4.4 InnoDB 스토리지 엔진의 잠금

생성일	@2021년 5월 30일 오후 10:26
태그	

InnoDB는 레코드 기반의 잠금 방식을 사용 → 뛰어난 동시성 처리!

5.1 버젼부터 InnoDB의 트랜잭션과 잠금, 그리고 대기중인 트랜잭션 목록을 조회할 수 있음

- INNODB_TRX
- INNODB_LOCKS
- INNODB_LOCK WAITS

→ 장시간 잠금을 가지고 있는 클라이언트를 종료시킬 수도 있다

1. InnoDB의 잠금 방식

Pessimistic Locking

- 현재 트랜잭션에서 변경하고자 하는 레코드에 대해 잠금을 획득/변경 작업을 처리하는 방식
  - 현재 변경하고자 하는 레코드를 다른 트랜잭션에서도 변경할 수 있다 (는 ***비관적 가정***)
  - InnoDB에서 이 방식을 채택
  - 높은 동시성 처리에는 비관적 잠금이 유리하다고 알려져 있음

Optimistic Locking

- 각 트랜잭션이 같은 레코드를 변경할 가능성이 희박할 것이라고 (**낙관적으로**) 가정
- 우선 먼저 변경 작업을 수행하고, 마지막에 잠금 충돌이 있었는지 확인
- 문제 있었을시 ROLLBACK

2. InnoDB의 잠금 종류

락 에스컬레이션

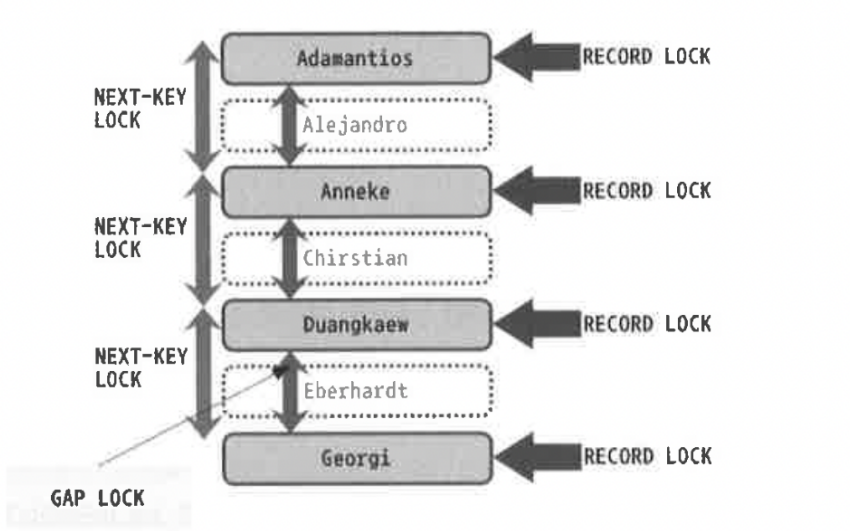
레코드락 → 페이지락 → 테이블락 이렇게 레벨 업 되는 경우

레코드 락

레코드 자체만을 잠그며, InnoDB에서는 레코드 자체가 아니라 인덱스의 레코드를 잠근다

- 인덱스가 하나도 없는 테이블이어도 내부적으로 자동 생성된 클러스터 인덱스로 잠금 설정

갭 락



[그림 4-1] InnoDB 잠금의 종류(점선의 레코드는 실제 존재하지 않는 레코드를 가정한 것임)

- 레코드와 레코드 사이의 간격을 잠그는 락
- 즉, 레코드와 바로 인접한 레코드 사이의 간격만을 잠그는 것 (사이 간격에 새 레코드가 Insert되는 것을 제어)
- 개념일 뿐, 자체적으로 사용되지는 않음

넥스트 키 락

레코드락과 갭 락을 합쳐놓은 형태의 잠금

- STATEMENT 포맷의 바이너리 로그를 사용하는 MySQL 서버에서는 REPEATABLE READ 격리 수준을 사용해야함
- 바이너리 로그(마스터)에 기록되는 쿼리가 슬레이브에서 실행될때, 마스터와 동일한 결과를 만들어내도록 보장하는 것이 주 목적

- 근데 이것으로 인해 데드락이나 트랜잭션을 기다리게 만드는 일이 자주 발생
- 가능하면 바이너리 로그 포맷을 ROW형태로 바꿔서 넥스트 키/갭 락을 줄이는 것이 좋음
    - 아직 이 포맷은 널리 사용되지 않고, STATEMENT 포맷에 비해 로그파일 크기가 커질 수 잇음

자동 증가(AUTO_INCREMENT) 락

AUTO_INCREMENT 컬럼 속성 : 자동 증가하는 숫자 값 채번

- 해당 설정이 사용된 테이블에 여러 개의 레코드가 동시에 INSERT 되는 경우, 저장되는 각 레코드는 중복없이 저장된 순서대로 증가하는 일련번호 값 가져야 함
- InnoDB에서 이를 위해 내부적으로 자동 증가 락(테이블 수준의 잠금)을 사용
- INSERT/REPLACE 같은 새 레코드 저장하는 쿼리에서만 필요함 (UPDATE, DELETE에서는 걸리지 않음)
- 트랜잭션과 상관 없이 AUTO_INCREMENT 값을 가져오는 순간만 락이 걸렸다가 즉시 해제됨
    - 동시에 INSERT 쿼리 2개가 실행되는 경우 한개 쿼리가 실행될때 나머지 쿼리는 자동 증가 락을 기다려야함
    - AUTO_INCREMENT 칼럼에 명시적으로 값 설정해도 해당 락이 걸림
- 자동 증가 락을 명시적으로 획득/해제하는 방법은 없음

5.1 버젼부터...
[innodb_autoinc_lock_mode] 파라미터로 ***자동 증가 락의 작동 방식을 변경***할 수 있음

- innodb_autoinc_lock_mode = 0 : 모든 INSERT 문장은 자동증가 락 사용
- innodb_autoinc_lock_mode = 1 : INSERT되는 레코드의 건수를 정확히 예측할 수 있을때, 자동 증가 락을 사용하지 않고, 훨씬 가볍고 빠른 래치(뮤텍스)를 이용해 처리
  INSERT ... (SELECT ...)의 경우처럼 <대량 INSERT>의 건 수를 예측 못할때는 래치사용 안함
  → 해당 쿼리문이 완료되기 전까지 자동증가 락이 해제되지 않음

→ 한번에 여러개의 자동 증가 값을 한번에 할당 되기 때문에 연속적인 INCREMENT 값을 가짐 (남는 값은 폐기하므로 대량 insert 이후의 레코드는 누락된 값 가질 수 있음)

- innodb_autoinc_lock_mode = 2 : 자동증가 락 걸지 않고 항상 래치(뮤텍스) 사용

하나의 INSERT 문장으로 생성되는 레코드여도 연속된 자동 증가 값을 보장 하지는 않음.

대량 INSERT시 다른 커넥션에서도 INSERT를 수행할 수 있어서 동시성이 높아지지만 마스터 슬레이브의 자동 증가 값이 달라질 가능성이 있음

3. 인덱스와 잠금

InnoDB의 잠금과 인덱스는 상당히 중요한 연관 관계가 있다

> 💡 InnoDB의 잠금은 인덱스를 잠금하기 때문에, 변경해야 할 레코드를 찾기위해 검색한 인덱스의 레코드를 모두 잠근다

```
// 예제 데이터베이스의 employees 테이블에는 아래와 같이 first_name 칼럼만
// 멤버로 담긴 ix_firstname이라는 인덱스가 준비돼 있다.
// KEY ix_firstname (first_name)

// employees 테이블에서 first_name='Georgi'인 사원은 전체 253명이 있으며,
// first_name='Georgi'이고 last_name='Klassen'인 사원은 딱 1명만 있는 것을 아래 쿼리로
// 확인할 수 있다.
```

```
mysql> UPDATE employees SET hire_date=NOW()
       WHERE first_name='Georgi' AND last_name='Klassen';
```

위 업데이트 문 수행시, 253개의 레코드가 다 락이 걸린다

(오라클은 다른 방식임)

만약 이 테이블에 인덱스가 없다면 테이블을 풀 스캔 해야 하므로 테이블에 있는 30만개 레코드를 모두 잠근다...

4. 트랜잭션 격리 수준과 잠금

위의 불필요한 레코드의 잠금현상 = InnoDB의 **넥스트 키 락** 때문에 발생하는 것

넥스트 키 락이 필요한 이유 : slave 복제를 위한 바이너리 로그 때문

갭락이나 넥스트 키 락의 사용을 줄이는 방법

- Row based binary log를 사용
- 바이너리 로그를 사용하지 않는 경우

버전	설정의 조합
MySQL 5.0	innodb_locks_unsafe_for_binlog=1 트랜잭션 격리 수준을 READ-COMMITTED로 설정
MySQL 5.1 이상	바이너리 로그를 비활성화 트랜잭션 격리 수준을 READ-COMMITTED로 설정
	레코드 기반의 바이너리 로그 사용 innodb_locks_unsafe_for_binlog=1 트랜잭션 격리 수준을 READ-COMMITTED로 설정

5.1 이상의 버전에서는 바이너리 로그가 활성화되면 최소 repeatable - read 이상의 격리 수준을 사용하도록 강제 되고 있음

→ 이 조합의 설정에서도 유니크키나 외래키에 대한 갭락은 없어지지 않는다

인덱스 검색시 불필요한 잠금의 감소

update 문장을 처리하기 위해 일치하는 레코드를 인덱스를 이용해 검색할때 우선 인덱스만 비교해서 일치하는 레코드에 대해 배타적 잠금을 걸게 됨

→ 그 다음 나머지 조건을 비교해서 일치하지 않는 레코드는 즉시 잠금 해제

5. 레코드 수준의 잠금 확인 및 해제

레코드가 자주 사용되지 않는다면 오랜 시간 동안 잠겨진 상태로 남아 있어도 잘 발견되지 않는다

MySql 5.1 버전부터 레코드 잠금과 잠금 대기에 대한 조회가 가능

커넥션 1	커넥션 2	커넥션 3
BEGIN;		
UPDATE employees SET birth_date=NOW() WHERE emp_no=100001;		
	UPDATE employees SET hire_date=NOW() WHERE emp_no=100001;	
		UPDATE employees SET birth_date=NOW(), hire_date=NOW() WHERE emp_no=100001;

```
mysql> SELECT * FROM information_schema.innodb_locks;
*************************** 1. row ***************************
lock_id     : 34A7:78:298:25
lock_trx_id : 34A7
lock_mode   : X
lock_type   : RECORD
lock_table  : 'employees'.'employees'
lock_index  : 'PRIMARY'
lock_space  : 78
lock_page   : 298
lock_rec    : 25
lock_data   : 100001

...

mysql> SELECT * FROM information_schema.innodb_trx;
*************************** 1. row ***************************
trx_id                  : 34A7
trx_state               : LOCK WAIT
trx_started             : 2011-08-10 14:46:55
trx_requested_lock_id   : 34A7:78:298:25
trx_wait_started        : 2011-08-10 14:46:55
trx_weight              : 2
trx_mysql_thread_id     : 100
trx_query               :  UPDATE employees SET birth_date=now(), hire_date=now() WHERE
emp_no=100001

...
```

- INNODB_LOCKS : 어떤 잠금이 존재하는지

- INNODB_TRX : 어떤 트랜잭션이 클라이언트에 의해 가동되며, 어떤 잠금을 기다리고 있는지
- INNODB_LOCK_WAITS : 락에 의한 프로세스 간의 의존관계 관리

위 테이블들을 조인해서 waiting, blocking transaction을 확인하면 된다

```
*************************** 1. row ***************************
waiting_trx_id   : 34A7
waiting_thread   : 100
waiting_query     : UPDATE employees SET birth_date=now(), hire_date=now() WHERE emp_
no=100001
blocking_trx_id  : 34A6
blocking_thread  : 99
blocking_query   : UPDATE employees SET hire_date=now() WHERE emp_no=100001

*************************** 2. row ***************************
waiting_trx_id   : 34A7
waiting_thread   : 100
waiting_query     : UPDATE employees SET birth_date=now(), hire_date=now() WHERE emp_
no=100001
blocking_trx_id  : 34A5
blocking_thread  : 18
blocking_query   : NULL

*************************** 3. row ***************************
waiting_trx_id   : 34A6
waiting_thread   : 99
waiting_query    : UPDATE employees SET hire_date=now() WHERE emp_no=100001
blocking_trx_id  : 34A5
blocking_thread  : 18
blocking_query   : NULL
```