面试真题

**【美团】MySQL什么情况下需要分库分表**

**应用场景**

1. 单表数据量过大
2. 表结构复杂（字段过多或索引过多）导致查询效率低
3. 并发访问量过高
4. 需要分布式管理实现数据的高可用性和负载均衡

**实现策略**

* 水平 拆分 数据行
* 垂直 拆分 字段

**常见分库分表中间件**

* Mycat（MySQL）
* ShardingSphere（Apache）

【美团】什么是分库分表？分库分表有哪些类型（或策略）？

**分库分表**是数据库性能优化的一种方法。

1）**水平 分 数据行**：将相同的表结构复制一份到另一个库中，每个库的表结构是一样的，可以减少单一数据库的读写压力，在大数量的情况下提高读写性能。

2）**垂直 分 字段**：可以根据业务功能或模块进行分库，如将用户数据、订单数据分别存储在不同的数据库中。

**【美团】乐观锁和悲观锁了解吗？**

乐观锁和悲观锁用于并发控制。

乐观地假设没有冲突，操作不加锁（提交时检查冲突有则回滚无则提交），通过版本号或时间戳实现，适用于读多写少

悲观地假设有并发冲突，操作时加锁，通过select for update或lock in share mode实现，适用于数据强一致性

【腾讯】MySQL 的存储引擎有哪些？为什么常用InnoDB？

MySQL 的存储引擎常用的主要有 3 个：

InnoDB

支持行级锁，高并发性能更优（常用原因）

支持事务，保证数据一致性（常用原因）

适合频繁的 读写混合 操作

MyISAM

仅支持表级锁，写操作性能差

不支持事务，崩溃后恢复较困难

查询性能高，适合读密集场景

MEMORY

所有数据在内存中，速度快、安全性不高

适用于存放临时数据（MSQL中使用该引擎作为临时表存放查询的中间结果）

【腾讯】B+ 树和 B 树的比较

B+ 树的所有数据仅存储在叶子节点，非叶节点仅存放索引，内部节点更紧凑，能容纳更多索引，提高磁盘I/O性能

B树的每个节点既存储索引也存储数据，占用更多节点空间

B+ 树叶子节点通过双向链表连接，支持范围查询和顺序遍历

B树的中序遍历效率较低，不便于范围查询

【腾讯】除了聚簇索引，还有什么索引？

聚簇索引：决定了数据在磁盘上的物理存储顺序，通常用于表的主键，适合范围查询、排序查询和高频的主键访问

非聚簇索引：索引和数据在不同的物理位置，索引指向数据的地址，适合多种查询条件

唯一索引：保证索引列中的值是唯一的，确保数据完整性

组合索引：通过组合多个列来加速复杂的多条件查询，提高多列查询的效率，遵循“最左前缀原则”

全文索引：用于处理大文本字段的全文检索，适用于关键词匹配

哈希索引：适合精确匹配、不支持范围查询。

【腾讯】二级索引存放的有哪些数据？

索引又可以分成聚簇索引和非聚簇索引（二级索引），它们区别就在于叶子节点存放的是什么数据：

* 聚簇索引的叶子节点存放的是完整的数据；
* 二级索引的叶子节点存放的是主键值。

**二级索引** 是逻辑概念，指非主键字段上的索引；而 **非聚簇索引** 是物理存储概念，指索引与数据分离的索引，在某些环境下（如 MyISAM），两者是相同的。

【腾讯】索引失效的情况

索引失效的情况：

* 当我们使用左或者左右模糊匹配的时候，也就是 like %xx 或者 like %xx%这两种方式都会造成索引失效；
* 当我们在查询条件中对索引列使用函数，就会导致索引失效。
* MySQL 在遇到字符串和数字比较的时候，会自动把字符串转为数字，然后再进行比较。如果字符串是索引列，而条件语句中的输入参数是数字的话，那么索引列字符串会发生隐式类型转换，由于隐式类型转换是通过 CAST 函数实现的，等同于对索引列使用了函数，所以就会导致索引失效。
* 当我们在查询条件中对索引列进行表达式计算，也是无法走索引的。
* 联合索引要能正确使用需要遵循最左匹配原则，也就是按照最左优先的方式进行索引的匹配，否则就会导致索引失效。

【腾讯】事务隔离级别有哪些？事务之间怎么避免脏读的？

1）脏读（Dirty Read）：

• 一个事务读取到另一个事务未提交的数据。如果该未提交事务最终被回滚，那么第一个事务读取的数据就是不一致的（脏的）。

2）不可重复读（Non-repeatable Read）：

• 在同一事务中，读取同一数据两次，但由于其他事务的提交，读取的结果不同。例如，事务 A 读取了一行数据，事务 B 修改并提交了这行数据，导致事务 A 再次读取时得到不同的值。

3）幻读（Phantom Read）：

• 在同一事务中，执行相同的查询操作，返回的结果集由于其他事务的插入而发生变化。例如，事务 A 查询符合某条件的记录，事务 B 插入了新记录并提交，导致事务 A 再次查询时看到不同的记录数量。

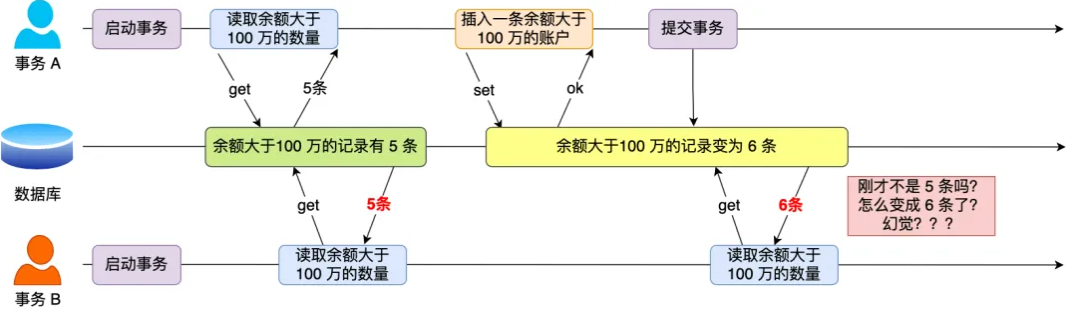
* **读未提交**，指一个事务还没提交时，它做的变更就能被其他事务看到；可能发生脏读、不可重复读、幻读现象；
* **读已提交**，指一个事务提交之后，它做的变更才能被其他事务看到；解决脏读（即不会读到未提交事务的数据）；
* **可重复读**，指一个事务执行过程中看到的数据，一直跟这个事务启动时看到的数据是一致的，**MySQL InnoDB 引擎的默认隔离级别**；解决不可重复读（即数据内容前后一致）；
* **串行化**；会对记录加上读写锁，在多个事务对这条记录进行读写操作时，如果发生了读写冲突的时候，后访问的事务必须等前一个事务执行完成，才能继续执行；解决幻读（即数据记录数量前后一致）。

所以，**要解决脏读现象，就要升级到「读提交」以上的隔离级别。**

【腾讯】什么情况下会出现幻读？

**在一个事务内多次查询某个符合查询条件的「记录数量」，如果出现前后两次查询到的记录数量不一样的情况，就意味着发生了「幻读」现象。**

举个栗子。假设有 A 和 B 这两个事务同时在处理，事务 A 先开始从数据库查询账户余额大于 100 万的记录，发现共有 5 条，然后事务 B 也按相同的搜索条件也是查询出了 5 条记录。接下来，事务 A 插入了一条余额超过 100 万的账号，并提交了事务，此时数据库超过 100 万余额的账号个数就变为 6。然后事务 B 再次查询账户余额大于 100 万的记录，此时查询到的记录数量有 6 条，**发现和前一次读到的记录数量不一样了，就感觉发生了幻觉一样，这种现象就被称为幻读。**



【腾讯】事务的 MVCC 是怎么实现的？

MVCC（多版本并发控制）：通过「事务的 Read View 里的字段」和「记录中的两个隐藏列」的比对，来控制并发事务访问同一个记录时的行为。

对于「读提交」和「可重复读」隔离级别的事务来说，它们是通过 Read View 来实现的，它们的区别在于创建 Read View 的时机不同：

* 「读提交」隔离级别是在每个 select 都会生成一个新的 Read View，也意味着，事务期间的多次读取同一条数据，前后两次读的数据可能会出现不一致，因为可能这期间另外一个事务修改了该记录，并提交了事务。
* 「可重复读」隔离级别是启动事务时生成一个 Read View，然后整个事务期间都在用这个 Read View，这样就保证了在事务期间读到的数据都是事务启动前的记录。



面向简历

MySQL存储引擎

MySQL 的存储引擎有哪些？为什么常用InnoDB？

读写分离

分库分表

MySQL什么情况下需要分库分表

分库分表有哪些类型/策略

索引

B+ 树和 B 树的比较

除了聚簇索引，还有什么索引？

二级索引存放的有哪些数据？

索引失效的情况

事务

事务隔离级别有哪些？事务之间怎么避免脏读的？

什么情况下会出现幻读？

事务的 MVCC 是怎么实现的？

日志

锁机制

乐观锁和悲观锁了解吗

知识框架

MySQL基础

MySQL的事务四大特性是怎么实现的

ACID原一隔持

ACID四大特性：

原子性（Atomicity）

含义：事务中所有操作要么全部完成，要么全部失败回滚。

实现：MySQL通过日志和回滚段来实现。

一致性（Consistency）

含义：事务在执行前后数据库从一个一致性状态转移到另一个一致性状态。

实现：MySQL通过检查约束、触发器和存储过程等手段来保证一致性。

隔离性（Isolation）

含义：多个事务并发执行感觉不到其他的存在。

实现：MySQL通过锁机制和多版本并发控制（MVCC）实现事务隔离级别：

读未提交、读已提交、可重复读、串行化。

持久性（Durability）

含义：事务一旦提交，改变永久保存。

实现：MySQL通过将事务日志写入磁盘来实现。

详细讲一讲MVCC

**版读写隔收**

多版本并发控制（**M**ulti-**V**ersion **C**oncurrency **C**ontrol）

允许多个事务读写并发执行，提高了数据库系统的性能。

1. **版本号**： 每个数据行一个版本号，通常是一个递增的整数。
2. **读操作**：根据事务启动时间和版本号判断。只有那些在事务启动前提交的数据版本对当前事务是可见的。
3. **写操作**：不直接修改，而是创建新数据版本。原始数据版本依然对其他事务可见，直到新的版本被成功提交。
4. **事务隔离级别**：例如，可重复读级别在整个事务期间只看到一个固定快照，而读已提交级别则在每个读操作时都获取当前最新的数据版本。

**回收过期数据**：定期清理已经过期的数据版本。通过后台任务如MySQL中的purge线程完成。

详细描述一条 SQL 语句在 MySQL 中的执行过程。

1. 先通过连接器校验权限
2. 利用分析器进行 SQL 语句的词法分析和语法分析，构建解析树
3. 使用优化器选择合适的索引和表连接顺序，最终选择一个最佳的执行计划
4. 利用执行器，调用引擎层查询数据，返回结果集给客户端

MySQL 的存储引擎有哪些？它们之间有什么区别？

1）**InnoDB**：

* 支持事务和外键，使用行级锁。
* 提供高并发性能，适用于高负载的 OLTP 应用。
* 数据以聚集索引的方式存储，提高查询效率。

2）**MyISAM**：

* 不支持事务和外键，使用表级锁。
* 适合读取多、更新少的场景，如数据仓库。

**3）MEMORY**：数据存在内存中，服务器重启后丢失，适用于临时数据存储。

4）**NDBCluster：**适合分布式应用，提供行级锁和自动分区。

5）**ARCHIVE**：支持高效的插入和压缩、不支持索引。适合日志数据存储。

MySQL 中 InnoDB 存储引擎与 MyISAM 存储引擎的区别是什么？

**MyISAM ：**

1）MyISAM 是基于 ISAM 引擎而来的，支持全文检索、数据压缩、空间函数，**不支持事务和行级锁，只有表级别锁**，它适用于 OLAP 场景，也就是分析类的，基本上都是读取，不会有什么写入动作的场景。

2）MyISAM 的索引也是 B+ 树，只是不像 InnoDB 那种叶子节点会存储完整的数据，MyISAM 的数据是独立于索引单独存储的，所以主键和非主键索引差别不大。

3）MyISAM 不支持崩溃后的安全恢复，而 InnoDB 有个 redolog 可以支持安全恢复。

4）MyISAM 写入性能差。因为锁的粒度太粗了，不支持行锁，只有表锁，所以写入的时候会对整张表加锁。不过有个并发插入的开关，开启之后当数据中间没有空洞的时候，也就是插入的新数据是从末尾插入时，读取数据是不会阻塞的。

**InnoDB（MySQL 默认引擎）：**

1）InnoDB 支持事务，实现了四种标准的隔离级别，利用 MVCC 来支持高并发，默认事务隔离级别为可重复读，支持行锁，利用行锁+间隙锁提供可重复读级别下防止幻读的能力，支持崩溃后的数据安全恢复。

2）支持外键，不过一般互联网项目都不会用外键的，性能太差，利用业务代码来实现约束即可。

3）由于 InnoDB 使用行级锁定和支持事务，因此在并发性能方面表现较好，特别是在多个用户同时对数据库进行读写操作时。

4）InnoDB 的主键索引称为聚簇索引，也就是数据和索引是放在一起的，这与 MyISAM 有所不同，并且它的辅助索引(非主键索引)只存储索引值与主键，因此当辅助索引不能覆盖查询的列时，需要通过找到的主键再去聚簇索引查询数据，这个过程称之为回表。

**MyISAM 和 InnoDB 适用场景**

InnoDB 更适合需要高并发、事务处理和数据完整性保证的场景，例如电商平台、金融系统和社交网络等。

而 MyISAM 更适合读操作远多于写操作且对数据完整性要求不高的场景，例如内容管理系统、博客平台和报表系统等。

如何使用 MySQL 的 EXPLAIN 语句进行查询分析？

explain 主要用来 SQL 分析，它主要的属性详解如下：

* id：查询的执行顺序的标识符，值越大优先级越高。简单查询的 id 通常为 1，复杂查询（如包含子查询或 UNION）的 id 会有多个。
* select\_type（**重要**）：查询的类型，如 SIMPLE（简单查询）、PRIMARY（主查询）、SUBQUERY（子查询）等。
* table：查询的数据表。
* type（**重要**）：访问类型，如 ALL（全表扫描）、index（索引扫描）、range（范围扫描）等。一般来说，性能从好到差的顺序是：const > eq\_ref > ref > range > index > ALL。
* possible\_keys：可能用到的索引。
* key（**重要**）：实际用到的索引。
* key\_len：用到索引的长度。
* ref：显示索引的哪一列被使用。
* rows（**重要**）：估计要读取的行数，值越小越好。
* filtered：显示查询条件过滤掉的行的百分比。一个高百分比表示查询条件的选择性好。
* Extra（**重要**）：额外信息，如 Using index（表示使用覆盖索引）、Using where（表示使用 WHERE 条件进行过滤）、Using temporary（表示使用临时表）、Using filesort（表示需要额外的排序步骤）。

type 详解：

* system：表示查询的表只有一行（系统表）。这是一个特殊的情况，不常见。
* const：表示查询的表最多只有一行匹配结果。这通常发生在查询条件是**主键**或**唯一索引**，并且是常量比较。
* eq\_ref：表示对于每个来自前一张表的行，MySQL 仅访问一次这个表。这通常发生在连接查询中使用主键或唯一索引的情况下。
* ref：MySQL 使用非唯一索引扫描来查找行。查询条件使用的索引是非唯一的（如普通索引）。
* range：表示 MySQL 会扫描表的一部分，而不是全部行。范围扫描通常出现在使用索引的范围查询中（如 BETWEEN、>, <, >=, <=）。
* index：表示 MySQL 扫描索引中的所有行，而不是表中的所有行。即使索引列的值覆盖查询，也需要扫描整个索引。
* all（性能最差）：表示 MySQL 需要扫描表中的所有行，即全表扫描。通常出现在没有索引的查询条件中。

MySQL 中如何进行 SQL 调优？

平时进行 SQL 调优，主要是通过观察慢 SQL，然后利用 explain 分析查询语句的执行计划，识别性能瓶颈，优化查询语句。

1）合理设计索引，利用联合索引进行覆盖索引的优化，避免回表的发生，减少一次查询和随机 I/O

2）避免 SELECT \* ，只查询必要的字段

3）避免在 SQL 中进行函数计算等操作，使得无法命中索引

4）避免使用 %LIKE，导致全表扫描

5）注意联合索引需满足最左匹配原则

6）不要对无索引字段进行排序操作

7）连表查询需要注意不同字段的字符集是否一致，否则也会导致全表扫描

除此之外，还可以利用缓存来优化，一些变化少或者访问频繁的数据设置到缓存中，减轻数据库的压力，提升查询的效率。

还可以通过业务来优化，例如少展示一些不必要的字段，减少多表查询的情况，将列表查询替换成分页分批查询等等。

MySQL 中 count(\*)、count(1) 和 count(字段名) 有什么区别？

在 MySQL 中，count(\*) 、count(1)和count(字段名) 都是用来**统计行数的聚合函数**，但它们有些许的区别：

**功能上：**

1）count(\*) 会统计表中所有行的数量，包括 null 值（不会忽略任何一行数据）。由于只是计算行数，不需要对具体的列进行处理，因此性能通常较高。

2）count(1) 和 count(\*) 几乎没差别，也会统计表中所有行的数量，包括 null 值。

3）count(字段名) 会统计指定字段不为 null 的行数。这种写法会对指定的字段进行计数，只会统计字段值不为 null 的行。

**效率上：**

1）count(1) 和 count(\*) 效率一致，网上其实众说纷纭，实际上当然得看官网怎么说！There is no performance difference. 没有差异的！

2）count(字段) 的查询就是全表扫描，正常情况下它还需要判断字段是否是 null 值，因此理论上会比 count(1) 和 count(\*) 慢。

但是如果字段不为 null，例如是主键，那么理论上也差不多，而且本质上它们的统计功能不一样，在需要统计 null 的时候，只能用 count(1) 和 count(\*)，不需要统计 null 的时候只能用count(字段)，所以也不用太纠结性能问题。

MySQL 中 DELETE、DROP 和 TRUNCATE 的区别是什么？

1. Drop 用于完全删除数据库表，包括数据和结构。
2. Truncate 只删除数据，不会删除表结构和索引等其他结构。
3. Delete 用于删除行数据，但保留表结构和相关的对象。

从性能来看，Drop > Truncate > Delete

MySQL 中 varchar 和 char 有什么区别？

CHAR 和 VARCHAR 是两种用于存储字符串的列类型，它俩最大的不同就是一个是固定长度，一个是可变长度。

* CHAR(n)：固定长度的字符串。CHAR 列的长度是固定的，即使存储的字符串长度小于定义的长度，**MySQL 也会在字符串的末尾填充空格以达到指定长度**（如果 char 类型的字符串后面有空格的话，innodb 会忽略）。
* VARCHAR(n)：可变长度的字符串。VARCHAR 列的长度是可变的，存储的字符串长度与实际数据长度相等，并且在存储数据时会额外增加 1 到 2 个字节（字符长度超过 255，则使用两个字节）用于存储字符串的长度信息。

理论上来说 CHAR 会比 VARCHAR 快，因为 VARCHAR 长度不固定，处理需要多一次运算，但是实际上这种运算耗时微乎其微，而固定大小在很多场景下比较浪费空间，除非存储的字符确认是固定大小或者本身就很短，不然业务上推荐使用 VARCHAR。

下面以表格方式总结方便横向对比：

| **特点** | **char** | **varchar** |
| --- | --- | --- |
| 存储方式 | 定长字符串（字符串长度小于定义的长度，会使用空格进行填充） | 变长字符串（不会额外填充空格） |
| 存储空间 | 始终占用固定长度空间 | 只占用实际需要的存储空间 |
| 性能影响 | 始终占用固定长度的存储空间，因此在存储时可能会浪费一些空间 (不需要记录额外长度信息，在某些情况下可能更快) | 只占用实际需要的存储空间，因此可以节省存储空间 （需要记录额外长度信息，占据1~2个字节），在某些情况下可能稍微影响性能） |
| 适用场景 | 适合存储固定且短的字符串 | 适合存储变化或较长的字符串 |

如何在 MySQL 中监控和优化慢 SQL？

可以利用 MySQL 自带的 slow\_query\_log 来监控慢 SQL，它是 MySQL 提供的一个日志功能，用于记录执行时间超过特定阈值的 SQL 语句。

对于慢查询，再使用 EXPLAIN 分析执行计划，查看查询的执行顺序、使用的索引、扫描的行数等，以识别潜在的性能瓶颈。

基于 EXPLAIN 再进行针对性的优化，常见的优化方向有：

* 根据 EXPLAIN 的结果，检查是否有合适的索引。若缺失索引，则添加（特别是在 WHERE、JOIN 和 ORDER BY 子句中使用的列上）
* 将复杂的 JOIN 查询拆分成多个简单查询，尽量小表驱动大表
* 避免 SELECT \* ，仅选择需要的字段

MySQL分库分表

**MySQL什么情况下需要分库分表**

**应用场景**

1. 单表数据量过大
2. 并发访问量过高
3. 表结构复杂（字段过多或索引过多）导致查询效率低
4. 需要分布式管理实现数据的高可用性和负载均衡

**实现策略**

* 水平 拆分 数据行
* 垂直 拆分 字段

**常见分库分表中间件**

* Mycat（MySQL）
* ShardingSphere（Apache）

什么是分库分表？分库分表有哪些类型（或策略）？

**分库分表**是数据库性能优化的一种方法。

1）**水平 分 数据行**：将相同的表结构复制一份到另一个库中，每个库的表结构是一样的，可以减少单一数据库的读写压力，在大数量的情况下提高读写性能。

2）**垂直 分 字段**：可以根据业务功能或模块进行分库，如将用户数据、订单数据分别存储在不同的数据库中。

如果组长要求你主导项目中的分库分表，大致的实施流程是？

1）**先分析业务需求**：

* 确定数据量及增长趋势，评估分库分表的必要性。（需要一定的预判但是不要过度设计）

2）**设计分库分表方案**：

* 选择适合的分库和分表策略（水平、垂直、哈希、范围等），并规划分库分表的结构。

3）**实现数据路由**：

* 根据分库分表策略设计数据路由机制，一般通过应用层代码或数据库中间件来实现，将请求路由到相应的数据库或表。

4）**数据迁移**：

* 将现有数据迁移到新的分库分表结构中，可以通过批量导入、ETL 工具或脚本来完成。

对数据库进行分库分表可能会引发哪些问题？

1）**事务问题**

我们使用关系型数据库，有很大一点在于它**保证事务的完整性**。

而分库之后单机事务就用不上了，必须使用分布式事务来解决，而分布式事务相对而言就比较重了，而且大部分的分布式事务只能保证最终一致性，所以**业务上会存在数据不一致的场景**。

2）**连表 JOIN 问题**

在一个库中的时候我们还可以利用 JOIN 来连表查询，而跨库了之后就无法使用 JOIN 了。

此时的解决方案就是在业务代码中进行关联，也就是先把一个表的数据查出来，然后通过得到的结果再去查另一张表，然后利用代码来关联得到最终的结果。

这种方式实现起来稍微比较复杂，不过也是可以接受的。

还有可以适当的冗余一些字段。比如以前的表就存储一个关联 ID，但是业务时常要求返回对应的 Name 或者其他字段。这时候就可以把这些字段冗余到当前表中，来去除需要关联的操作。

或者**通过宽表的形式查询**，比如将数据全量存储至 ES 中，利用 ES 来查询数据。

3）**全局 ID 唯一性问题**

以前单库单表直接使用数据库的自增 ID 即可，但是分库分表之后，使用自增 ID 会导致重复主键的情况，此时需要利用雪花算法或者其他全局唯一 ID 发号器来生成唯一主键。

4）**排序问题**

单表直接通过 order by 进行排序即可，分库分表后直接利用数据库是无法实现排序的。要么利用分库分表中间件的能力进行汇总排序，要么自己在业务代码中排序，要么利用 ES 存储全量数据排序查询。

5）**count 问题**

其实和排序问题类似，单表可以直接 count，分库分表后无法支持，只能多表 count 然后业务代码中累加，或者单独搞一个地方来维护总数，要么还是利用 ES。

MySQL主从同步&读写分离

什么是 MySQL 的主从同步机制？它是如何实现的？

主从复制指的是将数据库中的数据从主库（Master）复制到一个或多个从库（Slave）。主库负责处理所有的写操作（如插入、更新、删除），而从库则负责处理读操作（如查询）。

读写分离是基于主从复制的一种进一步优化策略，将数据库的读操作和写操作分离到不同的服务器上。具体来说：写操作（如新增、修改、删除数据）统一由主库处理，确保数据的一致性和完整性。读操作（如数据查询）则由一个或多个从库处理，提升读取性能并减少主库的负担。

优势：性能提升、负载均衡、数据备份

如何在 MySQL 中实现读写分离？

**做法一：代码封装**

利用代理类，对外暴露正常的读写接口，里面封装了逻辑，将读操作指向从库的数据源，写操作指向主库的数据源。

缺点：如果数据库宕机了，发生主从切换了之后，就得修改配置重启。如果系统是多语言的话，需要为每个语言都实现一个中间层代码，重复开发。

**做法二：使用中间件**

中间件一般而言是独立部署的系统，客户端与这个中间件的交互是通过 SQL 协议的。所以在客户端看来连接的就是一个数据库，通过 SQL 协议交互也可以屏蔽多语言的差异。常见的开源数据库中间件有： MySQL-Proxy、Atlas、ShardingSphere、Mycat 等。

缺点：整体架构多了一个系统需要维护，并且可能成为性能瓶颈，毕竟交互都需要经过它中转。

如何处理 MySQL 的主从同步延迟？

常见解决方式有以下几种：

* **二次查询**。如果从库查不到数据，则再去主库查一遍，由 API 封装这个逻辑即可，算是一个兜底策略，比较简单。不过等于读的压力又转移到主库身上了，如果有不法分子故意查询必定查不到的查询，这就对主库产生冲击了。
* **强制将写之后立马读的操作转移到主库上**。这种属于代码写死了，比如一些写入之后立马查询的操作，就绑定在一起，写死都走主库。不推荐，比较死板。
* **关键业务读写都走主库**，非关键还是读写分离。比如上面我举例的用户注册这种，可以读写主库，这样就不会有登陆报该用户不存在的问题，这种访问量频次应该也不会很多，所以看业务适当调整此类接口。
* **使用缓存**，主库写入后同步到缓存中，这样查询时可以先查询缓存，避免了延迟的问题，不过又引入了缓存数据一致性的问题。

如何在 MySQL 中避免单点故障？

一般会使用**主从架构**来避免单点故障，主数据库处理写操作，从数据库处理读操作，主数据库故障时可以切换到从数据库。

同时会对数据进行定期备份并存储在不同的物理位置，以便在发生故障时能够快速恢复数据。

并且需要建立监控系统，实时监控数据库的健康状态，并在发生故障时及时告警。

MySQL索引

**MySQL 数据排序是怎么实现的？**

排序过程中，如果排序字段命中索引，则利用**索引排序**。

反之，使用**文件排序**。

文件排序中，如果数据量少则在内存中排序，使用单路排序或者双路排序。

如果数据大则利用磁盘文件进行外部排序，一般使用归并排序。

MySQL 的索引类型有哪些？

按数据结构：

* B+树索引
* 哈希索引
* 倒排索引（Full-Text）
* R-树索引（多维空间树）

按索引性质：

* 普通索引（二级索引、辅助索引）
* 主键索引
* 联合索引
* 唯一索引
* 全文索引
* 空间索引

常见的基于 InnoDB B+ 树的索引：

* 聚簇索引（Clustered Index）
* 非聚簇索引（Non-clustered Index）

MySQL InnoDB 引擎中的聚簇索引和非聚簇索引有什么区别？

聚簇索引：

* 索引叶子结点存储的是数据行，可以直接访问完整数据。
* 每个表只能有一个聚簇索引，通常是主键索引，适合范围查询和排序。

非聚簇索引（非主键索引、辅助索引、二级索引）：

* 索引叶子节点存储的是数据行的主键和对应的索引列，需通过主键才能访问完整的数据行。
* 一个表可以有多个非聚簇索引（称之为非主键索引、辅助索引、二级索引），适用于快速查找特定列的数据。

请详细描述 MySQL 的 B+ 树中查询数据的全过程。

叶子（页目录）——槽——分组——数据行

1）数据从根节点找起，根据键值的大小确定左子树还是右子树，从上到下最终定位到叶子节点

2）叶子节点存储实际的数据行记录，但是一页有 16kb 大小，存储的数据行不止一条

3）叶子节点中数据行以组的形式划分，利用页目录结构，通过二分查找可以定位到对应的组

4）定位组后，利用链表遍历就可以找到对应的数据行

重点是先简单提下从根节点遍历到子节点的过程，然后提到叶子节点默认大小为 16KB ，所以理论上能存储很多记录，从而引出页目录，再通过二分查找才能对应记录。

为什么 MySQL 选择使用 B+ 树作为索引结构？

B+ 树在数据库系统中具有以下几个显著优势：

**1）高效的查找性能**：

B+ 树是一种自平衡树，每个叶子节点到根节点的路径长度相同，B+ 树在插入和删除节点时会进行分裂和合并操作，以保持树的平衡，但它又会有一定的冗余节点，使得删除的时候树结构的变化小，更高效。

查找、插入、删除等操作的时间复杂度为 O(log n)，能够保证在大数据量情况下也能有较快的响应时间。

**2）树的高度增长不会过快，使得查询磁盘的 I/O 次数减少**：

B+ 树不像红黑树，数据越多树的高度增长就越快。它是多叉树，**非叶子节点仅保存主键或索引值和页面指针**，使得每一页能容纳更多的记录，因此内存中就能存放更多索引，容易命中缓存，使得查询磁盘的 I/O 次数减少。

**3）范围查询能力强**：

B+ 树特别适合范围查询。因为叶子节点通过链表链接，从根节点定位到叶子节点查找到范围的起点之后，只需要顺序扫描链表即可遍历后续的数据，非常高效。

**B+ 树和 B 树区别**

1）B 树每个节点都存储了完整的数据，而 B+ 树非叶子节点仅存储 key 和指针，完整数据存储在叶子节点。这使得 B+ 树可以在内存中存放更多索引页，减少磁盘查询次数。

2）B+ 树叶子组成了链表，便于区间查找，而 B 树只能每一层遍历查找。

3）B+ 树查询时间更平均、稳定，都需要从根节点扫描到叶子节点。而 B 树则在非叶子节点就可能找到对应的数据返回。

MySQL 中的回表是什么？

**回表简介**

1）索引结构：MySQL 使用 B+ 树索引结构来加速数据的查找。B+ 树是一种多叉树，它的叶子节点中存储了完整的数据行，而非叶子节点存储了索引的键值和指向下一级节点的指针。

2）索引查询：当执行一个查询语句时，MySQL 会首先根据查询条件从 B+ 树的根节点开始进行查找，逐层下降直到找到满足条件的叶子节点。如果查询条件匹配了索引的键值，则可以直接从叶子节点中获取到完整的数据行，无需回表。

3）回表现象：在某些情况下，索引无法提供查询所需的所有信息，这时就需要进行回表操作。例如，如果查询语句需要返回的字段不在索引中，或者查询条件需要使用到索引之外的字段，就会发生回表现象。

4）回表过程：当发生回表时，MySQL 会根据叶子节点中的主键值再次去数据表中查找对应的数据行，以获取到缺失的数据。这个过程会增加额外的 IO 消耗和网络传输时间，降低查询性能。

5）回表优化：为了减少回表对性能的影响，可以采取一些优化措施。例如，使用覆盖索引来包含查询语句所需的所有字段，避免回表操作；或者使用索引的聚簇特性，将相关的字段放在一起，减少回表次数。

**怎么查看有没有发生回表？**

方案：使用 EXPLAIN 分析查询计划中的 type 列和 Extra 列来判断。如果 type 列的值为 ref，表示使用了索引进行查询；Extra 列中的 Using index 表示没有发生回表。

示例：SELECT id, name FROM users WHERE age > 25;

如果 age 列上有索引，但 id 和 name 列不在索引中，那么执行查询时可能会发生回表。因此可以采用覆盖索引，将查询字段都添加到一个联合索引中。

**回表经常出现的场景**

1）需要查询的字段不在索引中：当查询语句需要返回的字段不在索引中时，就会发生回表。例如，如果索引只包含了用户的 ID，而查询语句需要返回用户的姓名和年龄，就需要进行回表操作。

2）使用了非索引字段进行查询：如果查询语句需要使用到索引之外的字段进行条件过滤，也会导致回表。例如，如果有一个索引是用户的 ID，但查询语句需要根据用户的姓名进行查询，就需要回表操作。

3）覆盖索引不可用：覆盖索引是指索引包含了查询语句需要返回的所有字段。如果覆盖索引不可用，即索引中不包含所有需要的字段，就会发生回表。这通常发生在查询需要返回大量字段或者字段类型较大的情况下。

4）查询结果需要排序或分组：如果查询语句需要对结果进行排序或分组操作，而排序或分组的字段不在索引中，也会导致回表。因为排序或分组需要对完整的数据行进行操作。

5）索引选择性低：索引选择性是指索引中不同键值的唯一性程度。如果索引选择性低，即索引中的键值重复较多，那么在进行索引查询时，可能会有多条数据落在同一个叶子节点上，这就需要进行回表操作。

**注意事项**

回表并不一定是性能问题的根本原因，有时候回表是无法避免的。但在一些特定场景下，可以通过优化查询语句、设计合适的索引或者调整数据库的配置来减少回表的次数，提升查询性能，避免回表并不是银弹！！！

MySQL 索引的最左前缀匹配原则是什么？

MySQL 索引的最左前缀匹配原则指的是在使用**联合索引**时，查询条件必须从索引的最左侧开始匹配。如果一个联合索引包含多个列，查询条件必须包含第一个列的条件，然后是第二个列，以此类推。

**底层原理**：因为联合索引在 B+ 树中的排列方式遵循“从左到右”的顺序，例如联合索引 (first\_name, last\_name, age) 会按照 (first\_name, last\_name, age) 的顺序在 B+ 树中进行排序。

MySQL 在查找时会优先使用 first\_name 作为匹配依据，然后依次使用 last\_name 和 age。因此，组合索引能够从左到右依次高效匹配，跳过最左侧字段会导致无法利用该索引。

MySQL 的覆盖索引是什么？

MySQL 的覆盖索引（Covering Index）是指二级索引中包含了查询所需的所有字段，从而使查询可以仅通过访问二级索引而不需要访问实际的表数据（主键索引）。

**覆盖索引的优点**

* 减少I/O操作：因为查询可以直接从索引中获取所有需要的数据，避免了访问实际表的数据页，从而减少了I/O操作。
* 提高查询速度：索引比表数据更紧凑，因此从索引中读取数据比从表中读取要快。
* 减少内存占用：只需要读取索引页而不是表数据页，可以减少内存占用。

MySQL 的索引下推是什么？

索引下推（Index Condition Pushdown, ICP）是一种减少回表查询，提高查询效率的技术。它允许 MySQL 在使用索引查找数据时，将部分查询条件下推到存储引擎层过滤，从而减少需要从表中读取的数据行，减少了 IO（本该由 Server 层做操作，交由存储引擎层因此叫做 “下推” ） 。

**注意：索引下推是应用在联合索引上的。**

原本是查询到主键后会回表查询（因为like的不规范的模糊查询，导致了不能走索引覆盖，即还是要走回表查询），然后将结果返回应用层，应用层会执行where条件过滤，

而现在是引擎层直接能够识别并执行条件筛选语句如 like 语句条件过滤，即使是不满足前缀匹配的条件下，也能通过使用索引覆盖保存的信息筛选出记录（当然了，如果有没覆盖到的字段条件筛选还是要走回表查询的）

在什么情况下，不推荐为数据库建立索引？

一般有以下几种情况不推荐建立索引：

1）**对于数据量很小的表**

* 当表的数据量很小（如几百条记录）时，建立索引并不会显著提高查询性能，反而可能增加管理的复杂性。

2）**频繁更新的表**

* 对于频繁进行插入、更新和删除操作的表，索引会导致额外的维护开销，因为每次数据变更时都需要更新索引，这会影响性能。

3）**执行大量的 SELECT \***

* 此时二级索引可能不会显著提升性能，因为需要大量的回表查询，开销大，数据库最终可能会选择走全表扫描。

4）**低选择性字段（高度重复值的列）**

* 当索引字段的取值重复度高（如性别字段“男”、“女”），索引的效果不明显，且会增加存储空间的浪费。
* 但是，还有一种场景可以考虑，比如表里任务 status 列就 2 个类型，90 % 都是 1（已完成），10%（待执行） 是 2，这个场景会频繁查询 2（待执行）的任务来执行，此时可以建立索引，毕竟能过滤 90 % 的数据。

5）**低频查询的列**

* 对于查询频率极低的字段，建立索引的成本和维护负担可能超过带来的性能提升

6）**长文本字段**（非常长的 varchar 或 JSON、BLOB 和 TEXT 类型，这些类型的列通常包含大量数据）

* 数据量大排序时都无法用内存排，只能利用磁盘文件，排序很慢。
* 数据量大，每个页能存放的行数就少，扫描查询可能会涉及大量的 I/O。
* 文本字段过大都需要额外 blob 页存储，每次查询还需要查额外的页，也是随机 I/O 效率低。
* 这种类型的数据如果有查询需求，不应该放到 MySQL 中，可以需要采用 es 等组件来实现查询。

在 MySQL 中建索引时需要注意哪些事项？

1）不能盲目的建立索引，**索引并不是越多越好**，索引会占用空间，且每次修改的时可能都需要维护索引的数据，消耗资源。

2）对于字段的值有**大量重复的不要建立索引**。比如说：性别字段，在这种重复比例很大的数据行中，建立索引也不能提高检索速度。**但是也不绝对**，例如定时任务的场景，大部分任务都是成功，少部分任务状态是失败的，这时候通过失败状态去查询任务，实际上能过滤大部分成功的任务，效率还是可以的。

3）对于一些**长字段不应该建立索引**。比如 text、longtext 这种类型字段不应该建立索引。因为占据的内存大，扫描的时候大量加载至内存中还耗时，使得提升的性能可能不明显，甚至可能还会降低整体的性能，因为别的缓存数据可能因为它被踢出内存，下次查询还需要从磁盘中获取。

4）当**数据表的修改频率远大于查询频率**时，应该好好考虑是否需要建立索引。因为建立索引会减慢修改的效率，如果很少的查询较多的修改，则得不偿失。

5）对于需要频繁作为条件查询的字段应该建立索引。在 where 关键词后经常查询的字段，建立索引能提高查询的效率，如果有多个条件经常一起查询，则**可以考虑联合索引，减少索引数量**。

6）对经常在 order by、group by、distinct 后面的字段建立索引。这些操作通常需要对结果进行排序、分组或者去重，而索引可以帮助加快这些操作的速度。

MySQL 中使用索引一定有效吗？如何排查索引效果？

**索引不一定有效**。

例如查询条件中不包含索引列、低基数列索引效果不佳，或查询条件复杂且不匹配索引的顺序。

对于一些小表，MySQL可能选择全表扫描而非使用索引，因为全表扫描的开销可能更小。

最终是否用上索引是根据 MySQL 成本计算决定的，评估 CPU 和 I/O 成本最终选择用辅助索引还是全表扫描。有时候确实是全表扫描成本低所以没用上索引。但有时候由于一些统计数据的不准确，导致成本计算误判，而没用上索引。

**排查索引效果的方法**：使用 EXPLAIN 命令，通过在查询前加上EXPLAIN，可以查看 MySQL 选择的执行计划，了解是否使用了索引、使用了哪个索引、估算的行数等信息。

主要观察 EXPLAIN 结果以下几点：

* type（访问类型）：这个属性显示了查询使用的访问方法，例如 ALL、index、range 等。当查询使用索引时，这个属性通常会显示为 index 或 range ，表示查询使用了索引访问。如果这个值是 ALL ，则表示查询执行了全表扫描，没有使用索引。
* key（使用的索引）：这个属性显示了查询使用的索引，如果查询使用了索引，则会显示索引的名称。如果这个值是 NULL，则表示查询没有使用索引。
* rows（扫描的行数）：这个属性显示了查询扫描的行数，即查询返回的行数，需要评估下扫描量。

MySQL 中的索引数量是否越多越好？为什么？

**索引并不是越多越好**。因为索引**不论从时间还是空间上都是有一定成本的**

**1）从时间上**

每次对表中的数据进行增删改(INSERT、UPDATE 或 DELETE)的时候，索引也必须被更新，这会增加写入操作的开销。

例如删除了一个 name 为面试鸭的记录，不仅主键索引上需要修改，如果 name 字段有索引，那么 name 索引也需要修改，所以**索引越多需要修改的地方也就越多，时间开销就大了**，并且 B+ 树可能会有页分裂、合并等操作，时间开销就会更大。

还有一点需要注意：MySQL 有个查询优化器，它需要分析当前的查询，选择最优的计划，这过程就需要考虑选择哪个索引的查询成本低。如果索引过多，那么会导致优化器耗费更多的时间在选择上，甚至可能因为数据的不准确而选择了次优的索引。

**2）从空间上**

每建立一个二级索引，都需要新建一个 B+ 树，默认每个数据页都是 16kb，如果数据量很大，索引又很多，占用的空间可不小。

MySQL 三层 B+ 树能存多少数据？

在 MySQL 的 InnoDB 存储引擎中，B+ 树默认数据页大小为 16KB。

**参数**：

* 每个节点页大小为 **16KB**（即 **16384 字节**）。
* 假设每个数据记录的主键和数据大小为 **1KB**（一般会比这个小，但这里取整方便计算）。
* 每个内部节点（非叶子节点）存储的是指向子节点的指针和索引键。

**三层 B+ 树的存储计算**：

* **叶子节点**：第三层为叶子节点，每个叶子节点页可存储 **16 条数据记录**（16KB ÷ 1KB）。
* **第二层（中间层）**：假设每个指针**6 字节**和索引键（一般为 bigint）的大小为 **8 字节**，那么每个中间节点页可以指向 **1170 个叶子节点**（16KB \* 1024 ÷ （6+8）字节）。
* **第一层（根节点）**：根节点可以指向 **1170 个中间节点**。

由此，三层 B+ 树大致能存储的数据总量为：1170 \* 1170 \* 16 = 21902400，一棵三层的 B+ 树在 MySQL 中可以存储大约 **2000 万条记录**。

**InnoDB 中页的大小**

在 InnoDB 中，B+ 树的每个节点通常对应一个 **页（Page）**，默认页大小为 **16KB**。页的大小可以通过调整参数 innodb\_page\_size 来修改（通常为 4KB、8KB 或 16KB）。

注意：在 MySQL 中，B+ 树叶子节点存储的是数据行的完整信息，包括主键和其他字段。

每条记录（即每个数据行）在 B+ 树的叶子节点中按照主键顺序存储，这使得 InnoDB 的 B+ 树既支持高效的单条记录查询，也支持范围查询。

MySQL事务

数据库的脏读、不可重复读和幻读分别是什么？

**1）脏读（Dirty Read）**：

* 一个事务读取到另一个事务未提交的数据。如果该未提交事务最终被回滚，那么第一个事务读取的数据就是不一致的（脏的）。

**2）不可重复读（Non-repeatable Read）**：

* 在同一事务中，读取同一数据两次，但由于其他事务的提交，读取的结果不同。例如，事务 A 读取了一行数据，事务 B 修改并提交了这行数据，导致事务 A 再次读取时得到不同的值。

**3）幻读（Phantom Read）**：

* 在同一事务中，执行相同的查询操作，返回的结果集由于其他事务的插入而发生变化。例如，事务 A 查询符合某条件的记录，事务 B 插入了新记录并提交，导致事务 A 再次查询时看到不同的记录数量。

**扩展知识**

**几种读与隔离级别的关系：**

脏读、不可重复读和幻读是不同隔离级别下可能发生的问题：

* 读未提交允许脏读。
* 读已提交防止脏读，但可能出现不可重复读。
* 可重复读防止脏读和不可重复读，但仍可能出现幻读。
* 串行化防止所有三种问题，但性能开销较大。

**注意不可重复读与幻读的区别**

幻读是指在事务期间，对于数据总量的突然增加或减少，将别的事务提交的读取到了（针对于数据的数量），不可重复读是指对于同一条数据，在事务执行期间，里面的字段内容发生了变化，读取到了别的事务提交修改的数据。（针对于数据的内容）。

MySQL 中的事务隔离级别有哪些？

**1）读未提交（READ UNCOMMITTED）**：

* 这是最低的隔离级别，在该级别下，一个事务可以看到另一个事务尚未提交的数据修改。这可能会导致**脏读**问题，即读取到其他事务未提交的数据。

**2）读已提交（READ COMMITTED）**：

* 在这个级别下，一个事务只能看到已经提交的其他事务所做的修改。这可以避免脏读问题，但是可能会引发**不可重复读**问题，即在同一个事务中，相同的查询可能返回不同的结果。

**3）可重复读（REPEATABLE READ）**：

* 在这个级别下，确保在一个事务中的多个查询返回的结果是一致的。这可以避免不可重复读问题，但是可能会引发**幻读**问题，即在同一个事务中，多次查询可能返回不同数量的行（MySQL 默认的隔离级别）。

**4）串行化（SERIALIZABLE）**：

* 这是最高的隔离级别，在这个级别下，事务串行执行，即每个事务都会等待前一个事务执行完毕才会开始执行。这可以避免所有的并发问题，但是会大大降低并发性能。

MySQL 默认的事务隔离级别是什么？为什么选择这个级别？

MySQL 默认的隔离级别是可重复读（ Repeatable Read ），即 RR。

原因是为了兼容早期 binlog 的 statement 格式问题，如果是使用读已提交、读未提交等隔离级别，使用了 statement 格式的 binlog 会导致主从（备）数据库数据不一致问题。

你们生产环境的 MySQL 中使用了什么事务隔离级别？为什么？

MySQL 数据库的默认隔离级别是 RR（可重复读），但是很多大公司把隔离级别改成了 RC（读已提交），主要原因是为了**提高并发**和**降低死锁概率**。

为了解决幻读的问题 RR 相比 RC 多了间隙锁（gap lock）和临键锁（next-key lock）。而 RC 中修改数据仅用行锁，锁定的范围更小，因此相比而言 RC 的并发更高。

MySQL 是如何实现事务的？

MySQL 主要是通过：**锁、Redo Log 、Undo Log、MVCC** 来实现事务。

锁（行锁、间隙锁等等）机制，使用数据并发修改的控制，满足事务的隔离性。

Redo Log（重做日志），它会记录事务对数据库的所有修改，在崩溃时恢复未提交的更改，用来满足事务的持久性。

Undo Log（回滚日志），它会记录事务的反向操作，简单地说就是保存数据的历史版本，用于事务的回滚，使得事务执行失败之后可以恢复之前的样子。实现原子性和隔离性

MVCC（多版本并发控制），满足了非锁定读的需求，提高了并发度，实现了读已提交和可重复读两种隔离级别，实现了事务的隔离性。

看到这有小伙伴问一致性呢？

其实事务主要是为了实现一致性，具体是通过 AID，即原子性、隔离性和持久性来达到一致性的目的。

MySQL 中的 MVCC 是什么？

MVCC（Multi-Version Concurrency Control，多版本并发控制）是一种并发控制机制，允许多个事务同时读取和写入数据库，而无需互相等待，从而提高数据库的并发性能。

在 MVCC 中，数据库为每个事务创建一个数据快照。每当数据被修改时，MySQL 不会立即覆盖原有数据，而是生成新版本的记录。每个记录都保留了对应的版本号或时间戳。

多版本之间串联起来就形成了一条版本链，这样不同时刻启动的事务可以**无锁**地获得不同版本的数据(普通读)。此时读(普通读)写操作不会阻塞。

写操作可以继续写，无非就是会创建新的数据版本（但只有在事务提交后，新版本才会对其他事务可见。未提交的事务修改不会影响其他事务的读取），历史版本记录可供已经启动的事务读取。

如果 MySQL 中没有 MVCC，会有什么影响？

如果没有 MVCC，系统必须频繁地对读写操作进行加锁来保证数据的正确性，因为增加了锁的获取和释放的开销，会导致整体系统响应速度变慢，这种实现叫 LBCC (Lock-Based Concurrent Control)。

MySQL 事务的二阶段提交是什么？

MySQL 事务的二阶段提交是指在 MySQL 中，为了确保**redo log**（重做日志）和 **binlog**（二进制日志）之间的一致性，使用的一种机制。MySQL 通过二阶段提交来保证在**crash recovery**（崩溃恢复）时，不会出现**数据丢失**或**数据不一致**的情况。

**二阶段提交的两个阶段**：

* **准备阶段（Prepare Phase）**：在事务提交时，MySQL 的 InnoDB 引擎会先写入 **redo log**，并将其状态标记为**prepare**，表示事务已经准备提交但还未真正完成。此时的 redo log 是预提交状态，还未标记为完成提交。
* **提交阶段（Commit Phase）**：当 redo log 的状态变为 prepare 后，MySQL Server 会写入 **binlog**（记录用户的 DML 操作）。binlog 写入成功后，MySQL 会通知 InnoDB，将 redo log 状态改为**commit**，完成整个事务的提交过程。

binlog 和 redo log 的区别

**redo log**：

* **重做日志**（redo log）是 InnoDB 引擎内部的事务日志，用于记录数据页的物理修改。
* redo log 是固定大小的环形日志，主要用于**崩溃恢复**。它可以帮助 InnoDB 在崩溃后通过日志重做未写入数据页的数据修改，从而确保数据的持久性。

**binlog**：

* **二进制日志**（binlog）是 MySQL Server 层的日志，用于记录所有数据库的修改操作，包括增删改操作（DML）以及表结构的修改（DDL）。
* binlog 是**追加写入**的日志文件，主要用于**数据恢复**、**主从复制**、**数据备份**等场景。它记录的是 SQL 语句的逻辑修改操作，而非数据页的物理修改。

为什么需要二阶段提交

**如果没有二阶段提交**，关于这两个日志，要么就是先写完 redo log，再写 binlog 或者先写 binlog 再写 redo log。我们来分析一下会产生什么后果。

1）**先写完 redo log，再写 binlog**

写完 redo log 后，MySQL 异常宕机，binlog 还未写入数据。重启后 redo log 记录了，因此可以从 redo log 恢复事务的修改，但是 binlog 并没有本次事务提交的数据。后续通过 binlog 恢复的时候，本次事务的修改就丢了。

2）**先写完 binlog，再写 redo log**

写完 binlog 后，MySQL 异常宕机，redo log 还未写入数据。重启后因为 redo log 中没有记录，所以无法恢复本次事务的修改，但是 binlog 记录了本次事务提交的数据。后续通过 binlog 恢复的时候，本次事务的修改可以复原，但是这和原库的数据又不一致了。

如果有二阶段提交，MySQL 异常宕机恢复后如何保证数据一致呢？

1）**redo log 处于 prepare 阶段，binlog 还未写入**，此时 MySQL 异常宕机。

这个阶段很好理解，由于 redo log 还未 commit ，所以异常恢复后，redo log 中记录的数据也不作数，binlog 内也没有记录数据，此时数据是一致的。

2）**redo log 处于 prepare 阶段，binlog 已写入，但 redo log 还未 commit**，此时 MySQL 异常宕机。

此时仅需对比 redo log 中 prepare 的数据和 binlog 中的数据是都一致即可。

如果一致，则提交事务。不一致，则回滚事务。

直接先写完 redo log，再写 binlog，崩溃恢复后直接判断两个日志数据是否完整不就好了？为什么还要分二阶段?

理论上是可以的，但是从意义上来说有冲突。

因为 redo log 提交了，意味着事务已经提交，**此时是无法回滚的**。如果 binlog 没写入，此时数据就不一致了。

此时还有一种方案就是崩溃恢复时再补 binlog，补数据还是比较麻烦，还不如直接用两阶段了。

两阶段提交其实是一个经典的分布式解决方案，在协商场景下，每个人都被询问说 ok 了才提交，就是为了避免后续的回滚或者补数据的情况。

如何对比 redo log 和 binlog 是一致的？

两个日志都有一个字段：XID。

因此崩溃恢复的时候，扫描 redo log，如果发现有 prepare 的 redo log 则利用它的 XID 去 binlog 查询，如果找到对应的数据，则说明数据都保存了，事务可以提交，反之事务回滚。

组提交

组提交（Group Commit） 是 InnoDB 引擎中用于优化 redo log 的写入过程。它通过将多个事务的 redo log 刷盘操作合并为一次磁盘同步操作，从而减少 fsync 的调用次数，提高 MySQL 在高并发环境下的事务提交效率。

具体工作原理如下：

1. 当多个事务同时提交时，InnoDB 会先将这些事务的 redo log 记录写入到日志缓冲区中，但不会立即将每个事务的日志同步到磁盘。
2. 当组提交条件满足时（如等待时间到达、日志缓冲区达到一定大小等），InnoDB 会将多个事务的日志一次性进行磁盘同步（fsync），将这些事务的 redo log 一起持久化到磁盘。

通过组提交的方式，InnoDB 能够减少多次单独的 fsync 调用，从而降低磁盘 I/O 的压力。

组提交除了能应用到 redo log 中，实际 binlog 的刷盘也可以组提交，但是效果一般，因为它的 write 和 fsync 之间的间隔比较短。

不过可以通过设置以下参数实现 binlog 组提交的效果。

* binlog\_group\_commit\_sync\_delay 参数，表示延迟多少微秒后才调用 fsync;
* binlog\_group\_commit\_sync\_no\_delay\_count 参数，表示累积多少次以后才调用 fsync。

除此之外还有一个参数也需要关注一下，即 **innodb\_flush\_log\_at\_trx\_commit** 。

这个参数控制了 **redo log** 的刷盘策略，是影响组提交效果的重要配置：

* **innodb\_flush\_log\_at\_trx\_commit=1**：每个事务提交时都立即将 **redo log** 刷新到磁盘，最安全，但组提交效果较弱。
* **innodb\_flush\_log\_at\_trx\_commit=2**：每个事务提交时仅将日志写入操作系统缓存，定期刷新到磁盘，组提交效果更明显。
* **innodb\_flush\_log\_at\_trx\_commit=0**：不在事务提交时刷盘，数据可能丢失，但可以最大化组提交的效果。

MySQL 中长事务可能会导致哪些问题？

1）**长时间的锁竞争，阻塞资源**：

* 长事务持有锁的时间较长，容易导致其他事务在尝试获取相同锁时发生阻塞，从而增加系统的等待时间和降低并发性能。
* 业务线程也会因为长时间的数据库请求等待而阻塞，部分业务的阻塞可能还会影响到别的服务，导致产生雪崩，最终使得服务全面崩盘，导致非常严重的线上事故。

2）**死锁风险**：

* 长事务更容易产生死锁，因为多个事务可能在互相等待对方释放锁，导致系统无法继续执行。

3）**主从延迟**：

* 主库需要长时间执行，然后传输给从库，从库又要重放好久，期间可能有很长一段时间数据是不同步的。

4）**回滚导致时间浪费**：

* 如果长事务执行很长一段时间，中间突发状况导致抛错，使得事务回滚了，之前做的执行都浪费了。

MySQL日志

什么是 Write-Ahead Logging (WAL) 技术？它的优点是什么？MySQL 中是否用到了 WAL？

WAL（Write-Ahead Logging）技术是一种数据库事务日志管理技术，它确保在修改真正的数据之前，先将修改记录写入日志。这使得即使系统崩溃，通过日志也能恢复数据。保证了数据的持久性和一致性。

WAL 它的核心思想就是先写日志，再写数据，大致执行流程如下：

1）当一个事务开始时，所有对数据库的修改都会先记录到一个日志文件中，而不是直接应用到数据库文件，这些日志记录了数据的变更信息，可以用于恢复数据。

2）当日志记录被安全地写入磁盘后，才会将这些修改应用到数据库文件中。

MySQL 插入一条 SQL 语句，redo log 记录的是什么？

因为 redo log 是**物理日志**，记录“某页（Page）某位置的数据被修改为某值”。它不记录逻辑操作（如 “插入一行”），而是直接记录对页的变更。所以在插入操作中，**redo log** 记录的是**事务在数据页上的修改**

**数据页的插入点、记录的偏移量和插入的实际数据并更新页目录、页头等元数据**。

MySQL 的 Change Buffer 是什么？它有什么作用？

Change Buffer 是 MySQL InnoDB 存储引擎中的一个机制，**用于暂存对二级索引的插入和更新操作的变更**，而不立即执行这些操作，随后，当 InnoDB 进行合适的条件时（如页被读取或 Flush 操作），会将这些变更写入到二级索引中。

**作用**：

* **提高写入性能**：通过将对二级索引的变更暂存，可以减少对磁盘的频繁写入，提升插入和更新操作的性能。
* **批量处理**：Change Buffer 可以在后续的操作中批量处理这些变更，减少了随机写入的开销。

MySQL 的 Doublewrite Buffer 是什么？它有什么作用？

MySQL 的 Doublewrite Buffer 是 InnoDB 存储引擎中的一个机制，用于确保数据的安全性和一致性。其作用是将数据首先写入一个内存缓冲区（双写缓冲区），然后再将其写入数据文件。这种方式可以防止在写入过程中因崩溃或故障导致数据损坏，确保数据的一致性和完整性。

**工作原理简述**：

* 写入流程：当事务提交时，InnoDB 首先将数据写入 Doublewrite Buffer，再从该缓冲区将数据写入磁盘的实际数据文件。
* 恢复机制：在崩溃恢复时，InnoDB 会使用 Doublewrite Buffer 中的数据来修复损坏的页，保证数据不丢失。

MySQL 中的 Log Buffer 是什么？它有什么作用？

MySQL 中的 Log Buffer 是一个内存区域，用于暂时存储事务日志（redo log）的数据。在 InnoDB 存储引擎中，它的主要作用是提高性能，通过批量写入操作将日志数据从内存中写入磁盘，减少磁盘 I/O 操作的频率。

MySQL锁

**乐观锁和悲观锁了解吗？**

乐观锁和悲观锁用于并发控制。

**乐观锁**

* “乐观”地假设不会发生并发冲突
* 操作时不加锁提交时检查冲突有则回滚无则提交
* 通过版本号或时间戳来实现
* 适用于读多写少（如商品库存）

**悲观锁**

* “悲观”地假设会发生并发冲突
* 操作时加锁完成时释放
* 通过select for update或 lock in share mode实现
* 适用于数据强一致性（金融）

MySQL 中有哪些锁类型？

1）**行级锁（Row Lock）（重点）**：

* 仅对特定的行加锁，允许其他事务并发访问不同的行，适用于高并发场景。

2）**表级锁（Table Lock）（重点）**：

* 对整个表加锁，其他事务无法对该表进行任何读写操作，适用于需要保证完整性的小型表。

3）**意向锁（Intention Lock）**：

* 一种表锁，用于表示某个事务对某行数据加锁的意图，分为意向共享锁（IS）和意向排它锁（IX），主要用于行级锁与表级锁的结合。

4）**共享锁（Shared Lock）（重点）**：

* 允许多个事务并发读取同一资源，但不允许修改。只有在释放共享锁后，其他事务才能获得排它锁。

5）**排它锁（Exclusive Lock）（重点）**：

* 只允许一个事务对资源进行读写，其他事务在获得排它锁之前无法访问该资源。

6）**元数据锁（Metadata Lock, MDL）**：

* 用于保护数据库对象（如表和索引）的元数据，防止在进行 DDL 操作时其他事务对这些对象进行修改。

7）**间隙锁（Gap Lock）（重点）**：

* 针对索引中两个记录之间的间隙加锁，防止其他事务在这个间隙中插入新记录，以避免幻读。间隙锁不锁定具体行，而是锁定行与行之间的空间。

8）**临键锁（Next-Key Lock）（重点）**：

* 是行级锁和间隙锁的结合，锁定具体行和其前面的间隙，确保在一个范围内不会出现幻读。常用于支持可重复读的隔离级别。

9）**插入意向锁（Insert Intention Lock）**：

* 一种特殊的意向锁，用于指示事务打算在某个间隙中插入记录，允许其他事务进行共享锁，但在插入时会阻止其他的排它锁。

10）**自增锁（Auto Increment Lock）**：

* 在插入自增列时，加锁以保证自增值的唯一性，防止并发插入导致的冲突。通常在插入操作时被使用，以确保生成的自增 ID 是唯一的。

MySQL 中如果发生死锁应该如何解决？

**自动检测与回滚**：

* MySQL 自带死锁检测机制（innodb\_deadlock\_detect），当检测到死锁时，数据库会自动回滚其中一个事务，以解除死锁。通常会回滚事务中持有最少资源的那个。
* 也有锁等待超时的参数（innodb\_lock\_wait\_timeout），当获取锁的等待时间超过阈值时，就释放锁进行回滚。

**手动 kill 发生死锁的语句**：

* 可以通过命令，手动快速地找出被阻塞的事务及其线程 ID，然后手动 kill 它，及时释放资源。

MySQL 的乐观锁和悲观锁是什么？

**悲观锁（Pessimistic Locking）**：

* 假设会发生冲突，因此在操作数据之前就**对数据加锁**，确保其他事务无法访问该数据。常见于对数据一致性要求较高的场景。
* 实现方式：使用行级锁或表级锁，例如可以使用 SELECT ... FOR UPDATE 或 LOCK IN SHARE MODE 语句来加锁。

**乐观锁（Optimistic Locking）**：

* 假设不会发生冲突，因此在操作数据时不加锁，而是在更新数据时**进行版本控制或校验**。如果发现数据被其他事务修改，则会拒绝当前事务的修改，需重新尝试。
* 实现方式：通常通过版本号或时间戳来实现，每次更新时检查版本号或时间戳是否一致。