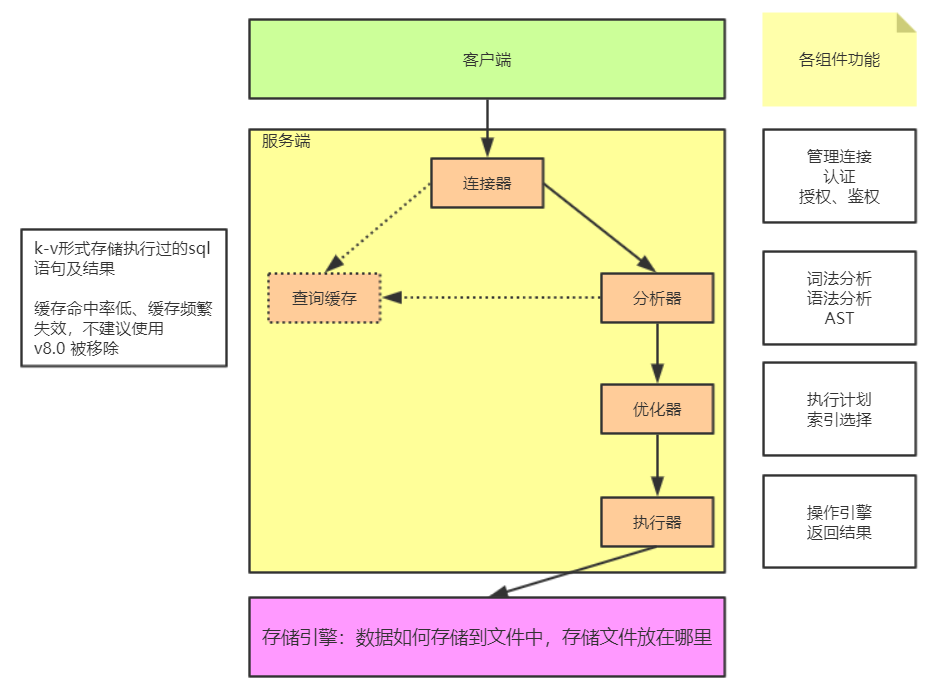
# =========基础=========

## Mysql架构 √



## 数据库的三大范式 √

**范式：**数据库设计时的一些规则。而这些规则是由一个姓范的人规定的，所以叫范式  
 三范式所追寻的原则，目的是：不存在冗余数据(同样的数据我不存第二遍)

 第一范式：要有主键 ，列不可分

 第二范式：不能存在部分依赖 【一个表只说明一个事物】

 第三范式：不能存在传递依赖【emp表除了dept\_id外不应该再有dept\_name】

## 左(外)连接、右(外)连接、内连接、笛卡儿积等；√

### 笛卡尔积：

笛卡尔发明的乘积。假设集合A={a, b}，集合B={0, 1, 2}，则两个集合的笛卡尔积为{(a, 0), (a, 1), (a, 2), (b, 0), (b, 1), (b, 2)}。

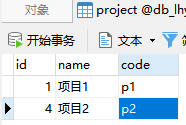
外连接：

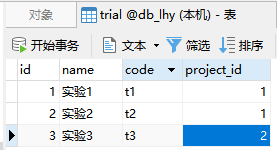
为了在操作时能保持这些将被舍弃的元组，提出了外连接的概念。使用外连接可以看到不满足连接条件的数据

### 左外连接left [outer] join：

左表信息将全部显示。outer可以省略，即left join默认就是left outer join

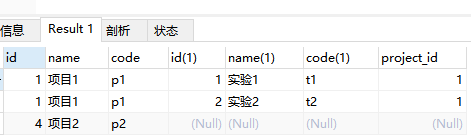
select \* from emp e left join dept d on e.deptno = d.deptno





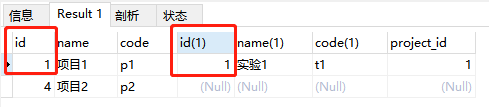
left join : 查出左表所有的和右表能关联上的

select \* from project t1 left join trial t2 on t1.id=t2.project\_id



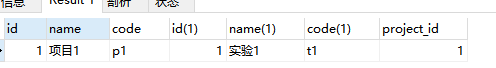
在on上再加一个and等值关联：查出了左表所有的，和右表符合符合t1.id=t2.project\_id and t1.id=t2.id的

select \* from project t1 left join trial t2 on t1.id=t2.project\_id and t1.id=t2.id



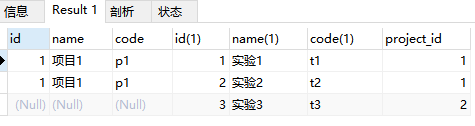
换成where ：查询出来的是，关联后，满足 t1.id=t2.id的

select \* from project t1 left join trial t2 on t1.id=t2.project\_id where t1.id=t2.id



Right join：查询的是右表所有的

select \* from project t1 right join trial t2 on t1.id=t2.project\_id



### 右外连接：

右表信息将全部显示

|  |
| --- |
| Oracle中支持全连接，即左右表全部信息都会现实出来，关键字是full outer join  mysql 中没有提供全连接的关键字，可以通过 "左外连接结果表 union 右外连接结果表" 的方式实现全连接 |

### 自连接：

一张表自己和自己连接

select e.ename, m.ename mgr from emp e, emp m where e.mgr = m.empno

### 内连接[inner] join

将两张表通过where或on或using指定连接条件进行连接

select \* from emp e join dept d on e.deptno = d.deptno

select \* from emp e JOIN dept d USING(deptno)

select \* from emp e, dept d where e.deptno = d.deptno;

### 等值连接：

连接条件用on或using或where指定，通过=进行连接

select \* from emp e join dept d on e.deptno = d.deptno

select \* from emp e JOIN dept d USING(deptno)

select \* from emp e, dept d where e.deptno = d.deptno;

### 非等值连接：

连接条件用 where 指定，将值不等的两列按照一定规则连接起来

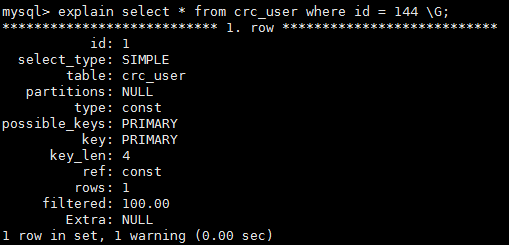
select e.deptno, e.ename, e.sal, sg.grade from emp e join salgrade sg where e.sal between sg.losal and sg.hisal

## DDL、DML、DCL分别指什么 √

1. DDL：数据定义语言，create，alert，drop，truncate
2. DML：数据操纵语言，select，insert，update，delete
3. DCL：数据控制语言，grant，revoke

## explain命令 √

explain显示了mysql如何使用索引来处理select语句以及连接表。可以帮助选择更好的索引和写出更优化的查询语句



## drop、truncate、delete的区别 √

Drop：删除表

Truncate：保留表结构，删除表数据

Delete 所有数据 和truncate区别：

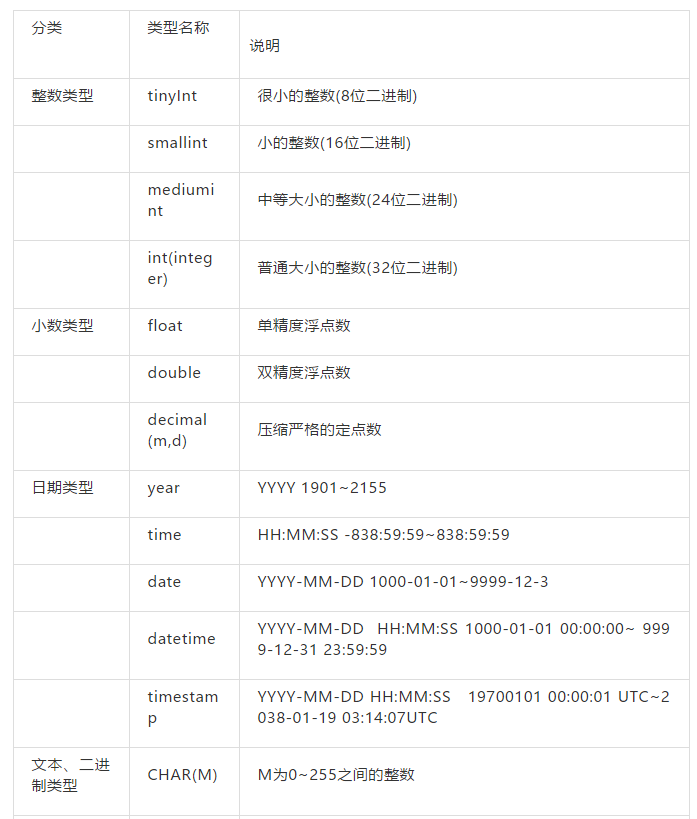
|  |  |  |
| --- | --- | --- |
| 区别 | Delete | truncate |
| 事务 | 经过事务，数据可以回滚 | 不经过事务，一旦删除永久消失，不支持回滚 |
| 效率 | 一行一行的删除，效率低 | 重建表结构 效率高 |
| 返回行数 | 返回删除掉的行数 | -- |
| 触发器 |  | 不触发任何 delete的触发器 |
| 主键自增时 | Delete后主键按照原来的值继续递增 | 主键将重新从 1 开始递增 |

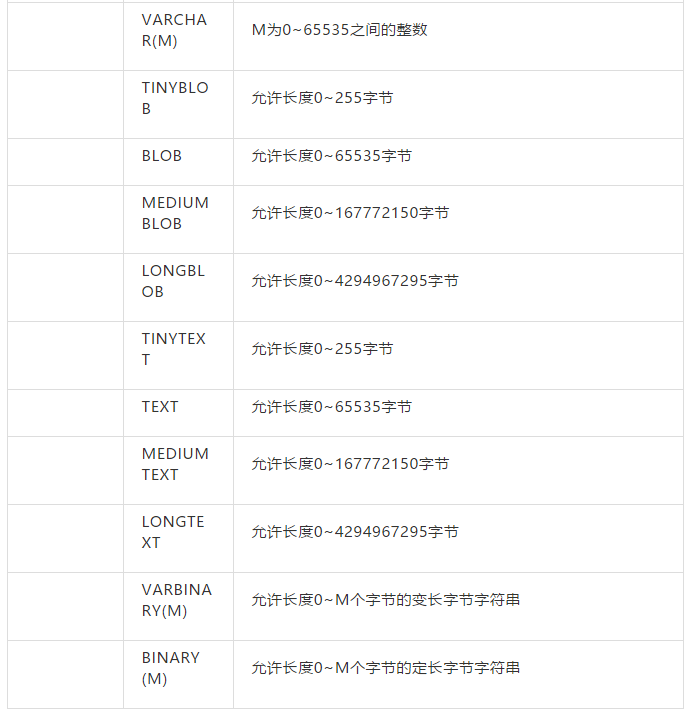
truncate虽然效率高，但容易发生误操作，不建议使用

## Exists √

exits(sub-query)，放在where后面作为一个条件判断。当exists后面的子查询能查出结果时，该条件即为真，通过where条件

## mysql有哪些数据类型 √





1、整数类型，包括TINYINT、SMALLINT、MEDIUMINT、INT、BIGINT，分别表示1字节、2字节、3字节、4字节、8字节整数。任何整数类型都可以加上UNSIGNED属性，表示数据是无符号的，即非负整数。

长度：整数类型可以被指定长度，例如：INT(11)表示长度为11的INT类型。长度在大多数场景是没有意义的，它不会限制值的合法范围，只会影响显示字符的个数，而且需要和UNSIGNED ZEROFILL属性配合使用才有意义。

例子，假定类型设定为INT(5)，属性为UNSIGNED ZEROFILL，如果用户插入的数据为12的话，那么数据库实际存储数据为00012。

2、实数类型，包括FLOAT、DOUBLE、DECIMAL。

DECIMAL可以用于存储比BIGINT还大的整型，能存储精确的小数。

而FLOAT和DOUBLE是有取值范围的，并支持使用标准的浮点进行近似计算。

计算时FLOAT和DOUBLE相比DECIMAL效率更高一些，DECIMAL你可以理解成是用字符串进行处理。

3、字符串类型，包括VARCHAR、CHAR、TEXT、BLOB

VARCHAR用于存储可变长字符串，它比定长类型更节省空间。

VARCHAR使用额外1或2个字节存储字符串长度。列长度小于255字节时，使用1字节表示，否则使用2字节表示。

VARCHAR存储的内容超出设置的长度时，内容会被截断。

CHAR是定长的，根据定义的字符串长度分配足够的空间。

CHAR会根据需要使用空格进行填充方便比较。

CHAR适合存储很短的字符串，或者所有值都接近同一个长度。

CHAR存储的内容超出设置的长度时，内容同样会被截断。

使用策略：

对于经常变更的数据来说，CHAR比VARCHAR更好，因为CHAR不容易产生碎片。

对于非常短的列，CHAR比VARCHAR在存储空间上更有效率。

使用时要注意只分配需要的空间，更长的列排序时会消耗更多内存。

尽量避免使用TEXT/BLOB类型，查询时会使用临时表，导致严重的性能开销。

4、枚举类型（ENUM），把不重复的数据存储为一个预定义的集合。

有时可以使用ENUM代替常用的字符串类型。

ENUM存储非常紧凑，会把列表值压缩到一个或两个字节。

ENUM在内部存储时，其实存的是整数。

尽量避免使用数字作为ENUM枚举的常量，因为容易混乱。

排序是按照内部存储的整数

5、日期和时间类型，尽量使用timestamp，空间效率高于datetime，

用整数保存时间戳通常不方便处理。

如果需要存储微妙，可以使用bigint存储。

看到这里，这道真题是不是就比较容易回答了。

## 视图：√

### 定义

视图(view)，也称虚表, 不占用物理空间，这个也是相对概念，因为视图本身的定义语句还是要存储在数据字典里的。视图只有逻辑定义，每次使用的时候,，只是重新执行SQL

视图是从一个或多个实际表中获得的，这些表的数据存放在数据库中。那些用于产生视图的 表叫做该视图的基表。一个视图也可以从另一个视图中产生。

视图的定义存在数据库中，与此定义相关的数据并没有再存一份于数据库中。通过视图看到 的数据存放在基表中。

视图看上去非常象数据库的物理表，对它的操作同任何其它的表一样。当通过视图修改数据时，实际上是在改变基表中的数据；相反地，基表数据的改变也会自动反映在由基表产生的视图中。

|  |
| --- |
| /\* 创建视图  CREATE [OR REPLACE] VIEW view  [(alias[, alias]...)]  AS subquery  [WITH READ ONLY];  \*/  create or replace view v\_avg\_sal as select deptno, avg(sal) asal from emp group by deptno;  --使用视图，直接用视图名称代替原来的 sql 语句  select \* from emp e, v\_avg\_sal t where e.sal > t.asal and t.deptno = e.deptno;  --查找视图的sql语句  SELECT view\_definition FROM information\_schema.views WHERE table\_name = 'v\_avg\_sal';  --查找视图表内容  select \* from v\_avg\_sal;  --查看所有视图的信息  show table status where comment ='view'  --更改视图数据，和更改表数据一样  create view v\_emp as select \* from emp;  insert into v\_emp(empno, ename) values(1111,'Ryan');  select \* from emp;  --删除视图  drop view v\_avg\_sal; |

### 注意点

视图是一种虚表，不占物理空间，通过视图看到的数据都来自于基表

修改视图的时候，也会修改对应基表数据。同样，修改基表也会体现在视图中

若一个视图依赖于多个基本表，则一次修改或删除该视图只能修改一个基本表的数据，否则 sql 语句执行会报错Can not modify more than one base table through a join view 'dept.v\_emp\_dept'

当视图执行的语句本身就需要执行时，利用视图是不会影响查询效率的。除非进行了多余的语句执行，会影响效率，比如下面一个例子，emp 其实被查询了两次。

|  |
| --- |
| create v\_emp as select \* from emp;  --此处视图其实是查询了两次emp  select \* from v\_emp;  --相当于以下语句  select \* from (select \* from emp) e; |

# =========事务===========

## 什么是数据库事务？√

事务（Transaction）是一个操作序列（集合）。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，是数据库环境中的逻辑工作单位。

## 事务控制语句 √

**事务控制语句包含以下 7 个关键字：**

1. BEGIN 显式地开启一个事务
2. COMMIT 会提交事务，并使已对数据库进行的所有修改成为永久性的
3. ROLLBACK ，会进行回滚。回滚会结束用户的事务，并撤销正在进行的所有未提交的修改
4. SAVEPOINT identifier，允许在事务中创建一个保存点，一个事务中可以有多个 SAVEPOINT
5. RELEASE SAVEPOINT identifier 删除一个事务的保存点，当没有指定的保存点时，执行该语句会抛出一个异常
6. ROLLBACK TO identifier 把事务回滚到保存点
7. SET TRANSACTION 用来设置事务的隔离级别。InnoDB 存储引擎提供事务的隔离级别有READ UNCOMMITTED、READ COMMITTED、REPEATABLE READ 和 SERIALIZABLE

**事务的结束可由以下操作进行触发：**

1. 用户显式执行 commit 语句提交操作或 Rollback 语句回退
2. 开启了自动提交，执行了DML语句后

通过SET AUTOCOMMIT=1 可以设置开启自动提交，但是一般会关闭自动提交，因为效率太低

1. 执行DDL语句，事务会自动提交
2. 用户正常断开连接时，Transaction自动提交
3. 系统崩溃或断电时事务自动回退

|  |
| --- |
| 一个事务语句示例： |

## 锁机制 √

事务可以保证在并发访问下，数据的一致性。

事务底层是用过锁机制做到数据一致性的。

加锁时需要考虑锁 粒度 问题，常见的数据库锁粒度包括：数据库、表、行

一般情况下，锁粒度越小，效率越高，逻辑控制越复杂。粒度越大，效率越低，逻辑控制越简单。在实际工作环境中，大部分操作都是行级锁。

## 数据库事物ACID（原子性、一致性、隔离性、持久性） √

**原子性：**事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；

**一致性：**执行事务前后，数据保持一致，多个事务对同一个数据读取的结果是相同的；

**隔离性：**并发访问数据库时，一个用户的事务不被其他事务所干扰，各并发事务之间数据库是独立的；

**持久性：**一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响。

## 事物的隔离级别 √

**关系型数据库常存在四种事务隔离级别：**

1. READ UNCOMMITTED：读未提交
2. READ COMMITTED：读已提交
3. **REPEATABLE READ ：可重复读（mysql 默认值）**
4. SERIALIZABLE：序列化，串行执行，读操作一定要等写操作执行完成

从上往下隔离级别越来越高，意味着数据越来越安全。

如何查看当前 session 的隔离级别是什么？

8.0之前：select @@tx\_isolation;

8.0之后：select @@transaction\_isolation;

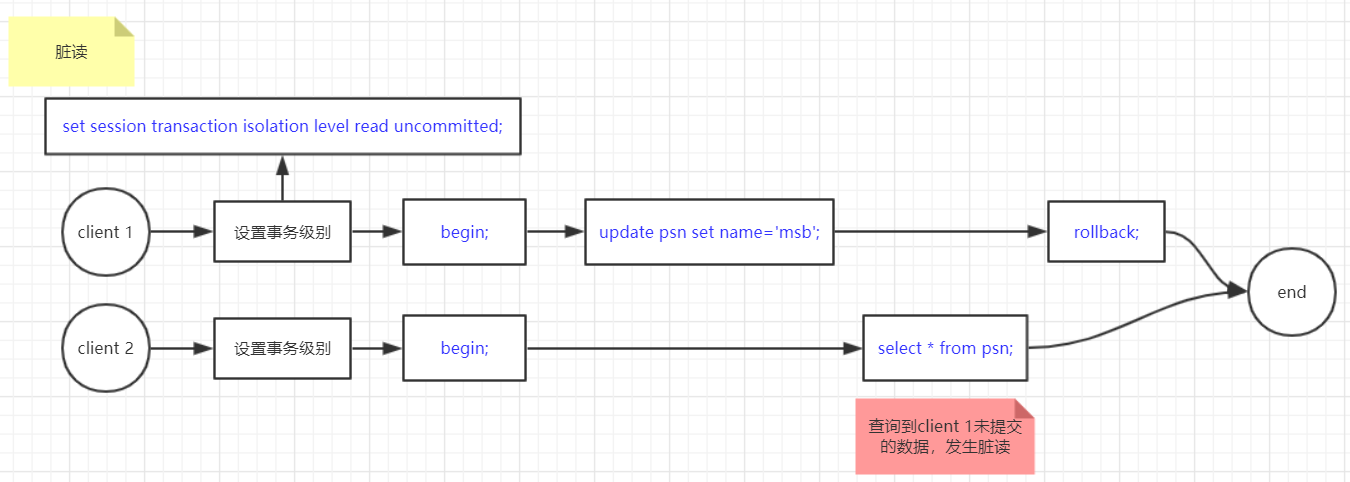
如何设置事务的隔离级别？

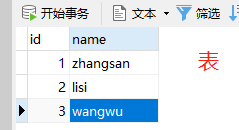
set global transaction isolation level read uncommitted

执行完该语句后，当前 session 的隔离级别不会立刻更新，需要重新开连接，或者做 session 级别的调整：set session transaction isolation level read uncommitted

不同的隔离级别会产生不同的问题，具体有如下 3 种数据不一致问题：

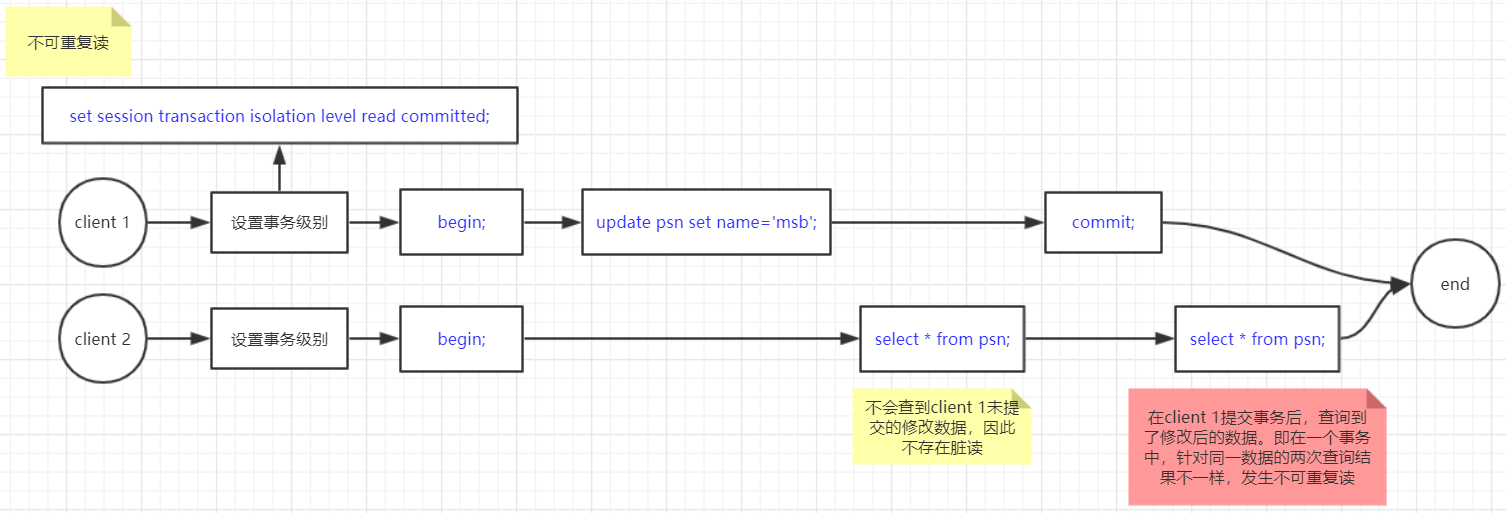
### 脏读、幻读、不可重复读

**脏读(Drity Read)：**(如有事务A和B，A读取了B未提交的数据) 





**不可重复读(Non-repeatable read):** (如有事务A和B，A负责读取，B负责写入，A连续读的过程中B写入了一次，A前后两次读出来的数据不一样)。



set session transaction isolation level read committed;

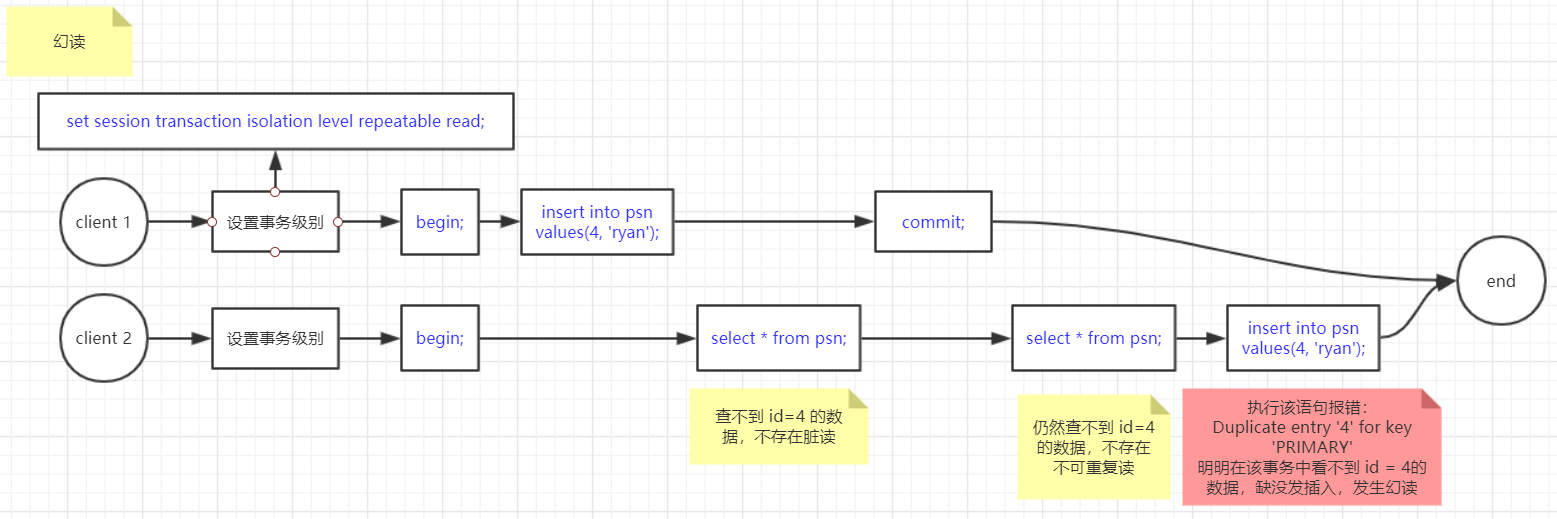
begin;

update psn set name='msb';

select \* from psn;

commit;

**幻读(Phantom Read):**在一个事务的两次查询中数据笔数不一致，例如有一个事务查询了几列(Row)数据，而另一个事务却在此时插入了新的几列数据，先前的事务在接下来的查询中，就会发现有几列数据是它先前所没有的。



不可重复读和幻读的差别在于侧重点不同，前者侧重于修改，后者侧重于新增、删除

set session transaction isolation level repeatable read;

begin;

insert into psn values(4, 'ryan');

select \* from psn;

commit;

insert into psn values(4, 'ryan');

## Mysql怎么解决幻读的？√ \*\*\*五颗星

概念

**MySQL InnoDB支持三种行锁定方式：**

* **行锁（Record Lock）**：锁直接加在索引记录上面。
* **间隙锁（Gap Lock）**：锁加在不存在的空闲空间，可以是两个索引记录之间，也可能是第一个索引记录之前或最后一个索引之后的空间。
* **Next-Key Lock：**行锁与间隙锁组合起来用就叫做Next-Key Lock。

当我们用范围条件而不是相等条件检索数据，并请求共享或排他锁时，InnoDB会给符合条件的已有数据记录的索引项加锁；对于键值在条件范围内但并不存在的记录，叫做“间隙（GAP)”，InnoDB也会对这个“间隙”加锁，这种锁机制就是所谓的间隙锁（Next-Key锁）。

举例来说，假如user表中只有101条记录，其empid的值分别是 1,2,...,100,101，下面的SQL：

select \* from  user where user\_id > 100 for update;

是一个范围条件的检索，InnoDB不仅会对符合条件的user\_id值为101的记录加锁，也会对user\_id大于101（这些记录并不存在）的“间隙”加锁。

产生幻读的原因是，行锁只能锁住行，但是新插入记录这个动作，要更新的是记录之间的“间隙”。因此，为了解决幻读问题，InnoDB 只好引入新的锁，也就是间隙锁 (Gap Lock)。

InnoDB使用间隙锁的目的，一方面是为了防止幻读，以满足相关隔离级别的要求，对于上面的例子，要是不使用间隙锁，如果其他事务插入了user\_id大于100的任何记录，那么本事务如果再次执行上述语句，就会发生幻读；另外一方面，是为了满足其恢复和复制的需要

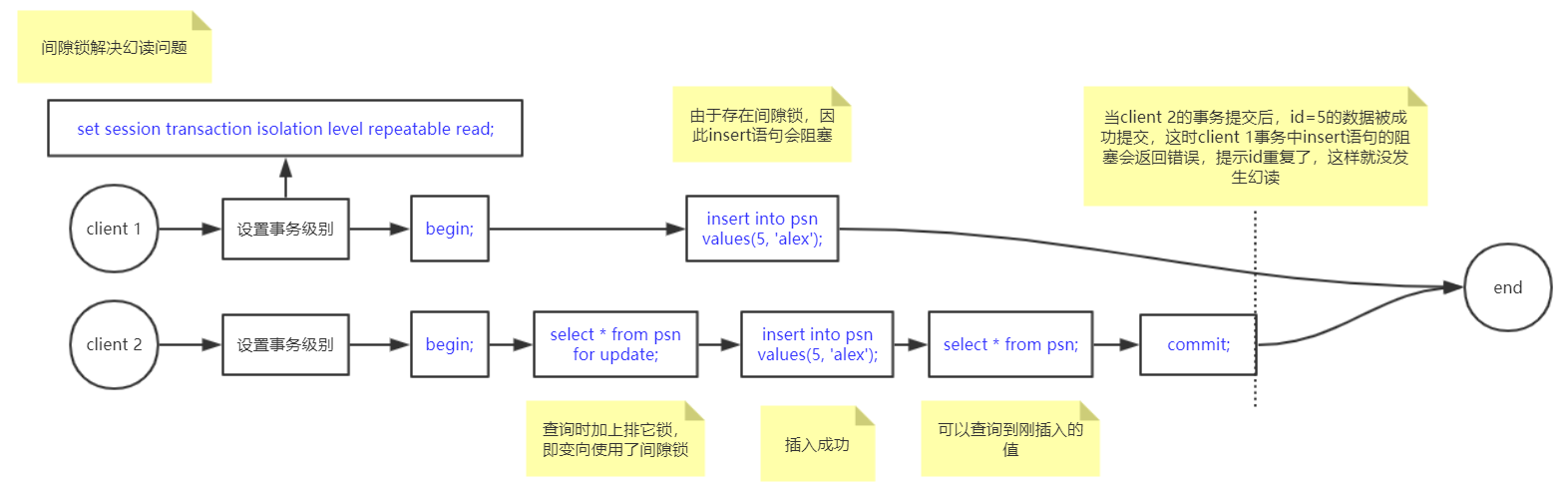
除了把事务隔离级别改成SERIALIZABLE外，还可以通过间隙锁解决幻读问题

间隙锁(Gap Lock)：锁住两行纪录之间的空隙，以避免数据新增、删除和更新操作。只在REPEATABLE READ级别下有用，解决幻读问题。

Next-Key Lock：单行锁（Record Lock）+间隙锁（Gap Lock）合起来就叫Next-Key Lock。

通过innodb\_locks\_unsafe\_for\_binlog参数配置为0，代表开启间隙锁；值为1代表关闭间隙锁，在MySQL8.0之前默认值为0，该参数在 8.0 版本被移除，说明一直是打开的。

解决幻读的实验步骤如下：



set session transaction isolation level repeatable read;

begin;

select \* from psn for update;

insert into psn values(5, 'alex');

select \* from psn;

commit;

|  |
| --- |
|  |

## 什么是事务的隔离级别？MySQL的默认隔离级别是什么？ √

为了达到事务的四大特性，数据库定义了4种不同的事务隔离级别，由低到高依次为Read uncommitted、Read committed、Repeatable read、Serializable，这四个级别可以逐个解决脏读、不可重复读、幻读这几类问题。



### SQL 标准定义了四个隔离级别：

* **READ-UNCOMMITTED(读取未提交)：**

最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读。

* **READ-COMMITTED(读取已提交)：**

允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生。

* **REPEATABLE-READ(可重复读)：**

对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。

* **SERIALIZABLE(可串行化)：**

最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。

这里需要注意的是：Mysql 默认采用的 REPEATABLE\_READ隔离级别 Oracle 默认采用的 READ\_COMMITTED隔离级别

事务隔离机制的实现基于锁机制和并发调度。其中并发调度使用的是MVVC（多版本并发控制），通过保存修改的旧版本信息来支持并发一致性读和回滚等特性。

因为隔离级别越低，事务请求的锁越少，所以大部分数据库系统的隔离级别都是READ-COMMITTED(读取提交内容):，但是你要知道的是InnoDB 存储引擎默认使用 \*\*REPEATABLE-READ（可重读）\*\*并不会有任何性能损失。

InnoDB 存储引擎在 分布式事务 的情况下一般会用到\*\*SERIALIZABLE(可串行化)\*\*隔离级别。

## 数据库中的隔离性是怎样实现的；

参照《数据库原理》

实现隔离级别的两种锁模式：共享锁和排他锁，这两种锁都是悲观锁，共享锁也叫S锁，是一种读锁，当一个事物获得了一条数据的共享锁，其它事务也能获得该共享锁，但不能获得排他锁，表明其它事务可读，但不可写。排他锁也叫X锁，是一种写锁，当一个事务对临界区加上排他锁，其它事务就不能获得该临界区的任何锁（共享锁，排他锁）。总结一下就是共享锁保证大家可以一起读，但只能一个人写，排他锁保证只能一个人去处理数据，其他人不能读也不能写。

## 原子性、一致性、持久性又是如何实现的；

**原子性：**undolog可以保证事务的原子性特点

**一致性：**两阶段提交保证了两个日志的一致性，要么都成功，要么都失败

**持久性：**根据binlog及redolog中记录的每个事务的状态，实现已提交的事务数据不丢失、未提交的事务数据进行回滚，这叫做 crash-safe，保证了事务的持久性特点

## Mysql如何解决幻读 √

除了把事务隔离级别改成SERIALIZABLE外，还可以通过间隙锁解决幻读问题

间隙锁(Gap Lock)：锁住两行纪录之间的空隙，以避免数据新增、删除和更新操作。只在RR级别下有用，解决幻读问题。

Next-Key Lock：单行锁（Record Lock）+间隙锁（Gap Lock）合起来就叫Next-Key Lock。

通过innodb\_locks\_unsafe\_for\_binlog参数配置为0，代表开启间隙锁；值为1代表关闭间隙锁，在MySQL8.0之前默认值为0，该参数在 8.0 版本被移除，说明一直是打开的。

# =========索引============

索引来源

|  |  |
| --- | --- |
| 常识：  |  | | --- | | 计算机数据是存在磁盘，磁盘的维度有两个指标：   1. 寻址（速度毫秒级） 2. 带宽 单位时间可以有多少字节流过去，多少数据量流过去。G/M 级别   内存   1. 寻址 纳秒   2-内存的带宽（内存数据往外走） 很大  （秒--> 毫秒--》微秒--》纳秒）【磁盘比内存寻址慢了10w倍】  I/O buffer： 成本问题  磁盘有磁道和扇区，一个扇区512Byte  有一个成本问题，如果访问一个硬盘，都是以一个扇区512字节来找，一个硬盘1T有很多个512Byte，要找到你存放数据的那个扇区，索引成本很大。  操作系统，无论你读多少，都是最少4K从磁盘上拿 （类比CPU缓存行的概念） |  **文件存数据** 最早的时候，数据存在文件里，比如data.txt ，Linux里有grep 、ark，你还可以用java写一个程序，基于IO流的文件查找。  随着文件的变大，读取速度变慢，为什么？因为文件变大，会受到硬盘读取的瓶颈，（I/O成为瓶颈） 数据库存储 数据库有data page，大小是4K。    数据库的表，行记录存到磁盘的时候，用了很多4K 4K的小格子，这个4K刚好和磁盘的一次读取大小4K对应上，这个4K定义小的话是浪费，应为读取磁盘一次还是4K，定义大于4K比如8K 16K无所谓。  你有1w行数据，会散落在这一个个4K的小格子里，光有小格子的话，查找数据的成本、复杂度，还和直接读取文件一样，还是要将一个个4K都读到内存，挨个的去找数据。走的还是全量IO，跟直接读取文件的IO量是一样的，所以一定会很慢。所以数据库只是建表了，没有建索引的话，读取速度还是很慢。  提速的话用索引，索引用的也是4K存储模型。索引存储在硬盘  比如你在身份证号上建立索引，    关系型数据库，建表的时候，必须给出建表schema建表语句，给出一共有多少列，每个列的类型，类型：规定了字节宽度。只要给出了schema，每一行数据的宽度就定死了。  这样的话，假设一行有10个字段，你插入一行，你只给出第一个和第七个字段，其他字段都没有给，数据向data page放的时候，那些没有数据的字段都会用0去开辟，用空的东西去补充那些字节，进行占位。好处是，将来进行增删改的时候，不用去移动数据，直接拿新的数据把原来位置进行复写就可以了。  明确：数据和索引都是在硬盘里的  内存是速度最快的地方，在内存里准备一个B+Tree，B+Tree树干是在内存里的，只存一些区间，数据区间和偏移，sql的where条件里只要命中索引了，查询在B+Tree会找到某一个页，这个页刚好在这个B+Tree区间里，然后把索引从磁盘读到内存，然后就找到哪个data page，把这个data page读到内存里，就找到记录了。  索引和数据在磁盘，B+Tree在内存，分而治之。减少磁盘IO的流量。  如果数据库表有几千万行，表很大，性能降低？如果表有索引  增删改变慢：需要找到索引，调整位置  查询速度？假设表有1T，B+Tree也能在内存装下，哪都没有溢出，没有问题   1. 一个人或者少量来查询，简单sql，where条件能够命中索引，查询依然很快。还是从内存的B+Tree，找到索引文件，再找到data page，带到内存。并不会因为数据量的变大，把别的data page也带到内存里去。 2. 并发大 的时候：会受到**硬盘带宽影响**速度    1. 大量sql查询不同的data page，每个data page的4K，是挨个的向内存去走，是并行的。（MySQL8.0引入了并行查询虽然还比较初级）此时就会有一个部分查询等待。    2. 或者复杂的sql到达：查询的时候不是要获取一个data page到内存了，而是多个data page到内存。   商业内存关系数据库  SAP公司的HANA数据库。内存级别的关系型数据库，内存约2T，硬件软件服务总和约2亿一个套餐。  数据在磁盘和内存中体积不一样。磁盘没有指针的概念。同个文件内存占用的更少。 |

## 什么是索引？ √

### 概念：

索引是为了加快对数据的搜索速度而设立的。索引是方案（schema）中的一个数据库对象，与表独立存放。

### 索引的作用：

在数据库中用来加速对表的查询，通过使用快速路径访问方法快速定位数据，减少了磁盘的I/O

### 索引创建的两种情况：

1. 自动: 当在表上定义一个 PRIMARY KEY 或者 UNIQUE 约束条件时，数据库自动创建一个对应的唯一索引
2. 手动: 用户可以创建索引以加速查询

### 索引要点：

* 1. 索引改善检索操作的性能，但降低数据插入、修改和删除的性能。在执行这些操作时，MySQL 必须动态地更新索引
  2. 索引数据可能要占用大量的存储空间
  3. 并非所有的数据都适合于索引。唯一性不好的数据（如省）从索引得到的好处不比具有更多可能值的数据（如姓名）从索引得到的好处多
  4. 索引用于数据过滤和数据排序。如果经常以某种特定的顺序排序数据，则该数据可能是索引的备选
  5. 可以在索引中定义多个列（如省加城市），这样的索引只在以省加城市的顺序排序时有用。如果想按城市排序，则这种索引没有用处
  6. 如果 sql 语句中存在运算，那么索引会无效，因此要避免索引无效的情况

## 索引存储在哪里？|| 聚簇索引与非聚簇索引索引的区别 √

要想知道索引存在哪里，就必须知道存储引擎是什么。存储引擎存储了 mysql 的所有数据（包括索引、表结构、表数据），不同的存储引擎会有不同的文件格式，不同的存放位置，因此按照存方式可将索引分为两类：

### 聚簇索引：数据和索引放在一个文件里

聚簇索引是将索引和整条记录存放在一起，找到索引就找到了记录；

非聚簇索引只存储索引字段和记录所在的位置，通过索引找到记录所在的位置，然后再根据记录所在位置去获取记录。

一般来讲一堆数据记录最多只能有一个聚簇索引，但可以有很多非聚簇索引；

Innobd中的主键索引是一种聚簇索引，非聚簇索引都是辅助索引，像复合索引、前缀索引、唯一索引。

Innodb使用的是聚簇索引，MyISam使用的是非聚簇索引

### **聚簇索引(聚集索引)**

　　聚簇索引就是按照每张表的主键构造一颗B+树，同时叶子节点中存放的就是整张表的行记录数据，也将聚集索引的叶子节点称为数据页。这个特性决定了索引组织表中数据也是索引的一部分，每张表只能拥有一个聚簇索引。

　　Innodb通过主键聚集数据，如果没有定义主键，innodb会选择非空的唯一索引代替。如果没有这样的索引，innodb会隐式的定义一个主键来作为聚簇索引。

聚簇索引的优缺点

　　优点：

　　　　1.数据访问更快，因为聚簇索引将索引和数据保存在同一个B+树中，因此从聚簇索引中获取数据比非聚簇索引更快

　　　　2.聚簇索引对于主键的排序查找和范围查找速度非常快  
　　缺点：

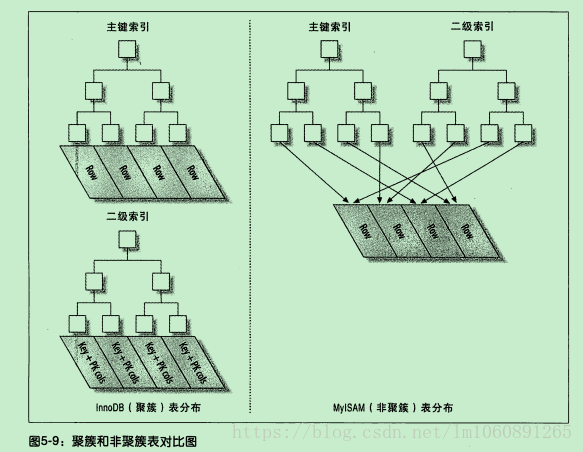
　　　　1.插入速度严重依赖于插入顺序，按照主键的**顺序插入**是最快的方式，否则将会出现页分裂，严重影响性能。因此，对于InnoDB表，我们一般都会定义一个**自增的ID列为主键**  
　　　　2.**更新主键的代价很高**，因为将会导致被更新的行移动。因此，对于InnoDB表，我们一般定义主键为不可更新。  
　　　　3.二级索引访问需要两次索引查找，第一次找到主键值，第二次根据主键值找到行数据。

### **辅助索引（非聚簇索引）**

　　在**聚簇索引之上创建的索引称之为辅助索引**，辅助索引访问数据总是需要二次查找。辅助索引叶子节点存储的不再是行的物理位置，而是主键值。通过辅助索引首先找到的是主键值，再通过主键值找到数据行的数据页，再通过数据页中的Page Directory找到数据行。

　　Innodb辅助索引的叶子节点并**不包含行记录的全部数据**，叶子节点除了包含键值外，还包含了相应行数据的聚簇索引键。

　　辅助索引的存在不影响数据在聚簇索引中的组织，所以一张表可以有多个辅助索引。在innodb中有时也称辅助索引为二级索引。



## 索引有哪些优缺点？ √

### 索引的优点

* 可以大大加快数据的检索速度，这也是创建索引的最主要的原因。
* 通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

### 索引的缺点

* **时间方面：**创建索引和维护索引要耗费时间，具体地，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，会降低增/改/删的执行效率；
* **空间方面：**索引需要占物理空间。

## 索引使用场景（重点）√

### Where



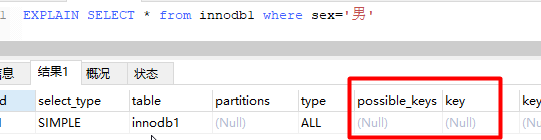
上图中，根据id查询记录，因为id字段仅建立了主键索引，因此此SQL执行可选的索引只有主键索引，如果有多个，最终会选一个较优的作为检索的依据。

-- 增加一个没有建立索引的字段

alter table innodb1 add sex char(1);

-- 按sex检索时可选的索引为null

EXPLAIN SELECT \* from innodb1 where sex='男';



可以尝试在一个字段未建立索引时，根据该字段查询的效率，然后对该字段建立索引（alter table 表名 add index(字段名)），同样的SQL执行的效率，你会发现查询效率会有明显的提升（数据量越大越明显）。

### order by

当我们使用order by将查询结果按照某个字段排序时，如果该字段没有建立索引，那么执行计划会将查询出的所有数据使用外部排序（将数据从硬盘分批读取到内存使用内部排序，最后合并排序结果），这个操作是很影响性能的，因为需要将查询涉及到的所有数据从磁盘中读到内存（如果单条数据过大或者数据量过多都会降低效率），更无论读到内存之后的排序了。

但是如果我们对该字段建立索引alter table 表名 add index(字段名)，那么由于索引本身是有序的，因此直接按照索引的顺序和映射关系逐条取出数据即可。而且如果分页的，那么只用取出索引表某个范围内的索引对应的数据，而不用像上述那取出所有数据进行排序再返回某个范围内的数据。（从磁盘取数据是最影响性能的）

### Join

对join语句匹配关系（on）涉及的字段建立索引能够提高效率

### 索引覆盖

如果要查询的字段都建立过索引，那么引擎会直接在索引表中查询而不会访问原始数据（否则只要有一个字段没有建立索引就会做全表扫描），这叫索引覆盖。因此我们需要尽可能的在select后只写必要的查询字段，以增加索引覆盖的几率。

这里值得注意的是不要想着为每个字段建立索引，因为优先使用索引的优势就在于其体积小。

## 索引分类 √

mysql 索引分为五类：

1. **主键索引:**

数据列不允许重复，不允许为NULL，一个表只能有一个主键。

当表没有主键时，会找唯一键，建立唯一索引。

当表没有主键也没有唯一键时，mysql 自动创建一个rowid列作为主键并建立主键索引。该列对用户不可见，对系统可见

主键无特殊情况时，建议设置“自动递增”，这样主键索引维护起来成本较低，不存在页分裂、页合并等影响性能的操作【当按照索引顺序插入时，page的填充率能达到15/16 ， 而乱序插入时只能到略大于 1/2 的填充率。会带来10%的TPS提升，并减少50%的空间浪费。】

主键的"自动递增"功能，通过"自增锁"实现

1. **唯一索引:**

数据列不允许重复，允许为NULL值

1. **普通索引：**

基本的索引类型，值可以为空

存在回表操作，需要遍历两次 B+ 树

覆盖索引：**要查询的列被所建的索引覆盖**。建立索引后，如果要查询非主键字段，则需要回表。如果只查询主键值或者索引的值（如 sql 语句是select id from emp where ename='KING'），此时直接可以从索引数据中获取，这就称为 "覆盖索引"。在InnoDB中因为覆盖索引少了一次回表，因此效率会提高。

1. **全文索引**

索引类型为“FULLTEXT”，可在“varchar”、“char”、“text”类型上创建

5.6之前，全文索引只存在与MyISAM中；5.6 开始，Innodb也开始支持全文索引

比较少用，本身 mysql 做大文本存储就很少用

1. **组合索引**

多列值组成一个索引，专门用于组合搜索

最左匹配原则：对于联合索引 (a,b,c) ，查询中能用到索引的是：a；ab；abc，ac。

对于其他顺序,或者其他字段我们不能使用该联合索引,这个就是mysql联合索引原则

|  |
| --- |
| **拓展：组合索引的优化选择**  注意：select \* from text where name like '%a'是不会走name索引的，而select \* from text where name like 'a%'会走name索引。  如果有个场景，存在根据name的查询、根据name和age的查询、根据age的查询，那么方案就有两种  1：组合索引name age，普通索引age  2：组合索引age name，普通索引name  如何选择？从空间考虑  从效率上考虑，两者几乎无差别。从空间上考虑，因为name字段长度长，因此方案1 更省空间，1 更优 |

**拓展**

## 组合索引的【索引下推】 \*\*\*\* √

|  |
| --- |
| 索引下推，5.7版本开始支持，前提是必须存在组合索引  假设我们有一个user表，建立了(username, age, sex)的联合索引。此时有这样一个语句select \* from user where username like 'M%' and sex = 1，这条语句的执行有 2 种方式：  根据联合索引，先找到名字符合条件的索引，然后回表获取整行数据，最后根据sex进行删选  根据联合索引，先找到满足名字条件的索引，然后直接筛选符合sex的索引（因为联合索引里有sex的值），最后回表获取整行数据  显然 2 回表时查询的全行数据较少，效率更高，这就是 mysql 的索引下推。默认索引下推是开启的，可通过select @@optimizer\_switch;查询是否开启，通过SET global optimizer\_switch = 'index\_condition\_pushdown=off';可以关闭索引下推  注意：如果语句是select \* from user where username like 'M%' and age > 18，那么直接利用联合索引(username, age, sex)和最左匹配原则，直接筛选出了符合username like 'M%' and age > 18的数据 |

可以通过 ALTER TABLE table\_name ADD UNIQUE (column); 创建唯一索引

可以通过 ALTER TABLE table\_name ADD UNIQUE (column1,column2); 创建唯一组合索引

普通索引: 基本的索引类型，没有唯一性的限制，允许为NULL值。

可以通过ALTER TABLE table\_name ADD INDEX index\_name (column);创建普通索引

可以通过ALTER TABLE table\_name ADD INDEX index\_name(column1, column2, column3)创建组合索引

全文索引：是目前搜索引擎使用的一种关键技术。

可以通过ALTER TABLE table\_name ADD FULLTEXT (column);创建全文索引

## 索引的维护，你说的维护索引需要什么成本？ √

B+树为了维护索引有序性，在插入新值的时候需要做必要的维护

在维护索引的时候需要需要分以下几种情况：

* 如果插入一个比较大的值，直接插入即可，几乎没有成本
* 如果插入的是中间的某一个值，需要逻辑上移动后续的元素，空出位置
* 如果需要插入的数据页满了，就需要单独申请一个新的数据页，然后移动部分数据过去，叫做页分裂，此时性能会受影响，同时空间的使用率也会降低，除了页分裂之外还包含页合并

因此，尽量使用自增主键作为索引，自增 id 可以避免页分裂（直接往后递增，不会insert到中间）

|  |
| --- |
| 有表T(ID,K)，表中R1~R5的(ID,k)值分别为(100,1)、(200,2)、(300,3)、(500,5)和(600,6)，两棵树的示例示意图如下。    B+树为了维护索引有序性，在插入新值的时候需要做必要的维护。以上面这个图为例，如果插入新的行ID值为700，则只需要在R5的记录后面插入一个新记录。如果新插入的ID值为400，就相对麻烦了，需要逻辑上挪动后面的数据，空出位置。  而更糟的情况是，如果R5所在的数据页已经满了，根据B+树的算法，这时候需要申请一个新的数据页，然后挪动部分数据过去。这个过程称为页分裂。在这种情况下，性能自然会受影响。  除了性能外，页分裂操作还影响数据页的利用率。原本放在一个页的数据，现在分到两个页中，整体空间利用率降低大约50%。  当然有分裂就有合并。当相邻两个页由于删除了数据，利用率很低之后，会将数据页做合并。合并的过程，可以认为是分裂过程的逆过程。  **主键长度越小，普通索引的叶子节点就越小，普通索引占用的空间也就越小** |

## 基础知识 √

### 局部性原理

程序和数据的访问都由聚集成群的倾向，在一段时间内，仅使用其中一小部分（称为空间局部性），或者最近访问过的程序代码和数据，很快又被访问的可能性很大（称为空间局部性）

### 磁盘预读

页是存储器的逻辑块，OS 往往将主存和存储区分割为连续的大小相等的块，每个存储块称为一页（也叫page，通常为 4k），主存和磁盘以页为单位交换数据

预读长度一般为页的整数倍 （就像缓存行一样）

### 提升I/O的优化方向

降低 I/O 发生次数

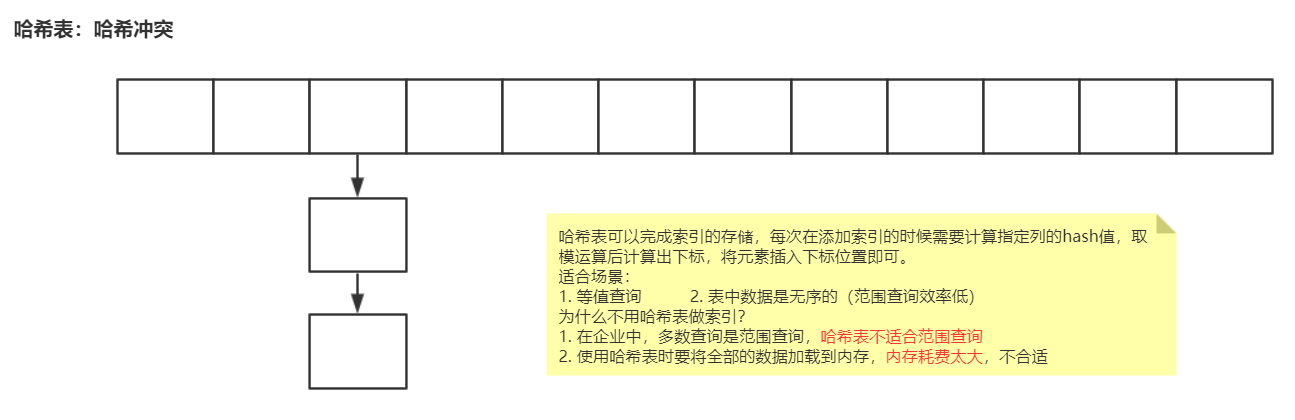
降低每次 I/O 读取的数据量

## mysql 索引底层使用的数据结构是什么？为什么不用哈希表、二叉树、AVL树、红黑树、B树？\*\*\*\*\*五颗星 √

### Mysql用的B+树

### 为什么不用哈希表？

|  |  |
| --- | --- |
| 概念 | 哈希表是key-value存储数据的结构，把值放在数组里，每次添加索引的时候，需要计算出指定列的hash值，取模运算后计算出下标，将元素插入下标位置。  如果存在hash冲突，就拉出来一个链表（想想hashMap的数组+链表）。 |
| 优点 | 新增快 （hash的无序性）； |
| 缺点 | 哈希索引做【范围查询】速度很慢（hash的无序性）  需要全部扫描 |
| 适合场景 | 1. 等值查询 2. 表中的数据是无序的（范围查询效率低） |
| 为什么不用hash表做索引？ | 1. 系统中的查询，多数是范围查询的，哈希表不适合范围查询 2. 使用哈希表时需要将全部的数据加载到内存，内存耗费太大，不适合。 |



### 追问：哈希表是无序的，那为什么不用有序数组？

**有序数组在等值查询和范围查询场景中的性能就都非常优秀。**



这里我们假设身份证号没有重复，这个数组就是按照身份证号递增的顺序保存的。这时候如果你要查ID\_card\_n2对应的名字，用二分法就可以快速得到，这个时间复杂度是O(log(N))。

同时很显然，这个索引结构支持范围查询。你要查身份证号在[ID\_card\_X, ID\_card\_Y]区间的User，可以先用二分法找到ID\_card\_X（如果不存在ID\_card\_X，就找到大于ID\_card\_X的第一个User），然后向右遍历，直到查到第一个大于ID\_card\_Y的身份证号，退出循环。

如果仅仅看查询效率，有序数组就是最好的数据结构了。但是，在需要更新数据的时候就麻烦了，你往中间插入一个记录就必须得挪动后面所有的记录，成本太高。

所以，**有序数组索引只适用于静态存储引擎**，比如你要保存的是2017年某个城市的所有人口信息，这类不会再修改的数据。

### 为什么不用二叉树？AVL树？红黑树？

**咋不屎二叉树：**二叉搜索树的特点是：每个节点的左儿子小于父节点，父节点又小于右儿子。

二叉树时间复杂度是O(log(N))，极端情况下退化成链表，时间复杂度变为O(n)，退化为顺序查找。

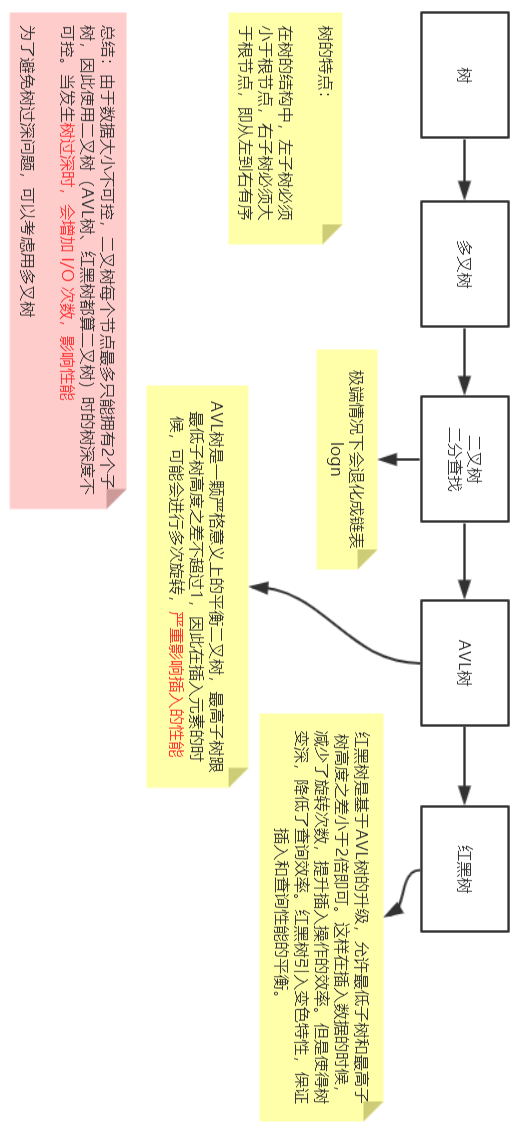
因此，为了获得较好的查找性能（维持O(log(N))的查询复杂度），就要构造一棵平衡二叉序树。为了做这个保证，更新的时间复杂度也是O(log(N))。

**咋不屎AVL树：AVL树**是自平衡二叉树。任何节点的两个子树的高之差不超过1。增加和删除可能会多次旋转来重新平衡这个树，严重影响性能。

**咋不屎红黑树：**红黑树是基于AVL树的升级，允许最低子树和最高子树高度之差小于2倍即可。这样在插入数据的时候，减少了旋转次数，提升插入删除操作效率，但是树变深（磁盘 io 次数增多），降低了查询效率。

**总结：**由于数据大小不可控，二叉树每个节点最多只能拥有2个子树，因此使用二叉树（AVL树，红黑树都是二叉树）时树的深度不可控。当发生树过深时，会增加磁盘I/O次数，影响性能。

|  |
| --- |
| 为了避免树过深问题，可以考虑使用“N叉”树。这里，“N叉”树中的“N”取决于数据块的大小  以InnoDB的一个整数字段索引为例，这个N差不多是1200。这棵树高是4的时候，就可以存1200的3次方个值，这已经17亿了。考虑到树根的数据块总是在内存中的，一个10亿行的表上一个整数字段的索引，查找一个值最多只需要访问3次磁盘。其实，树的第二层也有很大概率在内存中，那么访问磁盘的平均次数就更少了。 |



### 为啥不屎B树？

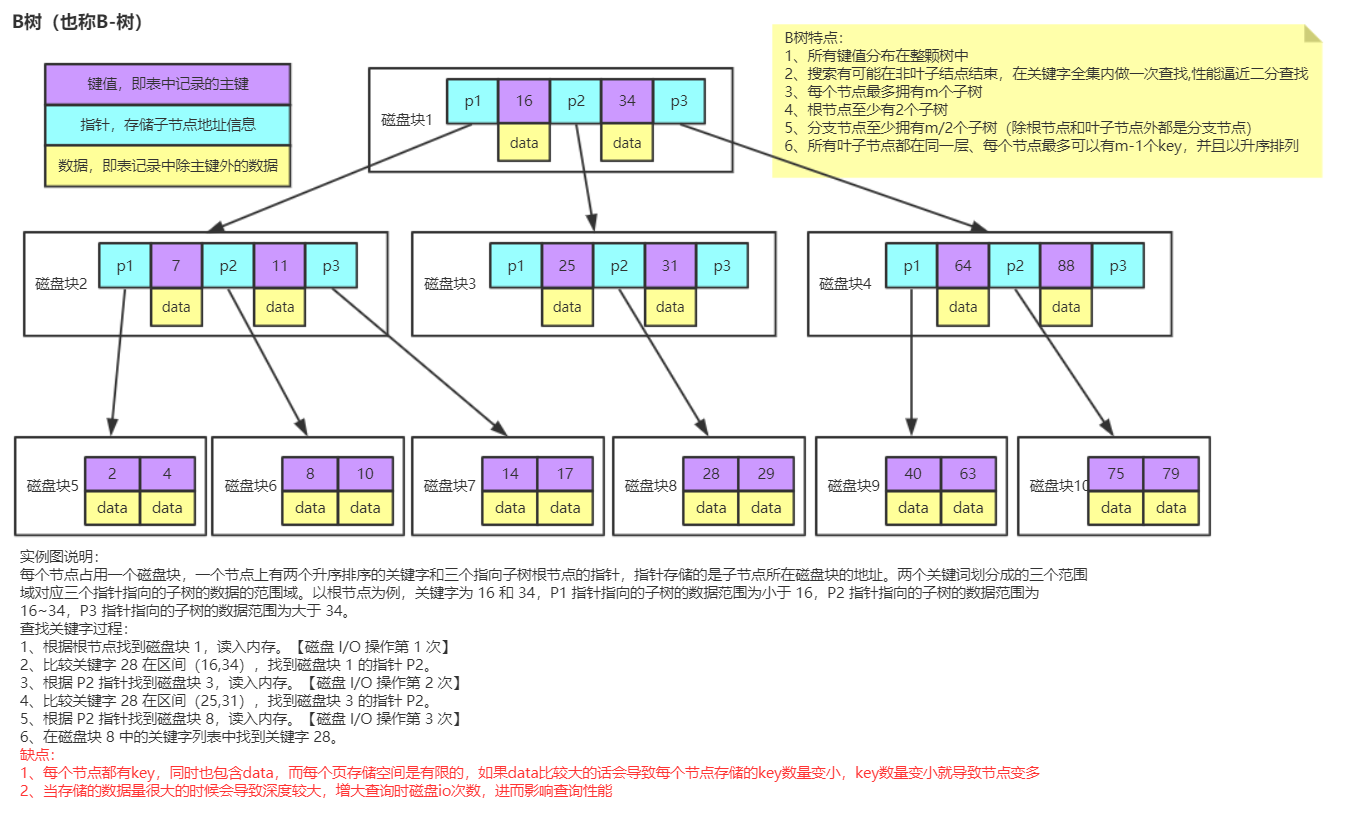
B树是平衡的多叉树，

在B-树中查找给定[关键字](https://baike.baidu.com/item/%E5%85%B3%E9%94%AE%E5%AD%97/7105697" \t "https://baike.baidu.com/item/_blank)的方法是，首先把根结点取来，在根结点所包含的关键字K1,…,Kn查找给定的关键字（可用顺序查找或二分查找法），若找到等于给定值的关键字，则查找成功；否则，一定可以确定要查找的关键字在Ki与Ki+1之间，Pi为指向子树根节点的指针，此时取[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88/2878304" \t "https://baike.baidu.com/item/_blank)Pi所指的结点继续查找，直至找到，或指针Pi为空时查找失败。

为啥不用呢？

缺点：

1. 每个节点都有key，同时也包含data，而每个页存储空间是有限的，如果data比较大的话，会导致每个节点存储的key数量变小，key数量变小就会导致节点变多。
2. 当存储的数据量很大的时候，会导致深度较大，增大查询时磁盘IO次数，影响查询性能。



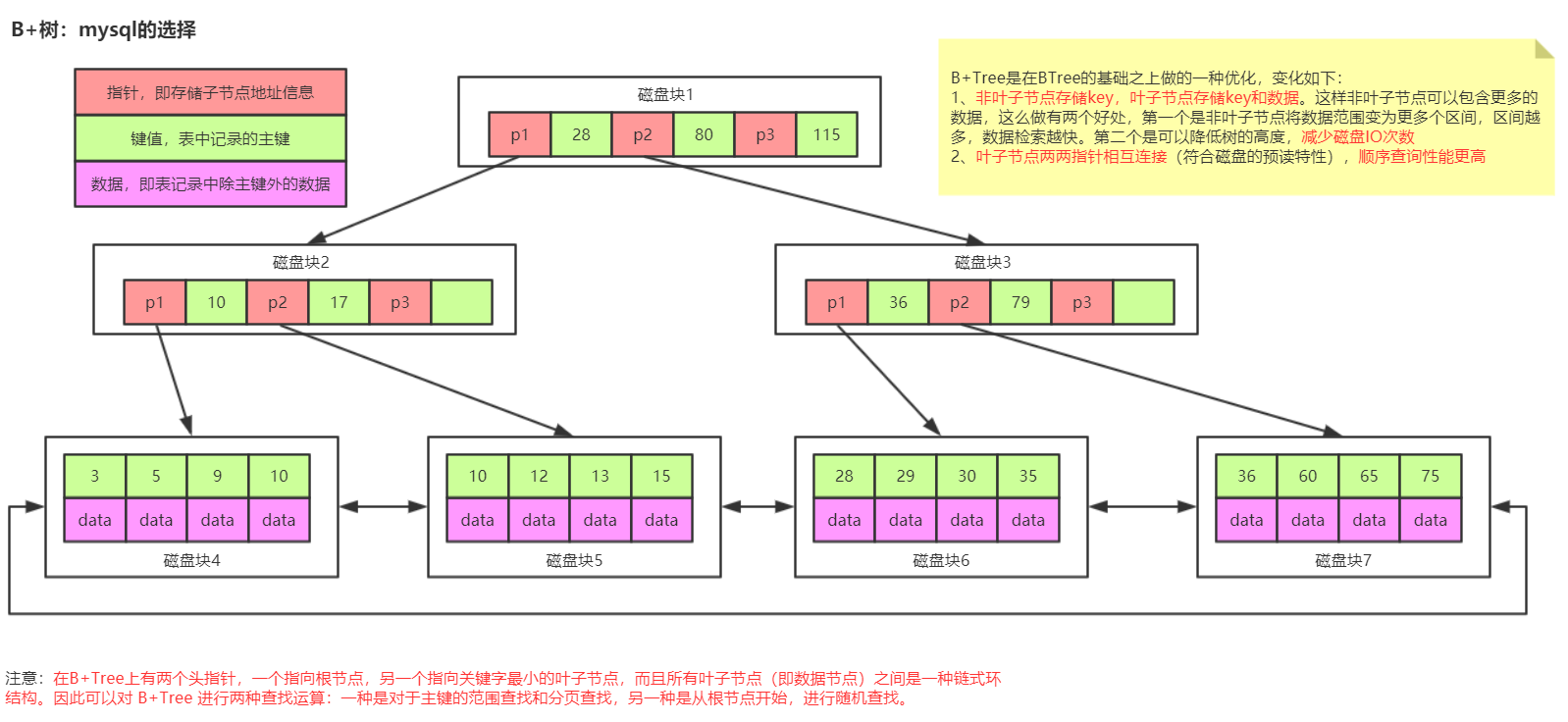
### B+树：

B+树是B树的一种变形形式，B+树上的叶子结点存储关键字以及相应记录的地址，叶子结点以上各层作为索引使用。

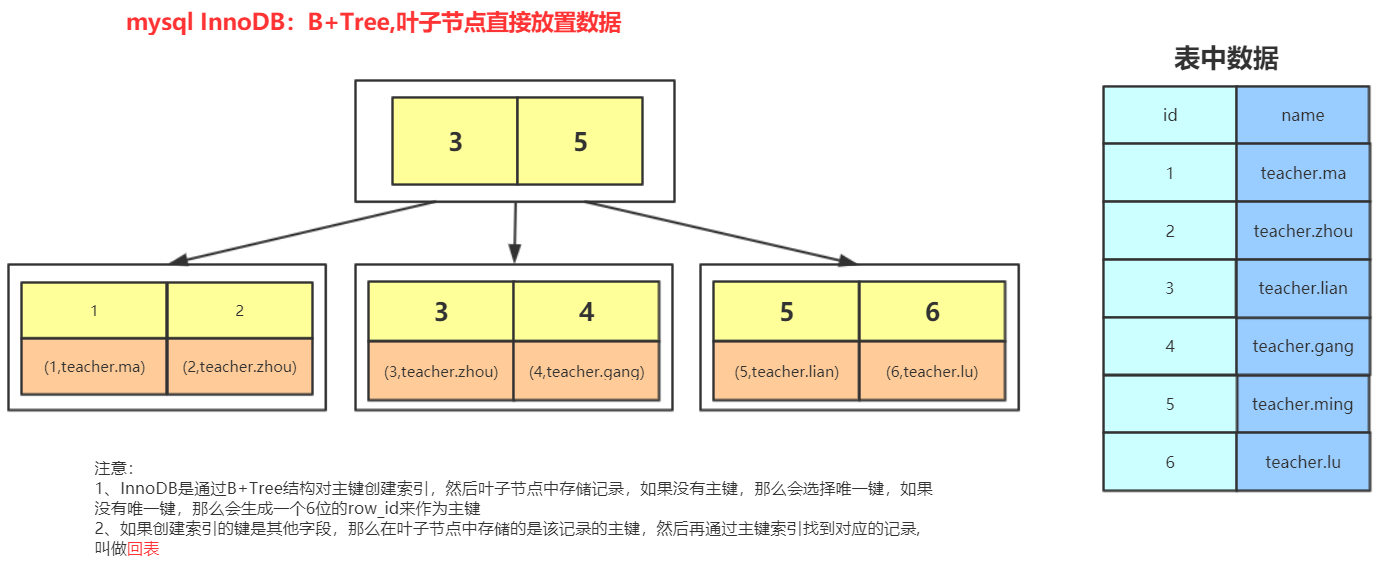
B+Tree是在BTree的基础上做的一种优化，变化如下：

1. 非叶子节点存储key，叶子节点存储key和数据。这样非叶子节点可以包含更多的数据，这么做有两个好处，第一个是非叶子节点将数据范围变为多个更多个区间，区间越多，数据检索越快。第二个是可以降低树的高度，减少磁盘IO次数。
2. 叶子节点两两指针相互连接（符合磁盘的预读特性），顺序查询性能更高。

**B-tree和B+tree的最大区别是，B+tree叶子节点间存在双向指针。这是MySQL官方文档里说的**

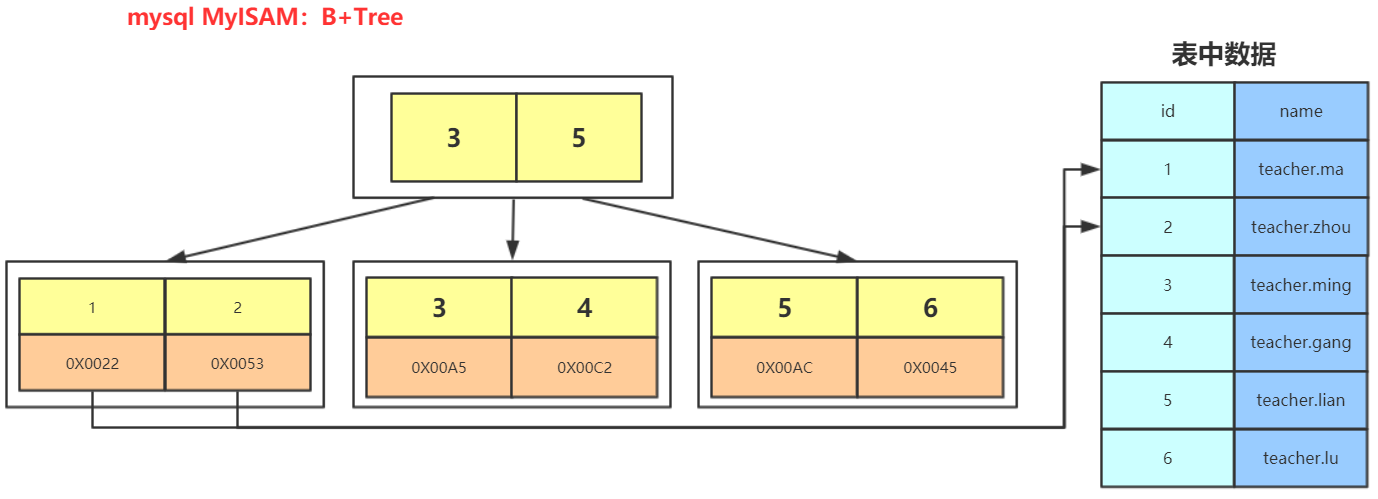


不同的存储引擎，B+树的实现略有不同。在InnoDB中，一个磁盘块默认大小是16k，由于数据和索引在一个文件里，因此叶子节点直接存放了整条数据。在MyISAM中，因为数据和索引不在一个文件里，因此叶子节点存放的是地信息址，地址指向了数据所在位置。



### ****回表：****

默认的当我们创建了一个主键或者唯一键时，mysql 会自动为该列创建索引。如果我们手动的给列 A 创建了索引，A 列索引也会以B+树进行文件存储，此时叶子节点存储的不是整行数据，而是该行数据的主键。当根据 A 列查找数据时，会先根据 A 列的索引找到对应行的主键，然后去主键索引结构里找到整行数据，这个过程叫做"回表"。回表只会出现在innodb中，为了节省空间。而在MyISAM中叶子节点直接存储了行数据的地址，根本不存在为了节省空间而存储索引后进行回表操作。



实际例子（可能是回表）

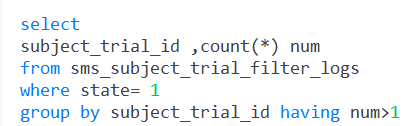
select

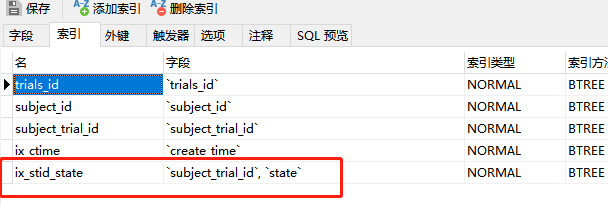
subject\_trial\_id ,count(\*) num

from sms\_subject\_trial\_filter\_logs

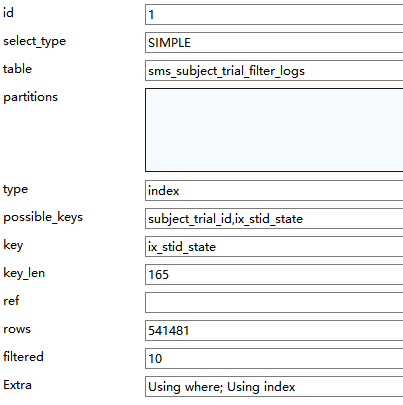
where state= 1

group by subject\_trial\_id having num>1





加了联合索引，sql里只用到这俩字段，应该是减少了回表查询，索引速度快了，没加之前是4秒-优化-》2秒

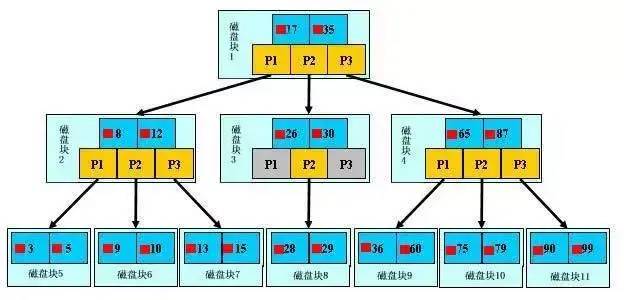


## 索引的数据结构（b树，hash）√

索引的数据结构和具体存储引擎的实现有关，在MySQL中使用较多的索引有Hash索引，B+树索引等，而我们经常使用的InnoDB存储引擎的默认索引实现为：B+树索引。对于哈希索引来说，底层的数据结构就是哈希表，因此在绝大多数需求为单条记录查询的时候，可以选择哈希索引，查询性能最快；其余大部分场景，建议选择BTree索引。

1）B树索引

mysql通过存储引擎取数据，基本上90%的人用的就是InnoDB了，按照实现方式分，InnoDB的索引类型目前只有两种：BTREE（B树）索引和HASH索引。B树索引是Mysql数据库中使用最频繁的索引类型，基本所有存储引擎都支持BTree索引。通常我们说的索引不出意外指的就是（B树）索引（实际是用B+树实现的，因为在查看表索引时，mysql一律打印BTREE，所以简称为B树索引）



查询方式：

主键索引区:PI(关联保存的时数据的地址)按主键查询,

普通索引区:si(关联的id的地址,然后再到达上面的地址)。所以按主键查询,速度最快

**B+tree性质：**

1.）n棵子tree的节点包含n个关键字，不用来保存数据而是保存数据的索引。

2.）所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。

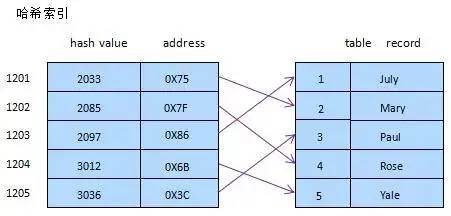
3.）所有的非终端结点可以看成是索引部分，结点中仅含其子树中的最大（或最小）关键字。

4.）B+ 树中，数据对象的插入和删除仅在叶节点上进行。

5.）B+树有2个头指针，一个是树的根节点，一个是最小关键码的叶节点。

**2）哈希索引**

简要说下，类似于数据结构中简单实现的HASH表（散列表）一样，当我们在mysql中用哈希索引时，主要就是通过Hash算法（常见的Hash算法有直接定址法、平方取中法、折叠法、除数取余法、随机数法），将数据库字段数据转换成定长的Hash值，与这条数据的行指针一并存入Hash表的对应位置；如果发生Hash碰撞（两个不同关键字的Hash值相同），则在对应Hash键下以链表形式存储。当然这只是简略模拟图。



## 说下 MySQL 的索引原理 √

索引用来快速地寻找那些具有特定值的记录。如果没有索引，一般来说执行查询时遍历整张表。

索引的原理很简单，就是把无序的数据变成有序的查询

把创建了索引的列的内容进行排序

对排序结果生成倒排表

在倒排表内容上拼上数据地址链

在查询的时候，先拿到倒排表内容，再取出数据地址链，从而拿到具体数据

## ****索引算法有哪些？****

索引算法有 BTree算法和Hash算法

**BTree算法**

BTree是最常用的mysql数据库索引算法，也是mysql默认的算法。因为它不仅可以被用在=,>,>=,<,<=和between这些比较操作符上，而且还可以用于like操作符，只要它的查询条件是一个不以通配符开头的常量， 例如：

-- 只要它的查询条件是一个不以通配符开头的常量

select \* from user where name like 'jack%';

-- 如果一通配符开头，或者没有使用常量，则不会使用索引，例如：

select \* from user where name like '%jack';

**Hash算法**

Hash Hash索引只能用于对等比较，例如=,<=>（相当于=）操作符。由于是一次定位数据，不像BTree索引需要从根节点到枝节点，最后才能访问到页节点这样多次IO访问，所以检索效率远高于BTree索引。

## 索引设计的原则？ √

* 适合索引的列是出现在where子句中的列，或者连接子句中指定的列
* 基数较小的类，索引效果较差，没有必要在此列建立索引
* 使用短索引，如果对长字符串列进行索引，应该指定一个前缀长度，这样能够节省大量索引空间
* 不要过度索引。索引需要额外的磁盘空间，并降低写操作的性能。在修改表内容的时候，索引会进行更新甚至重构，索引列越多，这个时间就会越长。所以只保持需要的索引有利于查询即可。

### 创建索引的原则（重中之重）

**a: 肯定在where条经常使用**

**b: 该字段的内容不是唯一的几个值(sex)**

**c: 字段内容不是频繁变化.**

索引虽好，但也不是无限制的使用，最好符合一下几个原则

1） 最左前缀匹配原则，组合索引非常重要的原则，mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。

2）较频繁作为查询条件的字段才去创建索引

3）更新频繁字段不适合创建索引

4）若是不能有效区分数据的列不适合做索引列(如性别，男女未知，最多也就三种，区分度实在太低)

5）尽量的扩展索引，不要新建索引。比如表中已经有a的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可。

6）定义有外键的数据列一定要建立索引。

7）对于那些查询中很少涉及的列，重复值比较多的列不要建立索引。

8）对于定义为text、image和bit的数据类型的列不要建立索引。

## 第三种方式：使用CREATE INDEX命令创建****创建索引的三种方式，删除索引****

第一种方式：在执行CREATE TABLE时创建索引

|  |
| --- |
| CREATE TABLE user\_index2 (  id INT auto\_increment PRIMARY KEY,  first\_name VARCHAR (16),  last\_name VARCHAR (16),  id\_card VARCHAR (18),  information text,  KEY name (first\_name, last\_name),  FULLTEXT KEY (information),  UNIQUE KEY (id\_card) ); |

第二种方式：使用ALTER TABLE命令去增加索引

|  |
| --- |
| ALTER TABLE table\_name ADD INDEX index\_name (column\_list); |

ALTER TABLE用来创建普通索引、UNIQUE索引或PRIMARY KEY索引。

其中table\_name是要增加索引的表名，column\_list指出对哪些列进行索引，多列时各列之间用逗号分隔。

索引名index\_name可自己命名，缺省时，MySQL将根据第一个索引列赋一个名称。另外，ALTER TABLE允许在单个语句中更改多个表，因此可以在同时创建多个索引。

第三种方式：使用CREATE INDEX命令创建

CREATE INDEX index\_name ON table\_name (column\_list);

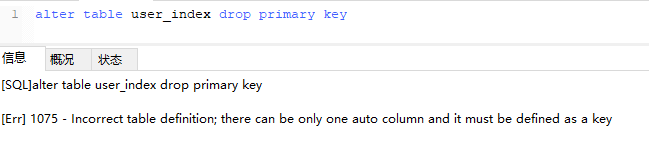
CREATE INDEX可对表增加普通索引或UNIQUE索引。（但是，不能创建PRIMARY KEY索引）

删除索引

根据索引名删除普通索引、唯一索引、全文索引：alter table 表名 drop KEY 索引名

|  |
| --- |
| alter table user\_index drop KEY name; alter table user\_index drop KEY id\_card; alter table user\_index drop KEY information; |

删除主键索引：alter table 表名 drop primary key（因为主键只有一个）。这里值得注意的是，如果主键自增长，那么不能直接执行此操作（自增长依赖于主键索引）：



需要取消自增长再行删除：

alter table user\_index  
-- 重新定义字段  
MODIFY id int,  
drop PRIMARY KEY

但通常不会删除主键，因为设计主键一定与业务逻辑无关。

## ****创建索引时需要注意什么？****

* 非空字段：应该指定列为NOT NULL，除非你想存储NULL。在mysql中，含有空值的列很难进行查询优化，因为它们使得索引、索引的统计信息以及比较运算更加复杂。你应该用0、一个特殊的值或者一个空串代替空值；
* 取值离散大的字段：（变量各个取值之间的差异程度）的列放到联合索引的前面，可以通过count()函数查看字段的差异值，返回值越大说明字段的唯一值越多字段的离散程度高；
* 索引字段越小越好：数据库的数据存储以页为单位一页存储的数据越多一次IO操作获取的数据越大效率越高。

## ****使用索引查询一定能提高查询的性能吗？为什么****

通常，通过索引查询数据比全表扫描要快。但是我们也必须注意到它的代价。

* 索引需要空间来存储，也需要定期维护， 每当有记录在表中增减或索引列被修改时，索引本身也会被修改。这意味着每条记录的INSERT，DELETE，UPDATE将为此多付出4，5 次的磁盘I/O。因为索引需要额外的存储空间和处理，那些不必要的索引反而会使查询反应时间变慢。使用索引查询不一定能提高查询性能，索引范围查询(INDEX RANGE SCAN)适用于两种情况:
* 基于一个范围的检索，一般查询返回结果集小于表中记录数的30%
* 基于非唯一性索引的检索

## ****百万级别或以上的数据如何删除****

关于索引：由于索引需要额外的维护成本，因为索引文件是单独存在的文件,所以当我们对数据的增加,修改,删除,都会产生额外的对索引文件的操作,这些操作需要消耗额外的IO,会降低增/改/删的执行效率。所以，在我们删除数据库百万级别数据的时候，查询MySQL官方手册得知删除数据的速度和创建的索引数量是成正比的。

1. 所以我们想要删除百万数据的时候可以先删除索引（此时大概耗时三分多钟）
2. 然后删除其中无用数据（此过程需要不到两分钟）
3. 删除完成后重新创建索引(此时数据较少了)创建索引也非常快，约十分钟左右。
4. 与之前的直接删除绝对是要快速很多，更别说万一删除中断,一切删除会回滚。那更是坑了。

## ****. 前缀索引****

语法：index(field(10))，使用字段值的前10个字符建立索引，默认是使用字段的全部内容建立索引。

前提：前缀的标识度高。比如密码就适合建立前缀索引，因为密码几乎各不相同。

实操的难度：在于前缀截取的长度。

我们可以利用select count(\*)/count(distinct left(password,prefixLen));，通过从调整prefixLen的值（从1自增）查看不同前缀长度的一个平均匹配度，接近1时就可以了（表示一个密码的前prefixLen个字符几乎能确定唯一一条记录）

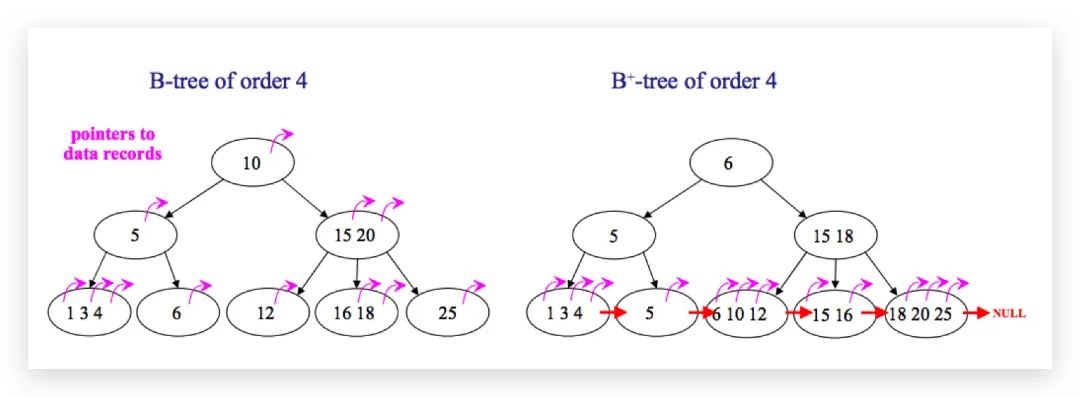
## 什么是最左前缀原则？√

* 顾名思义，就是最左优先，**在创建多列索引时，要根据业务需求，where子句中使用最频繁的一列放在最左边。**
* 最左前缀匹配原则，非常重要的原则，mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。
* =和in可以乱序，比如a = 1 and b = 2 and c = 3 建立(a,b,c)索引可以任意顺序，mysql的查询优化器会帮你优化成索引可以识别的形式

|  |
| --- |
| **最左匹配原理**  **基于MySQL的InnoDB引擎说明**    对于该表，如果按照 name 字段来建立索引的话，采用B+树的结构，大概的索引结构如下：    如果要进行模糊查找，查找name 以“张"开头的所有人的ID，即 sql 语句为：  select ID from table where name like '张%'  由于在B+树结构的索引中，索引项是按照索引定义里面出现的字段顺序排序的，索引在查找的时候，可以快速定位到 ID 为 100的张一，然后直接向右遍历所有张开头的人，直到条件不满足为止。也就是说，当找到第一个满足条件的人之后，直接向右遍历就可以了，由于索引是有序的，所有满足条件的人都会聚集在一起。而这种定位到最左边，然后向右遍历寻找的方式，就是我们所说的最左前缀原则。  **示例：一个(a,b,c)的组合索引。**   1. 通过a,c条件查询能不能使用或命中这个索引？-----能 2. 通过b,c条件查询能不能使用或命中这个索引？-----不能 3. 原因：索引文件具有B-Tree 的最左前缀匹配特性，如果左边的值未确定，那么无法使用此索引。 |

## ****B树和B+树的区别****

* 在B树中，你可以将键和值存放在内部节点和叶子节点；但在B+树中，内部节点都是键，没有值，叶子节点同时存放键和值。
* B+树的叶子节点有一条链相连，而B树的叶子节点各自独立。



## ****使用B树的好处****

B树可以在内部节点同时存储键和值，因此，把频繁访问的数据放在靠近根节点的地方将会大大提高热点数据的查询效率。这种特性使得B树在特定数据重复多次查询的场景中更加高效。

## ****使用B+树的好处****

由于B+树的内部节点只存放键，不存放值，因此，一次读取，可以在内存页中获取更多的键，有利于更快地缩小查找范围。B+树的叶节点由一条链相连，因此，当需要进行一次全数据遍历的时候，B+树只需要使用O(logN)时间找到最小的一个节点，然后通过链进行O(N)的顺序遍历即可。而B树则需要对树的每一层进行遍历，这会需要更多的内存置换次数，因此也就需要花费更多的时间

## ****Hash索引和B+树所有有什么区别或者说优劣呢?****

首先要知道Hash索引和B+树索引的底层实现原理：

hash索引底层就是hash表，进行查找时，调用一次hash函数就可以获取到相应的键值，之后进行回表查询获得实际数据。B+树底层实现是多路平衡查找树。对于每一次的查询都是从根节点出发，查找到叶子节点方可以获得所查键值，然后根据查询判断是否需要回表查询数据。

那么可以看出他们有以下的不同：

* hash索引进行等值查询更快(一般情况下)，但是却无法进行范围查询。

因为在hash索引中经过hash函数建立索引之后，索引的顺序与原顺序无法保持一致，不能支持范围查询。而B+树的的所有节点皆遵循(左节点小于父节点，右节点大于父节点，多叉树也类似)，天然支持范围。

* hash索引不支持使用索引进行排序，原理同上。
* hash索引不支持模糊查询以及多列索引的最左前缀匹配。原理也是因为hash函数的不可预测。AAAA和AAAAB的索引没有相关性。
* hash索引任何时候都避免不了回表查询数据，而B+树在符合某些条件(聚簇索引，覆盖索引等)的时候可以只通过索引完成查询。
* hash索引虽然在等值查询上较快，但是不稳定。性能不可预测，当某个键值存在大量重复的时候，发生hash碰撞，此时效率可能极差。而B+树的查询效率比较稳定，对于所有的查询都是从根节点到叶子节点，且树的高度较低。

因此，在大多数情况下，直接选择B+树索引可以获得稳定且较好的查询速度。而不需要使用hash索引。

## ****数据库为什么使用B+树而不是B树****

* B树只适合随机检索，而B+树同时支持随机检索和顺序检索；
* B+树空间利用率更高，可减少I/O次数，磁盘读写代价更低。一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。这样的话，索引查找过程中就要产生磁盘I/O消耗。B+树的内部结点并没有指向关键字具体信息的指针，只是作为索引使用，其内部结点比B树小，盘块能容纳的结点中关键字数量更多，一次性读入内存中可以查找的关键字也就越多，相对的，IO读写次数也就降低了。而IO读写次数是影响索引检索效率的最大因素；
* B+树的查询效率更加稳定。B树搜索有可能会在非叶子结点结束，越靠近根节点的记录查找时间越短，只要找到关键字即可确定记录的存在，其性能等价于在关键字全集内做一次二分查找。而在B+树中，顺序检索比较明显，随机检索时，任何关键字的查找都必须走一条从根节点到叶节点的路，所有关键字的查找路径长度相同，导致每一个关键字的查询效率相当。
* B-树在提高了磁盘IO性能的同时并没有解决元素遍历的效率低下的问题。B+树的叶子节点使用指针顺序连接在一起，只要遍历叶子节点就可以实现整棵树的遍历。而且在数据库中基于范围的查询是非常频繁的，而B树不支持这样的操作。
* 增删文件（节点）时，效率更高。因为B+树的叶子节点包含所有关键字，并以有序的链表结构存储，这样可很好提高增删效率。

## ****B+树在满足聚簇索引和覆盖索引的时候不需要回表查询数据****

在B+树的索引中，叶子节点可能存储了当前的key值，也可能存储了当前的key值以及整行的数据，这就是聚簇索引和非聚簇索引。在InnoDB中，只有主键索引是聚簇索引，如果没有主键，则挑选一个唯一键建立聚簇索引。如果没有唯一键，则隐式的生成一个键来建立聚簇索引。

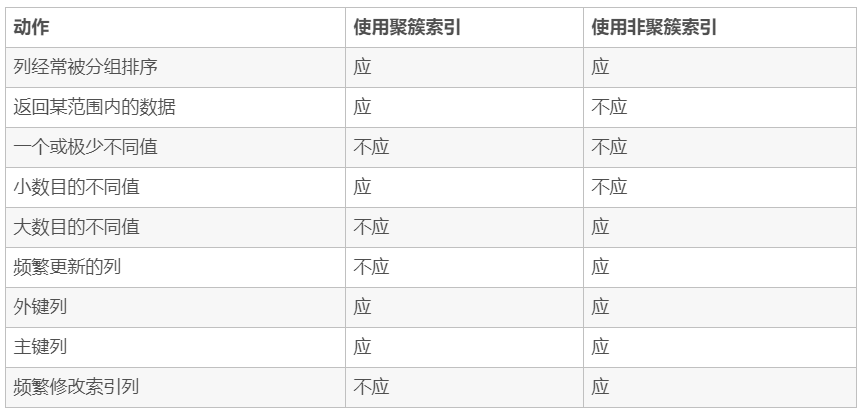
当查询使用聚簇索引时，在对应的叶子节点，可以获取到整行数据，因此不用再次进行回表查询。

## ****什么是聚簇索引？何时使用聚簇索引与非聚簇索引****

* 聚簇索引：将数据存储与索引放到了一块，找到索引也就找到了数据
* 非聚簇索引：将数据存储于索引分开结构，索引结构的叶子节点指向了数据的对应行，myisam通过key\_buffer把索引先缓存到内存中，当需要访问数据时（通过索引访问数据），在内存中直接搜索索引，然后通过索引找到磁盘相应数据，这也就是为什么索引不在key buffer命中时，速度慢的原因

澄清一个概念：innodb中，在聚簇索引之上创建的索引称之为辅助索引，辅助索引访问数据总是需要二次查找，非聚簇索引都是辅助索引，像复合索引、前缀索引、唯一索引，辅助索引叶子节点存储的不再是行的物理位置，而是主键值

何时使用聚簇索引与非聚簇索引



## 数据库表可以有几个聚簇索引？XDD

1个

## ****非聚簇索引一定会回表查询吗？****

不一定，这涉及到查询语句所要求的字段是否全部命中了索引，如果全部命中了索引，那么就不必再进行回表查询。

举个简单的例子，假设我们在员工表的年龄上建立了索引，那么当进行select age from employee where age < 20的查询时，在索引的叶子节点上，已经包含了age信息，不会再次进行回表查询。

## ****联合索引是什么？为什么需要注意联合索引中的顺序？****

MySQL可以使用多个字段同时建立一个索引，叫做联合索引。在联合索引中，如果想要命中索引，需要按照建立索引时的字段顺序挨个使用，否则无法命中索引。

具体原因为:

MySQL使用索引时需要索引有序，假设现在建立了"name，age，school"的联合索引，那么索引的排序为: 先按照name排序，如果name相同，则按照age排序，如果age的值也相等，则按照school进行排序。

当进行查询时，此时索引仅仅按照name严格有序，因此必须首先使用name字段进行等值查询，之后对于匹配到的列而言，其按照age字段严格有序，此时可以使用age字段用做索引查找，以此类推。因此在建立联合索引的时候应该注意索引列的顺序，一般情况下，将查询需求频繁或者字段选择性高的列放在前面。此外可以根据特例的查询或者表结构进行单独的调整。

## 什么是自适应哈希索引（AHI）

## MySQL 索引使用的注意事项 √

### 满足以下条件的字段，才应该创建索引.

* 肯定在where条经常使用
* 该字段的内容不是唯一的几个值(sex)
* 字段内容不是频繁变化.

### 使用索引的注意事项（索引失效情况）

1. 对于创建的多列索引，注意最左匹配原则
2. 对于使用like的查询，查询如果是 ‘%aaa’ 不会使用到索引，‘aaa%’ 会使用到索引。
3. 如果条件中有or，即使其中有条件带索引也不会使用。换言之，就是要求使用的所有字段，都必须建立索引, 我们建议大家尽量避免使用or 关键字
4. 如果列类型是字符串，那一定要在条件中将数据使用引号引用起来。否则不使用索引。(添加时,字符串必须’’), 也就是，如果列是字符串类型，就一定要用 ‘’ 把他包括起来.
5. 如果mysql估计使用全表扫描要比使用索引快，则不使用索引。

## 索引失效的情况，什么时候可能会出现，如何解决（上一问）√

## 什么是全文索引，全文索引中的倒排索引是什么原理（看ES部分）√

## 如果建了⼀个单列索引，查询的时候查出2列，会⽤到这个单列索引吗？

会用到，索引是在where条件起作用的。

## 如果建了⼀个包含多个列的索引，查询的时候只⽤了第⼀列，能不能⽤上这个索引？查三列呢？接上题，如果where条件后⾯带有⼀个 i + 5 < 100 会使⽤到这个索引吗？

## 数据库什么情况下索引会失效; √

1. **like查询**

对于使用like的查询，查询如果是 ‘%aaa’ 不会使用到索引‘aaa%’ 会使用到索引。

比如: explain select \* from dept where dname like '%aaa'\G不能使用索引，即，在like查询时，关键的 ‘关键字’ , 最前面，不能使用 % 或者 \_这样的字符。

**2 。Or**

如果条件中有or，要求使用的所有字段，都必须建立索引, 否则索引失效

如果or

select \* from crc\_user where login\_name ='aaa' or cn\_name = '张强'，login\_name有索引但是cn\_name没有索引，所以不生效

改写：改成 union all连接多个or，有索引的字段就会用到索引

|  |
| --- |
| 注：where只有一个字段的or且该字段有索引，还是会用到索引，两个字段就不会用到，很神奇，所以还是explain看看为好  例子：a、b字段都有索引  explain SELECT \* FROM `tb\_index` where a ='aaa' or a='aaa3' 就用到了索引    但是，where a ='aaa' or b='bbb3'就不走索引。 |

3.如果列类型是字符串，那一定要在条件中将数据使用引号引用起来。否则不使用索引。(添加时,字符串必须’’), 也就是，如果列是字符串类型，就一定要用 ‘’ 把他包括起来.

explain select \* from crc\_user where login\_name = 222 login\_name有索引但是失效

1. 如果mysql估计使用全表扫描要比使用索引快，则不使用索引（查询的数量是大表的大部分）。
2. 在 where 子句中对字段进行表达式操作，会使引擎放弃使用索引而进行全表扫描
3. 在where子句中对字段进行函数操作，会使引擎放弃使用索引而进行全表扫描

查询条件使用函数在索引列上，或者对索引列进行运算，运算包括(+，-，\*，/，! 等) 错误的例子：select \* from test where id-1=9; 正确的例子：select \* from test where id=10;

13,like "%\_" 百分号在前.

15,单独引用复合索引里非第一位置的索引列.

16,字符型字段为数字时在where条件里不添加引号.

17,对索引列进行运算.需要建立函数索引.

18,not in ,not exist.

19,当变量采用的是times变量，而表的字段采用的是date变量时.或相反情况。

20,B-tree索引 is null不会走,is not null会走

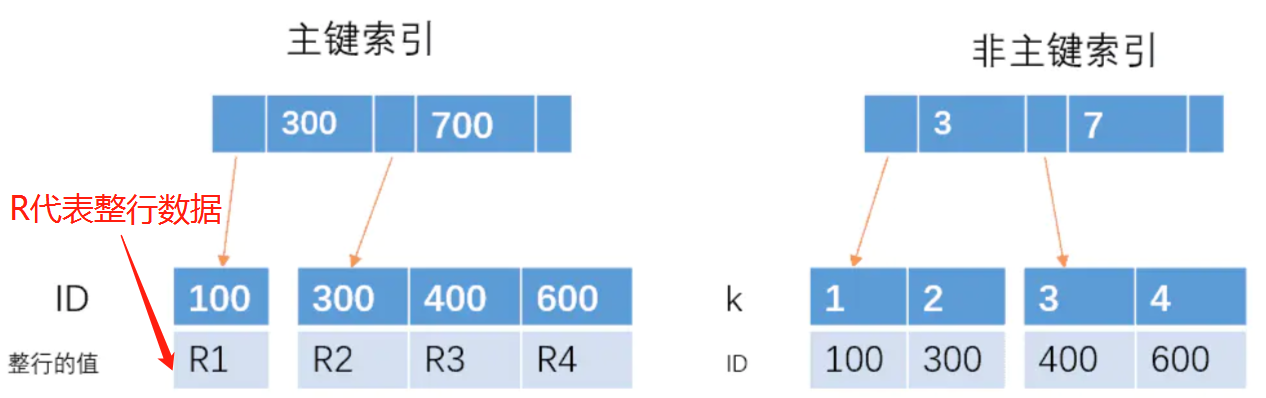
21,联合索引 is not null 只要在建立的索引列（不分先后）都会走, in null时 必须要和建立索引第一列一起使用,当建立索引第一位置条件是is null 时,其他建立索引的列可以是is null（但必须在所有列 都满足is null的时候）,或者=一个值； 当建立索引的第一位置是=一个值时,其他索引列可以是任何情况（包括is null =一个值）,以上两种情况索引都会走。其他情况不会走。

## 主键索引和非主键索引（二级索引）有什么区别 √

例如下表(其实就是上面的表中增加了一个k字段)，且ID是主键



主键索引和非主键索引的示意图如下：



### 主键索引和非主键索引的区别是：

* 非主键索引的叶子节点存放的是****主键的值****，
* 主键索引的叶子节点存放的是****整行数据****。
* 非主键索引也被称为****二级索引****，而主键索引也被称为****聚簇索引****。

### 根据这两种结构进行查询，看看区别：

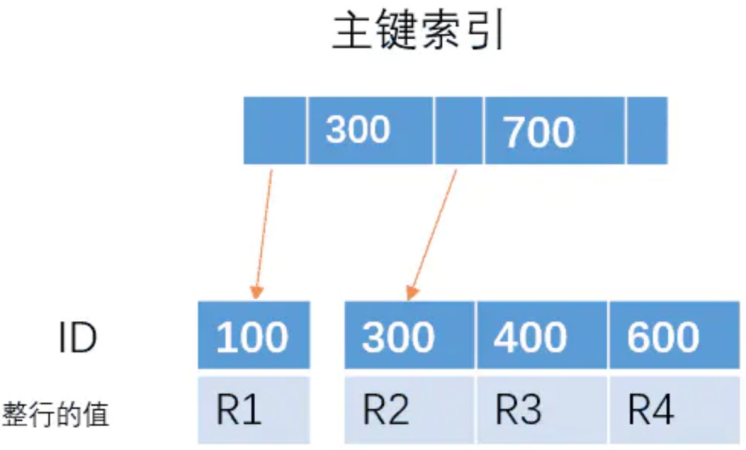
* 如果查询语句是 select \* from table where ID = 100，即主键查询的方式，则只需要搜索 ID 这棵 B+树。
* 如果查询语句是 select \* from table where k = 1，即非主键的查询方式，则先搜索k索引树，得到ID=100，再到ID索引树搜索一次，这个过程也被称为**回表**。

### 聚集索引和非聚集索引的区别

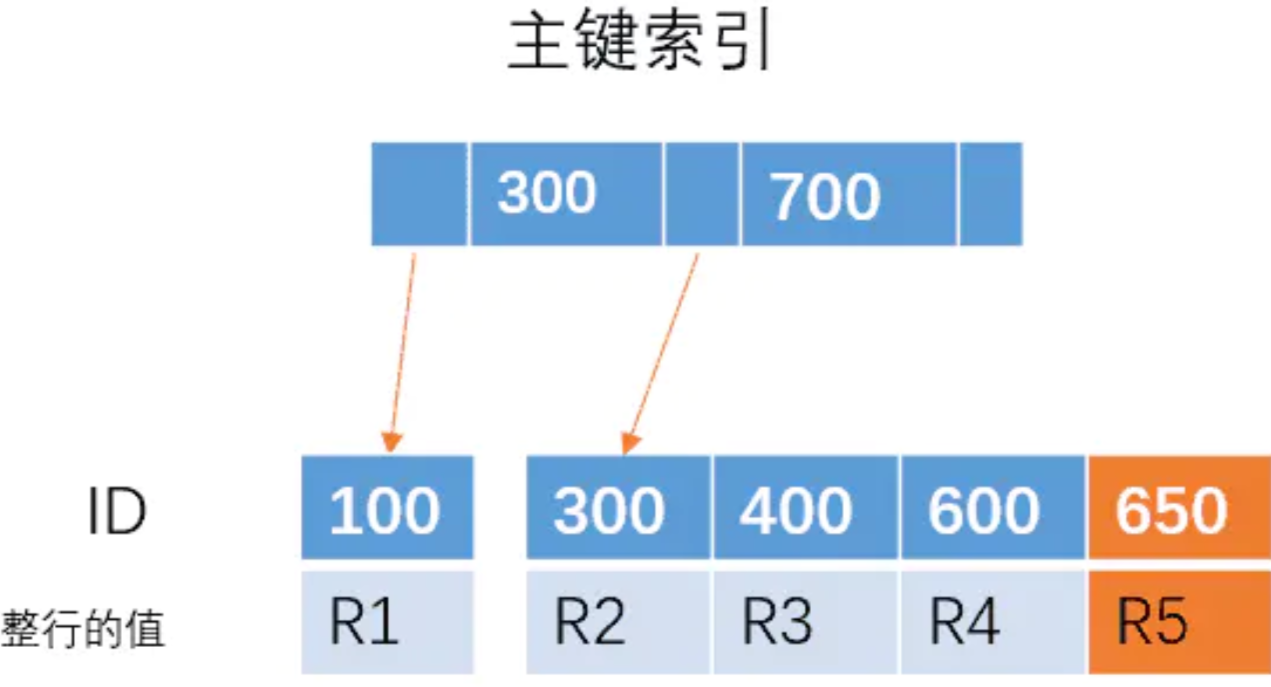
* 聚集索引表示表中存储的数据按照索引的顺序存储，检索效率比非聚集索引高，但对数据更新影响较大。（比如主键索引）
* 非聚集索引表示数据存储在一个地方，索引存储在另一个地方，索引带有指针指向数据的存储位置。非聚集索引检索效率比聚集索引低，但对数据更新影响较小。

## 为什么建议使用主键自增的索引？√

对于这棵主键索引的树：



如果插入 ID = 650 的一行数据，那么直接在最右边插入就可以了



但是如果插入的是 ID = 350 的一行数据，由于 B+ 树是有序的，那么需要将下面的叶子节点进行移动，腾出位置来插入 ID = 350 的数据，这样就会比较消耗时间。如果刚好 R4 所在的数据页已经满了，需要进行**页分裂**操作，这样会更加糟糕。

但是，如果主键是自增的，每次插入的 ID 都会比前面的大，那么每次只需要在后面插入就行， 不需要移动位置、分裂等操作，这样可以提高性能。也就是为什么建议使用主键自增的索引。

## 如何判断一个查询 sql 语句是否使用了索引 √

Explain分析，key字段



# =========优化===========

## Mysql数据库的优化技术

对mysql优化时一个综合性的技术，主要包括

a: 表的设计合理化(符合3NF)

b: 添加适当索引(index) [四种: 普通索引、主键索引、唯一索引unique、全文索引]

c: 分表技术(水平分割、垂直分割)

d: 读写[写: update/delete/add]分离

e: 存储过程 [模块化编程，可以提高速度]

f: 对mysql配置优化 [配置最大并发数my.ini, 调整缓存大小 ]

g: mysql服务器硬件升级

h: 定时的去清除不需要的数据,定时进行碎片整理(MyISAM)

说说 SQL 优化之道

limit 20000 加载很慢怎么解决

常见的数据库优化方案，在你的项目中数据库如何进行优化的

如果查询很慢，你会想到的第⼀个⽅式是什么？（数据库索引）

SQL优化思路，联合索引与底层树结构的映像关系，索引结构（B+、B-），为什么用这样的结构；

你目前为止遇到的最大数据量是多少？知道100万时候怎么设计吗？1000万呢？过几十亿呢？

MySQL有多少个参数可调，除了最大连接数。全部列出来，一个个分析。

聊下优化过的索引，怎么优化;

## 如何去定位慢查询

# =========存储引擎===========

存储引擎的 InnoDB与MyISAM区别，优缺点，使用场景

InnoDB的插入缓冲和两次写的概率和意义；

讲下 InnoDB 和 MyISAM 的区别？使用场景是

## ****MySQL存储引擎MyISAM与InnoDB区别****

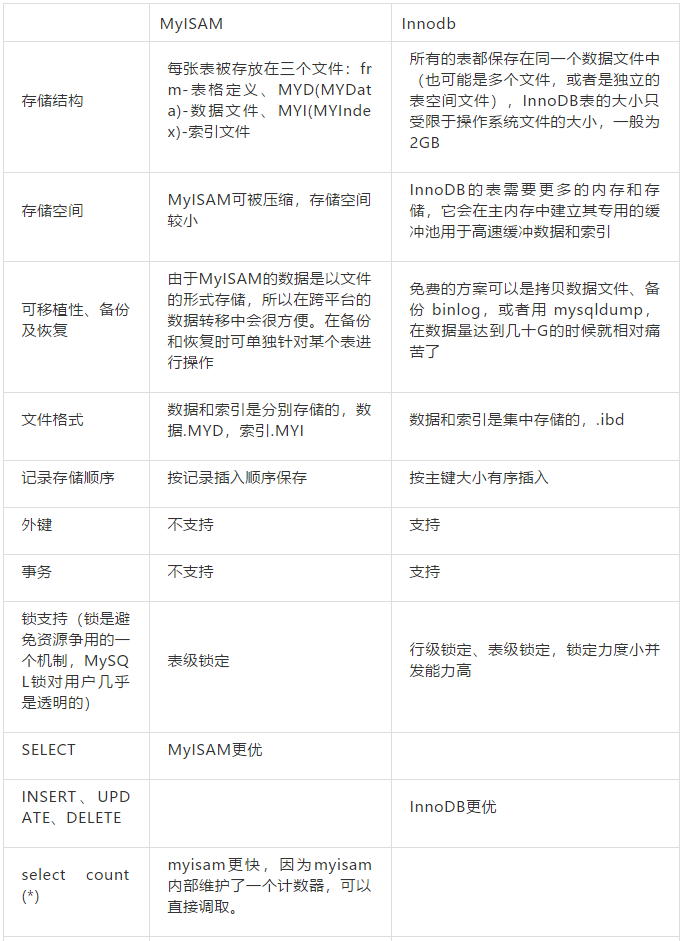
存储引擎Storage engine：MySQL中的数据、索引以及其他对象是如何存储的，是一套文件系统的实现。

常用的存储引擎有以下：

Innodb引擎：Innodb引擎提供了对数据库ACID事务的支持。并且还提供了行级锁和外键的约束。它的设计的目标就是处理大数据容量的数据库系统。

MyIASM引擎(原本Mysql的默认引擎)：不提供事务的支持，也不支持行级锁和外键。

MEMORY引擎：所有的数据都在内存中，数据的处理速度快，但是安全性不高。

=



## ****MyISAM索引与InnoDB索引的区别？****

InnoDB索引是聚簇索引，MyISAM索引是非聚簇索引。

InnoDB的主键索引的叶子节点存储着行数据，因此主键索引非常高效。

MyISAM索引的叶子节点存储的是行数据地址，需要再寻址一次才能得到数据。

InnoDB非主键索引的叶子节点存储的是主键和其他带索引的列数据，因此查询时做到覆盖索引会非常高效。

## InnoDB引擎的4大特性

插入缓冲（insert buffer)

二次写(double write)

自适应哈希索引(ahi)

预读(read ahead)

## ****存储引擎选择****

如果没有特别的需求，使用默认的Innodb即可。

MyISAM：以读写插入为主的应用程序，比如博客系统、新闻门户网站。

Innodb：更新（删除）操作频率也高，或者要保证数据的完整性；并发量高，支持事务和外键。比如OA自动化办公系统。

# =========锁==========

## Mysql锁机制 √

Mysql不同的存储引擎支持不同的锁机制

|  |  |
| --- | --- |
| 存储引擎 | 锁粒度 |
| **MyISAM和MEMORY** | 表级锁 |
| **InnoDB** | 既支持行级锁（默认），也支持表级锁 |

**表级锁：**

开销小，加锁快；

不会出现死锁；

锁定粒度大，发生锁冲突的概率最高，并发度最低。 ​

**行级锁：**

开销大，加锁慢；

会出现死锁；

锁定粒度最小，发生锁冲突的概率最低，并发度也最高。

### MyISAM表锁

MySQL的表级锁有两种模式：**共享锁（读锁），排它锁（写锁）**

MyISAM在执行查询语句之前，会自动给涉及的所有表加读锁，在执行更新操作前，会自动给涉及的表加写锁，这个过程并不需要用户干预，因此用户一般不需要使用命令来显式加锁。读写串行。

### InnoDB行锁

#### 事务及其ACID属性

事务是由一组SQL语句组成的逻辑处理单元，事务具有4属性，通常称为事务的ACID属性。

* 原子性（Actomicity）：事务是一个原子操作单元，其对数据的修改，要么全都执行，要么全都不执行。
* 一致性（Consistent）：在事务开始和完成时，数据都必须保持一致状态。
* 隔离性（Isolation）：数据库系统提供一定的隔离机制，保证事务在不受外部并发操作影响的“独立”环境执行。 持久性（Durable）：事务完成之后，它对于数据的修改是永久性的，即使出现系统故障也能够保持。

#### 并发事务带来的问题

相对于串行处理来说，并发事务处理能大大增加数据库资源的利用率，提高数据库系统的事务吞吐量，从而可以支持更多用户的并发操作，但与此同时，会带来一下问题：

**脏读：**

一个事务正在对一条记录做修改，在这个事务并提交前，这条记录的数据就处于不一致状态；这时，另一个事务也来读取同一条记录，如果不加控制，第二个事务读取了这些“脏”的数据，并据此做进一步的处理，就会产生未提交的数据依赖关系。这种现象被形象地叫做“脏读”

**不可重复读：**

一个事务在读取某些数据已经发生了改变、或某些记录已经被删除了！这种现象叫做“不可重复读”。

**幻读：**

一个事务按相同的查询条件重新读取以前检索过的数据，却发现其他事务插入了满足其查询条件的新数据，这种现象就称为“幻读”

上述出现的问题都是数据库读一致性的问题，可以通过事务的隔离机制来进行保证。

数据库的事务隔离越严格，并发副作用就越小，但付出的代价也就越大，因为事务隔离本质上就是使事务在一定程度上串行化，需要根据具体的业务需求来决定使用哪种隔离级别



#### InnoDB的行锁模式及加锁方法

**共享锁（s）：**

又称读锁。允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。若事务T对数据对象A加上S锁，则事务T可以读A但不能修改A，其他事务只能再对A加S锁，而不能加X锁，直到T释放A上的S锁。这保证了其他事务可以读A，但在T释放A上的S锁之前不能对A做任何修改。

**排他锁（x）**：

又称写锁。允许获取排他锁的事务更新数据，阻止其他事务取得相同的数据集共享读锁和排他写锁。若事务T对数据对象A加上X锁，事务T可以读A也可以修改A，其他事务不能再对A加任何锁，直到T释放A上的锁。

**mysql InnoDB引擎默认的修改数据语句：**\*\*update,delete,insert都会自动给涉及到的数据加上排他锁，**select语句默认不会加任何锁类型\*\*，如果加排他锁可以使用select …for update语句**，加共享锁可以使用select … lock in share mode语句。\*\*所以加过排他锁的数据行在其他事务种是不能修改数据的，也不能通过for update和lock in share mode锁的方式查询数据，但可以直接通过select …from…查询数据，因为普通查询没有任何锁机制。\*\*

**InnoDB行锁实现方式**

InnoDB行锁是通过给\*\*索引\*\*上的索引项加锁来实现的，这一点MySQL与Oracle不同，后者是通过在数据块中对相应数据行加锁来实现的。InnoDB这种行锁实现特点意味着：只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁！

### 总结：

**对于MyISAM的表锁，主要讨论了以下几点：**

共享读锁（S）之间是兼容的，但共享读锁（S）与排他写锁（X）之间，以及排他写锁（X）之间是互斥的，也就是说读和写是串行的。

在一定条件下，MyISAM允许查询和插入并发执行，我们可以利用这一点来解决应用中对同一表查询和插入的锁争用问题。

MyISAM默认的锁调度机制是写优先，这并不一定适合所有应用，用户可以通过设置LOW\_PRIORITY\_UPDATES参数，或在INSERT、UPDATE、DELETE语句中指定LOW\_PRIORITY选项来调节读写锁的争用

由于表锁的锁定粒度大，读写之间又是串行的，因此，如果更新操作较多，MyISAM表可能会出现严重的锁等待，可以考虑采用InnoDB表来减少锁冲突。

**\*\*对于InnoDB表，主要讨论了以下几项内容：\*\***

（1）InnoDB的行锁是基于索引实现的，如果不通过索引访问数据，InnoDB会使用表锁。

（2）在不同的隔离级别下，InnoDB的锁机制和一致性读策略不同。

在了解InnoDB锁特性后，用户可以通过设计和SQL调整等措施减少锁冲突和死锁，包括：

* - 尽量使用较低的隔离级别； 精心设计索引，并尽量使用索引访问数据，使加锁更精确，从而减少锁冲突的机会；
* - 选择合理的事务大小，小事务发生锁冲突的几率也更小；
* - 给记录集显式加锁时，最好一次性请求足够级别的锁。比如要修改数据的话，最好直接申请排他锁，而不是先申请共享锁，修改时再请求排他锁，这样容易产生死锁；
* - 不同的程序访问一组表时，应尽量约定以相同的顺序访问各表，对一个表而言，尽可能以固定的顺序存取表中的行。这样可以大大减少死锁的机会；
* - 尽量用相等条件访问数据，这样可以避免间隙锁对并发插入的影响； 不要申请超过实际需要的锁级别；除非必须，查询时不要显示加锁；
* - 对于一些特定的事务，可以使用表锁来提高处理速度或减少死锁的可能。

## 锁机制 √

事务可以保证在并发访问下，数据的一致性，那么事务是如何做到数据一致性的？底层是用过锁机制实现的。

加锁时需要考虑锁 粒度 问题，常见的数据库锁粒度包括：

数据库、表、行。

一般情况下，锁粒度越小，效率越高，逻辑控制越复杂。粒度越大，效率越低，逻辑控制越简单。在实际工作环境中，大部分操作都是行级锁

## 隔离级别与锁的关系 √

关系型数据库常存在四种事务隔离级别：

- `READ UNCOMMITTED`：读未提交

- `READ COMMITTED`：读已提交

- `REPEATABLE READ` ：可重复读（默认值）

- `SERIALIZABLE`：序列化，串行执行，读操作一定要等写操作执行完成

​ 从上往下隔离级别越来越高，意味着数据越来越安全

在Read Uncommitted级别下，读取数据不需要加共享锁，这样就不会跟被修改的数据上的排他锁冲突

在Read Committed级别下，读操作需要加共享锁，但是在语句执行完以后释放共享锁；

在Repeatable Read级别下，读操作需要加共享锁，但是在事务提交之前并不释放共享锁，也就是必须等待事务执行完毕以后才释放共享锁。

SERIALIZABLE 是限制性最强的隔离级别，因为该级别锁定整个范围的键，并一直持有锁，直到事务完成。

|  |
| --- |
| 如何查看当前 session 的隔离级别是什么？  8.0之前：`select @@tx\_isolation;`  8.0之后：`select @@transaction\_isolation;`  如何设置事务的隔离级别？  `set global transaction isolation level read uncommitted`  执行完该语句后，当前 session 的隔离级别不会立刻更新，需要重新开连接，或者做 session 级别的调整：``set session transaction isolation level read uncommitted`` |

## 按照锁的粒度分数据库锁有哪些？锁机制与InnoDB锁算法

在关系型数据库中，可以按照锁的粒度把数据库锁分为行级锁(INNODB引擎)、表级锁(MYISAM引擎)和页级锁(BDB引擎 )。

MyISAM和InnoDB存储引擎使用的锁：

* MyISAM采用表级锁(table-level locking)。
* InnoDB支持行级锁(row-level locking)和表级锁，默认为行级锁

行级锁，表级锁和页级锁对比

行级锁 行级锁是Mysql中锁定粒度最细的一种锁，表示只针对当前操作的行进行加锁。行级锁能大大减少数据库操作的冲突。其加锁粒度最小，但加锁的开销也最大。行级锁分为共享锁 和 排他锁。

特点：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。

表级锁 表级锁是MySQL中锁定粒度最大的一种锁，表示对当前操作的整张表加锁，它实现简单，资源消耗较少，被大部分MySQL引擎支持。最常使用的MYISAM与INNODB都支持表级锁定。表级锁定分为表共享读锁（共享锁）与表独占写锁（排他锁）。

特点：开销小，加锁快；不会出现死锁；锁定粒度大，发出锁冲突的概率最高，并发度最低。

页级锁 页级锁是MySQL中锁定粒度介于行级锁和表级锁中间的一种锁。表级锁速度快，但冲突多，行级冲突少，但速度慢。所以取了折衷的页级，一次锁定相邻的一组记录。

特点：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般

## ****从锁的类别上分MySQL都有哪些锁呢？像上面那样子进行锁定岂不是有点阻碍并发效率了****

从锁的类别上来讲，有共享锁和排他锁。

共享锁: 又叫做读锁。当用户要进行数据的读取时，对数据加上共享锁。共享锁可以同时加上多个。

排他锁: 又叫做写锁。当用户要进行数据的写入时，对数据加上排他锁。排他锁只可以加一个，他和其他的排他锁，共享锁都相斥。

用上面的例子来说就是用户的行为有两种，一种是来看房，多个用户一起看房是可以接受的。一种是真正的入住一晚，在这期间，无论是想入住的还是想看房的都不可以。

锁的粒度取决于具体的存储引擎，InnoDB实现了行级锁，页级锁，表级锁。

他们的加锁开销从大到小，并发能力也是从大到小。

## ****MySQL中InnoDB引擎的行锁是怎么实现的？****

答：InnoDB是基于索引来完成行锁

例: select \* from tab\_with\_index where id = 1 for update;

for update 可以根据条件来完成行锁锁定，并且 id 是有索引键的列，如果 id 不是索引键那么InnoDB将完成表锁，并发将无从谈起

## ****InnoDB存储引擎的锁的算法有三种****

* Record lock：单个行记录上的锁
* Gap lock：间隙锁，锁定一个范围，不包括记录本身
* Next-key lock：record+gap 锁定一个范围，包含记录本身

相关知识点：

1. innodb对于行的查询使用next-key lock
2. Next-locking keying为了解决Phantom Problem幻读问题
3. 当查询的索引含有唯一属性时，将next-key lock降级为record key
4. Gap锁设计的目的是为了阻止多个事务将记录插入到同一范围内，而这会导致幻读问题的产生
5. 有两种方式显式关闭gap锁：（除了外键约束和唯一性检查外，其余情况仅使用record lock） A. 将事务隔离级别设置为RC B. 将参数innodb\_locks\_unsafe\_for\_binlog设置为1

## 什么是死锁？如何查看? 怎么解决？XDD

死锁是指两个或多个事务在同一资源上相互占用，并请求锁定对方的资源，从而导致恶性循环的现象。

常见的解决死锁的方法

1、如果不同程序会并发存取多个表，尽量约定以相同的顺序访问表，可以大大降低死锁机会。

2、在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁产生概率；

3、对于非常容易产生死锁的业务部分，可以尝试使用升级锁定颗粒度，通过表级锁定来减少死锁产生的概率；

如果业务处理不好可以用分布式事务锁或者使用乐观锁

XDD

处理死锁两种方式：https://www.cnblogs.com/wudanyang/p/10703832.html

1、死锁超时：设置锁超时时间，默认50S，innodb\_lock\_wait\_timeout

2、死锁检测：开启死锁监测，innodb\_deadlock\_detect（true/false）

## ****数据库的乐观锁和悲观锁是什么？怎么实现的？****

数据库管理系统（DBMS）中的并发控制的任务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。乐观并发控制（乐观锁）和悲观并发控制（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，屏蔽一切可能违反数据完整性的操作。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违反数据完整性。在修改数据的时候把事务锁起来，通过version的方式来进行锁定。实现方式：乐一般会使用版本号机制或CAS算法实现。

两种锁的使用场景

从上面对两种锁的介绍，我们知道两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下（多读场景），即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。

但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行retry，这样反倒是降低了性能，所以一般多写的场景下用悲观锁就比较合适。

## MySQL遇到的死锁问题、如何排查与解决

## 悲观锁和乐观锁的原理和应用场景；

## 数据库死锁如何解决；

## Mysql自增锁 √

自增锁是针对自增列自增长的一个特殊的表级别锁，即针对`AUTO\_INCREMENT`关键字的具体实现

在高并发场景下，对于大量插入操作，如何让自增列依次自增，不重复、不断裂是自增锁要达到的效果

关于自增锁有一个配置项`innodb\_autoinc\_lock\_mode`，不同值代表不同的锁模式：

* 0：传统方式。串行自增的，并且是连续的。这样需要独占的串行锁，语句完成才释放锁，性能最低
* 1：连续方式。自增并且是连续的。当语句申请到自增锁就释放自增锁，自增锁就可以给其它语句使用，性能比 0 好很多
* 2：交错方式。多语句插入数据时，有可能自增的序列号和执行先后顺不一致且中间可能有断裂。一次分配一批自增值，然后下个语句就再进行分配一批自增值，阻塞很小性能最高，这是 mysql 8.0 中的默认值

## **mysql有哪些锁，意向锁有什么**用; √

意向共享锁，简称IS，作用在于：通知数据库接下来需要施加什么锁并对表加锁。如果需要对记录A加共享锁，那么此时innodb会先找到这张表，对该表加意向共享锁之后，再对记录A添加共享锁

意向排他锁，简称IX，作用在于：通知数据库接下来需要施加什么锁并对表加锁。如果需要对记录A加排他锁，那么此时innodb会先找到这张表，对该表加意向排他锁之后，再对记录A添加排他锁

这两个锁都是系统自动添加和自动释放的，人工无法干预，锁的粒度是表级别

## **Mysql间隙锁 √**

间隙锁是innodb中的概念，也是一种排它锁。间隙锁的作用是锁住行与行之间的间隙，不让新的数据插入其中。

间隙锁是为了解决在repeatable read（可重复读）隔离级别下发生的幻读问题，通过以下两个方面实现：

* 防止间隙内有新数据被插入
* 的数据（例如防止numer=3的记录通过update变成number=5）

**next-key lock = 行锁 + 间隙锁**

单行锁（Record Lock）+间隙锁（Gap Lock）合起来就叫Next-Key Lock。

通过innodb\_locks\_unsafe\_for\_binlog参数配置为0，代表开启间隙锁；值为1代表关闭间隙锁，在MySQL8.0之前默认值为0，该参数在 8.0 版本被移除，说明一直是打开的。

间隙锁锁的是行与行之间的间隙，间隙锁有如下规则：

* - 加锁的基本单位是（next-key lock），\*\*前开后闭\*\*原则
* - 插入过程中访问的行纪录会增加锁
* - 索引上的等值查询，给唯一索引加锁的时候，next-key lock退化为行锁
* - 索引上的等值查询，向右遍历时最后一个值不满足查询需求时，next-key lock退化为间隙锁
* - 唯一索引上的范围查询会访问到不满足条件的第一个值为止

区间是左闭右开。用例子说明一下，有如下一张表，其中 id 是主键：





## gap锁能完全避免幻读问题么XDD

可以，但是可以设置gap锁失效，Innodb\_locks\_unsafe\_for\_binlog=true

# =========分库分表===========

说说分库与分表设计

分库与分表带来的分布式困境与应对之策（如何解决分布式下的分库分表，全局表？）

如何选择合适的分布式主键方案

常见的几种分布式ID的设计方案

选择合适的数据存储方案

# =========日志 && MVCC============

## Mysql中的日志 √

MySQL中的日志有三类，binlog、redolog和undolog。其中binlog是【服务端】的日志文件，redolog和undolog是【存储引擎】层的日志文件，且只在存储引擎是innodb时才存在

## 什么是 binlog √

binlog称为二进制日志，是服务端的日志，它存储了 mysql 中的所有数据，主要有 2 个作用：

* 通过binlog进行数据恢复
* mysql 主从复制时，从节点通过binlog同步主节点数据

binlog是**逻辑日志**，记录了每个语句的原始逻辑，并且采用追加写的方式写入日志文件，即顺序写入

一般在企业中数据库会有备份系统，可以定期执行备份，备份的周期可以自己设置。通过binlog恢复数据的过程如下：

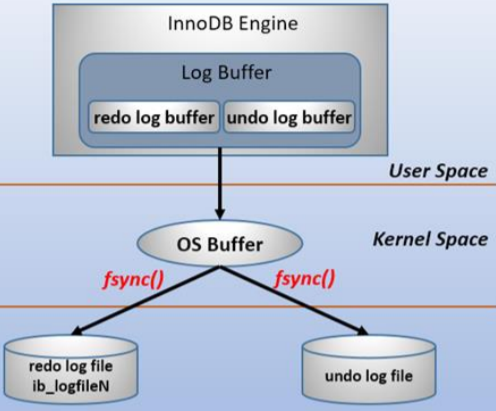
1. 找到最近一次的全量备份数据

2. 从备份的时间点开始，将备份的`binlog`取出来，重放到要恢复的那个时刻

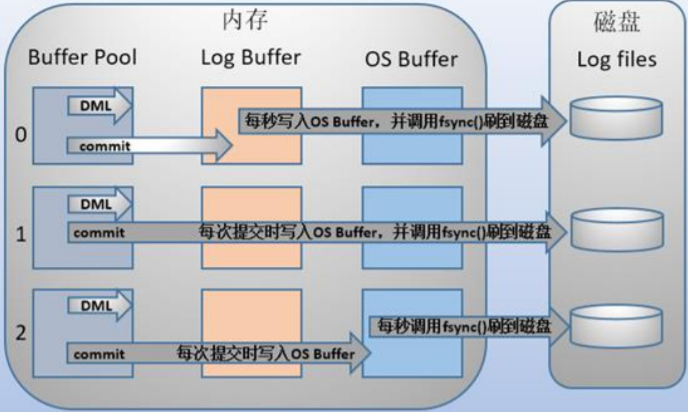
> 在 8.0 版本，`binlog`默认是开启的。可通过`select @@log\_bin;`查看当前配置

## 什么是redo日志、√

**redolog可以提高更新效率**。如果每次发生数据更新就写入磁盘触发IO，性能损耗严重。为了优化这一点，redolog用到的WAL技术，WAL的全称是Write-Ahead Logging，它的关键点是**先写日志再写磁盘**，并且**写日志是顺序写入内存**速度很快。redolog在【用户空间】中存在一个redo log buffer，当发生数据修改时，innodb引擎会先将记录写到redo log buffer中， 并更新内存，此时更新就算是完成了，同时innodb引擎会在合适的时机将redo log buffer写入到磁盘中



关于何时将redo log buffer内容写到磁盘，可通过innodb\_flush\_log\_at\_trx\_commit;来配置，该参数值可以为 0或1或2，具体含义看下图：



**nnodb\_flush\_log\_at\_trx\_commit：**

* **0：**

redulog在commit阶段，会将数据写到redo log buffer里，不同步执行刷到磁盘操作。然后 MySQL 会每秒执行一次redo log buffer的fsync操作将数据刷到磁盘。该模式下，当发生掉电时最多会丢失 1 秒内的全部操作。**速度最快，安全性最差**

* **1：**

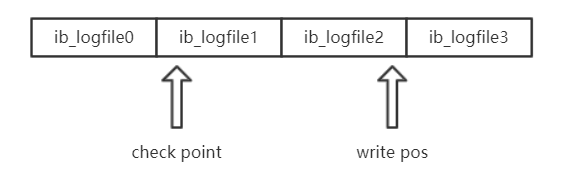
每次事务提交时都会把数据刷到磁盘上，这是 8.0 版本的默认值。该模式是**最安全但速度最慢**

* **2：**

每次事务提交时都会把数据写入到os buffer中，不同步执行刷到磁盘操作。然后 MySQL 会每秒执行一次os buffer的fsync操作。该模式速度较快，比 0 安全，如果有掉电保护组件的话，可以开启

redolog文件大小是固定的，是一个**物理日志**（记录的是在某个数据页上做了什么样的修改），使用的是循环写入的过程。

如下图所示，在redolog中存在两个指针，write pos代表下次数据写入的位置，可以一直往后写满ib\_logfile3和ib\_logfile0，最多写数据到check point位置。而check point到write pos之间则代表当前存在的数据



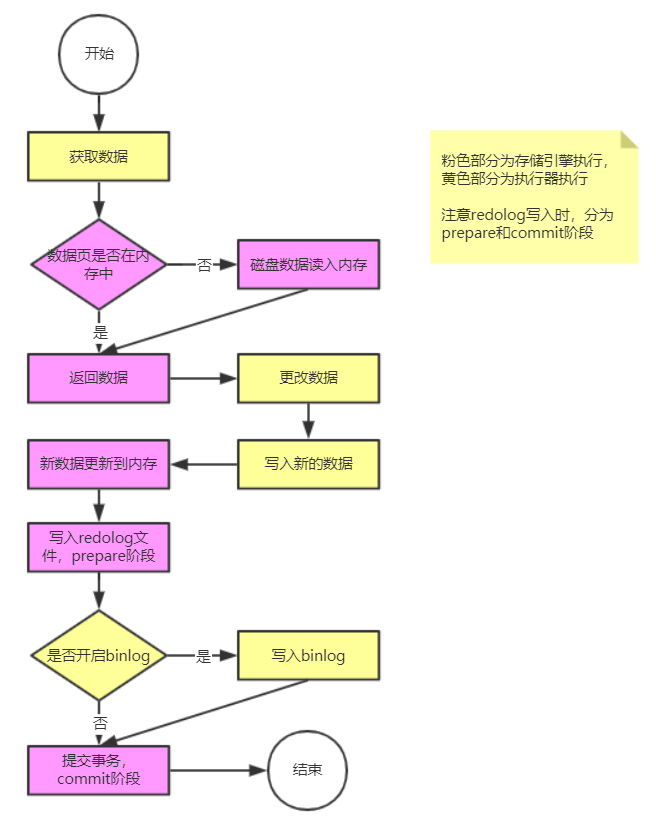
checkpoint是为了定期将db buffer的内容刷新到data file。当遇到内存不足、db buffer已满等情况时，需要将db buffer中的内容/部分内容（特别是脏数据）转储到data file中。在转储时，会记录checkpoint发生的”时刻“。在故障回复时候，只需要redo/undo最近的一次checkpoint之后的操作。

## mysql并发控制：MMVC、两阶段锁协议XDD

两阶段锁协议：1、加锁；2、释放锁

## 为什么redolog要分为两阶段？ \*\*\*\*\*\*

当出现redolog之后，数据的更新流程发生了改变



两阶段提交（2PC）

### 为什么 redolog要分为两阶段？

**为了保证redolog和binlog的数据一致性**。当恢复数据时，无论根据redolog还是binlog，得到的数据是一致的。

通过select @@log\_bin;可以查看是否开启binlog。如果没有开启，就不存在binlog，也没有两个日志文件数据据同步的说法了。

在 8.0 之前，可以通过参数innodb\_support\_xa配置决定是否开启两阶段提交，值为1时代表开启。而在 8.0 之后，始终启用两阶段提交

### crash-safe 是什么？

有了redolog之后，还有个好处：

mysql 可以保证即使数据库发生异常重启，根据binlog及redolog中记录的每个事务的状态，实现已提交的事务数据不丢失、未提交的事务数据进行回滚，这叫做 crash-safe，**保证了事务的持久性特点**

为了实现毫无差错的crash-safe，在写入redolog和binlog文件时必须进行一次fync操作将日志刷入磁盘，这样才能保证掉电时两个日志数据都写入了磁盘并且数据一致。因此redolog必需设置innodb\_flush\_log\_at\_trx\_commit=1。binlog也有决定何时进行fync的配置sync\_binlog，当值为0时代表刷新binlog\_cache中的信息到磁盘由os决定。当值大于0为N时，代表每N次事务提交后就刷新binlog cache中的信息到磁盘。因此还要设置sync\_binlog=1。两个参数值都得为1，俗称"双1"，是保证crash-safe的根本，官方也建议在需要保证事务持久性和一致性场景下设置成双1。

## 数据库宕机了如何进行恢复||数据恢复逻辑 √

情况一：开启binlog

binlog有记录，redolog状态commit：正常完成的事务，无需操作

binlog有记录，redolog状态prepare：在binlog写完提交事务之后的crash，提交事务

binlog无记录，redolog状态prepare：在binlog写完之前的crash，回滚事务

binlog无记录，redolog无记录：在redolog prepare之前crash，回滚事务

情况二：未开启binlog

redolog状态是commit，正常完成的事务，无需操作

redolog状态是prepare，回滚事务

## 什么是undo日志 √

**undolog可以保证事务的原子性**特点，还可以实现多版本并发控制（简称MVCC）。在操作任何数据之前，首先将数据备份到undolog中，然后进行数据的修改。如果出现了错误或者用户执行了ROLLBACK语句，系统可以利用undolog中的备份将数据恢复到事务开始之前的状态。

undolog是一个**逻辑日志**，可以这么理解：

* 当delete一条记录时，undolog中会记录一条对应的insert记录
* 当insert一条记录时，undolog中会记录一条对应的delete记录
* 当update一条记录时，它记录一条对应相反的update记录

当需要回滚时，执行undolog即可恢复到事务最开始的状态

## MVCC是什么？ √

多版本并发控制MVCC：是一种并发控制的方法，一般在数据库管理系统中，实现对数据库的并发访问，在编程语言中实现事务内存。

先了解两个概念：

* 当前读

像select lock in share mode(共享锁)，select for update ; update， insert ，delete(排它锁)这些操作都是一种当前读，为什么叫当前读？就是它读取的是记录的最新版本，读取时还要保证其他并发事务不能修改当前记录，会对读取的记录进行加锁

* 快照读

像不加锁的select操作就是快照读，即不加锁的非阻塞读；快照读的前提是隔离级别不是串行级别，串行级别下的快照读会退化成当前读；之所以出现快照读的情况，是基于提高并发性能的考虑，快照读的实现是基于多版本并发控制，即MVCC，可以认为MVCC是行锁的一个变种，但它在很多情况下，避免了加锁操作，降低了开销；既然是基于多版本，即快照读可能读到的并不一定是数据的最新版本，而有可能是之前的历史版本

**简单的说，MVCC就是为了实现读-写冲突不加锁**，**而这个读指的就是快照读，而非当前读，当前读实际上是一种加锁的操作，是悲观锁的实现**

## MVCC实现原理是什么？ √

MVCC的实现原理主要是依赖这三个东西：隐式字段，undo日志，Read View

### 隐式字段：

每行记录除了我们自定义的字段外，还有数据库隐式定义的DB\_TRX\_ID，DB\_ROLL\_PTR，DB\_ROW\_ID等字段。

|  |  |
| --- | --- |
| **DB\_TRX\_ID** | 最近修改(修改/插入)事务ID：记录 创建这条记录/最后一次修改该记录 的事务ID |
| **DB\_ROLL\_PTR** | 回滚指针，指向这条记录的上一个版本（存储于rollback segment里），配合undolog使用 |
| **DB\_ROW\_ID** | 隐含的自增ID（隐藏主键），如果数据表没有主键，InnoDB会自动以DB\_ROW\_ID产生一个聚簇索引 |
| **删除flag 隐藏字段** | 当记录被更新或删除并不代表真的删除，而是删除flag变了 |

### undo log

undo log主要分为两种

**insert undo log：**代表事务在insert新记录时产生的undo log，只在事务回滚时需要，并且在事务提交后可以被立即丢弃

**update undo log：**事务在进行update或delete时产生的undo log；不仅在事务回滚时需要，在快照读时也需要；所以不能随便删除，只有在快速读或事务回滚不涉及该日志时，对应的日志才会被 purge线程 统一清除

|  |
| --- |
| Purge（清除、清洗）线程：  - 为了实现InnoDB的MVCC机制，更新或者删除操作都只是设置一下老记录的deleted\_bit，并不真正将过时的记录删除  - 为了节省磁盘空间，InnoDB有专门的purge线程来清理deleted\_bit为true的记录。为了不影响MVCC的正常工作，purge线程自己也维护了一个read view（这个read view相当于系统中最老活跃事务的read view）；如果某个记录的deleted\_bit为true，并且DB\_TRX\_ID相对于purge线程的read view可见，那么这条记录一定是可以被安全清除的。 |

对MVCC有帮助的实质是update undo log ，undo log实际上就是存在rollback segment中旧记录链。

**例子说明MVCC**

|  |
| --- |
| 1. 有个事务向persion表插入了一条新记录，记录如下，隐式主键是1，事务ID和回滚指针假设为NULL   mvcc-part1   1. 现在来了事务1对该记录name字段进行了修改，改成了Tom  * - 在事务1修改该行(记录)数据时，数据库会先对该行加排它锁 * - 然后把该行数据拷贝到undo log中，作为旧记录。此时在undo log中有当前行的拷贝副本 * - 拷贝完毕后，修改该行name为Tom，并且修改隐藏字段的事务ID为当前事务1的ID。我们默认从1开始，之后递增，回滚指针指向拷贝到undo log的副本记录，既表示我的上一个版本就是它 * - 事务提交后，释放锁   mvcc-part2   1. 又来了事务2，修改了该条纪录，age变成了30  * 在事务2修改该行数据时，数据库也先为该行加锁 * 然后把该行数据拷贝到undo log中，作为旧记录，发现该行记录已经有undo log了，那么最新的旧数据作为链表的表头，插在该行记录的undo log最前面 * 修改该行age为30岁，并且修改隐藏字段的事务ID为当前事务2的ID，那就是2，回滚指针指向刚刚拷贝到undo log的副本记录 * 事务提交，释放锁   mvcc-part3  从上面可以看出，不同事务或者相同事务对同一记录的修改，会导致该记录的undo log成为一条记录版本链表，undo log的链首就是最新的旧记录，链尾就是最早的旧记录（当然像之前说的该undo log的节点可能会被purge线程清除掉，像图中的第一条insert undo log，其实在事务提交之后可能就被删除丢失了，不过这里为了演示，所以还放在这里） |

## mysql多版本并发控制，什么时间可以读到当前事物下的数据，可重复读使用原理XDD

1、创建版本小于等于当前事物版本

2、删除版本大于当前版本

## （什么是Write Ahead Log机制，什么是Double Write机制，什么是Check Point）

## 平时你们是怎么监控数据库的? 慢SQL是怎么排查的？（慢查询日志）

你们数据库是否⽀持emoji表情，如果不⽀持，如何操作?选择什么编码方式？如果支持一个表情占几个字节?(utf8mb4)；

## ****mysql有关权限的表都有哪几个****

MySQL服务器通过权限表来控制用户对数据库的访问，权限表存放在mysql数据库里，由mysql\_install\_db脚本初始化。这些权限表分别user，db，table\_priv，columns\_priv和host。下面分别介绍一下这些表的结构和内容：

user权限表：记录允许连接到服务器的用户帐号信息，里面的权限是全局级的。

db权限表：记录各个帐号在各个数据库上的操作权限。

table\_priv权限表：记录数据表级的操作权限。

columns\_priv权限表：记录数据列级的操作权限。

host权限表：配合db权限表对给定主机上数据库级操作权限作更细致的控制。这个权限表不受GRANT和REVOKE语句的影响。

## ****MySQL的binlog有有几种录入格式？分别有什么区别？****

有三种格式，statement，row和mixed。

statement模式下，每一条会修改数据的sql都会记录在binlog中。不需要记录每一行的变化，减少了binlog日志量，节约了IO，提高性能。由于sql的执行是有上下文的，因此在保存的时候需要保存相关的信息，同时还有一些使用了函数之类的语句无法被记录复制。

row级别下，不记录sql语句上下文相关信息，仅保存哪条记录被修改。记录单元为每一行的改动，基本是可以全部记下来但是由于很多操作，会导致大量行的改动(比如alter table)，因此这种模式的文件保存的信息太多，日志量太大。

mixed，一种折中的方案，普通操作使用statement记录，当无法使用statement的时候使用row。

此外，新版的MySQL中对row级别也做了一些优化，当表结构发生变化的时候，会记录语句而不是逐行记录。

# =========并发&集群============

MySQL并发情况下怎么解决（通过事务、隔离级别、锁）；

MySQL中的MVCC机制是什么意思，根据具体场景，MVCC是否有问题；

讲下MySQL的集群？集群遇到过什么问题？sql的优化？

数据库高并发下的优化思路;

项目 MySQL 的数据量和并发量有多大？量大后的影响有哪些，有考虑吗？SQL 调优有哪些技巧

项目数据库表是你设计的吗？一般要注意什么？如何考虑扩展性

# Sql题目

## 1，行列转换

### 行转列



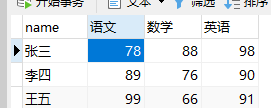
要求用一条 sql 语句查出如下结果：



思路：max(case when ) +group by

|  |
| --- |
| SELECT NAME,  max( CASE SUBJECT WHEN '语文' THEN score END ) ‘语文’,  max( CASE SUBJECT WHEN '数学' THEN score END ) ‘数学’,  max( CASE SUBJECT WHEN '英语' THEN score END ) ‘英语’  FROM  student\_score  GROUP BY  NAME |

### 列转行



要求用一条 sql 语句查出如下结果：

张三 语文 78

张三 数学 88

张三 英语 98

李四 语文 89

李四 数学 76

李四 英语 90

王五 语文 99

王五 数学 66

王五 英语 91

思路：使用union

