基础

1. 数据库的三范式是什么?

- 第一范式：强调的是列的原子性，即数据库表的每一列都是不可分割的原子数据项。
- 第二范式：要求实体的属性完全依赖于主关键字。所谓完全 依赖是指不能存在仅依赖主关键字一部分的属性。
- 第三范式：任何非主属性不依赖于其它非主属性。

2. MySQL 支持哪些存储引擎?

MySQL 支持多种存储引擎,比如 InnoDB,MyISAM,Memory,Archive 等等.在大多数的情况下,直接选择使用 InnoDB 引擎都是最合适的,InnoDB 也是 MySQL 的默认存储引擎。

MyISAM 和 InnoDB 的区别有哪些：

- InnoDB 支持事务，MyISAM 不支持
- InnoDB 支持外键，而 MyISAM 不支持
- InnoDB 是聚集索引，数据文件是和索引绑在一起的，必须要有主键，通过主键索引效率很高；MyISAM 是非聚集索引，数据文件是分离的，索引保存的是数据文件的指针，主键索引和辅助索引是独立的。
- Innodb 不支持全文索引，而 MyISAM 支持全文索引，查询效率上 MyISAM 要高；
- InnoDB 不保存表的具体行数，MyISAM 用一个变量保存了整个表的行数。
- MyISAM 采用表级锁(table-level locking)；InnoDB 支持行级锁(row-level locking)和表级锁,默认为行级锁。

3. 超键、候选键、主键、外键分别是什么?

- 超键：在关系中能唯一标识元组的属性集称为关系模式的超键。一个属性可以为作为一个超键，多个属性组合在一起也可以作为一个超键。超键包含候选键和主键。
- 候选键：是最小超键，即没有冗余元素的超键。
- 主键：数据库表中对储存数据对象予以唯一和完整标识的数据列或属性的组合。一个数据列只能有一个主键，且主键的取值不能缺失，即不能为空值（Null）。
- 外键：在一个表中存在的另一个表的主键称此表的外键。

4. SQL 约束有哪几种?

- NOT NULL: 用于控制字段的内容一定不能为空（NULL）。
- UNIQUE: 控件字段内容不能重复，一个表允许有多个 Unique 约束。
- PRIMARY KEY: 也是用于控件字段内容不能重复，但它在一个表只允许出现一个。
- FOREIGN KEY: 用于预防破坏表之间连接的动作，也能防止非法数据插入外键列，因为它必须是它指向的那个表中的值之一。
- CHECK: 用于控制字段的值范围。

5. MySQL 中的 varchar 和 char 有什么区别?

char 是一个定长字段,假如申请了 `char(10)` 的空间,那么无论实际存储多少内容.该字段都占用 10 个字符,而 varchar 是变长的,也就是说申请的只是最大长度,占用的空间为实际字符长度+1,最后一个字符存储使用了多长的空间.

在检索效率上来讲,char > varchar,因此在使用中,如果确定某个字段的值的长度,可以使用 char,否则应该尽量使用 varchar.例如存储用户 MD5 加密后的密码,则应该使用 char。

6. MySQL中 in 和 exists 区别

MySQL中的in语句是把外表和内表作hash 连接，而exists语句是对外表作loop循环，每次loop循环再对内表进行查询。一直大家都认为exists比in语句的效率要高，这种说法其实是不准确的。这个是要区分环境的。

如果查询的两个表大小相当，那么用in和exists差别不大。
如果两个表中一个较小，一个是大表，则子查询表大的用exists，子查询表小的用in。
not in 和not exists：如果查询语句使用了not in，那么内外表都进行全表扫描，没有用到索引；而not extsts的子查询依然能用到表上的索引。所以无论那个表大，用not exists都比not in要快。

7. drop、delete与truncate的区别

三者都表示删除，但是三者有一些差别：

	Delete	Truncate	Drop
类型	属于DML	属于DDL	属于DDL
回滚	可回滚	不可回滚	不可回滚
删除内容	表结构还在，删除表的全部或者一部分数据行	表结构还在，删除表中的所有数据	从数据库中删除表，所有的数据行，索引和权限也会被删除
删除速度	删除速度慢，需要逐行删除	删除速度快	删除速度最快

8. 什么是存储过程？有哪些优缺点?

存储过程是一些预编译的 SQL 语句。

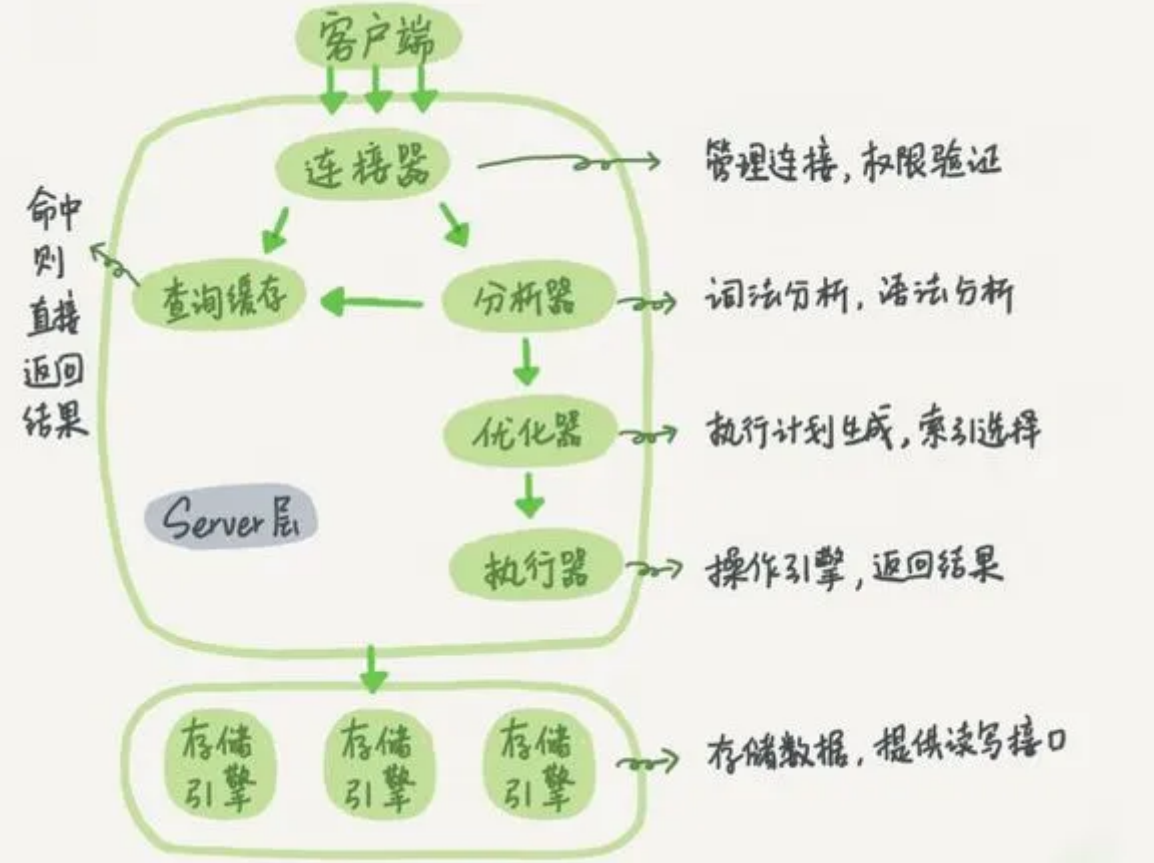
1、更加直白的理解：存储过程可以说是一个记录集，它是由一些 T-SQL 语句组成的代码块，这些 T-SQL 语句代码像一个方法一样实现一些功能（对单表或多表的增删改查），然后再给这个代码块取一个名字，在用到这个功能的时候调用他就行了。

2、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 T_SQL 语句 ，可以降低网络通信量，提高通信速率,可以一定程度上确保数据安全

但是,在互联网项目中,其实是不太推荐存储过程的,比较出名的就是阿里的《Java 开发手册》中禁止使用存储过程,我个人的理解是,在互联网项目中,迭代太快,项目的生命周期也比较短,人员流动相比于传统的项目也更加频繁,在这样的情况下,存储过程的管理确实是没有那么方便,同时,复用性也没有写在服务层那么好。

9. MySQL 执行查询的过程

1. 客户端通过 TCP 连接发送连接请求到 MySQL 连接器，连接器会对该请求进行权限验证及连接资源分配
2. 查缓存。（当判断缓存是否命中时，MySQL 不会进行解析查询语句，而是直接使用 SQL 语句和客户端发送过来的其他原始信息。所以，任何字符上的不同，例如空格、注解等都会导致缓存的不命中。）
3. 语法分析（SQL 语法是否写错了）。 如何把语句给到预处理器，检查数据表和数据列是否存在，解析别名看是否存在歧义。
4. 优化。是否使用索引，生成执行计划。
5. 交给执行器，将数据保存到结果集中，同时会逐步将数据缓存到查询缓存中，最终将结果集返回给客户端。



更新语句执行会复杂一点。需要检查表是否有排它锁，写 binlog，刷盘，是否执行 commit。

事务

1. 什么是数据库事务?

事务是一个不可分割的数据库操作序列，也是数据库并发控制的基本单位，其执行的结果必须使数据库从一种一致性状态变到另一种一致性状态。事务是逻辑上的一组操作，要么都执行，要么都不执行。

事务最经典也经常被拿出来说例子就是转账了。

假如小明要给小红转账1000元，这个转账会涉及到两个关键操作就是：将小明的余额减少1000元，将小红的余额增加1000元。万一在这两个操作之间突然出现错误比如银行系统崩溃，导致小明余额减少而小红的余额没有增加，这样就不对了。事务就是保证这两个关键操作要么都成功，要么都要失败。

2. 介绍一下事务具有的四个特征

事务就是一组原子性的操作，这些操作要么全部发生，要么全部不发生。事务把数据库从一种一致性状态转换成另一种一致性状态。

- 原子性。事务是数据库的逻辑工作单位，事务中包含的各操作要么都做，要么都不做
- 一致性。事 务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态。因此当数据库只包含成功事务提交的结果时，就说数据库处于一致性状态。如果数据库系统 运行中发生故障，有些事务尚未完成就被迫中断，这些未完成事务对数据库所做的修改有一部分已写入物理数据库，这时数据库就处于一种不正确的状态，或者说是 不一致的状态。
- 隔离性。一个事务的执行不能其它事务干扰。即一个事务内部的//操作及使用的数据对其它并发事务是隔离的，并发执行的各个事务之间不能互相干扰。

- 持续性。也称永久性，指一个事务一旦提交，它对数据库中的数据的改变就应该是永久性的。接下来的其它操作或故障不应该对其执行结果有任何影响。

3. 说一下MySQL 的四种隔离级别

- Read Uncommitted（读取未提交内容）

在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少。读取未提交的数据，也被称之为脏读（Dirty Read）。

- Read Committed（读取提交内容）

这是大多数数据库系统的默认隔离级别（但不是 MySQL 默认的）。它满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变。这种隔离级别 也支持所谓 的 不可重复读（Nonrepeatable Read），因为同一事务的其他实例在该实例处理其间可能会有新的 commit，所以同一 select 可能返回不同结果。

- Repeatable Read（可重读）

这是 MySQL 的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。不过理论上，这会导致另一个棘手的问题：幻读 （Phantom Read）。

- Serializable（可串行化）

通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争。

隔离级别	脏读	不可重复读	幻影读
READ-UNCOMMITTED	√	√	√
READ-COMMITTED	×	√	√
REPEATABLE-READ	×	×	√
SERIALIZABLE	×	×	×

MySQL 默认采用的 REPEATABLE_READ隔离级别 Oracle 默认采用的 READ_COMMITTED隔离级别

事务隔离机制的实现基于锁机制和并发调度。其中并发调度使用的是MVVC（多版本并发控制），通过保存修改的旧版本信息来支持并发一致性读和回滚等特性。

因为隔离级别越低，事务请求的锁越少，所以大部分数据库系统的隔离级别都是READ-COMMITTED(读取提交内容):，但是你要知道的是InnoDB 存储引擎默认使用 **REPEATABLE-READ（可重读）**并不会有任何性能损失。

InnoDB 存储引擎在 分布式事务 的情况下一般会用到**SERIALIZABLE(可串行化)**隔离级别。

4. 什么是脏读？幻读？不可重复读？

1、脏读：事务 A 读取了事务 B 更新的数据，然后 B 回滚操作，那么 A 读取到的数据是脏数据

2、不可重复读：事务 A 多次读取同一数据，事务 B 在事务 A 多次读取的过程中，对数据作了更新并提交，导致事务 A 多次读取同一数据时，结果 不一致。

3、幻读：系统管理员 A 将数据库中所有学生的成绩从具体分数改为 ABCDE 等级，但是系统管理员 B 就在这个时候插入了一条具体分数的记录，当系统管理员 A 改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。

不可重复读侧重于修改，幻读侧重于新增或删除（多了或少量行），脏读是一个事务回滚影响另外一个事务。

5. 事务的实现原理

事务是基于重做日志文件(redo log)和回滚日志(undo log)实现的。

每提交一个事务必须先将该事务的所有日志写入到重做日志文件进行持久化，数据库就可以通过重做日志来保证事务的原子性和持久性。

每当有修改事务时，还会产生 undo log，如果需要回滚，则根据 undo log 的反向语句进行逻辑操作，比如 insert 一条记录就 delete 一条记录。undo log 主要实现数据库的一致性。

6. MySQL事务日志介绍下?

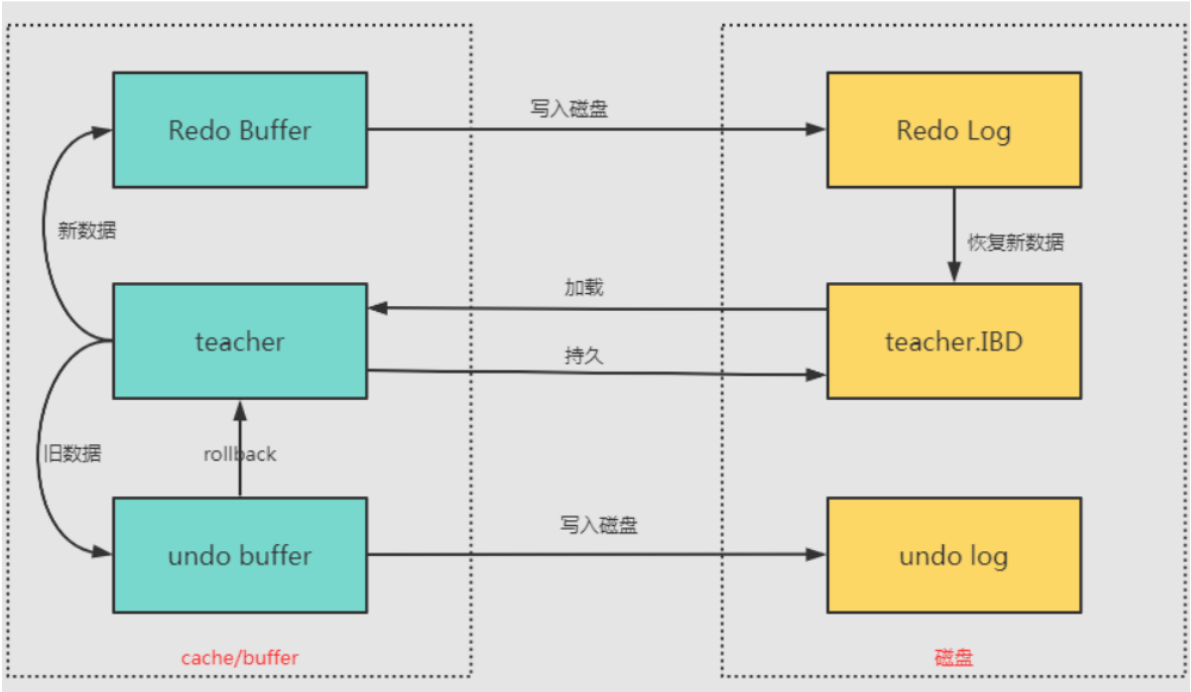
innodb 事务日志包括 redo log 和 undo log。

undo log 指事务开始之前，在操作任何数据之前，首先将需操作的数据备份到一个地方。redo log 指事务中操作的任何数据，将最新的数据备份到一个地方。

事务日志的目的：实例或者介质失败，事务日志文件就能派上用场。

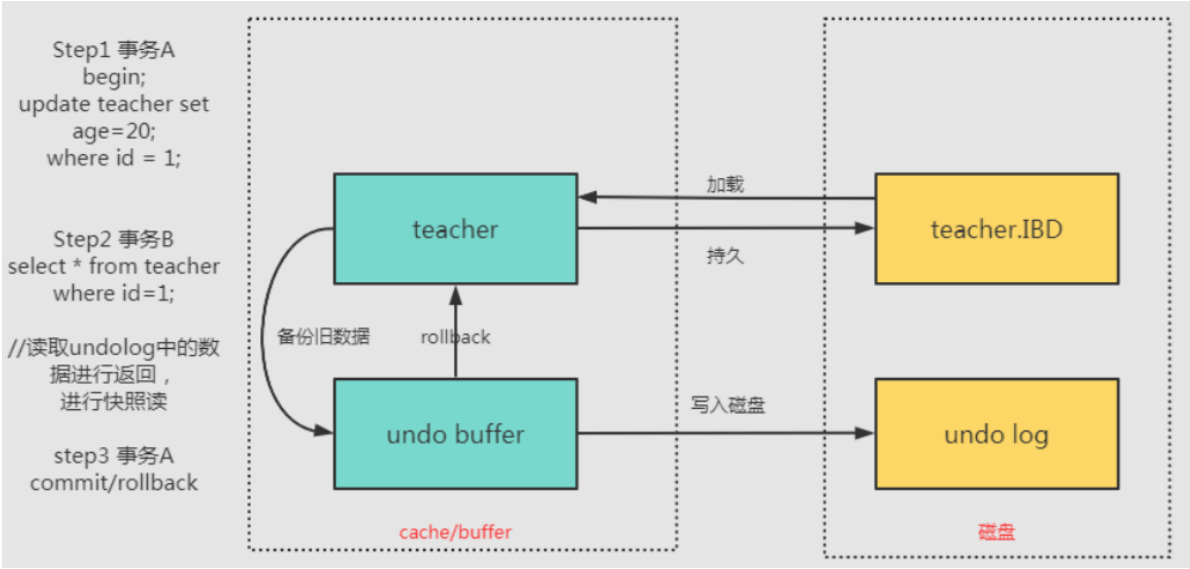
redo log

redo log 不是随着事务的提交才写入的，而是在事务的执行过程中，便开始写入 redo 中。具体的落盘策略可以进行配置 。防止在发生故障的时间点，尚有脏页未写入磁盘，在重启 MySQL 服务的时候，根据 redo log 进行重做，从而达到事务的未入磁盘数据进行持久化这一特性。RedoLog 是为了实现事务的持久性而出现的产物。



undo log

undo log 用来回滚行记录到某个版本。事务未提交之前，Undo 保存了未提交之前的版本数据，Undo 中的数据可作为数据旧版本快照供其他并发事务进行快照读。是为了实现事务的原子性而出现的产物,在 MySQL innodb 存储引擎中用来实现多版本并发控制。



7. 什么是MySQL的 binlog?

MySQL的 binlog 是记录所有数据库表结构变更（例如 CREATE、ALTER TABLE）以及表数据修改（INSERT、UPDATE、DELETE）的二进制日志。binlog 不会记录 SELECT 和 SHOW 这类操作，因为这类操作对数据本身并没有修改，但你可以通过查询通用日志来查看 MySQL 执行过的所有语句。

MySQL binlog 以事件形式记录，还包含语句所执行的消耗的时间，MySQL 的二进制日志是事务安全型的。binlog 的主要目的是复制和恢复。

binlog 有三种格式，各有优缺点：

- **statement**： 基于 SQL 语句的模式，某些语句和函数如 UUID, LOAD DATA INFILE 等在复制过程可能导致数据不一致甚至出错。
- **row**： 基于行的模式，记录的是行的变化，很安全。但是 binlog 会比其他两种模式大很多，在一些大表中清除大量数据时在 binlog 中会生成很多条语句，可能导致从库延迟变大。
- **mixed**： 混合模式，根据语句来选用是 statement 还是 row 模式。

8. 在事务中可以混合使用存储引擎吗?

尽量不要在同一个事务中使用多种存储引擎，MySQL服务器层不管理事务，事务是由下层的存储引擎实现的。

如果在事务中混合使用了事务型和非事务型的表（例如InnoDB和MyISAM表）,在正常提交的情况下不会有什么问题。

但如果该事务需要回滚，非事务型的表上的变更就无法撤销，这会导致数据库处于不一致的状态，这种情况很难修复，事务的最终结果将无法确定。所以，为每张表选择合适的存储引擎非常重要。

9. MySQL中是如何实现事务隔离的?

读未提交和串行化基本上是不需要考虑的隔离级别，前者不加锁限制，后者相当于单线程执行，效率太差。

MySQL 在可重复读级别解决了幻读问题，是通过行锁和间隙锁的组合 Next-Key 锁实现的。

详细原理看这篇文章：https://haicoder.net/note/MySQL-interview/MySQL-interview-MySQL-trans-level.html

10. 什么是 MVCC?

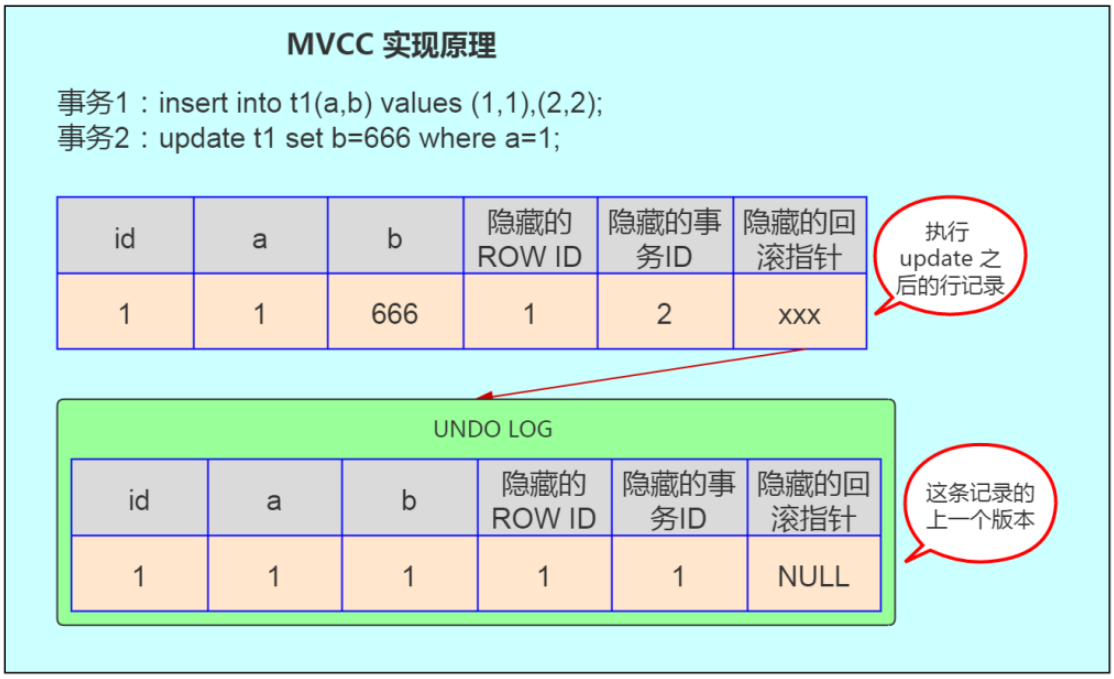
MVCC， 即多版本并发控制。MVCC 的实现，是通过保存数据在某个时间点的快照来实现的。根据事务开始的时间不同，每个事务对同一张表，同一时刻看到的数据可能是不一样的。

11. MVCC 的实现原理

对于 InnoDB ，聚簇索引记录中包含 3 个隐藏的列：

- ROW ID：隐藏的自增 ID，如果表没有主键，InnoDB 会自动按 ROW ID 产生一个聚集索引树。
- 事务 ID：记录最后一次修改该记录的事务 ID。
- 回滚指针：指向这条记录的上一个版本。

我们拿上面的例子，对应解释下 MVCC 的实现原理，如下图：



如图，首先 insert 语句向表 t1 中插入了一条数据，a 字段为 1，b 字段为 1， ROW ID 也为 1 ，事务 ID 假设为 1，回滚指针假设为 null。当执行 update t1 set b=666 where a=1 时，大致步骤如下：

- 数据库会先对满足 a=1 的行加排他锁；
- 然后将原记录复制到 undo 表空间中；
- 修改 b 字段的值为 666，修改事务 ID 为 2；
- 并通过隐藏的回滚指针指向 undo log 中的历史记录；
- 事务提交，释放前面对满足 a=1 的行所加的排他锁。

在前面实验的第 6 步中，session2 查询的结果是 session1 修改之前的记录，这个记录就是**来自 undolog** 中。

因此可以总结出 MVCC 实现的原理大致是：

InnoDB 每一行数据都有一个隐藏的回滚指针，用于指向该行修改前的最后一个历史版本，这个历史版本存放在 undo log 中。如果要执行更新操作，会将原记录放入 undo log 中，并通过隐藏的回滚指针指向 undo log 中的原记录。其它事务此时需要查询时，就是查询 undo log 中这行数据的最后一个历史版本。

MVCC 最大的好处是读不加锁，读写不冲突，极大地增加了 MySQL 的并发性。通过 MVCC，保证了事务 ACID 中的 I（隔离性）特性。

锁

1. 为什么要加锁?

当多个用户并发地存取数据时，在数据库中就会产生多个事务同时存取同一数据的情况。若对并发操作不加控制就可能会读取和存储不正确的数据，破坏数据库的一致性。

保证多用户环境下保证数据库完整性和一致性。

2. 按照锁的粒度分数据库锁有哪些?

在关系型数据库中，可以**按照锁的粒度把数据库锁分**为行级锁(INNODB引擎)、表级锁(MYISAM引擎)和页级锁(BDB引擎)。

行级锁

- 行级锁是MySQL中锁定粒度最细的一种锁，表示只针对当前操作的行进行加锁。行级锁能大大减少数据库操作的冲突。其加锁粒度最小，但加锁的开销也最大。行级锁分为共享锁 和 排他锁。
- 开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。

表级锁

- 表级锁是MySQL中锁定粒度最大的一种锁，表示对当前操作的整张表加锁，它实现简单，资源消耗较少，被大部分MySQL引擎支持。最常使用的MYISAM与INNODB都支持表级锁定。表级锁定分为表共享读锁（共享锁）与表独占写锁（排他锁）。
- 开销小，加锁快；不会出现死锁；锁定粒度大，发出锁冲突的概率最高，并发度最低。

页级锁

- 页级锁是MySQL中锁定粒度介于行级锁和表级锁中间的一种锁。表级锁速度快，但冲突多，行级冲突少，但速度慢。所以取了折衷的页级，一次锁定相邻的一组记录。BDB支持页级锁
- 开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般

MyISAM和InnoDB存储引擎使用的锁：

- MyISAM采用表级锁(table-level locking)。
- InnoDB支持行级锁(row-level locking)和表级锁，默认为行级锁

3. 从锁的类别上分MySQL都有哪些锁呢?

从锁的类别上来讲，有共享锁和排他锁。

- 共享锁: 又叫做读锁。 当用户要进行数据的读取时，对数据加上共享锁。共享锁可以同时加上多个。
- 排他锁: 又叫做写锁。 当用户要进行数据的写入时，对数据加上排他锁。排他锁只可以加一个，他和其他的排他锁，共享锁都相斥。

用上面的例子来说就是用户的行为有两种，一种是来看房，多个用户一起看房是可以接受的。 一种是真正的入住一晚，在这期间，无论是想入住的还是想看房的都不可以。

锁的粒度取决于具体的存储引擎，InnoDB实现了行级锁，页级锁，表级锁。

他们的加锁开销从大到小，并发能力也是从大到小。

4. 数据库的乐观锁和悲观锁是什么？怎么实现的?

数据库管理系统（DBMS）中的并发控制的任务是确保在多个事务同时存取数据库中同一数据时不破坏事务的隔离性和统一性以及数据库的统一性。乐观并发控制（乐观锁）和悲观并发控制（悲观锁）是并发控制主要采用的技术手段。

- 悲观锁：假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作。在查询完数据的时候就把事务锁起来，直到提交事务。实现方式：使用数据库中的锁机制
- 乐观锁：假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性。在修改数据的时候把事务锁起来，通过version的方式来进行锁定。实现方式：乐一般会使用版本号机制或CAS算法实现。

两种锁的使用场景

从上面对两种锁的介绍，我们知道两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下（多读场景），即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。

但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行retry，这样反倒是降低了性能，所以一般多写的场景下用悲观锁就比较合适。

5. InnoDB引擎的行锁是怎么实现的?

InnoDB是基于索引来完成行锁

例: select * from tab_with_index where id = 1 for update;

for update 可以根据条件来完成行锁锁定，并且 id 是有索引键的列，如果 id 不是索引键那么InnoDB将完成表锁，并发将无从谈起

6. 什么是死锁？怎么解决?

死锁是指两个或多个事务在同一资源上相互占用，并请求锁定对方的资源，从而导致恶性循环的现象。

常见的解决死锁的方法

1、如果不同程序会并发存取多个表，尽量约定以相同的顺序访问表，可以大大降低死锁机会。

2、在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁产生概率；

3、对于非常容易产生死锁的业务部分，可以尝试使用升级锁定颗粒度，通过表级锁定来减少死锁产生的概率；

如果业务处理不好可以用分布式事务锁或者使用乐观锁

7. 隔离级别与锁的关系

在Read Uncommitted级别下，读取数据不需要加共享锁，这样就不会跟被修改的数据上的排他锁冲突

在Read Committed级别下，读操作需要加共享锁，但是在语句执行完以后释放共享锁；

在Repeatable Read级别下，读操作需要加共享锁，但是在事务提交之前并不释放共享锁，也就是必须等待事务执行完毕以后才释放共享锁。

SERIALIZABLE 是限制性最强的隔离级别，因为该级别锁定整个范围的键，并一直持有锁，直到事务完成。

8. 优化锁方面的意见?

- 使用较低的隔离级别
- 设计索引，尽量使用索引去访问数据，加锁更加精确，从而减少锁冲突

- 选择合理的事务大小，给记录显示加锁时，最好一次性请求足够级别的锁。列如，修改数据的话，最好申请排他锁，而不是先申请共享锁，修改时在申请排他锁，这样会导致死锁
- 不同的程序访问一组表的时候，应尽量约定一个相同的顺序访问各表，对于一个表而言，尽可能的固定顺序的获取表中的行。这样大大的减少死锁的机会。
- 尽量使用相等条件访问数据，这样可以避免间隙锁对并发插入的影响
- 不要申请超过实际需要的锁级别
- 数据查询的时候不是必要，不要使用加锁。MySQL的MVCC可以实现事务中的查询不用加锁，优化事务性能：MVCC只在committed read（读提交）和 repeatable read （可重复读）两种隔离级别
- 对于特定的事务，可以使用表锁来提高处理速度活着减少死锁的可能。

分库分表

1. 为什么要分库分表?

分表

比如你单表都几千万数据了，你确定你能扛住么？绝对不行，单表数据量太大，会极大影响你的 sql执行的性能，到了后面你的 sql 可能就跑的很慢了。一般来说，就以我的经验来看，单表到几百万的时候，性能就会相对差一些了，你就得分表了。

分表就是把一个表的数据放到多个表中，然后查询的时候你就查一个表。比如按照用户 id 来分表，将一个用户的数据就放在一个表中。然后操作的时候你对一个用户就操作那个表就好了。这样可以控制每个表的数据量在可控的范围内，比如每个表就固定在 200 万以内。

分库

分库就是你一个库一般我们经验而言，最多支撑到并发 2000，一定要扩容了，而且一个健康的单库并发值你最好保持在每秒 1000 左右，不要太大。那么你可以将一个库的数据拆分到多个库中，访问的时候就访问一个库好了。

这就是所谓的分库分表。

#	分库分表前	分库分表后
并发支撑情况	MySQL 单机部署，扛不住高并发	MySQL从单机到多机，能承受的并发增加了多倍
磁盘使用情况	MySQL 单机磁盘容量几乎撑满	拆分为多个库，数据库服务器磁盘使用率大大降低
SQL 执行性能	单表数据量太大，SQL 越跑越慢	单表数据量减少，SQL 执行效率明显提升

2. 用过哪些分库分表中间件？不同的分库分表中间件都有什么优点和缺点?

这个其实就是看看你了解哪些分库分表的中间件，各个中间件的优缺点是啥？然后你用过哪些分库分表的中间件。

比较常见的包括：

- cobar
- TDDL
- atlas
- sharding-jdbc
- mycat

cobar

阿里 b2b 团队开发和开源的，属于 proxy 层方案。早些年还可以用，但是最近几年都没更新了，基本没啥人用，差不多算是被抛弃的状态吧。而且不支持读写分离、存储过程、跨库 join 和分页等操作。

TDDL

淘宝团队开发的，属于 client 层方案。支持基本的 crud 语法和读写分离，但不支持 join、多表查询等语法。目前使用的也不多，因为还依赖淘宝的 diamond 配置管理系统。

atlas

360 开源的，属于 proxy 层方案，以前是有一些公司在用的，但是确实有一个很大的问题就是社区最新的维护都在 5 年前了。所以，现在用的公司基本也很少了。

sharding-jdbc

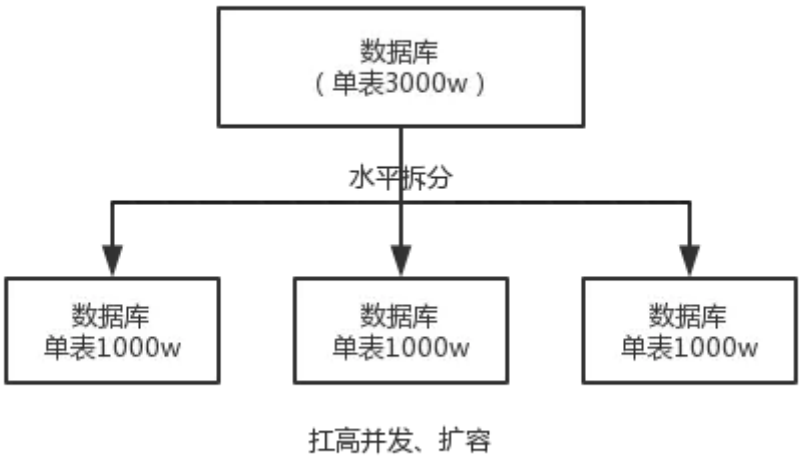
当当开源的，属于 client 层方案。确实之前用的还比较多一些，因为 SQL 语法支持也比较多，没有太多限制，而且目前推出到了 2.0 版本，支持分库分表、读写分离、分布式 id 生成、柔性事务（最大努力送达型事务、TCC 事务）。而且确实之前使用的公司会比较多一些（这个在官网有登记使用的公司，可以看到从 2017 年一直到现在，是有不少公司在用的），目前社区也还一直在开发和维护，还算是比较活跃，个人认为算是一个现在也**可以选择的方案**。

mycat

基于 cobar 改造的，属于 proxy 层方案，支持的功能非常完善，而且目前应该是非常火的而且不断流行的数据库中间件，社区很活跃，也有一些公司开始在用了。但是确实相比于 sharding jdbc 来说，年轻一些，经历的锤炼少一些。

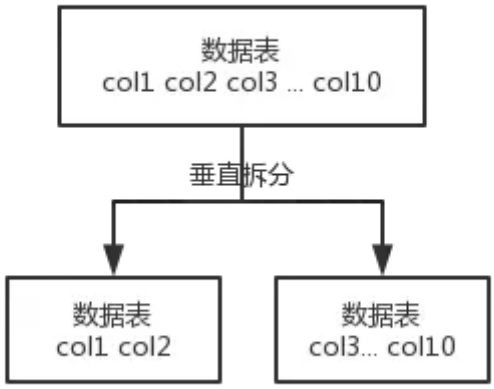
3. 如何对数据库如何进行垂直拆分或水平拆分的?

水平拆分的意思，就是把一个表的数据给弄到多个库的多个表里去，但是每个库的表结构都一样，只不过每个库表放的数据是不同的，所有库表的数据加起来就是全部数据。水平拆分的意义，就是将数据均匀放更多的库里，然后用多个库来抗更高的并发，还有就是用多个库的存储容量来进行扩容。



扛高并发、扩容

垂直拆分的意思，就是**把一个有很多字段的表给拆分成多个表，或者是多个库上去。**每个库表的结构都不一样，每个库表都包含部分字段。一般来说，会**将较少的访问频率很高的字段放到一个表里去**，然后**将较多的访问频率很低的字段放到另外一个表里去。**因为数据库是有缓存的，你访问频率高的行字段越少，就可以在缓存里缓存更多的行，性能就越好。这个一般在表层面做的较多一些。



两种**分库分表的方式**：

- 一种是按照 range 来分，就是每个库一段连续的数据，这个一般是按比如**时间范围**来的，但是这种一般较少用，因为很容易产生热点问题，大量的流量都打在最新的数据上了。
- 或者是按照某个字段hash一下均匀分散，这个较为常用。

range 来分，好处在于说，扩容的时候很简单，因为你只要预备好，给每个月都准备一个库就可以了，到了一个新的月份的时候，自然而然，就会写新的库了；缺点，但是大部分的请求，都是访问最新的数据。实际生产用 range，要看场景。

hash 分发，好处在于说，可以平均分配每个库的数据量和请求压力；坏处在于说扩容起来比较麻烦，会有一个数据迁移的过程，之前的数据需要重新计算 hash 值重新分配到不同的库或表

读写分离、主从同步（复制）

1. 什么是MySQL主从同步?

主从同步使得数据可以从一个数据库服务器复制到其他服务器上，在复制数据时，一个服务器充当主服务器（master），其余的服务器充当从服务器（slave）。

因为复制是异步进行的，所以从服务器不需要一直连接着主服务器，从服务器甚至可以通过拨号断断续续地连接主服务器。通过配置文件，可以指定复制所有的数据库，某个数据库，甚至是某个数据库上的某个表。

2. MySQL主从同步的目的？为什么要做主从同步?

1. 通过增加从服务器来提高数据库的性能，在主服务器上执行写入和更新，在从服务器上向外提供读功能，可以动态地调整从服务器的数量，从而调整整个数据库的性能。
2. 提高数据安全-因为数据已复制到从服务器，从服务器可以终止复制进程，所以，可以在从服务器上备份而不破坏主服务器相应数据
3. 在主服务器上生成实时数据，而在从服务器上分析这些数据，从而提高主服务器的性能
4. 数据备份。一般我们都会做数据备份，可能是写定时任务，一些特殊行业可能还需要手动备份，有些行业要求备份和原数据不能在同一个地方，所以主从就能很好的解决这个问题，不仅备份及时，而且还可以多地备份，保证数据的安全

3. 如何实现MySQL的读写分离?

其实很简单，就是基于主从复制架构，简单来说，就搞一个主库，挂多个从库，然后我们就单单只是写主库，然后主库会自动把数据给同步到从库上去。

4. MySQL主从复制流程和原理?

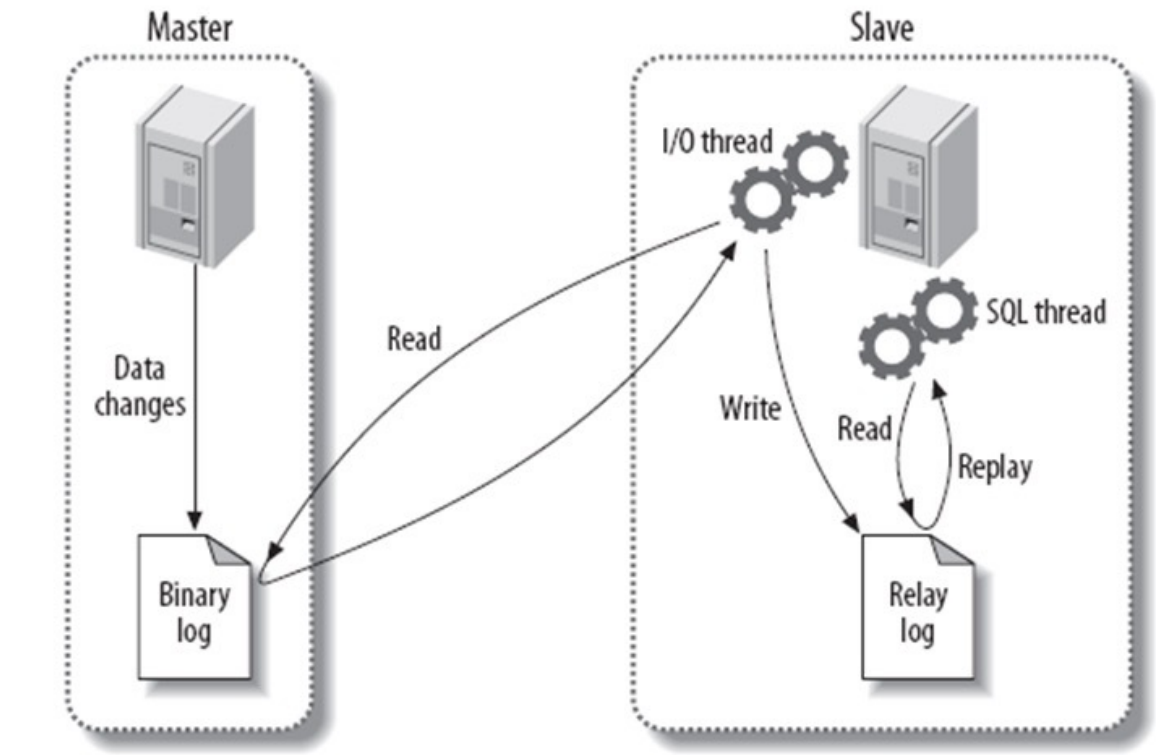
基本原理流程，是3个线程以及之间的关联

主：binlog线程——记录下所有改变了数据库数据的语句，放进master上的binlog中；

从：io线程——在使用start slave 之后，负责从master上拉取 binlog 内容，放进自己的relay log中；

从：sql执行线程——执行relay log中的语句；

复制过程如下：



Binary log：主数据库的二进制日志

Relay log：从服务器的中继日志

第一步：master在每个事务更新数据完成之前，将该操作记录串行地写入到binlog文件中。

第二步：salve开启一个I/O Thread，该线程在master打开一个普通连接，主要工作是binlog dump process。如果读取的进度已经跟上了master，就进入睡眠状态并等待master产生新的事件。I/O线程最终的目的是将这些事件写入到中继日志中。

第三步：SQL Thread会读取中继日志，并顺序执行该日志中的SQL事件，从而与主数据库中的数据保持一致。

5. MySQL主从同步延时问题如何解决?

MySQL 实际上在有两个同步机制，一个是半同步复制，用来 解决主库数据丢失问题；一个是并行复制，用来 解决主从同步延时问题。

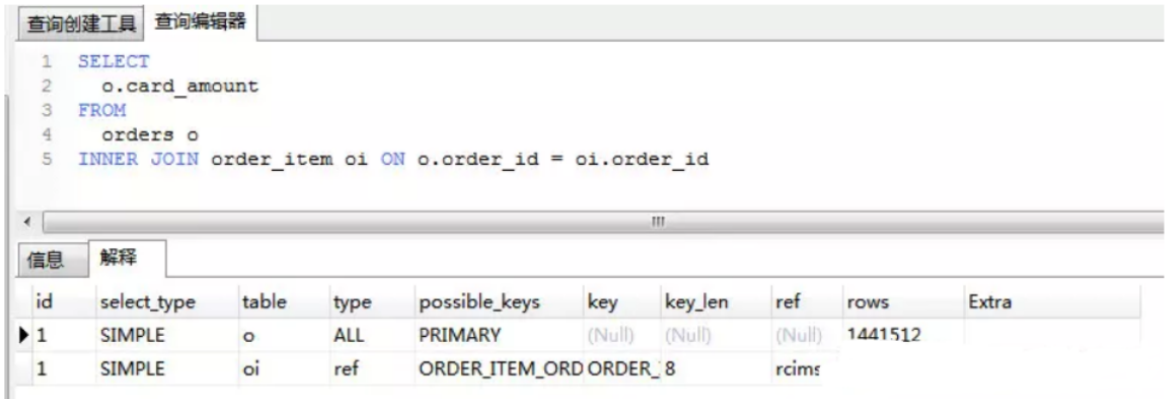
- 半同步复制（ semi-sync 复制），指的就是主库写入 binlog 日志之后，就会将强制此时立即将数据同步到从库，从库将日志写入自己本地的 relay log 之后，接着会返回一个 ack 给主库，主库接收到至少一个从库的 ack 之后才会认为写操作完成了。
- 并行复制，指的是从库开启多个线程，并行读取 relay log 中不同库的日志，然后并行重放不同库的日志，这是库级别的并行。

MySQL优化

1. 如何定位及优化SQL语句的性能问题?

对于低性能的SQL语句的定位，最重要也是最有效的方法就是使用执行计划，MySQL提供了explain命令来查看语句的执行计划。 我们知道，不管是哪种数据库，或者是哪种数据库引擎，在对一条SQL语句进行执行的过程中都会做很多相关的优化，对于查询语句，最重要的优化方式就是使用索引。

而执行计划，就是显示数据库引擎对于SQL语句的执行的详细情况，其中包含了是否使用索引，使用什么索引，使用的索引的相关信息等。



2. 大表数据查询，怎么优化

- 优化shema、sql语句+索引;
- 第二加缓存，memcached, redis;
- 主从复制，读写分离;
- 垂直拆分，根据你模块的耦合度，将一个大的系统分为多个小的系统，也就是分布式系统;
- 水平切分，针对数据量大的表，这一步最麻烦，最能考验技术水平，要选择一个合理的sharding key, 为了有好的查询效率，表结构也要改动，做一定的冗余，应用也要改，sql中尽量带sharding key，将数据定位到限定的表上去查，而不是扫描全部的表;

3. 超大分页怎么处理?

数据库层面,这也是我们主要集中关注的(虽然收效没那么大),类似于`select * from table where age > 20 limit 1000000`,10 这种查询其实也是有可以优化的余地的. 这条语句需要 load1000000 数据然后基本上全部丢弃,只取 10 条当然比较慢. 当时我们可以修改为`select * from table where id in (select id from table where age > 20 limit 1000000,10)`.这样虽然也 load 了一百万的数据,但是由于索引覆盖,要查询的所有字段都在索引中,所以速度会很快。

解决超大分页,其实主要是靠缓存,可预测性的提前查到内容,缓存至redis等k-V数据库中,直接返回即可.

在阿里巴巴《Java开发手册》中,对超大分页的解决办法是类似于上面提到的第一种.

> 【推荐】利用延迟关联或者子查询优化超多分页场景。
>
> 说明：MySQL并不是跳过offset行，而是取offset+N行，然后返回放弃前offset行，返回N行，那当offset特别大的时候，效率就非常的低下，要么控制返回的总页数，要么对超过特定阈值的页数进行SQL改写。
>
> 正例：先快速定位需要获取的id段，然后再关联：
>
> SELECT a.* FROM 表1 a, (select id from 表1 where 条件 LIMIT 100000,20) b where a.id=b.id

4. 统计过慢查询吗？对慢查询都怎么优化过?

在业务系统中，除了使用主键进行的查询，其他的我都会在测试库上测试其耗时，慢查询的统计主要由运维在做，会定期将业务中的慢查询反馈给我们。

慢查询的优化首先要搞明白慢的原因是什么？ 是查询条件没有命中索引？是load了不需要的数据列？还是数据量太大？

所以优化也是针对这三个方向来的，

- 首先分析语句，看看是否load了额外的数据，可能是查询了多余的行并且抛弃掉了，可能是加载了许多结果中并不需要的列，对语句进行分析以及重写。
- 分析语句的执行计划，然后获得其使用索引的情况，之后修改语句或者修改索引，使得语句可以尽可能的命中索引。
- 如果对语句的优化已经无法进行，可以考虑表中的数据量是否太大，如果是的话可以进行横向或者纵向的分表。

5. 如何优化查询过程中的数据访问

- 访问数据太多导致查询性能下降
- 确定应用程序是否在检索大量超过需要的数据，可能是太多行或列
- 确认MySQL服务器是否在分析大量不必要的数据行
- 查询不需要的数据。解决办法：使用limit解决
- 多表关联返回全部列。解决办法：指定列名
- 总是返回全部列。解决办法：避免使用SELECT *
- 重复查询相同的数据。解决办法：可以缓存数据，下次直接读取缓存
- 是否在扫描额外的记录。解决办法：
  使用explain进行分析，如果发现查询需要扫描大量的数据，但只返回少数的行，可以通过如下技巧去优化：
  使用索引覆盖扫描，把所有的列都放到索引中，这样存储引擎不需要回表获取对应行就可以返回结果。
- 改变数据库和表的结构，修改数据表范式
- 重写SQL语句，让优化器可以以更优的方式执行查询。

6. 如何优化关联查询

- 确定ON或者USING子句中是否有索引。
- 确保GROUP BY和ORDER BY只有一个表中的列，这样MySQL才有可能使用索引。

7. 数据库结构优化

一个好的数据库设计方案对于数据库的性能往往会起到事半功倍的效果。

需要考虑数据冗余、查询和更新的速度、字段的数据类型是否合理等多方面的内容。

1. **将字段很多的表分解成多个表**

对于字段较多的表，如果有些字段的使用频率很低，可以将这些字段分离出来形成新表。

因为当一个表的数据量很大时，会由于使用频率低的字段的存在而变慢。

2. **增加中间表**

对于需要经常联合查询的表，可以建立中间表以提高查询效率。

通过建立中间表，将需要通过联合查询的数据插入到中间表中，然后将原来的联合查询改为对中间表的查询。

3. **增加冗余字段**

设计数据表时应尽量遵循范式理论的规约，尽可能的减少冗余字段，让数据库设计看起来精致、优雅。但是，合理的加入冗余字段可以提高查询速度。

表的规范化程度越高，表和表之间的关系越多，需要连接查询的情况也就越多，性能也就越差。

注意：

冗余字段的值在一个表中修改了，就要想办法在其他表中更新，否则就会导致数据不一致的问题。

8. MySQL数据库cpu飙升到500%的话他怎么处理?

当 cpu 飙升到 500%时，先用操作系统命令 top 命令观察是不是 MySQLd 占用导致的，如果不是，找出占用高的进程，并进行相关处理。

如果是 MySQLd 造成的， show processlist，看看里面跑的 session 情况，是不是有消耗资源的 sql 在运行。找出消耗高的 sql，看看执行计划是否准确， index 是否缺失，或者实在是数据量太大造成。

一般来说，肯定要 kill 掉这些线程(同时观察 cpu 使用率是否下降)，等进行相应的调整(比如说加索引、改 sql、改内存参数)之后，再重新跑这些 SQL。

也有可能是每个 sql 消耗资源并不多，但是突然之间，有大量的 session 连进来导致 cpu 飙升，这种情况就需要跟应用一起来分析为何连接数会激增，再做出相应的调整，比如说限制连接数等。

9. 大表怎么优化?

类似的问题：某个表有近千万数据，CRUD比较慢，如何优化？分库分表了是怎么做的？分表分库了有什么问题？有用到中间件么？他们的原理知道么？

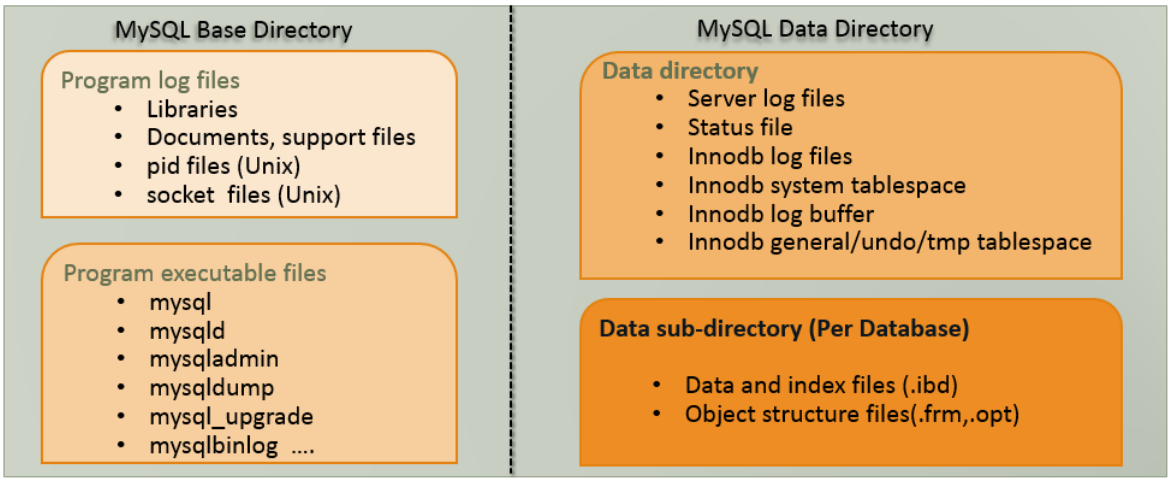
当MySQL单表记录数过大时，数据库的CRUD性能会明显下降，一些常见的优化措施如下：

- 限定数据的范围： 务必禁止不带任何限制数据范围条件的查询语句。比如：我们当用户在查询订单历史的时候，我们可以控制在一个月的范围内；
- 读/写分离： 经典的数据库拆分方案，主库负责写，从库负责读；
- 缓存： 使用MySQL的缓存，另外对重量级、更新少的数据可以考虑；
- 通过分库分表的方式进行优化，主要有垂直分表和水平分表。

这个博客记录了MySQL 5.7的物理和逻辑架构，还有其组件。在这个帖子中，我会尝试用图去说明SQL语句的执行流程和数据处理流程。MySQL的架构具备灵活性，因为它把不同的存储引擎作为插件。因此，MySQL的架构和行为也会随着存储引擎的改变而改变。

我们重点讨论InnoDB,因为它是MySQL的默认存储引擎。

MySQL物理架构



配置文件

- auto.cnf : 包含 server_uuid
- my.cnf : MySQL配置文件

形形色色的其他文件

```
-basedir=dir_name  //MySQL安装目录路径
-datadir=dir_name   //数据目录的路径，数据目录存储数据，状态，日志等
-pid-file=file_name  //MySQL服务器写ProcessID的文件路径
-socket=file_name, -S file_name  //在Unix系统上，使用的Unix套接字文件的名字，用于通过管道与本地服务器建立连接
-log-error=file_name        //记录错误和启动信息的日志文件名
```

MySQL逻辑架构

Client :

提供连接MySQL服务器功能的常用工具集

Server :

MySQL实例，真正提供数据存储和数据处理功能的MySQL服务器进程

mysqld:

MySQL服务器守护程序，在后台运行。它管理着客户端请求。mysqld是一个多线程的进程，允许多个会话连接，端口监听连接，管理MySQL实例

MySQL memory allocation:

MySQL的要求的内存空间是动态的，比如 innodb_buffer_pool_size (from 5.7.5), key_buffer_size。每个会话都有独一无二的执行计划，我们只能共享同一会话域内的数据集。

SESSION

为每个客户端连接分配一个会话，动态分配和回收。用于查询处理，每个会话同时具备一个缓冲区。每个会话是作为一个线程执行的

Parser

检测SQL语句语法，为每条SQL语句生成SQL_ID，用户认证也发生在这个阶段

Optimizer

创造一个有效率的执行计划（根据具体的存储引擎）。它将会重写查询语句。比如：InnoDB有共享缓冲区，所以，优化器会首先从预先缓存的数据中提取。使用 table statistics optimizer将会为SQL查询生成一个执行计划。用户权限检查也发生在这个阶段。

Metadata cache

缓存对象元信息和统计信息

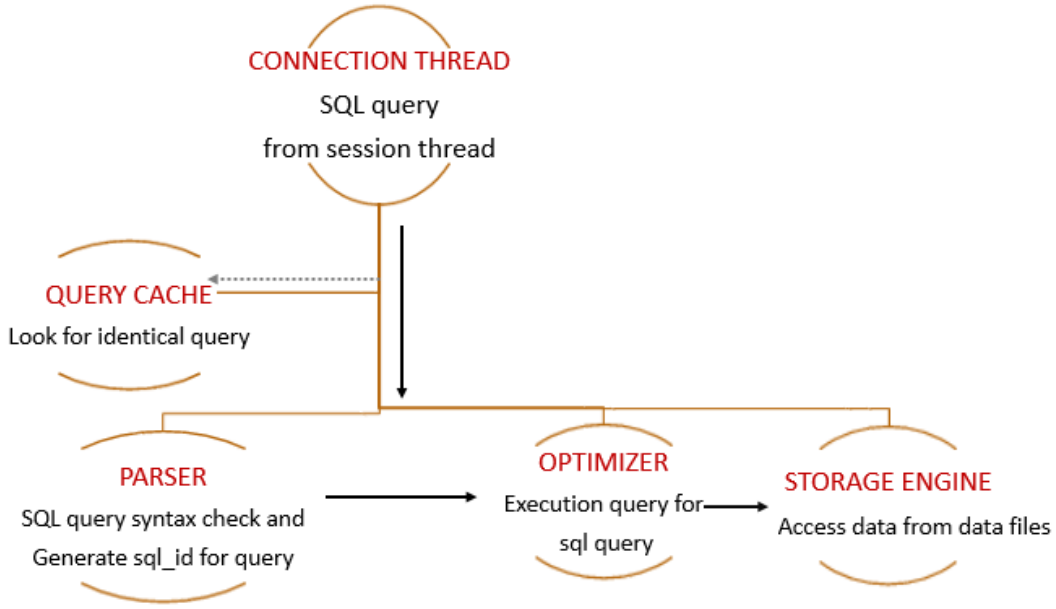
Query cache

共享在内存中的完全一样的查询语句。如果完全相同的查询在缓存命中，MySQL服务器会直接从缓存中去检索结果。缓存是会话间共享的，所以为一个客户生成的结果集也能为另一个客户所用。查询缓存基于SQL_ID。将SELECT语句写入视图就是查询缓存最好的例子。

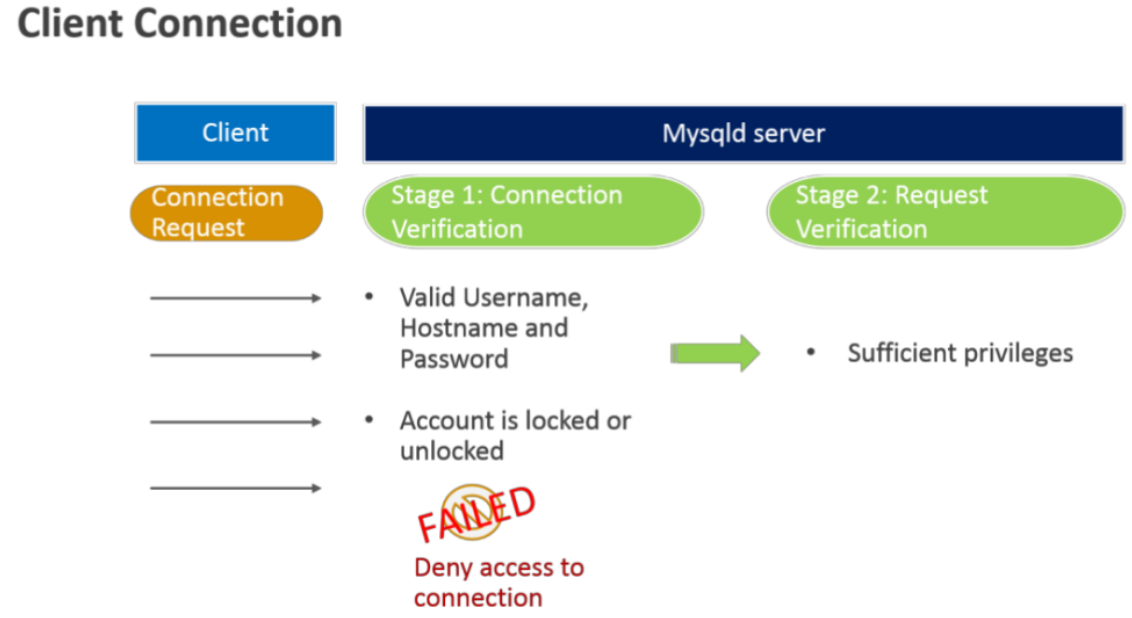
key cache

缓存表索引。MySQL keys是索引。如果索引数据量小，它将缓存索引结构和叶子节点（存储索引数据）。如果索引很大，它只会缓存索引结构，通常供MyISAM存储引擎使用

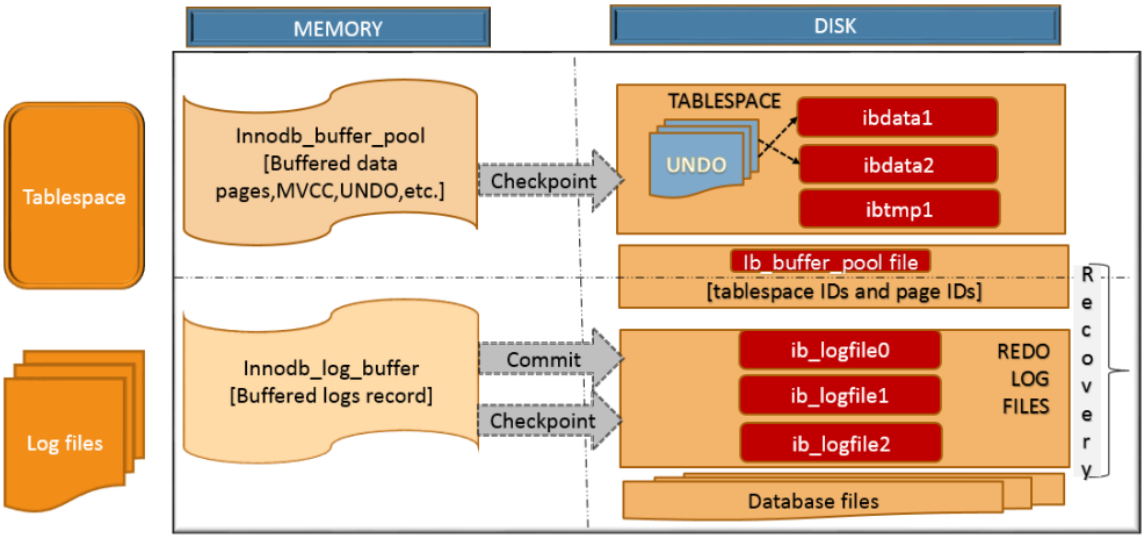
SQL执行



MySQL连接



InnoDB存储引擎架构



TABLESPACE

InnoDB存储空间被切分成tablespace,tablespace是一个与多个数据文件相关联的逻辑结构。

Pages

InnoDB最小的数据存储单元被也称作块。默认的页框是16KB,一个页包含多行。
可用页大小: 4kb,8kb,16kb,32kb,64kb
配置变量名 : innodb_page_size，在初始化mysqld时配置

Extents

一组页组成一个区，InnoDB为了更好的I/O吞吐率，每次读写都是按照区为单位。
一组16KB的页，一个区可以1MB，双写缓冲区（Doublewrite buffer ）每次分配/读/写都是以区为单位。

Segments

4个区构成一个Segments

InnoDB存储引擎

- ACID事务支持
- 行锁模式
- 事务REDO&UNDO支持
- 多数据文件
- 逻辑对象结构（InnoDB数据和日志缓冲区）
- InnoDB数据是百分百的具备逻辑结构，数据物理存储。
- InnoDB读取物理数据，创建逻辑结构[Blocks and Rows]
- 逻辑存储称为TABLESPACE

InnoDB 内存中组件

- InnoDB buffer pool
  InnoDB存储引擎的核心缓冲区。在这个缓冲区之中，加载表和索引数据
- InnoDB缓存表数据和索引数据的主要区域
  占据80%以上的物理内存，在专用数据库服务器中
- 所有会话的共享缓冲区
- InnoDB使用LRU页面置换算法

Change buffer

```
In a memory change buffer is a part of InnoDB buffer pool and on disk,
it is part of system tablespace, so even after database restart index
changes remain buffered.Change buffer is a special data structure that
caches changes to secondary index pages when affected pages not in the
buffer pool.
memory change buffer是InnoDB buffer pool的一部分，在磁盘上，也是系统tablespace的一部分。送印即使数据库重启…毛意思啊！无力…保留原文
```

Redo log buffer

redo logs缓冲区，保存写到redo log(重放日志)的数据。周期性的将缓冲区内的数据写入redo日志中。将内存中的数据写入磁盘的行为由innodb_log_at_trx_commit 和 innodb_log_at_timeout 调节。较大的redo日志缓冲区允许大型事务在事务提交前不进行写磁盘操作。
变量：innodb_log_buffer_size (default 16M)

在磁盘上的组件

系统表空间（tablespace）

除了存储表数据之外，InnoDB也支持查找表元信息，存储和检索MVCC信息以兑现服从ACID和事务隔离性等原则。它包含几种类型的InnoDB对象信息。
其包含的文件：

1. Table Data Pages
2. Table Index Pages
3. Data Dictionary
4. MVCC Control Data
5. Undo Space

6. Rollback Segments
7. Double Write Buffer (Pages Written in the Background to avoid OS caching) Insert Buffer (Changes to Secondary Indexes)

```
变量：
innodb_data_file_path = /ibdata/ibdata1:10M:autoextend
1
激活
innodb_file_per_table选项，你可以将每个新创建的表存储到不同的tablespace中。这种做法的优点是减少磁盘上数据文件中的碎片
```

通用tablespace

Shared tablespace to store multiple table data. Introduce in MySQL 5.7.6. A user has to create this using CREATE TABLESPACE syntax. TABLESPACE option can be used with CREATE TABLE to create a table and ALTER TABLE to move a table in general table.
共享的tablespace存储多个表信息，在MySQL 5.7.6时引入。用户只能使用CREATE TABLESPACE创建一个这样的表空间。 TABLESPACE选项可以在使用CREATE TABLE命令创建一个表然后 ALTER TABLE 将表移入通用空间时发挥作用。

```
- Memory advantage over innodb_file_per_table storage method.
- Support both Antelope and Barracuda file formats.
- Supports all row formats and associated features.
- Possible to create outside data directory.
```

InnoDB数据字典

在系统tablespace中的存储区域，由系统内部表（供mysql服务器使用的表）和对象元数据（表，索引，列信息）组成

双写缓冲区（Double write buffer）

系统tablespace的存储区域，InnoDB在写入物理文件之前先将页从InnoDB buffer pool写入此空间。mysqld进程突然崩溃会导致部分写问题。InnoDB可以从这个区域拿到一个备份。 Variable: inndb_doublewrite (default enable)

REDO logs

用于灾难恢复。mysqld启动的时候，InnoDB会尝试执行自动恢复，将不完整的事务更改矫正。还未完成更新数据文件的事务会在mysqld启动时会根据此日志记录中的信息被重放。它使用 LSN(Log Sequence Number)值来重放信息，因为mySQL会为每个事务赋予一个ID。因为大量数据更改不可能及时写道磁盘，所以得先记录到redo日志，然后再写入磁盘。
再redo日志，所有更改都会带有 row_id, 旧的列值，新的列值, session_id 和时间。

```
Innodb_log_file_in_group= [# of redo file groups]
Innodb_log_file_size= [每个日志文件大小]
```

UNDO日志和UNDO表空间

UNDO tablespace包含一个或多个undo日志文件。UNDO通过为事务（MVCC）保存被更改还未提交的值保持读一致性。未提交值从这个存储区域读取。UNDO日志也被叫做回滚数据段。
默认地，UNDO日志是系统表空间的一部分。但MySQL允许UNDO日志置于一个单独的表空间中[Introduce in MySQL 5.6]。这需要在初始化mysqld之前进行更改才起作用。

当我们配置单独UNDO表空间时，系统表空间的UNDO日志就被抑制了，但是一旦配置成单独的，我们只能删除UNDO日志的一部分，比如过期日志，而不能删除它。

```
- Variables : innodb_undo_tablespace : # of undo tablespaces, default
0 innodb_undo_directory:
Location for undo tablespace,default is, data_dir with 10MB size.
innodb_undo_logs :
# of undo logs, default , and max value is '128'
```

临时表空间

为临时表和相关对象提供存储功能，存储包括临时表未提交的数据。在MySQL 5.7.2引入，用于对临时表修改的回滚。 ibtmp1每次系统启动被重新创建，避免REDO日志对临时表的I/O操作。

```
innodb_temp_data_file_path = ibtmp1:12M:autoextend (default)
```

And All SET !!

存储引擎

```
Storage engine:
MySQL component that manages physical data (file management) and locations.
Storage engine responsible for SQL statement execution and fetching data from
data files. Use as a plugin and can load/unload from running MySQL server.Few of
them as following,
InnoDB :
Fully transactional ACID.
Offers REDO and UNDO for transactions.
Data storage in tablespace:
Multiple data files
Logical object structure using InnoDB data and log buffer
Row-level locking.
NDB (For MySQL Cluster):
Fully Transactional and ACID Storage engine.
Distribution execution of data and using multiple mysqld.
NDB use logical data with own buffer for each NDB engine.
Offers REDO and UNDO for transactions.
Row-level locking.
MyISAM:
Non-transactional storage engine
Speed for read
Data storage in files and use key, metadata and query cache
- FRM for table structure
- MYI for table index
- MYD for table data
Table-level locking.
MEMORY:
Non-transactional storage engine
All data stored in memory other than table metadata and structure.
Table-level locking.
ARCHIVE:
Non-transactional storage engine,
Store large amounts of compressed and unindexed data.
Allow INSERT, REPLACE, and SELECT, but not DELETE or UPDATE sql operations.
Table-level locking.
```

```
CSV:
Stores data in flat files using comma-separated values format.
Table structure need be created within MySQL server (.frm)
```

原文作者： lalit

参考

MySQL

https://blog.csdn.net/ThinkWon/article/details/104778621

https://haicoder.net/note/mysql-interview/mysql-interview-mysql-binlog.html

https://www.modb.pro/db/40241

https://www.jianshu.com/p/05da0fc0950e

https://blog.csdn.net/ThinkWon/article/details/104778621