MySQL学习笔记（Day033：锁_6 / 事物_1）

MySQL学习

一. AI自增锁

- 一个表一个 自增列
- `AUTO_INCREMENT` PK
- 在事物提交前就释放
  - 其他的锁是在事物提交时才释放
  - 如果AI锁在提交后才释放，那其他事物就没法插入了，无法实现并发

1.1. AI自增锁示例

```
--
-- 终端会话1
--
mysql> create table t_ai_1(a int auto_increment, b int , primary key(a));
Query OK, 0 rows affected (0.14 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into t_ai_1 values (NULL, 10);  -- 插入一个值，且事物没有提交
Query OK, 1 row affected (0.00 sec)

--
-- 终端会话2
--
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into t_ai_1 values(NULL, 20);  -- 尽管 终端会话1 中的事物没有提交，
                                             -- 但是终端会话2中的事物仍能提交，说明AI锁已经释放了
Query OK, 1 row affected (0.01 sec)

-- AI 锁 在事物提交前就释放了，类似latch，使用完就释放了


--
-- 终端会话1
--
mysql> rollback;
Query OK, 0 rows affected (0.02 sec)

--
-- 终端会话2
--
mysql> rollback;
Query OK, 0 rows affected (0.01 sec)

--
-- 终端会话1
--
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into t_ai_1 values (NULL, 30);
Query OK, 1 row affected (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.03 sec)

mysql> select * from t_ai_1;
+---+------+
| a | b    |
+---+------+
| 3 |   30 |  -- 因为之前的两次插入都rollback了，但是AI锁其实是提交的，导致现在插入时，自增列的序号从3开始
+---+------+
1 row in set (0.00 sec)
```

> 事物回滚后，自增值 不会 跟着回滚，导致自增值不连续，但是这个值连续也没什么意义。

1.2. AI自增锁的方式

插入类型	说明
insert-like	insert-like指所有的插入语句，如INSERT、REPLACE、INSERT…SELECT，REPLACE…SEECT、LOAD DATA等
simple inserts	simple inserts指能在插入前就确定插入行数的语句。这些语句包括INSERT、REPLACE等。需要注意的是：simple inserts不包含INSERT …ON DUPLICATE KEY UPDATE这类SQL语句
bulk inserts	bulk inserts指在插入前不能确定得到插入行数的语句，如INSERT…SELECT，REPLACE…SELECT，LOAD DATA
mixed-mode inserts	mixed-mode inserts指插入中有一部分的值是自增长的，有一部分是确定的。如INSERT INTO t1 (c1,c2) VALUES (1,'a'), (NULL,'b'), (5,'c'), (NULL,'d'); 也可以是指INSERT …ON DUPLICATE KEY UPDATE这类SQL语句

- 所有的插入都是insert-like
- 如果插入前能 确定行数 的，就是 `simple inserts`
  - `insert into table_1 values(NULL, 1), (NULL, 2);`
- 如果插入前 不能确定行数 的，就是 `bulk inserts`
  - `insert into table_1 select * from table_2;`
- 如果 部分自增长，部分指定 的，就是 `mixed-mode inserts`

> `insert ... on duplicate key update` 不推荐 使用
>
> 1. 非ANSI SQL 标准
> 2. 效果并非预期所期望的那样

1.2.1. 设置自增并发度

- `--innodb_autoinc_lock_mode={0|1|2}`
  - **`0` 传统方式**
    - 在 `SQL语句执行完之后` ，AI锁才释放
    - 例如：当 `insert ... select ...` 数据量很大时（比如执行10分钟），那在这个 `SQL执行完毕前` ，其他事物是 不能插入 的（AI锁未释放）
    - 这样可以保证在这个SQL语句内插入的数据，自增值是 连续 的，因为在这个10分钟内，AI自增锁是被这个SQL持有的，且没有释放
  - **`1` 默认参数**（ 大部分情况设置为`1` ）
    - bulk inserts， 同传统方式一样
    - 对于 `bulk inserts` 的方式，和 `0 - 传统方式` 一样，在SQL执行完之后，AI锁才释放
    - simple inserts， 并发方式
      - 在 `SQL运行完之前` ， `确定了自增值之后` ，就可以释放自增锁了

```
                          +
       bulk inserts       |       simple inserts
                          |
+-------------------------------------------------------+
                          |
    acquire AI_Lock       |     acquire AI_Lock
                          |
       insert ... select ... |        ai = ai + M
                          |
       ai = ai + N        |     release AI_Lock
                          |
    release AI_Lock       |        insert ... select ...
                          +
```

> 因为 `bulk inserts` 不知道要插入多少行，所以只能等insert结束后，才知道 `N` 的值，然后 `一次性(ai + N)`
> 而 `simple inserts` 知道插入的行数（M），所以可以先 `(ai + M)` ，然后将锁释放掉，给别的事物用，然后自己慢慢插入数据

  - **`2`**
    - 所有自增都可以并发方式（ `不同于Simple inserts的方式` ）
    - 同一SQL语句自增可能不连续
    - **`row-based binlog`**

```
for (i = ai; until_no_rec; i++) {
        acquire AI_Lock           # 插入前申请锁
            insert one record...  # 只插入一条记录
            ai = ai + 1           # 自增值+1
        release AI_Lock           # 释放锁
}
```

> 这样做的好处是，对于批量的、耗时的插入，SQL不会长时间的持有AI自增锁，而是插入 `一条（有且仅插入一条，而simple inserts是确定好的M条）` 语句后就 `释放` ，这样可以给别的事物使用，实现并发。
> 但是这种方式 `并发度是增加了` ，但是性能不一定变好，尤其是单线程导入数据时，要 `不断的申请和释放锁`
> 对于批量插入来说，自增就可能变的不连续了（需要和开发沟通，是否可以接受）

- `innodb_autoinc_lock_mode` 是 `read-only` 的， `需要` 修改后 `重启` MySQL实例

1.3. 自增列与索引

```
mysql> create table t_ai_2(a int auto_increment, b int , key(b, a));
ERROR 1075 (42000): Incorrect table definition; there can be only one auto column and it must be defined as a key
```

> 自增列必须定义为索引列，且 必须是第一个列，这样做的好处是，因为索引的第一个列是排序的，这样重启后，数据库可以直接找到 最后一个自增值，然后 +1 操作后，就可以作为 下一个自增列的值 了（否则要全表扫描了；MySQL的自增列的值 不是持久化 的）。

1.4. 自增的两个参数

- --auto_increment_increment = 1
- --auto_increment_offset = 1

```
mysql> show variables like "%auto%increment%";
+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| auto_increment_increment   | 1     |   -- 步长
| auto_increment_offset      | 1     |   -- 初始值
| innodb_autoextend_increment | 64    |
+----------------------------+-------+
3 rows in set (0.00 sec)
```

```
                                +
                                |
                                |
             Server  A          |            Server B
                                |
+-------------------------------------------------------------------------+
                                |
            Table User          |           Table User
                                |
     PK                         |     PK
             offset = 1         |           offset = 2
                                |
             increment = 2      |           increment = 4
                                |
     PK Value:                  |     PK Value:
              1 3 5 7 9 ...     |              2 4 6 8 10 ...
                                +
```

> 两个服务器上的User表的主键的值就没有交叉了
> 如果有三台服务器，则 A:[offset = 1, increment=3] ， B:[offset = 2, increment=3] ， C:[offset = 3, increment=3]
> 如果一开始不知道后面会有多少台服务器，则可以一开始把increment设置的大一点，比如是10，这样只会浪费一点自增值
> 这样做的目的是保证 每个节点 上产生的 自增值 是 全局唯一 的，这样做并不能用来做双主（比如一些额外的唯一索引能保证全局唯一么？）

二. SERIALIZABLE和REPEATABLE READ的区别

SERIALIZABLE 采用两阶段锁来保证隔离性

- 加锁阶段：只加锁，不放锁
- 解锁阶段：只放锁，不加锁。

且无论 读 还是 写 ， 都要加锁

但是这样做了以后， 失去了MVCC的特性 （非锁定的一致性读）

2.1. 两种隔离级别举例对比

- RR 隔离级别

```
--
-- 终端会话1
--
mysql> set tx_isolation="REPEATABLE-READ";
Query OK, 0 rows affected (0.00 sec)

mysql> create table t_lock_9(a int, b int, primary key(a));
Query OK, 0 rows affected (0.11 sec)

mysql> insert into t_lock_9 values(1,1);
Query OK, 1 row affected (0.00 sec)

mysql> select * from t_lock_9;
+---+------+
| a | b    |
+---+------+
| 1 |    1 |
+---+------+
1 row in set (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.02 sec)

mysql> select b from t_lock_9 where a=1;
+------+
| b    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)


--
-- 终端会话2
--
mysql> set tx_isolation="REPEATABLE-READ";
Query OK, 0 rows affected (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> update  t_lock_9 set b=2 where a = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> commit;
Query OK, 0 rows affected (0.03 sec)

--
-- 终端会话1
--
mysql> select b from t_lock_9 where a=1;  -- 再执行一次，得到的结果是1
+------+
| b    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)

mysql> select b from t_lock_9 where a=1 for update;  -- for update的去读，得到的结果是2
+------+
| b    |
+------+
|    2 |
+------+
1 row in set (0.00 sec)
```

> 终端会话1 中， RR隔离级别下，前两次读都是读取的快照，最后一次读取的当前更新的值

- SR隔离级别

```
-- 与之前一样，在MySQL5.6下测试，MySQL5.7.10版本测试没有锁，SR可以并发执行
--
-- 终端会话1
--
mysql> select * from t_lock_9;
+---+------+
| a | b    |
+---+------+
| 1 |    1 |
+---+------+
1 row in set (0.00 sec)

mysql> set tx_isolation='SERIALIZABLE';
Query OK, 0 rows affected (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select b from t_lock_9 where a=1;
+------+
| b    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)


--
-- 终端会话2
--
mysql> show engine innodb status\G
-- ------------省略部分输出------------
2 lock struct(s), heap size 360, 1 row lock(s)
MySQL thread id 3, OS thread handle 0x7f946bc94700, query id 30 localhost root cleaning up
TABLE LOCK table `burn_test`.`t_lock_9` trx id 5390 lock mode IS
RECORD LOCKS space id 15 page no 3 n bits 72 index `PRIMARY` of table `burn_test`.`t_lock_9` trx id 5390 lock mode S locks rec but not gap  -- 有S锁
Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
 0: len 4; hex 80000001; asc     ;;
 1: len 6; hex 00000000150c; asc       ;;
 2: len 7; hex 8c000000340110; asc     4  ;;
 3: len 4; hex 80000001; asc     ;;


mysql> set tx_isolation='SERIALIZABLE';
Query OK, 0 rows affected (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> update t_lock_9 set b=2 where a=1;  -- 在SR的隔离级别下，直接阻塞，因为a=1上有一个S锁
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
```

> SERIALIZABLE的隔离级别一般来说是不会去用

三. 事物（一）

3.1. 事物相关的主题

- transaction
- redo
- undo
- purge
- group commit
- XA
- transaction programming

3.2 开启一个事物

- 方式一
  - `begin;`
  - SQL…;
  - commit / rollback;
- 方式二
  - `start transaction;`
  - SQL…;
  - commit / rollback;

> 用到 `start transaction` 的原因是因为 `begin` 在 存储过程 中是一个关键字（代码块的开始）

3.3. 事物的ACID

- **A - Atomicity**（原子性）
  - redo
  - 事物内的SQL要么都执行完毕，要么全部回滚
- **C - consistency**（一致性）
  - undo
  - 事物的开启和结束后，没有破坏数据的结构和约束
- **`I - isolation`**（隔离性）
  - lock
  - 一个事物所做的修改，对其他事物是不可见的，好似串行执行的
- **D - Durable**（持久性）
  - redo & undo
  - 已经提交的事物是持久的，不考虑硬件损坏的情况

3.4. 事物的类型

- **Flat Transactions（平事物）**
  - 大部分的都是这个类型，
    - being;
    - SQL…;
    - commit / rollback;
- **Flat Transactions with Savepoints（带保存点的事物）**
  - 事物内执行完部分操作后，可以save work，如果回滚的话，可以回滚到该save work的状态
  - 在MySQL内 非常有用
- **Chained Transactions （链式事物）**
  - 在提交一个事物的时候，释放不需要的数据对象，将必要的处理上下文隐式的传给下一个要开始的事物
  - 提交事物操作和开始下一个事物操作将合并成为一个原子操作。这意味着下一个事物将看到上一个事物的结果，就好像在一个事物中执行的一样
  - 相当于在一个事物commit之后，立即打了一个begin（系统自动）
- Nested Transactions
  - InnoDB不支持
  - BDB支持
- **Distributed Transactions**
  - 在分布式环境下运行的扁平事物
  - 参与分布式的节点都要支持ACID
  - 举例：从招行转账10000到工行
    - 招行： - 10000
    - 工行： + 10000
- Long Transactions