零声教育出品 Mark 老师 QQ: 2548898954

事务

> 事务的本质是并发控制的单元，是用户定义的一个操作序列。这些操作要么都做，要么都不做，是一个不可分割的工作单位。

目的

> 事务将数据库从一种一致性状态转换为另一种一致性状态；保证系统始终处于一个完整且正确的状态；

组成

> 事务可由一条非常简单的 *SQL* 语句组成，也可以由一组复杂的 *SQL* 语句组成；

特征

> 在数据库提交事务时，可以确保要么所有修改都已经保存，要么所有修改都不保存；
>
> 事务是访问并更新数据库各种数据项的一个程序执行单元。
>
> 在 *MySQL innodb* 下，单条语句都具备事务；可以通过 `set autocommit = 0;` 设置当前会话手动提交；

事务控制语句

```
-- 显示开启事务
START TRANSACTION | BEGIN
-- 提交事务，并使得已对数据库做的所有修改持久化
COMMIT
-- 回滚事务，结束用户的事务，并撤销正在进行的所有未提交的修改
ROLLBACK
-- 创建一个保存点，一个事务可以有多个保存点
SAVEPOINT identifier
-- 删除一个保存点
RELEASE SAVEPOINT identifier
-- 事务回滚到保存点
ROLLBACK TO [SAVEPOINT] identifier
```

ACID特性

原子性（A）

> 事务操作要么都做（提交），要么都不做（回滚）；事务是访问并更新数据库各种数据项的一个程序执行单元，是不可分割的工作单位；通过 *undolog* 来实现回滚操作。*undolog* 记录的是事务每步具体操作，当回滚时，回放事务具体操作的逆运算；
>
> 事务包含的全部操作是一个不可分割的整体，要么全部执行，要么全部不执行；

隔离性（I）

> 描述：各个事务之间互相影响的程度；
>
> 目的：主要规定多个事务访问同一数据资源，各个事务对该数据资源访问的行为，不同的隔离性是应对不同的现象（脏读、不可重复读、幻读）；
>
> 事务的隔离性要求每个读写事务的对象对其他事务的操作对象能相互分离，并发事务之间不会相互影响，设定了不同程度的隔离级别，**通过适度破坏一致性，得以提高性能**；通过 *MVCC* 和 锁来实现；*MVCC* 时多版本并发控制，主要解决一致性非锁定**读**，通过记录和获取行版本，而不是使用锁来限制读操作，从而实现高效并发读性能。锁用来处理并发 *DML* 操作；数据库中提供粒度锁的策略，针对表（聚集索引 B+ 树）、页（聚集索引 B+ 树叶子节点）、行（叶子节点当中某一段记录行）三种粒度加锁；

持久性（D）

> 事务一旦完成，要将数据所做的变更记录下来，包括数据存储和多副本的网络备份；
>
> 事务提交后，事务 *DML* 操作将会持久化（写入 *redolog* 磁盘文件 哪一个页 页偏移值 具体数据）；即使发生宕机等故障，数据库也能将数据恢复。*redolog* 记录的是物理日志；

一致性（C）

> 事务的前后，所有的数据都保持一个一致的状态，不能违反数据的一致性检测（完整性约束检查）；
>
> 一致性指事务将数据库从一种一致性状态转变为下一种一致性的状态，在事务执行前后，数据库完整性约束没有被破坏；一个事务单元需要提交之后才会被其他事务可见。例如：一个表的姓名是唯一键，如果一个事务对姓名进行修改，但是在事务提交或事务回滚后，表中的姓名变得不唯一了，这样就破坏了一致性；一致性由原子性、隔离性以及持久性共同来维护的。

隔离级别

> ISO 和 ANIS SQL 标准制定了四种事务隔离级别的标准，各数据库厂商在正确性和性能之间做了妥协，并没有严格遵循这些标准；MySQL innodb默认支持的隔离级别是 REPEATABLE READ；

READ UNCOMMITTED

> 读未提交；该级别下读不加锁，写加排他锁，写锁在事务提交或回滚后释放锁；

READ COMMITTED

> 读已提交（*RC*）；从该级别后支持 *MVCC* (多版本并发控制)，也就是提供一致性非锁定读；此时读取操作读取历史快照数据；该隔离级别下读取历史版本的最新数据，所以读取的是已提交的数据；

REPEATABLE READ

> 可重复读（*RR*）；该级别下也支持 *MVCC*，此时读取操作读取事务开始时的版本数据；

SERIALIZABLE

> 可串行化；该级别下给读加了共享锁；所以事务都是串行化的执行；此时隔离级别最严苛；

命令

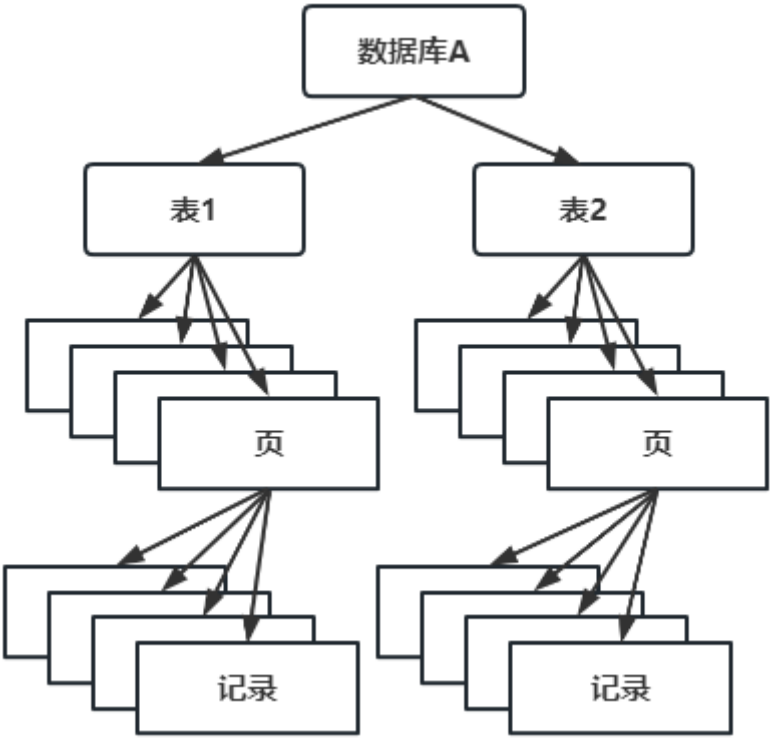
```
-- 设置隔离级别
SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL REPEATABLE READ;
-- 或者采用下面的方式设置隔离级别
SET @@tx_isolation = 'REPEATABLE READ';
SET @@global.tx_isolation = 'REPEATABLE READ';
-- 查看全局隔离级别
SELECT @@global.tx_isolation;
-- 查看当前会话隔离级别
SELECT @@session.tx_isolation;
SELECT @@tx_isolation;

-- 手动给读加 S 锁
SELECT ... LOCK IN SHARE MODE;
-- 手动给读加 X 锁
SELECT ... FOR UPDATE;
-- 查看当前锁信息
SELECT * FROM information_schema.innodb_locks;
```

锁

> 锁机制用于管理对共享资源的并发访问；用来实现事务的隔离级别；

锁类型

> 共享锁和排他锁都是行级锁；MySQL当中事务采用的是粒度锁；针对表（B+树）、页（B+树叶子节点）、行（B+树叶子节点当中某一段记录行）三种粒度加锁；
>
> 意向共享锁和意向排他锁都是表级别的锁；



共享锁（S）

> 事务读操作加的锁；对某一行加锁；
>
> 在 *SERIALIZABLE* 隔离级别下，默认帮读操作加共享锁；
>
> 在 *REPEATABLE READ* 隔离级别下，需手动加共享锁，可解决幻读问题；
>
> 在 *READ COMMITTED* 隔离级别下，没必要加共享锁，采用的是 *MVCC*；
>
> 在 *READ UNCOMMITTED* 隔离级别下，既没有加锁也没有使用 *MVCC*；

排他锁（X）

> 事务删除或更新加的锁；对某一行加锁；
>
> 在4种隔离级别下，都添加了排他锁，事务提交或事务回滚后释放锁；

意向共享锁（IS）

> 对一张表中某几行加的共享锁；

意向排他锁（IX）

> 对一张表中某几行加的排他锁；
>
> 目的：为了告诉其他事务，此时这条表被一个事务在访问；作用：排除表级别读写锁 （全面扫描加锁）；

锁的兼容性

锁	S	X	IS	IX	AI
S	兼容	冲突	兼容	冲突	冲突
X	冲突	冲突	冲突	冲突	冲突
IS	兼容	冲突	兼容	兼容	兼容
IX	冲突	冲突	兼容	兼容	兼容
AI	冲突	冲突	兼容	兼容	冲突

> 由于 *innodb* 支持的是行级别的锁，**意向锁并不会阻塞除了全表扫描以外的任何请求**；
>
> 意向锁之间是互相兼容的；
>
> *IS* 只对排他锁不兼容；
>
> 当想为某一行添加 *S* 锁，先自动为所在的页和表添加意向锁 *IS*，再为该行添加 *S* 锁；
>
> 当想为某一行添加 *X* 锁，先自动为所在的页和表添加意向锁 *IX*，再为该行添加 *X* 锁；
>
> 当事务试图读或写某一条记录时，会先在表上加上意向锁，然后才在要操作的记录上加上读锁或写锁。这样判断表中是否有记录加锁就很简单了，只要看下表上是否有意向锁就行了。意向锁之间是不会产生冲突的，也不和 *AUTO_INC* 表锁冲突，它只会**阻塞表级读锁或表级写锁**，另外，**意向锁也不会和行锁冲突**，**行锁只会和行锁冲突。**

锁算法

Record Lock

> 记录锁，单个行记录上的锁；

Gap Lock（重点）

> 间隙锁，锁定一个范围，但不包含记录本身；全开区间；*REPEATABLE READ* 级别及以上支持间隙锁；
>
> 如果 *REPEATABLE READ* 修改 `innodb_locks_unsafe_for_binlog = 0`，那么隔离级别相当于退化为 *READ COMMITTED*；

```
-- 查看是否支持间隙锁，默认支持，也就是 innodb_locks_unsafe_for_binlog = 0;
SELECT @@innodb_locks_unsafe_for_binlog;
```

Next-Key Lock

> 记录锁+间隙锁，锁定一个范围，并且锁住记录本身；左开右闭区间；

Insert Intention Lock

> 插入意向锁，`insert` 操作的时候产生；在多事务同时写入不同数据至同一索引间隙的时候，并不需要等待其他事务完成，不会发生锁等待。
>
> 假设有一个记录索引包含键值 4 和 7，两个不同的事务分别插入 5 和 6，每个事务都会产生一个加在 `4-7` 之间的插入意向锁，获取在插入行上的排它锁，但是不会被互相锁住，因为数据行并不冲突。

锁兼容

锁	GAP（持有）	Insert Intention（持有）	Record（持有）	Next-key（持有）
GAP（请求）	兼容	兼容	兼容	兼容
Insert Intention（请求）	冲突	兼容	兼容	冲突
Record（请求）	兼容	兼容	冲突	冲突
Next-key（请求）	兼容	兼容	冲突	冲突

> 横向：表示已经持有的锁；纵向：表示正在请求的锁；
>
> 一个事务已经获取了插入意向锁，对其他事务是没有任何影响的；
>
> 一个事务想要获取插入意向锁，如果有其他事务已经加了 *gap lock* 或 *Next-key lock* 则会阻塞；这个是重点，死锁之源；

AUTO-INC Lock（AI锁）

> 自增锁，是一种特殊的表级锁，发生在 *AUTO_INCREMENT* 约束下的插入操作；采用的一种特殊的表锁机制（较低概率造成 *B+* 树分裂）；完成对自增长值插入的 *SQL* 语句后立即释放；在大数据量的插入会影响插入性能，因为另一个事务中的插入会被阻塞；从*MySQL 5.1.22* 开始提供一种轻量级互斥量的自增长实现机制，该机制提高了自增长值插入的性能；

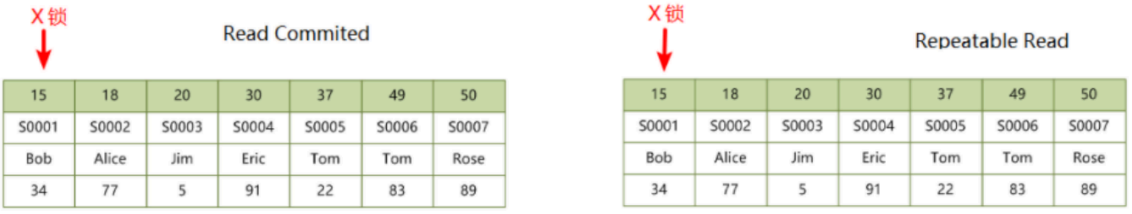
锁的对象

> 行级锁是针对表的索引加锁；索引包括聚集索引和辅助索引；
>
> 表级锁是针对页或表进行加锁；
>
> 重点考虑 InnoDB 在 read committed 和 repeatable read 级别下锁的情况；
>
> 如下图 students 表作为实例，其中 id 为主键，no（学号）为辅助唯一索引，name（姓名）和 age（年龄）为辅助普通索引，score（学分）无索引。

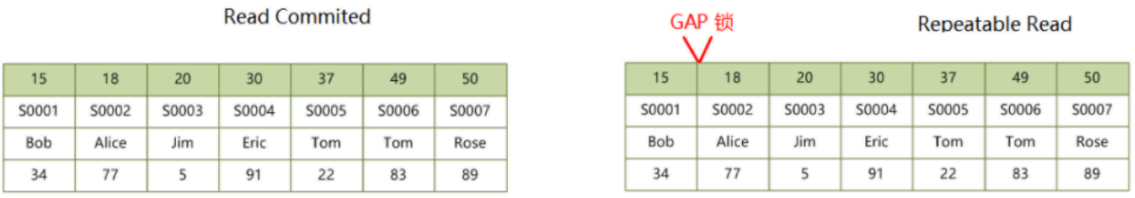
id	no	name	age	score
15	S0001	Bob	25	34
18	S0002	Alice	24	77
20	S0003	Jim	24	5
30	S0004	Eric	23	91
37	S0005	Tom	22	22
49	S0006	Tom	25	83
50	S0007	Rose	23	89

分别讨论

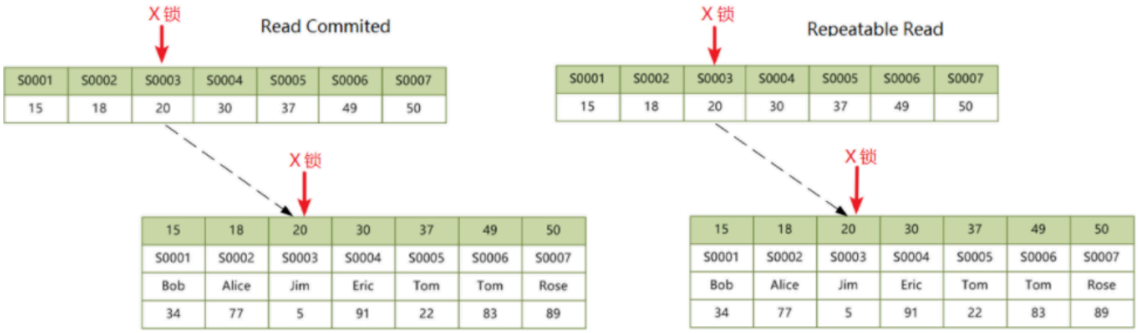
- 聚集索引，查询命中：`UPDATE students SET score = 100 WHERE id = 15;`



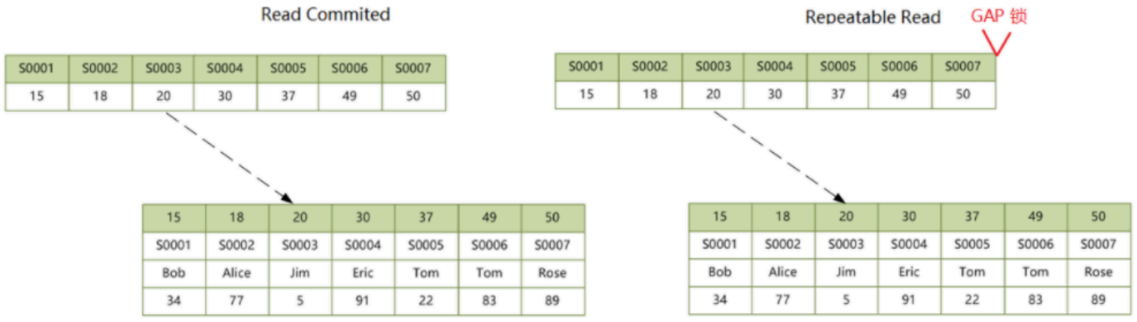
- 聚集索引，查询未命中：`UPDATE students SET score = 100 WHERE id = 16;`



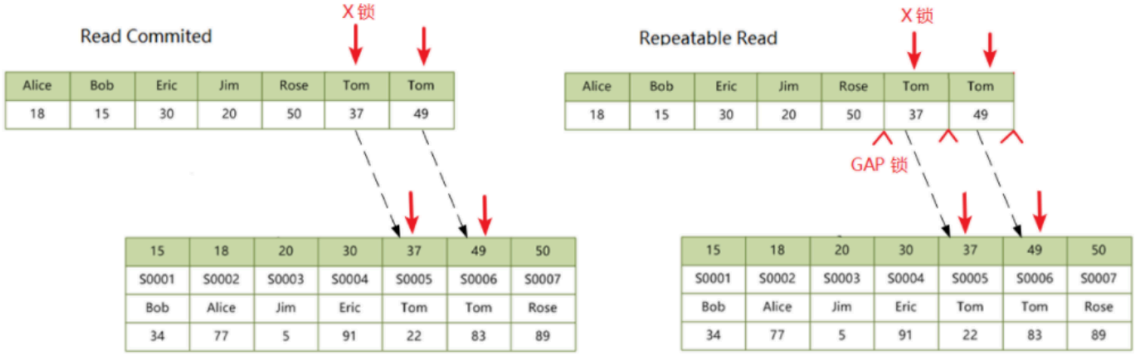
- 辅助唯一索引，查询命中：`UPDATE students SET score = 100 WHERE no = 'S0003';`



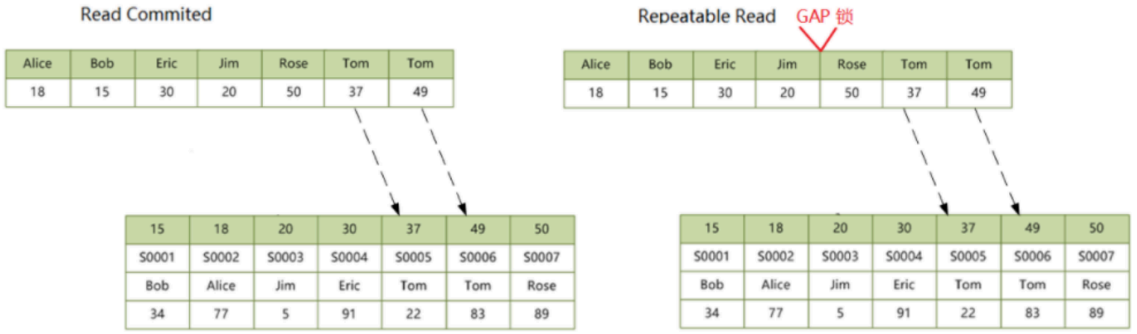
- 辅助唯一索引，查询未命中：`UPDATE students SET score = 100 WHERE no = 'S0008';`



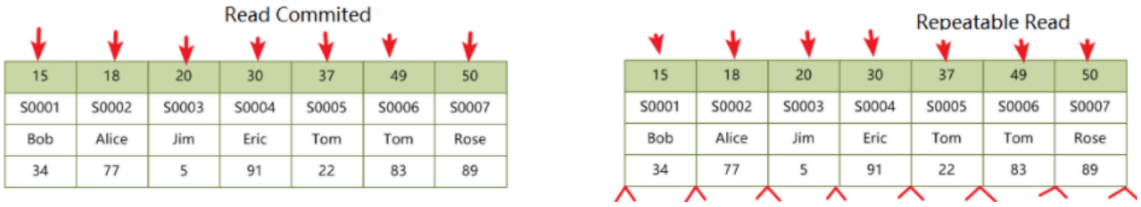
- 辅助非唯一索引，查询命中：`UPDATE students SET score = 100 WHERE name = 'Tom';`



- 辅助非唯一索引，查询未命中：`UPDATE students SET score = 100 WHERE name = 'John';`

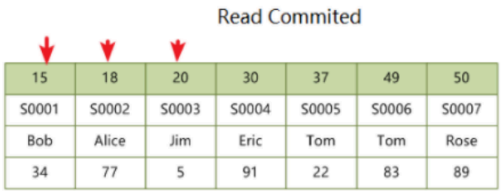


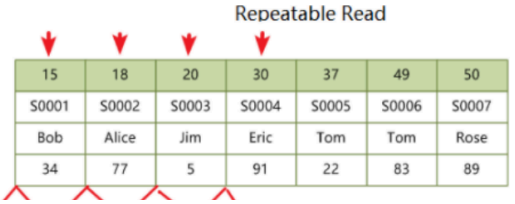
- 无索引：`UPDATE students SET score = 100 WHERE score = 22;`



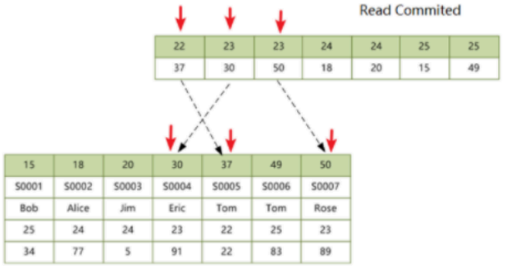
- 聚集索引，范围查询：`UPDATE students SET score = 100 WHERE id <= 20;`

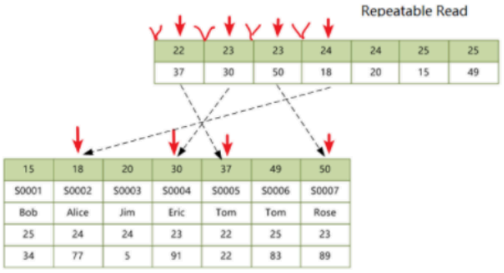
> 特殊情况下，不讨论，有时候加（20，30]



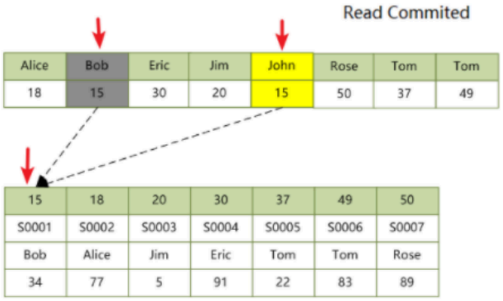


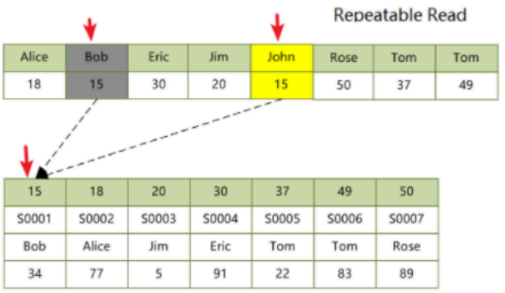
- 辅助索引，范围查询：`UPDATE students SET score = 100 WHERE age <= 23;`





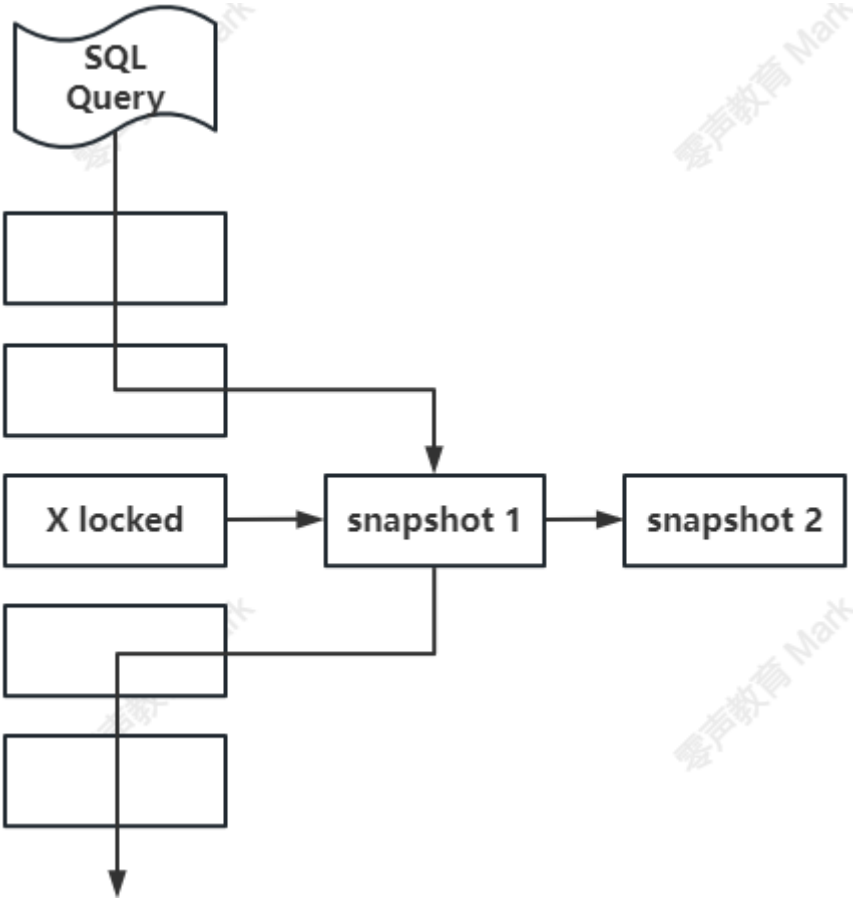
- 修改索引值：`UPDATE students SET name = 'John' WHERE id = 15;`





MVCC

> 多版本并发控制；用来实现一致性的非锁定读；非锁定读是指不需要等待访问的行上X锁的释放；
>
> 在 *read committed* 和 *repeatable read* 下，*innodb* 使用 *MVCC*；然后对于快照数据的定义不同；在 *read committed* 隔离级别下，对于快照数据总是读取被锁定行的最新一份快照数据；而在 *repeatable read* 隔离级别下，对于快照数据总是读取事务开始时的行数据版本；
>
> 思考：为什么读取快照数据不需要上锁?
>
> 因为没有事务需要对历史的数据进行修改操作；



read view

> 在 *read committed* 和 *read repeatable* 隔离级别下，*MVCC* 采用 *read view* 来实现的，它们的区别在于创建 *read view* 时机不同：
>
> - *read committed* 隔离级别会在事务中每个 `select` 都会生成一个新的 *read view*，也意味着在同一个事务多次读取同一条数据可能出现数据不一致；因为在多次读取期间可能有其他事务修改了该条记录，并提交了；
> - *read repeatable* 隔离级别是启动事务时生成一个 *read view*，在整个事务读取数据都才使用这个 *read view*，这样保证了在事务期间读到的数据都是事务启动前的记录；

构成

- `m_ids`：创建 *read view* 时，当前数据库活跃事务（开启未提交的事务）的事务 id 列表；
- `min_trx_id`：创建 *read view* 时，`m_ids` 中的最小事务 *id*；
- `max_trx_id`：创建 *read view* 时，当前数据库将为下一个事务分配的事务 id；并不一定是 `m_ids` 中的最大事务 *id*；
- `creator_trx_id`：创建 *read view* 所在事务的 *id*；

聚集索引隐藏列

- `trx_id`：当某个事务对某条聚集索引记录进行修改时，将会把当前事务的 *id* 赋值给 `trx_id`；
- `roll_pointer`：当某个事务对某条聚集索引记录进行修改时，会将上一个版本的记录写到 undo log，然后通过 `roll_pointer` 指向旧版本记录，通过它可以找到修改前的记录；

事务状态

- 已提交的事务
- 已启动未提交的事务
- 还没开始的事务

流程

- `trx_id < min_trx_id`；说明该记录在创建 *read_view* 之前已经提交，所以对当前事务可见；
- `trx_id >= max_trx_id`；说明该记录是在创建 *read_view* 之后启动事务生成的，所以对当前事务不可见；
- `min_trx_id <= trx_id < max_trx_id`；此时需要判断是否在 `m_ids` 列表中；
  - 在列表中；生成该版本记录的事务仍处于活跃状态，该版本记录对当前事务不可见；
  - 不在列表中；生成该版本记录的事务已经提交，该版本记录对当前事务可见；

redo

> *redo* 日志用来实现事务的持久性；内存中包含 *redo log buffer*，*磁盘*中包含 *redo log file*；
>
> 当事务提交时，必须先将该事务的所有日志写入到重做日志文件进行持久化，待事务的`commit` 操作完成才完成了事务的提交；
>
> *redo log* **顺序写**，记录的是对每个页的修改（页、页偏移量、以及修改的内容）；在数据库运行时不需要对 *redo log* 的文件进行读取操作；只有发生宕机的时候，才会拿*redo log* 进行恢复；

undo

> *undo* 日志用来帮助事务回滚以及 *MVCC* 的功能；存储在共享表空间中；*undo* 是逻辑日志，回滚时将数据库逻辑地恢复到原来的样子，根据 *undo log* 的记录，做之前的逆运算；比如事务中有 `insert` 操作，那么执行 `delete` 操作；对于 `update` 操作执行相反的 `update` 操作；
>
> 同时 *undo* 日志记录行的版本信息，用于处理 *MVCC* 功能；

并发读异常

脏读

> 事务（A）可以读到另外一个事务（B）中未提交的数据；也就是事务A读到脏数据；在读写分离的场景下，可以将 *slave* 节点设置为 *READ UNCOMMITTED*；此时脏读不影响，在 *slave* 上查询并不需要特别精准的返回值。

seq	session A	session B
1	`SET @@tx_isolation='READ UNCOMMITTED';`	`SET @@tx_isolation='READ UNCOMMITTED';`
2	`BEGIN;`	
3	`UPDATE account_t SET money = money - 100 WHERE name = 'A';`	
4		`BEGIN;`

seq	session A	session B
5		`SELECT money FROM account_t WHERE name = 'A';`
6		`SELECT money FROM account_t WHERE name = 'B';`
7	`UPDATE account_t SET money = money - 100 WHERE name = 'B';`	
8	`COMMIT`	`COMMIT`

不可重复读

> 事务（A) 可以读到另外一个事务（B）中提交的数据；通常发生在一个事务中两次读到的数据是不一样的情况；不可重复读在隔离级别 *READ COMMITTED* 存在。一般而言，不可重复读的问题是可以接受的，因为读到已经提交的数据，一般不会带来很大的问题，所以很多厂商（如*Oracle*、*SQL Server*）默认隔离级别就是 *READ COMMITTED*；

seq	session A	session B
1	`SET @@tx_isolation='READ COMMITTED';`	`SET @@tx_isolation='READ COMMITTED';`
2	`BEGIN;`	`BEGIN;`
3		`SELECT money FROM account_t WHERE name = 'A';`
4	`UPDATE account_t SET money = money - 100 WHERE name = 'A';`	
5	`COMMIT;`	`SELECT money FROM account_t WHERE name = 'A';`
6		`COMMIT;`

幻读

> 两次读取**同一个范围内的记录**得到的结果集不一样；快照读和当前读不一致；例如：以 *name* 为唯一键的表，一个事务中查询 `select * from t where name = 'mark';` 不存在，接下来 `insert into t(name) values ('mark');` 出现错误，此时另外一个事务也执行了`insert` 操作；幻读在隔离级别 *REPEATABLE READ* 及以下存在；但是可以在 *REPEATABLE READ* 级别下通过**读加锁**（使用 *next-key locking*）解决；

seq	session A	session B
1	`SET @@tx_isolation='REPEATABLE READ';`	`SET @@tx_isolation='REPEATABLE READ';`
2	`BEGIN;`	`BEGIN;`

seq	session A	session B
3		`SELECT * FROM account_t WHERE id >= 2;`
4	`INSERT INTO account_t(id,name,money) VALUES (4,'D',1000);`	
5	`COMMIT;`	
6		`INSERT INTO account_t(id,name,money) VALUES (4,'D',1000);`
7		`COMMIT;`

解决

seq	session A	session B
1	`SET @@tx_isolation='REPEATABLE READ';`	`SET @@tx_isolation='REPEATABLE READ';`
2	`BEGIN;`	`BEGIN;`
3		`SELECT * FROM account_t WHERE id >= 2 lock in share mode;`
4	`INSERT INTO account_t(id,name,money) VALUES (4,'D',1000);`	
5	`COMMIT;`	
6		`SELECT * FROM account_t WHERE id >= 2;`
7		`COMMIT;`

丢失更新

> 脏读、不可重复读、幻读都是一个事务写，一个事务读，由于一个事务的写导致另一个事务读到了不该读的数据；丢失更新是两个事务都是写；丢失更新分为**提交覆盖**和**回滚覆盖**；回滚覆盖数据库拒绝不可能产生，重点关注提交覆盖；

seq	session A	session B
1	`SET @@tx_isolation='REPEATABLE READ';`	`SET @@tx_isolation='REPEATABLE READ';`
2	`BEGIN;`	`BEGIN;`

seq	session A	session B
3	`SELECT money FROM account_t WHERE name = 'A' ;`	
4		`SELECT money FROM account_t WHERE name = 'A';`
5		`UPDATE account_t SET money = 1000+100 WHERE name = 'A';`
6		`COMMIT;`
7	`UPDATE account_t SET money = 1000-100 WHERE name = 'A';`	
8	`COMMIT;`	

区别

> 脏读和不可重复读的区别在于，脏读是读取了另一个事务未提交的数据，而不可重复读是读取了另一个事务提交之后的修改；本质上都是其他事务的修改影响了本事务的读取；
>
> 不可重复读和幻读比较类似；不可重复读是两次读取**同一条记录**，得到不一样的结果；而幻读是两次读取**同一个范围内的记录**得到的结果集不一样（可能不同个数，也可能相同个数内容不一样，比如x一行后又添加新行）；不可重复读是因为其他事务进行了 `update` 操作，幻读是因为其他事务进行了 `insert` 或者 `delete` 操作。

隔离级别下并发读异常

隔离级别	回滚覆盖	脏读	不可重复读	幻读	提交覆盖
READ UNCOMMITTED	*no*	*yes*	*yes*	*yes*	*yes*
READ COMMITTED	*no*	*no*	*yes*	*yes*	*yes*
REPEATABLE READ	*no*	*no*	*no*	*yes*（手动加锁）	*yes*（手动加锁）
SERIALIZABLE	*no*	*no*	*no*	*no*	*no*

并发死锁

> 死锁：两个或两个以上的事务在执行过程中，因争夺锁资源而造成的一种互相等待的现象；*MySQL* 中采用 *wait-for graph* （等待图-采用非递归深度优先的图算法实现）的方式来进行死锁检测；
>
> 异常报错：`deadlock found when trying to get lock`;

相反加锁顺序死锁

> 不同表的加锁顺序相反或者相同表不同行加锁顺序相反造成死锁；其中相同表不同行加锁顺序相反造成死锁有很多变种，其中容易忽略的是给辅助索引行加锁的时候，同时会给聚集索引行加锁；同时还可能出现在外键索引时，给父表加锁，同时隐含给子表加锁；触发器同样如此，这些都需要视情况分析；
>
> 调整加锁顺序；

锁冲突死锁

> *innodb* 在 *RR* 隔离级别下，最常见的是插入意向锁与 *gap* 锁冲突造成死锁；主要原理为：一个事务想要获取插入意向锁，如果有其他事务已经加了 *gap lock* 或 *Next-key lock* 则会阻塞；

案例

> 看 `并发死锁问题.sql`

查看死锁

系统表

```
-- 开启标准监控
CREATE TABLE innodb_monitor (a INT) ENGINE=INNODB;

-- 关闭标准监控
DROP TABLE innodb_monitor;

-- 开启锁监控
CREATE TABLE innodb_lock_monitor (a INT) ENGINE=INNODB;

-- 关闭锁监控
DROP TABLE innodb_lock_monitor
```

系统参数

```
-- 开启标准监控
set GLOBAL innodb_status_output=ON;

-- 关闭标准监控
set GLOBAL innodb_status_output=OFF;

-- 开启锁监控
set GLOBAL innodb_status_output_locks=ON;

-- 关闭锁监控
set GLOBAL innodb_status_output_locks=OFF;

-- 将死锁信息记录在错误日志中
set GLOBAL innodb_print_all_deadlocks=ON;
```

命令

```
-- 查看事务
select * from information_schema.INNODB_TRX;
-- 查看锁
select * from information_schema.INNODB_LOCKS;
-- 查看锁等待
select * from information_schema.INNODB_LOCK_WAITS;
```

死锁解决

> 对于顺序相反型，调整执行顺序；
>
> 对于锁冲突型，更换语句或者降低隔离级别；

如何避免死锁

- 尽可能以相同顺序来访问索引记录和表；
- 如果能确定幻读和不可重复读对应用影响不大，考虑将隔离级别降低为 *RC*；
- 添加合理的索引，不走索引将会为每一行记录加锁，死锁概率非常大；
- 尽量在一个事务中只锁定所需要的资源，减小死锁概率；
- 避免大事务，将大事务分拆成多个小事务；大事务占用资源多，耗时长，冲突概率变高；
- 避免同一时间点运行多个对同一表进行读写的概率；