

数据库部分：

数据库三大范式：

第一范式：每一列属性都是不可再分的属性值，确保每一列的原子性

第二范式：属性完全依赖于主键。每一行的数据只能与其中一列相关，即一行数据只做一件事。只要数据列中出现数据重复，就要把表拆分开来。

第三范式：属性不依赖于其它非主属性，属性直接依赖于主键，即每个属性都跟主键有直接关系而不是间接关系。

1.B+树以及为什么采用 B+树索引

2.数据库索引的使用：

主键索引：在创建表时，InnoDB 存储引擎默认会创建一个主键索引，也就是**聚簇索引**，其它索引都属于二级索引。

InnoDB 存储引擎：B+ 树索引的叶子节点保存**数据本身**；

MyISAM 存储引擎：B+ 树索引的叶子节点保存**数据的物理地址**；

InnoDB 存储引擎根据索引类型不同，分为聚簇索引和二级索引。它们区别在于，聚簇索引的叶子节点存放的是实际数据，所有完整的用户数据都存放在聚簇索引的叶子节点，而二级索引的叶子节点存放的是主键值，而不是实际数据。

在我们使用「主键索引」字段作为条件查询的时候，如果要查询的数据都在「聚簇索引」的叶子节点里，那么就会在「聚簇索引」中的 B+ 树检索到对应的叶子节点，然后**直接读取要查询的数据**。如下面这条语句：

```
// id 字段为主键索引
select * from t_user where id=1;
```

在我们使用「二级索引」字段作为条件查询的时候，如果要查询的数据都在「聚簇索引」的叶子节点里，那么需要检索两颗 B+树：

- 先在「二级索引」的 B+ 树找到对应的叶子节点，获取主键值；
- 然后用上一步获取的主键值，在「聚簇索引」中的 B+ 树检索到对应的叶子节点，然后获取要查询的数据。

这个过程叫做**回表**。

```
// name 字段为二级索引
select * from t_user where name="林某";
```

这里需要查询两个 B+ 树，所以就是二级索引，除此之外还有一种情况。

首先说明一下，mysql 官网并没有看到覆盖索引的准确描述，但如果通过搜索一次 B+ 树，也就是二级索引就搜索到了记录，那就是覆盖索引。

在我们使用「二级索引」字段作为条件查询的时候，如果要查询的数据在「二级索引」的叶子节点，那么只需要在「二级索引」的 B+ 树找到对应的叶子节点，然后读取要查询的数据，这个过程叫做**覆盖索引**。如下面这条语句：

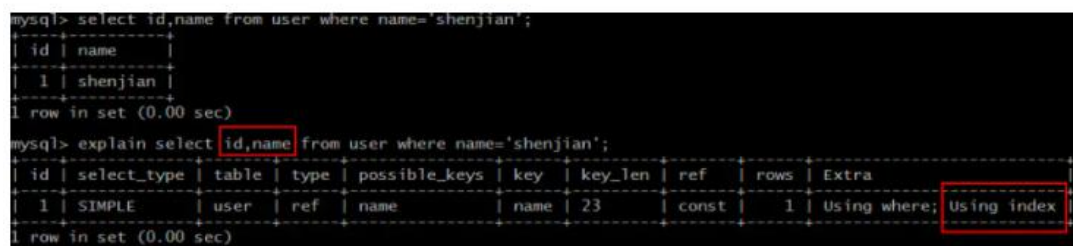
```
// name 字段为二级索引
select id from t_user where name="林某";
```

覆盖索引实现：！！！！

将被查询的字段，建立到联合索引里去。

```
create table user (
  id int primary key,
  name varchar(20),
  sex varchar(5),
  index(name)
)engine=innodb;
```

第一个 SQL 语句：



```
mysql> select id,name from user where name='shenjian';
+----+-----+
| id | name |
+----+-----+
| 1  | shenjian |
+----+-----+
1 row in set (0.00 sec)

mysql> explain select id,name from user where name='shenjian';
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE     | user  | ref  | name          | name | 23      | const | 1    | Using where; Using index |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

```
select id,name from user where name='shenjian';
```

能够命中 name 索引,索引叶子节点存储了主键 id,通过 name 的索引树即可获取 id 和 name,无需回表,符合索引覆盖,效率较高。

画外音, Extra: Using index。

第二个 SQL 语句:

```
mysql> select id,name,sex from user where name='shenjian';
+----+-----+-----+
| id | name  | sex   |
+----+-----+-----+
| 1  | shenjian | no    |
+----+-----+-----+
1 row in set (0.00 sec)

mysql> explain select id,name,sex from user where name='shenjian';
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE      | user  | ref  | name          | name | 23      | const | 1    | Using index condition |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

```
select id,name,sex* from user where name='shenjian';*
```

能够命中 name 索引,索引叶子节点存储了主键 id,但 sex 字段必须回表查询才能获取到,不符合索引覆盖,需要再次通过 id 值扫码聚集索引获取 sex 字段,效率会降低。

画外音, Extra: Using index condition。

如果把(name)单列索引升级为联合索引(name, sex)就不同了。

```
create table user (
```

```
id int primary key,
```

```
name varchar(20),
```

```
sex varchar(5),
```

```
index(name, sex)
```

```
)engine=innodb;
```

```
mysql> explain select id,name from user where name='shenjian';
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE      | user  | ref  | name          | name | 23      | const | 1    | Using where; Using index |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.01 sec)

mysql> explain select id,name,sex from user where name='shenjian';
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE      | user  | ref  | name          | name | 23      | const | 1    | Using where; Using index |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

可以看到:

```
select id,name ... where name='shenjian';
```

```
select id,name,sex* ... where name='shenjian';*
```

都能够命中索引覆盖，无需回表。

画外音，*Extra: Using index。*

3.联合索引非最左匹配

对主键字段建立的索引叫做聚簇索引，对普通字段建立的索引叫做二级索引。那么多个普通字段组合在一起创建的索引就叫做联合索引，也叫组合索引。

创建联合索引时，我们需要注意创建时的顺序问题，因为联合索引 (a, b, c) 和 (c, b, a) 在使用的时候会存在差别。

联合索引要能正确使用需要遵循最左匹配原则，也就是按照最左优先的方式进行索引的匹配。

比如，如果创建了一个 (a, b, c) 联合索引，如果查询条件是以下这几种，就可以匹配上联合索引：

- where a=1;
- where a=1 and b=2 and c=3;
- where a=1 and b=2;

需要注意的是，因为有查询优化器，所以 a 字段在 where 子句的顺序并不重要。

但是，如果查询条件是以下这几种，因为不符合最左匹配原则，所以就无法匹配上联合索引，联合索引就会失效：

- where b=2;
- where c=3;
- where b=2 and c=3;

有一个比较特殊的查询条件：where a = 1 and c = 3 ，符合最左匹配吗？

这种其实严格意义上来说是属于索引截断，不同版本处理方式也不一样。

MySQL 5.5 的话，前面 a 会走索引，在联合索引找到主键值后，开始回表，到主键索引读取数据行，然后再比对 c 字段的值。

从 MySQL5.6 之后，有一个**索引下推功能**，可以在索引遍历过程中，对索引中包含的字段先做判断，直接过滤掉不满足条件的记录，减少回表次数。

大概原理是：截断的字段会被下推到存储引擎层进行条件判断（因为 c 字段的值是在 (a, b, c) 联合索引里的），然后过滤出符合条件的数据后再返回给 Server 层。由于在引擎层就过滤掉大量的数据，无需再回表读取数据来进行判断，减少回表次数，从而提升了性能。

比如下面这条 `where a = 1 and c = 0` 语句，我们可以从执行计划中的 `Extra=Using index condition` 使用了索引下推功能。

可以参考这个文档：<https://www.jianshu.com/p/9b3406bcb199> 里面还包含 MySQL 查询优化的内容

3.事务篇

事务的特性：

事务是由 MySQL 的引擎来实现的，我们常见的 InnoDB 引擎它是支持事务的。

不过并不是所有的引擎都能支持事务，比如 MySQL 原生的 MyISAM 引擎就不支持事务，也正是这样，所以大多数 MySQL 的引擎都是用 InnoDB。

事务看起来感觉简单，但是要实现事务必须要遵守 4 个特性，分别如下：

- **原子性 (*Atomicity*)**：一个事务中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节，而且事务在执行过程中发生错误，会被回滚到事务开始前的状态，就像这个事务从来没有执行过一样；
- **一致性 (*Consistency*)**：数据库的完整性不会因为事务的执行而受到破坏，比如表中有一个字段为姓名，它有唯一约束，也就是表中姓名不能重复，如果一个事务对姓名字段进行了修改，但是在事务提交后，表中的姓名变得非唯一性了，这就破坏了事务的一致性要求，这时数据库就要撤销该事务，返回初始化的状态。
- **隔离性 (*Isolation*)**：数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。
- **持久性 (*Durability*)**：事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

InnoDB 引擎通过什么技术来保证事务的这四个特性的呢？

- 原子性和持久性是通过 redo log （重做日志）来保证的；
- 一致性是通过 undo log （回滚日志）来保证的；
- 隔离性是通过 MVCC（多版本并发控制）或锁机制来保证的；

脏读

如果一个事务「读到」了另一个「未提交事务修改过的数据」，就意味着发生了「脏读」现象。

不可重复读

在一个事务内多次读取同一个数据，如果出现前后两次读到的数据不一样的情况，就意味着发生了「不可重复读」现象。

幻读

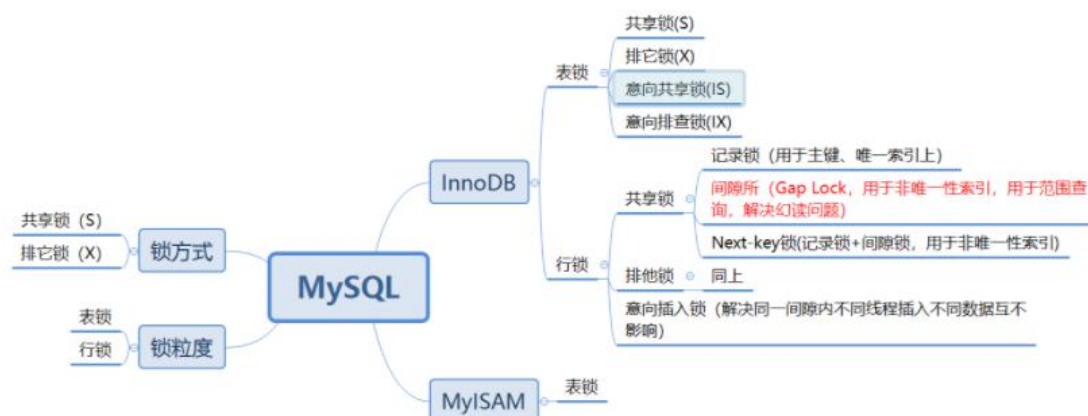
在一个事务内多次查询某个符合查询条件的「记录数量」，如果出现前后两次查询到的记录数量不一样的情况，就意味着发生了「幻读」现象。

- 脏读：读到其他事务未提交的数据；
- 不可重复读：前后读取的数据不一致；
- 幻读：前后读取的记录数量不一致。

SQL 标准提出了四种隔离级别来规避这些现象，隔离级别越高，性能效率就越低，这四个隔离级别如下：

- **读未提交 (*read uncommitted*)**，指一个事务还没提交时，它做的变更就能被其他事务看到；
- **读提交 (*read committed*)**，指一个事务提交之后，它做的变更才能被其他事务看到；
- **可重复读 (*repeatable read*)**，指一个事务执行过程中看到的数据，一直跟这个事务启动时看到的数据是一致的，**MySQL InnoDB 引擎的默认隔离级别**；
- **串行化 (*serializable*)**；会对记录加上读写锁，在多个事务对这条记录进行读写操作时，如果发生了读写冲突的时候，后访问的事务必须等前一个事务执行完成，才能继续执行；

4. 锁篇



属于分支 **innodb->行锁->共享锁**:

InnoDB 引擎是支持行级锁的, 而 MyISAM 引擎并不支持行级锁。

行级锁的类型主要有三类:

- Record Lock, 记录锁, 也就是仅仅把一条记录锁上;
- Gap Lock, 间隙锁, 锁定一个范围, 但是不包含记录本身;
- Next-Key Lock: Record Lock + Gap Lock 的组合, 锁定一个范围, 并且锁定记录本身。

乐观锁

用数据版本 (Version) 记录机制实现, 这是乐观锁最常用的一种实现方式。何谓数据版本? 即为数据增加一个版本标识, 一般是通过为数据库表增加一个数字类型的 “version” 字段来实现。当读取数据时, 将 version 字段的值一同读出, 数据每更新一次, 对此 version 值加 1。当我们提交更新的时候, 判断数据库表对应记录的当前版本信息与第一次取出来的 version 值进行对比, 如果数据库表当前版本号与第一次取出来的 version 值相等, 则予以更新, 否则认为是过期数据。

悲观锁

在进行每次操作时都要通过获取锁才能进行对相同数据的操作, 这点跟 java 中 synchronized 很相似, 共

享锁 (读锁) 和排它锁 (写锁) 是悲观锁的不同的实现

共享锁 (读锁)

共享锁又叫做读锁, 所有的事务只能对其进行读操作不能写操作, 加上共享锁后在事务结束之前 其他事务只能再加共享锁, 除此之外其他任何类型的锁都不能再加了。

排它锁 (写锁)

若某个事物对某一行加上了排他锁, 只能这个事务对其进行读写, 在此事务结束之前, 其他事务 不能对其进行加任何锁, 其他进程可以读取, 不能进行写操作, 需等待其释放。

表级锁

innodb 的行锁是在有索引的情况下, 没有索引的表是锁定全表的

行级锁

行锁又分共享锁和排他锁, 由字面意思理解, 就是给某一行加上锁, 也就是一条记录加上锁。
注意: 行级锁都是基于索引的, 如果一条 SQL 语句用不到索引是不会使用行级锁的, 会使用表级锁。

意向锁

意向共享锁 (IS): 表示事务准备给数据行加入共享锁, 也就是说一个数据行加共享锁前必须先取得该表的 IS 锁

意向排他锁 (IX): 类似上面, 表示事务准备给数据行加入排他锁, 说明事务在一个数据行加排他锁前必须先取得该表的 IX 锁。

意向锁是表级锁

事务要获取表 A 某些行的 S 锁必须要获取表 A 的 IS 锁

事务要获取表 A 某些行的 X 锁必须要获取表 A 的 IX 锁

意向共享锁和意向独占锁是表级锁, 不会和行级的共享锁和独占锁发生冲突, 而且意向锁之间也不会发生冲突, 只会和共享表锁 (`lock tables ... read`) 和独占表锁 (`lock tables ... write`) 发生冲突。

表锁和行锁是满足读读共享、读写互斥、写写互斥的。

如果没有「意向锁」, 那么加「独占表锁」时, 就需要遍历表里所有记录, 查看是否有记录存在独占锁, 这样效率会很慢。

那么有了「意向锁」, 由于在对记录加独占锁前, 先会加上表级别的意向独占锁, 那么在加「独占表锁」时, 直接查该表是否有意向独占锁, 如果有就意味着表里已经有记录被加了独占锁, 这样就不用去遍历表里的记录。

所以, 意向锁的目的是为了快速判断表里是否有记录被加锁。

| MVCC解决的问题是什么?

数据库并发场景有三种, 分别为:

- 1、读读: 不存在任何问题, 也不需要并发控制
- 2、读写: 有线程安全问题, 可能会造成事务隔离性问题, 可能遇到脏读、幻读、不可重复读
- 3、写写: 有线程安全问题, 可能存在更新丢失问题

MVCC是一种用来解决读写冲突的无锁并发控制, 也就是为事务分配单项增长的时间戳, 为每个修改保存一个版本, 版本与事务时间戳关联, 读操作只读该事务开始前的数据库的快照, 所以MVCC可以为数据库解决一下问题:

- 1、在并发读写数据库时, 可以做到在读操作时不用阻塞写操作, 写操作也不用阻塞读操作, 提高了数据库并发读写的性能
- 2、解决脏读、幻读、不可重复读等事务隔离问题, 但是不能解决更新丢失问题

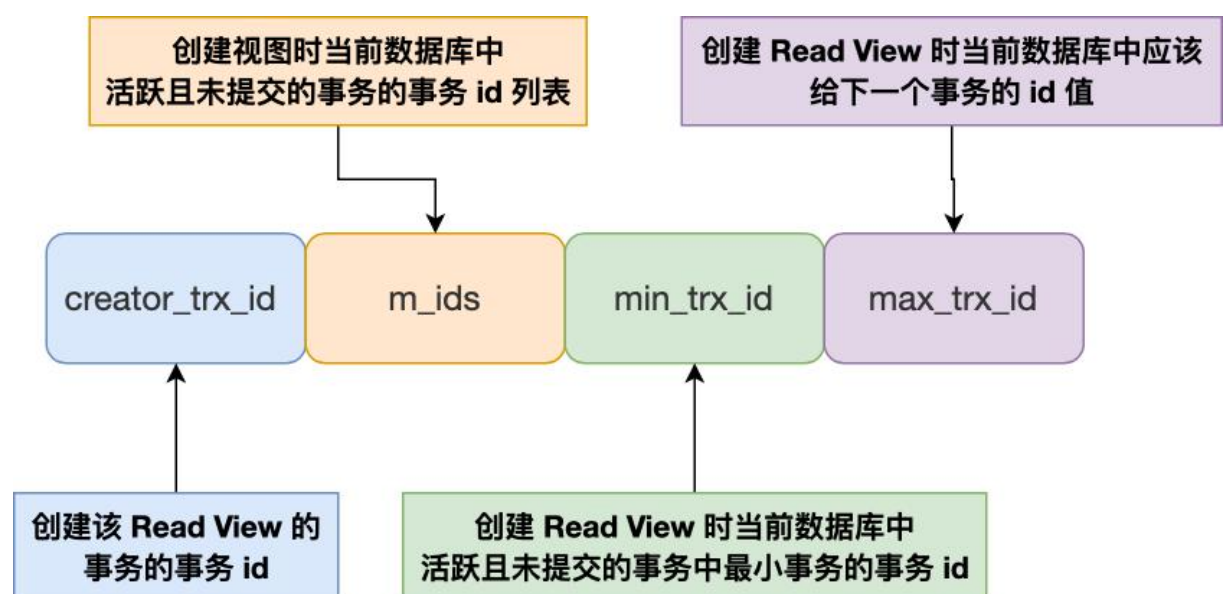
可重复读隔离级别是如何实现的？

「可重复读」隔离级别是启动事务时生成一个 Read View，然后整个事务期间都在用这个 Read View。

想要知道可重复读隔离级别是如何实现的，我们需要了解两个知识：

- Read View 中四个字段作用；
- 聚族索引记录中两个跟事务有关的隐藏列；

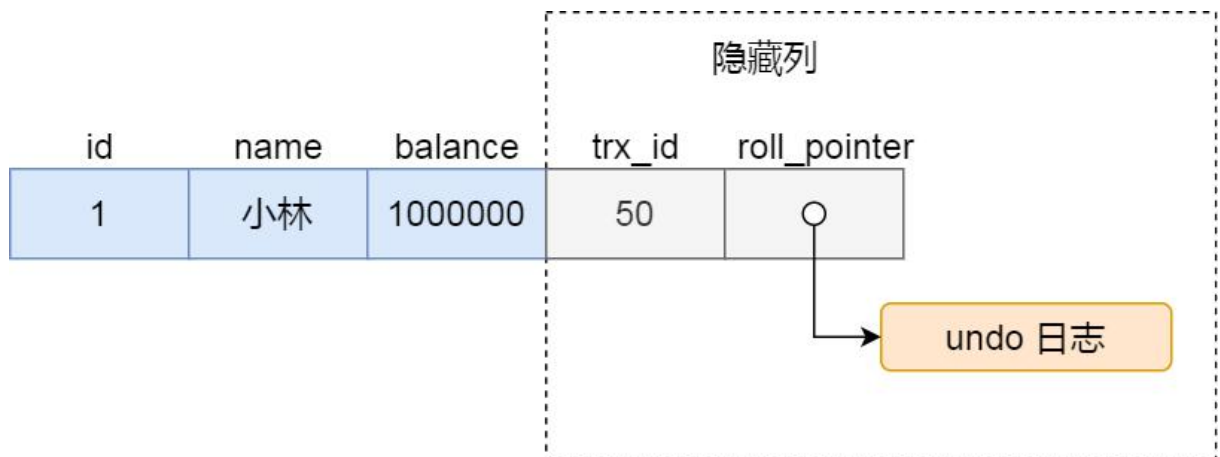
那 Read View 到底是个什么东西？



Read View 有四个重要的字段：

- **m_ids**：指的是创建 Read View 时当前数据库中**活跃且未提交的事务的事务 id 列表**，注意是一个列表。
- **min_trx_id**：指的是创建 Read View 时当前数据库中**活跃且未提交的事务中最小事务的事务 id**，也就是 m_ids 的最小值。
- **max_trx_id**：这个并不是 m_ids 的最大值，而是**创建 Read View 时当前数据库中应该给下一个事务的 id 值**；
- **creator_trx_id**：指的是**创建该 Read View 的事务的事务 id**。

知道了 Read View 的字段，我们还需要了解聚族索引记录中的两个隐藏列，假设在账户余额表插入一条小林余额为 100 万的记录，然后我把这两个隐藏列也画出来，该记录的整个示意图如下：

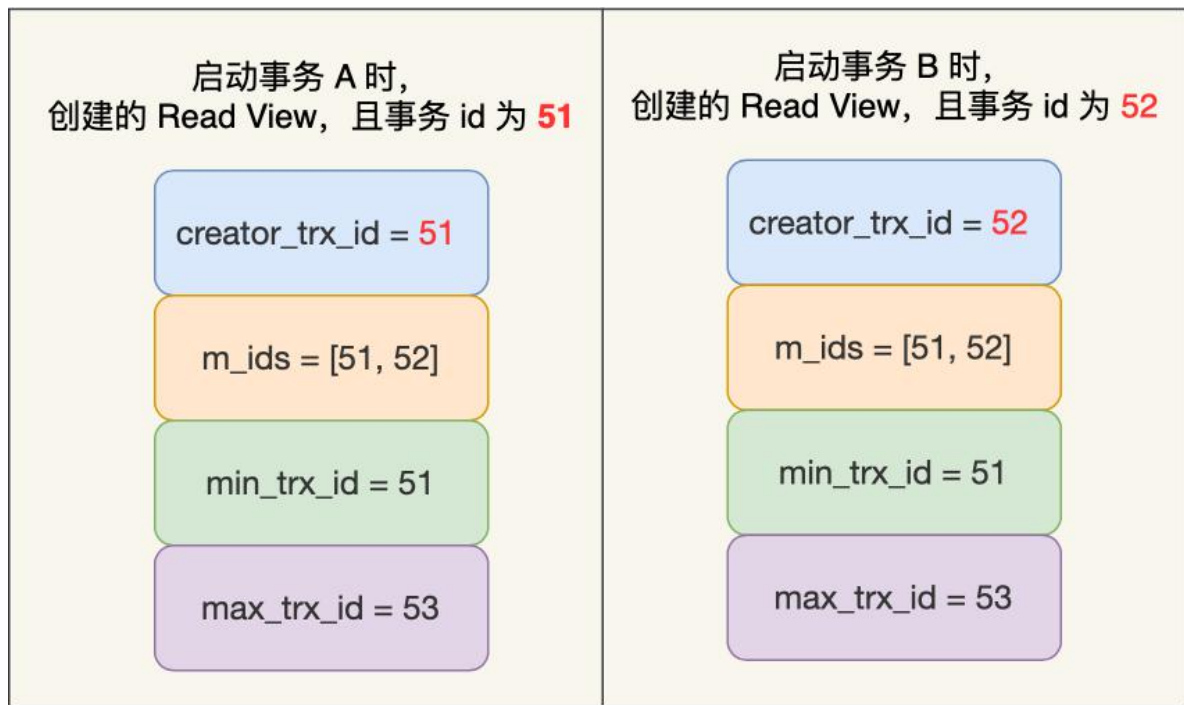


对于使用 InnoDB 存储引擎的数据库表，它的聚族索引记录中都包含下面两个隐藏列：

- `trx_id`，当一个事务对某条聚族索引记录进行改动时，就会把该事务的事务 `id` 记录在 `trx_id` 隐藏列里；
- `roll_pointer`，每次对某条聚族索引记录进行改动时，都会把旧版本的记录写入到 undo 日志中，然后这个隐藏列是个指针，指向每一个旧版本记录，于是就可以通过它找到修改前的记录。

了解完这两个知识点后，就可以跟大家说说可重复读隔离级别是如何实现的。

假设事务 A 和 事务 B 差不多同一时刻启动，那这两个事务创建的 Read View 如下：



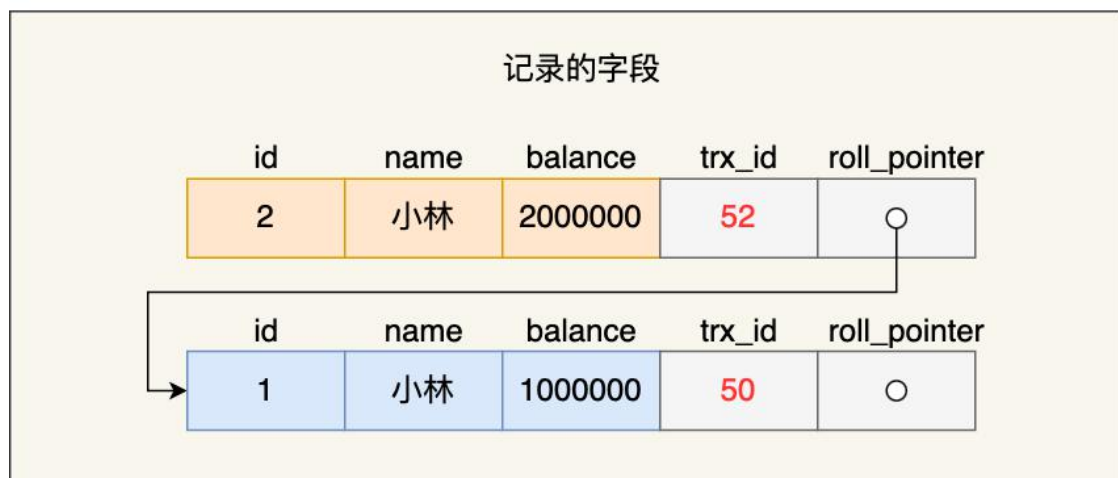
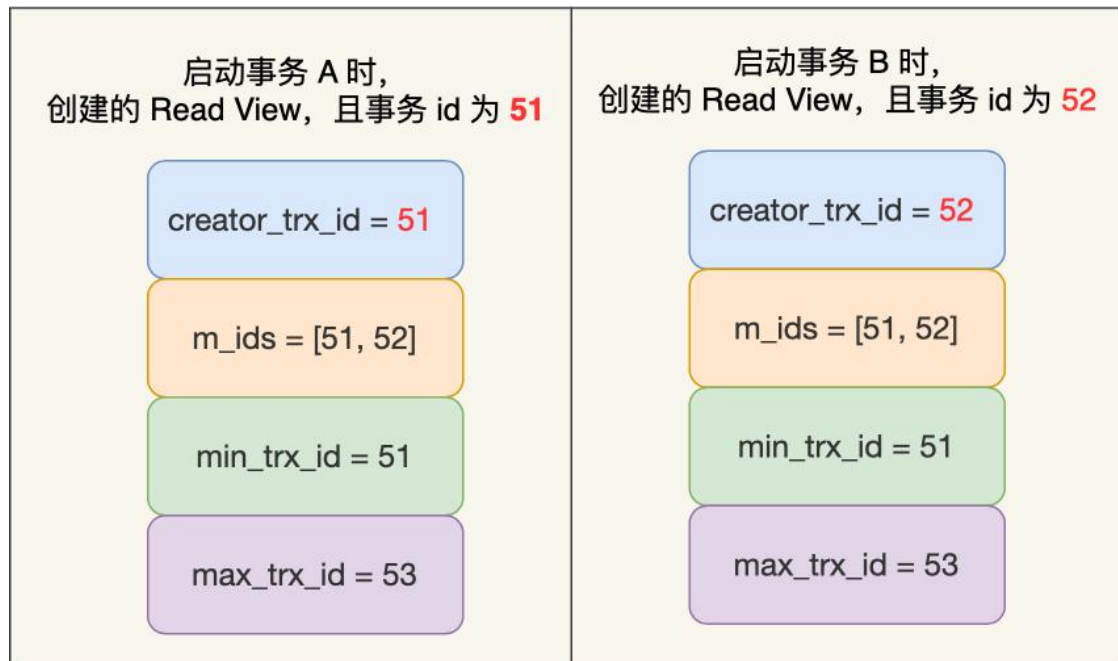
| 记录的字段 | | | | |
|-------|------|---------|--------|--------------|
| id | name | balance | trx_id | roll_pointer |
| 1 | 小林 | 1000000 | 50 | ○ |

事务 A 和 事务 B 的 Read View 具体内容如下：

- 在事务 A 的 Read View 中，它的事务 id 是 51，由于与事务 B 同时启动，所以此时活跃的事务的事务 id 列表是 51 和 52，活跃的事务 id 中最小的事务 id 是事务 A 本身，下一个事务 id 应该是 53。
- 在事务 B 的 Read View 中，它的事务 id 是 52，由于与事务 A 同时启动，所以此时活跃的事务的事务 id 列表是 51 和 52，活跃的事务 id 中最小的事务 id 是事务 A，下一个事务 id 应该是 53。

然后让事务 A 去读账户余额为 100 万的记录，在找到记录后，它会先看这条记录的 trx_id，此时发现 trx_id 为 50，通过和事务 A 的 Read View 的 m_ids 字段发现，该记录的事务 id 并不在活跃事务的列表中，并且小于事务 A 的事务 id，这意味着，这条记录的事务早就在事务 A 前提交过了，所以该记录对事务 A 可见，也就是事务 A 可以获取到这条记录。

接着，事务 B 通过 update 语句将这条记录修改了，将小林的余额改成 200 万，这时 MySQL 会记录相应的 undo log，并以链表的方式串联起来，形成[版本链](#)，如下图：



你可以在上图的「记录字段」看到，由于事务 B 修改了该记录，以前的记录就变成旧版本记录了，于是最新记录和旧版本记录通过链表的方式串起来，而且最新记录的 `trx_id` 是事务 B 的事务 id。

然后如果事务 A 再次读取该记录，发现这条记录的 `trx_id` 为 52，比自己的事务 id 还大，并且比下一个事务 id 53 小，这意味着，事务 A 读到是和自己同时启动事务的事务 B 修改的数据，这时事务 A 并不会读取这条记录，而是沿着 `undo log` 链条往下找旧版本的记录，直到找到 `trx_id` 等于或者小于事务 A 的事务 id 的第一条记录，所以事务 A 再一次读取到 `trx_id` 为 50 的记录，也就是小林余额是 100 万的这条记录。

「可重复读」隔离级别就是在启动时创建了 Read View，然后在事务期间读取数据的时候，在找到数据后，先会将该记录的 `trx_id` 和该事务的 Read View 里的字段做个比较：

- 如果记录的 `trx_id` 比该事务的 Read View 中的 `creator_trx_id` 要小，且不在 `m_ids` 列表里，这意味着这条记录的事务早就在该事务前提交过了，所以该记录对该事务可见；
- 如果记录的 `trx_id` 比该事务的 Read View 中的 `creator_trx_id` 要大，且在 `m_ids` 列表里，这意味着该事务读到的是和自己同时启动的另外一个事务修改的数据，这时就不应该读取这条记录，而是沿着 `undo log` 链条往下找旧版本的记录，直到找到 `trx_id` 等于或者小于该事务 `id` 的第一条记录。

就是通过这样的方式实现了，「可重复读」隔离级别下在事务期间读到的数据都是事务启动前的记录。

这种通过记录的版本链来控制并发事务访问同一个记录时的行为，这就叫 **MVCC**（多版本并发控制）。

数据结构部分：

1.数组和链表的区别。（很简单，但是很常考，记得要回答全面）

C++语言中可以用数组处理一组数据类型相同的数据，但不允许动态定义数组的大小，即在使用数组之前必须确定数组的大小。而在实际应用中，用户使用数组之前有时无法准确确定数组的大小，只能将数组定义成足够大小，这样数组中有些空间可能不被使用，从而造成内存空间的浪费。链表是一种常见的数据组织形式，它采用动态分配内存的形式实现。需要时可以用 `new` 分配内存空间，不需要时用 `delete` 将已分配的空间释放，不会造成内存空间的浪费。

从逻辑结构来看：数组必须事先定义固定的长度（元素个数），不能适应数据动态地增减的情况，即数组的大小一旦定义就不能改变。当数据增加时，可能超出原先定义的元素个数；当数据减少时，造成内存浪费；链表动态地进行存储分配，可以适应数据动态地增减的情况，且可以方便地插入、删除数据项。（数组中插入、删除数据项时，需要移动其它数据项）。

从内存存储来看：（静态）数组从栈中分配空间（用 `NEW` 创建的在堆中），对于程序员方便快捷，但是自由度小；链表从堆中分配空间，自由度大但是申请管理比较麻烦。

从访问方式来看：数组在内存中是连续存储的，因此，可以利用下标索引进行随机访问；链表是链式存储结构，在访问元素的时候只能通过线性的方式由前到后顺序访问，所以访问效率比数组要低。

数组 (Array)、栈 (Stack)、队列 (Queue)、链表 (Linked List)

树：堆(heap)、（B-树、B+树、）二叉查找树、AVL 树、红黑树、二叉树、哈夫曼树

图 (Graph)

散列表 (Hash)

- 数组：有序的元素序列，固定大小
- 栈：是一种运算受限的线性表，先进后出
- 队列：一种操作受限制的线性表，先进先出
- 链表：非连续、非顺序的存储结构，逻辑顺序是通过链表中的指针实现的

1.头插法：新插入的数据的指针指向最后的数据

2.尾插法：最后的数据的指针指向新插入的数据

- 堆：堆通常是一个可以被看做一棵完全二叉树的数组对象

- B-树：B 树，是一种平衡的多路搜索树（这里的平衡指根节点到叶子节点的高度一样）
- B+树：B+树是应文件系统所需而出的一种 B-树的变型树

（B-树和 B+树比较复杂，暂时不作拓展，多用在文件和数据库种，不说出来应该也不会问到）

- 二叉查找树：每个节点的左子树比该节点小，右子树比该节点大，子树同样是二叉查找树
- 平衡二叉树：基于二叉查找树，又叫 AVL 树（这里的平衡指左右子树深度差的绝对值不超过 1）
- 红黑树：一种特化的 AVL 树
- 二叉树：二叉树是每个结点最多有两个子树的树结构

1.完全二叉树：若设二叉树的深度为 h ，除第 h 层外，其它各层($1 \sim h-1$)的结点数都达到最大个数，第 h 层所有的结点都连续集中在最左边，这就是完全二叉树。

2.满二叉树：除了叶结点外每一个结点都有左右子叶且叶子结点都处在最底层的二叉树

3.二叉树遍历：前序中序后序分别是根左右、左根右、左右根

- 哈夫曼树：一种带权路径长度最短的二叉树，也称为最优二叉树
- 图：一些顶点的集合，节点之间的关系可以是任意的
- 散列表：又叫哈希表（Hash Table），是能够通过给定的关键字的值直接访问到具体对应的值的一个数据结构

树与二叉树的区别

1. 二叉树每个节点最多有 2 个子节点，树则无限制。
2. 二叉树中节点的子树分为左子树和右子树，即使某节点只有一棵子树，也要指明该子树是左子树还是右子树，即二叉树是有序的。
3. 树决不能为空，它至少有一个节点，而一棵二叉树可以是空的。

平衡二叉树的性质

平衡二叉树又叫 AVL 树，它的左子树上的所有节点的值都比根节点的值小，而右子树上的所有节点的值都比根节点的值大，且左子树与右子树的高度差最大为 1

红黑树的性质

红黑树基于 AVL 树，但没有追求左子树与右子树的高度差最大为 1 这个性质，而追求以下性质：

1. 每个结点是红色或黑色
 2. 根结点是黑色
 3. 每个叶子结点都是黑色的空结点
 4. 每个红色结点的两个子结点是黑色（从每个叶子到根的所有路径上不能有连续两个红色节点）
 5. 从任意结点到其每个叶子结点的路径包含相同数目的黑色结点
- 这些性质构造出红黑树的关键性质：从根到叶子的最长的可能路径不多于最短的可能路径的两倍长
 - 红黑树在任何不平衡的情况只需要最多三次旋转就能达到平衡，插入和删除的次数比 AVL 树少，查询效率稍逊于 AVL 树
 - 一棵有 n 个结点的红黑树的高度至多为 $2\lg(n+1)$
 - STL 容器中 map、multimap、set、multiset 都用到红黑树

至于变色和旋转的方法博主是参考这篇文章学习的：[红黑树的旋转与变色](#)，但是具体代码实现还需要进一步研究

常用的哈希函数

- 直接寻址法：取关键字或关键字的某个线性函数值为散列地址。

- 数字分析法：通过对数据的分析，发现数据中冲突较少的部分，并构造散列地址。
- 平方取中法：当无法确定关键字里哪几位的分布相对比较均匀时，可以先求出关键字的平方值，然后按需要取平方值的中间几位作为散列地址。这是因为：计算平方之后的中间几位和关键字中的每一位都相关，所以不同的关键字会以较高的概率产生不同的散列地址。
- 取随机数法：使用一个随机函数，取关键字的随机值作为散列地址，这种方式通常用于关键字长度不同的场合。
- 除留取余法：取关键字被某个不大于散列表的表长 n 的数 m 除后所得的余数 p 为散列地址。这种方式也可以在用过其他方法后再使用。该函数对 m 的选择很重要，一般取素数或者直接用 n

解决哈希表冲突的常用方法

- 开放地址法（也叫开放寻址法）：对计算出来的地址进行一个探测再哈希，比如往后移动一个地址，如果没人占用，就用这个地址。
- 再哈希法：在产生冲突之后，使用关键字的其他部分继续计算地址，如果还有冲突，则继续使用其他部分再计算地址。这种方式的缺点是时间增加了。
- 链地址法：链地址法其实就是对 Key 通过哈希之后落在同一个地址上的值，做一个链表。其实在很多高级语言的实现当中，也使用这种方式处理冲突。
- 建立一个公共溢出区：这种方式是建立一个公共溢出区，当地址存在冲突时，把新的地址放在公共溢出区里。

九种排序

- 直接插入排序：从第二个数开始，前面相邻的数进行比较，把大的数换到后面，如此循环
- 折半插入排序：在前面的有序区内不断使用二分法，即比本身小则 $low=mid+1$ ，比本身大则 $high=mid-1$ ，当 $low \leq high$ 时表示找到第一个比本身大的数，然后交换
- 希尔排序：实质为分组插入法，不断缩小步长进行插入排序
- 直接选择排序：从后面的数找出最小的数与本身交换
- 堆排序：选择排序的一种，近似完全二叉树，下标为 $n/2-1$ 的元素才有子树，构造大顶堆，即结点都比子树大，最后根节点就为最大的数，将最大的数和最后的数互换，继续将剩下的数构造大顶堆，如此循环

- 冒泡排序：交换排序的一种，每次都从第一个数开始，相邻的数进行比较，将最大的数后移，不断缩小比较次数
- 快速排序：交换排序的一种，先拿第一个数作为分界点，把数组排列成前半部分的数都比分界点小，后半部分的数都比分界点大，然后这 2 半部分继续同样的操作，递归完成
- 归并排序：分组归并，分组从最小的组开始进行过比较和交换，比如 10 个数，比较的区间为[0,1] [0,2] [3,4] [0,4] [5,6] [5,7] [8,9] [5,9] [0,9]
- 桶排序：将元素按大小分到固定长度的区间也就是桶中排序，再拿出来放回数组

| 各种常用排序算法 | | | | | | |
|----------|---------|-----------------|-----------------|-----------------|-----------------|-----|
| 类别 | 排序方法 | 时间复杂度 | | | 空间复杂度 | 稳定性 |
| | | 平均情况 | 最好情况 | 最坏情况 | 辅助存储 | |
| 插入排序 | 直接插入 | $O(n^2)$ | $O(n)$ | $O(n^2)$ | $O(1)$ | 稳定 |
| | shell排序 | $O(n^{1.3})$ | $O(n)$ | $O(n^2)$ | $O(1)$ | 不稳定 |
| 选择排序 | 直接选择 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 不稳定 |
| | 堆排序 | $O(n \log_2 n)$ | $O(n \log_2 n)$ | $O(n \log_2 n)$ | $O(1)$ | 不稳定 |
| 交换排序 | 冒泡排序 | $O(n^2)$ | $O(n)$ | $O(n^2)$ | $O(1)$ | 稳定 |
| | 快速排序 | $O(n \log_2 n)$ | $O(n \log_2 n)$ | $O(n^2)$ | $O(n \log_2 n)$ | 不稳定 |
| 归并排序 | | $O(n \log_2 n)$ | $O(n \log_2 n)$ | $O(n \log_2 n)$ | $O(1)$ | 稳定 |
| 基数排序 | | $O(d(r+n))$ | $O(d(n+rd))$ | $O(d(r+n))$ | $O(rd+n)$ | 稳定 |

注：基数排序的复杂度中，r代表关键字的基数，d代表长度，n代表关键字的个数

常用算法

- 基础：枚举，递归，分治，模拟，贪心，动态规划，剪枝，回溯
- 排序：冒泡、快速、直接选择和堆、直接插入和希尔排序、归并排序
- 查找：顺序查找、二分查找、索引查找、二叉排序树、哈希查找
- 图算法：深度优先遍历与广度优先遍历，最短路径，最小生成树，拓扑排序

