数据库

按照某种规则存放到计算机存储设备上可以被应用或用户访问的数据仓库

DBMS

数据库管理系统（DataBase Management System），指的是能够操作和管理数据库的软件

键

- 主键（primary key）
  - 一个列或多列的组合，其值能唯一地标识表中的每一行，通过它可强制表的实体完整性
  - 一个数据列只能有一个主键，且 **不能为空值（Null），默认唯一性约束，只有主键才能设置自动增长**
- 超键（super key）
  - 在关系中能 **唯一标识元素的属性集，超键包含候选键和主键**
- 候选键（candidate key）
  - 不含有多余属性的超键，是最小超键，即没有冗余元素的超键
- 外键（foreign key）
  - 在一个表中存在的另一个表的主键称此表的外键

更多：数据库原理—超键、候选键、主键、外键

表

表是一种结构化文件，用于存储某种特定类型的数据，数据在数据库中是以表的方式存储的，在数据库中可以创建多个表，同一数据库中表不能重名

列

表由列组成，列存储着表中某部分的信息。列是表中的一个字段，所有的表都是由一个或多个列组成的

行

表中的一个记录

数据类型

数据库中表的每一列都有相应的数据类型，限制了该列所存储的数据

SQL

结构化查询语言（Structure Query Language），是关系型数据语言的标准，无论哪一种数据库管理系统，都可以使用 SQL 来操作数据库中的数据。各个厂商在支持 SQL 标准的同时，在自己的数据库管理系统中做了一些扩展，这些扩展简称 **方言**

SQL 的分类

- DDL（Data Definition Language）：数据定义语言

    - 操作数据库对象：库、表、列等
- DML（Data Manipulation Language）：数据操作语言
    - 增删改数据库中的数据
- DCL（Data Control Language）：数据控制语言
    - 设置访问权限和安全级别
- DQL（Data Query Language）：数据查询语言
    - 查询数据库中的数据

完整性

- 实体完整性：指表中行的完整性，主要用于 **保证操作的数据非空、唯一且不重复。**即实体完整性要求 **每个关系有且仅有一个主键**
- 域完整性：指数据库表中的 **列必须满足某种特定的数据类型或约束**，其中约束又包括取值范围、精度等规定
- 参照完整性：属于表间规则，对于永久关系的相关表，在更新、插入或删除记录时，如果只改其中一个表，就会影响数据的完整性
- 自定义完整性：对数据表中 **字段属性的约束**，包括字段的值域、类型等约束

三大范式

- 第一范式（1NF）：数据表中的每一列都必须是不可拆分的最小单元，即确保每一列的原子性
- 第二范式（2NF）：满足 1NF 后，要求表中的所有列，都必须依赖于主键，而不能有任何一列与主键没有关系，即一个表只描述一件事情
- 第三范式（3NF）：满足 2NF 后，要求表中不能包含其它表的非主键字段，属性不依赖于其它非主属性

三大范式的区分

- 1NF：字段不可分
- 2NF：有主键，非主键字段依赖主键
- 3NF：非主键字段不能相互依赖

第一范式和第二范式在于有没有分出两张表，第二范式是说一张表中包含了所种不同的实体属性，那么要必须分成多张表，第三范式是要求已经分成了多张表，那么一张表中只能有另一张表中的主键，而不能有其他的任何信息

五大约束

- PRIMARY KEY：设置主键约束
- UNIQUE：设置唯一性约束，不能有重复值
- DEFAULT：默认值约束
- NOT NULL：设置非空约束，该字段不能为空
- FOREIGN KEY：设置外键约束

数据类型

- 整型：tinyint、smallint、mediumint、int、bigint
- 浮点型：float、double
    - **可以指定列宽**，如 double(5,2) 表示最多有 5 位，其中必须有 2 位小数
- 字符串
    - char：固定长度，如 char(10)，如果不足 10 位则会自动补足 10 位

    - varchar：可变长度，如 varchar(10)，如果不足 10 位不会补足，**性能不如 char 高**
    - text：适用于大文本内容
- 日期与时间
    - date：日期，格式为 `yyyy-MM-dd`
    - time：时间，格式为 `hh:mm:ss`
    - timestamp：时间戳，格式为 `yyyy-MM-dd hh:mm:ss`，**会自动赋值**
    - datetime：日期时间，格式为 `yyyy-MM-dd hh:mm:ss`

存储过程

存储过程就是作为可执行对象存放在数据库中的一个或多个 SQL 命令，其实就是能完成一定操作的一组 SQL 语句

优点

- **可以将代码封装，并保存在数据库中**
- **可以回传值，并可以接受参数**
- 无法使用 select 指令来运行，因为它是子程序，与查看表，数据表或用户定义函数不同
- 可以用在数据检验，强制实行商业逻辑等
- 存储过程是一个预编译的代码块，**执行效率比较高**
- 一个存储过程替代大量的 SQL 语句 ，可以 **降低网络通信量，提高通信速率**

缺点

- 不同的数据库不通用，**难以维护**
- 性能调校与撰写，受限于各种数据库系统
- 业务逻辑放在数据库上，难以迭代

```
delimiter //    # 定义结束符，确保在输入分号时，可以继续进行输入
create procedure pwhile()
begin
declare i int;  # 定义变量
declare sum int;
set i = 0;  # 为变量赋值
set sum = 0;
while i <= 10 do    # while循环
    set sum = sum + i;
    set i = i + 1;
end while;
select sum; # 显示sum的值
end
//  # 结束

call pwhile();  # 运行存储过程
```

```
# in: 向过程里传参数
delimiter //
create procedure pin(in sb int)
begin
select * from bank where id < sb;
end
//

# 调用含有in类型参数的存储过程
set @sb = 100;
call pin(@sb);
```

```
# out: 向过程外传参数
delimiter //
create procedure pout(out shit varchar(20))
begin
select bankname from bank where id = 1;
end
//

set @shit = '';
call pout(@shit);
```

```
## in和out合用
delimiter //
create procedure pinout(out fuck vatchar(20),in shit int)
begin
select bankname into fuck from bank where id = shit;
end
//

set @fuck = '';
set @shit = 1;
call pinout(@fuck,@shit);
select @fuck;
```

drop、delete、truncate 的区别

delete	drop
属于 DML	属于 DDL
不会自动提交	执行后会自动提交
事物提交后才生效，**可回滚**	不能回滚
会触发相应的 trigger	不会触发 trigger
可带 where	不可带 where
表结构在，删除部分或全部表内容	**表结构和内容删除**
速度最慢	速度最快

- truncate 与 drop 类似，区别是删除时删除的是 **表内容**，三者中速度中等
    - 如果是整理表内部的碎片，可以用 truncate 跟上 reuse stroage，再重新导入或插入数据

存储引擎

采用不同的技术将数据存储在文件或内存中，不同的技术有不同的存储机制。**存储引擎是 MySQL 特有的**，在不同的业务场景下选择不同的存储引擎，这样能够发挥 MySQL 的最佳性能

常用的存储引擎

- MyISAM：节省数据库空间，适用于数据读远大于修改
- InnoDB：支持事务，适用于如果数据修改较多时
- MEMORY：存储在内存中，速度快

MyISAM 与 InnoDB 的区别

	MyISAM	InnoDB
事务	不支持	支持
外键	不支持	支持
行锁	不支持，**只支持表锁**	支持
全文索引	支持	不支持
热备份	不支持	支持
count()	**直接返回值，因为保存了表的行数**	**先遍历表，再返回值**

视图

有的时候，我们可能只关心一张表中的某些字段，不需要查看全部的字段。使用视图就可以只关注需要的数据，同时，还可以保证数据表一些保密的数据不会泄露出来

视图其实就是一个 **查询结果**，基于查询的一种 **虚表**，视图可以 **隐藏表的实现细节**，即将查询出来的数据进行封装。使用视图可以让我们专注与逻辑，**但不提高查询效率**

- 视图建立在已有表的基础上，这些表被称为 **基表**，一个基表可以有 0 个或多个视图
- **向视图提供数据内容的语句为 SELECT 语句，可以将视图理解为存储起来的 SELECT 语句**
- **视图没有存储真正的数据，真正的数据还是存储在基表中**，视图只是向用户提供基表数据的 **另一种表现形式**
- 程序员虽然操作的是视图，但最终视图还会转成操作基表

```
# 创建视图
create view [视图名称] as [查询语句];

# 修改视图
alter view [视图名称] as [查询语句];

# 删除视图
drop view if exists [视图名称];
```

字符集与字符序

字符集（character set）

定义了字符以及字符的编码，MySQL 默认的字符集为 **latin1**

字符序（collation）

定义了字符的比较规则。每种字符集都可能有多种校对规则，并且都有一个默认的校对规则，并且每个校对规则只是针对某个字符集，和其他的字符集没有关系

- **每个字符集都有默认的字符序，一个字符集对应至少一种字符序，两个不同的字符集不能有相同的字符序**

```
# 查看所有字符集
show character set;

# 查看指定字符集
show character set where charset = 'utf8';

# 查看当前使用的字符集
show variables like 'character%';

# 查看所有字符序
show collation;

# 查看当前使用的字符序
show variables like 'collation%';
```

事务（Transaction）

可以 **保证多个操作的原子性**，对于数据库来说，事务可以保证一系列操作要么全成功，要么全失败，通常一个事务对应一个完整的业务

事务的四大特征 ACID

- 原子性（Atomicity）：事务开始后所有操作，要么全部做完，要么全部不做，不可能停滞在中间环节。事务执行过程中出错，会回滚到事务开始前的状态
- 一致性（Consistency）：事务的开始和结束，数据都必须保持一致状态
- 隔离性（Isolation）：同一时间，只允许一个事务请求同一数据，不同的事务互不影响
- 持久性（Durability）：事务完成之后，事务对数据库所作的更改将持久地保存在数据库中，不会被回滚

事务的隔离级别

- 读未提交（Read uncommitted）：一个事务读取到另外一个事务未提交的数据，读取到的数据叫做脏数据，一般只是理论上存在，数据库的默认隔离级别都高于该级别
    - 事务隔离的最低级别，任何情况都无法保证
- 读已提交（Read committed）：事务中的修改提交后，其他事务才可以看得到
    - 可避免脏读
- 重复读（Repeatable read）：保证了一个事务不会修改已经由另一个事务读取但未提交的数据，**MySQL 默认的隔离级别**
    - 可避免脏读、不可重复读
- 串行化（Serializable）：保证在 **同一个时间点上只有一个事务操作数据库**，但是这种级别一般很少使用，因为**吞吐量太低，用户体验不好**
    - 可避免脏读、不可重复读、幻读

并发事务带来的问题

- 脏读：一个事务读取到另外一个事务未提交的数据
- 不可重复读：一个事务读取到另一个事务已提交的数据，导致多次读取同一数据时，数据内容不一致
- 幻读：一个事务内读取到另一个事务插入的数据，导致前后多次读取，数据总量不一致
- 更新丢失：当多个事务更新同一数据时，由于不知道其他事务的存在，就会发生丢失更新问题，最后的更新覆盖了其他事务所做的更新

不可重复读和幻读的区别

- 不可重复读是读取了其他事务更改的数据，针对 update、delete 操作
    - 解决：使用行级锁，锁定该行，事务多次读取操作完成后释放锁，之后才允许其他事务更改刚才的数据
- 幻读是读取了其他事务新增的数据，针对 insert 操作
    - 解决：使用表级锁，锁定整张表，事务多次读取数据总量之后释放锁，之后才允许其他事务新增数据

更多：对于脏读，不可重复读，幻读的一点理解、数据库并发事务存在的问题

事务控制语句

```
# 开启事务
start transaction;
# 或者
begin;

# 提交
commit;
```

```
# 或者
commit work;

# 回滚
# 结束用户的事务，并撤销正在进行的所有未提交的修改
rollback;
# 或者
rollback work;

# 创建保存点
savepoint [保存点];

# 删除保存点
release savepoint [保存点];

# 回滚到保存点
rollback savepoint [保存点];

# 查看当前会话的隔离级别
select @@tx_isolation;

# 自动提交
# 0表示禁止，1表示开启
set autocommit = 0|1

# 设置事务隔离级别
# global全局，session本次会话
set [global | session] transaction isolation level [read uncommitted | read committed
| repeatable read | serializable];
```

触发器（trigger）

监视某种情况，并触发某种操作，它是一种保证数据完整性的方法，它是与表事件相关的特殊的存储过程，它的执行而是由事件来触发，当对一个表进行增删改操作时就会激活执行

不建议使用

非常消耗资源，如果使用的话，确定它是非常高效的，增删改非常频繁的表上不要使用触发器，不需要的触发器应及时删除。**触发器的功能基本都可以用存储过程来实现**

创建触发器

```
# trigger_time，指定了触发执行的时间，在事件之前或是之后，参数：before、after
# 触发事件，满足相应条件时触发，参数：insert、update、delete
# for each row，所有记录的操作满足条件都会触发该触发器，即触发器的触发频率是针对每一行数据触发一次
# 触发顺序（MySQL5.7+），定义多个触发器时，选择触发器执行的先后顺序，参数：follows、precedes
create trigger [触发器名]
[触发时间] [触发事件]
on [表名] for each row [触发顺序]
[执行语句];

# 多个执行语句的触发器
```

```
create trigger [触发器名]
[触发时间] [触发事件]
on [表名] for each row [触发顺序]
begin
    [执行语句]
end

# 查看触发器
show triggers;

# 删除触发器
drop trigger;
```

new 与 old

对同一个表相同触发时间的相同触发事件，只能定义一个触发器，可以使用 new 和 old 来引用触发器中发生变化的记录内容

- insert 型触发器
  - new 用来表示将要或已经插入的新数据
- update 型触发器
  - new 用来表示将要或已经被修改的新数据，old 用来表示将要或已经被修改的原数据
- delete 型触发器
  - old 用来表示将要或已经被删除的原数据

更多：MySQL触发器trigger的使用

索引

索引相当于一本字典目录，**能够提高数据库的查询效率**，表中每一个字段都可添加索引。**主键会自动添加索引**，所以查询时 **尽量使用主键查询，效率高**

为什么能够提高查询速度

索引的本质上是一个 **存储列值的数据结构。**如果在某列上使用了 B+ 树索引，那么这些列值在索引中是被排过序的，**有序的值** 是索引能提高查询性能的主要原因。索引就是通过事先排好序，从而在查找时可以应用二分查找等高效率的算法。一般的顺序查找，复杂度为 O(n)，而二分查找复杂度为 O(log2n)。当 n 很大时，二者的效率相差及其悬殊

为什么能够降低增删改速度

B+ 树是一颗平衡树，如果我们对这颗树增删改的话，那肯定会 **破坏它的原有结构。要维持平衡树，就必须做额外的工作。**正因为这些额外的 **工作开销**，导致索引会降低增删改的速度

索引的优缺点

优点

- 加快数据的检索速度
- 通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性
- 可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义

- 在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间
- 通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能

缺点

- 创建索引和维护索引要耗费时间
- 索引需要 **占物理空间**，除了数据表占数据空间之外，每一个索引还要占一定的物理空间。如果要建立聚簇索引，那么需要的空间就会更大
- 索引降低了增删改等维护任务的速度，因为大部分数据更新需要同时更新索引

索引的使用场景

什么时候创建索引

- 经常需要查询的表
- 数据特别多，数据分布范围广的表
- 经常使用 where 子句中的列，加快条件的判断速度
- 在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构
- 经常用于连接的列，这些列主要是一些外键，可以加快连接的速度
- 经常需要根据范围进行搜索的列，因为索引已经排序，其指定的范围是连续的
- 经常需要排序的列，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间

什么时候不要创建索引

- 经常需要增删改的表，很少需要查询的表
- 数据特别少的表，大部分情况下简单的全表扫描比建立索引更高效
- 但是对于特大型的表，建立和维护索引的代价将会随之增长。这种情况下，需要用到一种技术可以直接区分出需要查询的一组数据，而不是一条记录一条记录地匹配，例如可以使用分区技术
- 列名不经常作为连接条件或出现在 where 子句中
- 含有 text、image、bit 数据类型的列，这些列的数据量要么相当大，要么取值很少
- 当修改性能远远大于检索性能时，不应该创建索引。这是因为，**修改性能和检索性能是互相矛盾的**

索引的种类

- 单列索引：**一个索引只包含单个列**，但一个表中可以有多个单列索引
    - 普通索引：MySQL 中基本索引类型，没有什么限制，**允许在定义索引的列中插入重复值和空值**，纯粹为了查询数据更快一点
    - 唯一索引：索引列中的值必须是唯一的，但是 **允许为空值**
    - 主键索引：为表定义一个主键将自动创建主键索引，是一种特殊的唯一索引，**不允许有空值**
- 组合索引：在表中的多个字段组合上创建的索引，只有在查询条件中使用了这些字段的左边字段时，索引才会被使用，使用 **组合索引时遵循最左前缀集合**，即最左优先
- 全文索引：只有在 MyISAM 引擎上才能使用，**只能在 CHAR，VARCHAR，TEXT 类型字段上使用**全文索引，在一堆文字中，通过其中的某个关键字等，就能找到该字段所属的记录行
- 空间索引：空间索引是对空间数据类型的字段建立的索引，MySQL 中的空间数据类型有四种：GEOMETRY、POINT、LINESTRING、POLYGON

最左匹配原则

- 最左优先，以最左边的为起点任何连续的索引都能匹配上

- 索引可以简单如一个列 `(a)`，也可以复杂如多个列 `(a, b, c, d)`，即 **联合索引**。如果是联合索引，那么 key 也由多个列组成，同时，索引只能用于查找 key 是否 **存在（相等）**，遇到范围查询 `>、<、between、like` 等就 **不能进一步匹配** 了，后续退化为线性查找。因此，**列的排列顺序决定了可命中索引的列数**

聚集和非聚集索引

聚集索引：表中各行的物理顺序与键值的逻辑（索引）顺序相同，每个表只能有一个

非聚集索引：也叫做二级索引，非聚集索引指定表的逻辑顺序。数据存储在一个位置，索引存储在另一个位置，索引中包含指向数据存储位置的指针。可以有多个，小于 249 个

简单概括：

- 聚集索引就是以 **主键** 创建的索引
- 非聚集索引就是以 **非主键** 创建的索引

区别

- 聚集索引在叶子节点存储的是 **表中的数据**
- 非聚集索引在叶子节点存储的是 **主键和索引列**
- 使用非聚集索引查询出数据时，**拿到叶子上的主键再去查到想要查找的数据**，拿到主键再查找这个过程叫做 **回表**
- 非聚集索引在建立的时候也 **未必是单列** 的，可以多个列来创建索引
    - 最左匹配原则
    - **创建多个单列（非聚集）索引的时候，会生成多个索引树**，所以过多创建索引会占用磁盘空间

覆盖索引

- 如果不是聚集索引，叶子节点存储的是 **主键 + 列值**
- 最终还是要回表，也就是要通过主键 **再查找一次**，这样就会比较慢
- 覆盖索引就是把要 **查询出的列和索引是对应的**，不做回表操作

```
# 例如已经创建了(username,age)的索引
select username,age from user where username = 'jiage' and age = 20
```

- 很明显地知道，我们上边的查询是走索引的，并且，**要查询出的列在叶子节点都存在**！所以，就不用回表了。所以，**能使用覆盖索引就尽量使用吧**

索引的类型

索引是在存储引擎层实现的，不是在服务器层实现的，所以不同存储引擎具有不同的索引类型和实现，索引能够轻易将查询性能提升几个数量级

B+Tree 索引

MySQL 存储引擎的 **默认** 索引类型。查找速度很快，因为不需要进行全表扫描，只需要对树进行搜索。除了用于查找，还可以用于排序和分组

可以指定多个列作为索引列，多个索引列共同组成键。适用于全键值、键值范围和键前缀查找，其中键前缀查找只适用于最左前缀查找。如果不是按照索引列的顺序进行查找，则无法使用索引

InnoDB 的 B+Tree 索引分为 **主索引和辅助索引。**主索引的叶子节点 data 域记录着完整的数据记录，这种索引方式被称为聚簇索引。因为无法把数据行存放在两个不同的地方，所以一个表只能有一个聚簇索引

辅助索引的叶子节点的 data 域记录着主键的值，因此在使用辅助索引进行查找时，需要先查找到主键值，然后再到主索引中进行查找

哈希索引

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，效率极高。在 MySQL 中只有 Memory 引擎显式支持哈希索引

InnoDB 存储引擎有一个特殊的功能叫 **自适应哈希索引**，当某个索引值被使用的非常频繁时，会在 B+Tree 索引之上再创建一个哈希索引，这样就让 B+Tree 索引具有哈希索引的一些优点，比如快速的哈希查找

- 在绝大多数需求为单条记录查询的时候，可以选择哈希索引，查询性能最快；其余大部分场景，建议选择 B+Tree 索引

缺点

- 无法用于排序与分组
- 只支持精确查找，无法用于部分查找和范围查找
- 不支持最左匹配原则
- 不支持范围查询
- 如果哈希冲突很多，查找速度会变得很慢
- 哈希索引只包含哈希值和行指针，而不存储字段值，所以不能使用索引中的值来避免读取行。不过，访问内存中的行的速度很快，所以大部分情况下这一点对性能影响并不明显

空间数据索引（R-Tree）

相对于 BTREE，RTREE 的优势在于范围查找。MyISAM 存储引擎支持空间数据索引，可以用于地理数据存储。空间数据索引会从所有维度来索引数据，可以有效地使用任意维度来进行组合查询。**RTREE 在 MySQL 很少使用，仅支持 geometry 数据类型。**必须使用 GIS 相关的函数来维护数据

全文索引

全文索引使用倒排索引实现，它记录着关键词到其所在文档的映射。MyISAM 存储引擎支持全文索引，用于查找文本中的关键词，而不是直接比较索引中的值。查找条件使用 MATCH AGAINST，而不是普通的 where。它的出现是为了 **解决针对文本的模糊查询效率较低** 的问题

- InnoDB 存储引擎在 MySQL 5.6.4 版本中开始支持全文索引
- 可以在 `create table, alter table, create index` 使用，不过目前只有 char、varchar，text 列上可以创建全文索引

MyISAM 与 InnoDB

MyISAM

B+Tree 叶节点的 data 域存放的是数据记录的地址。在索引检索的时候，首先按照 B+Tree 搜索算法搜索索引，如果指定的 key 存在，则取出其 data 域的值，然后以 data 域的值为地址读取相应的数据记录。这被称为非聚簇索引

InnoDB

其数据文件本身就是索引文件。相比 MyISAM，索引文件和数据文件是分离的，其表数据文件本身就是按 B+Tree 组织的一个索引结构，树的叶节点 data 域保存了完整的数据记录。这个索引的 key 是数据表的主键，因此 InnoDB 表数据文件本身就是主索引。这被称为聚集索引。而其余的索引都作为辅助索引，辅助索引的 data 域存储相应记录主键的值而不是地址，这也是和 MyISAM 不同的地方。在根据主索引搜索时，直接找到 key 所在的节点即可取出数据；在根据辅助索引查找时，则需要先取出主键的值，再走一遍主索引。因此，在设计表的时候，不建议使用过长的字段作为主键，也不建议使用非单调的字段作为主键，这样会造成主索引频繁分裂

操作索引语句

```
# 创建表时创建
CREATE TABLE `test` (
    `id` INT (11) NOT NULL AUTO_INCREMENT,
    ...
    PRIMARY KEY (`id`)
    INDEX [索引名] (字段名(长度))
) ENGINE = INNODB DEFAULT CHARSET = utf8;

# 增加索引
alter table [表名] add [index | unique | primary key | fulltext | spatial] [索引名] (字段名);

# 创建索引
create index [索引名] on [表名] (字段名);
create unique index [索引名] on [表名] (字段名);

# 删除索引
drop index [索引名] on [表名];
alter table [表名] drop index [索引名];

# 显示索引信息，可以通过添加\G来格式化输出信息。
show index from [表名];
```

索引优化

- 独立的列：在进行查询时，索引列不能是表达式的一部分，也不能是函数的参数，否则无法使用索引
- 多列索引：在需要使用多个列作为条件进行查询时，使用多列索引比使用多个单列索引性能更好
- 索引列的顺序：**让选择性最强的索引列放在前面**
    - 索引的选择性是指：不重复的索引值和记录总数的比值 `COUNT(DISTINCT col) / COUNT(*)`。最大值为 1，此时每个记录都有唯一的索引与其对应。选择性越高，查询效率也越高
- 前缀索引：对于 blob、text 和 varchar 类型的列，必须使用前缀索引，只索引开始的部分字符。对于前缀长度的选取需要根据索引选择性来确定
- 覆盖索引：索引包含所有需要查询的字段的值
    - 索引通常远小于数据行的大小，只读取索引能大大减少数据访问量
    - 如 MyISAM 等存储引擎在内存中只缓存索引，而数据依赖于操作系统来缓存。因此，只访问索引可以不使用系统调用，缩短时间
    - 对于 InnoDB 引擎，若辅助索引能够覆盖查询，则无需访问主索引

聚簇索引

聚簇索引并不是一种索引类型，而是一种数据存储方式。聚簇表示数据行和相邻的键值紧密地存储在一起，InnoDB 的聚簇索引在同一个结构中保存了 B+Tree 索引和数据行。因为无法把数据行存放在两个不同的地方，所以一个表只能有一个聚簇索引

优点

- 可以把相关数据保存在一起，减少 I/O 操作
- 数据访问更快

缺点

- 聚簇索引最大限度提高了 I/O 密集型应用的性能，但是如果数据全部放在内存，就没必要用聚簇索引
- 插入速度严重依赖于插入顺序，按主键的顺序插入是最快的
- 更新操作代价很高，因为每个被更新的行都会移动到新的位置
- 当插入到某个已满的页中，存储引擎会将该页分裂成两个页面来容纳该行，页分裂会导致表占用更多的磁盘空间
- 如果行比较稀疏，或者由于页分裂导致数据存储不连续时，聚簇索引可能导致全表扫描速度变慢

权限管理

MySQL 的权限是如何实现的

服务器首先会检查你是否允许连接，因为创建用户的时候会加上主机限制，可以限制成本地、某个 IP、某个 IP 段、以及任何地方等，只允许你从配置的指定地方登陆。然后，如果你能连接，MySQL 会检查你发出的每个请求，看你是否有足够的权限实施它

MySQL 的权限

usage		连接权限，建立新用户时，会自动授予该权限
create	数据库、表或索引	创建数据库、表或索引权限
drop	数据库或表	删除数据库或表权限
grant option	数据库、表或保存的程序	赋予权限选项
references	数据库或表	
alter	表	更改表，比如添加字段、索引等
delete	表	删除数据权限
index	表	索引权限
insert	表	插入权限
select	表	查询权限
update	表	更新权限
create view	视图	创建视图权限
show view	视图	查看视图权限
alter routine	存储过程	更改存储过程权限
create routine	存储过程	创建存储过程权限
execute	存储过程	执行存储过程权限
file	服务器主机上的文件访问	文件访问权限
create temporary tables	服务器管理	创建临时表权限
lock tables	服务器管理	锁表权限
create user	服务器管理	创建用户权限
process	服务器管理	查看进程权限
reload	服务器管理	执行 flush, refresh, reload 等命令的权限
replication client	服务器管理	复制权限
replication slave	服务器管理	复制权限
show databases	服务器管理	查看数据库权限
shutdown	服务器管理	关闭数据库权限
super	服务器管理	执行kill线程权限

如何使用

- **只授予能满足需要的最小权限**，防止用户干坏事
- 创建用户的时候 **限制用户的登录主机**，一般是限制成指定 IP 或者内网 IP 段
- 初始化数据库的时候删除没有密码的用户。安装完数据库的时候会自动创建一些用户，这些用户默认没有密码
- 为每个用户设置满足密码复杂度的密码
- 定期清理不需要的用户，回收权限或者删除用户

```
# 创建一个只允许从本地登录的超级用户jack，并允许将权限赋予别的用户
grant all privileges on *.* to jack@'localhost' identified by "jack" with grant option;

# all privileges：表示所有权限，也可以使用select、update等权限
# on：指定权限针对哪些库和表
# *.*：中前面的*号用来指定数据库名，后面的*号用来指定表名
# to：将权限赋予某个用户
# jack@'localhost'：表示jack用户，@后面接限制的主机，可以是IP、IP段、域名以及%，%表示任何地方
# identified by:指定用户的登录密码
# with grant option：表示该用户可以将自己拥有的权限授权给别人
# 可以使用grant重复给用户添加权限，权限叠加

# 刷新权限，使权限生效
flush privileges;

# 查看当前用户的权限
show grants;

# 查看某个用户的权限
show grants for 'jack'@'%';

# 回收权限
revoke delete on *.* from 'jack'@'localhost';

# 删除用户
select host,user,password from user;
drop user 'jack'@'localhost';

# 重命名用户
rename user 'jack'@'%' to 'jim'@'%';
```

更多：MySQL之权限管理

更多

- InnoDB数据页结构
- MySQL的索引
- MySQL索引背后的数据结构及算法原理
- 索引的使用、原理和设计优化
- 聚集索引与非聚集索引的总结