# MySQL数据库基础篇

## 1 MySQL概述

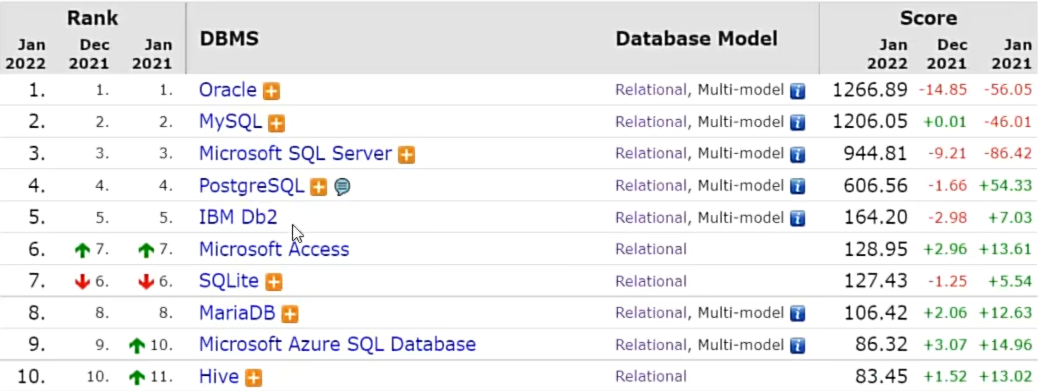
### 1.1 数据库相关概念

数据库(DataBase, DB), 存储数据的仓库, 数据时有组织的进行存储.

数据库管理系统(DataBase Management System, DBMS), 操纵和管理数据库的大型软件.

SQL(Structured Query Language), 操作关系型数据库的编程语言, 定义了一套操作关系型数据库的统一标准,

* **主流的关系型数据库管理系统**



### 1.2 下载安装

启动与停止

启动: net start mysql80

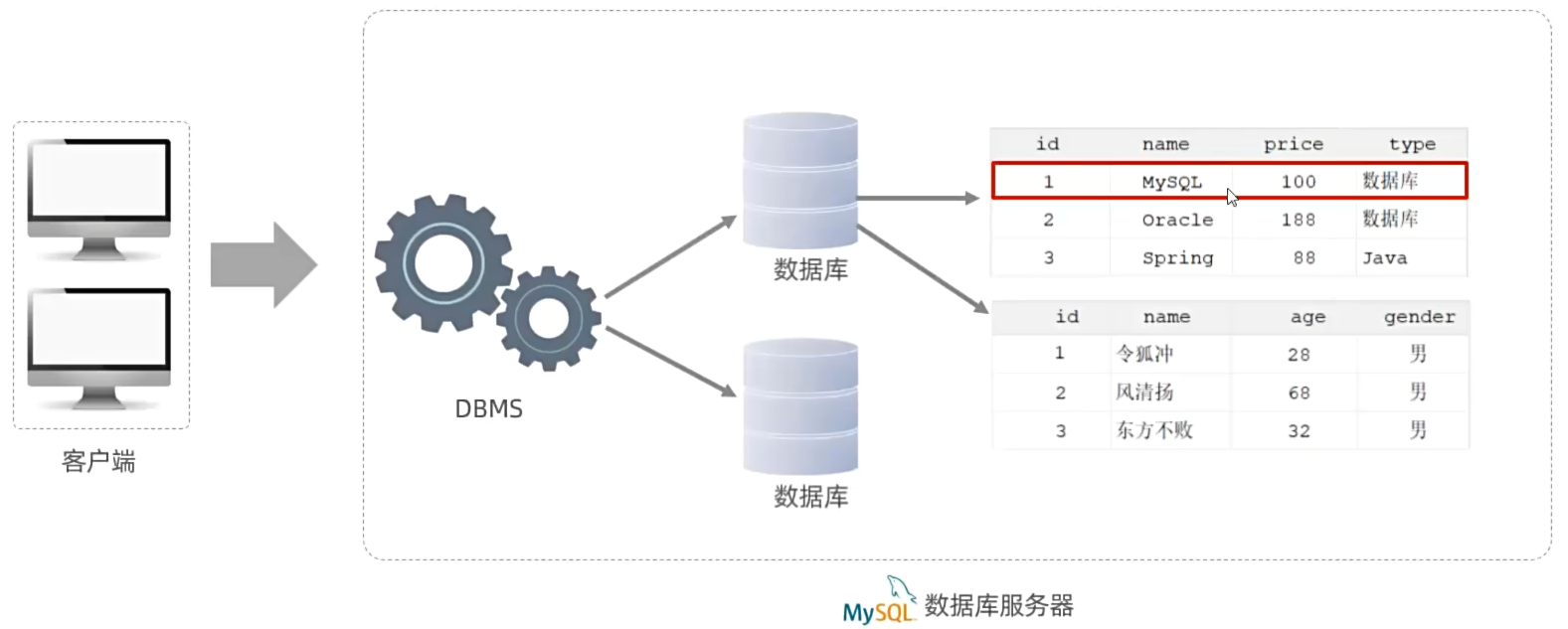
停止: net stop mysql80

客户端连接

方式一: MySQL提供的客户端命令行工具

方式二: 方式二：系统自带的命令行工具执行指令 mysql [-h 127.0.0.1] [-P330]-u root -p

### 1.3 数据模型



关系型数据库(RDBMS):

建立在关系模型基础上，由多张相互连接的二维表组成的数据库(通过表结构存储)。

特点:

1.使用表存储数据，格式统一，便于维护

2.使用SQL语言操作，标准统一，使用方便

## 2 SQL

### 2.1 通用语法

1. SQL语句可以单行或多行书写，以分号结尾。

2. SQL语句可以使用空格/缩进来增强语句的可读性。

3. MySQL数据库的SQL语句不区分大小写，关键字建议使用大写。

4. 注释：

单行注释：-- 注释内容或 # 注释内容(MySQL特有)

多行注释：/\*注释内容\*/

### 2.2 分类



### 2.3 DDL 数据定义语言

#### 2.3.1 DDL-数据库操作

查询

查询所有数据库

SHOW DATABASES

查询当前数据库

SELECT DATABASE();

创建

CREATE DATABASE [IF NOT EXISTS] 数据库名 [DEFAULT CHARSET 字符集] [COLLATE排序规则];

删除

DROP DATABASE [IF EXISTS] 数据库名;

使用

USE数据库名;

#### 2.3.2 DDL-表操作-查询

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;

#### 2.3.3 DDL-表操作-创建

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];

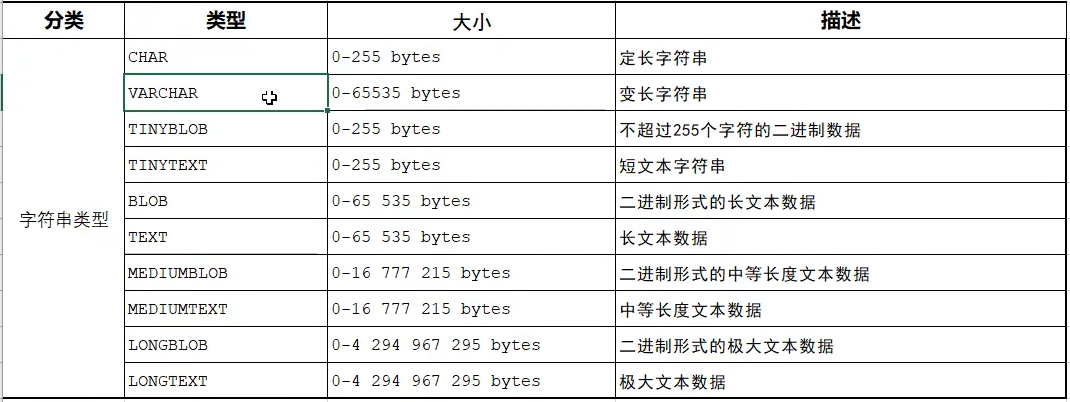
[…]为可选参数

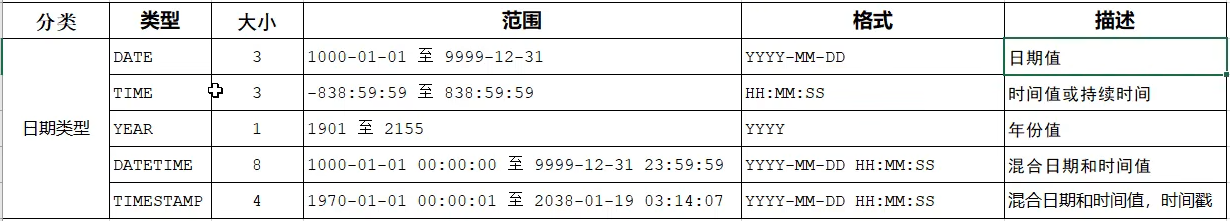
#### 2.3.4 DDL-表操作-数据类型



精度：整段数字的长度； 标度：小数后的数字长度

age TINYINT UNSIGNED





birthday date 2001-12-01

#### 2.3.5 DDL-表操作-修改

添加字段

ALTER TABLE 表名 ADD 字段名类型(长度) [COMMENT注释][约束];

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];

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;

#### 2.3.6 DDL-表操作-删除

删除表

DAOP TABLE [IF EXISTS] 表名；

删除指定表，并重新创建该表

TRUNCATE TABLE 表名；

### 2.4 DML 数据操作语言

#### 2.4.1 DML-添加数据

1. 给指定字段添加数据

2. 给全部字段添加数据

INSERT INTO 表名 VALUES (值1, 值2, …);

3. 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：

·插入数据时，指定的字段顺序需要与值的顺序是一一对应的。

·字符串和日期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国内。

#### 2.4.2 DML-修改数据

UPDATE 表名 SET 字段名1=值1, 字段名2=值2, [WHERE 条件];

注意：修改语句的条件可以有，也可以没有，如果没有条件，则会修改整张表的所有数据。

#### 2.4.3 DML-删除数据

DELETE FROM 表名 [WHERE 条件]

注意：

DELETE语句的条件可以有，也可以没有，如果没有条件，则会删除整张表的所有数据。

DELETE语句不能删除某一个字段的值(可以使用UPDATE)。

### 2.5 DQL 数据查询语言

查询关键字 SELECT

#### 2.5.1 DQL语法

SELECT

字段列表

FROM

表名列表

WHERE

条件列表

GROUP BY

分组字段列表

HAVING

分组后条件列表

ORDER BY

排序字段列表

LIMIT

分页参数

以上为**编写顺序**

#### 2.5.2 DQL-基本查询

1. 查询多个字段

SELECT 字段1, 字段2, 字段3 FROM 表名;

SELECT \* FROM表名;

2. 设置别名

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

3.去除重复记录

SELECT DISTINCT

字段列表FROM表名；

#### 2.5.3 DQL-条件查询

1. 语法

SELECT 字段列表 FROM 表名 WHERE 条件列表;

2. 条件



#### 2.5.4 DQL-聚合函数

1. 介绍: 将一列数据作为一个整体,进行纵向计算.

2. 常用聚合函数



3. 语法

SELECT 聚合函数 (字段列表) FROM 表名；

注意：null值不参与所有聚合函数运算。

#### 2.5.5 DQL-分组查询

1.语法

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

2. where与having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; 而having是分组之后对结果进行过滤。

判断条件不同：where不能对聚合函数进行判断，而having可以。

注意

·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。

#### 2.5.6 DQL-排序查询

1. 语法

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

2. 排序方式

ASC:升序（默认值）

DESC:降序

注意：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。

#### 2.5.7 DQL-分页查询

1.语法

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

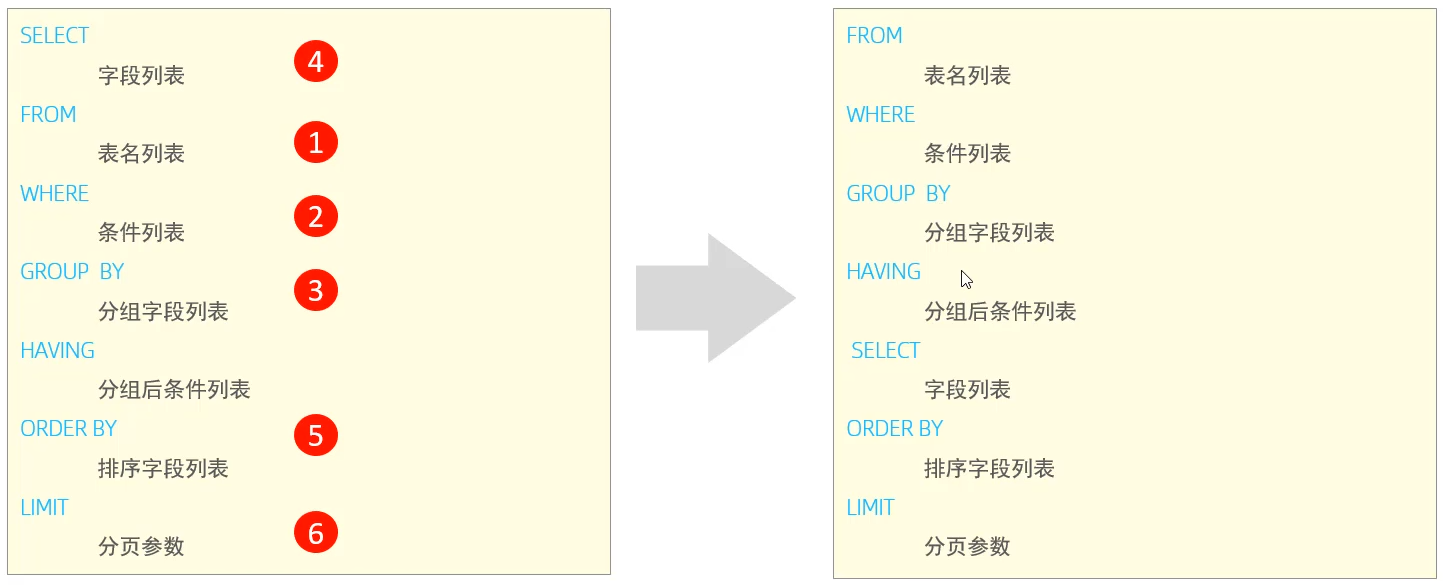
注意

·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，不同的数据库有不同的实现，MySQL中是LIMIT.

·如果查询的是第一页数据，起始索引可以省略，直接简写为limit 10。

#### 2.5.8 DQL-执行顺序



### 2.6 DCL 数据控制语言

#### 2.6.1 DCL-用户管理

1.查询用户

USE mysql;

SELECT FROM ser;

2.创建用户

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

3.修改用户密码

ALTER USER '用户名'@'主机名' IDENTIFIED WITH mysgl native password BY '新密码';

4.删除用户

DROP USER '用户名'@'主机名';

注意：

·主机名可以使用%通配。

·这类SQL开发人员操作的比较少，主要是DBA(Database Administrator数据库管理员)使用。

#### 2.6.2 DCL-权限控制



1.查询权限

SHOW GRANTS FOR '用户名'@'主机名'；

2.授予权限

GRANT 权限列表 ON 数据库名表名TO ‘用户名’@‘主机名’;

3.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：

·多个权限之间，使用逗号分隔

授权时，数据库名和表名可以使用\*进行通配，代表所有。

## 3 函数

函数是指一段可以直接被另一段程序调用的程序或代码。

### 3.1 字符串函数



### 3.2 数值函数



### 3.3 日期函数



### 3.4 流程函数



## 4 约束

### 4.1 概述

概念：约束是作用于表中字段上的规则，用于限制存储在表中的数据。

目的：保证数据库中数据的正确、有效性和完整性。

分类：



注意：约束是作用于表中字段上的，可以在创建表/修改表的时候添加约束。

### 4.2 外键约束

让两张表之间的数据建立连接，保证数据的一致性和完整性。

添加外键

CREATE TABLE 表名(

字段名数据类型，

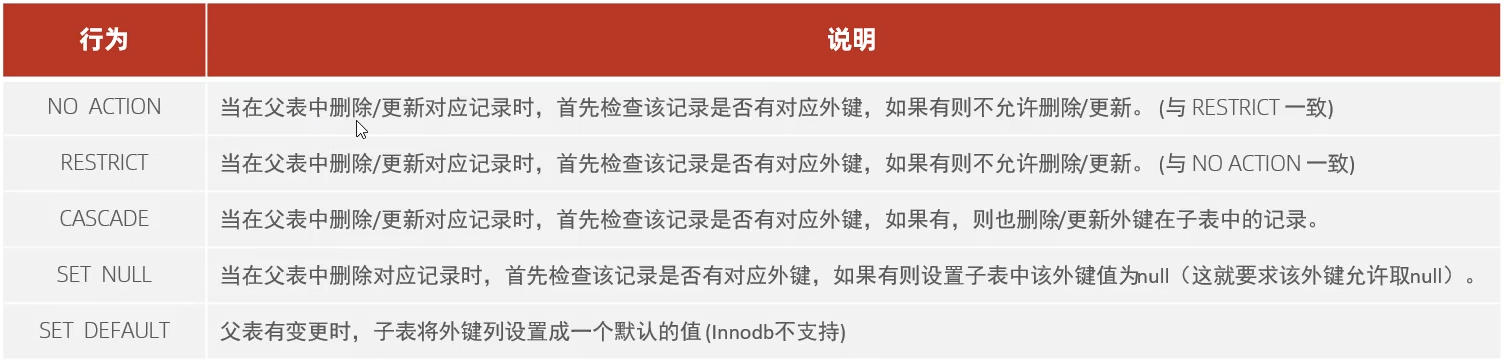
…

[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);

### 4.3 外键删除更新行为



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 CASCADE ON DELETE CASCADE

## 5 多表查询

### 5.1 多表关系概述

**多表查询**：

笛卡尔积：指在数学中，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） 添加where判断

项目开发中，在进行数据库表结构设计时，会根据业务需求及业务模块之间的关系，分析并设计表结构，由于业务之间相互关联，所以各个表结构之间也存在着各种联系，基本上分为三种：

>一对多（多对一）

>多对多

>一对一

* 一对多：



* 多对多：



* 一对一：



**多表查询分类：**

连接查询

内连接：相当于查询A、B交集部分数据

外连接：

左外连接：查询左表所有数据，以及两张表交集部分数据

右外连接：查询右表所有数据，以及两张表交集部分数据

自连接：当前表与自身的连接查询，自连接必须使用表别名

子查询

### 5.2 内连接

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…；

### 5.3 外连接

* 左外连接

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

相当于查询表1（左表）的所有数据包含表1和表2交集部分的数据. 保留左表中的所有记录，不管在右表中是否找到匹配的记录。

* 右外连接

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

相当于查询表2（右表）的所有数据包含表1和表2交集部分的数据

### 5.4 自连接

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。

SELECT A.NAME B.NAME FROM EMP A EMP B WHERE A.MANAGERID = B.ID;

### 5.5 联合查询union, union all

union查询：把多次查询的结果合并起来，形成一个新的查询结果集。

SELECT 字段列表 FROM 表A…;

UNION [ALL]

SELECT 字段列表 FROM 表B…;

union 去重, union all直接连接

### 5.6 子查询

SQL语句中嵌套SELECT语句，称为嵌套查询，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。

根据子查询结果不同，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，分为：WHERE之后、FROM之后、SELECT之后。

### 5.7 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)，最简单的形式，这种子查询成为标量子查询。

常用的操作符：= <> > >= < <=

### 5.8 列子查询

子查询返回的结果是一列（可以是多行），这种子查询称为列子查询。

常用的操作符：IN、NOT IN、ANY、SOME、ALL



### 5.9 行子查询

子查询返回的结果是一行（可以是多列），这种子查询称为行子查询。

常用的操作符：=、<>、IN、NOT IN

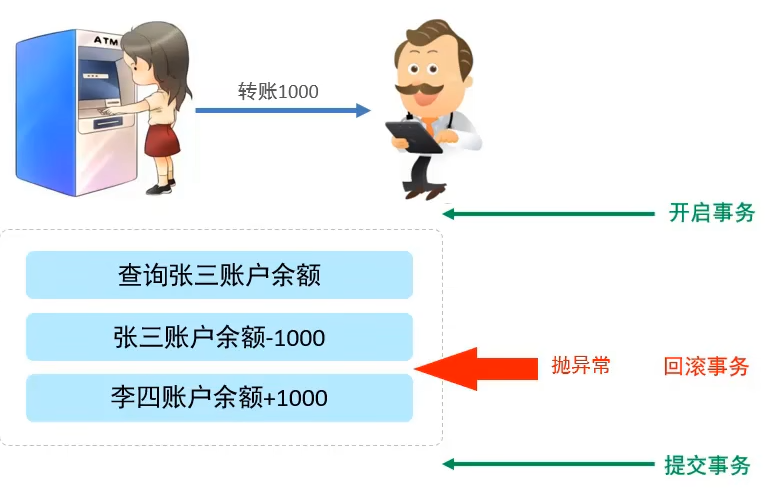
### 5.10 表子查询

子查询返回的结果是多行多列，这种子查询称为表子查询

常用的操作符：IN

## 6 事务

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，要么同时失败。



回滚：把临时修改的数据恢复。

默认MySQL的事务是自动提交的，也就是说，当执行一条DML语句，MySQL会立即隐式的提交事务。

### 6.1 操作演示

查看/设置事务提交方式

SELECT @@autocommit

SET @@autocommit=0; -- 0为手动提交

提交事务

COMMIT;

回滚事务

ROLLBACK;

方式二：不设置为0

开启事务

START TRANSACTION BEGIN;

提交事务

COMMIT;

回滚事务

ROLLBACK;

### 6.2 四大特性ACID

**原子性(Atomicity)**:事务是不可分割的最小操作单元，要么全部成功，要么全部失败。

**一致性(Consistency)**：事务完成时，必须使所有的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。

### 6.3 并发事务问题



### 6.4 隔离级别



从上到下隔离级别越来越高，数据越安全，性能越来越差

查看事务隔离级别

SELECT @@TRANSACTION\_ISOLATION;

设置事务隔离级别

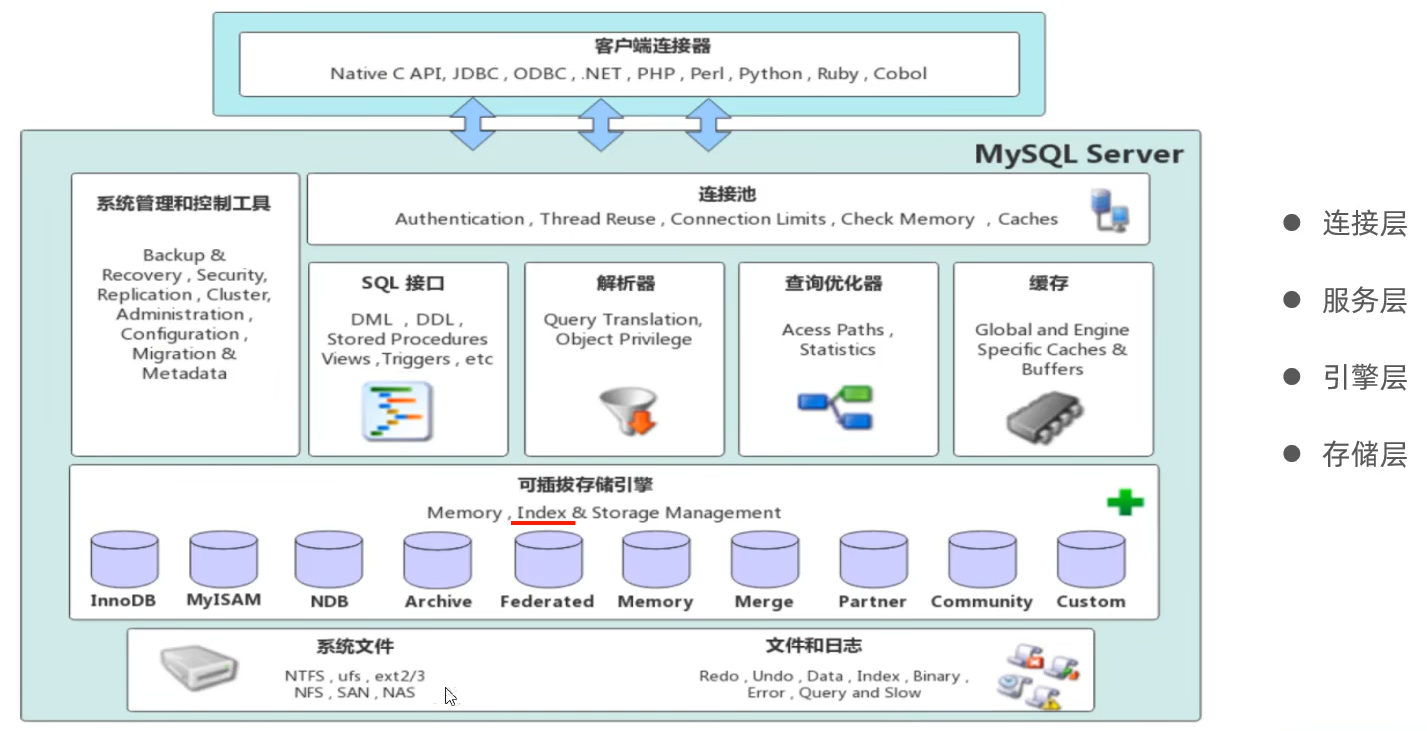
SET [SESSION | GLOBAL] TRANSACTION ISOLATION LEVEL {READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE};

session: 会话级别，仅对当前窗口有效

# 进阶篇

## 7 存储引擎

### 7.1 MySQL体系结构



* 连接层

最上层是一些客户端和链接服务，主要完成一些类似于连接处理、授权认证、及相关的安全方案。服务器也会为安全接入的每个客户端验证它所具有的操作权限。

* 服务层

第二层架构主要完成大多数的核心服务功能，如SQL接口，并完成缓存的查询，SQL的分析和优化，部分内置函数的执行。所有跨存储引擎的功能也在这一层实现，如过程、函数等。

* 引擎层

存储引擎真正的负责了MySQL中数据的存储和提取，服务器通过API和存储引擎进行通信。不同的存储引擎具有不同的功能，这样我们可以根据自己的需要，来选取合适的存储引擎。

* 存储层

主要是将数据存储在文件系统之上，并完成与存储引擎的交互。

### 7.2 存储引擎简介

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。存储引擎是基于表的，而不是基于库的，所以存储引擎也可被称为表类型。

MySQL5.5后，存储引擎默认为InnoDB。

语法:

创建表时，指定存储引擎:

CREATE TABLE表名（

字段1字段1类型[COMMENT字段1注释]，

…

字段n字段n类型[COMMENT字段n注释]

)ENGINE=INNODB [COMMENT表注释];

查看当前数据库支持的存储引擎

SHOW ENGINES

### 7.3 存储引擎特点

#### 7.3.1 InnoDB

InnoDB是一种兼顾高可靠性和高性能的通用存储引擎，在MySQL5.5之后，InnoDB是默认的MySQL存储引擎。

**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；

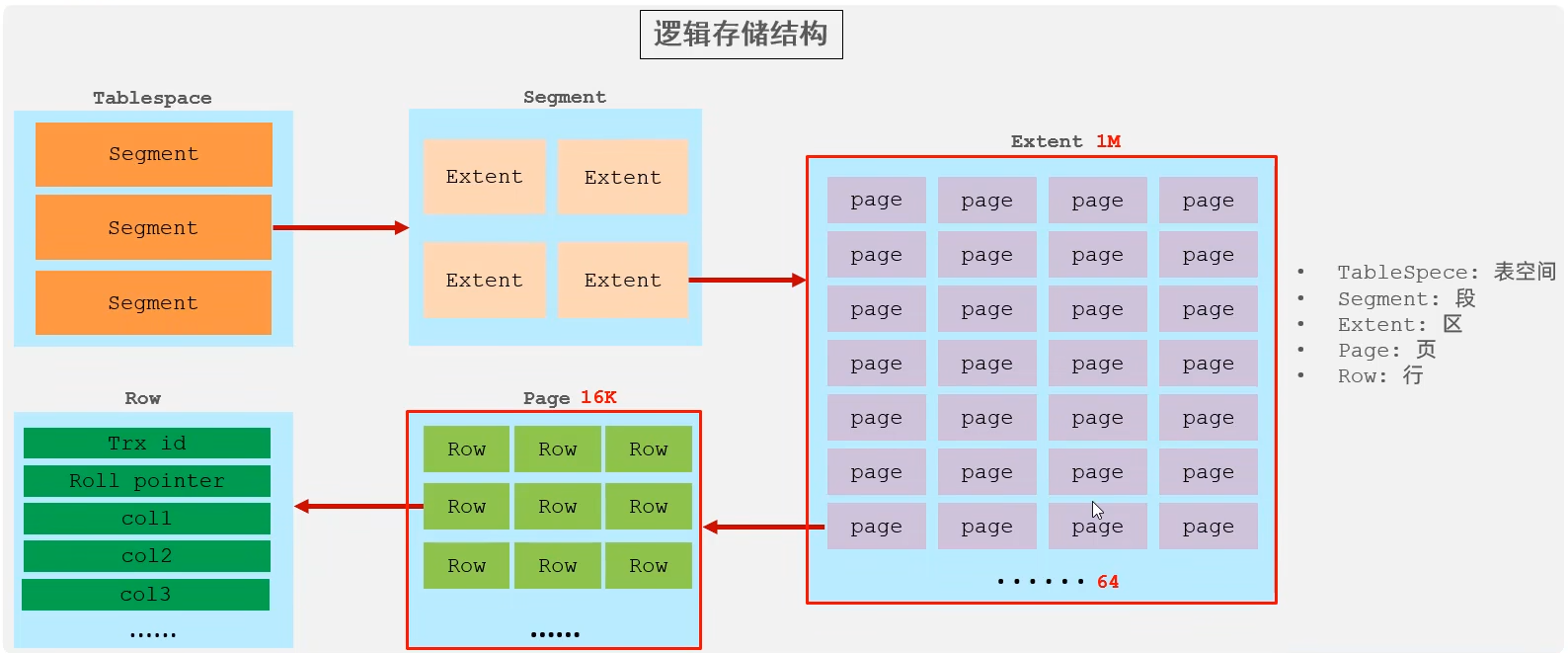
支持**外键**FOREIGN KEY:约束，保证数据的完整性和正确性.

**文件:**

xxx.ibd: xxx代表的是表名，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

**逻辑存储结构:**



索引是在引擎层, 不同的引擎对应不同的索引结构.

#### 7.3.2 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

#### 7.3.3 Memory

Memory引擎的表数据时存储在内存中的，由于受到硬件问题、或断电问题的影响，只能将这些表作为临时表或缓存使用。

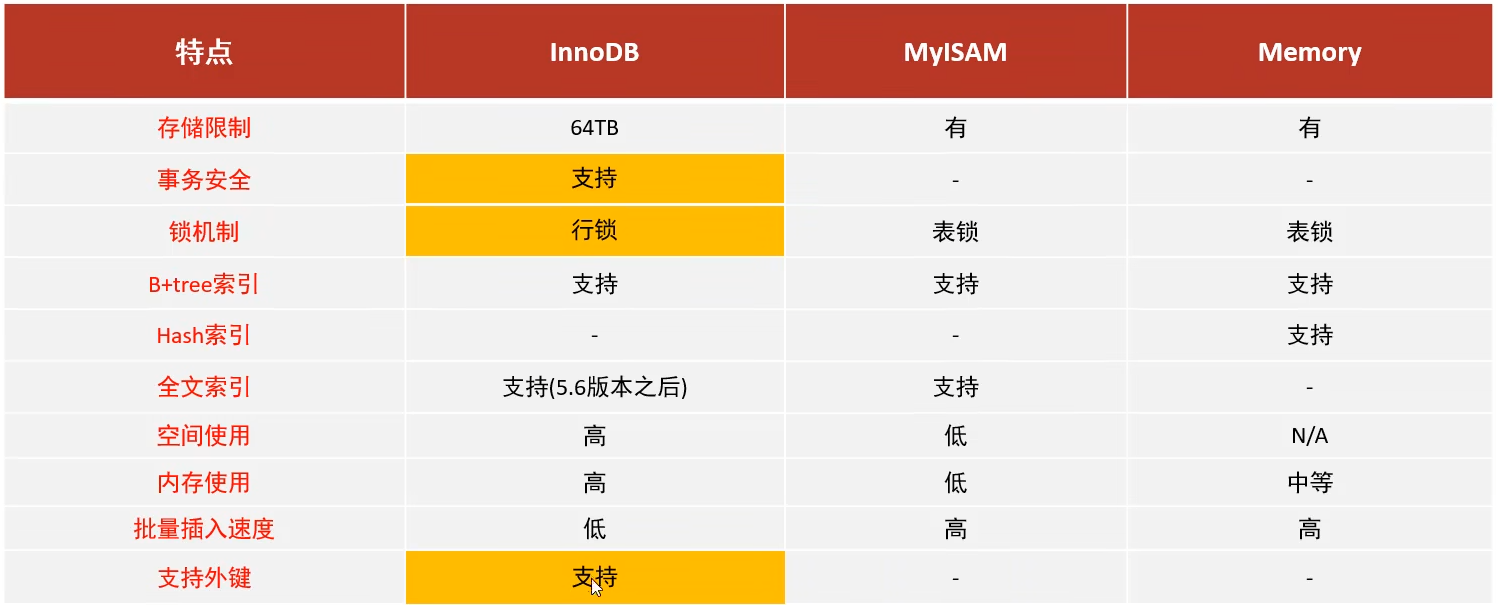
**特点:**

内存存放

hash索引(默认)

**文件:**

xxx.Sdi:存储表结构信息



### 7.4 存储引擎选择

**InnoDB**: 是Mysq的默认存储引擎，**支持事务、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之外，还包含很多的更新、删除操作，那么noDB存储引擎是比较合适的选择。

**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、并发性要求不是很高，那么选择这个存储引擎是非常合适的。适合非核心事务.

**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**。MEMORYI的缺陷就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。

## 8 索引

### 8.1 概述

索引(index)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找算法，这种数据结构就是索引。

**优缺点:**



数据库的查询占比较高, 劣势可以忽略不计.

### 8.2 结构

在引擎层实现, 不同的存储引擎有不同的结构.





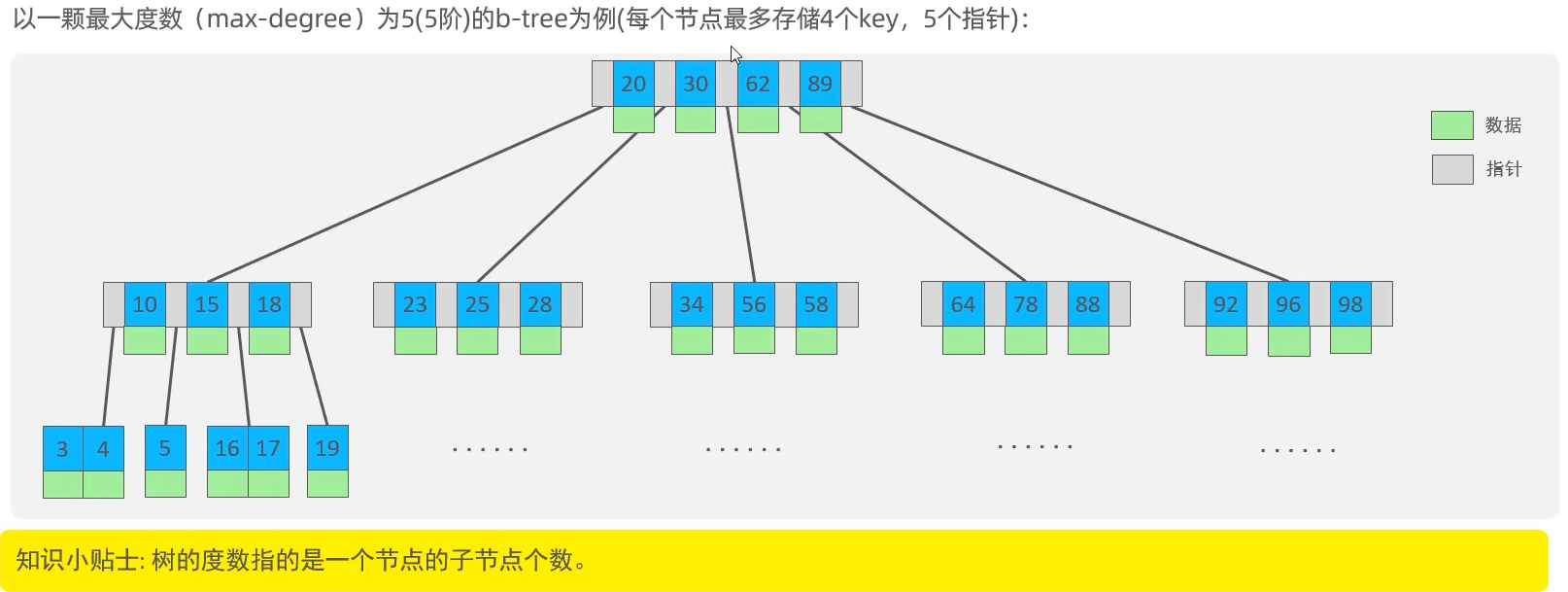
默认是B+Tree

**二叉树:**

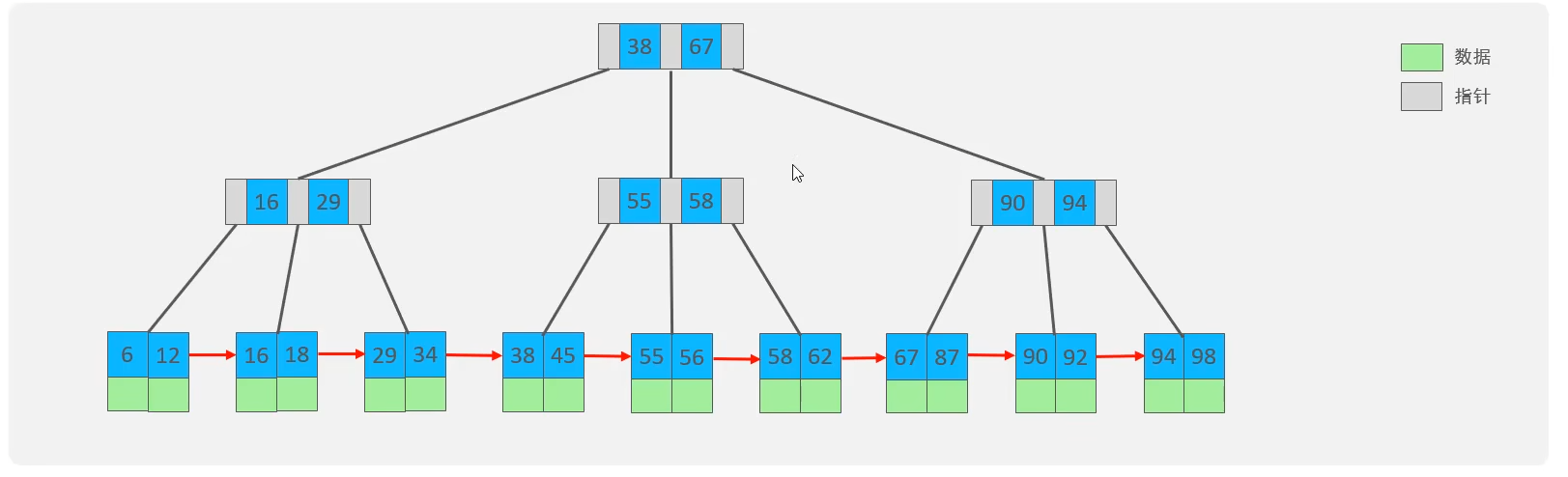
二叉树缺点：顺序插入时，会形成一个链表，查询性能大大降低。大数据量情况下，层级较深，检索速度慢。

红黑树：大数据量情况下，层级较深，检索速度慢。

**B-Tree(多路平衡查找树):**

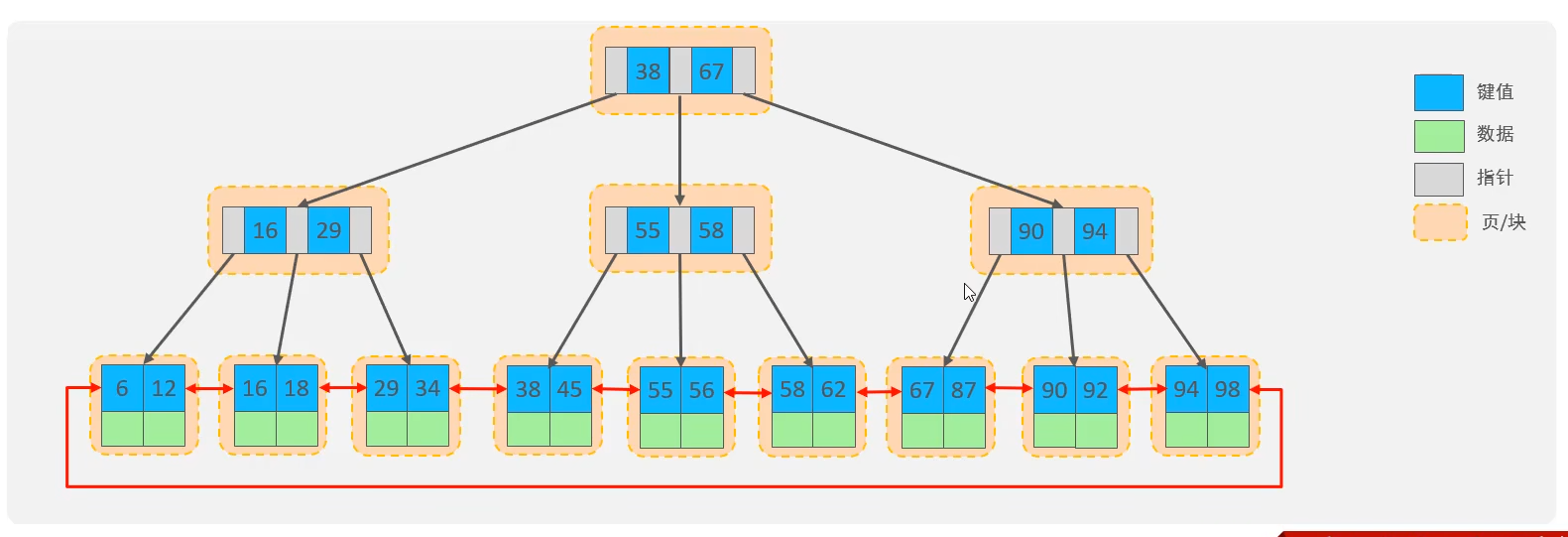


**B+Tree:**



所有元素都会出现在叶子节点, 形成单向链表. 底层上面的都是索引.

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双向链表指针，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。

>Hash索引特点

1. Hash索引只能用于对等比较(=，in),不支持范围查询(between,>,<,.)

2. 无法利用索引完成排序操作

3. 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

Memory引擎支持Hash, InnoDB具有自适应hash功能.

**为什么InnoDB选择B+Tree?**

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Hash索引，**B+tree支持范围匹配及排序操作**；

### 8.3 分类



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:



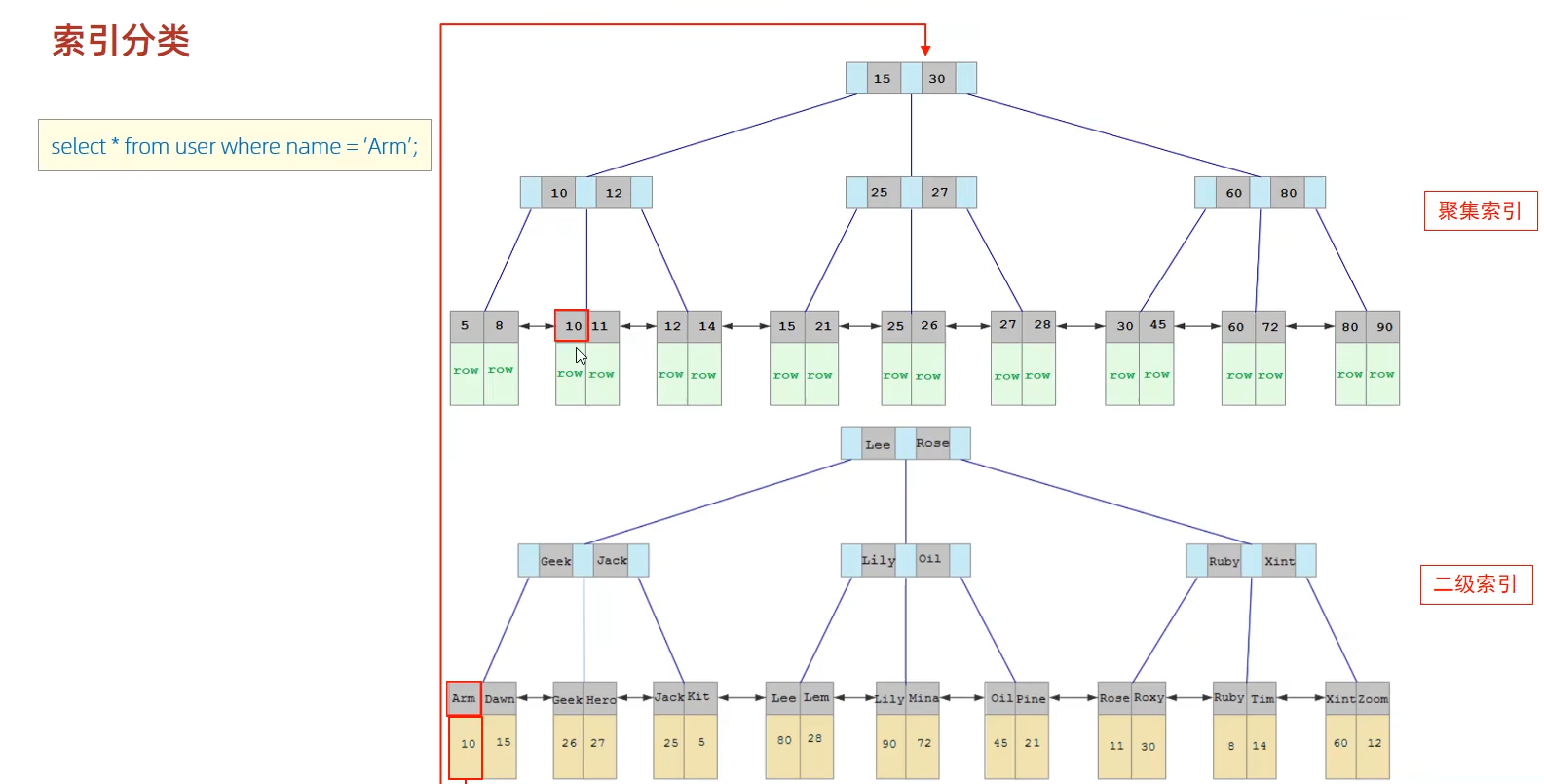
**聚集索引选取规则：**

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。

辅助/二级索引下面挂的是聚集索引id.



回表查询: 先查二级索引, 再回到聚集索引中获取所有数据.

**InnoDB主键索引的B+tree高度:**

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，主键即使为bigint,占用字节数为8。

高度为2:

n\*8+(n+1)\*6=16\*1024,算出n约为1170

1171\*16=18736

高度为3:

1171\*1171\*16=21939856

### 8.4 语法

创建索引

CREATE [UNIQUE | FULLTEXT] INDEX index\_name ON table\_name(index\_col\_name,...);

查看索引

SHOW INDEX FROM table\_name;

删除索引

DROP INDEX index\_name ON table\_name;

### 8.5 SQL性能分析

#### 8.5.1 SQL执行频率

MySQL客户端连接成功后，通过show [sessionl|global] status命令可以提供服务器状态信息。通过如下指令，可以查看当前数据库的INSERT、UPDATE、DELETE、SELECTI的访问频次：

SHOW GLOBAL STATUS LIKE 'Com\_\_\_\_\_\_\_\_’; --七个下划线

#### 8.5.2 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MySQL慢日志查询开关

slow query log=1

#设置慢日志的时间为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

#### 8.5.3 profile详情

show profiles能够在做SQL优化时帮助我们了解时间都耗费到哪里去了。通过have\_profiling参数，能够看到当前MySQL是否支持

**profile操作：**

SELECT @@have profiling

默认profiling是关闭的，可以通过set语句在session/globals级别开启profiling:

SET profiling = 1;

#查看每一条SQL的耗时基本情况

show profiles;

#查看指定query id的SQL语句各个阶段的耗时情况

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;

#### 8.5.4 explain执行计划

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 WHERE 条件;

各字段含义:

* id:

select查询的序列号，表示查询中执行select子句或者是操作表的顺序(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，常见的取值有SIMPLE(简单表，即不使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、al。 尽量往前优化.

* possible\_key

显示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。

* ref:

表示**查询时使用的列或常量的类型**，即哪些列与索引进行了比较。

* rows

MySQL认为必须要执行查询的行数，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。

### 8.6 索引使用

#### 8.6.1 最左前缀法则

如果索引了多列（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.

列之间查询时顺序变化, 但是不缺失, 还是可以走索引.

#### 8.6.2 失效

* 范围查询

联合索引中，出现范围查询（>，<），范围查询右侧的列索引失效.

>改为>=可以走索引.

* 索引列运算

不要在索引列上进行运算操作，索引将失效。

* 字符串不加引号

字符串不加引号造成隐式类型转换, 索引将失效. 能查出来, 但是没用到索引.

* 模糊查询

如果仅仅是尾部模糊匹配，索引不会失效。如果是头部模糊匹配，索引失效。

explain select\*from tb\_user where profession like'软件%';

* or连接的条件

用or分割开的条件，如果or前的条件中的列有索引，而后面的列中没有索引，那么涉及的索引都不会被用到。两侧都有索引才会生效, 使用的索引也是两个.

* 数据分布影响

如果MySQL评估使用索引比全表查询更慢，则不使用索引。

#### 8.6.3 索引提示

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。

use index:

explain select\*from tb\_user use index(idx\_user\_pro)where profession='软件工程';

ignore index:

explain select\*from tb\_user ignore index(idx user\_pro)where profession='软件工程';

force index:

explain select\*from tb\_user force index(idx\_user\_pro)where profession='软件工程';

#### 8.6.4 覆盖索引

**覆盖索引**（Covering Index）是指一个索引包含了查询所需要的所有列，从而避免回表操作。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够找到)，减少select\*。

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数据.

可以通过建立联合索引优化SQL语句. 把除id以外的所有colume联合建立起来就是覆盖索引.

#### 8.6.5 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**将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(column(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。

select count(distinct email)/count(\*) from tb user; --distinct 去重

select count(distinct substring(email,1,5))/count(\*) from tb\_user;

有一样的会进行徽标操作或精确匹配:

**回表操作（若为非覆盖索引）**：如果前缀索引是非覆盖索引（即索引中没有存储完整的行数据），则需要回表，通过主键或其他方式从原始数据表中读取完整的 email 字段值。

精确匹配：回表后，MySQL 将完整的 email 字段与查询条件中的值进行精确匹配。在上面的例子中，MySQL 会将 alice123@example.com 和 alice999@gmail.com 的完整字段与查询条件 alice123@example.com 进行比对，最终确定返回正确的记录。

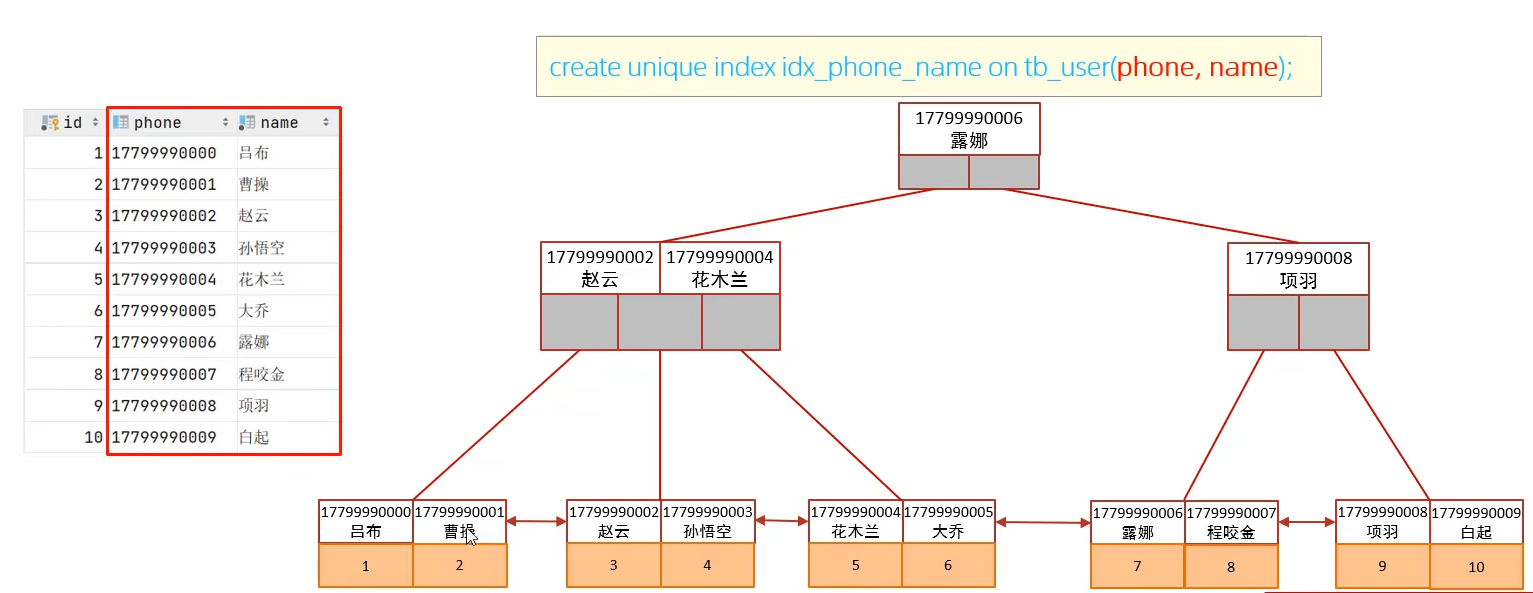
#### 8.6.6 单列&联合索引

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。

联合索引:



### 8.7 索引设计原则

1. 针对于数据量较大(100w)，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，字段的长度较长，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也就越大，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。

## 9 SQL优化

### 9.1 插入数据

* insert:

批量插入, 500-1000较好.

* 手动提交事务:

insert提交完后统一提交任务.

* 性能: 主键顺序插入>乱序
* 大批量插入数据:

#客户端连接服务端时，加上参数--local-infile

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;

#执行load指令将准备好的数据，加载到表结构中

load data local infile '/root/sql1.log' into table tb\_user fields terminated by ',' lines terminated by '\n';

### 9.2 主键优化

* InnoDB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)。

* 页(包含行)分裂:

页可以为空，也可以填充一半，也可以填充100%。每个页包含了2-N行数据(如果一行数据多大，会行溢出)，根据主键排列。

乱序插入可能出现页分裂.

* 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。

当页中删除的记录达到MERGE\_THRESHOLD(默认为页的50%),InnoDB会开始寻找最靠近的页（前或后）看看是否可以将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，在创建表或者创建索引时指定。

* 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。

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(无序, 长)做主键或者是其他自然主键，如身份证号。

业务操作时, 避免对主键的修改.

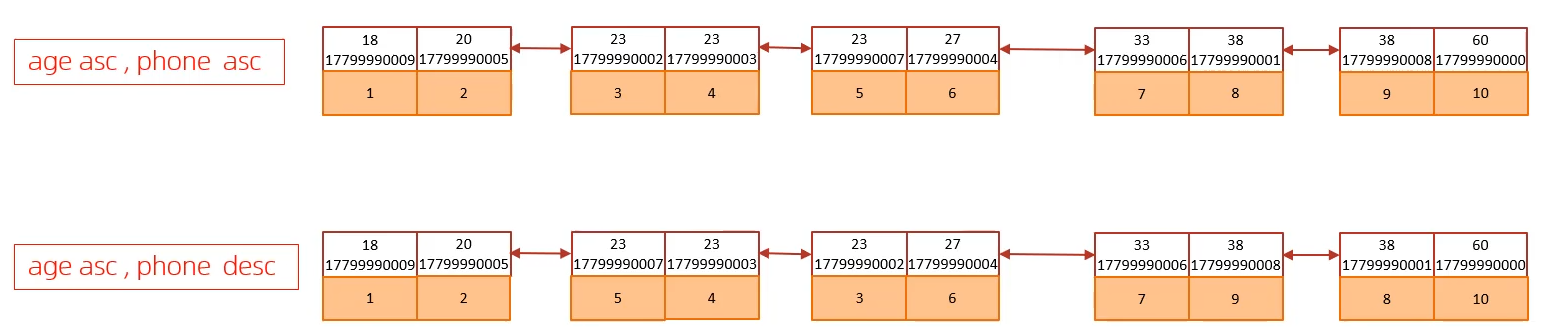
### 9.3 order by优化

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。

一个升序一个倒序的时候可以重新创建一个升一个倒的索引:

create index idx\_user\_age\_phone\_ad on tb\_user(age asc, phone desc);



**优化原则:**

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).

如果不可避免的出现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)。

### 9.4 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。

### 9.5 limit优化

分页查询.

一个常见又非常头疼的问题就是imit2000000,10,此时需要MySQL排序前2000010记录，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。

优化思路：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。

explain select \* from tb\_sku t, (select id from tb sku order by id limit 2000000,10) a where t.id=a.id;

### 9.6 count优化

MyISAM引擎把一个表的总行数存在了磁盘上，因此执行cout(\*)的时候会直接返回这个数，效率很高, 查询时没有where条件.

InnoDB引擎就麻烦了，它执行count(\*)的时候，需要把数据一行一行地从引擎里面读出来，然后累积计数。

count的几种用法:

count()是一个聚合函数，对于返回的结果集，一行行地判断，如果count函数的参数不是NULL, 累计值就加1，否则不加，最后返回累计值。

用法：count(\*)>count（主键）>count（字段）>count(1)

* count（主键）

InnoDB引擎会遍历整张表，把每一行的主键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nuul)。

* count（字段）

没有not null约束：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：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行，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。

最好的是自己计数, 使用其他架构.

按照效率排序的话，count（字段）<count(主键id)<count(1)≈count(\*),所以尽量使用count(\*)。

### 9.7 update优化

更新数据一定要**根据索引**. 有索引是行锁, 并且该索引不能失效. 没有索引升级为表锁, 并发性能下降.

## 10 视图/存储过程/触发器

## 11 锁

## 12 InnoDB引擎

## 13 MySQL管理