基础篇

通用语法及分类

- DDL: 数据定义语言，用来定义数据库对象（数据库、表、字段）
- DML: 数据操作语言，用来对数据库表中的数据进行增删改
- DQL: 数据查询语言，用来查询数据库中表的记录
- DCL: 数据控制语言，用来创建数据库用户、控制数据库的控制权限

DDL（数据定义语言）

数据定义语言

数据库操作

查询所有数据库:
`SHOW DATABASES;`
查询当前数据库：
`SELECT DATABASE();`
创建数据库：
`CREATE DATABASE [ IF NOT EXISTS ] 数据库名 [ DEFAULT CHARSET 字符集] [COLLATE 排序规则 ];`
删除数据库:
`DROP DATABASE [ IF EXISTS ] 数据库名;`
使用数据库：
`USE 数据库名;`

注意事项

- UTF8字符集长度为3字节，有些符号占4字节，所以推荐用utf8mb4字符集

表操作

查询当前数据库所有表：
`SHOW TABLES;`
查询表结构：
`DESC 表名;`
查询指定表的建表语句：
`SHOW CREATE TABLE 表名;`

创建表：

```
CREATE TABLE 表名(
    字段1 字段1类型 [COMMENT 字段1注释],
    字段2 字段2类型 [COMMENT 字段2注释],
    字段3 字段3类型 [COMMENT 字段3注释],
    ...
    字段n 字段n类型 [COMMENT 字段n注释]
)[ COMMENT 表注释 ];
```

所有的要用英文的格式
最后一个字段后面没有逗号

添加字段：
`ALTER TABLE 表名 ADD 字段名 类型(长度) [COMMENT 注释] [约束];`
例：`ALTER TABLE emp ADD nickname varchar(20) COMMENT '昵称';`

修改数据类型：
`ALTER TABLE 表名 MODIFY 字段名 新数据类型(长度);`

修改字段名和字段类型：
`ALTER TABLE 表名 CHANGE 旧字段名 新字段名 类型(长度) [COMMENT 注释] [约束];`

例：将emp表的nickname字段修改为username，类型为varchar(30)
`ALTER TABLE emp CHANGE nickname username varchar(30) COMMENT '昵称';`

删除字段：
`ALTER TABLE 表名 DROP 字段名;`

修改表名：
`ALTER TABLE 表名 RENAME TO 新表名`

删除表：
`DROP TABLE [IF EXISTS] 表名;`

删除表，并重新创建该表：
`TRUNCATE TABLE 表名;`

DML（数据操作语言）

添加数据

指定字段：
`INSERT INTO 表名 (字段名1, 字段名2, ...) VALUES (值1, 值2, ...);`
全部字段：
`INSERT INTO 表名 VALUES (值1, 值2, ...);`

批量添加数据：
`INSERT INTO 表名 (字段名1, 字段名2, ...) VALUES (值1, 值2, ...), (值1, 值2, ...), (值1, 值2, ...);`
`INSERT INTO 表名 VALUES (值1, 值2, ...), (值1, 值2, ...), (值1, 值2, ...);`

注意事项

- 字符串和日期类型数据应该包含在引号中
- 插入的数据大小应该在字段的规定范围内

更新和删除数据

修改数据：
`UPDATE 表名 SET 字段名1 = 值1, 字段名2 = 值2, ... [ WHERE 条件 ];`
例：
`UPDATE emp SET name = 'Jack' WHERE id = 1;`

删除数据：
`DELETE FROM 表名 [ WHERE 条件 ];`

DQL（数据查询语言）

语法：

```
SELECT
    字段列表
FROM
    表名字段
WHERE
    条件列表
GROUP BY
    分组字段列表
HAVING
    分组后的条件列表
ORDER BY
    排序字段列表
LIMIT
    分页参数
```

基础查询

查询多个字段：

`SELECT 字段1, 字段2, 字段3, ... FROM 表名;`

`SELECT * FROM 表名;`

设置别名：

`SELECT 字段1 [ AS 别名1 ], 字段2 [ AS 别名2 ], 字段3 [ AS 别名3 ], ... FROM 表名;`

`SELECT 字段1 [ 别名1 ], 字段2 [ 别名2 ], 字段3 [ 别名3 ], ... FROM 表名;`

去除重复记录：

`SELECT DISTINCT 字段列表 FROM 表名;`

转义：

`SELECT * FROM 表名 WHERE name LIKE '/_张三' ESCAPE '/'`

/ 之后的_不作为通配符

条件查询

语法：

`SELECT 字段列表 FROM 表名 WHERE 条件列表;`

条件：

比较运算符	功能
>	大于
>=	大于等于
<	小于
<=	小于等于
=	等于

比较运算符	功能
<> 或 !=	不等于
BETWEEN ... AND ...	在某个范围内（含最小、最大值）
IN(...)	在in之后的列表中的值，多选一
LIKE 占位符	模糊匹配（_匹配单个字符，%匹配任意个字符）
IS NULL	是NULL

逻辑运算符	功能
AND 或 &&	并且（多个条件同时成立）
OR 或 ||	或者（多个条件任意一个成立）
NOT 或 !	非，不是

例子：

```
-- 年龄等于30
select * from employee where age = 30;
-- 年龄小于30
select * from employee where age < 30;
-- 小于等于
select * from employee where age <= 30;
-- 没有身份证
select * from employee where idcard is null or idcard = '';
-- 有身份证
select * from employee where idcard;
select * from employee where idcard is not null;
-- 不等于
select * from employee where age != 30;
-- 年龄在20到30之间
select * from employee where age between 20 and 30;
select * from employee where age >= 20 and age <= 30;
-- 下面语句不报错，但查不到任何信息
select * from employee where age between 30 and 20;
-- 性别为女且年龄小于30
select * from employee where age < 30 and gender = '女';
-- 年龄等于25或30或35
select * from employee where age = 25 or age = 30 or age = 35;
select * from employee where age in (25, 30, 35);
-- 姓名为两个字
select * from employee where name like '__';
-- 身份证最后为X
select * from employee where idcard like '%X';
```

聚合查询（聚合函数）

常见聚合函数：

函数	功能
count	统计数量
max	最大值
min	最小值
avg	平均值
sum	求和

语法：

`SELECT 聚合函数(字段列表) FROM 表名;`

例：

`SELECT count(id) from employee where workaddress = "广东省";`

分组查询

语法：

`SELECT 字段列表 FROM 表名 [ WHERE 条件 ] GROUP BY 分组字段名 [ HAVING 分组后的过滤条件 ];`

where 和 having 的区别：

- 执行时机不同：where是分组之前进行过滤，不满足where条件不参与分组；having是分组后对结果进行过滤。
- 判断条件不同：where不能对聚合函数进行判断，而having可以。

例子：

```
-- 根据性别分组，统计男性和女性数量（只显示分组数量，不显示哪个是男哪个是女）
select count(*) from employee group by gender;
-- 根据性别分组，统计男性和女性数量
select gender, count(*) from employee group by gender;
-- 根据性别分组，统计男性和女性的平均年龄
select gender, avg(age) from employee group by gender;
-- 年龄小于45，并根据工作地址分组
select workaddress, count(*) from employee where age < 45 group by workaddress;
-- 年龄小于45，并根据工作地址分组，获取员工数量大于等于3的工作地址
select workaddress, count(*) address_count from employee where age < 45 group by
workaddress having address_count >= 3;
```

注意事项

- 执行顺序：where > 聚合函数 > having
- 分组之后，查询的字段一般为聚合函数和分组字段，查询其他字段无任何意义

排序查询

语法：

`SELECT 字段列表 FROM 表名 ORDER BY 字段1 排序方式1, 字段2 排序方式2;`

排序方式：

- ASC: 升序（默认）
- DESC: 降序

例子：

```
-- 根据年龄升序排序
SELECT * FROM employee ORDER BY age ASC;
SELECT * FROM employee ORDER BY age;
-- 两字段排序，根据年龄升序排序，入职时间降序排序(如果年龄相同那么就按这个)
SELECT * FROM employee ORDER BY age ASC, entrydate DESC;
```

注意事项

如果是多字段排序，当第一个字段值相同时，才会根据第二个字段进行排序

分页查询

语法：

`SELECT 字段列表 FROM 表名 LIMIT 起始索引, 查询记录数;`

例子：

```
-- 查询第一页数据，展示10条
SELECT * FROM employee LIMIT 0, 10;
-- 查询第二页
SELECT * FROM employee LIMIT 10, 10;
```

注意事项

- 起始索引从0开始，起始索引 = （查询页码 - 1） * 每页显示记录数
- 分页查询是数据库的方言，不同数据库有不同实现，MySQL是LIMIT
- 如果查询的是第一页数据，起始索引可以省略，直接简写 LIMIT 10

DQL执行顺序

FROM -> WHERE -> GROUP BY -> SELECT -> ORDER BY -> LIMIT

DCL

管理用户

查询用户：

```
USER mysql;
SELECT * FROM user;
```

创建用户：

`CREATE USER '用户名'@'主机名' IDENTIFIED BY '密码';`

修改用户密码：

`ALTER USER '用户名'@'主机名' IDENTIFIED WITH mysql_native_password BY '新密码';`

删除用户：

`DROP USER '用户名'@'主机名';`

例子：

```
-- 创建用户test，只能在当前主机localhost访问
create user 'test'@'localhost' identified by '123456';
-- 创建用户test，能在任意主机访问
create user 'test'@'%' identified by '123456';
create user 'test' identified by '123456';
-- 修改密码
alter user 'test'@'localhost' identified with mysql_native_password by '1234';
-- 删除用户
drop user 'test'@'localhost';
```

注意事项

- 主机名可以使用 % 通配

权限控制

常用权限：

权限	说明
ALL, ALL PRIVILEGES	所有权限
SELECT	查询数据
INSERT	插入数据
UPDATE	修改数据
DELETE	删除数据
ALTER	修改表
DROP	删除数据库/表/视图
CREATE	创建数据库/表

更多权限请看权限一览表

查询权限：

`SHOW GRANTS FOR '用户名'@'主机名';`

授予权限：

`GRANT 权限列表 ON 数据库名.表名 TO '用户名'@'主机名';`

撤销权限：

`REVOKE 权限列表 ON 数据库名.表名 FROM '用户名'@'主机名';`

注意事项

- 多个权限用逗号分隔
- 授权时，数据库名和表名可以用 * 进行通配，代表所有

函数

函数 是指一段可以直接被另外一段程序调用的程序或代码。

- 字符串函数
- 数值函数
- 日期函数
- 流程函数

字符串函数

常用函数：

函数	功能
CONCAT(s1, s2, ..., sn)	字符串拼接，将s1, s2, ..., sn拼接成一个字符串
LOWER(str)	将字符串全部转为小写
UPPER(str)	将字符串全部转为大写
LPAD(str, n, pad)	左填充，用字符串pad对str的左边进行填充，达到n个字符串长度
RPAD(str, n, pad)	右填充，用字符串pad对str的右边进行填充，达到n个字符串长度
TRIM(str)	去掉字符串头部和尾部的空格
SUBSTRING(str, start, len)	返回从字符串str从start位置起的len个长度的字符串

使用示例：

```
-- 拼接
SELECT CONCAT('Hello', 'World');
-- 小写
SELECT LOWER('Hello');
-- 大写
SELECT UPPER('Hello');
-- 左填充
SELECT LPAD('01', 5, '-');
-- 右填充
SELECT RPAD('01', 5, '-');
-- 去除空格
SELECT TRIM(' Hello World ');
-- 切片（起始索引为1）
SELECT SUBSTRING('Hello World', 1, 5);
```

数值函数

常见函数：

函数	功能
CEIL(x)	向上取整
FLOOR(x)	向下取整
MOD(x, y)	返回x/y的模
RAND()	返回0~1内的随机数
ROUND(x, y)	求参数x的四舍五入值，保留y位小数

日期函数

常用函数：

函数	功能
CURDATE()	返回当前日期
CURTIME()	返回当前时间
NOW()	返回当前日期和时间
YEAR(date)	获取指定date的年份
MONTH(date)	获取指定date的月份
DAY(date)	获取指定date的日期
DATE_ADD(date, INTERVAL expr type)	返回一个日期/时间值加上一个时间间隔expr后的时间值
DATEDIFF(date1, date2)	返回起始时间date1和结束时间date2之间的天数

例子：

```
-- DATE_ADD
SELECT DATE_ADD(NOW(), INTERVAL 70 YEAR);
```

流程函数

常用函数：

函数	功能
IF(value, t, f)	如果value为true，则返回t，否则返回f
IFNULL(value1, value2)	如果value1不为空，返回value1，否则返回value2

函数	功能
CASE WHEN [val1] THEN [res1] ... ELSE [default] END	如果val1为true，返回res1，... 否则返回default默认值
CASE [expr] WHEN [val1] THEN [res1] ... ELSE [default] END	如果expr的值等于val1，返回res1，... 否则返回default默认值

例子：

```
select
    name,
    (case when age > 30 then '中年' else '青年' end)
from employee;
select
    name,
    (case workaddress when '北京市' then '一线城市' when '上海市' then '一线城市'
else '二线城市' end) as '工作地址'
from employee;
```

约束

1. 概念：约束是用来作用于表中字段上的规则，用于限制存储在表中的数据。
2. 目的：保证数据库中的数据的正确、有效性和完整性

分类：

约束	描述	关键字
非空约束	限制该字段的数据不能为null	NOT NULL
唯一约束	保证该字段的所有数据都是唯一、不重复的	UNIQUE
主键约束	主键是一行数据的唯一标识，要求非空且唯一	PRIMARY KEY
默认约束	保存数据时，如果未指定该字段的值，则采用默认值	DEFAULT
检查约束（8.0.1版本后）	保证字段值满足某一个条件	CHECK
外键约束	用来让两张图的数据之间建立连接，保证数据的一致性和完整性	FOREIGN KEY

约束是作用于表中字段上的，可以再创建表/修改表的时候添加约束。

常用约束

约束条件	关键字
主键	PRIMARY KEY
自动增长	AUTO_INCREMENT

约束条件	关键字
不为空	NOT NULL
唯一	UNIQUE
逻辑条件	CHECK
默认值	DEFAULT

例子：

```
create table user(
    id int primary key auto_increment,
    name varchar(10) not null unique,
    age int check(age > 0 and age < 120),
    status char(1) default '1',
    gender char(1)
);
```

外键约束

外键用来让两张表的数据之间建立连接，从而保证数据的一致性和完整性。

添加外键：

```
CREATE TABLE 表名(
    字段名 字段类型,
    ...
    [CONSTRAINT] [外键名称] FOREIGN KEY(外键字段名) REFERENCES 主表(主表列名)
);
ALTER TABLE 表名 ADD CONSTRAINT 外键名称 FOREIGN KEY (外键字段名) REFERENCES 主表(主表列名);

-- 例子
alter table emp add constraint fk_emp_dept_id foreign key(dept_id) references dept(id);
```

删除外键：

`ALTER TABLE 表名 DROP FOREIGN KEY 外键名;`

删除/更新行为

行为	说明
NO ACTION	当在父表中删除/更新对应记录时，首先检查该记录是否有对应外键，如果有则不允许删除/更新（与RESTRICT一致）
RESTRICT	当在父表中删除/更新对应记录时，首先检查该记录是否有对应外键，如果有则不允许删除/更新（与NO ACTION一致）
CASCADE	当在父表中删除/更新对应记录时，首先检查该记录是否有对应外键，如果有则也删除/更新外键在子表中的记录

行为	说明
SET NULL	当在父表中删除/更新对应记录时，首先检查该记录是否有对应外键，如果有则设置子表中该外键值为null（要求该外键允许为null）
SET DEFAULT	父表有变更时，子表将外键设为一个默认值（Innodb不支持）

更改删除/更新行为：
`ALTER TABLE 表名 ADD CONSTRAINT 外键名称 FOREIGN KEY (外键字段) REFERENCES 主表名(主表字段名) ON UPDATE 行为 ON DELETE 行为;`

多表查询

多表关系

- 一对多（多对一）
- 多对多
- 一对一

一对多

案例：部门与员工
关系：一个部门对应多个员工，一个员工对应一个部门
实现：在多的一方建立外键，指向一的一方的主键

多对多

案例：学生与课程
关系：一个学生可以选多门课程，一门课程也可以供多个学生选修
实现：建立第三张中间表，中间表至少包含两个外键，分别关联两方主键

一对一

案例：用户与用户详情
关系：一对一关系，多用于单表拆分，将一张表的基础字段放在一张表中，其他详情字段放在另一张表中，以提升操作效率
实现：在任意一方加入外键，关联另外一方的主键，并且设置外键为唯一的（UNIQUE）

查询

合并查询（笛卡尔积，会展示所有组合结果）：
`select * from employee, dept;`

> 笛卡尔积：两个集合A集合和B集合的所有组合情况（在多表查询时，需要消除无效的笛卡尔积）

消除无效笛卡尔积：
`select * from employee, dept where employee.dept = dept.id;`

内连接查询

内连接查询的是两张表交集的部分

隐式内连接：

`SELECT 字段列表 FROM 表1, 表2 WHERE 条件 ...;`

显式内连接：

`SELECT 字段列表 FROM 表1 [ INNER ] JOIN 表2 ON 连接条件 ...;`

显式性能比隐式高

例子：

```
-- 查询员工姓名，及关联的部门的名称
-- 隐式
select e.name, d.name from employee as e, dept as d where e.dept = d.id;
-- 显式
select e.name, d.name from employee as e inner join dept as d on e.dept = d.id;
```

外连接查询

左外连接：
查询左表所有数据，以及两张表交集部分数据

`SELECT 字段列表 FROM 表1 LEFT [ OUTER ] JOIN 表2 ON 条件 ...;`

相当于查询表1的所有数据，包含表1和表2交集部分数据

右外连接：
查询右表所有数据，以及两张表交集部分数据

`SELECT 字段列表 FROM 表1 RIGHT [ OUTER ] JOIN 表2 ON 条件 ...;`

例子：

```
-- 左
select e.*, d.name from employee as e left outer join dept as d on e.dept =
d.id;
select d.name, e.* from dept d left outer join emp e on e.dept = d.id;  -- 这条语
句与下面的语句效果一样
-- 右
select d.name, e.* from employee as e right outer join dept as d on e.dept =
d.id;
```

左连接可以查询到没有dept的employee，右连接可以查询到没有employee的dept

自连接查询

当前表与自身的连接查询，自连接必须使用表别名

语法：

`SELECT 字段列表 FROM 表A 别名A JOIN 表A 别名B ON 条件 ...;`

自连接查询，可以是内连接查询，也可以是外连接查询

例子：

```sql
-- 查询员工及其所属领导的名字
select a.name, b.name from employee a, employee b where a.manager = b.id;
-- 没有领导的也查询出来
select a.name, b.name from employee a left join employee b on a.manager = b.id;
```

联合查询 union, union all

把多次查询的结果合并，形成一个新的查询集

语法：

```sql
SELECT 字段列表 FROM 表A ...
UNION [ALL]
SELECT 字段列表 FROM 表B ...
```

注意事项

- UNION ALL 会有重复结果，UNION 不会
- 联合查询比使用or效率高，不会使索引失效

子查询

SQL语句中嵌套SELECT语句，称谓嵌套查询，又称子查询。
`SELECT * FROM t1 WHERE column1 = ( SELECT column1 FROM t2);`
子查询外部的语句可以是 INSERT / UPDATE / DELETE / SELECT 的任何一个

根据子查询结果可以分为：

- 标量子查询（子查询结果为单个值）
- 列子查询（子查询结果为一列）
- 行子查询（子查询结果为一行）
- 表子查询（子查询结果为多行多列）

根据子查询位置可分为：

- WHERE 之后
- FROM 之后
- SELECT 之后

标量子查询

子查询返回的结果是单个值（数字、字符串、日期等）。
常用操作符：- < > > >= < <=

例子：

```sql
-- 查询销售部所有员工
select id from dept where name = '销售部';
-- 根据销售部部门ID，查询员工信息
select * from employee where dept = 4;
-- 合并（子查询）
select * from employee where dept = (select id from dept where name = '销售部');


-- 查询xxx入职之后的员工信息
select * from employee where entrydate > (select entrydate from employee where
name = 'xxx');
```

列子查询

返回的结果是一列（可以是多行）。

常用操作符：

操作符	描述
IN	在指定的集合范围内，多选一
NOT IN	不在指定的集合范围内
ANY	子查询返回列表中，有任意一个满足即可
SOME	与ANY等同，使用SOME的地方都可以使用ANY
ALL	子查询返回列表的所有值都必须满足

例子：

```sql
-- 查询销售部和市场部的所有员工信息
select * from employee where dept in (select id from dept where name = '销售部'
or name = '市场部');
-- 查询比财务部所有人工资都高的员工信息
select * from employee where salary > all(select salary from employee where dept
= (select id from dept where name = '财务部'));
-- 查询比研发部任意一人工资高的员工信息
select * from employee where salary > any (select salary from employee where
dept = (select id from dept where name = '研发部'));
```

行子查询

返回的结果是一行（可以是多列）。
常用操作符：=, <, >, IN, NOT IN

例子：

```sql
-- 查询与xxx的薪资及直属领导相同的员工信息
select * from employee where (salary, manager) = (12500, 1);
select * from employee where (salary, manager) = (select salary, manager from
employee where name = 'xxx');
```

表子查询

返回的结果是多行多列
常用操作符：IN

例子：

```
-- 查询与xxx1，xxx2的职位和薪资相同的员工
select * from employee where (job, salary) in (select job, salary from employee
where name = 'xxx1' or name = 'xxx2');
-- 查询入职日期是2006-01-01之后的员工，及其部门信息
select e.*, d.* from (select * from employee where entrydate > '2006-01-01') as
e left join dept as d on e.dept = d.id;
```

事务

事务是一组操作的集合，事务会把所有操作作为一个整体一起向系统提交或撤销操作请求，即这些操作要么同时成功，要么同时失败。

基本操作：

```
-- 1. 查询张三账户余额
select * from account where name = '张三';
-- 2. 将张三账户余额-1000
update account set money = money - 1000 where name = '张三';
-- 此语句出错后张三钱减少但是李四钱没有增加
模拟sql语句错误
-- 3. 将李四账户余额+1000
update account set money = money + 1000 where name = '李四';

-- 查看事务提交方式
SELECT @@AUTOCOMMIT;
-- 设置事务提交方式，1为自动提交，0为手动提交，该设置只对当前会话有效
SET @@AUTOCOMMIT = 0;
-- 提交事务
COMMIT;
-- 回滚事务
ROLLBACK;

-- 设置手动提交后上面代码改为：
select * from account where name = '张三';
update account set money = money - 1000 where name = '张三';
update account set money = money + 1000 where name = '李四';
commit;
```

操作方式二：

开启事务：
`START TRANSACTION 或 BEGIN TRANSACTION;`
提交事务：
`COMMIT;`
回滚事务：
`ROLLBACK;`

操作实例：

```
start transaction;
select * from account where name = '张三';
update account set money = money - 1000 where name = '张三';
update account set money = money + 1000 where name = '李四';
commit;
```

开启事务后，只有手动提交才会改变数据库中的数据。

四大特性ACID

- 原子性(Atomicity)：事务是不可分割的最小操作但愿，要么全部成功，要么全部失败
- 一致性(Consistency)：事务完成时，必须使所有数据都保持一致状态
- 隔离性(Isolation)：数据库系统提供的隔离机制，保证事务在不受外部并发操作影响的独立环境下运行
- 持久性(Durability)：事务一旦提交或回滚，它对数据库中的数据的改变就是永久的

并发事务

问题	描述
脏读	一个事务读到另一个事务还没提交的数据
不可重复读	一个事务先后读取同一条记录，但两次读取的数据不同
幻读	一个事务按照条件查询数据时，没有对应的数据行，但是再插入数据时，又发现这行数据已经存在

> 这三个问题的详细演示：https://www.bilibili.com/video/BV1Kr4y1i7ru?p=55cd

并发事务隔离级别：

隔离级别	脏读	不可重复读	幻读
Read uncommitted	√	√	√
Read committed	×	√	√
Repeatable Read(默认)	×	×	√
Serializable	×	×	×

- √表示在当前隔离级别下该问题会出现
- Serializable 性能最低；Read uncommitted 性能最高，数据安全性最差

查看事务隔离级别：
`SELECT @@TRANSACTION_ISOLATION;`
设置事务隔离级别：
`SET [ SESSION | GLOBAL ] TRANSACTION ISOLATION LEVEL {READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE };`
SESSION 是会话级别，表示只针对当前会话有效，GLOBAL 表示对所有会话有效

进阶篇

存储引擎

MySQL体系结构：
![层级描述](https://dhc.pythonanywhere.com/media/editor/MySQL体系结构层级含义_20220315034359342837.png "层级描述"

存储引擎就是存储数据、建立索引、更新/查询数据等技术的实现方式。存储引擎是基于表而不是基于库的，所以存储引擎也可以被称为表引擎。
默认存储引擎是InnoDB。

相关操作：

```
-- 查询建表语句
show create table account;
-- 建表时指定存储引擎
CREATE TABLE 表名(
    ...
) ENGINE=INNODB;
-- 查看当前数据库支持的存储引擎
show engines;
```

InnoDB

InnoDB 是一种兼顾高可靠性和高性能的通用存储引擎，在 MySQL 5.5 之后，InnoDB 是默认的 MySQL 引擎。

特点：

- DML 操作遵循 ACID 模型，支持**事务**
- **行级锁**，提高并发访问性能
- 支持**外键**约束，保证数据的完整性和正确性

文件：

- xxx.ibd: xxx代表表名，InnoDB 引擎的每张表都会对应这样一个表空间文件，存储该表的表结构（frm、sdi）、数据和索引。

参数：innodb_file_per_table，决定多张表共享一个表空间还是每张表对应一个表空间

知识点：

查看 Mysql 变量：
`show variables like 'innodb_file_per_table';`

从idb文件提取表结构数据：
（在cmd运行）
`ibd2sdi xxx.ibd`

InnoDB 逻辑存储结构：

MyISAM

MyISAM 是 MySQL 早期的默认存储引擎。

特点：

- 不支持事务，不支持外键
- 支持表锁，不支持行锁
- 访问速度快

文件：

- xxx.sdi: 存储表结构信息
- xxx.MYD: 存储数据
- xxx.MYI: 存储索引

Memory

Memory 引擎的表数据是存储在内存中的，受硬件问题、断电问题的影响，只能将这些表作为临时表或缓存使用。

特点：

- 存放在内存中，速度快
- hash索引（默认）

文件：

- xxx.sdi: 存储表结构信息

存储引擎特点

特点	InnoDB	MyISAM	Memory
存储限制	64TB	有	有
事务安全	支持	-	-
锁机制	行锁	表锁	表锁
B+tree索引	支持	支持	支持
Hash索引	-	-	支持
全文索引	支持（5.6版本之后）	支持	-
空间使用	高	低	N/A
内存使用	高	低	中等
批量插入速度	低	高	高
支持外键	支持	-	-

存储引擎的选择

在选择存储引擎时，应该根据应用系统的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。

- InnoDB: 如果应用对事物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之外，还包含很多的更新、删除操作，则 InnoDB 是比较合适的选择
- MyISAM: 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、并发性要求不高，那这个存储引擎是非常合适的。
- Memory: 将所有数据保存在内存中，访问速度快，通常用于临时表及缓存。Memory 的缺陷是对表的大小有限制，太大的表无法缓存在内存中，而且无法保障数据的安全性

电商中的足迹和评论适合使用 MyISAM 引擎，缓存适合使用 Memory 引擎。

性能分析

查看执行频次

查看当前数据库的 INSERT, UPDATE, DELETE, SELECT 访问频次：
`SHOW GLOBAL STATUS LIKE 'Com_____';` 或者 `SHOW SESSION STATUS LIKE 'Com_____';`
例：`show global status like 'Com_____'`

慢查询日志

慢查询日志记录了所有执行时间超过指定参数（long_query_time，单位：秒，默认10秒）的所有SQL语句的日志。
MySQL的慢查询日志默认没有开启，需要在MySQL的配置文件（/etc/my.cnf）中配置如下信息：

```
# 开启慢查询日志开关
slow_query_log=1
# 设置慢查询日志的时间为2秒，SQL语句执行时间超过2秒，就会视为慢查询，记录慢查询日志
long_query_time=2
```

更改后记得重启MySQL服务，日志文件位置：/var/lib/mysql/localhost-slow.log

查看慢查询日志开关状态：
`show variables like 'slow_query_log';`

profile

show profile 能在做SQL优化时帮我们了解时间都耗费在哪里。通过 have_profiling 参数，能看到当前 MySQL 是否支持 profile 操作：
`SELECT @@have_profiling;`
profiling 默认关闭，可以通过set语句在session/global级别开启 profiling：
`SET profiling = 1;`
查看所有语句的耗时：
`show profiles;`
查看指定query_id的SQL语句各个阶段的耗时：
`show profile for query query_id;`
查看指定query_id的SQL语句CPU的使用情况
`show profile cpu for query query_id;`

explain

EXPLAIN 或者 DESC 命令获取 MySQL 如何执行 SELECT 语句的信息，包括在 SELECT 语句执行过程中表如何连接和连接的顺序。
语法：

```
# 直接在select语句之前加上关键字 explain / desc
EXPLAIN SELECT 字段列表 FROM 表名 HWERE 条件;
```

EXPLAIN 各字段含义：

- id：select 查询的序列号，表示查询中执行 select 子句或者操作表的顺序（id相同，执行顺序从上到下；id不同，值越大越先执行）
- select_type：表示 SELECT 的类型，常见取值有 SIMPLE（简单表，即不适用表连接或者子查询）、PRIMARY（主查询，即外层的查询）、UNION（UNION中的第二个或者后面的查询语句）、 SUBQUERY（SELECT/WHERE之后包含了子查询）等
- type：表示连接类型，性能由好到差的连接类型为 NULL、system、const、eq_ref、ref、range、index、all
- possible_key：可能应用在这张表上的索引，一个或多个
- Key：实际使用的索引，如果为 NULL，则没有使用索引
- Key_len：表示索引中使用的字节数，该值为索引字段最大可能长度，并非实际使用长度，在不损失精确性的前提下，长度越短越好
- rows：MySQL认为必须要执行的行数，在InnoDB引擎的表中，是一个估计值，可能并不总是准确的
- filtered：表示返回结果的行数占需读取行数的百分比，filtered的值越大越好

索引

索引是帮助 MySQL **高效获取数据**的**数据结构（有序）**。在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查询算法，这种数据结构就是索引。

优缺点：

优点：

- 提高数据检索效率，降低数据库的IO成本
- 通过索引列对数据进行排序，降低数据排序的成本，降低CPU的消耗

缺点：

- 索引列也是要占用空间的
- 索引大大提高了查询效率，但降低了更新的速度，比如 INSERT、UPDATE、DELETE

索引结构

索引结构	描述
B+Tree	最常见的索引类型，大部分引擎都支持B+树索引
Hash	底层数据结构是用哈希表实现，只有精确匹配索引列的查询才有效，不支持范围查询
R-Tree(空间索引)	空间索引是 MyISAM 引擎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

索引结构	描述
Full-Text(全文索引)	是一种通过建立倒排索引，快速匹配文档的方式，类似于 Lucene, Solr, ES

索引	InnoDB	MyISAM	Memory
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	不支持	支持	不支持
Full-text	5.6版本后支持	支持	不支持

B-Tree

二叉树形成链表的缺点可以用红黑树来解决：
红黑树也存在大数据量情况下，层级较深，检索速度慢的问题。

为了解决上述问题，可以使用 B-Tree 结构。
B-Tree (多路平衡查找树) 以一棵最大度数（max-degree，指一个节点的子节点个数）为5（5阶）的 b-tree 为例（每个节点最多存储4个key，5个指针）

> B-Tree 的数据插入过程动画参照：https://www.bilibili.com/video/BV1Kr4y1i7ru?p=68
> 演示地址：https://www.cs.usfca.edu/~galles/visualization/BTree.html

B+Tree

结构图：

> 演示地址：https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

与 B-Tree 的区别：

- 所有的数据都会出现在叶子节点
- 叶子节点形成一个单向链表

MySQL 索引数据结构对经典的 B+Tree 进行了优化。在原 B+Tree 的基础上，增加一个指向相邻叶子节点的链表指针，就形成了带有顺序指针的 B+Tree，提高区间访问的性能。

Hash

哈希索引就是采用一定的hash算法，将键值换算成新的hash值，映射到对应的槽位上，然后存储在hash表中。
如果两个（或多个）键值，映射到一个相同的槽位上，他们就产生了hash冲突（也称为hash碰撞），可以通过链表来解决。

特点：

- Hash索引只能用于对等比较（=、in），不支持范围查询（betwwn、>、<、...）
- 无法利用索引完成排序操作
- 查询效率高，通常只需要一次检索就可以了，效率通常要高于 B+Tree 索引

存储引擎支持：

- Memory

- InnoDB: 具有自适应hash功能，hash索引是存储引擎根据 B+Tree 索引在指定条件下自动构建的

面试题

1. 为什么 InnoDB 存储引擎选择使用 B+Tree 索引结构?

- 相对于二叉树，层级更少，搜索效率高
- 对于 B-Tree，无论是叶子节点还是非叶子节点，都会保存数据，这样导致一页中存储的键值减少，指针也跟着减少，要同样保存大量数据，只能增加树的高度，导致性能降低
- 相对于 Hash 索引，B+Tree 支持范围匹配及排序操作

索引分类

分类	含义	特点	关键字
主键索引	针对于表中主键创建的索引	默认自动创建，只能有一个	PRIMARY
唯一索引	避免同一个表中某数据列中的值重复	可以有多个	UNIQUE
常规索引	快速定位特定数据	可以有多个	
全文索引	全文索引查找的是文本中的关键词，而不是比较索引中的值	可以有多个	FULLTEXT

在 InnoDB 存储引擎中，根据索引的存储形式，又可以分为以下两种：

分类	含义	特点
聚集索引(Clustered Index)	将数据存储与索引放一块，索引结构的叶子节点保存了行数据	必须有，而且只有一个
二级索引(Secondary Index)	将数据与索引分开存储，索引结构的叶子节点关联的是对应的主键	可以存在多个

演示图：


聚集索引选取规则：

- 如果存在主键，主键索引就是聚集索引
- 如果不存在主键，将使用第一个唯一(UNIQUE)索引作为聚集索引
- 如果表没有主键或没有合适的唯一索引，则 InnoDB 会自动生成一个 rowid 作为隐藏的聚集索引

思考题

1. 以下 SQL 语句，哪个执行效率高？为什么？

```
select * from user where id = 10;
select * from user where name = 'Arm';
-- 备注：id为主键，name字段创建的有索引
```

答：第一条语句，因为第二条需要回表查询，相当于两个步骤。

2. InnoDB 主键索引的 B+Tree 高度为多少?

答：假设一行数据大小为1k，一页中可以存储16行这样的数据。InnoDB 的指针占用6个字节的空间，主键假设为bigint，占用字节数为8.
可得公式：`n * 8 + (n + 1) * 6 = 16 * 1024`，其中 8 表示 bigint 占用的字节数，n 表示当前节点存储的key的数量，(n + 1) 表示指针数量（比key多一个）。算出n约为1170。

如果树的高度为2，那么他能存储的数据量大概为：`1171 * 16 = 18736`；
如果树的高度为3，那么他能存储的数据量大概为：`1171 * 1171 * 16 = 21939856`。

另外，如果有成千上万的数据，那么就要考虑分表，涉及运维篇知识。

语法

创建索引：
`CREATE [ UNIQUE | FULLTEXT ] INDEX index_name ON table_name (index_col_name, ...);`
如果不加 CREATE 后面不加索引类型参数，则创建的是常规索引

查看索引：
`SHOW INDEX FROM table_name;`

删除索引：
`DROP INDEX index_name ON table_name;`

案例：

```
-- name字段为姓名字段，该字段的值可能会重复，为该字段创建索引
create index idx_user_name on tb_user(name);
-- phone手机号字段的值非空，且唯一，为该字段创建唯一索引
create unique index idx_user_phone on tb_user (phone);
-- 为profession, age, status创建联合索引
create index idx_user_pro_age_stat on tb_user(profession, age, status);
-- 为email建立合适的索引来提升查询效率
create index idx_user_email on tb_user(email);

-- 删除索引
drop index idx_user_email on tb_user;
```

使用规则

最左前缀法则

如果索引关联了多列（联合索引），要遵守最左前缀法则，最左前缀法则指的是查询从索引的最左列开始，并且不跳过索引中的列。
如果跳跃某一列，索引将部分失效（后面的字段索引失效）。跳过的话，后面的排序就无从说起了。最左前缀法则在用select的时候，和放的位置是没有关系的，只要存在就行。

联合索引中，出现范围查询（<, >），范围查询右侧的列索引失效。可以用>=或者<=来规避索引失效问题。

索引失效情况

1. 在索引列上进行运算操作，索引将失效。如：`explain select * from tb_user where substring(phone, 10, 2) = '15';` 换成 `explain select * from tb_user where phone = '17799990015';` 这是可以的。
2. 字符串类型字段使用时，不加引号，索引将失效。如：`explain select * from tb_user where phone = 17799990015;`，此处phone的值没有加引号
3. 模糊查询中，如果仅仅是尾部模糊匹配，索引不会是失效；如果是头部模糊匹配，索引失效。如：`explain select * from tb_user where profession like '%工程';`，前后都有 % 也会失效。`explain select * from tb_user where profession like '软件%';` 这个是不会失效的，只有前面加了%才会失效。
4. 用 or 分割开的条件，如果 or 其中一个条件的列没有索引，那么涉及的索引都不会被用到。
5. 如果 MySQL 评估使用索引比全表更慢，则不使用索引。因为只要有一个没有索引，另外一个用不用索引都没有意义，都要进行全表扫描。所以就无需用索引。

SQL 提示

是优化数据库的一个重要手段，简单来说，就是在SQL语句中加入一些人为的提示来达到优化操作的目的。

例如，使用索引：
`explain select * from tb_user use index(idx_user_pro) where profession="软件工程";`
不使用哪个索引：
`explain select * from tb_user ignore index(idx_user_pro) where profession="软件工程";`
必须使用哪个索引：
`explain select * from tb_user force index(idx_user_pro) where profession="软件工程";`

use 是建议，实际使用哪个索引 MySQL 还会自己权衡运行速度去更改，force就是无论如何都强制使用该索引。

覆盖索引&回表查询

尽量使用覆盖索引（查询使用了索引，并且需要返回的列，在该索引中已经全部能找到），减少 select *。

explain 中 extra 字段含义：
`using index condition`：查找使用了索引，但是需要回表查询数据
`using where; using index;`：查找使用了索引，但是需要的数据都在索引列中能找到，所以不需要回表查询

覆盖索引：
如果在生成的二级索引（辅助索引）中可以一次性获得select所需要的字段，不需要回表查询。

如果在聚集索引中直接能找到对应的行，则直接返回行数据，只需要一次查询，哪怕是select *；
如果在辅助索引（二级索引）中找聚集索引，如`select id, name from xxx where name='xxx';`，也只需要通过辅助索引(name)查找到对应的id，返回name和name索引对应的id即可，只需要一次查询；
如果是通过辅助索引查找其他字段，则需要回表查询，如`select id, name, gender from xxx where name='xxx';`

所以尽量不要用`select *`，容易出现回表查询，降低效率，除非有联合索引包含了所有字段

面试题：一张表，有四个字段（id, username, password, status），由于数据量大，需要对以下SQL语句进行优化，该如何进行才是最优方案：

```
select id, username, password from tb_user where username='itcast';
```

解：给username和password字段建立联合索引，则不需要回表查询，直接覆盖索引。
username和password字段建立联合索引的叶子节点挂的就是 id 所以不需要三者同时建索引。

前缀索引

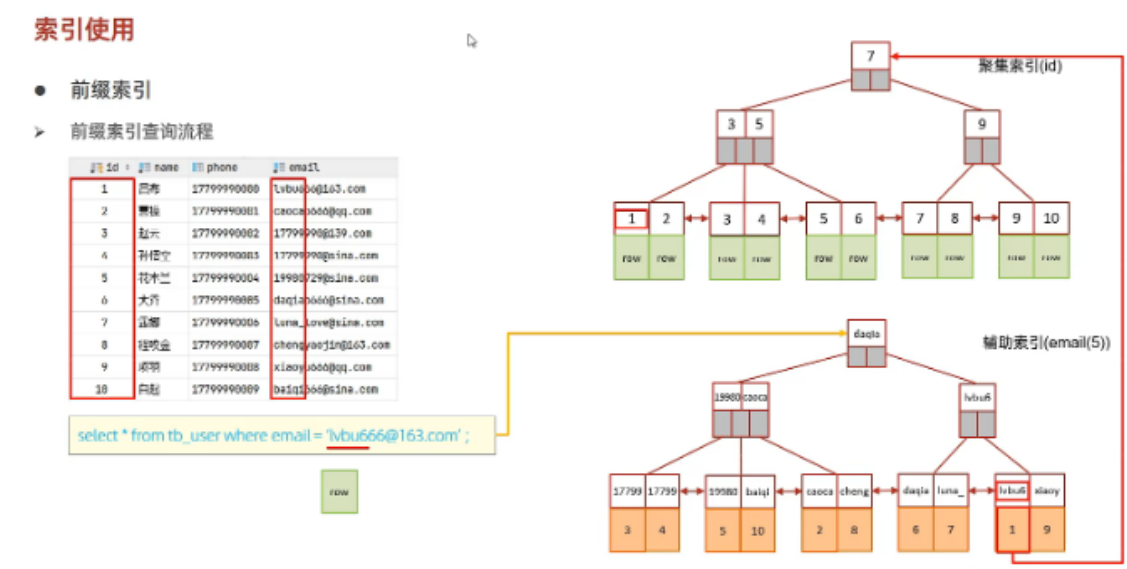
当字段类型为字符串（varchar, text等）时，有时候需要索引很长的字符串，这会让索引变得很大，查询时，浪费大量的磁盘IO，影响查询效率，此时可以只降字符串的一部分前缀，建立索引，这样可以大大节约索引空间，从而提高索引效率。

语法：`create index idx_xxxx on table_name(columnn(n));`
前缀长度：可以根据索引的选择性来决定，而选择性是指不重复的索引值（基数）和数据表的记录总数的比值，索引选择性越高则查询效率越高，唯一索引的选择性是1，这是最好的索引选择性，性能也是最好的。

求选择性公式：

```
select count(distinct email) / count(*) from tb_user;
select count(distinct substring(email, 1, 5)) / count(*) from tb_user;
```

前缀索引中是有可能碰到相同的索引的情况的（因为选择性可能不为1），所以使用前缀索引进行查询的时候，mysql 会有一个回表查询的过程，确定是否为所需数据。如图中的查询到lvbu6之后还要进行回表，回表完再查xiaoy，看到xiaoy是不需要的数据，则停止查下一个。



show index 里面的sub_part可以看到接取的长度

单列索引&联合索引

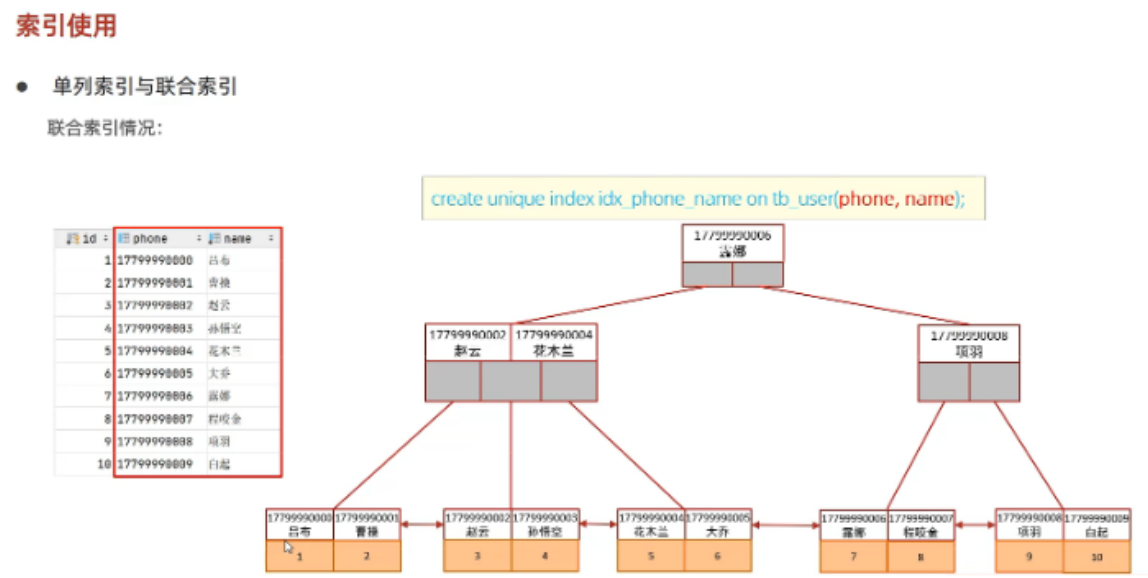
单列索引：即一个索引只包含单个列
联合索引：即一个索引包含了多个列
在业务场景中，如果存在多个查询条件，考虑针对于查询字段建立索引时，建议建立联合索引，而非单列索引。

单列索引情况：

`explain select id, phone, name from tb_user where phone = '17799990010' and name = '韩信';`

phone 和 name 都建立了索引情况下，这句只会用到phone索引字段。

联合索引的数据组织图：



注意事项

- 多条件联合查询时，MySQL优化器会评估哪个字段的索引效率更高，会选择该索引完成本次查询。

设计原则

1. 针对于数据量较大，且查询比较频繁的表建立索引
2. 针对于常作为查询条件（where）、排序（order by）、分组（group by）操作的字段建立索引
3. 尽量选择区分度高的列作为索引，尽量建立唯一索引，区分度越高，使用索引的效率越高
4. 如果是字符串类型的字段，字段长度较长，可以针对于字段的特点，建立前缀索引
5. 尽量使用联合索引，减少单列索引，查询时，联合索引很多时候可以覆盖索引，节省存储空间，避免回表，提高查询效率
6. 要控制索引的数量，索引并不是多多益善，索引越多，维护索引结构的代价就越大，会影响增删改的效率
7. 如果索引列不能存储NULL值，请在创建表时使用NOT NULL约束它。当优化器知道每列是否包含NULL值时，它可以更好地确定哪个索引最有效地用于查询

SQL 优化

插入数据

普通插入：

1. 采用批量插入（一次插入的数据不建议超过1000条，500 - 1000 为宜）
2. 手动提交事务
3. 主键顺序插入（主键顺序插入的效率大于乱序插入）

大批量插入：
如果一次性需要插入大批量数据，使用insert语句插入性能较低，此时可以使用MySQL数据库提供的load指令插入。

```
# 客户端连接服务端时，加上参数 --local-infile（这一行在bash/cmd界面输入）
mysql --local-infile -u root -p
# 设置全局参数local_infile为1，开启从本地加载文件导入数据的开关
set global local_infile = 1;
select @@local_infile;
# 执行load指令将准备好的数据，加载到表结构中，先要把表建立起来。
load data local infile '/root/sql1.log' into table 'tb_user' fields terminated
by ',' lines terminated by '\n';
```

主键优化

数据组织方式：在InnoDB存储引擎中，表数据都是根据主键顺序组织存放的，这种存储方式的表称为索引组织表（Index organized table, IOT）

主键的顺序的插入过程如下：

但是如果主键是乱序插入的话，就会导致需要插入的位置为中间的位置，会有页分裂的过程。

页分裂：页可以为空，也可以填充一般，也可以填充100%，每个页包含了2-N行数据（如果一行数据过大，会行溢出），根据主键排列。

页合并：当删除一行记录时，实际上记录并没有被物理删除，只是记录被标记（flaged）为删除并且它的空间变得允许被其他记录声明使用。当页中删除的记录到达 MERGE_THRESHOLD（默认为页的50%），InnoDB会开始寻找最靠近的页（前后）看看是否可以将这两个页合并以优化空间使用。

MERGE_THRESHOLD：合并页的阈值，可以自己设置，在创建表或创建索引时指定

> 文字说明不够清晰明了，具体可以看视频里的PPT演示过程：https://www.bilibili.com/video/BV1Kr4y1i7ru?p=90

主键设计原则：

- 满足业务需求的情况下，尽量降低主键的长度，二级索引的叶子节点保存的就是主键，所以主键小占用的空间也就会少。
- 插入数据时，尽量选择顺序插入，选择使用 AUTO_INCREMENT 自增主键
- 尽量不要使用 UUID 做主键或者是其他的自然主键，如身份证号，占用的空间大。
- 业务操作时，避免对主键的修改

order by优化

1. Using filesort：通过表的索引或全表扫描，读取满足条件的数据行，然后在排序缓冲区 sort buffer 中完成排序操作，所有不是通过索引直接返回排序结果的排序都叫 FileSort 排序
2. Using index：通过有序索引顺序扫描直接返回有序数据，这种情况即为 using index，不需要额外排序，操作效率高

如果order by字段全部使用升序排序或者降序排序，则都会走索引，但是如果一个字段升序排序，另一个字段降序排序，则不会走索引，explain的extra信息显示的是`Using index, Using filesort`，如果要优化掉Using filesort，则需要另外再创建一个索引，如：`create index idx_user_age_phone_ad on tb_user(age asc, phone desc);`，此时使用`select id, age,`

`phone from tb_user order by age asc, phone desc;`会全部走索引

总结：

- 根据排序字段建立合适的索引，多字段排序时，也遵循最左前缀法则
- 尽量使用覆盖索引
- 多字段排序，一个升序一个降序，此时需要注意联合索引在创建时的规则（ASC/DESC）
- 如果不可避免出现filesort，大数据量排序时，可以适当增大排序缓冲区大小 sort_buffer_size（默认256k）

group by优化

- 在分组操作时，可以通过索引来提高效率
- 分组操作时，索引的使用也是满足最左前缀法则的

如索引为`idx_user_pro_age_stat`，则句式可以是`select ... where profession order by age`，这样也符合最左前缀法则

limit优化

常见的问题如`limit 2000000, 10`，此时需要 MySQL 排序前2000000条记录，但仅仅返回2000000 - 2000010的记录，其他记录丢弃，查询排序的代价非常大。
优化方案：一般分页查询时，通过创建覆盖索引能够比较好地提高性能，可以通过覆盖索引加子查询形式进行优化

例如：

```
-- 此语句耗时很长
select * from tb_sku limit 9000000, 10;
-- 通过覆盖索引加快速度，直接通过主键索引进行排序及查询
select id from tb_sku order by id limit 9000000, 10;
-- 下面的语句是错误的，因为 MySQL 不支持 in 里面使用 limit
-- select * from tb_sku where id in (select id from tb_sku order by id limit
9000000, 10);
-- 通过连表查询即可实现第一句的效果，并且能达到第二句的速度
select * from tb_sku as s, (select id from tb_sku order by id limit 9000000, 10)
as a where s.id = a.id;
```

count优化

MyISAM 引擎把一个表的总行数存在了磁盘上，因此执行 count(*) 的时候会直接返回这个数，效率很高（前提是不适用where）；
InnoDB 在执行 count(*) 时，需要把数据一行一行地从引擎里面读出来，然后累计计数。
优化方案：自己计数，如创建key-value表存储在内存或硬盘，或者是用redis

count的几种用法：

- 如果count函数的参数（count里面写的那个字段）不是NULL（字段值不为NULL），累计值就加一，最后返回累计值
- 用法：count(*)、count(主键)、count(字段)、count(1)
- count(主键)跟count(*)一样，因为主键不能为空；count(字段)只计算字段值不为NULL的行；count(1)引擎会为每行添加一个1，然后就count这个1，返回结果也跟count(*)一样；count(null)返回0

各种用法的性能：

- count(主键)：InnoDB引擎会遍历整张表，把每行的主键id值都取出来，返回给服务层，服务层拿到主键后，直接按行进行累加（主键不可能为空）
- count(字段)：没有not null约束的话，InnoDB引擎会遍历整张表把每一行的字段值都取出来，返回给服务层，服务层判断是否为null，不为null，计数累加；有not null约束的话，InnoDB引擎会遍历整张表把每一行的字段值都取出来，返回给服务层，直接按行进行累加
- count(1)：InnoDB 引擎遍历整张表，但不取值。服务层对于返回的每一层，放一个数字 1 进去，直接按行进行累加
- count(*)：InnoDB 引擎并不会把全部字段取出来，而是专门做了优化，不取值，服务层直接按行进行累加

按效率排序：count(字段) < count(主键) < count(1) < count(*)，所以尽量使用 count(*)

update优化（避免行锁升级为表锁）

InnoDB 的行锁是针对索引加的锁，不是针对记录加的锁，并且该索引不能失效，否则会从行锁升级为表锁。

如以下两条语句：
`update student set no = '123' where id = 1;`，这句由于id有主键索引，所以只会锁这一行；
`update student set no = '123' where name = 'test';`，这句由于name没有索引，所以会把整张表都锁住进行数据更新，解决方法是给name字段添加索引，就可以由表锁变成行锁。

视图

视图（View）是一种虚拟存在的表。视图中的数据并不在数据库中实际存在，行和列数据来自定义视图的查询中使用的表，并且是在使用视图时动态生成的。
通俗的讲，视图只保存了查询的SQL逻辑，不保存查询结果。所以我们在创建视图的时候，主要的工作就落在创建这条SQL查询语句上。

创建视图

`CREATE [ OR REPLACE ] VIEW 视图名称[（列名列表）] AS SELECT 语句 [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]`

> 例子：`create or replace view stu_wll as select id,name from student where id<=10;`

查询视图

查看创建视图语句：`SHOW CREATE VIEW` 视图名称；

查看视图数据：`SELECT*FROM` 视图名称；
`show create view stu_v_1;`

修改视图

方式一：`CREATE[OR REPLACE] VIEW 视图名称[（列名列表)）] AS SELECT 语句[ WITH[ CASCADED | LOCAL ] CHECK OPTION ]`

方式二：`ALTER VIEW 视图名称 [（列名列表)] AS SELECT语句 [WITH [CASCADED | LOCAL] CHECK OPTION]`

删除视图

`DROP VIEW [IF EXISTS] 视图名称 [视图名称]`

视图检查选项

当使用WITH CHECK QPTION子句创建视图时，MySQL会通过视图检查正在更改的每个行，例如插入，更新，删除，以使其符合视图的定义。MySQL允许基于另一个视图创建视图，它还会检查依赖视图中的规则以保持一致性。为了确定检查的范围，mysql提供了两个选项：CASCADED 和 LOCAL ，默认值为CASCADED。

NOTE：如果没有开检查选项就不会进行检查。不同版本是不同含义的，要看版本。

CASCADED

`级联，一旦选择了这个选项，除了会检查创建视图时候的条件，还会检查所依赖视图的条件。`

> 比如下面的例子：创建stu_V_l 视图，id是小于等于 20的。
> `create or replace view stu_V_l as select id,name from student where id <=20;`
> 再创建 stu_v_2 视图，20 >= id >=10。
> `create or replace view stu_v_2 as select id,name from stu_v_1 where id >=10 with cascaded check option;`
> 再创建 stu_v_3 视图。
> `create or replace view stu_v_3 as select id,name from stu_v_2 where id<=15;`
> 这条数据能够成功，stu_v_3 没有开检查选项所以不会 去判断 id 是否小于等于15, 直接检查 是否满足 stu_v_2。
> `insert into stu_v_3 values(17,'Tom');`

LOCAL

本地的条件也会检查，还会向上检查。在向上找的时候，就要看是否上面开了检查选项，如果没开就不检查。和 CASCADED 的区别就是 CASCADED 不管上面开没开检查选项都会进行检查。

更新及作用

要使视图可更新，视图中的行与基础表中的行之间必须存在一对一的关系。如果视图包含以下任何一项，则该视图不可更新

1. 聚合函数或窗口函数 (SUM()、MIN()、MAX()、COUNT() 等)
2. DISTINCT
3. GROUP BY
4. HAVING
5. UNION 或者UNION ALL

> 例子： 使用了聚合函数，插入会失败。
> `create view stu_v_count as select count(*) from student;`
> `insert into stu_v_count values(10);`

作用

视图不仅可以简化用户对数据的理解，也可以简化他们的操作。那些被经常使用的查询可以被定义为视图，从而使得用户不必为以后的操作每次指定全部的条件。

> 安全
> 数据库可以授权，但不能授权到数据库特定行和特定的列上。通过视图用户只能查询和修改他们所能见到的数据
> 数据独立
> 视图可帮助用户屏蔽真实表结构变化带来的影响。

总而言之 类似于给表加上了一个外壳，通过这个外壳访问表的时候，只能按照所设计的方式进行访问与更新。

存储过程

存储过程是事先经过编译并存储在数据库中的一段SQL 语句的集合，调用存储过程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。
存储过程思想上很简单，就是数据库SQL 语言层面的代码封装与重用。

特点

1. 封装
2. 复用
3. 可以接收参数，也可以返回数据减少网络交互，效率提升

创建

```
CREATE PROCEDURE 存储过程名称( [参数列表] )

BEGIN

    SQL 语句

END;
```

NOTE: 在命令行中，执行创建存储过程的SQL时，需要通过关键字delimiter 指定SQL语句的结束符。默认是 分号作为结束符。

delimiter $ ，则 $ 符作为结束符。

调用

CALL 名称 ([参数])

查看

查询指定数据库的存储过程及状态信息

`SELECT* FROM INFORMATION_SCHEMA.ROUTINES WHERE ROUTINE_SCHEMA = 'xxx'`

存储过程名称；--查询某个存储过程的定义

`SHOW CREATE PROCEDURE`

删除

`DROP PROCEDURE [ IFEXISTS ] 存储过程名称`

游标

游标（CURSOR）是用来存储查询结果集的数据类型，在存储过程和函数中可以使用游标对结果集进行循环的处理。游标的使用包括游标的声明、OPEN、FETCH和CLOSE，其语法分别如下。

> 声明游标：
> `DECLARE 游标名称 CURSOR FOR 查询语句`

> 打开游标：
> `OPEN 游标名称`

> 获取游标记录：
> `FETCH 游标名称INTO变量[变量]`

条件处理程序：
条件处理程序（Handler）可以用来定义在流程控制结构执行过程中遇到问题时相应的处理步骤。具体语法为：

> DECLARE handler action HANDLER FOR condition value L condition value]..statement

handler_action CONTINUE：继续执行当前程序

EXIT：终止执行当前程序

condition_value :

`SQLSTATE sqlstate_value：状态码，如02000`

`SQLWARNING：所有以01开头的SQLSTATE代码的简写`

`NOT FOUND：所有以02开头的SQLSTATE代码的简写`

`SQLEXCEPTION：所有没有被SQLWARNING或NOT FOUND捕获的SQLSTATE代码的简写`

例子：

NOTE：要先声明普通变量，再申请游标。

要求：
`根据传入的参数uage，来查询用户表tb_user中，所有的用户年龄小于等于uage的用户姓名（name）和专业（profession），并将用户的姓名和专业插入到所创建的一张新表（id，name，profession）中。`

```
create procedure p1l(in uage int)

    begin

        declare uname varchar(100);

        decLare upro varchar(100);

        declare u_cursor cursor for select name,profession from tb_user where
age <= uage;

        当 条件处理程序的处理的状态码为02000的时候，就会退出。
```

```
        declare exit handler for SQLSTATE '02000'close u_cursor;
        drop table if exists tb_user_pro;
        create table if not exists tb_user_pro(
        id int primary key auto_increment,
        name varchar(100),
        profession varchar(100)
        );
        open u_cursor;
        while true do
        fetch u_cursor into uname,Upro;
        insert into tb_user_pro values(null,uname,Upro);
        end while;
        close u_cursor;
    end;
```

触发器

介绍
触发器是与表有关的数据库对象，指在insert/update/delete之前或之后，触发并执行触发器中定义的SQL语句集合。触发器的这种特性可以协助应用在数据库端确保数据的完整性，日志记录，数据校验等操作。
使用别名OLD和NEW来引用触发器中发生变化的记录内容，这与其他的数据库是相似的。现在触发器还只支持行级触发（比如说 一条语句影响了 5 行 则会被触发 5 次），不支持语句级触发（比如说 一条语句影响了 5 行 则会被触发 1 次）。

触发器类型	NEW 和 OLD
INSERT	NEW 表示将要或者已经新增的数据
UPDATE	OLD表示修改之前的数据，NEW表示将要或已经修改后的数据
DELETE	OLD表示将要或者已经删除的数据

锁

锁是计算机协调多个进程或线程并发访问某一资源的机制。在数据库中，除传统的计算资源（CPU、RAM、I/O）的争用以外，数据也是一种供许多用户共享的资源。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，锁冲突也是影响数据库并发访问性能的一个重要因素。从这个角度来说，锁对数据库而言显得尤其重要，也更加复杂。

NOTE : 针对事物才有加锁的意义。

分类：MySQL中的锁，按照锁的粒度分，分为以下三类：

1. 全局锁：锁定数据库中的所有表。
2. 表级锁：每次操作锁住整张表。
3. 行级锁：每次操作锁住对应的行数据。

全局锁：

全局锁就是对整个数据库实例加锁，加锁后整个实例就处于只读状态，后续的DML的写语句，DDL语句，已经更新操作的事务提交语句都将被阻塞。
其典型的使用场景是做全库的逻辑备份，对所有的表进行锁定，从而获取一致性视图，保证数据的完整性。

表锁：

表级锁，每次操作锁住整张表。锁定粒度大，发生锁冲突的概率最高，并发度最低。应用在MyISAM、InnoDB、BDB等存储引擎中。

对于表级锁，主要分为以下三类：

1. 表锁：对于表锁，分为两类：1.表共享读锁（read lock）所有的事物都只能读（当前加锁的客户端也只能读，不能写），不能写 2.表独占写锁（write lock），对当前加锁的客户端，可读可写，对于其他的客户端，不可读也不可写。
   读锁不会阻塞其他客户端的读，但是会阻塞写。写锁既会阻塞其他客户端的读，又会阻塞其他客户端的写。
2. 元数据锁（meta data lock，MDL），MDL加锁过程是系统自动控制，无需显式使用，在访问一张表的时候会自动加上。MDL锁主要作用是维护表元数据的数据一致性，在表上有活动事务的时候，不可以对元数据进行写入操作。在MySQL5.5中引入了MDL，当对一张表进行增删改查的时候，加MDL读锁（共享）;当对表结构进行变更操作的时候，加MDL写锁（排他）。
3. 意向锁: 为了避免DML在执行时，加的行锁与表锁的冲突，在InnoDB中引入了意向锁，使得表锁不用检查每行数据是否加锁，使用意向锁来减少表锁的检查。
   一个客户端对某一行加上了行锁，那么系统也会对其加上一个意向锁，当别的客户端来想要对其加上表锁时，便会检查意向锁是否兼容，若是不兼容，便会阻塞直到意向锁释放。

意向锁兼容性：

1. 意向共享锁（IS）：与表锁共享锁（read）兼容，与表锁排它锁（write）互斥。
2. 意向排他锁（lX）：与表锁共享锁（read）及排它锁（write）都互斥。意向锁之间不会互斥。

行锁：

行级锁，每次操作锁住对应的行数据。锁定粒度最小，发生锁冲突的概率最低，并发度最高。应用在InnoDB存储引擎中。
InnoDB的数据是基于索引组织的，行锁是通过对索引上的索引项加锁来实现的，而不是对记录加的锁。
对于行级锁，主要分为以下三类：

1. 行锁（Record Lock）：锁定单个行记录的锁，防止其他事务对此行进行update和delete。在RC（read commit ）、RR（repeat read）隔离级别下都支持。
2. 间隙锁（GapLock）：锁定索引记录间隙（不含该记录），确保索引记录间隙不变，防止其他事务在这个间隙进行insert，产生幻读。在RR隔离级别下都支持。比如说 两个临近叶子节点为 15 23，那么间隙就是指 [15 , 23],锁的是这个间隙。
3. 临键锁（Next-Key Lock）：行锁和间隙锁组合，同时锁住数据，并锁住数据前面的间隙Gap。在RR隔离级别下支持。

InnoDB实现了以下两种类型的行锁：

1. 共享锁（S）：允许一个事务去读一行，阻止其他事务获得相同数据集的排它锁。

2. 排他锁（X）：允许获取排他锁的事务更新数据，阻止其他事务获得相同数据集的共享锁和排他锁。

SQL	行锁类型	说明
insert	排他锁	自动加锁
update	排他锁	自动加锁
delete	排他锁	自动加锁
select	不加任何锁	
select lock in share mode	排他锁	需要手动在SELECT之后加LOCK IN SHARE MODE
select for update	排他锁	需要手动在SELECT之后加FOR UPDATE

行锁 - 演示

默认情况下，InnoDB在REPEATABLE READ事务隔离级别运行，InnoDB使用next-key 锁进行搜索和索引扫描，以防止幻读。

1. 针对唯一索引进行检索时，对已存在的记录进行等值匹配时，将会自动优化为行锁。
2. InnoDB的行锁是针对于索引加的锁，不通过索引条件检索数据，那么InnoDB将对表中的所有记录加锁，此时就会升级为表锁。

间隙锁/临键锁-演示

默认情况下，InnoDB在REPEATABLE READ事务隔离级别运行，InnoDB使用next-key 锁进行搜索和索引扫描，以防止幻读。

1. 索引上的等值查询（唯一索引），给不存在的记录加锁时，优化为间隙锁。
2. 索引上的等值查询（普通索引），向右遍历时最后一个值不满足查询需求时，next-key lock 退化为间隙锁。
3. 索引上的范围查询（唯一索引）--会访问到不满足条件的第一个值为止。

注意：间隙锁唯一目的是防止其他事务插入间隙。间隙锁可以共存，一个事务采用的间隙锁不会阻止另一个事务在同一间隙上采用间隙锁。

InnoDB 引擎

逻辑存储结构

表空间（ibd文件），一个mysql实例可以对应多个表空间，用于存储记录、索引等数据。

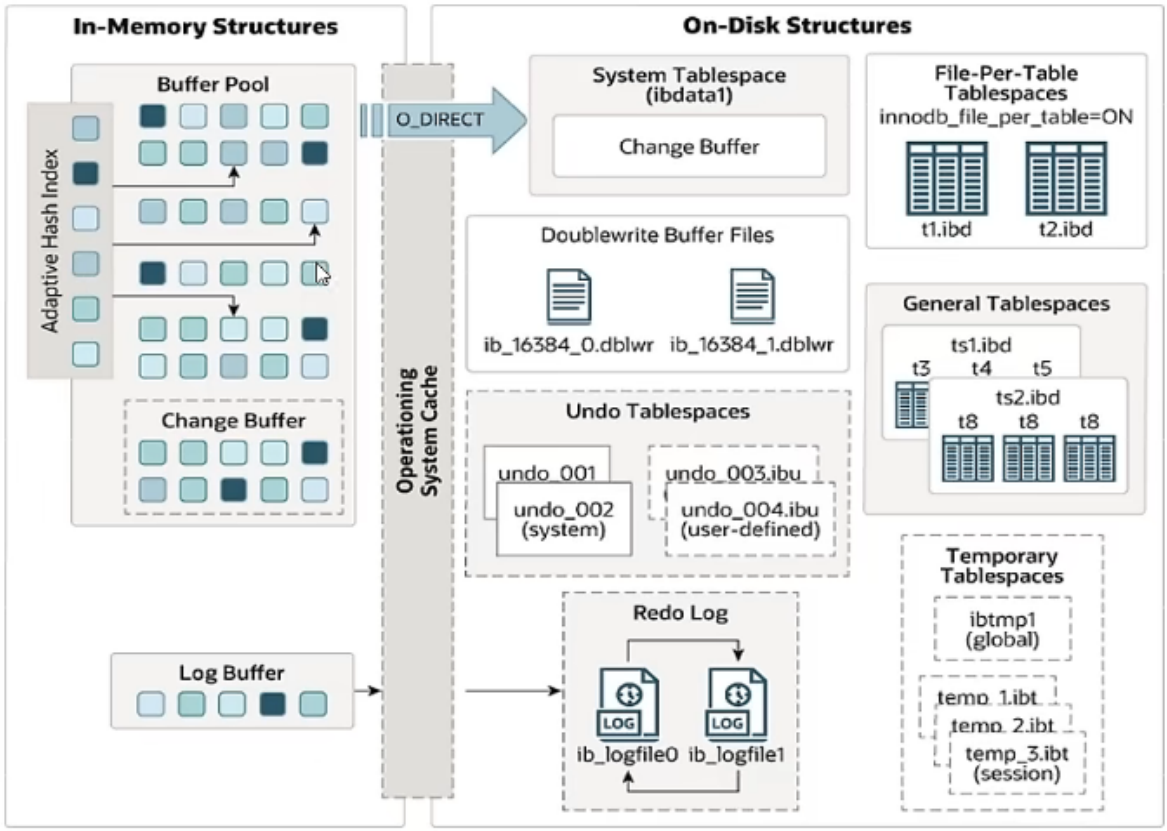
段，分为数据段（Leaf node segment）、索引段（Non-leaf node segment）、回滚段（Rollback segment），InnoDB是索引组织表，数据段就是B+树的叶子节点，索引段即为B+树的非叶子节点。段用来管理多个Extent（区）。

区，表空间的单元结构，每个区的大小为1M。默认情况下，InnoDB存储引擎页大小为16K，即一个区中一共有64个连续的页。

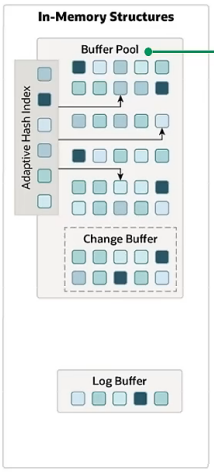
页，是InnoDB存储引擎磁盘管理的最小单元，每个页的大小默认为16KB。为了保证页的连续性，InnoDB存储引擎每从磁盘申请4-5个区。一页包含若干行。

行，InnoDB存储引擎数据是按进行存放的。

架构



Buffer Pool：缓冲池是主内存中的一个区域，里面可以缓存磁盘上经常操作的真实数据，在执行增删改查操作时，先操作缓冲池中的数据（若缓冲池没有数据，则从磁盘加载并缓存），然后再以一定频率刷新到磁盘，从而减少磁盘I0，加快处理速度。

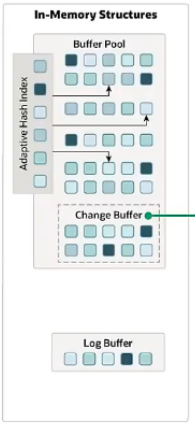


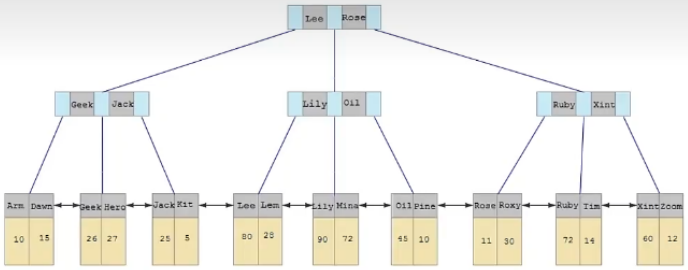
Buffer Pool：缓冲池是主内存中的一个区域，里面可以缓存磁盘上经常操作的真实数据，在执行增删改查操作时，先操作缓冲池中的数据（若缓冲池没有数据，则从磁盘加载并缓存），然后再以一定频率刷新到磁盘，从而减少磁盘IO，加快处理速度。

缓冲池以Page页为单位，底层采用链表数据结构管理Page。根据状态，将Page分为三种类型：

- free page：空闲page，未被使用。
- clean page：被使用page，数据没有被修改过。
- dirty page：脏页，被使用page，数据被修改过，也中数据与磁盘的数据产生了不一致。

架构-内存架构

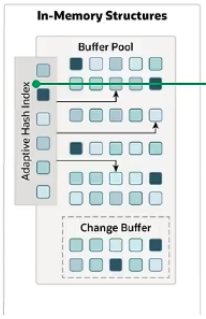




Change Buffer：更改缓冲区（针对于非唯一二级索引页），在执行DML语句时，如果这些数据Page没有在Buffer Pool中，不会直接操作磁盘，而会将数据变更存在更改缓冲区 Change Buffer 中，在未来数据被读取时，再将数据合并恢复到Buffer Pool中，再将合并后的数据刷新到磁盘中。

Change Buffer的意义是什么?

与聚集索引不同，二级索引通常是非唯一的，并且以相对随机的顺序插入二级索引。同样，删除和更新可能会影响索引树中不相邻的二级索引页，如果每一次都操作磁盘，会造成大量的磁盘IO。有了ChangeBuffer之后，我们可以在缓冲池中进行合并处理，减少磁盘IO。

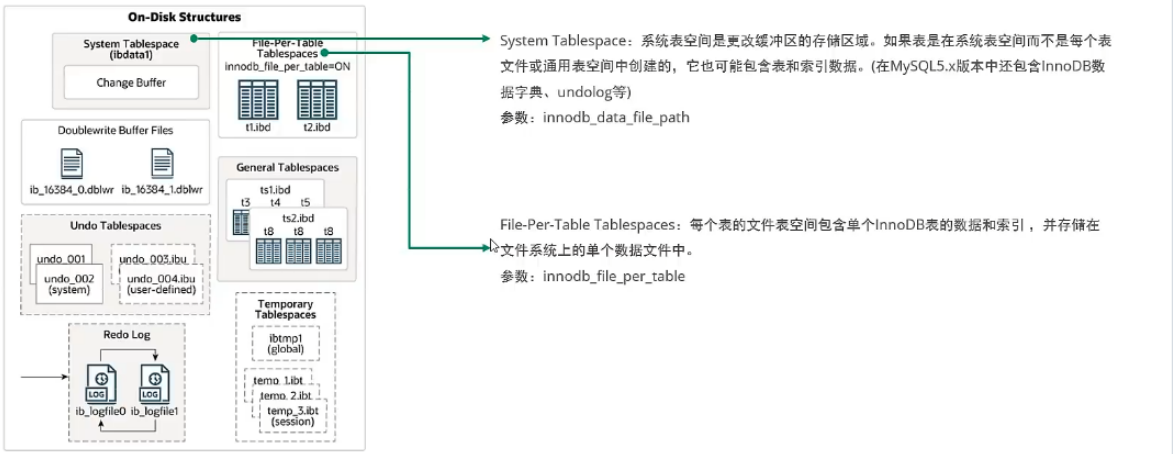


Adaptive Hash Index：自适应hash索引，用于优化对Buffer Pool数据的查询。InnoDB存储引擎会监控对表上各索引页的查询，如果观察到hash索引可以提升速度，则建立hash索引，称之为自适应hash索引。

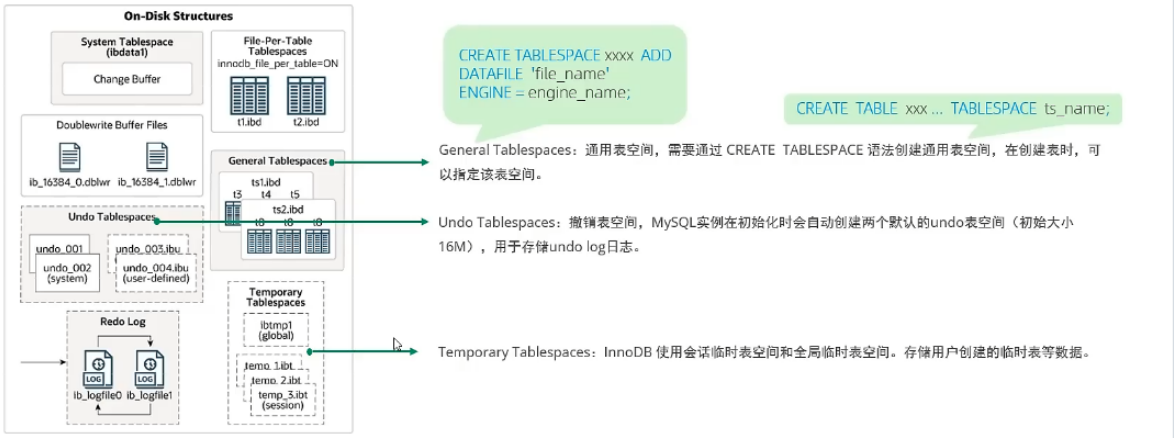
自适应哈希索引，无需人工干预，是系统根据情况自动完成。

参数： adaptive_hash_index

磁盘架构：



架构-磁盘结构



InnoDB的整个体系结构为：

当业务操作的时候直接操作的是内存缓冲区，如果缓冲区当中没有数据，则会从磁盘中加载到缓冲区，增删改查都是在缓冲区的，后台线程以一定的速率刷新到磁盘。

事务原理

事务是一组操作的集合，它是一个不可分割的工作单位，事务会把所有的操作作为一个整体一起向系统提交或撤销操作请求，即这些操作要么同时成功，要么同时败。具有ACID四大特征。

原子性，一致性，持久性这三大特性由 redo log 和 undo log 日志来保证的。
隔离性 是由锁机制和MVCC保证的。

redo log:

重做日志，记录的是事务提交时数据页的物理修改，是用来实现事务的持久性。
该日志文件由两部分组成：重做日志缓冲（redo log buffer）以及重做日志文件（redo log file），前者是在内存中，后者在磁盘中。当事务提交之后会把所有修改信息都存到该日志文件中，用于在刷新脏页到磁盘，发生错误时，进行数据恢复使用。

> 个人理解： 事物每次提交的时候都会将数据刷到redo log中而不是直接将buffer pool中的数据直接刷到磁盘中（ibd文件中），是因为redo log 是顺序写，性能处理的够快，直接刷到ibd中，是随机写，性能慢。所以脏页是在下一次读的时候，或者后台线程采用一定的机制进行刷盘到ibd中。

undo log:
回滚日志，用于记录数据被修改前的信息，作用包含两个：提供回滚和MVCC（多版本并发控制）。
undo log和redo log记录物理日志不一样，它是逻辑日志。可以认为当delete一条记录，undo log中会记录一条对应的insert记录，反之亦然，当update一条记录时，它记录一条对应相反的update记录。当执行rollback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：undo log在事务执行时产生，事务提交时，并不会立即删除undo log，因为这些日志可能还用于MVCC。
> Undo log存储：undo log采用段的方式进行管理和记录，存放在前面介绍的rollback segment回滚段中，内部包含1024个undo log segment。

MVCC

当前读:

读取的是记录的最新版本，读取时还要保证其他并发事务不能修改当前记录，会对读取的记录进行加锁。对于我们日常的操作，如：

- select...lock in share mode（共享锁）。
- select.....for update、update、insert、delete（排他锁）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，读取的是记录数据的可见版本，有可能是历史数据，不加锁，是非阻塞读。

- Read Committed：每次select，都生成一个快照读。
- Repeatable Read：开启事务后第一个select语句才是快照读的地方。
- Serializable：快照读会退化为当前读。

MVCC:

全称Multi-Version Concurrency Control，多版本并发控制。指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MySQL实现MVCC提供了一个非阻塞读功能。MVCC的具体实现，还需要依赖于数据库记录中的三个隐式字段、undo log日志、readView。

MVCC 实现原理:

有三个隐藏的字段:

MVCC-实现原理

- 记录中的隐藏字段

id	age	name
1	1	tom
3	3	cat

id	age	name	DB_TRX_ID	DB_ROLL_PTR

隐藏字段	含义
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，用于配合undo log，指向上一个版本。
DB_ROW_ID	隐藏主键，如果表结构没有指定主键，将会生成该隐藏字段。

> undo log回滚日志，在insert、update、delete的时候产生的便于数据回滚的日志。 当insert的时候，产生的undo log日志只在回滚时需要，在事务提交后，可被立即删除。 而update、delete的时候，产生的undo log日志不仅在回滚时需要，在快照读时也需要，不会立即被删除。

> undo log回滚日志，在insert、update、delete的时候产生的便于数据回滚的日志。
> 当insert的时候，产生的undo log日志只在回滚时需要，在事务提交后，可被立即删除。
> 而update、delete的时候，产生的undo log日志不仅在回滚时需要，在快照读时也需要，不会立即被删除。

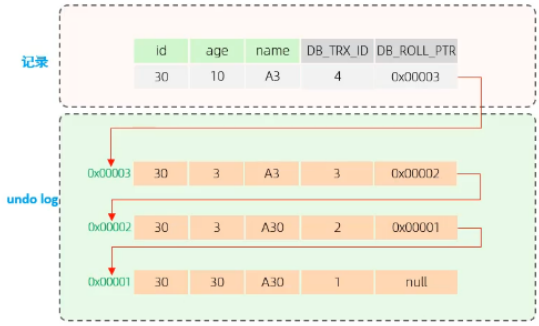
undo log 版本链：

undo log日志会记录原来的版本的数据，因为是通过undo log 日志进行回滚的。

MVCC-实现原理

- undo log版本链

事务2	事务3	事务4	事务5
开始事务	开始事务	开始事务	开始事务
修改id为30记录，age改为3		查询id为30的记录	
提交事务			
	修改id为30记录，name改为A3		
			查询id为30的记录
	提交事务		
		修改id为30记录，age改为10	
		查询id为30的记录	
			查询id为30的记录
		提交事务	



不同事务或相同事务对同一条记录进行修改，会导致该记录的undolog生成一条记录版本链表，链表的头部是最新的旧记录，链表尾部是最早的旧记录。

如何确定返回哪一个版本 这是由read view决定返回 undo log 中的哪一个版本。

- readview

ReadView（读视图）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。

ReadView中包含了四个核心字段：

字段	含义
m_ids	当前活跃的事务ID集合
min_trx_id	最小活跃事务ID
max_trx_id	预分配事务ID，当前最大事务ID+1（因为事务ID是自增的）
creator_trx_id	ReadView创建者的事务ID

> RC隔离级别下，在事务中每一次执行快照读时生成ReadView。
> RR隔离级别下，在事务中第一次执行快照读时生成ReadView，后续会复用。

> https://www.bilibili.com/video/BV1Kr4y1i7ru?p=145&spm_id_from=pageDriver&vd_source=bbc04b831b54029788a178a7c2e9ae20

MVCC 靠 隐藏字段 , undo log 版本链 , read view 实现的。

- 原子性-undo log
- 持久性-redo log
- 一致性-undo log + redo log
- 隔离性-锁 + MVCC

MVCC 靠 隐藏字段 , undo log 版本链 , read view 实现的。

- 原子性-undo log
- 持久性-redo log
- 一致性-undo log + redo log
- 隔离性-锁 + MVCC

MVCC-实现原理

- readview

不同的隔离级别，生成ReadView的时机不同：

- READ COMMITTED ：在事务中每一次执行快照读时生成ReadView。
- REPEATABLE READ：仅在事务中第一次执行快照读时生成ReadView，后续复用该ReadView。

数据类型

整型

类型名称	取值范围	大小
TINYINT	-128～127	1个字节
SMALLINT	-32768～32767	2个字节
MEDIUMINT	-8388608～8388607	3个字节
INT (INTEGHR)	-2147483648～2147483647	4个字节
BIGINT	-9223372036854775808～9223372036854775807	8个字节

无符号在数据类型后加 unsigned 关键字。

浮点型

类型名称	说明	存储需求

类型名称	说明	存储需求
FLOAT	单精度浮点数	4 个字节
DOUBLE	双精度浮点数	8 个字节
DECIMAL (M, D)，DEC	压缩的“严格”定点数	M+2 个字节

日期和时间

类型名称	日期格式	日期范围	存储需求
YEAR	YYYY	1901 ~ 2155	1 个字节
TIME	HH:MM:SS	-838:59:59 ~ 838:59:59	3 个字节
DATE	YYYY-MM-DD	1000-01-01 ~ 9999-12-3	3 个字节
DATETIME	YYYY-MM-DD HH:MM:SS	1000-01-01 00:00:00 ~ 9999-12-31 23:59:59	8 个字节
TIMESTAMP	YYYY-MM-DD HH:MM:SS	1980-01-01 00:00:01 UTC ~ 2040-01-19 03:14:07 UTC	4 个字节

字符串

类型名称	说明	存储需求
CHAR(M)	固定长度非二进制字符串	M 字节，1<=M<=255
VARCHAR(M)	变长非二进制字符串	L+1字节，在此，L< = M和 1<=M<=255
TINYTEXT	非常小的非二进制字符串	L+1字节，在此，L<2^8
TEXT	小的非二进制字符串	L+2字节，在此，L<2^16
MEDIUMTEXT	中等大小的非二进制字符串	L+3字节，在此，L<2^24
LONGTEXT	大的非二进制字符串	L+4字节，在此，L<2^32
ENUM	枚举类型，只能有一个枚举字符串值	1或2个字节，取决于枚举值的数目 (最大值为65535)
SET	一个设置，字符串对象可以有零个或 多个SET成员	1、2、3、4或8个字节，取决于集合 成员的数量（最多64个成员）

二进制类型

类型名称	说明	存储需求

类型名称	说明	存储需求
BIT(M)	位字段类型	大约 (M+7)/8 字节
BINARY(M)	固定长度二进制字符串	M 字节
VARBINARY (M)	可变长度二进制字符串	M+1 字节
TINYBLOB (M)	非常小的BLOB	L+1 字节，在此，L<2^8
BLOB (M)	小 BLOB	L+2 字节，在此，L<2^16
MEDIUMBLOB (M)	中等大小的BLOB	L+3 字节，在此，L<2^24
LONGBLOB (M)	非常大的BLOB	L+4 字节，在此，L<2^32

权限一览表

> 具体权限的作用详见官方文档

GRANT 和 REVOKE 允许的静态权限

Privilege	Grant Table Column	Context
`ALL [PRIVILEGES]`	Synonym for “all privileges”	Server administration
`ALTER`	`Alter_priv`	Tables
`ALTER ROUTINE`	`Alter_routine_priv`	Stored routines
`CREATE`	`Create_priv`	Databases, tables, or indexes
`CREATE ROLE`	`Create_role_priv`	Server administration
`CREATE ROUTINE`	`Create_routine_priv`	Stored routines
`CREATE TABLESPACE`	`Create_tablespace_priv`	Server administration
`CREATE TEMPORARY TABLES`	`Create_tmp_table_priv`	Tables
`CREATE USER`	`Create_user_priv`	Server administration
`CREATE VIEW`	`Create_view_priv`	Views
`DELETE`	`Delete_priv`	Tables
`DROP`	`Drop_priv`	Databases, tables, or views
`DROP ROLE`	`Drop_role_priv`	Server administration
`EVENT`	`Event_priv`	Databases
`EXECUTE`	`Execute_priv`	Stored routines
`FILE`	`File_priv`	File access on server host

Privilege	Grant Table Column	Context
GRANT OPTION	Grant_priv	Databases, tables, or stored routines
INDEX	Index_priv	Tables
INSERT	Insert_priv	Tables or columns
LOCK TABLES	Lock_tables_priv	Databases
PROCESS	Process_priv	Server administration
PROXY	See proxies_priv table	Server administration
REFERENCES	References_priv	Databases or tables
RELOAD	Reload_priv	Server administration
REPLICATION CLIENT	Repl_client_priv	Server administration
REPLICATION SLAVE	Repl_slave_priv	Server administration
SELECT	Select_priv	Tables or columns
SHOW DATABASES	Show_db_priv	Server administration
SHOW VIEW	Show_view_priv	Views
SHUTDOWN	Shutdown_priv	Server administration
SUPER	Super_priv	Server administration
TRIGGER	Trigger_priv	Tables
UPDATE	Update_priv	Tables or columns
USAGE	Synonym for “no privileges”	Server administration

GRANT 和 REVOKE 允许的动态权限

Privilege	Context
APPLICATION_PASSWORD_ADMIN	Dual password administration
AUDIT_ABORT_EXEMPT	Allow queries blocked by audit log filter
AUDIT_ADMIN	Audit log administration
AUTHENTICATION_POLICY_ADMIN	Authentication administration
BACKUP_ADMIN	Backup administration
BINLOG_ADMIN	Backup and Replication administration
BINLOG_ENCRYPTION_ADMIN	Backup and Replication administration

Privilege	Context
CLONE_ADMIN	Clone administration
CONNECTION_ADMIN	Server administration
ENCRYPTION_KEY_ADMIN	Server administration
FIREWALL_ADMIN	Firewall administration
FIREWALL_EXEMPT	Firewall administration
FIREWALL_USER	Firewall administration
FLUSH_OPTIMIZER_COSTS	Server administration
FLUSH_STATUS	Server administration
FLUSH_TABLES	Server administration
FLUSH_USER_RESOURCES	Server administration
GROUP_REPLICATION_ADMIN	Replication administration
GROUP_REPLICATION_STREAM	Replication administration
INNODB_REDO_LOG_ARCHIVE	Redo log archiving administration
NDB_STORED_USER	NDB Cluster
PASSWORDLESS_USER_ADMIN	Authentication administration
PERSIST_RO_VARIABLES_ADMIN	Server administration
REPLICATION_APPLIER	PRIVILEGE_CHECKS_USER for a replication channel
REPLICATION_SLAVE_ADMIN	Replication administration
RESOURCE_GROUP_ADMIN	Resource group administration
RESOURCE_GROUP_USER	Resource group administration
ROLE_ADMIN	Server administration
SESSION_VARIABLES_ADMIN	Server administration
SET_USER_ID	Server administration
SHOW_ROUTINE	Server administration
SYSTEM_USER	Server administration
SYSTEM_VARIABLES_ADMIN	Server administration
TABLE_ENCRYPTION_ADMIN	Server administration
VERSION_TOKEN_ADMIN	Server administration

Privilege	Context
`XA_RECOVER_ADMIN`	Server administration

图形化界面工具

- Workbench(免费): http://dev.mysql.com/downloads/workbench/
- navicat(收费，试用版30天): https://www.navicat.com/en/download/navicat-for-mysql
- Sequel Pro(开源免费，仅支持Mac OS): http://www.sequelpro.com/
- HeidiSQL(免费): http://www.heidisql.com/
- phpMyAdmin(免费): https://www.phpmyadmin.net/
- SQLyog: https://sqlyog.en.softonic.com/

安装

小技巧

1. 在SQL语句之后加上`\G`会将结果的表格形式转换成行文本形式
2. 查看Mysql数据库占用空间：

```
SELECT table_schema "Database Name"
     , SUM(data_length + index_length) / (1024 * 1024) "Database Size in MB"
FROM information_schema.TABLES
GROUP BY table_schema;
```

参考文献

> https://dhc.pythonanywhere.com/entry/share/?key=3ad29aad765a2b98b2b2a745d71bef715507ee9db8adbec98257bac0ad84cbe4#h1-u6743u9650u4E00u89C8u8868
> 这篇笔记是在别人的基础上完善而来，感谢B站的黑马程序员up主，也感谢路途博客。