数据库（核心版）

MRL Liu

2022 年 03 月 07 日

如何学习数据库？

数据库知识点非常繁多，但笔者认为所有的知识点可以分为**核心知识点**（数据库知识的基本知识）和**非核心知识点**（为了提高开发效率或者性能的额外机制，锦上添花）。其中核心知识点有：**SQL 操作**、**MySQL 架构**、**存储引擎架构**、**索引原理**、**事务原理**（锁、日志+MVCC）；非核心知识点有：**表连接**、**视图**、**储存过程**、**触发器**、**主从复制**、**读写分离**等。

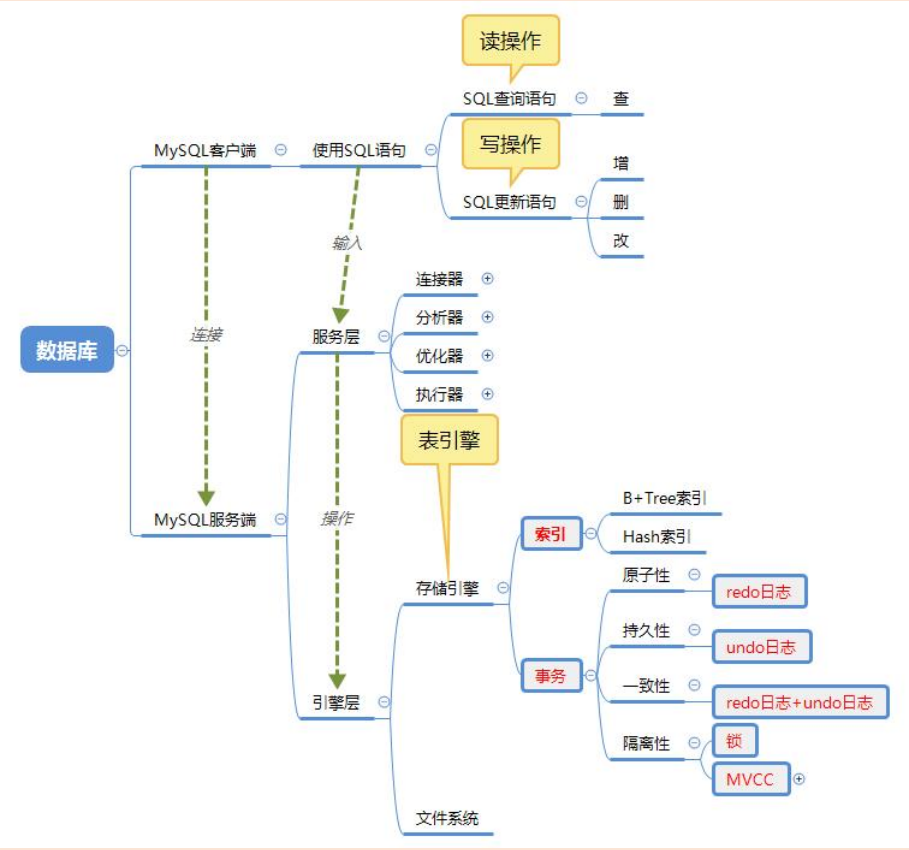
接下来简单介绍下核心知识点之间的关系：

SQL 语句从执行原理上可以整体分为 **SQL 查询语句**（查）和 **SQL 更新语句**（增删改），前者本质是读操作，后者本质是写操作。

MySQL 的服务端程序总体可以分为**服务层**（连接器+查询缓存+分析器+优化器+执行器）和**引擎层**（存储引擎+文件系统），其中**引擎层**是核心，它直接决定了数据在磁盘中如何存储和读取。数据库中经常提到的**索引**、**事务**、**日志**、**锁**、**MVCC** 其实都是由 MySQL 的**引擎层**实现的。大部分材料都默认讲解 MySQL 的默认存储 **InnoDB 引擎**。

索引是**存储引擎**中第一个重要的概念，它本质是数据结构，决定了在内存和磁盘中如何组合行记录，这决定了逻辑上如何读取这些数据。

事务是**存储引擎**为了解决**并发场景**提出的概念，由于数据库可能同时被多个用户操作，就会出现并发事务。**我们向 MySQL 提交的任何 SQL 语句都是包含在一个事务中的**，只不过是**显示提交**或者**隐式提交**的区别。事务有四大特性：**原子性**+**持久性**+**一致性**+**隔离性**。为了实现事务的这四大特性，InnoDB 引擎在内部使用了**日志**、**锁**和 **MVCC**。



从总体上看，数据库的操作是 **SQL 语言**，SQL 语言内部执行主要依靠**存储引擎**，**存储引擎**主要实现了**索引**和**事务**，为了实现事务的**四大特性**，存储引擎使用了**日志**、**锁**、**MVCC**。

一、数据库基础篇

1、数据库的基础辨析

为什么要使用数据库？

数据库技术用来帮助技术人员管理**海量数据**，用来**有组织地存储数据**。对比下常见的数据保存方式：

保存方式	优点	缺点	应用场景
保存在内存	**存取速度快**	**数据无法永久保存**，程序运行结束，数据消失。	应用程序执行过程中读写的临时数据
保存在文件	**数据可永久保存**	在**查询海量数据时效率低；操作速度比内存操作慢，需要频繁的 IO 操作**，所以存储速度慢	适用于存储少量、不需要频繁更新、安全要求较低的数据
数据保存在数据库	数据可**永久保存**； 数据**安全性高**； 可**高效管理海量数据**；	数据库移植不方便； 不支持集群； 不擅长业务逻辑的处理	适用于存储**海量**、可能**需要频繁更新**、**安全要求较高的数据**，一般用于后端存储用户数据

请简单介绍下数据库、数据库管理系统、SQL、MySQL、Navicat 的区别？

数据库（Database）是保存**有组织的数据**的**容器**（通常是一个文件或一组文件）。

数据库管理系统（DataBase- Management System，DBMS）是操作和管理**数据库**的大型软件。我们常说 XX 数据库，其实本质上是指 XX 数据库管理系统，使用的是一个软件。目前，较为流行的 DBMS 有 MySQL、SQL Server、Oracle 等。

SQL（Structured Query Language）是操作**关系型数据库**的结构化查询语言，请注意，SQL 不是某个数据库供应商独有的语言，几乎所有主流的数据库软件都使用相同的 SQL。

MySQL 是一种**关系数据库管理系统**（RDBMS），本质是一个软件，会为用户提供一个界面，只需单击一些按钮即可用于执行各种数据库操作。该软件底层使用 SQL 执行所有数据库操作。该软件是**开源免费**的。

Navicat、**DataGrip**、Sqlyog、是数据库的图形化软件，方便操作数据库。

非关系型数据库和关系型数据库区别，优势比较?

数据库有两种类型，分别是**关系型数据库**和**非关系型数据库**。

类型	定义	优点	缺点	常见
关系型数据库	建立在**关系模型基础**上，由多张能互相连接的**二维表**组成的数据库	使用表结构，格式一致，易于维护； **支持 SQL 语句**，可以满足各种复杂查询需求 **安全性高**，支持事务等安全机制	1）读写性能比较差； 2）建立在关系模型上，不可避免空间浪费； 3）固定的表结构，灵活度较低	MySQL，Microsoft SQL Server，Oracle，PostgreSQL 等
非关系型数据库又被称为 NoSQL（Not	以**对象**的形式存储数据的数据库	1）应用场景更广泛，存储数据的格式可以是 key-value、文档、图片等形式，； 2）可轻松进行海量数据的维护和处理；	1）不支持 SQL 语句； 2）安全性低，无法保证数据的完整性；	Neo4j，Redis，MongoDB 等

Only SQL)		3）**具有可扩展、高并发、高稳定性、成本低**的优势； 4）可以实现数据的分布式处理		

对于这两类数据库，对方的优势就是自己的弱势，反之亦然。

2、数据库的基础术语

关系数据库中的常见术语：表、列、行、字段、记录

表（table）是数据库中**存储某类数据**的**结构化文件**，数据库中的每个表的**表名都是唯一的**。表是二维的，所以有时也叫做**二维表**。

表由**列**（colomn）组织，每个**列**都有对应的**数据类型**（datatype），列也叫做**字段**。

表的数据是按**行**（row）插入的，**行**就是数据库中的一条**记录**（record）。

数据库中的约束是什么？

数据库的**约束**就是对**插入数据**进行**限定**，以保证数据的**有效性**和**完整性**。**约束**作用于表的**字段**，约束可以在**创建/修改**表时添加。数据库中主要有 6 大约束：

约束类型	含义	关键字
主键约束	保证该字段唯一且不重复，只能有一个	PRIMARY KEY
唯一约束	保证该字段的数据不能重复，可以有多个	UNIQUE
非空约束	限制该字段的数据不能为 null，可以有多个	NOT NULL
默认约束	插入数据时，如果未指定该字段的值，则采用默认值	DEFAULT
检查约束	保证字段值满足某一个条件	CHECK
外键约束	和其他表建立连接，保证数据一致性和完整性，可以有多个	FOREIGN KEY

外键约束有什么特点？

外键约束是用来在**两张表**之间**建立连接**。假设一个订单表和一张客户表，订单包含客户，显然订单表里需要有一个字段为“客户 ID”，设为外键约束。此时客户表就是**父表**（主表），订单表就是**子表**（从表），默认情况下，**父表不能删除子表中已引用的数据，子表不能添加父表中不存在的引用**。当然这种默认设置也可以更改，常见如下（熟读不记）：

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
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）

请比较下主键、联合主键、外键、索引、超键、候选键的区别

	定义	作用
主键	唯一记录一条数据的**字段**，**必须唯一且非空**。如果字段有多个则是**联合主键**（也叫**复合主键**）。	用来**唯一标识一条数据**，保证数据的**完整性**和**有效性**
外键	存储另一张表的主键的**字段**，**允许重复**，**可以为空**	用于**建立和其他表的联系**，保证数据的**完整性**和**有效性**
索引	存储引擎使用的**数据结构**，索引有很多类型，**一般不允许重复**，但可以为空。	用于**提升检索数据的效率**

你知道数据库中的多表关系有三类吗？可以分别列举一个例子吗？

	案例	**关系**	**实现**
一对一	用户与用户详情	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	在任意一方加入一个具有**唯一约束**的**外键**，关联另外一方的**主键**
一对多	部门与员工	一个部门对应多个员工，一个员工对应一个部门	在多的表中插入一个**外键**，指向一的一方的**主键**
多对多	学生与课程	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	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**

数据库的三大范式是什么？

数据库范式是设计数据库时需要遵循的规范。各种范式是条件递增的联系，越高的范式数据库冗余越小。常用的数据库**三大范式**为：

第一范式（1NF）：**每个列都不可以再拆分**，强调的是列的原子性。第一范式要求数据库中的表都是二维表。例如字段“地址”，如果业务中经常访问“地址”中的“城市”部分，则需要将“地址”拆分出省份、城市和详细地址等多个部分进行存储。

第二范式（2NF）：在第一范式的基础上，一个表中只能存储一种数据，不可以把多种数据存储在同一张数据库表中。**一个表必须有一个主键，非主键列完全依赖于主键**，而不能是依赖于主键的一部分（复合主键）。例如设计一个订单信息表，订单和商品之间可能是多对多的关系，所以可能需要将订单编号和商品编号作为数据库表的联合主键，其他列必须和依赖于联合主键整体，而不能只依赖于其中的订单编号部分。

第三范式（3NF）：在第二范式的基础上，**非主键列直接依赖于主键，不能间接依赖于其他非主键**。比如设计订单数据表时，客户编号应该作为一个外键和订单表建立相应关系，而不能在订单表中添加客户的其他信息（姓名、公司）等字段。

在设计数据库结构的时候，要尽量遵守三范式，如果不遵守，必须有足够的理由。比如性能。事实上我们经常会为了性能而妥协数据库的设计。

3、SQL 的基础语法

SQL 语言的四大分类是什么，分别起什么作用？

SQL 语言共分为四大类：

全称	**简写**	**说明**
数据**定义**语言	Data Definition Language,**DDL**	定义数据库对象（**数据库**、**表**、**字段**）
数据**控制**语言	Data Control Language,**DCL**	用来**管理用户**、**分配权限**、**设置属性**

数据**操作**语言	Data Manipulation Language,**DML**	用来对表中的数据进行**增删改**
数据**查询**语言	Data Query Language,**DQL**	用来**查询**中表的记录

1、数据定义语言 DDL。隐性提交的，不能 rollback，常见语句如下：

show databases/tables 展示所有数据库、所有表

create database/table/view/index/syn/cluster 创建表/视图/索引/同义词/簇

drop database/table/view/index/syn/cluster 删除表/视图/索引/同义词/簇

alter table 表名 rename to 修改表的名称

2. 数据控制语言 DCL。常见语句如下：

如：**grant**：授权权限。**rollback** [work] to [savepoint]：回退到某一点。

commit [work]：提交事务。可以显式提交、隐式提交及自动提交。

3. 数据操纵语言 DML。主要有三种形式：

插入：**insert into** 表名 (字段名 1, 字段名 2, ...) **values** (值 1, 值 2, ...);

更新：**update** 表名 **set** 字段名 1 = 值 1, 字段名 2 = 值 2, ... [**where** 条件];

删除：**delete from** 表名 [**where** 条件];

4. 数据查询语言 DQL。DQL 的语句由许多子句块组成：

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

你知道**多表查询**/**表连接**吗？请问什么是**交叉连接**、**笛卡尔积**、**内连接**、**外连接**、**自连接**？

单表查询是用 **SQL 语句**从**一张表**中进行查询（较为简单）；**多表查询（也叫表连接）**就是用 **SQL 语句**从**多张表**中进行查询。**笛卡尔积**就是出现在**多表查询**中，如下：

假设有两张表：员工表 employee 和部门表 dept，employee 表中字段 dept 和 dept 的 id 主键是**外键约束**。我们暂将含有**外键**的子表 employee 称为**左表**/**驱动表**，父表 dept 称为**右表**/**被驱动表**。

交叉连接：生成笛卡尔积（不使用任何匹配或者选取条件）

```
-- 省略写法：
select * from employee, dept;
-- 正式写法：
select * from employee,cross join dept;
```

注意：笛卡尔积：两个集合 A 集合和 B 集合的所有组合情况（假如表 A 有 5 条记录，表 B 有 6 条记录，则笛卡尔积后就有 30 条记录）

所以上述交叉连接会返回许多无效的记录，所以需要消除无效的笛卡尔积，要加 where 条件限制记录：

select * from employee, dept **where employee.dept = dept.id;**

内连接：查询的是左右两张表交集的部分，即左表和右表中相交的记录

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

外连接分为**左外连接**和**右外连接**

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

```
-- 查询员工所有信息及关联的部门的名称
-- 左外连接写法：含 e.dept 为空的情况
```

```
select e.name, d.name from employee as e left outer join dept as d on e.dept = d.id;
-- 右外连接写法：含 dept 为空的情况
select d.name, e.* from employee as e right outer join dept as d on e.dept = d.id;
```

自连接：当前表与自身表的连接查询，自连接必须使用表别名。

假设 employee 中存在字段 manager 记录每个员工的领导，存放每个员工 id，这时 employee 就自身与自身发生了连接：

```
-- 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;
```

请注意：长连接、短连接在数据库中和多表查询没关系，长连接指的是 MySQL 客户端长时间连接 MySQL 服务端。

你知道 select 语句中不同子句的执行顺序吗？

SQL 查询语句在编写时按照如下顺序编写：

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

但是 SQL 查询语句在 MySQL 执行时按照如下顺序执行：

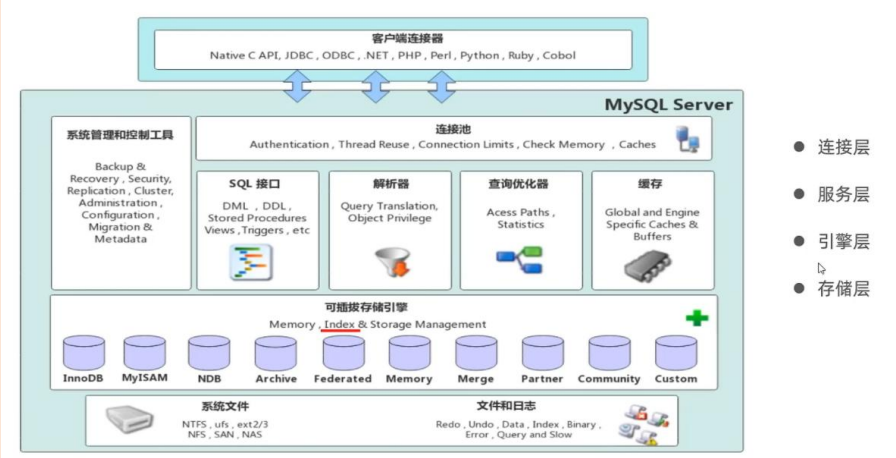
from -> **where** -> **group by** -> **having** -> **select** -> **order by** -> **limit**

二、数据库原理篇

1、MySQL 的架构

你能简述下 MySQL 的架构/体系结构吗？

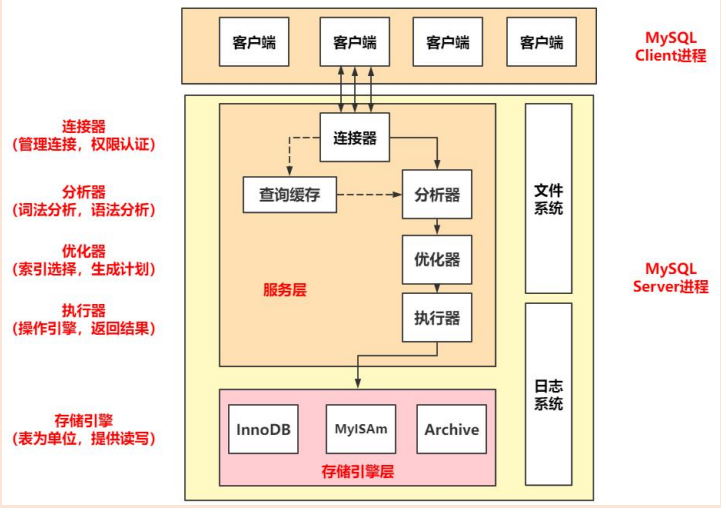
MySQL 采用了典型的 C/S 架构，即包括 MySQL Client 和 MySQL Server 两部分，这样子的好处是可以通过开发不同的客户端组件来让不同的开发环境支持 MySQL 服务。



MySQL Server 可以大致分为如下二层或者四层（不同文献分法不同，建议二层分法）：

分为两层	分为四层	主要任务	相关问题
服务层	连接层	连接管理、授权认证	MySQL 连接
	服务层	SQL 分析、调用引擎	SQL 的优化、数据库执行效率
引擎层	引擎层	表为单位，提供读写	选择合适的存储引擎、索引问题、表结构
	存储层	文件系统，存储数据	—

你能详细介绍一下 SQL 语句在 MySQL 的服务层中是怎么执行的吗？



首先，在输入 **SQL 语句**之前，程序员通过**客户端程序**连接到 **MySQL 服务器程序**。MySQL 中的**连接器**成功完成**连接管理**和**权限认证**后就可以处理**查询请求**了。

连接成功后，MySQL 根据查询请求，先到**查询缓存**中查看是否存在该请求的查询结果，

如果**有则直接返回**，如果**没有则继续继续输入分析器**。（查询缓存有弊端，只要有一个表被更新，所有查询缓存都清空，所以查询缓存的命中率非常低，查询缓存适合更新频率非常低的静态表）。

MySQL 的**分析器**首先对 SQL 查询语句进行“**词法分析**”，识别关键字“select”，然后**验证是否满足 SQL 语法**，不满足则会返回“ “You have an error in your SQL syntax”的错误，满足则继续输入 MySQL 的优化器。

MySQL 的**优化器**主要是生成执行计划，当 SQL 查询语句中有**多个索引**时，选择使用哪个索引，当是**多表查询**时，决定各个表的连接顺序。不同的方案可能执行效率不一样，优化器会决定选择哪个方案，然后输入 MySQL 的执行器。

MySQL 的**执行器**首先判断你对执行的**表是否有执行的权限**，如果没有则返回形如“SELECT command denied to user 'b'@'localhost' for table 'T'”的错误，如果有，**执行器**就会调用**存储引擎**提供的 API 来操作表。

2、MySQL 的存储引擎

数据库中的存储引擎有什么作用？

存储引擎主要任务是**基于二维表来存储数据**、**通过索引来更新或查询数据**。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**。

MySQL 的存储引擎是**弹性的可扩展式**，即可以根据不同的应用场景来选择不同的存储引擎。MySQL 在 5.5 之后的版本的默认存储引擎是 **InnoDB**，之前是 **MyISAM**。

InnoDB、MyISAM、Memory 三种存储引擎的区别是什么？

	应用场景	外键	事务	锁	B+tree 索引	Hash 索引	FullText 索引	存储文件	存储空间	内存使用	访问速度
InnoDB	MySQL 的**默认存储引擎**，绝大部分场景都能用，适合**并发性**和**安全性**高的场景（**应用最广泛**）	**支持**	**支持**	**行锁**	支持	不支持	支持（5.6 版本之后）	.ibd 存储表结构、数据和索引	高	高	慢
MyISAM	MySQL 过去默认的存储引擎，适合**并发性**和**安全性**低但是**读写操作频繁**的场景	不支持	不支持	**表锁**	支持	不支持	支持	.sdi 存储表结构; .MYD 存储数据; .MYI 存储索引	低	低	快
Memory	用作**临时表**和**缓存**	不支持	不支持	**表锁**	支持	**支持**	**不支持**	.sdi 存储表结构; **内存存储数据**	—	中等	快

进一步总结：

类型	定义	特点	缺点	应用场景
InnoDB	MySQL 的默	支持**事务**和**外键**，支持	存储空间和内存使用较	并发性和安

	认存储引擎	行级锁；支持 **B+tree 索引**和 **FullText 索引**；**不支持 Hash 索引**；	大； 访问速度较慢	全性要求高
MyISAM	MySQL 过去默认的存储引擎	不支持事务和外键；支持表级锁；支持 B+tree 索引和 FullText 索引；不支持 Hash 索引	安全性低，无法保证数据的完整性；查询速度快	对读写操作比较频繁，对并发性和安全性要求低
Memory	数据存储在内存中，表结构信息存储在磁盘中的存储引擎	不支持事务和外键；支持表级锁；支持 B+tree 索引和 Hash 索引；不支持 FullText 索引；	对表的大小有限制； 不支持事务，不保证数据的安全性；	临时表和缓存

3、MySQL 的索引

什么是索引？使用索引的优缺点是什么？

索引是**存储引擎**中用以**高效获取数据**的**有序**的**数据结构**。将二维表中的某个列设为索引列，那么根据该列查找数据时的效率教会比遍历扫描快很多。假如在**中大型表**中查询数据时表中没有索引，则需要全表扫描，查询效率非常低，如果具有索引则可以不需要全表扫描。

使用索引的优缺点如下：

索引的优点	**索引的缺点**
提高数据**检索效率**，降低数据库的 IO 成本（主要原因）； **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 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；	降低了**数据**进行**增加**、**删除**和**修改**的效率； 增加了数据存储的**空间成本**。

数据库中如何建立索引？哪些字段适合使用索引？

MySQL 中的主键索引和唯一索引在添加约束时一般都默认自动创建，此外也可以用 SQL 语句为一些列手动创建索引。

分类	含义	特点	关键字
主键索引	添加主键约束的列默认**自动创建**主键索引，也可**手动创建**。	只能有一个	PRIMARY
唯一索引	添加唯一约束的列默认**自动创建**唯一索引，也可**手动创建**。	可以有多个	UNIQUE
常规索引	可以快速定位特定数据，一般是**手动创建**。	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，一般是**手动创建**。	可以有多个	FULLTEXT
组合索引	在多个字段上创建的字段，	可以有多个	—

在手动创建索引时，一般选择如下列：

（1）需要**经常搜索、排序**的列

（2）添加了**外键约束**的列，可以加快**表连接**的速度

（3）经常作为**条件**来搜索的列

哪些列不适合建立索引呢，一般如下：

（1）**很少被查询过的列**。索引并不是越多越好，会降低插入、更新效率，提高存储成本

（2）经常**被修改的列**

索引也存在失效的情况，请列举？/如何优化索引的使用？/如何避免索引失效？

索引列使用了 **SQL 函数**或者**参加了运算**，索引失效。

索引列类型为**字符串**，**使用时不加引号**，索引失效。

索引列使用**模糊匹配**且以**“%”**开头，索引失效

where 语句中 or 有一边的条件字段没有索引时，索引失效

使用**复合索引**时，没有使用左侧的列查找，索引会失效

MySQL 内部认为使用**索引**比**全表查询**更慢，则不适用索引。

MySQL 中索引的数据结构类型有哪些?

索引结构	**描述**
B+Tree	**最常见**的索引类型，大部分存储引擎都**默认支持** 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
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

4、MySQL 的事务

什么是事务？SQL 语句如何使用事务？你知道什么长事务吗？

事务是**一组 SQL 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**。**事务**用来保证数据的**完整性**和**一致性**。操作数据库时，我们经常会使用到事务。事务也是在 **MySQL** 的**引擎层**支持的。

在 MySQL 中输入 SQL 语句时，很多默认都是自动提交事务（autocommit）的，即你只输入了一条 SQL 语句，系统也会默认把它封装在一个事务中提交，这是事务的隐式提交。

如果想要进行事务的**显示提交**，可以进行如下操作：

```
-- 开启事务
begin; --也可以使用 start transaction
-- 执行一组操作
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';
-- 提交事务
commit;
-- 回滚事务
rollback;
```

事务的自动提交设置也可能会被 set autocommit=0;来关闭，这样用户输入一条 SQL 语句时，事务被创建，该语句被执行，但是事务没有被自动提交，会被继续等待，直到用户主动执行 commit 或者 rollback 或者断开数据库连接，这样用户的所有 SQL 操作本质上都是在一个事务中，这样的事务也叫做**长事务**。**长事务应该尽可能被避免**。

你能介绍下事务四大特性（ACID）：原子性、一致性、隔离性、持久性?

原子性（Atomicity）:原子性是指事务包含的**所有操作要么全部成功，要么全部失败回滚**，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

一致性（Consistency）:**事务开始前和结束后，数据库的完整性约束没有被破坏**。比如 A 向 B 转账，不可能 A 扣了钱，B 却没收到。

隔离性（Isolation）:隔离性是并行场景下，当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，**多个并发事务之间要相互隔离**。同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰。比如 A 正在从一张银行卡中取钱，在 A 取钱的过程结束前，B 不能向这张卡转账。

持久性（Durability）:**持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的**，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

请注意，事务的原子性、一致性和持久性都是由事务本身决定的，即只要使用事务就具备这些特性，但是**事务**的**隔离性是需要手动设置隔离级别的**，即在并发场景中，根据需求来设置事务的隔离级别。

你知道并发事务中的脏读、不可重复读、幻读吗？

脏读、**不可重复读**和**幻读**都是一种并发事务问题，发生在多个事务同时执行时：

问题	描述
脏读	一个事务读到另一个事务还没提交的数据（一个事务的操作数据被其他事务的操作污染）
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（可能是脏读，也可能是当前事务的数据被其他事务提交后影响）
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

我们来举例说明脏读和不可重复读，假设存在一个长事务 A 和短事务 B，它们同时查询或者修改同一个值 1。在下述过程中，V1==2，说明该值被未提交的事务 B 污染了，这就是一种**脏读**现象；V1==1，V2==2，虽然 V1 没有被污染，但是 V2 被提交后的事务 B 污染了，这就是一种**不可重复读**现象。

事务A	事务B
启动事务 查询得到值1	启动事务
	查询得到值1
	将1改成2
查询得到值V1	
	提交事务B
查询得到值V2	
提交事务A	
查询得到值V3	

如何避免并发事务中可能出现的脏读、不可重复读、幻读呢？

想要解决这个问题，可以设置**事务隔离级别**：

需求	SQL 语句
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 };

注意 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

事务隔离级别设置得越高，效率越低，所以很多时候需要寻找一个平衡点：

类型	描述	脏读	不可重复读	幻读	并发实现
读未提交（Read uncommitted）	一个事务还没提交时，它的变更就可被其他事务看到	无法避免	无法避免	无法避免	
读提交（Read committed）	一个事务提交之后，它做的变更才就被其他事务看到	可避免	无法避免	无法避免	MVCC+锁
可重复读（Repeatable Read，默认）	一个事务执行过程中看到的数据，总是跟这个事务在启动时看到的数据是一致的。当然在可重复读隔离级别下，未提交变更对其他事务也是不可见的	可避免	可避免	无法避免	MVCC+锁
串行化（Serializable）	顾名思义是对于同一行记录，“写”会加“写锁”，“读”会加“读锁”。当出现读写锁冲突 的时候，后访问的事务必须等前一个事务执行完成，才能继续执行。	可避免	可 避免	可避免	锁

串行化性能最低；**读未提交**性能最高，**数据安全性**最差;MySQL 默认的事务隔离级别是**可重复读**，可以避免**脏读**和**不可重复**读问题，其他数据库的事务隔离级别更多是**读提交**。

5、MySQL 的锁

你知道数据库中的两阶段锁协议吗？

行级锁也叫做**记录锁**，针对表中的每行数据。在 InnoDB 事务中的 SQL 语句，如果 SQL 语句是增删改的更新语句，则会自动增加一个行级锁，但是执行完 SQL 语句后并不会立刻释放，而是直至该事物提交后才会释放，这个就是**两阶段锁协议**（InnoDB 事务中的锁必须先加锁后不能立即释放，而是要等待到提交事务的时候统一释放）。

事务 A 在执行 2 条 update 语句时会对 Id1 和 ID2 的两条记录都添加了行级锁，此时事务 B 想要更新 D1 的语句会被阻塞，直至事务 A 提交释放它加的两个行级锁。

根据这个**两阶段锁协议**的特点，我们可以在事务中将最可能造成锁冲突、最可能影响并发度的 SQL 更新语句尽可能往后放。例如,下面事务中的 3 条语句，由于影院的账户余额这条记录最可能和其他事务同时进行，所以 2 可以放到最后执行。

1. 从顾客A账户余额中扣除电影票价； 更新语句
2. 给影院B的账户余额增加这张电影票价； 更新语句
3. 记录一条交易日志。 插入语句

你知道数据库中的死锁和死锁检测吗？

当不同线程都互相依赖对方的资源时，就会导致这些线程都进入无线等待的状态造成死锁。数据库中的**死锁**现象很容易发生在 InnoDB 引擎的**行级锁**中，例如：

事务 A 的第 2 条更新语句依赖于事务 B 释放它的 ID2 的行级锁，事务 B 的第一条更新语句依赖于事务 A 的第 2 条更新语句释放它的 ID1 的行级锁。但是事务 A 和事务 B 都必须提交才能释放各自的行级锁，所以事务 A 和事务 B 都互相依赖，陷入无线等待，造成死锁现象。

死锁会大量消耗 CPU 资源，致使程序空转。所以需要**死锁检测**。在 MySQL 中设置参数数 innodb_deadlock_detect 为 on 则会开启死锁检测程序，该程序主动发现死锁后会主动回滚到死锁链条中的某一个事务，让其他事务得以继续执行。

但是**死锁检测程序会让每一个 InnoDB 事务添加锁时都查看是否被别人锁住，也会增加 CPU 消耗**。所以对于热点行的更新需要控制并发度，比如同一行最多只能有 10 个线程在更新，这种并发控制一般需要在数据库程序中设计。

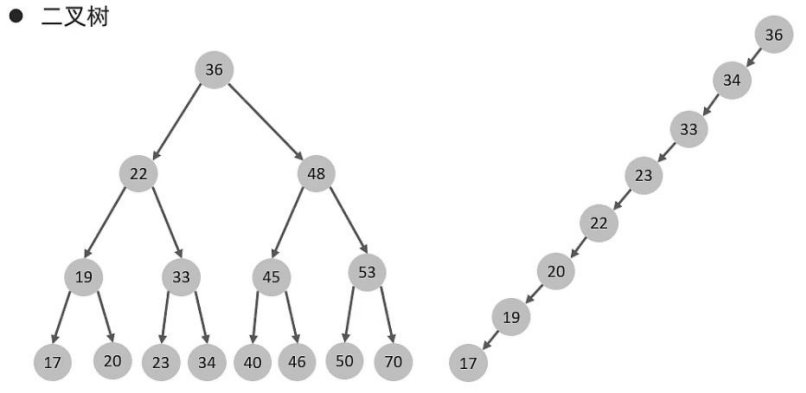
二、数据库原理篇

1、MySQL 的索引

搜索二叉树、红黑树、B-Tree、B+Tree、MySQL 中的 B+Tree、Hash 作为索引的区别是什么？

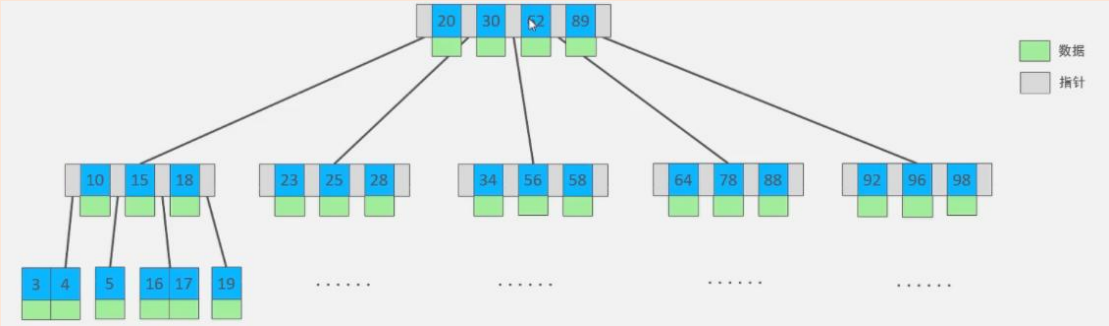
首先达成一个共识：大数据量情况下，树的层级（深度）越低，检索速度越快。

搜索二叉树的每个节点只存放一个数据，而且在顺序插入数据时，容易产生更多的层级。在数据量较大情况下，层级越深，检索速度越慢。



红黑树是一种高度平衡二叉树，可以让二叉树的节点尽可能平衡，但是其每个节点也只存在一个数据，所以在数据量较大情况下，层级越深，检索速度依旧慢。

B-Tree 是一种多路平衡查找树，其每个节点存放多个数据，可以极大减少树的层级。以一棵最大度数（max-degree，指一个节点的子节点个数）为 5 阶的 B-Tree 为例（每个节点最多存储 4 个 key，5 个指针）如下：



但是 B-Tree 的每个节点都单独存放一份数据，损耗了存储空间。

B+Tree

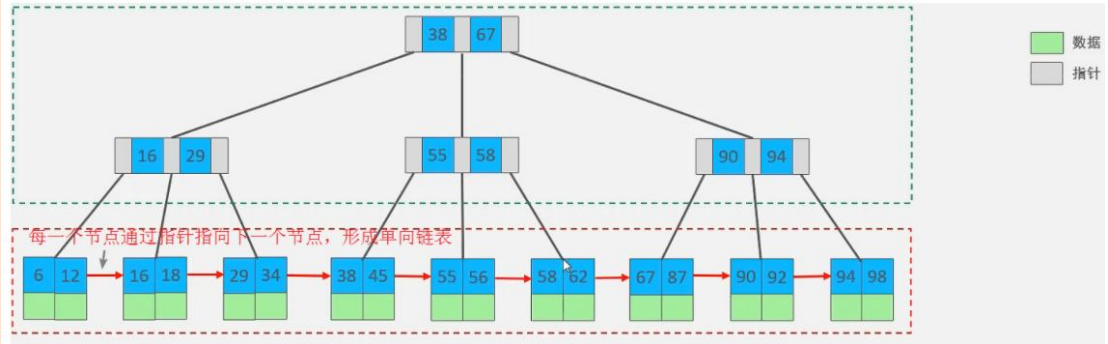
B+Tree 在 B-Tree 的基础上，做了如下改进：

（1）所有的数据都会出现在叶子节点，内部节点不存储数据，这样有 2 个好处：

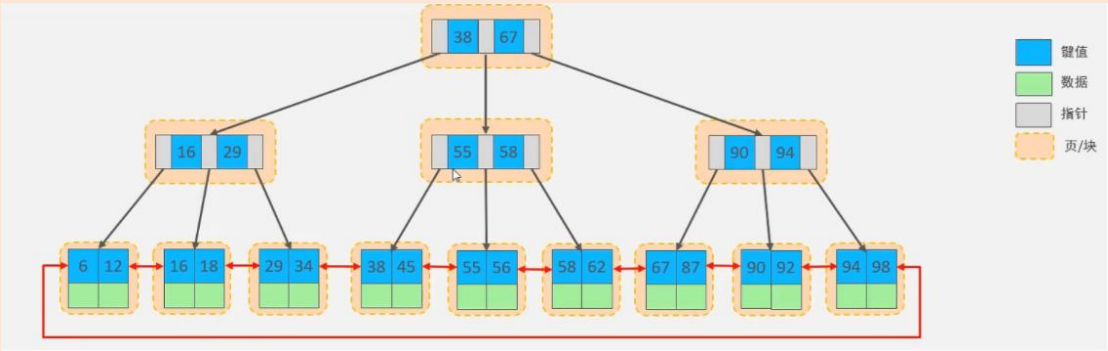
可以使磁盘的每一页存储的内部节点（key）更多，从而降低内存和磁盘的 IO 次数，提高效率。

由于叶子节点不存储数据，所以任何 key 的查找必须走一条从根结点到叶子结点的路径，所有 key 查询的路径长度相同，即每一个数据的查询效率相当，性能更加稳定。

（2）叶子节点形成一个单向链表。

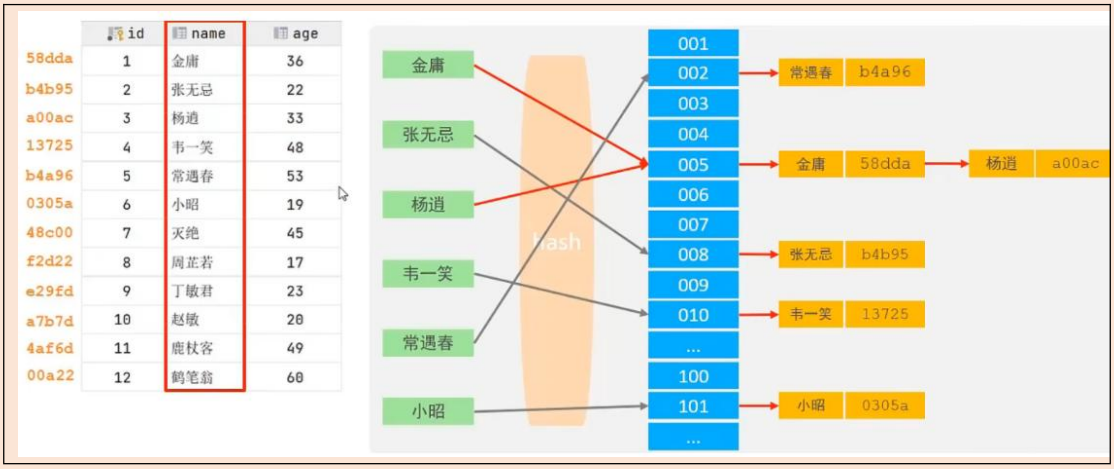


MySQL 索引数据结构对经典的 B+Tree 进行了优化。在原 B+Tree 的基础上，增加一个指向相邻叶子节点的链表指针，就形成了带有顺序指针的 B+Tree，提高区间访问的性能。



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。



优点：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

缺点：哈希索引由于哈希函数的特性，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

总结如下：

	结构特点	不足	备注
搜索二叉树	非平衡二叉树；每个节点存储	非平衡特性导致了顺序插入时容易产生更深的树。（极端情况	

	一个数据；	下，会形成链表）	
红黑树	高度平衡二叉树； 每个节点存储一个数据；	每个节点存储一个数据，在大数据下依然产生较深的树	很多编程语言内置数据类型的底层数据结构（map、set）
B-Tree	高度平衡二叉树； 每个节点存储多个数据；	每个节点都存储数据，导致需要的节点数更多，容易产生更深的数	B-Tree 在磁盘的存储空间中使用较多
B+Tree	高度平衡二叉树； 每个决策节点只存储键值； 叶节点存储键值和数据，并且形成**单链表**	查询效率不如 Hash	大多数存储引擎中支持的索引结构
MySQL 中的 B+Tree	高度平衡二叉树； 每个决策节点只存储键值； 叶节点存储键值和数据，并且形成**双链表**	查询效率不如 Hash； 但是支持等值查询、范围查询和排序操作	InnoDB 存储引擎默认支持的索引结构
Hash	哈希函数+链表设计；	查询效率优于 B+Tree； 但是只适合等值查询，不支持范围查询和排序操作	

你知道为什么 B+Tree 索引查询效率高、修改效率低吗？

这和 B+Tree 的结构特点紧密相关，它的内部节点不存储数据，叶节点存储数据，而且叶节点是一个有序的单链表或双链表。

假如查询的数据的条件字段是连续的（例如主键 ID 的聚集索引），那么只要找到第一个 ID，后续的递增的 ID 就可以快速找到。

假如查询的数据的条件字段不是连续的（例如非主键 ID 的非聚集索引），那么每个记录都根据根到叶的路径查询即可。

B+Tree 的修改效率低是因为 B+Tree 在插入新数据时，为了保证有序就要对数据页重新排序。

主键为什么要选择自增列？

在多数情况下，建议使用自增列的字段作为主键，这是因为使用自增列，**主键是顺序增加的，每次插入数据效率较高**；如果没有使用自增列（例如身份证号或学号），那么**每次插入的主键值过于随机，其插入数据的效率较低**。

MySQL 默认会根据每张表中的**主键**建立一个 B+Tree 索引，其中主键作为 key 存放在内部节点，叶节点存放 key 和 value（本行数据），各个叶节点本身会形成有序的链表结构。

如果每次插入数据时，主键（key）是递增的，MySQL 的存储引擎可以快速找到适当的位置并且插入，不需要额外移动数据。

如果每次插入数据时，主键（key）是非递增的，MySQL 的存储引擎就需要重新通过内

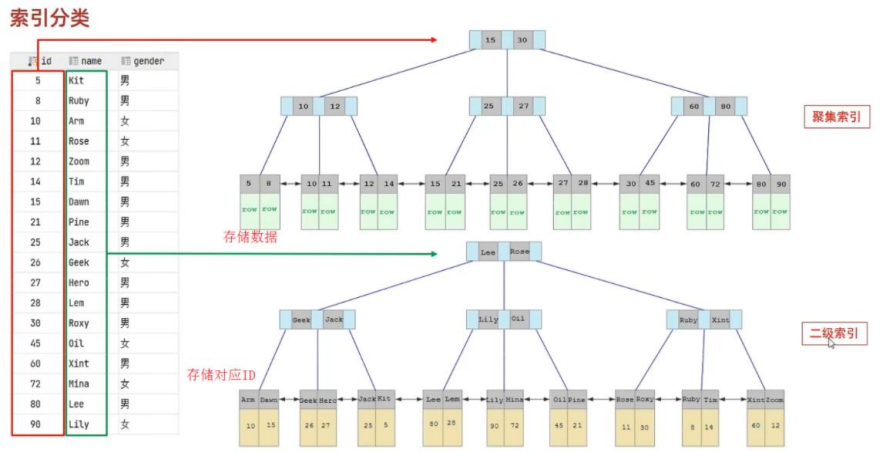
部节点来搜索插入位置，可能需要额外移动数据和频分的分页操作，操作额外的性能开销。

主键一般选择自增字段的 ID，但是部分需求下也会选择业务字段的非自增列，那么该业务字段必须是唯一约束的。

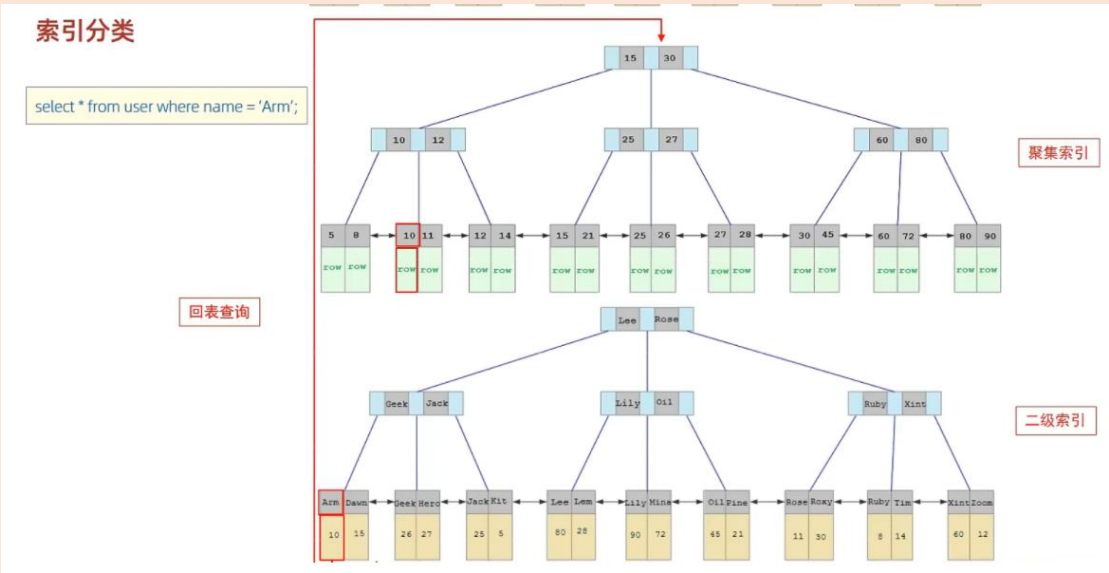
你知道聚集索引、非聚集索引、一级索引、二级索引、回表查询吗？简述一下它们的区别吗？

在 InnoDB 存储引擎中，根据 B+Tree 索引的存储形式，又可以分为以下两种聚集索引和二级索引（也叫非聚集索引）。

以下表为例，其中 id 字段是**主键约束**，所以该列自动生成一个**主键索引**，也就是聚集索引，那么 InnoDB 会生成一个如下的 B+Tree 作为**聚集索引**的结构信息，假设要根据 name 来查询，则又会根据 name 列生成一个 B+Tree 作为二级索引。



回表查询就是先走**二级索引**找到数据的 **ID 值**，然后根据 **ID 值**再找**聚集索引**找到对应的**数据**。



分类	含义	特点
聚集索引（一级索引）	将数据与索引一起存储，索引结构的叶节点保存了**行数据，**	**必须有且只能有一个**
非聚集索引（二级索引）	将数据与索引分开存储，索引结构的叶节点保存**行数据**中的**主键**	**可以有多个**

聚集索引选取策略：

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**

聚集索引和**非聚集索引**的根本区别在于**是否将数据（行记录）和索引一起存储**，聚集索引的叶节点保存行数据，二级索引保存主键，所以二级索引找到主键后还需要根据主键查找行数据，效率降低。

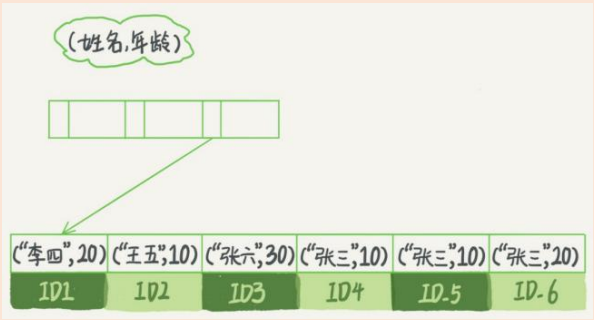
请注意由于**聚集索引**的叶节点存储的数据就是行数据，找到叶节点就找到了行数据，而且叶节点是有序的，这意味者索引的排列顺序和数据的排列顺序一致，所以只要找到第一个索引值记录，之后的索引记录就能快速找到。

什么是组合索引/联合索引/复合索引，它又是什么作用？

复合索引（也叫**联合/组合索引**）是指同时将**多个字段设为一个索引**，MySQL 允许创建最多包含 16 列的复合索引。**复合索引**在查询时遵循**最左前缀原则**。

```
create index 索引名称 on 表名称(字段 1,字段 2,字段 3);-- 为表创建一个复合索引
```

最左前缀原则指的是**匹配从复合索引的最左列开始，并且不能跳过复合索引中的列**。在复合索引中，记录首先按照复合索引的第一个字段排序，对于在第一个字段上取值相同的记录，系统再按照第二个字段的取值排序，以此类推。例如：存在查询某个名字的年龄的高频请求，将其建立为联合索引，假如查找“张三”，存储引擎可以快速定位到 ID4，然后向后遍历就可以得到所有需要的结果，遍历判断的条件就是依据最左前缀原则。



因此**只有复合索引的第一个字段出现在查询条件中，该复合索引才可能被使用**，否则就会失效。因此将应用频度高的字段，放置在复合索引的前面，会使系统最大可能地使用此索引，发挥复合索引的作用。

你知道索引下推吗？

索引下推（ICP）是 MySQL 5.6 之后的**针对扫描二级索引的一项优化改进**，总的来说就是把**索引过滤条件从服务层下推到引擎层**，可以减少存储引擎回表次数和服务层访问引擎层的次数。没有使用 ICP 时，MySQL 在引擎层执行一条 SQL 查询语句：先访问二级索引找到主键值，根据主键值找到完整行记录，将行记录交给 Server 层去检测是否满足 where 条件。这部分就是回表查询+server 层检测条件语句过滤。

使用 ICP 时，MySQL 在引擎层执行一条 SQL 查询语句：先访问二级索引找到主键值，然后判断 where 条件部分是否用索引中的列做检查，如果是再进行回表：根据主键值找到完整行记录，将行记录交给 Server 层去检测是否满足 where 条件。这部分就是回表查询+server 层检测条件语句过滤。

可以看到，**索引下推本质上就是在存储引擎中就进行一些索引字段的条件判断，来提高查找效率**。所以索引下推发生在 SQL 语句中存在查询字段和条件字段一样且都是索引的情况下。

你知道覆盖索引吗？

覆盖索引是指**索引上的信息**足够满足**查询请求**，不需要再回到主键索引上去取数据（索引字段包含查询字段）。例如 select id,age from user where age = 10;

为了提高查询速度，将 age 设为普通索引，由于 age 是查询字段，也是索引字段，所以就实现了覆盖索引，存储引擎只需要扫描一次 age 的 B+树就能满足要求。

如果继续查找 select id,age,name from user where age = 10;

因为 name 不是索引字段，所以通过 age 的 B+树后还需要根据 ID 值回表查找 name 的值，此时就不是覆盖索引。

优化方法可以将 age 索引删除，重新建立一个 age 和 name 的联合索引，那么就实现了覆盖索引。

由于覆盖索引可以减少树的搜索次数，显著提升查询性能，所以使用覆盖索引是一个常用的性能优化手段。

2、MySQL 的事务

你知道 MySQL 中事务的实现原理吗？

事务是 MySQL 用于并发场景下的机制，它的四大特性分别由如下机制实现（后文详细介绍）：

原子性：undo log（日志）

持久性：redo log（日志）

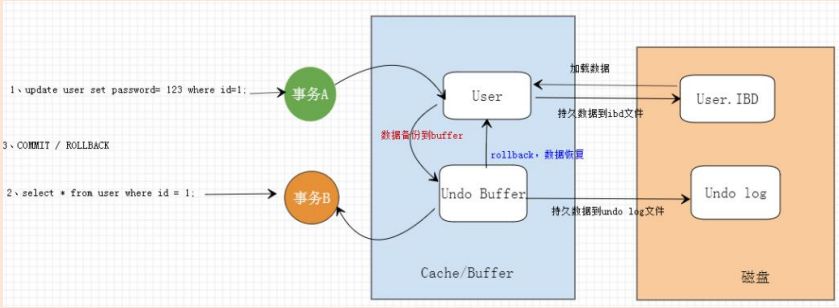
一致性：redo log+undo log（日志）

隔离性：MVCC+锁

你知道事务的原子性是如何实现的吗？

事务的**原子性**是指事务包含的**所有操作要么全部成功，要么全部失败回滚**，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

事务的原子性是由 InnoDB 引擎中的 undo log（回滚日志）实现的。undo log 是一种用于撤销回滚事务的日志，其本质上是一种命名为回滚段（roll back segment）的数据结构。



回滚日志主要有 2 个作用：实现事务的原子性、构建版本链用于 MVCC。

回滚日志在事务启动时产生，但不会在事务提交后就删除，因为事务还可能被回滚。

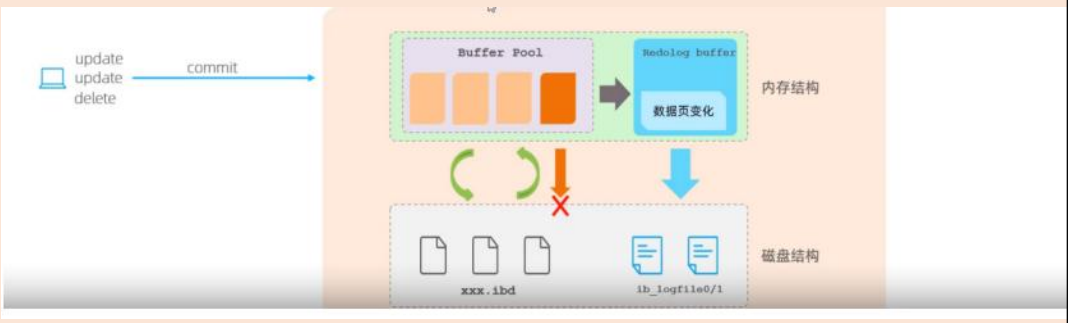
回滚日志是一种逻辑日志，所谓逻辑日志，就是 undo log 是记录一个操作过程，不会物理删除 undo log。从逻辑上可以认为当 delete 一条记录时，回滚日志中记录一条对应的 insert 记录，反之亦然，当 update 一条记录时，它记录一条对应相反的 update 记录。当执行 rollback 时，就可以从回滚日志的逻辑记录中读取到相应的内容并进行回滚。

你知道事务的持久性是如何实现的吗？

事务的**持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的**，即便是在数据库事务的系统遇到故障的情况下也不会丢失提交事务的操作。

事务的持久性是 InnoDB 引擎中的 redo log（重做日志）实现的。redo log 在内存中存在一个重做日志缓存（redo log buffer），在磁盘中存在一个重做日志文件（redo log file）。

假如一个事务中存在 SQL 更改语句，SQL 更改语句修改的数据先在更改缓存区，然后写入 BufferPool 中的脏页中，脏页会定时刷新到磁盘 ibd 文件中，但是如果从脏页直接刷新到 ibd 文件中效率会非常慢，而且这时如果程序报错停止运行，就会丢失数据。所以脏页的数据先写入 Redolog buffer 中，由 Redolog buffer 写入 redo log file，这种技术也叫做 WAL（Write-Ahead-Logging）技术，即先写日志，再写磁盘。。



为什么重做日志缓存写入磁盘就不会丢失数据而且效率高呢？

从 BufferPool 的脏页直接写入磁盘的 ibd 文件是一种随机 IO，每次插入都要搜索遍历插入位置，然后才能插入，效率自然低；

BufferPool 将所有的脏页数据写入 Redolog buffer，然后每次提交事务时才 Redolog buffer 写入 redo log file，这种日志 IO 是一种顺序 IO，即直接在日志文件后面追加即可，写入效率快，而且 redo log file 是固定大小的可循环写的文件。

总结：事务的持久性是由 redo log 日志实现的。redo log 日志是 MySQL 的 InnoDB 存储引擎的日志模块。具体讲，当执行 SQL 更新语句时，MySQL 的 InnoDB 存储引擎会将记录写入 redo log 中，然后更新内存数据。当 MySQL 程序空闲时，InnoDB 存储引擎再将 redo log 的数据更新到磁盘文件中。

redo log 日志的大小是固定的，如果一边写入数据一边将数据保存到磁盘中，如果现有 redo log 文件写满就会覆盖之前的记录，覆盖前会保存日志数据。具有缓存安全（crash-safe），即数据库发生异常重启，之前的记录都不会丢失。

你知道事务的一致性是如何实现的吗？

事务的一致性（Consistency）是指**事务开始前和结束后，数据库的完整性约束没有被破坏**。比如 A 向 B 转账，不可能 A 扣了钱，B 却没收到。

事务的一致性由事务的原子性和持久性决定，即保证事务的原子性和持久性就能保证事务的一致性。

你知道事务的隔离性是如何实现的吗？

事务的隔离性是指并行场景下，当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，**多个并发事务之间要相互隔离**。同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰。比如 A 正在从一张银行卡中取钱，在 A 取钱的过程结束前，B 不能向这张卡转账。

在 InnoDB 引擎中，事务的隔离性是通过建立**快照读**（SQL 查询语句，获得一个**可读视图**（consistent read view））和**当前读**（SQL 更新语句，同时加上**行级锁**）来实现的。

假如事务的隔离等级是**读未提交**，那么事务不需要什么快照读或者当前读，直接返回内存中记录更新的最新值，在并发场景中安全性最差；

假如事务的隔离等级是**读提交**，那么存储引擎在**执行每条 SQL 查询语句时创建一个一致性读视图**，执行完就**删除一致性读视图**，使用的机制是**多版本并发控制（MVCC）**，如果执行 SQL 更新语句，则需要当前读，并且加行级锁，读取该语句执行前已提交的事务的版本数据；

假如事务的隔离等级是**可重复读**，那么数据库在**执行事务开始时创建一个一致性读视图**，提交事务时就**删除一致性读视图**，使用的机制是**多版本并发控制（MVCC）**，如果执行 **SQL 更新语句**，则需要**当前读**，并且加**行级锁**，读取该语句执行前已提交的**事务**的版本数据；

假如事务的隔离等级是**串行化**，那么数据库在执行事务的每条 SQL 语句时都需要**加锁**，使用的机制是**锁机制。**

你知道数据库中的多版本并发控制（MVCC）吗？

多版本并发控制（MVCC）是 InnoDB 引擎（也包括其他很多存储引擎）在隔离级别为可重复读和提交读下实现事务的隔离性的机制。

多版本并发控制有什么特点？

解决了**读写并发问题**（即读不阻塞写，写不阻塞读）。普通的写锁只能串行执行（读写都不能并发），读锁只能实现读读并发。

解决了**一致性读问题**。如上所述，MVCC 实现了快照读。

降低了死锁的概率。因为 InnoDB 的 MVCC 采用了乐观锁的方式，读取数据时不需要加锁，对于写操作，也只锁住了必要的行。

MVCC 涉及 2 个概念和 2 个组件。2 个概念是快照读和当前读。2 个组件是版本链、可读视图（readview）。

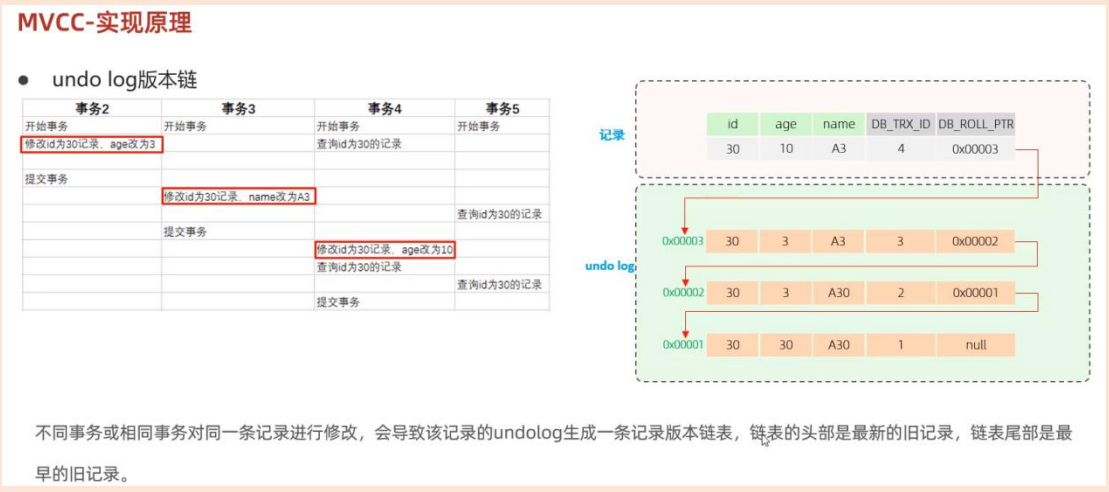
什么是快照读和当前读？

快照读（也叫**一致读，一致性不加锁**的读），指的是当前事务读取到的数据要么是事务开始前就已经存在的数据，要么是当前事务自身插入或者修改过的数据。与**快照读**相对应的就是**当前读**，**当前读**就是**读取最新数据**（这里不详细介绍当前读）

什么是版本链？

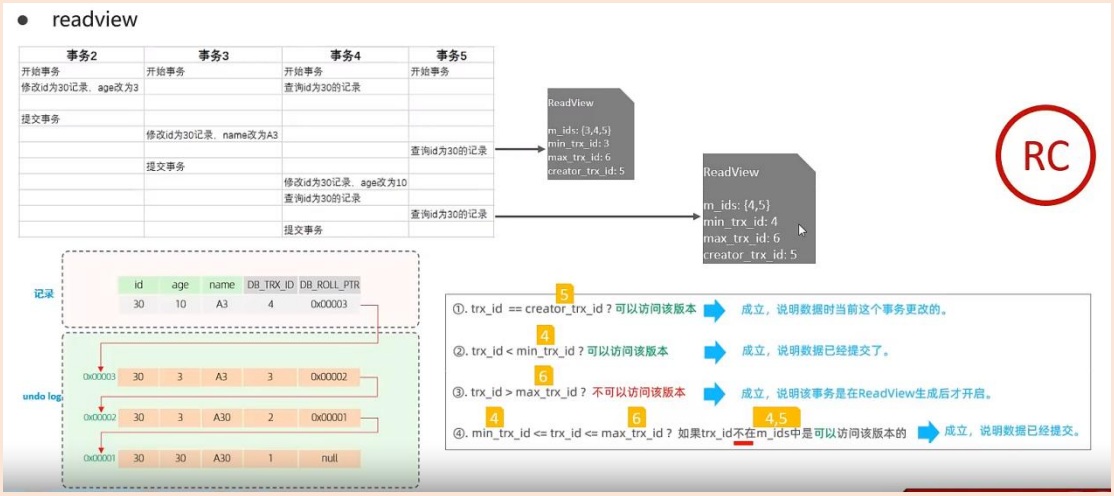
事务在实现原子性时使用了回滚日志（undo log），回滚日志是在执行 SQL 更新语句时产生的日志，即 **MySQL 数据库中的每条行记录都可能有多个历史版本（回滚日志）**。回滚日志是中存在一些特殊的字段，例如自己的版本 ID（=事务 ID）和指向下一条回滚日志的指针。

每次开启一个事务时，都会获得一个自增长的事务 ID，通过事务 ID 可以判断事务启动的时间顺序。InnoDB 中每个行记录的所有历史快照是一个**链表结构**。



什么是可读视图？

在读提交的隔离级别下，事务中执行每一条 SQL 查询语句时就会建立一个可读视图；在可重复读的隔离级别下，事务开始时建立一个可读视图。可读视图本质是一个数据结构，如下图：



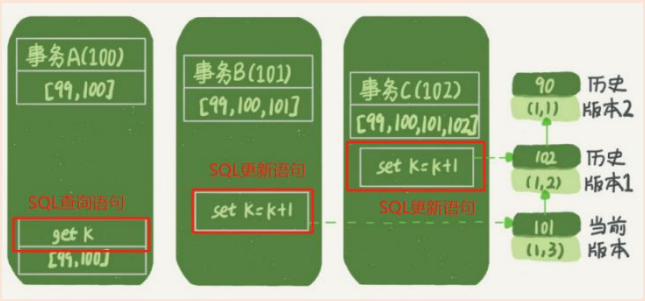
InnoDB 根据可读视图来读取数据，就能保证不同事务之间的数据的隔离性。

你知道快照读、一致性读、当前读吗？

快照读就是**一致性读**，指的是在 InnoDB 引擎的隔离级别为**读提交**时，引擎为在执行 **SQL 查询语句时建立可读视图**，只能读取到此时之前所有已经提交事务做出的更改；在 InnoDB 引擎的隔离级别为**可重复读**时，引擎为**每个事务开始时建立一个可读视图**，只能读取到该事务开始前所有已经提交事务做出的更改。

当前读是指在 InnoDB 引擎的隔离级别为**可重复读**时，事务中执行 **SQL 更新语句**时，引擎仍然可以为该语句读到当前点所有已经提交的事务，同时加**行级锁**。当前读一般在 **SQL 更新语句**中使用，如果想在 SQL 查询语句中使用，需要显示地加锁。

举个例子,在 InnoDB 引擎下，隔离级别为可重复读，存在如下先后启动的三个事务，它们分别查询或者读取某个行记录 k（事务 A 启动前为 1）：



其中事务 A 最后 k=1，因为其只有一个 SQL 查询语句，是一致性读，从事务启动时就获取了 k 的值，只要事务 A 中不更改它，它就始终是 1；

事务 C 最后 k=2，因为其只有一个 SQL 更新语句，是当前读，获取 k=1，然后加 1=2；

事务 B 最后 k=3，因为其只有一个 SQL 更新语句，是当前读，其始终在执行该语句时才读取 k，读取 k 之前已经提交的事务更新的数据，显然事务 C 已经提交了，此时 k=2+1=3.

假设事务 C 晚一点提交，如下：

事务A	事务B	事务C'
start transaction with consistent snapshot;		
	start transaction with consistent snapshot;	
		start transaction with consistent snapshot; update t set k=k+1 where id=1;
	update t set k=k+1 where id=1; select k from t where id=1;	
select k from t where id=1; commit;		commit;
	commit;	

事务 A 只有查询语句不影响，事务 C 最后 k=2，事务 C 的更新语句发生在事务 B 的更新语句之前，显然它的当前读对 k 加了行级锁，此时事务 B 的更新语句再读取 k 会陷入阻塞，等到事务 C 提交后释放锁（两阶段协议），事务 B 才能执行更新语句，即事务 B 本质上是在事务 C 提交后才能执行它的更新语句。

3、MySQL 的锁

你知道数据库中的两阶段锁协议吗？

行级锁也叫做**记录锁**，针对表中的每行数据。在 InnoDB 事务中的 SQL 语句，如果 SQL 语句是增删改的更新语句，则会自动增加一个行级锁，但是执行完 SQL 语句后并不会立刻释放，而是直至该事物提交后才会释放，这个就是**两阶段锁协议**（InnoDB 事务中的锁必须先加锁后不能立即释放，而是要等得到提交事务的时候统一释放）。

事务 A 在执行 2 条 update 语句时会对 Id1 和 ID2 的两条记录都添加了行级锁，此时事务 B 想要更新 D1 的语句会被阻塞，直至事务 A 提交释放它加的两个行级锁。

根据这个**两阶段锁协议**的特点，我们可以在事务中将最可能造成锁冲突、最可能影响并发度的 SQL 更新语句尽可能往后放。例如,下面事务中的 3 条语句，由于影院的账户余额这条记录最可能和其他事务同时进行，所以 2 可以放到最后执行。

你知道数据库中的死锁和死锁检测吗？

当不同线程都互相依赖对方的资源时，就会导致这些线程都进入无线等待的状态造成死

锁。数据库中的**死锁**现象很容易发生在 InnoDB 引擎的**行级锁**中，例如：

事务 A 的第 2 条更新语句依赖于事务 B 释放它的 ID2 的行级锁，事务 B 的第一条更新语句依赖于事务 A 的第 2 条更新语句释放它的 ID1 的行级锁。但是事务 A 和事务 B 都必须提交才能释放各自的行级锁，所以事务 A 和事务 B 都互相依赖，陷入无线等待，造成死锁现象。

死锁会大量消耗 CPU 资源，致使程序空转。所以需要**死锁检测**。在 MySQL 中设置参数数 innodb_deadlock_detect 为 on 则会开启死锁检测程序，该程序主动发现死锁后会主动回滚到死锁链条中的某一个事务，让其他事务得以继续执行。

但是**死锁检测程序会让每一个 InnoDB 事务添加锁时都查看是否被别人锁住，也会增加 CPU 消耗**。所以对于热点行的更新需要控制并发度，比如同一行最多只能有 10 个线程在更新，这种并发控制一般需要在数据库程序中设计。

4、MySQL 的日志

你能详细介绍一下 MySQL 中的日志吗？

MySQL 提供了如下 6 种日志：

架构层	日志种类	记录信息	作用
服务层	错误日志	MySQL 默认开启，记录数据库程序从启动到停止的所有报错信息	用于查找错误
	二进制日志（bin log）	MySQL 默认开启，记录数据库程序的所有 DDL（数据定义语言）语句和 DML（数据操作语言）语句	用于数据恢复和主从复制，不具备缓存安全，无固定大小。
	查询日志	MySQL 默认关闭，记录数据库程序的所有 SQL 查询语句的执行记录。	用来分析哪些 SQL 查询语句执行较慢，从而提供性能优化策略
	慢查询日志	MySQL 默认关闭，记录数据库程序的超过规定时间的 SQL 查询语句的执行记录，只记录超过规定时间的 SQL 查询语句。	
引擎层	重做日志（redo log）。	必须开启，执行 SQL 更新语句时，InnoDB 引擎会将记录先写	用以实现事务的持久性，读写速度快、缓存安全（数据库宕

		入 redo log 中，然后更新到磁盘文件中（Write ahead Logging，WAL）。	机也会安全写入磁盘中）、大小固定，循环覆盖写入。
	回滚日志（undo log）	必须开启，执行 SQL 更新语句时，InnoDB 引擎会将原有记录作为历史版本，新更新的行记录作为最新版本	用于实现事务的原子性，可以回滚事务，构建版本链实现 MVCC。每条行记录都有多条回滚日志，逻辑日志，事务提交也不一定删除。

你知道 InnoDB 引擎中的两阶段提交吗？

InnoDB 引擎的两阶段提交是一种为了防止数据库异常重启造成 redo log 日志和 binlog 日志写入数据不一样而设计的算法。InnoDB 引擎将 SQL 更新语句执行后的内存中的数据分成两个阶段来提交到 redo log 日志和 binlog 日志中：

（一）prepare 阶段。InnoDB 引擎写 redo-log 日志并将其标记为 prepare 状态，然后执行器写 binlog 日志提交。

（二）commit 阶段。InnoDB 引擎继续写 redo-log 日志将其标记为 commit 状态。

redo-log 日志是具备缓存安全的，即异常重启后会保存数据；binlog 日志不具备缓存安全。内存中的数据需要全部写入 redo-log 日志和 binlog 日志。

如果先写 redo-log 日志，然后数据库异常重启，binlog 日志没有写入；如果先写 binlog 日志，然后数据库异常重启，redo-log 日志没有写入。两阶段提交则先将写入 redo-log 日志中并及时将状态修改为 prepare 阶段，这样之后发生异常重启，redo-log 日志发现状态为 prepare 阶段就会再次写入 binlog 日志，然后提交事务。