# 01 三大范式？

第一范式：字段具有原子性，不可再分（字段单一职责）

第二范式：满足第一范式，每行应该被唯一区分，加一列存放每行的唯一标识符，称为主键（都要依赖主键）

第三范式：满足一二范式，且一个表不能包含其他表已存在的非主键信息（不间接依赖-不存在其他表的非主键信息）

**范式优点与缺点：**

优点：范式化，重复冗余数据少，更新快，修改少，查询时更少的distinct

缺点：因为一个表不存在冗余重复数据，**查询可能造成很多关联**，效率变低，可能使一些**索引策略无效**，范式化将列存在不同表中，这些列若在同一个表中可以是一个索引。

# 02InnoDB与MyISAM的区别？

2.1、MyISAM与InnoDB区别

* InnoDB聚簇索引，MyISAM非聚簇索引
* InnoDB数据与索引一起保存.ibd，MyISAM表结构.frm 索引.myi 数据.myd
* InnoDB支持事务、外键、行锁表锁，MyISAM不支持事务、外键、只支持表锁
* select count(\*)
* MyISAM查询更优，InnoDB更新更优
* 都是B+tree索引
* MyISAM支持全文索引，InnoDB5.6后支持

2.2、MyISAM

* 不支持事务，但是每次查询都是原子的
* 支持表级锁，每次操作对整个表加锁
* 存储表的总行数
* 一个MyISAM表有三个文件：表结构.frm 索引.myi 数据 .myd
* 采用非聚集索引，索引文件的数据域存储指向数据文件的指针。辅索引与主索引基本一致，但是辅索引不用保证唯一性。

2.3、Innodb

* 支持ACID事务，支持四种隔离级别
* 支持行级锁及外键约束，因此支持写并发
* 不存储总行
* 主键索引采用聚集索引(索引的数据域存储数据文件本身)，辅索引的数据域存储主键的值;因此从辅索引查找数据，需要先通过辅索引找到主键值，再访问辅索引；最好使用自增主键，防止插入数据时，为维持B+树结构，文件的大调整。

2.4、使用场景

大多数时候我们使用的都是 InnoDB 存储引擎，在某些读密集的情况下，使用 MyISAM 也是合适的。不过，前提是你的项目不介意 MyISAM 不支持事务、崩溃恢复等缺点（可是~我们一般都会介意啊！）。

* **MyISAM适合读多更新少的**：MyISAM索引跟数据分开放，因此有读取更快的说法。

* **InnoDB适合插入更新频繁的**：索引与数据一起放，建立索引更复杂，使用行锁，更新频繁效率更高
* 需要事务，高并发场景用Innodb：Innodb支持事务，采用行锁
* MyISAM查询比InnoDB快，更新InnoDB快

场景：MyISAM查询更优，InnoDB更新更优

**MyISAM适合读多，更新少的场景**。MyISAM使用非聚簇索引，数据和索引分开存的，而InnoDB数据和索引存一起的，数据量大时，一个内存页大小固定，读进内存的数据MyISAM就多一点(数据量小看不出差距，数据量大时差距就明显）。

因为MyISAM只把索引指针读进内存，可以存更多，查询速度也就更快，而且InnoDB还需要维护其他数据，比如其他隐藏字段 row\_id、tx\_id等

# 03自增主键理解？

**自增主键：**

InnoDB引擎的自增值，其实是**保存在了内存**里，并且到了MySQL 8.0版本后，才有了“自增值持久化”的能力。

也就是才实现了“如果发生重启，表的自增值可以恢复为MySQL重启前的值”，具体情况是：（查看表结构，会看到自增主键=多少）

* 在MySQL 5.7及之前的版本，**自增值保存在内存里**，并没有持久化。每次重启后，第一次打开表的时候，都会去找**自增值的最大值**max(id)，然后将max(id)+1作为这个表当前的自增值。
* **举例来说**：如果一个表当前数据行里最大的id是10，AUTO\_INCREMENT=11。这时候，我们删除id=10的行，AUTO\_INCREMENT还是11。但如果马上重启实例，重启后这个表的AUTO\_INCREMENT就会变成10。也就是说，MySQL重启可能会修改一个表的AUTO\_INCREMENT的值。
* 在MySQL 8.0版本，将自增值的变更记录在了redo log中，重启的时候依靠redo log恢复重启之前的值。

**自增值修改机制：**

1. 如果插入数据时id字段指定为0、null 或未指定值，那么就把这个表当前的 AUTO\_INCREMENT值填到自增字段；
2. 如果插入数据时id字段指定了具体的值，就直接使用语句里指定的值。

**自增值新增机制**：

1. 如果准备插入的值>=当前自增值，新的自增值就是“准备插入的值+1”；
2. 否则，自增值不变。

为什么自增主键不连续

* 在MySQL 5.7及之前的版本，自增值保存在内存里，并没有持久化
* 事务**回滚**（自增值不能回退，因为并发插入数据时，回退自增ID可能造成主键冲突）
* 唯一键冲突（由于表的自增值已变，但是主键发生冲突没插进去，下一次插入主键=现在变了的子增值+1，所以不连续）

**eg：**

假设，表t里面已经有了(1,1,1)这条记录，这时我再执行一条插入数据命令：

insert into t values(null, 1, 1); (自增id,唯一键c,普通字段d)

这个语句的执行流程就是：

1. 执行器调用InnoDB引擎接口写入一行，传入的这一行的值是(0,1,1);
2. InnoDB发现用户没有指定自增id的值，获取表t当前的自增值2；
3. 将传入的行的值改成(2,1,1)；
4. 将表的自增值改成3；
5. 继续执行插入数据操作，由于已经存在c=1的记录，所以报Duplicate key error，语句返回。

这个表的自增值改成3，是在真正执行插入数据的操作之前。这个语句真正执行的时候，因为碰到唯一键c冲突，所以id=2这一行并没有插入成功，但也没有将自增值再改回去。

所以，在这之后，再插入新的数据行时，拿到的自增id就是3。也就是说，出现了自增主键不连续的情况。

# 04Innodb为什么推介用自增ID

①主键页就会近乎于顺序的记录填满，提升了页面的最大填充率，不会有页的浪费。

②新插入的行一定会在原有的最大数据行下一行，mysql定位和寻址很快，不会为计算新行的位置而做出额外的消耗。

③减少了页分裂和碎片的产生

UUID：**大量的随机IO**+**页分裂导致移动大量的数据**+数据会有碎片。

总结：自增ID有序，会按顺序往最后插入，而UUID无序，随机生成，随机插入，会造成频繁页分裂，内存碎片化，大量随机IO

# 05什么是索引

* 排好序的数据结构，可以帮助快速查找数据
* 优缺点：索引可以提高查询速度，查询使用优化隐藏器提高性能，但是也会占据物理空间，降低增删改的速度，因为还要操作索引文件

# 06索引类型

覆盖索引+回表+索引下推+联合索引

* 普通索引：可以重复
* 唯一索引：唯一，可为空，表中只有一个主键索引，可多个唯一索引
* 主键索引
  + 唯一，不为空，叶子结点存出了行记录数据，主键索引也称聚簇索引，对应非主键索引的叶子结点存的主键的值（二级索引），用二级索引查需要回表操作（根据二级索引查到主键，再根据主键去主键索引查）
  + 一般推荐用自增主键，**保证空间利用率，减少页分裂**
* 全文索引
* 覆盖索引：索引字段覆盖了查询语句涉及的字段，直接通过索引文件就可以返回查询所需的数据，不必通过回表操作。
* 回表：通过索引找到主键，再根据主键id去主键索引查。
* 索引下推
  + 在根据索引查询过程中就根据查询条件过滤掉一些记录，减少最后的回表操作

假如执行select \* from stu where name=? and age=?   
没有索引下推先再存储引擎根据name筛选数据返回给server层，然后server层再根据age过滤  
有索引下推直接根据name和age在存储引擎层就筛选得到结果

# 07索引底层数据结构？

**B+树、hash**

hash底层是哈希表实现，等值查询，可以快速定位，一般情况效率很高，不稳定，当出现大量键重复哈希冲突，效率下降，不支持范围查询，无法用于排序分组，无法模糊查询，多列索引的最左前缀匹配原则，总要回表操作等。

# 08B树与B+树区别？为何用B+树？

B+树：非叶子结点不存data，只存key，查询更稳定，增大了广度(B+树出度更大，树高矮，节点小，磁盘IO次数少)；叶子结点下一级指针（范围查询）；索引冗余。

与红黑树相比：

更少查询次数：B+树出度更大，树高更低，查询次数更少

磁盘预读原理：为了减少IO操作，往往不严格按需读取，而是预读。B+树叶子**结点存储相临**，**读取会快一些**。

**存储更多索引结点：**B+树只在叶子结点储存数据，非叶子结点存索引，而一个结点就是磁盘一个内存页，内存页大小固定，那么相比B树这些可以·存更多的索引结点，出度更大，树高矮，查询次数少，磁盘IO少。

# 09索引设计原则（查询快，占用空间少）

* 出现在where子句或则连接子句中的列
* 基数小的表没必要
* 使用短索引，如果索引长字符串列，应该指定前缀长度
* 定义有外键的数据列一定索引
* 不要过度索引
* 更新频繁的不适合
* 区分度不高的不适合，如性别
* 尽量扩展索引，别新建索引，如(a)->(a,b)
* 字符串字段建立索引方法
  + 1、直接创建完整索引，这样可能比较占用空间；
  + 2、创建前缀索引，节省空间，但会增加查询扫描次数，并且不能使用覆盖索引；
  + 3、倒序存储，再创建前缀索引，用于绕过字符串本身前缀的区分度不够的问题；
  + 4、额外用一个字段进行索引，额外计算开销

总结：索引设计原则要求查询快，占用空间少；一般建在where条件，匹配度高的；要求基数大，区分度高，不要过大索引，尽量扩展，用联合索引，更新频繁不适合、使用短索引。

# 10索引失效场景？

* 以“%”开头的like语句，索引无效，后缀“%”不影响
* or语句前后没有同时使用索引
* 列类型是字符串，一定要在条件中将数据用引号引用，否则失效（隐式转换）
* 如果mysql估计使用全表扫描比索引快，则不用索引（键值少，重复数据多）
* 组合索引要遵守最左前缀原则——不使用第一列索引 失效
* 在索引字段上使用not，<>，！= （对它处理是全表扫描）
* 对索引字段进行计算操作，字段使用函数也会失效
* is null

# 11如何创建索引

* ALTER TABLE table\_name ADD INDEX index\_name (column\_list)；
* CREATE INDEX index\_name ON table\_name (column\_list)；
* 在create table时创建

# 12非聚簇索引一定会回表查询吗？

查询字段全部命中索引，覆盖索引，不走回表，直接从索引得到结果，不要查数据文件

总结：覆盖索引就不走回表

# 13联合索引的建立规则？

* 将**查询需求频繁**或者**字段选择性高**的列放在前面
* 索引的**复用**，可以少维护一些索引(a)->(a,b)
  + 如果既有联合查询，又有基于a、b各自的查询呢？：考虑的原则就是空间，将小的单独建索引

# 14最左匹配原则

从左往右匹配，直到遇到范围查询

建立联合索引（a,b,c）

索引是先根据a排序，a相同时b有序，a不同无序，以此类推。总之遇到范围查询就停。

（a,b）联合索引 [(2,4),(),()]  
 \|/ \|/  
 [(1,1),(1,2),(2,1)] [(2,4),(3,1),(3,2)]  
规律：a有顺序（1，1，2，2，2，3，3）b无顺序，a相同时b又有顺序，不同a之间b没有顺序，所以a=1,b>2走联合索引；a>1,b>2不走索引。  
  
select \* from table\_name where a = '1' and b = '2' and c = '3'  
//全值匹配查询，用到索引，与顺序无关，查询优化器，会自动优化查询顺序   
  
select \* from table\_name where a = '1'   
select \* from table\_name where a = '1' and b = '2'    
select \* from table\_name where a = '1' and b = '2' and c = '3'  
//匹配左边的列时，用到了索引  
  
select \* from table\_name where b = '2'   
select \* from table\_name where c = '3'  
select \* from table\_name where b = '1' and c = '3'  
//没有用到索引  
  
select \* from table\_name where a = '1' and c = '3'   
//a用到了索引，b、c没有到  
  
select \* from table\_name where a > 1 and a < 3 and b > 1;  
//只有a用到索引，在1<a<3的范围内b是无序的，不能用索引，找到1<a<3的记录后，只能根据条件 b > 1继续逐条过滤  
  
select \* from table\_name where a = 1 and b > 3;  
// a=1的情况下b是有序的，进行范围查找走的是联合索引 走 a b索引(a相同时b有序)

# 15前缀索引

尽量创建短索引，对长子字符串创索引可使用前缀索引，使用字段值前几个字符作为索引 index(filed(10))

# 16百万级数据如何删除

删除数据的速度和创建的索引数量是成正比的。先删索引，再删无用数据，再创建索引

# 17普通索引和唯一索引怎样选

* 查询比较
  + 查询会以页为单位将数据页加载进内存，不需要一条记录一条记录读取磁盘。然后唯一索引根据条件查询到记录时就返回结果，普通索引查到第一条记录往后遍历直到不满足条件，由于都在内存中，不需要磁盘读取那么大开销，带来的额外查询开销忽略不计，所以查询性能几乎一致
* 更新比较
  + 唯一索引由于更新时**要检查唯一性**，所以需要将数据页先加载进内存才能判断，此时直接操作内存，不需要操作change buffer
  + 补充：普通索引若数据再内存中直接内存中更新，否则会将更新操作先记录到channge buffer中，等下一次查询将数据读到内存中再进行change buffer里相关更新操作后将数据返回，这样一来，再**写多读少的情况下就减少了磁盘IO**，若写完就马上查询，就大可不必用change buffer，不但没提高多少效率还造成维护change buffer额外消耗
  + 将change buffer的操作对应到原始数据页的操作称为merge（可以查询来时读到内存再修改数据，后台线程也会merge，数据库正常关闭也会merge）
* 适合场景
  + 写多读少，选用普通索引更好，可以利用**change buffer**进行性能优化减少磁盘IO，将更新操作记录到change bufer，等查询来了将数据读到内存再进行修改.

**事务&隔离机制&日志&MVCC&锁**

# 18一条sql查询语句执行过程

**mysql分为server层与存储引擎层，server层包含连接器、分析器、优化器、执行器。**

接下来以一条sql查询语句执行过程介绍各个部分功能。客户端执行一条sql：

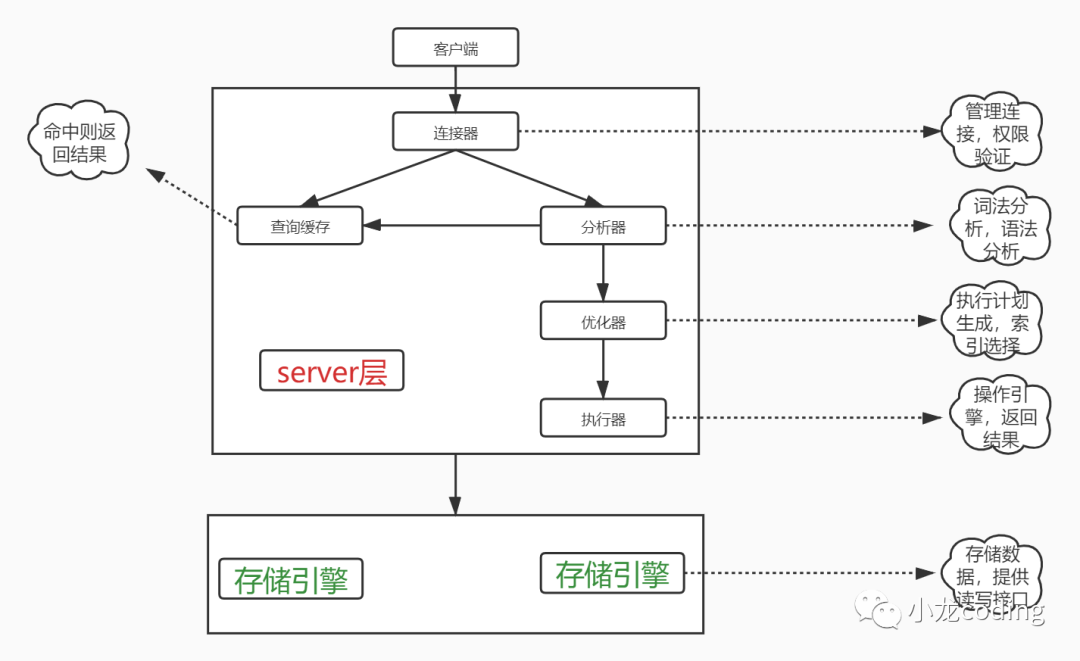
1、首先由连接器进行**身份验证，权限管理**

2、若开启了缓存，会**检查缓存**是否有该sql对应结果（缓存存储形式key-vlaue，key是执行的sql，value是对应的值）若开启缓存又有该sql的映射，将结果直接返回；

3、分析器进行**词法语法分析**

4、优化器会**生成执行计划**、选择索引等操作，选取最优执行方案

5、然后来到执行器，打开表调用存储引擎接口，逐行判断是否满足查询条件，满足放到结果集，最终返回给客户端；若用到索引，筛选行也会根据索引筛选。

                      图1、  一条sql语句执行流程

# 19两阶段提交（一条更新语句怎么执行？）

1、**引擎**先根据筛选条件**筛选**对应的行返回**给执行器**（若对应的行在内存直接返回，否则先去磁盘读取再返回）

2、**执行器执行相关更新操作**然后调用**引擎接口写回**更新后数据

3、引擎将新数据更新到内存，将更新操作记录到redolog，redolog处于prepare，告知执行器执行完，可提交事务

4、执行器生成该操作的binlog 并将binlog写入磁盘

5、执行器调用引擎事务提交接口，引擎把刚写入的redolog改为commit状态，更新完成。

# 20mysql的事务原理

事务：一系列操作组成，要么全部成功，要么全部失败

事务ACID特性

1. 原子性：一些列操作要么全部成功，要么全部失败
2. 隔离性：事务的结果只有提交了其他事务才可见
3. 一致性：数据库总时从一个一致状态变到另一个一致状态（事务修改前后的**数据总体保证一致 转账**）
4. 持久性：事务提交后，对数据修改永久的

事务的并发问题：

1. 脏读：读到未提交的数据
2. 不可重复读：一个事务下，两次读取数据不一致（侧重内容数据的修改）
3. 幻读：事务A 按照一定条件进行数据读取， 期间事务B 插入了相同搜索条件的新数据，事务A再次按照原先条件进行读取时，发现了事务B 新插入的数据 称为幻读（侧重新增或删除，插入数据读到多了一行）

隔离级别原理及解决问题分析：

1. 读未提交：原理：直接读取数据，不能解决任何并发问题
2. 读已提交：读操作不加锁，写操作加排他锁，解决了脏读。原理：利用MVCC实现，每一句语句执行前都会生成Read View（一致性视图）
3. 可重复读：MVCC实现，只有事务开始时会创建Read View，之后事务里的其他查询都用这个Read View。解决了脏读、不可重复读，快照读（普通查询，读取历史数据）使用MVCC解决了幻读，当前读（读取最新提交数据）通过间隙锁解决幻读（lock in share mode、for update、update、detete、insert），间隙锁在可重复读下才生效。（**默认隔离级别**）
4. 可串行化：原理：使用锁，读加共享锁，写加排他锁，串行执行

**总结：**读已提交和可重复读实现原理就是MVCC Read View不同的生成时机。可重复读只在事务开始时生成一个Read View，之后都用的这个；读已提交每次执行前都会生成Read View。

# 21ACID实现原理

原子性：undolog（记录事务开始前的老版本数据，可以保证原子操作，回滚，实现MVCC版本链）

隔离性：MVCC

持久性：redo log（记录事务开启后对数据的修改，可用于crash-safe）

211幻读问题详解

**幻读问题详解：**

1、创建tx实验表

DROP TABLE IF EXISTS `tx`;  
CREATE TABLE `tx` (  
 `age` int(5) DEFAULT NULL,  
 `name` varchar(5) DEFAULT NULL,  
 `id` int(5) NOT NULL,  
 PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;  
-- ----------------------------  
-- Records of tx  
-- ----------------------------  
INSERT INTO `tx` VALUES ('20', '张三', '1');  
INSERT INTO `tx` VALUES ('20', '李四', '2');

2、实验



结论（仔细理解，讲收获满满，本人认真总结的）：

1、发现RR隔离界别**若只快照读与当前读没有幻读问题**，快照读（普通查询，如select \* from table）读取旧的历史版本，用MVCC实现（MVCC原理下文分析），会在事务开始时生成一个Read View，之后都用这个Read View实现RR隔离级别。

当前读（select ... for update ，select ... lock in share mode ，update/insert/delete语句）读取最新数据版本，依靠间隙锁或则临键锁解决幻读，当你事务T1执行当前读，然后事务T2插入语句，事务T2会被阻塞住，插不进去。

2、当你事务T1中**先执行快照读，事务T2插入数据并提交，事务T1再执行当前读（比如以相同条件更新数据），会发现出现幻读**，更新到了新插入行的数据。

白话文解释：事务1先以某个条件比如age=20的查询得到2条数据，然后事务2插入新的数据age也为20然后提交事务，此时事务1更新age=20的数据，发现更新到了3行，把事务T2新插入的那行也更新了。

所以**幻读注重你插入新数据都修改改到了新插入的数据，而不可重复读是你修改了某个数据，两次查询得到不一致结果。**

**总结：（RR隔离界别并没有完全解决幻读）只使用快照都或则当前读不会幻读。若先快照读，然后当前读，期间按快照读相同条件插入数据，当前读就会发生幻读。**

# 22MVCC原理

多版本并发控制。

原理提炼总结：使用版本链+Read View

详解：

**版本链** 同一行数据可能有多个版本

innodb数据表每行数据记录会有几个隐藏字段，row\_id，事务ID，回滚指针。

1、Innodb采用主键索引（聚簇索引），会利用主键维护索引，若表没有主键，就用第一个非空唯一索引，若没有唯一索引，则用row\_id这个隐藏字段作为主键索引。

2、事务开启会向系统申请一个事务ID，严格递增，会向行记录插入最近操作它的那个事务的ID。

3、undolog会记录事务前老版本数据，然后行记录中回滚指针会指向老版本位置，如此形成一条版本链。因此可以利用undo log实现回滚，保证原子性，同时用于实现MVCC版本链。



**Read View**读已提交隔离级别下，会在每次查询都生成一个Read View，可重读读只在事务开始时生成一个Read View，以后每次查询都用这个Read View，以此实现不同隔离界别。

**Read View里面包含些什么？（一致性视图）**

**一个数组+up\_limit\_id(低水位)+low\_limit\_id(高水位)**（这里的up,low没写错，就是这么定义的）

数组里包含**事务启动时**当前活跃事务ID(未提交事务)，低水位就是活跃事务最小ID，高水位就是下一次将分配的事务ID，也就是目前最大事务ID+1。

**数据可见性规则是怎样实现的？**

数据版本的可见性规则，就是**基于数据的row、trx\_id**和这个**一致性视图（Read View）**的对比结果得到的。

 视图数组把所有的row trx\_id 分成了几种不同的情况

图4  数据版本可见性规则

**读取原理：**

某事务T要访问数据A，先获取该数据A中的事务id(获取最近操作它的事务的事务ID)，对比该事务T启动时刻生成的

**Read View:**

1、如果在readview的左边(比readview都小)，表示这个事务可以访问这数据（在左边意味着该事务已经提交)

2、如果在readview的右边(比readview都大），表示这个版本是由将来启动的事务生成的，是肯定不可见的；

3、如果当前事务在未提交事务集合中：

a、若 row trx\_id在数组中，表示这个版本是由还没提交的事务生成的，不可见；

b. 若 row trx\_id不在数组中，表示这个版本是已经提交了的事务生成的，可见。

不可以访问，获取roll\_pointer，通过版本链取上一版本

根据数据历史版本事务ID再重新与视图数组对比。

这样执行下来，虽然期间这一行数据被修改过，但是事务A不论在什么时候查询，看到这行数据的结果都是一致的，所以我们称之为一致性读。

# 23日志机制分析

前置知识，为了保证事务ACID中的一致性与原子性，mysql采用WAL，预写日志，先写日志，合适时再写磁盘。

innodb引擎级别有undo log与redo log，mysql server级别有bin log。

**undo log**

回滚日志

**作用：**undolog记录事务开始前老版本数据，用于实现回滚，保证原子性，实现MVCC，会将数据修改前的旧版本保存在undolog，然后行记录有个隐藏字段回滚指针指向老版本。

**redo log**

物理日志

**作用**：会记录事务开启后对数据做的修改，crash-safe

**特性**：**空间一定，写完后会循环写**，有两个指针write pos指向当前记录位置，checkpoint指向将擦除的位置，redolog相当于是个取货小车，货物太多时来不及一件一件入库太慢了这样。

就先将货物放入小车，等到货物不多或则小车满了或则店里空闲时再将小车货物送到库房。**用于crash-safe**，数据库异常断电等情况可用redo log恢复。

**写入流程**：先写redo log buffer，然后wite到文件系统的page cache，此时并没有持久化，然后fsync持久化到磁盘。

**写入策略：**根据innodb\_flush\_log\_at\_trx\_commit参数控制（我的记忆：innodb以事务的什么提交方式刷新日志）

0——>事务提交时只把redo log留在redo log buffer

1——>将redo log直接持久化到磁盘（所以有个双“1”配置，后面会讲）

2——>只是把redo log写到page cache

**bin log**

用于主备同步

**有3种格式：**

row：记录整行数据，更新记录更新前后的数据

缺点：记录每行数据，占空间

statement：记录整条sql语句

缺点：可能造成主从不一致

mysql> delete from t where a>=4 and b<=5 limit 1;

主库是索引a,那么删除a=4

备库是索引b,那么删除b=5

mixed：会判断statement格式下sql语句是否会造成主备不一致，不造成就statement格式，否则就row格式。

**写入机制**：

1、事务执行过程中将日志记录到binlog cache（系统为binlog分配了一块内存，每个线程一份）

2、事务提交时，执行器把binlog cache里的完整事务写入到binlog中，并清空binlog cache

* write：把日志写到文件系统的page cache，没有写磁盘，速度快
* fsync：将数据持久化到磁盘的操作，这时才占磁盘IOPS

根据sync\_binlog参数控制：

0——>只write，不fsync

1——>每次fsyncN

>1——>每次事务都write，等累积到N后才fsync。

可以将sync\_binlog设置大一点提高性能（可以提高IO性能，但是若发生异常，日志会丢失）

这里sync\_binlog和innodb\_flush\_log\_at\_trx\_commit配合设置双1模式

**两阶段提交：**

想要全面了解两阶段提交，我接下从这3个方面分析：

1、何为两阶段提交？

2、为什么要两阶段提交？

3、两阶段提交的过程是怎样的？

**何为两阶段提交？（2PC）**mysql中在server层级别有个binlog日志，归档日志，用于备份，主从同步复制，如果采用一主多从架构，主备切换，那就必须用到binlog进行主从同步。

此时事务提交就必须保证redolog与binlog的一致性，一般情况没有开启binlog日志，事务提交不会两阶段提交，若需要主从同步就必须开启binlog使用两阶段提交保证数据一致性。

**为什么要两阶段提交？**保证redolog与binlog一致性，保证事务在多个引擎的原子性。

**两阶段提交过程？**

**Prepare 阶段**：InnoDB 将回滚段undolog设置为 prepare 状态；将 redolog 写文件并刷盘；

（1、先写redolog，事务进入prepare状态）

**Commit 阶段**：Binlog 写入文件；binlog 刷盘；InnoDB commit；

（2、prepare成功，binlog写盘，然后事务进入commit状态，同时会在redolog记录commite标识，代表事务提交成功）

**redolog与binlog怎样联系起来的？**（XID）

* 崩溃恢复的时候，会按顺序扫描redo log，若redolog既有prepare又有commit，直接提交
* 如果碰到只有prepare、而没有commit的redo log，就拿着XID去binlog找对应的事务。

**怎样判断binlog是否完整？**

* statement格式的binlog，最后会有COMMIT
* row格式 末尾有XID event

**2pc不同时刻的崩溃恢复？**

* 1、redolog有commite标识，事务完整，直接提交事务
* 2、若redolog里面的事务只有完整的prepare，则判断对应事务的binlog是否存在并完整 （是-提交事务 | 否-回滚事务）

# 24Explain分析

type：表示MySQL在表中找到所需行的方式，或者叫访问类型

* **type=ALL，全表扫描，MySQL遍历全表来找到匹配行**
* **type=index，索引全扫描**
* **type=range，索引范围扫描**
* **type=eq\_ref，唯一索引**
* **type=NULL，MySQL不用访问表或者索引，直接就能够得到结果（性能最好）**

**possible\_keys**: 表示查询可能使用的索引

**key**: 实际使用的索引

**key\_len**: 使用索引字段的长度

**rows**: 扫描行的数量

**Extra**：

* using index：覆盖索引，不回表
* using where：回表查询
* using filesort：需要额外的排序，不能通过索引得到排序结果

24脏页？怎样刷新脏页？

内存数据页和磁盘数据页不一致。

**刷脏页情景：**

**redo log写满了**，停止所有更新操作，将checkpoint向前推进，推进那部分日志的脏页更新到磁盘。

**系统内存不够**，需要将一部分数据页淘汰，如果是干净页，直接淘汰就行了，脏页的话，需要全部同步到磁盘。

mysql自认为**空闲**时

mysql**正常关闭**之前

25MYSQL调优篇

# 25.1、一条sql执行很慢的原因？

一个 SQL 执行的很慢，我们要分两种情况讨论：

**1、大多数情况下很正常，偶尔很慢，则有如下原因**

(1)、数据库在**刷新脏页**，例如 redo log 写满了需要同步到磁盘。

(2)、执行的时候，**遇到锁**，如表锁、行锁。

(3)、**sql写的烂**了

**2、这条 SQL 语句一直执行的很慢，则有如下原因**

(1)、没有用上索引或则索引失效：例如该字段没有索引；由于对字段进行运算、函数操作导致无法用索引。

(2)、有索引可能会走全表扫描

怎样判断是否走全表扫描：

索引区分度（索引的值不同越多，区分度越高），称为基数，而数据量大时不可能全部扫描一遍得到基数，而是采样部分数据进行预测，那有可能预测错了，导致走全表扫描。

# 25.2、sql优化（定位低效率sql，慢查询怎样处理）

（1）数据库中设置SQL慢查询

方式一：修改配置文件 在 my.ini 增加几行: 主要是慢查询的定义时间（超过2秒就是慢查询），以及慢查询log日志记录（ slow\_query\_log）

[mysqlld]  
//定义查过多少秒的查询算是慢查询，我这里定义的是2秒  
long\_query\_time=2  
#5.8、5.1等版本配置如下选项  
log-slow-queries="mysql\_slow\_query.log"  
#5.5及以上版本配置如下选项  
slow-query-log=On  
slow\_query\_log\_file="mysql\_slow\_query. log"  
1/记录下没有使用索引的query  
log-query-not-using-indexestpspb16glos dndnorte/t

方式二：通过MySQL数据库开启慢查询:

mysql>set global slow\_query\_log=ON  
mysql>set global long\_query\_time = 3600;  
mysql>set global log\_querise\_not\_using\_indexes=ON;

（2）分析慢查询日志

可以通过以下命令定位低效率执行sql  
**show processlist**sql 可以用 **explain**分析执行计划

（3）优化

**索引**

1、尽量覆盖索引，5.6支持索引下推

2、组合索引符合最左匹配原则

3、避免索引失效

4、再写多读少的场景下，可以选择普通索引而不要唯一索引

更新时，普通索引可以使用change buffer进行优化，减少磁盘IO,将更新操作记录到change bufer，等查询来了将数据读到内存再进行修改.

5、索引建立原则（一般建在where和order by，基数要大，区分度要高，不要过度索引，外键建索引）

**sql语句**

1、分页查询优化

该方案适用于主键自增的表，可以把Limit查询转换成某个位置的查询。

select \* from tb\_sku where id>20000 limit 10;

2、优化insert语句

* 多条插入语句写成一条
* 在事务中插数据
* 数据有序插入（主键索引）

**数据库结构优化**

1、将字段多的表分解成多个表

有些字段使用频率高，有些低，数据量大时，会由于使用频率低的存在而变慢，可以考虑分开。

2、对于经常联合查询的表，可以考虑建立中间表

**优化器优化**

1、优化器使用MRR

**原理：MRR 【**Multi-Range Read**】将ID或键值读到buffer排序，通过把「随机磁盘读」，转化为「顺序磁盘读」，减少磁盘IO，从而提高了索引查询的性能。**

mysql >set optimizer\_switch='mrr=on';explain 查看 Extra多了一个MRRexplainselect\*from stu where age between 10 and 20;

对于 Myisam，在去磁盘获取完整数据之前，会先按照 rowid 排好序，再去顺序的读取磁盘。

对于 Innodb，则会按照聚簇索引键值排好序，再顺序的读取聚簇索引。

**磁盘预读：**请求一页的数据时，可以把后面几页的数据也一起返回，放到数据缓冲池中，这样如果下次刚好需要下一页的数据，就不再需要到磁盘读取（局部性原理）

**索引本身就是为了减少磁盘 IO，加快查询，而 MRR，则是把索引减少磁盘 IO 的作用，进一步放大**

https://zhuanlan.zhihu.com/p/148680235

**架构优化**

读/写分离（主库写，从库读）

**总结：**

1、先设置慢查询（my.ini或数据库命令）

2、分析慢查询日志

3、定位低效率sql（show processlist）

4、explain分析执行计划（是否索引失效，用到索引没，用了哪些）

5、优化（索引+sql语句+数据库结构优化+优化器优化+架构优化）

26主从同步

主从三条线程+binlog+relaylog(中继日志)

**原理：**

* binlog会在服务器启动生成，用于记录主库数据库变更记录，当binlog发生变更时，主结点的**log dump**线程会将其内容发给各个从结点，从结点的 IO线程接收binlog内容，并写入relay log(从节点上)，从结点的SQL线程读取relay log内容对数据库数据进行更新重放，保证主从一致性

**同步问题：**

* **全同步复制**：主库强制同步日志到从库，等**全部从库执行完**才返回客户端，性能差
* **半同步复制**：主库收到至少一个从库确认就认为操作成功，从库写入日志成功返回ack确认

# 26高可用架构

一主一备

**M-S结构**



主库A与备库B，客户端操作A，B把更新A的语句同步过来本地执行，数据就一致了，建议将备库设置为只读模式。

因为同步更新线程是超级权限不影响，而且设置为只读（1、可以标识哪个为备库2、当需要从备库查询时避免误操作）

**主备延迟**

1、解释：同一个事务，备库执行完时间与主库执行完时间之差

2、原因：

一般情况，日志从主库发到备库造成的时间很短的，主要原因是备库接收完这个binlog执行这个事务造成的时间，所以，主备延迟最直接的表现是，**备库消费中转日志（relay log）的速度，比主库生产binlog的速度要慢。**

3、主备延迟的来源

* 1、主备库部署机器性能差异
* 2、只考虑主库压力，忽略备库压力，备库写压力大，占用了cpu资源，导致同步延迟
  + **解决方案：**
    - 1、一主多从，分摊读压力（\*）
    - 2、通过binlog输出到外部系统，比如Hadoop这类系统，让外部系统提供统计类查询的能力。
* 3、大事务，大事务让主库执行很久，那么到备库也要执行很久，导致延迟很久，比如一次是删很多数据

**主备切换策略（**由于有主备延迟，导致有多种切换策略**）**

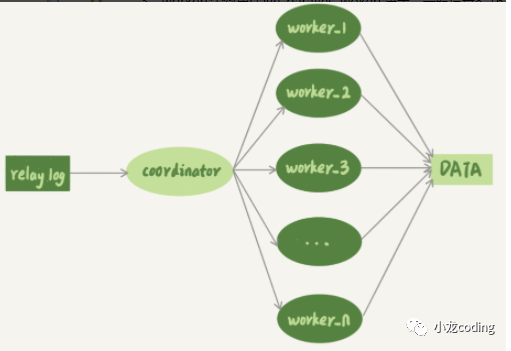
1、可靠性优先策略(实际保证这个)

* **切换流程**：等到主备数据同步再将备库设置为读写，业务转到备库B
* **判断备库B的同步延迟时间（seconds\_behind\_master）小于某个值时，将主库A设置为只读（readonly=ture）,此时系统会不可用主从都只读，然后继续等备库的同步延迟时间为0了，将B备库设置为读写（readonly=false）,然后业务请求转到B**
* **问题**：当主库设置为只读时，此时旧的主备都只读，系统不可用，所以要求备库同步延迟尽量短时才开始切换

2、备库并行复制

**前行知识**：若备库执行日志的速度一直慢于主库生成日志速度，延迟可能会达到小时级别，若主库持续高压力，备库可能始终追追不上主库节奏。采用备库并行复制解决。

**模型：**



* 1、coordinator负责读取中转日志和分发事务
* 2、各个workers负责真正执行
* 3、workers个数由slave\_paralles\_wokers决定，一般设置8-16（32核），备库还需要其他查询

**coordinator分发规则(每个版本须遵守**)

* 1、更新同一行的两个事务须分配到同一个worker
* 2、同一个事务不能拆分，需分配到同一个worker