스 레코드를 스캔하고 더 적은 수의 잠금을 설정하도록 테이블에 잘 선택된 인덱스를 추가하세요. `EXPLAIN SELECT`를 사용하여 MySQL 서버가 쿼리에 가장 적합한 것으로 간주하는 인덱스를 결정하세요.
- 잠금을 적게 사용합니다. `SELECT`가 이전 스냅샷의 데이터를 반환하도록 허용할 수 있는 경우, `FOR UPDATE` 또는 `FOR SHARE` 절을 추가하지 마세요. 동일한 트랜잭션 내의 각 일관된 읽기는 자체의 새로운 스냅샷에서 읽으므로 여기서는 `READ COMMITTED` 격리 수준을 사용하는 것이 좋습니다.
- 다른 방법이 없다면 테이블 수준 잠금으로 트랜잭션을 직렬화하세요. 올바른 사용 방법 `InnoDB` 테이블과 같은 트랜잭션 테이블을 사용하는 테이블을 잠그는 방법은 `SET autocommit = 0`(`START TRANSACTION`이 아님)으로 트랜잭션을 시작한 다음 테이블을 잠그고, 트랜잭션을 명시적으로 커밋할 때까지 테이블 잠금 해제를 호출하지 않는 것입니다. 예를 들어 테이블 `t1`에 쓰고 테이블 `t2`에서 읽어야 하는 경우 이렇게 할 수 있습니다:

  ```
  SET 자동 커밋=0;
  잠금 테이블 t1 쓰기, t2 읽기, ...;
  ... 여기서 테이블 T1 및 T2로 작업을 수행합니다 ...
  커밋;
  테이블 잠금 해제;
  ```

  테이블 수준 잠금은 테이블에 대한 동시 업데이트를 방지하여 바쁜 시스템에서 응답성이 저하되는 대신 교착 상태를 방지합니다.
- 트랜잭션을 직렬화하는 또 다른 방법은 단일 행만 포함하는 보조 '세마포어' 테이블을 만드는 것입니다. 각 트랜잭션이 다른 테이블에 액세스하기 전에 해당 행을 업데이트하도록 합니다. 이렇게 하면 모든 트랜잭션이 직렬 방식으로 발생합니다. 직렬화 잠금이 행 수준 잠금이기 때문에 `InnoDB`의 즉각적인 교착 상태 감지 알고리즘도 이 경우에도 작동한다는 점에 유의하세요. MySQL 테이블 수준 잠금을 사용하면 교착 상태를 해결하기 위해 시간 초과 방법을 사용해야 합니다.

15.7.6 트랜잭션 예약

`InnoDB`는 컨텐트 인식 트랜잭션 스케줄링(CATS) 알고리즘을 사용하여 잠금을 대기 중인 트랜잭션의 우선순

위를 정합니다. 여러 트랜잭션이 동일한 객체에 대한 잠금을 기다리는 경우, CATS 알고리즘은 어떤 트랜잭션이 먼저 잠금을 받을지 결정합니다.

CATS 알고리즘은 트랜잭션이 차단하는 트랜잭션 수에 따라 계산된 스케줄링 가중치를 할당하여 대기 중인 트랜잭션의 우선순위를 정합니다. 예를 들어 두 개의 트랜잭션이 동일한 객체에 대한 잠금을 대기 중인 경우 가장 많은 트랜잭션을 차단하는 트랜잭션에 더 큰 스케줄링 가중치가 할당됩니다. 가중치가 같으면 가장 오래 대기 중인 트랜잭션에 우선순위가 부여됩니다.

정보 스키마 `INNODB_TRX` 테이블의 `TRX_SCHEDULE_WEIGHT` 열을 쿼리하여 트랜잭션 스케줄링 가중치를 확인할 수 있습니다. 가중치는 대기 중인 트랜잭션에 대해서만 계산됩니다. 대기 중인 트랜잭션은 `TRX_STATE` 열에 보고된 대로 `LOCK WAIT` 트랜잭션 실행 상태의 트랜잭션입니다. 잠금을 기다리는 중이 아닌 트랜잭션은 NULL `TRX_SCHEDULE_WEIGHT` 값을 보고합니다.

`INNODB_METRICS` 카운터는 코드 레벨 트랜잭션 스케줄링 이벤트의 모니터링을 위해 제공됩니다. `INNODB_METRICS` 카운터 사용에 대한 자세한 내용은 15.15.6절, "InnoDB 정보_SCHEMA 메트릭 테이블"을 참조하십시오.

- 잠금 해제 시도

  레코드 잠금을 해제하려는 시도 횟수입니다. 단일 구조에 0개 이상의 레코드 잠금이 있을 수 있으므로 한 번의 시도로 0개 이상의 레코드 잠금이 해제될 수 있습니다.

- LOCK_RE_C_GRANT_ATTEMPS

  레코드 잠금을 부여하려는 시도 횟수입니다. 한 번의 시도로 0개 이상의 레코드 잠금이 부여될 수 있습니다.

- LOCK_SCHEDULE_REFRESH

  예약된 트랜잭션 가중치를 업데이트하기 위해 대기 그래프를 분석한 횟수입니다.

15.8 InnoDB 구성

이 섹션에서는 InnoDB 초기화, 시작 및 InnoDB 스토리지 엔진의 다양한 구성 요소와 기능에 대한 구성 정보 및 절차를 제공합니다. InnoDB 테이블에 대한 데이터베이스 작업 최적화에 대한 자세한 내용은 섹션 8.5, "InnoDB 테이블에 대한 최적화"를 참조하십시오.

15.8.1 InnoDB 시작 구성

InnoDB 구성과 관련하여 가장 먼저 결정해야 할 사항은 데이터 파일, 로그 파일, 페이지 크기 및 메모리 버퍼의 구성이며, 이 구성은 InnoDB를 초기화하기 전에 구성해야 합니다. InnoDB가 초기화된 후 구성을 수정하려면 간단한 절차가 필요할 수 있습니다.

이 섹션에서는 구성 파일에서 InnoDB 설정을 지정하는 방법에 대한 정보를 제공합니다.
InnoDB 초기화 정보 및 중요한 스토리지 고려 사항.

- MySQL 옵션 파일에서 옵션 지정하기
- InnoDB 초기화 정보 보기
- 중요한 스토리지 고려 사항
- 시스템 테이블스페이스 데이터 파일 구성
- InnoDB 이중 쓰기 버퍼 파일 구성
- 로그 구성 재실행
- 테이블 공간 구성 실행 취소
- 글로벌 임시 테이블 스페이스 구성
- 세션 임시 테이블 스페이스 구성
- 페이지 크기 구성

- 메모리 구성

MySQL 옵션 파일에서 옵션 지정하기

MySQL은 데이터 파일, 로그 파일 및 페이지 크기 설정을 사용하여 InnoDB를 초기화하기 때문에 InnoDB를 초기화하기 전에 MySQL이 시작할 때 읽는 옵션 파일에서 이러한 설정을 정의하는 것이 좋습니다. 일반적으로 MySQL 서버가 처음 시작될 때 InnoDB가 초기화됩니다.

서버가 시작될 때 읽는 모든 옵션 파일의 [mysqld] 그룹에 InnoDB 옵션을 배치할 수 있습니다. MySQL 옵션 파일의 위치는 4.2.2.2절 "옵션 파일 사용하기"에 설명되어 있습니다.

mysqld가 특정 파일(및 mysqld-auto.cnf)에서만 옵션을 읽도록 하려면
서버를 시작할 때 명령줄의 첫 번째 옵션으로 --defults-file 옵션을 사용합니다:

```
mysqld --defults-file = 경로_to_옵션_파일
```

InnoDB 초기화 정보 보기

시작 중에 InnoDB 초기화 정보를 보려면 명령 프롬프트에서 mysqld를 시작하면 초기화 정보가 콘솔에 인쇄됩니다.

예를 들어, Windows의 경우 mysqld가 C:\Program Files\MySQL\MySQL Server 8.2\bin에 있는 경우 다음과 같이 MySQL 서버를 시작합니다:

```
C:\> "C:\Program Files\MySQL\MySQL Server 8.2\bin\mysqld" --console
```

유닉스 계열 시스템에서 mysqld는 MySQL 설치의 bin 디렉터리에 있습니다:

```
$> bin/mysqld --user=mysql &
```

서버 출력을 콘솔로 보내지 않는 경우, 시작 후 오류 로그를 확인하여 시작 프로세스 중에 인쇄된 초기화 정보 InnoDB를 확인하세요.

다른 방법을 사용하여 MySQL을 시작하는 방법에 대한 자세한 내용은 2.9.5절. '자동으로 MySQL 시작 및 중지'를 참조하세요.

참고

InnoDB는 시작 시 모든 사용자 테이블 및 관련 데이터 파일을 열지 않습니다. 그러나 InnoDB는 데이터 사전에서 참조된 테이블스페이스 파일이 있는지 확인합니다. 테이블스페이스 파일을 찾을 수 없는 경우 InnoDB는 오류를 기록하고 시작 시퀀스를 계속 진행합니다. 재실행 로그에 참조된 테이블스페이스 파일은 애플리케이션 재실행을 위해 충돌 복구 중에 열릴 수 있습니다.

중요한 스토리지 고려 사항

시작 구성을 진행하기 전에 다음 스토리지 관련 고려 사항을 검토하세요.

- 경우에 따라 데이터 및 로그 파일을 별도의 물리적 디스크에 배치하여 데이터베이스 성능을 향상시킬 수 있습니다. InnoDB 데이터 파일에 원시 디스크 파티션(원시 장치)을 사용하여 I/O 속도를 높일 수도 있습니다. 시스템 테이블 스페이스에 원시 디스크 파티션 사용을 참조하세요.

- InnoDB는 사용자 데이터를 보호하기 위해 커밋, 롤백 및 충돌 복구 기능을 갖춘 트랜잭션 안전(ACID 준수) 스토리지 엔진입니다. **그러나** 기본 운영 체제 또는 하드웨어가 광고된 대로 작동하지 않는 경우에는 **그렇게 할 수 없습니다.** 많은 운영 체제 또는 디스크 하위 시스템은 성능 향상을 위해 쓰기 작업을 지연하거나 순서를 변경할 수 있습니다. 일부 운영 체제에서는 파일에 대해 쓰여지지 않은 데이터가 모두 플러시될 때까지 기다려야 하는 fsync() 시스템 호출이 실제로 데이터가 안정적인 스토리지로 플러시되기 전에 반환될 수 있습니다. 이 때문에 운영 체제 충돌이나 정전으로 인해 최근에 커밋한 데이터가 파괴되거나 최악의 경우 손상될 수 있습니다.

쓰기 작업이 재순서화되었기 때문입니다. 데이터 무결성이 중요한 경우 프로덕션 환경에서 사용하기 전에 "풀 더 플러그" 테스트를 수행하세요. macOS에서 `InnoDB`는 특수한 `fcntl()` 파일 플러시 방법을 사용합니다. Linux에서는 **쓰기 백 캐시를 비활성화하는 것이** 좋습니다.

ATA/SATA 디스크 드라이브의 경우, `hdparm -W0 /dev/hda`와 같은 명령으로 쓰기 백 캐시를 비활성화할 수 있습니다. **일부 드라이브 또는 디스크 컨트롤러는 쓰기-백 캐시를 비활성화하지 못할 수 있습니다.**

- 사용자 데이터를 보호하는 `InnoDB` 복구 기능과 관련하여 `InnoDB`는 기본적으로 활성화된 (`innodb_doublewrite=ON`) 이중 쓰기 버퍼라는 구조와 관련된 파일 플러시 기술을 사용합니다. 이중 쓰기 버퍼는 예기치 않은 종료 또는 정전 후 복구에 안전성을 더하고, `fsync()` 작업의 필요성을 줄여 대부분의 Unix에서 성능을 개선합니다. 데이터 무결성 또는 장애 발생 가능성을 염려하는 경우 `innodb_doublewrite` 옵션을 활성화된 상태로 유지하는 것이 좋습니다. 이중 쓰기 버퍼에 대한 자세한 내용은 섹션 15.11.1, "InnoDB 디스크 I/O"를 참조하십시오.
- `InnoDB`와 함께 NFS를 사용하기 전에 MySQL과 함께 NFS 사용에 설명된 잠재적인 문제를 검토하세요.

시스템 테이블스페이스 데이터 파일 구성

`innodb_data_file_path` 옵션은 `InnoDB` 시스템 테이블 스페이스 데이터 파일의 이름, 크기 및 속성을 정의합니다. MySQL 서버를 초기화하기 전에 이 옵션을 구성하지 않으면 기본 동작은 12MB보다 약간 큰 단일 자동 확장 데이터 파일(`ibdata1`)을 생성하는 것입니다:

```
mysql> SHOW 변수에 'innodb_data_file_path' 같은 변수를 입력합니다;
+----------------------+--------------------------+
|                      변수_이름| 값                       |
+----------------------+--------------------------+
| innodb_data_파일_경로 | ibdata1:12M:자동 확장 |
+----------------------+--------------------------+
```

전체 데이터 파일 사양 구문에는 파일 이름, 파일 크기, 자동 확장 속성 및 최대 속성입니다:

```
file_name:file_size[:autoextend[:max:max_file_size]]
```

파일 크기는 크기 값에 `K`, `M` 또는 `G`를 추가하여 킬로바이트, 메가바이트 또는 기가바이트로 지정합니다. 데이터 파일 크기를 킬로바이트 단위로 지정하는 경우 1024의 배수로 지정합니다. 그렇지 않으면 킬로바이트 값은 가장 가까운 메가바이트(MB) 경계로 반올림됩니다. 파일 크기의 합계는 최소한 12MB보다 약간 커야 합니다.

세미콜론으로 구분된 목록을 사용하여 둘 이상의 데이터 파일을 지정할 수 있습니다. 예를 들면 다음과 같습니다:

```
[mysqld]
innodb_data_file_path=ibdata1:50M;ibdata2:50M:autoextend
```

자동 확장 및 최대 속성은 마지막에 지정한 데이터 파일에만 사용할 수 있습니다.

자동 확장 속성을 지정하면 필요한 공간에 따라 데이터 파일의 크기가 64MB 단위로 자동으로 증가합니다. `innodb_autoextend_increment` 변수는 증분 크기를 제어합니다.

자동 확장 데이터 파일의 최대 크기를 지정하려면 자동 확장 속성 뒤에 최대 속성을 사용합니다. 최대 속성은 디스크 사용량을 제한하는 것이 매우 중요한 경우에만 사용하십시오. 다음 구성에서는 `ibdata1`을 500MB 한도까지 늘릴 수 있습니다:

```
[mysqld] innodb_data_파일_경로=ibdata1:12M:자동 확장:최대:500M
```

이중 쓰기 버퍼 페이지를 위한 충분한 공간을 확보하기 위해 *첫 번째* 시스템 테이블스페이스 데이터 파일에 최소 파일 크기가 적용됩니다. 다음 표는 각 `InnoDB` 페이지 크기에 대한 최소 파일 크기를 보여줍니다. 기본 `InnoDB` 페이지 크기는 16384(16KB)입니다.

페이지 크기(innodb_page_size)	최소 파일 크기
16384 (16KB) 이하	3MB
32768 (32KB)	6MB
65536 (64KB)	12MB

디스크가 꽉 차면 다른 디스크에 데이터 파일을 추가할 수 있습니다. 자세한 내용은 시스템 테이블 공간 크

기 조정을 참조하세요.

개별 파일의 크기 제한은 운영 체제에 따라 결정됩니다. 대용량 파일을 지원하는 운영 체제에서는 파일 크기를 4GB 이상으로 설정할 수 있습니다. 원시 디스크 파티션을 데이터 파일로 사용할 수도 있습니다. 시스템 테이블 공간에 원시 디스크 파티션 사용을 참조하세요.

InnoDB는 파일 시스템 최대 파일 크기를 인식하지 못하므로 최대 파일 크기가 2GB와 같이 작은 값인 파일 시스템에서는 주의해야 합니다.

시스템 테이블스페이스 파일은 기본적으로 데이터 디렉터리(datadir)에 생성됩니다. 다른 위치를 지정하려면 innodb_data_home_dir 옵션을 사용합니다. 예를 들어, myibdata라는 디렉터리에 시스템 테이블스페이스 데이터 파일을 생성하려면 이 구성을 사용합니다:

```
[mysqld]
innodb_data_home_dir = /myibdata/
innodb_data_file_path=ibdata1:50M:autoextend
```

innodb_data_home_dir의 값을 지정할 때는 후행 슬래시가 필요합니다. InnoDB는 디렉터리를 생성하지 않으므로 서버를 시작하기 전에 지정한 디렉터리가 존재하는지 확인하세요. 또한 MySQL 서버에 디렉터리에 파일을 생성할 수 있는 적절한 액세스 권한이 있는지 확인하세요.

InnoDB는 각 데이터 파일의 디렉터리 경로를 데이터 파일 이름에 innodb_data_home_dir의 값을 텍스트로 연결하여 형성합니다. innodb_data_home_dir이 정의되지 않은 경우 기본값은 데이터 디렉터리인 "./"입니다. (MySQL 서버는 실행을 시작할 때 현재 작업 디렉터리를 데이터 디렉터리로 변경합니다.)

또는 시스템 테이블스페이스 데이터 파일의 절대 경로를 지정할 수도 있습니다. 다음 구성은 앞의 구성과 동일합니다:

```
[mysqld]
innodb_data_file_path=/myibdata/ibdata1:50M:autoextend
```

innodb_data_file_path에 절대 경로를 지정하는 경우, 이 설정은 innodb_data_home_dir 설정과 연결되지 않습니다. 시스템 테이블스페이스 파일은 지정된 절대 경로에 생성됩니다. 서버를 시작하기 전에 지정된 디렉터리가 존재해야 합니다.

InnoDB 이중 쓰기 버퍼 파일 구성

InnoDB 이중쓰기버퍼 저장 영역은 이중 쓰기 파일에 상주하므로 이중 쓰기 페이지의 저장 위치와 관련하여 유연성을 제공합니다. 이전 릴리스에서는 이중 쓰기 버퍼가
저장 영역이 시스템 테이블 스페이스에 상주합니다. innodb_doublewrite_dir 변수는 InnoDB가 시작할 때 이중 쓰기 파일을 생성하는 디렉터리를 정의합니다. 디렉터리가 지정되지 않은 경우 기본적으로 데이터 디렉터리가 지정되는 innodb_data_home_dir 디렉터리에 이중 쓰기 파일이 생성됩니다.

이중 쓰기 파일을 innodb_data_home_dir 디렉터리가 아닌 다른 위치에 생성하려면 innodb_doublewrite_dir 변수를 구성합니다. 예를 들어

```
innodb_doublewrite_dir=/path/to/doublewrite_directory
```

다른 이중 쓰기 버퍼 변수를 사용하여 이중 쓰기 파일 수, 스레드당 페이지 수 및 이중 쓰기 배치 크기를 정의할 수 있습니다. 이중 쓰기 버퍼 구성에 대한 자세한 내용은 섹션 15.6.4, "이중 쓰기 버퍼"를 참조하십시오.

로그 구성 재실행

재실행 로그 파일이 차지하는 디스크 공간의 양은 시작 또는 런타임에 설정할 수 있는

`innodb_redo_log_capacity` 변수에 의해 제어되며, 예를 들어 옵션 파일에서 이 변수를 8GB로 설정하려면 다음 항목을 추가합니다:

```
[mysqld]
innodb_redo_log_capacity = 8589934592
```

런타임에 재실행 로그 용량을 구성하는 방법에 대한 자세한 내용은 재실행 로그 용량 구성을 참조하세요.

`innodb_redo_log_capacity` 변수는 더 이상 사용되지 않는 `innodb_log_file_size` 및 `innodb_log_files_in_group` 변수를 대체합니다. `innodb_redo_log_capacity` 설정이 정의된 경우 `innodb_log_file_size` 및 `innodb_log_files_in_group` 설정은 무시되며, 그렇지 않은 경우 이 설정은 `innodb_redo_log_capacity` 설정 (`innodb_log_files_in_group` *.

innodb_log_file_size = innodb_redo_log_capacity). 이러한 변수가 설정되어 있지 않으면 innodb_redo_log_capacity가 기본값인 104857600 바이트(100MB)로 설정됩니다. 최대 설정은 128GB입니다.

InnoDB는 32개의 재실행 로그 파일을 유지하려고 시도하며, 각 파일은 1/32 * innodb_redo_log_capacity와 동일합니다. 재실행 로그 파일은 innodb_log_group_home_dir 변수에 의해 다른 디렉터리가 지정되지 않는 한 데이터 디렉터리의 #innodb_redo 디렉터리에 있습니다. innodb_log_group_home_dir이 정의된 경우, 재실행 로그 파일은 해당 디렉터리의 #innodb_redo 디렉터리에 있습니다. 자세한 내용은 섹션 15.6.5, "재실행 로그"를 참조하세요.

MySQL 서버 인스턴스를 초기화할 때 innodb_log_files_in_group 및 innodb_log_file_size 변수를 구성하여 다른 재실행 로그 파일 수와 다른 재실행 로그 파일 크기를 정의할 수 있습니다.

innodb_log_files_in_group은 로그 그룹에 있는 로그 파일 수를 정의합니다. 기본값 및 권장값은 2입니다.

innodb_log_file_size는 로그 그룹에 있는 각 로그 파일의 크기(바이트)를 정의합니다. 결합된 로그 파일 크기(innodb_log_file_size * innodb_log_files_in_group)는 512GB보다 약간 작은 최대값을 초과할 수 없습니다. 예를 들어 255GB 로그 파일 쌍은 한도에 근접하지만 한도를 초과하지는 않습니다. 기본 로그 파일 크기는 48MB입니다. 일반적으로 로그 파일의 합산 크기는 서버가 워크로드 활동의 최고점과 최저점을 부드럽게 처리할 수 있을 만큼 충분히 커야 하며, 이는 종종 1시간 이상의 쓰기 활동을 처리할 수 있는 충분한 재실행 로그 공간이 있다는 것을 의미합니다. A
로그 파일 크기가 클수록 버퍼 풀에서 체크포인트 플러시 활동이 줄어들어 디스크 I/O가 감소합니다. 자세한 내용은 섹션 8.5.4, "InnoDB 재실행 로깅 최적화"를 참조하세요.

innodb_log_group_home_dir은 InnoDB 로그 파일의 디렉터리 경로를 정의합니다. 예를 들어, 이 옵션을 사용하면 잠재적인 I/O 리소스 충돌을 피하기 위해 InnoDB 재실행 로그 파일을 InnoDB 데이터 파일과 다른 물리적 스토리지 위치에 배치할 수 있습니다:

```
[mysqld]
innodb_log_group_home_dir = /dr3/iblogs
```

참고

InnoDB는 디렉터리를 생성하지 않으므로 서버를 시작하기 전에 로그 디렉터리가 있는지 확인해야 합니다. 필요한 디렉터리가 있으면 Unix 또는 DOS mkdir 명령을 사용하여 생성합니다.

MySQL 서버에 로그 디렉터리에 파일을 만들 수 있는 적절한 액세스 권한이 있는지 확인합니다. 일반적으로 서버는 파일을 생성해야 하는 모든 디렉터리에 대한 액세스 권한이 있어야 합니다.

테이블 공간 구성 실행 취소

실행 취소 로그는 기본적으로 MySQL 인스턴스가 초기화될 때 생성된 두 개의 실행 취소 테이블스페이스에 저장됩니다.

`innodb_undo_directory` 변수는 `InnoDB가` 기본 실행 취소 테이블스페이스를 생성하는 경로를 정의합니다. 이 변수가 정의되지 않은 경우 기본 실행 취소 테이블스페이스가 데이터 디렉터리에 생성됩니다. `innodb_undo_directory` 변수는 동적이지 않습니다. 이 변수를 구성하려면 서버를 다시 시작해야 합니다.

실행 취소 로그의 I/O 패턴으로 인해 실행 취소 테이블스페이스는 SSD 스토리지에 적합한 후보입니다.

추가 실행 취소 테이블 스페이스 구성에 대한 자세한 내용은 섹션 15.6.3.4, "테이블 스페이스 실행 취소"를 참조하십시오.

글로벌 임시 테이블 스페이스 구성

글로벌 임시 테이블 스페이스에는 사용자가 만든 임시 테이블의 변경 사항에 대한 롤백 세그먼트가 저장됩니다.

ibtmp1이라는 이름의 단일 자동 확장 글로벌 임시 테이블스페이스 데이터 파일입니다.
디렉터리에 기본적으로 저장됩니다. 초기 파일 크기는 12MB보다 약간 큽니다.

innodb_temp_data_file_path 옵션은 글로벌 임시 테이블스페이스 데이터 파일의 경로, 파일 이름 및 파일 크기를 지정합니다. 파일 크기는 크기 값에 K, M 또는 G를 추가하여 KB, MB 또는 GB 단위로 지정합니다. 파일 크기 또는 합산 파일 크기는 12MB보다 약간 커야 합니다.

글로벌 임시 테이블 스페이스 데이터 파일의 대체 위치를 지정하려면
시작 시 innodb_temp_data_file_path 옵션을 추가합니다.

세션 임시 테이블 스페이스 구성

MySQL 8.2에서는 InnoDB가항상 내부 임시 테이블의 온디스크 스토리지 엔진으로 사용됩니다.

innodb_temp_tablespaces_dir 변수는 InnoDB가 세션 임시 테이블스페이스를 생성하는 위치를 정의합니다. 기본 위치는 데이터 디렉터리의 #innodb_temp 디렉터리입니다.

세션 임시 테이블스페이스의 대체 위치를 지정하려면 시작할 때 innodb_temp_tablespaces_dir 변수를 구성합니다. 정규화된 경로 또는 데이터 디렉터리를 기준으로 한 경로가 허용됩니다.

페이지 크기 구성

innodb_page_size 옵션은 MySQL 인스턴스의 모든 InnoDB 테이블 스페이스에 대한 페이지 크기를 지정합니다. 이 값은 인스턴스를 생성할 때 설정되며 이후에도 일정하게 유지됩니다. 유효한 값은 64KB, 32KB, 16KB(기본값), 8KB 및 4KB입니다. 또는 페이지 크기를 바이트 단위로 지정할 수 있습니다(65536, 32768, 16384, 8192, 4096).

기본 16KB 페이지 크기는 광범위한 워크로드, 특히 테이블 스캔이 포함된 쿼리 및 대량 업데이트가 포함된 DML 작업에 적합합니다. 단일 페이지에 많은 행이 포함되어 경합이 문제가 될 수 있는 소규모 쓰기가 많은 OLTP 워크로드에는 더 작은 페이지 크기가 더 효율적일 수 있습니다. 일반적으로 작은 블록 크기를 사용하는 SSD 저장 장치에서도 페이지 크기가 작을수록 더 효율적일 수 있습니다. InnoDB 페이지 크기를 저장 장치 블록 크기에 가깝게 유지하면 디스크에 다시 쓰여지는 변경되지 않은 데이터의 양을 최소화할 수 있습니다.

메모리 구성

MySQL은 데이터베이스 작업의 성능을 향상시키기 위해 다양한 캐시 및 버퍼에 메모리를 할당합니다. InnoDB에 메모리를 할당할 때는 항상 운영 체제에서 요구하는 메모리, 다른 애플리케이션에 할당된 메모리, 기타 MySQL 버퍼 및 캐시에 할당된 메모리를 고려해야 합니다. 예를 들어 MyISAM 테이블을 사용하는 경우 키 버퍼에 할당된 메모리 양(key_buffer_size)을 고려하세요. MySQL 버퍼 및 캐시에 대한 개요는 섹션 8.12.3.1, "MySQL이 메모리를 사용하는 방법"을 참조하세요.

InnoDB 전용 버퍼는 다음 매개변수를 사용하여 구성합니다:

- `innodb_buffer_pool_size`는 `InnoDB` 테이블, 인덱스 및 기타 보조 버퍼에 대한 캐시된 데이터를 저장하는 메모리 영역인 버퍼 풀의 크기를 정의합니다. 버퍼 풀의 크기는 시스템 성능에 중요하며 일반적으로 다음과 같이 권장됩니다.

  `innodb_buffer_pool_size`는 시스템 메모리의 50~75%로 구성됩니다. 기본 버퍼 풀 크기는 128MB입니다. 추가 지침은 섹션 8.12.3.1, "MySQL이 메모리를 사용하는 방법"을 참조하십시오. `InnoDB` 버퍼 풀 크기를 구성하는 방법에 대한 자세한 내용은 섹션 15.8.3.1, "InnoDB 버퍼 풀 크기 구성"을 참조하십시오. 버퍼 풀 크기는 시작 시 또는 동적으로 구성할 수 있습니다.

  메모리가 많은 시스템에서는 버퍼 풀을 여러 버퍼 풀 인스턴스로 분할하여 동시성을 향상시킬 수 있습니다. 버퍼 풀 인스턴스 수는 by `innodb_buffer_pool_instances` 옵션으로 제어됩니다. 기본적으로 `InnoDB`는 하나의 버퍼 풀 인스턴스를 생성합니다. 버퍼 풀 인스턴스 수는 시작 시 구성할 수 있습니다. 자세한 내용은 섹션 15.8.3.2, "여러 버퍼 풀 인스턴스 구성"을 참조하세요.

- `innodb_log_buffer_size`는 `InnoDB`가 디스크의 로그 파일에 기록하는 데 사용하는 버퍼의 크기를 정의합니다. 기본 크기는 16MB입니다. 로그 버퍼가 크면 대용량 트랜잭션을 실행할 수 있습니다.
  를 사용하면 트랜잭션이 커밋되기 전에 로그를 디스크에 쓰지 않아도 됩니다. 많은 행을 업데이트, 삽입 또는 삭제하는 트랜잭션이 있는 경우 디스크 I/O를 절약하기 위해 로그 버퍼의 크기를 늘리는 것을 고려할 수 있습니다. `innodb_log_buffer_size`는 시작 시 구성할 수 있습니다. 관련 정보는 섹션 8.5.4, "InnoDB 재실행 로깅 최적화"를 참조하십시오.

경고

32비트 GNU/Linux x86에서 메모리 사용량을 너무 높게 설정하면 `glibc`가 프로세스 힙이 스레드 스택을 초과하여 커지도록 허용하여 서버 장애를 일으킬 수 있습니다. 글로벌 및 스레드별 버퍼와 캐시를 위해 `mysqld` 프로세스에 할당된 메모리가 2GB에 가깝거나 초과하는 경우 위험합니다.

MySQL의 전역 및 스레드당 메모리 할당을 계산하는 다음과 유사한 공식을 사용하여 MySQL 메모리 사용량을 추정할 수 있습니다. MySQL 버전 및 구성에 따라 버퍼 및 캐시를 고려하여 공식을 수정해야 할 수도 있습니다. MySQL 버퍼 및 캐시에 대한 개요는 섹션 8.12.3.1, 'MySQL에서 메모리를 사용하는 방법'을 참조하세요.

```
innodb_buffer_pool_size
+ 키_버퍼_크기
+ max_connections*(sort_buffer_size+read_buffer_size+binlog_cache_size)
+ 최대 연결 수*2MB
```

각 스레드는 스택(보통 2MB이지만 오라클에서 제공하는 MySQL 바이너리에서는 256KB에 불과합니다.)을 사용하며 최악의 경우 `sort_buffer_size + read_buffer_size` 추가 메모리도 사용합니다.

Linux에서 커널이 대용량 페이지 지원을 사용하도록 설정된 경우 `InnoDB`는 대용량 페이지를 사용하여 버퍼 풀에 메모리를 할당할 수 있습니다. 섹션 8.12.3.3, "대용량 페이지 지원 사용"을 참조하십시오.

15.8.2 읽기 전용 작업을 위한 InnoDB 구성

MySQL 데이터 디렉터리가 읽기 전용 미디어에 있는 `InnoDB` 테이블을 쿼리할 수 있습니다.
서버 시작 시 `--innodb-read-only` 구성 옵션을 설정합니다.

사용 방법

읽기 전용 작업을 위해 인스턴스를 준비하려면 읽기 전용 매체에 저장하기 전에 필요한 모든 정보가 데이터 파일에 플러시되었는지 확인하세요. 변경 버퍼링을 비활성화한 상태로 서버를 실행하고 (`innodb_change_buffering=0`) 느리게 종료합니다.

전체 MySQL 인스턴스에 대해 읽기 전용 모드를 사용하려면 서버를 시작할 때 다음 구성 옵션을 지정하세요:

- `--innodb-read-only=1`
- 인스턴스가 DVD 또는 CD와 같은 읽기 전용 미디어에 있거나 `/var` 디렉터리를 모두가 쓸 수 없는 경우: `--pid-file = 경로_온_쓰기 가능한_미디어` 및 `--event-scheduler = 사용 안 함`
- `--innodb-temp-data-file-path`. 이 옵션은 `InnoDB` 임시 테이블스페이스 데이터 파일의 경로, 파일 이름 및 파일 크기를 지정합니다. 기본 설정은 `ibtmp1:12M:autoextend`로, 데이터 디렉터리에 `ibtmp1` 임시 테이블스페이스 데이터 파일을 생성합니다. 읽기 전용 작업을 위한 인스턴스를 준비하려면 `innodb_temp_data_file_path`를 데이터 디렉터리 외부의 위치로 설정합니다. 경로는 데이터 디렉터리에 상대적이어야 합니다. 예를 들어

```
--innodb-temp-data-file-path=../../../tmp/ibtmp1:12M:autoextend
```

`innodb_read_only`를 활성화하면 모든 스토리지 엔진에 대한 테이블 생성 및 삭제 작업이 방지됩니다. 이러한 작업은 `mysql` 시스템 데이터베이스의 데이터 사전 테이블을 수정하지만 이러한 테이블은

InnoDB 스토리지 엔진을 사용하며 innodb_read_only가 활성화된 경우 수정할 수 없습니다. 데이터 사전 테이블을 수정하는 모든 작업(예: 분석 테이블 및 변경 테이블 *tbl_name* 엔진=*엔진_이름*)에도 동일한 제한이 적용됩니다.

또한 MySQL 시스템 데이터베이스의 다른 테이블은 MySQL의 InnoDB 스토리지 엔진을 사용합니다. 8.2. 이러한 테이블을 읽기 전용으로 설정하면 테이블을 수정하는 작업에 제한이 발생합니다. 예를 들어, 읽기 전용 모드에서는 사용자 생성, 부여, 취소 및 플러그인 설치 작업이 허용되지 않습니다.

사용 시나리오

이 작동 모드는 다음과 같은 상황에서 적합합니다:

- DVD 또는 CD와 같은 읽기 전용 저장 매체에 MySQL 애플리케이션 또는 MySQL 데이터 집합을 배포합니다.
- 일반적으로 데이터 웨어하우징 구성에서 동일한 데이터 디렉터리를 동시에 쿼리하는 여러 MySQL 인스턴스. 이 기술을 사용하여 부하가 많은 MySQL 인스턴스에서 발생할 수 있는 병목 현상을 방지하거나, 다양한 인스턴스에 대해 서로 다른 구성 옵션을 사용하여 특정 종류의 쿼리에 맞게 각 인스턴스를 조정할 수 있습니다.
- 아카이브된 백업 데이터와 같이 보안 또는 데이터 무결성 등의 이유로 읽기 전용 상태로 설정된 데이터를 쿼리합니다.

참고

이 기능은 주로 읽기 전용 측면에 기반한 원시 성능보다는 배포 및 배포의 유연성을 위한 것입니다. 전체 서버를 읽기 전용으로 설정할 필요가 없는 읽기 전용 쿼리의 성능을 조정하는 방법은 8.5.3절, 'InnoDB 읽기 전용 트랜잭션 최적화'를 참조하세요.

작동 방식

innodb-read-only 옵션을 통해 서버가 읽기 전용 모드로 실행되는 경우, 특정 InnoDB 기능 및 구성 요소가 축소되거나 완전히 꺼집니다:

- 변경 버퍼링이 수행되지 않으며, 특히 변경 버퍼에서 병합이 수행되지 않습니다. 읽기 전용 작업을 위해 인스턴스를 준비할 때 변경 버퍼가 비어 있는지 확인하려면 변경 버퍼링을 비활성화하고 (innodb_change_buffering=0) 먼저 느린 종료를 수행합니다.
- 시작 시 크래시 복구 단계가 없습니다. 인스턴스가 읽기 전용 상태가 되기 전에 느린 종료를 수행했어야 합니다.
- 읽기 전용 작업에서는 재실행 로그가 사용되지 않으므로 인스턴스를 읽기 전용으로 설정하기 전에 innodb_log_file_size를 가능한 가장 작은 크기(1MB)로설정할 수 있습니다.

- 대부분의 백그라운드 스레드가 꺼집니다. 읽기 전용 모드에서 허용되는 임시 파일에 대한 쓰기를 위한 I/O 읽기 스레드와 I/O 쓰기 스레드 및 페이지 플러시 코디네이터 스레드는 그대로 유지됩니다. 버퍼 풀 크기 조정 스레드도 활성 상태로 유지되어 버퍼 풀의 온라인 크기 조정을 가능하게 합니다.
- 교착 상태, 모니터 출력 등에 대한 정보는 임시 파일에 기록되지 않습니다. 따라서 엔진 `INNODB` 상태 표시가출력을 생성하지 않습니다.
- 일반적으로 쓰기 작업의 동작을 변경하는 구성 옵션 설정 변경은 서버가 읽기 전용 모드일 때는 적용되지 않습니다.
- 격리 수준을 적용하기 위한 MVCC 처리가 꺼져 있습니다. 업데이트 및 삭제가 불가능하므로 모든 쿼리는 최신 버전의 레코드를 읽습니다.
- 실행 취소 로그는 사용되지 않습니다. `innodb_undo_tablespaces` 및 `innodb_undo_directory` 구성 옵션.

15.8.3 InnoDB 버퍼 풀 구성

이 섹션에서는 InnoDB 버퍼 풀에 대한 구성 및 튜닝 정보를 제공합니다.

15.8.3.1 InnoDB 버퍼 풀 크기 구성

오프라인 또는 서버가 실행 중인 상태에서 InnoDB 버퍼 풀 크기를 구성할 수 있습니다. 이 섹션에 설명된 동작은 두 가지 방법 모두에 적용됩니다. 온라인으로 버퍼 풀 크기를 구성하는 방법에 대한 자세한 내용은 온라인으로 InnoDB 버퍼 풀 크기 구성을 참조하세요.

innodb_buffer_pool_size를 늘리거나 줄일 때, 작업은 청크 단위로 수행됩니다. 청크 크기는 innodb_buffer_pool_chunk_size 구성 옵션으로 정의되며, 기본값은 128M입니다. 자세한 내용은 InnoDB 버퍼 풀 청크 크기 구성을 참조하세요.

버퍼 풀 크기는 항상 innodb_buffer_pool_chunk_size의 배수이거나 같아야 합니다.
* innodb_buffer_pool_instances. innodb_buffer_pool_size를 구성하는 경우
를 innodb_buffer_pool_chunk_size * innodb_buffer_pool_instances의 배수 또는 같지 않은 값으로 변경하면 버퍼 풀 크기가 innodb_buffer_pool_chunk_size * innodb_buffer_pool_instances의 배수 또는 같은 값으로 자동 조정됩니다.

다음 예제에서는 innodb_버퍼풀_size가 8G로 설정되어 있고, innodb_버퍼풀_instances는 16으로 설정되어 있습니다. innodb_버퍼풀_chunk_size는 기본값인 128M입니다.

8G는 innodb_버퍼 풀 인스턴스=16 * innodb_버퍼 풀 청크 크기=128M의 배수인 2G이므로 8G가 유효한 innodb_버퍼 풀 크기 값입니다.

```
$> mysqld --innodb-buffer-pool-size=8G --innodb-buffer-pool-instances=16
```

```
mysql> SELECT @@innodb_buffer_pool_size/1024/1024/1024;
+----------------------------------------+
| @@innodb_buffer_pool_size/1024/1024/1024 |
+----------------------------------------+
|8                          .000000000000 |
+----------------------------------------+
```

이 예제에서는 innodb_버퍼풀_size가 9G로 설정되어 있고, innodb_버퍼풀_instances가는 16으로 설정되어 있으며, 기본값인 128M이 innodb_buffer_pool_chunk_size입니다.
이 경우 9G는 innodb_buffer_pool_instances=16 *의 배수가 아닙니다.
innodb_버퍼풀_청크_size=128M이므로 innodb_버퍼풀_size는 innodb_버퍼풀_청크_size * innodb_버퍼풀_instances의 배수인 10G로 조정됩니다.

```
$> mysqld --innodb-buffer-pool-size=9G --innodb-buffer-pool-instances=16
```

```
mysql> SELECT @@innodb_buffer_pool_size/1024/1024/1024;
+----------------------------------------+
| @@innodb_buffer_pool_size/1024/1024/1024 |
+----------------------------------------+
|10                         .000000000000 |
+----------------------------------------+
```

InnoDB 버퍼 풀 청크 크기 구성

innodb_buffer_pool_chunk_size는 1MB(1048576바이트) 단위로 늘리거나 줄일 수 있지만, 시작 시, 명령줄 문자열 또는 MySQL 구성 파일에서만 수정할 수 있습니다.

명령줄:

```
$> mysqld --innodb-buffer-pool-chunk-size=134217728
```

구성 파일입니다:

```
[mysqld]
```

```
innodb_buffer_pool_chunk_size=134217728
```

innodb_buffer_pool_chunk_size를 변경할 때는 다음 조건이 적용됩니다:

- 버퍼 풀이 초기화될 때 새로운 innodb_버퍼풀_chunk_size 값 * innodb_버퍼풀_instances가 현재 버퍼 풀 크기보다 크면, innodb_버퍼풀_chunk_size는 innodb_버퍼풀_size / innodb_버퍼풀_instances로 잘려나갑니다.

  예를 들어 버퍼 풀의 크기가 2GB(2147483648 바이트), 버퍼 풀 인스턴스 4개, 청크 크기가 1GB(1073741824 바이트)로 초기화되어 있는 경우, 아래와 같이 청크 크기가 innodb_buffer_pool_size / innodb_buffer_pool_instances와 같은 값으로 잘려집니다:

```
$> mysqld --innodb-buffer-pool-size=2147483648 --innodb-buffer-pool-instances=4
--innodb-buffer-pool-chunk-size=1073741824;
```

```
mysql> SELECT @@innodb_buffer_pool_size;
+----------------------------+
| @@innodb_buffer_pool_size |
+----------------------------+
                 |2147483648 |
+----------------------------+

mysql> SELECT @@innodb_buffer_pool_instances;
+-------------------------------+
| @@innodb_buffer_pool_instances |
+-------------------------------+
|4 |
+-------------------------------+

# 청크 크기가 시작 시 1GB(1073741824 바이트)로 설정되었지만
innodb_버퍼_풀_size / innodb_버퍼_풀_인스턴스로 # 잘림

mysql> SELECT @@innodb_buffer_pool_chunk_size;
+---------------------------------+
| @@innodb_buffer_pool_chunk_size |
+---------------------------------+
                        |536870912 |
+---------------------------------+
```

- 버퍼 풀 크기는 항상 innodb_buffer_pool_chunk_size의 배수이거나 같아야 합니다.
  * innodb_buffer_pool_instances. innodb_buffer_pool_chunk_size를 변경하면 innodb_buffer_pool_size는 innodb_buffer_pool_chunk_size * innodb_buffer_pool_instances의 배수 또는 같은 값으로 자동 조정됩니다. 조정은 버퍼 풀이 초기화될 때 발생합니다. 이 동작은 다음 예제에서 설명합니다:

```
# 버퍼 풀의 기본 크기는 128MB(134217728 바이트)입니다.

mysql> SELECT @@innodb_buffer_pool_size;
+---------------------------+
| @@innodb_buffer_pool_size |
+---------------------------+
                  |134217728 |
+---------------------------+

# 청크 크기도 128MB(134217728 바이트)입니다. mysql>

SELECT @@innodb_buffer_pool_chunk_size;
+---------------------------------+
| @@innodb_buffer_pool_chunk_size |
+---------------------------------+
                        |134217728 |
+---------------------------------+

# 단일 버퍼 풀 인스턴스가 있습니다. mysql> SELECT

@@innodb_buffer_pool_instances;
+--------------------------------+
| @@innodb_buffer_pool_instances |
```

```
+---------------------------------+
|1 |
+---------------------------------+

# 시작 시 청크 크기가 1MB(1048576바이트) 감소 # (134217728 -
1048576 = 133169152):

$> mysqld --innodb-buffer-pool-chunk-size=133169152

mysql> SELECT @@innodb_buffer_pool_chunk_size;
+---------------------------------+
| @@innodb_buffer_pool_chunk_size |
+---------------------------------+
                        |133169152 |
+---------------------------------+

# 버퍼 풀 크기가 134217728 에서 266338304 로 증가했습니다.
# 버퍼 풀 크기가 자동으로 다음과 같은 값으로 조정됩니다.
# 또는 innodb_버퍼_풀_청크_크기 * innodb_버퍼_풀_인스턴스의 배수입니다.

mysql> SELECT @@innodb_buffer_pool_size;
+---------------------------+
| @@innodb_buffer_pool_size |
+---------------------------+
                  |266338304 |
+---------------------------+
```

이 예는 동일한 동작을 보여 주지만 여러 버퍼 풀 인스턴스가 있는 경우를 보여줍니다:

```
# 버퍼 풀의 기본 크기는 2GB(2147483648 바이트)입니다.

mysql> SELECT @@innodb_buffer_pool_size;
+---------------------------+
| @@innodb_buffer_pool_size |
+---------------------------+
                 |2147483648 |
+---------------------------+

# 청크 크기는 0.5GB(536870912 바이트)입니다. mysql>

SELECT @@innodb_buffer_pool_chunk_size;
+---------------------------------+
| @@innodb_buffer_pool_chunk_size |
+---------------------------------+
                        |536870912 |
+---------------------------------+

# 버퍼 풀 인스턴스가 4개 있습니다.

mysql> SELECT @@innodb_buffer_pool_instances;
+---------------------------------+
| @@innodb_buffer_pool_instances |
+---------------------------------+
|4 |
+---------------------------------+

# 시작 시 청크 크기가 1MB(1048576바이트)씩 감소합니다 # (536870912
- 1048576 = 535822336):

$> mysqld --innodb-buffer-pool-chunk-size=535822336

mysql> SELECT @@innodb_buffer_pool_chunk_size;
+---------------------------------+
| @@innodb_buffer_pool_chunk_size |
+---------------------------------+
                        |535822336 |
+---------------------------------+

# 버퍼 풀 크기가 2147483648 에서 4286578688 로 증가했습니다.
# 버퍼 풀 크기가 자동으로 다음과 같은 값으로 조정됩니다.
# 또는 innodb_버퍼_풀_청크_크기 * innodb_버퍼_풀_인스턴스의 배수입니다.
```

```
mysql> SELECT @@innodb_buffer_pool_size;
+----------------------------+
| @@innodb_buffer_pool_size |
+----------------------------+
                  |4286578688 |
+----------------------------+
```

이 값을 변경하면 위의 예시와 같이 버퍼 풀의 크기가 커질 수 있으므로 innodb_buffer_pool_chunk_size를 변경할 때 주의해야 합니다. innodb_buffer_pool_chunk_size를 변경하기 전에, 결과 버퍼 풀 크기가 허용 가능한지 확인하기 위해 innodb_buffer_pool_size에 미치는 영향을 계산하세요.

참고

잠재적인 성능 문제를 방지하려면 청크 수(innodb_버퍼풀_크기/innodb_버퍼풀_청크_크기)가 1000을 초과하지 않아야 합니다.

온라인에서 InnoDB 버퍼 풀 크기 구성하기

innodb_buffer_pool_size 구성 옵션은 SET 문을 사용하여 동적으로 설정할 수 있으므로 서버를 다시 시작하지 않고도 버퍼 풀의 크기를 조정할 수 있습니다. 예를 들어

```
mysql> SET GLOBAL innodb_buffer_pool_size=402653184;
```

참고

버퍼 풀 크기는 innodb_buffer_pool_chunk_size * innodb_buffer_pool_instances의 배수이거나 같아야 합니다. 이러한 변수 설정을 변경하려면 서버를 다시 시작해야 합니다.

버퍼 풀의 크기를 조정하기 전에 InnoDB API를 통해 수행되는 활성 트랜잭션 및 작업을 완료해야 합니다. 크기 조정 작업을 시작할 때 모든 활성 트랜잭션이 완료될 때까지 작업이 시작되지 않습니다. 크기 조정 작업이 진행 중이면 버퍼 풀에 액세스해야 하는 새 트랜잭션 및 작업은 크기 조정 작업이 완료될 때까지 기다려야 합니다. 이 규칙의 예외는 버퍼 풀이 조각 모음되는 동안 버퍼 풀에 대한 동시 액세스가 허용되고 버퍼 풀 크기가 줄어들 때 페이지가 철회된다는 것입니다. 동시 액세스를 허용하면 페이지가 철회되는 동안 일시적으로 사용 가능한 페이지가 부족해질 수 있다는 단점이 있습니다.

참고

버퍼 풀 크기 조정 작업이 시작된 후에 중첩된 트랜잭션이 시작되면 실패할 수 있습니다.

온라인 버퍼 풀 크기 조정 진행 상황 모니터링

예를 들어, Innodb_buffer_pool_resize_status 변수는 버퍼 풀 크기 조정 진행률을 나타내는 문자열 값을 보고합니다:

```
mysql> SHOW STATUS WHERE Variable_name='InnoDB_buffer_pool_resize_status';
+---------------------------------+ -------------------------------------+
| 변수_이름| 값                        |
+---------------------------------+ -------------------------------------+
| Innodb_buffer_pool_resize_status | 다른 해시 테이블도 크기 조정. |
+---------------------------------+ -------------------------------------+
```

Innodb_buffer_pool_resize_status_code를 사용하여 온라인 버퍼 풀 크기 조정 작업을 모니터링할 수도 있습니다.

프로그래밍 방식의 모니터링에 적합한 숫자 값을 보고하는 `Innodb_buffer_pool_resize_status_progress` 상태 변수를 사용합니다.

`Innodb_buffer_pool_resize_status_code` 상태 변수는 온라인 버퍼 풀 크기 조정 작업의 단계를 나타내는 상태 코드를 보고합니다. 상태 코드는 다음과 같습니다:

- 0: 크기 조정 작업 진행 중 없음
- 1: 크기 조정 시작
- 2: AHI(적응형 해시 인덱스) 비활성화하기
- 3: 블록 인출하기
- 4: 글로벌 잠금 획득
- 5: 풀 크기 조정
- 6: 해시 크기 조정
- 7: 크기 조정 실패

`Innodb_buffer_pool_resize_status_progress` 상태 변수는 각 단계의 진행률을 나타내는 백분율 값을 보고합니다. 백분율 값은 각 버퍼 풀 인스턴스가 처리된 후에 업데이트됩니다. 상태 (`Innodb_buffer_pool_resize_status_code`로 보고됨)가 한 상태에서 다른 상태로 변경되면 백분율 값은 0으로 재설정됩니다.

다음 쿼리는 버퍼 풀 크기 조정 진행률을 나타내는 문자열 값, 작업의 현재 단계를 나타내는 코드 및 해당 단계의 현재 진행률을 백분율 값으로 표시하여 반환합니다:

```
SELECT variable_name, variable_value
 FROM performance_schema.global_status
 WHERE LOWER(variable name) LIKE "innodb buffer pool resize%";
```

버퍼 풀 크기 조정 진행 상황은 서버 오류 로그에서도 확인할 수 있습니다. 이 예는 버퍼 풀의 크기를 늘릴 때 기록되는 메모를 보여줍니다:

```
[참고] InnoDB: 버퍼 풀 크기를 134217728 에서 4294967296 로 조정. (단위=134217728) [참고]
InnoDB: 적응형 해시 인덱스 비활성화.
[참고] InnoDB: 버퍼 풀 0 : 31 청크(253952 블록)가 추가되었습니다. [참고]
InnoDB: 버퍼 풀 0: 해시 테이블의 크기가 조정되었습니다.
[참고] InnoDB: lock_sys, 적응형 해시 인덱스, 사전에서 해시 테이블 크기 조정. [참고] InnoDB:
버퍼 풀 크기를 134217728 에서 4294967296 로 조정 완료. [참고] InnoDB: 적응형 해시 인덱스를 다
시 활성화했습니다.
```

이 예는 버퍼 풀의 크기를 줄일 때 기록되는 메모를 보여줍니다:

```
[참고] InnoDB: 버퍼 풀 크기를 4294967296 에서 134217728 로 조정. (단위=134217728) [참고]
InnoDB: 적응형 해시 인덱스 비활성화.
[참고] InnoDB: 버퍼 풀 0 : 마지막 253952 블록 인출 시작.
[참고] InnoDB: 버퍼 풀 0: 사용 가능한 목록에서 253952개 블록을 인출했습니다.
0 페이지. (253952/253952)
[참고] InnoDB: 버퍼 풀 0 : 대상 253952 블록을 철회했습니다. [참고] InnoDB:
버퍼 풀 0: 31개 청크(253952블록)가 해제되었습니다. [참고] InnoDB: 버퍼 풀 0:
해시 테이블 크기가 조정되었습니다.
[참고] InnoDB: lock_sys, 적응형 해시 인덱스, 사전에서 해시 테이블 크기 조정. [참고] InnoDB:
버퍼 풀 크기를 4294967296 에서 134217728 로 조정 완료. [참고] InnoDB: 적응형 해시 인덱스를 다
시 활성화했습니다.
```

로그 오류 상세도=3으로 서버를 시작하면 온라인 버퍼 풀 크기 조정 작업 중에 추가 정보가 오류 로그에 기록됩니다. 추가 정보에는 Innodb_buffer_pool_resize_status_code가 보고하는 상태 코드와 Innodb_buffer_pool_resize_status_progress가 보고하는 백분율 진행률 값이 포함됩니다.

```
[참고] [MY-012398] [InnoDB] 버퍼 풀 크기 조정 요청. (새 크기: 1073741824 바이트) [참고] [MY-013954]
[InnoDB] 상태 코드 1: 버퍼 풀 크기를 134217728 에서 1073741824 (단위=134217728)로 조정 중입니다.
[참고] [MY-013953] [InnoDB] 상태 코드 1: 100% 완료 [참고] [MY-
013952] [InnoDB] 상태 코드 1: 완료됨
[참고] [MY-013954] [InnoDB] 상태 코드 2: 적응형 해시 인덱스 비활성화. [참고] [MY-
011885] [InnoDB] 적응형 해시 인덱스 비활성화.
```

```
[참고] [MY-013953] [InnoDB] 상태 코드 2: 100% 완료 [참고] [MY-
013952] [InnoDB] 상태 코드 2: 완료됨
[참고] [MY-013954] [InnoDB] 상태 코드 3: 축소할 블록을 인출 중입니다. [참고] [MY-013953]
[InnoDB] 상태 코드 3: 100% 완료
[참고] [MY-013952] [InnoDB] 상태 코드 3: 완료됨
[참고] [MY-013954] [InnoDB] 상태 코드 4: 버퍼 풀 전체 래칭 중입니다. [참고] [MY-
013953] [InnoDB] 상태 코드 4: 14% 완료
[참고] [MY-013953] [InnoDB] 상태 코드 4: 28% 완료 [참고] [MY-
013953] [InnoDB] 상태 코드 4: 42% 완료 [참고] [MY-013953]
[InnoDB] 상태 코드 4: 57% 완료 [참고] [MY-013953] [InnoDB] 상태
코드 4: 71% 완료 [참고] [MY-013953] [InnoDB] 상태 코드 4: 85%
완료 [참고] [MY-013953] [InnoDB] 상태 코드 4: 100% 완료 [참고]
[MY-013952] [InnoDB] 상태 코드 4: 완료됨
[참고] [MY-013954] [InnoDB] 상태 코드 5: 풀 크기 조정 시작 중
[참고] [MY-013954] [InnoDB] 상태 코드 5: 버퍼 풀 0 : 청크 1~8로 크기 조정 중입니다. [참고] [MY-
011891] [InnoDB] 버퍼 풀 0 : 청크 7개(57339 블록)가 추가되었습니다.
[참고] [MY-013953] [InnoDB] 상태 코드 5: 100% 완료 [참고] [MY-
013952] [InnoDB] 상태 코드 5: 완료됨
[참고] [MY-013954] [InnoDB] 상태 코드 6: 해시 테이블 크기 조정 중입니다. [참고]
[MY-011892] [InnoDB] 버퍼 풀 0 : 해시 테이블 크기가 조정되었습니다. [참고] [MY-
013953] [InnoDB] 상태 코드 6: 100% 완료
[참고] [MY-013954] [InnoDB] 상태 코드 6: 다른 해시 테이블도 크기 조정 중입니다.
[참고] [MY-011893] [InnoDB] lock_sys, 적응형 해시 인덱스, 사전에서 해시 테이블 크기 조정. [참고] [MY-
011894] [InnoDB] 버퍼 풀 크기를 134217728 에서 1073741824 로 변경 완료. [참고] [MY-011895] [InnoDB] 적
응형 해시 인덱스 다시 활성화.
[참고] [MY-013952] [InnoDB] 상태 코드 6: 완료됨
[참고] [MY-013954] [InnoDB] 상태 코드 0: 220826 6:25:46에 버퍼 풀 크기 조정 완료. [참고] [MY-013953]
[InnoDB] 상태 코드 0: 100% 완료
```

온라인 버퍼 풀 크기 조정 내부

크기 조정 작업은 백그라운드 스레드에서 수행됩니다. 버퍼 풀의 크기를 늘릴 때 크기 조정 작업이 수행됩니다:

- 청크 단위로 페이지를 추가합니다(청크 크기는 `innodb_buffer_pool_chunk_size`로 정의됨).
- 해시 테이블, 목록 및 포인터를 변환하여 메모리의 새 주소를 사용하도록 합니다.
- 무료 목록에 새 페이지를 추가합니다.

이러한 작업이 진행되는 동안 다른 스레드는 버퍼 풀에 액세스하지 못하도록 차단됩니다. 버퍼 풀의 크기를 줄

이는 경우 크기 조정 작업이 수행됩니다:

- 버퍼 풀을 조각 모음하고 페이지를 인출(해제)합니다.
- 청크 단위로 페이지를 제거합니다(청크 크기는 `innodb_buffer_pool_chunk_size`로 정의됨).

- 해시 테이블, 목록 및 포인터를 변환하여 메모리의 새 주소를 사용하도록 합니다.

이러한 작업 중 버퍼 풀 조각 모음과 페이지 인출 작업만 다른 스레드가 동시에 버퍼 풀에 액세스할 수 있도록 허용합니다.

15.8.3.2 여러 버퍼 풀 인스턴스 구성

수 기가바이트 범위의 버퍼 풀이 있는 시스템의 경우 버퍼 풀을 별도의 인스턴스로 나누면 여러 스레드가 캐시된 페이지를 읽고 쓸 때 경합을 줄여 동시성을 향상시킬 수 있습니다. 이 기능은 일반적으로 버퍼 풀 크기가 수 기가바이트 범위인 시스템을 위한 것입니다. 여러 버퍼 풀 인스턴스는 `innodb_buffer_pool_instances` 구성 옵션을 사용하여 구성할 수 있으며, `innodb_buffer_pool_size` 값을 조정할 수도 있습니다.

`InnoDB` 버퍼 풀이 크면 메모리에서 검색하여 많은 데이터 요청을 충족할 수 있습니다. 여러 스레드가 한 번에 버퍼 풀에 액세스하려고 하면 병목 현상이 발생할 수 있습니다. 여러 버퍼 풀을 활성화하여 이러한 경합을 최소화할 수 있습니다. 버퍼 풀에 저장되거나 버퍼 풀에서 읽혀지는 각 페이지는 해싱 함수를 사용하여 버퍼 풀 중 하나에 무작위로 할당됩니다. 각 버퍼

풀은 자체 프리 리스트, 플러시 리스트, LRU 및 버퍼 풀에 연결된 기타 모든 데이터 구조를 관리합니다.

여러 버퍼 풀 인스턴스를 사용하도록 설정하려면 innodb_buffer_pool_instances 구성 옵션을 1(기본값)에서 최대 64(최대값)보다 큰 값으로 설정합니다. 이 옵션은 다음과 같은 경우에 적용됩니다.
를 1GB 이상의 크기로 설정한 경우에만 사용할 수 있습니다. 지정한 총 크기는 모든 버퍼 풀로 나뉩니다. 최상의 효율성을 위해 다음 조합을 지정합니다.
각 버퍼 풀 인스턴스가 최소 1GB가 되도록 innodb_버퍼풀_instances 및 innodb_버퍼풀_size를 설정합니다.

InnoDB 버퍼 풀 크기 수정에 대한 자세한 내용은 15.8.3.1절, 'InnoDB 버퍼 풀 크기 구성'을 참조하세요.

15.8.3.3 버퍼 풀 스캔 저항성 만들기

InnoDB는 엄격한 LRU 알고리즘을 사용하는 대신, 버퍼 풀로 가져와서 다시는 액세스되지 않는 데이터의 양을 최소화하는 기술을 사용합니다. 이 기법의 목표는 미리 읽기 및 전체 테이블 스캔이 나중에 액세스할 수도 있고 그렇지 않을 수도 있는 새 블록을 가져오는 경우에도 자주 액세스하는('핫') 페이지가 버퍼 풀에 남아 있도록 하는 것입니다.

새로 읽은 블록은 LRU 목록의 가운데에 삽입됩니다. 새로 읽은 모든 페이지는 기본적으로 LRU 목록의 꼬리에서 3/8인 위치에 삽입됩니다. 버퍼 풀에서 페이지가 처음 액세스되면 페이지가 목록의 앞쪽(가장 최근에 사용된 끝)으로 이동합니다. 따라서 한 번도 액세스되지 않은 페이지는 LRU 목록의 앞부분에 도달하지 못하며 엄격한 LRU 접근 방식보다 더 빨리 "노후화"됩니다. 이 방식은 LRU 목록을 두 개의 세그먼트로 나누며, 삽입 지점의 하류에 있는 페이지는 "오래된" 페이지로 간주되어 LRU 퇴출의 대상이 됩니다.

InnoDB 버퍼 풀의 내부 작동 방식에 대한 설명과 LRU 알고리즘에 대한 자세한 내용은 15.5.1절. "버퍼 풀"을 참조하세요.

LRU 목록의 삽입 지점을 제어하고 테이블 또는 인덱스 스캔을 통해 버퍼 풀로 가져온 블록에 InnoDB가 동일한 최적화를 적용할지 여부를 선택할 수 있습니다. 구성 매개변수 innodb_old_blocks_pct는 LRU 목록에서 "오래된" 블록의 비율을 제어합니다. innodb_old_blocks_pct의 기본값은 37이며, 이는 원래의 고정 비율인 3/8에 해당합니다. 값 범위는 5(버퍼 풀의 새 페이지가 매우 빠르게 노후화됨)에서 95(버퍼 풀의 5%만 핫 페이지용으로 예약되어 있어 알고리즘이 익숙한 LRU 전략에 가깝게 됨)까지입니다.

미리 읽기로 인해 버퍼 풀이 휘젓는 것을 방지하는 최적화를 통해 테이블 또는 인덱스 스캔으로 인한 유사한 문제를 방지할 수 있습니다. 이러한 스캔에서는 일반적으로 데이터 페이지가 몇 번 연속해서 빠르게 액세스되고 다시는 건드리지 않습니다. 구성 매개변수 innodb_old_blocks_time은 페이지에 처음 액세스한 후 LRU 목록의 맨 앞(가장 최근에 사용된 끝)으로 이동하지 않고 액세스할 수 있는 시간 창(밀리초)을 지정합니다. innodb_old_blocks_time의 기본값은 1000입니다. 이 값을 늘리면 버퍼 풀에서 점점 더 많은 블록이 더 빨리 노후화될 가능성이 높아집니다.

innodb_old_blocks_pct와 innodb_old_blocks_time은 모두 MySQL 옵션 파일(my.cnf 또는 my.ini)에서 지정하거나 런타임에 SET GLOBAL 문을 사용하여 변경할 수 있습니다. 런타임에 값을 변경하려면 전

역 시스템 변수를 설정할 수 있는 충분한 권한이 필요합니다. 섹션 5.1.9.1, "시스템 변수 권한"을 참조하세요.

이러한 매개변수 설정의 효과를 측정하는 데 도움이 되도록 `SHOW ENGINE INNODB STATUS` 명령은 버퍼 풀 통계를 보고합니다. 자세한 내용은 InnoDB 표준 모니터를 사용하여 버퍼 풀 모니터링하기를 참조하세요.

이러한 매개변수의 효과는 하드웨어 구성, 데이터 및 워크로드의 세부 사항에 따라 크게 달라질 수 있으므로 성능이 중요한 환경이나 프로덕션 환경에서 이러한 설정을 변경하기 전에 항상 벤치마킹을 통해 효과를 확인해야 합니다.

대부분의 활동이 주기적인 일괄 보고 쿼리가 있는 OLTP 유형으로 대규모 스캔을 초래하는 혼합 워크로드의 경우, 일괄 실행 중에 `innodb_old_blocks_time` 값을 설정하면 일반 워크로드의 작업 집합을 버퍼 풀에 유지하는 데 도움이 될 수 있습니다.

버퍼 풀에 완전히 들어갈 수 없는 큰 테이블을 스캔할 때 innodb_old_blocks_pct를 작은 값으로 설정하면 한 번만 읽은 데이터가 버퍼 풀의 상당 부분을 차지하지 않도록 할 수 있습니다. 예를 들어, innodb_old_blocks_pct=5로 설정하면 한 번만 읽는 이 데이터가 버퍼 풀의 5%로 제한됩니다.

메모리에 맞는 작은 테이블을 스캔할 때는 버퍼 풀 내에서 페이지를 이동하는 데 드는 오버헤드가 적으므로 innodb_old_blocks_pct를 기본값으로 두거나 innodb_old_blocks_pct=50과 같이 더 높게 설정할 수 있습니다.

innodb_old_blocks_time 매개변수의 효과는 innodb_old_blocks_pct 매개변수보다 예측하기 어렵고, 상대적으로 작으며, 워크로드에 따라 더 많이 달라집니다. 최적의 값에 도달하려면, innodb_old_blocks_pct 조정으로 인한 성능 개선이 충분하지 않은 경우 자체 벤치마크를 수행하십시오.

15.8.3.4 InnoDB 버퍼 풀 프리페칭 구성(읽기 앞서)

미리 읽기 요청은 이러한 페이지가 곧 필요할 것으로 예상하여 버퍼 풀에 있는 여러 페이지를 비동기적으로 미리 가져오기 위한 I/O 요청입니다. 이 요청은 모든 페이지를 한 번에 가져옵니다. InnoDB는 두 가지 읽기 선행 알고리즘을 사용하여 I/O 성능을 개선합니다:

선형 미리 읽기는 순차적으로 액세스되는 버퍼 풀의 페이지를 기반으로 곧 어떤 페이지가 필요할지 예측하는 기술입니다. 구성 매개변수 innodb_read_ahead_threshold를 사용하여 비동기 읽기 요청을 트리거하는 데 필요한 순차적 페이지 액세스 횟수를 조정하여 InnoDB가 읽기 앞서 작업을 수행하는 시기를 제어할 수 있습니다. 이 매개변수가 추가되기 전에는 InnoDB가 현재 익스텐션의 마지막 페이지를 읽을 때만 다음 익스텐션 전체에 대한 비동기 프리페치 요청을 실행할지 여부를 계산했습니다.

구성 매개변수 innodb_read_ahead_threshold는 InnoDB가순차적 페이지 액세스 패턴을 감지하는 데 얼마나 민감한지를 제어합니다. 한 범위에서 순차적으로 읽은 페이지 수가 innodb_read_ahead_threshold보다 크거나 같으면 InnoDB는 다음 범위 전체에 대한 비동기 읽기 앞서 작업을 시작합니다. innodb_read_ahead_threshold는 0-64 사이의 임의의 값으로 설정할 수 있습니다. 기본값은 56입니다. 값이 높을수록 액세스 패턴 검사가 더 엄격해집니다. 예를 들어 값을 48로 설정하면 InnoDB는 현재 범위의 48페이지가 순차적으로 액세스된 경우에만 선형 읽기 사전 요청을 트리거합니다. 값이 8인 경우, InnoDB는
를 설정하면 범위 내에서 8페이지까지 순차적으로 액세스하는 경우에도 비동기식 미리 읽기가 가능합니다. 이 매개변수의 값은 MySQL 구성 파일에서 설정하거나 다음을 사용하여 동적으로 변경할 수 있습니다.
글로벌 시스템 변수를 설정할 수 있는 충분한 권한이 필요한 SET GLOBAL 문을 사용할 수 있습니다. 섹션 5.1.9.1, "시스템 변수 권한"을 참조하십시오.

무작위 미리 읽기는 페이지가 읽힌 순서와 관계없이 버퍼 풀에 이미 있는 페이지를 기반으로 페이지가 곧 필요할 수 있는 시기를 예측하는 기술입니다. 버퍼 풀에서 동일한 범위의 페이지가 13개 연속으로 발견되면 InnoDB는 비동기적으로 요청을 발행합니다.
를 사용하여 범위의 나머지 페이지를 미리 가져옵니다. 이 기능을 사용하려면 구성 변수

`innodb_random_read_ahead`를 `ON`으로 설정합니다.

`SHOW ENGINE INNODB STATUS` 명령은 미리 읽기 알고리즘의 효과를 평가하는 데 도움이 되는 통계를 표시합니다. 통계에는 다음과 같은 글로벌 상태 변수에 대한 카운터 정보가 포함됩니다:

- `Innodb_버퍼_풀_read_ahead`
- `Innodb_buffer_pool_read_ahead_evicted`
- `Innodb_버퍼_풀_read_ahead_rnd`

이 정보는 `innodb_random_read_ahead` 설정을 미세 조정할 때 유용할 수 있습니다.

I/O 성능에 대한 자세한 내용은 섹션 8.5.8, 'InnoDB 디스크 I/O 최적화' 및 섹션 8.12.1, '디스크 I/O 최적화'를 참조하세요.

15.8.3.5 버퍼 풀 플러싱 구성

InnoDB는 버퍼 풀에서 더티 페이지를 플러시하는 등 백그라운드에서 특정 작업을 수행합니다. 더티 페이지는 수정되었지만 아직 디스크의 데이터 파일에 기록되지 않은 페이지를 말합니다.

MySQL 8.2에서 버퍼 풀 플러싱은 페이지 클리너 스레드에 의해 수행됩니다. 페이지 클리너 스레드 수는 기본값이 4인 innodb_page_cleaners 변수에 의해 제어됩니다. 그러나 페이지 클리너 스레드 수가 버퍼 풀 인스턴스 수를 초과하는 경우 innodb_page_cleaners는 innodb_buffer_pool_instances와 동일한 값으로 자동 설정됩니다.

버퍼 풀 플러시는 더티 페이지의 비율이 innodb_max_dirty_pages_pct_lwm 변수에 정의된 로워 워터 마크 값에 도달하면 시작됩니다. 기본 로워 워터 마크는 버퍼 풀 페이지의 10%입니다. innodb_max_dirty_pages_pct_lwm 값이 0이면 이 조기 플러싱 동작이 비활성화됩니다.

innodb_max_dirty_pages_pct_lwm 임계값의 목적은 버퍼 풀의 더티 페이지 비율을 제어하고 더티 페이지의 양이 기본값이 90인 innodb_max_dirty_pages_pct 변수에 정의된 임계값에 도달하는 것을 방지하는 것입니다. InnoDB는 버퍼 풀의 더티 페이지 비율이 innodb_max_dirty_pages_pct 임계값에 도달하면 버퍼 풀 페이지를 공격적으로 플러시합니다.

innodb_max_dirty_pages_pct_lwm을 구성할 때는 항상 이 값이 innodb_max_dirty_pages_pct 값.

추가 변수를 통해 버퍼 풀 플러싱 동작을 미세 조정할 수 있습니다:

- innodb_flush_neighbors 변수는 버퍼 풀에서 페이지를 플러시할 때 다른 더티 페이지도 같은 범위에서 플러시할지 여부를 정의합니다.
  - 기본 설정인 0은 innodb_flush_neighbors를 비활성화합니다. 같은 범위의 더티 페이지는 플러시되지 않습니다. 이 설정은 탐색 시간이 중요한 요소가 아닌 비회전식 스토리지(SSD) 장치에 권장됩니다.
  - 1로 설정하면 연속된 더티 페이지를 동일한 범위로 플러시합니다.
  - 2로 설정하면 더러운 페이지가 같은 정도로 플러시됩니다.

  테이블 데이터가 기존 HDD 저장 장치에 저장된 경우 인접 페이지를 한 번의 작업으로 플러시하면 개별 페이지를 서로 다른 시간에 플러시하는 것보다 I/O 오버헤드(주로 디스크 탐색 작업의 경우)가 줄어듭니다. SSD에 저장된 테이블 데이터의 경우 탐색 시간은 중요한 요소가 아니므로 이 설정을 비활성화하여 쓰기 작업을 분산할 수 있습니다.

- innodb_lru_scan_depth 변수는 버퍼 풀 인스턴스별로 페이지 클리너 스레드가 플러시할 더티 페이지를 찾기 위해 버퍼 풀 LRU 목록의 얼마나 아래까지 스캔할지를 지정합니다. 이 작업은 페이지 클리너 스레드가 초당 한 번 수행하는 백그라운드 작업입니다.

  일반적으로 기본값보다 작은 설정이 대부분의 워크로드에 적합합니다. 필요 이상으로 크게 설정하면 성능에 영향을 줄 수 있습니다. 일반적인 워크로드에서 여유 I/O 용량이 있는 경우에만 값을 늘리는 것을 고려하세요.

반대로 쓰기 집약적인 워크로드로 인해 I/O 용량이 포화 상태인 경우, 특히 버퍼 풀이 큰 경우에는 값을 줄이세요.

`innodb_lru_scan_depth`를 조정할 때는 낮은 값으로 시작하여 여유 페이지가 거의 표시되지 않는 것을 목표로 설정을 상향 조정합니다. 또한, 버퍼 풀 인스턴스 수를 변경할 때 `innodb_lru_scan_depth` * `innodb_buffer_pool_instances`는 페이지 클리너 스레드가 매초 수행하는 작업량을 정의하므로, 이 값을 조정하는 것도 고려하세요.

`innodb_flush_neighbors` 및 `innodb_lru_scan_depth` 변수는 주로 쓰기 집약적인 워크로드를 위한 것입니다. DML 활동이 많은 경우 플러싱이 충분히 공격적이지 않으면 플러싱이 뒤처질 수 있으며, 플러싱이 너무 공격적일 경우 디스크 쓰기로 인해 I/O 용량이 포화될 수 있습니다. 이상적인 설정은 다음에 따라 달라집니다.

워크로드, 데이터 액세스 패턴, 스토리지 구성(예: 데이터가 HDD 또는 SSD 장치에 저장되어 있는지 여부)에 따라 달라집니다.

적응형 플러싱

InnoDB는 적응형 플러싱 알고리즘을 사용하여 재실행 로그 생성 속도와 현재 플러싱 속도에 따라 플러싱 속도를 동적으로 조정합니다. 이는 플러싱 활동이 현재 워크로드에 보조를 맞추도록 함으로써 전반적인 성능을 원활하게 하기 위한 것입니다. 플러싱 속도를 자동으로 조정하면 I/O가 폭증할 때 발생할 수 있는 처리량 급감을 방지하는 데 도움이 됩니다.
버퍼 풀 플러싱으로 인한 활동은 일반적인 읽기 및 쓰기 활동에 사용할 수 있는 I/O 용량에 영향을 미칩니다.

일반적으로 많은 재실행 항목을 생성하는 쓰기 집약적인 워크로드와 관련된 급격한 체크포인트는 예를 들어 처리량에 갑작스러운 변화를 일으킬 수 있습니다. 샤프 체크포인트는 InnoDB가 로그 파일의 일부를 재사용하려고 할 때 발생합니다. 그러기 전에 로그 파일의 해당 부분에 재실행 항목이 있는 모든 더티 페이지를 플러시해야 합니다. 로그 파일이 가득 차면 급격한 체크포인트가 발생하여 처리량이 일시적으로 감소합니다. 이 시나리오는 innodb_max_dirty_pages_pct 임계값에 도달하지 않은 경우에도 발생할 수 있습니다.

적응형 플러싱 알고리즘은 버퍼 풀의 더티 페이지 수와 재실행 로그 레코드가 생성되는 속도를 추적하여 이러한 시나리오를 방지하는 데 도움이 됩니다. 이 정보를 기반으로 초당 버퍼 풀에서 플러시할 더티 페이지 수를 결정하여 갑작스러운 워크로드 변화를 관리할 수 있습니다.

innodb_adaptive_flushing_lwm 변수는 재실행 로그 용량에 대한 저수위 표시를 정의합니다. 이 임계값을 초과하면 innodb_adaptive_flushing 변수가 비활성화되어 있더라도 적응형 플러싱이 활성화됩니다.

내부 벤치마킹 결과, 이 알고리즘은 시간이 지나도 처리량을 유지할 뿐만 아니라 전체 처리량도 크게 향상시킬 수 있는 것으로 나타났습니다. 그러나 적응형 플러싱은 워크로드의 I/O 패턴에 상당한 영향을 미칠 수 있으며 모든 경우에 적합하지 않을 수 있습니다. 재실행 로그가 가득 찰 위험이 있을 때 가장 큰 이점을 제공합니다. 적응형 플러싱이 워크로드의 특성에 적합하지 않은 경우 비활성화할 수 있습니다. 적응형 플러싱은 기본적으로 활성화되어 있는 innodb_adaptive_flushing 변수에 의해 제어됩니다.

innodb_flushing_avg_loops는 InnoDB가 이전에 계산된 플러싱 상태의 스냅샷을 유지하는 반복 횟수를 정의하여 적응형 플러싱이 포그라운드 워크로드 변경에 얼마나 빨리 응답하는지를 제어합니다.
innodb_flushing_avg_loops 값이 높을수록 InnoDB가 이전에 계산된 스냅샷을 더 오래 유지하므로 적응형 플러싱이 더 느리게 응답합니다.
높은 값을 설정할 때는 재실행 로그 사용률이 75%(비동기 플러싱이 시작되는 하드코딩된 한계)에 도달하지 않도록 하고, innodb_max_dirty_pages_pct 임계값이 더티 페이지 수를 워크로드에 적합한 수준으로 유지하도록 하는 것이 중요합니다.

워크로드가 일정하고, 로그 파일 크기(innodb_log_file_size)가 크며, 로그 공간 사용률이 75%에 도달하지 않는 작은 스파이크가 있는 시스템은 가능한 한 원활하게 플러시를 유지하려면 높은

innodb_flushing_avg_loops 값을 사용해야 합니다. 부하가 극도로 급증하거나 로그 파일이 많은 공간을 제공하지 않는 시스템의 경우, 값이 작으면 플러싱이 워크로드 변화를 면밀히 추적할 수 있고 로그 공간 사용률 75%에 도달하는 것을 방지하는 데 도움이 됩니다.

플러싱이 늦어지면 버퍼 풀 플러싱 속도가 innodb_io_capacity 설정에 정의된 대로 InnoDB에서 사용할 수 있는 I/O 용량을 초과할 수 있다는 점에 유의하세요. innodb_io_capacity_max 값은 이러한 상황에서 I/O 용량의 상한을 정의하여 I/O 활동의 급증으로 인해 서버의 전체 I/O 용량이 소모되지 않도록 합니다.

innodb_io_capacity 설정은 모든 버퍼 풀 인스턴스에 적용됩니다. 더티 페이지가 플러시되면 I/O 용량이 버퍼 풀 인스턴스 간에 균등하게 분배됩니다.

유휴 기간 동안 버퍼 플러싱 제한하기

innodb_idle_flush_pct 변수는 데이터베이스 페이지가 수정되지 않는 기간인 유휴 기간 동안의 버퍼 풀 플러시 속도를 제한합니다. 이 값은 백분율로 해석됩니다.

의 값(InnoDB에서 사용할 수 있는 초당 I/O 작업 수를 정의)을 설정합니다. 기본값은 100, 즉 innodb_io_capacity 값의 100%입니다. 유휴 기간 동안 플러시를 제한하려면 innodb_idle_flush_pct를 100 미만으로 설정합니다.

유휴 기간 동안 페이지 플러시를 제한하면 솔리드 스테이트 스토리지 장치의 수명을 연장하는 데 도움이 될 수 있습니다. 유휴 기간 동안 페이지 플러싱을 제한하면 유휴 기간이 길어지면 종료 시간이 길어지고 서버 장애가 발생할 경우 복구 시간이 길어지는 등의 부작용이 발생할 수 있습니다.

15.8.3.6 버퍼 풀 상태 저장 및 복원하기

서버 재시작 후 워밍업 기간을 줄이기 위해 InnoDB는 서버 종료 시 각 버퍼 풀에 대해 가장 최근에 사용된 페이지의 비율을 저장하고 서버 시작 시 이 페이지를 복원합니다. 저장되는 최근 사용 페이지의 백분율은 innodb_buffer_pool_dump_pct 구성 옵션으로 정의됩니다.

사용량이 많은 서버를 다시 시작한 후에는 일반적으로 버퍼 풀에 있던 디스크 페이지가 메모리로 다시 가져오기 때문에(동일한 데이터가 쿼리되고 업데이트되는 등) 처리량이 꾸준히 증가하는 워밍업 기간이 있습니다. 시작 시 버퍼 풀을 복원하는 기능을 사용하면 워밍업 기간을 단축할 수 있습니다.
재시작 전에 버퍼 풀에 있던 디스크 페이지를 다시 로드하여 DML 작업이 해당 행에 액세스할 때까지 기다리지 않고 바로 액세스할 수 있습니다. 또한 I/O 요청을 대량으로 일괄 처리할 수 있으므로 전체 I/O 속도가 빨라집니다. 페이지 로딩은 백그라운드에서 이루어지며 데이터베이스 시작을 지연시키지 않습니다.

종료 시 버퍼 풀 상태를 저장하고 시작 시 복원하는 것 외에도 서버가 실행되는 동안 언제든지 버퍼 풀 상태를 저장하고 복원할 수 있습니다. 예를 들어, 꾸준한 워크로드에서 안정적인 처리량에 도달한 후 버퍼 풀의 상태를 저장할 수 있습니다. 또한 해당 작업에만 필요한 데이터 페이지를 버퍼 풀로 가져오는 보고서 또는 유지 관리 작업을 실행한 후 또는 다른 일반적인 워크로드를 실행한 후 이전 버퍼 풀 상태를 복원할 수도 있습니다.

버퍼 풀의 크기는 수 기가바이트에 달할 수 있지만, InnoDB가 디스크에 저장하는 버퍼 풀 데이터는 이에 비하면 매우 작습니다. 적절한 페이지를 찾는 데 필요한 테이블스페이스 ID와 페이지 ID만 디스크에 저장됩니다. 이 정보는 INNODB_BUFFER_PAGE_LRU 정보 스키마 테이블에서 파생됩니다. 기본적으로 테이블스페이스 ID 및 페이지 ID 데이터는 ib_buffer_pool이라는 파일에 저장되며, 이 파일은 InnoDB 데이터 디렉터리에 저장됩니다. 파일 이름과 위치는 innodb_buffer_pool_filename 구성 파라미터를 사용하여 수정할 수 있습니다.

데이터는 일반 데이터베이스 작업과 마찬가지로 버퍼 풀에서 캐시되고 에이징되므로 디스크 페이지가 최근에 업데이트되었거나 DML 작업에 아직 로드되지 않은 데이터가 포함되어 있어도 문제가 없습니다. 로딩 메커니즘은 더 이상 존재하지 않는 요청된 페이지를 건너뜁니다.

기본 메커니즘에는 덤프 및 로드 작업을 수행하기 위해 파견되는 백그라운드 스레드가 포함됩니다.

압축된 테이블의 디스크 페이지는 압축된 형태로 버퍼 풀에 로드됩니다. DML 작업 중에 페이지 콘텐츠에 액

세스할 때는 페이지가 평소처럼 압축 해제됩니다. 페이지 압축을 푸는 작업은 CPU를 많이 사용하는 프로세스이므로 버퍼 풀 복원 작업을 수행하는 단일 스레드보다는 연결 스레드에서 작업을 수행하는 것이 동시성 측면에서 더 효율적입니다.

버퍼 풀 상태 저장 및 복구와 관련된 작업은 다음 항목에 설명되어 있습니다:

- 버퍼 풀 페이지의 덤프 비율 구성하기
- 종료 시 버퍼 풀 상태 저장 및 시작 시 복원하기
- 온라인으로 버퍼 풀 상태 저장 및 복원하기
- 버퍼 풀 덤프 진행률 표시

- 버퍼 풀 로드 진행률 표시
- 버퍼 풀 로드 작업 중단
- 성능 스키마를 사용한 버퍼 풀 로드 진행 상황 모니터링

버퍼 풀 페이지의 덤프 비율 구성하기

버퍼 풀에서 페이지를 덤프하기 전에, 가장 최근에 사용한 버퍼 풀 페이지의 백분율을 설정하여 덤프할 버퍼 풀 페이지의 비율을 구성할 수 있습니다.
옵션을 설정합니다. 서버가 실행되는 동안 버퍼 풀 페이지를 덤프하려는 경우 다음과 같은 옵션을 구성할 수 있습니다.
동적으로:

```
SET GLOBAL innodb_buffer_pool_dump_pct=40;
```

서버 종료 시 버퍼 풀 페이지를 덤프하려면 구성 파일에서 `innodb_buffer_pool_dump_pct`를 설정하세요.

```
[mysqld]
innodb_buffer_pool_dump_pct=40
```

`innodb_buffer_pool_dump_pct` 기본값은 25(가장 최근에 사용한 페이지의 25% 덤프)입니다.

종료 시 버퍼 풀 상태 저장 및 시작 시 복원하기

서버 종료 시 버퍼 풀의 상태를 저장하려면 서버를 종료하기 전에 다음 문을 실행합니다:

```
GLOBAL innodb_buffer_pool_dump_at_shutdown=ON으로 설정합니다;
```

innodb_buffer_pool_dump_at_shutdown은 기본적으로 활성화되어 있습니다.

서버 시작 시 버퍼 풀 상태를 복원하려면 서버를 시작할 때 `--innodb-buffer-pool-load-at-startup` 옵션을 지정합니다:

```
mysqld --innodb-buffer-pool-load-at-startup=ON;
```

innodb_buffer_pool_load_at_startup은 기본적으로 활성화되어 있습니다.

온라인으로 버퍼 풀 상태 저장 및 복원하기

MySQL 서버가 실행되는 동안 버퍼 풀의 상태를 저장하려면 다음 문을 실행합니다:

```
GLOBAL innodb_buffer_pool_dump_now=ON으로 설정합니다;
```

MySQL이 실행되는 동안 버퍼 풀 상태를 복원하려면 다음 문을 실행합니다:

```
GLOBAL innodb_buffer_pool_load_now=ON으로 설정합니다;
```

버퍼 풀 덤프 진행률 표시

버퍼 풀 상태를 디스크에 저장할 때 진행률을 표시하려면 다음 문을 실행합니다:

```
'Innodb_buffer_pool_dump_status'와 같은 상태를 표시합니다;
```

작업이 아직 시작되지 않은 경우 "시작되지 않음"이 반환됩니다. 작업이 완료되면 완료 시간이 인쇄됩니다(예: 110505 12:18:02에 완료됨). 작업이 진행 중이면 상태 정보가 제공됩니다(예: 버퍼 풀 5/7 덤핑 중, 237/2873 페이지).

버퍼 풀 로드 진행률 표시

버퍼 풀을 로드할 때 진행률을 표시하려면 다음 문을 실행합니다:

```
'Innodb_buffer_pool_load_status'와 같은 상태를 표시합니다;
```

작업이 아직 시작되지 않은 경우 "시작되지 않음"이 반환됩니다. 작업이 완료되면 완료 시간이 인쇄됩니다(예: 110505 12:23:24에 완료됨). 작업이 진행 중이면 상태 정보가 제공됩니다(예: 로드된 123/22301 페이지).

버퍼 풀 로드 작업 중단

버퍼 풀 로드 작업을 중단하려면 다음 문을 실행합니다:

```
GLOBAL innodb_buffer_pool_load_abort=ON으로 설정합니다;
```

성능 스키마를 사용한 버퍼 풀 로드 진행 상황 모니터링

성능 스키마를 사용하여 버퍼 풀 로드 진행 상황을 모니터링할 수 있습니다.

다음 예제에서는 stage/innodb/버퍼 풀 로드 단계 이벤트 계측기와 관련 소비자 테이블을 활성화하여 버퍼 풀 로드 진행 상황을 모니터링하는 방법을 설명합니다.

이 예제에서 사용된 버퍼 풀 덤프 및 로드 절차에 대한 자세한 내용은 다음을 참조하십시오.
섹션 15.8.3.6, "버퍼 풀 상태 저장 및 복원"을 참조하십시오. 퍼포먼스 스키마 스테이지 이벤트 인스트루먼트 및 관련 컨슈머에 대한 자세한 내용은 27.12.5절. "퍼포먼스 스키마 스테이지 이벤트 테이블"을 참조하십시오.

1. 스테이지/인노드/버퍼 풀 로드 기기를 활성화합니다:

```
mysql> UPDATE performance_schema.setup_instruments SET ENABLED = 'YES'
       WHERE NAME LIKE 'stage/innodb/buffer%';
```

2. 이벤트_스테이지_현재, 이벤트_스테이지_역사, 이벤트_스테이지_역사_길이를 포함하는 스테이지 이벤트 소비자 테이블을 사용하도록 설정합니다.

```
mysql> UPDATE performance_schema.setup_consumers SET ENABLED = 'YES'
       WHERE NAME LIKE '%stages%';
```

3. innodb_buffer_pool_dump_now를 활성화하여 현재 버퍼 풀 상태를 덤프합니다.

```
mysql> SET GLOBAL innodb_buffer_pool_dump_now=ON;
```

4. 버퍼 풀 덤프 상태를 확인하여 작업이 완료되었는지 확인합니다.

```
mysql> SHOW STATUS LIKE 'Innodb_buffer_pool_dump_status'\G
*************************** 1. 행 *************************** 변수_이름:
Innodb_버퍼_풀_덤프_상태
        Value: 150202 16:38:58에 버퍼 풀(들) 덤프가 완료되었습니다.
```

5. innodb_buffer_pool_load_now를 활성화하여 버퍼 풀을 로드합니다:

```
mysql> SET GLOBAL innodb_buffer_pool_load_now=ON;
```

6. 성능 스키마 events_stages_current 테이블을 쿼리하여 버퍼 풀 로드 작업의 현재 상태를 확인합니다. WORK_COMPLETED 열에는 로드된 버퍼 풀 페이지 수가 표시됩니다. WORK_ESTIMATED 열은 남은 작업의 예상치를 페이지 단위로 제공합니다.

```
mysql> SELECT EVENT_NAME, WORK_COMPLETED, WORK_ESTIMATED
       FROM performance_schema.events_stages_current;
+-------------------------------+----------------+---------------------+
| 이벤트_이름                     | 작업_완료 | 작업_추정 |
+-------------------------------+----------------+---------------------+
| stage/innodb/버퍼 풀 로드                    |5353            |7167 |
+-------------------------------+----------------+---------------------+
```

버퍼 풀 로드 작업이 완료된 경우 `events_stages_current` 테이블은 빈 집합을 반환합니다. 이 경우 `events_stages_history` 테이블을 확인하여 완료된 이벤트의 데이터를 확인할 수 있습니다. 예를 들어

```
mysql> SELECT EVENT_NAME, WORK_COMPLETED, WORK_ESTIMATED
       FROM performance_schema.events_stages_history;
+-------------------------------+----------------+---------------------+
| 이벤트_이름                    | 작업_완료 | 작업_추정 |
+-------------------------------+----------------+---------------------+
스테이지/인노드/버퍼 풀 로드                        |7167 |7167       |7167 |7167
+-------------------------------+----------------+---------------------+
```

참고

시작 시 innodb_buffer_pool_load_at_startup을 사용하여 버퍼 풀을 로드할 때 성능 스키마를 사용하여 버퍼 풀 로드 진행 상황을 모니터링할 수도 있습니다. 이 경우, 시작 시 스테이지/이노드/버퍼 풀 로드 계측기 및 관련 컨슈머가 활성화되어 있어야 합니다. 자세한 내용은 섹션 27.3, "성능 스키마 시작 구성"을 참조하세요.

15.8.3.7 핵심 파일에서 버퍼 풀 페이지 제외하기

코어 파일은 실행 중인 프로세스의 상태와 메모리 이미지를 기록합니다. 버퍼 풀은 주 메모리에 있고 실행 중인 프로세스의 메모리 이미지는 코어 파일에 덤프되므로, 버퍼 풀이 큰 시스템에서는 mysqld 프로세스가 종료될 때 대용량 코어 파일이 생성될 수 있습니다.

대용량 핵심 파일은 작성에 걸리는 시간, 디스크 공간을 차지하는 양, 대용량 파일 전송과 관련된 문제 등 여러 가지 이유로 문제가 될 수 있습니다.

코어 파일 크기를 줄이려면 innodb_buffer_pool_in_core_file 변수를 비활성화하여 코어 덤프에서 버퍼 풀 페이지를 생략할 수 있습니다. 이 변수는 기본적으로 활성화되어 있습니다.

디버깅 목적으로 조직 내부 또는 외부에서 공유할 수 있는 핵심 파일에 데이터베이스 페이지를 덤프하는 것이 우려되는 경우 보안 측면에서도 버퍼 풀 페이지를 제외하는 것이 바람직할 수 있습니다.

참고

일부 디버깅 시나리오에서는 mysqld 프로세스가 종료된 시점의 버퍼 풀 페이지에 있는 데이터에 액세스하는 것이 유용할 수 있습니다. 버퍼 풀 페이지를 포함할지 제외할지 확실하지 않은 경우 MySQL 지원팀에 문의하세요.

innodb_buffer_pool_in_core_file을 비활성화하면 core_file 변수가 활성화되어 있고 운영 체제가 Linux 3.4 이상에서 지원되는 madvise() 시스템 호출에 대한 MADV_DONTDUMP 비 POSIX 확장을 지원하는 경우에만 적용됩니다. MADV_DONTDUMP 확장을 사용하면 지정된 범위의 페이지가 코어 덤프에서 제외됩니다.

운영 체제가 MADV_DONTDUMP 확장을 지원한다고 가정하고, 서버를 시작하려면 --코어 파일 및 --innodb-버퍼-풀-인-코어 파일=OFF 옵션을 사용하여 버퍼 풀 페이지 없이 코어 파일을 생성할 수 있습니다.

```
$> mysqld --core-file --innodb-buffer-pool-in-core-file=OFF
```

core_file 변수는 기본적으로 읽기 전용이며 비활성화되어 있습니다. 시작 시 --core- file 옵션을 지정하면 활성화됩니다. innodb_buffer_pool_in_core_file 변수는 동적입니다. 시작 시 지정하거나 SET 문을 사용하여 런타임에 구성할 수 있습니다.

```
mysql> SET GLOBAL innodb_buffer_pool_in_core_file=OFF;
```

innodb_buffer_pool_in_core_file 변수가 비활성화되어 있지만 운영 체제에서 MADV_DONTDUMP를 지원하지 않거나 madvise() 오류가 발생하면 MySQL 서버 오류 로그에 경고가 기록되고, 의도하지 않게 버퍼 풀 페이지를 포함하는 코어 파일을 쓰지 않도록 core_file 변수가 비활성화됩니다. 읽기 전용 core_file 변수가 비활성화되면 다시 활성화하려면 서버를 다시 시작해야 합니다.

다음 표에는 코어 파일 생성 여부와 버퍼 풀 페이지 포함 여부를 결정하는 구성 및 MADV_DONTDUMP 지원 시나리오가 나와 있습니다.

표 15.4 핵심 파일 구성 시나리오

core_file 변수	innodb_buffer_pool_ 변수	파일 MADV_DONTDUMP 지원	결과
꺼짐(기본값)	결과와 관련이 없음	결과와 관련이 없음	코어 파일이 생성되지 않음
켜기	켜기(기본값)	결과와 관련이 없음	버퍼 풀 페이지로 코어 파일 생성
켜기	꺼짐	예	버퍼 풀 페이지 없이 코어 파일 생성
켜기	꺼짐	아니요	코어 파일이 생성되지 않고 코어_파일이 비활성화되며 서버 오류 로그에 경고가 기록됩니다.

innodb_buffer_pool_in_core_file 변수를 비활성화하면 코어 파일 크기가 줄어드는 것은 버퍼 풀의 크기에 따라 다르지만 InnoDB 페이지 크기에도 영향을 받습니다. 페이지 크기가 작을수록 같은 양의 데이터에 더 많은 페이지가 필요하며, 페이지가 많을수록 페이지 메타데이터가 많아집니다. 다음 표는 페이지 크기가 다른 1GB 버퍼 풀에 대해 볼 수 있는 크기 감소 예시를 제공합니다.

표 15.5 버퍼 풀 페이지 포함 및 제외된 코어 파일 크기

innodb_page_size 설정	버퍼 풀 페이지 포함 (innodb_buffer_pool_in_co)	버퍼 풀 페이지 제외 (ei_nfnioldeb=_ObNuffer_pool_in_cor e_fil
4KB	2.1GB	0.9GB
64KB	1.7GB	0.7GB

15.8.4 InnoDB의 스레드 동시성 구성

InnoDB는 운영 체제 스레드를 사용하여 사용자 트랜잭션의 요청을 처리합니다. (트랜잭션은 커밋 또는 롤백하기 전에 InnoDB에 많은 요청을 발행할 수 있습니다.) 컨텍스트 전환이 효율적인 최신 운영 체제 및 멀티코어 프로세서가 탑재된 서버에서는 대부분의 워크로드가 동시 스레드 수에 대한 제한 없이 잘 실행됩니다.

스레드 간 컨텍스트 전환을 최소화하는 것이 도움이 되는 상황에서 InnoDB는 여러 가지 기술을 사용하여 동시에 실행되는 운영 체제 스레드의 수(따라서 한 번에 처리되는 요청의 수)를 제한할 수 있습니다. InnoDB가 사용자 세션에서 새 요청을 수신할 때 동시에 실행 중인 스레드 수가 사전 정의된 제한에 도달하면 새 요청은 잠시 동안 절전 모드로 전환된 후 다시 시도됩니다. 절전 모드 이후에도 일정을 변경할 수 없는 요청은 선입선출 대기

열에 넣어져 결국 처리됩니다. 잠금을 기다리는 스레드는 동시에 실행 중인 스레드 수에 포함되지 않습니다.

구성 매개변수 innodb_thread_concurrency를 설정하여 동시 스레드 수를 제한할 수 있습니다. 실행 중인 스레드 수가 이 제한에 도달하면 추가 스레드는 큐에 배치되기 전에 구성 매개변수 innodb_thread_sleep_delay로 설정한 마이크로초 동안 절전 모드로 전환됩니다.

구성 옵션 innodb_adaptive_max_sleep_delay를 innodb_thread_sleep_delay에 허용할 수 있는 가장 높은 값으로 설정하면 InnoDB가 자동으로 조정합니다.

현재 스레드 스케줄링 활동에 따라 innodb_thread_sleep_delay를 높이거나 낮춥니다. 이 동적 조정은 시스템 부하가 적을 때와 최대 용량에 가깝게 작동할 때 스레드 스케줄링 메커니즘이 원활하게 작동하는 데 도움이 됩니다.

MySQL 및 InnoDB의 여러 릴리스에서 innodb_thread_concurrency의 기본값과 동시 스레드 수에 대한 암시적 기본 제한이 변경되었습니다. innodb_thread_concurrency의 기본값은 0이므로 기본적으로 동시 실행 스레드 수에 대한 제한이 없습니다.

InnoDB는 동시 스레드 수가 제한되어 있는 경우에만 스레드를 절전 모드로 전환합니다. 스레드 수에 제한이 없는 경우, 모든 스레드가 동일하게 스케줄링되기 위해 경쟁합니다. 즉, innodb_thread_concurrency가 0 이면 innodb_thread_sleep_delay 값은 무시됩니다.

스레드 수에 제한이 있는 경우(innodb_thread_concurrency가 0보다 큰 경우) InnoDB는 *단일 SQL 문을* 실행하는 동안 이루어진 여러 요청이 다음에서 설정한 제한을 준수하지 않고 InnoDB에 들어갈 수 있도록 허용하여 컨텍스트 전환 오버헤드를 줄입니다. innodb_thread_concurrency. InnoDB 내에서 SQL 문(예: 조인)은 여러 행 작업으로 구성될 수 있으므로, InnoDB는 최소한의 오버헤드로 스레드를 반복적으로 예약할 수 있는 지정된 수의 '티켓'을 할당합니다.

새 SQL 문이 시작될 때, 스레드에는 티켓이 없으며, innodb_thread_concurrency를 준수해야 합니다. 스레드에 InnoDB에 들어갈 수 있는 권한이 부여되면, 이후에 InnoDB에 들어가 행 작업을 수행하는 데사용할 수 있는 티켓이 할당됩니다. 티켓이
가 소진되면 스레드가 퇴출되고, innodb_thread_concurrency가 다시 관찰되어 스레드가 대기 중인 스레드의 선입 선출 대기열에 다시 배치될 수 있습니다. 스레드가 다시 한 번 InnoDB에 들어갈 수 있는 권한이 부여되면 티켓이 다시 할당됩니다. 할당된 티켓의 수는 전역 옵션 innodb_concurrency_tickets에 의해 지정되며, 기본값은 5000개입니다. 잠금을 대기 중인 스레드에는 잠금을 사용할 수 있게 되면 하나의 티켓이 할당됩니다.

이러한 변수의 올바른 값은 환경과 워크로드에 따라 다릅니다. 다양한 값을 시도하여 애플리케이션에 적합한 값을 결정하세요. 동시에 실행되는 스레드 수를 제한하기 전에, 멀티코어 및 멀티프로세서 컴퓨터에서 InnoDB의 성능을 향상시킬 수 있는 구성 옵션(예: innodb_adaptive_hash_index)을 검토하세요.

MySQL 스레드 처리에 대한 일반적인 성능 정보는 5.1.12.1절. '연결 인터페이스'를 참조하세요.

15.8.5 백그라운드 InnoDB I/O 스레드 수 구성하기

InnoDB는 백그라운드 스레드를 사용하여 다양한 유형의 I/O 요청을 처리합니다. 데이터 페이지에서 읽기 및 쓰기 I/O를 서비스하는 백그라운드 스레드 수를 구성할 수 있습니다. innodb_read_io_threads 및 innodb_write_io_threads 구성 매개변수입니다. 이 매개변수는 각각 읽기 및 쓰기 요청에 사용되는 백그라운드 스레드 수를 나타냅니다. 이 매개변수는 지원되는 모든 플랫폼에서 유효합니다. 이러한 매개변수의 값은 MySQL 옵션 파일(my.cnf 또는 my.ini)에서 설정할 수 있으며, 값을 동적으로 변경할 수는 없습니다. 이러한 매개변수의 기본값은 4이며 허용되는 값의 범위는 1~64입니다.

이러한 구성 옵션의 목적은 하이엔드 시스템에서 InnoDB의 확장성을 높이기 위한 것입니다. 각 백그라운드 스레드는 최대 256개의 보류 중인 I/O 요청을 처리할 수 있습니다. 백그라운드 I/O의 주요 원인은 읽기 전 요청입니다. InnoDB는 대부분의 백그라운드 스레드가 작업을 균등하게 공유하는 방식으로 들어오는 요청의 부하를 분산하려고 시도합니다. 또한 InnoDB는 요청을 통합할 가능성을 높이기 위해 동일한 범위의 읽기 요청을 동일한 스레드에 할당하려고 시도합니다. 고급 I/O 하위 시스템을 사용하는 경우, 엔진 INNODB 상태 표시 출력에 읽기 요청 보류가 64개 이상의 innodb_read_io_threads가 표시되는 경우, innodb_read_io_threads 값을 늘려 성능을 개선할 수 있습니다.

Linux 시스템에서 InnoDB는 기본적으로 비동기 I/O 하위 시스템을 사용하여 데이터 파일 페이지에 대한 읽기 및 쓰기 요청을 수행하며, 이로 인해 InnoDB 백그라운드 스레드가 서비스하는 방식이 변경됩니다.

이러한 유형의 I/O 요청. 자세한 내용은 섹션 15.8.6, "Linux에서 비동기 I/O 사용"을 참조하세요.

InnoDB I/O 성능에 대한 자세한 내용은 섹션 8.5.8, "InnoDB 디스크 I/O 최적화"를 참조하세요.

15.8.6 Linux에서 비동기 I/O 사용

InnoDB는 Linux에서 비동기 I/O 하위 시스템(기본 AIO)을 사용하여 데이터 파일 페이지에 대한 읽기 및 쓰기 요청을 수행합니다. 이 동작은 Linux 시스템에만 적용되며 기본적으로 활성화되어 있는 innodb_use_native_aio 구성 옵션에 의해 제어됩니다. 다른 Unix-시스템과 마찬가지로, InnoDB는 동기식 I/O만 사용합니다. 이전까지 InnoDB는 Windows 시스템에서만 비동기 I/O를 사용했습니다. Linux에서 비동기 I/O 하위 시스템을 사용하려면 libaio 라이브러리가 필요합니다.

동기식 I/O를 사용하면 쿼리 스레드가 I/O 요청을 큐에 대기시키고, InnoDB 백그라운드 스레드가 큐에 대기 중인 요청을 한 번에 하나씩 검색하여 각각에 대해 동기식 I/O 호출을 실행합니다. I/O 요청이 가 완료되고 I/O 호출이 반환되면 요청을 처리 중인 InnoDB 백그라운드 스레드가 I/O 완료 루틴을 호출하고 다음 요청을 처리하기 위해 반환합니다. 병렬로 처리할 수 있는 요청의 수는 *n이며,* 여기서 *n은* InnoDB 백그라운드 스레드의 수입니다.
InnoDB 백그라운드 스레드 수는 innodb_read_io_threads 및 innodb_write_io_threads에 의해 제어됩니다. 15.8.5절, "백그라운드 InnoDB I/O 스레드 수 구성하기"를 참조하세요.

기본 AIO를 사용하면 쿼리 스레드가 I/O 요청을 운영 체제에 직접 디스패치하므로 백그라운드 스레드 수에 따른 제한이 제거됩니다. InnoDB 백그라운드 스레드는 I/O 이벤트가 완료된 요청을 알릴 때까지 기다립니다. 요청이 완료되면 백그라운드 스레드는 I/O 완료 루틴을 호출하고 I/O 이벤트 대기를 재개합니다.

네이티브 AIO의 장점은 일반적으로 엔진 INNODB 상태\G 출력에 보류 중인 읽기/쓰기가 많이 표시되는 I/O 바인딩이 많은 시스템에 대한 확장성입니다. 네이티브 AIO를 사용할 때 병렬 처리가 증가한다는 것은 I/O 스케줄러의 유형이나 디스크 어레이 컨트롤러의 속성이 I/O 성능에 더 큰 영향을 미친다는 것을 의미합니다.

I/O 바인딩이 많은 시스템에서 네이티브 AIO의 잠재적인 단점은 운영 체제에 한 번에 전송되는 I/O 쓰기 요청의 수를 제어할 수 없다는 것입니다. 병렬 처리를 위해 운영 체제로 전송되는 I/O 쓰기 요청이 너무 많으면 경우에 따라 I/O 활동량과 시스템 성능에 따라 I/O 읽기 고갈이 발생할 수 있습니다.

OS의 비동기 I/O 하위 시스템 문제로 인해 InnoDB가 시작되지 않는 경우 innodb_use_native_aio=0으로 서버를 시작할 수 있습니다. 또한 InnoDB가 tmpdir 위치, tmpfs 파일 시스템 및 tmpfs에서 비동기 I/O를 지원하지 않는 Linux 커널의 조합과 같은 잠재적인 문제를 감지하는 경우 이 옵션이 시작 중에 자동으로 비활성화 될 수 있습니다.

15.8.7 InnoDB I/O 용량 구성

InnoDB 마스터 스레드 및 기타 스레드는 백그라운드에서 다양한 작업을 수행하며, 대부분은 버퍼 풀에서 더티 페이지를 플러시하고

버퍼를 적절한 보조 인덱스로 변경합니다. InnoDB는 서버의 정상적인 작동에 악영향을 미치지 않는 방식으로 이러한 작업을 수행하려고 시도합니다. 사용 가능한 I/O 대역폭을 추정하고 사용 가능한 용량을 활용하도록 활동을 조정하려고 시도합니다.

innodb_io_capacity 변수는 InnoDB에서 사용할 수 있는 전체 I/O 용량을 정의합니다. 시스템이 초당 수행할 수 있는 I/O 작업 수(IOPS)와 비슷하게 설정해야 합니다. innodb_io_capacity를 설정하면 InnoDB는 설정된 값을 기반으로 백그라운드 작업에 사용할 수 있는 I/O 대역폭을 추정합니다.

innodb_io_capacity를 100 이상의 값으로 설정할 수 있습니다. 기본값은 200입니다. 일반적으로 최대 7200RPM의 하드 드라이브와 같은 소비자 수준의 저장 장치에는 100 정도의 값이 적합합니다. 더 빠른 하드 드라이브, RAID 구성 및 SSD(솔리드 스테이트 드라이브)는 더 높은 값을 사용하면 이점이 있습니다.

이 설정은 가능한 한 낮게 유지하는 것이 이상적이지만, 백그라운드 활동이 뒤처질 정도로 낮게 설정해서는 안 됩니다. 값이 너무 높으면 버퍼 풀에서 데이터가 제거되고 버퍼가 너무 빨리 변경되어 캐싱이 큰 이점을 제공하지 못합니다. 더 높은 I/O 속도를 처리할 수 있는 바쁜 시스템의 경우 더 높은 값을 설정하여 서버가 높은 행 속도와 관련된 백그라운드 유지 관리 작업을 처리할 수 있도록 할 수 있습니다.
변경합니다. 일반적으로 InnoDB에 사용되는 드라이브 수에 따라 값을 늘릴 수 있습니다.
I/O. 예를 들어 여러 디스크 또는 SSD를 사용하는 시스템에서 값을 늘릴 수 있습니다.

일반적으로 저사양 SSD의 경우 기본 설정인 200이면 충분합니다. 고급형 버스 연결 SSD의 경우, 예를 들어 1000과 같이 더 높은 설정을 고려하세요. 개별 5400RPM 또는 7200RPM 드라이브가 있는 시스템의 경우, 이 값을 100으로 낮출 수 있는데, 이는 약 100 IOPS를 수행할 수 있는 이전 세대 디스크 드라이브에서 사용할 수 있는 초당 입출력 작업(IOPS)의 예상 비율을 나타냅니다.

100만과 같이 높은 값을 지정할 수 있지만 실제로는 그렇게 큰 값은 거의 이점이 없습니다. 일반적으로 20000보다 낮은 값으로는 워크로드에 충분하지 않다고 확신하는 경우가 아니라면 20000보다 높은 값은 사용하지 않는 것이 좋습니다.

innodb_io_capacity를 조정할 때 쓰기 워크로드를 고려하세요. 쓰기 워크로드가 많은 시스템에서는 이 값을 높게 설정하는 것이 좋습니다. 쓰기 워크로드가 적은 시스템에서는 낮은 설정으로도 충분할 수 있습니다.

innodb_io_capacity 설정은 버퍼 풀 인스턴스별 설정이 아닙니다. 사용 가능한 I/O 용량은 플러시 작업을 위해 버퍼 풀 인스턴스 간에 균등하게 분배됩니다.

MySQL 옵션 파일(my.cnf 또는 my.ini)에서 innodb_io_capacity 값을 설정하거나, 글로벌 시스템 변수를 설정할 수 있는 충분한 권한이 필요한 SET GLOBAL 문을 사용하여 런타임에 수정할 수 있습니다. 섹션 5.1.9.1, "시스템 변수 권한"을 참조하세요.

체크포인트에서 I/O 용량 무시하기

기본적으로 활성화되어 있는 innodb_flush_sync 변수를 사용하면 체크포인트에서 발생하는 I/O 활동 버스트 중에 innodb_io_capacity 설정이 무시됩니다. innodb_io_capacity 설정에 정의된 I/O 속도를 준수하려면 innodb_flush_sync를 비활성화합니다.

MySQL 옵션 파일(my.cnf 또는 my.ini)에서 innodb_flush_sync 값을 설정하거나, 글로벌 시스템 변수를 설정할 수 있는 충분한 권한이 필요한 SET GLOBAL 문을 사용하여 런타임에 수정할 수 있습니다. 섹션 5.1.9.1, "시스템 변수 권한"을 참조하세요.

최대 I/O 용량 구성

플러싱 활동이 뒤처지면 InnoDB는 더 높은 비율의 I/
O 초당 작업 수(IOPS)를 초과할 수 있습니다. innodb_io_capacity_max 변수는 이러한 상황에서 InnoDB 백그라운드 작업에서 수행되는 최대 IOPS 수를 정의합니다.

시작 시 innodb_io_capacity 설정을 지정하고 innodb_io_capacity_max 값을 지정하지 않으면 기

본적으로 innodb_io_capacity_max 값의 두 배 또는 2000 중 큰 값으로 지정됩니다.

innodb_io_capacity_max를 구성할 때는 보통 innodb_io_capacity의 2배가 좋은 시작점입니다. 기본값인 2000은 SSD 또는 둘 이상의 일반 디스크 드라이브를 사용하는 워크로드를 위한 것입니다. SSD 또는 여러 개의 디스크 드라이브를 사용하지 않는 워크로드에 2000을 설정하면 너무 높을 수 있으며, 너무 많은 플러싱을 허용할 수 있습니다. 일반 디스크 드라이브가 하나인 경우 200에서 400 사이로 설정하는 것이 좋습니다. 고급 버스 연결 SSD의 경우 2500과 같이 더 높은 설정을 고려하십시오. innodb_io_capacity 설정과 마찬가지로, 이 설정은 가능한 한 낮게 유지하되, InnoDB가 innodb_io_capacity 설정 이상으로 IOPS 속도를 충분히 확장할 수 없을 정도로 낮게 설정해서는 안 됩니다.

innodb_io_capacity_max를 조정할 때 쓰기 워크로드를 고려하세요. 쓰기 워크로드가 큰 시스템에서는 이 값을 높게 설정하는 것이 좋습니다. 쓰기 워크로드가 적은 시스템에서는 이보다 낮은 설정으로도 충분할 수 있습니다.

innodb_io_capacity_max 값보다 낮은 값으로 설정할 수 없습니다.

SET 문을 사용하여 innodb_io_c용량_max를 DEFAULT로 설정하면(SET GLOBAL innodb_io_c용량_max=DEFAULT) innodb_io_c용량_max가 최대값으로 설정됩니다.

innodb_io_capacity_max 제한은 모든 버퍼 풀 인스턴스에 적용됩니다. 버퍼 풀 인스턴스별 설정이 아닙니다.

15.8.8 스핀 잠금 폴링 구성

InnoDB 뮤텍스와 rw-락은 일반적으로 짧은 간격으로 예약됩니다. 멀티코어 시스템에서는 스레드가 일정 기간 동안 뮤텍스 또는 rw-잠금을 획득할 수 있는지 지속적으로 확인하는 것이 더 효율적일 수 있습니다.
의 시간을 잠자기 상태로 유지합니다. 이 기간 동안 뮤텍스 또는 rw-lock을 사용할 수 있게 되면 스레드는 동일한 시간 슬라이스에서 즉시 계속할 수 있습니다. 그러나 여러 스레드가 뮤텍스나 rw-lock과 같은 공유 객체를 너무 자주 폴링하면 프로세서가 서로의 캐시 일부를 무효화하는 '캐시 핑퐁'이 발생할 수 있습니다. InnoDB는 폴링 활동의 동기화를 해제하기 위해 폴링 사이에 무작위 지연을 강제함으로써 이 문제를 최소화합니다. 무작위 지연은 스핀-대기 루프로 구현됩니다.

스핀-대기 루프의 지속 시간은 루프에서 발생하는 PAUSE 명령의 수에 따라 결정됩니다. 이 숫자는 0에서 innodb_spin_wait_delay 값을 포함하지 않는 범위의 정수를 임의로 선택하고 해당 값에 50을 곱하여 생성됩니다. 예를 들어, innodb_spin_wait_delay 설정이 6인 경우 다음 범위에서 정수가 임의로 선택됩니다:

```
{0,1,2,3,4,5}
```

선택한 정수에 50을 곱하면 6가지 가능한 PAUSE 명령어 값 중 하나가 생성됩니다:

```
{0,50,100,150,200,250}
```

이 값 집합의 경우 250은 스핀 대기 루프에서 발생할 수 있는 최대 PAUSE 명령어 수입니다.
innodb_spin_wait_delay를 5로 설정하면 5개의 가능한 값 집합이 생성됩니다.
{0,50,100,150,200}, 여기서 200은 최대 PAUSE 명령어 수입니다. 이러한 방식으로 innodb_spin_wait_delay 설정은 스핀 잠금 폴링 사이의 최대 지연을 제어합니다.

모든 프로세서 코어가 빠른 캐시 메모리를 공유하는 시스템에서는 최대 지연을 줄이거나 innodb_spin_wait_delay=0을 설정하여 사용 중 루프를 완전히 비활성화할 수 있습니다. 여러 프로세서 칩이 있는 시스템에서는 캐시 무효화의 효과가 더 클 수 있으며 최대 지연이 증가할 수 있습니다.

100MHz 펜티엄 시대에는 innodb_spin_wait_delay 단위가 1마이크로초와 동등하도록 보정되었습니다. 이 시간 등가성은 유지되지 않았지만, 상대적으로 더 긴 PAUSE 명령어를 가진 스카이레이크 세대의 프로세서가 도입될 때까지 다른 CPU 명령어에 비해 프로세서 주기 측면에서 PAUSE 명령어 지속 시간은 상당히 일정하게 유지되었습니다. innodb_spin_wait_pause_multiplier 변수는 PAUSE 명령

지속 시간의 차이를 설명할 수 있는 방법을 제공합니다.

`innodb_spin_wait_pause_multiplier` 변수는 PAUSE 명령 값의 크기를 제어합니다. 예를 들어, `innodb_spin_wait_delay` 설정이 6이라고 가정할 때, `innodb_spin_wait_pause_multiplier` 값을 50(기본값이자 이전에 하드코딩된 값)에서 5로 줄이면 더 작은 PAUSE 명령 값 집합을 생성할 수 있습니다:

```
{0,5,10,15,20,25}
```

PAUSE 명령어 값을 늘리거나 줄일 수 있는 기능을 통해 다양한 프로세서 아키텍처에 맞게 `InnoDB`를 미세 조정할 수 있습니다. 예를 들어, 비교적 긴 PAUSE 명령어를 사용하는 프로세서 아키텍처에는 더 작은 PAUSE 명령어 값이 적합할 수 있습니다.

`innodb_spin_wait_delay` 및 `innodb_spin_wait_pause_multiplier` 변수는 동적입니다. MySQL 옵션 파일에서 지정하거나 런타임에 `SET GLOBAL`을 사용하여 수정할 수 있습니다.

문을 사용해야 합니다. 런타임에 변수를 수정하려면 전역 시스템 변수를 설정할 수 있는 충분한 권한이 필요합니다. 섹션 5.1.9.1, "시스템 변수 권한"을 참조하십시오.

15.8.9 퍼지 구성

InnoDB는 SQL 문으로 행을 삭제할 때 데이터베이스에서 행을 즉시 물리적으로 제거하지 않습니다. 행과 해당 인덱스 레코드는 `InnoDB`가 삭제를 위해 기록된 실행 취소 로그 레코드를 삭제할 때만 물리적으로 제거됩니다. 다중 버전 동시성 제어(MVCC) 또는 롤백에 더 이상 행이 필요하지 않은 경우에만 발생하는 이 제거 작업을 퍼지라고 합니다.

퍼지는 주기적인 일정에 따라 실행됩니다. `InnoDB` 트랜잭션 시스템에 의해 유지 관리되는 커밋된 트랜잭션의 실행 취소 로그 페이지 목록인 기록 목록에서 실행 취소 로그 페이지를 구문 분석하고 처리합니다. Purge는 실행 취소 로그 페이지를 처리한 후 기록 목록에서 해당 페이지를 해제합니다.

퍼지 스레드 구성

퍼지 작업은 하나 이상의 퍼지 스레드에 의해 백그라운드에서 수행됩니다. 제거 스레드 수는 `innodb_purge_threads` 변수에 의해 제어됩니다. 기본값은 4입니다.

DML 작업이 단일 테이블에 집중되어 있는 경우 테이블에 대한 제거 작업이 단일 제거 스레드에 의해 수행되므로 제거 작업이 느려지고 제거 지연이 증가하며 DML 작업에 큰 개체 값이 포함된 경우 테이블 공간 파일 크기가 증가할 수 있습니다. `innodb_max_purge_lag` 설정을 초과하면 사용 가능한 퍼지 스레드 간에 퍼지 작업이 자동으로 재분배됩니다. 이 시나리오에서 활성 제거 스레드가 너무 많으면 사용자 스레드와의 경합이 발생할 수 있으므로 `innodb_purge_threads` 설정을 적절히 관리합니다. `innodb_max_purge_lag` 변수는 기본적으로 0으로 설정되어 있으며, 이는 기본적으로 최대 퍼지 지연이 없음을 의미합니다.

DML 작업이 소수의 테이블에 집중되어 있는 경우, 사용 중인 테이블에 액세스하기 위해 스레드가 서로 경합하지 않도록 `innodb_purge_threads` 설정을 낮게 유지합니다. DML 작업이 여러 테이블에 분산되어 있는 경우 `innodb_purge_threads` 설정을 더 높게 고려하십시오. 제거 스레드의 최대 개수는 32개입니다.

`innodb_purge_threads` 설정은 허용되는 최대 퍼지 스레드 수입니다. 퍼지 시스템은 사용되는 퍼지 스레드 수를 자동으로 조정합니다.

퍼지 배치 크기 구성

`innodb_purge_batch_size` 변수는 기록 목록에서 구문 분석 및 처리를 한 번에 제거하는 실행 취소 로그 페이지의 수를 정의합니다. 기본값은 300입니다. 멀티스레드 제거 구성에서 코디네이터 제거 스레드는 `innodb_purge_batch_size`를 `innodb_purge_threads`로 나누고 각 제거 스레드에 해당 페이지 수를 할당합니다.

또한 퍼지 시스템은 더 이상 필요하지 않은 실행 취소 로그 페이지를 해제합니다. 실행 취소 로그를 128번

반복할 때마다 수행됩니다. 일괄 처리에서 구문 분석 및 처리되는 실행 취소 로그 페이지 수를 정의하는 것 외에도, `innodb_purge_batch_size` 변수는 실행 취소 로그를 통해 128회 반복할 때마다 제거가 해제되는 실행 취소 로그 페이지 수를 정의합니다.

`innodb_purge_batch_size` 변수는 고급 성능 튜닝 및 실험을 위한 것입니다. 대부분의 사용자는 `innodb_purge_batch_size`를 기본값에서 변경할 필요가 없습니다.

최대 퍼지 지연 구성

`innodb_max_purge_lag` 변수는 원하는 최대 퍼지 지연을 정의합니다. 퍼지 지연이 `innodb_max_purge_lag` 임계값을 초과하면 `INSERT`, `UPDATE` 및 삭제 작업을 수행하여 퍼지 작업이 따라잡을 수 있는 시간을 허용합니다. 기본값은 0으로, 최대 퍼지 지연이 없고 지연이 발생하지 않습니다.

`InnoDB` 트랜잭션 시스템은 업데이트 또는 삭제 작업에 의해 인덱스 레코드가 삭제 표시된 트랜잭션의 목록을 유지 관리합니다. 목록의 길이는 퍼지 지연입니다.

퍼지 지연 지연은 다음 공식으로 계산됩니다:

```
(PURGE_LAG/INNODB_MAX_PURGE_LAG - 0.9995) * 10000
```

지연은 퍼지 배치가 시작될 때 계산됩니다.

트랜잭션의 크기가 100바이트에 불과하고 제거되지 않은 테이블 행의 크기가 100MB라고 가정할 때 문제가 있는 워크로드에 대한 일반적인 innodb_max_purge_lag 설정은 1000000(100만)일 수 있습니다.

퍼지 지연은 트랜잭션의 트랜잭션 섹션에 히스토리 목록 길이 값으로 표시됩니다.
엔진 INNODB 상태 표시 출력.

```
mysql> SHOW 엔진 인노드 상태;
...
--------------
트랜잭션
--------------
Trx ID 카운터 0 290328385
trx의 n:o < 0에 대해 퍼지 완료 290315608 undo n:o < 0 17
히스토리 목록 길이 20
```

히스토리 목록 길이는 일반적으로 수천 개 미만의 낮은 값이지만 쓰기 작업량이 많거나 트랜잭션이 오래 실행되면 다음과 같은 트랜잭션의 경우에도 이 값이 증가할 수 있습니다.
는 읽기 전용입니다. 오래 실행되는 트랜잭션으로 인해 히스토리 목록 길이가 늘어날 수 있는 이유는 반복 읽기 같은 일관된 읽기 트랜잭션 격리 수준에서 트랜잭션이 해당 트랜잭션의 읽기 보기를 만들 때와 동일한 결과를 반환해야 하기 때문입니다.
따라서 InnoDB 다중 버전 동시성 제어(MVCC) 시스템은 해당 데이터에 의존하는 모든 트랜잭션이 완료될 때까지 실행 취소 로그에 데이터의 사본을 보관해야 합니다. 다음은 히스토리 목록 길이를증가시킬 수 있는 장기 실행 트랜잭션의 예입니다:

- 동시 DML의 양이 상당히 많은 상태에서 --single-transaction 옵션을 사용하는 mysqldump 작업입니다.
- 자동 커밋을 비활성화한 후 SELECT 쿼리를 실행하고 명시적인 커밋 또는 롤백.

퍼지 지연이 커지는 극단적인 상황에서 과도한 지연을 방지하려면
innodb_max_purge_lag_delay 변수를 설정하여 지연을 제한할 수 있습니다. 이 변수는
innodb_max_purge_lag_delay 변수는 innodb_max_purge_lag 임계값을 초과할 때 부과되는 지연의 최대 지연을 마이크로초 단위로 지정합니다. 지정된 innodb_max_purge_lag_delay 값은 innodb_max_purge_lag 공식에 의해 계산된 지연 기간의 상한입니다.

테이블 공간 잘라내기 제거 및 실행 취소

제거 시스템은 실행 취소 테이블스페이스를 잘라내는 작업도 담당합니다.
innodb_purge_rseg_truncate_frequency 변수를 구성하여 제거 시스템에서 잘라낼 실행 취소 테이블스페이스를 찾는 빈도를 제어할 수 있습니다. 자세한 내용은 실행 취소 테이블 공간 잘라내기를 참조하십시오.

15.8.10 InnoDB에 대한 옵티마이저 통계 구성

이 섹션에서는 InnoDB에 대한 영구 및 비영구 옵티마이저 통계를 구성하는 방법에 대해 설명합니다. 테이블.

영구 옵티마이저 통계는 서버를 다시 시작할 때에도 유지되므로 요금제 안정성을 높이고 쿼리 성능을 일관성 있게 유지할 수 있습니다. 또한 영구 옵티마이저 통계는 이러한 추가적인 이점을 통해 제어 및 유연성을 제공합니다:

- innodb_stats_auto_recalc 구성 옵션을 사용하여 테이블을 크게 변경한 후 통계가 자동으로 업데이트되는지 여부를 제어할 수 있습니다.

- STATS_PERSISTENT, STATS_AUTO_RECALC 및 STATS_SAMPLE_PAGES를 사용할 수 있습니다. 절을 CREATE TABLE 및 ALTER TABLE 문과 함께 사용하여 개별 테이블에 대한 옵티마이저 통계를 구성할 수 있습니다.
- mysql.innodb_table_stats에서 옵티마이저 통계 데이터를 쿼리할 수 있습니다. mysql.innodb_index_stats 테이블.
- mysql.innodb_table_stats의 last_update 열과 mysql.innodb_index_stats 테이블에서 통계가 마지막으로 업데이트된 시기를 확인할 수 있습니다.
- mysql.innodb_table_stats 및 mysql.innodb_index_stats 테이블을 수동으로 수정하여 특정 쿼리 최적화 계획을 강제로 적용하거나 데이터베이스를 수정하지 않고 대체 계획을 테스트할 수 있습니다.

영구 옵티마이저 통계 기능은 기본적으로 사용하도록 설정되어 있습니다(innodb_stats_persistent=ON).

비영구 옵티마이저 통계는 서버를 다시 시작할 때마다 그리고 다른 작업 후에 지워지고 다음 테이블 액세스 시 다시 계산됩니다. 따라서 다른 추정치가 생성될 수 있습니다.
를 사용하여 통계를 다시 계산할 때 실행 계획의 선택이 달라지고 쿼리 성능이 달라질 수 있습니다.

이 섹션에서는 정확한 통계와 분석 테이블 실행 시간 간의 균형을 맞추려고 할 때 유용할 수 있는 분석 테이블 복잡성 추정에 대한 정보도 제공합니다.

15.8.10.1 퍼시스턴트 옵티마이저 통계 매개변수 구성

영구 옵티마이저 통계 기능은 통계를 디스크에 저장하고 서버가 다시 시작될 때마다 통계를 영구적으로 유지하여 옵티마이저가 특정 쿼리에 대해 매번 일관된 선택을 할 수 있도록 함으로써 플랜의 안정성을 향상시킵니다.

옵티마이저 통계는 innodb_stats_perististent=ON 또는 개별 테이블이 STATS_PERSISTENT=1로 정의된 경우 디스크에 지속됩니다. innodb_stats_perististent는 기본적으로 활성화됩니다.

이전에는 서버를 재시작하거나 다른 유형의 작업을 수행한 후 옵티마이저 통계가 지워지고 다음 테이블 액세스 시 다시 계산되었습니다. 따라서 통계를 다시 계산할 때 다른 추정치가 생성되어 쿼리 실행 계획의 선택이 달라지고 쿼리 성능에 변동이 생길 수 있었습니다.

영구 통계는 mysql.innodb_table_stats 및 mysql.innodb_index_stats 테이블. InnoDB 영구 통계 테이블을 참조하십시오.

옵티마이저 통계를 디스크에 유지하지 않으려면 섹션 15.8.10.2, "비영구 옵티마이저 통계 매개변수 구성"을 참조하세요.

퍼시스턴트 옵티마이저 통계에 대한 자동 통계 계산 구성

기본적으로 활성화되어 있는 innodb_stats_auto_recalc 변수는 테이블의 행이 10% 이상 변경될 때 통계가 자동으로 계산되는지 여부를 제어합니다. 또한 테이블을 생성하거나 변경할 때 STATS_AUTO_RECALC 절을 지정하여 개별 테이블에 대한 자동 통계 재계산을 구성할 수도 있습니다.

백그라운드에서 발생하는 자동 통계 재계산의 비동기적 특성으로 인해 innodb_stats_auto_recalc가 활성화되어 있어도 테이블의 10% 이상에 영향을 미치는 DML 작업을 실행한 후 통계가 즉시 재계산되지 않을 수 있습니다. 경우에 따라 통계 재계산이 몇 초 지연될 수 있습니다. 최신 통계가 즉시 필요한 경우, 테이블 분석을 실행하여 동기식(포그라운드) 통계 재계산을 시작하십시오.

innodb_stats_auto_recalc가 비활성화되어 있으면 인덱싱된 열을 크게 변경한 후 `ANALYZE TABLE` 문을 실행하여 옵티마이저 통계의 정확성을 보장할 수 있습니다.

데이터를 로드한 후 실행하는 설정 스크립트에 분석 테이블을 추가하고 활동이 적은 시간에 일정에 따라 분석 테이블을 실행하는 것도 고려할 수 있습니다.

기존 테이블에 인덱스가 추가되거나 컬럼이 추가 또는 삭제되는 경우, 인덱스 통계는 innodb_stats_auto_recalc 값에 관계없이 계산되어 innodb_index_stats 테이블에 추가됩니다.

개별 테이블에 대한 옵티마이저 통계 매개 변수 구성

innodb_stats_persistent, innodb_stats_auto_recalc 및 innodb_stats_persistent_sample_pages는 전역 변수입니다. 이러한 시스템 전체 설정을 재정의하고 개별 테이블에 대한 옵티마이저 통계 파라미터를 구성하려면 CREATE에서 STATS_PERSISTENT, STATS_AUTO_RECALC 및 STATS_SAMPLE_PAGES 절을 정의할 수 있습니다.
TABLE 또는 ALTER TABLE 문을 사용합니다.

- STATS_PERSISTENT는 InnoDB 테이블에 대한 영구 통계를 활성화할지 여부를 지정합니다. DEFAULT 값을 사용하면 테이블에 대한 영구 통계 설정이 innodb_stats_persistent 설정에 의해 결정됩니다. 값이 1이면 테이블에 대한 영구 통계가 활성화되고 값이 0이면 이 기능이 비활성화됩니다. 개별 테이블에 대해 영구 통계를 사용하도록설정한 후 테이블 데이터가 로드된 후 테이블 분석을 사용하여 통계를 계산합니다.

- STATS_AUTO_RECALC는 영구 통계를 자동으로 다시 계산할지 여부를 지정합니다. DEFAULT 값을 사용하면테이블에 대한 영구 통계 설정이 innodb_stats_auto_recalc 설정에 의해 결정됩니다. 값이 1이면 테이블 데이터의 10%가 변경될 때 통계가 다시 계산됩니다. 값 0은 테이블에 대한 자동 재계산을 방지합니다. 0 값을 사용하는 경우 테이블을 크게 변경한 후 테이블 분석을 사용하여 통계를 다시 계산합니다.

- STATS_SAMPLE_PAGES는 예를 들어 분석 테이블 작업으로 인덱싱된 열에 대해 카디널리티 및 기타 통계가 계산될 때 샘플링할 인덱스 페이지 수를 지정합니다.

세 절 모두 다음 CREATE TABLE 예제에서 지정됩니다:

```
CREATE TABLE `t1` (
`id` int(8) NOT NULL 자동 증가,
데이터` varchar(255),
날짜` 날짜/시간, 기본 키
(`id`),
INDEX `DATE_IX` (`date`)
) 엔진=InnoDB, 통계_지속성
  =1, 통계_자동재계산=1, 통계_
  샘플_페이지=25;
```

InnoDB 옵티마이저 통계를 위한 샘플 페이지 수 구성하기

옵티마이저는 키 분포에 대한 추정 통계를 사용하여 인덱스의 상대적 선택성에 따라 실행 계획에 사용할 인덱스를 선택합니다. 테이블 분석과 같은 작업을 수행하면 InnoDB가테이블의 각 인덱스에서 무작위로 페이지를 샘플링하여 인덱스의 카디널리티를 추정합니다. 이 샘플링 기법을 랜덤 다이브라고 합니다.

innodb_stats_persistent_sample_pages는 샘플링된 페이지의 수를 제어합니다. 런타임에 설정을 조정하여 옵티마이저에서 사용하는 통계 추정치의 품질을 관리할 수 있습니다. 기본값은 20입니다. 다음과 같은 문제가 발생하면 설정을 수정하는 것이 좋습니다:

1. *통계가 충분히 정확하지 않아 최적화 프로그램이* `EXPLAIN` 출력에 표시된 것처럼 *차선책을 선택합니다.* 인덱스의 실제 카디널리티(인덱스 열에서 `SELECT DISTINCT`를 실행하여 결정됨)와 `mysql.innodb_index_stats` 테이블의 추정치를 비교하여 통계의 정확성을 확인할 수 있습니다.

   통계가 충분히 정확하지 않다고 판단되는 경우에는
   통계 추정치가 나올 때까지 `innodb_stats_perpetistent_sample_pages`를 늘려야 합니다.

는 충분히 정확합니다. 그러나 innodb_stats_persistent_sample_pages를 너무 많이 늘리면 분석 테이블이 느리게 실행될 수 있습니다.

2. *테이블 분석이 너무 느립니다*. 이 경우, 분석 테이블 실행 시간이 허용 가능한 수준이 될 때까지 innodb_stats_persistent_sample_pages 값을 줄여야 합니다. 그러나 값을 너무 많이 줄이면 첫 번째 문제인 부정확한 통계와 차선의 쿼리 실행 계획이 발생할 수 있습니다.

   정확한 통계와 테이블 분석 실행 시간 간에 균형을 맞출 수 없는 경우 테이블의 인덱싱된 열 수를 줄이거나 파티션 수를 제한하여 테이블 분석 복잡성을 줄이는 것을 고려합니다. 고유하지 않은 각 인덱스에 기본 키 열이 추가되므로 테이블의 기본 키 열 수 또한 고려해야 합니다.

   관련 정보는 섹션 15.8.10.3, 'InnoDB 테이블에 대한 분석 테이블 복잡성 추정'을 참조하십시오.

영구 통계 계산에 삭제 표시된 레코드 포함

기본적으로 InnoDB는 통계를 계산할 때 커밋되지 않은 데이터를 읽습니다. 테이블에서 행을 삭제하는 커밋되지 않은 트랜잭션의 경우, 행 추정치 및 인덱스 통계를 계산할 때 삭제 표시가 있는 레코드가 제외되므로 테이블에서 동시에 운영 중인 다른 트랜잭션에 대해 READ UNCOMMITTED 이외의 트랜잭션 격리 수준을 사용하여 최적의 실행 계획이 되지 않을 수 있습니다. 이 시나리오를 방지하려면, 영구 옵티마이저 통계를 계산할 때 삭제 표시가 있는 레코드가 포함되도록 innodb_stats_include_delete_marked를 활성화할 수 있습니다.

innodb_stats_include_delete_marked를 활성화하면 테이블 분석은 통계를 다시 계산할 때 삭제 표시가 있는 레코드를 고려합니다.

innodb_stats_include_delete_marked는 모든 InnoDB 테이블에 영향을 미치는 전역 설정이며, 영구 옵티마이저 통계에만 적용됩니다.

InnoDB 영구 통계 테이블

영구 통계 기능은 mysql 데이터베이스의 내부적으로 관리되는 테이블인 innodb_table_stats 및 innodb_index_stats에 의존합니다. 이 테이블은 모든 설치, 업그레이드 및 소스에서 빌드 절차에서 자동으로 설정됩니다.

표 15.6 innodb_table_stats의 열

열 이름	설명
데이터베이스_이름	데이터베이스 이름
테이블_이름	테이블 이름, 파티션 이름 또는 하위 파티션 이름
last_update	마지막으로 InnoDB 이 행 업데이트
n_rows	테이블의 행 수
클러스터된_인덱스_크기	기본 색인의 크기(페이지 단위)
합계_오타_인덱스_크기	기타(주 인덱스가 아닌) 인덱스의 총 크기(페이지 단위)

표 15.7 innodb_index_stats의 열

열 이름	설명
데이터베이스_이름	데이터베이스 이름
테이블_이름	테이블 이름, 파티션 이름 또는 하위 파티션 이름
index_name	색인 이름
last_update	행이 마지막으로 업데이트된 시간을 나타내는 타임스탬프입니다.

열 이름	설명
stat_name	통계의 이름이며, 통계값 열에 값이 보고됩니다.
stat_value	에 명명된 통계의 값입니다. stat_name 열
sample_size	통계_값 열에 제공된 예상치에 대해 샘플링된 페이지 수입니다.
통계_설명	통계에 이름이 지정된 통계에 대한 설명 stat_name 열

innodb_table_stats 및 innodb_index_stats 테이블에는 인덱스 통계가 마지막으로 업데이트된 시점을 보여주는 last_update 열이 포함되어 있습니다:

```
mysql> SELECT * FROM innodb_table_stats \G
*************************** 1. 행 *************************** 데
           이터베이스_이름: 사킬라
              table_name: 배우
            마지막 업데이트: 2014-05-28 16:16:44
                  n_rows: 200
       클러스터된_인덱스_크기: 1
mysql> SELECT * FROM innodb_index_stats \G
*************************** 1. 행 *************************** 데
   이터베이스_이름: 사킬라
      테이블_이름: 액터 인덱스
       _이름: 기본
     last_update: 2014-05-28 16:16:44
       stat_name: n_diff_pfx01
      stat_value: 200
     sample_size: 1
     ...
```

innodb_table_stats 및 innodb_index_stats 테이블은 수동으로 업데이트할 수 있으므로 데이터베이스를 수정하지 않고도 특정 쿼리 최적화 계획을 강제로 적용하거나 대체 계획을 테스트할 수 있습니다. 통계를 수동으로 업데이트하는 경우 FLUSH TABLE *tbl_name* 문을 사용하여 업데이트된 통계를 로드합니다.

영구 통계는 서버 인스턴스와 관련이 있으므로 로컬 정보로 간주됩니다. 따라서 자동 통계 재계산이 수행될 때 innodb_table_stats 및 innodb_index_stats 테이블은 복제되지 않습니다. 통계의 동기식 재계산을 시작하기 위해 테이블 분석을 실행하면 해당 문이 복제되고(로깅을 억제하지 않은 경우) 복제본에서 재계산이 수행됩니다.

InnoDB 영구 통계 테이블 예제

innodb_table_stats 테이블에는 각 테이블에 대해 하나의 행이 포함됩니다. 다음 예는 수집되는 데이터 유형을 보여줍니다.

테이블 t1에는 기본 인덱스(열 a, b), 보조 인덱스(열 c, d) 및 고유 인덱스(열 e, f)가 포함되어 있습니다:

```
CREATE TABLE t1 (
a INT, b INT, c INT, d INT, e INT, f INT,
기본 키(a, b), 키 i1(c, d), 고유 키 i2uniq(e, f)
) 엔진=innodb;
```

샘플 데이터 5행이 삽입되면 테이블 `t1`이 다음과 같이 나타납니다:

```
mysql> SELECT * FROM t1;
+---+---+------+------+------+---------+
| A | B | C | D | E | F|
```

```
+---+---+------+------+------+---------+
| 1 | 1 |   10 |11 | 100 | 101 |
| 1 | 2 |   10 |   11 |  200 |  102 |
| 1 | 3 |   10 |   11 |  100 |  103 |
| 1 | 4 |   10 |   12 |  200 |  104 |
| 1 | 5 |   10 |12 | 100 | 105 |
+---+---+------+------+------+---------+
```

통계를 즉시 업데이트하려면 테이블 분석을 실행합니다(innodb_stats_auto_recalc가 활성화된 경우, 변경된 테이블 행의 10% 임계값에 도달했다고 가정하면 몇 초 내에 통계가 자동으로 업데이트됩니다):

```
mysql> ANALYZE TABLE t1;
+---------+---------+----------+-------------+
| 표 | 연산           | 메시지 유형 | 메시지 텍스트 |
+---------+---------+----------+-------------+
| test.t1 | 분석 | 상태| 확인      |
+---------+---------+----------+-------------+
```

테이블 t1에 대한 테이블 통계는 InnoDB가 테이블 통계를 마지막으로 업데이트한 시간(2014-03-14 14:36:34), 테이블의 행 수(5), 클러스터된 인덱스 크기(1페이지) 및 다른 인덱스의 합산 크기(2페이지)를 보여줍니다.

```
mysql> SELECT * FROM mysql.innodb_table_stats WHERE table_name like 't1'\G
*************************** 1. 행 *************************** 데
           이터베이스_이름: 테스트
              테이블_이름: T1
             마지막 업데이트: 2014-03-14 14:36:34
                  n_rows: 5
       클러스터된_인덱스_크기: 1
        합계_오타_인덱스_크기: 2
```

innodb_index_stats 테이블은 각 인덱스에 대해 여러 행을 포함합니다. innodb_index_stats 테이블의 각 행은 stat_name 열에 이름이 지정되고 stat_description 열에 설명되어 있는 특정 인덱스 통계와 관련된 데이터를 제공합니다. 예를 들어

```
mysql> SELECT index_name, stat_name, stat_value, stat_description
       FROM mysql.innodb_index_stats WHERE table_name like 't1';
+------------+-------------+------------+--------------------------------------+
| index_name | stat_name |   stat_value | stat_description                     |
+------------+-------------+------------+--------------------------------------+
| 기본         | N_DIFF_PFX01 |          1 | a                                    |
| 기본         | N_DIFF_PFX02 |          5 | a,b                                  |
| 기본         | N_LEAF_PAGES |          1 | 색인 내 리프 페이지 수                       |
| 기본         | 크기           |          1 | 색인 내 페이지 수                           |
| i1         | N_DIFF_PFX01 |          1 | c                                    |
| i1         | N_DIFF_PFX02 |          2 | c,d                                  |
| i1         | N_DIFF_PFX03 |          2 | c,d,a                                |
| i1         | N_DIFF_PFX04 |          5 | C,D,A,B                              |
| i1         | N_LEAF_PAGES |          1 | 색인 내 리프 페이지 수                       |
| i1         | 크기           |          1 | 색인 내 페이지 수                           |
| i2uniq     | N_DIFF_PFX01 |          2 | e                                    |
| i2uniq     | N_DIFF_PFX02 |          5 | e,f                                  |
| i2uniq     | N_LEAF_PAGES |          1 | 색인 내 리프 페이지 수                       |
| i2uniq| 크기|1             | 색인      내 페이지 수                    |
                                                                              -+
```

stat_name 열에는 다음과 같은 유형의 통계가 표시됩니다:

- 크기: 여기서 stat_name=크기인 경우, stat_value 열은 인덱스의 총 페이지 수를 표시합니다.

- `n_leaf_pages`: 여기서 `stat_name=n_leaf_페이지`, `stat_value` 열은 인덱스에 있는 리프 페이지의 수를 표시합니다.

- *`n_diff_pfxNN`*: `stat_name=n_diff_pfx01`인 경우, `stat_value` 열은 인덱스의 첫 번째 열에 있는 고유 값의 수를 표시합니다. `stat_name=n_diff_pfx02`인 경우, `stat_value` 열은 인덱스의 처음 두 열에 있는 고유 값의 수를 표시합니다,

등입니다. stat_name=n_diff_pfxNN인 경우 stat_description 열에는 쉼표로 구분된 인덱스 열 목록이 계산됩니다.

카디널리티 데이터를 제공하는 *n_diff_pfxNN* 통계를 더 자세히 설명하기 위해 이전에 소개한 t1 테이블의 예를 다시 한 번 살펴보겠습니다. 아래 그림과 같이, t1 테이블은 기본 인덱스(열 a, b), 보조 인덱스(열 c, d) 및 고유 인덱스(열 e, f)로 생성됩니다:

```
CREATE TABLE t1 (
  a INT, b INT, c INT, d INT, e INT, f INT,
  기본 키(a, b), 키 i1(c, d), 고유 키 i2uniq(e, f)
) 엔진=innodb;
```

샘플 데이터 5행이 삽입되면 테이블 t1이 다음과 같이 나타납니다:

```
mysql> SELECT * FROM t1;
+---+---+------+------+------+---------+
| A | B | C | D | E | F|
+---+---+------+------+------+---------+
| 1 | 1 |   10 |   11 |  100 |  101 |
| 1 | 2 |   10 |   11 |  200 |  102 |
| 1 | 3 |   10 |   11 |  100 |  103 |
| 1 | 4 |   10 |   12 |  200 |  104 |
| 1 | 5 |   10 |12  | 100 | 105 |
+---+---+------+------+------+---------+
```

인덱스 이름, 통계 이름, 통계 값 및 통계 설명을 쿼리할 때, 여기서 stat_name LIKE 'n_diff%'인 경우 다음과 같은 결과 집합이 반환됩니다:

```
mysql> SELECT index_name, stat_name, stat_value, stat_description
       FROM mysql.innodb_index_stats
       WHERE table_name 같은 't1' AND stat_name 같은 'n_diff%';
+------------+--------------+------------+----------------------+
| 인덱스 이름 | 통계           이름 | 통계 값 | 통계 설명 |
| 기본       | n_diff_pfx01 |          1 | a                    |
| 기본       | n_diff_pfx02 |          5 | a,b                  |
| i1         | n_diff_pfx01 |          1 | c                    |
| i1         | n_diff_pfx02 |          2 | c,d                  |
| i1         | n_diff_pfx03 |          2 | c,d,a                |
| i1         | n_diff_pfx04 |          5 | C,D,A,B              |
| i2uniq     | n_diff_pfx01 |          2 | e                    |
| i2uniq|    n_diff_pfx02 |5           | e,f                  |
+------------+--------------+------------+----------------------+
```

PRIMARY 인덱스의 경우 두 개의 n_diff% 행이 있습니다. 행 수는 인덱스의 열 수와 같습니다.

참고

고유하지 않은 인덱스의 경우 InnoDB는 기본 키의 열을 추가합니다.

- 여기서 index_name=PRIMARY 및 stat_name=n_diff_pfx01에서 stat_value는 1이며, 이는 인덱스의 첫 번째 열(열 a)에 단일 고유 값이 있음을 나타냅니다. 열 a의 고유 값 수는 단일 고유 값(1)이 있는 테이블 t1에서 열 a의 데이터를 보면 확인할 수 있습니다. 카운트된 열(a)은 결과 집합의 stat_description 열에 표시됩니다.

- 여기서 index_name=PRIMARY 및 stat_name=n_diff_pfx02에서 stat_value는 5이며, 이는 인덱스의 두 열(a,b)에 고유값이 5개 있음을 나타냅니다. 열의 고유 값 수는 (1,1), (1,2), (1,3), (1,4),

(1,5) 등 다섯 개의 고유 값이 있는 테이블 t1의 열 a와 b의 데이터를 보면 확인할 수 있습니다. 카운트된 열(a,b)은 결과 집합의 stat_description 열에 표시됩니다.

보조 인덱스(i1)의 경우 4개의 n_diff% 행이 있습니다. 보조 인덱스(c,d)에는 두 개의 열만 정의되어 있지만 보조 인덱스에는 4개의 n_diff% 행이 있는 이유는 InnoDB의

는 기본 키가 있는 모든 고유하지 않은 인덱스에 접미사를 붙입니다. 결과적으로 보조 인덱스 열(c,d)과 기본 키 열(a,b)을 모두 설명하기 위해 두 개가 아닌 네 개의 n_diff% 행이 있습니다.

- 여기서 index_name=i1 및 stat_name=n_diff_pfx01에서 stat_value는 1이며, 이는 인덱스의 첫 번째 열(열 c)에 단일 고유값이 있음을 나타냅니다. 열(c)의 고유 값 수는 테이블 t1에서 열(c)의 데이터를 보면 확인할 수 있으며, 여기에는 단일 고유 값(10)이 있습니다. 카운트된 열(c)은 결과 집합의 stat_description 열에 표시됩니다.

- 인덱스_이름이 i1이고 stat_이름이 n_diff_pfx02인 경우, stat_값은 2이며 인덱스의 처음 두 열(c,d)에 고유값이 두 개 있음을 나타냅니다. 열 c와 d의 고유 값 수는 (10,11) 및 (10,12)라는 두 개의 고유 값이 있는 테이블 t1의 열 c와 d의 데이터를 보면 확인할 수 있습니다 카운트된 열(c,d)은 결과 집합의 stat_description 열에 표시됩니다.

- 여기서 index_name=i1 및 stat_name=n_diff_pfx03에서 stat_value는 2이며, 이는 인덱스의 처음 세 열(c,d,a)에 고유값이 두 개 있음을 나타냅니다. 열 c, d, a의 고유 값 수는 (10,11,1) 및 (10,12,1)이라는 두 개의 고유 값이 있는 테이블 t1의 열 c, d, a의 데이터를 보면 확인할 수 있습니다 카운트된 열(c,d,a)은 결과 집합의 stat_description 열에 표시됩니다.

- 인덱스_이름=i1 및 stat_이름=n_diff_pfx04에서 stat_값은 5이며, 이는 인덱스의 네 열(c,d,a,b)에 고유값이 5개 있음을 나타냅니다. 열의 고유 값 수는 (10,11,1,1), (10,11,1,2) 등 5개의 고유 값이 있는 테이블 t1의 열 c, d, a, b의 데이터를 보면 확인할 수 있습니다, (10,11,1,3), (10,12,1,4) 및 (10,12,1,5). 카운트된 열(c,d,a,b)은 결과 집합의 통계_설명 열입니다.

고유 인덱스(i2uniq)의 경우 두 개의 n_diff% 행이 있습니다.

- 인덱스_이름=i2uniq 및 stat_이름=n_diff_pfx01에서 stat_값은 2이며, 이는 인덱스의 첫 번째 열(열 e)에 고유값이 두 개 있음을 나타냅니다. 열(e)의 고유 값 수는 테이블 t1에서 열(e)의 데이터를 보면 (100) 및 (200)이라는 두 개의 고유 값이 있는 것을 확인할 수 있습니다 카운트된 열(e)은 결과 집합의 stat_description 열에 표시됩니다.

- 여기서 index_name=i2uniq 및 stat_name=n_diff_pfx02에서 stat_value는 5이며, 이는 인덱스의 두 열(e,f)에 고유값이 5개 있음을 나타냅니다. 열의 고유 값 수는 (100,101), (200,102), (100,103), (200,104), (100,105)의 다섯 가지 고유 값이 있는 테이블 t1의 열 e 및 f의 데이터를 보면 확인할 수 있습니다. 카운트된 열(e,f)은 결과 집합의 stat_description 열에 표시됩니다.

innodb_index_stats 테이블을 사용하여 인덱스 크기 검색하기

테이블, 파티션 또는 하위 파티션의 인덱스 크기는 innodb_index_stats 테이블을 사용하여 검색할 수 있습니다. 다음 예에서는 테이블 t1에 대한 인덱스 크기를 검색합니다. 테이블 t1 및 해당 인덱스 통계에 대한 정의는 InnoDB 영구 통계 테이블 예제를 참조하세요.

```
mysql> SELECT SUM(stat_value) pages, index_name,
       SUM(stat_value)*@@innodb_page_size size
       FROM mysql.innodb_index_stats WHERE table_name='t1'
       AND stat_name = 'size' GROUP BY index_name;
+-------+------------+---------+
| 페이지 | 인덱스_이름 | 크기 |
+-------+------------+---------+
|1     | 기본|          16384 |
      |1 | i1|         16384 |
      |1 | i2uniq|     16384 |
+-------+------------+---------+
```

파티션 또는 하위 파티션의 경우, 동일한 쿼리에 수정된 `WHERE` 절을 사용하여 인덱스 크기를 검색할 수 있습니다. 예를 들어, 다음 쿼리는 테이블 `t1`의 파티션에 대한 인덱스 크기를 검색합니다:

```
mysql> SELECT SUM(stat_value) pages, index_name,
       SUM(stat_value)*@@innodb_page_size size
       FROM mysql.innodb_index_stats WHERE table_name = 't1#P%' AND
       stat_name = 'size' GROUP BY index_name;
```

15.8.10.2 비영구 옵티마이저 통계 매개변수 구성하기

이 섹션에서는 비영구 옵티마이저 통계를 구성하는 방법을 설명합니다. 최적화 프로그램 통계는 innodb_stats_persistent=OFF이거나 `STATS_PERSISTENT=0`으로 개별 테이블을 생성하거나 변경할 때 디스크에 지속되지 않습니다. 대신 통계는 메모리에 저장되며 서버가 종료되면 손실됩니다. 또한 통계는 특정 작업 및 특정 조건에 따라 주기적으로 업데이트됩니다.

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.

옵티마이저 통계 업데이트

비영구 옵티마이저 통계는 언제 업데이트됩니다:

- 분석 테이블을 실행합니다.
- 테이블 상태 표시, 인덱스 표시를 실행하거나 정보 스키마 테이블 또는 통계 테이블에 `innodb_stats_on_metadata` 옵션이 활성화되어 있습니다.

  `innodb_stats_on_metadata`의 기본 설정은 `OFF`입니다. `innodb_stats_on_met`데이터를 활성화하면 테이블 또는 인덱스 수가 많은 스키마의 액세스 속도가 저하되고 `InnoDB` 테이블이 포함된 쿼리의 실행 계획 안정성이 저하될 수 있습니다. `innodb_stats_on_met`데이터는 `SET` 문을 사용하여 전역적으로 구성됩니다.

  ```
  설정 글로벌 innodb_stats_on_metadata=ON
  ```

  참고

  `innodb_stats_on_metadata`는 옵티마이저 통계가 비영구적으로 구성된 경우에만 적용됩니다(innodb_stats_persistent가 비활성화되어 있는 경우).

- 기본값인 `--auto-rehash` 옵션을 활성화한 상태에서 `mysql` 클라이언트를 시작합니다. 자동 재해시 옵션을 사용하면 모든 `InnoDB` 테이블이 열리고 테이블을 열면 통계가 다시 계산됩니다.

  `mysql` 클라이언트의 시작 시간을 개선하고 통계를 업데이트하려면 `--disable-auto-rehash` 옵션을 사용하여 자동 재해시를 해제할 수 있습니다. 자동 리해시 기능을 사용하면 대화형 사용자를 위해 데이터베이스, 테이블 및 열 이름을 자동으로 완성할 수 있습니다.

- 먼저 테이블이 열립니다.
- InnoDB는 마지막으로 통계가 업데이트된 이후 테이블의 1/16이 수정되었음을 감지합니다.

샘플링된 페이지 수 구성하기

MySQL 쿼리 옵티마이저는 키 분포에 대한 추정 통계를 사용하여 인덱스의 상대적 선택성에 따라 실행 계획에 사용할 인덱스를 선택합니다. InnoDB는 옵티마이저 통계를 업데이트할 때 테이블의 각 인덱스에서 무작위로 페이지를 샘플링하여 인덱스의 카디널리티를 추정합니다. (이 기법을 랜덤 다이브라고 합니다.)

통계 추정치의 품질을 제어하여 쿼리 최적화 도구에 더 나은 정보를 제공하려면 매개변수를 사용하여 샘플링된 페이지 수를 변경할 수 있습니다.

`innodb_stats_transient_sample_pages`. 샘플링된 페이지의 기본 개수는 8개로, 정확한 추정치를 생성하기에 부족하여 쿼리에서 인덱스 선택이 잘못될 수 있습니다.

옵티마이저를 사용합니다. 이 기술은 조인에 사용되는 큰 테이블과 테이블에 특히 중요합니다. 이러한 테이블에 대한 불필요한 전체 테이블 스캔은 상당한 성능 문제가 될 수 있습니다. 이러한 쿼리 조정에 대한 팁은 섹션 8.2.1.23, '전체 테이블 스캔 피하기'를 참조하십시오. `innodb_stats_transient_sample_pages`는 런타임에 설정할 수 있는 전역 매개 변수입니다.

`innodb_stats_transient_sample_pages`의 값은 `innodb_stats_perpetistent=0`일 때 모든 `InnoDB` 테이블 및 인덱스의 인덱스 샘플링에 영향을 줍니다. 인덱스 샘플 크기를 변경할 때 다음과 같은 잠재적으로 중요한 영향에 유의하세요:

- 1 또는 2와 같은 작은 값은 카디널리티의 부정확한 추정을 초래할 수 있습니다.
- `innodb_stats_transient_sample_pages` 값을 늘리면 더 많은 디스크 읽기가 필요할 수 있습니다. 8보다 훨씬 큰 값(예: 100)을 사용하면 테이블을 열거나 테이블 상태 표시를 실행하는 데 걸리는 시간이 크게 느려질 수 있습니다.
- 옵티마이저는 인덱스 선택도에 대한 다양한 추정치를 기반으로 매우 다른 쿼리 계획을 선택할 수 있습니다.

시스템에 가장 적합한 `innodb_stats_transient_sample_pages` 값이 무엇이든, 옵션을 설정하고 해당 값으로 둡니다. 과도한 I/O를 요구하지 않으면서 데이터베이스의 모든 테이블에 대해 합리적으로 정확한 추정치를 제공하는 값을 선택합니다. 통계는 자동으로
를 분석 테이블 실행 시 이외의 다양한 시점에 재계산하는 경우, 인덱스 샘플 크기를 늘리고 분석 테이블을 실행한 다음 다시 샘플 크기를 줄이는 것은 의미가 없습니다.

일반적으로 작은 테이블은 큰 테이블보다 더 적은 수의 인덱스 샘플이 필요합니다. 데이터베이스에 큰 테이블이 많은 경우, 작은 테이블이 대부분인 경우보다 `innodb_stats_transient_sample_pages`의 값을 더 높게 사용하는 것이 좋습니다.

15.8.10.3 InnoDB 테이블에 대한 분석 테이블 복잡성 추정

`InnoDB` 테이블의 분석 테이블 복잡도는 다음에 따라 달라집니다:

- `innodb_stats_persistent_sample_pages`에 정의된 대로 샘플링된 페이지 수입니다.
- 테이블에서 인덱싱된 열의 수입니다.
- 파티션의 수입니다. 테이블에 파티션이 없는 경우 파티션 수는 1로 간주됩니다. 이러한 매개 변수를 사용하여

테이블 복잡성을 추정하는 대략적인 공식은 다음과 같습니다:

`innodb_stats_persistent_sample_pages` 값 * 테이블의 인덱싱된 열 수 *
파티션 수

일반적으로 결과 값이 클수록 테이블 분석의 실행 시간이 길어집니다.

참고

`innodb_stats_persistent_sample_pages`는 글로벌 수준에서 샘플링되는 페이지 수를 정의합니다. 개별 테이블에 대해 샘플링되는 페이지 수를 설정하려면 `CREATE TABLE` 또는 `ALTER TABLE`과 함께 `STATS_SAMPLE_PAGES` 옵션을 사용합니다. 자세한 내용은 섹션 15.8.10.1, "영구 옵티마이저 통계 매개변수 구성"을 참조하십시오.

`innodb_stats_persistent=OFF`인 경우, 샘플링되는 페이지 수는 `innodb_stats_transient_sample_pages`에 의해 정의됩니다. 자세한 내용은 15.8.10.2절. "비영구 최적화 프로그램 통계 매개변수 구성"을 참조하십시오.

분석 테이블 복잡성을 추정하는 보다 심층적인 접근 방식을 보려면 다음 예를 고려하십시오.

빅 O 표기법에서 분석 테이블 복잡도는 다음과 같이 설명됩니다:

```
O(n_sample
  * (N_COLS_IN_UNIQ_I
     + N_COLS_IN_NON_UNIQ_I
     + n_cols_in_pk * (1 + n_non_uniq_i))
  * n_part)
```

어디에:

- n_sample은 샘플링된 페이지 수입니다(INNODB_STATS_PERSISTENT_SAMPLE_PAGES)
- n_cols_in_uniq_i는 모든 고유 인덱스에 있는 모든 열의 총 개수입니다(기본 키 열은 포함되지 않음).
- N_COLS_IN_NON_UNIQ_I는 모든 고유하지 않은 인덱스에 있는 모든 열의 총 개수입니다.
- n_cols_in_pk는 기본 키의 열 수입니다(기본 키가 정의되지 않은 경우 InnoDB 내부적으로 단일 열 기본 키를 생성합니다)
- N_NON_UNIQ_I는 테이블에 있는 고유하지 않은 인덱스의 수입니다.
- n_part는 파티션 수입니다. 파티션이 정의되지 않은 경우 테이블은 단일 파티션으로 간주됩니다.

이제 기본 키(열 2개), 고유 인덱스(열 2개) 및 두 개의 비고유 인덱스(각각 열 2개)가 있는 다음 테이블(테이블 t)을 고려해 보겠습니다:

```
CREATE TABLE t (
  a INT,
  b INT,
  c INT,
  d INT,
  e INT,
  f INT,
  g INT,
  h INT,
  기본 키(a, b),
  고유 키 i1uniq(c, d), 키
  i2nonuniq(e, f),
  키 i3nonuniq(g, h)
);
```

위에서 설명한 알고리즘에 필요한 열 및 인덱스 데이터를 보려면 테이블 t에 대해 mysql.innodb_index_stats 영구 인덱스 통계 테이블을 쿼리합니다. n_diff_pfx% 통계는 각 인덱스에 대해 카운트되는 열을 표시합니다. 예를 들어, 기본 키 인덱스의 경우 열 a와 b가 계산됩니다. 고유하지 않은 인덱스의 경우 사용자 정의 열과 함께 기본 키 열(a,b)이 카운트됩니다.

참고

InnoDB 영구 통계 테이블에 대한 자세한 내용은 섹션 15.8.10.1, "영구 옵티마이저 통계 매개변수 구성"을 참조하십시오.

```
mysql> SELECT index_name, stat_name, stat_description
       FROM mysql.innodb_index_stats WHERE
       database_name='test' AND
       table_name='t' AND
       stat_name을 'n_diff_pfx%'와 같이 설정합니다;
  +------------+-------------+----------------------+
  | 인덱스 이름 | 통계 이름 |      통계 설명 |
```

```
+------------+-------------+--------------------+
| 기본         | n_diff_pfx01 | a                  |
| 기본         | n_diff_pfx02 | a,b                |
| i1uniq     | n_diff_pfx01 | c                  |
| i1uniq     | n_diff_pfx02 | c,d                |
| i2nonuniq  | n_diff_pfx01 | e                  |
| i2nonuniq  | n_diff_pfx02 | e,f                |
```

```
| i2nonuniq   | n_diff_pfx03 | e,f,a           |
| i2nonuniq   | n_diff_pfx04 | e,f,a,b         |
| i3nonuniq   | n_diff_pfx01 | g               |
| i3nonuniq   | n_diff_pfx02 | g,h             |
| i3nonuniq   | n_diff_pfx03 | g,h,a           |
| i3nonuniq | n_diff_pfx04 | g,h,a,b         |
+------------+--------------+-----------------+
```

위에 표시된 인덱스 통계 데이터와 테이블 정의를 기반으로 다음 값을 확인할 수 있습니다:

- 기본 키 열을 포함하지 않는 모든 고유 인덱스의 모든 열의 총 개수인 `n_cols_in_uniq_i`는 2(`c` 및 `d`) 입니다.
- 모든 고유하지 않은 인덱스에 있는 모든 열의 총 개수인 `N_COLS_IN_NON_UNIQ_I`는 4(`E`, `F`, `G`) 입니다. 및 `h`)
- 기본 키의 열 수인 n_cols_in_pk는 2(`a` 및 `b`) 입니다.
- 테이블에 있는 고유하지 않은 인덱스의 개수인 `N_NON_UNIQ_I`는 2(`I2NONUNIQ` 및 `I3NONUNIQ`)입니다.)
- 파티션 수인 n_part는 1입니다.

이제 `innodb_stats_persistent_sample_pages` * (2 + 4 + 2 * (1 + 2)) * 1
을 계산하여 스캔되는 리프 페이지 수를 결정할 수 있습니다. 와
기본값이 `20`으로 설정되어 있고 기본 페이지 크기가 `16KiB`(`innodb_page_size=16384`)인 경우, 테이블 `t`에 대해 20 * 12 * `16384`바이트, 즉 약 `4MiB`가 읽혀진다고 추정할 수 있습니다.

참고

일부 리프 페이지가 이미 버퍼 풀에 캐시되어 있을 수 있으므로 디스크에서 4MB를 모두 읽지 못할 수도 있습니다.

15.8.11 인덱스 페이지에 대한 병합 임계값 구성하기

인덱스 페이지에 대한 `MERGE_THRESHOLD` 값을 구성할 수 있습니다. 행이 삭제되거나 `UPDATE` 작업으로 행이 단축될 때 인덱스 페이지의 "페이지 가득 차 있음" 비율이 `MERGE_THRESHOLD` 값 아래로 떨어지면 `InnoDB`는 인덱스 페이지를 인접한 인덱스 페이지와 병합하려고 시도합니다.
기본 `MERGE_THRESHOLD` 값은 이전에 하드코딩된 값인 50입니다. 최소 `MERGE_THRESHOLD` 값은 1이고 최대 값은 50입니다.

인덱스 페이지의 "페이지 가득 찼음" 비율이 기본 `MERGE_THRESHOLD` 설정인 50% 미만으로 떨어지면 `InnoDB`는 인덱스 페이지를 인접한 페이지와 병합하려고 시도합니다. 두 페이지가 모두 50%에 가까워지면 페이지가 병합된 직후 페이지 분할이 발생할 수 있습니다. 이러한 병합-분할 동작이 자주 발생하면 성능에 부정적인 영향을 미칠 수 있습니다. 빈번한 병합 분할을 방지하려면 `MERGE_THRESHOLD` 값을 낮추어 `InnoDB`가 더 낮은 "페이지 가득 찼음" 비율에서 페이지 병합을 시도하도록 할 수 있습니다. 페이지가 가득 찬 비율을 낮춰 페이지를 병합하면 인덱스 페이지에 더 많은 공간을 확보할 수 있고 병합 분할 동작을 줄이는 데 도움이 됩니다.

인덱스 페이지에 대한 `MERGE_THRESHOLD`는 테이블 또는 개별 인덱스에 대해 정의할 수 있습니다. 개별 인

덱스에 대해 정의된 `MERGE_THRESHOLD` 값은 테이블에 대해 정의된 `MERGE_THRESHOLD` 값보다 우선합니다. 정의되지 않은 경우 기본값은 50입니다.

테이블에 대한 MERGE_THRESHOLD 설정

`CREATE TABLE` 문의 *`table_option`* `COMMENT` 절을 사용하여 테이블의 `MERGE_THRESHOLD` 값을 설정할 수 있습니다. 예를 들면 다음과 같습니다:

```
CREATE TABLE t1 (
   id INT,
  KEY id_index (id)
```

```
) comment='merge_threshold=45';
```

table_option을 사용하여 기존 테이블에 대한 MERGE_THRESHOLD 값을 설정할 수도 있습니다. COMMENT 절을 ALTER TABLE로 변경합니다:

```
CREATE TABLE t1 (
   id INT,
  KEY id_index (id)
);

ALTER TABLE t1 COMMENT='MERGE_THRESHOLD=40';
```

개별 인덱스에 대한 MERGE_THRESHOLD 설정하기

개별 인덱스에 대한 MERGE_THRESHOLD 값을 설정하려면 다음 예제와 같이 CREATE TABLE, ALTER TABLE 또는 CREATE INDEX와 함께 *index_option* COMMENT 절을 사용할 수 있습니다:

- CREATE TABLE을 사용하여 개별 인덱스에 대해 MERGE_THRESHOLD를 설정합니다:

```
CREATE TABLE t1 (
   id INT,
  KEY id_index (id) COMMENT 'MERGE_THRESHOLD=40'
);
```

- ALTER TABLE을 사용하여 개별 인덱스에 대해 MERGE_THRESHOLD를 설정합니다:

```
CREATE TABLE t1 (
   id INT,
  KEY id_index (id)
);

ALTER TABLE t1 DROP KEY id_index;
ALTER TABLE t1 ADD KEY id_index (id) COMMENT 'MERGE_THRESHOLD=40';
```

- CREATE INDEX를 사용하여 개별 인덱스에 대해 MERGE_THRESHOLD를 설정합니다:

```
CREATE TABLE t1 (id INT);
CREATE INDEX id_index ON t1 (id) COMMENT 'MERGE_THRESHOLD=40';
```

참고

기본 키 또는 고유 키 인덱스 없이 InnoDB 테이블을 생성할 때 InnoDB가 생성하는 클러스터 인덱스인 GEN_CLUST_INDEX의 인덱스 수준에서는 MERGE_THRESHOLD 값을 수정할 수 없습니다. 테이블에 대한 MERGE_THRESHOLD를 설정해야만 GEN_CLUST_INDEX에 대한 MERGE_THRESHOLD 값을 수정할 수 있습니다.

인덱스에 대한 MERGE_THRESHOLD 값 쿼리하기

인덱스의 현재 MERGE_THRESHOLD 값은 인덱스의 INNODB_INDEXES 테이블. 예를 들어

```
mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_INDEXES WHERE NAME='id_index' \G
*************************** 1. 행 ***************************
       INDEX_ID: 91
           NAME: id_index
       TABLE_ID: 68
           TYPE: 0
       N_FIELDS: 1
        PAGE_NO: 4
            공간: 57
merge_threshold: 40
```

table_option COMMENT 절을 사용하여 명시적으로 정의한 경우 SHOW CREATE TABLE을 사용하여 테이블의 MERGE_THRESHOLD 값을 볼 수 있습니다:

```
mysql> SHOW CREATE TABLE t2 \G
*************************** 1. 행 ***************************
       이블: t2
테이블을 만듭니다: CREATE TABLE `t2` (
  `id` int(11) 기본값 NULL,
  KEY `id_index` (`id`) COMMENT 'MERGE_THRESHOLD=40'
) 엔진=InnoDB 기본 문자 집합=utf8mb4
```

참고

인덱스 수준에서 정의된 MERGE_THRESHOLD 값은 테이블에 대해 정의된 MERGE_THRESHOLD 값보다우선합니다. 정의되지 않은 경우, 기본값은 50%(이전에 하드코딩된 값인 MERGE_THRESHOLD=50)입니다.

마찬가지로, *index_option* COMMENT 절을 사용하여 명시적으로 정의한 경우 SHOW INDEX를 사용하여 인덱스의 MERGE_THRESHOLD 값을 볼 수 있습니다:

```
mysql> SHOW INDEX FROM t2 \G
*************************** 1. 행 ***************************
        이블: t2
   비고유: 1 키_이름:
     id_index
 Seq_in_index: 1
  열_이름: id
    콜레이션: 카디
  널리티: 0
     Sub_part: NULL 포장됨
       : NULL
         Null: YES 인덱스_
   유형: BTREE
      댓글:
인덱스_댓글: merge_threshold=40
```

MERGE_THRESHOLD 설정의 효과 측정하기

INNODB_METRICS 테이블은 두 개의 카운터를 제공합니다.
인덱스 페이지 병합의 MERGE_THRESHOLD 설정.

```
mysql> SELECT NAME, COMMENT FROM INFORMATION_SCHEMA.INNODB_METRICS
       WHERE NAME은 '%index_page_merge%'와 같습니다;
+----------------------------+-----------------------------------------+
| 이름| 댓글                      |
+----------------------------+-----------------------------------------+
| index_page_merge_attempts|   인덱스 페이지 병합 시도                          횟수 |
| 인덱스 페이지 병합 성공 횟수 | 인덱스 페이지 병합 성공 횟수 | 인덱스 페이지 병합 성공 횟수
+----------------------------+-----------------------------------------+
```

MERGE_THRESHOLD 값을 낮출 때의 목표는 다음과 같습니다:

- 페이지 병합 시도 횟수 및 페이지 병합 성공 횟수 감소
- 비슷한 수의 페이지 병합 시도 및 페이지 병합 성공 횟수

`MERGE_THRESHOLD` 설정이 너무 작으면 빈 페이지 공간이 과도하게 많아 데이터 파일이 커질 수 있습니다.

`INNODB_METRICS` 카운터 사용에 대한 자세한 내용은 섹션 15.15.6, 'InnoDB 정보_SCHEMA 메트릭 테이블'을 참조하십시오.

15.8.12 전용 MySQL 서버에 자동 구성 활성화하기

innodb_dedicated_server를 활성화하면 `InnoDB는` 다음 변수를 자동으로 구성합니다:

- `innodb_buffer_pool_size`

- innodb_redo_log_capacity

참고

innodb_log_file_size 및 innodb_log_files_in_group은 더 이상 사용되지 않으며 innodb_redo_log_capacity로 대체됩니다.

- innodb_flush_method

MySQL 인스턴스가 사용 가능한 모든 시스템 리소스를 사용할 수 있는 전용 서버에 상주하는 경우에만 innodb_dedicated_server를 활성화하는 것을 고려하세요. 예를 들어, Docker 컨테이너 또는 전용 VM에서 MySQL Server를 실행하는 경우 innodb_dedicated_server를 활성화하는 것을 고려하십시오.
는 MySQL만 실행합니다. MySQL 인스턴스가 다른 애플리케이션과 시스템 리소스를 공유하는 경우 innodb_dedicated_server를 활성화하는 것은 권장되지 않습니다.

다음 정보에서는 각 변수가 자동으로 구성되는 방법을 설명합니다.

- innodb_buffer_pool_size

  버퍼 풀 크기는 서버에서 감지된 메모리 양에 따라 구성됩니다.

표 15.8 자동으로 구성된 버퍼 풀 크기

감지된 서버 메모리	버퍼 풀 크기
1GB 미만	128MB(기본값)
1GB ~ 4GB	*감지된 서버 메모리* * 0.5
4GB 이상	*감지된 서버 메모리* * 0.75

- innodb_redo_log_capacity

  재실행 로그 용량은 서버에서 감지된 메모리 양에 따라 구성되며, 경우에 따라 innodb_buffer_pool_size가 명시적으로 구성되었는지 여부에 따라 구성됩니다. innodb_buffer_pool_size가 명시적으로 구성되지 않은 경우 기본값이 사용됩니다.

경고

innodb_buffer_pool_size가 감지된 서버 메모리 양보다 큰 값으로 설정된 경우 자동 재실행 로그 용량 구성 동작이 정의되지 않습니다.

표 15.9 자동으로 구성된 로그 파일 크기

감지된 서버 메모리	버퍼 풀 크기	로그 용량 재실행
1GB 미만	구성되지 않음	100MB
1GB 미만	1GB 미만	100MB
1GB ~ 2GB	해당 없음	100MB

2GB ~ 4GB	구성되지 않음	1GB
2GB ~ 4GB	구성된 모든 값	라운드(0.5 * *감지된 서버 메모리*(GB)) * 0.5 GB
4GB ~ 10.66GB	해당 없음	라운드(0.75 * *감지된 서버 메모리*(GB)) * 0.5GB
10.66GB ~ 170.66GB	해당 없음	라운드(0.5625 * *감지된 서버 메모리*(GB)) * 1GB
170.66GB 이상	해당 없음	128GB

- `innodb_log_file_size`(더 이상 사용되지 않음)

로그 파일 크기는 자동으로 구성된 버퍼 풀 크기에 따라 구성됩니다.

표 15.10 자동으로 구성된 로그 파일 크기

버퍼 풀 크기	로그 파일 크기
8GB 미만	512MB
8GB ~ 128GB	1024MB
128GB 이상	2048MB

- `innodb_log_files_in_group`(사용 중단됨)

  로그 파일 수는 자동으로 구성된 버퍼 풀 크기에 따라 구성됩니다.

  표 15.11 자동으로 구성된 로그 파일 수

버퍼 풀 크기	로그 파일 수
8GB 미만	라운드(*버퍼 풀 크기*)
8GB ~ 128GB	라운드(*버퍼 풀 크기* * 0.75)
128GB 이상	64

  참고

  반올림된 버퍼 풀 크기 값이 2GB 미만인 경우 최소 `innodb_log_files_in_group` 값인 2가 적용됩니다.

- `innodb_flush_method`

  innodb_dedicated_server가 활성화된 경우 플러시 메서드는 `O_DIRECT_NO_FSYNC`로 설정됩니다. `O_DIRECT_NO_FSYNC` 설정을 사용할 수 없는 경우 기본 `innodb_flush_method` 설정이 사용됩니다.

  `InnoDB`는 I/O를 플러시하는 동안 `O_DIRECT`를 사용하지만 각 쓰기 작업 후에는 `fsync()` 시스템 호출을 건너뜁니다.

  경고

  새 파일을 만든 후, 파일 크기를 늘린 후, 파일을 닫은 후에 `fsync()`가 호출되어 파일 시스템 메타데이터 변경 사항이 동기화되도록 합니다. 각 쓰기 작업 후에는 여전히 `fsync()` 시스템 호출을 건너뜁니다.

  재실행 로그 파일과 데이터 파일이 서로 다른 저장 장치에 있고 배터리로 백업되지 않는 장치 캐시에서 데이터 파일 쓰기가 플러시되기 전에 예기치 않은 종료가 발생하는 경우 데이터 손실이 발생할 수 있습니다. 재실행 로그 파일과 데이터 파일에 서로 다른 저장 장치를 사용 중이거나 사용하려는 경우, 데이터 파일이 배터리로 백업

되지 않는 캐시가 있는 장치에 있는 경우, 대신 O_DIRECT를 사용하세요.

자동으로 구성된 옵션이 옵션 파일이나 다른 곳에서 명시적으로 구성된 경우 명시적으로 지정된 설정이 사용되며 이와 유사한 시작 경고가 stderr에 인쇄됩니다:

```
[경고] [000000] InnoDB: 옵션 innodb_dedicated_server가 무시됨
의 경우 innodb_버퍼_pool_size=134217728 이 명시적으로 지정되어 있기 때문입니다.
```

한 옵션을 명시적으로 구성해도 다른 옵션이 자동으로 구성되는 것을 막지는 못합니다.

innodb_dedicated_server가 활성화되고 innodb_buffer_pool_size가 명시적으로 구성된 경우, 버퍼 풀 크기를 기반으로 구성된 변수는 계산된 버퍼 풀 크기 값을 사용합니다.

를 명시적으로 정의된 버퍼 풀 크기 값이 아닌 서버에서 감지된 메모리 양에 따라 설정합니다.

자동으로 구성된 설정은 MySQL 서버가 시작될 때마다 평가되고 필요한 경우 재구성됩니다.

15.9 InnoDB 테이블 및 페이지 압축

이 섹션에서는 `InnoDB` 테이블 압축 및 `InnoDB` 페이지 압축 기능에 대한 정보를 제공합니다. 페이지 압축 기능은 투명 페이지 압축이라고도 합니다.

`InnoDB`의 압축 기능을 사용하면 데이터가 압축된 형태로 저장되는 테이블을 만들 수 있습니다. 압축은 원시 성능과 확장성을 모두 개선하는 데 도움이 될 수 있습니다. 압축은 디스크와 메모리 간에 전송되는 데이터의 양이 줄어들고 디스크와 메모리에서 차지하는 공간이 줄어든다는 것을 의미합니다. 인덱스 데이터도 압축되므로 보조 인덱스가 있는 테이블의 경우 이점이 더욱 증폭됩니다. 압축은 SSD 저장 장치에 특히 중요할 수 있는데, 그 이유는 SSD 저장 장치는 HDD 장치에 비해 용량이 낮은 경향이 있기 때문입니다.

15.9.1 InnoDB 테이블 압축

이 섹션에서는 파일별 테이블 테이블 스페이스 또는 일반 테이블 스페이스에 있는 `InnoDB` 테이블에서 지원되는 `InnoDB` 테이블 압축에 대해 설명합니다. 테이블 압축은 테이블 생성 또는 테이블 변경과 함께 `ROW_FORMAT=COMPRESSED` 속성을 사용하여 활성화합니다.

15.9.1.1 테이블 압축 개요

프로세서와 캐시 메모리의 속도가 디스크 저장 장치보다 빨라졌기 때문에 많은 워크로드가 디스크에 종속되어 있습니다. 데이터 압축을 사용하면 CPU 사용률을 높이는 대신 적은 비용으로 데이터베이스 크기를 줄이고, I/O를 줄이며, 처리량을 향상시킬 수 있습니다. 압축은 자주 사용하는 데이터를 메모리에 보관할 수 있는 충분한 RAM이 있는 시스템에서 읽기 집약적인 애플리케이션에 특히 유용합니다.

`ROW_FORMAT=COMPRESSED`로 생성된 `InnoDB` 테이블은 구성된 `innodb_page_size` 값보다 디스크에서 더 작은 페이지 크기를 사용할 수 있습니다. 페이지가 작을수록 디스크에서 읽고 쓰는 데 필요한 I/O가 줄어들어 SSD 장치에 특히 유용합니다.

압축된 페이지 크기는 `CREATE TABLE` 또는 `ALTER TABLE KEY_BLOCK_SIZE` 파라미터를 통해 지정합니다. 페이지 크기가 다르면 시스템 테이블 스페이스에는 압축된 테이블을 저장할 수 없으므로 테이블을 시스템 테이블 스페이스가 아닌 파일 단위 테이블 스페이스 또는 일반 테이블 스페이스에 배치해야 합니다. 자세한 내용은 15.6.3.2절. "파일 단위 테이블 테이블 스페이스" 및 15.6.3.3절. "일반 테이블 스페이스"를 참조하십시오.

압축 수준은 `KEY_BLOCK_SIZE` 값에 관계없이 동일합니다. `KEY_BLOCK_SIZE`에 더 작은 값을 지정하면 점점 더 작은 페이지의 I/O 이점을 얻을 수 있습니다. 그러나 너무 작은 값을 지정하면 데이터 값을 각 페이지에 여러 행에 맞도록 충분히 압축할 수 없을 때 페이지를 재구성하는 데 추가 오버헤드가 발생합니다. 각 인덱스의 키 열 길이에 따라 테이블에 대해 `KEY_BLOCK_SIZE`를 얼마나 작게 지정할 수 있는지에 대한 엄격한 제한이

있습니다. 너무 작은 값을 지정하면 `CREATE TABLE` 또는 `ALTER TABLE` 문이 실패합니다.

버퍼 풀에서 압축된 데이터는 작은 페이지에 보관되며, 페이지 크기는 `KEY_BLOCK_SIZE` 값에 따라 결정됩니다. 열 값을 추출하거나 업데이트하기 위해 MySQL은 압축되지 않은 데이터로 버퍼 풀에 비압축 페이지도 생성합니다. 버퍼 풀 내에서 비압축 페이지에 대한 모든 업데이트도 동등한 압축 페이지에 다시 기록됩니다. 압축 페이지와 비압축 페이지의 추가 데이터를 모두 수용할 수 있도록 버퍼 풀의 크기를 조정해야 할 수도 있지만, 공간이 필요할 때 비압축 페이지는 버퍼 풀에서 제거된 후 다음 액세스 시 다시 압축 해제됩니다.

15.9.1.2 압축 테이블 만들기

압축된 테이블은 파일 단위 테이블 공간 또는 일반 테이블 공간에서 만들 수 있습니다. InnoDB 시스템 테이블 스페이스에는 테이블 압축을 사용할 수 없습니다. 시스템 테이블 스페이스(공간 0, 공간

.ibdata 파일)에는 사용자가 만든 테이블이 포함될 수 있지만, 압축되지 않는 내부 시스템 데이터도 포함됩니다. 따라서 압축은 테이블별 파일 또는 일반 테이블 스페이스에 저장된 테이블(및 인덱스)에만 적용됩니다.

파일 단위 테이블 공간에서 압축 테이블 만들기

테이블별 파일 테이블 공간에서 압축 테이블을 생성하려면 `innodb_file_per_table`을 사용하도록 설정해야 합니다(기본값). 이 매개변수는 MySQL 구성 파일(`my.cnf` 또는 `my.ini`)에서 설정하거나 `SET` 문을 사용하여 동적으로 설정할 수 있습니다.

`innodb_file_per_table` 옵션을 구성한 후 `CREATE TABLE` 또는 `ALTER TABLE` 문에 `ROW_FORMAT=COMPRESSED` 절 또는 `KEY_BLOCK_SIZE` 절 또는 둘 다를 지정하여 파일 단위 테이블 테이블 공간에 압축 테이블을 생성합니다.

예를 들어 다음과 같은 문을 사용할 수 있습니다:

```
SET GLOBAL innodb_file_per_table=1;
테이블 t1 생성
 (c1 INT PRIMARY KEY)
 ROW_FORMAT=COMPRESSED
 KEY_BLOCK_SIZE=8;
```

일반 테이블 스페이스에서 압축 테이블 만들기

일반 테이블스페이스에서 압축 테이블을 생성하려면 테이블스페이스를 생성할 때 지정한 일반 테이블스페이스에 대해 `FILE_BLOCK_SIZE`를 정의해야 한다. `FILE_BLOCK_SIZE` 값은 `innodb_page_size` 값과 압축 테이블의 페이지 크기와 관련하여 유효한 압축 페이지 크기여야 하며, `CREATE TABLE` 또는 `ALTER TABLE KEY_BLOCK_SIZE`에 의해 정의됩니다.
절의 값은 `FILE_BLOCK_SIZE/1024`와 같아야 합니다. 예를 들어, innodb_page_size=16384이고 `FILE_BLOCK_SIZE=8192`인 경우 테이블의 `KEY_BLOCK_SIZE`는 8이어야 합니다. 자세한 내용은 섹션 15.6.3.3, "일반 테이블 스페이스"를 참조하십시오.

다음 예제에서는 일반 테이블 스페이스를 생성하고 압축 테이블을 추가하는 방법을 보여줍니다. 이 예에서는 기본 innodb_page_size가 16K라고 가정합니다. FILE_BLOCK_SIZE가 8192이면 압축 테이블의 KEY_BLOCK_SIZE가 8이어야 합니다.

```
mysql> CREATE TABLESPACE `ts2` ADD DATAFILE 'ts2.ibd' FILE_BLOCK_SIZE = 8192 엔진=InnoDB;

mysql> CREATE TABLE t4 (c1 INT PRIMARY KEY) TABLESPACE ts2 ROW_FORMAT=COMPRESSED KEY_BLOCK_SIZE=8;
```

참고

- MySQL 8.2부터 압축 테이블의 테이블스페이스 파일은 `InnoDB` 페이지 크기 대신 물리적 페이지 크기를 사용하여 생성되므로 빈 압축 테이블에 대한 테이블스페이스 파일의 초기 크기가 이전 MySQL 릴리스보다 작아집니다.
- `ROW_FORMAT=COMPRESSED`를 지정하는 경우, `KEY_BLOCK_SIZE`를 생략할 수 있습니다.
  설정의 기본값은 `innodb_page_size` 값의 절반입니다.

으며, 압축이 자동으로 활성화됩니다.

- 유효한 `KEY_BLOCK_SIZE` 값을 지정하면 `ROW_FORMAT=COMPRESSED`를 생략할 수 있으며, 압축이 자동으로 활성화됩니다.

- `KEY_BLOCK_SIZE`에 가장 적합한 값을 결정하려면 일반적으로 이 절의 값이 다른 동일한 테이블의 복사본을 여러 개 만든 다음 결과 `.ibd` 파일의 크기를 측정합니다.
  를 참조하여 실제 워크로드에서 각각의 성능이 얼마나 우수한지 확인하세요. 일반 테이블 공간의 경우 테이블을 삭제해도 일반 테이블 공간 `.ibd` 파일의 크기가 줄어들지 않으며 운영 체제에 디스크 공간이 반환되지 않는다는 점에 유의하세요. 자세한 내용은 섹션 15.6.3.3, "일반 테이블 스페이스"를 참조하십시오.

- `KEY_BLOCK_SIZE` 값은 힌트로 취급되며, 필요한 경우 `InnoDB`에서 다른 크기를 사용할 수 있습니다. 테이블별 파일 테이블스페이스의 경우, KEY_BLOCK_SIZE는 `innodb_PAGE_SIZE` 값보다 작거나 같을 수 있습니다. `innodb_page_size`보다 큰 값을 지정하는 경우
  값으로 지정하면 지정된 값이 무시되고 경고가 발행되며 KEY_BLOCK_SIZE는

innodb_page_size 값. innodb_strict_mode가 ON인 경우, 잘못된 KEY_BLOCK_SIZE 값을 지정하면 오류가 반환됩니다. 일반 테이블스페이스의 경우, 유효한 키 블록 크기 값은 테이블스페이스의 파일 블록 크기 설정에 따라 달라진다. 자세한 내용은 섹션 15.6.3.3, "일반 테이블 스페이스"를 참조하십시오.

- InnoDB는 32KB 및 64KB 페이지 크기를 지원하지만 이러한 페이지 크기는 압축을 지원하지 않습니다. 자세한 내용은 innodb_page_size 설명서를 참조하세요.
- InnoDB 데이터 페이지의 기본 비압축 크기는 16KB입니다. 옵션 값의 조합에 따라 MySQL은 테이블스페이스 데이터 파일(.ibd 파일)에 1KB, 2KB, 4KB, 8KB 또는 16KB의 페이지 크기를 사용합니다. 실제 압축 알고리즘은 KEY_BLOCK_SIZE 값의 영향을 받지 않으며, 이 값은 압축된 각 청크의 크기를 결정하고, 이는 다시 각 압축 페이지에 패킹할 수 있는 행 수에 영향을 줍니다.
- 파일 단위 테이블 테이블 스페이스에서 압축 테이블을 생성할 때 KEY_BLOCK_SIZE를 InnoDB 페이지 크기와 동일하게 설정해도 일반적으로 압축이 많이 발생하지 않습니다. 예를 들어, KEY_BLOCK_SIZE=16을 설정하면 일반적으로일반적인 InnoDB 페이지 크기는 16KB입니다. 이 설정은 긴 BLOB, VARCHAR 또는 TEXT 열이 많은 테이블에 여전히 유용할 수 있습니다. 이러한 값은 압축이 잘 되는 경우가 많으므로 15.9.1.5절 "InnoDB 테이블의 압축 작동 방식"에 설명된 대로 오버플로 페이지가 더 적게 필요할 수 있기 때문입니다. 일반 테이블 스페이스의 경우 InnoDB 페이지 크기와 동일한 KEY_BLOCK_SIZE 값은 허용되지 않습니다. 자세한 내용은 섹션 15.6.3.3, "일반 테이블 스페이스"를 참조하십시오.
- 테이블의 모든 인덱스(클러스터된 인덱스 포함)는 CREATE TABLE 또는 ALTER TABLE 문에 지정된 대로 동일한 페이지 크기를 사용하여 압축됩니다. ROW_FORMAT 및 KEY_BLOCK_SIZE와 같은 테이블 특성은 InnoDB 테이블에 대한 CREATE INDEX 구문의 일부가 아니며, 지정된 경우 무시됩니다(지정된 경우 SHOW CREATE TABLE 문의 출력에 나타나지만).
- 성능 관련 구성 옵션은 섹션 15.9.1.3, 'InnoDB 테이블의 압축 조정'을 참조하십시오.

압축 테이블에 대한 제한 사항

- 압축된 테이블은 InnoDB 시스템 테이블 스페이스에 저장할 수 없습니다.
- 일반 테이블 스페이스에는 여러 테이블이 포함될 수 있지만 압축된 테이블과 압축되지 않은 테이블은 동일한 일반 테이블 스페이스 내에 공존할 수 없습니다.
- 압축은 개별 행이 아닌 전체 테이블과 관련된 모든 인덱스에 적용되며, 절 이름이 ROW_FORMAT임에도 불구하고 압축은 개별 행에 적용되지 않습니다.
- InnoDB는 압축된 임시 테이블을 지원하지 않습니다. innodb_strict_mode가 활성화된 경우(기본값), 행 형식이 압축되거나 키 블록 크기가 지정되면 CREATE TEMPORARY TABLE이 오류를 반환합니다. innodb_strict_mode가 비활성화되면 경고가 표시되고 압축되지 않은 행 형식을 사용하여 임시 테이블이 생성됩니다. 임시 테이블에 대한 ALTER TABLE 작업에도 동일한 제한이 적용됩니다.

15.9.1.3 InnoDB 테이블에 대한 압축 조정

대부분의 경우 InnoDB 데이터 저장 및 압축에 설명된 내부 최적화를 통해 시스템이 압축된 데이터로 잘 실행되도록 보장합니다. 그러나 압축의 효율성은 데이터의 특성에 따라 달라지므로 압축된 테이블의 성능에 영향을 미치는 결정을 내릴 수 있습니다:

- 압축할 테이블.
- 사용할 압축 페이지 크기.
- 시스템이 데이터를 압축 및 압축 해제하는 데 소요되는 시간 등 런타임 성능 특성에 따라 버퍼 풀의 크기를 조정할지 여부. 워크로드가 데이터 웨어하우스(주로 쿼리)에 더 가까운지 아니면 OLTP 시스템(쿼리와 DML의 혼합)에 더 가까운지 여부.

- 시스템에서 압축된 테이블에 대해 DML 작업을 수행하고 데이터가 배포되는 방식이 런타임에 많은 비용이 드는 압축 실패로 이어지는 경우 추가 고급 구성 옵션을 조정할 수 있습니다.

이 섹션의 지침을 사용하여 이러한 아키텍처 및 구성 선택을 하세요. 장기 테스트를 수행하고 압축된 테이블을 프로덕션에 적용할 준비가 되면 15.9.1.4절, '런타임에 InnoDB 테이블 압축 모니터링'을 참조하여 실제 조건에서 이러한 선택의 효과를 검증하는 방법을 확인하세요.

압축을 사용해야 하는 경우

일반적으로 압축은 적절한 수의 문자열 열이 포함되어 있고 데이터를 쓰는 횟수보다 읽는 횟수가 훨씬 많은 테이블에서 가장 잘 작동합니다. 특정 상황에서 압축이 도움이 되는지 여부를 예측할 수 있는 확실한 방법은 없으므로 항상 대표적인 구성에서 실행되는 특정 워크로드 및 데이터 세트로 테스트하세요. 압축할 테이블을 결정할 때는 다음 요소를 고려하세요.

데이터 특성 및 압축

데이터 파일의 크기를 줄이는 데 있어 압축의 효율성을 결정하는 핵심 요소는 데이터 자체의 특성입니다. 압축은 데이터 블록에서 반복되는 바이트 문자열을 식별하는 방식으로 작동한다는 점을 기억하세요. 완전히 무작위화된 데이터는 최악의 경우입니다. 일반적인 데이터에는 반복되는 값이 있는 경우가 많으므로 효과적으로 압축됩니다. 문자 문자열은 `CHAR`, `VARCHAR`, `TEXT` 또는 `BLOB` 열에 정의되어 있든 상관없이 잘 압축되는 경우가 많습니다. 반면, 대부분 이진 데이터(정수 또는 부동 소수점 숫자)를 포함하는 테이블이나 이전에 압축된 데이터(예: JPEG 또는 PNG 이미지)는 일반적으로 압축이 잘 되지 않거나 크게 또는 전혀 압축되지 않을 수 있습니다.

각 InnoDB 테이블에 대해 압축을 설정할지 여부를 선택합니다. 테이블과 모든 인덱스는 동일한(압축된) 페이지 크기를 사용합니다. 테이블의 모든 열에 대한 데이터를 포함하는 기본 키(클러스터된) 인덱스가 보조 인덱스보다 더 효과적으로 압축될 수 있습니다. 긴 행이 있는 경우 압축을 사용하면 동적 행 형식에서 설명한 대로 긴 열 값이 "페이지 외부"에 저장될 수 있습니다. 이러한 오버플로 페이지는 압축이 잘 될 수 있습니다. 이러한 고려 사항을 고려할 때, 많은 애플리케이션에서 일부 테이블은 다른 테이블보다 더 효과적으로 압축되며, 일부 테이블을 압축한 경우에만 워크로드가 가장 잘 수행될 수 있습니다.

특정 테이블을 압축할지 여부를 결정하려면 실험을 수행하세요. 압축되지 않은 테이블의 .ibd 파일 사본에 LZ77 압축을 구현하는 유틸리티(예: `gzip` 또는 WinZip)를 사용하여 데이터를 얼마나 효율적으로 압축할 수 있는지 대략적으로 추정할 수 있습니다. MySQL은 페이지 크기(기본적으로 16KB)를 기준으로 데이터를 청크로 압축하기 때문에 파일 기반 압축 도구보다 MySQL 압축 테이블의 압축률이 낮을 것으로 예상할 수 있습니다. 또한
페이지 형식에는 압축되지 않은 일부 내부 시스템 데이터가 포함됩니다. 파일 기반 압축 유틸리티는 훨씬 더 큰 데이터 청크를 검사할 수 있으므로 MySQL이 개별 페이지에서 찾을 수 있는 것보다 큰 파일에서 반복되는 문자열을 더 많이 찾을 수 있습니다.

특정 테이블에서 압축을 테스트하는 또 다른 방법은 압축되지 않은 테이블의 일부 데이터를 테이블별 파일

테이블 공간에 있는 유사한 압축 테이블(인덱스가 모두 동일)로 복사하고 결과 `.ibd` 파일의 크기를 살펴보는 것입니다. 예를 들어

```
사용 테스트;
SET GLOBAL innodb_file_per_table=1;
SET GLOBAL autocommit=0;

-- 한두 개의 행이 있는 비압축 테이블을 만듭니다.
CREATE TABLE big_table AS SELECT * FROM information_schema.columns;
INSERT INTO big_table SELECT * FROM big_table;
INSERT INTO big_table SELECT * FROM big_table;
INSERT INTO big_table SELECT * FROM big_table;
INSERT INTO big_table SELECT * FROM big_table;
INSERT INTO big_table SELECT * FROM big_table;
INSERT INTO big_table SELECT * FROM big_table;
INSERT INTO big_table SELECT * FROM big_table;
INSERT INTO big_table SELECT * FROM big_table;
INSERT INTO big_table SELECT * FROM big_table;
INSERT INTO big_table SELECT * FROM big_table;
```

```
커밋;
ALTER TABLE big_table ADD id int 부호 없음 NOT NULL PRIMARY KEY auto_increment;

SHOW CREATE TABLE big_table\G

select count(id) from big_table;

-- 압축되지 않은 테이블에 필요한 공간을 확인합니다.
\ls -l 데이터/테스트/빅테이블.ibd

CREATE TABLE key_block_size_4 LIKE big_table;
ALTER TABLE key_block_size_4 key_block_size=4 row_format=compressed;

INSERT INTO key_block_size_4 SELECT * FROM big_table;
commit;

-- 압축된 테이블에 필요한 공간 확인
-- 특정 압축 설정으로.
\ls -l 데이터/테스트/키_블록_크기_4.ibd
```

이 실험에서 다음과 같은 숫자가 생성되었는데, 물론 테이블 구조와 데이터에 따라 상당히 달라질 수 있습니다:

```
-rw-rw---- 1 cirrus 직원 310378496 Jan 9 13:44 data/test/big_table.ibd
-rw-rw---- 1 cirrus 직원 83886080 Jan 9 15:10 data/test/key_block_size_4.ibd
```

특정 워크로드에 압축이 효율적인지 확인합니다:

- 간단한 테스트를 위해 다른 압축 테이블이 없는 MySQL 인스턴스를 사용하여 정보 스키마 `INNODB_CMP` 테이블에 대한 쿼리를 실행합니다.
- 여러 개의 압축 테이블이 있는 워크로드와 관련된 보다 정교한 테스트의 경우, 정보 스키마 `INNODB_CMP_PER_INDEX` 테이블에 대해 쿼리를 실행합니다. 이 테이블의 통계는 `INNODB_CMP_PER_INDEX` 테이블은 수집 비용이 많이 들기 때문에 해당 테이블을 쿼리하기 전에 구성 옵션 `innodb_cmp_per_index_enabled`를 활성화해야 하며, 이러한 테스트를 개발 서버 또는 중요하지 않은 복제본 서버로 제한할 수 있습니다.
- 테스트 중인 압축 테이블에 대해 몇 가지 일반적인 SQL 문을 실행합니다.
- `INFORMATION_SCHEMA.`INNODB_CMP를 쿼리하여 전체 압축 작업 대비 성공적인 압축 작업의 비율을 검사합니다.
  `INFORMATION_SCHEMA.`INNODB_CMP_PER_INDEX를 비교하고 COMPRESS_OPS와 `compress_ops_ok`.
- 압축 작업이 성공적으로 완료되는 비율이 높으면 테이블이 압축에 적합한 후보일 수 있습니다.
- 압축 실패 비율이 높은 경우 15.9.1.6절. "OLTP 워크로드용 압축"에 설명된 대로 `innodb_compression_level`, `innodb_compression_failure_threshold_pct` 및 `innodb_compression_pad_pct_max` 옵션을 조정하고 추가 테스트를 시도할 수 있습니다.

데이터베이스 압축과 애플리케이션 압축 비교

애플리케이션에서 데이터를 압축할지 테이블에서 압축할지 결정하고 동일한 데이터에 두 가지 유형의 압축을

모두 사용하지 마십시오. 애플리케이션에서 데이터를 압축하고 그 결과를 압축된 테이블에 저장하면 공간을 추가로 절약할 가능성이 거의 없으며 이중 압축으로 인해 CPU 사이클만 낭비하게 됩니다.

데이터베이스에서 압축

사용하도록 설정하면 MySQL 테이블 압축이 자동으로 수행되며 모든 열과 인덱스 값에 적용됩니다. 열은 여전히 `LIKE`와 같은 연산자로 테스트할 수 있으며, 인덱스 값이 압축된 경우에도 정렬 작업은 여전히 인덱스를 사용할 수 있습니다. 인덱스는 데이터베이스 전체 크기의 상당 부분을 차지하는 경우가 많으므로 압축을 통해 스토리지, I/O 또는 프로세서 시간을 크게 절약할 수 있습니다. 압축 및 압축 해제 작업은 예상되는 부하를 처리할 수 있는 규모의 강력한 시스템인 데이터베이스 서버에서 이루어집니다.

애플리케이션에서 압축하기

애플리케이션에서 텍스트와 같은 데이터를 데이터베이스에 삽입하기 전에 압축하는 경우, 일부 열만 압축하고 다른 열은 압축하지 않음으로써 잘 압축되지 않는 데이터에 대한 오버헤드를 절약할 수 있습니다.
이 접근 방식은 데이터베이스 서버가 아닌 클라이언트 시스템에서 압축 및 압축 해제를 위해 CPU 사이클을 사용하므로 클라이언트가 많은 분산 애플리케이션이나 클라이언트 시스템에 여유 CPU 사이클이 있는 경우에 적합할 수 있습니다.

하이브리드 접근 방식

물론 이러한 접근 방식을 결합할 수도 있습니다. 일부 애플리케이션의 경우 일부 압축 테이블과 일부 비압축 테이블을 사용하는 것이 적절할 수 있습니다. 일부 데이터는 외부에서 압축하여 비압축 테이블에 저장하고 애플리케이션의 다른 테이블은 MySQL이 압축하도록 허용하는 것이 가장 좋습니다. 언제나 그렇듯이, 올바른 결정을 내리기 위해서는 사전 설계와 실제 테스트가 중요합니다.

워크로드 특성 및 압축

압축할 테이블(및 페이지 크기)을 선택하는 것 외에도 워크로드는 성능의 또 다른 주요 결정 요인입니다. 애플리케이션이 업데이트가 아닌 읽기가 주를 이루는 경우, 인덱스 페이지가 압축된 데이터에 대해 유지 관리하는 페이지별 '수정 로그'를 위한 공간이 부족해지면 페이지를 재구성하고 다시 압축해야 하는 페이지 수가 줄어듭니다. 업데이트가 주로 인덱싱되지 않은 열이나 "페이지 외부"에 저장되는 BLOB 또는 큰 문자열을 포함하는 열을 변경하는 경우 압축 오버헤드가 허용될 수 있습니다. 테이블에 대한 유일한 변경 사항이 단조롭게 증가하는 기본 키를 사용하는 `INSERT`이고 보조 인덱스가 거의 없는 경우 인덱스 페이지를 재구성하고 다시 압축할 필요가 거의 없습니다. MySQL은 행을 "삭제 표시"하고 삭제할 수 있으므로
압축되지 않은 데이터를 수정하여 '제자리'에 있는 압축 페이지에서 테이블의 삭제 작업은 상대적으로 효율적입니다.

일부 환경에서는 데이터를 로드하는 데 걸리는 시간이 런타임 검색만큼이나 중요할 수 있습니다. 특히 데이터 웨어하우스 환경에서는 많은 테이블이 읽기 전용이거나 대부분 읽기일 수 있습니다. 이러한 경우, 디스크 읽기 횟수나 스토리지 비용 절감 효과가 크지 않다면 압축으로 인한 로드 시간 증가라는 대가를 지불할 수 있을 수도 있고 그렇지 않을 수도 있습니다.

기본적으로 압축은 데이터를 압축하고 압축을 푸는 데 CPU 시간을 사용할 수 있을 때 가장 잘 작동합니다. 따라서 워크로드가 CPU에 종속되지 않고 I/O에 종속되는 경우 압축을 통해 전반적인 성능을 향상시킬 수 있습니다. 다양한 압축 구성으로 애플리케이션 성능을 테스트할 때는 프로덕션 시스템의 계획된 구성과 유사한 플랫폼에서 테스트하세요.

구성 특성 및 압축

디스크에서 데이터베이스 페이지를 읽고 쓰는 작업은 시스템 성능에서 가장 느린 부분입니다. 압축은 CPU 시

간을 사용하여 데이터를 압축 및 압축 해제함으로써 I/O를 줄이려고 시도하며, 프로세서 주기에 비해 I/O가 상대적으로 부족한 리소스일 때 가장 효과적입니다.

특히 빠른 멀티코어 CPU를 사용하는 다중 사용자 환경에서 실행할 때 이러한 문제가 자주 발생합니다. 압축된 테이블의 페이지가 메모리에 있는 경우 MySQL은 페이지의 비압축 복사본을 위해 버퍼 풀에 추가 메모리(일반적으로 16KB)를 사용하는 경우가 많습니다. 적응형 LRU 알고리즘은 워크로드가 I/O 바운드 방식으로 실행되는지 CPU 바운드 방식으로 실행되는지를 고려하여 압축 페이지와 비압축 페이지 간에 메모리 사용의 균형을 맞추려고 시도합니다. 하지만 버퍼 풀 전용 메모리가 더 많은 구성이 메모리 제약이 심한 구성보다 압축 테이블을 사용할 때 더 잘 실행되는 경향이 있습니다.

압축 페이지 크기 선택

압축 페이지 크기의 최적 설정은 테이블과 인덱스에 포함된 데이터의 유형과 분포에 따라 달라집니다. 압축된 페이지 크기는 항상 최대 레코드 크기보다 커야 하며, 그렇지 않으면 B 트리 페이지 압축에서 설명한 대로 작업이 실패할 수 있습니다.

압축 페이지 크기를 너무 크게 설정하면 공간이 다소 낭비되지만 페이지를 자주 압축할 필요는 없습니다. 압축 페이지 크기를 너무 작게 설정하면 삽입 또는 업데이트 시 시간이 많이 걸리는 재압축이 필요할 수 있으며, B-트리 노드를 더 자주 분할해야 하므로 데이터 파일 크기가 커지고 인덱싱 효율이 떨어질 수 있습니다.

일반적으로 압축 페이지 크기를 8K 또는 4K 바이트로 설정합니다. InnoDB 테이블의 최대 행 크기가 약 8K라는 점을 감안할 때, 일반적으로 `KEY_BLOCK_SIZE=8`을 선택하는 것이 안전합니다.

15.9.1.4 런타임 시 InnoDB 테이블 압축 모니터링

전반적인 애플리케이션 성능, CPU 및 I/O 사용률, 디스크 파일 크기는 압축이 애플리케이션에 얼마나 효과적인지 알 수 있는 좋은 지표입니다. 이 섹션에서는 15.9.1.3절 "InnoDB 테이블의 압축 튜닝"의 성능 튜닝 조언을 기반으로 초기 테스트 중에 나타나지 않을 수 있는 문제를 찾는 방법을 설명합니다.

압축 테이블에 대한 성능 고려 사항을 자세히 알아보려면 예제 15.1, "압축 정보 스키마 테이블 사용"에 설명된 정보 스키마 테이블을 사용하여 런타임에 압축 성능을 모니터링할 수 있습니다. 이러한 테이블에는 내부 메모리 사용량과 전체적으로 사용되는 압축률이 반영되어 있습니다.

`INNODB_CMP` 테이블은 사용 중인 각 압축 페이지 크기(`KEY_BLOCK_SIZE`)에 대한 압축 활동에 대한 정보를 보고합니다. 이 테이블의 정보는 시스템 전체에 대한 것으로 데이터베이스의 모든 압축 테이블에 대한 압축 통계를 요약합니다. 이 데이터를 사용하면 다른 압축 테이블에 액세스하지 않을 때 이 테이블을 검사하여 테이블을 압축할지 여부를 결정하는 데 도움이 될 수 있습니다. 서버에 상대적으로 적은 오버헤드가 발생하므로 프로덕션 서버에서 주기적으로 쿼리하여 압축 기능의 전반적인 효율성을 확인할 수 있습니다.

`INNODB_CMP_PER_INDEX` 테이블은 개별 테이블 및 인덱스에 대한 압축 활동에 대한 정보를 보고합니다. 이 정보는 압축 효율성을 평가하고 한 번에 한 테이블 또는 인덱스씩 성능 문제를 진단하는 데 더 타겟팅되고 더 유용합니다. (각 `InnoDB` 테이블은 클러스터된 인덱스로 표시되므로, MySQL은 이 맥락에서 테이블과 인덱스를 크게 구분하지 않습니다). `INNODB_CMP_PER_INDEX` 테이블은 상당한 오버헤드를 수반하므로 다양한 워크로드, 데이터 및 압축 설정의 효과를 개별적으로 비교할 수 있는 개발 서버에 더 적합합니다. 실수로 이 모니터링 오버헤드가 부과되는 것을 방지하려면, `INNODB_CMP_PER_INDEX` 테이블을 쿼리하기 전에 `innodb_cmp_per_index_enabled` 구성 옵션을 활성화해야 합니다.

고려해야 할 주요 통계는 압축 및 압축 해제 작업의 수와 수행에 소요된 시간입니다. MySQL은 수정 후 압축된 데이터를 더 이상 포함할 수 없을 정도로 가득 차면 B-트리 노드를 분할하므로 "성공한" 압축 작업의 수와 전체 압축 작업의 수를 비교합니다. `INNODB_CMP` 및 `INNODB_CMP_PER_INDEX` 테이블의 정보와 전반적인 애플리케이션 성능 및 하드웨어 리소스 사용률에 따라 하드웨어 구성을 변경하거나 버퍼 풀의 크기를 조정하거나 다른 페이지 크기를 선택하거나 압축할 다른 테이블 집합을 선택할 수 있습니다.

압축 및 압축 해제에 필요한 CPU 시간이 많은 경우, 동일한 데이터, 애플리케이션 워크로드 및 압축된 테이블 집합에 대해 더 빠른 CPU 또는 멀티코어 CPU로 변경하면 성능을 개선하는 데 도움이 될 수 있습니다. 버퍼 풀의 크기를 늘리는 것도 성능에 도움이 될 수 있습니다.
더 많은 비압축 페이지가 메모리에 남을 수 있으므로 압축된 형태로만 메모리에 존재하는 페이지를 압축 해제할

필요가 줄어듭니다.

전체적으로 압축 작업 수가 많으면(애플리케이션의 `INSERT`, `UPDATE`, `DELETE` 작업 수 및 데이터베이스 크기와 비교하여) 압축된 테이블 중 일부가 너무 많이 업데이트되어 효과적인 압축이 이루어지지 않고 있음을 나타낼 수 있습니다. 이 경우 더 큰 페이지 크기를 선택하거나 압축할 테이블을 더 신중하게 선택하세요.

"성공한" 압축 작업 수(`COMPRESS_OPS_OK`)가 총 압축 작업 수(`COMPRESS_OPS`)의 높은 비율이면 시스템이 잘 작동하고 있는 것입니다. 이 비율이 낮으면 MySQL이 B 트리 노드를 재구성, 재압축 및 분할하는 빈도가 바람직하지 않은 것입니다. 이 경우 일부 테이블을 압축하지 않거나 `KEY_BLOCK_SIZE`

를 설정할 수 있습니다. 애플리케이션에서 '압축 실패' 횟수가 전체의 1% 또는 2%를 초과하는 테이블에 대해서는 압축을 해제할 수 있습니다. (이러한 실패율은 데이터 로드와 같은 일시적인 작업 중에는 허용될 수 있습니다.)

15.9.1.5 InnoDB 테이블의 압축 작동 방식

이 섹션에서는 InnoDB 테이블의 압축에 대한 몇 가지 내부 구현 세부 사항을 설명합니다. 여기에 제시된 정보는 성능을 조정하는 데 도움이 될 수 있지만 압축의 기본적인 사용을 위해 반드시 알아야 할 필요는 없습니다.

압축 알고리즘

일부 운영 체제는 파일 시스템 수준에서 압축을 구현합니다. 파일은 일반적으로 가변 크기 블록으로 압축되는 고정 크기 블록으로 나뉘며, 이는 쉽게 조각화로 이어집니다.
블록 내부의 내용이 수정될 때마다 전체 블록이 디스크에 기록되기 전에 다시 압축됩니다. 이러한 특성으로 인해 이 압축 기술은 업데이트 집약적인 데이터베이스 시스템에서 사용하기에 부적합합니다.

MySQL은 LZ77 압축 알고리즘을 구현하는 잘 알려진 zlib 라이브러리의 도움으로 압축을 구현합니다. 이 압축 알고리즘은 성숙하고 강력하며 CPU 사용률과 데이터 크기 감소 모두에서 효율적입니다. 이 알고리즘은 "무손실" 방식이므로 압축되지 않은 원본 데이터를 압축된 형태에서 항상 재구성할 수 있습니다. LZ77 압축은 압축할 데이터 내에서 반복되는 데이터 시퀀스를 찾는 방식으로 작동합니다. 데이터의 값 패턴에 따라 압축률이 결정되지만, 일반적인 사용자 데이터는 50% 이상 압축되는 경우가 많습니다.

애플리케이션에 의해 수행되는 압축이나 다른 데이터베이스 관리 시스템의 압축 기능과 달리, InnoDB 압축은 사용자 데이터와 인덱스에 모두 적용됩니다. 대부분의 경우 인덱스가 전체 데이터베이스 크기의 40~50% 이상을 차지할 수 있으므로 이 차이는 상당히 큽니다. 압축이 데이터 세트에 대해 잘 작동하는 경우, InnoDB 데이터 파일(테이블별 파일 테이블 스페이스 또는 일반 테이블 스페이스 `.ibd` 파일)의 크기는 압축되지 않은 크기의 25~50% 또는 그보다 작을 수 있습니다. 워크로드에 따라 이 작은 데이터베이스는 다음과 같은 결과를 가져올 수 있습니다.
CPU 사용률 증가에 따른 적당한 비용으로 I/O 감소와 처리량 증가를 얻을 수 있습니다. 압축 수준과 CPU 오버헤드 간의 균형은 `innodb_compression_level` 구성 옵션을 수정하여 조정할 수 있습니다.

InnoDB 데이터 스토리지 및 압축

InnoDB 테이블의 모든 사용자 데이터는 B-트리 인덱스(클러스터된 인덱스)로 구성된 페이지에 저장됩니다. 일부 다른 데이터베이스 시스템에서는 이러한 유형의 인덱스를 "인덱스 구성 테이블"이라고 합니다. 인덱스 노드의 각 행에는 (사용자 지정 또는 시스템 생성) 기본 키의 값과 테이블의 다른 모든 열이 포함됩니다.

InnoDB 테이블의 보조 인덱스도 B-트리이며, 인덱스 키와 클러스터된 인덱스의 행에 대한 포인터라는 값 쌍을 포함합니다. 포인터는 실제로 테이블의 기본 키 값으로, 인덱스 키와 기본 키 이외의 열이 필요한 경우 클러스터된 인덱스에 액세스하는 데 사용됩니다. 보조 인덱스 레코드는 항상 단일 B-트리 페이지에 맞아야 합니다.

B-트리 노드(클러스터된 인덱스와 보조 인덱스 모두)의 압축은 다음 섹션에서 설명하는 것처럼 긴 `VARCHAR`, `BLOB` 또는 `TEXT` 열을 저장하는 데 사용되는 오버플로 페이지의 압축과 다르게 처리됩니다.

B-Tree 페이지 압축

B트리 페이지는 자주 업데이트되기 때문에 특별한 관리가 필요합니다. B-트리 노드가 분할되는 횟수를 최소화하고 콘텐츠의 압축을 풀고 다시 압축해야 하는 횟수를 최소화하는 것이 중요합니다.

MySQL이 사용하는 한 가지 기술은 B-tree 노드에서 일부 시스템 정보를 압축되지 않은 형태로 유지하여 특정 인플레이스 업데이트를 용이하게 하는 것입니다. 예를 들어, 이를 통해 압축 작업 없이 행을 삭제 표시하고 삭제할 수 있습니다.

또한 MySQL은 인덱스 페이지가 변경될 때 불필요한 압축 해제 및 재압축을 피하려고 시도합니다. 각 B-트리 페이지 내에서 시스템은 압축되지 않은 "수정 로그"를 유지하여 페이지에 대한 변경 사항을 기록합니다. 전체 페이지를 완전히 재구성하지 않고도 이 수정 로그에 작은 레코드의 업데이트 및 삽입을 기록할 수 있습니다.

수정 로그를 위한 공간이 부족해지면 InnoDB는 페이지 압축을 풀고 변경 사항을 적용한 후 페이지를 다시 압축합니다. 재압축에 실패하면(압축 실패라고 알려진 상황), B-트리 노드가 분할되고 업데이트 또는 삽입이 성공할 때까지 프로세스가 반복됩니다.

OLTP 애플리케이션과 같이 쓰기 집약적인 워크로드에서 잦은 압축 실패를 방지하기 위해 MySQL은 때때로 페이지에 약간의 빈 공간(패딩)을 예약하여 수정 로그가 더 빨리 채워지고 페이지가 분할되지 않도록 충분한 공간이 남아있을 때 페이지를 다시 압축합니다.
시스템에서 페이지 분할 빈도를 추적하기 때문에 각 페이지에 남는 패딩 공간의 양은 달라집니다. 압축 테이블에 자주 쓰기를 수행하는 바쁜 서버에서는 `innodb_compression_failure_threshold_pct` 및 `innodb_compression_pad_pct_max` 구성 옵션을 사용하여 이 메커니즘을 미세 조정할 수 있습니다.

일반적으로 MySQL은 InnoDB 테이블의 각 B 트리 페이지가 최소 2개의 레코드를 수용할 수 있어야 합니다. 압축 테이블의 경우 이 요구 사항이 완화되었습니다. B-트리 노드(기본 키 또는 보조 인덱스)의 리프 페이지는 하나의 레코드만 수용하면 됩니다.
레코드는 압축되지 않은 형태로 페이지별 수정 로그에 맞아야 합니다. innodb_strict_mode가 `ON`인 경우 MySQL은 테이블 생성 또는 인덱스 생성 중에 최대 행 크기를 확인합니다. 행이 맞지 않으면 다음과 같은 오류 메시지가 발생합니다: 오류 `HY000: 너무 큰 행입니다`.

innodb_strict_mode가 OFF일 때 테이블을 생성하고 후속 `INSERT` 또는 `UPDATE` 문이 압축된 페이지 크기에 맞지 않는 인덱스 항목을 생성하려고 시도하면 오류 `42000`과 함께 작업이 실패합니다: 행 크기가 너무 큽니다. (이 오류 메시지는 레코드가 너무 큰 인덱스의 이름을 지정하거나 인덱스 레코드의 길이 또는 특정 인덱스 페이지의 최대 레코드 크기를 언급하지 않습니다.) 이 문제를 해결하려면 `ALTER TABLE`을 사용하여 테이블을 다시 빌드하고 더 큰 압축 페이지 크기(`KEY_BLOCK_SIZE`)를 선택하거나, 열 접두사 인덱스를 줄이거나, `ROW_FORMAT=DYNAMIC` 또는 `ROW_FORMAT=COMPACT`로 압축을 완전히 비활성화합니다.

압축 테이블도 지원하는 일반 테이블스페이스에는 innodb_strict_mode를 적용할 수 없습니다. 일반 테이블스페이스에 대한 테이블스페이스 관리 규칙은 `innodb_strict_mode`와 무관하게 엄격하게 적용됩니다. 자세한 내용은 13.1.21절, "테이블스페이스 생성 문"을 참조한다.

BLOB, VARCHAR 및 TEXT 열 압축하기

InnoDB 테이블에서 기본 키에 속하지 않는 `BLOB`, `VARCHAR` 및 `TEXT` 열은 별도로 할당된 오버플로 페이지에 저장될 수 있습니다. 이러한 열을 오프페이지 열이라고 합니다. 이러한 열의 값은 단일 링크된 오버플로 페이지 목록에 저장됩니다.

`ROW_FORMAT=DYNAMIC` 또는 `ROW_FORMAT=COMPRESSED`로 생성된 테이블의 경우, `BLOB`, `TEXT` 또는 `VARCHAR` 열의 값은 길이와 전체 행의 길이에 따라 완전히 오프페이지에 저장될 수 있습니다. 오프페이지에

저장된 열의 경우 클러스터된 인덱스 레코드에는 오버플로 페이지에 대한 20바이트 포인터만 열당 하나씩 포함됩니다. 열이 페이지 외부에 저장되는지 여부는 페이지 크기와 행의 전체 크기에 따라 달라집니다. 행이 너무 길어서 클러스터된 인덱스의 페이지에 완전히 들어갈 수 없는 경우, MySQL은 행이 페이지에 맞을 때까지 가장 긴 열을 선택하여 페이지 외부에 저장합니다.

클러스터된 인덱스 페이지입니다. 위에서 언급했듯이 행이 압축된 페이지에 자체적으로 맞지 않는 경우 오류가 발생합니다.

참고

`ROW_FORMAT=DYNAMIC` 또는 `ROW_FORMAT=COMPRESSED`로 생성된 테이블의 경우 40바이트 이하인 `TEXT` 및 `BLOB` 열은 항상 인라인으로 저장됩니다.

`ROW_FORMAT=REDUNDANT` 및 ROW_FORMAT=COMPACT를 사용하는 테이블은 처음 768바이트를 저장합니다. 기본 키와 함께 클러스터된 인덱스 레코드의 `BLOB`, `VARCHAR` 및 `TEXT` 열입니다. 768-

바이트 접두사 뒤에는 나머지 열 값을 포함하는 오버플로 페이지에 대한 20바이트 포인터가 이어집니다.

테이블이 COMPRESSED 형식인 경우 오버플로 페이지에 기록된 모든 데이터는 "있는 그대로" 압축됩니다. 즉, MySQL은 전체 데이터 항목에 zlib 압축 알고리즘을 적용합니다. 압축된 오버플로 페이지에는 데이터 외에 페이지 체크섬과 링크로 구성된 압축되지 않은 헤더와 트레일러가 포함됩니다.
를 다음 오버플로 페이지로 이동하는 등의 작업을 수행합니다. 따라서 텍스트 데이터의 경우처럼 데이터의 압축률이 높은 경우 긴 BLOB, TEXT 또는 VARCHAR 열의 경우 매우 큰 스토리지 절감 효과를 얻을 수 있습니다. JPEG와 같은 이미지 데이터는 일반적으로 이미 압축되어 있으므로 압축 테이블에 저장해도 큰 이점이 없으며, 이중 압축으로 인해 CPU 사이클이 낭비되어 공간을 거의 또는 전혀 절약하지 못할 수 있습니다.

오버플로 페이지는 다른 페이지와 크기가 동일합니다. 페이지 외부에 저장된 열 개 열이 포함된 행은 열의 총 길이가 8K 바이트에 불과하더라도 10개의 오버플로 페이지를 차지합니다. 압축되지 않은 테이블에서는 압축되지 않은 오버플로 페이지 10개가 16만 바이트를 차지합니다. 페이지 크기가 8K인 압축 테이블에서는 80K 바이트만 차지합니다. 따라서 열 값이 긴 테이블에는 압축 테이블 형식을 사용하는 것이 더 효율적인 경우가 많습니다.

테이블별 파일 테이블 스페이스의 경우 16K 압축 페이지 크기를 사용하면 이러한 데이터가 잘 압축되는 경우가 많기 때문에 BLOB, VARCHAR 또는 TEXT 열의 스토리지 및 I/O 비용을 줄일 수 있습니다.
따라서 B-트리 노드 자체는 많은 페이지를 차지하지만 오버플로 페이지가 더 적게 필요합니다.
를 사용할 수 있습니다. 일반 테이블스페이스는 16K 압축 페이지 크기(KEY_BLOCK_SIZE)를 지원하지 않습니다. 자세한 내용은 섹션 15.6.3.3, "일반 테이블 스페이스"를 참조하십시오.

압축 및 InnoDB 버퍼 풀

압축된 InnoDB 테이블에서 모든 압축된 페이지(1K, 2K, 4K 또는 8K)는 16K 바이트의 비압축 페이지(또는 innodb_page_size가설정된 경우 더 작은 크기)에 해당합니다. 페이지의 데이터에 액세스하기 위해 MySQL은 버퍼 풀에 아직 없는 경우 디스크에서 압축된 페이지를 읽은 다음 페이지를 원래 형식으로 압축을 해제합니다. 이 섹션에서는 압축 테이블의 페이지와 관련하여 InnoDB가 버퍼 풀을 관리하는 방법을 설명합니다.

I/O를 최소화하고 페이지 압축을 해제할 필요성을 줄이기 위해 버퍼 풀에 압축된 형태와 압축되지 않은 형태의 데이터베이스 페이지가 모두 포함되는 경우가 있습니다. 다른 필수 데이터베이스 페이지를 위한 공간을 확보하기 위해 MySQL은 압축되지 않은 페이지를 버퍼 풀에서 내보내면서
압축된 페이지를 메모리에 저장합니다. 또는 한동안 페이지에 액세스하지 않은 경우 다른 데이터를 위한 공간을 확보하기 위해 페이지의 압축된 형태가 디스크에 기록될 수 있습니다. 따라서 언제든지 버퍼 풀에는 압축된 페이지와 압축되지 않은 페이지가 모두 포함되거나 압축된 페이지만 포함되거나 둘 다 포함되지 않을 수 있습니다.

MySQL은 최근 사용 빈도가 높은(자주 액세스하는) 데이터가 메모리에 유지되는 경향이 있도록 최근 사용 빈도가 가장 낮은(LRU) 목록을 사용하여 메모리에 보관할 페이지와 제거할 페이지를 추적합니다. 압축 테이블에 액세스할 때 MySQL은 적응형 LRU 알고리즘을 사용하여 메모리에서 압축 페이지와 비압축 페이지의 적절한 균형을 유지합니다. 이 적응형 알고리즘은 시스템이 I/O 바인딩 방식으로 실행되는지 CPU 바인딩 방식으로 실행되는지에 민감하게 반응합니다. 목표는 너무 많은

CPU가 바쁠 때 페이지를 압축 해제하는 데 걸리는 처리 시간을 줄이고, 압축된 페이지를 압축 해제하는 데 사용할 수 있는 여유 사이클이 있을 때 과도한 I/O를 수행하지 않도록 합니다.

메모리에서). 시스템이 I/O 바인딩된 경우 알고리즘은 다른 디스크 페이지가 메모리에 상주할 공간을 더 많이 확보하기 위해 두 복사본이 아닌 압축되지 않은 페이지 복사본을 퇴거하는 것을 선호합니다.

시스템이 CPU에 바인딩된 경우 MySQL은 압축된 페이지와 압축되지 않은 페이지를 모두 제거하여 "핫" 페이지에 더 많은 메모리를 사용할 수 있도록 하고 압축된 형태로만 메모리에 있는 데이터를 압축 해제할 필요성을 줄입니다.

압축 및 InnoDB 재실행 로그 파일

압축된 페이지가 데이터 파일에 기록되기 전에 MySQL은 페이지의 사본을 재실행 로그에 기록합니다(마지막으로 데이터베이스에 기록된 이후 다시 압축된 경우). 이 작업은 다음을 위해 수행됩니다.

는 `zlib` 라이브러리가 업그레이드되어 압축된 데이터에 호환성 문제가 발생하는 경우에도 충돌 복구에 재실행 로그를 사용할 수 있도록 보장합니다. 따라서 다음과 같은 경우 로그 파일 크기가 약간 증가하거나 더 자주 체크포인트가 필요할 수 있습니다.

압축을 사용합니다. 로그 파일 크기 또는 체크포인트 빈도의 증가 정도는 재구성 및 재압축이 필요한 방식으로 압축 페이지를 수정하는 횟수에 따라 달라집니다.

파일 단위 테이블 테이블 스페이스에서 압축 테이블을 생성하려면 `innodb_file_per_table`을 사용하도록 설정해야 합니다. 일반 테이블 스페이스에서 압축 테이블을 생성할 때는 `innodb_file_per_table` 설정에 의존하지 않습니다. 자세한 내용은 섹션 15.6.3.3, "일반 테이블 스페이스"를 참조하십시오.

15.9.1.6 OLTP 워크로드를 위한 압축

기존에는 데이터 웨어하우스 구성과 같이 읽기 전용 또는 읽기 위주의 워크로드에 주로 `InnoDB` 압축 기능을 권장했습니다. 속도가 빠르지만 상대적으로 크기가 작고 가격이 비싼 SSD 저장 장치의 등장으로 `OLTP` 워크로드에도 압축이 매력적으로 다가왔습니다. 트래픽이 많은 대화형 웹사이트에서는 삽입, 업데이트, 삭제 작업을 자주 수행하는 애플리케이션에 압축 테이블을 사용하여 스토리지 요구 사항과 초당 입출력 작업 수(IOPS)를 줄일 수 있습니다.

이러한 구성 옵션을 사용하면 쓰기 집약적인 작업의 성능과 확장성에 중점을 두고 특정 MySQL 인스턴스에 대한 압축 작동 방식을 조정할 수 있습니다:

- `innodb_compression_level`을 사용하면 압축 정도를 높이거나 낮출 수 있습니다. 값이 높을수록 저장 장치에 더 많은 데이터를 저장할 수 있지만, 처리하는 동안 CPU 오버헤드가 증가합니다.
  압축. 저장 공간이 중요하지 않거나 데이터를 특별히 압축할 필요가 없을 것으로 예상되는 경우 값을 낮게 설정하면 CPU 오버헤드를 줄일 수 있습니다.

- `innodb_compression_failure_threshold_pct`는 압축된 테이블을 업데이트하는 동안 압축 실패에 대한 차단 지점을 지정합니다. 이 임계값을 통과하면 MySQL은 각각의 새 압축 페이지 내에 추가 여유 공간을 남겨두기 시작하여 `innodb_compression_pad_pct_max`에 지정된 페이지 크기 비율까지 여유 공간의 양을 동적으로 조정합니다.

- `innodb_compression_pad_pct_max`를 사용하면 전체 페이지를 다시 압축할 필요 없이 압축된 행에 변경 사항을 기록하기 위해 각 페이지 내에 예약된 최대 공간을 조정할 수 있습니다. 값이 높을수록 재압축 없이 더 많은 변경 사항을 기록할 수 있습니다.
  페이지. MySQL은 각 압축 테이블 내의 페이지에 가변적인 양의 여유 공간을 사용하며, 런타임에 지정된 비율의 압축 작업이 '실패'하여 압축된 페이지를 분할하는 데 많은 비용이 드는 경우에만 사용합니다.

- `innodb_log_compressed_pages`를 사용하면 재압축된 페이지의 이미지가 재실행 로그에 기록되지 않도록 설정할 수 있습니다. 압축된 데이터를 변경할 때 재압축이 발생할 수 있습니다. 이 옵션은 복구 중에 다른 버전의 `zlib` 압축 알고리즘을 사용할 경우 발생할 수 있는 손상을 방지하기 위해 기본적으로 활성화되어 있습니다. `zlib` 버전이 변경되지 않는 것이 확실하다면, 압축 데이터를 수정하는 워크로드에 대한 재실행 로그 생성을 줄이려면 `innodb_log_compressed_pages`를 비활성화하세요.

압축 데이터로 작업할 때는 페이지의 압축 버전과 비압축 버전을 동시에 메모리에 보관해야 하는 경우가 있으므로, OLTP 스타일 워크로드에서 압축을 사용할 때는 `innodb_buffer_pool_size` 구성 옵션의 값을 늘릴 준비를 하시기 바랍니다.

15.9.1.7 SQL 압축 구문 경고 및 오류

이 섹션에서는 테이블별 테이블 공간 및 일반 테이블 공간에서 테이블 압축 기능을 사용할 때 발생할 수 있는 구문 경고 및 오류에 대해 설명합니다.

테이블별 파일 테이블 스페이스에 대한 SQL 압축 구문 경고 및 오류

innodb_strict_mode가 활성화된 경우(기본값), `innodb_file_per_table`이 비활성화되어 있는 경우 `CREATE TABLE` 또는 `ALTER TABLE` 문에서 `ROW_FORMAT=COMPRESSED` 또는 `KEY_BLOCK_SIZE`를 지정하면 다음과 같은 오류가 발생합니다.

```
오류 1031(HY000): 't1'에 대한 테이블 스토리지 엔진에 이 옵션이 없습니다.
```

참고

현재 구성에서 압축 테이블 사용을 허용하지 않는 경우 테이블이 생성되지 않습니다.

innodb_strict_mode가 비활성화되어 있는 경우, CREATE TABLE 또는 ALTER TABLE 문에서 ROW_FORMAT=COMPRESSED 또는 KEY_BLOCK_SIZE를 지정하면 다음과 같은 경고가 출력됩니다.

```
mysql> SHOW WARNINGS;
+---------+------+ ----------------------------------------------------------------+
| 레벨|    코드 | 메시지                                                                |
+---------+------+ ----------------------------------------------------------------+
| 경고 | 1478 | InnoDB: KEY_BLOCK_SIZE에는 테이블당 innodb_file_per_table이 필요합니다. |
| 경고 | 1478 | InnoDB: KEY_BLOCK_SIZE=4 무시.                                        |
| 경고 | 1478 | InnoDB: ROW_FORMAT=COMPRESSED에는 테이블당 innodb_file_per_table이 필요합니다. |
| 경고 | 1478 | InnoDB: ROW_FORMAT=DYNAMIC 가정.                                      |
+---------+------+ ----------------------------------------------------------------+
```

참고

이러한 메시지는 오류가 아닌 경고일 뿐이며, 옵션이 지정되지 않은 것처럼 테이블이 압축되지 않고 생성됩니다.

"비엄격" 동작을 사용하면 소스 데이터베이스에 압축된 테이블이 포함되어 있더라도 압축된 테이블을 지원하지 않는 데이터베이스로 mysqldump 파일을 가져올 수 있습니다. 이 경우 MySQL은 작업을 방지하는 대신 ROW_FORMAT=DYNAMIC으로 테이블을 생성합니다.

덤프 파일을 새 데이터베이스로 가져오고 테이블을 원래 데이터베이스에 있는 그대로 다시 만들려면 서버에 innodb_file_per_table 구성 매개변수에 대한 적절한 설정이 있는지 확인합니다.

KEY_BLOCK_SIZE 속성은 ROW_FORMAT이 COMPRESSED로 지정되거나 생략된 경우에만 허용됩니다. 다른 ROW_FORMAT과 함께 KEY_BLOCK_SIZE를 지정하면 SHOW WARNINGS로 볼 수 있는 경고가 생성됩니다. 그러나 테이블은 압축되지 않으며 지정된 KEY_BLOCK_SIZE는 무시됩니다).

레벨	코드	메시지
경고	1478	InnoDB: ROW_FORMAT=COMPRESSED가 아닌 경우 *KEY_BLOCK_SIZE=n* 무시.

innodb_strict_mode를 활성화하여 실행하는 경우, KEY_BLOCK_SIZE와 COMPRESSED 이외의 ROW_FORMAT을 조합하면 경고가 아닌 오류가 발생하고 테이블이 생성되지 않습니다.

표 15.12, "ROW_FORMAT 및 KEY_BLOCK_SIZE 옵션"에서는 다음과 같은 개요를 제공합니다.

테이블 생성 또는 테이블 변경과 함께 사용되는 행_형식 및 키_블록_크기 옵션.

표 15.12 ROW_FORMAT 및 KEY_BLOCK_SIZE 옵션

옵션	사용 참고 사항	설명
행_형식=리던던트	MySQL 5.0.3 이전에 사용된 스토리지 형식	보다 덜 효율적 ROW_FORMAT=COMPACT; for 이전 버전과의 호환성
행_형식=컴팩트	MySQL 5.0.3 이후의 기본 저장소 형식	768바이트의 긴 열 값의 접두사를

옵션	사용 참고 사항	설명
		클러스터된 색인 페이지와 함께 오버플로 페이지에 저장된 남은 바이트
행_포맷=동적		클러스터된 인덱스 페이지 내에 값이 맞는 경우 값을 저장하고, 그렇지 않은 경우 오버플로 페이지에 대한 20바이트 포인터만 저장합니다(접두사 없음).
행_형식=압축		zlib를 사용하여 테이블과 인덱스를 압축합니다.
KEY_BLOCK_SIZE=n		1, 2, 4, 8 또는 16킬로바이트의 압축 페이지 크기를 지정합니다. 행_형식=압축. 일반 테이블스페이스의 경우 InnoDB 페이지 크기와 동일한 KEY_BLOCK_SIZE 값은 허용되지 않습니다.

표 15.13, "InnoDB Strict 모드가 꺼져 있을 때 테이블 생성/변경 경고 및 오류"에는 CREATE TABLE 또는 ALTER TABLE 문의 특정 구성 매개변수 및 옵션 조합에서 발생하는 오류 조건과 SHOW TABLE STATUS 출력에 옵션이 표시되는 방식이 요약되어 있습니다.

innodb_strict_mode가 OFF인 경우 MySQL은 테이블을 생성하거나 변경하지만 아래와 같이 특정 설정은 무시합니다. MySQL 오류 로그에서 경고 메시지를 확인할 수 있습니다. 언제 innodb_strict_mode가 ON인 경우 이러한 지정된 옵션 조합은 오류를 생성하며 테이블이 생성되거나 변경되지 않습니다. 오류 조건에 대한 전체 설명을 보려면 SHOW ERRORS 문(예제)을 실행합니다:

```
mysql> CREATE TABLE x (id INT PRIMARY KEY, c INT)
-> engine=innodb key_block_size=33333;
오류 1005 (HY000): 'test.x' 테이블을 만들 수 없습니다(오류 번호: 1478)

mysql> SHOW ERRORS;
+-------+------+-----------------------------------------+
| 레벨 | 코드 | 메시지                                  |
+-------+------+-----------------------------------------+
| 오류 | 1478 | InnoDB: 잘못된 키 블록 크기 = 33333.    |
| 오류 | 1005 | 'test.x' 테이블을 만들 수 없음(오류 번호: 1478) |
+-------+------+-----------------------------------------+
```

표 15.13 InnoDB Strict 모드가 꺼져 있을 때의 테이블 생성/변경 경고 및 오류

구문	경고 또는 오류 조건	테이블 상태 표시와 같이 ROW_FORMAT 결과 표시
행_형식=리던던트	없음	중복
행_형식=컴팩트	없음	컴팩트
행_형식=압축 또는 ROW_FORMAT=DYNAMIC 또는 KEY_BLOCK_SIZE가 지정됩니다.	파일 단위의 경우 무시됨 테이블 테이블스페이스는 innodb_file_per_table이 활성화되어 있지 않으면 사용할 수 없습니다. 일반 테이블스페이스는 모든 행 형식을 지원합니다. 섹션 15.6.3.3, "일반 테이블 스페이스"를 참조하십시오.	테이블별 파일 테이블 스페이스의 기본 행 형식, 일반 테이블 스페이스의 지정된 행 형식

구문	경고 또는 오류 조건	테이블 상태 표시와 같이 ROW_FORMAT 결과 표시
잘못된 키 블록 크기(1, 2, 4, 8 또는 16이 아님)가 지정되었습니다.	KEY_BLOCK_SIZE는 무시됩니다.	지정된 행 형식 또는 기본 행 형식
행_형식=압축와 유효한 KEY_BLOCK_SIZE가 지정되어 있습니다.	None; KEY_BLOCK_SIZE 지정이 사용됩니다.	압축
KEY_BLOCK_SIZE는 중복, 컴팩트 또는 동적 행 형식	KEY_BLOCK_SIZE는 무시됩니다.	중복, 컴팩트 또는 다이나믹
ROW_FORMAT이 중복, 콤팩트, 동적 또는 압축 중 하나가 아닙니다.	MySQL 구문 분석기가 인식하면 무시됩니다. 그렇지 않으면 오류가 발생합니다.	기본 행 형식 또는 N/A

innodb_strict_mode가 ON인 경우 MySQL은 유효하지 않은 ROW_FORMAT 또는 KEY_BLOCK_SIZE 매개변수를 거부하고 오류를 발생시킵니다. Strict 모드는 기본적으로 ON입니다. innodb_strict_mode가 OFF인 경우 MySQL은 무시된 유효하지 않은 매개변수에 대해 오류 대신 경고를 발행합니다.

SHOW TABLE STATUS를 사용하면 선택한 KEY_BLOCK_SIZE를 볼 수 없습니다. SHOW CREATE TABLE 문은 테이블을 생성할 때 무시된 경우에도 KEY_BLOCK_SIZE를 표시합니다. 테이블의 실제 압축 페이지 크기는 MySQL에서 표시할 수 없습니다.

일반 테이블 스페이스에 대한 SQL 압축 구문 경고 및 오류

- 테이블 스페이스를 만들 때 일반 테이블 스페이스에 대해 FILE_BLOCK_SIZE가 정의되지 않은 경우 테이블 스페이스에 압축 테이블을 포함할 수 없습니다. 압축 테이블을 추가하려고 하면 다음 예와 같이 오류가 반환됩니다:

```
mysql> CREATE TABLESPACE `ts1` ADD DATAFILE 'ts1.ibd' Engine=InnoDB;

mysql> CREATE TABLE t1 (c1 INT PRIMARY KEY) TABLESPACE ts1 ROW_FORMAT=COMPRESSED
       KEY_BLOCK_SIZE=8;
오류 1478(HY000): InnoDB: 테이블 스페이스 `ts1`에 COMPRESSED 테이블을 포함할 수 없습니다.
```

- 일반 테이블 스페이스에 잘못된 키 블록 크기가 있는 테이블을 추가하려고 하면 다음 예와 같이 오류가 반환됩니다:

```
mysql> CREATE TABLESPACE `ts2` ADD DATAFILE 'ts2.ibd' FILE_BLOCK_SIZE = 8192 엔진=InnoDB;

mysql> CREATE TABLE t2 (c1 INT PRIMARY KEY) TABLESPACE ts2 ROW_FORMAT=COMPRESSED
       KEY_BLOCK_SIZE=4;
오류 1478(HY000): InnoDB: 테이블 스페이스 `ts2`가 블록 크기 8192를 사용하며 실제 페이지 크기
가 4096인 테이블을 포함할 수 없습니다.
```

  일반 테이블스페이스의 경우 테이블의 KEY_BLOCK_SIZE는 테이블스페이스의 FILE_BLOCK_SIZE를 1024로 나눈 값과 같아야 합니다. 예를 들어 테이블스페이스의 FILE_BLOCK_SIZE가 8192인 경우 테이

블의 KEY_BLOCK_SIZE는 8이어야 합니다.

- 압축된 테이블을 저장하도록 구성된 일반 테이블 스페이스에 압축되지 않은 행 형식의 테이블을 추가하려고 하면 다음 예와 같이 오류가 반환됩니다:

```
mysql> CREATE TABLESPACE `ts3` ADD DATAFILE 'ts3.ibd' FILE_BLOCK_SIZE = 8192 엔진=InnoDB;

mysql> CREATE TABLE t3 (c1 INT PRIMARY KEY) TABLESPACE ts3 ROW_FORMAT=COMPACT;
오류 1478(HY000): InnoDB: 테이블 스페이스 `ts3`이 블록 크기 8192를 사용하며 물리적 페이지 크
기 16384의 테이블을 포함할 수 없습니다.
```

일반 테이블스페이스에는 innodb_strict_mode를 적용할 수 없습니다. 일반 테이블스페이스에 대한 테이블스페이스 관리 규칙은 innodb_strict_mode와 무관하게 엄격하게 적용됩니다. 자세한 내용은 섹션 13.1.21, "테이블스페이스 생성 문"을 참조한다.

일반 테이블 스페이스와 함께 압축 테이블을 사용하는 방법에 대한 자세한 내용은 섹션 15.6.3.3, "일반 테이블 스페이스"를 참조하십시오.

15.9.2 InnoDB 페이지 압축

`InnoDB`는 파일 단위 테이블 공간에 있는 테이블에 대해 페이지 수준 압축을 지원합니다. 이 기능을 *투명 페이지 압축이라고* 합니다. 페이지 압축은 `CREATE TABLE` 또는 `ALTER TABLE`로 `COMPRESSION` 속성을 지정하여 활성화할 수 있습니다. 지원되는 압축 알고리즘에는 `Zlib` 및 `LZ4`가 포함됩니다.

지원 플랫폼

페이지 압축을 사용하려면 스파스 파일 및 홀 펀칭 지원이 필요합니다. 페이지 압축은 NTFS를 사용하는 Windows와 커널 수준에서 홀 펀칭을 지원하는 다음과 같은 MySQL 지원 Linux 플랫폼 하위 집합에서 지원됩니다:

- 커널 버전 3.10.0-123 이상을 사용하는 RHEL 7 및 파생 배포판
- OEL 5.10(UEK2) 커널 버전 2.6.39 이상
- OEL 6.5(UEK3) 커널 버전 3.8.13 이상
- OEL 7.0 커널 버전 3.8.13 이상
- SLE11 커널 버전 3.0-x
- SLE12 커널 버전 3.12-x
- OES11 커널 버전 3.0-x
- 우분투 14.0.4 LTS 커널 버전 3.13 이상
- 우분투 12.0.4 LTS 커널 버전 3.2 이상
- Debian 7 커널 버전 3.2 이상

참고

특정 Linux 배포판에서 사용 가능한 모든 파일 시스템이 홀 펀칭을 지원하지 않을 수 있습니다.

페이지 압축 작동 방식

페이지가 기록되면 지정된 압축 알고리즘을 사용하여 압축됩니다. 압축된 데이터는 디스크에 기록되며, 홀 펀칭 메커니즘이 페이지 끝에서 빈 블록을 해제합니다. 압축에 실패하면 데이터가 그대로 기록됩니다.

Linux의 홀 펀치 크기

Linux 시스템에서 파일 시스템 블록 크기는 홀 펀칭에 사용되는 단위 크기입니다. 따라서 페이지 압축은 페이지 데이터를 `InnoDB` 페이지 크기에서 파일 시스템 블록 크기를 뺀 크기보다 작거나 같은 크기로 압축할 수 있는 경우에만 작동합니다. 예를 들어 innodb_page_size=16K이고 파일 시스템 블록 크기가 4K인 경우, 홀 펀칭이 가능하려면 페이지 데이터를 12K 이하로 압축해야 합니다.

Windows의 홀 펀치 크기

Windows 시스템에서 스파스 파일의 기본 인프라는 NTFS 압축을 기반으로 합니다. 홀 펀칭 크기는 NTFS 압축 단위로, NTFS 클러스터 크기의 16배입니다. 클러스터 크기와 해당 압축 단위는 다음 표에 나와 있습니다:

표 15.14 Windows NTFS 클러스터 크기 및 압축 단위

클러스터 크기	압축 장치
512 바이트	8 KB
1 KB	16 KB
2 KB	32 KB
4 KB	64 KB

Windows 시스템에서 페이지 압축은 페이지 데이터를 `InnoDB` 페이지 크기에서 압축 단위 크기를 뺀 크기 이하로 압축할 수 있는 경우에만 작동합니다.

기본 NTFS 클러스터 크기는 4KB이며, 압축 단위 크기는 64KB입니다. 즉, 최대 innodb_page_size도 64KB이므로 기본 제공 Windows NTFS 구성에서는 페이지 압축의 이점이 없습니다.

Windows에서 페이지 압축이 작동하려면 파일 시스템을 4K보다 작은 클러스터 크기로 생성해야 하며, innodb_page_size는 압축 단위 크기의 두 배 이상이어야 합니다. 예를 들어, Windows에서 페이지 압축이 작동하려면 클러스터 크기가 512바이트(압축 단위는 8KB)인 파일 시스템을 구축하고 `innodb_page_size` 값을 16K 이상으로 `InnoDB`를 초기화하면 됩니다.

페이지 압축 사용

페이지 압축을 사용하도록 설정하려면 `CREATE TABLE` 문에 `COMPRESSION` 속성을 지정합니다. 예를 들면 다음과 같습니다:

```
CREATE TABLE t1 (c1 INT) COMPRESSION="zlib";
```

`ALTER TABLE` 문에서 페이지 압축을 활성화할 수도 있습니다. 그러나 `ALTER TABLE ... COMPRESSION`은 테이블스페이스 압축 속성만 업데이트합니다. 새 압축 알고리즘을 설정한 후 발생하는 테이블스페이스에 대한 쓰기는 새 설정을 사용하지만 기존 페이지에 새 압축 알고리즘을 적용하려면 `OPTIMIZE TABLE`을 사용하여 테이블을 다시 작성해야 합니다.

```
ALTER TABLE t1 COMPRESSION="zlib";
OPTIMIZE TABLE t1;
```

페이지 압축 비활성화

페이지 압축을 비활성화하려면 `ALTER TABLE`을 사용하여 `COMPRESSION=None`을 설정합니다. `COMPRESSION=None`을 설정한 후 발생하는 테이블 스페이스에 대한 쓰기는 더 이상 페이지 압축을 사용하지 않습니다. 기존 페이지의 압축을 해제하려면 `COMPRESSION=None`을 설정한 후 `OPTIMIZE TABLE`을 사용하여 테이블을 다시 작성해야 합니다.

```
ALTER TABLE t1 COMPRESSION="None";
OPTIMIZE TABLE t1;
```

페이지 압축 메타데이터

페이지 압축 메타데이터는 정보 스키마 `INNODB_TABLESPACES` 테이블의 다음 열에서 찾을 수 있습니다:

- `FS_BLOCK_SIZE`: 홀 펀칭에 사용되는 단위 크기인 파일 시스템 블록 크기입니다.
- 파일 크기: 압축되지 않은 파일의 최대 크기를 나타내는 파일의 겉보기 크기입니다.
- `ALLOCATED_SIZE`: 파일의 실제 크기, 즉 디스크에 할당된 공간의 양입니다.

참고

유닉스 계열 시스템에서 `ls -l` *`tablespace_name`*`.ibd`는 겉보기 파일 크기(`FILE_SIZE`에 해당)를 바이트 단위로 표시합니다. 실제 공간을 확인하려면 다음과 같이 하세요.

> 디스크에 할당된 공간(ALLOCATED_SIZE와 동일)을 확인하려면 du --block-size=1 *tablespace_name*.ibd를 사용합니다. 블록 크기=1 옵션은 할당된 공간을 블록이 아닌 바이트 단위로 출력하므로 ls -l 출력과 비교할 수 있습니다.
>
> 테이블 만들기 표시를 사용하여 현재 페이지 압축 설정(Zlib, Lz4 또는 없음)을 확인합니다. 테이블에는 압축 설정이 다른 여러 페이지가 혼합되어 있을 수 있습니다.

다음 예제에서는 정보 스키마 INNODB_TABLESPACES 테이블에서 employees 테이블에 대한 페이지 압축 메타데이터를 검색합니다.

```
# Zlib 페이지 압축으로 직원 테이블 만들기

CREATE TABLE employees (
    emp_no                      INTNOT NULL,
    birth_date
                                NOTNULL
    ,  first_name  VARCHAR(14)  NOT  NULL,
    last_name    VARCHAR(16)    NOT    NULL,
    gender ENUM ('M','F') NOT NULL,
    hire_date
                                NOTNULL
    , PRIMARY KEY (emp_no)
) COMPRESSION="zlib";

# 데이터 삽입(표시되지 않음)

# mysql> SELECT SPACE, NAME, FS_BLOCK_SIZE, FILE_SIZE, ALLOCATED_SIZE FROM
       INFORMATION_SCHEMA.INNODB_TABLESPACES WHERE NAME='employees/직원'\G
*************************** 1. 행 ***************************
SPACE: 45
이름: 직원/직원
FS_BLOCK_SIZE: 4096
파일 크기: 23068672
할당된 크기: 19415040
```

직원 테이블의 페이지 압축 메타데이터에 따르면 겉보기 파일 크기는 23068672 바이트이고 실제 파일 크기(페이지 압축 포함)는 19415040 바이트입니다. 파일 시스템 블록 크기는 4096바이트이며, 이는 홀 펀칭에 사용되는 블록 크기입니다.

페이지 압축을 사용하여 테이블 식별

페이지 압축이 활성화된 테이블을 식별하려면 정보 스키마를 확인하면 됩니다.

TABLES 테이블의 CREATE_OPTIONS 열에서 COMPRESSION 속성으로 정의된 테이블을 확인할 수 있습니다:

```
mysql> SELECT TABLE_NAME, TABLE_SCHEMA, CREATE_OPTIONS FROM INFORMATION_SCHEMA.TABLES
       WHERE CREATE_OPTIONS LIKE '%COMPRESSION=%';
+------------+-------------+----------------------+
| table_name | table_schema | create_options      |
+------------+-------------+----------------------+
| employees | test|          COMPRESSION="zlib" |
+------------+-------------+----------------------+
```

테이블 만들기 표시를 사용하면 압축 속성도 표시됩니다(사용 중인 경우).

페이지 압축 제한 및 사용 참고 사항

- 파일 시스템 블록 크기(또는 Windows의 경우 압축 단위 크기)*2를 초과하면 페이지 압축이 비활성화됩니다. > `innodb_page_size`.
- 시스템 테이블 공간, 임시 테이블 공간 및 일반 테이블 공간과 같은 공유 테이블 공간에 있는 테이블에는 페이지 압축이 지원되지 않습니다.
- 로그 테이블스페이스 실행 취소에는 페이지 압축이 지원되지 않습니다.
- 재실행 로그 페이지에는 페이지 압축이 지원되지 않습니다.

- 공간 인덱스에 사용되는 R-tree 페이지는 압축되지 않습니다.
- 압축 테이블에 속하는 페이지(`ROW_FORMAT=COMPRESSED`)는 그대로 유지됩니다.
- 복구하는 동안 업데이트된 페이지는 압축되지 않은 형태로 기록됩니다.
- 사용된 압축 알고리즘을 지원하지 않는 서버에서 페이지 압축 테이블스페이스를 로드하면 I/O 오류가 발생합니다.
- 페이지 압축을 지원하지 않는 이전 버전의 MySQL로 다운그레이드하기 전에 페이지 압축 기능을 사용하는 테이블의 압축을 해제합니다. 테이블 압축을 해제하려면 `ALTER TABLE ... COMPRESSION=None`을 실행하고 테이블 최적화를 실행합니다.
- 페이지 압축 테이블스페이스는 사용된 압축 알고리즘을 두 서버 모두에서 사용할 수 있는 경우 Linux와 Windows 서버 간에 복사할 수 있습니다.
- 페이지 압축 테이블스페이스 파일을 한 호스트에서 다른 호스트로 이동할 때 페이지 압축을 유지하려면 스파스 파일을 보존하는 유틸리티가 필요합니다.
- NVMFS는 펀치홀 기능을 활용하도록 설계되었기 때문에 다른 플랫폼보다 NVMFS를 사용하는 Fusion-io 하드웨어에서 더 나은 페이지 압축을 달성할 수 있습니다.
- `InnoDB` 페이지 크기가 크고 파일 시스템 블록 크기가 상대적으로 작은 상태에서 페이지 압축 기능을 사용하면 쓰기 증폭이 발생할 수 있습니다. 예를 들어, 4KB 파일 시스템 블록 크기에 최대 `InnoDB` 페이지 크기가 64KB인 경우 압축은 향상될 수 있지만 버퍼 풀에 대한 수요가 증가하여 I/O가 증가하고 쓰기 증폭이 발생할 수 있습니다.

15.10 InnoDB 행 형식

테이블의 행 형식에 따라 행이 물리적으로 저장되는 방식이 결정되며, 이는 쿼리 및 DML 작업의 성능에 영향을 미칠 수 있습니다. 단일 디스크 페이지에 더 많은 행이 들어갈수록 쿼리 및 인덱스 조회가 더 빠르게 작동하고 버퍼 풀에 필요한 캐시 메모리가 줄어들며 업데이트된 값을 기록하는 데 필요한 I/O가 줄어듭니다.

각 테이블의 데이터는 페이지로 나뉩니다. 각 테이블을 구성하는 페이지는 B-트리 인덱스라는 트리 데이터 구조로 배열됩니다. 테이블 데이터와 보조 인덱스는 모두 이 유형의 구조를 사용합니다. 전체 테이블을 나타내는 B-트리 인덱스를 클러스터된 인덱스라고 하며, 기본 키 열에 따라 구성됩니다. 클러스터된 인덱스 데이터 구조의 노드에는 행에 있는 모든 열의 값이 포함됩니다. 보조 인덱스 구조의 노드에는 인덱스 열과 기본 키 열의 값이 포함됩니다.

가변 길이 열은 열 값이 B-트리 인덱스 노드에 저장되는 규칙의 예외입니다. B-트리 페이지에 맞추기에는 너무 긴 가변 길이 열은 오버플로 페이지라고 하는 별도로 할당된 디스크 페이지에 저장됩니다. 이러한 열을 오프페이지 열이라고 합니다. 페이지 외부 열의 값은 단일 링크된 오버플로 페이지 목록에 저장되며, 이러한 각

열에는 하나 이상의 자체 오버플로 페이지 목록이 있습니다. 열 길이에 따라 가변 길이 열 값의 전체 또는 접두사가 B-트리에 저장되어 저장 공간을 낭비하거나 별도의 페이지를 읽어야 하는 상황을 방지할 수 있습니다.

`InnoDB` 스토리지 엔진은 네 가지 행 형식을 지원합니다: 중복, 컴팩트, 동적, 그리고 압축.

표 15.15 InnoDB 행 형식 개요

행 형식	컴팩트한 스토리지 특성	향상된 가변 길이 열 스토리지	대형 인덱스 키 접두사 지원	압축 지원	지원되는 테이블스페이스 유형
중복	아니요	아니요	아니요	아니요	시스템, 테이블별 파일, 일반

행 형식	컴팩트한 스토리지 특성	향상된 가변 길이 열 스토리지	대형 인덱스 키 접두사 지원	압축 지원	지원되는 테이블스페이스 유형
컴팩트	예	아니요	아니요	아니요	시스템, 테이블별 파일, 일반
다이나믹	예	예	예	아니요	시스템, 테이블별 파일, 일반
압축	예	예	예	예	테이블별 파일, 일반

다음 항목에서는 행 형식 저장소 특성과 테이블의 행 형식을 정의하고 결정하는 방법에 대해 설명합니다.

- 중복 행 형식
- 컴팩트 행 형식
- 다이나믹 행 형식
- 압축 행 형식
- 테이블의 행 형식 정의하기
- 테이블의 행 형식 결정하기

중복 행 형식

중복 형식은 이전 버전의 MySQL과의 호환성을 제공합니다.

중복 행 형식을 사용하는 테이블은 B 트리 노드 내의 인덱스 레코드에 처음 768바이트의 가변 길이 열 값(`VARCHAR`, `VARBINARY`, `BLOB` 및 `TEXT` 유형)을 저장하고 나머지는 오버플로 페이지에 저장합니다. 768바이트보다 큰 고정 길이 열 은 가변 길이 열로 인코딩되며 페이지 외부에 저장할 수 있습니다. 예를 들어, `CHAR(255)` 열은 `utf8mb4`와 같이 문자 집합의 최대 바이트 길이가 3보다 큰 경우 768바이트를 초과할 수 있습니다.

열 값이 768바이트 이하인 경우 오버플로 페이지가 사용되지 않고 값이 B-트리 노드에 전적으로 저장되므로 I/O가 일부 절감될 수 있습니다. 이 방법은 비교적 짧은 `BLOB` 열 값에 적합하지만, B-트리 노드가 키 값이 아닌 데이터로 채워져 효율성이 저하될 수 있습니다. `BLOB` 열이 많은 테이블은 B-트리 노드가 너무 꽉 차서 너무 적은 행을 포함하게 되어 행이 짧거나 열 값이 페이지 외부에 저장되는 경우보다 전체 인덱스의 효율성이 떨어질 수 있습니다.

중복 행 형식 스토리지 특성

중복 행 형식에는 다음과 같은 저장소 특성이 있습니다:

- 각 인덱스 레코드에는 6바이트 헤더가 포함됩니다. 헤더는 연속된 레코드를 서로 연결하고 행 수준 잠금을 위해 사용됩니다.
- 클러스터된 인덱스의 레코드에는 모든 사용자 정의 열에 대한 필드가 포함됩니다. 또한 6바이트 트랜잭션 ID 필드와 7바이트 롤 포인터 필드도 있습니다.
- 테이블에 기본 키가 정의되어 있지 않은 경우, 클러스터된 각 인덱스 레코드에는 6바이트 행 ID 필드도 포함됩니다.
- 각 보조 인덱스 레코드에는 보조 인덱스에 없는 클러스터된 인덱스 키에 대해 정의된 모든 기본 키 열이 포함되어 있습니다.

- 레코드에는 레코드의 각 필드에 대한 포인터가 포함됩니다. 레코드에 있는 필드의 총 길이가 128바이트 미만인 경우 포인터는 1바이트이고, 그렇지 않은 경우 2바이트입니다. 포인터의 배열을 레코드 디렉토리라고 합니다. 포인터가 가리키는 영역은 레코드의 데이터 부분입니다.
- 내부적으로 CHAR(10)과 같은 고정 길이 문자 열은 고정 길이 형식으로 저장됩니다. VARCHAR 열에서 후행 공백은 잘리지 않습니다.
- 768바이트보다 큰 고정 길이 열은 가변 길이 열로 인코딩되며 페이지 외부에 저장할 수 있습니다. 예를 들어, 문자 집합의 최대 바이트 길이가 3보다 큰 경우 CHAR(255) 열은 utf8mb4와 마찬가지로 768바이트를 초과할 수 있습니다.
- SQL NULL 값은 레코드 디렉터리에 1바이트 또는 2바이트를 예약합니다. 가변 길이 열에 저장된 경우 SQL NULL 값은 레코드의 데이터 부분에 0바이트를 예약합니다. 고정 길이 열의 경우, 열의 고정 길이가 레코드의 데이터 부분에 예약됩니다. NULL 값에 고정 공간을 예약하면 인덱스 페이지 조각화를 일으키지 않고도 열을 NULL에서 NULL이 아닌 값으로 제자리에서 업데이트할 수 있습니다.

컴팩트 행 형식

COMPACT 행 형식은 중복 행 형식에 비해 행 저장 공간을 약 20% 줄여주지만, 일부 작업에서 CPU 사용량이 증가합니다. 워크로드가 다음과 같은 일반적인 워크로드인 경우
캐시 적중률과 디스크 속도에 의해 제한되는 경우 컴팩트 포맷이 더 빠를 수 있습니다. 워크로드가 CPU 속도에 의해 제한되는 경우 컴팩트 포맷이 더 느릴 수 있습니다.

COMPACT 행 형식을 사용하는 테이블은 B-tree 노드 내의 인덱스 레코드에 처음 768바이트의 가변 길이 열 값(VARCHAR, VARBINARY, BLOB 및 TEXT 유형)을 저장하고 나머지는 오버플로 페이지에 저장합니다.
768바이트보다 큰 고정 길이 열은 가변 길이 열로 인코딩되며, 페이지 외부에 저장할 수 있습니다. 예를 들어
, CHAR(255)
열은 문자 집합의 최대 바이트 길이가 3보다 큰 경우 utf8mb4와 마찬가지로 768바이트를 초과할 수 있습니다
.

열 값이 768바이트 이하인 경우 오버플로 페이지가 사용되지 않고 값이 B-트리 노드에 완전히 저장되므로 I/O가 일부 절감될 수 있습니다. 이 방법은 비교적 짧은 BLOB 열 값에 적합하지만, B-트리 노드가 키 값이 아닌 데이터로 채워져 효율성이 저하될 수 있습니다. BLOB 열이 많은 테이블은 B-트리 노드가 너무 꽉 차서 너무 적은 행을 포함하게 되어 행이 짧거나 열 값이 페이지 외부에 저장되는 경우보다 전체 인덱스의 효율성이 떨어질 수 있습니다.

컴팩트 행 형식 스토리지 특성

COMPACT 행 형식에는 다음과 같은 저장소 특성이 있습니다:

- 각 인덱스 레코드에는 5바이트 헤더가 포함되며, 그 앞에 가변 길이 헤더가 올 수 있습니다. 이 헤더는 연속된 레코드를 연결하고 행 수준 잠금을 위해 사용됩니다.

- 레코드 헤더의 가변 길이 부분에는 NULL 열을 나타내는 비트 벡터가 포함되어 있습니다. 인덱스에서 NULL이 될 수 있는 열의 수가 N인 경우 비트 벡터는 CEILING(*N*/8) 바이트를 차지합니다. (예를 들어, NULL이 될 수 있는 열이 9개에서 16개 사이인 경우 비트 벡터는 2바이트를 사용합니다.) NULL인 열은 이 벡터에서 비트 이외의 공간을 차지하지 않습니다. 헤더의 가변 길이 부분에는 가변 길이 열의 길이도 포함됩니다. 각 길이는 열의 최대 길이에 따라 1바이트 또는 2바이트가 소요됩니다. 인덱스의 모든 열이 NULL이 아니고 길이가 고정된 경우 레코드 헤더에 가변 길이 부분이 없습니다.

- NULL이 아닌 각 가변 길이 필드에 대해 레코드 헤더에는 열의 길이가 1바이트 또는 2바이트 단위로 포함됩니다. 열의 일부가 오버플로 페이지에 외부에 저장되어 있거나 최대 길이가 255바이트를 초과하고 실제 길이가 127바이트를 초과하는 경우에만 2바이트가 필요합니다. 외부에 저장된 열의 경우 2바이트 길이는 내부에 저장된 부분의 길이에 외부에 저장된 부분에 대한 20바이트 포인터를 더한 값을 나타냅니다. 내부 부분은 768바이트이므로 길이는 768+20입니다. 20바이트 포인터는 열의 실제 길이를 저장합니다.

- 레코드 헤더 뒤에는 NULL이 아닌 열의 데이터 내용이 이어집니다.
- 클러스터된 인덱스의 레코드에는 모든 사용자 정의 열에 대한 필드가 포함됩니다. 또한 6바이트 트랜잭션 ID 필드와 7바이트 롤 포인터 필드도 있습니다.
- 테이블에 기본 키가 정의되어 있지 않은 경우, 클러스터된 각 인덱스 레코드에는 6바이트 행 ID 필드도 포함됩니다.
- 각 보조 인덱스 레코드에는 보조 인덱스에 없는 클러스터된 인덱스 키에 대해 정의된 모든 기본 키 열이 포함됩니다. 기본 키 열이 가변 길이인 경우, 보조 인덱스가 고정 길이 열에 정의되어 있더라도 각 보조 인덱스의 레코드 헤더에는 해당 길이를 기록할 수 있는 가변 길이 부분이 있습니다.
- 내부적으로 가변 길이가 아닌 문자 집합의 경우 CHAR(10)과 같은 고정 길이 문자 열이 사용됩니다. 는 고정 길이 형식으로 저장됩니다.

  VARCHAR 열에서 후행 공백은 잘리지 않습니다.
- 내부적으로, utf8mb3 및 utf8mb4와 같은 가변 길이 문자 집합의 경우 InnoDB는 후행 공백을 잘라내어 CHAR(*N*)을 N바이트 단위로 저장하려고 시도합니다. CHAR(*N*) 열 값의 바이트 길이가 N바이트를 초과하는 경우 후행 공백은 열 값 바이트 길이의 최소값으로 잘립니다. CHAR(*N*) 열의 최대 길이는 최대 문자 바이트 길이 × *N입니다*.

  CHAR(*N*)에는 최소 N바이트가 예약되어 있습니다. 대부분의 경우 최소 공간 *N을* 예약하면 인덱스 페이지 조각화를 일으키지 않고 열 업데이트를 제자리에서 수행할 수 있습니다. 이에 비해, 중복 행 형식을 사용할 때 CHAR(*N*) 열은 최대 문자 바이트 길이 × *N을* 차지합니다.

  768바이트보다 큰 고정 길이 열은 가변 길이 필드로 인코딩되며 페이지 외부에 저장할 수 있습니다. 예를 들어, 문자 집합의 최대 바이트 길이가 3보다 큰 경우 문자(255) 열은 768바이트를 초과할 수 있습니다(utf8mb4의 경우와 마찬가지로).

다이나믹 행 형식

동적 행 형식은 COMPACT 행 형식과 동일한 저장 특성을 제공하지만 긴 가변 길이 열에 대한 향상된 저장 기능을 추가하고 큰 인덱스 키 접두사를 지원합니다.

ROW_FORMAT=DYNAMIC으로 테이블을 생성하면 InnoDB는 긴 가변 길이 열 값(VARCHAR, VARBINARY, BLOB 및 TEXT 유형의 경우)을 완전히 페이지 외부에 저장할 수 있으며 클러스터된 인덱스 레코드에는 오버플로 페이지에 대한 20바이트 포인터만 포함되어 있습니다. 768바이트보다 큰 고정 길이 필드는 가변 길이 필드로 인코딩됩니다. 예를 들어, 문자 집합의 최대 바이트 길이가 3보다 크면 CHAR(255) 열이 768바이트를 초과할 수 있습니다(utf8mb4와 마찬가지로).

열을 페이지 외부에 저장할지 여부는 페이지 크기와 행의 전체 크기에 따라 달라집니다. 행이 너무 길면 클러스터된 인덱스 레코드가 B-트리 페이지에 맞을 때까지 가장 긴 열이 페이지 외부 저장을 위해 선택됩니다. 40바이

트 이하인 텍스트 및 블롭 열은 일렬로 저장됩니다.

동적 행 형식은 적합할 경우 인덱스 노드에 전체 행을 저장하는 효율성을 유지하지만(COMPACT 및 중복 형식과 마찬가지로), 동적 행 형식은 채우기 문제를 피합니다.
긴 열의 데이터 바이트 수가 많은 B-트리 노드. 긴 데이터 값의 일부가 페이지 외부에 저장되는 경우 일반적으로 전체 값을 페이지 외부에 저장하는 것이 가장 효율적이라는 아이디어에 기반한 것이 DYNAMIC 행 형식입니다. DYNAMIC 형식을 사용하면 더 짧은 열이 B-트리 노드에 남아있을 가능성이 높으므로 주어진 행에 필요한 오버플로 페이지 수가 최소화됩니다.

동적 행 형식은 최대 3072바이트의 인덱스 키 접두사를 지원합니다.

DYNAMIC 행 형식을 사용하는 테이블은 시스템 테이블 스페이스, 테이블별 파일 테이블 스페이스 및 일반 테이블 스페이스에 저장할 수 있습니다. 시스템 테이블스페이스에 동적 테이블을 저장하려면 innodb_file_per_table을 비활성화하고일반 CREATE TABLE 또는 ALTER TABLE 문을 사용하거나, CREATE TABLE 또는 ALTER TABLE과 함께 TABLESPACE [=] innodb_system table 옵션을 사용합니다.

innodb_file_per_table 변수는 일반 테이블 스페이스에는 적용되지 않으며, 시스템 테이블 스페이스에 동적 테이블을 저장하기 위해 TABLESPACE [=] innodb_system table 옵션을 사용할 때도 적용되지 않습니다.

동적 행 형식 스토리지 특성

동적 행 형식은 COMPACT 행 형식의 변형입니다. 저장소 특성에 대해서는 COMPACT 행 형식 저장소 특성을 참조하십시오.

압축 행 형식

압축 행 형식은 다이나믹과 동일한 스토리지 특성 및 기능을 제공합니다.
행 형식이지만 테이블 및 인덱스 데이터 압축에 대한 지원을 추가합니다.

COMPRESSED 행 형식은 테이블 및 인덱스 데이터를 압축하고 더 작은 페이지 크기를 사용하는 추가 저장 공간 및 성능 고려 사항과 함께 페이지 외부 저장에 대해 DYNAMIC 행 형식과 유사한 내부 세부 정보를 사용합니다. 압축 행 형식의 경우, KEY_BLOCK_SIZE 옵션은 클러스터된 인덱스에 저장되는 열 데이터의 양과 오버플로 페이지에 배치되는 열 데이터의 양을 제어합니다. COMPRESSED 행 형식에 대한 자세한 내용은 섹션 15.9, "InnoDB 테이블 및 페이지 압축"을 참조하십시오.

압축 행 형식은 최대 3072바이트의 인덱스 키 접두사를 지원합니다.

COMPRESSED 행 형식을 사용하는 테이블은 파일별 테이블 공간 또는 일반 테이블 공간에서 만들 수 있습니다. 시스템 테이블 스페이스는 COMPRESSED 행 형식을 지원하지 않습니다. 테이블별 파일 테이블 스페이스에 COMPRESSED 테이블을 저장하려면 innodb_file_per_table 변수를 사용하도록 설정해야 합니다. 일반 테이블 스페이스에는 innodb_file_per_table 변수를 적용할 수 없습니다. 일반 테이블스페이스는 모든 행 형식을 지원하지만 물리적 페이지 크기가 다르기 때문에 압축된 테이블과 압축되지 않은 테이블이 동일한 일반 테이블스페이스에 공존할 수 없다는 점에 주의해야 합니다. 자세한 내용은 섹션 15.6.3.3, "일반 테이블 스페이스"를 참조하십시오.

압축 행 형식 스토리지 특성

압축 행 형식은 COMPACT 행 형식의 변형입니다. 저장소 특성에 대해서는 COMPACT 행 형식 저장소 특성을 참조하십시오.

테이블의 행 형식 정의하기

InnoDB 테이블의 기본 행 형식은 innodb_default_row_format 변수에 의해 정의되며, 기본값은 DYNAMIC입니다. 기본 행 형식은 ROW_FORMAT 테이블 옵션이 명시적으로 정의되지 않았거나 ROW_FORMAT=DEFAULT가 지정된 경우에 사용됩니다.

테이블의 행 형식은 CREATE TABLE 또는 ALTER TABLE 문에서 ROW_FORMAT 테이블 옵션을 사용하여

명시적으로 정의할 수 있습니다. 예를 들면 다음과 같습니다:

```
CREATE TABLE t1 (c1 INT) ROW_FORMAT=DYNAMIC;
```

명시적으로 정의된 ROW_FORMAT 설정은 기본 행 형식을 재정의합니다. 지정 ROW_FORMAT=DEFAULT는 암시적 기본값을 사용하는 것과 동일합니다.

innodb_default_row_format 변수는 동적으로 설정할 수 있습니다:

```
mysql> SET GLOBAL innodb_default_row_format=DYNAMIC;
```

유효한 innodb_default_row_format 옵션에는 DYNAMIC, COMPACT 및 REDUNDANT가 포함됩니다. 시스템 테이블스페이스에서 사용하도록 지원되지 않는 COMPRESSED 행 형식은 기본값으로 정의할 수 없습니다. 이 옵션은 CREATE TABLE 또는 ALTER TABLE 문에서만 명시적으로 지정할 수 있습니다. innodb_default_row_format 변수를 COMPRESSED로 설정하려고 하면 오류가 반환됩니다:

```
mysql> SET GLOBAL innodb_dault_row_format=COMPRESSED;
ERROR 1231 (42000): 'innodb_default_row_format' 변수를
'COMPRESSED' 값으로 설정할 수 없습니다.
```

새로 생성된 테이블은 ROW_FORMAT 옵션이 명시적으로 지정되지 않았거나 ROW_FORMAT=DEFAULT를 사용하는 경우 innodb_default_row_format 변수에 정의된 행 형식을 사용합니다. 예를 들어, 다음 CREATE TABLE 문은 innodb_default_row_format 변수에 정의된 행 형식을 사용합니다.

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

```
CREATE TABLE t2 (c1 INT) ROW_FORMAT=DEFAULT;
```

ROW_FORMAT 옵션을 명시적으로 지정하지 않거나 ROW_FORMAT=DEFAULT를 사용하는 경우 테이블을 재구축하는 작업은 테이블의 행 형식을 innodb_default_row_format 변수에 정의된 형식으로 자동 변경합니다.

테이블 재구축 작업에는 테이블 재구축이 필요한 경우 ALGORITHM=COPY 또는 ALGORITHM=INPLACE를 사용하는 ALTER TABLE 작업이 포함됩니다. 자세한 내용은 섹션 15.12.1, "온라인 DDL 작업"을 참고한다. OPTIMIZE TABLE도 테이블 재구축 작업입니다.

다음 예제에서는 명시적으로 정의된 행 형식 없이 생성된 테이블의 행 형식을 자동으로 변경하는 테이블 재구성 작업을 보여 줍니다.

```
mysql> SELECT @@innodb_default_row_format;
+-----------------------------+
| @@innodb_default_row_format |
+-----------------------------+
| 동적                        |
+-----------------------------+

mysql> CREATE TABLE t1 (c1 INT);

mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_TABLES WHERE NAME LIKE 'test/t1' \G
*************************** 1. 행 ***************************
     TABLE_ID: 54
         이름: test/t1 플래
         그: 33
       N_COLS: 4
        공간: 35
   ROW_FORMAT: 동적
ZIP_PAGE_SIZE: 0
   SPACE_TYPE: Single

mysql> SET GLOBAL innodb_default_row_format=COMPACT;

mysql> ALTER TABLE t1 ADD COLUMN (c2 INT);

mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_TABLES WHERE NAME LIKE 'test/t1' \G
*************************** 1. 행 ***************************
     TABLE_ID: 55
         이름: test/t1 플래
         그: 1
       N_COLS: 5
        공간: 36
   ROW_FORMAT: Compact
ZIP_PAGE_SIZE: 0
   SPACE_TYPE: Single
```

기존 테이블의 행 형식을 다음에서 변경하기 전에 다음과 같은 잠재적 문제를 고려하십시오.

중복 또는 컴팩트에서 다이나믹으로.

- 중복 및 COMPACT 행 형식은 최대 767바이트의 인덱스 키 접두사 길이를 지원하는 반면, DYNAMIC 및 COMPRESSED 행 형식은 3072바이트의 인덱스 키 접두사 길이를 지원합니다. 복제 환경에서 innodb_default_row_format 변수가 원본에서 DYNAMIC으로 설정되어 있고 복제본에서 COMPACT로 설정되어 있는 경우, 행 형식을 명시적으로 정의하지 않는 다음 DDL 문은 원본에서 성공하지만 복제본에서는 실패합니다:

```
CREATE TABLE t1 (c1 INT PRIMARY KEY, c2 VARCHAR(5000), KEY i1(c2(3070));
```

관련 정보는 섹션 15.22, "InnoDB 제한"을 참조하세요.

- 행 형식을 명시적으로 정의하지 않은 테이블을 가져오면 소스 서버의 `innodb_default_row_format` 설정이 대상 서버의 설정과 다를 경우 스키마 불일치 오류가 발생합니다. 자세한 내용은 15.6.1.3절. "InnoDB 테이블 가져오기"를 참조하세요.

테이블의 행 형식 결정하기

테이블의 행 형식을 확인하려면 테이블 상태 표시를 사용합니다:

```
mysql> SHOW TABLE STATUS IN test1\G
*************************** 1. 행 *************************** 이
           름: t1
         엔진: InnoDB 버전:
        10
     행_형식: 동적 행: 0
 평균_행_길이: 0
    데이터_길이: 16384
최대_데이터_길이: 0
     Index_길이: 16384
      Data_free: 0
     자동 증가: 1
    생성_시간: 2016-09-14 16:29:38 업데이트_
    시간: NULL
     Check_time: NULL
      콜레이션: utf8mb4_0900_ai_ci 체크섬:
       NULL
     Create_옵션:
            댓글:
```

또는 정보 스키마 `INNODB_TABLES` 테이블을 쿼리합니다:

```
mysql> SELECT NAME, ROW_FORMAT FROM INFORMATION_SCHEMA.INNODB_TABLES WHERE NAME='test1/t1';
+----------+-------------+
| 이름 |     행_형식 |
+----------+-------------+
| test1/t1 | 동적        |
+----------+-------------+
```

15.11 InnoDB 디스크 I/O 및 파일 공간 관리

DBA는 I/O 하위 시스템이 포화 상태가 되지 않도록 디스크 I/O를 관리하고, 저장 장치가 가득 차지 않도록 디스크 공간을 관리해야 합니다. ACID 설계 모델에서는 중복으로 보일 수 있지만 데이터 안정성을 보장하는 데 도움이 되는 일정량의 I/O가 필요합니다. 이러한 제약 조건 내에서 InnoDB는 데이터베이스 작업과 디스크 파일 구성을 최적화하여 디스크 I/O 양을 최소화하려고 노력합니다. 때로는 데이터베이스가 사용 중이 아닐 때까지 또는 빠른 종료 후 데이터베이스를 재시작하는 경우와 같이 모든 것이 일관된 상태가 될 때까지 I/O가 연기되기

이 섹션에서는 기본 종류의 MySQL 테이블(`InnoDB` 테이블이라고도 함)의 I/O 및 디스크 공간에 대한 주요 고려 사항에 대해 설명합니다:

- 쿼리 성능을 개선하기 위해 사용되는 백그라운드 I/O의 양을 제어합니다.
- 추가 I/O를 희생하여 내구성을 강화하는 기능을 활성화 또는 비활성화합니다.
- 표를 여러 개의 작은 파일, 몇 개의 큰 파일 또는 두 가지를 결합한 파일로 구성할 수 있습니다.
- 재실행 로그 파일의 크기와 로그 파일이 가득 찰 때 발생하는 I/O 활동의 균형을 맞출 수 있습니다.
- 최적의 쿼리 성능을 위해 테이블을 재구성하는 방법.

15.11.1 InnoDB 디스크 I/O

InnoDB는 가능한 경우 비동기 디스크 I/O를 사용하여 I/O 작업을 처리하는 스레드를 여러 개 생성하고, I/O가 진행되는 동안 다른 데이터베이스 작업을 계속 진행할 수 있도록 허용합니다. Linux 및 Windows 플랫폼에서 InnoDB는 사용 가능한 OS 및 라이브러리 함수를 사용하여 "기본" 비동기 I/O를 수행합니다. 다른 플랫폼에서도 InnoDB는 여전히 I/O 스레드를 사용하지만, 스레드는 실제로 I/O 요청이 완료될 때까지 대기할 수 있으며, 이 기술을 "시뮬레이션된" 비동기 I/O라고 합니다.

미리 읽기

InnoDB는 데이터가 곧 필요할 가능성이 높다고 판단되면 읽기 전 작업을 수행하여 해당 데이터를 버퍼 풀로 가져와 메모리에서 사용할 수 있도록 합니다. 연속된 데이터에 대해 몇 번의 큰 읽기 요청을 하는 것이 분산된 작은 요청을 여러 번 하는 것보다 더 효율적일 수 있습니다. InnoDB에는 두 가지 읽기 전 휴리스틱이 있습니다:

- 순차적 미리 읽기에서 InnoDB는 테이블 스페이스의 세그먼트에 대한 액세스 패턴이 순차적임을 감지하면 데이터베이스 페이지의 읽기 일괄 처리를 I/O 시스템에 미리 게시합니다.
- 무작위 읽기에서 InnoDB는 테이블 스페이스의 일부 영역이 버퍼 풀로 완전히 읽혀지는 중임을 감지하면 나머지 읽은 내용을 I/O 시스템에 게시합니다.

읽기 전 휴리스틱 구성에 대한 자세한 내용은 15.8.3.4절, "InnoDB 버퍼 풀 프리페칭 구성(읽기 전)"을 참조하세요.

이중 쓰기 버퍼

InnoDB는 이중 쓰기 버퍼라는 구조와 관련된 새로운 파일 플러시 기술을 사용하며, 대부분의 경우 기본적으로 활성화되어 있습니다(innodb_doublewrite=ON). 이 기술은 예기치 않은 종료 또는 정전 후 복구에 안전성을 더하고, fsync() 작업의 필요성을 줄여 대부분의 Unix에서 성능을 개선합니다.

데이터 파일에 페이지를 쓰기 전에 InnoDB는 먼저 이중 쓰기 버퍼라는 저장 영역에 페이지를 씁니다. 쓰기와 이중 쓰기 버퍼에 대한 플러시가 완료된 후에야 InnoDB는 데이터 파일의 적절한 위치에 페이지를 씁니다. 페이지 쓰기 도중에 운영 체제, 스토리지 하위 시스템 또는 예기치 않은 mysqld 프로세스 종료가 발생하여 페이지가 찢어진 상태가 되는 경우, 나중에 복구 중에 InnoDB가 이중 쓰기 버퍼에서 페이지의 올바른 사본을 찾을 수 있습니다.

이중 쓰기 버퍼에 대한 자세한 내용은 섹션 15.6.4, "이중 쓰기 버퍼"를 참조하세요.

15.11.2 파일 공간 관리

innodb_data_file_path 구성 옵션을 사용하여 구성 파일에서 정의한 데이터 파일은 InnoDB 시스템 테이블스페이스를 형성합니다. 파일은 논리적으로 연결되어 시스템 테이블스페이스를 형성합니다. 스트라이

핑은 사용되지 않습니다. 시스템 테이블 스페이스 내에서 테이블이 할당되는 위치를 정의할 수 없습니다. 새로 생성된 시스템 테이블 스페이스에서 `InnoDB`는 첫 번째 데이터 파일부터 공간을 할당합니다.

모든 테이블과 인덱스를 시스템 테이블 스페이스 내에 저장할 때 발생하는 문제를 방지하려면 새로 생성된 각 테이블을 별도의 테이블 스페이스 파일(확장자 `.ibd`)에 저장하는 `innodb_file_per_table` 구성 옵션(기본값)을 활성화할 수 있습니다. 이렇게 저장된 테이블의 경우 디스크 파일 내에서 조각화가 적고, 테이블이 잘릴 때 해당 공간이 시스템 테이블 스페이스 내에서 InnoDB에 의해 계속 예약되지 않고 운영 체제로 반환됩니다. 자세한 내용은 섹션 15.6.3.2, "테이블별 파일 테이블 스페이스"를 참조하십시오.

일반 테이블 스페이스에 테이블을 저장할 수도 있습니다. 일반 테이블 스페이스는 `CREATE TABLESPACE` 구문을 사용하여 만든 공유 테이블 스페이스입니다. MySQL 데이터 디렉토리 외부에서 생성할 수 있으며 여러 테이블을 보유할 수 있고 모든 행 형식의 테이블을 지원합니다. 자세한 내용은 섹션 15.6.3.3, "일반 테이블 스페이스"를 참조하세요.

페이지, 범위, 세그먼트 및 테이블 공간

각 테이블스페이스는 데이터베이스 페이지로 구성됩니다. MySQL 인스턴스의 모든 테이블스페이스는 동일한 페이지 크기를 갖습니다. 기본적으로 모든 테이블스페이스의 페이지 크기는 16KB이며, MySQL 인스턴스를 생성할 때 `innodb_page_size` 옵션을 지정하여 페이지 크기를 8KB 또는 4KB로 줄일 수 있습니다. 페이지 크기를 32KB 또는 64KB로 늘릴 수도 있습니다. 자세한 내용은 `innodb_page_size` 설명서를 참조하세요.

페이지 크기는 최대 16KB 크기의 페이지(연속된 16KB 페이지 64개, 8KB 페이지 128개, 4KB 페이지 256개)의 경우 1MB 크기의 익스텐션으로 그룹화됩니다. 페이지 크기가 32KB인 경우 범위 크기는 2MB입니다. 페이지 크기가 64KB인 경우 범위 크기는 4MB입니다. 테이블 스페이스 내부의 "파일"을 `InnoDB`에서는 세그먼트라고 합니다. (이러한 세그먼트는 실제로 많은 테이블 스페이스 세그먼트를 포함하는 롤백 세그먼트와는 다릅니다.)

테이블 스페이스 내에서 세그먼트가 커지면 `InnoDB`는 처음 32페이지를 한 번에 하나씩 세그먼트에 할당합니다. 그 후 `InnoDB`는 세그먼트에 전체 확장을 할당하기 시작합니다. `InnoDB`는 데이터의 우수한 순차성을 보장하기 위해 큰 세그먼트에 한 번에 최대 4개의 익스텐트를 추가할 수 있습니다.

`InnoDB`의 각 인덱스에는 두 개의 세그먼트가 할당됩니다. 하나는 B-트리의 비리프 노드용이고, 다른 하나는 리프 노드용입니다. 리프 노드가 실제 테이블 데이터를 포함하므로 디스크에서 리프 노드를 연속적으로 유지하면 순차적 I/O 작업을 더 잘 수행할 수 있습니다.

테이블 스페이스의 일부 페이지에는 다른 페이지의 비트맵이 포함되어 있으므로 `InnoDB` 테이블스페이스는 세그먼트 전체에 할당할 수 없으며 개별 페이지로만 할당할 수 있습니다.

테이블 공간에서 사용 가능한 여유 공간을 `SHOW TABLE STATUS` 문을 실행하여 요청하면 `InnoDB`는 테이블 공간에 확실히 여유가 있는 확장을 보고합니다. `InnoDB`는 항상 정리 및 기타 내부 목적을 위해 일부 확장을 예약하며, 이러한 예약된 확장은 여유 공간에 포함되지 않습니다.

테이블에서 데이터를 삭제하면 `InnoDB`는 해당 B-트리 인덱스를 축소합니다. 해제된 공간을 다른 사용자가 사용할 수 있는지 여부는 삭제 패턴에 따라 달라집니다.
개별 페이지 또는 테이블 스페이스의 확장을 삭제할 수 있습니다. 테이블을 삭제하거나 테이블에서 모든 행을 삭제하면 다른 사용자에게 공간이 제공되지만 삭제된 행은 물리적으로만 제거된다는 점을 기억하세요.
는 트랜잭션 롤백 또는 일관된 읽기에 더 이상 필요하지 않은 후 일정 시간이 지나면 자동으로 수행되는 퍼지 작업에 의해 제거됩니다. (섹션 15.3, "InnoDB 다중 버전 관리" 참조).

예약된 파일 세그먼트 페이지의 비율 구성하기

`innodb_segment_reserve_factor` 변수를 사용하면 빈 페이지로 예약된 테이블 스페이스 파일 세그먼트 페이지의 비율을 정의할 수 있습니다. 향후 성장을 위해 일정 비율의 페이지가 예약되어 B-트리의 페이지가 연속적으로 할당될 수 있습니다. 예약된 페이지의 비율을 수정하는 기능을 사용하면 데이터 조각화 또는 스토리지 공간의 비효율적인 사용 문제를 해결하기 위해 `InnoDB`를 미세 조정할 수 있습니다.

이 설정은 테이블별 파일 및 일반 테이블 스페이스에 적용할 수 있습니다. 테이블별

기본 설정은 12.5%입니다.

`innodb_segment_reserve_factor` 변수는 동적이며 `SET`을 사용하여 구성할 수 있습니다.
문을 사용합니다. 예를 들어

```
mysql> SET GLOBAL innodb_segment_reserve_factor=10;
```

페이지가 테이블 행과 관련되는 방식

4KB, 8KB, 16KB 및 32KB `innodb_page_size` 설정의 경우 최대 행 길이는 데이터베이스 페이지 크기의 절반에 약간 못 미칩니다. 예를 들어, 기본 16KB `InnoDB` 페이지 크기에서 최대 행 길이는 8KB보다 약간 작습니다. 64KB `innodb_page_size` 설정의 경우 최대 행 길이는 16KB보다 약간 작습니다.

행이 최대 행 길이를 초과하지 않는 경우 모든 행이 페이지 내에 로컬로 저장됩니다. 행이 최대 행 길이를 초과하는 경우 행이 최대 행 길이 제한에 맞을 때까지 가변 길이 열이 외부 페이지 외부 저장소로 선택됩니다. 가변 길이 열을 위한 외부 페이지 외부 저장소는 행 형식에 따라 다릅니다:

- *컴팩트 및 중복 행 형식*

  가변 길이 열을 외부 페이지 외부 저장소로 선택하면 InnoDB는 처음 768바이트는 행에 로컬로 저장하고 나머지는 오버플로 페이지에 외부에 저장합니다. 이러한 각 열에는 자체 오버플로 페이지 목록이 있습니다. 768바이트 접두사에는 열의 실제 길이를 저장하고 나머지 값이 저장되는 오버플로 목록을 가리키는 20바이트 값이 함께 제공됩니다. 섹션 15.10, "InnoDB 행 형식"을 참조하십시오.

- *동적 및 압축 행 형식*

  가변 길이 열을 외부 페이지 외부 저장소로 선택한 경우 InnoDB는 20바이트 포인터를 행에 로컬로 저장하고 나머지는 오버플로 페이지에 외부에 저장합니다. 섹션 15.10, "InnoDB 행 형식"을 참조하십시오.

LONGBLOB 및 LONGTEXT 열은 4GB 미만이어야 하며, BLOB을 포함한 총 행 길이는 4GB 미만이어야 합니다. 및 TEXT 열의 용량은 4GB 미만이어야 합니다.

15.11.3 InnoDB 체크포인트

로그 파일을 매우 크게 만들면 체크포인트하는 동안 디스크 I/O가 줄어들 수 있습니다. 로그 파일의 총 크기를 버퍼 풀만큼 크게 설정하거나 그 이상으로 설정하는 것이 좋습니다.

체크포인트 처리 작동 방식

InnoDB는 퍼지 체크포인팅으로 알려진 체크포인트 메커니즘을 구현합니다. InnoDB는 수정된 데이터베이스 페이지를 버퍼 풀에서 소량씩 플러시합니다. 버퍼 풀을 한 번에 플러시할 필요가 없으므로 체크포인트 프로세스 중에 사용자 SQL 문 처리가 중단될 수 있습니다.

충돌 복구 중에 InnoDB는 로그 파일에 기록된 체크포인트 레이블을 찾습니다. 이 레이블 이전의 데이터베이스에 대한 모든 수정 사항이 데이터베이스의 디스크 이미지에 존재한다는 것을 알고 있습니다. 그런 다음 InnoDB는 체크포인트에서 앞으로 로그 파일을 스캔하여 기록된 수정 사항을 데이터베이스에 적용합니다.

15.11.4 테이블 조각 모음

보조 인덱스에 무작위로 삽입하거나 삭제하면 인덱스가 조각화될 수 있습니다. 조각화는 디스크에 있는 인덱스 페이지의 물리적 순서가 페이지에 있는 레코드의 인덱스 순서와 일치하지 않거나 인덱스에 할당된 64페이지 블록에 사용되지 않는 페이지가 많다는 것을 의미합니다.

파편화의 한 가지 증상은 테이블이 '사용해야 하는' 공간보다 더 많은 공간을 차지한다는 것입니다. 정확히 얼마나 많은 공간을 차지하는지는 파악하기 어렵습니다. 모든 InnoDB 데이터와 인덱스는 B-트리에 저장되며, 그

채우기 계수는 50%에서 100%까지 다양할 수 있습니다. 조각화의 또 다른 증상은 이와 같은 테이블 스캔이 "해야 하는" 시간보다 더 많은 시간이 걸린다는 것입니다:

```
SELECT COUNT(*) FROM t WHERE non_indexed_column <> 12345;
```

앞의 쿼리에서는 MySQL이 큰 테이블에 대해 가장 느린 쿼리 유형인 전체 테이블 스캔을 수행해야 합니다.

인덱스 스캔 속도를 높이려면 주기적으로 "null" `ALTER TABLE` 작업을 수행하여 MySQL이 테이블을 다시 작성하도록 할 수 있습니다:

```
ALTER TABLE tbl_name ENGINE=INNODB
```

`ALTER TABLE tbl_name` FORCE를 사용하여 테이블을 다시 작성하는 "null" 변경 작업을 수행할 수도 있습니다.

`ALTER TABLE tbl_name ENGINE=INNODB` 와 `ALTER TABLE tbl_name FORCE` 는 모두 온라인 DDL 을 사용한다. 자세한 내용은 섹션 15.12, "InnoDB 및 온라인 DDL"을 참고한다.

조각 모음 작업을 수행하는 또 다른 방법은 mysqldump를 사용하여 테이블을 텍스트 파일에 덤프하고 테이블을 삭제한 다음 덤프 파일에서 다시 로드하는 것입니다.

인덱스에 삽입이 항상 오름차순으로 이루어지고 레코드가 끝 부분부터 삭제되는 경우 `InnoDB` 파일 공간 관리 알고리즘은 인덱스의 조각화가 발생하지 않도록 보장합니다.

15.11.5 테이블 잘라내기로 디스크 공간 확보하기

`InnoDB` 테이블을 잘라낼 때 운영 체제 디스크 공간을 확보하려면 테이블을 자체 .ibd 파일에 저장해야 합니다. 테이블을 자체 .ibd 파일에 저장하려면 테이블을 생성할 때 `innodb_file_per_table`을 활성화해야 합니다. 또한 잘리는 테이블과 다른 테이블 사이에 외래 키 제약 조건이 있을 수 없으며, 그렇지 않으면 `TRUNCATE TABLE` 작업이 실패합니다. 그러나 동일한 테이블에 있는 두 열 사이의 외래 키 제약 조건은 허용됩니다.

테이블이 잘리면 테이블이 삭제되고 새 `.ibd` 파일로 다시 생성되며, 확보된 공간은 운영 체제로 반환됩니다. 이는 `InnoDB` 시스템 테이블 스페이스 내에 저장된 `InnoDB` 테이블(`innodb_file_per_table=OFF`일 때 생성된 테이블) 및 공유 일반 테이블 스페이스에 저장된 테이블을 잘라내는 것과는 대조적으로, 테이블이 잘린 후 해제된 공간은 `InnoDB`만 사용할 수 있습니다.

테이블을 잘라내어 디스크 공간을 운영 체제로 반환하는 기능은 물리적 백업을 더 작게 만들 수 있다는 의미이기도 합니다. 시스템 테이블 스페이스(`innodb_file_per_table=OFF`일 때 생성된 테이블) 또는 일반 테이블 스페이스에 저장된 테이블을 잘라내면 테이블 스페이스에 사용되지 않는 공간 블록이 남습니다.

15.12 InnoDB 및 온라인 DDL

온라인 DDL 기능은 즉시 및 제자리 테이블 변경과 동시 DML을 지원합니다. 이 기능의 장점은 다음과 같습니다:

- 테이블을 몇 분 또는 몇 시간 동안 사용할 수 없게 만드는 것이 현실적으로 불가능한 바쁜 프로덕션 환경에서 응답성과 가용성이 향상됩니다.
- 인플레이스 작업의 경우, `LOCK` 절을 사용하여 DDL 작업 중 성능과 동시성 간의 균형을 조정할 수 있습니다. LOCK 절을 참고한다.
- 테이블 복사 방식보다 디스크 공간 사용량과 I/O 오버헤드가 적습니다.

일반적으로 온라인 DDL을 활성화하기 위해 특별한 조치를 취할 필요는 없습니다. 기본적으로 MySQL은 가능

한 한 잠금을 최소화하면서 허용되는 대로 즉시 또는 제자리에서 작업을 수행합니다.

`ALTER TABLE` 문의 `ALGORITHM` 및 `LOCK` 절을 사용하여 DDL 작업의 여러 측면을 제어할 수 있습니다. 이러한 절은 테이블 및 열 사양과 쉼표로 구분하여 문 끝에 배치됩니다. 예를 들어

```
ALTER TABLE tbl_name ADD PRIMARY KEY (column), ALGORITHM=INPLACE, LOCK=NONE;
```

`LOCK` 절은 제자리에서 수행되는 작업에 사용할 수 있으며 작업 중 테이블에 대한 동시 액세스 수준을 미세 조정하는 데 유용합니다. 즉시 수행되는 연산에는 `LOCK=DEFAULT`만 지원됩니다. `ALGORITHM` 절은 주로 성능 비교를 위한 것이며, 문제가 발생할 경우 이전 테이블 복사 동작에 대한 대체 수단으로 사용됩니다. 예를 들어

- 실수로 테이블을 읽기, 쓰기 또는 둘 다 사용할 수 없게 만드는 것을 방지하려면 제자리에서 `ALTER TABLE` 작업을 수행하는 동안 `ALTER TABLE` 문에 `LOCK=NONE`(읽기 및 쓰기 허용) 또는 `LOCK=SHARED`(읽기 허용)와 같은 절을 지정합니다. 요청된 동시성 수준을 사용할 수 없는 경우 작업이 즉시 중단됩니다.
- 알고리즘 간의 성능을 비교하려면 `ALGORITHM=INSTANT`, `ALGORITHM=INPLACE` 및 `ALGORITHM=COPY`를 사용하여 문을 실행합니다. 또한 `old_alter_table` 구성 옵션을 활성화하여 `ALGORITHM=COPY`를 강제로 사용하도록 설정한 상태에서 문을 실행할 수도 있습니다.
- 테이블을 복사하는 `ALTER TABLE` 연산으로 서버를 묶어두지 않으려면 `ALGORITHM=INSTANT` 또는 `ALGORITHM=INPLACE`를 포함합니다. 지정된 알고리즘을 사용할 수 없는 경우 문이 즉시 중지됩니다.

15.12.1 온라인 DDL 운영

이 섹션의 다음 항목에서 DDL 작업에 대한 온라인 지원 세부 정보, 구문 예제 및 사용 참고 사항을 확인할 수 있습니다.

- 인덱스 작업
- 기본 키 연산
- 열 작업
- 생성된 열 작업
- 외래 키 연산
- 테이블 작업
- 테이블 스페이스 작업
- 파티셔닝 작업

색인 작업

다음 표는 인덱스 작업에 대한 온라인 DDL 지원에 대한 개요를 제공합니다. 별표는 추가 정보, 예외 또는 종속성을 나타냅니다. 자세한 내용은 구문 및 사용 참고 사항을 참조하세요.

표 15.16 인덱스 작업에 대한 온라인 DDL 지원

운영	즉시	제자리에서	테이블 다시 빌드	동시 DML 허용	메타데이터만 수정

보조 인덱스 생성 또는 추가	아니요	예	아니요	예	아니요
인덱스 삭제하기	아니요	예	아니요	예	예
인덱스 이름 바꾸기	아니요	예	아니요	예	예
추가 전체 텍스트 index	아니요	예*	아니요*	아니요	아니요
추가 공간 인덱스	아니요	예	아니요	아니요	아니요

운영	즉시	제자리에서	테이블 다시 빌드	동시 DML 허용	메타데이터만 수정
인덱스 유형 변경	예	예	아니요	예	예

구문 및 사용 참고 사항

- 보조 인덱스 생성 또는 추가

```
테이블 (col_list)에 인덱스 이름 생성;
ALTER TABLE tbl_name ADD INDEX name (col_list);
```

인덱스가 생성되는 동안 테이블은 읽기 및 쓰기 작업에 계속 사용할 수 있습니다. 테이블에 액세스하는 모든 트랜잭션이 완료된 후에만 `CREATE INDEX` 문이 완료되므로 인덱스의 초기 상태는 테이블의 가장 최근 내용을 반영합니다.

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 전체 프로세스의 속도를 높일 수 있음을 의미합니다.

새로 생성된 보조 인덱스에는 `CREATE INDEX` 또는 `ALTER TABLE` 문 실행이 완료된 시점의 테이블에 커밋된 데이터만 포함됩니다. 여기에는
커밋되지 않은 값, 이전 버전의 값 또는 삭제하도록 표시되었지만 아직 제거되지 않은 값은
이전 인덱스입니다.

이 작업의 성능, 공간 사용량 및 의미에 영향을 미치는 몇 가지 요인이 있습니다. 자세한 내용은 섹션 15.12.8, "온라인 DDL 제한"을 참조하세요.

- 인덱스 삭제하기

```
테이블에 인덱스 이름을 삭제합니다;
ALTER TABLE tbl_name DROP INDEX name;
```

인덱스가 삭제되는 동안 테이블은 읽기 및 쓰기 작업에 계속 사용할 수 있습니다. `DROP INDEX` 문은 테이블에 액세스하는 모든 트랜잭션이 완료된 후에만 완료되므로 인덱스의 초기 상태는 테이블의 가장 최근 내용을 반영합니다.

- 인덱스 이름 바꾸기

```
ALTER TABLE tbl_name RENAME INDEX old_index_name TO new_index_name, ALGORITHM=INPLACE, LOCK=NONE;
```

- 전체 텍스트 인덱스 추가

```
테이블(열)에 전체 텍스트 인덱스 이름 만들기;
```

첫 번째 전체 텍스트 인덱스를 추가하면 사용자 정의 `FTS_DOC_ID` 열이 없는 경우 테이블이 다시 작성됩니다. 테이블을 다시 빌드하지 않고도 추가 `FULLTEXT` 인덱스를 추가할 수 있습니다.

- 공간 인덱스 추가

```
CREATE TABLE geom (g GEOMETRY NOT NULL);
ALTER TABLE geom ADD SPATIAL INDEX(g), ALGORITHM=INPLACE, LOCK=SHARED;
```

- 인덱스 유형 변경({BTREE | HASH} 사용)

```
ALTER TABLE tbl_name DROP INDEX i1, ADD INDEX i1(key_part,...) USING BTREE, ALGORITHM=INSTANT;
```

기본 키 연산

다음 표는 기본 키 작업에 대한 온라인 DDL 지원에 대한 개요를 제공합니다. 별표는 추가 정보, 예외 또는 종속성을 나타냅니다. 구문 및 사용 참고 사항을 참조하세요.

표 15.17 기본 키 작업에 대한 온라인 DDL 지원

운영	즉시	제자리에서	테이블 다시 빌드	동시 DML 허용	메타데이터만 수정
기본 키 추가하기	아니요	예*	예*	예	아니요
기본 키 삭제	아니요	아니요	예	아니요	아니요
기본 키 삭제 및 다른 키 추가	아니요	예	예	예	아니요

구문 및 사용 참고 사항

- 기본 키 추가하기

```
ALTER TABLE tbl_name ADD PRIMARY KEY (column), ALGORITHM=INPLACE, LOCK=NONE;
```

테이블을 제자리에 다시 빌드합니다. 데이터가 크게 재구성되므로 비용이 많이 드는 작업입니다. 열을 NOT NULL로 변환해야 하는 경우 특정 조건에서는 ALGORITHM=INPLACE가 허용되지 않습니다.

클러스터된 인덱스를 재구성하려면 항상 테이블 데이터를 복사해야 합니다. 따라서 테이블을 생성할 때 기본 키를 정의하는 것이 가장 좋으며, 나중에 ALTER TABLE ... ADD PRIMARY KEY를 나중에 실행하는 것보다 테이블을 만들 때 기본 키를 정의하는 것이 좋습니다.

UNIQUE 또는 PRIMARY KEY 인덱스를 생성할 때 MySQL은 몇 가지 추가 작업을 수행해야 합니다. UNIQUE 인덱스의 경우 MySQL은 테이블에 키에 대한 중복 값이 없는지 확인합니다. PRIMARY KEY 인덱스의 경우, MySQL은 PRIMARY KEY 열에 NULL이 포함되어 있지 않은지도 확인합니다.

ALGORITHM=COPY 절을 사용하여 기본 키를 추가하면 MySQL은 연결된 열의 NULL 값을 기본값으로 변환합니다: 숫자의 경우 0, 문자 기반 열 및 BLOB의 경우 빈 문자열, 날짜 시간의 경우 0000-00-00 00:00:00로 변환합니다. 이는 오라클에서 사용하지 말 것을 권장하는 비표준 동작입니다.
ALGORITHM=INPLACE를 사용하여 기본 키 추가하기
는 SQL_MODE 설정에 strict_trans_tables 또는 strict_all_tables 플래그가 포함된 경우에만 허용되며, SQL_MODE 설정이 strict인 경우 ALGORITHM=INPLACE가 허용되지만 요청된 기본 키 열에 NULL 값이 포함된 경우 문이 여전히 실패할 수 있습니다. ALGORITHM=INPLACE 동작은 보다 표준을 준수합니다.

기본 키가 없는 테이블을 생성하는 경우 InnoDB가 기본 키를 선택하는데, 이 키는 NOT NULL 열에 정의된 첫 번째 고유 키이거나 시스템 생성 키일 수 있습니다. 추가 숨겨진 열에 대한 불확실성과 잠재적인 공간 요구 사항을 피하려면 CREATE TABLE 문의 일부로 PRIMARY KEY 절을 지정합니다.

MySQL은 원본 테이블의 기존 데이터를 원하는 인덱스 구조를 가진 임시 테이블로 복사하여 새로운 클러스터된 인덱스를 만듭니다. 데이터가 임시 테이블에 완전히 복사되면 원래 테이블의 이름이 다른 임시 테이블 이름

으로 변경됩니다. 클러스터된 새 인덱스를 구성하는 임시 테이블의 이름이 원본 테이블의 이름으로 변경되고 원본 테이블은 데이터베이스에서 삭제됩니다.

보조 인덱스에 대한 작업에 적용되는 온라인 성능 향상은 기본 키 인덱스에는 적용되지 않습니다. InnoDB 테이블의 행은 기본 키를 기반으로 구성된 클러스터된 인덱스에 저장되며, 일부 데이터베이스 시스템에서는 이를 "인덱스 구성 테이블"이라고 부릅니다.
테이블 구조는 기본 키와 밀접하게 연결되어 있으므로 기본 키를 재정의하려면 여전히 데이터를 복사해야 합니다.

기본 키에 대한 연산이 `ALGORITHM=INPLACE`를 사용하는 경우 데이터가 여전히 복사되더라도 `ALGORITHM=COPY`를 사용하는 것보다 더 효율적이므로 다음과 같습니다:

  - `ALGORITHM=INPLACE`에는 실행 취소 로깅 또는 관련 재실행 로깅이 필요하지 않습니다. 이러한 작업은 `ALGORITHM=COPY`를 사용하는 DDL 문에 오버헤드를 추가합니다.
  - 보조 인덱스 항목은 미리 정렬되어 있으므로 순서대로 로드할 수 있습니다.
  - 보조 인덱스에 무작위 액세스 삽입이 없기 때문에 변경 버퍼는 사용되지 않습니다.

- 기본 키 삭제

```
ALTER TABLE tbl_name DROP PRIMARY KEY, ALGORITHM=COPY;
```

  `ALGORITHM=COPY`만 동일한 키에 새 키를 추가하지 않고 기본 키를 삭제하는 것을 지원합니다. `ALTER TABLE` 문을 사용합니다.

- 기본 키를 삭제하고 다른 키 추가

```
ALTER TABLE tbl_name DROP PRIMARY KEY, ADD PRIMARY KEY (column), ALGORITHM=INPLACE, LOCK=NONE;
```

  데이터가 상당히 재구성되므로 비용이 많이 드는 작업입니다.

열 작업

다음 표에서는 열 작업에 대한 온라인 DDL 지원에 대한 개요를 제공합니다. 별표는 추가 정보, 예외 또는 종속성을 나타냅니다. 자세한 내용은 구문 및 사용 참고 사항을 참조하세요.

표 15.18 열 작업에 대한 온라인 DDL 지원

운영	즉시	제자리에서	테이블 다시 빌드	동시 DML 허용	메타데이터만 수정
열 추가하기	예*	예	아니요*	예*	예
열 삭제	예*	예	예	예	예
열 이름 바꾸기	예*	예	아니요	예*	예
열 재정렬	아니요	예	예	예	아니요
열 기본값 설정	예	예	아니요	예	예
열 데이터 유형 변경	아니요	아니요	예	아니요	아니요
확장 `VARCHAR` 열 크기	아니요	예	아니요	예	예

열 기본값 삭제하기	예	예	아니요	예	예
자동 증가 값 변경하기	아니요	예	아니요	예	아니요*

운영	즉시	제자리에서	테이블 다시 빌드	동시 DML 허용	메타데이터만 수정
열을 NULL로 만들기	아니요	예	예*	예	아니요
열을 NULL이 아닌 열로 만들기	아니요	예*	예*	예	아니요
ENUM 또는 SET 의 정의 수정하기 열	예	예	아니요	예	예

구문 및 사용 참고 사항

- 열 추가하기

```
ALTER TABLE tbl_name ADD COLUMN column_name 칼럼_정의, 알고리즘=인스턴트;
```

INSTANT는 MySQL 8.2의 기본 알고리즘입니다.

INSTANT 알고리즘이 열을 추가할 때 다음과 같은 제한 사항이 적용됩니다:

- 문은 열 추가를 INSTANT 알고리즘을 지원하지 않는 다른 ALTER TABLE 작업과 결합할 수 없습니다.
- INSTANT 알고리즘은 테이블의 모든 위치에 열을 추가할 수 있습니다.
- ROW_FORMAT=COMPRESSED를 사용하는 테이블, 전체 텍스트 인덱스가 있는 테이블, 데이터 사전 테이블 공간에 있는 테이블 또는 임시 테이블에는 열을 추가할 수 없습니다. 임시 테이블은 ALGORITHM=COPY만 지원합니다.
- MySQL은 INSTANT 알고리즘이 열을 추가할 때 행 크기를 확인하고 추가가 제한을 초과하는 경우 다음과 같은 오류를 발생시킵니다.

  오류 4092(HY000): 이 최대 가능한 행 크기가 최대 허용 가능한 행 크기를 초과하므로 ALGORITHM=INSTANT로 열을 추가할 수 없습니다. ALGORITHM=INPLACE/COPY를 시도하십시오.

- INSTANT 알고리즘으로 열을 추가한 후 테이블의 내부 표현에서 최대 열 수는 1022개를 초과할 수 없습니다. 오류 메시지는 다음과 같습니다:

  오류 4158 (HY000): ALGORITHM=INSTANT를 사용하여 더 이상 *tbl_name*에 열을 추가할 수 없습니다. ALGORITHM=INPLACE/COPY를 시도하십시오.

- INSTANT 알고리즘은 내부 mysql 테이블과 같은 시스템 스키마 테이블에 열을 추가하거나 삭제

할 수 없습니다.

동일한 `ALTER TABLE` 문에 여러 열을 추가할 수 있습니다. 예를 들어

```
ALTER TABLE t1 ADD COLUMN c2 INT, ADD COLUMN c3 INT, ALGORITHM=INSTANT;
```

각 `ALTER TABLE ...` 뒤에 새 행 버전이 생성됩니다. 하나 이상의 열을 추가하거나, 하나 이상의 열을 삭제하거나, 하나 이상의 열을 추가 및 삭제하는 `ALGORITHM=INSTANT` 작업
를 사용할 수 있습니다. `INFORMATION_SCHEMA.INNODB_TABLES.TOTAL_ROW_VERSIONS` 열은 테이블의 행 버전 수를 추적합니다. 이 값은 열이 즉시 추가되거나 삭제될 때마다 증가합니다. 초기 값은 0입니다.

```
mysql> SELECT NAME, TOTAL_ROW_VERSIONS FROM INFORMATION_SCHEMA.INNODB_TABLES
       WHERE NAME LIKE 'test/t1';
```

```
+---------+---------------------+
| 이름|     총_행_버전 |
+---------+---------------------+
| test/t1 |0 |
+---------+---------------------+
```

즉시 추가되거나 삭제된 열이 있는 테이블이 테이블 재구축 `ALTER TABLE` 또는 `OPTIMIZE TABLE` 작업을 통해 재구축되면 `TOTAL_ROW_VERSIONS` 값이 0으로 재설정됩니다. 각 행 버전에는 테이블 메타데이터를 위한 추가 공간이 필요하므로 허용되는 최대 행 버전 수는 64개입니다. 행 버전 제한에 도달하면 `ALGORITHM=INSTANT`를 사용하는 `ADD COLUMN` 및 `DROP COLUMN` 작업이 거부되고 `COPY` 또는 `INPLACE` 알고리즘을 사용하여 테이블을 다시 작성하라는 오류 메시지와 함께 거부됩니다.

`오류 4080 (HY000): 테이블 테스트/T1에 대한 최대 행 버전에 도달했습니다. 더 이상 열을 즉시 추가하거나 삭제할 수 없습니다. 복사/붙여넣기를 사용하세요.`

다음 정보_체계 열은 즉시 추가된 열에 대한 추가 메타데이터를 제공합니다. 자세한 내용은 해당 열의 설명을 참조하십시오. 26.4.9절, "INFORMATION_SCHEMA INNODB_COLUMNS 테이블" 및 26.4.23절, "INFORMATION_SCHEMA INNODB_TABLES 테이블"을 참조하세요.

- `innodb_columns.default_value`
- `innodb_columns.has_default`
- `innodb_tables.instant_cols`

자동 증가 열을 추가할 때 동시 DML은 허용되지 않습니다. 데이터가 크게 재구성되므로 비용이 많이 드는 작업입니다. 최소한 `ALGORITHM=INPLACE, LOCK=SHARED`가 필요합니다.

열을 추가하는 데 ALGORITHM=INPLACE를 사용하면 테이블이 다시 작성됩니다.

- 열 삭제

```
ALTER TABLE tbl_name DROP COLUMN column_name, ALGORITHM=INSTANT;
```

INSTANT는 MySQL 8.2의 기본 알고리즘입니다.

`INSTANT` 알고리즘을 사용하여 열을 삭제하는 경우 다음과 같은 제한 사항이 적용됩니다:

- 열 삭제는 `ALGORITHM=INSTANT`를 지원하지 않는 다른 `ALTER TABLE` 작업과 동일한 문에서 결합할 수 없습니다.
- `ROW_FORMAT=COMPRESSED`를 사용하는 테이블, 전체 텍스트 인덱스가 있는 테이블, 데이터 사전 테이블 스페이스에 있는 테이블 또는 임시 테이블에서는 열을 삭제할 수 없습니다. 임시 테이블은 `ALGORITHM=COPY`만 지원합니다.

예를 들어, 동일한 `ALTER TABLE` 문에 여러 열을 삭제할 수 있습니다:

```
ALTER TABLE t1 DROP COLUMN c4, DROP COLUMN c5, ALGORITHM=INSTANT;
```

`ALGORITHM=INSTANT`를 사용하여 열을 추가하거나 삭제할 때마다 새 행 버전이 생성됩니다. `INFORMATION_SCHEMA.INNODB_TABLES.TOTAL_ROW_VERSIONS` 열은 테이블의 행 버전 수를 추적

합니다. 이 값은 열이 즉시 추가되거나 삭제될 때마다 증가합니다. 초기 값은 0입니다.

```
mysql> SELECT NAME, TOTAL_ROW_VERSIONS FROM INFORMATION_SCHEMA.INNODB_TABLES
        WHERE NAME LIKE 'test/t1';
+---------+--------------------+
| 이름|     총_행_버젼 |
+---------+--------------------+
| test/t1 |0 |
+---------+--------------------+
```

즉시 추가되거나 삭제된 열이 있는 테이블이 테이블 재구축 ALTER TABLE 또는 OPTIMIZE TABLE 작업을 통해 재구축되면 TOTAL_ROW_VERSIONS 값이 0으로 재설정됩니다. 각 행 버전에는 테이블 메타데이터를 위한 추가 공간이 필요하므로 허용되는 최대 행 버전 수는 64개입니다. 행 버전 제한에 도달하면 ALGORITHM=INSTANT를 사용하는 ADD COLUMN 및 DROP COLUMN 작업이 거부되고 COPY 또는 INPLACE 알고리즘을 사용하여 테이블을 다시 작성하라는 오류 메시지와 함께 거부됩니다.

오류 4080 (HY000): 테이블 테스트/t1에 대한 최대 행 버전에 도달했습니다. 더 이상 열을 즉시 추가하거나 삭제할 수 없습니다. 복사/붙여넣기를 사용하세요.

알고리즘=INSTANT가 아닌 다른 알고리즘을 사용하면 데이터가 크게 재구성되므로 비용이 많이 드는 작업이 됩니다.

- 열 이름 바꾸기

```
ALTER TABLE tbl CHANGE old_col_name new_col_name 데이터 유형, 알고리즘=인스턴트, 잠금=NONE;
```

동시 DML을 허용하려면 데이터 유형은 동일하게 유지하고 열 이름만 변경하세요.

동일한 데이터 유형과 [NOT] NULL 속성을 유지하면서 열 이름만 변경하면 항상 온라인에서 작업을 수행할 수 있습니다.

다른 테이블에서 참조된 열의 이름을 바꾸는 것은 ALGORITHM=INPLACE에서만 허용됩니다. ALGORITHM=INSTANT, ALGORITHM=COPY 또는 연산이 이러한 알고리즘을 사용하게 하는 다른 조건을 사용하는 경우 ALTER TABLE 문이 실패합니다.

ALGORITHM=INSTANT는 가상 열의 이름 변경을 지원하지만 ALGORITHM=INPLACE는 지원하지 않습니다.

ALGORITHM=INSTANT 및 ALGORITHM=INPLACE는 동일한 문에서 가상 열을 추가하거나 삭제할 때 열 이름 변경을 지원하지 않습니다. 이 경우 ALGORITHM=COPY만 지원됩니다.

- 열 재정렬

열을 재정렬하려면 변경 또는 수정 작업에서 FIRST 또는 AFTER를 사용합니다.

```
ALTER TABLE tbl_name MODIFY COLUMN col_name column_definition FIRST, ALGORITHM=INPLACE, LOCK=NONE;
```

데이터가 상당히 재구성되므로 비용이 많이 드는 작업입니다.

- 열 데이터 유형 변경

```
ALTER TABLE tbl_name CHANGE c1 c1 BIGINT, ALGORITHM=COPY;
```

열 데이터 유형 변경은 ALGORITHM=COPY에서만 지원됩니다.

- VARCHAR 열 크기 확장

```
ALTER TABLE tbl_name CHANGE COLUMN c1 c1 VARCHAR(255), ALGORITHM=INPLACE, LOCK=NONE;
```

VARCHAR 열에 필요한 길이 바이트 수는 동일하게 유지되어야 합니다. 크기가 0~255바이트인 VARCHAR 열

의 경우 값을 인코딩하는 데 1바이트의 길이 바이트가 필요합니다. 크기가 256바이트 이상인 VARCHAR 열의 경우 두 개의 길이 바이트가 필요합니다. 결과적으로, 제자리 ALTER TABLE은 VARCHAR 열 크기를 0에서 255바이트까지 또는 256바이트에서 더 큰 크기로 늘리는 것만 지원합니다. In-place ALTER TABLE은 VARCHAR 열의 크기를 256바이트 미만에서 256바이트 이상으로 늘리는 것을 지원하지 않습니다. 이 경우 필요한 길이 바이트 수가 1에서 2로 변경되며, 이는 테이블 복사(ALGORITHM=COPY)에서만 지원됩니다.

예를 들어, 제자리에서 ALTER TABLE을 사용하여 단일 바이트 문자 집합의 VARCHAR 열 크기를 VARCHAR(255)에서 VARCHAR(256)으로 변경하려고 하면 이 오류가 반환됩니다:

```
ALTER TABLE tbl_name ALGORITHM=INPLACE, CHANGE COLUMN c1 c1 VARCHAR(256);
오류 0A000: 알고리즘=인플레이스가 지원되지 않습니다. 이유: 변경할 수 없습니다.
```

```
열 유형을 INPLACE로 설정합니다. ALGORITHM=COPY를 사용해 보십시오.
```

참고

VARCHAR 열의 바이트 길이는 문자 집합의 바이트 길이에 따라 달라집니다.

제자리에서 ALTER TABLE을 사용하여 VARCHAR 크기를 줄이는 것은 지원되지 않습니다. VARCHAR 크기를 줄이려면 테이블 복사(ALGORITHM=COPY)가 필요합니다.

- 열 기본값 설정

```
ALTER TABLE tbl_name ALTER COLUMN col SET DEFAULT 리터럴, ALGORITHM=INSTANT;
```

테이블 메타데이터만 수정합니다. 기본 열 값은 데이터 사전에 저장됩니다.

- 열 기본값 삭제하기

```
ALTER TABLE tbl ALTER COLUMN col DROP DEFAULT, ALGORITHM=INSTANT;
```

- 자동 증가 값 변경하기

```
ALTER TABLE 테이블 자동 인크립션 = 다음 값, 알고리즘 = 인플레이스, 잠금 = 없음;
```

데이터 파일이 아닌 메모리에 저장된 값을 수정합니다.

복제 또는 샤딩을 사용하는 분산 시스템에서 테이블의 자동 증가 카운터를 특정 값으로 재설정하는 경우가 있습니다. 그러면 테이블에 삽입되는 다음 행은 자동 증가 열에 지정된 값을 사용합니다. 주기적으로 모든 테이블을 비웠다가 다시 로드하고 자동 증가 시퀀스를 1부터 다시 시작하는 데이터 웨어하우징 환경에서도 이 기술을 사용할 수 있습니다.

- 열을 NULL로 만들기

```
ALTER TABLE tbl_name MODIFY COLUMN column_name 데이터_유형 NULL, ALGORITHM=INPLACE, LOCK=NONE;
```

테이블을 제자리에 다시 빌드합니다. 데이터가 상당히 재구성되므로 비용이 많이 드는 작업입니다.

- 열을 NULL이 아닌 열로 만들기

```
ALTER TABLE tbl_name MODIFY COLUMN column_name data_type NOT NULL, ALGORITHM=INPLACE, LOCK=NONE;
```

테이블을 제자리에 다시 빌드합니다. 작업이 성공하려면 STRICT_ALL_TABLES 또는 STRICT_TRANS_TABLES SQL_MODE가 필요합니다. 열에 NULL 값이 포함되어 있으면 작업이 실패합니다. 서버는 참조 무결성 손실을 초래할 수 있는 외래 키 열에 대한 변경을 금지합니다. 섹션 13.1.9, "테이블 문 변경"을 참조하십시오. 데이터가 크게 재구성되므로 비용이 많이 드는 작업입니다.

- ENUM 또는 SET 열의 정의 수정하기

```
CREATE TABLE t1 (c1 ENUM('a', 'b', 'c'));
ALTER TABLE t1 MODIFY COLUMN c1 ENUM('a', 'b', 'c', 'd'), ALGORITHM=INSTANT;
```

데이터 유형의 저장소 크기가 변경되지 않는 한 유효한 멤버 값 목록 *끝에* 새 열거형 또는 집합 멤버를 추가하여 열 또는 집합 열의 정의를 수정하는 작업은 즉시 또는 제자리에서 수행할 수 있습니다. 예를 들어 멤버가 8개인

`SET` 열에 멤버를 추가하면 값당 필요한 저장 공간이 1바이트에서 2바이트로 변경되므로 테이블 복사본이 필요합니다.

목록 중간에 멤버를 추가하면 기존 멤버의 번호가 변경되므로 테이블 복사본이 필요합니다.

생성된 열 작업

다음 표에서는 생성된 열 작업에 대한 온라인 DDL 지원에 대한 개요를 제공합니다. 자세한 내용은 구문 및 사용 참고 사항을 참조하세요.

표 15.19 생성된 열 작업에 대한 온라인 DDL 지원

운영	즉시	제자리에서	테이블 다시 빌드	동시 DML 허용	메타데이터만 수정
추가 저장된 열	아니요	아니요	예	아니요	아니요
저장된 열 순서 수정하기	아니요	아니요	예	아니요	아니요
드롭 저장된 열	아니요	예	예	예	아니요
추가 가상 열	예	예	아니요	예	예
수정 가상 열 순서	아니요	아니요	예	아니요	아니요
드롭 가상 열	예	예	아니요	예	예

구문 및 사용 참고 사항

- `STORED` 열 추가

```
ALTER TABLE t1 ADD COLUMN (c2 INT 항상 생성된 AS (c1 + 1) 저장됨), ALGORITHM=COPY;
```

서버에서 표현식을 평가해야 하므로 `ADD` COLUMN은 저장된 열에 대한 제자리 연산이 아닙니다(임시 테이블을 사용하지 않고 수행됨).

- 저장된 열 순서 수정하기

```
ALTER TABLE t1 MODIFY COLUMN c2 INT GENERATED ALWAYS AS (c1 + 1) STORED FIRST, ALGORITHM=COPY;
```

테이블을 제자리에 다시 빌드합니다.

- 저장된 열 삭제

```
ALTER TABLE t1 DROP COLUMN c2, ALGORITHM=INPLACE, LOCK=NONE;
```

테이블을 제자리에 다시 빌드합니다.

- 가상 열 추가하기

```
ALTER TABLE t1 ADD COLUMN (c2 INT 항상 (c1 + 1) VIRTUAL로 생성됨), ALGORITHM=INSTANT;
```

가상 열 추가는 즉시 수행하거나 파티션되지 않은 테이블의 경우 제자리에서 수행할 수 있습니다. 파

티션이 분할된 테이블의 경우 가상 열을 추가하는 것은 제자리 작업이 아닙니다.

- 가상 열 순서 수정

```
ALTER TABLE t1 MODIFY COLUMN c2 INT GENERATED ALWAYS AS (c1 + 1) VIRTUAL FIRST, ALGORITHM=COPY;
```

- 가상 열 삭제하기

```
ALTER TABLE t1 DROP COLUMN c2, ALGORITHM=INSTANT;
```

가상 열 삭제는 파티션이 없는 테이블의 경우 즉시 또는 제자리에서 수행할 수 있습니다.

외래 키 연산

다음 표는 외래 키 작업에 대한 온라인 DDL 지원에 대한 개요를 제공합니다. 별표는 추가 정보, 예외 또는 종속성을 나타냅니다. 자세한 내용은 구문 및 사용 참고 사항을 참조하세요.

표 15.20 외래 키 연산을 위한 온라인 DDL 지원

운영	즉시	제자리에서	테이블 다시 빌드	동시 DML 허용	메타데이터만 수정
외래 키 제약 조건 추가	아니요	예*	아니요	예	예
외래 키 제약 조건 삭제	아니요	예	아니요	예	예

구문 및 사용 참고 사항

- 외래 키 제약 조건 추가

  외래 키 검사가 비활성화되어 있으면 `INPLACE` 알고리즘이 지원됩니다. 그렇지 않으면 `COPY` 알고리즘이 지원됩니다.

```
ALTER TABLE tbl1 ADD CONSTRAINT fk_name FOREIGN KEY index (col1)
  REFERENCES tbl2(col2) referential_actions;
```

- 외래 키 제약 조건 삭제

```
ALTER TABLE tbl DROP FORIGN KEY fk_name;
```

  외래 키 삭제는 외래 키 삭제 옵션을 활성화 또는 비활성화한 상태에서 온라인으로 수행할 수 있습니다.

  특정 테이블의 외래 키 제약 조건 이름을 모르는 경우 다음 문을 실행하고 각 외래 키에 대한 `CONSTRAINT` 절에서 제약 조건 이름을 찾습니다:

```
SHOW 테이블 만들기 테이블\G
```

  또는 정보 스키마 `TABLE_CONSTRAINTS` 테이블을 쿼리하고 `CONSTRAINT_NAME` 및 `CONSTRAINT_TYPE` 열을 사용하여 외래 키 이름을 식별합니다.

  또한 외래 키와 관련 인덱스를 단일 문에 넣을 수도 있습니다:

```
ALTER TABLE 테이블 DROP FORIGN KEY 제약 조건, DROP INDEX 인덱스;
```

참고

변경하려는 테이블에 외래 키가 이미 있는 경우(즉, `FOREIGN KEY ... REFERENCE` 절을 포함하는 자식 테이블인 경우) 추가

제한은 온라인 DDL 작업에 적용되며, 직접적으로 관련되지 않은 작업도 포함됩니다.
외래 키 열입니다:

- 부모 테이블에 대한 변경으로 인해 자식 테이블에 관련 변경이 발생하는 경우, 자식 테이블의 `ALTER TABLE`은 다른 트랜잭션이 커밋될 때까지 기다릴 수 있으며, 이 경우 `CASCADE` 또는 `SET NULL` 매개 변수를 사용하여 `ON UPDATE` 또는 `ON DELETE` 절을 통해 관련 변경을 수행합니다.
- 같은 방식으로 테이블이 외래 키 관계의 상위 테이블인 경우 `FOREIGN KEY` 절이 포함되어 있지 않더라도 다음을 기다릴 수 있습니다.

INSERT, UPDATE 또는 DELETE 문으로 인해 자식 테이블에서 ON UPDATE 또는 ON DELETE 작업이 수행되는 경우 ALTER TABLE이 완료되도록 합니다.

테이블 작업

다음 표는 테이블 작업에 대한 온라인 DDL 지원에 대한 개요를 제공합니다. 별표는 추가 정보, 예외 또는 종속성을 나타냅니다. 자세한 내용은 구문 및 사용 참고 사항을 참조하세요.

표 15.21 테이블 작업에 대한 온라인 DDL 지원

운영	즉시	제자리에서	테이블 다시 빌드	동시 DML 허용	메타데이터만 수정
변경 ROW_FORMAT	아니요	예	예	예	아니요
변경 KEY_BLOCK_SIZE	아니요	예	예	예	아니요
영구 테이블 통계 설정	아니요	예	아니요	예	예
문자 집합 지정	아니요	예	예*	예	아니요
문자 집합 변환	아니요	아니요	예*	아니요	아니요
테이블 최적화	아니요	예*	예	예	아니요
FORCE 옵션으로 재구축	아니요	예*	예	예	아니요
널 리빌드 수행	아니요	예*	예	예	아니요
테이블 이름 바꾸기	예	예	아니요	예	예

구문 및 사용 참고 사항

- ROW_FORMAT 변경하기

```
ALTER TABLE tbl_name ROW_FORMAT = row_format, ALGORITHM=INPLACE, LOCK=NONE;
```

데이터가 상당히 재구성되므로 비용이 많이 드는 작업입니다.

ROW_FORMAT 옵션에 대한 자세한 내용은 테이블 옵션을 참조하십시오.

- 키 블록 크기 변경하기

```
ALTER TABLE tbl_name KEY_BLOCK_SIZE = value, ALGORITHM=INPLACE, LOCK=NONE;
```

데이터가 상당히 재구성되므로 비용이 많이 드는 작업입니다.

키 블록 크기 옵션에 대한 자세한 내용은 테이블 옵션을 참조하십시오.

- 영구 테이블 통계 옵션 설정

```
ALTER TABLE tbl_name STATS_PERSISTENT=0, STATS_SAMPLE_PAGES=20, STATS_AUTO_RECALC=1, ALGORITHM=INPLAC
```

테이블 메타데이터만 수정합니다.

영구 통계에는 STATS_PERSISTENT, STATS_AUTO_RECALC 및 STATS_SAMPLE_PAGES가 포함됩니다. 자세한 내용은 섹션 15.8.10.1, "영구 옵티마이저 통계 매개변수 구성"을 참조하십시오.

- 문자 집합 지정

```
ALTER TABLE tbl_name 문자 집합 = 문자 집합_이름, 알고리즘=인플레이스, 잠금=없음;
```

새 문자 인코딩이 다른 경우 테이블을 다시 작성합니다.

- 문자 집합 변환

```
ALTER TABLE tbl_name CONVERT TO CHARACTER SET charset_name, ALGORITHM=COPY;
```

새 문자 인코딩이 다른 경우 테이블을 다시 작성합니다.

- 테이블 최적화

```
테이블 tbl_name을 최적화합니다;
```

전체 텍스트 인덱스가 있는 테이블에는 제자리 작업이 지원되지 않습니다. 이 작업은 알고리즘을 사용할 수 있지만 알고리즘 및 잠금 구문은 허용되지 않습니다.

- FORCE 옵션으로 테이블 재구성하기

```
ALTER TABLE tbl_name FORCE, ALGORITHM=INPLACE, LOCK=NONE;
```

MySQL 5.6.17부터 ALGORITHM=INPLACE를 사용합니다. 전체 텍스트 인덱스가 있는 테이블에는 ALGORITHM=INPLACE가 지원되지 않습니다.

- "널" 리빌드 수행

```
ALTER TABLE tbl_name 엔진=InnoDB, 알고리즘=인플레이스, 잠금=없음;
```

MySQL 5.6.17부터 ALGORITHM=INPLACE를 사용합니다. 전체 텍스트 인덱스가 있는 테이블에는 ALGORITHM=INPLACE가 지원되지 않습니다.

- 테이블 이름 바꾸기

```
ALTER TABLE old_tbl_name RENAME TO new_tbl_name, ALGORITHM=INSTANT;
```

테이블 이름 변경은 즉시 또는 제자리에서 수행할 수 있습니다. MySQL은 복사본을 만들지 않고 *tbl_name* 테이블에 해당하는 파일의 이름을 바꿉니다. (RENAME TABLE 문을 사용하여 테이블 이름을 바꿀 수도 있습니다. 섹션 13.1.36, "RENAME TABLE 문" 참조). 이름이 변경된 테이블에 대해 특별히 부여된 권한은 새 이름으로 마이그레이션되지 않습니다. 수동으로 변경해야 합니다.

테이블 스페이스 작업

다음 표에서는 테이블 스페이스 작업에 대한 온라인 DDL 지원에 대한 개요를 제공합니다. 자세한 내용은 구문 및 사용 참고 사항을 참조하세요.

표 15.22 테이블스페이스 운영을 위한 온라인 DDL 지원

운영	즉시	제자리에서	테이블 다시 빌드	동시 DML 허용	메타데이터만 수정
일반 테이블 스페이스 이름 바꾸기	아니요	예	아니요	예	예
일반 활성화 또는 비활성화	아니요	예	아니요	예	아니요

운영	즉시	제자리에서	테이블 다시 빌드	동시 DML 허용	메타데이터만 수정
테이블 스페이스 암호화					
테이블별 테이블 공간 암호화 사용 또는 사용 안 함	아니요	아니요	예	아니요	아니요

구문 및 사용 참고 사항

- 일반 테이블 스페이스 이름 바꾸기

```
테이블스페이스 테이블스페이스_이름을 새 테이블스페이스_이름으로 변경합니다;
```

테이블 스페이스 변경 ... RENAME TO는 INPLACE 알고리즘을 사용하지만 알고리즘 조항.

- 일반 테이블 공간 암호화 사용 또는 사용 안 함

```
ALTER TABLESPACE tablespace_name ENCRYPTION='Y';
```

테이블 스페이스 변경 ... 암호화는 INPLACE 알고리즘을 사용하지만 알고리즘 조항.

관련 정보는 섹션 15.13, "InnoDB 저장 데이터 암호화"를 참조하세요.

- 테이블별 테이블 공간 암호화 사용 또는 사용 안 함

```
ALTER TABLE tbl_name ENCRYPTION='Y', ALGORITHM=COPY;
```

관련 정보는 섹션 15.13, "InnoDB 저장 데이터 암호화"를 참조하세요.

파티셔닝 작업

일부 ALTER TABLE 파티셔닝 절을 제외하고, 파티셔닝된 테이블에 대한 온라인 DDL 작업은 InnoDB 테이블은 일반 InnoDB 테이블에 적용되는 것과 동일한 규칙을 따릅니다.

일부 ALTER TABLE 파티셔닝 절은 파티셔닝되지 않은 일반 InnoDB 테이블과 동일한 내부 온라인 DDL API를 거치지 않습니다. 따라서 ALTER TABLE 분할 절에 대한 온라인 지원은 다양합니다.

다음 표는 각 ALTER TABLE 분할 문의 온라인 상태를 보여줍니다. 사용되는 온라인 DDL API에 관계없이 MySQL은 가능한 경우 데이터 복사 및 잠금을 최소화하려고 시도합니다.

ALGORITHM=COPY를 사용하거나 "ALGORITHM=기본값, LOCK=기본값"만 허용하는 ALTER TABLE 파티셔닝 옵션은 COPY 알고리즘을 사용하여 테이블을 다시 파티셔닝합니다. 즉, 새 분할 방식으로 분할된 새 테이블이 생성됩니다. 새로 생성된 테이블에는 ALTER TABLE 문에 의해 적용된 모든 변경 사항이 포함되며 테이블 데이터는 새 테이블 구조로 복사됩니다.

표 15.23 파티셔닝 작업을 위한 온라인 DDL 지원

파티셔닝 조항	즉시	제자리에서	DML 허용	참고
파티션 기준	아니요	아니요	아니요	허용 알고리즘 = 복사, 잠금 = {기본값 |

파티셔닝 조항	즉시	제자리에서	DML 허용	참고	
				공유| 독점}	
파티션 추가	아니요	예*	예*	알고리즘=인플락 잠금 ={기본값| 없음| 공 유| 독점} 지원 범위 및 파티션 나열, 알고리즘 인플레이스 잠금={기 본값| 공유| 제외} 해시 및 키 파티션, ALGORITHM=COPY, LOCK={공유| 독점} 경우 모든 파티션 유형. 범위 또는 목록으로 분할된 테이블의 기 존 데이터를 복사하 지 않습니다. 동시 쿼리 와 함께 허용됩니다. 알고리즘=복사 를 사용하여 해시 또는 목록으로 분할 된 테이블의 경우 MySQL이 공유 잠 금을 유지하면서 데 이터를 복사하기 때 문입니다.	E, E, E,
파티션 삭제	아니요	예*	예*	알고리즘=인플레이스 잠금={기본값| 없음 공유| 독점}은 다음과 같습니다. 지원됩니다. Does 범위 또는 목록으로 분할된 테이블의 데 이터를 복사하지 않 습니다.	E 켜기
				DROP 파티션 알고리즘=인플랙	

파티셔닝 조항	즉시	제자리에서	DML 허용	참고
				에서 데이터 이동 호환되는 파티션이 있는 다른 파티션에 파티션을 삭제했습니다 . . . VALUES 정의. 다른 파티션으로 옮길 수 없는 데이터는 삭제됩니다.
파티션 삭제	아니요	아니요	아니요	알고리즘=기본 잠금=기본값만 허용합니다.
파티션 가져오기	아니요	아니요	아니요	알고리즘=기본 잠금=기본값만 허용합니다.
파티션 잘라내기	아니요	예	예	기존 데이터를 복사하지 않습니다. 행만 삭제할 뿐 테이블 자체나 파티션의 정의는 변경하지 않습니다.
통합 파티션	아니요	예*	아니요	알고리즘=인플레이스 잠금={기본값| 공유 독점}은 다음과 같습니다. 지원됩니다.
파티션 재구성	아니요	예*	아니요	알고리즘=인플레이스 잠금={기본값| 공유 독점}은 다음과 같습니다. 지원됩니다.
교환 파티션	아니요	예	예	
파티션 분석	아니요	예	예	
파티션 확인	아니요	예	예	
파티션 최적화	아니요	아니요	아니요	알고리즘 및 LOCK 절은 무시됩니다. 전체 테이블을 다시 작성합니다. 섹션 24.3.4, "파티션

파티셔닝 조항	즉시	제자리에서	DML 허용	참고
				EXCLUSIVE}는 지원됩니다.
파티션 복구	아니요	예	예	
파티셔닝 제거	아니요	아니요	아니요	알고리즘=복사, 잠금={기본값| 공유| 독점} 권한 부여

파티션이 분할되지 않은 테이블에 대한 비파티션 온라인 `ALTER TABLE` 작업은 일반 테이블에 적용되는 것과 동일한 규칙을 따릅니다. 그러나 `ALTER TABLE`은 각 테이블 파티션에서 온라인 작업을 수행하므로 여러 파티션에서 작업이 수행되므로 시스템 리소스에 대한 요구가 증가합니다.

테이블 분할 변경 절에 대한 자세한 내용은 분할 옵션 및 섹션 13.1.9.1, "테이블 분할 작업 변경"을 참조하십시오. 파티셔닝에 대한 일반적인 정보는 24장, *파티셔닝을 참고한다*.

15.12.2 온라인 DDL 성능 및 동시성

온라인 DDL은 MySQL 운영의 여러 측면을 개선합니다:

- 테이블에 액세스하는 애플리케이션은 DDL 작업이 진행되는 동안 테이블에 대한 쿼리 및 DDL 작업을 진행할 수 있으므로 응답성이 향상됩니다. MySQL 서버 리소스에 대한 잠금 및 대기 시간이 줄어들어 DDL 작업에 포함되지 않은 작업의 경우에도 확장성이 향상됩니다.
- 인스턴트 연산은 데이터 사전의 메타데이터만 수정합니다. 작업 실행 단계에서 테이블에 대한 독점 메타데이터 잠금이 잠시 걸릴 수 있습니다. 테이블 데이터는 영향을 받지 않으므로 즉각적인 작업이 가능합니다. 동시 DML이 허용됩니다.
- 온라인 작업은 테이블 복사 방법과 관련된 디스크 I/O 및 CPU 주기를 피하므로 데이터베이스의 전체 부하를 최소화합니다. 부하를 최소화하면 DDL 작업 중에 우수한 성능과 높은 처리량을 유지하는 데 도움이 됩니다.
- 온라인 작업은 테이블 복사 작업보다 버퍼 풀에 데이터를 덜 읽으므로 메모리에서 자주 액세스하는 데이터의 제거가 줄어듭니다. 자주 액세스하는 데이터를 제거하면 DDL 작업 후 일시적인 성능 저하가 발생할 수 있습니다.

LOCK 절

기본적으로 MySQL은 DDL 작업 중에 가능한 한 잠금을 적게 사용합니다. 다음과 같은 경우 인플레이스 작업 및 일부 복사 작업에 대해 `LOCK` 절을 지정하여 보다 제한적인 잠금을 적용할 수 있습니다.
필수입니다. `LOCK` 절이 특정 DDL 작업에 허용되는 것보다 덜 제한적인 수준의 잠금을 지정하면 오류가 발생

하고 문이 실패합니다. `LOCK` 절은 가장 제한적인 것부터 가장 제한적인 것까지 순서대로 아래에 설명되어 있습니다:

- `LOCK=NONE`:

  동시 쿼리 및 DML을 허용합니다.

  예를 들어, 고객 가입 또는 구매와 관련된 테이블에 이 절을 사용하면 긴 DDL 작업 중에 테이블을 사용할 수 없게 되는 것을 방지할 수 있습니다.

- 잠금=공유:

동시 쿼리는 허용하지만 DML은 차단합니다.

예를 들어, 데이터 웨어하우스 테이블에서 이 절을 사용하면 DDL 작업이 완료될 때까지 데이터 로드 작업을 지연시킬 수 있지만 쿼리는 장시간 지연시킬 수 없습니다.

- LOCK=기본값:

  가능한 한 많은 동시성(동시 쿼리, DML 또는 둘 다)을 허용합니다. 잠금 생략 절을 지정하는 것은 LOCK=DEFAULT를 지정하는 것과 동일합니다.

  DDL 문의 기본 잠금 수준이 테이블의 가용성 문제를 일으키지 않을 것으로 예상되는 경우 이 절을 사용합니다.

- 잠금=독점:

  동시 쿼리 및 DML을 차단합니다.

  가능한 한 짧은 시간 내에 DDL 작업을 완료하는 것이 주요 관심사이고 쿼리 및 DML 동시 액세스가 필요하지 않은 경우 이 절을 사용합니다. 서버가 유휴 상태여야 하는 경우 예기치 않은 테이블 액세스를 피하기 위해 이 절을 사용할 수도 있습니다.

온라인 DDL 및 메타데이터 잠금

온라인 DDL 작업은 세 단계로 나누어 볼 수 있습니다:

- *1단계: 초기화*

  초기화 단계에서 서버는 스토리지 엔진 기능, 문에 지정된 작업, 사용자 지정 알고리즘 및 잠금 옵션을 고려하여 작업 중에 허용되는 동시성의 양을 결정합니다. 이 단계에서는 현재 테이블 정의를 보호하기 위해 공유 업그레이드 가능한 메타데이터 잠금이 수행됩니다.

- *2단계: 실행*

  이 단계에서는 문이 준비되고 실행됩니다. 메타데이터 잠금이 배타적으로 업그레이드되는지 여부는 초기화 단계에서 평가된 요소에 따라 달라집니다. 독점 메타데이터 잠금이 필요한 경우, 문 준비 중에 잠시만 수행됩니다.

- *3단계: 커밋 테이블 정의*

  커밋 테이블 정의 단계에서 메타데이터 잠금이 전용으로 업그레이드되어 이전 테이블 정의를 내리고 새 정의를 커밋합니다. 일단 부여되면 독점 메타데이터 잠금은 짧은 기간 동안 유지됩니다.

위에서 설명한 독점 메타데이터 잠금 요구 사항으로 인해 온라인 DDL 작업은 테이블에 메타데이터 잠금을 보유한 동시 트랜잭션이 커밋 또는 롤백될 때까지 기다려야 할 수 있습니다. DDL 작업 전 또는 작업 중에 시작된 트랜잭션은 변경되는 테이블에 메타데이터 잠금을 유지할 수 있습니다. 오래 실행 중이거나 비활성 상

태인 트랜잭션의 경우, 온라인 DDL 작업이 시간 초과될 수 있습니다.

독점 메타데이터 잠금을 기다리는 중입니다. 또한 온라인 DDL 작업에서 요청된 보류 중인 독점 메타데이터 잠금은 테이블의 후속 트랜잭션을 차단합니다.

다음 예제에서는 독점 메타데이터 잠금을 기다리는 온라인 DDL 작업과 보류 중인 메타데이터 잠금이 테이블의 후속 트랜잭션을 차단하는 방법을 보여 줍니다.

세션 1:

```
mysql> CREATE TABLE t1 (c1 INT) ENGINE=InnoDB;
mysql> START TRANSACTION;
mysql> SELECT * FROM t1;
```

세션 1 `SELECT` 문은 테이블 t1에 대한 공유 메타데이터 잠금을 사용합니다.

세션 2:

```
mysql> ALTER TABLE t1 ADD COLUMN x INT, ALGORITHM=INPLACE, LOCK=NONE;
```

테이블 정의 변경 사항을 커밋하기 위해 테이블 t1에 대한 독점 메타데이터 잠금이 필요한 세션 2의 온라인 DDL 작업은 세션 1 트랜잭션이 커밋되거나 롤백될 때까지 기다려야 합니다.

세션 3:

```
mysql> SELECT * FROM t1;
```

세션 3에서 실행된 `SELECT` 문은 세션 2의 `ALTER TABLE` 작업에서 요청한 독점 메타데이터 잠금이 부여되기를 기다리며 차단됩니다.

전체 프로세스 목록 표시를사용하여 트랜잭션이 메타데이터 잠금을 기다리고 있는지 확인할 수 있습니다.

```
mysql> 전체 프로세스 목록 표시\G
...
*************************** 2. 행 ***************************
     Id: 5
   사용자: root 호스
   트: localhost
     db: test
명령: Query
   시간: 44
  상태입니다: 테이블 메타데이터 잠금을 기다리는 중
   Info: ALTER TABLE t1 ADD COLUMN x INT, ALGORITHM=INPLACE, LOCK=NONE
...
*************************** 4. 행 ***************************
     Id: 7
   사용자: root 호스
   트: localhost
     db: test
명령: Query
   시간: 5
  상태입니다: 테이블 메타데이터 잠금을 기다리는 중 정
   보: SELECT * FROM t1
4행 세트(0.00초)
```

메타데이터 잠금 정보는 세션 간의 메타데이터 잠금 종속성, 세션이 대기 중인 메타데이터 잠금, 현재 메타데이터 잠금을 보유하고 있는 세션에 대한 정보를 제공하는 성능 스키마 메타데이터_잠금 테이블을 통해서도 노출됩니다. 자세한 내용은 섹션 27.12.13.3, "메타데이터 잠금 테이블"을 참조하세요.

온라인 DDL 성능

DDL 작업의 성능은 주로 작업이 즉시 수행되는지, 제자리에서 수행되는지, 테이블을 다시 빌드하는지 여부에 따라 결정됩니다.

DDL 작업의 상대적인 성능을 평가하기 위해 `ALGORITHM=INSTANT`, `ALGORITHM=INPLACE` 및 `ALGORITHM=COPY`를 사용하여 결과를 비교할 수 있습니다. `old_alter_table`을 활성화한 상태에서 문을 실행하여 `ALGORITHM=COPY`를 사용하도록 강제할 수도 있습니다.

테이블 데이터를 수정하는 DDL 작업의 경우 명령이 완료된 후 표시되는 "영향을 받는 행" 값을 확인하여 DDL 작업이 변경 사항을 제자리에서 수행할지 아니면 테이블 복사를 수행할지 결정할 수 있습니다. 예를 들면 다음과 같습니다:

- 열의 기본값을 변경합니다(빠르게, 테이블 데이터에는 영향을 주지 않음):

```
쿼리 확인, 영향을 받는 행 0개(0.07초)
```

- 인덱스 추가(시간이 걸리지만 영향을 받는 행이 0개이면 테이블이 복사되지 않았음을 나타냅니다):

```
쿼리 확인, 영향을 받은 행 0개(21.42초)
```

- 열의 데이터 유형 변경(상당한 시간이 소요되며 테이블의 모든 행을 다시 작성해야 함):