本文目录：

事务的四大特性?

事务特性ACID：**原子性**（`Atomicity`）、**一致性**（`Consistency`）、**隔离性**（`Isolation`）、**持久性**（`Durability`）。

- **原子性**是指事务包含的所有操作要么全部成功，要么全部失败回滚。
- **一致性**是指一个事务执行之前和执行之后都必须处于一致性状态。比如a与b账户共有1000块，两人之间转账之后无论成功还是失败，它们的账户总和还是1000。
- **隔离性**。跟隔离级别相关，如`read committed`，一个事务只能读到已经提交的修改。
- **持久性**是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

数据库的三大范式

第一范式1NF

确保数据库表字段的原子性。

比如字段`userInfo`：`广东省 10086'`，依照第一范式必须拆分成`userInfo`：`广东省` `userTel`：`10086`两个字段。

第二范式2NF

首先要满足第一范式，另外包含两部分内容，一是表必须有一个主键；二是非主键列必须完全依赖于主键，而不能只依赖于主键的一部分。

举个例子。假定选课关系表为`student_course`(student_no, student_name, age, course_name, grade, credit)，主键为(student_no, course_name)。其中学分完全依赖于课程名称，姓名年龄完全依赖学号，不符合第二范式，会导致数据冗余（学生选n门课，姓名年龄有n条记录）、插入异常（插入一门新课，因为没有学号，无法保存新课记录）等问题。

可以拆分成三个表：学生：`student`(stuent_no, student_name, 年龄)；课程：`course`(course_name, credit)；选课关系：`student_course_relation`(student_no, course_name, grade)。

第三范式3NF

首先要满足第二范式，另外非主键列必须直接依赖于主键，不能存在传递依赖。即不能存在：非主键列 A 依赖于非主键列 B，非主键列 B 依赖于主键的情况。

假定学生关系表为Student(student_no, student_name, age, academy_id, academy_telephone)，主键为"学号"，其中学院id依赖于学号，而学院地点和学院电话依赖于学院id，存在传递依赖，不符合第三范式。

可以把学生关系表分为如下两个表：学生：(student_no, student_name, age, academy_id)；学院：(academy_id, academy_telephone)。

2NF和3NF的区别?

- 2NF依据是非主键列是否完全依赖于主键，还是依赖于主键的一部分。
- 3NF依据是非主键列是直接依赖于主键，还是直接依赖于非主键。

事务隔离级别有哪些?

先了解下几个概念：脏读、不可重复读、幻读。

- **脏读**是指在一个事务处理过程里读取了另一个未提交的事务中的数据。
- **不可重复读**是指在对于数据库中的某行记录，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，另一个事务修改了数据并提交了。
- **幻读**是当某个事务在读取某个范围内的记录时，另外一个事务又在该范围内插入了新的记录，当之前的事务再次读取该范围的记录时，会产生幻行，就像产生幻觉一样，这就是发生了幻读。

不可重复读和脏读的区别是，脏读是某一事务读取了另一个事务未提交的脏数据，而不可重复读则是读取了前一事务提交的数据。

幻读和不可重复读都是读取了另一条已经提交的事务，不同的是不可重复读的重点是修改，幻读的重点在于新增或者删除。

事务隔离就是为了解决上面提到的脏读、不可重复读、幻读这几个问题。

MySQL数据库为我们提供的四种隔离级别：

- **Serializable** (串行化)：通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。
- **Repeatable read** (可重复读)：MySQL的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行，解决了不可重复读的问题。
- **Read committed** (读已提交)：一个事务只能看见已经提交事务所做的改变。可避免脏读的发生。
- **Read uncommitted** (读未提交)：所有事务都可以看到其他未提交事务的执行结果。

查看隔离级别：

```
select @@transaction_isolation;
```

设置隔离级别：

```
set session transaction isolation level read uncommitted;
```

索引

什么是索引?

索引是存储引擎用于提高数据库表的访问速度的一种**数据结构**。

索引的优缺点?

优点：

- **加快数据查找的速度**

- 为用来排序或者是分组的字段添加索引，可以加快分组和排序的速度
- 加快表与表之间的连接

缺点：

- 建立索引需要**占用物理空间**
- 会降低表的增删改的效率，因为每次对表记录进行增删改，需要进行**动态维护索引**，导致增删改时间变长

索引的作用?

数据是存储在磁盘上的，查询数据时，如果没有索引，会加载所有的数据到内存，依次进行检索，读取磁盘次数较多。有了索引，就不需要加载所有数据，因为B+树的高度一般在2-4层，最多只需要读取2-4次磁盘，查询速度大大提升。

什么情况下需要建索引?

1. 经常用于查询的字段
2. 经常用于连接的字段建立索引，可以加快连接的速度
3. 经常需要排序的字段建立索引，因为索引已经排好序，可以加快排序查询速度

什么情况下不建索引?

1. `where`条件中用不到的字段不适合建立索引
2. 表记录较少
3. 需要经常增删改
4. **参与列计算**的列不适合建索引
5. **区分度不高**的字段不适合建立索引，如性别等

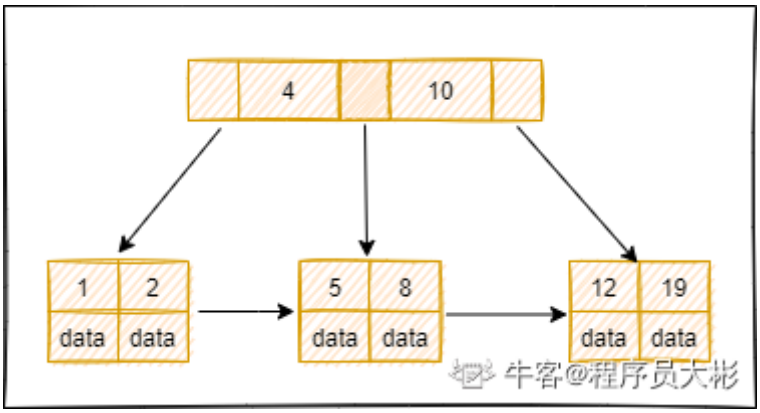
索引的数据结构

索引的数据结构主要有B+树和哈希表，对应的索引分别为B+树索引和哈希索引。InnoDB引擎的索引类型有B+树索引和哈希索引，默认的索引类型为B+树索引。

B+树索引

B+ 树是基于B 树和叶子节点顺序访问指针进行实现，它具有B树的平衡性，并且通过顺序访问指针来提高区间查询的性能。

在 B+ 树中，节点中的 `key` 从左到右递增排列，如果某个指针的左右相邻 `key` 分别是 keyi 和 keyi+1，则该指针指向节点的所有 `key` 大于等于 keyi 且小于等于 keyi+1。



进行查找操作时，首先在根节点进行二分查找，找到 `key` 所在的指针，然后递归地在指针所指向的节点进行查找。直到查找到叶子节点，然后在叶子节点上进行二分查找，找出 `key` 所对应的数据项。

MySQL 数据库使用最多的索引类型是 `BTREE` 索引，底层基于B+树数据结构来实现。

```
mysql> show index from blog\G;
*************************** 1. row ***************************
        Table: blog
   Non_unique: 0
     Key_name: PRIMARY
  Seq_in_index: 1
  Column_name: blog_id
    Collation: A
  Cardinality: 4
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
Index_comment:
      Visible: YES
   Expression: NULL
```

哈希索引

哈希索引是基于哈希表实现的，对于每一行数据，存储引擎会对索引列进行哈希计算得到哈希码，并且哈希算法要尽量保证不同的列值计算出的哈希码值是不同的，将哈希码的值作为哈希表的key值，将指向数据行的指针作为哈希表的value值。这样查找一个数据的时间复杂度就是O(1)，一般多用于精确查找。

Hash索引和B+树索引的区别?

- 哈希索引**不支持排序**，因为哈希表是无序的。
- 哈希索引**不支持范围查找**。
- 哈希索引**不支持模糊查询**及多列索引的最左前缀匹配。
- 因为哈希表中会**存在哈希冲突**，所以哈希索引的性能是不稳定的，而B+树索引的性能是相对稳定的，每次查询都是从根节点到叶子节点。

为什么B+树比B树更适合实现数据库索引?

- 由于B+树的数据都存储在叶子结点中，叶子结点均为索引，方便扫库，只需要扫一遍叶子结点即可，但是B树因为其分支结点同样存储着数据，我们要找到具体的数据，需要进行一次中序遍历按序来扫，所以

B+树更加适合在区间查询的情况，而在数据库中基于范围的查询是非常频繁的，所以通常B+树用于数据库索引。

- B+树的节点只存储索引key值，具体信息的地址存在于叶子节点的地址中。这就使以页为单位的索引中可以存放更多的节点。减少更多的I/O支出。
- B+树的查询效率更加稳定，任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

索引有什么分类?

1、**主键索引**：名为primary的唯一非空索引，不允许有空值。

2、**唯一索引**：索引列中的值必须是唯一的，但是允许为空值。唯一索引和主键索引的区别是：唯一约束的列可以为`null`且可以存在多个`null`值。唯一索引的用途：唯一标识数据库表中的每条记录，主要是用来防止数据重复插入。创建唯一索引的SQL语句如下：

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name UNIQUE KEY(column_1,column_2,...);
```

3、**组合索引**：在表中的多个字段组合上创建的索引，只有在查询条件中使用了这些字段的左边字段时，索引才会被使用，使用组合索引时需遵循最左前缀原则。

4、**全文索引**：只有在`MyISAM`引擎上才能使用，只能在`CHAR`、`VARCHAR`和`TEXT`类型字段上使用全文索引。

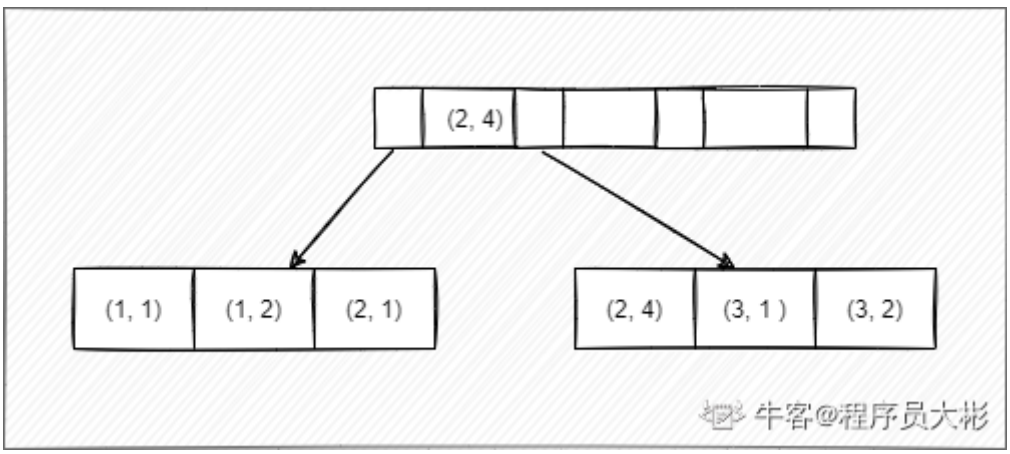
什么是最左匹配原则?

如果 SQL 语句中用到了组合索引中的最左边的索引，那么这条 SQL 语句就可以利用这个组合索引去进行匹配。当遇到范围查询(`>`、`<`、`between`、`like`)就会停止匹配，后面的字段不会用到索引。

对`(a,b,c)`建立索引，查询条件使用 a/ab/abc 会走索引，使用 bc 不会走索引。

对`(a,b,c,d)`建立索引，查询条件为`a = 1 and b = 2 and c > 3 and d = 4`，那么a、b和c三个字段能用到索引，而d无法使用索引。因为遇到了范围查询。

如下图，对(a, b) 建立索引，a 在索引树中是全局有序的，而 b 是全局无序，局部有序（当a相等时，会根据b进行排序）。直接执行`b = 2`这种查询条件无法使用索引。



当a的值确定的时候，b是有序的。例如`a = 1`时，b值为1，2是有序的状态。当`a = 2`时候，b的值为1，4也是有序状态。 当执行`a = 1 and b = 2`时a和b字段能用到索引。而执行`a > 1 and b = 2`时，a字段能用到索引，b字段用不到索引。因为a的值此时是一个范围，不是固定的，在这个范围内b值不是有序的，因此b字段无法使用索引。

什么是聚集索引?

InnoDB使用表的主键构造主键索引树，同时叶子节点中存放的即为整张表的记录数据。聚集索引叶子节点的存储是逻辑上连续的，使用双向链表连接，叶子节点按照主键的顺序排序，因此对于主键的排序查找和范围查找速度比较快。

聚集索引的叶子节点就是整张表的行记录。InnoDB 主键使用的是聚簇索引。聚集索引要比非聚集索引查询效率高很多。

对于`InnoDB`来说，聚集索引一般是表中的主键索引，如果表中没有显示指定主键，则会选择表中的第一个不允许为`NULL`的唯一索引。如果没有主键也没有合适的唯一索引，那么`InnoDB`内部会生成一个隐藏的主键作为聚集索引，这个隐藏的主键长度为6个字节，它的值会随着数据的插入自增。

什么是覆盖索引?

`select`的数据列只用从索引中就能够取得，不需要**回表**进行二次查询，也就是说查询列要被所使用的索引覆盖。对于`innodb`表的二级索引，如果索引能覆盖到查询的列，那么就可以避免对主键索引的二次查询。

不是所有类型的索引都可以成为覆盖索引。覆盖索引要存储索引列的值，而哈希索引、全文索引不存储索引列的值，所以MySQL使用b+树索引做覆盖索引。

对于使用了覆盖索引的查询，在查询前面使用`explain`，输出的extra列会显示为`using index`。

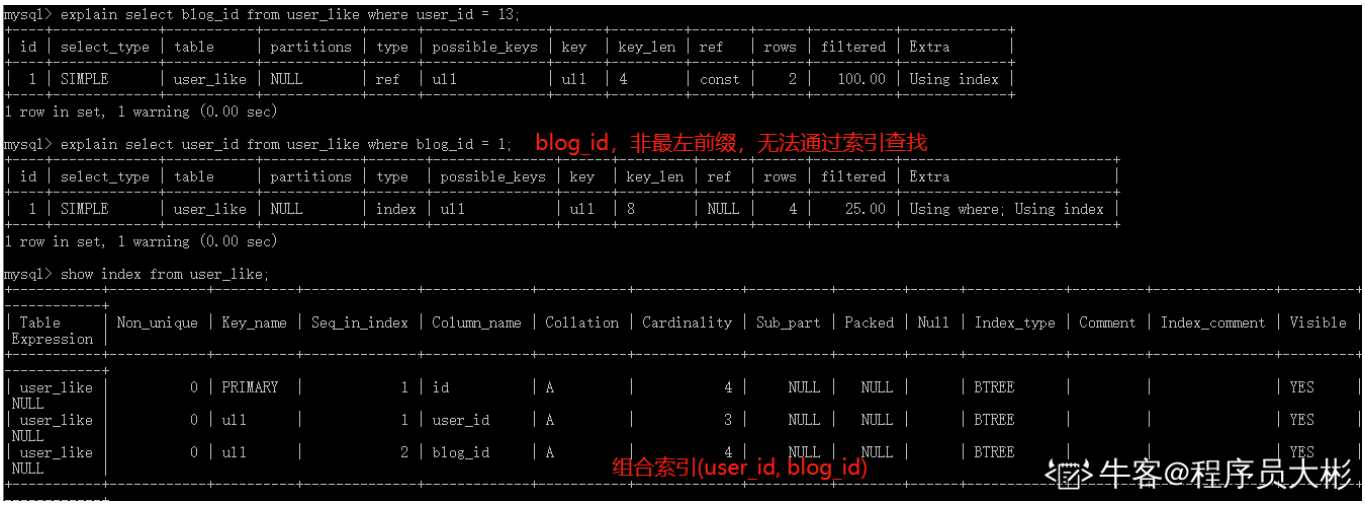
比如`user_like`用户点赞表，组合索引为`(user_id, blog_id)`，`user_id`和`blog_id`都不为`null`。

```
explain select blog_id from user_like where user_id = 13;
```

`explain`结果的`Extra`列为`Using index`，查询的列被索引覆盖，并且where筛选条件符合最左前缀原则，通过**索引查找**就能直接找到符合条件的数据，不需要回表查询数据。

```
explain select user_id from user_like where blog_id = 1;
```

`explain`结果的`Extra`列为`Using where; Using index`， 查询的列被索引覆盖，where筛选条件不符合最左前缀原则，无法通过索引查找找到符合条件的数据，但可以通过**索引扫描**找到符合条件的数据，也不需要回表查询数据。



索引的设计原则?

- 索引列的**区分度越高**，索引的效果越好。比如使用性别这种区分度很低的列作为索引，效果就会很差。
- 尽量使用**短索引**，对于较长的字符串进行索引时应该指定一个较短的前缀长度，因为较小的索引涉及到的磁盘I/O较少，查询速度更快。
- 索引不是越多越好，每个索引都需要额外的物理空间，维护也需要花费时间。
- 利用**最左前缀原则**。

索引什么时候会失效?

导致索引失效的情况：

- 对于组合索引，不是使用组合索引最左边的字段，则不会使用索引
- 以%开头的like查询如`%abc`，无法使用索引；非%开头的like查询如`abc%`，相当于范围查询，会使用索引
- 查询条件中列类型是字符串，没有使用引号，可能会因为类型不同发生隐式转换，使索引失效
- 判断索引列是否不等于某个值时
- 对索引列进行运算
- 查询条件使用`or`连接，也会导致索引失效

什么是前缀索引?

有时需要在很长的字符列上创建索引，这会造成索引特别大且慢。使用前缀索引可以避免这个问题。

前缀索引是指对文本或者字符串的前几个字符建立索引，这样索引的长度更短，查询速度更快。

创建前缀索引的关键在于选择足够长的前缀以**保证较高的索引选择性**。索引选择性越高查询效率就越高，因为选择性高的索引可以让MySQL在查找时过滤掉更多的数据行。

建立前缀索引的方式：

```
// email列创建前缀索引
ALTER TABLE table_name ADD KEY(column_name(prefix_length));
```

常见的存储引擎有哪些?

MySQL中常用的四种存储引擎分别是： **MyISAM**、**InnoDB**、**MEMORY**、**ARCHIVE**。MySQL 5.5版本后默认的存储引擎为`InnoDB`。

InnoDB存储引擎

InnoDB是MySQL**默认的事务型存储引擎**，使用最广泛，基于聚簇索引建立的。InnoDB内部做了很多优化，如能够自动在内存中创建自适应hash索引，以加速读操作。

优点：支持事务和崩溃修复能力；引入了行级锁和外键约束。

缺点：占用的数据空间相对较大。

适用场景：需要事务支持，并且有较高的并发读写频率。

MyISAM存储引擎

数据以紧密格式存储。对于只读数据，或者表比较小、可以容忍修复操作，可以使用MyISAM引擎。MyISAM会将表存储在两个文件中，数据文件`.MYD`和索引文件`.MYI`。

优点：访问速度快。

缺点：MyISAM不支持事务和行级锁，不支持崩溃后的安全恢复，也不支持外键。

适用场景：对事务完整性没有要求；表的数据都会只读的。

MEMORY存储引擎

MEMORY引擎将数据全部放在内存中，访问速度较快，但是一旦系统奔溃的话，数据都会丢失。

MEMORY引擎默认使用哈希索引，将键的哈希值和指向数据行的指针保存在哈希索引中。

优点：访问速度较快。

缺点：

1. 哈希索引数据不是按照索引值顺序存储，无法用于排序。
2. 不支持部分索引匹配查找，因为哈希索引是使用索引列的全部内容来计算哈希值的。
3. 只支持等值比较，不支持范围查询。
4. 当出现哈希冲突时，存储引擎需要遍历链表中所有的行指针，逐行进行比较，直到找到符合条件的行。

ARCHIVE存储引擎

ARCHIVE存储引擎非常适合存储大量独立的、作为历史记录的数据。ARCHIVE提供了压缩功能，拥有高效的插入速度，但是这种引擎不支持索引，所以查询性能较差。

MyISAM和InnoDB的区别?

1. **是否支持行级锁**：`MyISAM`只有表级锁，而`InnoDB`支持行级锁和表级锁，默认为行级锁。
2. **是否支持事务和崩溃后的安全恢复**： `MyISAM`不提供事务支持。而`InnoDB`提供事务支持，具有事务、回滚和崩溃修复能力。
3. **是否支持外键：** `MyISAM`不支持，而`InnoDB`支持。

4. **是否支持MVCC**： `MyISAM`不支持， `InnoDB`支持。应对高并发事务，MVCC比单纯的加锁更高效。
5. `MyISAM`不支持聚集索引， `InnoDB`支持聚集索引。

MVCC 实现原理?

MVCC(`Multiversion concurrency control`) 就是同一份数据保留多版本的一种方式，进而实现并发控制。在查询的时候，通过`read view`和版本链找到对应版本的数据。

作用：提升并发性能。对于高并发场景，MVCC比行级锁开销更小。

MVCC 实现原理如下：

MVCC 的实现依赖于版本链，版本链是通过表的三个隐藏字段实现。

- `DB_TRX_ID`： 当前事务id，通过事务id的大小判断事务的时间顺序。
- `DB_ROLL_PRT`： 回滚指针，指向当前行记录的上一个版本，通过这个指针将数据的多个版本连接在一起构成`undo log`版本链。
- `DB_ROLL_ID`： 主键，如果数据表没有主键，InnoDB会自动生成主键。

每条表记录大概是这样的：

name	age	DB_ROW_ID	DB_TRX_ID	DB_ROLL_PTR
大彬	18	1	牛客@程序员大彬	

使用事务更新行记录的时候，就会生成版本链，执行过程如下：

1. 用排他锁锁住该行；
2. 将该行原本的值拷贝到`undo log`，作为旧版本用于回滚；
3. 修改当前行的值，生成一个新版本，更新事务id，使回滚指针指向旧版本的记录，这样就形成一条版本链。

下面举个例子方便大家理解。

1、初始数据如下，其中`DB_ROW_ID`和`DB_ROLL_PTR`为空。

name	age	DB_ROW_ID	DB_TRX_ID	DB_ROLL_PTR
大彬	18	1	牛客@程序员大彬	

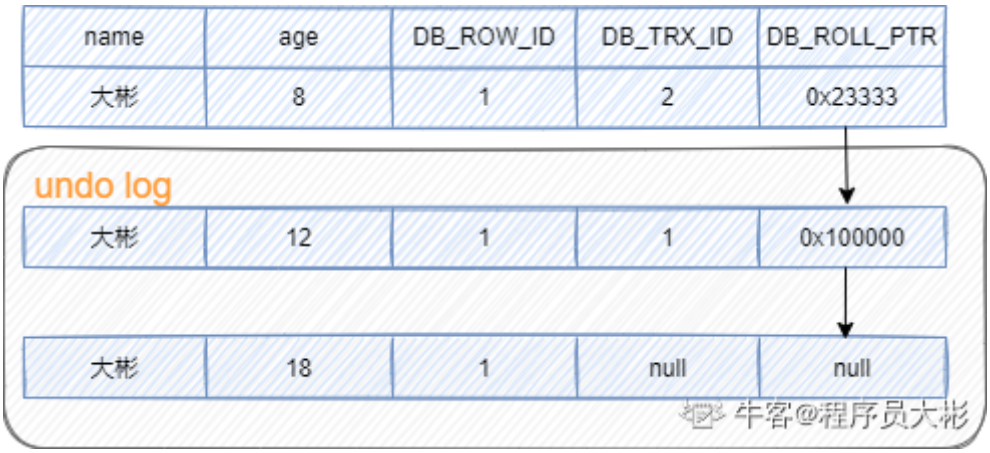
2、事务A对该行数据做了修改，将`age`修改为12，效果如下：

name	age	DB_ROW_ID	DB_TRX_ID	DB_ROLL_PTR
大彬	12	1	1	0x100000

undo log

大彬	18	1	牛客@程序员大彬	

3、之后事务B也对该行记录做了修改，将`age`修改为8，效果如下：



4、此时undo log有两行记录，并且通过回滚指针连在一起。

接下来了解下read view的概念。

`read view`可以理解成将数据在每个时刻的状态拍成“照片”记录下来。在获取某时刻t的数据时，到t时间点拍的“照片”上取数据。

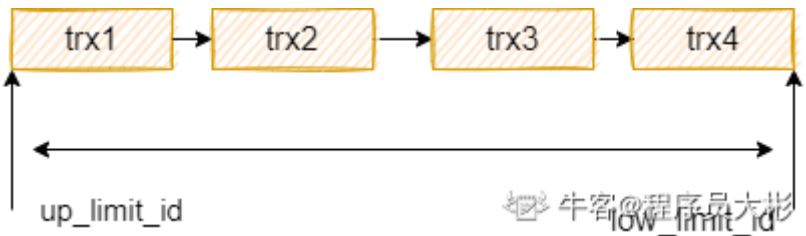
在`read view`内部维护一个活跃事务链表，表示生成`read view`的时候还在活跃的事务。这个链表包含在创建`read view`之前还未提交的事务，不包含创建`read view`之后提交的事务。

不同隔离级别创建read view的时机不同。

- read committed：每次执行select都会创建新的read_view，保证能读取到其他事务已经提交的修改。
- repeatable read：在一个事务范围内，第一次select时更新这个read_view，以后不会再更新，后续所有的select都是复用之前的read_view。这样可以保证事务范围内每次读取的内容都一样，即可重复读。

read view的记录筛选方式

前提：`DATA_TRX_ID`表示每个数据行的最新的事务ID；`up_limit_id`表示当前快照中的最先开始的事务；`low_limit_id`表示当前快照中的最慢开始的事务，即最后一个事务。



- 如果`DATA_TRX_ID < up_limit_id`：说明在创建`read view`时，修改该数据行的事务已提交，该版本的记录可被当前事务读取到。
- 如果`DATA_TRX_ID >= low_limit_id`：说明当前版本的记录的事务是在创建`read view`之后生成的，该版本的数据行不可以被当前事务访问。此时需要通过版本链找到上一个版本，然后重新判断该版本的记录对当前事务的可见性。
- 如果`up_limit_id <= DATA_TRX_ID < low_limit_i`：
  i. 需要在活跃事务链表中查找是否存在ID为`DATA_TRX_ID`的值的事务。
  ii. 如果存在，因为在活跃事务链表中的事务是未提交的，所以该记录是不可见的。此时需要通过版本链找到上一个版本，然后重新判断该版本的可见性。
  iii. 如果不存在，说明事务trx_id 已经提交了，这行记录是可见的。

总结：InnoDB 的`MVCC`是通过 `read view` 和版本链实现的，版本链保存有历史版本记录，通过`read view` 判断当前版本的数据是否可见，如果不可见，再从版本链中找到上一个版本，继续进行判断，直到找到一个可见的版本。

快照读和当前读

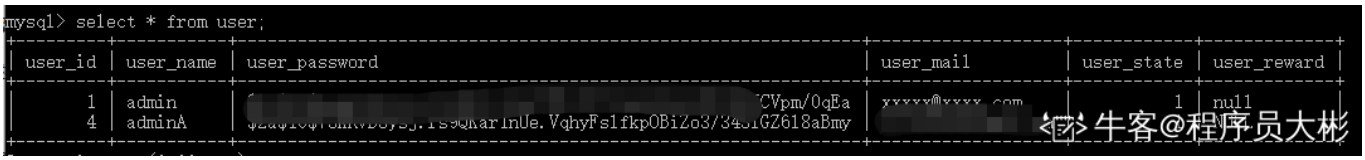
表记录有两种读取方式。

- 快照读：读取的是快照版本。普通的`SELECT`就是快照读。通过mvcc来进行并发控制的，不用加锁。
- 当前读：读取的是最新版本。`UPDATE、DELETE、INSERT、SELECT … LOCK IN SHARE MODE、SELECT … FOR UPDATE` 是当前读。

快照读情况下，InnoDB通过`mvcc`机制避免了幻读现象。而`mvcc`机制无法避免当前读情况下出现的幻读现象。因为当前读每次读取的都是最新数据，这时如果两次查询中间有其它事务插入数据，就会产生幻读。

下面举个例子说明下：

1、首先，user表只有两条记录，具体如下：



2、事务a和事务b同时开启事务`start transaction`；

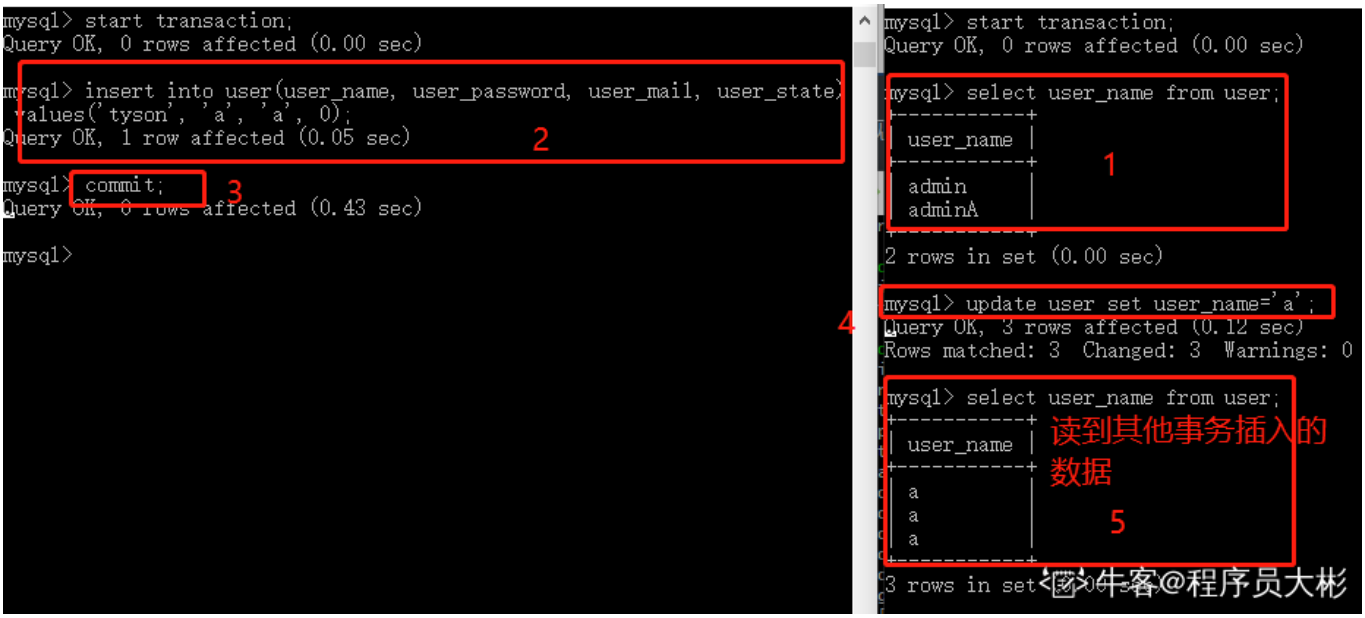
3、事务a插入数据然后提交；

```
insert into user(user_name, user_password, user_mail, user_state) values('tyson', 'a', 'a', 0);
```

4、事务b执行全表的update；

```
update user set user_name = 'a';
```

5、事务b然后执行查询，查到了事务a中插入的数据。（下图左边是事务b，右边是事务a。事务开始之前只有两条记录，事务a插入一条数据之后，事务b查询出来是三条数据）



以上就是当前读出现的幻读现象。

那么MySQL是如何避免幻读?

- 在快照读情况下，MySQL通过`mvcc`来避免幻读。
- 在当前读情况下，MySQL通过`next-key`来避免幻读（加行锁和间隙锁来实现的）。

next-key包括两部分：行锁和间隙锁。行锁是加在索引上的锁，间隙锁是加在索引之间的。

`Serializable`隔离级别也可以避免幻读，会锁住整张表，并发性极低，一般不会使用。

共享锁和排他锁

SELECT 的读取锁定主要分为两种方式：共享锁和排他锁。

```
select * from table where id<6 lock in share mode;--共享锁
select * from table where id<6 for update;--排他锁
```

这两种方式主要的不同在于`LOCK IN SHARE MODE`多个事务同时更新同一个表单时很容易造成死锁。

申请排他锁的前提是，没有线程对该结果集的任何行数据使用排它锁或者共享锁，否则申请会受到阻塞。在进行事务操作时，MySQL会对查询结果集的每行数据添加排它锁，其他线程对这些数据的更改或删除操作会被阻塞（只能读操作），直到该语句的事务被`commit`语句或`rollback`语句结束为止。

`SELECT... FOR UPDATE`使用注意事项：

1. `for update`仅适用于innodb，且必须在事务范围内才能生效。
2. 根据主键进行查询，查询条件为`like`或者不等于，主键字段产生**表锁**。
3. 根据非索引字段进行查询，会产生**表锁**。

大表怎么优化?

某个表有近千万数据，查询比较慢，如何优化?

当MySQL单表记录数过大时，数据库的性能会明显下降，一些常见的优化措施如下：

- 限定数据的范围。比如：用户在查询历史信息的时候，可以控制在一个月的时间范围内；
- 读写分离： 经典的数据库拆分方案，主库负责写，从库负责读；
- 通过分库分表的方式进行优化，主要有垂直拆分和水平拆分。

bin log/redo log/undo log

MySQL日志主要包括查询日志、慢查询日志、事务日志、错误日志、二进制日志等。其中比较重要的是 `bin log` （二进制日志）和 `redo log` （重做日志）和 `undo log` （回滚日志）。

bin log

`bin log` 是MySQL数据库级别的文件，记录对MySQL数据库执行修改的所有操作，不会记录select和show语句，主要用于恢复数据库和同步数据库。

redo log

`redo log` 是innodb引擎级别，用来记录innodb存储引擎的事务日志，不管事务是否提交都会记录下来，用于数据恢复。当数据库发生故障，innoDB存储引擎会使用 `redo log` 恢复到发生故障前的时刻，以此来保证数据的完整性。将参数 `innodb_flush_log_at_tx_commit` 设置为1，那么在执行commit时会将 `redo log` 同步写到磁盘。

undo log

除了记录 `redo log` 外，当进行数据修改时还会记录 `undo log` ， `undo log` 用于数据的撤回操作，它保留了记录修改前的内容。通过 `undo log` 可以实现事务回滚，并且可以根据 `undo log` 回溯到某个特定的版本的数据，**实现MVCC。**

bin log和redo log有什么区别?

1. `bin log` 会记录所有日志记录，包括InnoDB、MyISAM等存储引擎的日志； `redo log` 只记录innoDB自身的事务日志。
2. `bin log` 只在事务提交前写入到磁盘，一个事务只写一次；而在事务进行过程，会有 `redo log` 不断写入磁盘。
3. `bin log` 是逻辑日志，记录的是SQL语句的原始逻辑； `redo log` 是物理日志，记录的是在某个数据页上做了什么修改。

讲一下MySQL架构?

MySQL主要分为 Server 层和存储引擎层：

- **Server 层**：主要包括连接器、查询缓存、分析器、优化器、执行器等，所有跨存储引擎的功能都在这一层实现，比如存储过程、触发器、视图，函数等，还有一个通用的日志模块 binglog 日志模块。
- **存储引擎**： 主要负责数据的存储和读取。server 层通过api与存储引擎进行通信。

Server 层基本组件

- **连接器：** 当客户端连接 MySQL 时，server层会对其进行身份认证和权限校验。
- **查询缓存:** 执行查询语句的时候，会先查询缓存，先校验这个 sql 是否执行过，如果有缓存这个 sql，就会直接返回给客户端，如果没有命中，就会执行后续的操作。
- **分析器:** 没有命中缓存的话，SQL 语句就会经过分析器，主要分为两步，词法分析和语法分析，先看 SQL 语句要做什么，再检查 SQL 语句语法是否正确。
- **优化器：** 优化器对查询进行优化，包括重写查询、决定表的读写顺序以及选择合适的索引等，生成执行计划。
- **执行器：** 首先执行前会校验该用户有没有权限，如果没有权限，就会返回错误信息，如果有权限，就会根据执行计划去调用引擎的接口，返回结果。

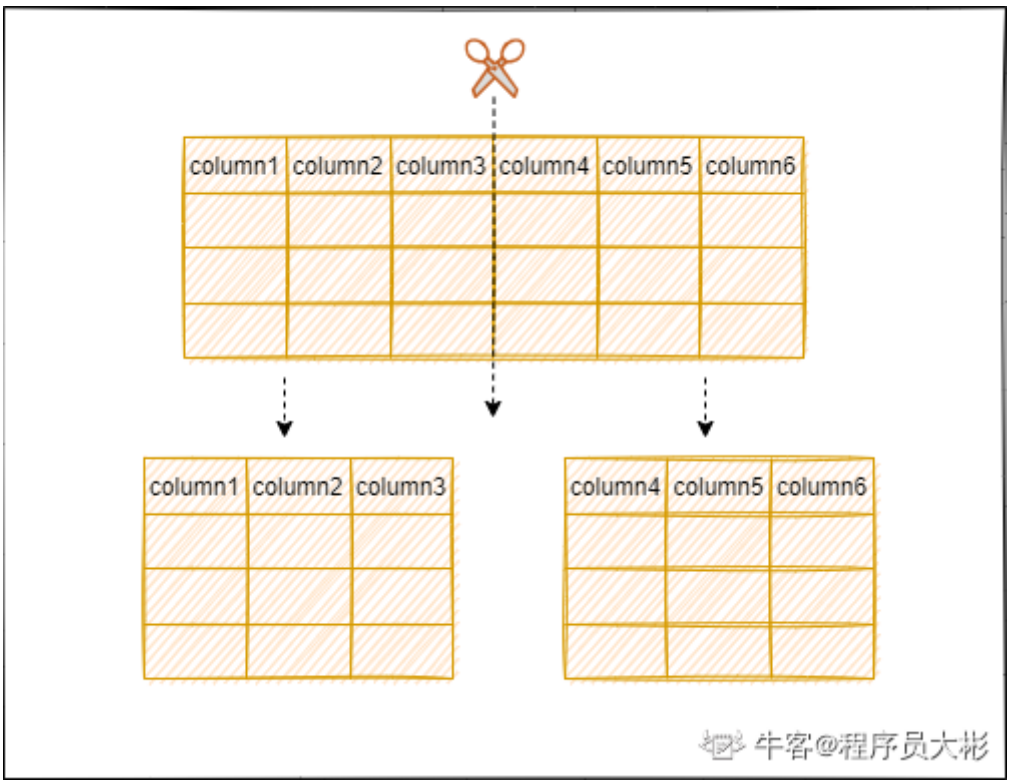
分库分表

当单表的数据量达到1000W或100G以后，优化索引、添加从库等可能对数据库性能提升效果不明显，此时就要考虑对其进行切分了。切分的目的就在于减少数据库的负担，缩短查询的时间。

数据切分可以分为两种方式：垂直划分和水平划分。

垂直划分

垂直划分数据库是根据业务进行划分，例如购物场景，可以将库中涉及商品、订单、用户的表分别划分出成一个库，通过降低单库的大小来提高性能。同样的，分表的情况就是将一个大表根据业务功能拆分成一个个子表，例如商品基本信息和商品描述，商品基本信息一般会展示在商品列表，商品描述在商品详情页，可以将商品基本信息和商品描述拆分成两张表。



优点：行记录变小，数据页可以存放更多记录，在查询时减少I/O次数。

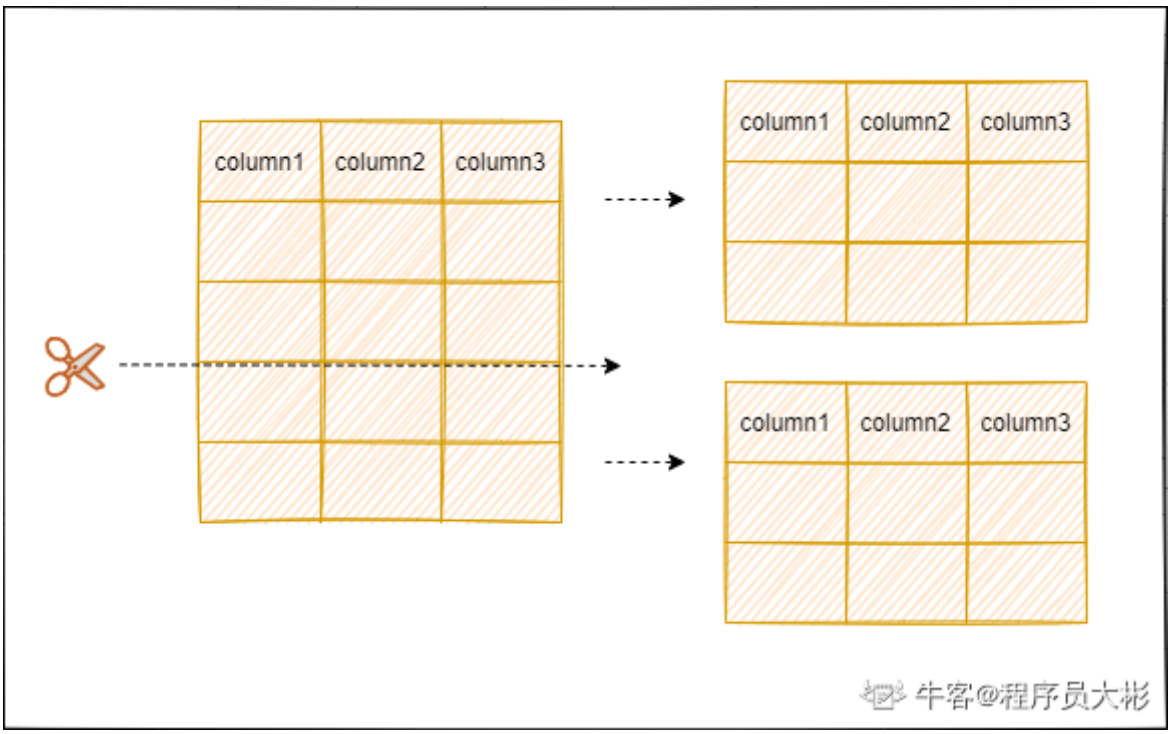
缺点：

- 主键出现冗余，需要管理冗余列；
- 会引起表连接JOIN操作，可以通过在业务服务器上进行join来减少数据库压力；

- 依然存在单表数据量过大的问题。

水平划分

水平划分是根据一定规则，例如时间或id序列值等进行数据的拆分。比如根据年份来拆分不同的数据库。每个数据库结构一致，但是数据得以拆分，从而提升性能。



优点：单库（表）的数据量得以减少，提高性能；切分出的表结构相同，程序改动较少。

缺点：

- 分片事务一致性难以解决
- 跨节点`join`性能差，逻辑复杂
- 数据分片在扩容时需要迁移

什么是分区表?

分区表是一个独立的逻辑表，但是底层由多个物理子表组成。

当查询条件的数据分布在某一个分区的时候，查询引擎只会去某一个分区查询，而不是遍历整个表。在管理层面，如果需要删除某一个分区的数据，只需要删除对应的分区即可。

分区表类型

按照范围分区。

```
CREATE TABLE test_range_partition(
       id INT auto_increment,
       createdate DATETIME,
       primary key (id,createdate)
   )
```

```
   PARTITION BY RANGE (TO_DAYS(createdate) ) (
      PARTITION p201801 VALUES LESS THAN ( TO_DAYS('20180201') ),
      PARTITION p201802 VALUES LESS THAN ( TO_DAYS('20180301') ),
      PARTITION p201803 VALUES LESS THAN ( TO_DAYS('20180401') ),
      PARTITION p201804 VALUES LESS THAN ( TO_DAYS('20180501') ),
      PARTITION p201805 VALUES LESS THAN ( TO_DAYS('20180601') ),
      PARTITION p201806 VALUES LESS THAN ( TO_DAYS('20180701') ),
      PARTITION p201807 VALUES LESS THAN ( TO_DAYS('20180801') ),
      PARTITION p201808 VALUES LESS THAN ( TO_DAYS('20180901') ),
      PARTITION p201809 VALUES LESS THAN ( TO_DAYS('20181001') ),
      PARTITION p201810 VALUES LESS THAN ( TO_DAYS('20181101') ),
      PARTITION p201811 VALUES LESS THAN ( TO_DAYS('20181201') ),
      PARTITION p201812 VALUES LESS THAN ( TO_DAYS('20190101') )
   );
```

在 `/var/lib/mysql/data/` 可以找到对应的数据文件，每个分区表都有一个使用#分隔命名的表文件：

```
   -rw-r----- 1 MySQL MySQL    65 Mar 14 21:47 db.opt
   -rw-r----- 1 MySQL MySQL  8598 Mar 14 21:50 test_range_partition.frm
   -rw-r----- 1 MySQL MySQL 98304 Mar 14 21:50 test_range_partition#P#p201801.ibd
   -rw-r----- 1 MySQL MySQL 98304 Mar 14 21:50 test_range_partition#P#p201802.ibd
   -rw-r----- 1 MySQL MySQL 98304 Mar 14 21:50 test_range_partition#P#p201803.ibd
...
```

list分区

对于`List`分区，分区字段必须是已知的，如果插入的字段不在分区时枚举值中，将无法插入。

```
create table test_list_partiotion
   (
       id int auto_increment,
       data_type tinyint,
       primary key(id,data_type)
   )partition by list(data_type)
   (
       partition p0 values in (0,1,2,3,4,5,6),
       partition p1 values in (7,8,9,10,11,12),
       partition p2 values in (13,14,15,16,17)
   );
```

hash分区

可以将数据均匀地分布到预先定义的分区中。

```
create table test_hash_partiotion
   (
       id int auto_increment,
       create_date datetime,
       primary key(id,create_date)
   )partition by hash(year(create_date)) partitions 10;
```

分区的问题?

1. 打开和锁住所有底层表的成本可能很高。当查询访问分区表时，MySQL 需要打开并锁住所有的底层表，这个操作在分区过滤之前发生，所以无法通过分区过滤来降低此开销，会影响到查询速度。可以通过批量操作来降低此类开销，比如批量插入、`LOAD DATA INFILE` 和一次删除多行数据。
2. 维护分区的成本可能很高。例如重组分区，会先创建一个临时分区，然后将数据复制到其中，最后再删除原分区。
3. 所有分区必须使用相同的存储引擎。

查询语句执行流程?

查询语句的执行流程如下：权限校验、查询缓存、分析器、优化器、权限校验、执行器、引擎。

举个例子，查询语句如下：

```
select * from user where id > 1 and name = '大彬';
```

1. 首先检查权限，没有权限则返回错误；
2. MySQL8.0以前会查询缓存，缓存命中则直接返回，没有则执行下一步；
3. 词法分析和语法分析。提取表名、查询条件，检查语法是否有错误；
4. 两种执行方案，先查 `id > 1` 还是 `name = '大彬'`，优化器根据自己的优化算法选择执行效率最好的方案；
5. 校验权限，有权限就调用数据库引擎接口，返回引擎的执行结果。

更新语句执行过程?

更新语句执行流程如下：分析器、权限校验、执行器、引擎、`redo log`（`prepare` 状态）、`binlog`、`redo log`（`commit` 状态）

举个例子，更新语句如下：

```
update user set name = '大彬' where id = 1;
```

1. 先查询到 id 为1的记录，有缓存会使用缓存。
2. 拿到查询结果，将 name 更新为大彬，然后调用引擎接口，写入更新数据，innodb 引擎将数据保存在内存中，同时记录 `redo log`，此时 `redo log` 进入 `prepare` 状态。
3. 执行器收到通知后记录 `binlog`，然后调用引擎接口，提交 `redo log` 为 `commit` 状态。
4. 更新完成。

为什么记录完 `redo log`，不直接提交，而是先进入 `prepare` 状态?

假设先写 `redo log` 直接提交，然后写 `binlog`，写完 `redo log` 后，机器挂了，`binlog` 日志没有被写入，那么机器重启后，这台机器会通过 `redo log` 恢复数据，但是这个时候 `binlog` 并没有记录该数据，后续进行机器备份的时候，就会丢失这一条数据，同时主从同步也会丢失这一条数据。

exist和in的区别?

`exists`用于对外表记录做筛选。`exists`会遍历外表，将外查询表的每一行，代入内查询进行判断。当`exists`里的条件语句能够返回记录行时，条件就为真，返回外表当前记录。反之如果`exists`里的条件语句不能返回记录行，条件为假，则外表当前记录被丢弃。

```
select a.* from A awhere exists(select 1 from B b where a.id=b.id)
```

`in`是先把后边的语句查出来放到临时表中，然后遍历临时表，将临时表的每一行，代入外查询去查找。

```
select * from Awhere id in(select id from B)
```

子查询的表比较大的时候，使用`exists`可以有效减少总的循环次数来提升速度；**当外查询的表比较大的时候**，使用`in`可以有效减少对外查询表循环遍历来提升速度。

MySQL中int(10)和char(10)的区别?

int(10)中的10表示的是显示数据的长度，而char(10)表示的是存储数据的长度。

truncate、delete与drop区别?

相同点：

1. `truncate`和不带`where`子句的`delete`、以及`drop`都会删除表内的数据。
2. `drop`、`truncate`都是`DDL`语句（数据定义语言），执行后会自动提交。

不同点：

1. truncate 和 delete 只删除数据不删除表的结构；drop 语句将删除表的结构被依赖的约束、触发器、索引;
2. 一般来说，执行速度: drop > truncate > delete。

having和where区别?

- 二者作用的对象不同，`where`子句作用于表和视图，`having`作用于组。
- `where`在数据分组前进行过滤，`having`在数据分组后进行过滤。

什么是MySQL主从同步?

主从同步使得数据可以从一个数据库服务器复制到其他服务器上，在复制数据时，一个服务器充当主服务器（`master`），其余的服务器充当从服务器（`slave`）。

因为复制是异步进行的，所以从服务器不需要一直连接着主服务器，从服务器甚至可以通过拨号断断续续地连接主服务器。通过配置文件，可以指定复制所有的数据库，某个数据库，甚至是某个数据库上的某个表。

为什么要做主从同步?

1. 读写分离，使数据库能支撑更大的并发。
2. 在主服务器上生成实时数据，而在从服务器上分析这些数据，从而提高主服务器的性能。
3. 数据备份，保证数据的安全。

乐观锁和悲观锁是什么?

数据库中的并发控制是确保在多个事务同时存取数据库中同一数据时不破坏事务的隔离性和统一性以及数据库的统一性。乐观锁和悲观锁是并发控制主要采用的技术手段。

- 悲观锁：假定会发生并发冲突，在查询完数据的时候就把事务锁起来，直到提交事务。实现方式：使用数据库中的锁机制。
- 乐观锁：假设不会发生并发冲突，只在提交操作时检查是否数据是否被修改过。给表增加 `version` 字段，在修改提交之前检查 `version` 与原来取到的 `version` 值是否相等，若相等，表示数据没有被修改，可以更新，否则，数据为脏数据，不能更新。实现方式：乐观锁一般使用版本号机制或 `CAS` 算法实现。

用过processlist吗?

`show processlist` 或 `show full processlist` 可以查看当前 MySQL 是否有压力，正在运行的 `SQL`，有没有慢 `SQL` 正在执行。返回参数如下：

1. **id**：线程ID，可以用 `kill id` 杀死某个线程
2. **db**：数据库名称
3. **user**：数据库用户
4. **host**：数据库实例的IP
5. **command**：当前执行的命令，比如 `Sleep`， `Query`， `Connect` 等
6. **time**：消耗时间，单位秒
7. state
   ：执行状态，主要有以下状态：
   - Sleep，线程正在等待客户端发送新的请求
   - Locked，线程正在等待锁
   - Sending data，正在处理 `SELECT` 查询的记录，同时把结果发送给客户端
   - Kill，正在执行 `kill` 语句，杀死指定线程
   - Connect，一个从节点连上了主节点
   - Quit，线程正在退出
   - Sorting for group，正在为 `GROUP BY` 做排序
   - Sorting for order，正在为 `ORDER BY` 做排序
8. **info**：正在执行的 `SQL` 语句