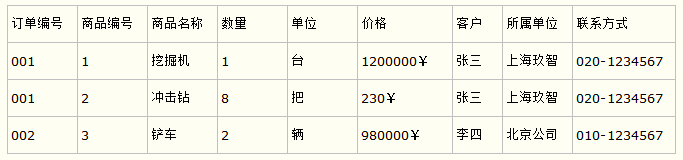
# 数据库设计三大范式

## 第一范式(确保每列保持原子性)

如果数据库表中的所有字段值都是不可分解的原子值，该数据库表满足了第一范式。

## 第二范式(确保表中的每列都和主键相关)

第二范式需要确保数据库表中的每一列都和主键相关，而不能只与主键的某一部分相关（主要针对联合主键而言）。也就是说在一个数据库表中，一个表中只能保存一种数据，不可以把多种数据保存在同一张数据库表中。

比如要设计一个订单信息表，将订单编号和商品编号作为数据库表的联合主键。

产生一个问题：这个表中是以订单编号和商品编号作为联合主键。这样在该表中商品名称、单位、商品价格等信息不与该表的主键相关，而仅仅是与商品编号相关。所以在这里违反了第二范式的设计原则。而如果把这个订单信息表进行拆分，把商品信息分离到另一个表中，把订单项目表也分离到另一个表中，就非常完美了。如下所示。



这样设计，在很大程度上减小了数据库的冗余。如果要获取订单的商品信息，使用商品编号到商品信息表中查询即可。

## 第三范式(确保每列都和主键列直接相关,而不是间接相关)

第三范式需要确保数据表中的每一列数据都和主键直接相关，而不能间接相关。

比如在设计一个订单数据表的时候，可以将客户编号作为一个外键和订单表建立相应的关系。而不可以在订单表中添加关于客户其它信息（比如姓名、所属公司等）的字段。如下面这两个表所示的设计就是一个满足第三范式的数据库表。



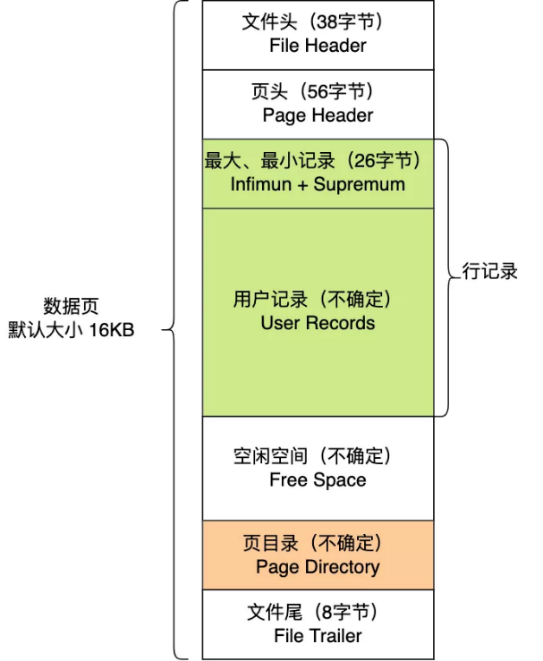
这样在查询订单信息的时候，就可以使用客户编号来引用客户信息表中的记录，也不必在订单信息表中多次输入客户信息的内容，减小了数据冗余。

# 数据的读取

1. InnoDB 的数据是按「数据页」为单位来读写的，也就是说，当需要读一条记录的时候，并不是将这个记录本身从磁盘读出来，而是以页为单位，将其整体读入内存。
2. 数据库的 I/O 操作的最小单位是页，InnoDB 数据页的默认大小是 16KB，意味着数据库每次读写都是以 16KB 为单位的，一次最少从磁盘中读取 16K 的内容到内存中，一次最少把内存中的 16K 内容刷新到磁盘中。

# 数据页的结构

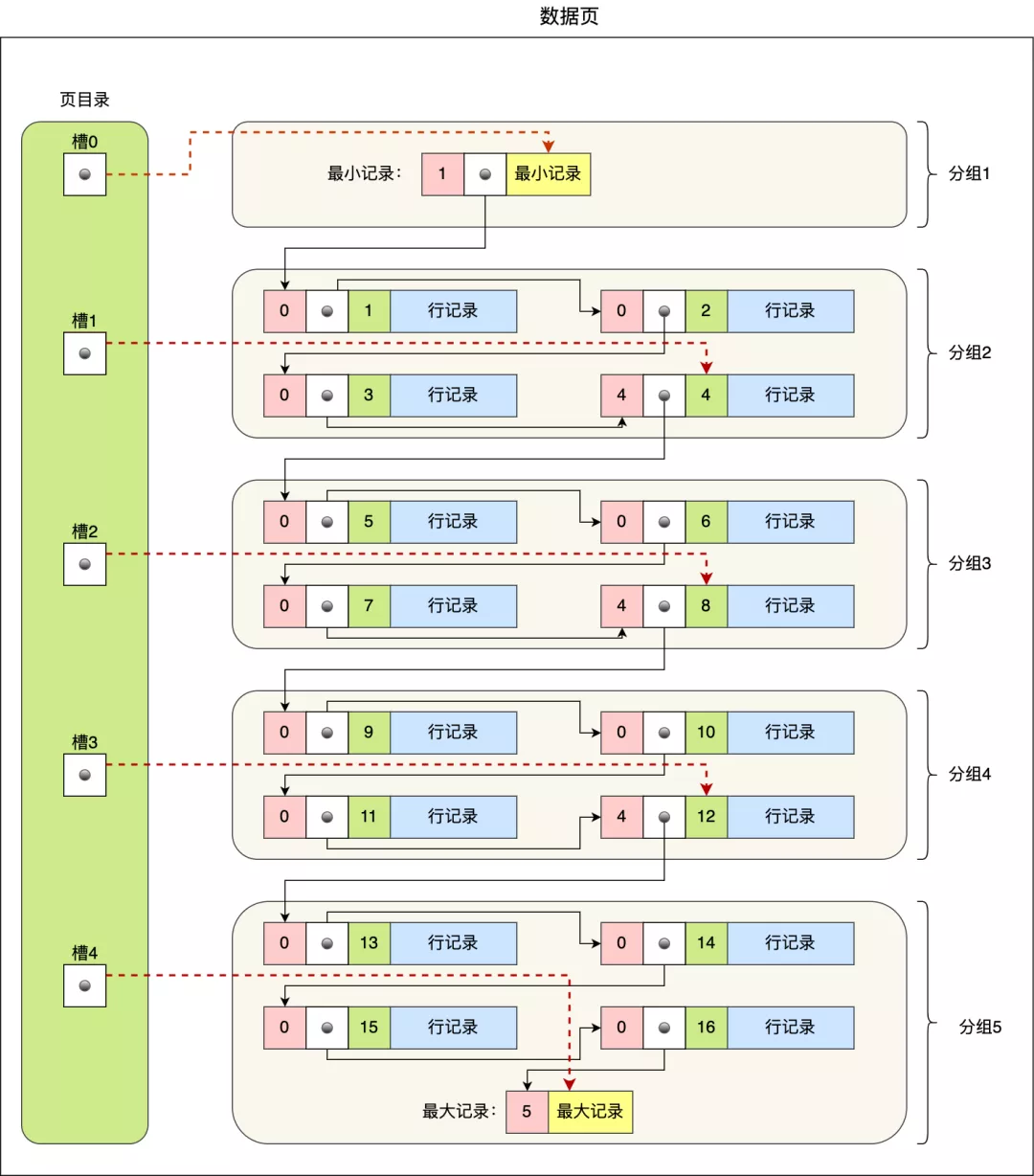
数据页的主要作用是存储记录，也就是数据库的数据



# 用户记录及页目录的组织方式

数据页中的记录按照「主键」顺序组成单向链表，单向链表的特点就是插入、删除非常方便，但是检索效率不高，最差的情况下需要遍历链表上的所有节点才能完成检索。

数据页中有一个页目录，起到记录的索引作用



1. 将所有的记录划分成几个组，这些记录包括最小记录和最大记录，但不包括标记为“已删除”的记录；
2. 每个记录组的最后一条记录就是组内最大的那条记录，并且最后一条记录的头信息中会存储该组一共有多少条记录，作为 n\_owned 字段（上图中粉红色字段）
3. 页目录用来存储每组最后一条记录的地址偏移量，这些地址偏移量会按照先后顺序存储起来，每组的地址偏移量也被称之为槽（slot），每个槽相当于指针指向了不同组的最后一个记录。
4. InnoDB 对每个分组中的记录条数都是有规定的，槽内的记录就只有几条：

* 第一个分组中的记录只能有 1 条记录；
* 最后一个分组中的记录条数范围只能在 1-8 条之间；
* 剩下的分组中记录条数范围只能在 4-8 条之间。

从图可以看到，页目录就是由多个槽组成的，槽相当于分组记录的索引。然后，因为记录是按照「主键值」从小到大排序的，所以我们通过槽查找记录时，可以使用二分法快速定位要查询的记录在哪个槽（哪个记录分组），定位到槽后，再遍历槽内的所有记录，找到对应的记录，无需从最小记录开始遍历整个页中的记录链表。

# 为什么推荐使用自增ID作为主键？

自增ID可以保证每次插入时B+索引是从右边扩展的，可以避免B+树和频繁合并和分裂（对比使用UUID）。如果使用字符串主键和随机主键，会使得数据随机插入，效率比较差。

如果表使用自增主键，那么每次插入新的记录，记录就会顺序添加到当前索引节点的后续位置，当一页写满，就会自动开辟一个新的页。 如果使用非自增主键（如果身份证号或学号等），由于每次插入主键的值近似于随机，因此每次新纪录都要被插到现有索引页得中间某个位置， 频繁的移动、分页操作造成了大量的碎片，得到了不够紧凑的索引结构，后续不得不通过OPTIMIZE TABLE（optimize table）来重建表并优化填充页面。

# 为什么要使用索引？

1. 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。

2. 可以大大加快 数据的检索速度（大大减少的检索的数据量）, 这也是创建索引的最主要的原因。

3. 帮助服务器避免排序和临时表

# 索引创建原则

1．选择唯一性索引

唯一性索引的值是唯一的，可以更快速的通过该索引来确定某条记录。例如，学生表中学号是具有唯一性的字段。为该字段建立唯一性索引可以很快的确定某个学生的信息。如果使用姓名的话，可能存在同名现象，从而降低查询速度。

2．为经常需要排序、分组和联合操作的字段建立索引

经常需要ORDER BY、GROUP BY、DISTINCT和UNION等操作的字段，排序操作会浪费很多时间。如果为其建立索引，可以有效地避免排序操作。

3．为常作为查询条件的字段建立索引

如果某个字段经常用来做查询条件，那么该字段的查询速度会影响整个表的查询速度。因此，为这样的字段建立索引，可以提高整个表的查询速度。

4．限制索引的数目

索引的数目不是越多越好。每个索引都需要占用磁盘空间，索引越多，需要的磁盘空间就越大。修改表时，对索引的重构和更新很麻烦。越多的索引，会使更新表变得很浪费时间。

5．尽量使用数据量少的索引

如果索引的值很长，那么查询的速度会受到影响。例如，对一个CHAR(100)类型的字段进行全文检索需要的时间肯定要比对CHAR(10)类型的字段需要的时间要多。

6．尽量使用前缀来索引

如果索引字段的值很长，最好使用值的前缀来索引。例如，TEXT和BLOG类型的字段，进行全文检索会很浪费时间。如果只检索字段的前面的若干个字符，这样可以提高检索速度。

7．删除不再使用或者很少使用的索引

表中的数据被大量更新，或者数据的使用方式被改变后，原有的一些索引可能不再需要。数据库管理员应当定期找出这些索引，将它们删除，从而减少索引对更新操作的影响。

8 . 最左匹配原则，非常重要的原则。

mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如a 1=”” and=”” b=”2” c=”“> 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。

# MySQL索引类型

MySQL官方对索引的定义为：索引（Index）是帮助MySQL高效获取数据的数据结构。MySQL索引常用有：主键索引、唯一索引、普通索引、全文索引和组合索引。

唯一索引:

索引列的所有值都只能出现一次，即必须唯一，值可以为空。

主键索引:

主键是一种唯一性索引，但它必须指定为PRIMARY KEY，每个表只能有一个主键。

普通索引:

基本的索引类型，值可以为空，没有唯一性的限制。

全文索引:

全文索引的索引类型为FULLTEXT。全文索引可以在varchar、char、text类型的列上创建。可以通过alter table或create index命令创建。FULLTEXT 用于搜索很长一篇文章的时候，效果最好。alter table 表名 add FULLTEXT(‘字段名’)

组合索引:

指多个字段上创建的索引，只有在查询条件中使用了创建索引时的第一个字段，索引才会被使用。

# MySQL索引结构

1. B树
2. B+树
3. Hash索引

索引列会被存储在匹配到的hash bucket里面的表里，这个表里会有实际的数据行指针，再根据实际的数据行指针查找对应的数据行。

要查找一行数据或者处理一个where子句，SQL引擎需要做下面几件事

1. 根据where条件里面的参数生成合适的哈希函数
2. 索引列进行匹配，匹配到对应hash bucket，找到对应hash bucket意味着也找到了对应的数据行指针（row pointer）
3. 读取数据

缺点

1、因为Hash索引比较的是经过Hash计算的值，所以只能进行等式比较，不能用于范围查询

2、由于哈希值是按照顺序排列的，但是哈希值映射的真正数据在哈希表中就不一定按照顺序排列，所以无法利用Hash索引来加速任何排序操作

3、不能用部分索引键来搜索，因为组合索引在计算哈希值的时候是一起计算的。

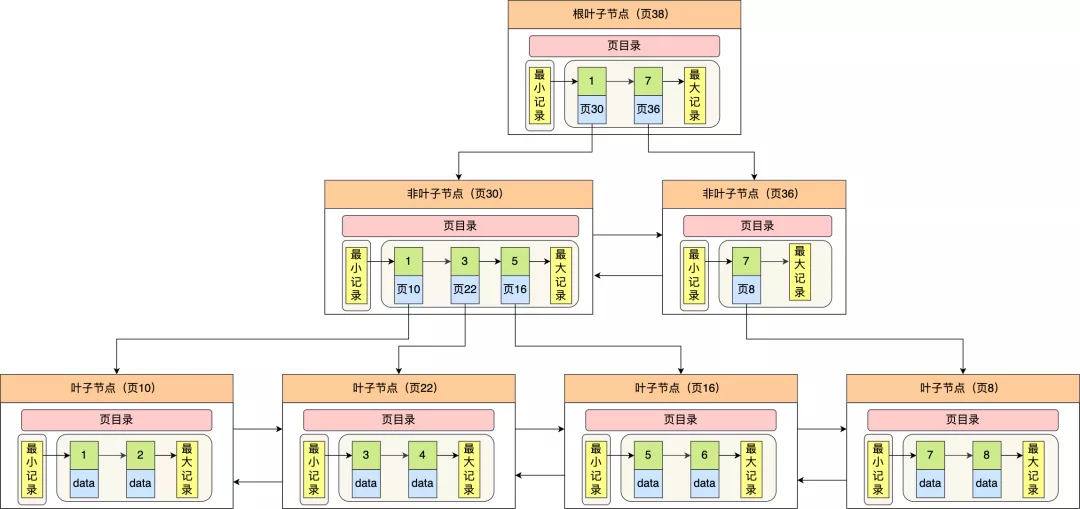
4、当哈希值大量重复且数据量非常大时，其检索效率并没有Btree索引高的。

# 为什么使用B+树？

1. B+树自平衡的多叉树，比二叉树更矮，查询效率更高
2. B+ 树的非叶子节点不存放实际的记录数据，仅存放索引。B 树的每个节点都包含数据（索引+记录），而用户的记录数据的大小很有可能远远超过了索引数据，我们只是想读取这些节点的索引数据来做比较查询，而「非 查询节点」里的记录数据对我们是没用的，这样不仅增多磁盘 I/O 操作次数，也占用内存资源。
3. B+ 树的插入和删除效率更高。B+ 树有大量的冗余节点，这样使得删除一个节点的时候，可以直接从叶子节点中删除，甚至可以不动非叶子节点，这样删除非常快。B+ 树的插入也是一样，有冗余节点，插入可能存在节点的分裂（如果节点饱和），但是最多只涉及树的一条路径。而且 B+ 树会自动平衡，不需要像更多复杂的算法，类似红黑树的旋转操作等。
4. B+ 树所有叶子节点间还有一个双向链表进行连接，这种设计对范围查找非常有帮助。比如说我们想知道 12 月 1 日和 12 月 12 日之间的订单，这个时候可以先查找到 12 月 1 日所在的叶子节点，然后利用链表向右遍历，直到找到 12 月12 日的节点，这样就不需要从根节点查询了，进一步节省查询需要的时间。

# B+ 树是如何进行查询的？

1. 磁盘的 I/O 操作次数对索引的使用效率至关重要，因此在构造索引的时候，我们更倾向于采用“矮胖”的 B+ 树数据结构，这样所需要进行的磁盘 I/O 次数更少，而且 B+ 树 更适合进行关键字的范围查询。
2. InnoDB 里的 B+ 树中的每个节点都是一个数据页，结构示意图如上



1. 只有叶子节点（最底层的节点）才存放了数据，非叶子节点（其他上层节）仅用来存放目录项作为索引。
2. 非叶子节点分为不同层次，通过分层来降低每一层的搜索量；
3. 所有节点按照索引键大小排序，构成一个双向链表，便于范围查询；
4. 在定位记录所在哪一个页时，也是通过二分法快速定位到包含该记录的页。定位到该页后，又会在该页内进行二分法快速定位记录所在的分组（槽号），最后在分组内进行遍历查找。

# 聚集索引和二级索引

1. 聚集索引的叶子节点存放的是实际数据，所有完整的用户记录都存放在聚集索引的叶子节点；因为表的数据都是存放在聚集索引的叶子节点里，所以 InnoDB 存储引擎一定会为表创建一个聚集索引，且由于数据在物理上只会保存一份，所以聚簇索引只能有一个。
2. 一张表只能有一个聚簇索引，那为了实现非主键字段的快速搜索，就引出了二级索引（非聚簇索引/辅助索引），它也是利用了 B+ 树的数据结构，但是二级索引的叶子节点存放的是主键值，不是实际数据。
3. 如果某个查询语句使用了二级索引，但是查询的数据不是主键值，这时在二级索引找到主键值后，需要去聚簇索引中获得数据行，这个过程就叫作「回表」，也就是说要查两个 B+ 树才能查到数据。
4. 当查询的数据是主键值时，因为只在二级索引就能查询到，不用再去聚簇索引查，这个过程就叫作「索引覆盖」，也就是只需要查一个 B+ 树就能找到数据。

# 索引失败

1. 对索引使用左或者左右模糊匹配，也就是 like %xx 或者 like %xx% 这两种方式都会造成索引失效。原因在于查询的结果可能是多个，不知道从哪个索引值开始比较，于是就只能通过全表扫描的方式来查询。
2. 对索引进行函数/对索引进行表达式计算，因为索引保持的是索引字段的原始值，而不是经过函数计算的值，自然就没办法走索引。
3. 应该尽量避免在 where 子句中使用 != 或 not in 或 <> 操作符，因为这几个操作符都会导致索引失效而进行全表扫描。
4. WHERE 子句中的 OR语句，只要有条件列不是索引列，就会进行全表扫描。
5. 当查询条件左右两侧类型不匹配的时候会发生隐式转换，隐式转换带来的影响就是可能导致索引失效而进行全表扫描。select \* from news where date\_str = 201701

# MySQL优化

## 应用层层面

1. 尽量避免使用select \*
2. 数据类型优化
3. 读写分离
4. 表中的行或列不要太多
5. 批量插入代替循环单条插入
6. 对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。

核心服务层面

使用查询缓存

用多个小表代替一个大表，注意不要过度设计

## 存储引擎层面

创建高性能索引

1. （见上面的索引失败情况）
2. 多个单列索引并不是最佳选择，可以使用组合索引尽量覆盖索引。覆盖索引的好处 如果一个索引包含所有需要的查询的字段的值，直接根据索引的查询结果返回数据，而无需读表，能够极大的提高性能。
3. 组合索引符合最左匹配原则

# 索引的缺点

1. 当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

2. 索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

3. 创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

# 分表

拆分其实又分**垂直拆分**和**水平拆分**

案例： 简单购物系统暂设涉及如下表：

1.产品表（数据量10w，稳定）

2.订单表（数据量200w，且有增长趋势）

3.用户表 （数据量100w，且有增长趋势）

**垂直拆分**

解决问题：表与表之间的io竞争

不解决问题：单表中数据量增长出现的压力

方案： 把产品表和用户表放到一个server上 订单表单独放到一个server上

**水平拆分**

解决问题：单表中数据量增长出现的压力

不解决问题：表与表之间的io争夺

方案：**用户表** 通过性别拆分为男用户表和女用户表，**订单表** 通过已完成和完成中拆分为已完成订单和未完成订单，**产品表** 未完成订单放一个server上，已完成订单表盒男用户表放一个server上，女用户表放一个server上(女的爱购物 哈哈)。

# MySQL事务

我们在数据库操作前先开启事务，等所有数据库操作执行完成后，才提交事务，对于已经提交的事务来说，该事务对数据库所做的修改将永久生效，如果中途发生发生中断或错误，那么该事务期间对数据库所做的修改将会被回滚到没执行该事务之前的状态。

1. 事务是由 MySQL 的引擎来实现的，我们常见的 InnoDB 引擎它是支持事务的。
2. 不过并不是所有的引擎都能支持事务，比如 MySQL 原生的 MyISAM 引擎就不支持事务，也正是这样，所以大多数 MySQL 的引擎都是用 InnoDB。

原子性（Atomicity）：一个事务中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节，而且事务在执行过程中发生错误，会被回滚到事务开始前的状态，就像这个事务从来没有执行过一样；

一致性（Consistency）：数据库的完整性不会因为事务的执行而受到破坏，比如表中有一个字段为姓名，它有唯一约束，也就是表中姓名不能重复，如果一个事务对姓名字段进行了修改，但是在事务提交后，表中的姓名变得非唯一性了，这就破坏了事务的一致性要求，这时数据库就要撤销该事务，返回初始化的状态。

隔离性（Isolation）：数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。

持久性（Durability）：事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

# InnoDB 如何保证事务的四个特性？

1. 持久性是通过 redo log （重做日志）来保证的；

当做数据修改的时候，不仅在内存中操作，还会在redo log中记录这次操作。当事务提交的时候，会将redo log日志进行刷盘(redo log一部分在内存中，一部分在磁盘上)。当数据库宕机重启的时候，会将redo log中的内容恢复到数据库中，再根据undo log和binlog内容决定回滚数据还是提交数据。

1. 原子性和一致性是通过 undo log（回滚日志） 来保证的；

undo log记录了这些回滚需要的信息，当事务执行失败或调用了rollback，导致事务需要回滚，便可以利用undo log中的信息将数据回滚到修改之前的样子。

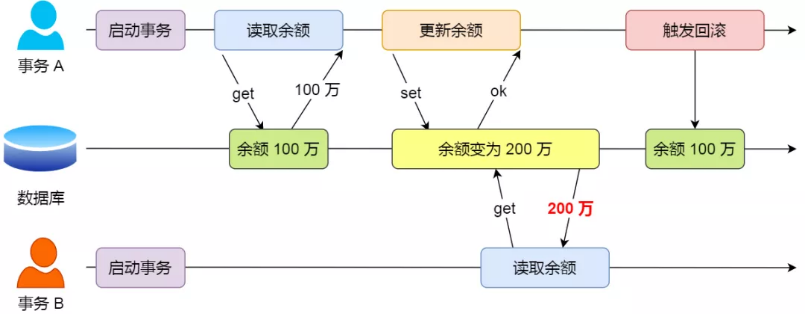
1. 隔离性是通过 MVCC（多版本并发控制） 或锁机制来保证的；

# 并行事务会引发什么问题？

在同时处理多个事务的时候，就可能出现脏读（dirty read）、不可重复读（non-repeatable read）、幻读（phantom read）的问题。

脏读

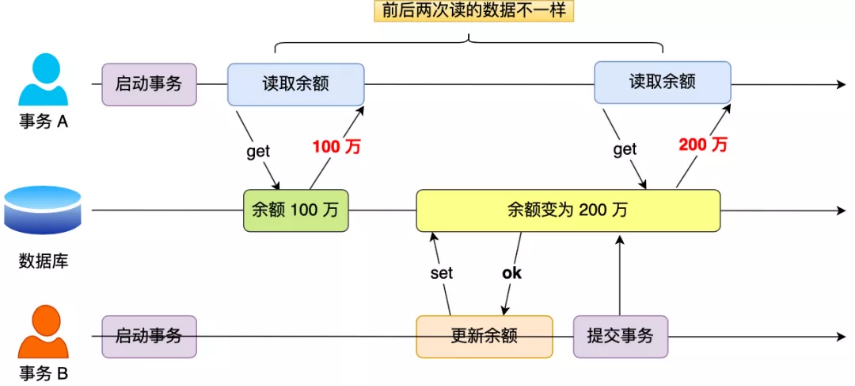
如果一个事务「读到」了另一个「未提交事务修改过的数据」，就意味着发生了「脏读」现象。



因为事务 A 是还没提交事务的，也就是它随时可能发生回滚操作，如果事务 A 发生了回滚，那么事务 B 刚才得到的数据就是过期的数据，这种现象就被称为脏读。

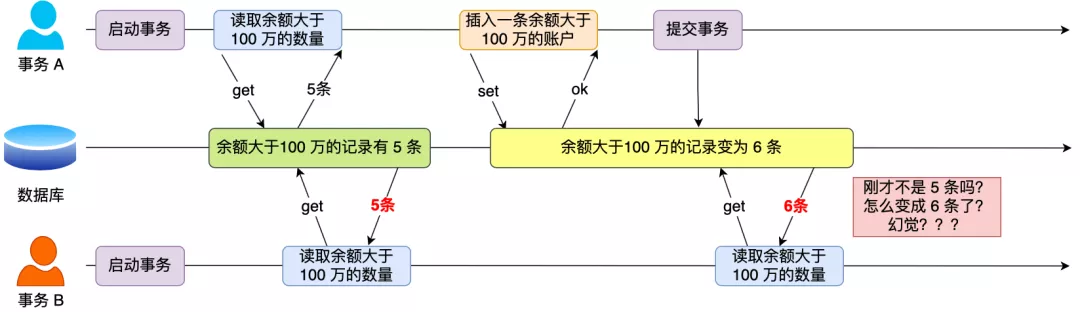
不可重复读

在一个事务内多次读取同一个数据，如果出现前后两次读到的数据不一样的情况，就意味着发生了「不可重复读」现象。



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。



# 幻读是怎么被解决的？

InnoDB 存储引擎的默认事务隔离级别是「可重复读」，但是在这个隔离级别下，在多个事务并发的时候，会出现幻读的问题，所谓的幻读是指在同一事务下，连续执行两次同样的查询语句，第二次的查询语句可能会返回之前不存在的行。

因此 InnoDB 存储引擎自己实现了行锁，通过 next-key 锁（记录锁和间隙锁的组合）来锁住记录本身和记录之间的“间隙”，防止其他事务在这个记录之间插入新的记录，从而避免了幻读现象。

MySQL 里除了普通select查询是快照读，其他都是**当前读**，比如update、insert、delete，这些语句执行前都会查询最新版本的数据，然后再做进一步的操作。

select ... for update 这种查询语句是当前读，每次执行的时候都是读取最新的数据。

**因此，要讨论「可重复读」隔离级别的幻读现象，是要建立在「当前读」的情况下。**

接下来，我们假设select ... for update当前读是不会加锁的（实际上是会加锁的）

比如，下面事务 A 查询语句会锁住(2, +∞]范围的记录，然后期间如果有其他事务在这个锁住的范围插入数据就会被阻塞。



next-key 锁的加锁规则其实挺复杂的，在一些场景下会退化成记录锁或间隙锁

需要注意的是，next-key lock 锁的是索引，而不是数据本身，所以如果 update 语句的 where 条件没有用到索引列，那么就会全表扫描，在一行行扫描的过程中，不仅给行加上了行锁，还给行两边的空隙也加上了间隙锁，相当于锁住整个表，然后直到事务结束才会释放锁。

# SELECT FOR UPDATE

FOR UPDATE 是一种行级锁，又叫排它锁。仅适用于 InnoDB，并且必须开启事务，在 BEGIN 与 COMMIT 之间才生效。

开启两个 MySQL 事务

事务1

 BEGIN  
 SELECT id, user\_id, goods\_id, state FROM goods\_order WHERE id = 2 FOR UPDATE  
 UPDATE goods\_order SET state = 2 WHERE id=2  
 COMMIT

事务2

 BEGIN  
 SELECT id, user\_id, goods\_id, state FROM goods\_order WHERE id = 2 FOR UPDATE  
 UPDATE goods\_order SET state = 3 WHERE id=2  
 COMMIT

当 事务1 执行完 SELECT ... FOR UPDATE 后（此时事务还未结束）， 事务2 执行 SELECT ... FOR UPDATE 语句时将会阻塞在那，直到 事务1 中的事务结束（执行完COMMIT）。

其中一个使用场景是用于修改订单状态，修改订单状态往往需要两个步骤：

1. 查询订单状态。
2. 修改订单状态。

当有两个任务同时请求时，有可能出现如下情况：

1. 任务A查询到订单状态为1。
2. 任务B查询到订单状态为1。
3. 任务A修改订单状态为2。
4. 任务B修改订单状态为3。

其中，任务B将订单状态改为3的前提是订单状态为1，但是上述情况下任务B修改订单时订单状态已变成2了，并不符合预期，通过 SELECT ... FRO UPDATE 就可以解决上述问题。

# 事务的隔离级别

SQL 标准提出了四种隔离级别来规避这些现象，隔离级别约高，性能效率就越低，这四个隔离级别如下：

读未提交（read uncommitted），指一个事务还没提交时，它做的变更就能被其他事务看到；

读提交（read committed），指一个事务提交之后，它做的变更才能被其他事务看到；

可重复读（repeatable read），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；

串行化（serializable ）；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；

按隔离水平高低排序如下：



针对不同的隔离级别，并发事务时可能发生的现象也会不同。



1. 要解决脏读现象，就要升级到「读提交」以上的隔离级别；要解决不可重复读现象，就要升级到「可重复读」的隔离级别。
2. 要解决幻读现象不建议将隔离级别升级到「串行化」，因为这样会导致数据库在并发事务时性能很差。InnoDB 引擎的默认隔离级别虽然是「可重复读」，但是它通过next-key lock 锁（行锁和间隙锁的组合）来锁住记录之间的“间隙”和记录本身，防止其他事务在这个记录之间插入新的记录，这样就避免了幻读现象。

# 隔离级别案例



在不同隔离级别下，事务 A 执行过程中查询到的余额可能会不同：

1. 在「读未提交」隔离级别下，事务 B 修改余额后，虽然没有提交事务，但是此时的余额已经可以被事务 A 看见了，于是事务 A 中余额 V1 查询的值是 200 万，余额 V2、V3 自然也是 200 万了；
2. 在「读提交」隔离级别下，事务 B 修改余额后，因为没有提交事务，所以事务 A 中余额 V1 的值还是 100 万，等事务 B 提交完后，最新的余额数据才能被事务 A 看见，因此额 V2、V3 都是 200 万；
3. 在「可重复读」隔离级别下，事务 A 只能看见启动事务时的数据，所以余额 V1、余额 V2 的值都是 100 万，当事务 A 提交事务后，就能看见最新的余额数据了，所以余额 V3 的值是 200 万；
4. 在「串行化」隔离级别下，事务 B 在执行将余额 100 万修改为 200 万时，由于此前事务 A 执行了读操作，这样就发生了读写冲突，于是就会被锁住，直到事务 A 提交后，事务 B 才可以继续执行，所以从 A 的角度看，余额 V1、V2 的值是 100 万，余额 V3 的值是 200万。

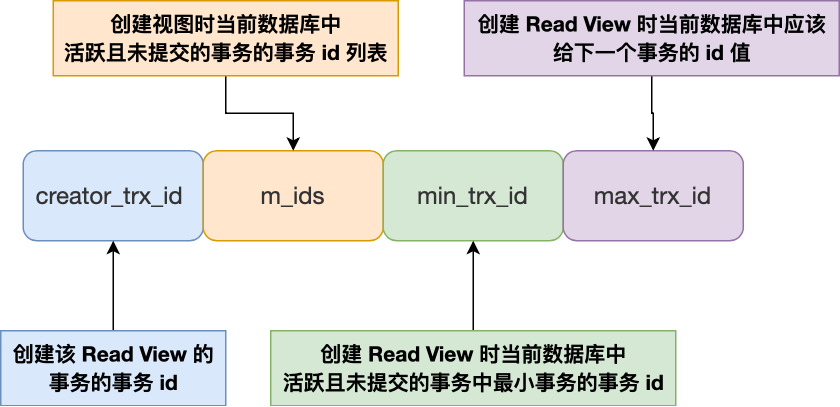
# 四种隔离级别具体是如何实现的呢？

1. 对于「读未提交」隔离级别的事务来说，因为可以读到未提交事务修改的数据，所以直接读取最新的数据就好了；
2. 对于「串行化」隔离级别的事务来说，通过加读写锁的方式来避免并行访问；
3. 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。
4. 「读提交」和「可重复读」这两个隔离级别实现是通过「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，这就叫 MVCC（多版本并发控制）。

# 可重复读隔离级别是如何实现的？

「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。想要知道可重复读隔离级别是如何实现的，我们需要了解两个知识：

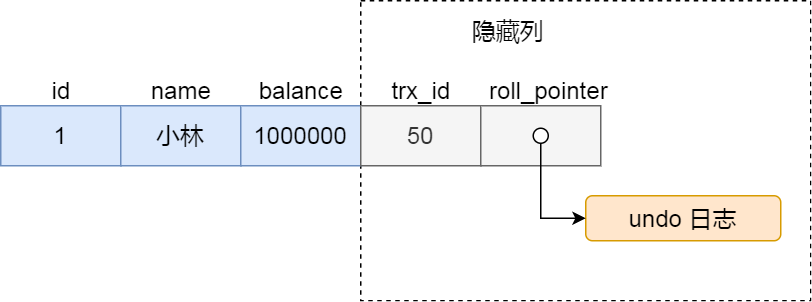
* Read View 中四个字段作用；
* 聚族索引记录中两个跟事务有关的隐藏列；



Read View 有四个重要的字段：

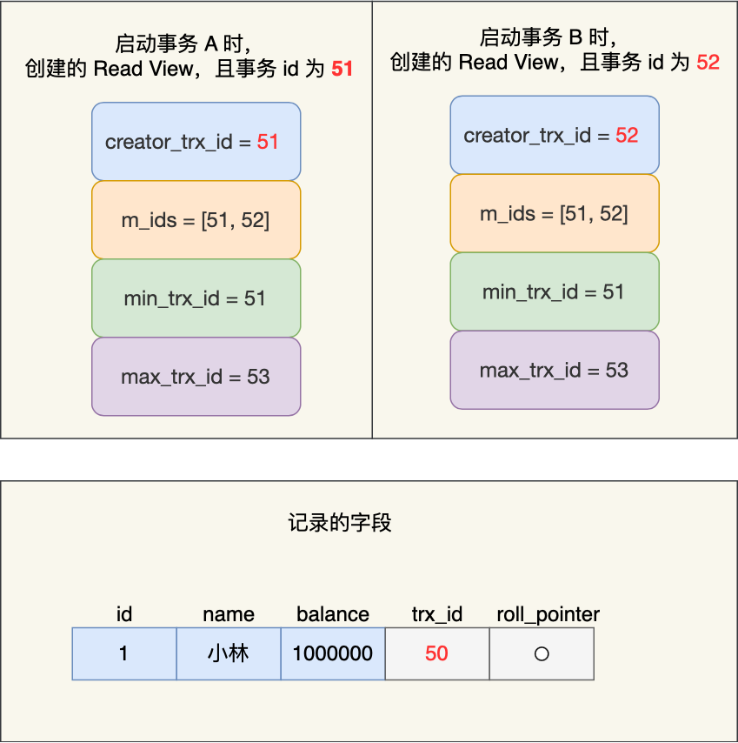
* m\_ids ：指的是创建 Read View 时当前数据库中**活跃且未提交的事务的事务 id 列表**，注意是一个列表。
* min\_trx\_id ：指的是创建 Read View 时当前数据库中**活跃且未提交的事务中最小事务的事务 id**，也就是 m\_ids 的最小值。
* max\_trx\_id ：这个并不是 m\_ids 的最大值，而是**创建 Read View 时当前数据库中应该给下一个事务的 id 值**；
* creator\_trx\_id ：指的是**创建该 Read View 的事务的事务 id**。

知道了 Read View 的字段，我们还需要了解聚族索引记录中的两个隐藏列，假设在账户余额表插入一条小林余额为 100 万的记录，然后我把这两个隐藏列也画出来，该记录的整个

示意图如下：

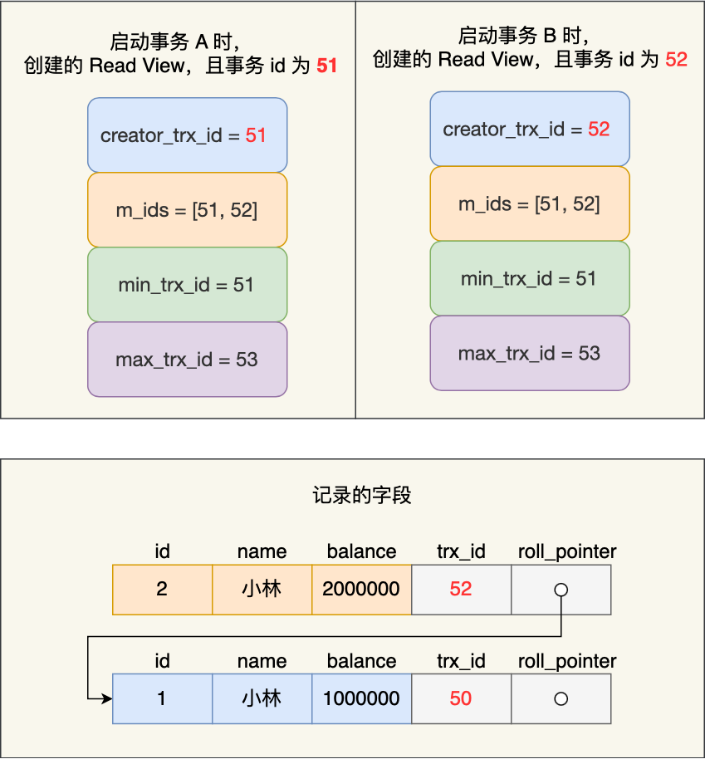
对于使用 InnoDB 存储引擎的数据库表，它的聚族索引记录中都包含下面两个隐藏列：

* trx\_id，当一个事务对某条聚族索引记录进行改动时，就会**把该事务的事务 id 记录在 trx\_id 隐藏列里**；
* roll\_pointer，每次对某条聚族索引记录进行改动时，都会把旧版本的记录写入到 undo 日志中，然后**这个隐藏列是个指针，指向每一个旧版本记录**，于是就可以通过它找到修改前的记录。

了解完这两个知识点后，就可以跟大家说说可重复读隔离级别是如何实现的。假设事务 A 和 事务 B 差不多同一时刻启动，那这两个事务创建的 Read View 如下：

事务 A 和 事务 B 的 Read View 具体内容如下：

* 在事务 A 的 Read View 中，它的事务 id 是 51，由于与事务 B 同时启动，所以此时活跃的事务的事务 id 列表是 51 和 52，活跃的事务 id 中最小的事务 id 是事务 A 本身，下一个事务 id 应该是 53。
* 在事务 B 的 Read View 中，它的事务 id 是 52，由于与事务 A 同时启动，所以此时活跃的事务的事务 id 列表是 51 和 52，**活跃的事务 id 中最小的事务 id 是事务 A**，下一个事务 id 应该是 53。

然后让事务 A 去读账户余额为 100 万的记录，在找到记录后，它会先看这条记录的 trx\_id，此时发现 trx\_id 为 50，通过和事务 A 的 Read View 的 m\_ids 字段发现，**该记录的事务 id 并不在活跃事务的列表中，并且小于事务 A 的事务 id，这意味着，这条记录的事务早就在事务 A 前提交过了，所以该记录对事务 A 可见**，也就是事务 A 可以获取到这条记录。接着，事务 B 通过 update 语句将这条记录修改了，将小林的余额改成 200 万，这时 MySQL 会记录相应的 undo log，并以链表的方式串联起来，形成**版本链**，如下图：

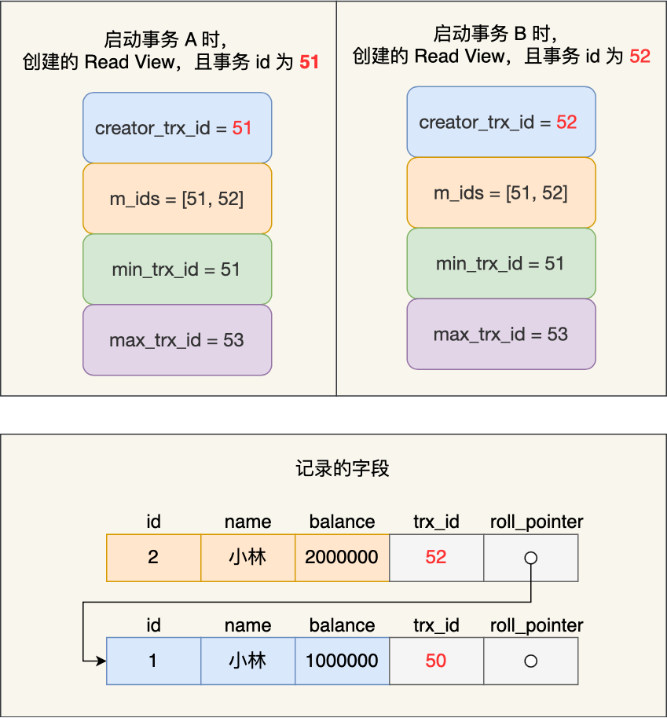
你可以在上图的「记录字段」看到，由于事务 B 修改了该记录，以前的记录就变成旧版本记录了，于是最新记录和旧版本记录通过链表的方式串起来，而且最新记录的 trx\_id 是事务 B 的事务 id。然后如果事务 A 再次读取该记录，**发现这条记录的 trx\_id 为 52，比自己的事务 id 还大，并且比下一个事务 id 53 小，这意味着，事务 A 读到是和自己同时启动事务的事务 B 修改的数据，这时事务 A 并不会读取这条记录，而是沿着 undo log 链条往下找旧版本的记录，直到找到 trx\_id 等于或者小于事务 A 的事务 id 的第一条记录**，所以事务 A 再一次读取到 trx\_id 为 50 的记录，也就是小林余额是 100 万的这条记录。「可重复读」隔离级别就是在启动时创建了 Read View，然后在事务期间读取数据的时候，在找到数据后，先会将该记录的 trx\_id 和该事务的 Read View 里的字段做个比较：

* 如果记录的 trx\_id 比该事务的 Read View 中的 creator\_trx\_id 要小，且不在 m\_ids 列表里，这意味着这条记录的事务早就在该事务前提交过了，所以该记录对该事务可见；
* 如果记录的 trx\_id 比该事务的 Read View 中的 creator\_trx\_id 要大，且在 m\_ids 列表里，这意味着该事务读到的是和自己同时启动的另外一个事务修改的数据，这时就不应该读取这条记录，而是沿着 undo log 链条往下找旧版本的记录，直到找到 trx\_id 等于或者小于该事务 id 的第一条记录。

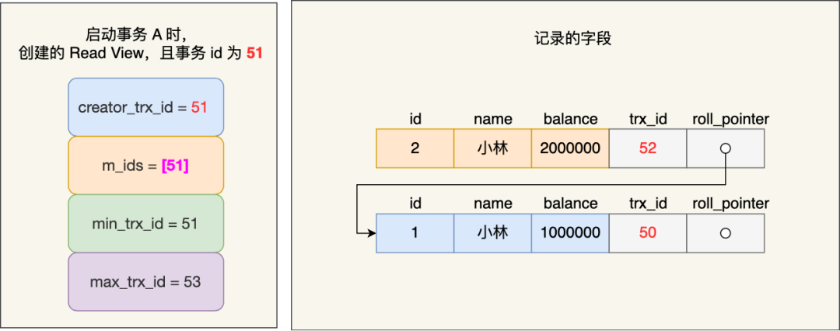
就是通过这样的方式实现了，「可重复读」隔离级别下在事务期间读到的数据都是事务启动前的记录。**这种通过记录的版本链来控制并发事务访问同一个记录时的行为，这就叫 MVCC（多版本并发控制）。**

# 读提交隔离级别是如何实现的？

「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。那读提交隔离级别是怎么实现呢？

我们还是以前面的例子来聊聊。假设事务 A 和 事务 B 差不多同一时刻启动，然后事务 B 将小林的账户余额修改成了 200 万，但是事务 B 还未提交，这时事务 A 读到的数据，应该还是小林账户余额为 100 万的数据，那具体怎么做到的呢？

事务 A 在找到小林这条记录时，会看这条记录的 trx\_id，**发现和事务 A 的 Read View 中的 creator\_trx\_id 要大，而且还在 m\_ids 列表里，说明这条记录被事务 B 修改过，而且还可以知道事务 B 并没有提交事务**，因为如果提交了事务，那么这条记录的 trx\_id 就不会在 m\_ids 列表里。因此，**事务 A 不能读取该记录，而是沿着 undo log 链条往下找**。当事务 B 修改数据并提交了事务后，这时事务 A 读到的数据，就是小林账户余额为 200 万的数据，那具体怎么做到的呢？



事务 A 在找到小林这条记录时，会看这条记录的 trx\_id，**发现和事务 A 的 Read View 中的 creator\_trx\_id 要大，而且不在 m\_ids 列表里，说明该记录的 trx\_id 的事务是已经提交过的了，于是事务 A 就可以读取这条记录**，这也就是所谓的读已提交机制。

# InnoDB与MyISAM引擎的区别

1. InnoDB支持事务，MyISAM不支持，对于InnoDB每一条SQL语言都默认封装成事务，自动提交，这样会影响速度，所以最好把多条SQL语言放在begin和commit之间，组成一个事务；
2. InnoDB支持外键，而MyISAM不支持。对一个包含外键的InnoDB表转为MYISAM会失败；
3. InnoDB是聚集索引，使用B+Tree作为索引结构，数据文件是和（主键）索引绑在一起的（表数据文件本身就是按B+Tree组织的一个索引结构），必须要有主键，通过主键索引效率很高。但是辅助索引需要两次查询，先查询到主键，然后再通过主键查询到数据。因此，主键不应该过大，因为主键太大，其他索引也都会很大。

MyISAM是非聚集索引，也是使用B+Tree作为索引结构，索引和数据文件是分离的，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。

也就是说：InnoDB的B+树主键索引的叶子节点就是数据文件，辅助索引的叶子节点是主键的值；而MyISAM的B+树主键索引和辅助索引的叶子节点都是数据文件的地址指针。

1. InnoDB不保存表的具体行数，执行select count(\*) from table时需要全表扫描。而MyISAM用一个变量保存了整个表的行数，执行上述语句时只需要读出该变量即可，速度很快（注意不能加有任何WHERE条件）；

那么为什么InnoDB没有了这个变量呢？

因为InnoDB的事务特性，在同一时刻表中的行数对于不同的事务而言是不一样的，因此count统计会计算对于当前事务而言可以统计到的行数，而不是将总行数储存起来方便快速查询。

1. InnoDB不支持全文索引，而MyISAM支持全文索引，在涉及全文索引领域的查询效率上MyISAM速度更快高；PS：5.7以后的InnoDB支持全文索引了
2. InnoDB支持表、行(默认)级锁，而MyISAM支持表级

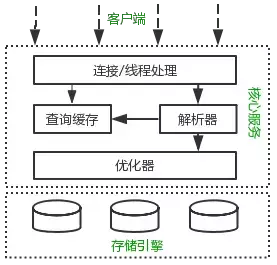
 InnoDB的行锁是实现在索引上的，而不是锁在物理行记录上。潜台词是，如果访问没有命中索引，也无法使用行锁，将要退化为表锁

1. InnoDB表必须有唯一索引（如主键）（用户没有指定的话会自己找/生产一个隐藏列Row\_id来充当默认主键），而MyISAM可以没有

如何选择：

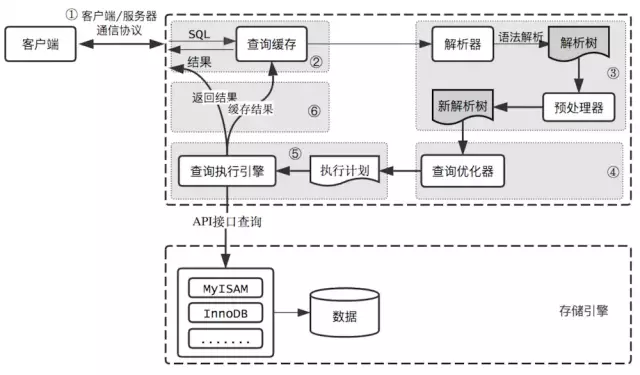
1. 是否要支持事务，如果要请选择InnoDB，如果不需要可以考虑MyISAM；
2. 如果表中绝大多数都只是读查询，可以考虑MyISAM，如果既有读也有写，请使用InnoDB。
3. 系统奔溃后，MyISAM恢复起来更困难，能否接受；
4. MySQL5.5版本开始InnoDB已经成为MySQL的默认引擎(之前是MyISAM)，说明其优势是有目共睹的，如果你不知道用什么，那就用InnoDB，至少不会差。

# MySQL逻辑架构



1. MySQL逻辑架构整体分为三层，最上层为客户端层，并非MySQL所独有，诸如：连接处理、授权认证、安全等功能均在这一层处理。
2. MySQL大多数核心服务均在中间这一层，包括查询解析、分析、优化、缓存、内置函数(比如：时间、数学、加密等函数)。所有的跨存储引擎的功能也在这一层实现：存储过程、触发器、视图等。
3. 最下层为存储引擎，其负责MySQL中的数据存储和提取。和Linux下的文件系统类似，每种存储引擎都有其优势和劣势。中间的服务层通过API与存储引擎通信，这些API接口屏蔽了不同存储引擎间的差异。

# MySQL查询过程



总结一下MySQL整个查询执行过程，总的来说分为5个步骤：

1. 客户端向MySQL服务器发送一条查询请求
2. 服务器首先检查查询缓存，如果命中缓存，则立刻返回存储在缓存中的结果。否则进入下一阶段
3. 服务器进行SQL解析、预处理、再由优化器生成对应的执行计划
4. MySQL根据执行计划，调用存储引擎的API来执行查询
5. 将结果返回给客户端，同时缓存查询结果

## 客户端/服务端通信协议

MySQL客户端/服务端通信协议是“半双工”的：在任一时刻，要么是服务器向客户端发送数据，要么是客户端向服务器发送数据，这两个动作不能同时发生。一旦一端开始发送消息，另一端要接收完整个消息才能响应它，所以我们无法也无须将一个消息切成小块独立发送，也没有办法进行流量控制。

客户端用一个单独的数据包将查询请求发送给服务器，所以当查询语句很长的时候，需要设置max\_allowed\_packet参数。但是需要注意的是，如果查询实在是太大，服务端会拒绝接收更多数据并抛出异常。

## 查询缓存（复杂）

在解析一个查询语句前，如果查询缓存是打开的，那么MySQL会检查这个查询语句是否命中查询缓存中的数据。如果当前查询恰好命中查询缓存，在检查一次用户权限后直接返回缓存中的结果。这种情况下，查询不会被解析，也不会生成执行计划，更不会执行。

MySQL将缓存存放在一个引用表（不要理解成table，可以认为是类似于HashMap的数据结构），通过一个哈希值索引，这个哈希值通过查询本身、当前要查询的数据库、客户端协议版本号等一些可能影响结果的信息计算得来。所以两个查询在任何字符上的不同（例如：空格、注释），都会导致缓存不会命中。

如果查询中包含任何用户自定义函数、存储函数、用户变量、临时表、mysql库中的系统表，其查询结果都不会被缓存。比如函数NOW()或者CURRENT\_DATE()会因为不同的查询时间，返回不同的查询结果，再比如包含CURRENT\_USER或者CONNECION\_ID()的查询语句会因为不同的用户而返回不同的结果，将这样的查询结果缓存起来没有任何的意义。

## 语法解析和预处理

MySQL通过关键字将SQL语句进行解析，并生成一颗对应的解析树。这个过程解析器主要通过语法规则来验证和解析。比如SQL中是否使用了错误的关键字或者关键字的顺序是否正确等等。预处理则会根据MySQL规则进一步检查解析树是否合法。比如检查要查询的数据表和数据列是否存在等等。

## 查询优化（复杂）

多数情况下，一条查询可以有很多种执行方式，最后都返回相应的结果。优化器的作用就是找到这其中最好的执行计划。

MySQL使用基于成本的优化器，它尝试预测一个查询使用某种执行计划时的成本，并选择其中成本最小的一个。在MySQL可以通过查询当前会话的last\_query\_cost的值来得到其计算当前查询的成本。

MySQL的查询优化器是一个非常复杂的部件，它使用了非常多的优化策略来生成一个最优的执行计划：

1. 重新定义表的关联顺序（多张表关联查询时，并不一定按照SQL中指定的顺序进行，但有一些技巧可以指定关联顺序）
2. 优化MIN()和MAX()函数（找某列的最小值，如果该列有索引，只需要查找B+Tree索引最左端，反之则可以找到最大值，具体原理见下文）
3. 提前终止查询（比如：使用Limit时，查找到满足数量的结果集后会立即终止查询）
4. 优化排序（在老版本MySQL会使用两次传输排序，即先读取行指针和需要排序的字段在内存中对其排序，然后再根据排序结果去读取数据行，而新版本采用的是单次传输排序，也就是一次读取所有的数据行，然后根据给定的列排序。对于I/O密集型应用，效率会高很多）

## 查询执行引擎

在完成解析和优化阶段以后，MySQL会生成对应的执行计划，查询执行引擎根据执行计划给出的指令逐步执行得出结果。整个执行过程的大部分操作均是通过调用存储引擎实现的接口来完成，这些接口被称为handlerAPI。查询过程中的每一张表由一个handler实例表示。

实际上，MySQL在查询优化阶段就为每一张表创建了一个handler实例，优化器可以根据这些实例的接口来获取表的相关信息，包括表的所有列名、索引统计信息等。存储引擎接口提供了非常丰富的功能，但其底层仅有几十个接口，这些接口像搭积木一样完成了一次查询的大部分操作。

## 返回结果给客户端

查询执行的最后一个阶段就是将结果返回给客户端。即使查询不到数据，MySQL仍然会返回这个查询的相关信息，比如改查询影响到的行数以及执行时间等等。

如果查询缓存被打开且这个查询可以被缓存，MySQL也会将结果存放到缓存中。

结果集返回客户端是一个增量且逐步返回的过程。有可能MySQL在生成第一条结果时，就开始向客户端逐步返回结果集了。这样服务端就无须存储太多结果而消耗过多内存，也可以让客户端第一时间获得返回结果。需要注意的是，结果集中的每一行都会以一个满足①中所描述的通信协议的数据包发送，再通过TCP协议进行传输，在传输过程中，可能对MySQL的数据包进行缓存然后批量发送。

# MySQL锁

## 全局锁

### 全局锁是怎么用的？

要使用全局锁，则要执行这条命：Flush tables with read lock (FTWRL)

执行后，整个数据库就处于只读状态了，这时其他线程执行以下操作，都会被阻塞：

对数据的增删查改操作，比如 select、insert、delete、update等语句；

对表结构的更改操作，比如 alter table、drop table 等语句。

如果要释放全局锁，则要执行这条命令：unlock tables

同时，当会话断开了，全局锁会被自动释放。

### 全局锁应用场景是什么？

全局锁主要应用于做全库逻辑备份，这样在备份数据库期间，不会因为数据或表结构的更新，而出现备份文件的数据与预期的不一样。

### 加全局锁又会带来什么缺点呢？

加上全局锁，意味着整个数据库都是只读状态。

那么如果数据库里有很多数据，备份就会花费很多的时间，关键是备份期间，业务只能读数据，而不能更新数据，这样会造成业务停滞。

### 既然使用全局锁会影响业务，那有什么其他方式可以避免？

如果数据库的引擎支持的事务支持可重复读的隔离级别，那么在备份数据库之前先开启事务，会先创建 Read View，然后整个事务执行期间都在用这个 Read View，而且由于 MVCC 的支持，备份期间业务依然可以对数据进行更新操作。

因为在可重复读的隔离级别下，即使其他事务更新了表的数据，也不会影响备份数据库时的 Read View，这就是事务四大特性中的隔离性，这样备份期间备份的数据一直是在开启事务时的数据。

InnoDB 存储引擎默认的事务隔离级别正是可重复读，因此可以采用这种方式来备份数据库。

但是，对于 MyISAM 这种不支持事务的引擎，在备份数据库时就要使用全局锁的方法。

## 表级锁

### 表锁

如果我们想对学生表（t\_student）加表锁，可以使用下面的命令：

表级别的共享锁，也就是读锁；

lock tables t\_student read;

表级别的独占锁，也就是写锁；

lock tables t\_stuent wirte;

表锁除了会限制别的线程的读写外，也会限制本线程接下来的读写操作。

不过尽量避免在使用 InnoDB 引擎的表使用表锁，因为表锁的颗粒度太大，会影响并发性能

### 元数据锁

我们不需要显示的使用 MDL，因为当我们对数据库表进行操作时，会自动给这个表加上 MDL：

* 对一张表进行 CRUD 操作时，加的是 MDL 读锁；
* 对一张表做结构变更操作的时候，加的是 MDL 写锁；

MDL 是在事务提交后才会释放，这意味着事务执行期间，MDL 是一直持有的。

那如果数据库有一个长事务（所谓的长事务，就是开启了事务，但是一直还没提交），那在对表结构做变更操作的时候，可能会发生意想不到的事情，比如下面这个顺序的场景：

* 首先，线程 A 先启用了事务（但是一直不提交），然后执行一条 select 语句，此时就先对该表加上 MDL 读锁；
* 然后，线程 B 也执行了同样的 select 语句，此时并不会阻塞，因为「读读」并不冲突；
* 接着，线程 C 修改了表字段，此时由于线程 A 的事务并没有提交，也就是 MDL 读锁还在占用着，这时线程 C 就无法申请到 MDL 写锁，就会被阻塞，
* 那么在线程 C 阻塞后，后续有对该表的 select 语句，就都会被阻塞，如果此时有大量该表的 select 语句的请求到来，就会有大量的线程被阻塞住，这时数据库的线程很快就会爆满了。

为什么线程 C 因为申请不到 MDL 写锁，而导致后续的申请读锁的查询操作也会被阻塞？

* 这是因为申请 MDL 锁的操作会形成一个队列，队列中写锁获取优先级高于读锁，一旦出现 MDL 写锁等待，会阻塞后续该表的所有 CRUD 操作。
* 所以为了能安全的对表结构进行变更，在对表结构变更前，先要看看数据库中的长事务，是否有事务已经对表加上了 MDL 读锁，如果可以考虑 kill 掉这个长事务，然后再做表结构的变更。

### 意向锁

在使用 InnoDB 引擎的表里对某些记录加上「共享锁」之前，需要先在表级别加上一个「意向共享锁」；

在使用 InnoDB 引擎的表里对某些纪录加上「独占锁」之前，需要先在表级别加上一个「意向独占锁」；

意向共享锁和意向独占锁是表级锁，不会和行级的共享锁和独占锁发生冲突，而且意向锁之间也不会发生冲突，只会和共享表锁（lock tables … read）和独占表锁（lock tables … write）发生冲突。

如果没有「意向锁」，那么加「独占表锁」时，就需要遍历表里所有记录，查看是否有记录存在独占锁，这样效率会很慢。

那么有了「意向锁」，由于在对记录加独占锁前，先会加上表级别的意向独占锁，那么在加「独占表锁」时，直接查该表是否有意向独占锁，如果有就意味着表里已经有记录被加了独占锁，这样就不用去遍历表里的记录。

所以，意向锁的目的是为了快速判断表里是否有记录被加锁。

### AUTO-INC锁

AUTO-INC 锁是特殊的表锁机制，锁不是再一个事务提交后才释放，而是再执行完插入语句后就会立即释放。

在插入数据时，会加一个表级别的 AUTO-INC 锁，然后为被 AUTO\_INCREMENT 修饰的字段赋值递增的值，等插入语句执行完成后，才会把 AUTO-INC 锁释放掉。

## 行级锁

InnoDB 引擎是支持行级锁的，而 MyISAM 引擎并不支持行级锁。

### Record Lock

记录锁，也就是仅仅把一条记录锁上；

### Gap Lock

间隙锁是一种加在两个索引之间的锁，或者加在第一个索引之前，或最后一个索引之后的间隙。间隙锁只阻止其他事务插入到间隙中，他们不阻止其他事务在同一个间隙上获得间隙锁。（间隙锁和插入意向锁冲突）

### Next-Key Lock

Next-key锁是记录和间隙锁的组合，指的是加在某条记录以及这条记录前面间隙上的锁。

# next-key lock的加锁规则

**加锁的基本单位是 next-key lock**，它是由记录锁和间隙锁组合而成的，**next-key lock 是前开后闭区间，而间隙锁是前开后开区间**。但是，next-key lock 在一些场景下会退化成记录锁或间隙锁。**我的 MySQL 的版本是 8.0.26，不同版本的加锁规则可能是不同的**

## **唯一索引等值查询**

当我们用唯一索引进行等值查询的时候，查询的记录存不存在，加锁的规则也会不同：

**当查询的记录是存在的，在用「唯一索引进行等值查询」时，next-key lock 会退化成「记录锁」**。

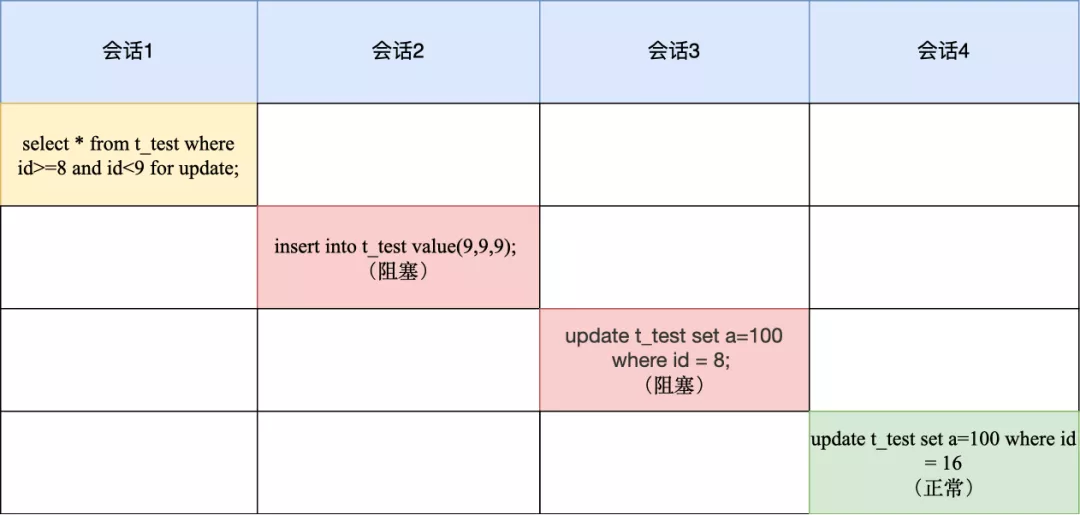
**当查询的记录是不存在的，在用「唯一索引进行等值查询」时，next-key lock 会退化成「间隙锁」**。

## **唯一索引范围查询**

举个例子，下面这两条查询语句，查询的结果虽然是一样的，但是加锁的范围是不一样的。

 select \* from t\_test where id=8 for update;  
 select \* from t\_test where id>=8 and id<9 for update;

做个实验就知道了。



会话 1 加锁变化过程如下：

1. 最开始要找的第一行是 id = 8，因此 next-key lock(4,8]，但是由于 id 是唯一索引，且该记录是存在的，因此会退化成记录锁，也就是只会对 id = 8 这一行加锁；
2. 由于是范围查找，就会继续往后找存在的记录，也就是会找到 id = 16 这一行停下来，然后加 next-key lock (8, 16]，但由于 id = 16 不满足 id < 9，所以会退化成间隙锁，加锁范围变为 (8, 16)。

所以，会话 1 这时候主键索引的锁是记录锁 id=8 和间隙锁(8, 16)。

会话 2 由于往间隙锁里插入了 id = 9 的记录，所以会被锁住了，而 id = 8 是被加锁的，因此会话 3 的语句也会被阻塞。

由于 id = 16 并没有加锁，所以会话 4 是可以正常被执行。

## **非唯一索引等值查询**

当查询的记录存在时，除了会加 next-key lock 外，还额外加间隙锁，规则是向下遍历到第一个不符合条件的值才能停止，也就是会加两把锁。

当查询的记录不存在时，只会加 next-key lock，然后会退化为间隙锁，也就是只会加一把锁。

## **非唯一索引范围查询**

非唯一索引和主键索引的范围查询的加锁也有所不同，不同之处在于**普通索引范围查询，next-key lock 不会退化为间隙锁和记录锁，所以只保留next-key lock的锁不变**。

来看下面这个案例：



会话 1 加锁变化过程如下：

1. 最开始要找的第一行是 b = 8，因此 next-key lock(4,8]，但是由于 b 不是唯一索引，并不会退化成记录锁。
2. 但是由于是范围查找，就会继续往后找存在的记录，也就是会找到 b = 16 这一行停下来，然后加 next-key lock (8, 16]，因为是普通索引查询，所以并不会退化成间隙锁。

所以，会话 1 的普通索引 b 有两个 next-key lock，分别是 (4,8] 和(8, 16]。这样，你就明白为什么会话 2 、会话 3 、会话 4 的语句都会被锁住了。

# MySQL死锁案例

## 初始准备

//id是自增主键，name是非唯一索引，balance普通字段

| **id** | **name** | **balance** |
| --- | --- | --- |
| 1 | Eason | 100 |
| 2 | William | 100 |

CREATE TABLE `account`  (

  `id`  int(11) NOT NULL AUTO\_INCREMENT,

  `name` varchar(255) DEFAULT NULL,

  `balance`  int(11) DEFAULT NULL,

   PRIMARY KEY (`id`),

   KEY `idx\_name`  (`name`) USING BTREE

) ENGINE=InnoDB AUTO\_INCREMENT=3 DEFAULT CHARSET=utf8;

## 模拟并发

开启两个终端模拟事务并发情况，执行顺序以及实验现象如下：



1）事务A执行更新操作，更新成功

mysql> update account set balance =1000  where name ='William';

Query OK,  1 row affected (0.01 sec)

2）事务B执行更新操作，更新成功

mysql> update account set balance =1000  where name ='Eason';

Query OK,  1 row affected (0.01 sec)

3）事务A执行插入操作，陷入阻塞

mysql> insert into account values(null,'Juliet',100);

**这个时候该操作被阻塞，无法执行。**

4）事务B执行插入操作，插入成功，同时事务A的插入由阻塞变为死锁error。

mysql> insert into account values(null,'Zack',100);

Query OK,  1 row affected (0.01 sec)



## 分析加锁情况

**事务A中update语句的加锁情况**

update account set balance =1000 where name ='William';

**事务A执行完update更新语句，会持有锁：**

* Next-key Lock：(E, W]
* Gap Lock ：(W, +∞)

**事务A中insert语句的加锁情况**

insert into account values(null,'Juliet',100);

**间隙锁：**

* 因为Juliet(J在E和W之间)，所以需要请求加(E,W)的间隙锁

**插入意向锁（Insert Intention）**

* 插入意向锁是在插入一行记录操作之前设置的一种间隙锁，这个锁释放了一种插入方式的信号，即事务A需要插入意向锁(E,W)

**因此**，事务A的update语句和insert语句执行完，它是持有了 **(E, W]的 Next-Key锁**，**(W, +∞)的Gap锁**，想拿到 **(E,W)的插入意向排它锁**

**事务B中update语句的加锁情况**

update account set balance =1000 where name ='Eason';

**事务B执行完update更新语句，会持有锁：**

* Next-key Lock：(-∞, E]
* Gap Lock ：(E, W)

**事务B中insert语句的加锁情况**

insert into account values(null,'Zack',100);

**间隙锁：**

* 因为Zack(Y在W之后)，所以需要请求加(W,+∞)的间隙锁

**插入意向锁（Insert Intention）**

* 插入意向锁是在插入一行记录操作之前设置的一种间隙锁，这个锁释放了一种插入方式的信号，即事务A需要插入意向锁(W,+∞)

**所以**，事务B的update语句和insert语句执行完，它是持有了 **（-∞，E]的 Next-Key锁**，**（E，W）的Gap锁**，想拿到 **(W,+∞)的间隙锁，即插入意向排它锁**

## 总结

接下来呢，让我们一起还原死锁真相吧

* 事务A执行完Update William的语句，持有（E，W]的Next-key Lock，（W，+∞）的Gap Lock ，插入成功~
* 事务B执行完Update Eason语句，持有（-∞，E]的 Next-Key Lock，（E，W）的Gap Lock，插入成功~
* 事务A执行Insert Juliet的语句时，因为需要（E，W）的插入意向锁，但是（E，W）在事务B怀里，所以它陷入心塞~
* 事务B执行Insert Zack的语句时，因为需要(W,+∞) 的插入意向锁，但是(W,+∞) 在事务A怀里，所以它也陷入心塞。
* 事务A持有（W，+∞）的Gap Lock，在等待（E，W）的插入意向锁，事务B持有（E，W）的Gap锁，在等待(W,+∞) 的插入意向锁，所以形成了死锁的闭环（Gap锁与插入意向锁会冲突的）
* 事务A,B形成了死锁闭环后，因为InnoDB的底层机制，它会让其中一个事务让出资源，另外的事务执行成功，这就是为什么你最后看到事务B插入成功了，但是事务A的插入显示了Deadlock found ~

# 慢查询日志

1. 记录所有执行时间超过long\_query\_time秒的所有查询或不适用于索引的查询
2. long\_query\_time默认时间是10秒。即超过10秒的查询都认为是慢查询。
3. 慢查询日志默认名称：主机名-slow.log

# 慢SQL的原因

1. 无索引或者索引失效

- [索引失效的情况](https://www.chenmx.net/?p=316):https://www.chenmx.net/?p=316

2. 锁等待

* + 行锁是基于索引加的锁，如果我们在更新操作时，索引失效，行锁也会升级为表锁。
  + 除了锁升级还要注意死锁。

MyISAM最小支持表锁，所以不讨论了。InnoDB最小支持行锁，但也要注意锁升级。

当一张表使用大量行锁，会导致失误执行率下降，匆匆而可能造成其它事务长时间锁

等待和更多的锁冲突发生，致使性能严重下降，更有可能会将行锁升级为表锁。

3. 不恰当的`SQL`语句

* + 例如`select \*`操作一般都需要回表，除非`where`条件是主键。
  + 对非索引字段进行排序。

4. 在大数据表中使用`<LIMIT M,N>`分页查询。

# Explain命令

在日常工作中，我们会有时会开慢查询去记录一些执行时间比较久的SQL语句，找出这些SQL语句并不意味着完事了，些时我们常常用到explain这个命令来查看一个这些SQL语句的执行计划，查看该SQL语句有没有使用上了索引，有没有做全表扫描，这都可以通过explain命令来查看。所以我们深入了解MySQL的基于开销的优化器，还可以获得很多可能被优化器考虑到的访问策略的细节，以及当运行SQL语句时哪种策略预计会被优化器采用。

 -- 实际SQL，查找用户名为Jefabc的员工  
 select \* from emp where name = 'Jefabc';  
 -- 查看SQL是否使用索引，前面加上explain即可  
 explain select \* from emp where name = 'Jefabc';

img

expain出来的信息有10列，分别是id、select\_type、table、type、possible\_keys、key、key\_len、ref、rows、Extra

**概要描述：** id:选择标识符 select\_type:表示查询的类型。 table:输出结果集的表 partitions:匹配的分区 type:表示表的连接类型 possible\_keys:表示查询时，可能使用的索引 key:表示实际使用的索引 key\_len:索引字段的长度 ref:列与索引的比较 rows:扫描出的行数(估算的行数) filtered:按表条件过滤的行百分比 Extra:执行情况的描述和说明

**下面对这些字段出现的可能进行解释：**

一、 **id**

SELECT识别符。这是SELECT的查询序列号

**我的理解是SQL执行的顺序的标识，SQL从大到小的执行**

1. id相同时，执行顺序由上至下
2. 如果是子查询，id的序号会递增，id值越大优先级越高，越先被执行
3. id如果相同，可以认为是一组，从上往下顺序执行；在所有组中，id值越大，优先级越高，越先执行

 -- 查看在研发部并且名字以Jef开头的员工，经典查询  
 explain select e.no, e.name from emp e left join dept d on e.dept\_no = d.no where e.name like 'Jef%' and d.name = '研发部';

img

**二、select\_type**

***\*示查询中每个select子句的类型\****

(1) SIMPLE(简单SELECT，不使用UNION或子查询等)

(2) PRIMARY(子查询中最外层查询，查询中若包含任何复杂的子部分，最外层的select被标记为PRIMARY)

(3) UNION(UNION中的第二个或后面的SELECT语句)

(4) DEPENDENT UNION(UNION中的第二个或后面的SELECT语句，取决于外面的查询)

(5) UNION RESULT(UNION的结果，union语句中第二个select开始后面所有select)

(6) SUBQUERY(子查询中的第一个SELECT，结果不依赖于外部查询)

(7) DEPENDENT SUBQUERY(子查询中的第一个SELECT，依赖于外部查询)

(8) DERIVED(派生表的SELECT, FROM子句的子查询)

(9) UNCACHEABLE SUBQUERY(一个子查询的结果不能被缓存，必须重新评估外链接的第一行)

**三、table**

显示这一步所访问数据库中表名称（显示这一行的数据是关于哪张表的），有时不是真实的表名字，可能是简称，例如上面的e，d，也可能是第几步执行的结果的简称

**四、type**

对表访问方式，表示MySQL在表中找到所需行的方式，又称“访问类型”。

常用的类型有： **ALL、index、range、 ref、eq\_ref、const、system、\***\*NULL（从左到右，性能从差到好）\*\*

ALL：Full Table Scan， MySQL将遍历全表以找到匹配的行

index: Full Index Scan，index与ALL区别为index类型只遍历索引树

range:只检索给定范围的行，使用一个索引来选择行

ref: 表示上述表的连接匹配条件，即哪些列或常量被用于查找索引列上的值

eq\_ref: 类似ref，区别就在使用的索引是唯一索引，对于每个索引键值，表中只有一条记录匹配，简单来说，就是多表连接中使用primary key或者 unique key作为关联条件

const、system: 当MySQL对查询某部分进行优化，并转换为一个常量时，使用这些类型访问。如将主键置于where列表中，MySQL就能将该查询转换为一个常量，system是const类型的特例，当查询的表只有一行的情况下，使用system

NULL: MySQL在优化过程中分解语句，执行时甚至不用访问表或索引，例如从一个索引列里选取最小值可以通过单独索引查找完成。

**五、possible\_keys**

**指出MySQL能使用哪个索引在表中找到记录，查询涉及到的字段上若存在索引，则该索引将被列出，但不一定被查询使用（该查询可以利用的索引，如果没有任何索引显示 null）**

该列完全独立于EXPLAIN输出所示的表的次序。这意味着在possible\_keys中的某些键实际上不能按生成的表次序使用。 如果该列是NULL，则没有相关的索引。在这种情况下，可以通过检查WHERE子句看是否它引用某些列或适合索引的列来提高你的查询性能。如果是这样，创造一个适当的索引并且再次用EXPLAIN检查查询

**六、Key**

**key列显示MySQL实际决定使用的键（索引），必然包含在possible\_keys中**

如果没有选择索引，键是NULL。要想强制MySQL使用或忽视possible\_keys列中的索引，在查询中使用FORCE INDEX、USE INDEX或者IGNORE INDEX。

**七、key\_len**

**表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度（key\_len显示的值为索引字段的最大可能长度，并非实际使用长度，即key\_len是根据表定义计算而得，不是通过表内检索出的）**

不损失精确性的情况下，长度越短越好

**八、ref**

**列与索引的比较，表示上述表的连接匹配条件，即哪些列或常量被用于查找索引列上的值**

**九、rows**

**估算出结果集行数，表示MySQL根据表统计信息及索引选用情况，估算的找到所需的记录所需要读取的行数**

**十、Extra**

**该列包含MySQL解决查询的详细信息,有以下几种情况：**

Using where:不用读取表中所有信息，仅通过索引就可以获取所需数据，这发生在对表的全部的请求列都是同一个索引的部分的时候，表示mysql服务器将在存储引擎检索行后再进行过滤

Using temporary：表示MySQL需要使用临时表来存储结果集，常见于排序和分组查询，常见 group by ; order by

Using filesort：当Query中包含 order by 操作，而且无法利用索引完成的排序操作称为“文件排序”

 -- 测试Extra的filesort  
 explain select \* from emp order by name;

Using join buffer：改值强调了在获取连接条件时没有使用索引，并且需要连接缓冲区来存储中间结果。如果出现了这个值，那应该注意，根据查询的具体情况可能需要添加索引来改进能。

Impossible where：这个值强调了where语句会导致没有符合条件的行（通过收集统计信息不可能存在结果）。

Select tables optimized away：这个值意味着仅通过使用索引，优化器可能仅从聚合函数结果中返回一行

No tables used：Query语句中使用from dual 或不含任何from子句

 -- explain select now() from dual;

**总结：\***\* • EXPLAIN不会告诉你关于触发器、存储过程的信息或用户自定义函数对查询的影响情况 • EXPLAIN不考虑各种Cache • EXPLAIN不能显示MySQL在执行查询时所作的优化工作 • 部分统计信息是估算的，并非精确值 • EXPALIN只能解释SELECT操作，其他操作要重写为SELECT后查看执行计划。\*\*

通过收集统计信息不可能存在结果