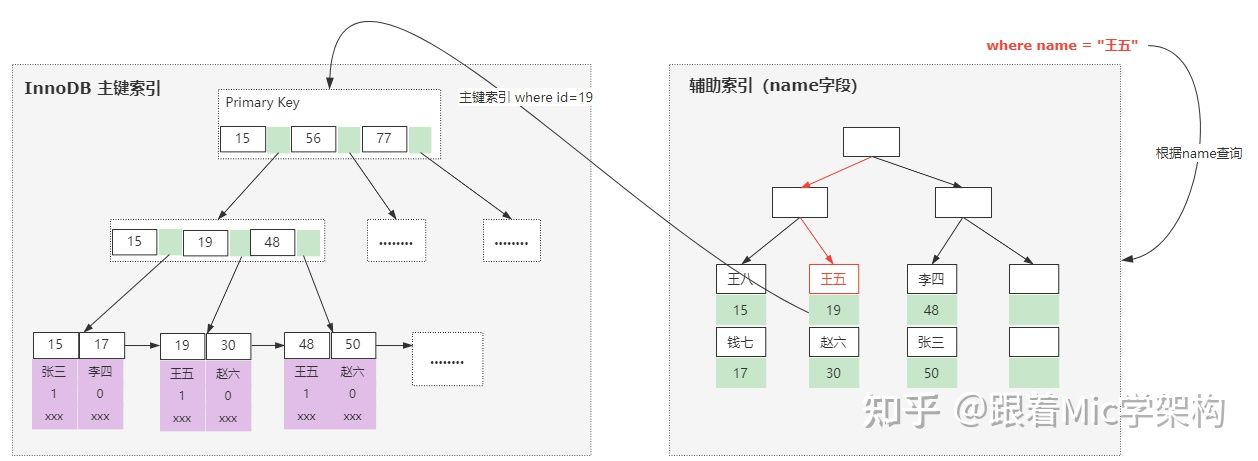
# 什么是聚集索引和非聚集索引？

1. 简单来说，聚集索引就是基于主键创建的索引，除了主键索引以外的其他索引，称为非聚集索引，也叫做[二级索引](https://www.zhihu.com/search?q=%E4%BA%8C%E7%BA%A7%E7%B4%A2%E5%BC%95&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A2475915157%7D)。
2. 由于在InnoDB引擎里面，一张表的数据对应的物理文件本身就是按照B+树来组织的一种索引结构，而聚集索引就是按照每张表的主键来构建一棵B+树，然后叶子节点里面存储了这个表的每一行数据记录。
3. 所以基于InnoDB这样的特性，聚集索引并不仅仅是一种索引类型，还代表着一种数据的存储方式。
4. 同时也意味着每个表里面必须要有一个主键，如果没有主键，InnoDB会默认选择或者添加一个隐藏列作为主键索引来存储这个表的数据行。一般情况是建议使用自增id作为主键，这样的话id本身具有连续性使得对应的数据也会按照顺序存储在磁盘上，写入性能和检索性能都很高。否则，如果使用uuid这种随机id，那么在频繁插入数据的时候，就会导致随机磁盘IO，从而导致性能较低。
5. 需要注意的是，InnoDB里面只能存在一个聚集索引，原因很简单，如果存在多个聚集索引，那么意味着这个表里面的数据存在多个副本，造成磁盘空间的浪费，以及数据维护的困难。
6. 由于在InnoDB里面，主键索引表示的是一种数据存储结构，所以如果是基于非聚集索引来查询一条完整的记录，最终还是需要访问主键索引来检索。



# 什么是索引？索引的优缺点？

索引是提高数据库查找效率的数据结构。

优点：

* 可以提高数据库查找效率。
* 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。
* 使用顺序索引在进行范围查找时效率很高。

缺点：

* 创建索引和维护索引耗费空间和时间，且随数据量的增大而增大。
* 在数据库进行插入、删除、修改操作时索引需要动态维护，耗费时间。

# 顺序索引和散列索引的区别？

索引是提高数据库查找效率的数据结构，分为顺序索引和散列索引。

* 顺序索引中的记录基于搜索码值顺序排序，常用的数据结构为B+树。
* 散列索引中的记录基于搜索码值的散列函数的值平均、随机地分布在若干个散列桶中。

散列索引缺点：不适合进行范围查找。

# 非聚集索引范围查找效率如何？

非聚集索引查找需要先找到主键，然后再根据主键查询数据，但范围内的主键很可能不相邻，所以非聚集索引范围查找效率不高。

# 数据库索引用B+树不用平衡二叉树（红黑）

* **磁盘IO空间浪费**：在AVL和红黑树的查找过程中，需要从磁盘读取每个节点的数据，但是操作系统从磁盘读入内存是按页读取的，如果使用AVL或红黑树，每个节点只保存一个key值，每次读取会比较浪费，从而意味着需要更多的IO操作，也更加费时。
* **范围索引和遍历更加方便**：B+树的叶子结点都是相链的，因此对整棵树的遍历只需要一次线性遍历叶子结点即可。而且由于叶子结点上数据顺序排列并且相连，所以便于区间查找和搜索。而AVL和红黑树则需要进行每一层的递归遍历。相邻的元素可能在内存中不相邻，所以缓存命中性没有B+树好。

# 数据库索引用B+树不用B树

* **IO次数更少：**由于B+树在内部节点上不包含数据信息，因此在内存页中能够存放更多的key。 数据存放的更加紧密，具有更好的空间局部性。因此访问叶子节点上关联的数据也具有更好的缓存命中率。
* **范围索引和遍历更加方便**：B+树的叶子结点都是相链的，因此对整棵树的遍历只需要一次线性遍历叶子结点即可。而且由于叶子结点上数据顺序排列并且相连，所以便于区间查找和搜索。而B树则需要进行每一层的递归遍历。相邻的元素可能在内存中不相邻，所以缓存命中性没有B+树好。

# 索引下推Index Condition Pushdown(ICP)

简单来说，索引条件下推ICP就是尽可量利用二级索引筛除不符合where条件的记录，减少回表的次数。

例：一张表拥有联合索引(a, b) 使用select \* from table where a=1 and b like ‘%d’。如果不使用索引下推，那么根据最左匹配原则只能使用索引a进行筛选，使用索引下推，则b也能按‘%d’过滤。

没有使用ICP的情况下，MySQL查询过程如下：

* 存储引擎读取索引记录；
* 根据索引中的主键值，定位并读取完整的行记录；
* 存储引擎把记录交给Server层去检测该记录是否满足WHERE条件。

使用ICP的情况下，查询过程如下：

* 读取索引记录（不是完整的行记录）；
* 判断WHERE条件部分能否用索引中的列筛选；
* 条件满足，使用索引中的主键去定位并读取完整的行记录（就是所谓的回表）；
* 存储引擎把记录交给Server层，Server层检测该记录是否满足WHERE条件的其余部分。

# Multi-Range Read(MRR)

Multi-Range Read是MySQL5.6的优化措施，目的是为了减少磁盘的随机IO访问，并且将随机访问转化为顺序的数据访问，所以MRR优化措施对IO-bound型的SQL查询语句可能带来极大的性能提升。简单来说就是二级索引查询到的主键是随机顺序的，先将主键进行排序，然后再通过聚集索引查询数据。

工作方式：

1. 将查询得到的辅助索引键值存放于缓存之中，注意，这时候缓存中的数据是根据辅助索引的键值排序的。
2. 将缓存中的数据根据row ID(主键)进行重排序。
3. 然后再根据row ID(主键)的顺序去访问。

# 索引合并INDEX MERGE

简单来说，索引合并就是对多个索引分别进行条件扫描，然后将它们各自的结果进行合并。

这种算法一般用于以下三种情况：

* or条件的并集（union 或者 union all）
* and条件的交集
* 综合前面两种情况

# 覆盖索引

回表指的就是非聚集索引从叶子节点拿到数据(主键的键值)之后，还需要再根据主键键值去扫描主键索引的B+树，这种操作就叫做回表，也就是说他需要扫描两颗B+树，这也就是为什么在InnoDB中主键索引的效率相比较其他索引是最高的。（因为其他索引得到主键键值后还需要扫描聚集索引）。

前面我们说到了回表操作，那么就还有这么一种场景是不需要回表的：比如说我们一个查询只需要查询当前索引的值和主键的值，而不需要查其他数据，这时候就不需要回表了，直接就可以返回，这种也称之为**覆盖索引。**

# ****事务的隔离级别****

## 脏读

事务A可以读取到事务B未提交的数据就是脏读，如果事务B回滚，那么事务读取的数据就是错误的。

## 不可重复读

事务A前后两次读取同一条记录，事务B在这两次读取之间提交了这个记录的修改，那么事务A两次读到的同一条记录就是不一样的。不可重复读注重同一条记录的修改和更新。

## 幻读

事务A进行了一个范围查询，假设第一次只能查出一条记录，这时候事务B插入了一条记录并提交，然后事务A再次执行同一个查询，这时候就能查出来两条记录。幻读注重查询范围内记录的插入和删除。

## Read Uncommitted(未提交读)

可能出现脏读、不可重复读、幻读。

## Read Committed(已提交读)

不可能出现脏读，可能出现不可重复读、幻读。

## Repeatable Read(可重复读)

不可能出现脏读、不可重复读，可能出现幻读。InnoDB默认的隔离级别是可重复读，并通过临键锁解决了幻读问题。

## Serializable(串行化)

这是最高的隔离级别。它要求事务序列化执行，事务只能一个接着一个地执行，不能并发执行。在这个级别，可以解决上面提到的所有并发问题，但可能导致大量的超时现象和锁竞争，通常不会用这个隔离级别。

# LBCC

LBCC，基于锁的并发控制，英文全称Lock Based Concurrency Control。这种方案比较简单粗暴，就是一个事务去读取一条数据的时候，就上锁，不允许其他事务来操作(当然这个锁的实现也比较重要，如果我们只锁定当前一条数据依然无法解决幻读问题)。

## 当前读

这个概念其实很好理解，MySQL加锁之后就是当前读。假如当前事务只是加共享锁，那么其他事务就不能有排他锁，也就是不能修改数据；而假如当前事务需要加排他锁，那么其他事务就不能持有任何锁。总而言之，能加锁成功，就确保了除了当前事务之外，其他事务不会对当前数据产生影响，所以自然而然的，当前事务读取到的数据就只能是最新的，而不会是快照数据(后文MVCC会解释快照读概念)。

LBCC方案中，如果我们的业务系统是读多写少的话，这种方案就会极大影响了效率，所以我们就有了另一种解决方案：MVCC。

# MVCC

MVCC,多版本的并发控制，英文全称：Multi Version Concurrency Control。就是当我们在修改数据的时候，可以为这条数据创建一个快照，后面就可以直接读取这个快照。

为了实现MVCC机制，InnoDB内部为每一行添加了两个隐藏列：DB\_TRX\_ID和DB\_ROLL\_PTR。DB\_TRX\_ID存储了插入或更新语句的最近事务的ID，DB\_ROLL\_PTR是回滚指针，回滚指针指向记录修改前的undo log记录，读取记录的时候会根据指针去读取undo log中的记录。

## MVCC查询两大规则

1. 如果DB\_TRX\_ID小于等于当前事务id的数据说明该数据在当前事务之前修改，则直接读取。
2. 如果DB\_TRX\_ID大于当前事务id的数据说明该数据在当前事务之后修改，则读取其DB\_TRX\_ID回滚指针里DB\_TRX\_ID小于等于当前事务id的数据。

# 锁

## 定义

锁是一种用于保证在并发场景下每个事务仍能以一致性的方式读取和修改数据的方式，当一个事务对某一条数据上锁之后，其他事务就不能修改或者只能阻塞等待锁的释放，所以锁的粒度大小一定程度上可以影响到访问数据库的性能。

## 全局锁

FLUSH TABLE WITH READ LOCK;--加全局锁

UNLOCK TABLES;-- 解锁

执行这个语句之后整个数据库都只允许读，不允许写。

## 表锁

1. 通过LOCK命令加锁

LOCK TABLE t1 READ;共享锁，此时本线程只能读t1表，其他线程也只能读t1表

LOCK TABLE t2 WRITE;排他锁，此时本线程能读写t2表，其他线程不能读写t2表

UNLOCK TABLE; --解锁

1. MDL锁

MDL（metadata lock）即元数据锁，主要是用来保证读写的正确性。

当对一个表做增删改查操作的时候，加MDL读锁（事务提交后自动释放）。

当对一个表结构做变更操作时，加MDL写锁（事务提交后自动释放）。

MDL读锁之间不互斥，可以对一张表加多个MDL读锁，读写和写锁之间互斥。

PS：MDL锁是在Server层实现的，所以所有引擎都具有MDL锁。

## 通过LOCK加锁和MDL锁的区别

通过LOCK加锁不仅限制了其他线程，也限制自己线程只能操作加锁的表。

## 行锁

行锁，从名字上来看，就是锁住一行数据，然而，行锁的实际实现算法会相对复杂，有时候并不仅仅只是锁住某一条数据，这个后面再展开。

正常的思路是：锁住一行数据之后，其他事务就不能来访问这条数据了，那么我们想象，假如事务A访问了一条数据，只是拿出来读一下，并不想去修改，正好事务B也来访问这条数据，也仅仅只是想拿出来读一下，并不想去修改，这时候如果因此阻塞了，就有点浪费性能了。所以为了优化这种读数据的场景，我们又把行锁分为了两大类型：共享锁和排他锁。

### 共享锁

共享锁，Shared Lock，又称之为读锁，S锁，就是说一条数据被加了S锁之后，其他事务也能来读数据，可以共享一把锁。

我们可以通过如下语句加共享锁：

select \* from test where id=1 LOCK IN SHARE MODE;

加锁之后，直到加锁的事务结束(提交或者回滚)就会释放锁。

### 排他锁

排他锁，Exclusive Lock，又称之为写锁，X锁。就是说一条数据被加了X锁之后，其他事务想来访问这条数据只能阻塞等待锁的释放，具有排他性。

当我们在修改数据，如：insert,update,delete的时候MySQL就会自动加上排他锁，同样的，我们可以通过如下sql语句手动加上排他锁：

select \* from test where id=1 for update;

在InnoDB引擎中，是允许行锁和表锁共存的。

但是这样就会有一个问题，假如事务A给t表其中一行数据上锁了，这时候事务B想给t表上一个表锁，这时候怎么办呢？事务B怎么知道t表有没有行锁的存在，如果采用全表遍历的情况，当表中的数据很大的话，加锁都要加半天，所以MySQL中就又引入了意向锁。

### 锁锁的到底是什么？

InnoDB引擎中，锁锁的是索引：

* 假如一张表没有索引，MySQL会进行锁表(其实锁住的是隐藏列ROWID的主键索引)。
* 假如我们对辅助索引加锁，那么辅助索引所对应的主键索引也会被锁住。
* 主键索引被锁住，实际上就等于是整条记录都被锁住了(主键索引叶子节点存储了整条数据)。

**在mysql中，行锁又衍生了其他几种算法锁，分别是记录锁、间隙锁、临键锁，这3个锁为了解决当前读的不可重复读和幻读问题，因为快照读不会读取当前事务启动后其他事务改变的记录。**

### 记录锁（Record Lock）

记录锁解决的是当前读的不可重复读问题。记录锁锁的是表中的某一条记录，记录锁的出现条件必须是精准命中索引并且索引是唯一索引，如主键id，记录锁在RC和RR隔离级别中都存在，因为需要保证记录的可重复读。

### 间隙锁（Gap Lock）和临键锁（Next-Key Lock）

间隙锁和临键锁解决的是当前读的幻读问题。间隙锁锁的是一个区间，临键锁锁的是一条记录及其两侧的区间。间隙锁和临键锁只在RR隔离级别中存在。

1. 没有索引

没有索引的情况下，不管是等号查询还是范围查询，整张表的每行都会加X锁（排他锁），并且每个区间都会加间隙锁，这种情况相当于对所有记录加了临键锁，锁住了整个表。

1. 普通索引

不管是等号查询还是范围查询，如果查询到符合条件的记录，对这些记录及其两侧区间加临键锁。如果没有查询到，对查找结果所在区间加间隙锁。

1. 唯一索引
   1. 等号查询

如果查询到符合条件的记录对这条记录加记录锁，不加间隙锁，如果没有查到，将对查找结果所在区间加间隙锁。

* 1. 范围查询

如果查询到符合条件的记录，对这些记录及其两侧区间加临键锁。如果没有查询到，对两侧区间加间隙锁。

* 间隙锁和临键锁的共同点：都锁住了区间。临键锁=记录锁+间隙锁。
* 间隙锁和临键锁的区别：间隙锁在没有查询到记录时使用，临键锁在查询到记录时使用。

**临键锁在以下两个条件时会降级成为间隙锁或者记录锁：**

* 当查询未命中任务记录时，会降级为间隙锁。
* 当使用主键或者唯一索引命中了一条记录时，会降级为记录锁。

# 死锁

## 死锁的检测

目前数据库大部分采用wait-for graph（等待图）的方式来进行死锁检测，InnoDB引擎也是采用这种方式来检测死锁。数据库中会记录两种信息：

* 锁的信息链表
* 事务的等待链表

wait-for graph 算法会根据这两个信息构建一张图，当图中存在回路，则证明存在死锁。

## 死锁的避免

* 尽量将长事务拆分成多个小事务。
* 查询时避免没有where条件语句查询，并尽可能使用索引查询。
* 可以的话尽量使用等值查询。

# Select语句的执行流程

一条select查询大致经过以下六个步骤：

1. 客户端发起一个请求时，首先会建立一个连接
2. 服务端会检查缓存，如果命中则直接返回，否则继续之后后面步骤
3. 服务器端根据收到的sql语句进行解析，然后对其进行词法分析，语法分析以及预处理
4. 由优化器生成执行计划
5. 调用存储引擎层API来执行查询
6. 返回查询到的结果

## 建立连接

MySQL服务端和客户端的通信方式采用的是半双工协议。半双工指的是服务器和客户端可以双向通信，但不能同时发送消息，单工指的是单向通信，全双工是双向通信，且可以同时发送消息。

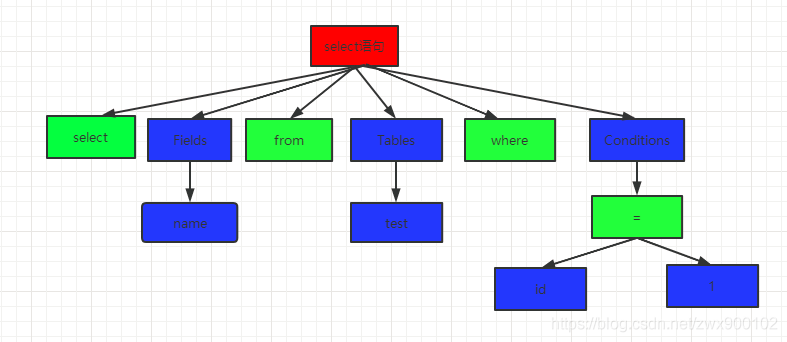
## 查询缓存

MySQL的缓存使用条件非常苛刻，查询语句必须与缓存中完全一致。而且一旦表里面有一行数据变动了，那么关于这种表的所有缓存都会失效。所以一般我们都是不建议使用缓存，MySQL最新的8.0版本已经将缓存模块去掉了。

## 解析器和预处理器

### 词法解析和语法解析(Parser)

这一步主要的工作就是检查sql语句的语法对不对，在这里，首先会把我们整个SQL语句打碎，比如：select name from test where id=1，就会被打散成select, name, from, test, where, id, =, 1 这8个字符，并且能识别出关键字和非关键字，然后根据sql语句生成一个数据结构，也叫做解析树(select\_lex)。



### 预处理器(Preprocessor)

经过了前面的词法和语法解析，那么至少我们一条sql语句的语法格式是满足要求了，接下来我们还需要做什么呢？自然是检查表名，列名以及其他一些信息等是不是真实存在的，**预处理就是做一个表名和字段名等相关信息合法性的检测**。

### 查询优化器(Query Optimizer)

经过上面的步骤，到这里就得到了一句有效的sql语句了。而对一个查询语句，尤其是复杂的多表查询语句，我们可以有很多种执行方式，每种执行方式的效率也不一样，所以这时候就需要查询优化器去选择一种它认为最高效的执行方式。MySQL 里面使用的是基于开销（cost）的优化器，哪种执行计划开销最小，就选择哪种。

### 存储引擎查询

当Server层得到了一条sql语句的执行计划后，这时候就会去调用存储引擎层对应的API，执行查询（在B+树中进行搜索）。

### 返回结果

最后，将查询出得到的结果返回Server层，如果开启了缓存，Server层返回数据的同时还会写入缓存。MySQL将查询结果返回是一个增量的逐步返回过程。例如：当我们处理完所有查询逻辑并开始执行查询并且生成第一条结果数据的时候，MySQL就可以开始逐步的向客户端传输数据了。

# SQL语句

以前4个字符创建前缀索引

alter table x\_test add index(x\_name(4));

创建全文索引

alter table article where fulltext index title\_content\_index (title, content);

全文索引查询

SELECT \* FROM article WHERE MATCH(title, content) AGAINST (‘查询字符串’);