

MySQL面试热点与MySQL高级特性,性能优化

讲师：徐磊

## 课程介绍

MySQL是使用最为广泛的开源数据库系统，是后端开发工程师，架构师，运维工程师，DBA，面试中几乎必定被问到的内容。本次课程以面试真题为起点,结合MySQL必问的热点技能为方向，覆盖MySQL面试相关的方方面面，梳理和总结相关知识点。通过本课程的学习，不仅可以深入了解MySQL数据库，也能助力轻松面试，获得心仪的工作机会。

## 主要内容目标

- MySQL常见基础面试热点
- MySQL语句的执行顺序
- MySQL内外，全连接深入
- 存储引擎的选择和区别
- 存储过程，函数，触发器
- 锁问题深入
- 事物与隔离级别深入
- MySQL数据库优化问题
- MySQL数据库优化方案
- MySQL索引深入
- MySQL索引失效的避免
- MySQL其他性能优化方案
- MySQL常见问题和大厂面试题解决

## 第一章 Mysql数据库基础面试热点

### 目标 掌握数据库范式

什么是范式

创建表的规则，指导我们后期如何去设计自己的表。

数据库表规范化的好处：

- 减少数据的冗余。
- 减少后期java代码的工作量。

三大范式小结

范式	特点
第一范式	原子性每列不可再拆分
第二范式	不产生局部依赖，每列都完全依赖于主键，一张表只描述一件事情
第三范式	不产生传递依赖，所有的列都直接依赖于主键,使用外键关联，外键都来源与其他表的主键

## 反三范式

反3NF：为了提高数据的性能，增加冗余字段，以便提高查询性能

## 目标 Select语句执行顺序

了解SQL Select语句完整的执行顺序，有利于熟练掌握SQL的编写，Select语句的执行顺序也是开发中常考的内容。

[SQL语言](#)不同于其他编程语言的最明显特征是处理代码的顺序。在大多数数据库语言中，代码按编码顺序被处理。但在SQL语句中，第一个被处理的是FROM，而不是第一出现的SELECT。

代码编写顺序

- select distinct 查询字段
- from 表名
- JOIN 表名
- ON 连接条件
- where 查询条件
- group by 分组字段
- having 分组后条件
- order by 排序条件
- limit 查询起始位置, 查询条数

```
SELECT DISTINCT <select_list>
FROM <left_table>
<join_type> JOIN <right_table>
ON <join_condition>
WHERE <where_condition>
GROUP BY <group_by_list>
HAVING <having_condition>
ORDER BY <order_by_condition>
LIMIT <limit_number>
```

测试脚本：

```
CREATE TABLE tb_customer
(
    customer_id VARCHAR(10) NOT NULL,
    city VARCHAR(10) NOT NULL,
    PRIMARY KEY(customer_id)
)ENGINE=INNODB DEFAULT CHARSET=UTF8;

CREATE TABLE tb_order(
order_id INT PRIMARY KEY AUTO_INCREMENT,
customer_id VARCHAR(23)
)ENGINE=INNODB DEFAULT CHARSET=UTF8;;

INSERT INTO tb_customer(customer_id,city) VALUES('163','hangzhou');
INSERT INTO tb_customer(customer_id,city) VALUES('9you','shanghai');
INSERT INTO tb_customer(customer_id,city) VALUES('tx','hangzhou');
INSERT INTO tb_customer(customer_id,city) VALUES('baidu','hangzhou');

INSERT INTO tb_order(customer_id) VALUES('163');
```

```
INSERT INTO tb_order(customer_id) VALUES('163');
INSERT INTO tb_order(customer_id) VALUES('9you');
INSERT INTO tb_order(customer_id) VALUES('9you');
INSERT INTO tb_order(customer_id) VALUES('9you');
INSERT INTO tb_order(customer_id) VALUES('tx');
INSERT INTO tb_order(customer_id) VALUES(NULL);
```

查询来自杭州，并且订单数少于2的客户。

```
SELECT a.customer_id, COUNT(b.order_id) AS total_orders
FROM tb_customer a LEFT JOIN tb_order b ON a.customer_id = b.customer_id
WHERE a.city = 'hangzhou'
GROUP BY a.customer_id
HAVING COUNT(b.order_id) < 2
ORDER BY total_orders DESC;
```

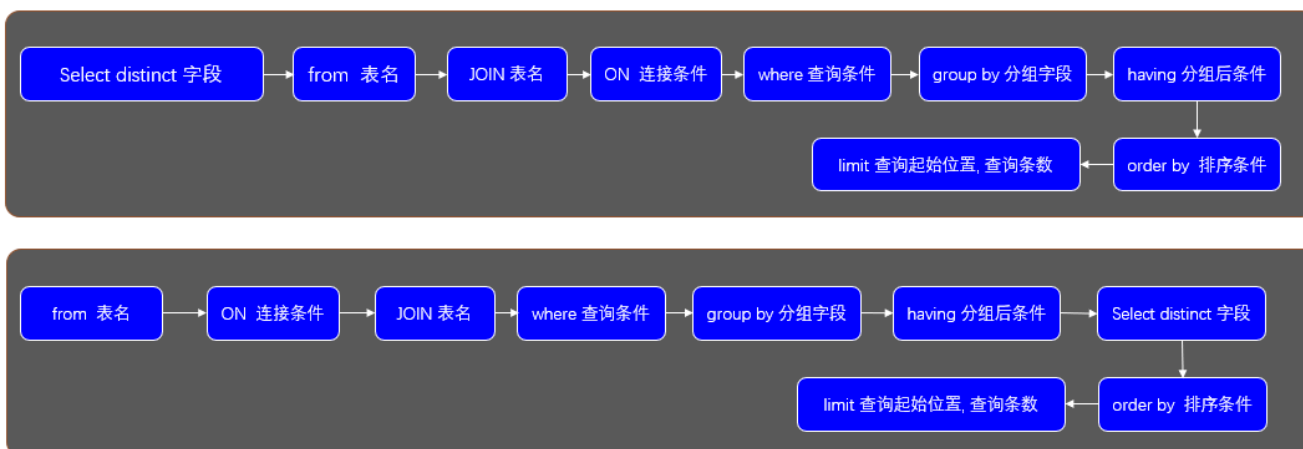
### Mysql读取顺序

- from 表名
- ON 连接条件
- JOIN 表名
- where 查询条件
- group by 分组字段
- having 分组后条件
- select distinct 查询字段
- order by 排序条件
- limit 查询起始位置, 查询条数

### 整体过程

1. 先对多表进行关系, 根据条件找出符合条件的记录
2. 在符合条件的基础上进行再次where条件筛选
3. 对筛选出来的内容进行分组操作
4. 分组完成后, 使用having再次筛选出满足条件的记录
5. 取所满足条件的记录
6. 对取出的记录进行排序
7. 最终从取出的记录当中获取多少条记录显示出来

### 示例图



## 目标 内连接的介绍

内连接：内连接可整合多张表，获取相关关系或者公共部分的记录

### 数据准备

```
# 创建部门表
create table dept(
    id int primary key auto_increment,
    name varchar(20)
);

insert into dept (name) values ('开发部'),('市场部'),('财务部');

# 创建员工表
create table employee (
    id int primary key auto_increment,
    name varchar(10),
    gender char(1), -- 性别
    salary double, -- 工资
    join_date date, -- 入职日期
    dept_id int,
    foreign key (dept_id) references dept(id) -- 外键, 关联部门表(部门表的主键)
);

insert into employee(name,gender,salary,join_date,dept_id) values('孙悟空','男',7200,'2013-02-24',1);
insert into employee(name,gender,salary,join_date,dept_id) values('猪八戒','男',3600,'2010-12-02',2);
insert into employee(name,gender,salary,join_date,dept_id) values('唐僧','男',9000,'2008-08-08',2);
insert into employee(name,gender,salary,join_date,dept_id) values('白骨精','女',5000,'2015-10-07',3);
insert into employee(name,gender,salary,join_date,dept_id) values('蜘蛛精','女',4500,'2011-03-14',1);
```

### 笛卡尔积的介绍

```
SELECT 列名 FROM 左表,右表
```

内连接分为：隐式内连接，显式内连接

- 隐式内连接

```
select 列名 from 左表,右表 where 主表.主键=从表.外键
```

- 显式内连接

```
--使用INNER JOIN ... ON语句，可以省略INNER
select 列名 from 左表 inner join 右表 on 主表.主键=从表.外键
--显式内连接与隐式内连接查询结果是一样的**
select * from dept d inner join emp e on d.id = e.dept_id;
```

## 目标 左外连接的介绍

外连接：外连接分为两种，一种是左连接（Left JOIN）和右连接（Right JOIN）

- 左外连接：在内连接的基础上，保证左表中所有的记录都出现。相应记录使用NULL和它匹配。
- 左外连接：使用LEFT OUTER JOIN ... ON，OUTER可以省略

```
-- 需求：在部门表中增加一个销售部，需要查询所有的部门和员工，将部门表设置成左表，员工表设置成右表
select * from dept;
insert into dept values(null,'销售部');
-- 使用内连接查询
select * from employee e dept d inner join on d.id = e.dept_id;
-- 使用左外连接查询
select * from dept d left join employee e on d.id = e.dept_id;
```

## 目标 右外连接的介绍

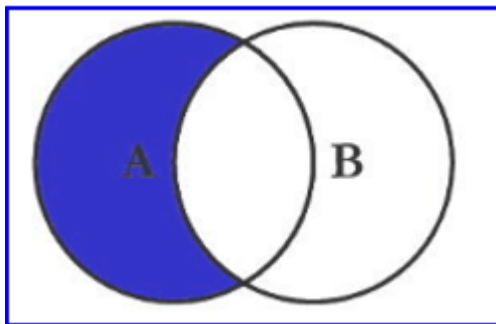
右外连接概念：在内连接的基础上，保证右表中所有的数据都显示。左表中如果没有匹配的数据，使用NULL匹配。

- 右连接的语法：
- 右外连接：使用RIGHT OUTER JOIN ... ON，OUTER可以省略
- 右连接的案例：

```
-- 需求：在员工表中增加一个员工：'沙僧','男',6666,'2013-02-24',null
select * from emp;
insert into emp values(null,'沙僧','男',6666,'2013-02-24',null);
-- 希望员工的信息全部显示出来
-- 使用内连接查询
select * from dept d inner join emp e on d.id = e.dept_id;
-- 使用右外连接查询
select * from dept d right join emp e on d.id = e.dept_id;
```

## 目标 查询左边独有的数据

查询左表独有数据



作用 查询A的独有数据 语句

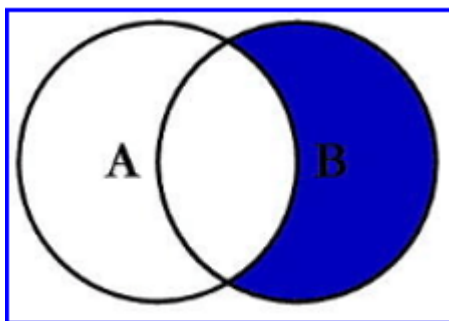
```
select <select_list> from tableA A Left Join tableB B on A.Key = B.Key where B.key IS NULL
```

示例

```
SELECT * from employee e LEFT JOIN department d on e.depart_id = d.id WHERE d.id IS NULL;
```

目标 查询右表独有的数据

查询右表独有数据



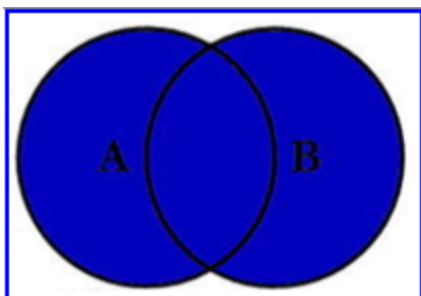
作用 查询B的独有数据 语句

```
select <select_list> from tableA A Right Join tableB B on A.Key = B.Key where A.key IS NULL
```

示例

```
SELECT * from employee e RIGHT JOIN department d on e.depart_id = d.id WHERE e.id IS NULL;
```

目标 全连接介绍



作用

查询两个表的全部信息

语句

```
Select <select_list> from tableA A Full Outer Join tableB B on A.Key = B.Key
```

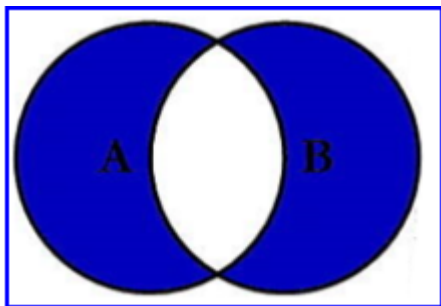
注:MySQL 默认不支持此种写法 Oracle支持

示例

```
SELECT * from employee e LEFT JOIN department d on e.depart_id = d.id
UNION
SELECT * from employee e RIGHT JOIN department d on e.depart_id = d.id
```

## 目标 查询左右表各独有的数据

图示



作用

查询A和B各自的独有的数据

语句

```
Select <select_list> from tableA A Full Outer Join tableB B on A.Key = B.Key where
A.key = null or B.key=null
```

示例

```
SELECT * from employee e LEFT JOIN department d on e.depart_id = d.id WHERE d.id is NULL
UNION
SELECT * from employee e RIGHT JOIN department d on e.depart_id = d.id WHERE e.depart_id
is NULL
```

## 目标 表的级联操作

### 什么是级联操作

在修改和删除主表的主键时，同时更新或删除副表的外键值，称为级联操作 `ON UPDATE CASCADE` -- 级联更新，主键发生更新时，外键也会更新 `ON DELETE CASCADE` -- 级联删除，主键发生删除时，从表关联的全部数据都会被直接删除。

具体操作：

- 删除employee表
- 重新创建employee表，添加级联更新和级联删除

```
CREATE TABLE employee (
    id INT PRIMARY KEY AUTO_INCREMENT,
    NAME VARCHAR(30),
    age INT,
    dep_id INT,
    -- 添加外键约束,并且添加级联更新和级联删除
    CONSTRAINT employee_dep_fk FOREIGN KEY (dep_id) REFERENCES department(id) ON UPDATE
    CASCADE ON DELETE CASCADE
);
```

- 再次添加数据到员工表和部门表

```
INSERT INTO employee (NAME, age, dep_id) VALUES ('张三', 20, 1);
INSERT INTO employee (NAME, age, dep_id) VALUES ('李四', 21, 1);
INSERT INTO employee (NAME, age, dep_id) VALUES ('王五', 20, 1);
INSERT INTO employee (NAME, age, dep_id) VALUES ('老王', 20, 2);
INSERT INTO employee (NAME, age, dep_id) VALUES ('大王', 22, 2);
INSERT INTO employee (NAME, age, dep_id) VALUES ('小王', 18, 2);
```

## 第二章 存储引擎的选择

### 目标 存储引擎的概念和查看

#### 存储引擎的概念

插件式存储引擎是Mysql中最重要的特性之一，用户可以根据应用的需要选择如何存储数据和索引，是否使用事物等，从而改善你的应用的整体功能。这些不同的技术以及配套的相关功能在MySQL中被称作存储引擎,MySQL默认支持多种存储引擎，以适应不同领域数据库的需求，用户可以通过选择不同的存储引擎，提高应用的效率，提供灵活的存储，用户也可以按照自己的需求定制和使用存储引擎。

#### 查看存储引擎

```
show engines;
```

Engine	Support	Comment	Transactions	XA	Savepoints
FEDERATED	NO	Federated MySQL storage engine	(NULL)	(NULL)	(NULL)
MRG_MYISAM	YES	Collection of identical MyISAM tables	NO	NO	NO
MyISAM	YES	MyISAM storage engine	NO	NO	NO
BLACKHOLE	YES	/dev/null storage engine (anything you write to it disappears)	NO	NO	NO
CSV	YES	CSV storage engine	NO	NO	NO
MEMORY	YES	Hash based, stored in memory, useful for temporary tables	NO	NO	NO
ARCHIVE	YES	Archive storage engine	NO	NO	NO
InnoDB	DEFAULT	Supports transactions, row-level locking, and foreign keys	YES	YES	YES
PERFORMANCE_SCHEMA	YES	Performance Schema	NO	NO	NO

#### 查看当前存储引擎

- 查看系统当前支持的存储引擎，需要使用如下命令：

```
show variables like '%storage_engine%';
```

效果如下：



1 结果		2 信息	3 表数据	4 信息
		(只读)		
<input type="checkbox"/>	Variable_name	Value		
<input type="checkbox"/>	default_storage_engine	InnoDB		
<input type="checkbox"/>	storage_engine	InnoDB		

可以看到：

- 当前MySQL默认的存储引擎是InnoDB
- 当前MySQL正在使用的存储引擎也是InnoDB

## 目标 存储引擎的创建和修改

### 引入

创建新表时，如果不指定存储引擎，那么系统就会使用默认的存储引擎，MySQL 5.5之前的默认存储引擎是MyISAM，5.5之后改为了InnoDB,如果需要修改存储引擎可以在核心配置文件中配置如下操作

```
default-storage-engine=INNODB
```

也可以在创建表的时候，通过增加ENGINE关键字设置新表的存储引擎，例如下面的例子：

```
CREATE TABLE `test1` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(20) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM default CHARSET=utf8

CREATE TABLE `test2` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(20) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8
```

也可以把一个已经存在的表的存储引擎，修改成其他表的存储引擎，操作如下

```
alter table test1 ENGINE = innodb;
show create table test1
```

## 目标 常见存储引擎的区别

常用存储引擎的对比

特 点	MyISAM	InnoDB	MEMORY	MERGE	NDB
存储限制	有	64TB	有	没有	有
事务安全		支持			
锁机制	表锁	行锁	表锁	表锁	行锁
B 树索引	支持	支持	支持	支持	支持
哈希索引			支持		支持
全文索引	支持				
集群索引		支持			
数据缓存		支持	支持		支持
索引缓存	支持	支持	支持	支持	支持
数据可压缩	支持				
空间使用	低	高	N/A	低	低
内存使用	低	高	中等	低	高
批量插入的速度	高	低	高	高	高
支持外键		支持	<a href="https://blog.csdn.net/qq_36045946">https://blog.csdn.net/qq_36045946</a>		

除了上表列出了，重点介绍下MyISAM和InnoDB区别

- 主外键
  - MyISAM : 不支持
  - InnoDB: 支持
- 事务
  - MyISAM:不支持
  - InnoDB:支持
- 行表锁
  - MyISAM 表锁 操作一条记录也会锁住整个表 不适合高并发的操作
  - InnoDB 行锁 操作时,只锁某一行,不对其它行有影响 适合高并发的操作
- 缓存
  - MyISAM : 只缓存索引,不缓存数据
  - InnoDB:不仅缓存索引,还要缓存真实数据,对内存要求比较高,而且内存大小对性能有决定性的影响
- 关注点
  - MyISAM性能
  - InnoDB:事务
- 默认安装
  - MyISAM:是
  - InnoDB:是

小结

在选择存储引擎时，应根据应用特点选择合适的存储引擎。对于复杂的应用系统，还可以根据实际情况选择多种存储引擎进行组合。

下面是几种常用存储引擎的适用环境。

● **MyISAM**: 默认的 MySQL 插件式存储引擎。如果应用是以读操作和插入操作为主，只有很少的更新和删除操作，并且对事务的完整性、并发性要求不是很高，那么选择这个存储引擎是非常适合的。MyISAM 是在 Web、数据仓储和其他应用环境下最常使用的存储引擎之一。

● **InnoDB**: 用于事务处理应用程序，支持外键。如果应用对事务的完整性有比较高的要求，在并发条件下要求数据的一致性，数据操作除了插入和查询以外，还包括很多的更新、删除操作，那么 InnoDB 存储引擎应该是一个比较合适的选择。InnoDB 存储引擎除了有效地降低由于删除和更新导致的锁定，还可以确保事务的完整提交（Commit）和回滚（Rollback），对于类似计费系统或者财务系统等对数据准确性要求比较高的系统，InnoDB 都是合适的选择。

## 第三章 存储过程，函数，触发器

---

### 目标 存储过程,函数的介绍

#### 引入

MySQL从5.0版本开始支持存储过程和函数。

存储过程和函数是事先经过编译和存储在数据库中的一段SQL语句的集合，然后直接通知调用执行即可，所以调用存储过程和函数可以简化应用开发人员的很多工作，减少数据在数据库和应用服务器之间的传输，对于提高数据处理的效率是有好处的。

**存储过程和函数的区别在于函数必须有返回值**，而存储过程没有，存储过程的参数可以使用**IN，OUT，INOUT**类型，而函数的参数只能是IN类型的，

创建，删除，修改存储过程或者函数都需要权限，例如创建存储过程或者函数需要CREATE ROUTINE权限，修改或者删除存储过程或者函数需要ALTER ROUTINE权限，执行存储过程或者函数需要EXECUTE权限。

#### 小结

- 存储过程和函数是事先经过编译和存储在数据库中的一段SQL语句的集合
- 减少数据在数据库和应用服务器之间的传输，对于提高数据处理的效率是有好处的
- 存储过程的操作要确定是否有权限。

### 目标 一个简单的存储过程

# 02:14:16

本次直播时长



2892

观看人气峰值



72

增加粉丝



1082

获得积分

我知道了

[查看更多数据](#)

测试数据：

```
CREATE TABLE student (  
  id int primary key auto_increment,  
  name varchar(20),  
  age int,  
  sex varchar(5),  
  address varchar(100),  
  math int,  
  english int  
);  
  
INSERT INTO student(NAME,age,sex,address,math,english) VALUES ('马云',55,'男','杭州',66,78),  
( '马化腾',45,'女','深圳',98,87),( '马景涛',55,'男','香港',56,77),( '柳岩',20,'女','湖南',76,65),  
( '柳青',20,'男','湖南',86,NULL),( '刘德华',57,'男','香港',99,99),( '马德',22,'女','香港',99,99),  
( '德玛西亚',18,'男','南京',56,65);  
  
CREATE TABLE users(  
  id INT PRIMARY KEY AUTO_INCREMENT,  
  NAME VARCHAR(20),  
  address VARCHAR(30) DEFAULT '广州'  
);  
  
-- 添加一条记录,不使用默认地址  
INSERT INTO users(NAME , address) VALUES('李四','广州');  
INSERT INTO users(NAME , address) VALUES('王五','广州');
```

一个简单的存储过程

```
-- 创建存储过程
DELIMITER $$
CREATE PROCEDURE testa()
BEGIN
    SELECT * FROM student WHERE id=2;
END $$

-- 调用存储过程
call testa();
```

## 小结

存储过程：

- 1.创建格式：create procedure 存储过程名
- 2.包含一个以上代码块，代码块使用begin和end 之间
- 3.在命令行中创建需要定义分隔符 delimiter \$\$
- 4.存储过程调用使用call命令

存储过程的特点：

- 1.能完成较复杂的判断和运算，而且处理逻辑都封装在数据库端，调用者不需要自己处理业务逻辑，一旦逻辑发生变化，只需要修改存储过程即可，而对调用者程序完全没有影响。
- 2.可编程性强，灵活
- 3.SQL编程的代码可重复使用
- 4.执行速度相对快一些
- 5.减少网络之间数据传输，节省开销

## 目标 存储过程的删除,查看操作

- 删除存储过程/函数

```
-- 删除存储过程
DROP PROCEDURE testa1;

-- 删除函数
DROP FUNCTION testa1;
```

- 查看存储过程或者函数

```
-- 查看存储过程或者函数的状态
SHOW PROCEDURE STATUS LIKE 'testa';

-- 查看存储过程或者函数的定义
SHOW CREATE PROCEDURE testa;
```

## 目标 存储过程的变量

需求1: 编写存储过程，使用变量取id=2的用户名。

```

DELIMITER $$
CREATE PROCEDURE testa3()
BEGIN
DECLARE my_uname VARCHAR(32) DEFAULT '';
SET my_uname='itheima';
SELECT NAME INTO my_uname FROM student WHERE id=2;
SELECT my_uname;
END $$

CALL testa3();

```

## 小结

- 1.变量的声明使用declare,一句declare只声明一个变量，变量必须先声明后使用
- 2.变量具有数据类型和长度，与mysql的SQL数据类型保持一致，因此甚至还能指定默认值、字符集和排序规则等
- 3.变量可以通过set来赋值，也可以通过select into的方式赋值
- 4.变量需要返回，可以使用select语句,如：select 变量名

**需求2：**统计表users,student的行数量和student表中英语最高分，数学最高分的注册时间。

```

DELIMITER $$
CREATE PROCEDURE stats_users_students5()
BEGIN
    BEGIN
        DECLARE users_sum INT DEFAULT 0;
        DECLARE students_sum INT DEFAULT 0;
        SELECT COUNT(*) INTO users_sum FROM users;
        SELECT COUNT(*) INTO students_sum FROM student;
        SELECT users_sum,students_sum;
    END;

    BEGIN
        DECLARE max_math INT;
        DECLARE max_english INT;
        SELECT MAX(math),MAX(english) INTO max_math,max_english FROM student ;
        SELECT users_sum,students_sum,max_math,max_english;
    END;
END;
$$
CALL stats_users_students5();

```

## 小结

- 1.变量是有作用域的，作用范围在begin与end块之间，end结束变量的作用范围即结束。
- 2.需要多个块之间传递值，可以使用全局变量，即放在所有代码块之前。
- 3.传参变量是全局的，可以在多个块之间起作用

## 目标 存储过程的传入参数IN

- 需求：编写存储过程，传入id，返回该用户的name

```
-- 需求: 编写存储过程, 传入id, 返回该用户的name
DELIMITER $$
CREATE PROCEDURE getName(my_uid INT)
BEGIN
    DECLARE my_uname VARCHAR(32) DEFAULT '';
    SELECT NAME INTO my_uname FROM student WHERE id=my_uid;
    SELECT my_uname;
END;
$$
CALL getName(2);
```

## 小结

- 1.传入参数: 类型为IN,表示该参数的值必须在调用存储过程时指定, 如果不显式指定为IN, 那么默认就是IN类型。
- 2.IN类型参数一般只用于传入, 在调用存储过程中一般不作修改和返回
- 3.如果调用存储过程中需要修改和返回值, 可以使用OUT类型参数

## 目标 存储过程的传出参数OUT

- 需求: 调用存储过程时, 传入uid返回该用户的uname

```
-- 需求: 调用存储过程时, 传入uid返回该用户的uname
DELIMITER $$
CREATE PROCEDURE getName22(IN my_uid INT,OUT my_uname VARCHAR(32))
BEGIN
    SELECT NAME INTO my_uname FROM student WHERE id=my_uid;
    SELECT my_uname;
END;
$$

SET @uname:='';
CALL getName22(2,@uname);
SELECT @uname AS myName;
```

- 小结
  - 1.传出参数: 在调用存储过程中, 可以改变其值, 并可返回
  - 2.OUT是传出参数, 不能用于传入参数值
  - 3.调用存储过程时, OUT参数也需要指定, 但必须是变量, 不能是常量
  - 4.如果既需要传入, 同时又需要传出, 则可以使用INOUT类型参数

## 目标 存储过程的可变参数INOUT

- 需求: 调用存储过程时, 参数my\_uid和my\_uname,既是传入, 也是传出参数

```
-- 需求: 调用存储过程时, 参数my_uid和my_uname,既是传入, 也是传出参数
DELIMITER $$
CREATE PROCEDURE getName33(INOUT my_uid INT,INOUT my_uname VARCHAR(32))
BEGIN
    SET my_uid=2;
    SET my_uname='hxf3';
```

```

SELECT id,NAME INTO my_uid,my_uname FROM student WHERE id=my_uid;
SELECT my_uid,my_uname;
END;
$$

SET @uname:='';
SET @uid:=0;
CALL getName33(@uid,@uname);
SELECT @uname AS myName;

```

- 小结
  - 1.可变变量INOUT：调用时可传入值，在调用过程中，可修改其值，同时也可返回值。
  - 2.INOUT参数集合了IN和OUT类型的参数功能
  - 3.INOUT调用时传入的是变量，而不是常量

## 目标 存储过程条件语句

### (1) 存储过程的条件语句

- 需求：编写存储过程，如果用户uid是偶数则就给出uname,其它情况只返回uid

```

-- 需求：编写存储过程，如果用户uid是偶数则就给出uname,其它情况只返回uid
DELIMITER $$
CREATE PROCEDURE getName44(IN my_uid INT )
BEGIN
DECLARE my_uname VARCHAR(32) DEFAULT '';
IF(my_uid%2=0)
THEN
    SELECT NAME INTO my_uname FROM student WHERE id=my_uid;
    SELECT my_uname;
ELSE
    SELECT my_uid;
END IF;
END;
$$
CALL getName44(1);
CALL getName44(2);

```

- 1.条件语句最基本的结构: if() then ...else ...end if;
- 2.If判断返回逻辑真或者假，表达式可以是任意返回真或者假的表达式

### (2) 存储过程的条件语句应用示例

- 需求：根据用户传入的uid参数判断：（1）如果状态status为1，则给用户score加10分 （2）如果状态status为2，则给用户score加20分 （3）其它情况加30分

```

DELIMITER $$
CREATE PROCEDURE addscore1(IN my_uid INT )
BEGIN
DECLARE my_status INT DEFAULT 0;

```



```

SELECT STATUS INTO my_status FROM student WHERE id=my_uid;
IF(my_status =1)
THEN
    UPDATE student SET math=math+10 , english=english+10 WHERE id=my_uid;
ELSEIF(my_status =2)
THEN
    UPDATE student SET math=math+20 , english=english+20 WHERE id=my_uid;
ELSE
    UPDATE student SET math=math+30 , english=english+30 WHERE id=my_uid;
END IF;
END;
$$
CALL addscore1(1);

```

## 目标 存储过程循环语句

### (1) while循环

- 需求：使用循环语句，向表uesrs中插入10条uid连续的记录。

```

-- 需求：使用循环语句，向表student(uid)中插入10条uid连续的记录。
DELIMITER $$
CREATE PROCEDURE insertdata()
BEGIN
DECLARE i INT DEFAULT 0;
WHILE(i< 10) DO
BEGIN
    SELECT i;
    SET i=i+1;
    INSERT INTO users(NAME , address) VALUES("孙悟空" , "广州");
END ;
END WHILE;
END;
$$
CALL insertdata();

```

- 1.while语句最基本的结构: while() do...end while;
- 2.while判断返回逻辑真或者假，表达式可以是任意返回真或者假的表达式

### (2)repeat循环语句

- 需求：使用repeat循环向表users插入10条uid连续的记录

```

DELIMITER $$
CREATE PROCEDURE insertdata2()
BEGIN
DECLARE i INT DEFAULT 100;
REPEAT
BEGIN
    SELECT i;
    SET i=i+1;
    INSERT INTO users(NAME) VALUES('黑马');
END ;

```

```
UNTIL i >= 110
```

```
END REPEAT;
```

```
END;
```

```
$$
```

```
CALL insertdata3();
```

1. **repeat**语句最基本的结构: repeat...until ...end REPEAT;

2. **until**判断返回逻辑真或者假, 表达式可以是任意返回真或者假的表达式  
只有当until语句为真时, 循环结束。

## 目标 光标的使用

在存储过程和函数中, 可以使用光标 (有时也称为游标) 对结果集进行循环的处理, 光标的使用包括了:

- 光标的申明
- OPEN
- FETCH
- CLOSE
- 需求: 编写存储过程, 使用光标, 把id为偶数的记录逐一更新用户名。

-- 编写存储过程, 使用光标, 把id为偶数的记录逐一更新用户名。

```
DELIMITER $$
```

```
CREATE PROCEDURE testcursor()
```

```
BEGIN
```

```
DECLARE stopflag INT DEFAULT 0;
```

```
DECLARE my_uname VARCHAR(20);
```

```
DECLARE uname_cur CURSOR FOR SELECT NAME FROM student WHERE id%2=0 ;
```

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET stopflag=1;
```

```
OPEN uname_cur; -- 打开光标
```

```
FETCH uname_cur INTO my_uname; -- 光标向前走一步, 取出一条记录放到变量my_uname中。
```

```
WHILE( stopflag=0 ) DO -- 如果光标还没有到结尾, 就继续
```

```
BEGIN
```

```
    UPDATE student SET NAME=CONCAT(my_uname, '_cur') WHERE NAME=my_uname;
```

```
    FETCH uname_cur INTO my_uname;
```

```
END ;
```

```
END WHILE;
```

```
CLOSE uname_cur;
```

```
END;
```

```
$$
```

```
DELIMITER ;
```

注意: 变量, 条件, 处理程序, 光标, 都是通过DECLARE定义的, 它们之间是有先后顺序要求的, 变量和条件必须在最前面声明, 然后才能是光标的申明, 最后才可以是处理程序的申明。

## 目标 简单的自定义函数

- 需求: 编写函数, 传入一个用户uid, 返回用户的uname

-- 需求: 编写函数, 传入一个用户uid, 返回用户的uname

```
DELIMITER $$
```

```
CREATE FUNCTION getFName1(my_uid INT) RETURNS VARCHAR(32)
```

```
READS SQL DATA # READS SQL DATA表示子程序包含读数据的语句，但不包含写数据的语句。
```

```
BEGIN
  DECLARE my_uname VARCHAR(32) DEFAULT '';
  SELECT NAME INTO my_uname FROM student WHERE id=my_uid;
  RETURN my_uname;
END;
$$
```

```
SELECT getFName1(3);
```

1. 创建函数使用create FUNCTION 函数名 (参数 ) RETURNS 返回类型
2. 函数体放在begin和end之间
3. Return指定函数的返回值
4. 函数调用: SELECT getuname()

## 目标 自定义函数综合应用示例

- 需求：输入用户ID，获得address, id, name组合的UUID值，在游戏中作为用户的唯一标识

```
-- 需求：输入用户ID，获得address, id, name组合的UUID值，
-- 在全区游戏中作为用户的唯一标识
DELIMITER $$
CREATE FUNCTION getuuid2(my_uid INT) RETURNS VARCHAR(30) CHARSET utf8
  READS SQL DATA # READS SQL DATA表示子程序包含读数据的语句，但不包含写数据的语句。
BEGIN
  DECLARE UUID VARCHAR(30) DEFAULT '';
  SELECT CONCAT(address , id , NAME) INTO UUID FROM student WHERE id=my_uid;
  RETURN UUID;
END;
$$

SELECT getuuid2(2);
```

- 需求：输入参数id，查询全部学生中数学分数高于id学生数学成绩的总和

```
-- 需求：输入参数id，查询出数学成绩高于该参数的学生的数学成绩总和
DELIMITER $$
CREATE FUNCTION mathAll(my_uid INT) RETURNS INT
  READS SQL DATA
BEGIN
  DECLARE math_id INT DEFAULT 0;
  DECLARE math_all INT DEFAULT 0;
  SELECT math INTO math_id FROM student WHERE id= my_uid;
  SELECT SUM(math) INTO math_all FROM student WHERE math > math_id;
  RETURN math_all ;
END;
$$
SELECT mathAll(2);
```

## 目标 MySQL触发器示例

## 什么是触发器

Mysql从5.0.2开始支持触发器功能，触发器是与表有关的数据库对象，在满足定义条件时触发，并执行触发器中定义的语句集合、触发器这种特性可以协助应用在数据库端确定数据的完整性。接下来以需求的方式演示。

- 需求：出于审计目的，当有人往表users插入一条记录时，把插入的uid,uname,插入动作和操作时间记录下来。

```
-- 出于审计目的，当有人往表users插入一条记录时，把插入的uid,uname,插入动作和操作时间记录下来。
DELIMITER $$
CREATE TRIGGER tr_users_insert AFTER INSERT ON users
FOR EACH ROW
BEGIN
    INSERT INTO oplog(uid,uname,ACTION,optime)
    VALUES(NEW.uid,NEW.uname,'insert',NOW());
END;
$$

1.创建触发器使用create TRIGGER 触发器名
2.什么时候触发? After INSERT ON users,除了after还有before,
是在对表操作之前(BEFORE)或者之后(AFTER)触发动作的。
3.对什么操作事件触发? after INSERT ON users , 操作事件包括insert,UPDATE,DELETE
4.对什么表触发? after INSERT ON users
5.影响的范围? For EACH ROW
```

- 需求：出于 审计目的，当删除users表时，记录删除前该记录的主要字段值

```
Delimiter $$
CREATE trigger tr_users_delete before delete on users
for each row
begin
    insert into oplog(uid,uname,action,optime,old_value,new_value)
    values(OLD.uid,OLD.uname,'delete',now(),OLD.regtime,OLD.regtime);
End;
$$
```

## 目标 事件调度器 (EVENT-SCHEDULE)

事件调度器是MySQL中提供的可做定时操作处理，或者周期操作处理的一个对象。

事件调度器的使用如下：

- 先确认是否开启了事件调度的支持，事件调度器开启后就可以定义时间调度器使用了

```
show variables like '%event_scheduler%';
set global event_scheduler =on;
```

- 事件调度器的入门案例

```
DELIMITER $$
CREATE EVENT IF NOT EXISTS event_hello
ON SCHEDULE EVERY 3 SECOND
ON COMPLETION PRESERVE
DO
```

```

BEGIN
    INSERT INTO users(NAME , address) VALUES('王五','广州');
END$$
DELIMITER ;

CREATE EVENT IF NOT EXISTS event_hello 创建使用create EVENT
ON SCHEDULE EVERY 3 minute 说明什么时候执行, 多长时间执行一次
ON COMPLETION preserve 调度计划执行完成后是否还保留
DO sql; 这个调度计划要做什么?

```

- 事件调度器计划示例

单次计划任务示例:

on schedule at '2016-12-12 04:00:00' 在 2016-12-12 04:00:00执行一次

重复计划任务

on schedule every 1 second 每秒执行一次

on schedule every 1 minuter 每分钟执行一次

on schedule every 1 day 每天执行一次

指定时间范围的重复计划任务

on schedule every 1 day starts '2016-12-12 20:20:20' 每天在20: 20: 20执行一次

on schedule every 1 minute starts '2016-12-12 9:00:00' ends '2016-12-12 11:00:00'

## 目标 综合案例

设计一个福彩3D的开奖存储过程, 每3分钟开奖一次。

- 第一步: 创建一张奖号表, 存储三个中奖号码, 以及生成时间。
- 第二步: 定义一个存储过程, 随机生成3个中奖号码存入到奖号表中。
- 第三步: 做一个事件调度器, 每隔3s调用一次存储过程。

-- 创建奖号的存储操作

```

CREATE TABLE lucky_num(
    id INT PRIMARY KEY AUTO_INCREMENT,
    num1 INT ,
    num2 INT ,
    num3 INT ,
    ctime DATETIME
)
-- 创建一个存储过程, 负责生成3个随机号码给表存储
DELIMITER $$
CREATE PROCEDURE create_lucky_num()
BEGIN
    INSERT INTO lucky_num(num1,num2,num3,ctime)
    SELECT FLOOR(RAND()*9)+1, FLOOR(RAND()*9)+1, FLOOR(RAND()*9)+1,NOW();
END $$
-- 开启事件调度器
DELIMITER $$
CREATE EVENT IF NOT EXISTS create_lucky_num
ON SCHEDULE EVERY 3 SECOND
ON COMPLETION PRESERVE

```

```
DO
    BEGIN
        CALL create_lucky_num;
    END$$
DELIMITER ;
```

## 第四章 锁问题

### 目标 锁的概念和分类

在现实生活中是为我们想要隐藏于外界所使用的一种工具,在计算机中,是协调多个进程或线程并发访问某一资源的一种机制,在数据库当中,除了传统的计算资源(CPU、RAM、I/O等等)的争用之外,**数据也是一种供许多用户共享访问的资源,如何保证数据并发访问的一致性、有效性,是所有数据库必须解决的一个问题**,锁的冲突也是影响数据库并发访问性能的一个重要因素。锁对数据库而言显得尤为重要。

在购买商品时,商品库存只有1个时,两个人同时买时,谁买到的问题,会用到事务,先从库存表中取出物品的数据,然后插入订单,付款后,插入付款表信息。更新商品的数量,在这个过程中,使用锁可以对有限的资源进行保护,实现隔离和并发的矛盾

#### 锁分类

- 按操作分
  - 读锁(共享锁): 针对同一份数据,多个读取操作可以同时进行而不互相影响
  - 写锁(排它锁): 当前写操作没有完成前,会阻断其他写锁和读锁
- 按粒度分
  - 表锁
  - 行锁
  - 页锁

### 目标 表锁

偏向MyISAM存储引擎,开销小,加锁快,无死锁,锁定粒度大,发生锁冲突的概率最高,并发最低,整张表就只能一个人使用

#### 示例

1.建立一张Myisam引擎的表

```
CREATE TABLE `user` (
  `id` INT(11) NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MYISAM DEFAULT CHARSET=utf8;

INSERT INTO USER(NAME) VALUES('孙悟空'),('猪八戒');
```

2.查看表有没有被锁过

```
show open tables;
```

3.对表加锁

```
lock table user read, user write;
```

4.对表进行解锁

```
unlock tables
```

目标 表锁读锁的演示

表锁中读写锁对操作和性能产生哪些影响

对user添加读锁 lock table user read;(共享锁)

- 当前连接:

是否可以查看自己	可以
是否可以更新	不可以
能不能读别的表	不可以，当前表还没有解锁,不能放下当前, 操作别的内容

- 另一个连接

是否可以查看	可以
是否可以更新	当更新时, 处理阻塞状态,等待解锁后, 才能进行更新
能不能读别的表	可以

目标 表锁写锁的演示

对user添加写锁 lock table user write;(排它)

- 当前连接

能否读自己锁过的表	可以
能否改自己锁过的表	可以
能否读取别的表	不可以

- 另一个连接

能否操作没有加过锁的表	可以
能否对被锁过的表进行操作	阻塞，等待解锁时, 才能查到

目标 表锁分析

```
show status like 'table%';
```

- Table\_locks\_immediate:产生表级锁定的次数,表示可以立即获取锁的查询次数
- Table\_locks\_waited:出现表级锁定争用而发生等待的次数
- Myisam的读写锁调度是写优先,这也是myisam不适合做写为主表的引擎
- 因为写锁后, 其它线程不能做任何操作,大量更新使用, 查询很难得到锁, 从而造成永久阻塞
- 淘宝: 买家(偏向读锁), 卖家(偏向写锁)

## 目标 行锁的概述和事物

偏向InnoDB存储引擎,基于行锁, 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。

InnoDB与MyISAM的最大不同点:一是支持事务, 二是采用了行级锁。

### 什么是事务

事物是一批操作, 要么同时成功, 要么同时失败!

### 转账的操作

```
- 创建数据表
CREATE TABLE account (
    id INT PRIMARY KEY AUTO_INCREMENT,
    NAME VARCHAR(10),
    balance DOUBLE
);

-- 添加数据
INSERT INTO account (NAME, balance) VALUES ('Jack', 1000), ('Rose', 1000);
模拟Jack给Rose转500元钱, 一个转账的业务操作最少要执行下面的2条语句:
Jack账号-500
Rose账号+500
-- 还原
update account set balance = 1000;
-- jack转账200给rose
-- jack扣钱
update account set balance = balance - 500 where name = 'jack';
-- rose加钱
update account set balance = balance + 500 where name = 'rose';
```

假设当Jack账号上-500元,服务器崩溃了。Rose的账号并没有+500元, 数据就出现问题了。我们需要保证其中一条SQL语句出现问题, 整个转账就算失败。只有两条SQL都成功了转账才算成功。这个时候就需要用到事务。

## 目标 手动提交事务

MYSQL中可以有两种方式进行事务的操作

- 1) 手动提交事务
- 2) 自动提交事务(5.5以后, 默认 事务会自动提交)



## 手动提交事务的SQL语句

功能	SQL语句
开启事务	start transaction
提交事务	commit
回滚事务	rollback

总结: 如果事务中SQL语句没有问题, commit提交事务, 会对数据库数据的数据进行改变。如果事务中SQL语句有问题, rollback回滚事务, 会回退到开启事务时的状态。

## 目标 取消自动提交

- MySQL默认每一条DML(增删改)语句都是一个单独的事务, 每条语句都会自动开启一个事务, 执行完毕自动提交事务, MySQL默认开始自动提交事务。
- 查看MySQL是否开启自动提交事务

```
select @@autocommit;    -- 只要是@@开头, 表示mysql中的全局变量
SHOW VARIABLES LIKE '%commit%';
```

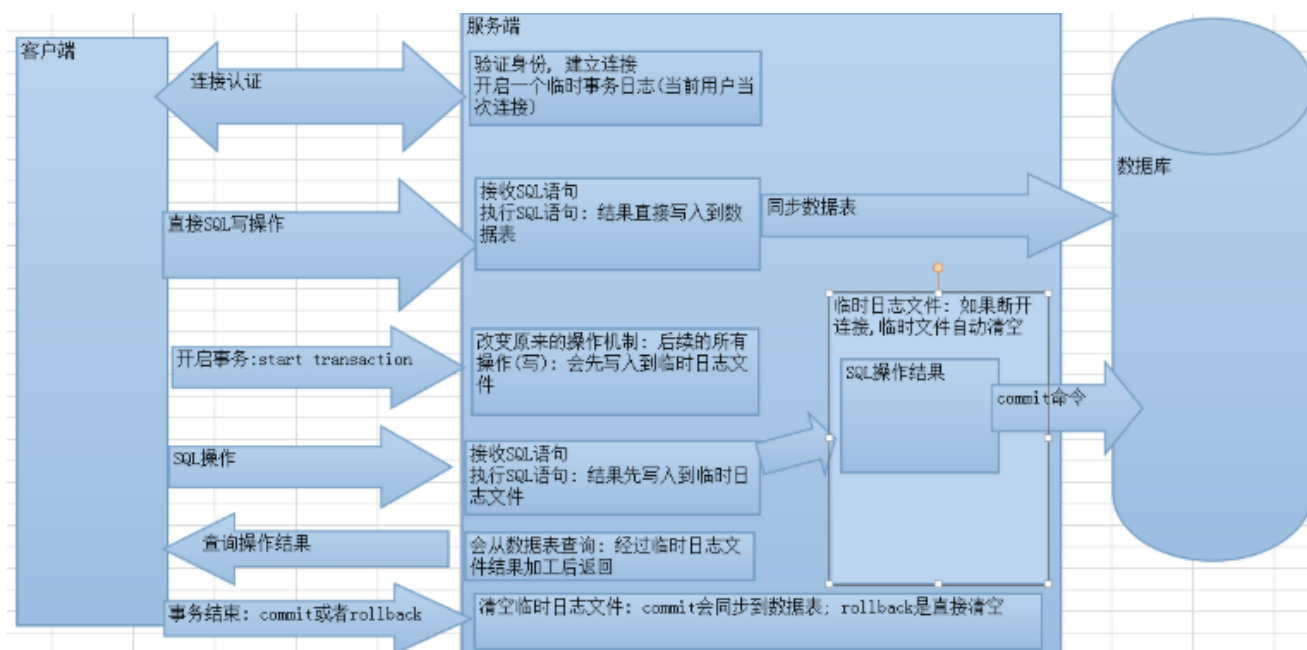
- 取消自动提交事务

```
####MySQL数据库默认是自动提交。
####能否让MySQL数据库永远是开启事物。永远需要手工提交呢?
SHOW VARIABLES LIKE '%commit%';
SELECT @@autocommit; # 1代表了自动开启了提交事物
SET autocommit = 0; # 1是自动提交事物, 0是开启全局事物手动提交 (开启事物)
```

## 目标 事务原理

事务开启之后, 所有的操作都会临时保存到事务日志中, 事务日志只有在得到commit命令才会同步到数据表中, 其他任何情况都会清空事务日志(rollback, 断开连接)

### 原理图



## 事务的步骤

- 1) 客户端连接数据库服务器, 创建连接时创建此用户临时日志文件
- 2) 开启事务以后, 所有的操作都会先写入到临时日志文件中
- 3) 所有的查询操作从表中查询, 但会经过日志文件加工后才返回
- 4) 如果事务提交commit则将日志文件中的数据写到表中, rollback否则清空日志文件。

## 目标 事物回滚点

### 什么是回滚点

在某些成功的操作完成之后, 后续的操作有可能成功有可能失败, 但是不管成功还是失败, 前面操作都已经成功, 可以在当前成功的位置设置一个回滚点。可以供后续失败操作返回到该位置, 而不是返回所有操作, 这个点称之为回滚点。

### 回滚点的操作语句

回滚点的操作语句	语句
设置回滚点	savepoint 名字
回到回滚点	rollback to 名字

### 具体操作:

- 1) 将数据还原到1000
- 2) 开启事务
- 3) 让Jack账号减3次钱, 每次10块
- 4) 设置回滚点: savepoint three\_times;
- 5) 让Jack账号减4次钱, 每次10块

6) 回到回滚点: rollback to three\_times;

总结: 设置回滚点可以让我们在失败的时候回到回滚点, 而不是回到事务开启的时候。

## 目标 事务的四大特性ACID

特性	含义
原子性 (Atomicity)	事务不要再拆分, 整个事务是一个整体, 要么全部成功, 要么全部失败
一致性 (Consistency)	事务开启**前和事务结束后, 对数据库影响状态是一致。如: 转账前总金额是2000, 转账后总金额也是2000.**
隔离性 (Isolation)	如果同时运行多个事务, 事务之间不能相互影响。
持久性 (Durability)	只要事务提交, 对数据库的影响是持久的。关机也不会改变。

## 目标 四种隔离级别和可能出现的问题

并发事务处理会带来一些问题, 所以事物与事物之间需要进行适当的隔离, 但是隔离是存在级别的, 并不是隔离的越高级越好, 隔离越高级, 性能越差!

级别	名字	隔离级别	脏读	不可重复读	幻读	数据库默认隔离级别
1	读未提交	read uncommitted	是	是	是	
2	读已提交	read committed	否	是	是	Oracle和SQL Server
3	可重复读	repeatable read	否	否	是	MySQL默认
4	串行化	serializable	否	否	否	

事物有四种隔离级别。

- 读未提交: 出现脏读 (一个事物读到了其他事物未提交的数据)
- 读已提交: 出现不可重复读 (一个事物多次读取到别人已提交的数据)
- 可重复读: 出现幻读 (一个事物的修改操作, 发现新增加了修改的数据)
- 串行化: 读是共享锁, 写是排他锁。

## 目标 四种隔离级别的演示

- 事物的隔离级别 (读未提交): 出现脏读

```
REPEATABLE-READ: 可重复读, 是MySQL的默认隔离级别。
SHOW VARIABLES LIKE '%isolation%';
-- 或
SELECT @@tx_isolation;
-- 设置隔离级别成为: 读未提交。出现脏读
SET GLOBAL TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

问题：出现脏读，一个事物读取到了其他事物未提交的数据。

- 事物的隔离级别（读已提交）：不会出现脏读，但是出现不可重复读

```
SHOW VARIABLES LIKE '%isolation%';
-- 或
SELECT @@tx_isolation;
-- 设置隔离级别成为：读可提交，不会出现脏读，但是会出现不可重复读。
SET GLOBAL TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

可以解决脏读问题：不会读取到其他事物未提交的数据。但是会出现不可重复读。

问题：出现不可重复读，一次事物中，多次读取到的数据不一样，读取到了别人已提交的数据。

- 事物的隔离级别(可重复读)：可重复读演示：不会出现不可重复读，会出现幻读

```
SHOW VARIABLES LIKE '%isolation%';
-- 或
SELECT @@tx_isolation;
SET GLOBAL TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

可以解决的问题：不会出现不可重复读，一次事物多次读取到数据是一样的。

问题：出现了幻读。

- 事物的隔离级别(可串行化)：serializable可串行化。所有问题都不会出现

```
SHOW VARIABLES LIKE '%isolation%';
-- 或
SELECT @@tx_isolation;
SET GLOBAL TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

## 目标 演示行锁

行锁偏向InnoDB存储引擎,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。

InnoDB与MyISAM的最大不同点:一是支持事务,二是采用了行级锁。

接下来演示一下MySQL默认行锁的使用。

- 执行更新操作（可重复读）
  - 自己可以查看到更新的内容
  - 连接2看不到更新的内容,只有commit后,才能看到更新的内容
- 连接1执行更新操作,连接2也执行更新操作,更新同一条记录
  - 连接1没有提交事务时,连接2更新处于阻塞状态
  - 当commit时,连接2才会继续执行 连接2更新也要commit
- 连接1和连接2同时更新数据,但更新的不是同一条记录
  - 不会影响

## 目标 如何锁定一行数据

如何在操作时锁定一行操作。

- 在查询之后添加for update, 其它操作会被阻塞,直到锁定的行提交commit;

查看行锁的使用信息:

```
show status like 'innodb_row_lock%';
```

## 目标 悲观锁,乐观锁

### • 悲观锁

- 就是很悲观, 它对于数据被外界修改持保守态度, **认为数据随时会修改**。
- 整个数据处理中需要将数据加锁。悲观锁一般都是依靠关系数据库提供的锁机制
- 事实上关系数据库中的行锁, 表锁不论是读写锁都是悲观锁

### • 乐观锁

- 顾名思义, 就是很乐观, 每次自己操作数据的时候认为没有人回来修改它, 所以不去加锁。
- 但是在更新的时候会去判断在此期间数据有没有被修改
- 需要用户自己去实现, 不会发生并发抢占资源, 只有在提交操作的时候检查是否违反数据完整性

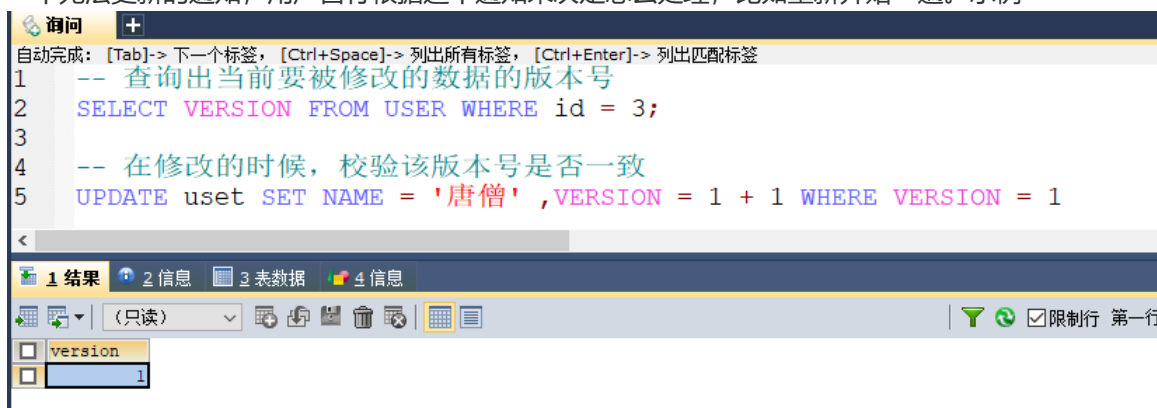
### • 悲观锁,乐观锁使用前提

- 对于读操作远多于写操作的时候, 这时候一个更新操作加锁会阻塞所有读取, 降低了吞吐量。最后还要释放锁, 锁是需要一些开销的, 这时候可以选择**乐观锁**
- 如果是读写比例差距不是非常大或者你的系统没有响应不及时, 吞吐量瓶颈问题, 那就不要去使用乐观锁, 它增加了复杂度, 也带来了业务额外的风险。

### • 乐观锁的实现方式

#### ◦ 版本号

- 就是给数据增加一个版本标识, 在数据库上就是表中增加一个version字段每次更新把这个字段加1
- 读取数据的时候把version读出来, 更新的时候比较version
- 如果还是开始读取的version就可以更新了
- 如果现在的version比老的version大, 说明有其他事务更新了该数据, 并增加了版本号, 这时候得到一个无法更新的通知, 用户自行根据这个通知来决定怎么处理, 比如重新开始一遍。示例



#### ◦ 时间戳

- 和版本号基本一样, 只是通过时间戳来判断而已, 注意时间戳要使用数据库服务器的时间戳不能是业务系统的时间
- 同样是在需要乐观锁控制的table中增加一个字段, 名称无所谓, 字段类型使用时间 (timestamp)
- 和上面的version类似, 也是在更新提交的时候检查当前数据库中数据的时间戳和自己更新前取到的时间戳进行对比如果一致则OK, 否则就是版本冲突。

## 第五章 Mysql面试指南-数据库优化

## 目标 Mysql性能优化概述

在应用开发的过程中，由于前期数据量少，开发人员编写的SQL语句或者数据库整体解决方案都更重视在功能上的实现，但是当应用系统正式上线后，随着生成数据量的急剧增长，很多SQL语句和数据库整体方案开始逐渐显露出了性能问题，对生成的影响也越来越大，此时Mysql数据库的性能问题成为系统应用的瓶颈，因此需要进行Mysql数据库的性能优化。

### 性能下降的表现

- 执行时间长
- 等待时间长

### 性能下降的原因

- 查询语句写的不好，各种连接，各种子查询导致用不上索引或者没有建立索引
- 建立的索引失效，建立了索引,在真正执行时,没有用上建立的索引
- 关联查询太多join
- 服务器调优及配置参数导致，如果设置的不合理,比例不恰当,也会导致性能下降,sql变慢
- 系统架构的问题

### 数据库优化的目的：

- 避免出现页面访问错误。
- 增加数据库的稳定性：很多数据库的问题都是由于查询低效引起的。
- 优化用户体验，流畅页面访问速度，支撑业务正常开张。

### 掌握

- 如果写出高质量的sql
- 如果保证索引不失效
- 搭建高并发，高可靠的系统架构。

## 目标 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

- 表的设计合理化(符合3NF，有时候要进行反三范式操作)
- 添加适当索引(index) (重点)
- 分表技术(水平分割、垂直分割)
- 主从复制，读写分离
- 对mysql配置优化 [配置最大并发数my.ini, 调整缓存大小]
- 系统应用优化等（集合缓存技术）
- 服务器的硬件优化

## 目标 一般优化步骤和慢查询定位

### 通过show status命令了解SQL的执行频率

MySQL客户端连接成功后，通过使用show [session|global] status 命令可以提供服务器状态信息。其中的session来表示当前的连接的统计结果，global来表示自数据库上次启动至今的统计结果。默认是session级别的。

下面的例子：

```
show status like 'Com_%';  
-- 试图连接MySQL服务器的次数  
show status like 'connections';  
-- 显示慢查询次数  
show status like 'slow_queries';
```

其中Com\_XXX表示XXX语句所执行的次数。

重点注意：Com\_select, Com\_insert, Com\_update, Com\_delete通过这几个参数，可以容易地了解到当前数据库的应用是以插入更新为主还是以查询操作为主，以及各类的SQL大致的执行比例是多少。 还有几个常用的参数便于用户了解数据库的基本情况。

- Connections：试图连接MySQL服务器的次数
- Uptime：服务器工作的时间（单位秒）
- Slow\_queries：慢查询的次数（默认是慢查询时间10s）

## 慢查询日志

慢查询日志：记录具体执行效率较低的SQL语句的日志信息。

在默认情况下mysql慢查询日志记录是关闭的，同时慢查询日志默认不记录：**管理语句和不使用索引进行查询的语句。**

MySQL的慢查询日志是MySQL提供的一种日志记录,它用来记录在MySQL中响应时间超过阈值的语句 具体指运行时间超过long\_query\_time值的SQL，则会被记录到慢查询日志中。long\_query\_time的默认值为10S，意思是运行10S以上的语句。就会被认作是慢查询,默认情况下，Mysql数据库并不启动慢查询日志，需要我们手动来设置这个参数，如果不是调优需要的话，一般不建议启动该参数，因为开启慢查询日志会或多或少带来一定的性能影响。

通过慢查询日志定位执行效率较低的SQL语句。慢查询日志记录了所有执行时间超过long\_query\_time所设置的SQL语句。

- 查看是否开启慢查询日志

```
show variables like '%slow_query_log%'  
开启  
set global slow_query_log=1;  
只对当前数据库生效，如果重启后，则会失效  
如果想永久生效,必须要修改配置文件  
slow_query_log = 1  
slow_query_log_file=地址
```

- 那么开启了慢查询日志后，什么样的SQL才会记录到慢查询日志里面呢？

```
show variables like 'long_query_time'  
set global long_query_time=4;
```

- 要断开连接后,才能生效

```
show global variables like 'long_query_time';  
select sleep(4)  
show global status like '%slow_queries%';
```

## 慢查询日志分析工具mysqldumpslow

慢查询日志中可能会出现很多的日志记录，我们可以通过慢查询日志工具进行分析，MySQL默认安装了mysqldumpslow工具实现对慢查询日志信息的分析。

### 示例

```
-- 得到返回记录集最多的10个SQL。
mysqldumpslow.pl -s r -t 10 C:\soft\DESKTOP-8GVEK4U-slow.log
-- 得到访问次数最多的10个SQL
mysqldumpslow.pl -s c -t 10 C:\soft\DESKTOP-8GVEK4U-slow.log
-- 得到按照时间排序的前10条里面含有左连接的查询语句。
mysqldumpslow.pl -s t -t 10 -g "left join" C:\soft\DESKTOP-8GVEK4U-slow.log
-- 另外建议在使用这些命令时结合 | 和more 使用，否则有可能出现刷屏的情况。
mysqldumpslow.pl -s r -t 20 C:\soft\DESKTOP-8GVEK4U-slow.log

Count: 4 (执行了多少次)   Time=375.01s (每次执行的时间) (1500s) (一共执行了多少时间)   Lock=0.00s
(0s) (等待锁的时间)   Rows=10200.3 (每次返回的记录数) (40801) (总共返回的记录数), username[password]@[10.194.172.41]
```

### 参数

```
-s 按照那种方式排序
c: 访问计数
l: 锁定时间
r: 返回记录
al: 平均锁定时间
ar: 平均访问记录数
at: 平均查询时间
-t 是top n的意思，返回多少条数据。
-g 可以跟上正则匹配模式，大小写不敏感。
```

第三方的慢查询日志分析工具：mysqsla，myprofi，pt-query-diges。

## Show Profile分析

**概述** Show Profile是mysql提供的可以用来分析当前会话中sql语句执行的资源消耗情况的工具，可用于sql调优的测量。默认情况下处于关闭状态，并保存最近15次的运行结果。把一条sql在mysql当中每一个环节耗费的时候都记录下来。使用

```
-- 1.查看当前版本是否支持
Show variables like 'profiling';
-- 2.打开profile
set profiling = on
-- 3.查看结果
show profiles
-- 4.诊断sql
show profile cpu,block io for query 88;
```

### 当出现以下选项时, 要进行优化

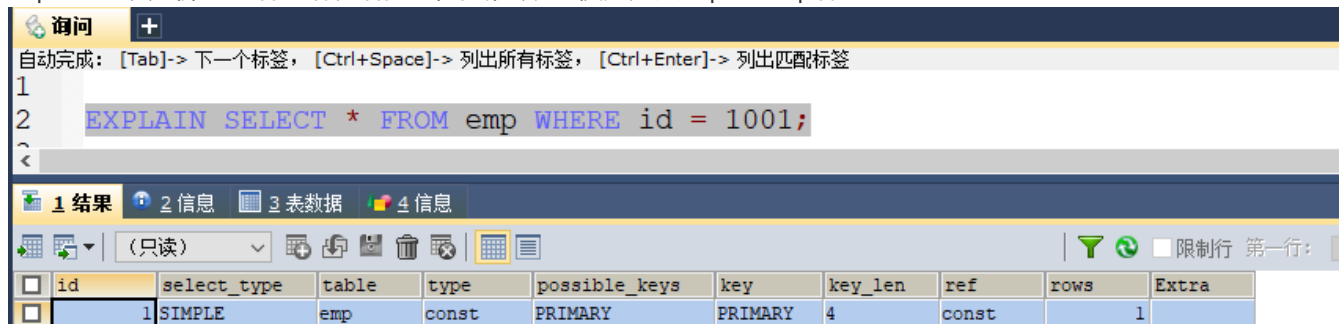
- Creating tmp table 创建临时表copy数据到临时表,用完再进行删除



- Copying to tmp table on disk 把内存中临时表复制到磁盘
- Locked 被锁定

## 通过EXPLAIN分析低效SQL的执行计划

查询执行计划, 使用explain关键字, 可以模拟优化器执行的SQL语句, 从而知道MYSQL是如何处理sql语句的 通过 Explain可以分析查询语句或表结构的性能瓶颈。 使用方法explain sql语句



分析包含信息

- id
  - select查询的序列号包含一组数字,表示查询中执行select子句或操作表的顺序
  - 相同,顺序走
  - 不同,看谁大,大的先执行
- type 访问类型排列,从上到下是最好的方式到最差的方式
  - system
    - 表中有一行记录(系统表) 这是const类型的特例,平时不会出现
  - const
    - 表示通过索引一次就找到了
    - const用于比较primary 或者 unique索引. 直接查询主键或者唯一索引
    - 因为只匹配一行数据,所以很快
  - eq\_ref
    - 唯一性索引扫描
    - 对于每个索引键,表中只有一条记录与之匹配
    - 常见于主键或唯一索引扫描
  - ref
    - 非唯一性索引扫描,返回匹配某个单独值的所有行
    - 本质上也是一种索引访问
    - 它返回所有匹配某个单独值的行
    - 可能会找到多个符合条件的行,
    - 所以它应该属于查找和扫描的混合体
  - range
    - 只检索给定范围的行,使用一个索引来选择行
    - key列显示使用了哪个索引
    - 一般就是在你的where语句中出现between\ in等查询
    - 这种范围扫描索引比全表扫描要好
    - 因为它只需要开始于索引的某一点.而结束语另一点
    - 不用扫描全部索引
  - index

- Full Index Scan
  - index与All区别为index类型只遍历索引树,通常比All要快,因为索引文件通常比数据文件要小
  - all和index都是读全表,但index是从索引中读取,all是从硬盘当中读取
- ALL
  - 将全表进行扫描,从硬盘当中读取数据如果出现了All 数据量非常大,一定要去做优化
- 要求
  - 一般来说,保证查询至少达到range级别, 最好能达到ref
- key:
  - 实际使用的索引,如果为NULL,则没有使用索引
  - 查询中若使用了覆盖索引,则该索引仅出现在key列表中
  - possible\_keys与key关系 理论应该用到哪些索引 实际用到了哪些索引覆盖索引
  - 查询的字段和建立的字段刚好吻合,这种我们称为覆盖索引
- select\_type:查询类型,主要用于区别普通查询,联合查询,子查询等复杂查询
  - SIMPLE 简单select查询,查询中不包含子查询或者UNION
  - PRIMARY查询中若包含任何复杂的子查询,最外层查询则被标记为primary
  - SUBQUERY: 在select或where中包含了子查询
  - DERIVED: 在from列表中包含的子查询被标记为derived(衍生)把结果放在临时表当中
  - UNION 若第二个select出现的union之后,则被标记为union 若union包含在from子句的子查询中,外层select将被标记为deriver
  - UNION RESULT从union表获取结果select, 两个UNION合并的结果集在最后
- table 显示这一行的数据是关于哪张表的
- partitions 如果查询是基于分区表的话, 会显示查询访问的分区
- possible\_keys key与keys主要作用,是查看是否使用了建立的索引, 也即判断索引失效 在建立多个索引 的情况下,mysql最终用到了哪一个索引 possible\_keys 显示可能应用在这张表中的索引,一个或者多个 查询涉及到的字段上若存在索引,则该索引将被列出,但不一定被查询实际使用 可能自己创建了4个索引,在执行的时候,可能根据内部的自动判断,只使用了3个
- key\_len:表示索引中使用的字节数,可通过该列计算查询中使用的索引长度 .
- ref 索引是否被引入到, 到底引用到了哪几个索引
- rows根据表统计信息及索引选用情况,大致估算出找到所需的记录所需要读取的行数
- filtered:满足查询的记录数量的比例, 注意是百分比, 不是具体记录数, 值越大越好, filtered列的值依赖统计信息, 并不十分准确
- Extra:产生的值
  - Using filesort说明mysql会对数据使用一个外部的索引排序,而不是按照表内的索引顺序进行Mysql中无法利用索引完成排序操作称为"文件排序"
  - Using temporary使用了临时表保存中间结果,Mysql在对查询结果排序时, 使用了临时表,常见于排序 orderby 和分组查询group by
  - use index表示相应的select中使用了覆盖索引,避免访问了表的数据行, 效率很好如果同时出现using where 表明索引被用来执行索引键值的查找如果没有同时出现using where 表明索引 用来读取数据而非执行查找动作
  - using where: 表明使用了wher过滤
  - using join buffer: 使用了连接缓存
  - impossible where: where 子句的值总是false 不能用来获取任何元组

## 目标 常用的SQL优化-索引优化

## 什么是索引

索引是解决SQL性能问题的重要手段之一，使用索引可以帮助用户解决大多数的SQL性能问题。索引就是数据结构，**索引是Mysql高效获取数据的数据结构**，类似新华字典的索引目录，可以通过索引目录快速查到你想要的字。排好序的快速查找数据

## 为什么要建立索引

提高查询效率，没有排序之前一个一个往后找，通过索引进行排序之后，可以直接定义到想要的位置，**排好序的快速查找数据结构-->就是索引**

## 建立索引的优劣势

### 优势

- 索引类似大学图书馆建立的书目索引提高数据检索的效率
- 降低数据库的IO成本
- 通过索引对数据项进行排序，降低数据排序成本，降低了CPU的消耗

### 索引劣势

- 一般来说，索引本身也很大，索引往往以文件的形式存储到磁盘上
- 索引也是一张表，该表保存了主键与索引字段，并指向实体表的记录。所以索引也是要占磁盘空间的
- 虽然索引提高了查询速度，但是会降低更新表的速度。
- 因为更新表时，MYSQL不仅要保存数据，还要保存一下索引文件每次更新添加了索引列的字段，会调整因为更新所带来的键值变化后索引的信息

## 目标 索引分类

- **单值索引**
  - 一个索引只包含单个列，一个表可以有多个单值索引，一般来说，一个表建立索引不要超过5个
- **唯一索引**
  - 索引列的值必须唯一，但允许有空值
- **复合索引**
  - 一个索引包含多个列
- **全文索引**
  - MySQL全文检索是利用查询关键字和查询列内容之间的相关度进行检索，可以利用全文索引来提高匹配的速度。

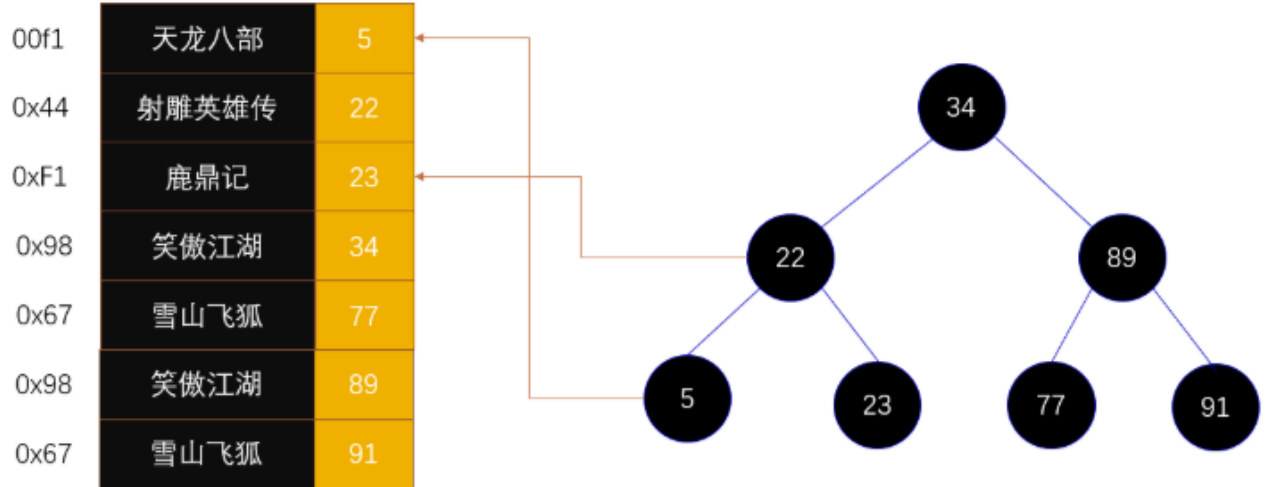
## 目标 索引快速查找数据的原因和索引类型

为了加快数据的查找，可以维护**二叉查找树**，每个节点分别包含索引键值和一个指向对应数据记录的物理地址的指针，这样就可以运用二叉查找在一定的复杂度内获取相应的数据，从而快速的检索出符合条件的记录

### 二叉查找树

- 左子树的键值小于根的键值

- 右子树的键值大于根的键值



除了二叉树还有B-tree索引, 我平时所说的索引,如果没有特别指定, 都是指B树结构组织的索引,其中聚焦索引,次要索引,复合索引,前缀索引,唯一默认都是B+树索引,B+树索引之外, 还有哈希索引(Hash index)等

索引是在 MySQL 的存储引擎层中实现的,而不是在服务器层实现的。所以每种存储引擎的索引都不一定完全相同,也不是所有的存储引擎都支持所有的索引类型。MySQL 目前提供了以下 4 种索引。

- B-Tree 索引: 最常见的索引类型,大部分引擎都支持 B 树索引。
- HASH 索引: 只有 Memory 引擎支持,使用场景简单。
- R-Tree 索引 (空间索引): 空间索引是 MyISAM 的一个特殊索引类型,主要用于地理空间数据类型,通常使用较少,不做特别介绍。
- Full-text (全文索引): 全文索引也是 MyISAM 的一个特殊索引类型,主要用于全文索引, InnoDB 从 MySQL 5.6 版本开始提供对全文索引的支持。

表 18-1 MyISAM、InnoDB、Memory 三个常用引擎支持的索引类型比较

索引	MyISAM 引擎	InnoDB 引擎	Memory 引擎
B-Tree 索引	支持	支持	支持
HASH 索引	不支持	不支持	支持
R-Tree 索引	支持	不支持	不支持
Full-text 索引	支持	暂不支持	不支持

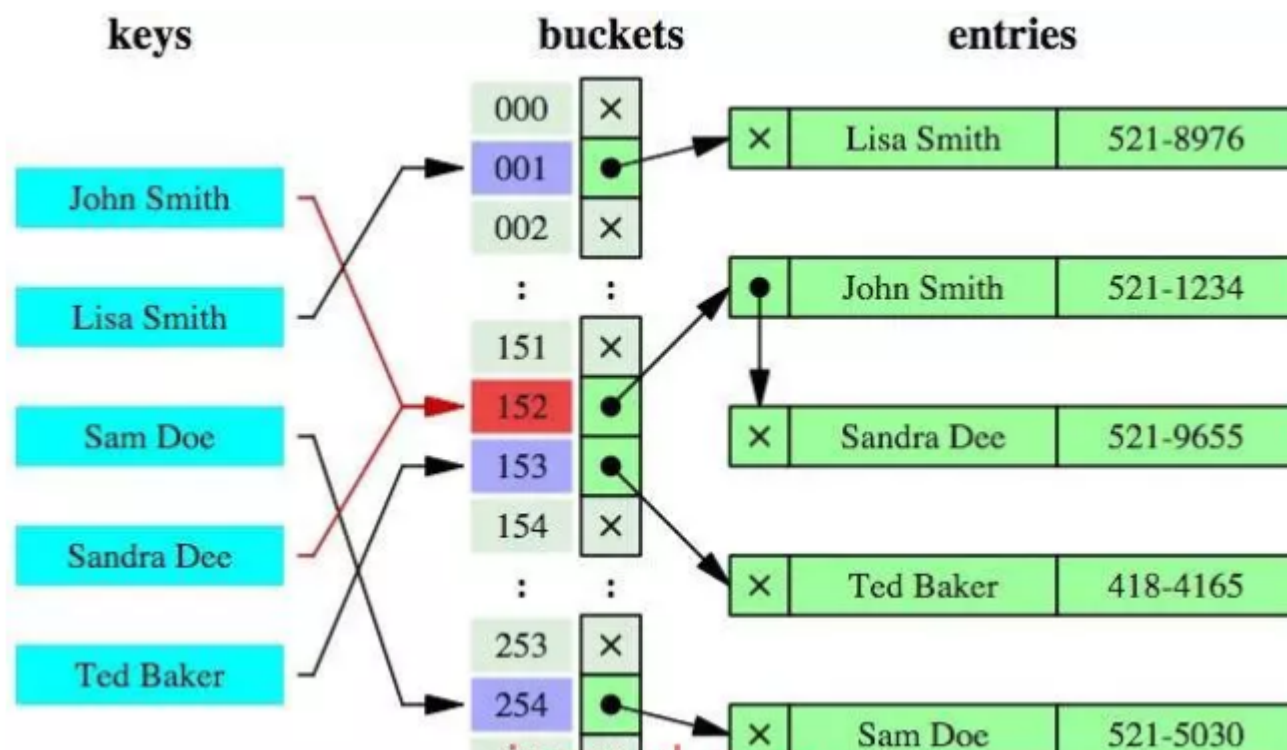
比较常用到的索引就是 B-Tree 索引和 Hash 索引。Hash 索引相对简单,只有 Memory/Heap 引擎支持 Hash 索引。Hash 索引适用于 Key-Value 查询,通过 Hash 索引要比通过 B-Tree 索引查询更迅速;Hash 索引不适用范围查询,例如<、>、<=、>=这类操作。如果使用 Memory/Heap 引擎并且 where 条件中不使用“=”进行索引列,那么不会用到索引。Memory/Heap 引擎只有在“=”的条件下才会使用索引。

B-Tree 索引比较复杂,下面将详细分析 MySQL 是如何利用 B-Tree 索引的。

### 目标 Hash索引

哈希索引就是采用一定的**哈希算法**,把键值换算成新的哈希值,检索时不需要类似B+树那样从根节点到叶子节点逐级查找,只需一次哈希算法即可**立刻定位到相应的位置,速度非常快**

本质上就是**把键值换算成新的哈希值**,根据这个**哈希值来定位**。



看起来哈希索引很牛逼啊，但其实哈希索引有好几个局限(根据他本质的原理可得)：

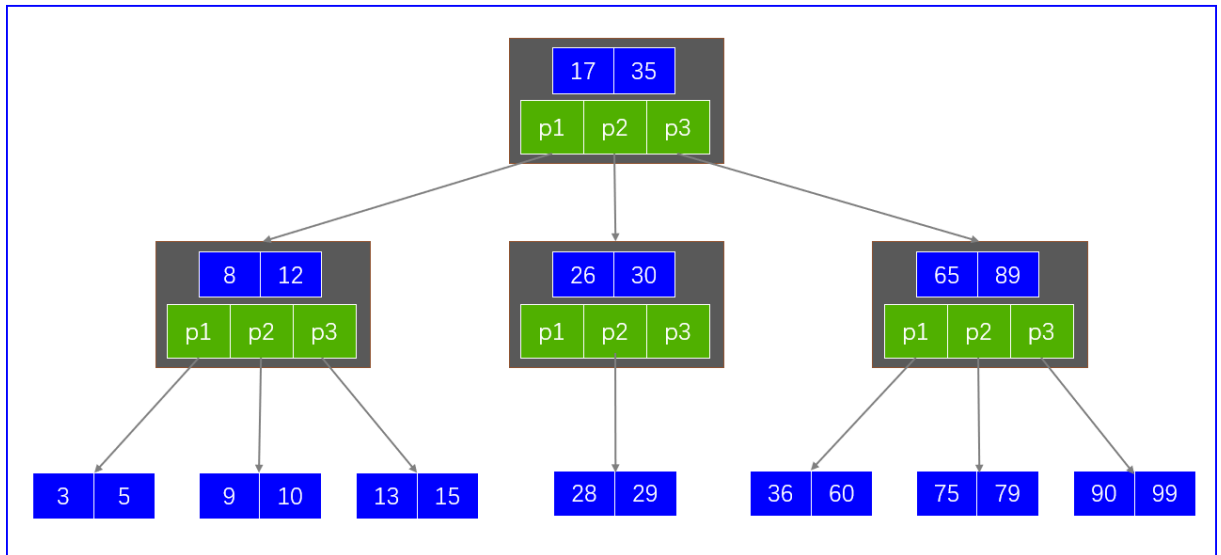
- 哈希索引也没办法利用索引完成**排序**
- 不支持**最左匹配原则**
- 在有大量重复键值情况下，哈希索引的效率也是极低的---->**哈希碰撞**问题。
- **不支持范围查询**

## 目标 B-Tree

### 平衡多路查找树 特性

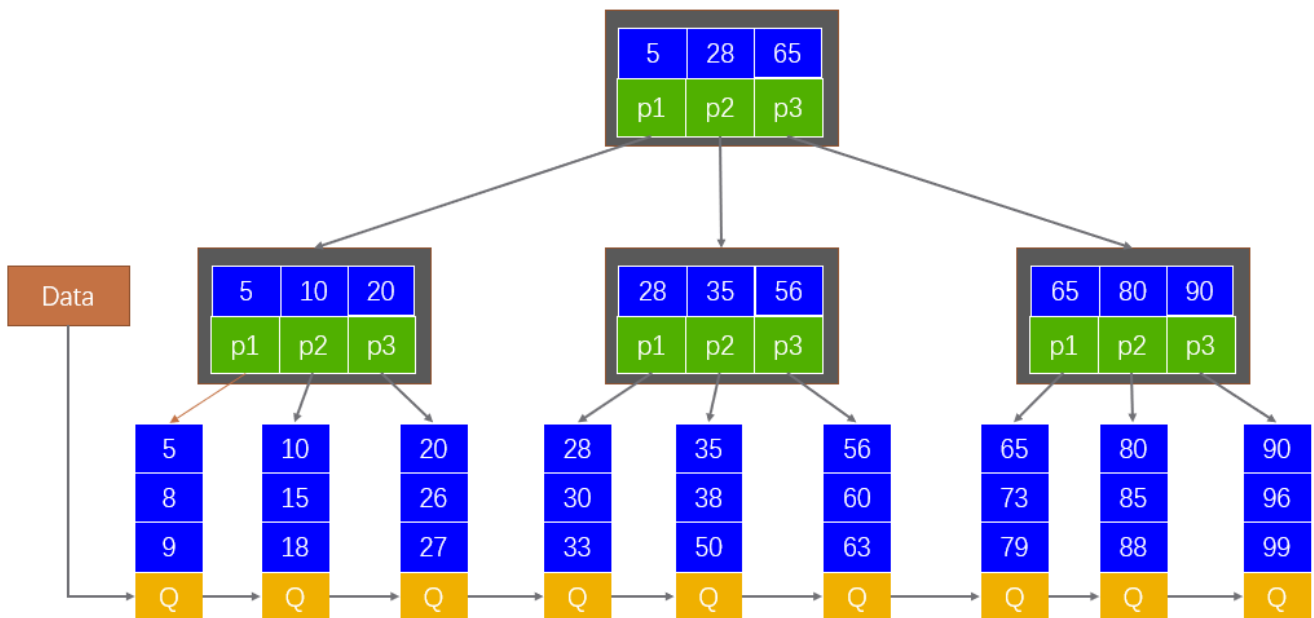
- m阶B-Tree满足以下条件：
  - 0.根节点至少包括两个孩子
  - 1.树中每个节点最多有m个孩子( $m \geq 2$ )
  - 2.除了根节点和叶子节点外，其它每个节点至少有 $\lceil m/2 \rceil$ 个孩子。
  - 3.所有叶子节点都在同一层
  - 节点中数据key从左到右递增排列

◦ 示例图



## 目标 B+Tree

B+树是B树的变体,基本与B-Tree相同



• 不同点

- 非叶子节点的子树指针与关键字个数相同, 这样使得B+树每个节点所能保存的关键字大大增加
- 非叶子节点的子树指针, 指向关键字值 $[k[i], k[i+1]]$ 的子树( $10 < 18 < 20$ )
- 非叶子节点仅用来做索引, 数据都保存在叶子节点中
- 所有叶子节点均有一个链指针指向下一个叶子节点
  - 链接起来, 能够方便我们直接在叶子节点做范围统计
  - 而不是再回到子节点中
  - 一旦定位到某个叶子节点, 便可以从该叶子节点横向的去跨子树去做统计

• 采用B+Tree做为主流索引数据结构的原因

- B+树的磁盘读写代价更低
  - 内部的结构并没有指向关键字的具体指针
  - 不存放数据,只存放索引信息, 因此其内部结点相对B树更小。如果把所有同一内部结点的关键字存放在同一盘块中, 那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说I/O读写次数也就降低了
- B+树的查询效率更加稳定
  - 内部节点并不是最终指向文件内容的节点,只是叶子节点中关键字的索引,
  - 所以它任何关键字的查找,必须走一条从根节点到叶子节点的路
  - 所有关键字查询的长度相同,导致每一个数据查询的效率也几乎是相同
- B+树更有利于对数据库的扫描
  - B树在提高IO性能同时,并没有解决元素遍历效率低下问题, B+树只需要遍历叶子节点,就可以解决对全部关键字信息的扫描, 对数据库中, 频繁使用的范围查询,性能更高

## 目标 索引的基本操作

### 创建索引

```
create [UNIQUE|primary|fulltext] index 索引名称 ON 表名(字段(长度))
```

```
CREATE INDEX emp_name_index ON employee(NAME);
```

### 测试脚本

```
--创建MyISAM模式表方便批量跑数据
CREATE TABLE `logs1` (
  `id` INT(11) NOT NULL AUTO_INCREMENT,
  `logtype` VARCHAR(255) DEFAULT NULL,
  `num` INT ,

  PRIMARY KEY (`id`)
) ENGINE=MYISAM AUTO_INCREMENT=1811 DEFAULT CHARSET=utf8 ;

--建存储过程
DELIMITER $$
CREATE PROCEDURE insertdata2()
BEGIN
DECLARE n INT DEFAULT 1;
  loopname:LOOP
    INSERT INTO `logs1`(`logtype`,`num`) VALUES ( '/index',FLOOR(RAND()*99)+1 );
    SET n=n+1;
  IF n=1000000 THEN
    LEAVE loopname;
  END IF;
  END LOOP loopname;
END;
$$

--执行存储过程
CALL insertdata2();

--数据插入成功后修改表模式InnoDB 时间稍微久点
```

```
ALTER TABLE `logs1` ENGINE=INNODB;
```

没有使用索引查询的时间如下：

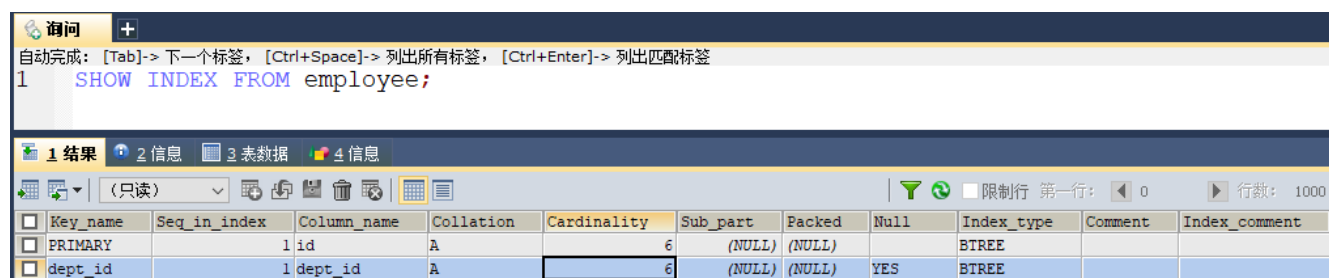
```
SELECT COUNT(num) FROM logs1 WHERE num = 15; # 耗时接近5s
```

创建索引后查询的时间如下：

```
-- 为num创建一个索引
CREATE INDEX index_num ON logs1(num);
-- 再次查询耗时
SELECT COUNT(num) FROM logs1 WHERE num = 15; # 耗时接近0.153s
```

查看索引

```
show index from 表名
```



自动完成: [Tab]-> 下一个标签, [Ctrl+Space]-> 列出所有标签, [Ctrl+Enter]-> 列出匹配标签

```
1 SHOW INDEX FROM employee;
```

Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment
PRIMARY	1	id	A	6	(NULL)	(NULL)		BTREE		
dept_id	1	dept_id	A	6	(NULL)	(NULL)	YES	BTREE		

删除索引

```
drop index[索引名称] on 表名
```

```
DROP INDEX emp_name_index ON employee;
```

更改索引

```
alter table tab_name add primary key(column_list)
-- 添加一个主键,索引必须是唯一索引,不能为NULL
alter table tab_name add unique index_name(column_list)
-- 创建的索引是唯一索引,可以为NULL
alter table tab_name add index index_name(column_list)
-- 普通索引,索引值可出现多次
alter table tab_name add fulltext index_name(column_list)
-- 全文索引
```

## 目标 索引建立选择

- 适合建立索引
  - 1.主键自动建立唯一索引:primary
  - 2.频繁作为查询条件的字段应该创建索引, 比如银行系统银行帐号,电信系统的手机号
  - 3.查询中与其它表关联的字段,外键关系建立索引, 比如员工表的部门外键



- 4.频繁更新的字段不适合建立索引，每次更新不单单更新数据,还要更新索引
  - 5.where条件里用不到的字段不建立索引
  - 6.查询中排序的字段,排序的字段若通过索引去访问将大大提升排序速度，索引能够提高检索的速度和排序的速度
  - 7.查询中统计或分组的字段
- 不适合建立索引
  - 记录比较少
  - 经常增删改的表
    - 索引提高了查询的速度，同时却会降低更新表的速度,因为建立索引后, 如果对表进行INSERT,UPDATE和DELETE, MYSQL不仅要保存数据,还要保存一下索引文件
  - 数据重复的表字段
    - 如果某个数据列包含了许多重复的内容,为它建立索引 就没有太大在的实际效果，比如表中的某一个字段为国籍,性别，数据的差异率不高,这种建立索引就没有太多意义。

## 目标 MySQL如何正确使用索引

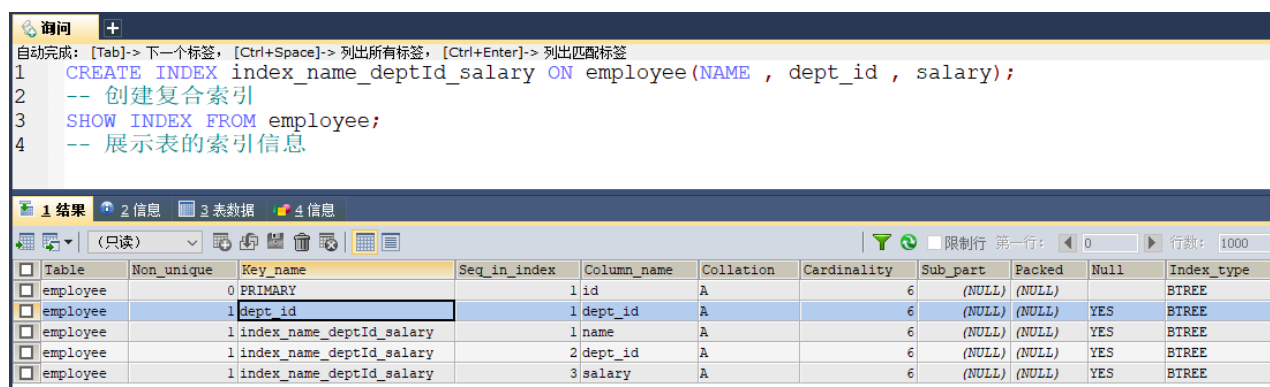
索引使用的关键是：查询尽量使用索引，避免索引失效。

### 全值匹配(最好)

- 建立复合索引(name,dept\_id,salary)

示例：

```
CREATE INDEX index_name_deptId_salary ON employee(NAME , dept_id , salary);
-- 创建复合索引
SHOW INDEX FROM employee;
-- 展示表的索引信息
```



自动完成: [Tab]-> 下一个标签, [Ctrl+Space]-> 列出所有标签, [Ctrl+Enter]-> 列出匹配标签

```
1 CREATE INDEX index_name_deptId_salary ON employee(NAME , dept_id , salary);
2 -- 创建复合索引
3 SHOW INDEX FROM employee;
4 -- 展示表的索引信息
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type
employee	0	PRIMARY	1	id	A	6	(NULL)	(NULL)		BTREE
employee	1	dept_id	1	dept_id	A	6	(NULL)	(NULL)	YES	BTREE
employee	1	index_name_deptId_salary	1	name	A	6	(NULL)	(NULL)	YES	BTREE
employee	1	index_name_deptId_salary	2	dept_id	A	6	(NULL)	(NULL)	YES	BTREE
employee	1	index_name_deptId_salary	3	salary	A	6	(NULL)	(NULL)	YES	BTREE

- 使用到了一个索引

```
EXPLAIN SELECT * FROM employee WHERE NAME = '白骨精'
```

- 使用到了2个索引

```
EXPLAIN SELECT * FROM employee WHERE NAME = '白骨精' AND dept_id= 2
```

- 使用用到了3个索引

```
EXPLAIN SELECT * FROM employee WHERE NAME = '白骨精' AND dept_id= 2 AND salary = 7200
```

## 最佳左前缀法则

如果索引有多列,要遵守最左前缀法则,指的就是从索引的最左列开始 并且不跳过索引中的列

- 跳过第一个,索引失效 (后续没有单独列索引的情况下)

```
EXPLAIN SELECT * FROM employee WHERE dept_id = 2 AND salary = 7200
```

- 跳过前两个,索引失效

```
EXPLAIN SELECT * FROM employee WHERE salary = 7200
```

- 跳过中间一个,只有第一个生效

```
EXPLAIN SELECT * FROM employee WHERE name = '白骨精' AND salary = 7200
```

- 顺序可以乱

```
EXPLAIN SELECT * FROM employee WHERE dept_id = 2 AND salary = 7200 and name='白骨精'
```

## 不在索引列上做任何操作

- 计算,函数,类型转换会导致索引失效而转向全表扫描
- 示例 正常状态

```
EXPLAIN SELECT * FROM employee WHERE NAME='白骨精'
```

- 示例 添加了运算

```
EXPLAIN SELECT * FROM employee WHERE trim(NAME)='白骨精'
```

## 范围条件右边的索引失效

- 全部使用

```
EXPLAIN SELECT * FROM employee WHERE NAME = '白骨精' AND dept_id= 2 AND salary = 7200
```

- 使用了范围

```
EXPLAIN SELECT * FROM employee WHERE NAME = '白骨精' AND dept_id > 2 AND salary = 7200
```

## !=或者<>及is not null

mysql在使用不等于(!=或者<>)的时候无法使用索引会导致全表扫描

- 等于

```
EXPLAIN SELECT * FROM employee WHERE NAME = '白骨精'
```

- 不等于

```
EXPLAIN SELECT * FROM employee WHERE NAME != '白骨精'  
EXPLAIN SELECT * FROM employee WHERE NAME <> '白骨精'
```

- is not null 无法使用索引

```
EXPLAIN SELECT * FROM employee WHERE NAME = is not null
```

## or引起索引失效

- 少用or 用or连接时, 会导致索引失效

```
EXPLAIN SELECT * FROM employee WHERE NAME = '白骨精' or dept_id = 1 or salary = 7200
```

## like引起索引失效

like以通配符开头(%)索引失效变成全表扫描

- 使用%开头,不会使用索引

```
EXPLAIN SELECT * FROM employee WHERE NAME like '%精'
```

- 使用%结尾, 可以使用索引

```
EXPLAIN SELECT * FROM employee WHERE NAME like '白%'
```

- 两边都使用%, 不会使用索引

```
EXPLAIN SELECT * FROM employee WHERE NAME like '%白%'
```

- 尽量使用覆盖索引 查询的字段和建立索引的字段刚好吻合,这种我们称为覆盖索引(覆盖索引后即可解决索引失效)

```
EXPLAIN SELECT name FROM employee WHERE NAME like '%白%'
```

## 字符串不加引号索引失效

```
EXPLAIN SELECT * FROM employee WHERE NAME = '110';
```

## 目标 大批量数据分页操作如何优化

- 传统查询分析

- 使用limit 随着offset增大, 查询的速度会越来越慢, 会把前面的数据都取出,找到对应位置。

```
SELECT * FROM logs1 LIMIT 0,3;  
SELECT * FROM logs1 LIMIT 9900000,3;
```

- 优化后分页查询

- 使用子查询优化方式1

```
select * from logs1 e inner join (SELECT id from logs1 limit 9900000 ,10 ) et on  
e.id = et.id 。
```

- 使用子查询优化方式2

```
select * from logs1 where id >=(SELECT id from logs1 limit 9900000 , 10) limit 10
```

- 使用 id 限定优化

- 记录上一页最大的id号 使用范围查询,限制是只能使用于明确知道id的情况, 不过一般建立表的时候, 都会添加基本的id字段, 这为分页查询带来很多便利

```
select * from logs1 where id between 9900000 and 9900010  
select * from orders_history where id > 1000000 limit 100;
```

- 项目设计优化

- 没有必要提供太多的分页查询。