数据库100问（基于mysql）

数据库相关知识

1. 查询sql的执行流程，涉及到mysql的那些组件
2. sql更新语句涉及到哪些日志，这些日志如何保证mysql 数据的可恢复的
3. mysql事务的隔离级别，隔离级别的实现原理（MVCC多版本并发控制），如何选择事务的隔离级别
4. 索引的实现原理
5. 全局锁、表锁、行锁的原理，各自的优劣
6. mysql explain分析sql语句的用法
7. count（*） 太慢如何做统计
8. 如何设计索引
9. order by是怎么工作的

1. 慢sql查询如何优化

1. 使用慢查询日志、或`show processlist;` 定位慢查询sql（查看info字段）
2. 使用explain 查询sql是不是走了索引
3. 没有走索引的需要优化sql让sql走索引
4. 如果走了索引，查询很慢，则要考虑根据查询条件优化索引

2. 什么是索引

索引是帮助数据库高效获取数据的排好序的数据结构
索引的数据结构：
1. 二叉树 可能会退化成链表导致查询效率低下
2. 红黑树 树的深度过高导致磁盘IO次数过多
3. Hash表 由于存储是无序的，无法进行范围查找
4. B+Tree(目前最常用的索引结构) 一般的索引树层级都在1-3层

3. 数据类型（mysql）

1. 数值类型

```
>   tinyint     1byte    -128 127
>   smallint    2bytes   -32768 32767
>   mediumint   3bytes
>   int/integer 4byte
>   bigint      8bytes
>   float       4byte
>   double      8byte
>   decimal（5,2）：总共五位，小数点后两位
```

2. 时间和日期类型

> date 3bytes yyyy-mm-dd time 3byte hh:mm:ss year datetime yyyy-mm-dd hh:mm:ss timestamp 时间戳

3. 字符串类型 （单引号）

> varchar 变长字符串 char 定长字符串 0-255 bytes TEXT

mysql的配置

查看mysql中的各种配置参数：

```
--查看引擎相关的
show variables like '%engine%';
-- 查看数据库事务的隔离级别
show variables like 'transaction-isolation';
```

4. DDL、DML、DQL

DDL

DDL： data definition language 数据定义语言

数据库操作

> 查看所有数据库： `sql show databases;` 创建数据库：`sql create database [if not exists] mydb1 [charset=utf8]` 切换或选择要操作的数据库： `sql use mydb1;` 删除数据库：`sql drop database [if exists] mydb1;` 修改数据库访问编码： `sql alter database mydb1 character set utf8;`

表操作

创建库：

```
create table [if not exists] tablename(
  id int(10) [auto_increment] [comment '主键id'],
  name varchar(255) [not null] [comment '名称'],
  age int,
  birth date
)[];
```

查看表：

> `sql show tables;` 查看表的创建语句 `sql show create table table1` 查看表结构 `sql desc table1;`

修改表结构

> 添加表列：`sql alter table table1 add column1 varchar(20) [default null];` 修改列名：`sql alter table table1 change `column1` `column2` varchar(30) [default null];` 删除列：`sql alter table script_execute_request_record_1 drop column script_content,drop column request_script_content;` 修改表名：`sql rename table `table1` to `table2`;`

DML

DML: data manipulation language 数据操作语言 数据插入：

> 指定字段赋值：`sql insert into table1(column1,column2) values(value1,value2),(value3,value4);` 全字段赋值：`sql insert into table1 values(value1,value2,value3),(value4,value5,value6);` 数据修改：`sql update table1 set column1=v1,column2=v2 [where column1 = v3];` 数据删除：`sql delete from table1 [where column = v1];sql truncate [table] table1;` 删除表，然后重新创建表

约束（mysql）constraint

创建表时为表中的某些列添加限制条件

主键约束 primary key

唯一标识表数据的每一行（可以多个列组合） 表只允许有一个主键 系统默认会在所在的列和列组合上创建对应的唯一索引 主键约束列不能为null 创建单列主键

```
create table table1(
    id int(11) primary key,
    name varchar(20) not null
);
create table table1(
    id int(11) auto_increment,
    name varchar(20) not null,
    [constraint pk1] primary key(id)
);
```

创建多列主键 (主键参与列都不能为空)

```
create table table1(
  name varchar(20) not null,
  age int(11) not null
  [constraint pk1] primary key(name,age)
);
```

修改表结构后添加主键

```
alter table table1 add primary key(name,age);
```

删除主键约束

```
alter table `table1` drop primary key;
```

自增长约束 auto_increment

一个表只能有一个自增长约束 为主键自动赋值

```
create table table1(
     id int(11) auto_increment,
     name varchar(20) not null,
     [constraint pk1] primary key(id)
 ) auto_increment = 100; --从100开始自增
 insert into table1(name) value('lisi');
```

修改的方式添加：

```
alter table table1 auto_increment = 200;
```

delete 删除数据auto_increment自增会保留 truncate 删除数据auto_increment自增长会从起始值开始

非空约束 not null

创建表时指定字段非空 修改字段时指定

```
alter table table1 modify column1 varchar(20) not null;
```

删除非空约束

```
alter table table1 modify column1 varchar(20)
```

唯一约束

在mysql中NULL和任何值都不匹配、不相同,所以唯一约束可以允许多个NULL值出现,创建唯一约束时要保证约束列不能为NULL

指所有记录中字段的值不能重复出现 创建 字段名后加上unique

```
create table table1(
    id int(11) primary key,
    name varchar(20) unique not null  --列指定
);
```

在所有字段的最后指定唯一约束的列

```
create table table1(
    id int(11) primary key,
    name varchar(20)  not null,
    gender tinyint not null,
    UNIQUE KEY `uk1` (`name`,`gender`)
);
```

修改添加约束：

```
alter table table1 add constraint uk1 unique(name,age);
```

删除约束

```
alter table table1 drop index uk1;
```

默认值约束

给字段默认值

```
create table table1(
    id int(11) primary key,
    name varchar(20) not null default 'unknown'
);
```

零填充约束

填充长度到字段的指定长度

```
create table table1(
    id int(11) zerofill primary key, --整型zerofill默认长度是10
    name varchar(20) not null default 'unknown'
);
```

外键约束 约束多张表之间的关系

> 主表必须时存在于数据库中 从表的外键必须是主表的主键

创建

```
    create table table1(
      id int(11) primary key,
      name varchar(20)  not null,
      gender tinyint not null,
      UNIQUE KEY `uk1` (`name`,`gender`)

  );
    --从表
  create table table2(
     id int(11) primary key,
     name varchar(30) not null,
     dept_id int(11) not null,
     constraint t1_pk foreign key(dept_id) references table1(id); --外键约束
  )
```

修改方式创建

```
alter table table2 add constraint t1_pk foreign key(dept_id) references
table1(id) on delete cascade on update cascade; --级联删除级联更新
```

删除外键

```
alter table table2 drop foreign key ti_pk;
```

> 多对多的关系使用中间表来进行约束

DQL -data query language 数据查询语言

查询操作

```
select
      [all|disntinct] column1 as a1,
      [all|disntinct] column2 as a2,
from
   table1|view1
[where [condition]]
group by [column1]
having expression
order by [column1] [asc|desc]
limit [0,100]
```

关键字

> 去重某一列 `distinct` 去重所有列 `distinct *` 如果列具有NULL值，并且对该列使用DISTINCT子句，MySQL将保留一个NULL值，并删除其它的NULL值，因为DISTINCT子句将所有NULL值视为相同的值。
>
> 查询区间 `between a and b`
>
> `least ()` 查询最小值 (如果比较的值存在null，不会进行比较，直接返回nul)
>
> `greatest ()` 查询最大值 (如果比较的值存在null，不会进行比较，直接返回nul)
>
> `ifnull(a,0)` 如果a的值不是null，返回a，如果是null，返回0

聚合函数

> count() 统一某一字段时，不统计null值行
>
> sum() 不统计null值的行
>
> min() 不统计null值的行
>
> max() 不统计null值的行
>
> avg() 平均值，不统计null值的行

执行顺序(在前面执行的字句产生的结果才能在后面的字句中使用)由于sql的书写顺序与执行顺序的不一致，了解sql的执行顺序有助于写好sql

1. from table1 [left|right] join table2 先确定表
2. join on (join condition)
3. where (where condition) select 中指定的别名无法在where中使用，而from ... join ...指定的表别名可以使用
4. group by col1,col2 (当出现group by时，select查询的字段必须是group by的字段或聚合函数)
5. having having_condition （可以使用select字句中的字段别名） 分组操作会执行聚合函数count(*) as c，avg(col3) as a,sum(col4) as s等，在having中可以使用聚合函数产生的别名字段
6. select col1,col2,count(*) as c，avg(col3) as a,sum(col4) as s.....
7. distinct
8. union [all] 求并集
9. order by col1 [desc|asc]排序
10. limit 100,10

gpt对执行顺序的回答

- FROM: 首先，数据库从指定的表中选择数据。
- JOIN: 如果在查询中使用了JOIN子句，数据库会根据指定的条件将多个表连接起来。
- WHERE: 在连接和表选择之后，应用WHERE子句中指定的条件来过滤行。只有满足条件的行会被保留。
- GROUP BY: 如果存在GROUP BY子句，数据库会按照指定的列对结果进行分组。

- HAVING: HAVING子句用于在GROUP BY之后对分组进行过滤，只保留满足条件的分组。
- SELECT: 在执行完所有的连接、过滤和分组操作后，数据库会根据SELECT子句中指定的列选择要返回的数据。
- DISTINCT: 如果使用了DISTINCT关键字，数据库会确保返回的结果中没有重复的行。
- ORDER BY: ORDER BY子句用于对结果进行排序，可以根据一个或多个列进行升序或降序排序。
- LIMIT: 最后，LIMIT子句用于限制返回的结果行数。

需要注意的是，虽然上述顺序是一般情况下的执行顺序，但数据库优化器可能会对查询进行优化，例如重新排列子句的顺序以提高性能。此外，子查询、联合查询等复杂情况可能会影响执行顺序。在实际使用中，你可以通过查看查询的执行计划（EXPLAIN语句）来了解数据库是如何处理你的查询的。

分组

> `group by column1 column2 having condition` 分组查询的select字段只能是分组字段或者聚合函数 `having` 是对`group by`的结果进行筛选

分页

> `limit n` 显示默认的前n条 `limit m,n` 从第m条开始，显示n条数据

查询并插入insert into ... select .... *插入的字段名称必须与查询到的名称同名，且字段类型尽量保持一致*

```
select
   column1,column2
from
   table2
where
   [condition]
```

正则表达式

> `regexp ^a` 是否以a开头 `regexp b$` 是否以b结尾 `regexp .a .`匹配任意单个字符（除了换行符） `regexp [xyz]` []内任意一个字符是否在匹配对象中出现 `regexp [^abc]` 匹配除了 [...] 中字符的所有字符,^在[]中出现表示取反的意思，其他地方表示以某某开头 `'abc' regexp [^a]` 返回1 `regexp (taa)*` *匹配0个或多个taa，包括空字符串 `regexp a+` a出现一次或多次 `regexp a?` a出现0次或一次 `regexp a|b` 匹配a或b `regexp ^(a|b)` 以a或b开头 `regexp a{m}` 匹配m个a `regexp a{m,}` 匹配m个及多个a `regexp a{m,n}` 匹配m到n个a `regexp ^d(abc)` ()内的abc存在时匹配，普通字符匹配

多表操作

多表关系

> 一对一 一对多 多对多

交叉连接(笛卡尔积)

> 两张表项乘 会产生大量冗余数据

```
select * from a,b;
```

内连接（两张表的交集） [inner] join

```
-- 隐式内连接
select * from a,b where a.id = b.cid;
-- 显式内连接
select * from a inner join b on a.id = b.cid;
```

```
-- 分组、排序、join
select
   t1.deptno as deptno,
   t1.name as name,
   count(1) as count
from dept3 t1
inner join emp3 t2
on t1.deptno = t2.deptId
group by t1.deptno,t1.name
having count >3  order by count desc;
```

外连接

left [outer] join

```
-- 右表没有对应的数据补null
select * from dept a left outer join emp b on a.deptno = b.dept_id;
```

right [outer] join

```
--左表没有对应的数据补null
select * from dept a right outer join emp b on a.deptno = b.dept_id;
```

full [outer] join (mysql 中使用的union,将查询的结果上下拼接，并去重)

```
select * from dept a left outer join emp b on a.deptno = b.dept_id
union --会自动去重
select * from dept a right outer join emp b on a.deptno = b.dept_id;
```

union all 表示不去重

```
select * from dept a left outer join emp b on a.deptno = b.dept_id
union all
select * from dept a right outer join emp b on a.deptno = b.dept_id
```

子查询 select的嵌套查询 子查询返回结果类型

> 单行单列

```
   select
       *
   from
      emp3
   where age = (
      select
         max(age)
      from emp3);
```

> 单行多列

```
   --多表查询实现
   select * from emp3 a join dept3 b  on a.deptid = b.deptno and b.name in
('aaaa','bbbbb');
   --子查询实现
   select
      *
   from emp3 a
   where a.deptid in (
      select
         deptno
      from dept3
      where name in ('aaaa','bbbbb'));
```

> 多行单列

```
   --多表查询实现
   select * from emp3 a join dept3 b on a.deptid = b.deptno and b.name =
'aaaa' and a.age <= 20;
   -- 子查询
   select * from emp3 a where deptid in (select deptno from dept3 where
name = 'aaaa') and a.age <=20;
```

> 多行多列

```
select * from (select * from dept3 where name = 'aaaa') t1 join (select *
from emp3 where age <30) t2 on t1.deptno=t2.deptid
```

相关关键字

> all --集合中的所有元素

```
select * from table1 where age > all(select age from table2)  --大于子查询中
的所有年龄
```

```
select * from emp3 where deptid <> all(select deptno form dept3); --不与集合
中的任何值匹配
```

> any/some --集合中的任意值比较

```
select * from emp3 where age > any(select age from table2) --大于子查询中的任
意一个人的年龄
```

> in --是其中的任意一个

```
select * from emp3 where detpid in(select deptno from dept);
```

> exists 存在数据满足并返回数据 (用in可以达到一样的效果)

```
select * from emp3 a where exists  ( select * from emp3 b where a.deptid
=b.deptno );
```

多表关联查询的效率会比子查询高，尽量使用关联查询 表自关联

> 自关联表必须取别名

```
select a.name as name ,b.name as manager from emp1 e1 left join emp1 e2 on
e1.manager = e2.id;
```

5. mysql的函数

聚合函数

常见的

> count()、sum() min() max() avg() **group_concat()** 合并某一列

```
select group_concat(emp_name) from emp; --默认分隔符 ,
select group_concat(emp_name separator ';') from emp; --指定；分隔符
select department, group_concat(emp_name separator ';') from emp group by
department; -- 先分组后拼接
select department, group_concat(emp_name order by salary desc separator
';') from emp group by department; -- 先分组后按某顺序排序后拼接
```

数学函数

> abs(x) 求绝对值 ceil(x) 向上取整 ceil(1.4)= 2 floor(x) 向下取整 floor(1.4) = 1 greatest(a,b,c) 取列表最大值 least(a,b,c) 取列表最小值 max(column1) 要传列名，或与列名相关的表达式 min(column1) 要传列名，或与列名相关的表达式 mod(5,2) 取余 power(2,3) 求2的3次方 rand()*100 100以内的随机数 round(3.433232，3) 四舍五入，保留3位小数 truncate(x,y) 截取x的小数保留y位小数

字符串函数

```
  select char_length('hello')/character_length('hello')  --查询字符串长度，一
个汉字代表一个字符
  select length('你好') --length返回字节数
  select concat('hello ,','world'); --返回合并字符串
  select concat_ws('-','hello','world') --返回用分隔符合并的字符串
  select field('aaaa','aaaa','bbb','ccc') --=1  返回字符串在字符串列表中第一次出
现的位置
  select ltrim('    hello') --去除字符串左边的空格
  select rtrim('world     ')--去除右端空格
  select trim('   hello world     ')--去除两端空格
  select mid('helloworld',2,3) --从第二个字符开始截取，截取长度为3
  select position('abc' in 'hello abc world')  --= 6 判断一个字符串在另一个字符
串第一次出现的位置
  select replace('helloaaaworld','aaa','bbb') --将第一个字符串中的aaa替换成bbb
  select reverse('hello') =  --olleh 字符串反转
  select right('hello' ,3) --= llo 返回右边的几个字符
  select strcmp('hello','world') --= -1 比较两个字符串的大小
  select substr('hello',2,3)/substring('hello',2,3) --字符串截取,从第二个字符开
始截取，截取三个
  select ucase('hello') --小写变大写
  select upper('hello') --将字符串转为大写
  select lcase('HELLO') --将字符串转为小写
  select lower('HELLO') --将字符串转为小写
```

日期函数

```
   select unix_timestamp(); --获取时间戳（ms值）
   select unix_timestamp('2022-01-01 08:08:08'); --字符串日期转时间戳
   select from_unix_time(unix_timestamp(),'%Y-%m-%d %H:%i:%s'); --时间戳毫秒
```

```
值转为字符串格式
   select current_date()/curdate(); --获取当前年月日
   select current_time()/curtime(); --时分秒
   select current_timestamp(); 获取年月日时分秒
   select date('2022-01-01 08:08:08'); --从日期字符串中获取年月日
   select datediff('2022-01-20','2022-01-02'); --获取日期之间的差值
   select timediff('08:08:08','08:07:08'); --获取时间差值，秒级
   select date_format('2022-1-20 8:8:8','%Y-%m-%d %H:%i:%s'); --日期格式化
   select str_to_date('2022-1-20 8:8:8','%Y-%m-%d %H:%i:%s')  --字符串转日期
   select date_sub('2022-01-20', interval 2 day/second/month/year/minute)
= 2022-01-18 --将日期进行减法
   select date_add('2022-01-20', interval 2 day/second/month/year/minute) =
2022-01-22 --将字符串日期进行假发
   select extract(hour/month/year/minute/second/day from '2022-01-01
08:08:08')--从日期中获取小时/天等
   select last_day('2022-01-20') --获取给定日期的所在月最后一天
   select makedate('2021',53) --2021-02-22 指定年份的某一天的日期
   select year/month/minute  ('2022-01-01 08:08:08') --获取各种级别时间
   select quarter('2022-01-20 08:08:08') --获取季度
   select monthname('2022-01-20 08:08:08') --获取月名称
   select dayofmonth('2022-01-20 08:08:08') --这个月的第几天
   select dayofweek('2022-01-20 08:08:08') --1是星期日，2是星期1
   select dayofyear('2022-01-20 08:08:08') --是一年的第几天
   select dayname('2022-01-20 08:08:08') --指定日期周几
   select week('2022-01-20 08:08:08') --获取指定日期是第几周，从0开始
   select weekday('2022-01-20 08:08:08') --是周几
   select yearweek('2022-01-20 08:08:08') --年份和第几周
```

控制流函数

if 逻辑判断

```
select if(5>3,'大于','小于'); --类似于java中的三目运算符
select ifnull('abc','0'); --如果是null，返回'0'
select isnull('abc'); --判断表达式是否为null
select nullif(12,12); --如果两个表达式一样，返回null,否则返回第一个表达式的值
```

case when语句

```
   select
      *,
      case [column1]
         when column1=1 then 'hello'
         when column1=2 then 'world'
         when column1=5 then 'hello world'
         else
            'wrong'
      end as info
```

```
    from
       table1;
```

窗口函数（mysql8.0之后添加的）

进行聚合的同时保留分组内的所有数据 序号函数

```
row_number()|rank()|dense_rank() over(
   partition by ...
   order by ...
)
```

```
    --对部门员工按薪资排序，并给出排名
    --row_number()
    select
       dname,
       ename,
       salary,
       row_number() over(partition by dname order by salary desc) as rn
     from
       emp;
    --rank():薪资相同时rank值相同，序号不连续
     select
       dname,
       ename,
       salary,
       rank() over(partition by dname order by salary desc) as rn1
     from
       emp;
     --dense_rank() 薪资相同时rank值相同，序号连续
     select
       dname,
       ename,
       salary,
       dense_rank() over(partition by dname order by salary desc) as rn2
    from
       emp;
    -- 分组求topn
    select * from  (
       select
             dname,
             ename,
             salary,
             row_number() over(partition by dname order by salary desc) as
rn
       from
          emp
    ) as t
    where t.rn <= 3
```

```
    -- 全体排序，去掉partition by
    select
       dname,
       ename,
       salary,
       row_number() over(order by salary desc) as rn
    from
       emp;
```

分布函数 查询分组内小于等于当前rank值的行数/分组内总行数

```
    --cume_dist()
    select
       dname,
       ename,
       salary
       cume_dist() over(partition by dname order by salary) as rn1,
       cume_dist() over(order by salary) as rn1
    from
       emp;
    --percent_rank() 不常用
    select
       dname,
       ename,
       salary
       percent_rank() over(partition by dname order by salary) as rn1
    from
       emp;
```

前后函数 返回位于当前行的前n行（lag（exp，n））或后 n行（lead（exp，n））的exp值 应用查询前一名同学和当前同学成绩的差值

```
    --
     select
       dname,
       ename,
       salary,
       hirdate,
       lag(hirdate ,1,'2000-01-01') over(partition by dname order by
hirdate) as time1, --排序后上一行的值，没有的补默认值2000-01-01
       lag(hirdate ,2) over(partition by dname order by hirdate) as time2
--排序后上2行的值，没有的补null
       --lack(hirdate ,1,'2000-01-01') over(partition by dname order by
hirdate) as time1 --排序后下一行的值
    from
       emp;
```

头尾函数 返回第一个或最后一个的表达式值 场景，截止到当前，按日期排序查询的第一个入职和最后一个入职员工的薪资

```
   select
      dname,
      ename,
      salary,
      first_value(salary) over (partition by dname order by hiredate) as
fisrt,
      last_value(salary) over (partition by dname order by hiredate) as
last
    from
      emp;
```

其他 nth_value(exp,n) 返回窗口中第n个exp的值 场景：截止到当前薪资，显示每个员工薪资中排第二或第三的薪资

```
    select
      dname,
      ename,
      salary,
      nth_value(salary,2) over (partition by dname  order by hiredate) as
v2,
      nth_value(salary,3) over (partition by dname order by hiredate) as
v3
    from
      emp;
```

ntile() 将分区中的有序数据分为n个等级，记录等级数 场景，将每个部门员工按照入职日期分成3组

```
 select
      dname,
      ename,
      salary,
      ntile(3) over (partition by dname order by hiredate) as v1

    from
      emp;
```

6. 视图view

数据库中只存储视图定义，视图中的数据还是存储在原来的物理表中

> 简化代码,将常用的查询结果集封装成视图 保证数据安全

创建

```
create [or replace] [algorithm = { undifiend|merge|temptable}]
   view view_name[(column_list)]
   as select_statement
[with [cascade|local] check option]
```

```
create or replace
view view1_emp
as
select ename,job from emp;

show full tables; --查询视图和表

select * from view1_emp;
```

修改

```
alter view view1_emp
as
select a.deptno,b.dname,a.loc.b.ename from dept a,emp b where a.deptno =
b.deptno
```

更新视图数据（实际上是更新原表数据） sql中有聚合函数、有distinct字段、有group by having不能更新、有union、union all 、join不能更新、from字句中有视图不能更新、包含子查询不能更新 **一般情况下只将视图作为虚拟表，不要通过视图更新数据 重命名**

```
rename table view1_emp to view2_emp;
```

删除视图

```
drop view if exists view2_emp;
```

7.存储过程（mysql5.0之后的版本）

> 一组sql的语句集 sql语言的代码封装与重用

创建

```
--删除
drop  procedure if exists proc01;
delimiter $$
create procedure proc01()
```

```
begin
   select empno,ename from emp;
end $$
delimiter ; --将分隔符恢复到默认
```

调用

```
call proc01();
```

使用变量

```
delimiter $$
create procedure proc02()
begin
   declare var_name01 varchar(20) default 'qqq'; --定义
   set var_name01  = 'lisi'; --赋值
   select var_name01; --使用
end  $$
delimiter ;
```

```
delimiter $$
create procedure proc03()
begin
   declare var_name01 varchar(20) default 'qqq'; --定义
   select  ename into var_name01 from emp where empno = 1001;
   set var_name01  = 'lisi'; --赋值
   select var_name01; --使用
end  $$
delimiter ;
```

使用用户变量

全局有效，不需要声明

```
delimiter $$
create procedure proc04()
begin
   set @var_name01  = 'lisi'; --赋值
   select  @var_name01;
end  $$
delimiter ;
select @var_name01; --可在外部使用
```

系统变量

全局变量：mysql启动时服务器给的默认值my.ini 会话变量：mysql将当前所有的全局变量复制一份作为会话变量 全局变量操作

```
show global variables; --查看所有全局变量
show @@global.auto_increment_increment;
set global sort_buffer_size = 40000;
set @@global.sort_buffer_size = 40000;
```

会话变量操作

```
show session variables; --查看所有全局变量
show @@session.auto_increment_increment;
set session sort_buffer_size = 40000;
set @@session.sort_buffer_size = 40000;
```

存储过程传参in

入参

```
delimiter $$
create procedure proc06(in param_empno int)
begin
   select * from user where id = param_empno
end $$
delimiter ;
call proc06(1001);
```

存储过程传参out

出参

```
delimiter $$
create procedure proc07(in param_empno int,out out_ename varchar(20))
begin
   select username into out_ename from user where id = param_empno;
end $$
delimiter ;
call proc07(1,@out_ename);
select @out_ename;
```

存储过程传参inout

可修改参数并传出

```
delimiter $$
create procedure proc08(inout num int)
begin
   set num = num * 10;
end $$
delimiter ;

set @inout_num = 2;
call proc08(@inout_num);
select @inout_num;


delimiter $$
create procedure proc09(inout inout_ename varchar(30),inout sal int)
begin
   select concat_ws('_',deptno,ename) into inout_ename from emp where
emp.ename = inout_ename;
   set sal = sal*12;
end $$
delimiter ;
set @inout_name = 'lisi';
set @inout_sal = 3000;
call proc09(@inout_name,@inout_sal);
select @inout_name;
select @inout_sal;
```

流程控制 if elseif then else end if

```
delimiter $$
create procedure proc10(in socre int)
begin
 if score <60
   then select 'not pass';
 elseif score >=60 and score <80
   then  select 'pass';
 elseif score >=80 and score <90
   then select 'great pass';
 elseif score >=90 and score <=100
   then select 'full pass' ;
 else
   then select 'error';
  end if;
end $$
delimiter ;

set @score = 55;
call proc10(80);
```

```
delimiter $$
create procedure proc11(in in_ename varchar(30))
begin
declare var_sal decimal(7,2);
declare var_result varchar(20);
select
   sal into var_sal,
where ename = in_ename;
if var_sal<10000
   then set var_result = 'shiyongxinzi';
elseif var_sal<20000
   then set var_result = 'zhuanzhengxinzi';
else
   set var_result = 'creatorxinzi';
 end if;
select var_result;
end $$
delimiter ;
set @name = 'guanyu';
call proc11(@name);
```

流程控制 case when end case

```
delimiter $$
create procedure proc12(in pay_type int)
begin
   case pay_type
      when 1 then select 'weixinzifu';
      when 2 then select 'zifubaozifu';
      when 3 then select 'yinhangkazifu';
      else select 'other';
   end case;
end
delimiter $$;
call proc12(1);
```

```
delimiter $$
create procedure proc13(in in_ename varchar(30))
begin
declare var_sal decimal(7,2);
declare var_result varchar(20);
select
   sal into var_sal,
where ename = in_ename;
case
   when var_sal<10000
      then set var_result = 'shiyongxinzi';
   when  var_sal<20000
```

```
        then set var_result = 'zhuanzhengxinzi';
    else
        set var_result = 'creatorxinzi';
 end case;
select var_result;
end $$
delimiter ;
set @name = 'guanyu';
call proc13(@name);
```

流程控制--循环 while、repeat

```
--while 循环 leave 跳出循环
--向表中添加数据
delimiter $$
create procedure proc14(in insert_count int)
begin
    declare i int default 1;
    lable: while i <= insert_count
        insert into user(username,password) values(concat('user-
',i),'123456');
        if i = 5
            then leave lebel; --跳出标签位置的循环
        end if;
        set i = i+1;
    end while label;
end $$
delimiter ;
call proc14(10);
```

```
--while 循环 iterate 跳过本次循环继续进行下一次循环
--向表中添加数据
delimiter $$
create procedure proc15(in insert_count int)
begin
    declare i int default 1;
    lable: while i <= insert_count
        insert into user(username,password) values(concat('user-
',i),'123456');
        set i = i+1;
        if i = 5
            then iterate lebel; --跳出标签位置的循环
        end if;
    end while label;
end $$
delimiter ;
call proc14(10);
```

repeat until

```
--repeat 循环 iterate 跳过本次循环继续进行下一次循环
--向表中添加数据
delimiter $$
create procedure proc15(in insert_count int)
begin
   declare i int default 1;
   lable: repeat i <= insert_count
      insert into user(username,password) values(concat('user-
',i),'123456');
      set i = i+1;
      until i > insert_count --跳出循环，不要加分号
   end repeat label;
   select 'repeat end';
end $$
delimiter ;
call proc15(10);
```

loop

```
delimiter $$
create procedure proc16(in insert_count int)
begin
   declare i int default 1;
   lable: loop i <= insert_count
      insert into user(username,password) values(concat('user-
',i),'123456');
      set i = i+1;
      if i > insert_count --跳出循环
         then leave label;
      end if;
   end loop label;
   select 'repeat end';
end $$
delimiter ;
call proc16(10);
```

游标cursor 对查询结果集进行额外处理

```
drop procedure if exists proc16;
delimiter $$
create procedure proc16(in in_dname varchar(30))
   declare var_empno int;
   declare var_ename varchar(50);
   declare var_sal decimal(7,2);
--声明游标
   declare mycursor cursor for
```

```
      select empno,ename,sal,from dept a,emp b
      where a.deptno = b.deptno and a.dname = in_dname;
 --打开游标
 open mycursor;
--通过游标获取数据
fetch mycursor into var_empno,var_ename,var_sal;
l1:loop --loop 没有推出，会报错
select var_empno,var_ename,var_sal;
end loop l1;
--关闭游标
close mycursor;
begin
end $$
delimter ;

call proc16('sales');
```

异常处理handler句柄 对存储过程的错误异常处理

```
drop procedure if exists proc16
delimiter $$
create procedure proc17(in in_dname varchar(30))
   declare var_empno int;
   declare var_ename varchar(50);
   declare var_sal decimal(7,2);
   --句柄的操作
   --定义异常标记值
   declare flag int default 1;

   --触发条件


--声明游标
   declare mycursor cursor for
      select empno,ename,sal,from dept a,emp b
      where a.deptno = b.deptno and a.dname = in_dname;
   --定义异常处理句柄，异常处理完成后，剩余代码是否执行（continue/exit）
   declare continue handler for NOT FOUND set flag = 0
   --打开游标
   open mycursor;
   --通过游标获取数据
   fetch mycursor into var_empno,var_ename,var_sal;
   l1:loop --loop 没有推出，会报错
   if flag =1
      then
         select var_empno,var_ename,var_sal;
   else
      leave l1;
   end if;
   end loop l1;
   --关闭游标
   close mycursor;
```

```
begin
end $$
delimter ;
call proc17('sales');
```

使用场景：可以通过存储过程创建下个月的每张表

8.存储函数

创建存储函数

```
drop function if exists my_function1;
--无参数存储函数
delimiter $$
create function my_function1() returns int
begin
   declare cnt int default 0;
   select count(*) into cnt from emp;
   return cnt;
end $$
delimiter ;

--调用
select my_function1();

--有参数存储函数
drop function if exists my_function2;
--无参数存储函数
delimiter $$
create function my_function2(in param_empno int) returns varchar(50)
begin
   declare out_empname varchar(50);
   select ename into out_empname  from emp where empno = param_empno;
   return out_empname;
end $$
delimiter ;

--调用
select my_function2(1001);
```

9. 触发器

> 特殊的存储过程，对代码进行封装。 当表中的数据发生变更时（insert/update/delete），会自动触发 触发器的调用,不需要手动调用 触发器可以作为日志记录工具来使用 mysql只支持行级触发 定义什么条件触发 什么时候触发 触发频率 不能对本表进行触发操作 尽量少用触发器 对增删改频繁的表不要使用触发器 **查看触发器**

```
show triggers;
```

创建触发器

```
--单条语句触发
create trigger my_trigger1 after|before insert|update|delete
on user
for each row
insert into user_logs values(null,now(),'new user');
--多条语句触发
create trigger my_trigger2 after|before insert|update|delete
on user
for each row
delimiter $$
begin
   insert into user_logs values(null,now(),'new user');
   insert into user_logs values(null,now(),'new user');
end $$
delimiter ;
```

new old 在触发器内部获取更新前后的数据

```
--insert
drop trigger if exists my_trigger3;
create trigger my_trigger3 after insert
on user
for each row
insert into user_logs values(null,now(),concat_ws('_','new
user:',NEW.id,NEW.name,NEW.password));

--update
drop trigger if exists my_trigger4;
create trigger my_trigger4 after update
on user
for each row
insert into user_logs values(null,now(),concat_ws('_','update user,old
data:',OLD.id,OLD.name,OLD.password));
--update
drop trigger if exists my_trigger5;
create trigger my_trigger5 after update
on user
for each row
insert into user_logs values(null,now(),concat_ws('_','update user,new
data:',NEW.id,NEW.name,NEW.password));
--delete
drop trigger if exists my_trigger6;
create trigger my_trigger6 after delete
on user
for each row
insert into user_logs values(null,now(),concat_ws('_','delete user,old
data:',OLD.id,OLD.name,OLD.password));
```

10. mysql的索引

> 通过某种算法构建某种数据结构，提供在某一列中快速找到某一行数据的功能 索引的目的是提供查询数据的效率

分类

按数据结构

- hash索引 ： 根据索引列为每一列的值生成一个hash算法（极端情况下会产生哈希冲突），
  - hash索引由于是无序索引，在进行范围查找和排序时会很低效
  - hash索引对磁盘利用率不高，桶之间的分布时随机的，可能会对磁盘利用率有影响
  - hash索引插入效率高

> b+tree索引：生成树型结构的数据，通过数值大小进行排序，有效支持范围查找，顺序存储，磁盘利用率高，插入时可能需要对索引进行重新排序，效率较低

按功能

> 单列索引 ： 普通索引、唯一索引、主键索引 --对表中某一列创建索引 组合索引 ： 多列组合成一个索引 全文索引 空间索引

在mysql中的索引又分为主键索引（聚簇索引）和普通索引（二级索引，非主键索引）

- 主键索引：索引树的叶子结点键是主键信息，值存储的是整行数据
- 普通索引：索引树的叶子结点的键是索引信息，值存储的是主键值 回表的概念：通过普通索引树查询数据时，首先根据普通索引查询到主键值，让后通过主键值到主键索引树上取出索引对应的行数据 查询普通索引后通过回表查询主键索引，实际上是使用了普通索引和主键索引进行了联合查询，所以在设计索引时需要注意索引的合理性 **在设计索引时，应根据实际的应用场景合理设计索引中的字段，使用覆盖索引（即索引中的字段完全包含查询语句中需要获取的字段），尽量避免因为查询过程中的回表带来的时间开销**

单列索引-普通索引

```
create table student(
   id int primary key,
   card_id varchar(30),
   gender varchar(20),
   age int,
   birth date,
   score double,
   phone_num varchar(11),
   index index_name (`name`)  --建表时创建普通索引，允许值重复
);
create index index_gender on student(gender); --单独创建普通索引
alter table student add index index_age(age); --修改表结构的方式创建索引
--查看表的所有索引
show index from sutdent;
--删除索引
```

```
drop index index_gender on student;
alter table student drop index index_age;
```

单列索引-唯一索引，列必须唯一，且不能为空值

```
create table student2(
   id int primary key,
   name varchar(30) not null,
   card_id varchar(30),
   gender varchar(20),
   age int,
   birth date,
   score double,
   phone_num varchar(11),
   unique index index_name (`name`)  --建表时创建唯一索引
);
alter table student2 add unique index index_age(age);
create unique index_gender on student2(gender);
drop unique index index_gender on student2; --删除
alter table student2 drop index index_age; --删除
```

单列索引-主键索引

主键列唯一且不能为空，主键列会创建主键索引

组合索引--使用表中多个字段添加索引

```
create index index_phone_name on student(phone_num,name);
create unique index u_index_card_name on student(card_id,name); --按最左原则进行匹配，查询条件中必须有card_id才能进行索引匹配，如果直接使用name进行条件过滤，是不会进行索引匹配的
select * from student where name = 'lisi'; --不走索引，因为无法匹配card_id
select * from student where card_id = '1002032';--走索引，使用card_id进行匹配
select * from student where name = 'lisi' and card_id = '1002032'; --mysql会优化查询条件顺序，走索引，使用完整的索引列进行匹配
```

全文索引 fulltext

> 不需要使用where语句进行参数匹配，数据量大时使用全文索引比like快 5.6 之后的版本MyISAM、InnoDB支持全文索引 只有数据类型为char、varchar、text的字段才能使用全文索引 表中存在大量数据时再创建全文索引的效率高 使用时需要修改全文索引最小查询长度（如果为3，则查询3个及3个以上的字符进行匹配） 查看字段设置show variables like '%ft%';

```
create table article(
   id in primary key auto_increment,
   title varchar(255) not null,
```

```
    content varchar(1024),
    writing_date date
);
create fulltext index index_content on article(content);--插入数据后创建，会提高创建的效率
select * from article where match(content) aginst('you');
```

空间索引--

> 对空间数据类型结构支持 索引字段不能为空

索引的原理

> 索引以文件的形式存储在磁盘上 索引的查找过程中会产生IO 索引的结构组织要尽量减少查找过程中的磁盘IO

hash算法

> 通过字段的值计算哈希值，定位数据非常快 不能进行范围查找，因为散列表的值时无序的，无法进行大小的比较

搜索二叉树

> 分为左子树、右子树，根节点，左子树的值比根节点小，右子树的值比根节点大 有可能产生不平衡，树的层级太深，极端情况下可能产生类似于链表的结构，导致查询的io次数很多 查询范围也会比较麻烦

平衡搜索二叉树

> 左子树和右子树都是平衡二叉树 左子树比中间小，右子树比中间大 左子树和右子树的深度之差绝对值小于等于1 插入需要旋转 查询范围时，需要回旋，查询效率较低 存放数据量过大的情况下，树的高度会很高，查询效率会很慢

Btree树（平衡的多路查找树） mysql默认情况下就是用的b+tree

> 分为b-tree、b+tree

b-tree（以非叶子结点指针数量为3(max.degree为3)为例，一个节点最多存储两个数据值），叶子结点和非叶子结点都有数据

> 一个节点最多存储两个数据， 每个非叶子节点由2个key和3个指针组成 所有叶子节点具有相同的深度，等于树高h。 添加数据时，如果被添加的节点存储数据量大于等于3，节点进行分裂，则将中间的值向上提升为父节点，两边进行分裂 一个节点存在三个指针，左边的指向比节点中小的子节点，中间的指向夹在当前节点两个值中间的数值节点，右边的指向比右边值大的子节点 查找值时，使用指针进行遍历，根据值的比较快速定位到对应的节点中的值 查找范围时也能很快定位，不需要进行旋转遍历节点的最大深度越大，树的高度越低，进行磁盘io的次数越少

b+tree(所有的数据都是在叶子结点上)

> 构建树的过程与b-tree一致 当结点上的数据大于3个时，将中间结点向上提升，但底层的叶子结点依然包含中间结点的数据，且将叶子结点使用双向链表进行连接 最底层的叶子节点使用双向链表进行链

> 接，存储了索引上的所有数据（方便范围查找），由于底层的叶子结点具有相同的深度，所以可以方便的创建双向链表 叶子结点在磁盘上存放在连续的空间上，进一步提高查找速度

B-树和B+树的区别 B-树内部节点是保存数据的;而B+树内部节点是不保存数据的，只作索引作用，它的叶子节点才保存数据。 B+树相邻的叶子节点之间是通过链表指针连起来的，B-树却不是。 查找过程中，B-树在找到具体的数值以后就结束，而B+树则需要通过索引找到叶子结点中的数据才结束 B-树中任何一个关键字出现且只出现在一个结点中，而B+树可以出现多次。

- 在MyISAM中，叶子结点除了存储索引列的值以外，还存储了真实数据存放的地址，查找时先查找到地址，再通过地址取数据（回表）
- 在InnoDB中，叶子结点存放的是数据，比MyISAM效率会高一点，但索引结构占用硬盘空间更大

索引的优缺点 *优点*

> 大大加快数据查询速度 使用分组和排序进行数据查询时，可以显著减少查询时分组和排序的时间 创建唯一索引能保证数据库表中每一行数据的唯一性 在实现数据的参考完整性方面，可以加速表和表之间的连接 *缺点* 创建索引和维护索引需要消耗时间 索引要占用磁盘空间 对数据的频繁增删改，索引需要进行动态维护，降低了维护速度

索引创建原则

> 更新频繁的列不应设置索引 数据量小的表不用索引 重复数据多的字段不应该设置索引（如性别，各种有穷枚举） 首先应该考虑where和order by 涉及的列上建立索引,合理设置max_length_for_sort_data和sort_buffer_size，尽量避免文件排序（文件排序的explain分析extra字段会显示using filesort）

普通索引和唯一索引的选择

普通索引允许索引值不唯一，可以使用到change buffer，在日志数据、账单数据等这类历史数据库（写多读少的场景），建议使用普通索引。如果写入的数据马上需要进行查询，可以使用唯一索引

索引相关的概念和面试题

1. 什么是聚集索引：数据行的物理顺序与列值（一般是主键的那一列）的 逻辑顺序相同，一个表中只能拥有一个聚集索引，索引的叶子结点对应的数据节点，可以直接获取到对应的全部列的数据，非聚集索引索引没有覆盖到对应的列时需要进行二次查询（回表）
2. 非聚集索引：索引中索引的逻辑顺序与磁盘上的无力存储顺序不同，一个表中可以有多个非聚集索引
3. 回表：select所需获取的列有非索引列，索引需要到表中找到相应的列信息，这叫做回表
4. 覆盖索引：select查询的数据列只用从索引中就能获取到，不必读取数据行，查询的列被所建的索引所覆盖

mysql中的锁

全局锁：对整个库加锁，使用命令`flush tables with read lock`对库进行全局加锁，一般都是在进行数据库备份时（一般的备份都是在从库上进行备份）避免其他线程对数据操作带来数据不一致性的问题 表级锁：通过lock tables ...read/write来对某一个张表进行加锁 MDL锁（metadata lock）：对表的元数据进行加锁，避免DDL操作带来的表结构的变更导致的查询不一致的问题 行锁：通过减少锁冲突来提高业务的并发度。在InnoDB事务中，行锁是在需要时才添加上的，在事务结束时才释放（两阶段锁）。如果在事务中需要锁多个行，就把最有可能导致锁冲突、影响并发度的锁尽量往后放 行锁容易造成数据库的死锁，对于死锁的策略：

- 设置innodb_lock_wait_timeout来等待锁超时
- 将参数innodb_deadlock_detect设置为on，发起死锁检测

在备份数据库时，最好确保表使用的是InnoDB引擎，同时在mysqldump中开启--signle-transaction参数

11.mysql存储引擎

> 数据存储的底层软件组织 不同存储引擎提供不同的存储方式、索引技巧和锁粒度不同 需要根据具体的业务需求选择不同的存储引擎

查看存储引擎

```
show engines;
```

MyISAM mysql 5.5之前的默认的引擎，不支持事物，不支持外键 查询快

InnoDB 目前版本mysql的默认引擎 支持事务 支持外键

为表指定引擎

```
show varibables like `%default_storage_engine%`; --查看当前的存储引擎
create table1(id int,name varchar(20)) engine = MyISAM; --创建时指定引擎
alter table table1 engine = InnoDB; --修改存储引擎
```

12.事务transaction

> mysql InnoDB支持事务 事务用来维护数据库的完整性，保证sql要么全部执行，要么全部不执行

事务操作

```
set autocommit = 0;--取消数据库的自动提交
begin;--开启事务
update account set money = money -200 where id = 1002;
update account set money = money + 200 where id = 1003;
commit; --提交事务
rollback;
set autocommit = 1;--恢复自动提交
```

事务特性 ACID

> 原子性 事务是一个不可分割的整体，事务内的sql要么全部执行，要么全部不执行 一致性 系统从一个正确状态迁移到另一个状态，结果保证完全一致 隔离性 数据库对多事务并发执行的一个控制，描述了多个事务并发执行时，事务与事务之间操作数据的可见性。 持久性 事务一旦提交，则其结果是永久的

事务的隔离级别 isolate：将一个事务与其他事务隔离开 隔离级别：

> read uncommitted 读未提交，一个事务能读取到另一个未提交事务的数据（脏读） read committed 读已提交，一个事务要等到另一个事务提交后才能读取到相关的数据，可避免脏读，会造成不可重复读 repeatable read 可重复读 （mysql默认级别），造成幻读，mysql的InnoDB可以解决幻读问题 serializable 序列化串行

不可重复读：A事务在没有提交事务期间读取B事务操作的数据是不同的（也就是B事务事务开启前和事务提交后的数据是不同的），针对数据update的情况

- 事务A读取了某一行数据的值。
- 此时，事务B修改了该行数据的值并提交事务。
- 事务A再次尝试读取同一行数据，但此时读到的数据已经被事务B修改过，与之前读取的值不同。 *可重复读 在可重复读隔离级别下，事务的主要特点包括：
- 读取一致性: 事务在开始时会创建一个读取视图，这个视图包含了事务开始时数据库中的快照。在事务执行期间，所有的读操作都基于这个读取视图，从而保证了事务内部多次读取同一数据时的一致性。
- 幻读问题: 可重复读隔离级别可以避免不可重复读问题，但是仍然可能会出现幻读问题。幻读指的是，在同一个事务中，某次查询返回了一个范围内的记录，但是在随后的查询中，由于其他事务插入了新的数据，同样的查询可能返回了不同的记录。为了解决幻读问题，可以使用锁或者在更高的隔离级别下操作，如Serializable。
- 锁机制: MySQL在可重复读隔离级别下会自动对查询的数据行加锁，以确保事务之间的隔离性。当事务在读取数据时，会对读取的数据行加共享锁，防止其他事务对这些数据进行修改。而当事务执行INSERT、UPDATE、DELETE等修改操作时，会对涉及的数据行加排他锁，防止其他事务同时进行读取或修改。

幻读：针对多条数据的情况，一次事务中前后读取到的数据量发生了变化，A事务开启事务执行delete操作，B事务执行插入操作然后提交，此时A事务查询数据总条数发现数据量和删除前的一样，和预期不一样，针对数据delete，insert的情况 操作

```
show variables like `%isloation%`;
set autocommit = 0;
set session transaction isolation level read uncommitted;--设置隔离级别为读未提交，引起脏读
set session transaction isolation level read committed; --引起不可重复读
set session transaction isolation level repeatable read; --引起幻读
```

```
show variables like `%isloation%`;
set autocommit = 0;
```

12 mysql锁机制

锁是计算机协调多个进程或线程并发访问资源的机制 锁分类

> 表锁，锁定整个表 MyISAM使用，不会出现死锁，查询效率高 行锁，操作时锁定当前行 InnoDB使用，开销大，可能出现死锁，锁粒度小，发生锁冲突小

操作分类

> 读锁 写锁 **MyISAM引擎锁操作**

```
    --指定表引擎为MyISAM
    lock table tb_book read; --加读锁，只能读，不能修改，可重复加锁

    unlock tables; --释放锁

    lock table tb_book write; --独享锁，仅加锁的session能读写，当 其他session访问时不允许读也不允许写
    unlock tables; --释放锁
```

InnoDB行锁 开销大，加锁慢，粒度小，发生冲突概率低 InnoDb支持事务，采取行锁

> 共享锁，多个事务对同一数据共享一把锁，都能访问数据，但不能修改数据 排他锁 不能与其他锁并存，如果一个事务获取了一行数据的排他锁，其他事务不能再回去该行数据的其他锁，只有获取锁的事物可读可写 InnoDB对insert，update,delete 默认加上排他锁 InnoDB通过间隙锁和next-key lock可以在RC的隔离级别下解决幻读

13. mysql日志

分类

> 错误日志 查看：`sqlshow variables like 'log_error%'` 存放错误信息 二进制日志 mysql8.0之前需要 在my.ini 添加 log_bin = mysqlbin(二进制文件名前缀),重启服务器

- statement格式 记录sql
- row 记录数据变更信息
- mix 混合格式
- 查看binlog日志格式 `sql show variables like ‘binlog_format’;`
- 查询指定的binlog 日志 `sql show binlog events in 'binlog.10000' from 10 limit 1,3;`

> 查询日志

- 查看是否开启 show variables like 'general_log';
- set global general_log = 1; 开启查询日志

> 慢查询日志

- 是否开启 `sql show variables like '%slow_query_log%';`
- 临时开启 `sql set global slow_query_log = 1;`
- 限制慢查询超时时间 `sql show variables likes 'long_query_time%';set gobal long_query_time = 10; --限制为10s`

14. mysql 优化

整体优化方案

> 设计上优化（库表关系优化） 查询上优化 索引上优化 存储上优化

查看服务器状态信息

```
    show session status like 'Com_____'; --当前会话的增删改查、事务提交次数
    show global status like 'Com_____'; --查看全局的增删改查、事务提交次数
    show status like 'Innodb_rows%' --查看针对InnoDB的统计
```

定位低效执行的sql

- 开启慢查询日志，通过慢查询日志进行定位慢查询sql 配置信息

```
slow_query_log = 1
slow_query_log_file = /path/to/slow-query.log
long_query_time = <阈值，以秒为单位>
```

- 定位低效执行的sql：

```
    show processlist; --查询客户端连接服务器的线程执行状态信息
```

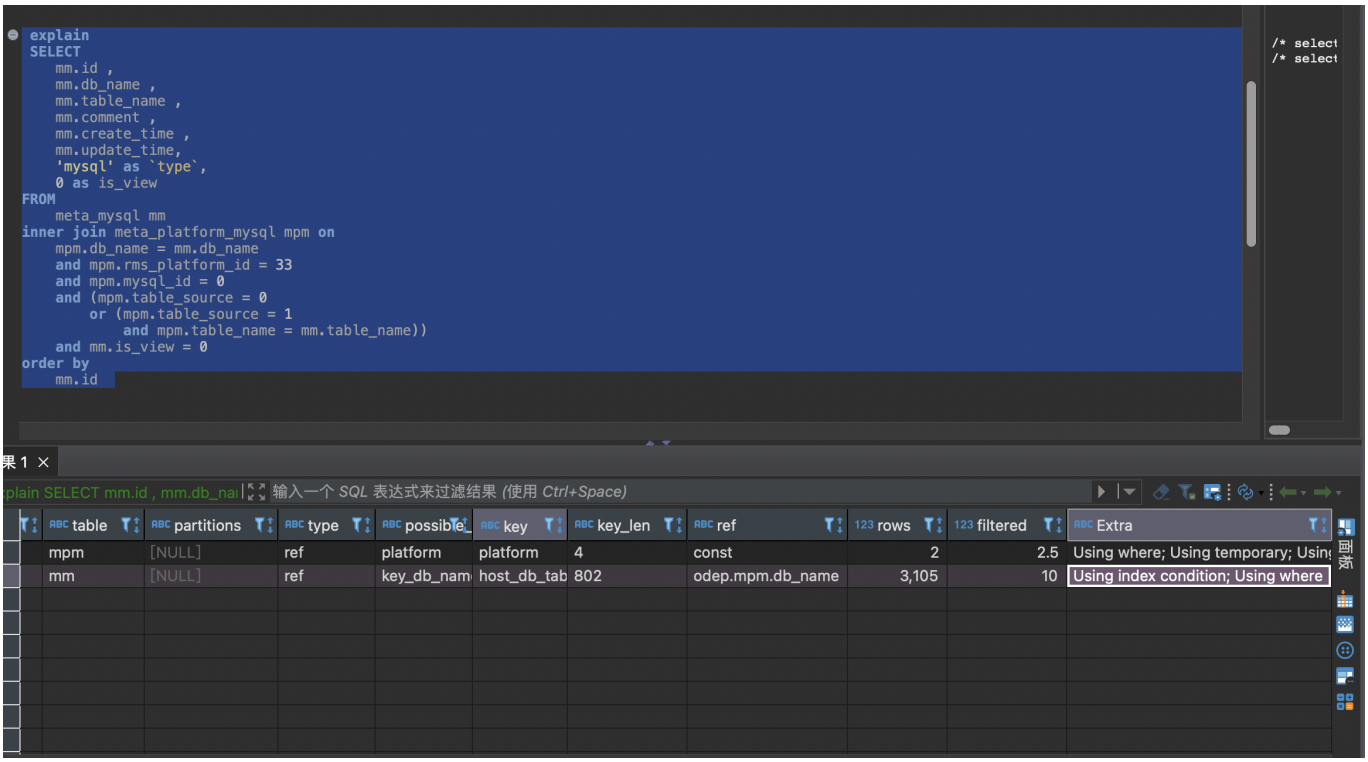
explain分析执行计划

通过分析执行计划，可以确定以下优化点：

- 索引优化：通过分析执行计划中的索引扫描操作，可以确定是否需要创建新的索引或修改现有的索引以提高查询性能。
- 表优化：通过分析执行计划中的表扫描操作，可以确定是否需要对表进行分区、分片或垂直拆分等优化操作。
- 连接优化：通过分析执行计划中的连接操作，可以确定是否需要调整连接顺序、使用连接方式（如嵌套循环连接、哈希连接、排序-合并连接等）以提高查询性能。
- 查询重写：通过分析执行计划中的过滤条件等信息，可以确定是否需要对查询语句进行重写以减少查询时间和资源消耗。
- 数据统计优化：通过分析执行计划中的代价估算信息，可以确定是否需要收集更准确的数据统计信息以提高查询优化器的准确性和效率。

```
explain
 SELECT
    mm.id ,
    mm.db_name ,
    mm.table_name ,
    mm.comment ,
    mm.create_time ,
    mm.update_time,
    'mysql' as `type`,
    0 as is_view
FROM
    meta_mysql mm
```

```
    inner join meta_platform_mysql mpm on
        mpm.db_name = mm.db_name
        and mpm.rms_platform_id = 33
        and mpm.mysql_id = 0
        and (mpm.table_source = 0
            or (mpm.table_source = 1
                and mpm.table_name = mm.table_name))
        and mm.is_view = 0
    order by
        mm.id
```



> id： select 查询的序号，表加载顺序,id值越大，加载越优先,而对于相同id的行，则表示从上往下依次执行。

> select_type: 查询类型

- simple 不包含子查询和union
- primary 主查询，子查询的最外层查询
- subquery 子查询中的第一个select
- dependent subquery 子查询中的第一个select，依赖了外面的查询
- union sql语句中出现了union关键字
- union result 使用union关键字
- derived 在from中包含子查询（衍生）
- materialized 物化子查询
- union 在union中的第二个和随后的select被标记为union
- union result union的结果
- uncacheable subquery 子查询，结果无法缓存，必须针对外部查询的每一行重新评估

> table： 当前这一行查询的表，如果在sql中定义的别名，则展示表的别名

> type： 表连接类型 查询效率 system>const>eq_ref>ref>range>index>all

- null 不访问任何表任何索引直接返回结果 `sql select now()`
- system 查询系统表，少量数据，不需要进行磁盘io，system是const类型的特例，msyql5.7以上的版本访问系统表是all
- const 查询条件是主键索引或唯一索引，且判断条件是常量 `where id =1`，最多只返回一条数据
- eq_ref 主键索引(primary key)或者非空唯一索引(unique not null)等值扫描,使用了索引的全部组成部分，前表的每一行对应后表只有一行被扫描，join查询，且命中的左表主键索引（主键列），且用等值比较 (如果不是一对一的关系，则是all) ```a join b on a.id = b.pid``
- ref 非主键非唯一索引等值扫描,查询条件是非唯一索引 多表查询时，前表的每一行对应后表只有一行被扫描，join查询，且命中的左表普通索引最左前缀（普通索引列），且用等值比较 (如果不是一对一的关系，则是all)
- fulltext：全文索引
- index_merge：此类型表示使用了索引合并优化，表示一个查询里面用到了多个索引
- unique_subquery：该类型和eq_ref类似，但是使用了IN查询，且子查询是主键或者唯一索引
- index_subquery：和unique_subquery类似，只是子查询使用的是非唯一索引
- range：范围扫描，表示检索了指定范围的行，主要用于有限制的索引扫描。比较常见的范围扫描是带有BETWEEN子句或WHERE子句里有>、>=、<、<=、IS NULL、<=>、BETWEEN、LIKE、IN()等操作符。
- index：全索引扫描，和ALL类似，只不过index是全盘扫描了索引的数据
- all 全表扫描，性能最差

最左前缀原则，指的是索引按照最左优先的方式匹配索引 如果一个联合索引的字段顺序为a,b,c 当查询条件为a，c时，不会走索引中的c条件查询 最左匹配原则在遇到范围查询的时候，就会停止匹配。 原因分析： 当联合索引为（a,b,c）时 a在全局上是有序的，而当a相等时才按照b进行排序，b相等时才按照c进行排序，所以b在a相等的情况下局部有序，c在a,b相等的情况下局部有序 如果a使用不等值判断（如 > < between and），如 a > 1 and b = 1,则接下来b不会走索引,只有a会走索引 a = 1 and b > 2 and c = 3 :只有a、b走索引 a = 1 and b between 1 and 3 and c =3 也是只有a,b 走索引

> possible keys 可能用到的索引，这一列的数据是在优化过程的早期创建的，因此有些索引可能对于后续优化过程是没用的。 key 实际用到的索引 key_len 索引字段长度（字节）指用到的索引列的字段长度（如果索引列为 int+varchar(20)+int,用了前两个，则长度为24 ） row 扫描行的数量,数值越小越好 filtered 符合查询条件的数据百分比,百分比越大效率越高

> extra 执行情况的说明和描述

- using index 使用覆盖索引的情况下，extra字段会显示该值
- Using index condition 当where中使用的联合索引有部分条件生效时出现
- (Using where) where 子句中出现了非索引的查询条件，有可能是全表扫描，当where中存在有效索引时可能出现：Using where，Using index
- using filesort 说明mysql会对数据使用一个外部的索引排序，而不是按照索引顺序进行读取，称为文件排序，效率低. *MySQL中无法利用索引完成的排序操作称为“文件排序”*
- using temporary 需要建立临时表暂存结果，常见于order by 和group by，效率低
-
- Impossible HAVING HAVING子句始终为false，不会命中任何行
- Impossible WHERE WHERE子句始终为false，不会命中任何行

show profile分析 mysql 5.0.37之后保本支持

```
select @@have_profiling;
set profiling = 1;--打开profile支持
select ...
select ...
select ....
show profiles; --查看以上sql执行的耗时情况
show profile for query 20; --查询某一条sql的执行耗时状态
show profile cpu for query 20; --查询cpu资源消耗情况
```

trace分析优化器执行计划

```
set optimizer_trace = "ebable = on" ,end_markers_in_json=on;
set optimizer_trace_max_mem_size = 10000000;
select * from information_schma.optimizer_trace \G;
```

索引优化 索引优化能解决大多数mysql的性能优化问题

> 避免索引失效 --当使用组合索引时，保证索引的全值匹配（至少要保证最左原则） 最左前缀匹配，保证组合索引最左边的字段能匹配 组合列索引时，当使用范围查询时，范围查询之后不能在使用其他匹配条件，其他匹配条件不会走索引 查询条件等号左侧不能做运算，否则索引失效 字符串索引时，如果不加单引号索引会失效 查询列尽量用名称覆盖，尽量使用覆盖索引（字段都是索引列），避免使用*号 （尽量让查询能从索引树中拿到所有的列数据，而直接使用*需要从磁盘中读取原表数据---*回表*） 组合列索引时，用or分割开的条件，如果or前的条件列没有索引，而后面的列有索引，则涉及的索引都不会被用到 以%开头的like模糊查询索引失效，当select字段是最左前缀规则下的索引列时（不使用*号），则以%开头的like还是可以走索引的 如果mysql认为频繁使用索引列比全表更慢，则不会使用索引列（索引中出现大量的重复值时会出现这种情况） is NULL,is not null 有时索引有效，有时索引失效 in 走索引， not in 索引失效（使用主键索引列匹配in 和not in都能走索引） 避免使用大量重复数据的列作为索引列 单列和复合索引，尽量使用复合索引 *复合索引 name+status+address相当于name，name+status,name+status+address三个索引*

回表 mysql中的索引是根据索引列的值进行排序的，所以索引节点中存储了参与索引列的值，当一次select查询中所查询的列存在非索引的字段，mysql在索引中找不到对应列的数据，这时需要查找主键索引取回相应的数据列，这个过程叫回表。 回表查询回消耗一定的时间，所以在设计索引或编写sql时，应该尽量保证查询列都在索引中，避免回表带来的时间消耗

叶子节点存储其他列的值（非主键列）的情况，有时被称为"覆盖索引"，使用覆盖索引可以避免查询时回表，但是这样会增加索引的存储空间 MySQL的查询优化器在决定是否使用覆盖索引时，会考虑多个因素，包括表的结构、数据分布、查询条件等。在某些情况下，优化器可能无法选择合适的覆盖索引。因此，即使你尝试了上述步骤，也不能保证每个查询都会使用覆盖索引。

sql优化

> 大批量插入数据

- InnoDB下通过load infile 插入数据时，主键顺序插入可以提高插入效率
- `sql set global local_infile = 1;`

- `sql load data local infile 'dxxxx' into table tb_user fields terminated by ',' lines terminated by '\n';`
- 插入时关闭唯一索引关闭唯一索引校验可以提交插入效率
- `sql set unique_checks = '1';set unique_checks = '1';`

> insert 优化

- 将多次插入使用values(...),(...),(...)
- 关闭自动提交事务，开启事务，批量插入，批量提交
- 保证插入数据的主键有序

> order by优化

- 避免using filesort排序（select不要使用*，不要出现非索引字段），使用Using index排序，filesort是mysql查询到行数据后，将数据拷贝到file_buffer中使用排序算法对数据进行排序
- 避免多个排序字段时排序方式不同，都用升序asc或降序desc
- 多个排序字段尽量和组合索引字段顺序一致
- 尽量在排序字段上设置索引，避免在文件中的排序
- 设置max_length_for_sort_data的大小和sort_buffer_size的大小，优先使用一次扫描

> 优化子查询

- 多表查询的效率高于子查询
- 子查询会形成临时表，需要两个步骤实现查询

> limit 查询优化

- 在索引上进行排序，根据索引排序，然后再与原来的表进行关联查询
- `sql selectv * from tn_user a,(select id from tb_user order by id limit 100000,10) b where a.id = b.id;`
- 将limit的查询转化为自增的主键的范围查询
- `sql select * from tb_user where id >9000000 limit 10;`

15. jdbc 操作mysql

16. sql注入

> 不使用prepareStatement时，使用拼接的方式改变原来sql的逻辑，导致与原来sql不同的意图

```
select * from user
where username='aaa' and password ='bbb' or 1=1 --拼接的字符串
```

> 使用PrepareStatement,使用占位符的形式进行参数的填充，对sql进行预编译

MVCC- Multi-Version Concurrency Control 多版本并发控制

- 事务版本号 事务每次开启前，都会从数据库获得一个自增长的事务ID，可以从事务ID判断事务的执行先后顺序。这就是事务版本号。

- 隐式字段 对于InnoDB存储引擎，每一行记录都有两个隐藏列trx_id、roll_pointer，如果表中没有主键和非NULL唯一键时，则还会有第三个隐藏的主键列row_id。
- undo log undo log，回滚日志，用于记录数据被修改前的信息。在表记录修改之前，会先把数据拷贝到undo log里，如果事务回滚，即可以通过undo log来还原数据。 可以这样认为，当delete一条记录时，undo log 中会记录一条对应的insert记录，当update一条记录时，它记录一条对应相反的update记录。 作用: 事务回滚时，保证原子性和一致性。 用于MVCC快照读。
- 版本链 多个事务并行操作某一行数据时，不同事务对该行数据的修改会产生多个版本，然后通过回滚指针（roll_pointer），连成一个链表，这个链表就称为版本链。
- 快照读和当前读 快照读： 读取的是记录数据的可见版本（有旧的版本）。不加锁,普通的select语句都是快照读 当前读：读取的是记录数据的最新版本，显式加锁的都是当前读，如：

```
select * from core_user where id > 2 for update;
select * from account where id>2 lock in share mode;
```

- Read View 它就是事务执行SQL语句时，产生的读视图。实际上在innodb中，每个SQL语句执行前都会得到一个Read View。 它主要是用来做可见性判断的，即判断当前事务可见哪个版本的数据

MVCC查询流程：通过 Read View+ Undo Log 实现，Undo Log 保存了历史快照，Read View可见性规则帮助判断当前版本的数据是否可见。

> 获取事务自己的版本号，即事务ID 获取Read View 查询得到的数据，然后Read View中的事务版本号进行比较。 如果不符合Read View的可见性规则， 即就需要Undo log中历史快照; 最后返回符合规则的数据 在读已提交（RC）隔离级别下，同一个事务里面，每一次查询都会产生一个新的Read View副本，这样就可能造成同一个事务里前后读取数据可能不一致的问题（不可重复读并发问题）。 在可重复读（RR）隔离级别下，一个事务里只会获取一次read view，都是副本共用的，从而保证每次查询的数据都是一样的。 mysql可重复读的的隔离级别下并不能完全解决幻读

mvvc参考文档

mysql left join的优化方案(回答来自于chatgpt):

1. 选择合适的字段： 在 SELECT 语句中，只选择需要的字段，避免选择过多无用的字段，减少数据传输的开销。
2. 索引： 确保参与 LEFT JOIN 的字段和 WHERE 子句中的字段都有适当的索引。索引可以加快数据匹配和过滤，提高查询性能。
3. 避免大表驱动小表： 当一个表非常大，而另一个表相对较小时，优化器可能会选择从大表中获取数据并驱动小表的匹配。可以通过重写查询，或者使用适当的提示来指导优化器选择更合适的执行计划。
4. 使用子查询替代 JOIN： 在某些情况下，使用子查询代替 LEFT JOIN 可能会提高性能。优化器可能会更好地处理子查询，从而生成更高效的执行计划。
5. LIMIT 结果集大小： 如果可能的话，在 LEFT JOIN 操作之后使用 LIMIT 限制返回的结果集大小。这可以减少数据的传输和处理开销。
6. 分步执行： 如果查询复杂，可以考虑分步执行，先获取左表的部分数据(使用where进行过滤)，然后再执行 LEFT JOIN。这可能会减少内存和计算开销。
7. 使用 EXPLAIN 分析执行计划： 使用数据库的 EXPLAIN 命令分析查询的执行计划，了解优化器选择的路径，并根据分析结果进行调整。
8. 考虑数据量和频率： 考虑实际数据量和查询的频率，优化查询时要根据具体情况进行权衡和决策。
9. 定期维护： 定期进行数据库的维护，包括重新生成索引、收集统计信息等，以保持查询性能。

10. 使用适当的数据库引擎： 根据数据量和查询需求，选择合适的数据库引擎，如 InnoDB、MyISAM 等。
11. 在join的两张表中如果能进行where条件筛选，则首先应该进行条件筛选

left join为什么要避免大表驱动小表

大表驱动小表可能存在以下问题：

1. 内存和磁盘开销： 当大表驱动小表时，数据库系统需要将整个大表的数据加载到内存中，这可能会导致内存不足或缓存失效，从而增加了磁盘访问的开销，影响性能。
2. 数据传输延迟： 由于大表需要从磁盘读取数据，可能会导致数据传输延迟，从而增加了查询的响应时间。
3. 性能下降： 大表驱动小表可能导致查询性能下降，因为数据库系统需要处理更多的数据，执行时间会变长。
4. 查询优化困难： 大表驱动小表会使查询优化器选择合适的执行计划变得更加困难，因为优化器需要考虑如何在处理大表的情况下尽量优化小表的匹配。

避免大表驱动小表可采取以下措施：

1. 重写查询： 可以尝试重写查询，改变表的连接顺序，使小表驱动大表。这可能需要对查询进行优化，以便更好地利用索引。
2. 使用子查询： 可以使用子查询代替 LEFT JOIN，以避免大表驱动小表。子查询可能会更有效地处理数据，从而提高查询性能。
3. 添加索引： 确保参与连接操作的字段和筛选条件字段都有适当的索引。这有助于优化查询执行计划，减少大表的访问开销。
4. 分步执行： 如果可能，可以将查询拆分为多个步骤，先获取小表的部分数据，然后再执行 LEFT JOIN。这可以减少内存和磁盘开销。

sql优化的建议

选择合适的字段： 在 SELECT 查询中，只选择需要的字段，避免选择过多不必要的字段，减少数据传输开销。 使用索引： 确保查询中涉及的字段都有适当的索引，以加速数据检索。但不要过度索引，因为索引也会增加插入、更新和删除操作的开销。 避免使用通配符：%： 在使用 LIKE 子句进行模糊匹配时，避免在通配符前使用 %，这会导致无法使用索引。 避免使用 SELECT *： 尽量避免使用 SELECT* 查询所有字段，明确列出所需字段。 根据需求合理使用 JOIN或子查询： 使用 INNER JOIN、LEFT JOIN、RIGHT JOIN 等连接操作时，确保连接字段上有索引，并避免在大表上执行 JOIN 操作。 使用 EXISTS 替代 IN 和 NOT IN： 在子查询中，使用 EXISTS 条件来代替 IN 和 NOT IN，它通常更高效。 避免子查询： 尽量避免使用嵌套子查询，可以尝试使用 JOIN 操作来达到相同的效果。 分页优化： 在分页查询中，使用 LIMIT 和 OFFSET，同时确保 OFFSET 值不过大，以免导致性能问题。 使用 UNION 替代 OR： 当需要多个条件时，使用 UNION 连接多个查询，而不是使用 OR 条件。 使用 UNION ALL： 如果查询需要返回重复记录，请使用 UNION ALL 而不是 UNION，因为 UNION ALL 不进行去重操作，性能更高。 选择合适的数据类型： 选择最适合数据的数据类型，不要使用过大或过小的数据类型。 定期维护： 定期进行数据库的维护，包括重建索引、收集统计信息等。 避免 ORDER BY RAND()： 在排序时避免使用 ORDER BY RAND()，因为它会导致性能问题。可以使用其他方法来实现随机排序。 批量操作： 尽量使用批量插入、更新和删除操作，减少单次操作的开销。 查询计划分析： 使用 EXPLAIN 或类似工具分析查询计划，了解查询的执行计划，找到潜在的性能问题。 分区表： 对于大表，可以考虑使用分区表来提高查询性能。 缓存： 使用缓存技术，如缓存查询结果或使用缓存数据库，以减轻数据库负担。 服务器优化： 配置数据库服务器参数，如内存分配、线程数等，以适应负载。 分析和监控： 定期分析数据库性能，使用监控工具来追踪数据库的性能瓶颈。 编写优化的存储过程： 在适当的情况下，使用存储过程来封装一系列操作，减少网络开销。

大表的limit优化

limit是通过从表的第一行开始遍历到指定位置的方法来实现寻找分页数据的，这样在数据量小的时候是非常有效的，但是碰到数据量非常大的情况时，会遇到效率不佳的问题 select * from table_name where name = 'aaa' limit 1000000,20 这种语句，由于limit在执行时会扫描1000000条数据才能得到结果，所以数值越大，耗时越长可以改成如下的写法 select * from table_name where name ='aaa' and id > 1000000 limit 20

面试题

工作中具体的sql优化的例子

1. 关于limit 1000000 10的优化，使用where id > 1000000 limit 10这种写法避免全表扫描
2. 关于left join的优化： 尽量避免使用left join ，分步骤查询，或者使用子查询，查询字段，尽量要使用

有一个联合索引中有三个字段 a,b,c，下面那个查询速度快 select * from t1 where a =1 and c = 1; select * from t1 where b = 1 and a = 1; 第二条sql快，按照索引的最左匹配原则，虽然第二条sql使用b在前a在后，但是查询时时可以按照最左匹配原则走联合索引

字节面试题： 有一张用户表user，大概有100万行数据，age >8的数据大概有10万行， 有name和age两个字段，在age上做了索引，下面的sql要扫描多少行数据 select name，age from user where age > 8 and name = "zhangsan" limit 50; 答案：大概会扫描10万行数据，limit条件不会影响扫描的行数，where子句中使用了索引列age使用不等值判断会扫描age > 8的所有数据

innodb事务原理

redo log 和undo log： 保证事务的原子性、一致性、持久性 锁和mvcc：保证事务的隔离性

redo log 重做日志 （数据的持久性）

分为redo log buffer 重做日志缓冲和redo log file（重做日志文件），前者在内存中，后者在磁盘中 记录物理日志，记录数据的内容 当事务提交之后把所有的修改信息都给存放到该日志文件中，用于在刷新脏页到磁盘发生错误时，进行数据的恢复

undo log 回滚日志 （事务的原子性和一致性）

记录数据修改前的信息，提供回滚和MVCC 记录逻辑日志，记录insert、update、delete等操作的反操作 当执行rollback操作时，就可以从undo log中的逻辑记录读取到相应的内容并进行回滚 undo log销毁：在事务执行时产生，事务提交后不会立即删除，日志还可用于MVCC undo log存储，采用段的方式管理和记录，insert时undo log日志只在回滚时删除，事务提交后可以被立即删除 update、delete 产生的undo log日志不仅回滚时需要，在快照读时也需要，不会立即被删除

gpt Undo Log（回滚日志）： Undo Log用于实现事务的原子性和一致性，以及提供事务的回滚功能。在事务执行期间，所有被修改的数据都会在Undo Log中记录相应的修改前的数据，即事务的"撤销操作"。如果事务回滚，系统会利用Undo Log中的信息将数据恢复到事务开始前的状态。同时，Undo Log还能够支持多版本并发控制（MVCC），使得多个事务能够并发地访问数据库而不互相干扰。

Redo Log（重做日志）： Redo Log用于实现事务的持久性，确保在数据库崩溃等情况下数据的一致性。在事务提交时，修改的数据会首先写入到Redo Log中，然后再更新到数据库文件中。如果数据库发生崩溃，系统可以利用Redo Log中的信息重新应用事务的修改，从而恢复数据库到最近一次崩溃点的状态。这种机制被称为“重做”。

概念

当前读：读取的记录时最新的版本，读取时保证其他事务不能修改当前记录 select * from user lock in share mode 可以保证读取到最新的数据（默认隔离级别repeatable read） select ...for update 、update、insert、delete 都是一种当前读

快照读： 简单的select（不加锁）就是快照读，快照读读的是记录数据的课件版本，有可能是历史数据，不加锁，是非阻塞读 read commited：每次select都生成一个快照读 repeatable read：开启事务后的第一个select语句才是快照读的地方 serializable 快照读会退化为当前读

MVCC

维护一个数据的多个版本，使得读写操作没有冲突，快照读为MVCC提供了一个非阻塞读的功能 MVCC不需要加锁 MVCC提供了一个非阻塞读的功能，MVCC的具体实现需要依赖于数据库记录中的三个隐式字段、undo log、readView 隐式字段 DB_TRX_ID:最近修改事务id，记录插入这条记录或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，配合undo log指向上一个版本 DB_ROW_ID：隐藏主键，表结构中没有指定主键，将会生成该隐式字段

readView

是快照读SQL执行时MVCC提取数据的依据，记录并维护系统当前活跃的事务（未提交的）id 根据当前事务id于四个字段值的比较判断是否生成read 四个字段： m_ids:当前活跃事务ID集合 min_trx_id：最小活跃事务ID ma_trx_id：预分配事务id，当前最大事务ID+1 creator_trx_id: readView创建者的事务ID

read committed下事务每次执行快照读生成一个readView （每次读都会有不同的readview，可能出现不可重复读） repeatable read下仅在事务中第一次执行快照读时生成readview，后续复用该readview（保证可重复读）

mysql如何做分库分表

分库分表的原因： 大表会影响查询性能，其次数据增删改的性能也收到影响，如果是使用的mysql主从复制做读写隔离，则会导致主从延迟过高，比如订单系统，用户下单后主库数据已经写入，从库还没来得及更新，这时订单已经生效，但是用户再次查询订单时可能从库里还没有数据，会导致用户重复下单的情况。

分表方案：

垂直分表：将使用频率高的字段放在一张表中，剩下的使用频率低的字段放在另一个张表里

垂直划分之后，以前的查询表的sql需要从两张表中来获取数据，如果使用了数据库中间件，中间件会自动实现重写查询逻辑，不使用中间件需要自己来改写sql 原sql

```
select order_id,order_sn ,source from orders where order_id = '001';
```

拆分之后

```
select a.order_id,a.order_sn,b.source from orders01 a,orders02 b where
a.id=b.id and a.order_id = '001';
```

另外，对于text这种大字段类型，可以将text类型拆分到子表中，

水平分表：按照表中的记录进行分片，单表不建议超过500w

对于特定的表，分表策略也不一样 根据表字段取模，范围、hash 对于财务或计费类的系统，可以按月来分表

分库分表应该注意的问题：

1. 避免出现分库之后容量太满的情况
2. 避免单表数据量过的问题
3. 分表的字段要选择合理，避免查询过于复杂

分表策略：

MySQL分表策略是在处理大量数据时的一种常用技术，旨在提高数据库性能和可维护性。以下是几种常见的MySQL分表策略：

按范围分表： 将数据根据某个范围值分散到不同的表中。例如，可以根据时间范围将数据分布到不同的月份表中。

按哈希分表： 将数据的哈希值作为依据，将数据分散到不同的表中。这有助于均匀分布数据，但可能会导致难以预测的查询性能。

按模数分表： 将数据的ID或其他唯一标识符通过取模运算，将数据分布到一系列表中。这对于预测性能和查询分布很有帮助。

按业务分表： 根据业务逻辑将数据分散到不同的表中。例如，可以根据地理位置或产品类型进行分表。

垂直分表： 将表的列拆分为不同的表，以避免数据冗余并提高查询效率。这常用于将经常访问的列与不经常访问的列分开。

水平分表： 将表的行按照某个标准分布到不同的表中。例如，可以根据用户ID的范围来分表。

分区表： MySQL支持分区表，允许将表按照预定义的条件划分为多个分区，每个分区实际上可以存储在不同的磁盘上，从而提高查询效率。

在同时写入MySQL和Redis时，可以使用以下几种方法来保证数据的一致性：

事务（Transaction）： 使用数据库的事务机制来确保数据的一致性。在写入MySQL和Redis之前，开启一个事务，然后将写入操作放入同一个事务中，最后提交事务。如果写入MySQL或Redis的任何一个操作失败，可以回滚整个事务，确保数据的一致性。 双写模式（Dual-write）： 在写入MySQL和Redis之前，先写入MySQL，然后再写入Redis。如果写入MySQL成功但写入Redis失败，可以通过定期或异步的方式进行数据修复，将MySQL中的数据同步到Redis中，保持数据的一致性。 消息队列（Message Queue）： 将写入MySQL和Redis的操作放入消息队列中，消费者从消息队列中读取消息并执行写入操作。这样可以确保写入MySQL和Redis的顺序一致，并且可以通过重试机制来保证写入操作的可靠性。 使用分布式事务（Distributed Transaction）： 如果MySQL和Redis分别部署在不同的节点上，可以使用分布式事务来保证数据的一致性。一种常见的方式是使用两阶段提交（Two-Phase Commit）协议来确保所有参与节点的数据操作都成功或都

失败。 以上是一些常用的方法，具体选择哪种方法取决于系统的需求和实际情况。需要权衡数据一致性、性能、可靠性等因素，选择适合的方案。

mysql datetime 和timestamp的区别

相同点： 两个数据类型存储时间的格式一致。均为 YYYY-MM-DD HH:MM:SS 两个数据类型都包含「日期」和「时间」部分。 两个数据类型都可以存储微秒的小数秒（秒后6位小数秒）

建表时自动设置和更新时间戳： CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP

1. 占用空间 datetime 占用8个字节 yyyy-mm-dd hh:mm:ss timestamp 占用4个字节 yyyy-mm-dd hh:mm:ss
2. 表示范围 datetime '1000-01-01 00:00:00.000000' to '9999-12-31 23:59:59.999999'

timestamp '1970-01-01 00:00:01.000000' to '2038-01-19 03:14:07.999999'

timestamp翻译为汉语即"时间戳"，它是当前时间到 Unix元年(1970 年 1 月 1 日 0 时 0 分 0 秒)的秒数。对于某些时间的计算，如果是以 datetime 的形式会比较困难，假如我是 1994-1-20 06:06:06 出生，现在的时间是 2016-10-1 20:04:50 ，那么要计算我活了多少秒钟用 datetime 还需要函数进行转换，但是 timestamp 直接相减就行。

3. 时区 timestamp 只占 4 个字节，而且是以utc的格式储存， 它会自动检索当前时区并进行转换。datetime以 8 个字节储存，不会进行时区的检索. 如果存进去的是NULL，timestamp会自动储存当前时间（now()），而 datetime会储存 NULL。

4. 使用上的选择 如果想计算时间的变化，使用timestamp 例如飞机从北京起飞时间为北京时间 2021-10-10 11:05:00，到达纽约时纽约时间为2021-10-10 09:50:00，这种情况下使用timestamp来存储时间，两个时间都会被转化为utc时间，更加方便计算 如果只是记录信息的创建时间、修改时间，不涉及到加减转换，使用datetime更直观

数据库表设计的一些经验规则

- 采取领域模型驱动的方式和自顶向下的思路进行数据库设计，首先分析系统业务，根据职责定义对象，设计对应的逻辑表
- 合理的数据结构：确保数据库表的字段和数据类型合理，避免存储冗余信息，合理选择字段类型和字段长度
- 合理设计主键：选择唯一标识每条记录的主键，通常以自增 ID或具有唯一性的字段（或组合字段）
- 根据查询热点合理设计索引，提高查询性能
- 性能优化：定期进行性能优化，包括索引重建、查询优化和定期清理无用数据
- 针对所有表的主键和外间建立索引
- 尽量少使用存储过程
- 遵循数据库设计三范式： 第一范式： 数据库表中的所有字段值都是不可分解的原子值，确保每一列都保持原子性，不可再拆分 第二范式： 在第一范式的基础上，需要确保数据库表中的每一列都和主键相关，而不能只与主键的某一部分相关，确保表中的每一列都和主键相关 第三范式： 确保数据表中的每一列数据都和主键直接相关，而不能间接相关

设计数据库应该注意以下几点：

1. 遵循数据库的三范式，减少数据冗余并确保数据一致性

2. 设计合适的主键，尽量不使用外键（性能问题），在代码中维护外键，维护数据完整性，主键尽量避免业务相关，一般采用自增的int类型（innodb中的主键时聚集索引，使用int类型的数据能提升查询效率）
3. 对于业务字段选择合理的数据类型，避免设置null值保证数据完整性，同时避免在查询时对null值的额外处理
4. 定长字符字段使用char类型，非定长字符字段用varchar类型，对于需要进行时间计算的字段选timestamp类型，不需要时间计算的字段选datetime类型
5. 默认为表添加新增时间create_time、update_time两个字段进行业务的排查
6. 表、字段、索引命名尽量简单易懂，让人一看就知道代表什么业务含义
7. 设计表时表的字段不宜过多，对于过多的字段应考虑拆分子表
8. 对于一些可能存在删除的业务的表，优先考虑设计逻辑删除字段，避免数据恢复的困难以及物理删除带来的主键不连续的问题
9. 根据业务查询条件合理设置索引，提高数据查询效率
10. 对于货币字段，一般使用decimal字段来保证精度
11. 对于图片、视频、音频不应该直接存储在数据库中，应该将文件存储在hdfs、文件服务器上，在mysql对应的字段中保存文件路径或是文件的链接地址
12. 对于经常需要进行join的字段，可以考虑在表中进行冗余，减少join查询
13. 使用mysql的主从复制、读写分离来保证数据库的安全和性能

对于索引设计应该考虑：

1. 对于一张表，索引的个数不应该操作五个，索引过多会降低写入速度
2. 区分度不高的字段，不应该添加索引，比如性别、业务状态等字段
3. 能使用联合字段索引尽量使用，因为联合字段索引可以包含单字段的索引
4. 索引应该建在小字段上

查询大批量数据时，避免全表扫描的方法：

1. 首先考虑在where或者order by的列上建立索引
2. 尽量避免where子句中的null判断
3. 避免like 查询时前%的逻辑匹配
4. 使用or连接查询条件时查询字段必须全有索引
5. 联合字段索引遵循最左匹配原则，谨慎使用非等值匹配的情况
6. 避免在索引列上进行运算
7. 避免索引列查询时的字段类型不匹配
8. 使用in查询索引列时，要控制in匹配的数量，如果in匹配的数量过多会导致in查询放弃索引

```
explain
select * from data_main dm2  where dm2.word_type in
(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21);
```

这条语句会导致扫描全表

事务失效可能的原因

1. 在service中方法A调用service自身的标注了@Transcational注解的方法会导致事务失效
2. @Transactional注解的方法是private或final类型的，会导致事务失效

3. 可能由于在spring中选择的事务传播机制不当导致的事务失效

spring中编程式事务的使用

1. 使用TransactionTemplate方法

```
    @Autowired
    private TransactionTemplate transactionTemplate;

    public Object test(){
        transactionTemplate.execute(new TransactionCallback<Object>() {
            @Override
            public Object doInTransaction(TransactionStatus status) {
                try{
                    doSomething();

                }catch (Exception e){
                    status.setRollbackOnly();
                }
                return null;
            }
        });
        return null;
    }
    public void doSomething(){
        System.out.println("hello world");
    }
```

2. 直接使用PlatformTransactionManager来获取事务并开启事务，执行业务逻辑，然后提交或回滚事务

```
        //定义一个JdbcTemplate，用来方便执行数据库增删改查
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        //1.定义事务管理器，给其指定一个数据源（可以把事务管理器想象为一个人，这个人来
负责事务的控制操作）
        PlatformTransactionManager platformTransactionManager = new
DataSourceTransactionManager(dataSource);
        //2.定义事务属性：TransactionDefinition，
        // TransactionDefinition可以用来配置事务的属性信息，比如事务隔离级别、事务
超时时间、事务传播方式、是否是只读事务等等。
        TransactionDefinition transactionDefinition = new
DefaultTransactionDefinition();
        //3.开启事务：调用platformTransactionManager.getTransaction开启事务操
作，得到事务状态(TransactionStatus)对象
        TransactionStatus transactionStatus =
platformTransactionManager.getTransaction(transactionDefinition);
        //4.执行业务操作，下面就执行2个插入操作
        try {
            System.out.println("before:" +
jdbcTemplate.queryForList("SELECT * from t_user"));
            jdbcTemplate.update("insert into t_user (name) values (?)",
```

```
"test1-1");
            jdbcTemplate.update("insert into t_user (name) values (?)",
"test1-2");
            //5.提交事务: platformTransactionManager.commit
            platformTransactionManager.commit(transactionStatus);
        } catch (Exception e) {
            //6.回滚事务: platformTransactionManager.rollback
            platformTransactionManager.rollback(transactionStatus);
        }
        System.out.println("after:" + jdbcTemplate.queryForList("SELECT *
from t_user"));
```

防止sql注入的方法:

- 使用预处理语句和参数化查询 （prepared statements），这是目前最有效的防止sql注入的方法
- 使用存储过程，限制用户输入类型，可能一定程度避免sql注入
- 对页面输入参数进行验证
- 使用ORM框架，避免使用sql拼接
- 限制数据库账号权限
- 使用web应用防火墙（WAF）
- 避免显示数据库错误信息
- 定期更新数据库管理系统
- 使用类型安全的sql接口
- 进行安全审计和代码审查