# Mysql

### 1、InnoDB和MyISAM的区别和使用场景

事务：InnoDB支持事务，而MyISAM不支持

外键：InnoDB支持外键，而MyISAM不支持

锁：InnoDB支持行级锁（并发量高）和表锁，MyISAM只支持表锁

索引：InnoDB不支持全文索引，MyISAM支持全文索引（全文索引是基于分词的索引，能更快速地在文本中检索一段信息内容）

使用场景：

InnoDB适合需要高并发或事务的场景，

MyISAM适合读操作远远大于写操作且不需要事务的场景

|  |  |  |
| --- | --- | --- |
| 属性 | InnoDB | MyISAM |
| **事务** | 支持事务 | 不支持事务 |
| **外键** | 支持外键 | 不支持外键 |
| **锁** | 支持行锁（并发更高）和表锁 | 只支持表锁（不会死锁） |
| **索引** | 不支持全文索引 | 支持全文索引（查询更快） |
| **适用场景** | 并发量高或者需要事务 | 读远多于写且不需要事务 |

### 2、如何防止sql注入？

在编写sql语句时，使用参数传递的方式去给变量赋值而不是直接嵌入到sql语句中

在mybatis的mapper中，我们可以用#{value}的方式去传递参数防止sql注入。

### 3、怎样实现幂等

幂等：也就是相同条件下对一个业务的操作,不管操作多少次,结果都是一样

实现方案：唯一索引（通常是主键），乐观锁（通常是version字段），token+redis（令牌-缓存）

token：token是服务端生成的一串字符串，当客户端登录后，服务端将token作为令牌发送给客户端，之后客户端访问服务端时，只需要校验token即可，不需要再验证用户和密码，redis作为缓存保存令牌到服务端。

### 4、一条sql语句的执行流程

Mysql分为server层和存储引擎层

建立连接：server层的连接器验证客户端发来的用户名和密码是否正确，如果错误就返回错误提示，如果正确就校验用户的权限

查询缓存：当执行sql查询语句时，mysql会先去缓存中查询是否有记录，如果有记录就直接返回，没有的话就去数据库中查询，然后将查询记录记录到缓存中

分析器：分析器的作用是对sql语句进行词法分析和语法分析，词法分析就是对sql语句中的关键词进行分析，比如查询到select关键词就知道这是一条查询语句，此外还会分析被操作的表，条件语句的字段等，语法分析就是判断这条语法是否正确。

优化器：对sql语句进行优化

执行器：调用存储引擎的接口，执行sql语句，执行之前会判断一下有没有权限

### 5、Mysql的事务特性（ACID）

原子性：单个事务是不可分割的最小工作单元，事务中的所有操作，要么全都执行成功，要么全都执行失败。通过undolog日志解决

一致性：数据库总是从一个一致性的状态转换到另外一个一致性的状态。如果事务成功完成，那么所有变化都将成功应用，如果失败，就会回滚到原始状态。通过undolog日志解决

持久性：一旦事务提交，则其所做的修改就会永久保存到数据库中。此时即使系统崩溃，修改的数据也不会丢失。通过redolog日志解决

隔离性：事务与事务之间互不影响。通过加锁实现

### 6、 Mysql事务的隔离级别，脏读、不可重复度，幻读。

脏读：事务读取到了未提交的数据，比如A查询到了B修改了但未提交的数据

不可重复读：事务对同一内容的两次查询结果不一致，比如A查询到数据不存在，B插入并提交，A再次查询时发现存在

幻读：事务在插入前检查到数据不存在，插入时却发现数据已经存在无法插入，比如A发现数据不存在准备插入，但B此时插入数据并提交

不可重复读重点在于update，而幻读的重点在于insert或delete。

Mysql事务的隔离级别从低到高为：

读未提交RC：所有的事务都可以读取其他事务未提交的结果，会发生脏读，不可重复读，幻读

读已提交RU：事务开始时，只能读取到已提交的修改，会发生不可重复读，幻读

可重复读RR：事务开始时，在同一条件的查询返回的内容是一致的，会发生幻读，mysql默认隔离级别Mysql-可重复读的隔离级别在什么情况下会出现幻读\_可重复读为什么会出现幻读\_rsjssc的博客-CSDN博客

可串行化：所有事务依次执行，事务之间不能互相干扰，能防止脏读，不可重复读，幻读

### 7、 锁的类型，行锁和表锁，共享锁和独占锁

|  |  |  |  |
| --- | --- | --- | --- |
| 锁的类型 | 是否会死锁 | 并发量 | 适用场景 |
| 表锁 | 不会死锁 | 并发量低 | 1、读多写少（上一次锁连续读）  2、写特别多（如果用行锁，会导致其他事务长时间锁等待，锁冲突频率变高） |
| 行锁 | 会死锁 | 并发量高 | 并发量高的场合 |
| 页面锁 | 会死锁 | 并发量中 |  |

行锁：行锁就是加在单行上的锁，只存在于InnoDB引擎中，分为共享锁（读锁）和独占锁（写锁）

|  |  |  |
| --- | --- | --- |
| 类型 | 作用 | 类型 |
| 共享锁（读锁） | 获得共享锁的事务被允许读一行，并阻止其他事务获得该行的独占锁 | 共享锁（读锁） |
| 独占锁（写锁） | 获得独占锁的事务被允许写一行，并阻止其他事务获得该行的共享锁和独占锁 | 独占锁（写锁） |

语法：LOCK TABLE {TABLENAME} READ / LOCK TABLE {TABLENAME} WRITE

读锁和写锁的

共同点： 自动加锁

不同点：共享锁可能会死锁，独占锁不会 / 读锁读取数据后释放，写锁要事务结束后才释放

语法：SELECT...+LOCK IN SHARE MODE / INSERT...+FOR UPDATE

此外还有意向共享锁和意向独占锁，这是InnoDB自动添加的，事务在获取读锁和写锁前必须先获取该表的对应意向锁。

表锁：表锁是加在整个表的锁

表锁也有两种锁，共享锁和独占锁，作用与行锁的基本相同

### 8、内关联，全关联，左关联，右关联的区别

内关联（INNER JOIN）：所有的查询结果在关联的两张表中都有记录

全关联：联结了那些在相关表中没有关联的行，分为左关联和右关联

左关联（LEFT JOIN）：从From子句的左边表中选择所有行，可以没有右边的表的行

左关联（RIGHT JOIN）：从From子句的右边表中选择所有行，可以没有左边的表的行

### 9、索引，索引的分类

索引的作用相当于是图书的目录，它是对数据表中一列或多列值进行排序的一种存储结构

索引的分类：普通索引，主键索引，复合索引，唯一索引，外键索引，全文索引

普通索引：

mysql表中最基本的索引

语法：CREAT INDEX index\_name ON TABLE table\_name(column)

主键索引：

在Mysql创建表指定主键的时候，会自动在主键上建立一个主键索引，标志着主键具有唯一性且不为空

复合索引：

复合索引也叫组合索引，在建立索引的时候使用表中的多个字段，就比如说使用身份证号和手机号建立索引

语法：CREAT INDEX index\_name ON TABLE table\_name(column1,column2)

唯一索引：

唯一索引标志着该字段具有唯一性

语法：CREAT UNIQUE INDEX index\_name ON TABLE table\_name(column)

如果创建表的时候加索引语法为：

CREATE TABLE tablename(

propname1 type1,

……

propnamen type..n,

UNIQUE INDEX|KEY index\_name (column [(length)] [ ASC | DESC ] ) );

//简化为：UNIQUE INDEX index\_name (column);

### 10、索引的优缺点和使用规则

索引的优点主要体现在：

提高了用户查询的速度，提高了性能

降低了查询中分组和排序的时间

加速表与表之间的连接

通过索引的唯一性，可以确保表中数据的唯一性

索引的缺点主要体现在：

空间：索引占用了磁盘空间

时间：当数据量较大时，索引的创建和维护也是非常耗费时间的

每次对数据进行增删改查时，索引也要进行动态维护，降低了数据的维护速度

总结，以下条件不适合添加索引：

在查询中很少使用的列

数据值很少的列

写操作远多于读操作的列（修改频率大于查询频率的列）

使用索引的规则：

对查询频率高的字段创建索引

对经常需要分组、排序和联合操作的字段创建索引

尽量使用唯一索引

多使用短索引

索引的数目不要不多（不然会降低增删改的效率）

### 11、索引的原理，B+树的优点

首先我们了解一下B树（Balance Tree），即平衡树的意思，下图即是一颗B树

B树的每一个节点代表一个磁盘块，每一个节点上面存储了多个键值对和指针

B树相对于平衡二叉树，每个节点存储了更多的键（Key）和数据（Data），并且每个节点拥有更多的子节点，子节点的个数就是B树的阶数，比如上图就是的B树就是三阶B树

基于这个性质，B树查询数据读取磁盘的次数会减少，查找的效率会比平衡二叉树高。

B+树是对B树的进一步优化，下图是一颗B+树

在B+树中，非叶子节点不会存储数据（Data），只会存储键值（Key），而所有的数据都存储在叶子节点上，并且所有数据都是按照顺序排列的

在数据库中，页（节点）的大小是固定的，InnoDB默认为16kb，页如果不存储数据，就可以存储更多的键值，树的阶数会提升，

这样一来我们查询数据访问磁盘的次数会进一步减少，查询的效率更高，根节点默认存储在内存，所以我们访问一次三阶B+树索引的数据只需要2次磁盘IO

因为B+树索引中所有的数据都存储在叶子节点上，并且按顺序排列，所以对数据的范围查找，分组，排序都会更加简单

MyISAM中的B+树索引实现与innodb中的略有不同。在MyISAM中，B+树索引的叶子节点并不存储数据，而是存储数据的文件地址。

B+树相对于B树的特点：

只有叶子节点才会存放数据，非叶子节点只存放索引

B+树中每个节点都通过双向链表连接，叶子节点中的数据通过单向链表连接

非叶子节点中的索引会出现在子节点中，而且是子节点中索引的最大（最小）值

非叶子节点的索引树=子节点数

B+树相对于B树的优点：

普通查询，范围查询，分组查询，排序的速度更快

插入和删除的速度更快

### 12、聚集索引和非聚集索引

以主键为B+树索引的键值构建的B+树索引就是聚集索引

以主键以外的字段作为B+树索引的键值构建的B+树索引就是非聚集索引

非聚集索引与聚集索引的区别在于非聚集索引的叶子节点不存储表中的数据，而是存储该列对应的主键，想要查找数据我们还需要根据主键再去聚集索引中进行查找，这个再根据聚集索引查找数据的过程，我们称为回表。

### 13、联合索引的原理

联合索引是对多个列进行索引

联合索引也是通过B+树实现的，特点如下：

B+树的键值数量不是1，而是2或2以上

B+树在对第一个索引排序的基础上，再去对第二个索引排序，再去对第三个，依次排序

联合索引遵循最左匹配原则

最左匹配原则：在联合索引中查询时，从左边的字段开始匹配，若条件中的字段顺序与联合索引中字段顺序一致，就可以走索引查询，否则不走

### 14、索引什么时候会失效

使用LIKE模糊查询时%或者\_放在最前面（会变为全表查询）

使用or语句or的两边没有同时使用索引

联合索引查询时没有遵守最左匹配原则

索引列数据类型隐式转化（varchar不加单引号的话可能会自动转换为int型）

对索引列进行计算或使用函数（改变了列值，在索引中找不到）

order by使用错误（排序与索引中的排序不一致）

全盘扫描速度比索引快

Where 子句中使用参数

### 15、高性能数据库优化实战经验：

冷门又重要的6个技巧，高并发数据库优化，架构师一定要收藏\_哔哩哔哩\_bilibili

1.打破范式设计，冗余少量字段方便查询，需要注意源表和冗余表保证同一事务写。

2.关联关系在业务层面约束，不依赖数据库外键

3.字段拓展性，如模板信息这种结构不清晰的字段使用json类型，json检索的问题我的想法是少量key使用虚拟列并建立索引，多条件检索直接异构es

4.冷热分离，源表拆分成多张表，可以把频繁变更的字段放在主表，使用率较低的放在副表，判断依据可以是创建时间、业务域

5.服务拆分在分片字段选择上尽量考虑使用本地事务，让同业务的不同sql命中同一个分表，以避免使用分布式事务

6.尽量使用单表维度sql，原因：join性能差，后期分库分表更方便，前瞻性设计要考虑使用哪种ID主键策略

### 16、sql语句中group by的使用

group by 根据字面意思就是对数据进行分组，所谓分组就是将数据集划分为若干个不同的小区域，针对每个小区域单独数据处理

注意：group by语句中select 查询的字段要么就是分组依据，要么在聚合函数中

select 类别,sun(分数) from 成绩 group by 类别 order by sum(分数) desc

### 17、sql书写顺序

SELECT <字段名>

FROM <表名>

JOIN <表名>

ON <连接条件>

WHERE <筛选条件>

GROUP BY <字段名>

HAVING <筛选条件>

UNION

ORDER BY <字段名>

LIMIT <限制行数>;

### 18、关联语句join与合并语句union

Join是关联语句，可以把两个表通过连接条件合并一个数据集

inner join：内连接，只有在两个表中都符合连接条件的记录才会被收集

left join：左连接的结果包括左表中的所有记录和右表中满足连接条件的记录

right join：右连接的结果包括右表中的所有记录和左表中满足连接条件的记录

full join：全连接的结果是左右表的并集

cross join：左右表做笛卡尔积

union (all)：union能将多个select的记录进行合并，这多个记录必须列数一致，列名一致，列数据类型对应一致，如果是union all则返回的记录集包括重复记录