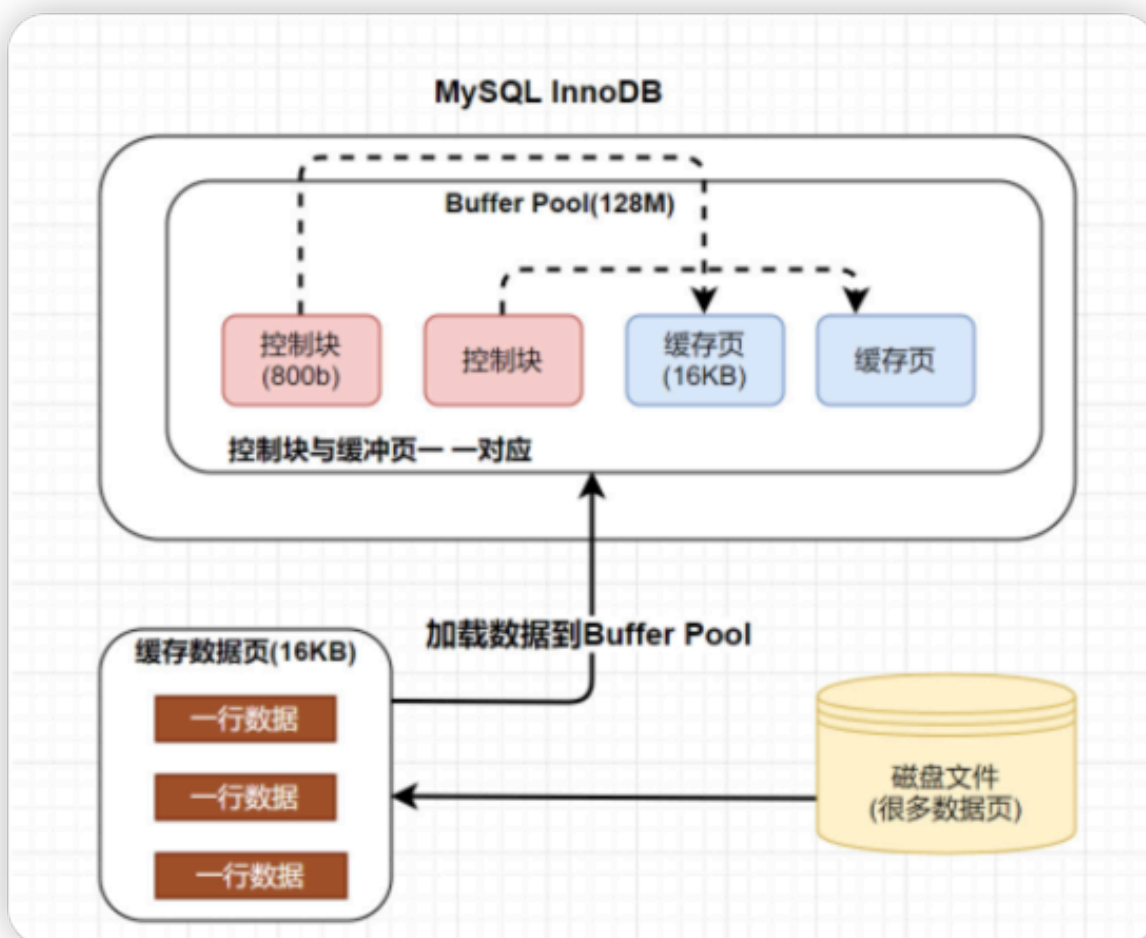
1.什么是BufferPool?

Buffer Pool基本概念

Buffer Pool：缓冲池，简称BP。其作用是用来缓存表数据与索引数据，减少磁盘IO操作，提升效率。

Buffer Pool由**缓存数据页(Page)** 和 对缓存数据页进行描述的**控制块** 组成, 控制块中存储着对应缓存页的所属的 表空间、数据页的编号、以及对应缓存页在Buffer Pool中的地址等信息.

Buffer Pool默认大小是128M, 以Page页为单位，Page页默认大小16K，而控制块的大小约为数据页的5%，大 概是800字节。



> 注: Buffer Pool大小为128M指的就是缓存页的大小，控制块则一般占5%，所以每次会多申请6M的内存空间用于存放控制块

如何判断一个页是否在BufferPool中缓存？

MySQl中有一个哈希表数据结构，它使用表空间号+数据页号，作为一个key，然后缓冲页对应的控制块作为value。

数据页缓存的Hash表

KEY	VALUE
表空间号+数据页号	对应控制块
表空间号+数据页号	对应控制块
....	....

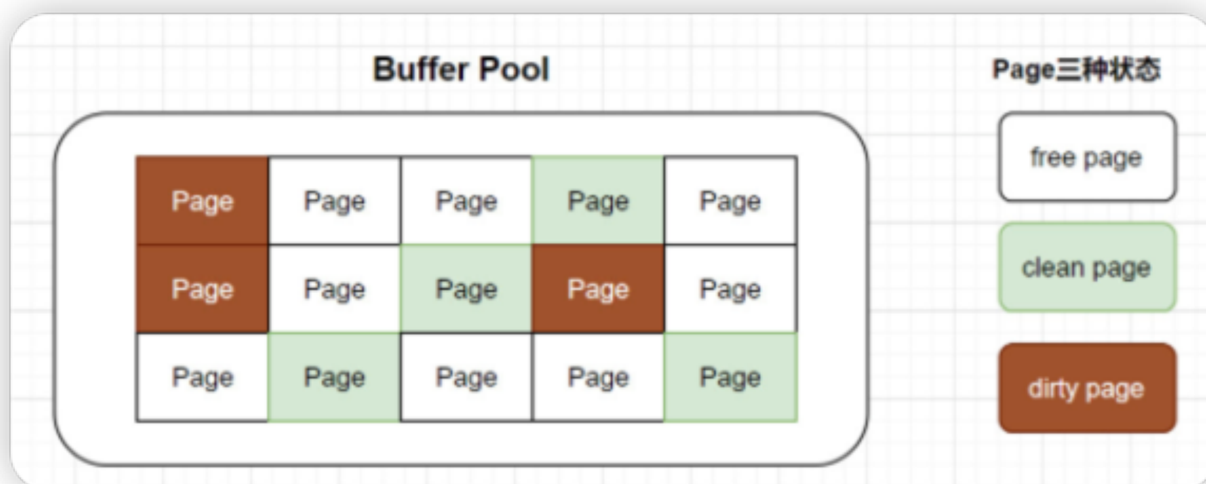
- **当需要访问某个页的数据时，先从哈希表中根据表空间号+页号看看是否存在对应的缓冲页。**
- **如果有，则直接使用；如果没有，就从free链表中选出一个空闲的缓冲页，然后把磁盘中对应的页加载到该缓冲页的位置**

2.InnoDB如何管理Page页?

Page页分类

BP的底层采用链表数据结构管理Page。在InnoDB访问表记录和索引时会在Page页中缓存，以后使用可以减少磁盘IO操作，提升效率。

Page根据状态可以分为三种类型：



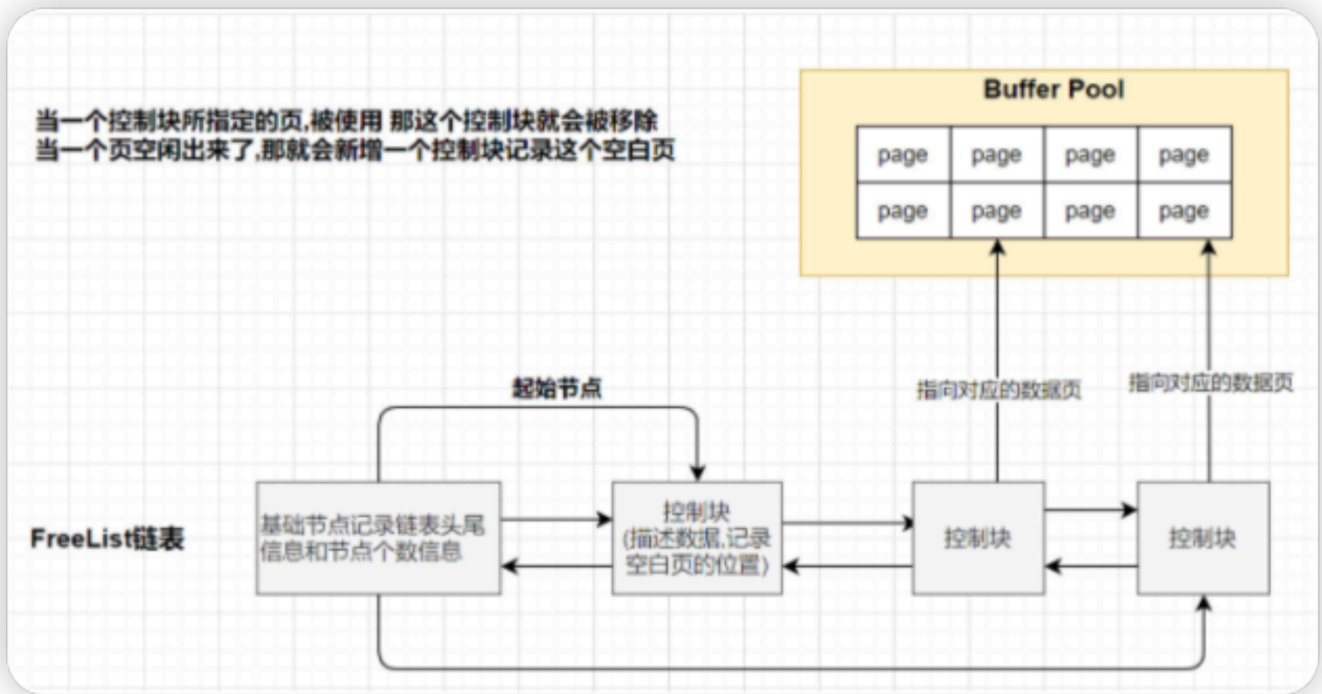
- free page： 空闲page，未被使用
- clean page：被使用page，数据没有被修改过
- dirty page：脏页，被使用page，数据被修改过，Page页中数据和磁盘的数据产生了不一致

Page页如何管理

针对上面所说的三种page类型，InnoDB通过三种链表结构来维护和管理

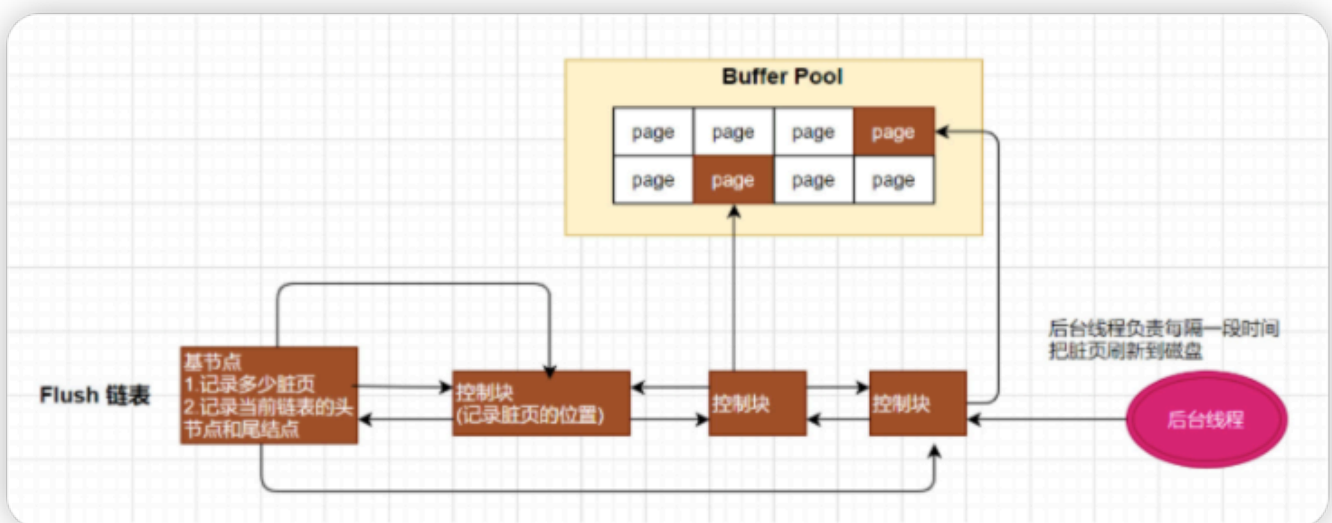
1. free list：表示空闲缓冲区，管理free page

- free链表是把所有空闲的缓冲页对应的控制块作为一个个的节点放到一个链表中，这个链表便称之为free链表
- 基节点: free链表中只有一个基节点是不记录缓存页信息(单独申请空间)，它里面就存放了free链表的头节点的地址，尾节点的地址，还有free链表里当前有多少个节点。



2.flush list： 表示需要刷新到磁盘的缓冲区，管理dirty page，内部page按修改时间排序。

- InnoDB引擎为了提高处理效率，在每次修改缓冲页后，并不是立刻把修改刷新到磁盘上，而是在未来的某个时间点进行刷新操作. 所以需要使用到flush链表存储脏页，凡是被修改过的缓冲页对应的控制块都会作为节点加入到flush链表.
- flush链表的结构与free链表的结构相似



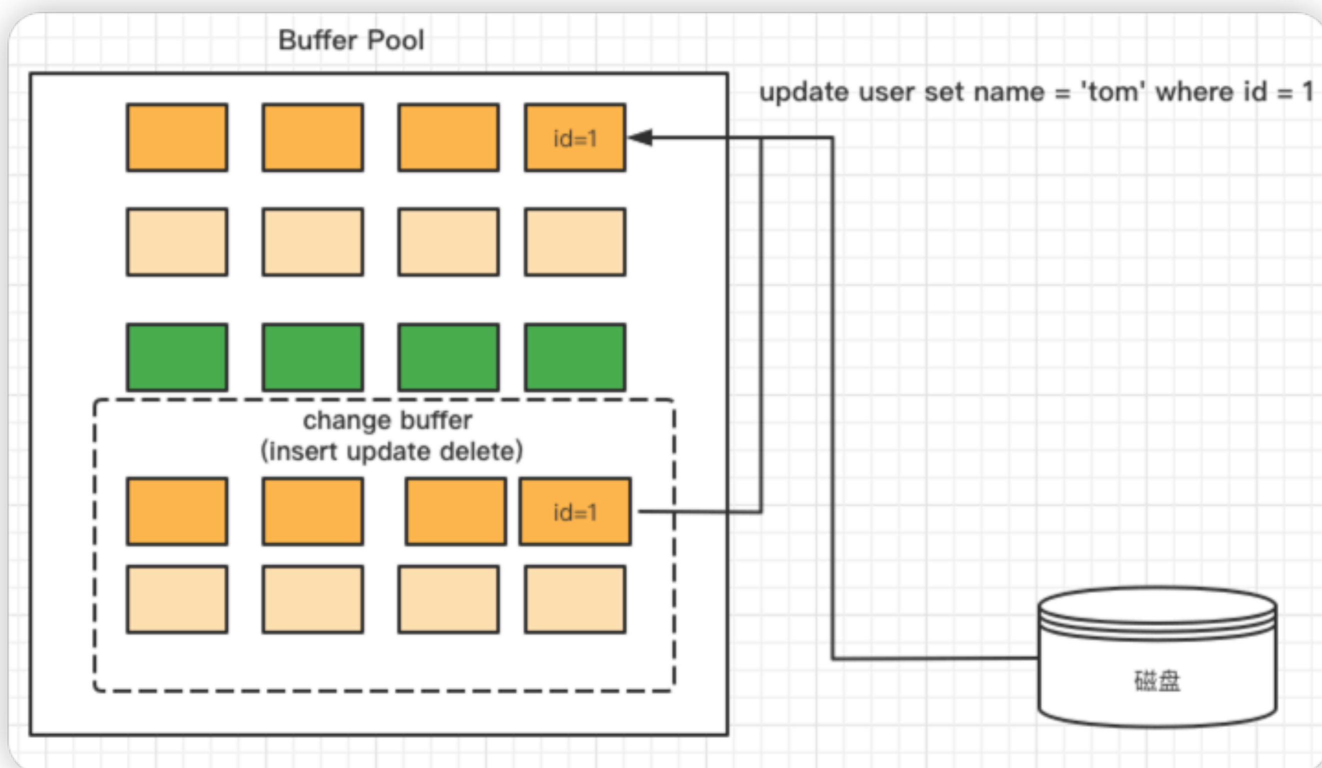
3.lru list：表示正在使用的缓冲区，管理clean page和dirty page，缓冲区以midpoint为基点，前面链表称为new列表区，存放经常访问的数据，占63%；后面的链表称为old列表区，存放使用较少数据，占37%

3.为什么写缓冲区，仅适用于非唯一普通索引页?

change Buffer基本概念

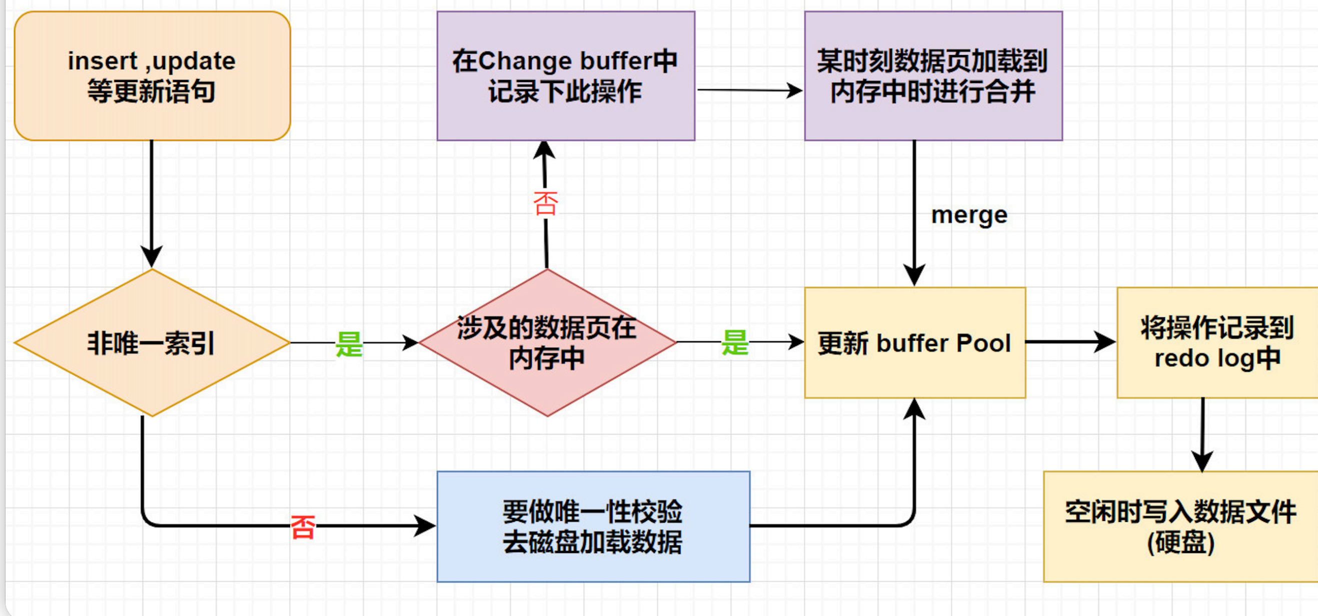
Change Buffer：写缓冲区,是针对二级索引(辅助索引) 页的更新优化措施。

作用：在进行DML操作时，如果请求的辅助索引（二级索引）没有在缓冲池中时，并不会立刻将磁盘页加载到缓冲池，而是在CB记录缓冲变更，等未来数据被读取时，再将数据合并恢复到BP中。



1. ChangeBuffer用于存储SQL变更操作，比如Insert/Update/Delete等SQL语句
2. ChangeBuffer中的每个变更操作都有其对应的数据页，并且该数据页未加载到缓存中；
3. 当ChangeBuffer中变更操作对应的数据页加载到缓存中后，InnoDB会把变更操作Merge到数据页上；
4. InnoDB会定期加载ChangeBuffer中操作对应的数据页到缓存中，并Merge变更操作；

change buffer更新流程



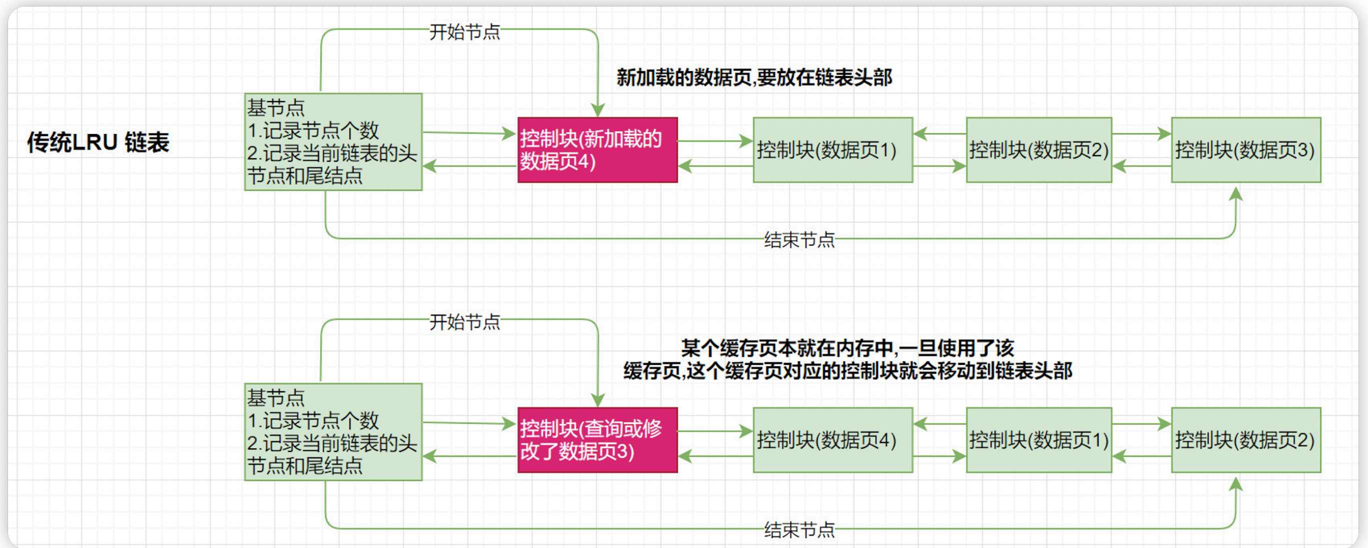
写缓冲区，仅适用于非唯一普通索引页，为什么?

- 如果在索引设置唯一性，在进行修改时，InnoDB必须要做唯一性校验，因此必须查询磁盘，做一次IO操 作。会直接将记录查询到BufferPool中，然后在缓冲池修改，不会在ChangeBuffer操作。

4.MySQL为什么改进LRU算法?

普通LRU算法

LRU = Least Recently Used（最近最少使用）：就是末尾淘汰法，新数据从链表头部加入，释放空间时从末尾淘汰.



1. 当要访问某个页时，如果不在Buffer Pool，需要把该页加载到缓冲池,并且把该缓冲页对应的控制块作为节点添加到LRU链表的头部。
2. 当要访问某个页时，如果在Buffer Pool中，则直接把该页对应的控制块移动到LRU链表的头部
3. 当需要释放空间时,从最末尾淘汰

普通LRU链表的优缺点

优点

- 所有最近使用的数据都在链表表头，最近未使用的数据都在链表表尾,保证热数据能最快被获取到。

缺点

- 如果发生全表扫描（比如：没有建立合适的索引 or 查询时使用select *等），则有很大可能将真正的热数据淘汰掉.
- 由于MySQL中存在预读机制，很多预读的页都会被放到LRU链表的表头。如果这些预读的页都没有用到的话，这样，会导致很多尾部的缓冲页很快就会被淘汰。

链表中原本的数据

head → 8页 12页 7页 6页 5页 100页 10页 40页 ← tail

SQL查询来了新的数据

60页 61页 62页 63页 64页 65页 66页 67页

根据LRU算法被置换之后的列表

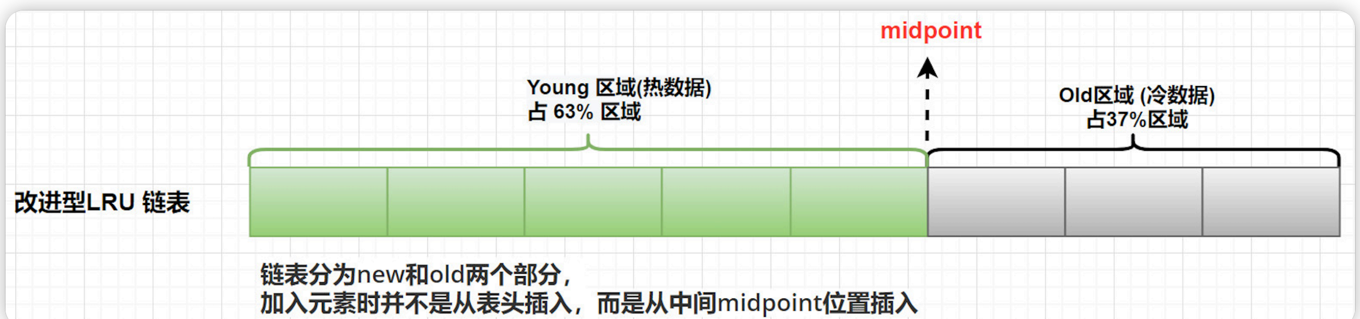
head → 67页 66页 65页 64页 63页 62页 61页 60页 ← tail

8页 12页 7页 6页 5页 100页 10页 40页

被淘汰的这部分数据，
才是经常被使用的数据

改进型LRU算法

改进型LRU：将链表分为new和old两个部分，加入元素时并不是从表头插入，而是从中间midpoint位置插入(就是说从磁盘中新读出的数据会放在冷数据区的头部)，如果数据很快被访问，那么page就会向new列表头部移动，如果数据没有被访问，会逐步向old尾部移动，等待淘汰。



冷数据区的数据页什么时候会被转到到热数据区呢？

1. 如果该数据页在LRU链表中存在时间超过1s，就将其移动到链表头部（链表指的是整个LRU链表)
2. 如果该数据页在LRU链表中存在的时间短于1s，其位置不变(由于全表扫描有一个特点，就是它对某个页的频繁访问总耗时会很短)
3. 1s这个时间是由参数 `innodb_old_blocks_time` 控制的

5.使用索引一定可以提升效率吗?

索引就是排好序的,帮助我们进行快速查找的数据结构.

简单来讲，索引就是一种将数据库中的记录按照特殊形式存储的数据结构。通过索引，能够显著地提高数据查询的效率，从而提升服务器的性能.

索引的优势与劣势

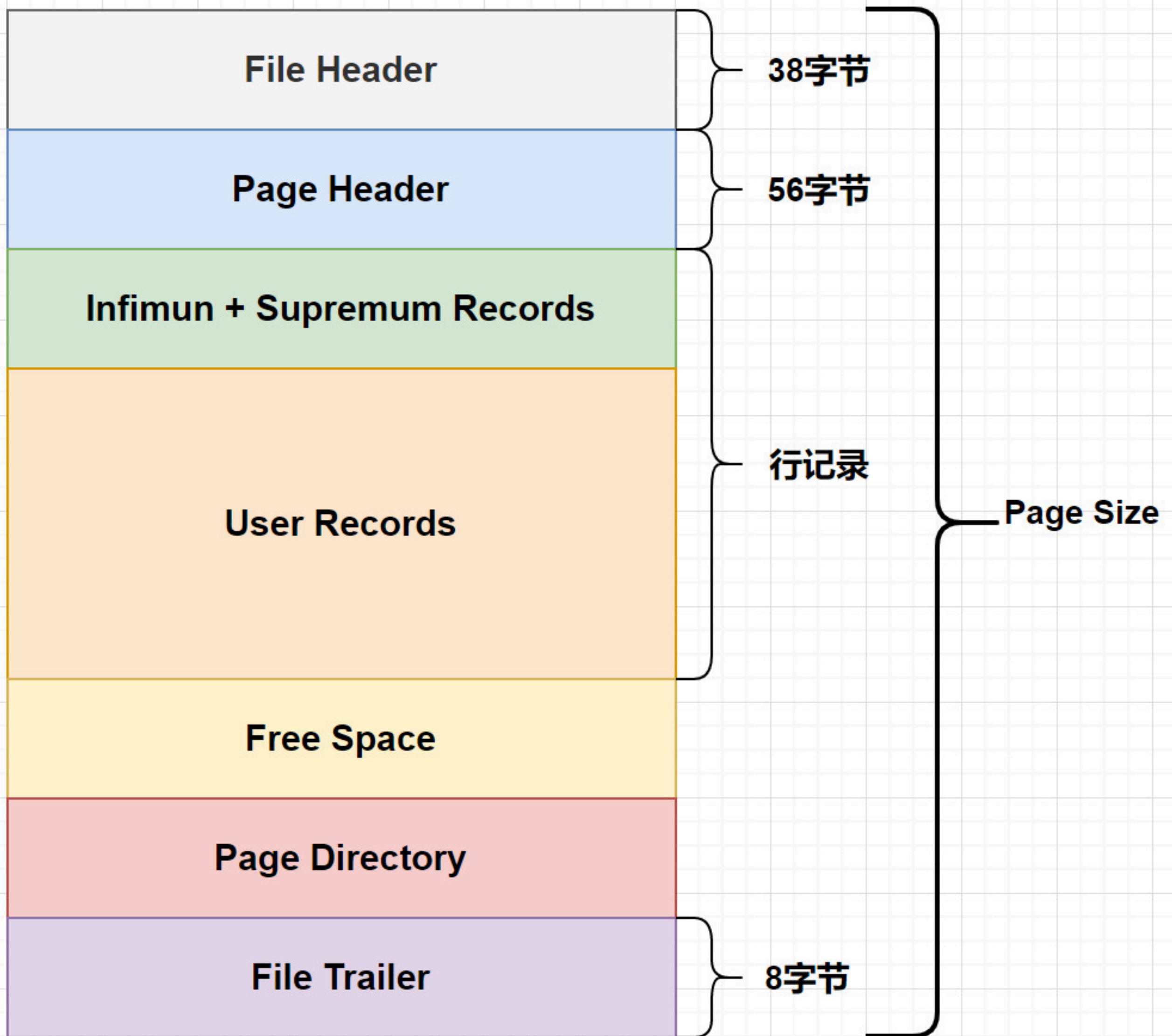
- 优点
    - 提高数据检索的效率,降低数据库的IO成本
    - 通过索引列对数据进行排序,降低数据排序的成本,降低了CPU的消耗
- 缺点
    - 创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加
    - 索引需要占物理空间，除了数据表占用数据空间之外，每一个索引还要占用一定的物理空间
    - 当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，降低了数据的维护速度
- 创建索引的原则
    - 在经常需要搜索的列上创建索引，可以加快搜索的速度；

- 在作为主键的列上创建索引，强制该列的唯一性和组织表中数据的排列结构；
- 在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；
- 在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；
- 在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；
- 在经常使用在WHERE子句中的列上面创建索引，加快条件的判断速度。

6.介绍一下Page页的结构?

Page是整个InnoDB存储的最基本构件，也是InnoDB磁盘管理的最小单位，与数据库相关的所有内容都存储在这种Page结构里。

Page分为几种类型，常见的页类型有数据页（B+tree Node）Undo页（Undo Log Page）系统页（System Page） 事务数据页（Transaction System Page）等



Page 各部分说明

名称	占用大小	说明
File Header	**38字节**	**文件头, 描述页信息**
Page Header	**56字节**	**页头,页的状态**
Infimum + Supremum	**26字节**	**最大和最小记录,这是两个虚拟的行记录**
User Records	**不确定**	**用户记录,存储数据行记录**
Free Space	**不确定**	**空闲空间,页中还没有被使用的空间**
Page Directory	**不确定**	**页目录,存储用户记录的相对位置**
File Trailer	**8字节**	**文件尾,校验页是否完整**

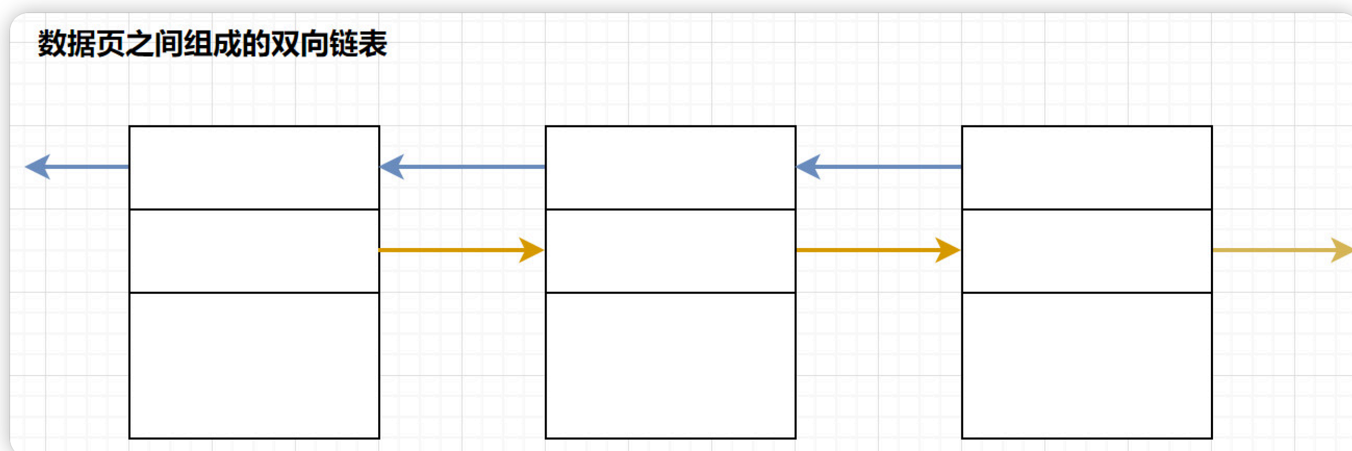
- File Header 字段用于记录 Page 的头信息，其中比较重要的是 FIL_PAGE_PREV 和 FIL_PAGE_NEXT 字段，通过这两个字段，我们可以找到该页的上一页和下一页，实际上所有页通过两个字段可以形成一条双向链表
- Page Header 字段用于记录 Page 的状态信息。
- Infimum 和 Supremum 是两个伪行记录，Infimum（下确界）记录比该页中任何主键值都要小的值，Supremum （上确界）记录比该页中任何主键值都要大的值，这个伪记录分别构成了页中记录的边界。
- User Records 中存放的是实际的数据行记录
- Free Space 中存放的是空闲空间，被删除的行记录会被记录成空闲空间
- Page Directory 记录着与二叉查找相关的信息
- File Trailer 存储用于检测数据完整性的校验和等数据。

页结构整体上可以分为三大部分，分别为通用部分(文件头、文件尾)、存储记录空间、索引部分。

1. 通用部分 (File Header&File Trailer)

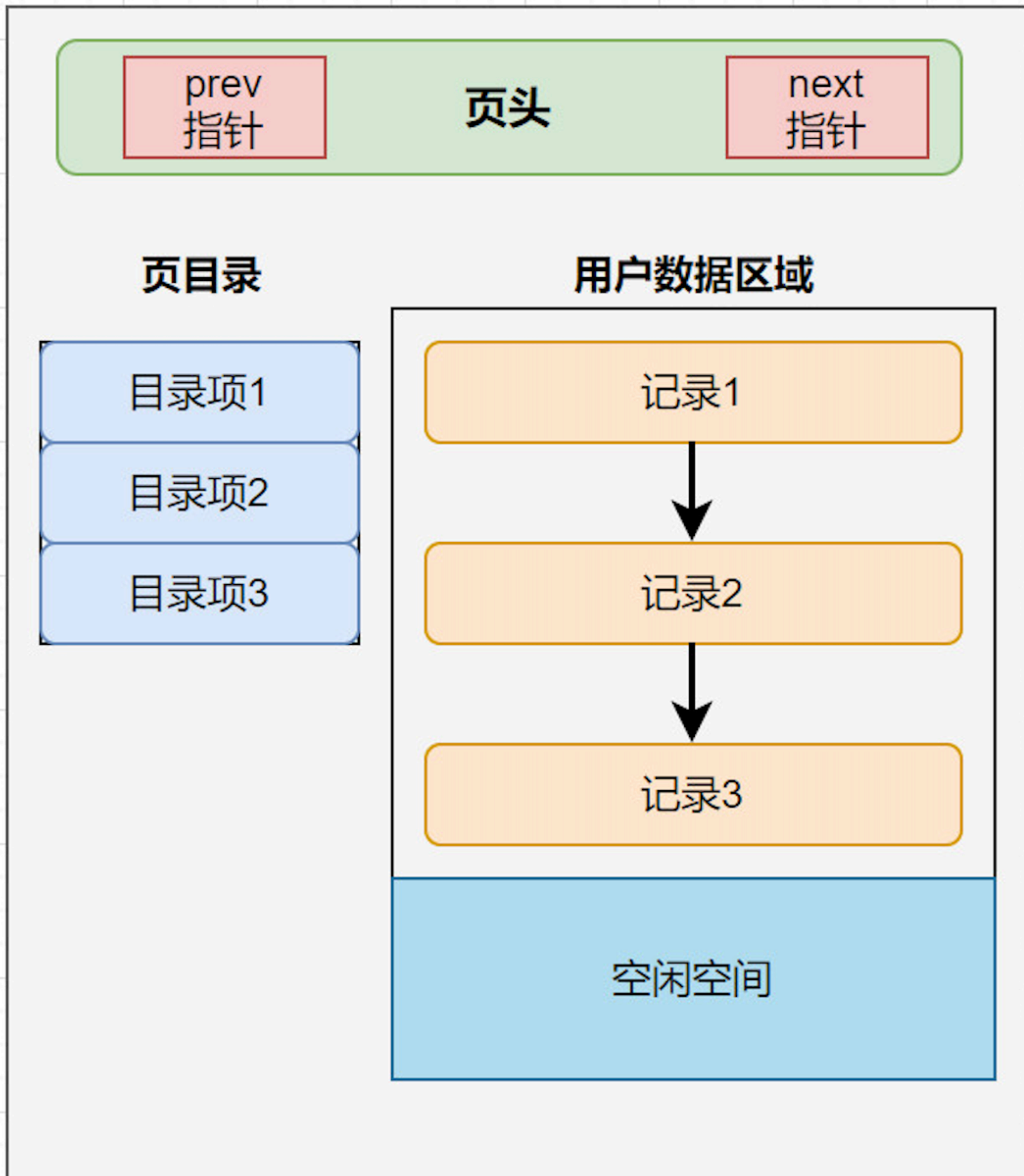
通用部分：主要指文件头和文件尾，将页的内容进行封装，通过文件头和文件尾校验的CheckSum方式来确保页的传输是完整的。

其中比较重要的是在文件头中的 `FIL_PAGE_PREV` 和 `FIL_PAGE_NEXT` 字段，通过这两个字段，我们可以找到该页的上一页和下一页，实际上所有页通过两个字段可以形成一条双向链表



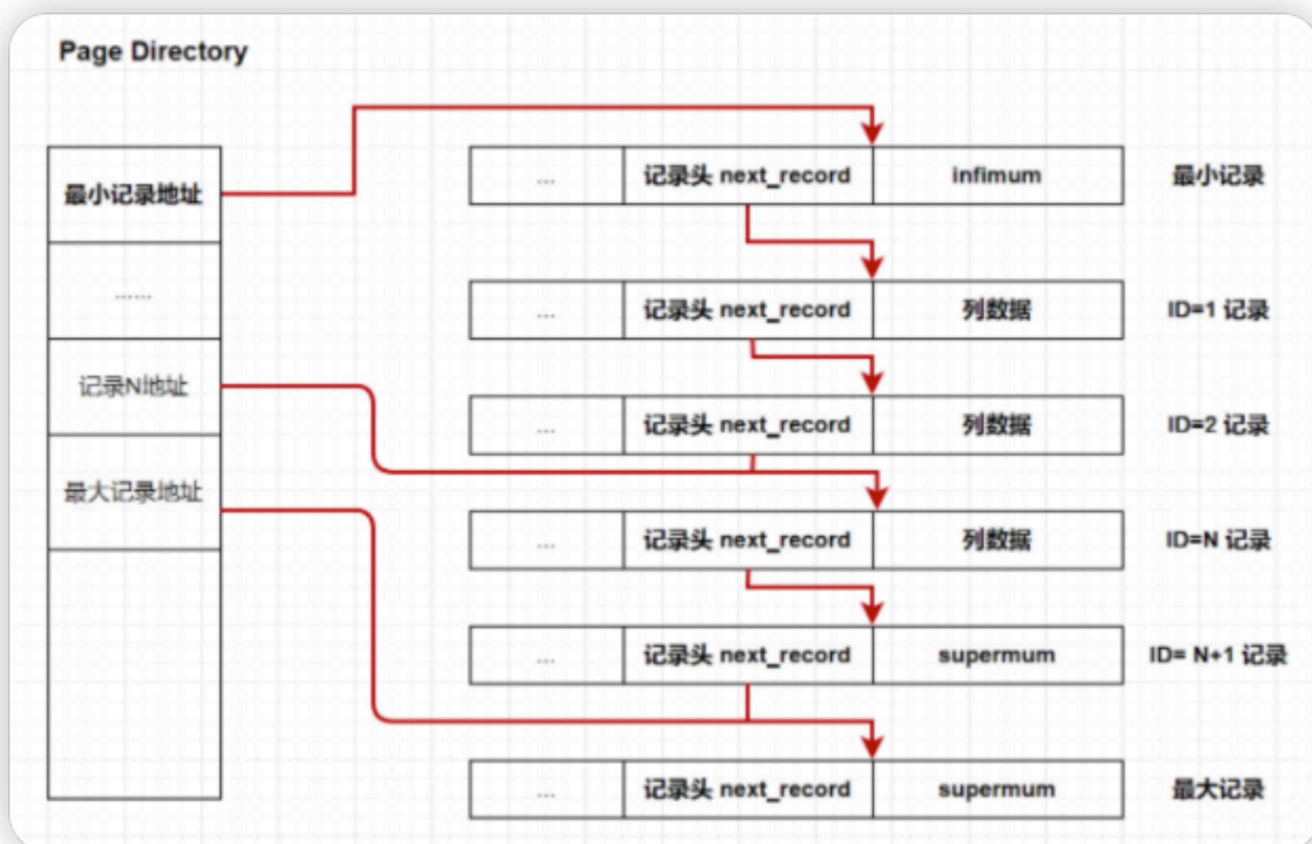
2. 记录部分(User Records&Free Space)

页的主要作用是存储记录，所以“最小和最大记录”和“用户记录”部分占了页结构的主要空间。另外空闲空间是个灵活的部分，当有新的记录插入时，会从空闲空间中进行分配用于存储新记录



3)数据目录部分 (Page Directory)

数据页中行记录按照主键值由小到大顺序串联成一个单链表(**页中记录是以单向链表的形式进行存储的**)，且单链表的链表头为最小记录，链表尾为最大记录。并且为了更快速地定位到指定的行记录，通过 `Page Directory` 实现目录的功能，借助 `Page Directory`使用二分法快速找到需要查找的行记录。



7.说一下聚簇索引与非聚簇索引?

聚集索引与非聚集索引的区别是：叶节点是否存放一整行记录

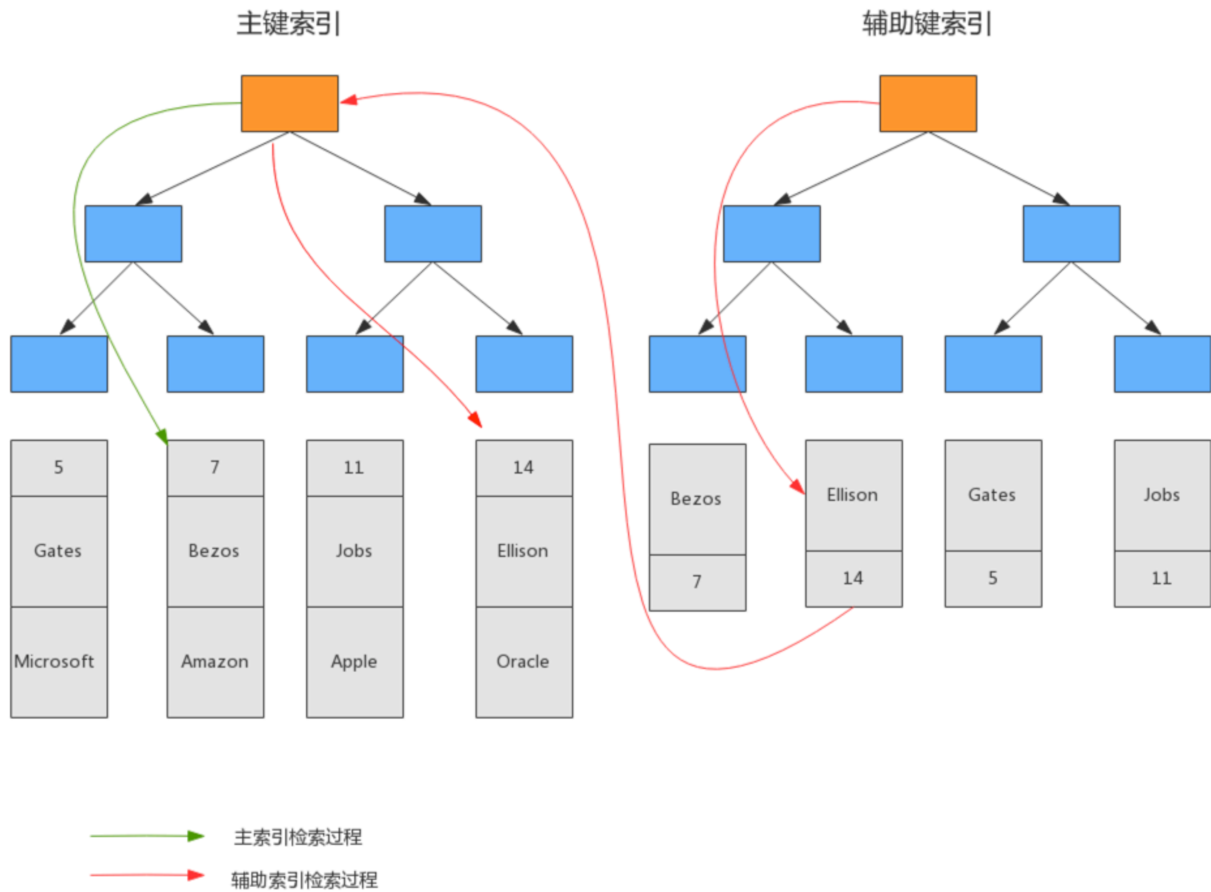
- **聚簇索引**: 将数据存储与索引放到了一块,索引结构的叶子节点保存了行数据.
- **非聚簇索引**：将数据与索引分开存储，索引结构的叶子节点指向了数据对应的位置.

InnoDB 主键使用的是聚簇索引，MyISAM 不管是主键索引，还是二级索引使用的都是非聚簇索引。

在InnoDB引擎中，主键索引采用的就是聚簇索引结构存储。

聚簇索引（聚集索引）

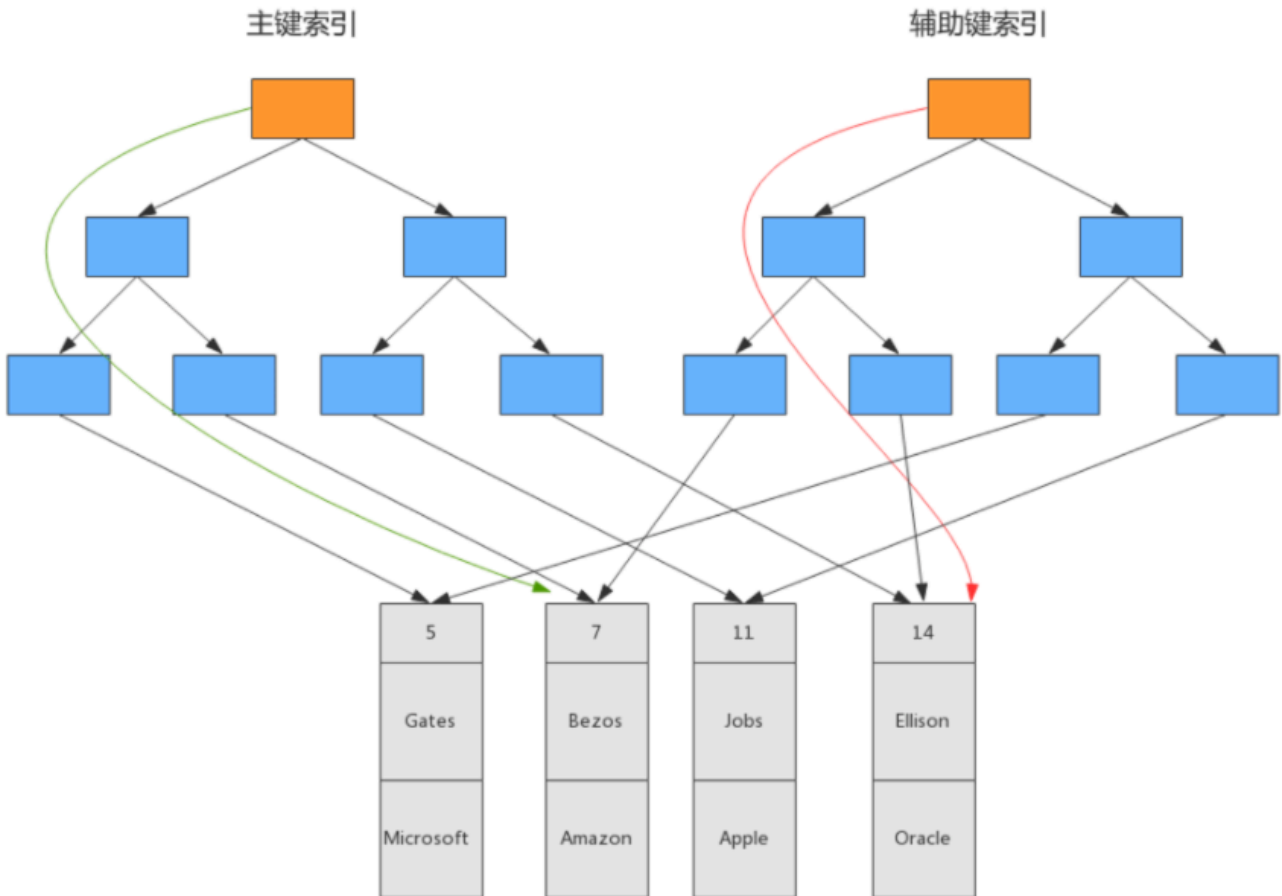
- 聚簇索引是一种数据存储方式，InnoDB的聚簇索引就是按照主键顺序构建 B+Tree结构。B+Tree 的叶子节点就是行记录，行记录和主键值紧凑地存储在一起。 这也意味着 InnoDB 的主键索引就是数据表本身，它按主键顺序存放了整张表的数据，占用的空间就是整个表数据量的大小。通常说的主键索引就是聚集索引。
- InnoDB的表要求必须要有聚簇索引：
    - 如果表定义了主键，则主键索引就是聚簇索引
    - 如果表没有定义主键，则第一个非空unique列作为聚簇索引
    - 否则InnoDB会从建一个隐藏的row-id作为聚簇索引
- 辅助索引
  InnoDB辅助索引，也叫作二级索引，是根据索引列构建 B+Tree结构。但在 B+Tree 的叶子节点中只存了索引列和主键的信息。二级索引占用的空间会比聚簇索引小很多， 通常创建辅助索引就是为了提升查询效率。一个表InnoDB只能创建一个聚簇索引，但可以创建多个辅助索引。



非聚簇索引

与InnoDB表存储不同，MyISM使用的是非聚簇索引， 非聚簇索引的两棵B+树看上去没什么不同 ， 节点的结构完全一致只是存储的内容不同而已，主键索引B+树的节点存储了主键，辅助键索引B+树存储了辅助键。

表数据存储在独立的地方，这两颗B+树的叶子节点都使用一个地址指向真正的表数据，对于表数据来说，这两个键没有任何差别。由于 索引树是独立的，通过辅助键检索无需访问主键的索引树 。



聚簇索引的优点

1. 当你需要取出一定范围内的数据时，用聚簇索引也比用非聚簇索引好。
2. 当通过聚簇索引查找目标数据时理论上比非聚簇索引要快，因为非聚簇索引定位到对应主键时还要多一次目标记录寻址,即多一次I/O。
3. 使用覆盖索引扫描的查询可以直接使用页节点中的主键值。

聚簇索引的缺点

1. 插入速度严重依赖于插入顺序 。
2. 更新主键的代价很高，因为将会导致被更新的行移动 。
3. 二级索引访问需要两次索引查找，第一次找到主键值，第二次根据主键值找到行数据。

8.索引有哪几种类型?

1）普通索引

- **这是最基本的索引类型，基于普通字段建立的索引，没有任何限制。**

```
CREATE INDEX <索引的名字> ON tablename (字段名);
ALTER TABLE tablename ADD INDEX [索引的名字] (字段名);
CREATE TABLE tablename ( [...], INDEX [索引的名字] (字段名)
);
```

2）唯一索引

- **与"普通索引"类似，不同的就是：索引字段的值必须唯一，但允许有空值 。**

```
CREATE UNIQUE INDEX <索引的名字> ON tablename (字段名);
ALTER TABLE tablename ADD UNIQUE INDEX [索引的名字] (字段名);
CREATE TABLE tablename ( [...], UNIQUE [索引的名字] (字段名)
;
```

3）主键索引

- **它是一种特殊的唯一索引，不允许有空值。在创建或修改表时追加主键约束即可，每个表只能有一个主键。**

```
CREATE TABLE tablename ( [...], PRIMARY KEY (字段名) );
ALTER TABLE tablename ADD PRIMARY KEY (字段名);
```

4）复合索引

- **用户可以在多个列上建立索引，这种索引叫做组复合索引（组合索引）。复合索引可以代替多个单一索引，相比多个单一索引复合索引所需的开销更小。**

```
CREATE INDEX <索引的名字> ON tablename (字段名1，字段名2...);

ALTER TABLE tablename ADD INDEX [索引的名字] (字段名1，字段名2...);

CREATE TABLE tablename ( [...], INDEX [索引的名字] (字段名1，字段名2...) );
```

- **复合索引使用注意事项：**
  - **何时使用复合索引，要根据where条件建索引，注意不要过多使用索引，过多使用会对更新操作效率有很大影响。**
  - **如果表已经建立了(col1，col2)，就没有必要再单独建立（col1）；如果现在有(col1)索引，如果查询需要col1和col2条件，可以建立(col1,col2)复合索引，对于查询有一定提高。**

5) 全文索引

查询操作在数据量比较少时，可以使用like模糊查询，但是对于大量的文本数据检索，效率很低。如果使用全文索引，查询速度会比like快很多倍。

在MySQL 5.6 以前的版本，只有MyISAM存储引擎支持全文索引，从MySQL 5.6开始MyISAM和InnoDB存储引擎均支持。

```
CREATE FULLTEXT INDEX <索引的名字> ON tablename (字段名);

ALTER TABLE tablename ADD FULLTEXT [索引的名字] (字段名);

CREATE TABLE tablename ( [...], FULLTEXT KEY [索引的名字] (字段名) ;
```

全文索引方式有自然语言检索 `IN NATURAL LANGUAGE MODE`和布尔检索 `IN BOOLEAN MODE`两种

和常用的like模糊查询不同，全文索引有自己的语法格式，使用 match 和 against 关键字，比如

```
SELECT * FROM users3 WHERE MATCH(NAME) AGAINST('aabb');

-- * 表示通配符,只能在词的后面
SELECT * FROM users3 WHERE MATCH(NAME) AGAINST('aa*'  IN
BOOLEAN MODE);
```

全文索引使用注意事项：

- 全文索引必须在字符串、文本字段上建立。
- 全文索引字段值必须在最小字符和最大字符之间的才会有效。（innodb：3-84；myisam：4-84）

9.介绍一下最佳左前缀法则?

1)最佳左前缀法则

最佳左前缀法则：如果创建的是联合索引,就要遵循该法则. 使用索引时，where后面的条件需要从索引的最左前列开始使用,并且不能跳过索引中的列使用。

- 场景1：按照索引字段顺序使用，三个字段都使用了索引,没有问题。

  ```
  EXPLAIN SELECT * FROM users WHERE user_name = 'tom'
  AND user_age = 17 AND user_level = 'A';
  ```

- 场景2: 直接跳过user_name使用索引字段，索引无效，未使用到索引。

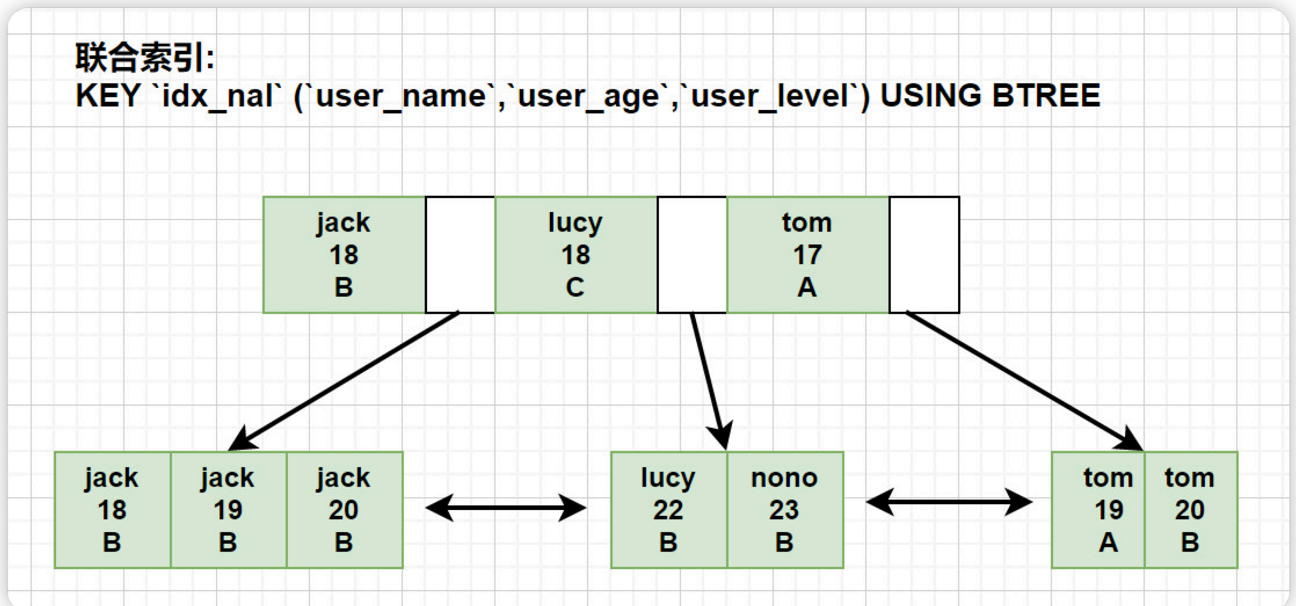
  ```
  EXPLAIN SELECT * FROM users WHERE user_age = 17 AND
  user_level = 'A';
  ```

- 场景3: 不按照创建联合索引的顺序,使用索引

```
EXPLAIN SELECT * FROM users WHERE
user_age = 17 AND user_name = 'tom' AND user_level =
'A';
```

where后面查询条件顺序是 `user_age`、`user_level`、`user_name`与我们创建的索引顺序 `user_name`、`user_age`、`user_level`不一致, 为什么还是使用了索引, 原因是因为MySql底层优化器对其进行了优化。

- 最佳左前缀底层原理
MySQL创建联合索引的规则是: 首先会对联合索引最左边的字段进行排序(例子中是 `user_name`), 在第一个字段的基础之上 再对第二个字段进行排序 (例子中是 `user_age`) .



- 最佳左前缀原则其实是和B+树的结构有关系, 最左字段肯定是有序的, 第二个字段则是无序的(联合索引的排序方式是: 先按照第一个字段进行排序,如果第一个字段相等再根据第二个字段排序). 所以如果直接使用第二个字段 `user_age` 通常是使用不到索引的.

10.什么是索引下推?

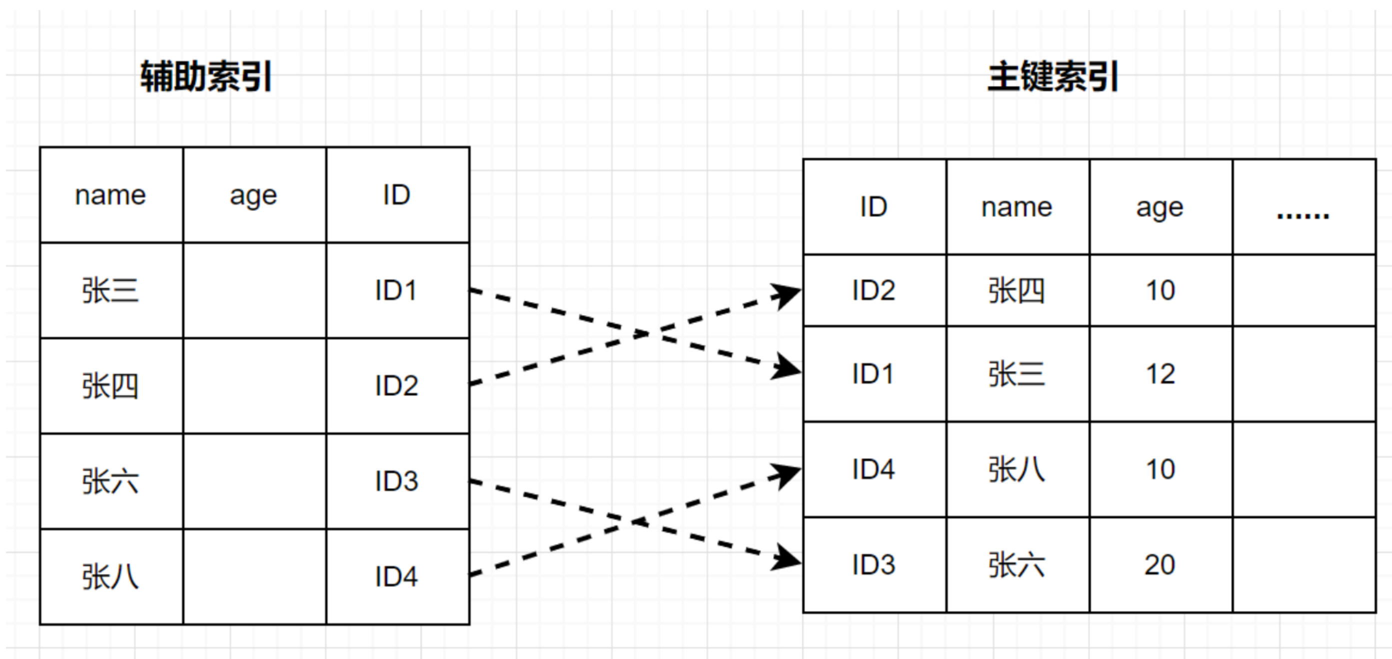
索引下推（index condition pushdown ）简称ICP，在Mysql5.6的版本上推出，用于优化查询。

需求: 查询users表中 "名字第一个字是张，年龄为10岁的所有记录"。

```
SELECT * FROM users WHERE user_name LIKE '张%' AND user_age = 10;
```

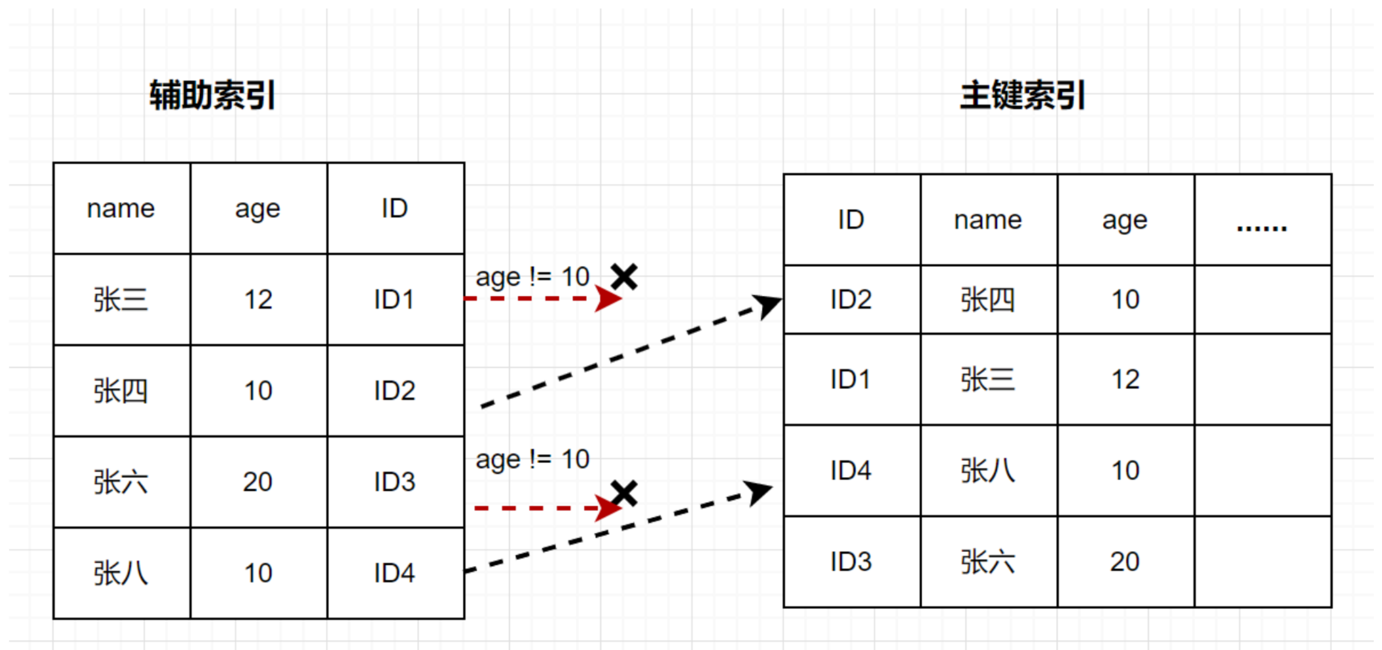
根据最左前缀法则，该语句在搜索索引树的时候，只能匹配到名字第一个字是'张'的记录，接下来是怎么处理的呢？当然就是从该记录开始，逐个回表，到主键索引上找出相应的记录，再比对 `age` 这个字段的值是否符合。

图1: 在 (name,age) 索引里面特意去掉了 age 的值，这个过程 InnoDB 并不会去看 age 的值，只是按顺序把"name 第一个字是'张'"的记录一条条取出来回表。因此，需要回表 4 次



MySQL 5.6引入了索引下推优化，可以在索引遍历过程中，对索引中包含的字段先做判断，过滤掉不符合条件的记录，减少回表次数。

图2: InnoDB 在 (name,age) 索引内部就判断了 age 是否等于 10，对于不等于 10 的记录，直接判断并跳过,减少回表次数.



总结

如果没有索引下推优化（或称ICP优化），当进行索引查询时，首先根据索引来查找记录，然后再根据where条件来过滤记录；

在支持ICP优化后，MySQL会在取出索引的同时，判断是否可以进行where条件过滤再进行索引查询，也就是说提前执行where的部分过滤操作，在某些场景下，可以大大减少回表次数，从而提升整体性能。

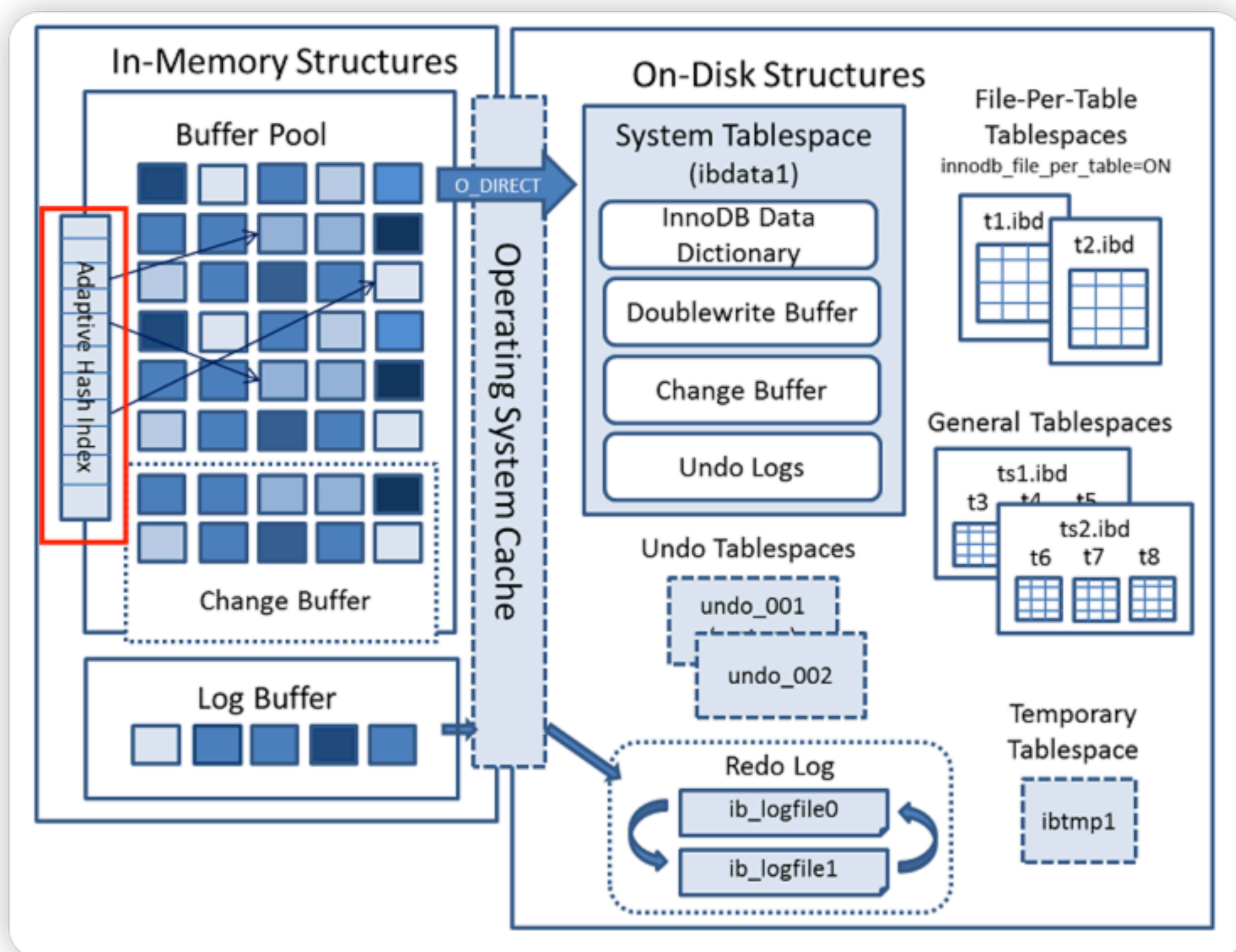
11.什么是自适应哈希索引?

自适应Hash索引（Adatptive Hash Index，内部简称AHI）是InnoDB的三大特性之一，还有两个是 Buffer Pool简称BP、双写缓冲区（Doublewrite Buffer）。

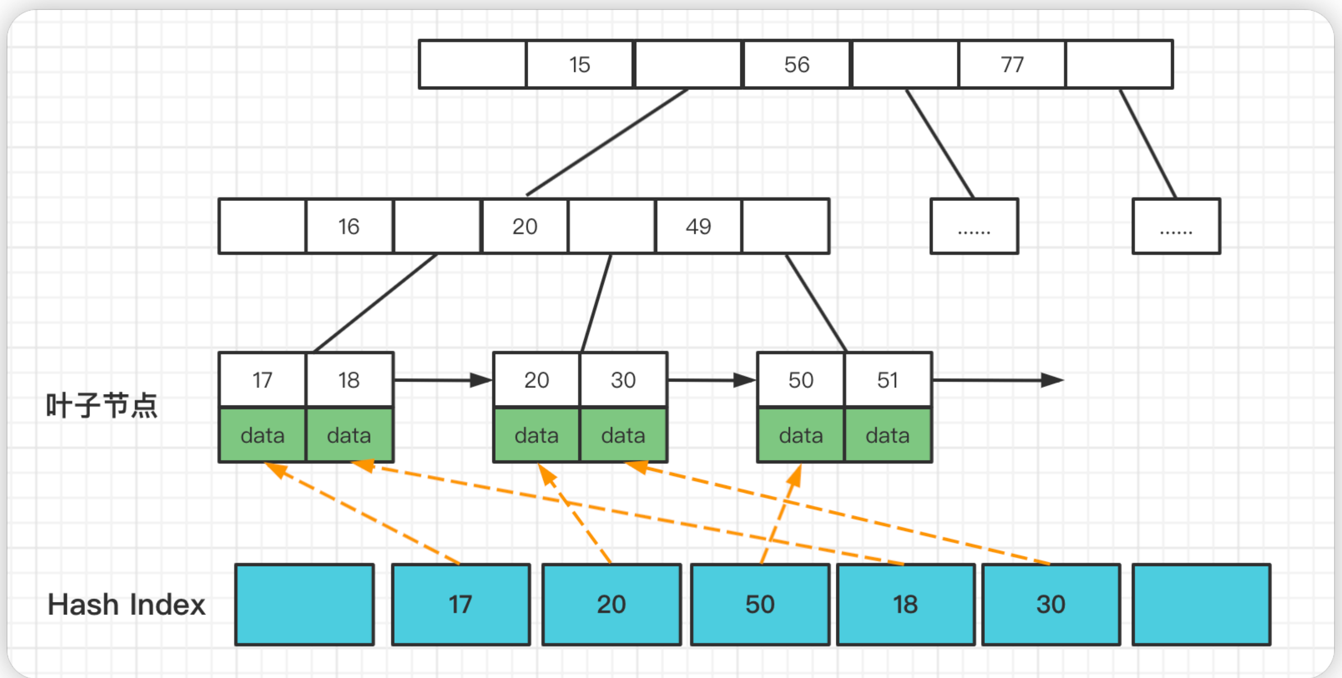
1、自适应即我们不需要自己处理，当InnoDB引擎根据查询统计发现某一查询满足hash索引的数据结构特点，就会给其建立一个hash索引；

2、hash索引底层的数据结构是散列表（Hash表），其数据特点就是比较适合在内存中使用，自适应Hash索引存在于InnoDB架构中的缓存中（不存在于磁盘架构中），见下面的InnoDB架构图。

3、自适应hash索引只适合搜索等值的查询，如select * from table where index_col='xxx'，而对于其他查找类型，如范围查找，是不能使用的；



Adaptive Hash Index是针对B+树Search Path的优化，因此所有会涉及到Search Path的操作，均可使用此Hash索引进行优化.



根据索引键值(前缀)快速定位到叶子节点满足条件记录的Offset，减少了B+树Search Path的代价，将B+树从Root节点至Leaf节点的路径定位，优化为Hash Index的快速查询。

InnoDB的自适应Hash索引是默认开启的，可以通过配置下面的参数设置进行关闭。

```
innodb_adaptive_hash_index = off
```

自适应Hash索引使用分片进行实现的，分片数可以使用配置参数设置：

```
innodb_adaptive_hash_index_parts = 8
```

12.为什么LIKE以%开头索引会失效?

like查询为范围查询，%出现在左边，则索引失效。%出现在右边索引未失效.

场景1: 两边都有% 或者 字段左边有%,索引都会失效。

```
EXPLAIN SELECT * FROM users WHERE user_name LIKE '%tom%';

EXPLAIN SELECT * FROM users WHERE user_name LIKE '%tom';
```

场景2: 字段右边有%,索引生效

```
EXPLAIN SELECT * FROM users WHERE user_name LIKE 'tom%';
```

解决%出现在左边索引失效的方法，使用覆盖索引

```
EXPLAIN SELECT user_name FROM users WHERE user_name LIKE
'%jack%';

EXPLAIN SELECT user_name,user_age,user_level FROM users
WHERE user_name LIKE '%jack%';
```

对比场景1可以知道, 通过使用覆盖索引 `type = index`,并且 `extra = Using index`,从全表扫描变成了全索引扫描.

like 失效的原因

1. **%号在右:** 由于B+树的索引顺序，是按照首字母的大小进行排序，%号在右的匹配又是匹配首字母。所以可以在B+树上进行有序的查找，查找首字母符合要求的数据。所以有些时候可以用到索引.
2. **%号在左:** 是匹配字符串尾部的数据，我们上面说了排序规则，尾部的字母是没有顺序的，所以不能按照索引顺序查询，就用不到索引.
3. **两个%%号:** 这个是查询任意位置的字母满足条件即可，只有首字母是进行索引排序的，其他位置的字母都是相对无序的，所以查找任意位置的字母是用不上索引的.

13.自增还是UUID？数据库主键的类型该如何选择？

auto_increment的优点：

1. 字段长度较uuid小很多，可以是bigint甚至是int类型，这对检索的性能会有所影响。
2. 在写的方面，因为是自增的，所以主键是趋势自增的，也就是说新增的数据永远在后面，这点对于性能有很大的提升。
3. 数据库自动编号，速度快，而且是增量增长，按顺序存放，对于检索非常有利。
4. 数字型，占用空间小，易排序，在程序中传递也方便。

auto_increment的缺点：

1. 由于是自增，很容易通过网络爬虫知晓当前系统的业务量。
2. 高并发的情况下，竞争自增锁会降低数据库的吞吐能力。
3. 数据迁移或分库分表场景下，自增方式不再适用。

UUID的优点：

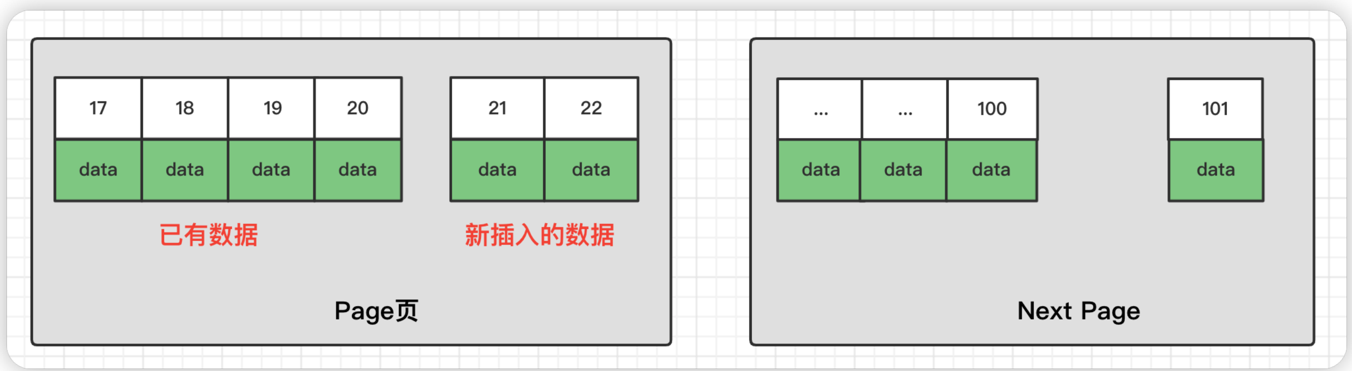
1. 不会冲突。进行数据拆分、合并存储的时候，能保证主键全局的唯一性
2. 可以在应用层生成，提高数据库吞吐能力

UUID的缺点：

1. 影响插入速度， 并且造成硬盘使用率低。与自增相比，最大的缺陷就是随机io，下面我们会去具体解释
2. 字符串类型相比整数类型肯定更消耗空间，而且会比整数类型操作慢。

uuid 和自增 id 的索引结构对比

1、**使用自增 id 的内部结构**

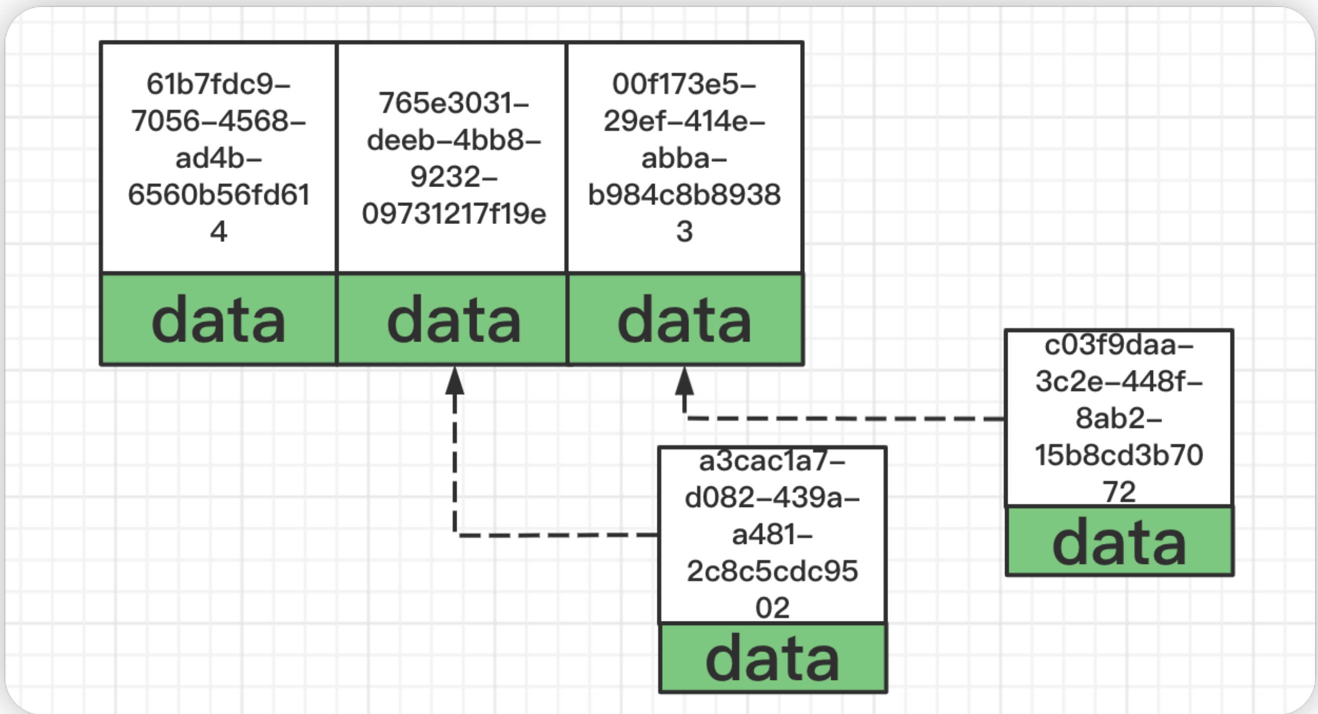


自增的主键的值是顺序的，所以 InnoDB 把每一条记录都存储在一条记录的后面。

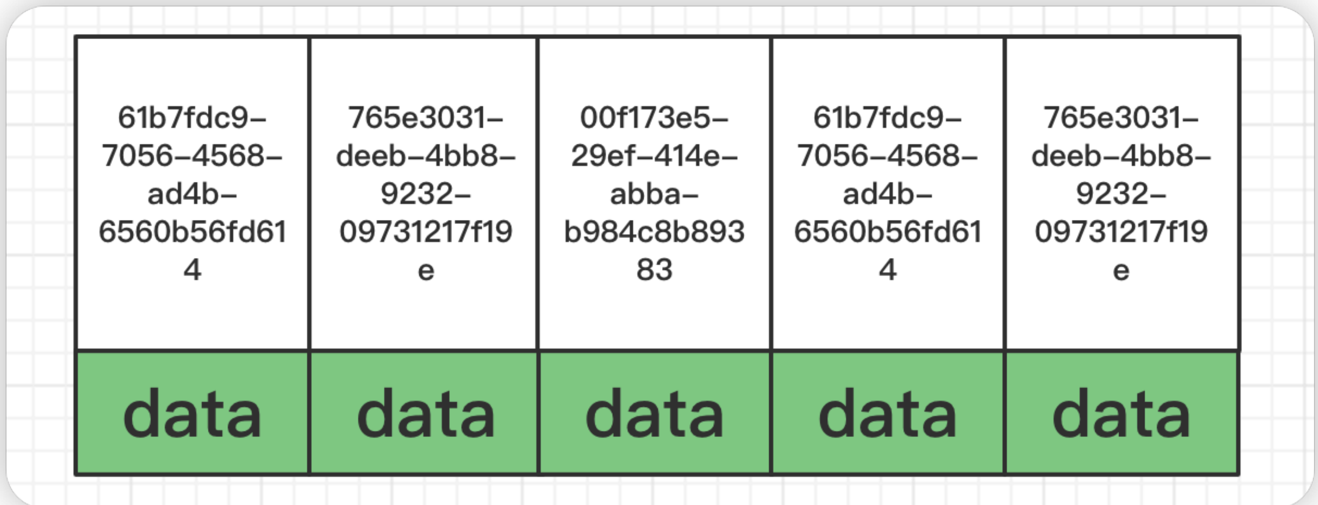
- 当达到页面的最大填充因子时候（InnoDB 默认的最大填充因子是页大小的 15/16，会留出 1/16 的空间留作以后的修改）。
- 下一条记录就会写入新的页中，一旦数据按照这种顺序的方式加载，主键页就会近乎于顺序的记录填满，提升了页面的最大填充率，不会有页的浪费。
- 新插入的行一定会在原有的最大数据行下一行，MySQL 定位和寻址很快，不会为计算新行的位置而做出额外的消耗。减少了页分裂和碎片的产生。

2、**使用 uuid 的索引内部结构**

插入UUID： 新的记录可能会插入之前记录的中间，因此需要移动之前的记录



被写满已经刷新到磁盘上的页可能会被重新读取



因为 uuid 相对顺序的自增 id 来说是毫无规律可言的，新行的值不一定要比之前的主键的值要大，所以 innodb 无法做到总是把新行插入到索引的最后，而是需要为新行寻找新的合适的位置从而来分配新的空间。

这个过程需要做很多额外的操作，数据的毫无顺序会导致数据分布散乱，将会导致以下的问题：

1. 写入的目标页很可能已经刷新到磁盘上并且从缓存上移除，或者还没有被加载到缓存中，innodb 在插入之前不得不先找到并从磁盘读取目标页到内存中，这将导致大量的随机 IO。
2. 因为写入是乱序的，innodb 不得不频繁的做页分裂操作，以便为新的行分配空间，页分裂导致移动大量的数据，一次插入最少需要修改三个页以上。
3. 由于频繁的页分裂，页会变得稀疏并被不规则的填充，最终会导致数据会有碎片。
4. 在把随机值（uuid 和雪花 id）载入到聚簇索引（InnoDB 默认的索引类型）以后，有时候会需要做一次 OPTIMEIZE TABLE 来重建表并优化页的填充，这将又需要一定的时间消耗。

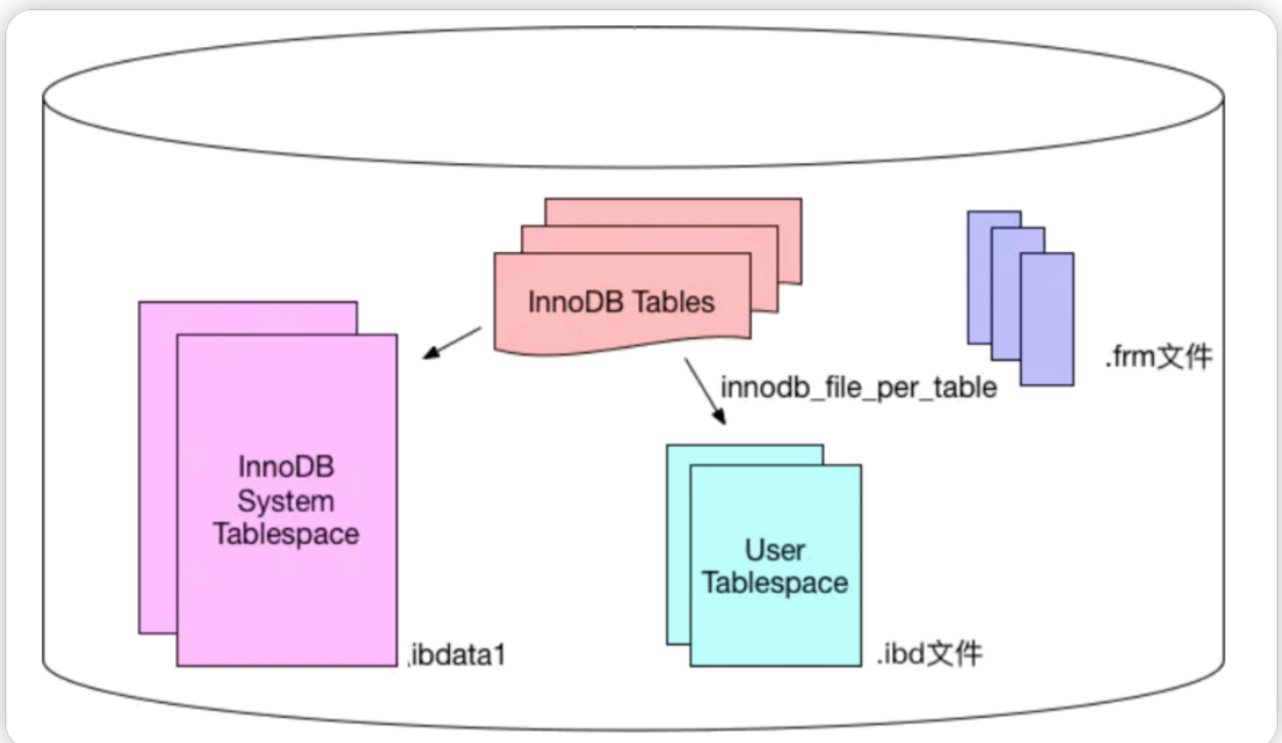
结论：使用 InnoDB 应该尽可能的按主键的自增顺序插入，并且尽可能使用单调的增加的聚簇键的值来插入新行。如果是分库分表场景下，分布式主键ID的生成方案 优先选择雪花算法生成全局唯一主键（雪花算法生成的主键在一定程度上是有序的）。

14.InnoDB与MyISAM的区别?

InnoDB和MyISAM是使用MySQL时最常用的两种引擎类型，我们重点来看下两者区别。

- 事务和外键
  InnoDB支持事务和外键，具有安全性和完整性，适合大量insert或update操作
  MyISAM不支持事务和外键，它提供高速存储和检索，适合大量的select查询操作
- 锁机制
  InnoDB支持行级锁，锁定指定记录。基于索引来加锁实现。
  MyISAM支持表级锁，锁定整张表。

- 索引结构
  InnoDB使用聚集索引（聚簇索引），索引和记录在一起存储，既缓存索引，也缓存记录。
  MyISAM使用非聚集索引（非聚簇索引），索引和记录分开。
- 并发处理能力
  MyISAM使用表锁，会导致写操作并发率低，读之间并不阻塞，读写阻塞。
  InnoDB读写阻塞可以与隔离级别有关，可以采用多版本并发控制（MVCC）来支持高并发
- 存储文件
  InnoDB表对应两个文件，一个.frm表结构文件，一个.ibd数据文件。InnoDB表最大支持64TB；
  MyISAM表对应三个文件，一个.frm表结构文件，一个MYD表数据文件，一个.MYI索引文件。从MySQL5.0开始默认限制是256TB。



MyISAM 适用场景

- 不需要事务支持（不支持）

- 并发相对较低（锁定机制问题）
- 数据修改相对较少，以读为主
- 数据一致性要求不高

InnoDB 适用场景

- 需要事务支持（具有较好的事务特性）
- 行级锁定对高并发有很好的适应能力
- 数据更新较为频繁的场景
- 数据一致性要求较高
- 硬件设备内存较大，可以利用InnoDB较好的缓存能力来提高内存利用率，减少磁盘IO

两种引擎该如何选择?

- 是否需要事务？有，InnoDB
- 是否存在并发修改？有，InnoDB
- 是否追求快速查询，且数据修改少？是，MyISAM
- 在绝大多数情况下，推荐使用InnoDB

扩展资料：各个存储引擎特性对比

Feature	MyISAM	Memory	InnoDB	Archive	NDB
B-tree indexes	Yes	Yes	Yes	No	No
Backup/point-in-time recovery(note 1)	Yes	Yes	Yes	Yes	Yes
Cluster database support	No	No	No	No	Yes
Clustered indexes	No	No	Yes	No	No
Compressed data	Yes（note2)	No	Yes	Yes	No
Data caches	No	N/A	Yes	No	Yes
Encrypted data	Yes（note3)	Yes（note3)	Yes（note4)	Yes（note3)	Yes（note3)
Foreign key support	No	No	Yes	No	Yes（note5)
Full-text search indexes	Yes	No	Yes（note6)	No	no
Geospatial data type support	Yes	No	Yes	Yes	Yes
Geospatial indexing support	Yes	No	Yes（note7)	No	No
Hash indexes	No	Yes	No（note8)	No	Yes
Index caches	Yes	N/A	Yes	No	Yes
Locking granularity	Table	Table	Row	Row	Row
MVCC	No	No	Yes	No	No
Replication support(note1)	Yes	Limited（note9)	Yes	Yes	Yes
Storage limits	256TB	RAM	64TB	None	384EB
T-tree indexes	No	No	No	No	Yes
Transactions	No	No	Yes	No	Yes
Update statistics for data dictionary	Yes	Yes	Yes	Yes	Yes

15.B树和B+树的区别是什么？

1）B-Tree介绍

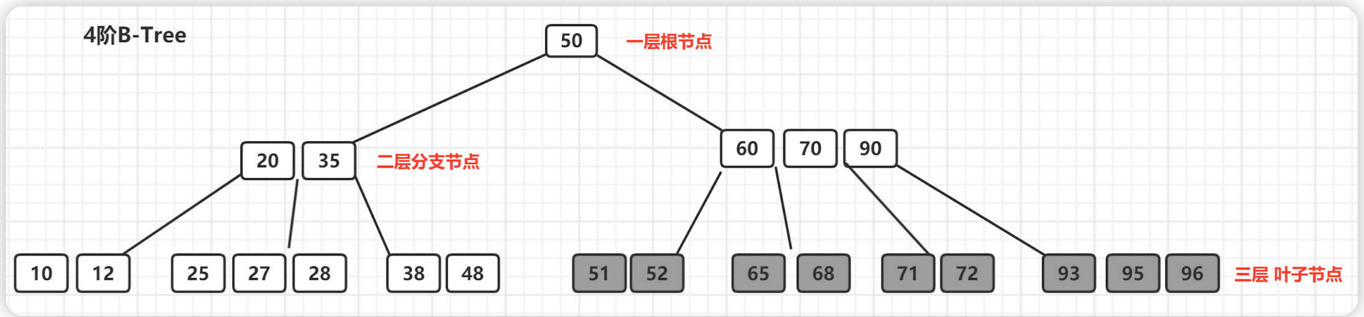
B-Tree是一种平衡的多路查找树,B树允许一个节点存放多个数据. 这样可以在尽可能减少树的深度的同时,存放更多的数据(把瘦高的树变的矮胖).

B-Tree中所有节点的子树个数的最大值称为B-Tree的阶,用m表示.一颗m阶的B树,如果不为空,就必须满足以下条件.

m阶的B-Tree满足以下条件:

1. 每个节点最多拥有m-1个关键字(根节点除外),也就是m个子树
2. 根节点至少有两个子树(可以没有子树,有就必须是两个)

3. 分支节点至少有(m/2)颗子树 (除去根节点和叶子节点其他都是分支节点)
4. 所有叶子节点都在同一层,并且以升序排序



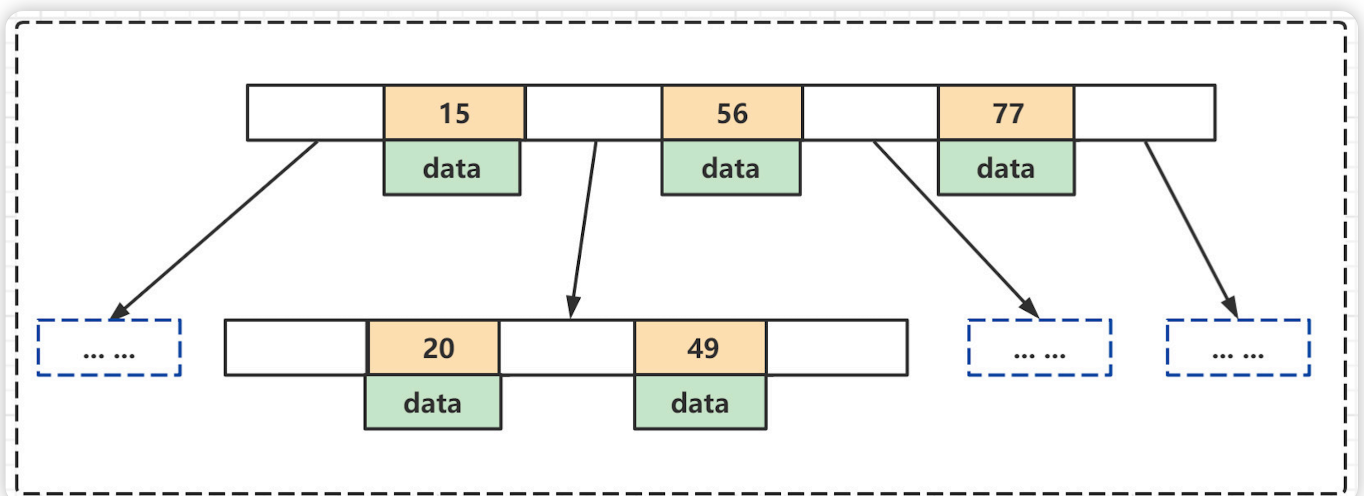
什么是B-Tree的阶？

所有节点中，节点【60,70,90】拥有的子节点数目最多，四个子节点（灰色节点），所以上面的B-Tree为4阶B树。

B-Tree结构存储索引的特点

为了描述B-Tree首先定义一条记录为一个键值对[key, data]，key为记录的键值，对应表中的主键值(聚簇索引)，data为一行记录中除主键外的数据。对于不同的记录，key值互不相同

- 索引值和data数据分布在整棵树结构中
- 白色块部分是指针,存储着子节点的地址信息。
- 每个节点可以存放多个索引值及对应的data数据
- 树节点中的多个索引值从左到右升序排列



B-Tree的查找操作

B-Tree的每个节点的元素可以视为一次I/O读取，树的高度表示最多的I/O次数，在相同数量的总元素个数下，每个节点的元素个数越多，高度越低，查询所需的I/O次数越少.

B-Tree总结

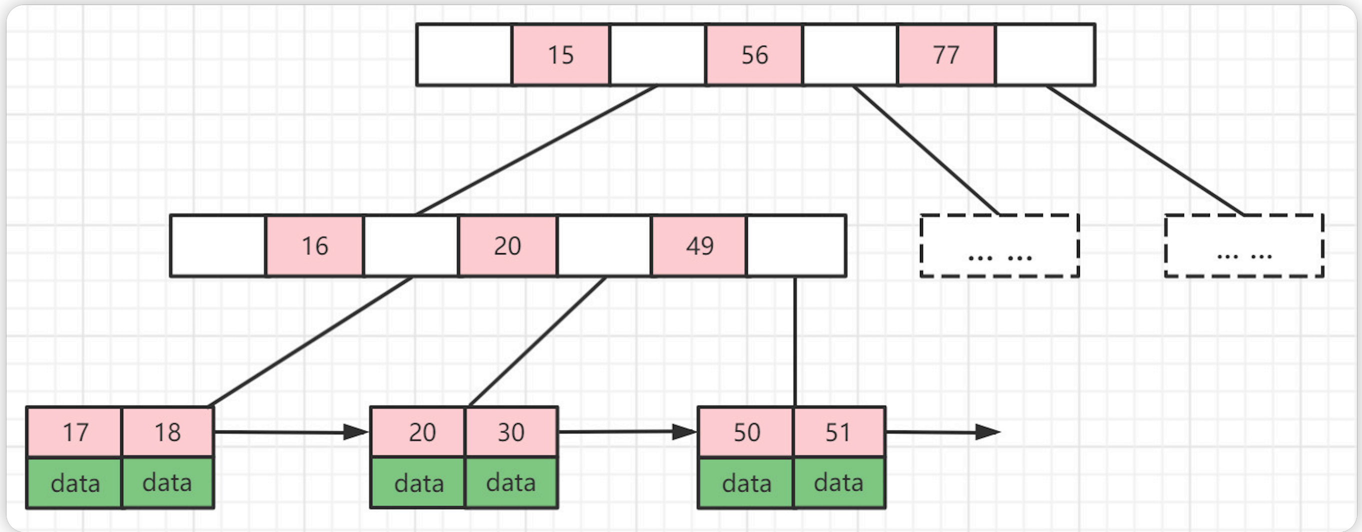
- 优点: B树可以在内部节点存储键值和相关记录数据，因此把频繁访问的数据放在靠近根节点的位置将大大提高热点数据的查询效率。
- 缺点: B树中每个节点不仅包含数据的key值,还有data数据. 所以当data数据较大时,会导致每个节点存储的key值减少,并且导致B树的层数变高. 增加查询时的IO次数.
- 使用场景: B树主要应用于文件系统以及部分数据库索引，如MongoDB，大部分关系型数据库索引则是使用B+树实现

2）B+Tree

B+Tree是在B-Tree基础上的一种优化，使其更适合实现存储索引结构，InnoDB存储引擎就是用B+Tree实现其索引结构。

B+Tree的特征

- 非叶子节点只存储键值信息.
- 所有叶子节点之间都有一个链指针.
- 数据记录都存放在叶子节点中.

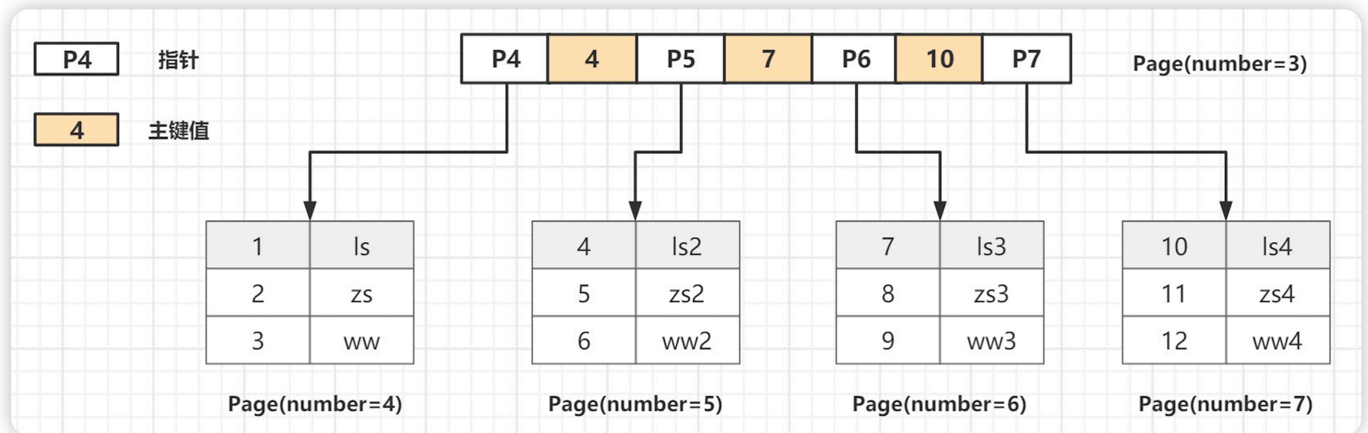


B+Tree的优势

1. B+Tree是B Tree的变种，B Tree能解决的问题，B+Tree也能够解决（降低树的高度，增大节点存储数据量）
2. B+Tree扫库和扫表能力更强，如果我们要根据索引去进行数据表的扫描，对B Tree进行扫描，需要把整棵树遍历一遍，而B+Tree只需要遍历他的所有叶子节点即可（叶子节点之间有引用）。
3. B+Tree磁盘读写能力更强，他的根节点和支节点不保存数据区，所有根节点和支节点同样大小的情况下，保存的关键字要比B Tree要多。而叶子节点不保存子节点引用。所以，B+Tree读写一次磁盘加载的关键字比B Tree更多。
4. B+Tree排序能力更强，如上面的图中可以看出，B+Tree天然具有排序功能。
5. B+Tree查询效率更加稳定，每次查询数据，查询IO次数一定是稳定的。当然这个每个人的理解都不同，因为在B Tree如果根节点命中直接返回，确实效率更高。

16.一个B+树中大概能存放多少条索引记录?

MySQL设计者将一个B+Tree的节点的大小设置为等于一个页. (这样做的目的是每个节点只需要一次I/O就可以完全载入), InnoDB的一个页的大小是16KB,所以每个节点的大小也是16KB, 并且B+Tree的根节点是保存在内存中的,子节点才是存储在磁盘上.



假设一个B+树高为2，即存在一个根节点和若干个叶子节点，那么这棵B+树的存放总记录数为：

根节点指针数*单个叶子节点记录行数.

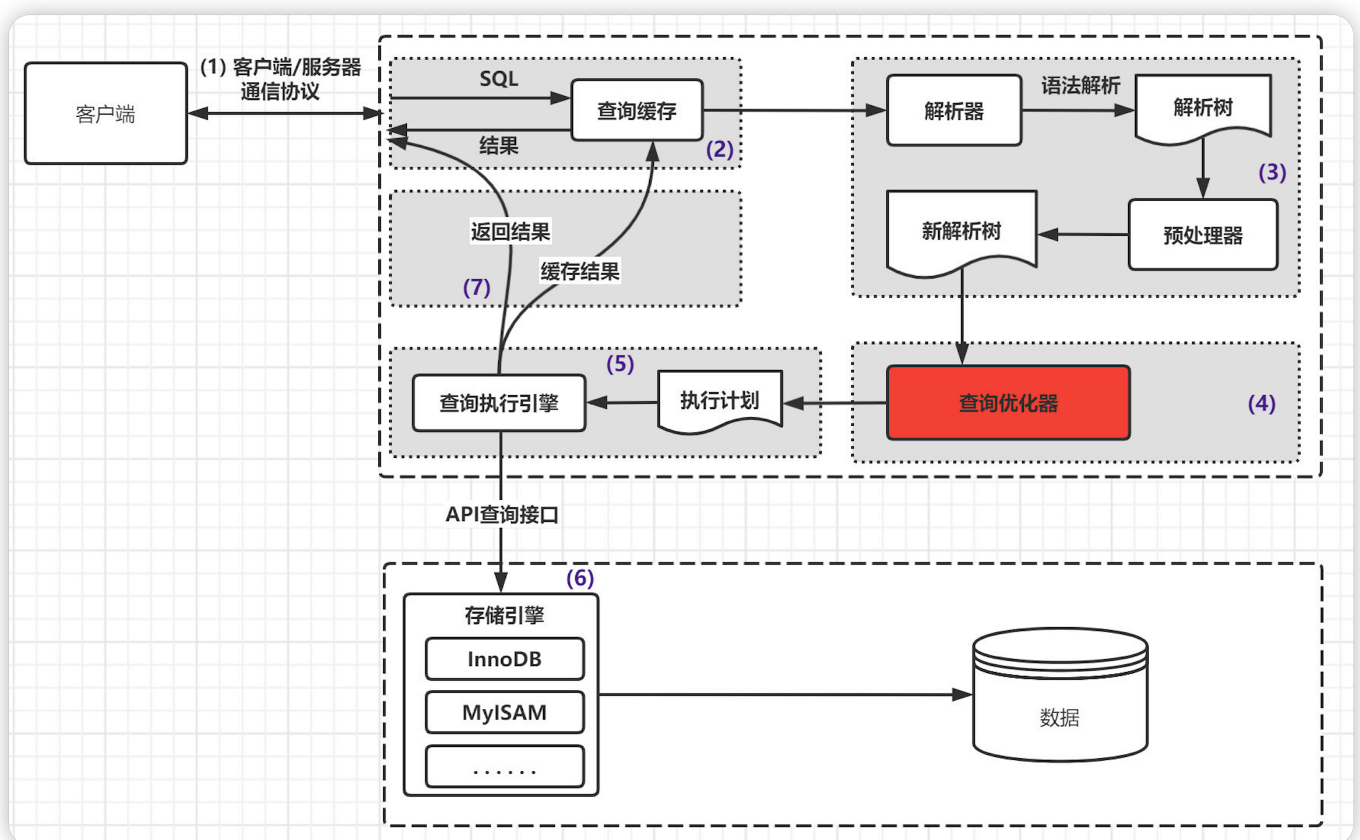
- **计算根节点指针数**: 假设表的主键为INT类型,占用的就是4个字节,或者是BIGINT占用8个字节, 指针大小为6个字节,那么一个页(就是B+Tree中的一个节点) ,大概可以存储: 16384B / (4B+6B) = 1638 ,一个节点最多可以存储1638个索引指针.
- **计算每个叶子节点的记录数**:我们假设一行记录的数据大小为1k,那么一页就可以存储16行数据,16KB / 1KB = 16.
- **一颗高度为2的B+Tree可以存放的记录数为**: 1638 * 16=26208 条数据记录, 同样的原理可以推算出一个高度3的B+Tree可以存放: 1638 * 1638 * 16 = 42928704条这样的记录.

所以InnoDB中的B+Tree高度一般为1-3层,就可以满足千万级别的数据存储,在查找数据时一次页的查找代表一次 IO，所以通过主键索引查询通常只需要 1-3 次 IO 操作即可查找到数据。

17.explain 用过吗，有哪些主要字段?

使用 `EXPLAIN` 关键字可以模拟优化器来执行SQL查询语句，从而知道MySQL是如何处理我们的SQL语句的。分析出查询语句或是表结构的性能瓶颈。

MySQL查询过程



通过explain我们可以获得以下信息：

- 表的读取顺序
- 数据读取操作的操作类型
- 哪些索引可以被使用
- 哪些索引真正被使用
- 表的直接引用

- **每张表的有多少行被优化器查询了**

Explain使用方式: **explain+sql语句**, 通过执行explain可以获得sql语句执行的相关信息

```
explain select * from users;
```

18.type字段中有哪些常见的值?

type字段在 MySQL 官网文档描述如下:

> **The join type. For descriptions of the different types.**

type字段显示的是连接类型 (join type表示的是用什么样的方式来获取数据), 它描述了找到所需数据所使用的扫描方式, 是较为重要的一个指标。

下面给出各种连接类型,按照从最佳类型到最坏类型进行排序:

```
-- 完整的连接类型比较多
system > const > eq_ref > ref > fulltext > ref_or_null >
index_merge > unique_subquery > index_subquery > range >
index > ALL

-- 简化之后,我们可以只关注一下几种
system > const > eq_ref > ref > range > index > ALL
```

> **一般来说,需要保证查询至少达到 range级别,最好能到ref,否则就要就行SQL的优化调整**

下面介绍type字段不同值表示的含义:

type类型	解释
system	不进行磁盘IO,查询系统表,仅仅返回一条数据
const	查找主键索引,最多返回1条或0条数据. 属于精确查找
eq_ref	查找唯一性索引,返回数据最多一条, 属于精确查找
ref	查找非唯一性索引,返回匹配某一条件的多条数据,属于精确查找,数据返回可能是多条.
range	查找某个索引的部分索引,只检索给定范围的行,属于范围查找. 比如: > 、 < 、in 、between
index	查找所有索引树,比ALL快一些,因为索引文件要比数据文件小.
ALL	不使用任何索引,直接进行全表扫描

19.Extra有哪些主要指标，各自的含义是什么?

Extra 是 EXPLAIN 输出中另外一个很重要的列，该列显示MySQL在查询过程中的一些详细信息

extra类型	解释
Using filesort	MySQL中无法利用索引完成的排序操作称为 “文件排序”
Using index	表示直接访问索引就能够获取到所需要的数据（覆盖索引），不需要通过索引回表
Using index condition	搜索条件中虽然出现了索引列，但是有部分条件无法使用索引， 会根据能用索引的条件先搜索一遍再匹配无法使用索引的条件。
Using join buffer	使用了连接缓存, 会显示join连接查询时,MySQL选择的查询算法
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
Using where	意味着全表扫描或者在查找使用索引的情况下，但是还有查询条件不在索引字段当中

20.如何进行分页查询优化?

- 一般性分页

  一般的分页查询使用简单的 limit 子句就可以实现。limit格式如下：

  ```
  SELECT * FROM 表名 LIMIT [offset,] rows
  ```

  - 第一个参数指定第一个返回记录行的偏移量，注意从0开始；
  - 第二个参数指定返回记录行的最大数目；
  - 如果只给定一个参数，它表示返回最大的记录行数目；

  思考1：如果偏移量固定，返回记录量对执行时间有什么影响?

```
select * from user limit 10000,1;
select * from user limit 10000,10;
select * from user limit 10000,100;
select * from user limit 10000,1000;
select * from user limit 10000,10000;
```

结果：在查询记录时，返回记录量低于100条，查询时间基本没有变化，差距不大。随着查询记录量越大，所花费的时间也会越来越多。

思考2：如果查询偏移量变化，返回记录数固定对执行时间有什么影响?

```
select * from user limit 1,100;
select * from user limit 10,100;
select * from user limit 100,100;
select * from user limit 1000,100;
select * from user limit 10000,100;
```

结果：在查询记录时，如果查询记录量相同，偏移量超过100后就开始随着偏移量增大，查询时间急剧的增加。（这种分页查询机制，每次都会从数据库第一条记录开始扫描，越往后查询越慢，而且查询的数据越多，也会拖慢总查询速度。）

- 分页优化方案

  优化1: 通过索引进行分页

  直接进行limit操作 会产生全表扫描,速度很慢. Limit限制的是从结果集的M位置处取出N条输出,其余抛弃.

  假设ID是连续递增的,我们根据查询的页数和查询的记录数可以算出查询的id的范围，然后配合 limit使用

```
EXPLAIN SELECT * FROM user WHERE id  >= 100001 LIMIT
100;
```

优化2：利用子查询优化

```
-- 首先定位偏移位置的id
SELECT id FROM user_contacts LIMIT 100000,1;

-- 根据获取到的id值向后查询.
EXPLAIN SELECT * FROM user_contacts WHERE id >=
(SELECT id FROM user_contacts LIMIT 100000,1) LIMIT 100;
```

原因：使用了id做主键比较(id>=)，并且子查询使用了覆盖索引进行优化。

21.如何做慢查询优化?

MySQL 慢查询的相关参数解释：

- **slow_query_log**：是否开启慢查询日志，`ON(1)`表示开启，`OFF(0)` 表示关闭。
- **slow-query-log-file**：新版（5.6及以上版本）MySQL数据库慢查询日志存储路径。
- **long_query_time**： 慢查询**阈值**，当查询时间多于设定的阈值时，记录日志。

慢查询配置方式

1. **默认情况下slow_query_log的值为OFF，表示慢查询日志是禁用的**

```
mysql> show variables like '%slow_query_log%';
+---------------------+-------------------------------+
| Variable_name       | Value                         |
+---------------------+-------------------------------+
| slow_query_log      | ON                            |
| slow_query_log_file | /var/lib/mysql/test-slow.log |
+---------------------+-------------------------------+
```

2. **可以通过设置slow_query_log的值来开启**

```
mysql> set global slow_query_log=1;
```

3. **使用** `set global slow_query_log=1` 开启了慢查询日志只对当前数据库生效，MySQL重启后则会失效。如果要永久生效，就必须修改配置文件my.cnf（其它系统变量也是如此）

```
-- 编辑配置
vim /etc/my.cnf

-- 添加如下内容
slow_query_log =1
slow_query_log_file=/var/lib/mysql/ruyuan-slow.log

-- 重启MySQL
service mysqld restart

mysql> show variables like '%slow_query%';
+---------------------+---------------------------------+
| Variable_name       | Value                           |
+---------------------+---------------------------------+
| slow_query_log      | ON                              |
| slow_query_log_file | /var/lib/mysql/ruyuan-slow.log |
+---------------------+---------------------------------+
```

4. 那么开启了慢查询日志后，什么样的SQL才会记录到慢查询日志里面呢？ 这个是由参数 `long_query_time`控制，默认情况下long_query_time的值为10秒.

```
mysql> show variables like 'long_query_time';
+-----------------+-----------+
| Variable_name   | Value     |
+-----------------+-----------+
| long_query_time | 10.000000 |
+-----------------+-----------+

mysql> set global long_query_time=1;
Query OK, 0 rows affected (0.00 sec)

mysql>  show variables like 'long_query_time';
+-----------------+-----------+
| Variable_name   | Value     |
+-----------------+-----------+
| long_query_time | 10.000000 |
+-----------------+-----------+
```

5. **修改了变量long_query_time，但是查询变量long_query_time的值还是10，难道没有修改到呢？注意：使用命令 set global long_query_time=1 修改后，需要重新连接或新开一个会话才能看到修改值。**

```
mysql> show variables like 'long_query_time';
+-----------------+----------+
| Variable_name   | Value    |
+-----------------+----------+
| long_query_time | 1.000000 |
+-----------------+----------+
```

6. `log_output` 参数是指定日志的存储方式。`log_output='FILE'` 表示将日志存入文件，默认值是'FILE'。`log_output='TABLE'` 表示将日志存入数据库，这样日志信息就会被写入到 mysql.slow_log 表中。

```
mysql> SHOW VARIABLES LIKE '%log_output%';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| log_output    | FILE  |
+---------------+-------+
```

> **MySQL数据库支持同时两种日志存储方式，配置的时候以逗号隔开即可，如：log_output='FILE,TABLE'。日志记录到系统的专用日志表中，要比记录到文件耗费更多的系统资源，因此对于需要启用慢查询日志，又需要能够获得更高的系统性能，那么建议优先记录到文件.**

7. 系统变量 `log-queries-not-using-indexes`：未使用索引的查询也被记录到慢查询日志中（可选项）。如果调优的话，建议开启这个选项。

```
mysql> show variables like
'log_queries_not_using_indexes';
+-------------------------------+-------+
| Variable_name                 | Value |
+-------------------------------+-------+
| log_queries_not_using_indexes | OFF   |
+-------------------------------+-------+

mysql> set global log_queries_not_using_indexes=1;
Query OK, 0 rows affected (0.00 sec)

mysql> show variables like
'log_queries_not_using_indexes';
+-------------------------------+-------+
```

```
| Variable_name                 | Value |
+-------------------------------+-------+
| log_queries_not_using_indexes | ON    |
+-------------------------------+-------+
```

3) 慢查询测试

1. **执行 test_index.sql 脚本,监控慢查询日志内容**

```
[root@localhost mysql]# tail -f /var/lib/mysql/ruyuan-slow.log
/usr/sbin/mysqld, Version: 5.7.30-log (MySQL Community Server (GPL)). started with:
Tcp port: 0  Unix socket: /var/lib/mysql/mysql.sock
Time                 Id Command    Argument
```

2. **执行下面的SQL,执行超时 (超过1秒) 我们去查看慢查询日志**

```
SELECT * FROM test_index WHERE
hobby = '20009951' OR hobby = '10009931' OR hobby = '30009931'
OR dname = 'name4000' OR dname = 'name6600' ;
```

3. **日志内容**

我们得到慢查询日志后，最重要的一步就是去分析这个日志。我们先来看下慢日志里到底记录了哪些内容。

如下图是慢日志里其中一条SQL的记录内容，可以看到有时间戳，用户，查询时长及具体的SQL等信息.

```
# Time: 2022-02-23T13:50:45.005959Z
# User@Host: root[root] @ localhost []  Id:     3
# Query_time: 3.724273  Lock_time: 0.000371 Rows_sent: 5
Rows_examined: 5000000
SET timestamp=1645624245;
select * from test_index where hobby = '20009951' or hobby
= '10009931' or hobby = '30009931' or dname = 'name4000'
or dname = 'name6600';
```

- **Time: 执行时间**
- **User: 用户信息 ,Id信息**
- **Query_time: 查询时长**
- **Lock_time: 等待锁的时长**
- **Rows_sent:查询结果的行数**
- **Rows_examined: 查询扫描的行数**
- **SET timestamp: 时间戳**
- **SQL的具体信息**

慢查询SQL优化思路

1) SQL性能下降的原因

在日常的运维过程中，经常会遇到DBA将一些执行效率较低的SQL发过来找开发人员分析，当我们拿到这个SQL语句之后，在对这些SQL进行分析之前，需要明确可能导致SQL执行性能下降的原因进行分析，执行性能下降可以体现在以下两个方面：

- **等待时间长**

  ```
  锁表导致查询一直处于等待状态，后续我们从MySQL锁的机制去分析SQL执行的原理
  ```

- **执行时间长**

```
1.查询语句写的烂
2.索引失效
3.关联查询太多join
4.服务器调优及各个参数的设置
```

2) 慢查询优化思路

1. 优先选择优化高并发执行的SQL,因为高并发的SQL发生问题带来后果更严重.

```
比如下面两种情况:
   SQL1: 每小时执行10000次, 每次20个IO 优化后每次18个IO,每小时节省2万次IO
   SQL2: 每小时10次,每次20000个IO,每次优化减少2000个IO,每小时节省2万次IO
   SQL2更难优化,SQL1更好优化.但是第一种属于高并发SQL,更急需优化 成本更低
```

2. 定位优化对象的性能瓶颈(在优化之前了解性能瓶颈在哪)

```
在去优化SQL时,选择优化分方向有三个:
  1.IO(数据访问消耗的了太多的时间,查看是否正确使用了索引) ,
  2.CPU(数据运算花费了太多时间, 数据的运算分组 排序是不是有问题)
  3.网络带宽(加大网络带宽)
```

3. 明确优化目标

```
需要根据数据库当前的状态
数据库中与该条SQL的关系
当前SQL的具体功能
最好的情况消耗的资源,最差情况下消耗的资源,优化的结果只有一个给用户一个好的体验
```

4. 从explain执行计划入手

```
只有explain能告诉你当前SQL的执行状态
```

5. 永远用小的结果集驱动大的结果集

```
小的数据集驱动大的数据集,减少内层表读取的次数

类似于嵌套循环
for(int i = 0; i < 5; i++){
  for(int i = 0; i < 1000; i++){

  }
}
如果小的循环在外层,对于数据库连接来说就只连接5次,进行5000次操作,如果1000在外,则需要进行1000次数据库连接,从而浪费资源, 增加消耗.这就是为什么要小表驱动大表。
```

6. 尽可能在索引中完成排序

```
排序操作用的比较多,order by 后面的字段如果在索引中,索引本来就是排好序的,所以速度很快,没有索引的话,就需要从表中拿数据,在内存中进行排序,如果内存空间不够还会发生落盘操作
```

7. 只获取自己需要的列

```
不要使用select  * ,select * 很可能不走索引,而且数据量过大
```

8. 只使用最有效的过滤条件

```
误区 where后面的条件越多越好,但实际上是应该用最短的路径访问到数据
```

9. 尽可能避免复杂的join和子查询

```
每条SQL的JOIN操作 建议不要超过三张表
将复杂的SQL, 拆分成多个小的SQL 单个表执行,获取的结果 在程序中进行
封装
如果join占用的资源比较多,会导致其他进程等待时间变长
```

0. 合理设计并利用索引

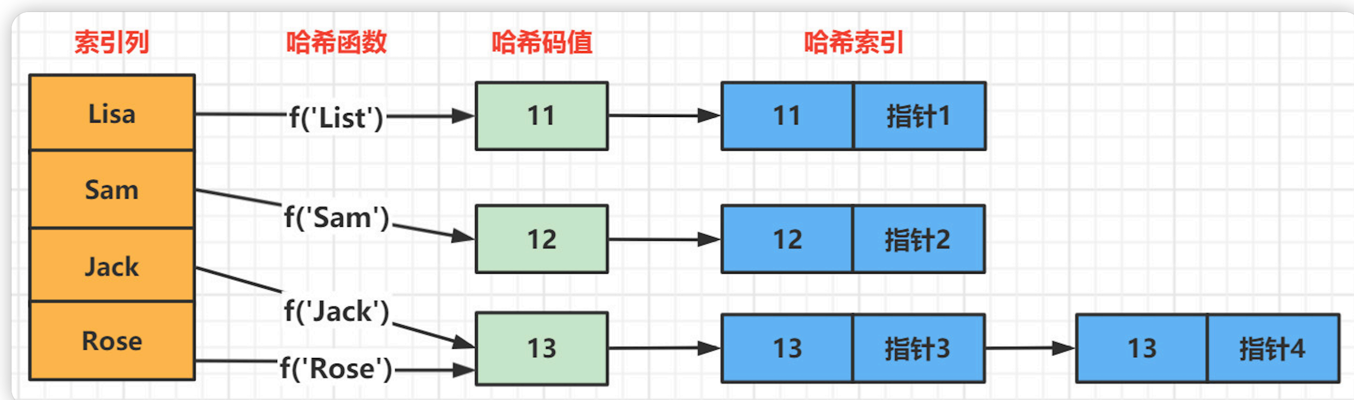
```
如何判定是否需要创建索引?
  1.较为频繁的作为查询条件的字段应该创建索引.
  2.唯一性太差的字段不适合单独创建索引，即使频繁作为查询条件.（唯一
性太差的字段主要是指哪些呢？如状态字段，类型字段等等这些字段中的数
据可能总共就是那么几个几十个数值重复使用）（当一条Query所返回的数
据超过了全表的15%的时候，就不应该再使用索引扫描来完成这个Query
了）.
  3.更新非常频繁的字段不适合创建索引.（因为索引中的字段被更新的时
候，不仅仅需要更新表中的数据，同时还要更新索引数据，以确保索引信息
是准确的）.
  4.不会出现在WHERE子句中的字段不该创建索引.

如何选择合适索引?
  1.对于单键索引，尽量选择针对当前Query过滤性更好的索引.
  2.选择联合索引时,当前Query中过滤性最好的字段在索引字段顺序中排列
要靠前.
  3.选择联合索引时,尽量索引字段出现在w中比较多的索引.
```

22.Hash索引有哪些优缺点?

MySQL中索引的常用数据结构有两种: 一种是B+Tree,另一种则是Hash.

Hash底层实现是由Hash表来实现的，是根据键值 <key,value> 存储数据的结构。非常适合根据key查找value值，也就是单个key查询，或者说等值查询。



对于每一行数据，存储引擎都会对所有的索引列计算一个哈希码，哈希码是一个较小的值,如果出现哈希码值相同的情况会拉出一条链表.

Hsah索引的优点

- 因为索引自身只需要存储对应的Hash值,所以索引结构非常紧凑, 只需要做等值比较查询，而不包含排序或范围查询的需求，都适合使用哈希索引 .
- 没有哈希冲突的情况下,等值查询访问哈希索引的数据非常快.(如果发生Hash冲突,存储引擎必须遍历链表中的所有行指针,逐行进行比较,直到找到所有符合条件的行).

Hash索引的缺点

- 哈希索引只包含哈希值和行指针，而不存储字段值，所以不能使用索引中的值来避免读取行。
- 哈希索引只支持等值比较查询。不支持任何范围查询和部分索引列匹配查找。
- 哈希索引数据并不是按照索引值顺序存储的，所以也就无法用于排序。

23.说一下InnoDB内存相关的参数优化?

Buffer Pool参数优化

1.1 缓冲池内存大小配置

一个大的日志缓冲区允许大量的事务在提交之前不写日志到磁盘。因此，如果你有很多事务的更新，插入或删除操作，通过设置这个参数会大量的减少磁盘I/O的次数数。
建议: 在专用数据库服务器上，可以将缓冲池大小设置为服务器物理内存的60% - 80%.

- **查看缓冲池大小**

```
mysql> show variables like '%innodb_buffer_pool_size%';
+-------------------------+-----------+
| Variable_name           | Value     |
+-------------------------+-----------+
| innodb_buffer_pool_size | 134217728 |
+-------------------------+-----------+

mysql> select 134217728 / 1024 / 1024;
+-------------------------+
| 134217728 / 1024 / 1024 |
+-------------------------+
|            128.00000000 |
+-------------------------+
```

- **在线调整InnoDB缓冲池大小**
  innodb_buffer_pool_size可以动态设置，允许在不重新启动服务器的情况下调整缓冲池的大小.

```
mysql> SET GLOBAL innodb_buffer_pool_size = 268435456; -
- 512
Query OK, 0 rows affected (0.10 sec)

mysql> show variables like '%innodb_buffer_pool_size%';
+-------------------------+-----------+
| Variable_name           | Value     |
+-------------------------+-----------+
| innodb_buffer_pool_size | 268435456 |
+-------------------------+-----------+
```

监控在线调整缓冲池的进度

```
mysql> SHOW STATUS WHERE
Variable_name='InnoDB_buffer_pool_resize_status';
+----------------------------------+--------------------
--------------------------------------------------+
| Variable_name                    | Value
                                                  |
+----------------------------------+--------------------
--------------------------------------------------+
| Innodb_buffer_pool_resize_status | Size did not change
(old size = new size = 268435456. Nothing to do. |
+----------------------------------+--------------------
--------------------------------------------------+
```

1.3 InnoDB 缓存性能评估

当前配置的innodb_buffer_pool_size是否合适，可以通过分析InnoDB缓冲池的缓存命中率来验证。

- 以下公式计算InnoDB buffer pool 命中率:

```
命中率 = innodb_buffer_pool_read_requests /
(innodb_buffer_pool_read_requests+innodb_buffer_pool_rea
ds)* 100

参数1: innodb_buffer_pool_reads: 表示InnoDB缓冲池无法满足的
请求数。需要从磁盘中读取。
参数2: innodb_buffer_pool_read_requests: 表示从内存中读取页
的请求数。
```

```
mysql> show status like 'innodb_buffer_pool_read%';
+---------------------------------------+-------+
| Variable_name                         | Value |
+---------------------------------------+-------+
| Innodb_buffer_pool_read_ahead_rnd     | 0     |
| Innodb_buffer_pool_read_ahead         | 0     |
| Innodb_buffer_pool_read_ahead_evicted | 0     |
| Innodb_buffer_pool_read_requests      | 12701 |
| Innodb_buffer_pool_reads              | 455   |
+---------------------------------------+-------+

-- 此值低于90%，则可以考虑增加innodb_buffer_pool_size。
mysql> select 12701 / (455 + 12701) * 100 ;
+-----------------------------+
| 12701 / (455 + 12701) * 100 |
+-----------------------------+
|                     96.5415 |
+-----------------------------+
```

1.4 Page管理相关参数

查看Page页的大小(默认16KB),`innodb_page_size`只能在初始化MySQL实例之前配置，不能在之后修改。如果没有指定值，则使用默认页面大小初始化实例。

```
mysql> show variables like '%innodb_page_size%';
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| innodb_page_size | 16384 |
+------------------+-------+
```

Page页管理状态相关参数

```
mysql> show global status like
'%innodb_buffer_pool_pages%';
+----------------------------------+-------+
| Variable_name                    | Value |
+----------------------------------+-------+
| Innodb_buffer_pool_pages_data    | 515   |
| Innodb_buffer_pool_pages_dirty   | 0     |
| Innodb_buffer_pool_pages_flushed | 334   |
| Innodb_buffer_pool_pages_free    | 15868 |
| Innodb_buffer_pool_pages_misc    | 0     |
| Innodb_buffer_pool_pages_total   | 16383 |
+----------------------------------+-------+
```

pages_data: InnoDB缓冲池中包含数据的页数。 该数字包括脏页面和干净页面。

pages_dirty: 显示在内存中修改但尚未写入数据文件的InnoDB缓冲池数据页的数量（脏页刷新）。

pages_flushed: 表示从InnoDB缓冲池中刷新脏页的请求数。

pages_free: 显示InnoDB缓冲池中的空闲页面

pages_misc: 缓存池中当前已经被用作管理用途或hash index而不能用作为普通数据页的数目

pages_total: 缓存池的页总数目。单位是page。

24.InnoDB日志相关的参数优化了解过吗?

1.日志缓冲区相关参数配置

日志缓冲区的大小。一般默认值16MB是够用的，但如果事务之中含有blog/text等大字段，这个缓冲区会被很快填满会引起额外的IO负载。配置更大的日志缓冲区,可以有效的提高MySQL的效率.

- **innodb_log_buffer_size 缓冲区大小**

  ```
  mysql> show variables like 'innodb_log_buffer_size';
  +------------------------+----------+
  | Variable_name          | Value    |
  +------------------------+----------+
  | innodb_log_buffer_size | 16777216 |
  +------------------------+----------+
  ```

- **innodb_log_files_in_group 日志组文件个数**
  日志组根据需要来创建。而日志组的成员则需要至少2个，实现循环写入并作为冗余策略。

  ```
  mysql> show variables like 'innodb_log_files_in_group';
  +---------------------------+-------+
  | Variable_name             | Value |
  +---------------------------+-------+
  | innodb_log_files_in_group | 2     |
  +---------------------------+-------+
  ```

- **innodb_log_file_size 日志文件大小**
  参数innodb_log_file_size用于设定MySQL日志组中每个日志文件的大小(默认48M)。此参数是一个全局的静态参数，不能动态修改。
  参数innodb_log_file_size的最大值，二进制日志文件大小（innodb_log_file_size * innodb_log_files_in_group）不能超过512GB.所以单个日志文件的大小不能超过256G.

```
mysql> show variables like 'innodb_log_file_size';
+----------------------+----------+
| Variable_name        | Value    |
+----------------------+----------+
| innodb_log_file_size | 50331648 |
+----------------------+----------+
```

2.日志文件参数优化

首先我们先来看一下日志文件大小设置对性能的影响

- 设置过小
  1. 参数 `innodb_log_file_size`设置太小，就会导致MySQL的日志文件(redo log）频繁切换，频繁的触发数据库的检查点（Checkpoint），导致刷新脏页到磁盘的次数增加。从而影响IO性能。
  2. 处理大事务时，将所有的日志文件写满了，事务内容还没有写完，这样就会导致日志不能切换.
- 设置过大
  参数 `innodb_log_file_size`如果设置太大，虽然可以提升IO性能，但是当MySQL由于意外宕机时，二进制日志很大，那么恢复的时间必然很长。而且这个恢复时间往往不可控，受多方面因素影响。

优化建议:

如何设置合适的日志文件大小？

- 根据实际生产场景的优化经验,一般是计算一段时间内生成的事务日志（redo log）的大小， 而MySQL的日志文件的大小最少应该承载一个小时的业务日志量(官网文档中有说明)。

想要估计一下InnoDB redo log的大小，需要抓取一段时间内Log SequenceNumber（日志顺序号）的数据,来计算一小时内产生的日志大小.

> **Log sequence number**
>
> **自系统修改开始，就不断的生成redo日志。为了记录一共生成了多少日志，于是mysql设计了全局变量log sequence number，简称lsn，但不是从0开始，是从8704字节开始。**

```
-- pager分页工具，只获取 sequence的信息
mysql> pager grep sequence;
PAGER set to 'grep sequence'

-- 查询状态,并倒计时一分钟
mysql> show engine innodb status\G select sleep(60);
Log sequence number 5399154
1 row in set (0.00 sec)

1 row in set (1 min 0.00 sec)

-- 一分时间内所生成的数据量 5406150
mysql> show engine innodb status\G;
Log sequence number 5406150

-- 关闭pager
mysql> nopager;
PAGER set to stdout
```

有了一分钟的日志量,据此推算一小时内的日志量

```
mysql> select (5406150 - 5399154) / 1024 as kb_per_min;
+------------+
| kb_per_min |
+------------+
|     6.8320 |
+------------+

mysql> select (5406150 - 5399154) / 1024 * 60 as
kb_per_min;
+------------+
| kb_per_min |
+------------+
|   409.9219 |
+------------+
```

太大的缓冲池或非常不正常的业务负载可能会计算出非常大(或非常小)的日志大小。这也是公式不足之处，需要根据判断和经验。但这个计算方法是一个很好的参考标准。

25.InnoDB IO线程相关参数优化了解过吗?

数据库属于 IO 密集型的应用程序，其主要职责就是数据的管理及存储工作。从内存中读取一个数据库数据的时间是微秒级别，而从一块普通硬盘上读取一个IO是在毫秒级别。要优化数据库，IO操作是必须要优化的，尽可能将磁盘IO转化为内存IO。

1) 参数: query_cache_size&have_query_cache

MySQL查询缓存会保存查询返回的完整结果。当查询命中该缓存，会立刻返回结果，跳过了解析，优化和执行阶段。

查询缓存会跟踪查询中涉及的每个表，如果这些表发生变化，那么和这个表相关的所有缓存都将失效。

1. 查看查询缓存是否开启

```
-- 查询是否支持查询缓存
mysql> show variables like 'have_query_cache';
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| have_query_cache | YES   |
+------------------+-------+

-- 查询是否开启查询缓存 默认关闭
mysql> show variables like '%query_cache_type%';
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| query_cache_type | OFF   |
+------------------+-------+
```

2. 开启缓存,在my.ini中添加下面一行参数

```
query_cache_size=128M
query_cache_type=1

query_cache_type:
设置为0，OFF,缓存禁用
设置为1，ON,缓存所有的结果
设置为2，DENAND,只缓存在select语句中通过SQL_CACHE指定需要缓存的查询
```

3. 测试能否缓存查询

```
mysql> show status like '%Qcache%';
+-------------------------+---------+
| Variable_name           | Value   |
+-------------------------+---------+
| Qcache_free_blocks      | 1       |
| Qcache_free_memory      | 1031832 |
| Qcache_hits             | 0       |
| Qcache_inserts          | 0       |
| Qcache_lowmem_prunes    | 0       |
| Qcache_not_cached       | 1       |
| Qcache_queries_in_cache | 0       |
| Qcache_total_blocks     | 1       |
+-------------------------+---------+
```

- **Qcache_free_blocks**:缓存中目前剩余的blocks数量（如果值较大，则查询缓存中的内存碎片过多）
- **Qcache_free_memory**:空闲缓存的内存大小
- **Qcache_hits**:命中缓存次数
- **Qcache_inserts**: 未命中然后进行正常查询
- **Qcache_lowmem_prunes**:查询因为内存不足而被移除出查询缓存记录

- **Qcache_not_cached**: 没有被缓存的查询数量
- **Qcache_queries_in_cache**:当前缓存中缓存的查询数量
- **Qcache_total_blocks**:当前缓存的block数量

优化建议: Query Cache的使用需要多个参数配合，其中最为关键的是 query_cache_size 和 query_cache_type ， 前者设置用于缓存 ResultSet 的内存大小，后者设置在何场景下使用 Query Cache。

MySQL数据库数据变化相对不多，query_cache_size 一般设置为256MB比较合适 ,也可以通过计算Query Cache的命中率来进行调整

```
( Qcache_hits / ( Qcache_hits + Qcache_inserts ) * 100) )
```

2. 参数: innodb_max_dirty_pages_pct 该参数是InnoDB 存储引擎用来控制buffer pool中脏页的百分比，当脏页数量占比超过这个参数设置的值时，InnoDB会启动刷脏页的操作。

```
-- innodb_max_dirty_pages_pct 参数可以动态调整，最小值为0， 最大值为99.99，默认值为 75。
mysql> show variables like 'innodb_max_dirty_pages_pct';
+----------------------------+-----------+
| Variable_name              | Value     |
+----------------------------+-----------+
| innodb_max_dirty_pages_pct | 75.000000 |
+----------------------------+-----------+
```

优化建议: 该参数比例值越大，从内存到磁盘的写入操作就会相对减少，所以能够一定程度下减少写入操作的磁盘IO。但是，如果这个比例值过大，当数据库 Crash 之后重启的时间可能就会很长，因为会有大量的事务数据需要从日志文件恢复出来写入数据文件中.最大不建议超过90,一般重启恢复的数据在超过1GB的话,启动速度就会变慢.

3) 参数: innodb_old_blocks_pct&innodb_old_blocks_time

`innodb_old_blocks_pct` 用来确定LRU链表中old sublist所占比例,默认占用37%

```
mysql> show variables like '%innodb_old_blocks_pct%';
+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| innodb_old_blocks_pct | 37    |
+-----------------------+-------+
```

`innodb_old_blocks_time` 用来控制old sublist中page的转移策略，新的page页在进入LRU链表中时，会先插入到old sublist的头部，然后page需要在old sublist中停留innodb_old_blocks_time这么久后，下一次对该page的访问才会使其移动到new sublist的头部，默认值1秒.

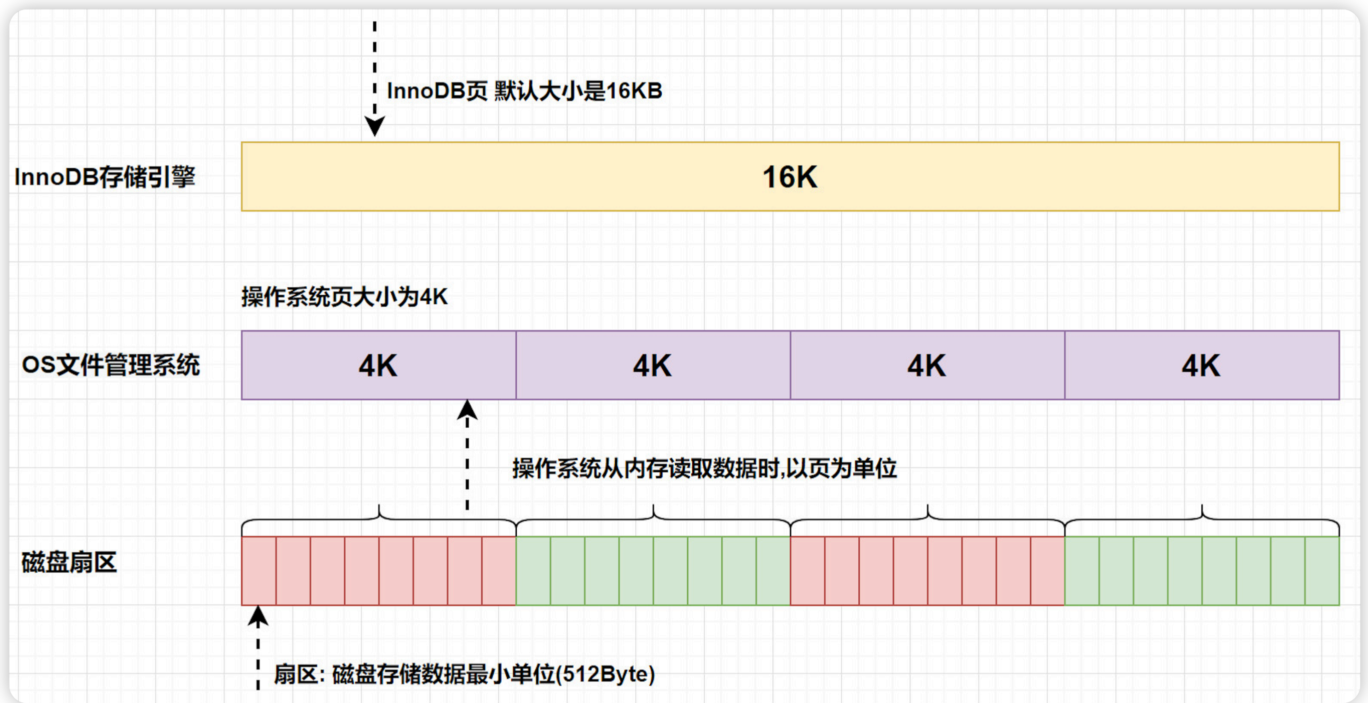
```
mysql> show variables like '%innodb_old_blocks_time%';
+------------------------+-------+
| Variable_name          | Value |
+------------------------+-------+
| innodb_old_blocks_time | 1000  |
+------------------------+-------+
```

优化建议: 在没有大表扫描的情况下，并且数据多为频繁使用的数据时，我们可以增加innodb_old_blocks_pct的值，并且减小innodb_old_blocks_time的值。让数据页能够更快和更多的进入的热点数据区。

26.什么是写失效?

InnoDB的页和操作系统的页大小不一致，InnoDB页大小一般为16K，操作系统页大小为4K，InnoDB的页写入到磁盘时，一个页需要分4次写。

如果存储引擎正在写入页的数据到磁盘时发生了宕机，可能出现页只写了一部分的情况，比如只写了4K，就宕机了，这种情况叫做部分写失效（partial page write），可能会导致数据丢失。



双写缓冲区 Doublewrite Buffer

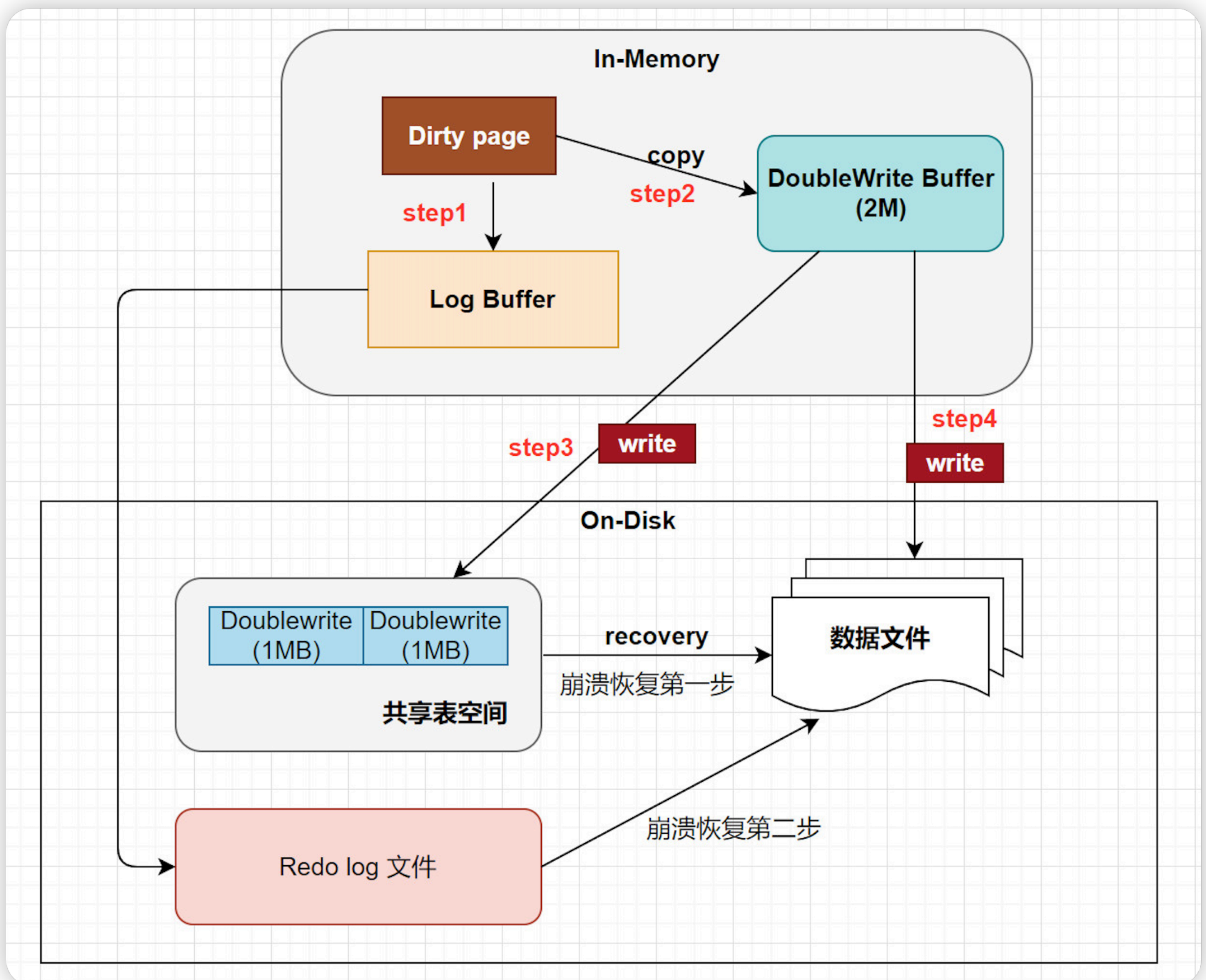
为了解决写失效问题，InnoDB实现了double write buffer Files, 它位于系统表空间，是一个存储区域。

在BufferPool的page页刷新到磁盘真正的位置前，会先将数据存在Doublewrite 缓冲区。这样在宕机重启时，如果出现数据页损坏，那么在应用redo log之前，需要通过该页的副本来还原该页，然后再进行redo log重做，double write实现了InnoDB引擎数据页的可靠性.

默认情况下启用双写缓冲区，如果要禁用Doublewrite 缓冲区，可以将`innodb_doublewrite`设置为0。

```
mysql> show variables like '%innodb_doublewrite%';
+--------------------+-------+
| Variable_name      | Value |
+--------------------+-------+
| innodb_doublewrite | ON    |
+--------------------+-------+
1 row in set (0.01 sec)
```

数据双写流程



- **step1**：当进行缓冲池中的脏页刷新到磁盘的操作时,并不会直接写磁盘,每次脏页刷新必须要先写double write .
- **step2**：通过memcpy函数将脏页复制到内存中的double write buffer .

- **step3**: double write buffer再分两次、每次1MB, 顺序写入共享表空间的物理磁盘上, **第一次写**.
- **step4**: 在完成double write页的写入后，再将double wirite buffer中的页写入各个表的**独立表空间**文件中(数据文件 .ibd), **第二次写**。

为什么写两次？

可能有的同学会有疑问，为啥写两次，刷一次数据文件保存数据不就可以了，为什么还要写共享表空间 ?其实是因为共享表空间是在ibdbata文件中划出2M连续的空间，专门给double write刷脏页用的, 由于在这个过程中, **double write页的存储是连续的，因此写入磁盘为顺序写，性能很高**；完成double write后，再将脏页写入实际的各个表空间文件，这时写入就是离散的了.

27.什么是行溢出?

行记录格式

1) 行格式分类

表的行格式决定了它的行是如何物理存储的，这反过来又会影响查询和DML操作的性能。如果在单个page页中容纳更多行，查询和索引查找可以更快地工作，缓冲池中所需的内存更少，写入更新时所需的I/O更少。

InnoDB存储引擎支持四种行格式：Redundant、Compact、Dynamic 和 Compressed .

查询MySQL使用的行格式,默认为: dynamic

```
mysql> show variables like 'innodb_default_row_format';
+---------------------------+---------+
| Variable_name             | Value   |
+---------------------------+---------+
| innodb_default_row_format | dynamic |
+---------------------------+---------+
```

指定行格式语法

```
CREATE TABLE <table_name(column_name)> ROW_FORMAT=行格式名称
ALTER TABLE <table_name> ROW_FORMAT=行格式名称
```

2) COMPACT 行记录格式

Compact 设计目标是高效地存储数据，一个页中存放的行数据越多，其性能就越高。

Compact行记录由两部分组成: 记录放入额外信息 和 记录的真实数据.

记录额外信息部分

服务器为了描述一条记录而添加了一些额外信息(元数据信息)，这些额外信息分为3类，分别是: 变长字段长度列表、NULL值列表和记录头信息.

- **变长字段长度列表**

  MySQL支持一些变长的数据类型，比如VARCHAR(M)、VARBINARY(M)、各种TEXT类型，各种BLOB类型，这些变长的数据类型占用的存储空间分为两部分：

1. 真正的数据内容
2. 占用的字节数

变长字段的长度是不固定的，所以在存储数据的时候要把这些数据占用的字节数也存起来，读取数据的时候才能根据这个长度列表去读取对应长度的数据。

在 `Compact`行格式中，把所有变长类型的列的长度都存放在记录的开头部位形成一个列表，按照列的顺序逆序存放,这个列表就是 **变长字段长度列表**。

- **NULL值列表**

  表中的某些列可能会存储NULL值，如果把这些NULL值都放到记录的真实数据中会比较浪费空间，所以Compact行格式把这些值为NULL的列存储到NULL值列表中。(如果表中所有列都不允许为 NULL，就不存在NULL值列表)

- **记录头信息**

  记录头信息是由固定的5个字节组成，5个字节也就是40个二进制位，不同的位代表不同的意思，这些头信息会在后面的一些功能中看到。

名称	大小(单位:bit)	描述
预留位1	1	没有使用
预留位2	1	没有使用
delete_mask	1	标记该记录是否被删除
min_rec_mask	1	标记该记录是否是本层B+树的非叶子节点中的最小记录
n_owned	4	表示当前分组中管理的记录数
heap_no	13	表示当前记录在记录堆中的位置信息
record_type	3	表示当前记录的类型: 0 表示普通记录, 1 表示B+树非叶子节点记录, 2 表示最小记录,3表示最大记录
next_record	16	表示下一条记录的相对位置

1. delete_mask

   这个属性标记着当前记录是否被删除，占用1个二进制位，值为0 的时候代表记录并没有被删除，为1 的时候代表记录被删除掉了

2. min_rec_mask

   B+树的每层非叶子节点中的最小记录都会添加该标记。

3. n_owned

   代表每个分组里，所拥有的记录的数量，一般是分组里主键最大值才有的。

4. heap_no

在数据页的User Records中插入的记录是一条一条紧凑的排列的，这种紧凑排列的结构又被称为堆。为了便于管理这个堆，把记录在堆中的相对位置给定一个编号——heap_no。所以heap_no这个属性表示当前记录在本页中的位置。

5. record_type

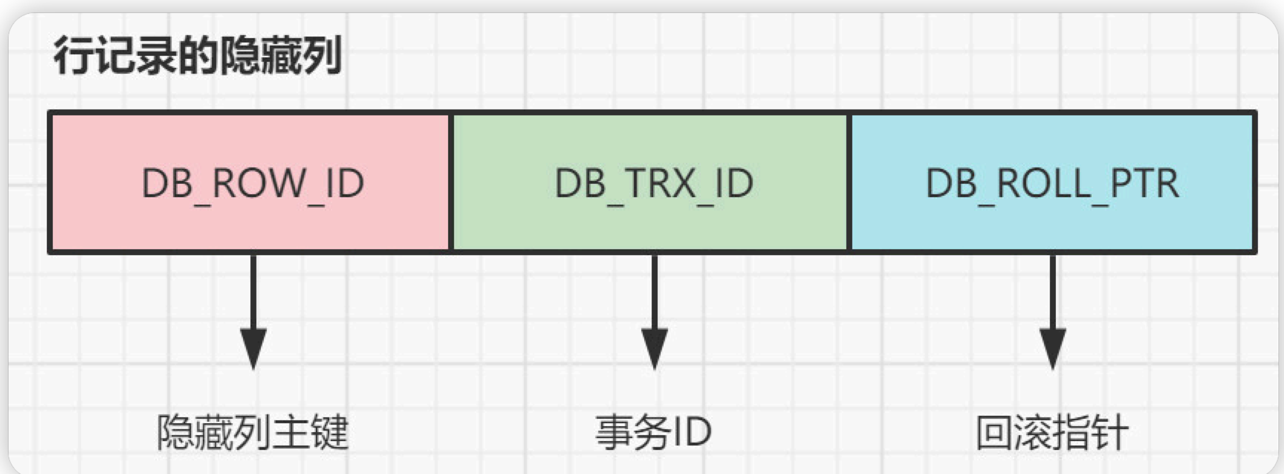
   这个属性表示当前记录的类型，一共有4种类型的记录， 0 表示普通用户记录， 1 表示B+树非叶节点记录， 2 表示最小记录， 3 表示最大记录。

6. next_record

   表示从当前记录的真实数据到下一条记录的真实数据的地址偏移量，可以理解为指向下一条记录地址的指针。值为正数说明下一条记录在当前记录后面，为负数说明下一条记录在当前记录的前面。

- **记录真实数据部分**

  记录的真实数据除了插入的那些列的数据，MySQL会为每个记录默认的添加一些列（也称为隐藏列），具体的列如下：



列名	是否必须	占用空间	描述
row_id	否	6字节	行ID,唯一标识一条记录
transaction_id	是	6字节	事务ID
roll_pointer	是	7字节	回滚指针

生成隐藏主键列的方式有:

```
1. 服务器会在内存中维护一个全局变量，每当向某个包含隐藏的row_id列的表中插入一条记录时，就会把该变量的值当作新记录的row_id列的值，并且把该变量自增1。
2. 每当这个变量的值为256的倍数时，就会将该变量的值刷新到系统表空间的页号为7的页面中一个Max Row ID的属性处。
3. 当系统启动时，会将页中的Max Row ID属性加载到内存中，并将该值加上256之后赋值给全局变量，因为在上次关机时该全局变量的值可能大于页中Max Row ID属性值。
   4.
```

3) Compact中的行溢出机制

什么是行溢出？

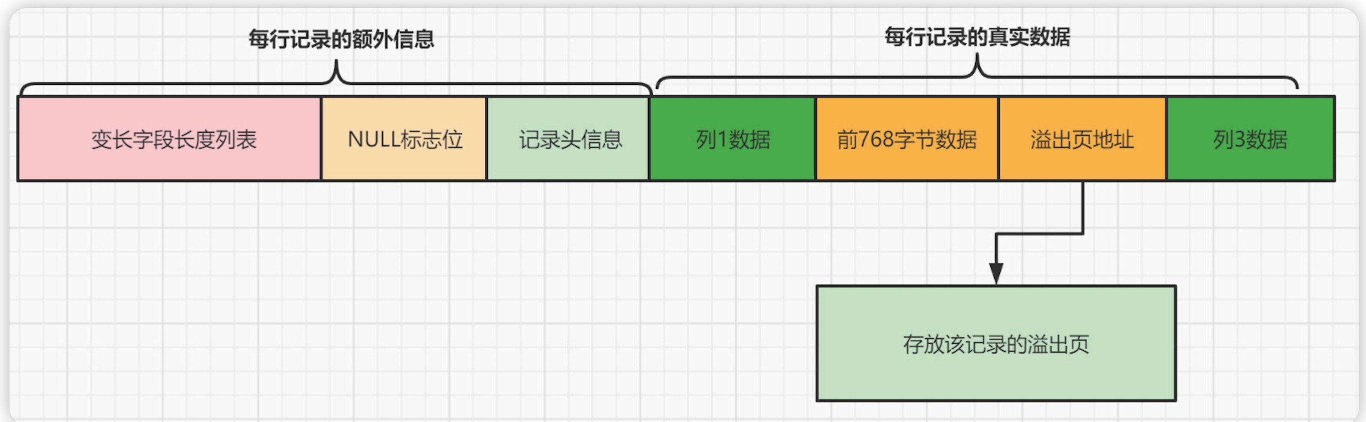
MySQL中是以页为基本单位,进行磁盘与内存之间的数据交互的,我们知道一个页的大小是16KB,16KB = 16384字节.而一个varchar(m) 类型列最多可以存储65532个字节,一些大的数据类型比如TEXT可以存储更多.

如果一个表中存在这样的大字段,那么一个页就无法存储一条完整的记录.这时就会发生行溢出,多出的数据就会存储在另外的溢出页中.

总结: 如果某些字段信息过长，无法存储在B树节点中，这时候会被单独分配空间，此时被称为溢出页，该字段被称为页外列。

Compact中的行溢出机制

InnoDB 规定一页至少存储两条记录(B+树特点)，如果页中只能存放下一条记录，InnoDB存储引擎会自动将行数据存放到溢出页中.
当发生行溢出时，数据页只保存了前768字节的前缀数据，接着是20个字节的偏移量，指向行溢出页.



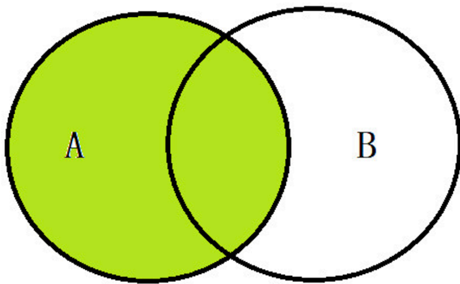
28.如何进行JOIN优化?

JOIN 是 MySQL 用来进行联表操作的，用来匹配两个表的数据，筛选并合并出符合我们要求的结果集。

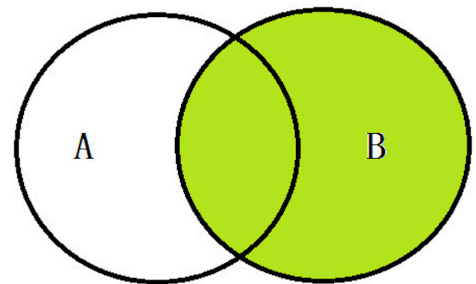
JOIN 操作有多种方式，取决于最终数据的合并效果。常用连接方式的有以下几种:

SQL JOINS

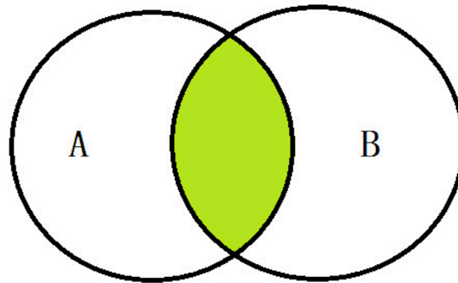
绿色区域是查询的结果集



LEFT JOIN
左外链接



RIGHT JOIN
右外链接



INNER JOIN
内连接

什么是驱动表？

- 多表关联查询时,第一个被处理的表就是驱动表,使用驱动表去关联其他表.
- 驱动表的确定非常的关键,会直接影响多表关联的顺序,也决定后续关联查询的性能

驱动表的选择要遵循一个规则:

- 在对最终的结果集没有影响的前提下,优先选择结果集最小的那张表作为驱动表

3) 三种JOIN算法

1.Simple Nested-Loop Join （简单的嵌套循环连接）

- **简单来说嵌套循环连接算法就是一个双层for 循环 ，通过循环外层表的行数据，逐个与内层表的所有行数据进行比较来获取结果.**
- **这种算法是最简单的方案，性能也一般。对内循环没优化。**

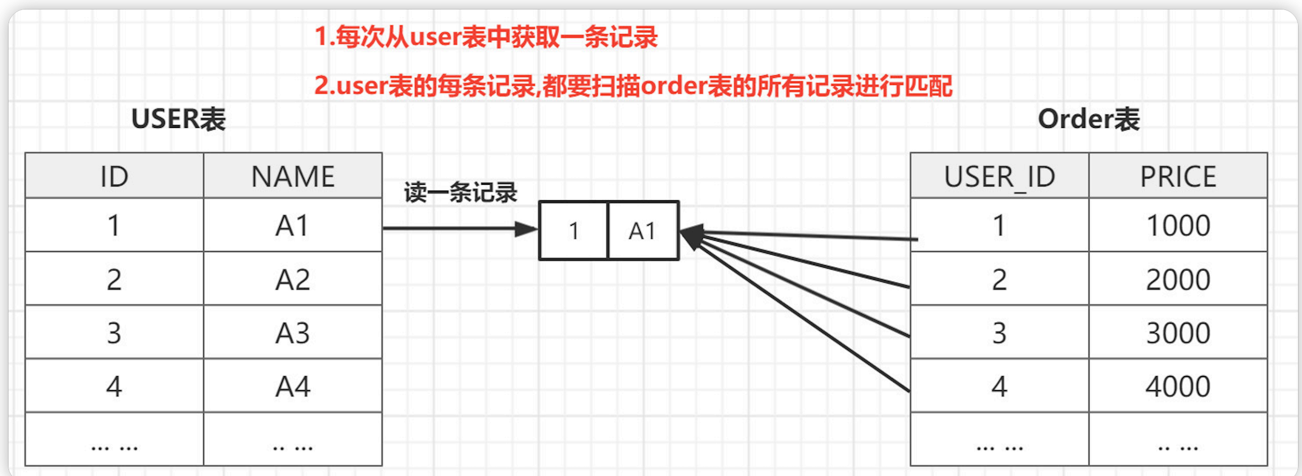
- 例如有这样一条SQL:

```
-- 连接用户表与订单表 连接条件是 u.id = o.user_id
select * from user t1 left join order t2 on t1.id =
t2.user_id;
-- user表为驱动表,order表为被驱动表
```

- 转换成代码执行时的思路是这样的:

```
for(user表行 uRow : user表){
    for(Order表的行 oRow : order表){
        if(uRow.id = oRow.user_id){
            return uRow;
        }
    }
}
```

- 匹配过程如下图

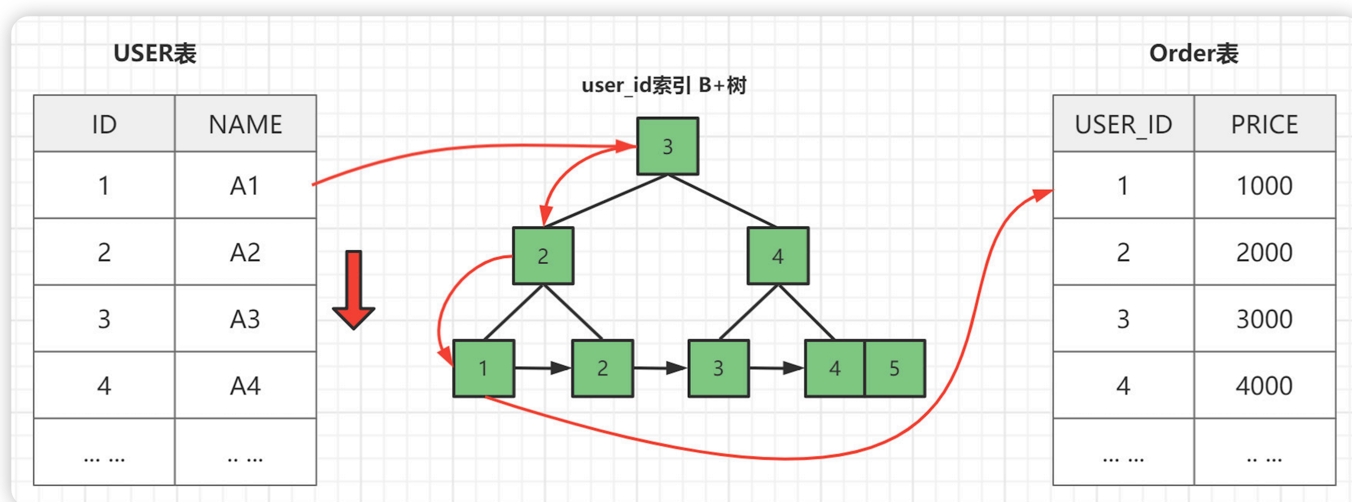


- **SNL 的特点**
    - **简单粗暴容易理解，就是通过双层循环比较数据来获得结果**
    - **查询效率会非常慢,假设 A 表有 N 行，B 表有 M 行。SNL 的开销如下：**
        - **A 表扫描 1 次。**

- **B 表扫描 M 次。**
- **一共有 N 个内循环，每个内循环要 M 次，一共有内循环 N * M 次**

2) Index Nested-Loop Join（索引嵌套循环连接）

- Index Nested-Loop Join 其优化的思路：**主要是为了减少内层表数据的匹配次数**，最大的区别在于，用来进行 join 的字段已经在被驱动表中建立了索引。
- 从原来的 `匹配次数 = 外层表行数 * 内层表行数`，变成了 `匹配次数 = 外层表的行数 * 内层表索引的高度` ，极大的提升了 join的性能。
- 当 `order` 表的 `user_id` 为索引的时候执行过程会如下图：

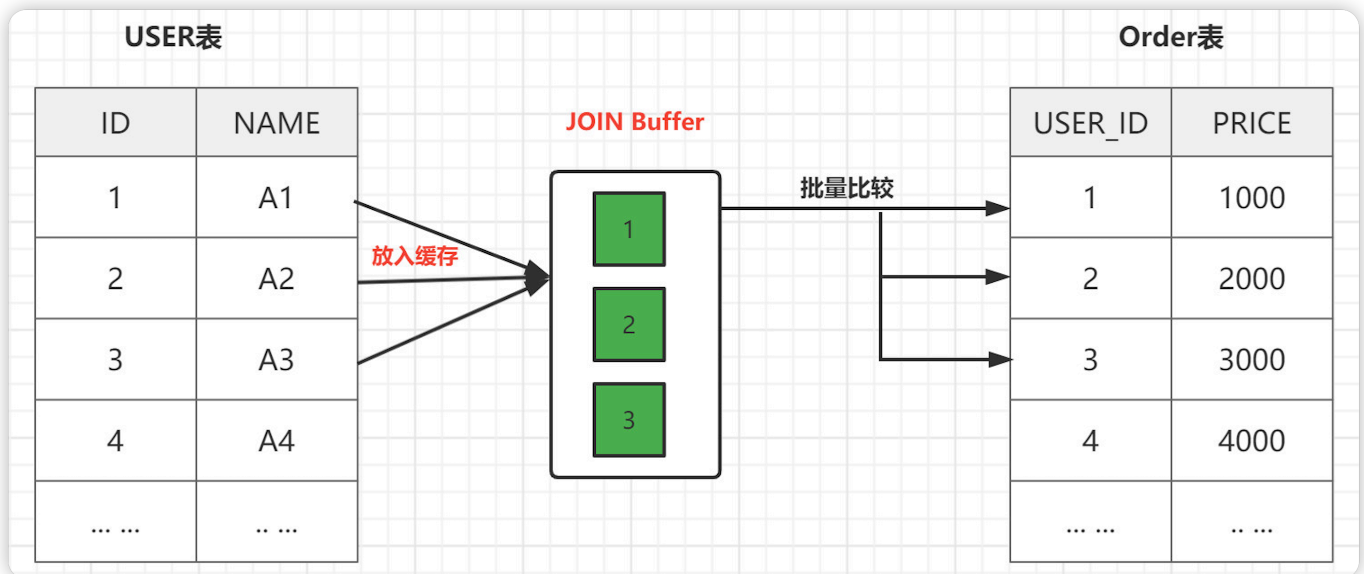


注意：使用Index Nested-Loop Join 算法的前提是匹配的字段必须建立了索引。

3) Block Nested-Loop Join(块嵌套循环连接)

如果 join 的字段有索引，MySQL 会使用 INL 算法。如果没有的话，MySQL 会如何处理？

因为不存在索引了，所以被驱动表需要进行扫描。这里 MySQL 并不会简单粗暴的应用 SNL 算法，而是加入了 buffer 缓冲区，降低了内循环的个数，也就是被驱动表的扫描次数。



- 在外层循环扫描 user表中的所有记录。扫描的时候，会把需要进行 join 用到的列都缓存到 buffer 中。buffer 中的数据有一个特点，里面的记录不需要一条一条地取出来和 order 表进行比较，而是整个 buffer 和 order表进行批量比较。
- 如果我们把 buffer 的空间开得很大，可以容纳下 user 表的所有记录，那么 order 表也只需要访问一次。
- MySQL 默认 buffer 大小 256K，如果有 n 个 join 操作，会生成 n-1 个 join buffer。

```
mysql> show variables like '%join_buffer%';
+------------------+--------+
| Variable_name    | Value  |
+------------------+--------+
| join_buffer_size | 262144 |
+------------------+--------+


mysql> set session join_buffer_size=262144;
Query OK, 0 rows affected (0.00 sec)
```

4) JOIN优化总结

1. 永远用小结果集驱动大结果集(其本质就是减少外层循环的数据数量)
2. 为匹配的条件增加索引(减少内层表的循环匹配次数)
3. 增大join buffer size的大小（一次缓存的数据越多，那么内层包的扫表次数就越少）
4. 减少不必要的字段查询（字段越少，join buffer 所缓存的数据就越多

29.索引哪些情况下会失效?

1. 查询条件包含 or，会导致索引失效。
2. 隐式类型转换，会导致索引失效，例如 age 字段类型是 int，我们 where age = “1”，这样就会触发隐式类型转换
3. like 通配符会导致索引失效，注意:”ABC%” 不会失效，会走 range 索引，”% ABC” 索引会失效
4. 联合索引，查询时的条件列不是联合索引中的第一个列，索引失效。
5. 对索引字段进行函数运算。
6. 对索引列运算（如，+、-、*、/），索引失效。
7. 索引字段上使用（!= 或者 < >，not in）时，会导致索引失效。
8. 索引字段上使用 is null， is not null，可能导致索引失效。
9. 相 join 的两个表的字符编码不同，不能命中索引，会导致笛卡尔积的循

环计算
0. mysql 估计使用全表扫描要比使用索引快，则不使用索引。

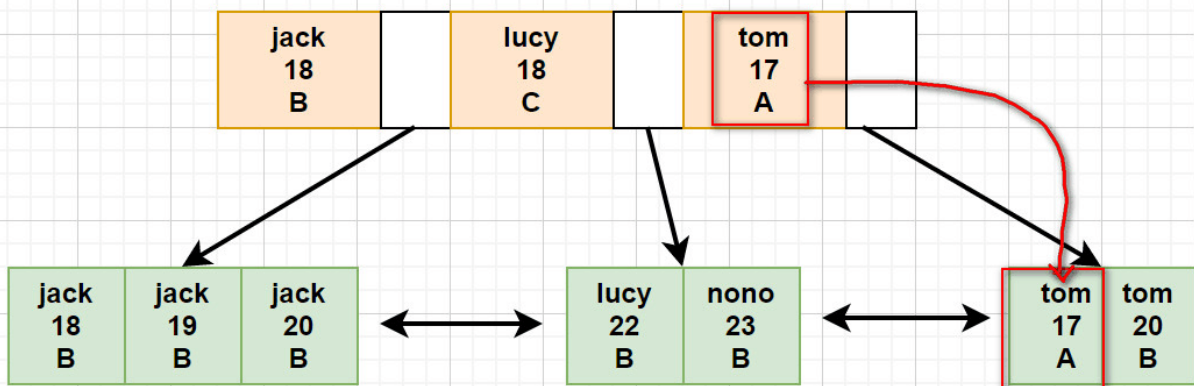
30.什么是覆盖索引？

覆盖索引是一种避免回表查询的优化策略：只需要在一棵索引树上就能获取SQL所需的所有列数据，无需回表，速度更快。

具体的实现方式:

- *将被查询的字段建立普通索引或者联合索引*，这样的话就可以直接返回索引中的的数据，不需要再通过聚集索引去定位行记录，避免了回表的情况发生。

```
EXPLAIN SELECT user_name,user_age,user_level FROM users
WHERE user_name = 'tom' AND user_age = 17;
```



覆盖索引的定义与注意事项:

- 如果一个索引包含了 所有需要查询的字段的值 (不需要回表)，这个索引

就是覆盖索引。
- MySQL只能使用B+Tree索引做覆盖索引 (因为只有B+树能存储索引列值)
- 在explain的Extra列, 如果出现 **`Using index` 表示 使用到了覆盖索引 , 所取的数据完全在索引中就能拿到

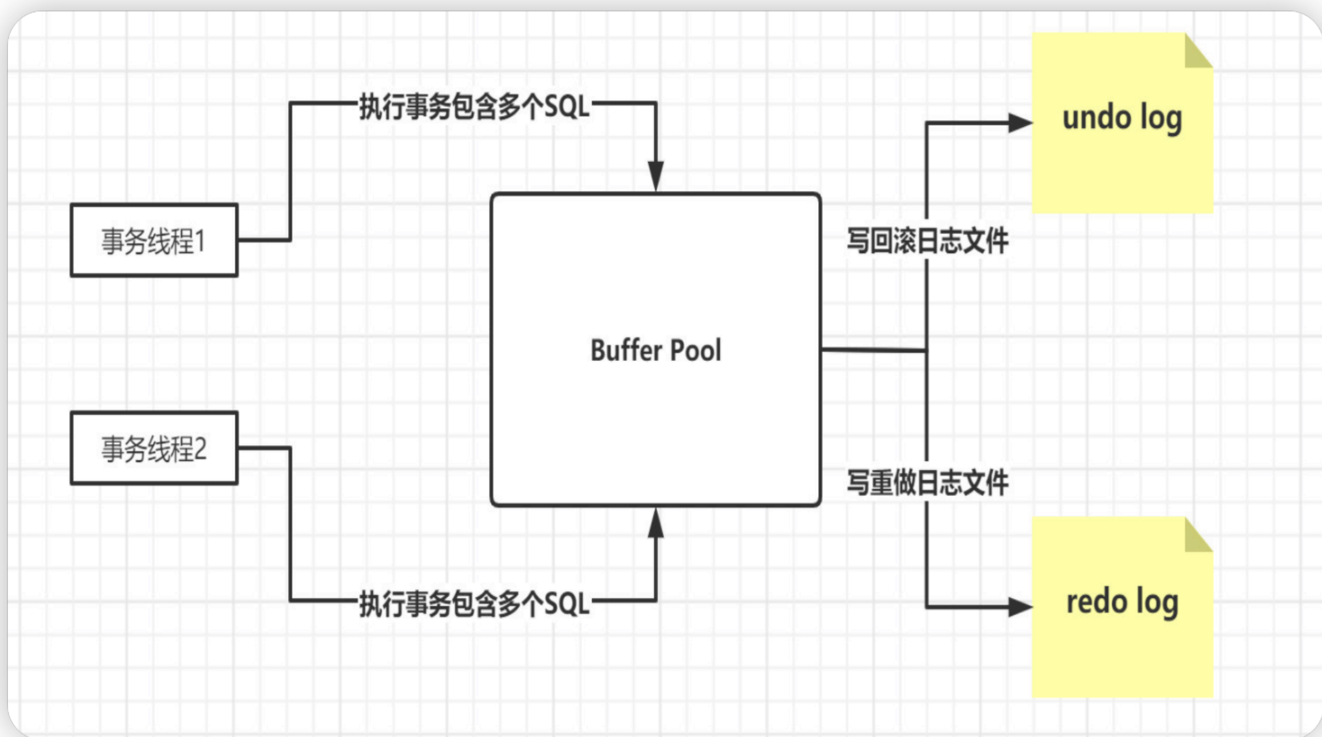
31.介绍一下MySQL中事务的特性?

在关系型数据库管理系统中，一个逻辑工作单元要成为事务，必须满足这 4 个特性，即所谓的 ACID：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability）。

1）原子性

原子性：事务作为一个整体被执行，包含在其中的对数据库的操作要么全部被执行，要么都不执行。

InnoDB存储引擎提供了两种事务日志：redo log(重做日志)和undo log(回滚日志)。其中redo log用于保证事务持久性；undo log则是事务原子性和隔离性实现的基础。

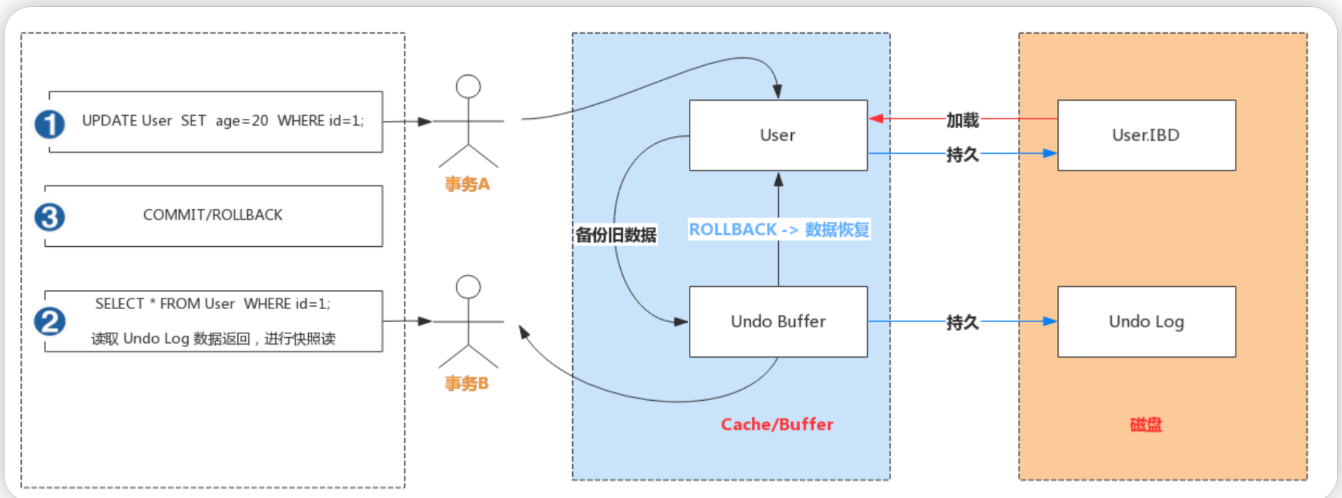


每写一个事务,都会修改Buffer Pool,从而产生相应的Redo/Undo日志:

- 如果要回滚事务，那么就基于undo log来回滚就可以了，把之前对缓存页做的修改都给回滚了就可以了。
- 如果事务提交之后，redo log刷入磁盘，结果MySQL宕机了，是可以根据redo log恢复事务修改过的缓存数据的。

实现原子性的关键，是当事务回滚时能够撤销所有已经成功执行的sql语句。

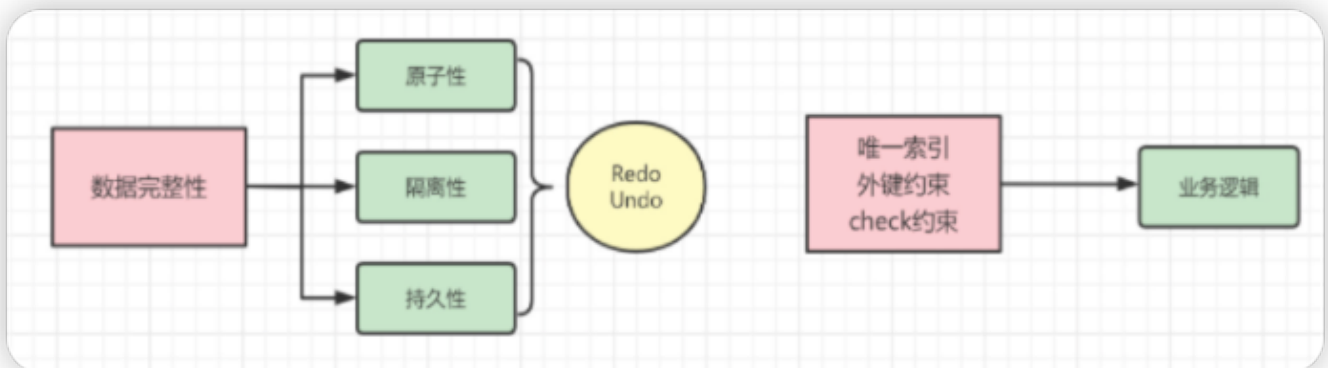
InnoDB 实现回滚，靠的是undo log： 当事务对数据库进行修改时，InnoDB 会生成对应的undo log ；如果事务执行失败或调用了rollback，导致事务需要回滚，便可以利用undo log中的信息将数据回滚到修改之前的样子。



2）一致性

一致性：事务应确保数据库的状态从一个一致状态转变为另一个一致状态。*一致状态*的含义是数据库中的数据应满足完整性约束。

- 约束一致性：创建表结构时所指定的外键、唯一索引等约束。
- 数据一致性：是一个综合性的规定，因为它是由原子性、持久性、隔离性共同保证的结果，而不是单单依赖于某一种技术。



3）隔离性

隔离性：指的是一个事务的执行不能被其他事务干扰，即一个事务内部的操作及使用的数据对其他的并发事务是隔离的。

不考虑隔离性会引发的问题:

- **脏读**：一个事务读取到了另一个事务修改但未提交的数据。

- **不可重复读**: 一个事务中多次读取同一行记录的结果不一致，后面读取的跟前面读取的结果不一致。
- **幻读** : 一个事务中多次按相同条件查询，结果不一致。后续查询的结果和面前查询结果不同，多了或少了几行记录。

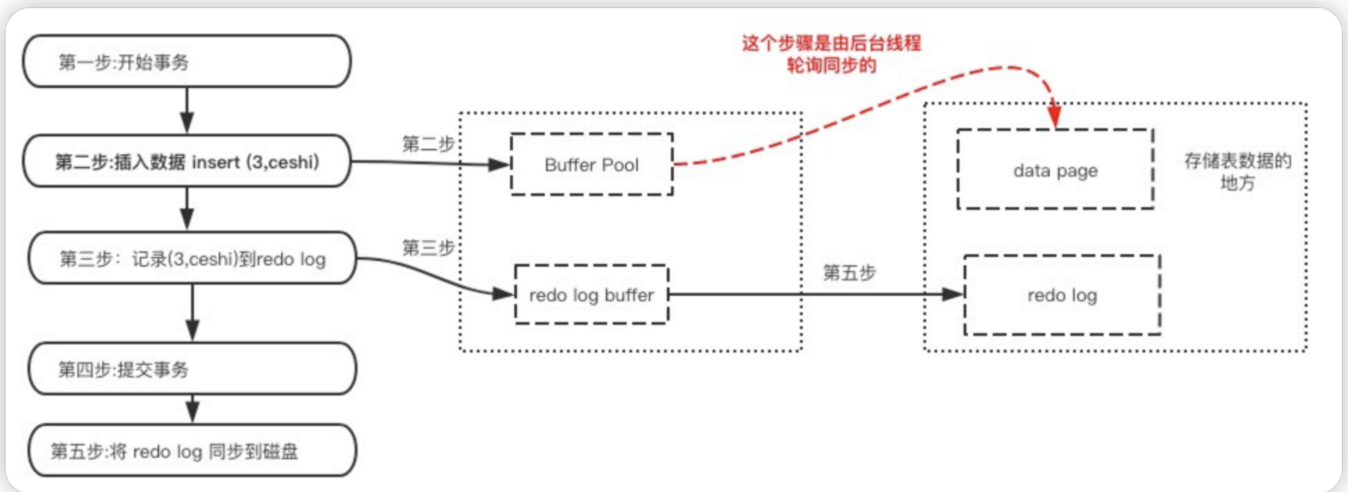
数据库事务的隔离级别有4个，由低到高依次为Read uncommitted 、Read committed、Repeatable read 、Serializable ，这四个级别可以逐个解决脏读 、不可重复读 、幻读 这几类问题。

4）持久性

持久性：指的是一个事务一旦提交，它对数据库中数据的改变就应该是永久性的，后续的操作或故障不应该对其有任何影响，不会丢失。

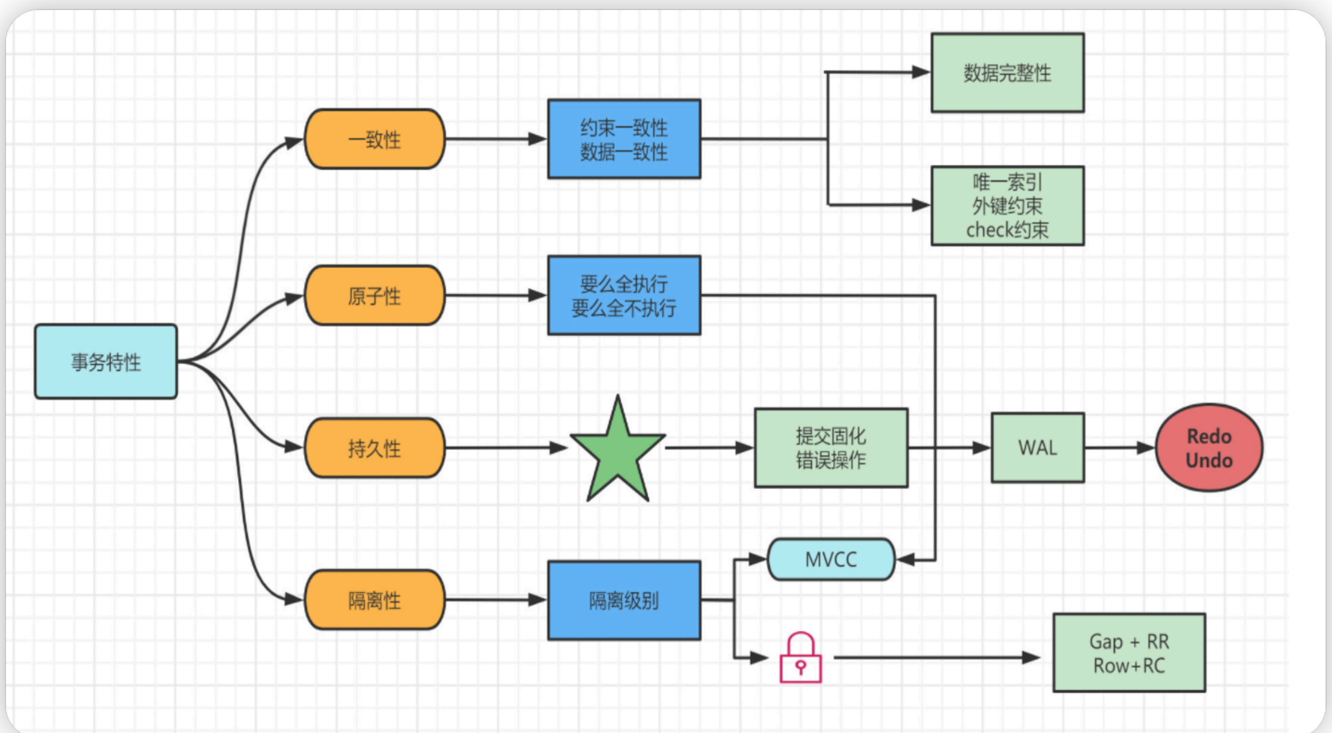
MySQL 事务的持久性保证依赖的日志文件: `redo log`

- redo log 也包括两部分：一是内存中的日志缓冲(redo log buffer)，该部分日志是易失性的；二是磁盘上的重做日志文件(redo log file)，该部分日志是持久的。redo log是物理日志，记录的是数据库中物理页的情况 。
- 当数据发生修改时，InnoDB不仅会修改Buffer Pool中的数据，也会在redo log buffer记录这次操作；当事务提交时，会对redo log buffer进行刷盘，记录到redo log file中。如果MySQL宕机，重启时可以读取redo log file中的数据，对数据库进行恢复。这样就不需要每次提交事务都实时进行刷脏了。



5）ACID总结

- 事务的持久化是为了应对系统崩溃造成的数据丢失.
- 只有保证了事务的一致性，才能保证执行结果的正确性
- 在非并发状态下，事务间天然保证隔离性，因此只需要保证事务的原子性即可保证一致性.
- 在并发状态下，需要严格保证事务的原子性、隔离性。



32.MySQL 的可重复读怎么实现的?

可重复读（repeatable read）定义： 一个事务执行过程中看到的数据，总是跟这个事务在启动时看到的数据是一致的。

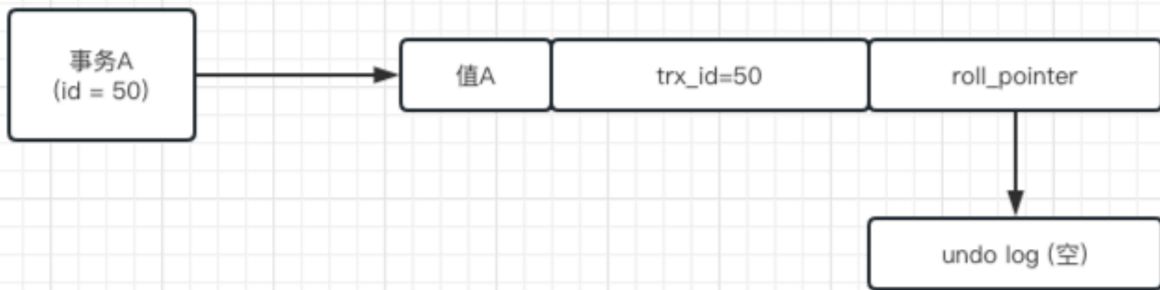
MVCC

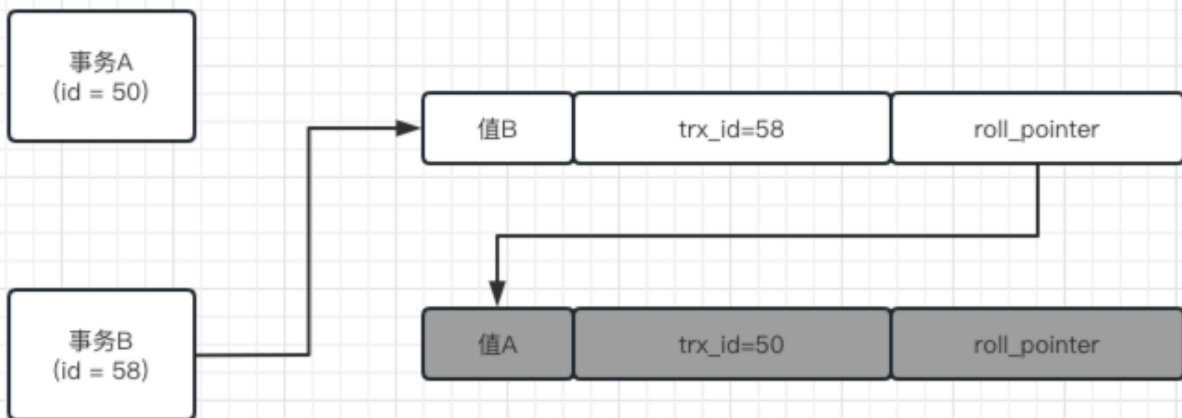
- MVCC，多版本并发控制, 用于实现**读已提交**和**可重复读**隔离级别。
- MVCC的核心就是 Undo log多版本链 + Read view，“MV”就是通过Undo log来保存数据的历史版本，实现多版本的管理，“CC”是通过Read-view来实现管理，通过 Read-view原则来决定数据是否显示。同时针对不同的隔离级别， Read view的生成策略不同，也就实现了不同的隔离级别。

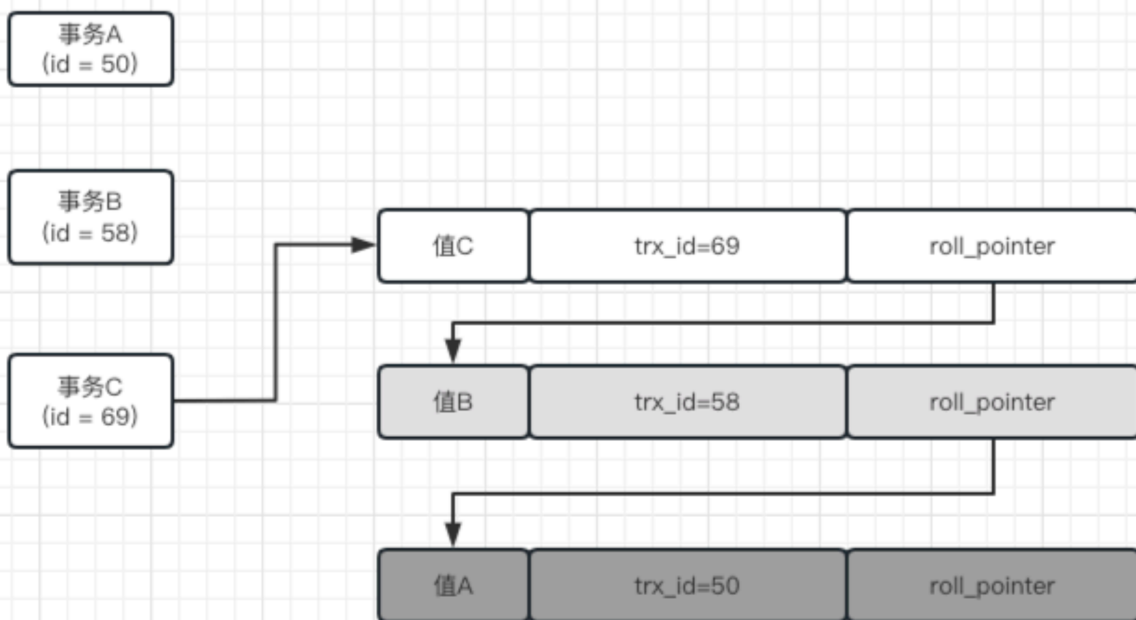
Undo log 多版本链

每条数据都有两个隐藏字段:

- trx_id: 事务id,记录最近一次更新这条数据的事务id.
- roll_pointer: 回滚指针,指向之前生成的undo log


1.事务A插入数据
事务A
(id = 50)
值A
trx_id=50
roll_pointer
undo log (空)


2.事务B修改事务A插入的数据
事务A
(id = 50)
事务B
(id = 58)
值B
trx_id=58
roll_pointer
值A
trx_id=50
roll_pointer


3.接着事务C 又来更新该条数据
事务A
(id = 50)
事务B
(id = 58)
事务C
(id = 69)
值C
trx_id=69
roll_pointer
值B
trx_id=58
roll_pointer
值A
trx_id=50
roll_pointer

每一条数据都有多个版本,版本之间通过undo log链条进行连接通过这样的设计方式,可以保证每个事务提交的时候,一旦需要回滚操作,可以保证同一个事务只能读取到比当前版本更早提交的值,不能看到更晚提交的值。

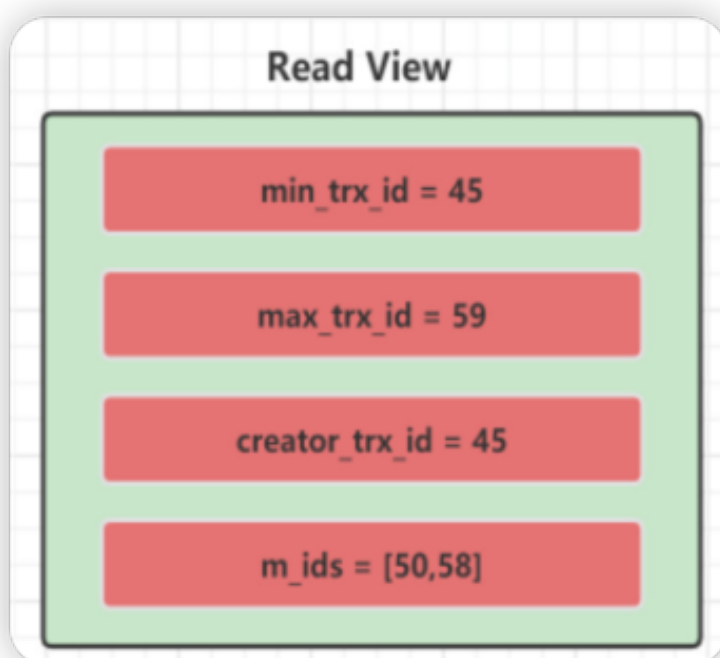
ReadView

Read View是 InnoDB 在实现 MVCC 时用到的一致性读视图，即 consistent read view，用于支持 RC（Read Committed，读提交）和 RR（Repeatable Read，可重复读）隔离级别的实现.

Read View简单理解就是对数据在某个时刻的状态拍成照片记录下来。那么之后获取某时刻的数据时就还是原来的照片上的数据，是不会变的.

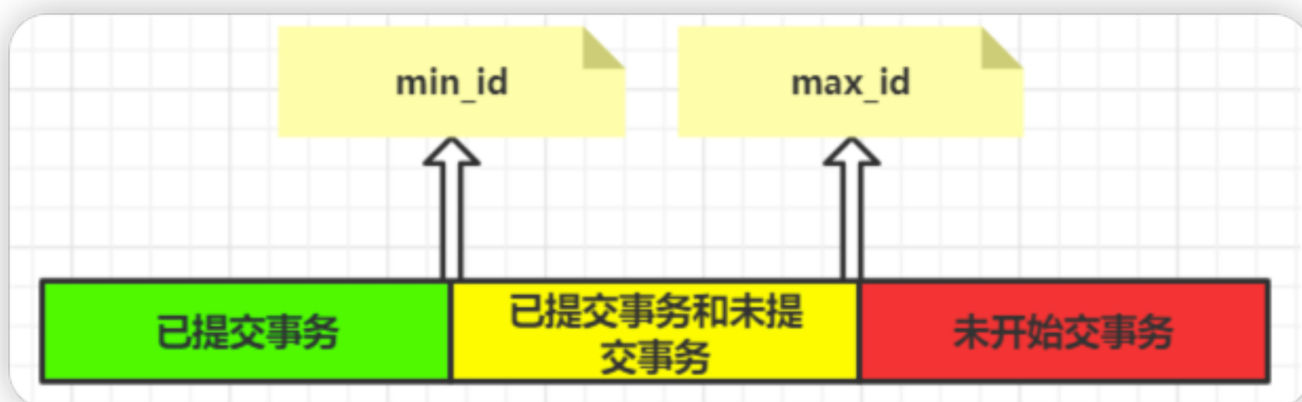
Read View中比较重要的字段有4个:

- `m_ids` : 用来表示MySQL中哪些事务正在执行,但是没有提交.
- `min_trx_id`: 就是m_ids里最小的值.
- `max_trx_id` : 下一个要生成的事务id值,也就是最大事务id
- `creator_trx_id`: 就是你这个事务的id



当一个事务第一次执行查询sql时，会生成一致性视图 read-view（快照），查询时从 undo log 中最新的一条记录开始跟 read-view 做对比，如果不符合比较规则，就根据回滚指针回滚到上一条记录继续比较，直到得到符合比较条件的查询结果。

Read View判断记录某个版本是否可见的规则如下



1.如果当前记录的事务id落在绿色部分（trx_id < min_id），表示这个版本是已提交的事务生成的，可读。
2.如果当前记录的事务id落在红色部分（trx_id > max_id），表示这个版本是由将来启动的事务生成的，不可读。

3. 如果当前记录的事务id落在黄色部分（min_id <= trx_id <= max_id），则分为两种情况：
4. 若当前记录的事务id在未提交事务的数组中，则此条记录不可读；
5. 若当前记录的事务id不在未提交事务的数组中，则此条记录可读。

RC 和 RR 隔离级别都是由 MVCC 实现，区别在于：

- RC 隔离级别时，read-view 是每次执行 select 语句时都生成一个；
- RR 隔离级别时，read-view 是在第一次执行 select 语句时生成一个，同一事务中后面的所有 select 语句都复用这个 read-view 。

33.Repeatable Read 解决了幻读问题吗?

可重复读（repeatable read）定义： 一个事务执行过程中看到的数据，总是跟这个事务在启动时看到的数据是一致的。

不过理论上会出现幻读，简单的说幻读指的的当用户读取某一范围的数据行时，另一个事务又在该范围插入了新行，当用户在读取该范围的数据时会发现有新的幻影行。

注意在可重复读隔离级别下，普通的查询是快照读，是不会看到别的事务插入的数据的。因此， 幻读在“当前读”下才会出现（查询语句添加for update，表示当前读）；

在 MVCC 并发控制中，读操作可以分为两类: 快照读（`Snapshot Read`）与当前读 （`Current Read`）。

- 快照读
  快照读是指读取数据时不是读取最新版本的数据，而是基于历史版本读取的一个快照信息（mysql读取undo log历史版本)，快照读可以使普通的SELECT 读取数据时不用对表数据进行加锁，从而解决了因为对数据库表的加锁而导致的两个如下问题

  1. 解决了因加锁导致的修改数据时无法对数据读取问题.
  2. 解决了因加锁导致读取数据时无法对数据进行修改的问题.
- 当前读
  当前读是读取的数据库最新的数据，当前读和快照读不同，因为要读取最新的数据而且要保证事务的隔离性，所以当前读是需要对数据进行加锁的（`插入/更新/删除操作，属于当前读，需要加锁` ，`select for update` 为当前读）

表结构

id	key	value
0	0	0
1	1	1

假设 select * from where value=1 for update，只在这一行加锁（注意这只是假设），其它行不加锁，那么就会出现如下场景：

	session A	session B	session C
T1	Q1: begin; select * from t where value=1 for update; result:(1,1,1)		
T2		update t set value=1 where id=0;	
T3	Q2: begin; select * from t where value=1 for update; result:(0,0,1),(1,1,1)		
T4			insert into t value(6,6,1);
T5	Q3: begin; select * from t where value=1 for update; result:(0,0,1),(1,1,1),(6,6,1)		
T6	commit;		

Session A的三次查询Q1-Q3都是select * from where value=1 for update，查询的value=1的所有row。

- T1：Q1只返回一行(1,1,1)；
- T2：session B更新id=0的value为1，此时表t中value=1的数据有两行
- T3：Q2返回两行(0,0,1),(1,1,1)
- T4：session C插入一行(6,6,1)，此时表t中value=1的数据有三行

- T5：Q3返回三行(0,0,1),(1,1,1),(6,6,1)
- T6：session A事物commit。

其中Q3读到value=1这一样的现象，就称之为幻读，**幻读指的是一个事务在前后两次查询同一个范围的时候，后一次查询看到了前一次查询没有看到的行。**

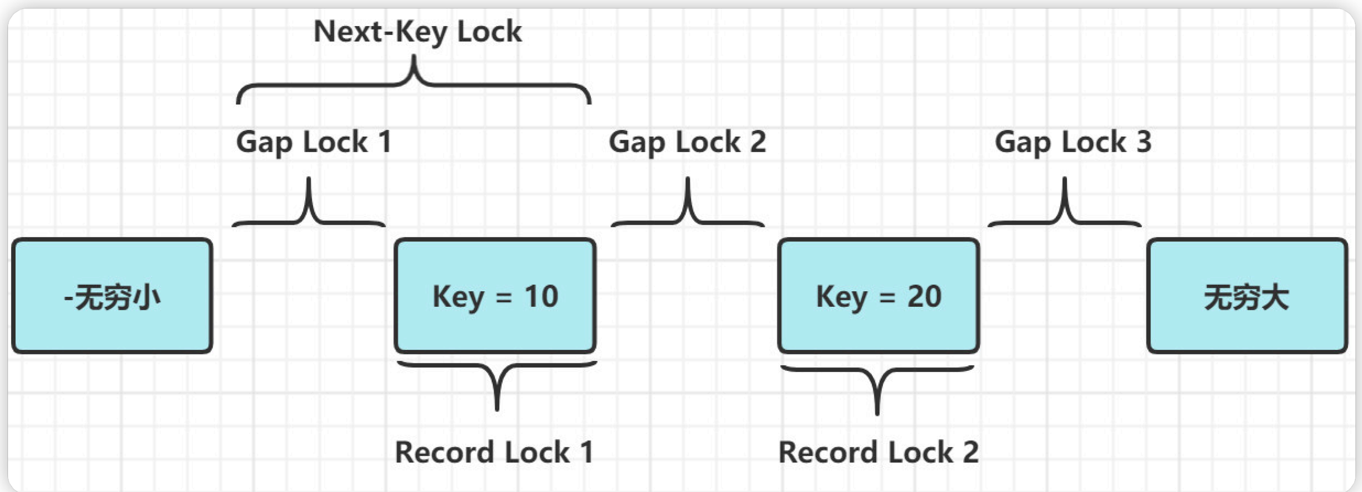
先对“幻读”做出如下解释：

- 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，而不是快照读,因为在可重复读隔离级别下，普通的查询是快照读，是不会看到别的事务插入的数据的。

Next-key Lock 锁

产生幻读的原因是，行锁只能锁住行，但是新插入记录这个动作，要更新的是记录之间的“间隙”。因此，Innodb 引擎为了解决「可重复读」隔离级别使用「当前读」而造成的幻读问题，就引出了 next-key 锁，就是记录锁和间隙锁的组合。

- RecordLock锁：锁定单个行记录的锁。（记录锁，RC、RR隔离级别都支持）
- GapLock锁：间隙锁，锁定索引记录间隙(不包括记录本身)，确保索引记录的间隙不变。（范围锁，RR隔离级别支持）
- Next-key Lock 锁：记录锁和间隙锁组合，同时锁住数据，并且锁住数据前后范围。（记录锁+范围锁，RR隔离级别支持）



总结

- RR隔离级别下间隙锁才有效，RC隔离级别下没有间隙锁；
- RR隔离级别下为了解决“幻读”问题：“快照读”依靠MVCC控制，“当前读”通过间隙锁解决；
- 间隙锁和行锁合称next-key lock，每个next-key lock是前开后闭区间；
- 间隙锁的引入，可能会导致同样语句锁住更大的范围，影响并发度。

34.请说一下数据库锁的种类?

MySQL数据库由于其自身架构的特点,存在多种数据存储引擎, MySQL中不同的存储引擎支持不同的锁机制。

- **MyISAM**和**MEMORY**存储引擎采用的表级锁，
- **InnoDB**存储引擎既支持行级锁，也支持表级锁，默认情况下采用行级锁。
- **BDB**采用的是页面锁，也支持表级锁

按照数据操作的类型分

- 读锁（共享锁）：针对同一份数据，多个读操作可以同时进行而不会互相影响。
- 写锁（排他锁）：当前写操作没有完成前，它会阻断其他写锁和读锁。

按照数据操作的粒度分

- 表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低。
- 行级锁： 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。
- 页面锁：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

按照操作性能可分为乐观锁和悲观锁

- 乐观锁：一般的实现方式是对记录数据版本进行比对，在数据更新提交的时候才会进行冲突检测，如果发现冲突了，则提示错误信息。
- 悲观锁：在对一条数据修改的时候，为了避免同时被其他人修改，在修改数据之前先锁定，再修改的控制方式。共享锁和排他锁是悲观锁的不同实现，但都属于悲观锁范畴。

35.请说一下共享锁和排他锁?

行级锁分为共享锁和排他锁两种。

行锁的是mysql锁中粒度最小的一种锁，因为锁的粒度很小，所以发生资源争抢的概率也最小，并发性能最大，但是也会造成死锁，每次加锁和释放锁的开销也会变大。

使用MySQL行级锁的两个前提

- 使用 innoDB 引擎
- 开启事务 (隔离级别为 `Repeatable Read`)

InnoDB行锁的类型

- **共享锁**（S）：当事务对数据加上共享锁后, 其他用户可以并发读取数

据，但任何事务都不能对数据进行修改（获取数据上的排他锁），直到已释放所有共享锁。

- **排他锁**（X）：如果事务T对数据A加上排他锁后，则其他事务不能再对数据A加任任何类型的封锁。获准排他锁的事务既能读数据，又能修改数据。

加锁的方式

- InnoDB引擎默认更新语句，**update,delete,insert 都会自动给涉及到的数据加上排他锁**，select语句默认不会加任何锁类型，如果要加可以使用下面的方式:
- 加共享锁（S）：select * from table_name where ... **lock in share mode**;
- 加排他锁（x）：select * from table_name where ... **for update**;

锁兼容

- 共享锁只能兼容共享锁, 不兼容排它锁
- 排它锁互斥共享锁和其它排它锁

共享锁与排它锁的兼容性		
	共享锁	排他锁
共享锁	兼容	冲突
排它锁	冲突	冲突

36.InnoDB 的行锁是怎么实现的?

InnoDB行锁是通过对索引数据页上的记录加锁实现的，主要实现算法有 3 种：Record Lock、Gap Lock 和 Next-key Lock。

- **RecordLock锁：锁定单个行记录的锁。（记录锁，RC、RR隔离级别都支持）**
- **GapLock锁：间隙锁，锁定索引记录间隙，确保索引记录的间隙不变。（范围锁，RR隔离级别支持）**
- **Next-key Lock 锁：记录锁和间隙锁组合，同时锁住数据，并且锁住数据前后范围。（记录锁+范围锁，RR隔离级别支持）**

> 注意： InnoDB这种行锁实现特点意味着：只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁

在RR隔离级别，InnoDB对于记录加锁行为都是先采用Next-Key Lock，但是当SQL操作含有唯一索引时，Innodb会对Next-Key Lock进行优化，降级为RecordLock，仅锁住索引本身而非范围。

各种操作加锁的特点

1）select ... from 语句：InnoDB引擎采用MVCC机制实现非阻塞读，所以对于普通的select语句，InnoDB不加锁

2）select ... from lock in share mode语句：追加了共享锁，InnoDB会使用Next-Key Lock锁进行处理，如果扫描发现唯一索引，可以降级为RecordLock锁。

3）select ... from for update语句：追加了排他锁，InnoDB会使用Next-Key Lock锁进行处理，如果扫描发现唯一索引，可以降级为RecordLock锁。

4）update ... where 语句：InnoDB会使用Next-Key Lock锁进行处理，如果扫描发现唯一索引，可以降级为RecordLock锁。

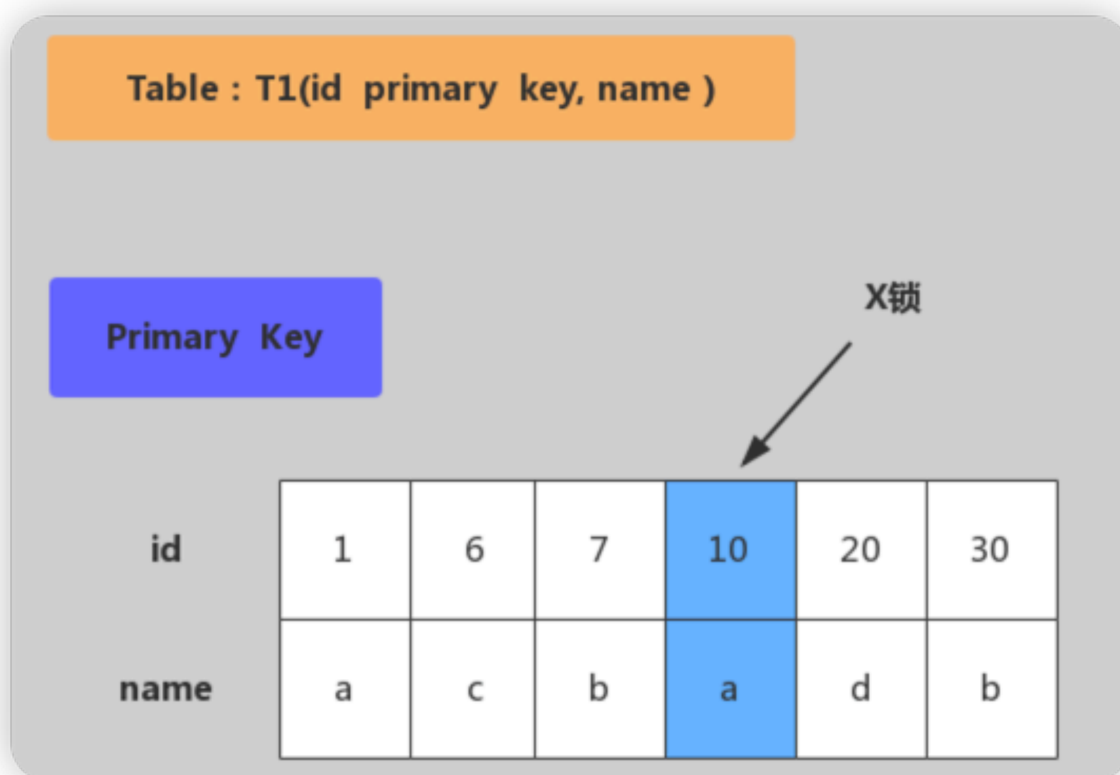
5）delete ... where 语句：InnoDB会使用Next-Key Lock锁进行处理，如果扫描发现唯一索引，可以降级为RecordLock锁。

6）insert语句：InnoDB会在将要插入的那一行设置一个排他的RecordLock锁。

下面以“update t1 set name=‘lisi’ where id=10”操作为例，举例子分析下 InnoDB 对不同索引的加锁行为，以RR隔离级别为例。

1. 主键加锁

   加锁行为：仅在id=10的主键索引记录上加X锁。

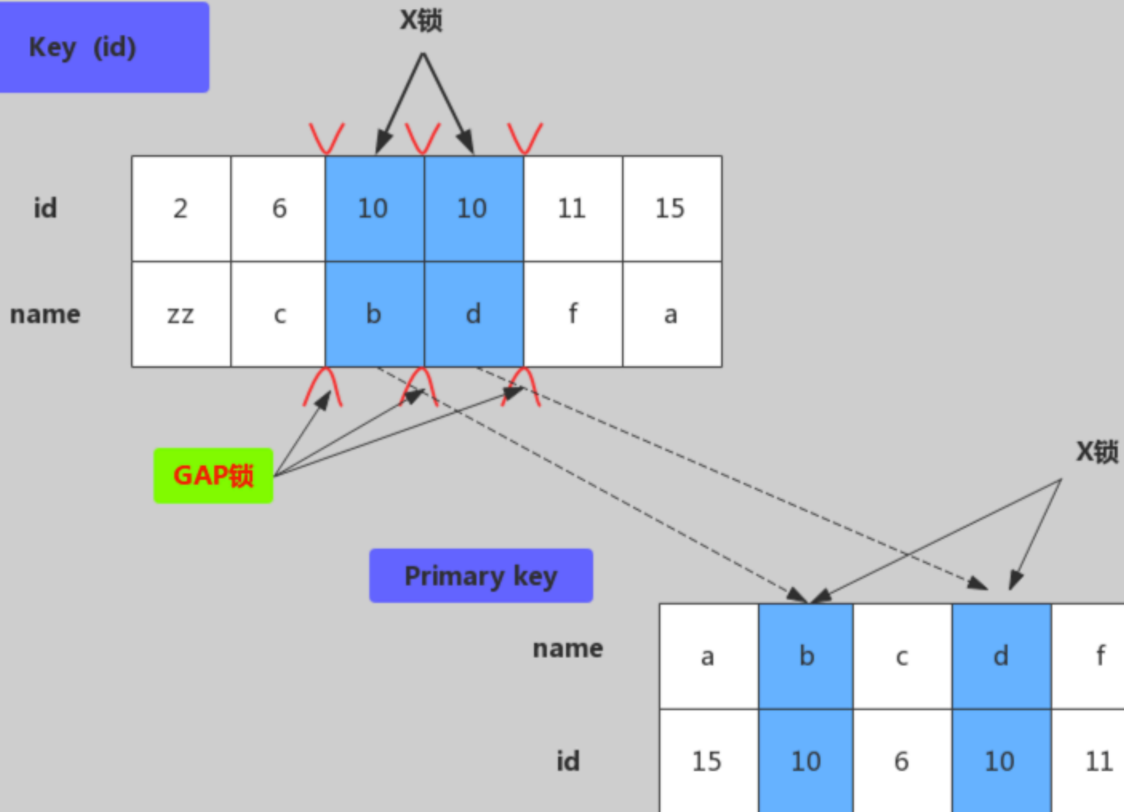


2. 唯一键加锁

   加锁行为：现在唯一索引id上加X锁，然后在id=10的主键索引记录上加X锁。

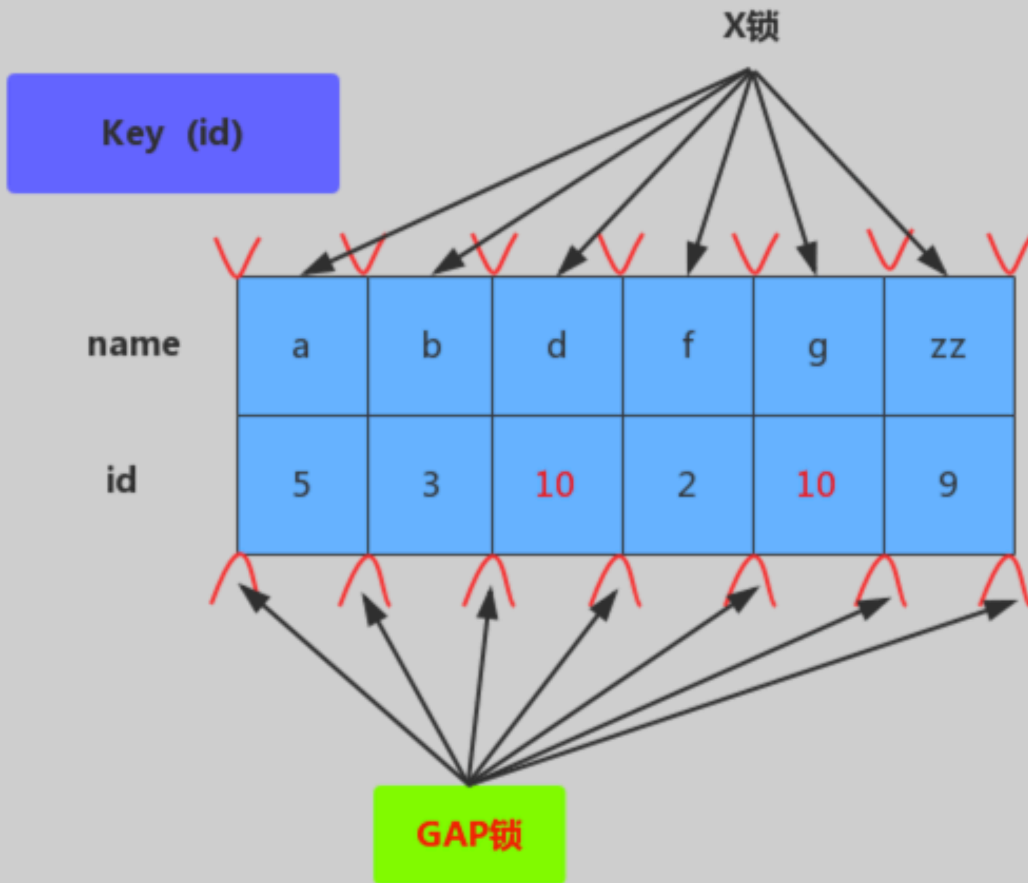
3. 非唯一键加锁

加锁行为：对满足id=10条件的记录和主键分别加X锁，然后在(6,c)-(10,b)、(10,b)-(10,d)、(10,d)-(11,f)范围分别加Gap Lock。



4. 无索引加锁

   加锁行为：表里所有行和间隙都会加X锁。（当没有索引时，会导致全表锁定，因为InnoDB引擎锁机制是基于索引实现的记录锁定）。



37.并发事务会产生哪些问题

事务并发处理可能会带来一些问题，如下：

- 更新丢失
  当两个或多个事务更新同一行记录，会产生更新丢失现象。可以分为回滚覆盖和提交覆盖。
  - 回滚覆盖：一个事务回滚操作，把其他事务已提交的数据给覆盖了。
  - 提交覆盖：一个事务提交操作，把其他事务已提交的数据给覆盖了。

- 脏读
  一个事务读取到了另一个事务修改但未提交的数据。
- 不可重复读
  一个事务中多次读取同一行记录不一致，后面读取的跟前面读取的不一致。
- 幻读
  一个事务中多次按相同条件查询，结果不一致。后续查询的结果和面前查询结果不同，多了或少了几行记录。

“更新丢失”、”脏读”、“不可重复读”和“幻读”等并发事务问题，其实都是数据库一致性问题，为了解决这些问题，MySQL数据库是通过事务隔离级别来解决的，数据库系统提供了以下 4 种事务隔离级别供用户选择。

事务隔离级别	回滚覆盖	脏读	不可重复读	提交覆盖	幻读
读未提交	x	可能发生	可能发生	可能发生	可能发生
读已提交	x	x	可能发生	可能发生	可能发生
可重复读	x	x	x	x	可能发生
串行化	x	x	x	x	x

事务隔离级别	回滚覆盖	脏读	不可重复读	提交覆盖	幻读
读未提交	x	可能发生	可能发生	可能发生	可能发生
读已提交	x	x	可能发生	可能发生	可能发生
可重复读	x	x	x	x	可能发生
串行化	x	x	x	x	x

- **读未提交**
  Read Uncommitted 读未提交：解决了回滚覆盖类型的更新丢失，但可能发生脏读现象，也就是可能读取到其他会话中未提交事务修改的数据。
- **已提交读**

Read Committed 读已提交：只能读取到其他会话中已经提交的数据，解决了脏读。但可能发生不可重复读现象，也就是可能在一个事务中两次查询结果不一致。

- **可重复度**
  Repeatable Read 可重复读：解决了不可重复读，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。不过理论上会出现幻读，简单的说幻读指的的当用户读取某一范围的数据行时，另一个事务又在该范围插入了新行，当用户在读取该范围的数据时会发现有新的幻影行。
- **可串行化**
  所有的增删改查串行执行。它通过强制事务排序，解决相互冲突，从而解决幻度的问题。这个级别可能导致大量的超时现象的和锁竞争，效率低下。

数据库的事务隔离级别越高，并发问题就越小，但是并发处理能力越差（代价）。读未提交隔离级别最低，并发问题多，但是并发处理能力好。以后使用时，可以根据系统特点来选择一个合适的隔离级别，比如对不可重复读和幻读并不敏感，更多关心数据库并发处理能力，此时可以使用Read Commited隔离级别。

事务隔离级别，针对Innodb引擎，支持事务的功能。像MyISAM引擎没有关系。

事务隔离级别和锁的关系

1）事务隔离级别是SQL92定制的标准，相当于事务并发控制的整体解决方案，本质上是对锁和MVCC使用的封装，隐藏了底层细节。

2）锁是数据库实现并发控制的基础，事务隔离性是采用锁来实现，对相应操作加不同的锁，就可以防止其他事务同时对数据进行读写操作。

3）对用户来讲，首先选择使用隔离级别，当选用的隔离级别不能解决并发问题或需求时，才有必要在开发中手动的设置锁。

MySQL默认隔离级别：可重复读

Oracle、SQLServer默认隔离级别：读已提交

一般使用时，建议采用默认隔离级别，然后存在的一些并发问题，可以通过悲观锁、乐观锁等实现处理。

38.说一下MVCC内部细节

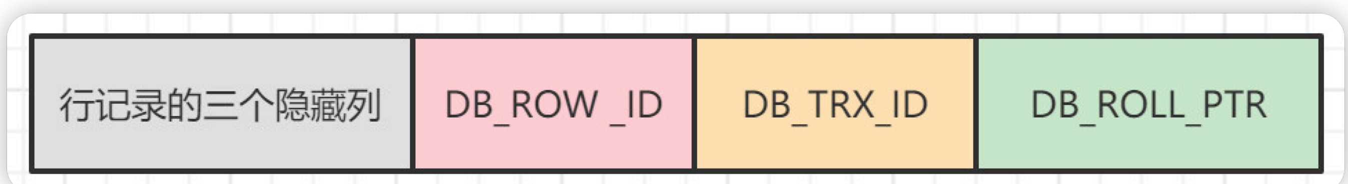
MVCC概念

MVCC（Multi Version Concurrency Control）被称为多版本并发控制，是指在数据库中为了实现高并发的数据访问，对数据进行多版本处理，并通过事务的可见性来保证事务能看到自己应该看到的数据版本。

> MVCC最大的好处是读不加锁，读写不冲突。在读多写少的系统应用中，读写不冲突是非常重要的，极大的提升系统的并发性能，这也是为什么现阶段几乎所有的关系型数据库都支持 MVCC 的原因，不过目前MVCC只在 Read Commited 和 Repeatable Read 两种隔离级别下工作。

回答这个面试题时，主要介绍以下的几个关键内容：

1）行记录的三个隐藏字段



- `DB_ROW_ID`：如果没有为表显式的定义主键，并且表中也没有定义唯一

索引，那么InnoDB会自动为表添加一个row_id的隐藏列作为主键。

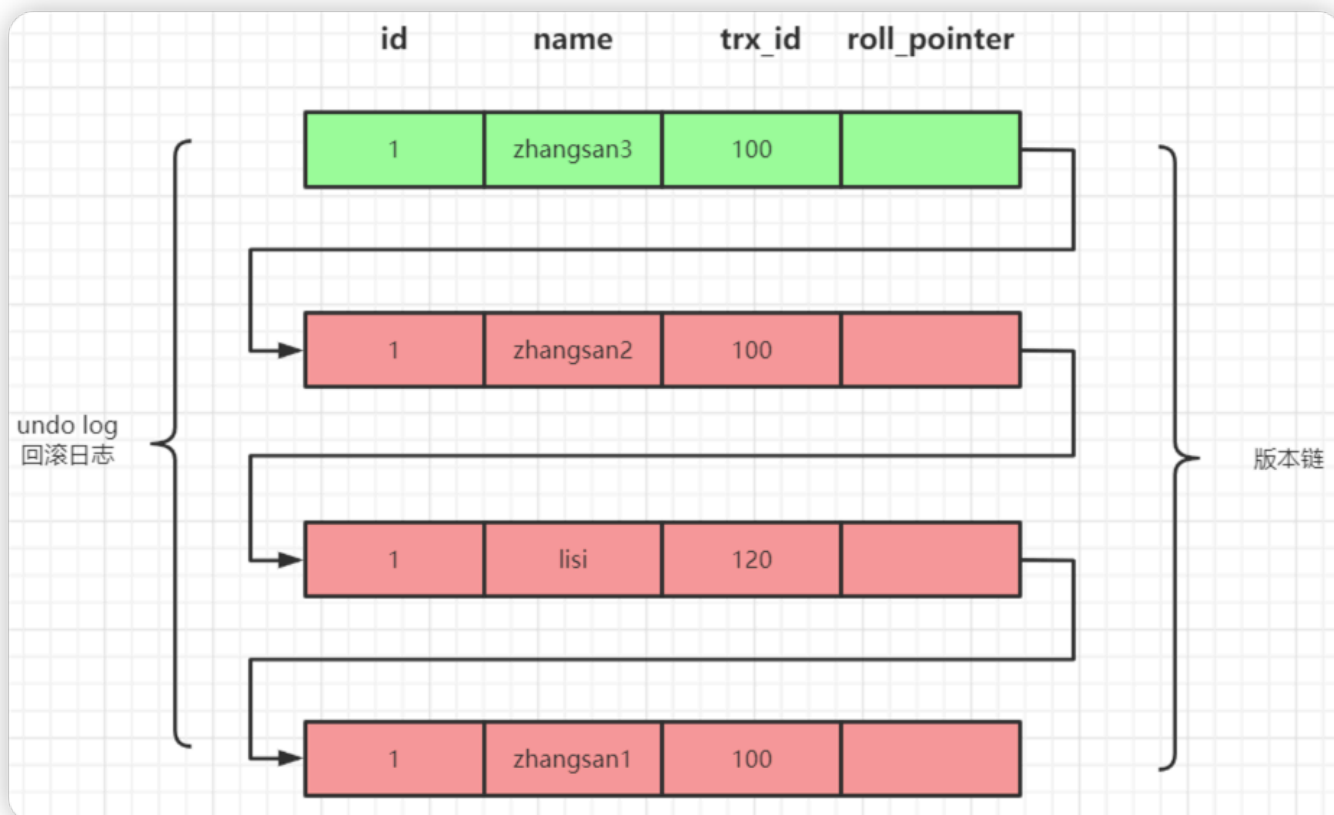
- `DB_TRX_ID`：事务中对某条记录做增删改时,就会将这个事务的事务ID写入到trx_id中.
- `DB_ROLL_PTR`: 回滚指针,指向undo log的指针

2）Undo log 多版本链

举例：事务 T-100 和 T-120 对表中 id = 1 的数据行做 update 操作，事务 T-130 进行 select 操作，即使 T-100 已经提交修改，三次 select 语句的结果都是“lisi”。

T-100	T-120	T-130		
update user set name = 'zhangsan1' where id = 1;				
	update user set name = 'lisi where id = 1;			
	commit;			
		1 select name from user where id = 1;	lisi	readview : [100] 120
update user set name = 'zhangsan2' where id = 1;				
update user set name = 'zhangsan3' where id = 1;				
		2 select name from user where id = 1;	lisi	readview : [100] 120
commit;				
		3 select name from user where id = 1;	lisi	readview : [100] 120

- 每一条数据都有多个版本,版本之间通过undo log链条进行连接



3）ReadView

Read View是 InnoDB 在实现 MVCC 时用到的一致性读视图，即 consistent read view，用于支持 RC（Read Committed，读提交）和 RR（Repeatable Read，可重复读）隔离级别的实现.

Read View简单理解就是对数据在每个时刻的状态拍成照片记录下来。那么之后获取某时刻的数据时就还是原来的照片上的数据，是不会变的.

Read View中比较重要的字段有4个:

- `m_ids` : 用来表示MySQL中哪些事务正在执行,但是没有提交.
- `min_trx_id`: 就是m_ids里最小的值.
- `max_trx_id` : 下一个要生成的事务id值,也就是最大事务id
- `creator_trx_id`: 就是你这个事务的id

通过Read View判断记录的某个版本是否可见的方式总结:

- trx_id = creator_trx_id
  如果被访问版本的trx_id,与readview中的creator_trx_id值相同,表明当前事务在访问自己修改过的记录,该版本可以被当前事务访问.

- trx_id < min_trx_id
  如果被访问版本的trx_id,小于readview中的min_trx_id值,表明生成该版本的事务在当前事务生成readview前已经提交,该版本可以被当前事务访问.

- trx_id >= max_trx_id
  如果被访问版本的trx_id,大于或等于readview中的max_trx_id值,表明生成该版本的事务在当前事务生成readview后才开启,该版本不可以被当前事务访问.

- trx_id > min_trx_id && trx_id < max_trx_id
  如果被访问版本的trx_id,值在readview的min_trx_id和max_trx_id之间，就需要判断trx_id属性值是不是在m_ids列表中?

    - 在：说明创建readview时生成该版本的事务还是活跃的,该版本不可以被访问
    - 不在：说明创建readview时生成该版本的事务已经被提交,该版本可以被访问

何时生成ReadView快照

- 在 **读已提交（Read Committed， 简称RC）** 隔离级别下，**每一次**读取数据前都生成一个ReadVIew。
- 在 **可重复读** （Repeatable Read，简称RR）隔离级别下，在一个事务中，只在 **第一次**读取数据前生成一个ReadVIew。

4）快照读（`Snapshot Read`）与当前读 （`Current Read`）

在 MVCC 并发控制中，读操作可以分为两类: 快照读（`Snapshot Read`）与当前读 （`Current Read`） 。

- 快照读
  快照读是指读取数据时不是读取最新版本的数据，而是基于历史版本读取的一个快照信息（mysql读取undo log历史版本) ，快照读可以使普通的SELECT 读取数据时不用对表数据进行加锁，从而解决了因为对数据库表的加锁而导致的两个如下问题

  1. 解决了因加锁导致的修改数据时无法对数据读取问题.
  2. 解决了因加锁导致读取数据时无法对数据进行修改的问题.
- 当前读
  当前读是读取的数据库最新的数据，当前读和快照读不同，因为要读取最新的数据而且要保证事务的隔离性，所以当前读是需要对数据进行加锁的（`Update delete insert select ....lock in share mode` ，`select for update` 为当前读）

总结一下

- 并发环境下，写-写操作有加锁解决方案，但为了提高性能，InnoDB存储引擎提供MVCC，目的是为了解决读-写，写-读操作下不加锁仍能安全进行。
- MVCC的过程，本质就是访问版本链，并判断哪个版本可见的过程。该判断算法是通过版本上的trx_id与快照ReadView的若干个信息进行对比。
- 快照生成的时机因隔离级别不同，读已提交隔离级别下，每一次读取前都会生成一个快照ReadView；而可重复读则仅在一个事务中，第一次读取前生成一个快照。

39.说一下MySQL死锁的原因和处理方法

1) 表的死锁

产生原因:

用户A访问表A（锁住了表A），然后又访问表B；另一个用户B访问表B（锁住了表B），然后企图访问表A；这时用户A由于用户B已经锁住表B，它必须等待用户B释放表B才能继续，同样用户B要等用户A释放表A才能继续，这就死锁就产生了。

用户A--》A表（表锁）--》B表（表锁）

用户B--》B表（表锁）--》A表（表锁）

解决方案：

这种死锁比较常见，是由于程序的BUG产生的，除了调整的程序的逻辑没有其它的办法。

仔细分析程序的逻辑，对于数据库的多表操作时，尽量按照相同的顺序进行处理，尽量避免同时锁定两个资源，如操作A和B两张表时，总是按先A后B的顺序处理， 必须同时锁定两个资源时，要保证在任何时刻都应该按照相同的顺序来锁定资源。

2) 行级锁死锁

产生原因1：

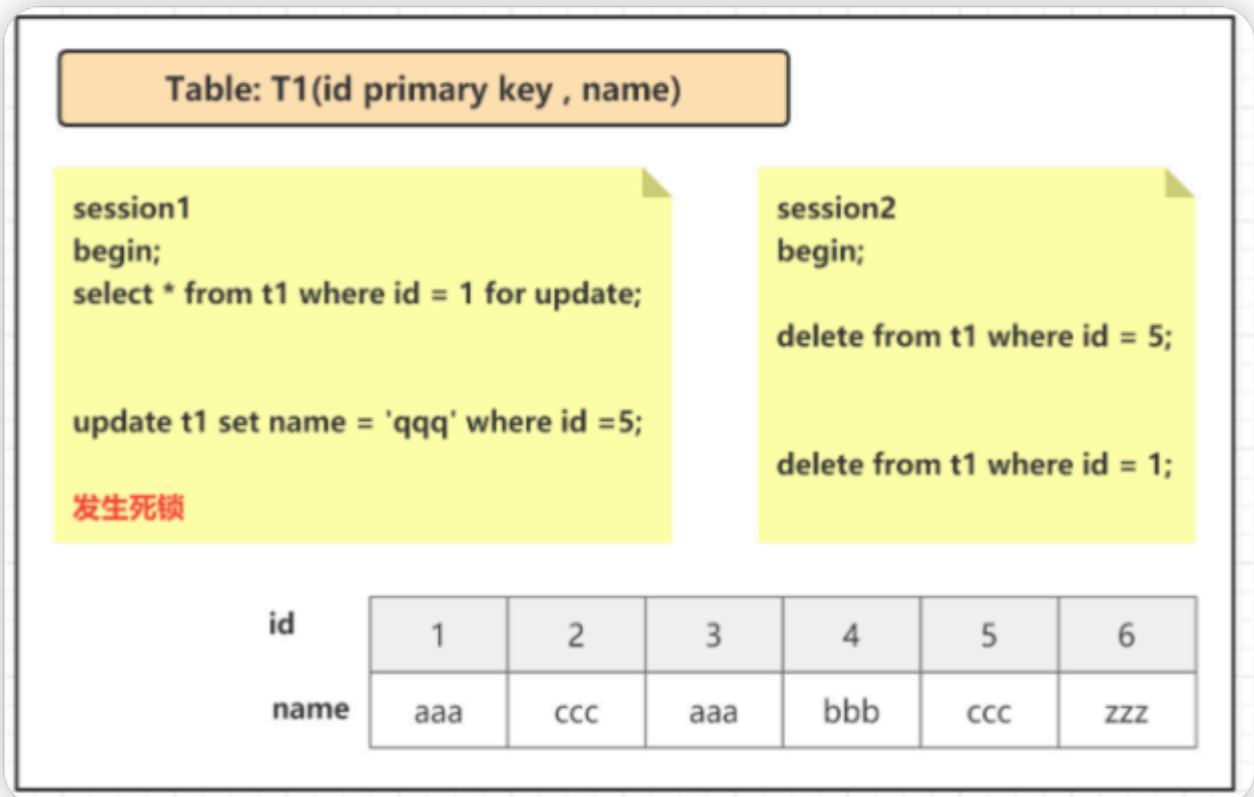
如果在事务中执行了一条没有索引条件的查询，引发全表扫描，把行级锁上升为全表记录锁定（等价于表级锁），多个这样的事务执行后，就很容易产生死锁和阻塞，最终应用系统会越来越慢，发生阻塞或死锁。

解决方案1：

SQL语句中不要使用太复杂的关联多表的查询；使用explain“执行计划"对SQL语句进行分析，对于有全表扫描和全表锁定的SQL语句，建立相应的索引进行优化。

产生原因2：

- 两个事务分别想拿到对方持有的锁，互相等待，于是产生死锁



产生原因3：每个事务只有一个SQL,但是有些情况还是会发生死锁.

1. 事务1,从name索引出发，读到的[hdc, 1]，[hdc, 6]均满足条件, 不仅会加name索引上的记录X锁, 而且会加聚簇索引上的记录X锁, 加锁顺序为先[1,hdc,100], 后[6,hdc,10]
2. 事务2，从pubtime索引出发，[10,6],[100,1]均满足过滤条件，同样也会加聚簇索引上的记录X锁，加锁顺序为[6,hdc,10]，后[1,hdc,100]。
3. 但是加锁时发现跟事务1的加锁顺序正好相反，两个Session恰好都持有了第一把锁，请求加第二把锁，死锁就发生了。

Table: T2(id primary key , name key , blogid,pubtime key ,comment)

session1
update t2 set comment = 'abc' where name = 'hdc';

session2
select * from t2 where pubtime > 5 for update;

key(name)

name	bbb	hdc	hdc	xxx	www	yyy
id	10	1	6	8	10	4

key(pubtime)

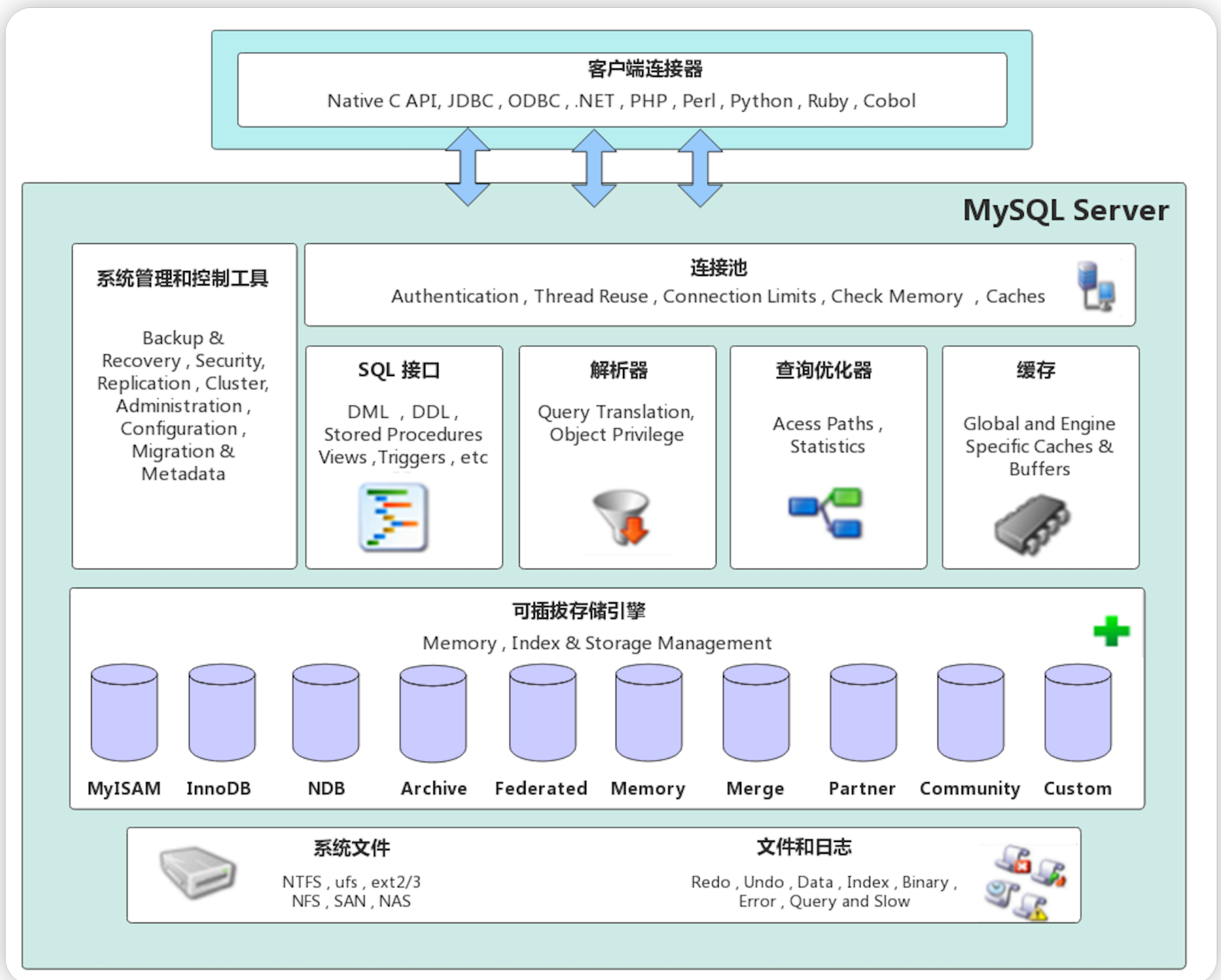
pubtime	1	3	5	10	4	100
id	10	4	8	6	100	1

Primary Key

id	1	4	6	8	10	100
name	hdc	yyy	hdc	xxx	www	bbb
blogid	a	b	c	d	e	f
pubtime	10	3	100	5	1	20
comment				good		

解决方案: 如上面的原因2和原因3，对索引加锁顺序的不一致很可能会导致死锁，所以如果可以，尽量以相同的顺序来访问索引记录和表。在程序以批量方式处理数据的时候，如果事先对数据排序，保证每个线程按固定的顺序来处理记录，也可以大大降低出现死锁的可能；

40.介绍一下MySQL的体系架构?



MySQL Server架构自顶向下大致可以分网络连接层、服务层、存储引擎层和系统文件层。

一、网络连接层

- 客户端连接器（Client Connectors）：提供与MySQL服务器建立的支持。目前几乎支持所有主流的服务端编程技术，例如常见的 Java、C、Python、.NET等，它们通过各自API技术与MySQL建立连接。

二、服务层（MySQL Server）

服务层是MySQL Server的核心，主要包含系统管理和控制工具、连接池、SQL接口、解析器、查询优化器和缓存六个部分。

- **连接池（Connection Pool）**：负责存储和管理客户端与数据库的连接，一个线程负责管理一个连接。
- **系统管理和控制工具（Management Services & Utilities）**：例如备份恢复、安全管理、集群管理等
- **SQL接口（SQL Interface）**：用于接受客户端发送的各种SQL命令，并且返回用户需要查询的结果。比如DML、DDL、存储过程、视图、触发器等。
- **解析器（Parser）**：负责将请求的SQL解析生成一个"解析树"。然后根据一些MySQL规则进一步检查解析树是否合法。
- **查询优化器（Optimizer）**：当"解析树"通过解析器语法检查后，将交由优化器将其转化成执行计划，然后与存储引擎交互。

  > **select uid,name from user where gender=1;**
  >
  > **选取--》投影--》联接 策略**
  >
  > **1）select先根据where语句进行选取，并不是查询出全部数据再过滤**
  >
  > **2）select查询根据uid和name进行属性投影，并不是取出所有字段**
  >
  > **3）将前面选取和投影联接起来最终生成查询结果**

- **缓存（Cache&Buffer）**： 缓存机制是由一系列小缓存组成的。比如表缓存，记录缓存，权限缓存，引擎缓存等。如果查询缓存有命中的查询结果，查询语句就可以直接去查询缓存中取数据。

三、**存储引擎层（Pluggable Storage Engines）**

存储引擎负责MySQL中数据的存储与提取，与底层系统文件进行交互。MySQL存储引擎是插件式的，服务器中的查询执行引擎通过接口与存储引擎进行通信，接口屏蔽了不同存储引擎之间的差异 。现在有很多种存储引擎，各有各的特点，最常见的是MyISAM和InnoDB。

四、系统文件层（File System）

该层负责将数据库的数据和日志存储在文件系统之上，并完成与存储引擎的交互，是文件的物理存储层。主要包含日志文件，数据文件，配置文件，pid 文件，socket 文件等。

- 日志文件
  - 错误日志（Error log）
    默认开启，show variables like '%log_error%'
  - 通用查询日志（General query log）
    记录一般查询语句，show variables like '%general%';
  - 二进制日志（binary log）
    记录了对MySQL数据库执行的更改操作，并且记录了语句的发生时间、执行时长；但是它不记录select、show等不修改数据库的SQL。主要用于数据库恢复和主从复制。
    show variables like '%log_bin%'; //是否开启
    show variables like '%binlog%'; //参数查看
    show binary logs;//查看日志文件
  - 慢查询日志（Slow query log）
    记录所有执行时间超时的查询SQL，默认是10秒。
    show variables like '%slow_query%'; //是否开启
    show variables like '%long_query_time%'; //时长
- 配置文件
  用于存放MySQL所有的配置信息文件，比如my.cnf、my.ini等。
- 数据文件

    - **db.opt 文件：记录这个库的默认使用的字符集和校验规则。**
    - **frm 文件：存储与表相关的元数据（meta）信息，包括表结构的定义信息等，每一张表都会有一个frm 文件。**
    - **MYD 文件：MyISAM 存储引擎专用，存放 MyISAM 表的数据（data)，每一张表都会有一个 .MYD 文件。**
    - **MYI 文件：MyISAM 存储引擎专用，存放 MyISAM 表的索引相关信息，每一张 MyISAM 表对应一个 .MYI 文件。**
    - **ibd文件和 IBDATA 文件：存放 InnoDB 的数据文件（包括索引）。InnoDB 存储引擎有两种表空间方式：独享表空间和共享表空间。独享表空间使用 .ibd 文件来存放数据，且每一张 InnoDB 表对应一个 .ibd 文件。共享表空间使用 .ibdata 文件，所有表共同使用一个（或多个，自行配置）.ibdata 文件。**
    - **ibdata1 文件：系统表空间数据文件，存储表元数据、Undo日志等。**
    - **ib_logfile0、ib_logfile1 文件：Redo log 日志文件。**
- **pid 文件**
  pid 文件是 mysqld 应用程序在 Unix/Linux 环境下的一个进程文件，和许多其他 Unix/Linux 服务端程序一样，它存放着自己的进程 id。
- **socket 文件**
  socket 文件也是在 Unix/Linux 环境下才有的，用户在 Unix/Linux 环境下客户端连接可以不通过 TCP/IP 网络而直接使用 Unix Socket 来连接 MySQL。

41.undo log、redo log、 bin log的作用是什么？

undo log 基本概念

- undo log是一种用于撤销回退的日志，在数据库事务开始之前，MySQL

会先记录更新前的数据到 undo log日志文件里面，当事务回滚时或者数据库崩溃时，可以利用 undo log来进行回退。
- Undo Log产生和销毁：Undo Log在事务开始前产生；事务在提交时，并不会立刻删除undo log，innodb会将该事务对应的undo log放入到删除列表中，后面会通过后台线程purge thread进行回收处理。

注意: undo log也会产生redo log，因为undo log也要实现持久性保护。

undo log的作用

1. **提供回滚操作【undo log实现事务的原子性】**
   在数据修改的时候，不仅记录了redo log，还记录了相对应的undo log，如果因为某些原因导致事务执行失败了，可以借助undo log进行回滚。
   *undo log 和 redo log 记录物理日志不一样，它是*逻辑日志。可以认为当delete一条记录时，undo log中会记录一条对应的insert记录，反之亦然，当update一条记录时，它记录一条对应相反的update记录。
2. **提供多版本控制(MVCC)【*undo log实现多版本并发控制（MVCC）*】**
   MVCC，即多版本控制。在MySQL数据库InnoDB存储引擎中，用undo Log来实现多版本并发控制(MVCC)。当读取的某一行被其他事务锁定时，它可以从undo log中分析出该行记录以前的数据版本是怎样的，从而让用户能够读取到当前事务操作之前的数据【快照读】。

redo log 基本概念

- InnoDB引擎对数据的更新，是先将更新记录写入redo log日志，然后会在系统空闲的时候或者是按照设定的更新策略再将日志中的内容更新到磁盘之中。这就是所谓的预写式技术（Write Ahead logging）。这种技术可以大大减少IO操作的频率，提升数据刷新的效率。
- redo log：被称作重做日志, 包括两部分：一个是内存中的日志缓冲：`redo log buffer`，另一个是磁盘上的日志文件：`redo log file`。

redo log的作用

- mysql 每执行一条 DML 语句，先将记录写入 redo log buffer 。后续某个时间点再一次性将多个操作记录写到 redo log file 。当故障发生致使内存数据丢失后，InnoDB会在重启时，经过重放 redo，将Page恢复到崩溃之前的状态 **通过Redo log可以实现事务的持久性** 。

bin log基本概念

- binlog是一个二进制格式的文件，用于记录用户对数据库更新的SQL语句信息，例如更改数据库表和更改内容的SQL语句都会记录到binlog里，但是不会记录SELECT和SHOW这类操作。
- binlog在MySQL的Server层实现(引擎共用)
- binlog为逻辑日志,记录的是一条SQL语句的原始逻辑
  - binlog不限制大小,追加写入,不会覆盖以前的日志.
  - 默认情况下，binlog日志是二进制格式的，不能使用查看文本工具的命令（比如，cat，vi等）查看，而使用mysqlbinlog解析查看。

bin log的作用

1. 主从复制：在主库中开启Binlog功能，这样主库就可以把Binlog传递给从库，从库拿到Binlog后实现数据恢复达到主从数据一致性。
2. 数据恢复：通过mysqlbinlog工具来恢复数据。

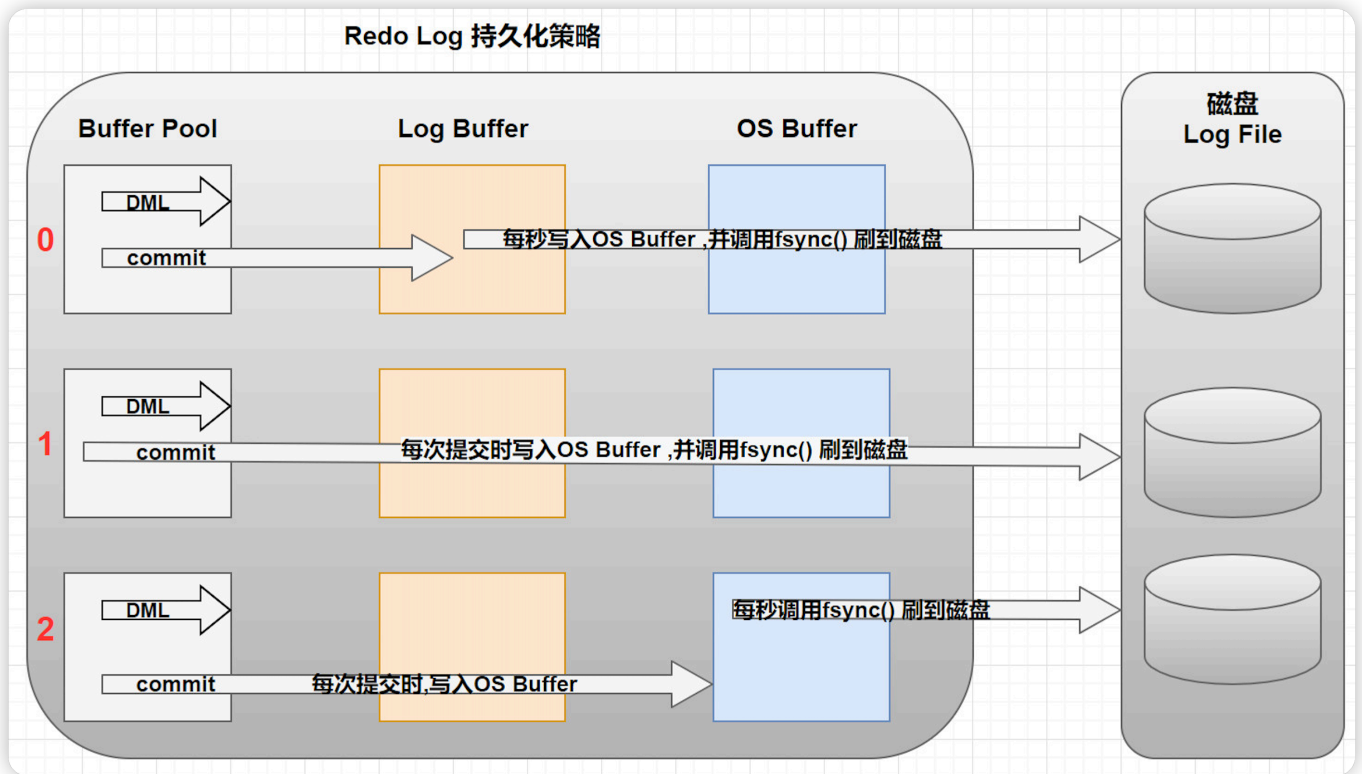
42.redo log与undo log的持久化策略?

redo log持久化

缓冲区数据一般情况下是无法直接写入磁盘的，中间必须经过操作系统缓冲区(OS Buffer)。因此， redo log buffer 写入 redo logfile 实际上是先写入 OS Buffer，然后再通过系统调用 fsync() 将其刷到 redo log file.

Redo Buffer 持久化到 redo log 的策略，可通过 `Innodb_flush_log_at_trx_commit` 设置：

参数值	含义
0 (延迟写)	事务提交时不会将 `redo log buffer`中日志写入到 `os buffer`， 而是每秒写入 `os buffer`并调用 `fsync()`写入到 `redo log file`中。 也就是说设置为0时是(大约)每秒刷新写入到磁盘中的，当系统崩溃，会丢失1秒钟的数据。
1 (实时写，实时刷)	事务每次提交都会将 `redo log buffer`中的日志写入 `os buffer`并 调用 `fsync()`刷到 `redo log file`中。这种方式即使系统崩溃也不会丢失任何数据，但是因为每次提交都写入磁盘，IO的性能较差。
2 (实时写，延时刷)	每次提交都仅写入到 `os buffer`，然后是每秒调用 `fsync()`将 `os buffer`中的日志写入到 `redo log file`。



一般建议选择取值2，因为 MySQL 挂了数据没有损失，整个服务器挂了才会损失1秒的事务提交数据

undo log持久化

MySQL中的Undo Log严格的讲不是Log，而是数据，因此他的管理和落盘都跟数据是一样的：

- Undo的磁盘结构并不是顺序的，而是像数据一样按Page管理
- Undo写入时，也像数据一样产生对应的Redo Log (因为undo也是对页面的修改，记录undo这个操作本身也会有对应的redo)。
- Undo的Page也像数据一样缓存在Buffer Pool中，跟数据Page一起做LRU换入换出，以及刷脏。Undo Page的刷脏也像数据一样要等到对应的Redo Log 落盘之后

当事务提交的时候，innodb不会立即删除undo log，因为后续还可能会用到undo log，如隔离级别为repeatable read时，事务读取的都是开启事务时的最新提交行版本，只要该事务不结束，该行版本就不能删除，即undo log不能删除。

但是在事务提交的时候，会将该事务对应的undo log放入到删除列表中，未来通过purge来删除。并且提交事务时，还会判断undo log分配的页是否可以重用，如果可以重用，则会分配给后面来的事务，避免为每个独立的事务分配独立的undo log页而浪费存储空间和性能。

43.bin log与undo log的区别?

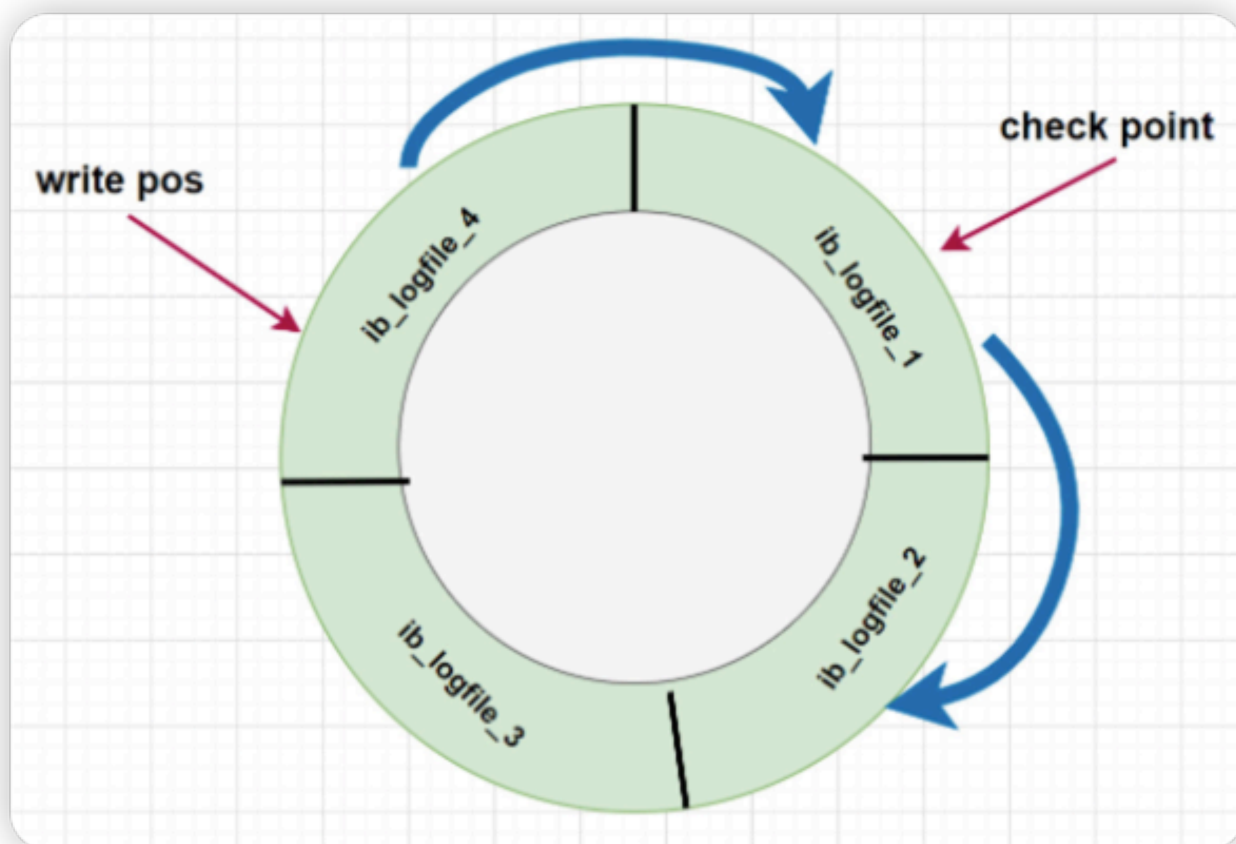
1）redo log是InnoDB引擎特有的；binlog是MySQL的Server层实现的，所有引擎都可以使用。

2）redo log是物理日志，记录的是“在XXX数据页上做了XXX修改”；binlog是逻辑日志，记录的是原始逻辑，其记录是对应的SQL语句。

- 物理日志: 记录的是每一个page页中具体存储的值是多少，在这个数据页上做了什么修改. 比如: 某个事物将系统表空间中的第100个页面中偏移量为1000处的那个字节的值1改为2.
- 逻辑日志: 记录的是每一个page页面中具体数据是怎么变动的，它会记录一个变动的过程或SQL语句的逻辑, 比如: 把一个page页中的一个数据从1改为2，再从2改为3,逻辑日志就会记录1->2,2->3这个数据变化的过程.

3）redo log是循环写的，空间一定会用完，需要write pos和check point搭配；binlog是追加写，写到一定大小会切换到下一个，并不会覆盖以前的日志

- Redo Log 文件内容是以顺序循环的方式写入文件，写满时则回溯到第一个文件，进行覆盖写。



- **write pos**: 表示日志当前记录的位置，当ib_logfile_4写满后，会从ib_logfile_1从头开始记录；
- **check point**: 表示将日志记录的修改写进磁盘，完成数据落盘，数据落盘后checkpoint会将日志上的相关记录擦除掉，即 `write pos -> checkpoint` 之间的部分是redo log空着的部分，用于记录新的记录，`checkpoint -> write pos` 之间是redo log 待落盘的数据修改记录
- 如果 write pos 追上 checkpoint，表示写满，这时候不能再执行新的更新，得停下来先擦掉一些记录，把 checkpoint 推进一下。

3）Redo Log作为服务器异常宕机后事务数据自动恢复使用，Binlog可以作为主从复制和数据恢复使用。Binlog没有自动crash-safe能力

CrashSafe指MySQL服务器宕机重启后，能够保证：

- 所有已经提交的事务的数据仍然存在。
- 所有没有提交的事务的数据自动回滚。

44.MySQL的binlog有几种日志格式？分别有什么区别？

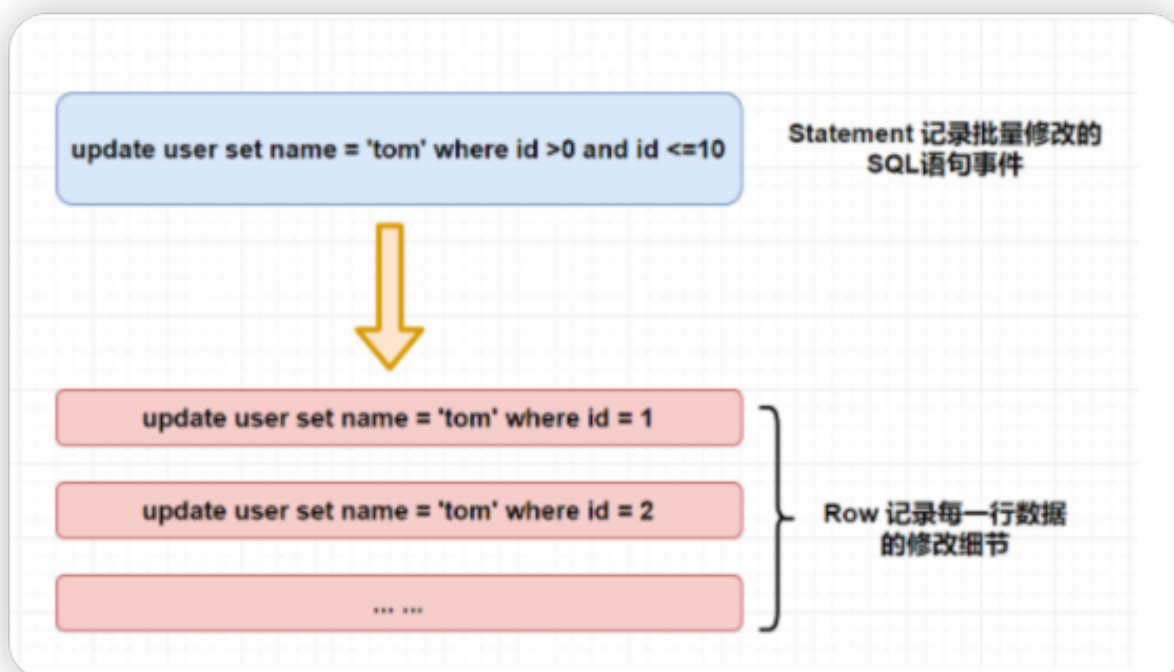
binlog日志有三种模式

1）ROW（row-based replication, RBR）：日志中会记录每一行数据被修改的情况，然后在slave端对相同的数据进行修改。

- 优点：能清楚记录每一个行数据的修改细节，能完全实现主从数据同步和数据的恢复。而且不会出现某些特定情况下存储过程或function无法被正确复制的问题。
- 缺点：批量操作，会产生大量的日志，尤其是alter table会让日志量暴涨。

2）STATMENT（statement-based replication, SBR）：记录每一条修改数据的SQL语句（批量修改时，记录的不是单条SQL语句，而是批量修改的SQL语句事件）, slave在复制的时候SQL进程会解析成和原来master端执行过的相同的SQL再次执行。简称SQL语句复制。

- 优点：日志量小，减少磁盘IO，提升存储和恢复速度
- 缺点：在某些情况下会导致主从数据不一致，比如last_insert_id()、now()等函数。



3）MIXED（mixed-based replication, MBR）：以上两种模式的混合使用，一般会使用STATEMENT模式保存binlog，对于STATEMENT模式无法复制的操作使用ROW模式保存binlog，MySQL会根据执行的SQL语句选择写入模式。

企业场景如何选择binlog的模式

1. 如果生产中使用MySQL的特殊功能相对少（存储过程、触发器、函数）。选择默认的语句模式，Statement。
2. 如果生产中使用MySQL的特殊功能较多的，可以选择Mixed模式。
3. 如果生产中使用MySQL的特殊功能较多，又希望数据最大化一致，此时最好Row 模式；但是要注意，该模式的binlog日志量增长非常快.

45.mysql 线上修改大表结构有哪些风险?

在线修改大表的可能影响

- 在线修改大表的表结构执行时间往往不可预估，一般时间较长。

- 由于修改表结构是表级锁，因此在修改表结构时，影响表写入操作。
- 如果长时间的修改表结构，中途修改失败，由于修改表结构是一个事务，因此失败后会还原表结构，在这个过程中表都是锁着不可写入。
- 修改大表结构容易导致数据库CPU、IO等性能消耗，使MySQL服务器性能降低。
- 在线修改大表结构容易导致主从延时，从而影响业务读取。

修改方式：

1. 对表加锁(表此时只读)
2. 复制原表物理结构
3. 修改表的物理结构
4. 把原表数据导入中间表中 ，数据同步完后，**锁定中间表，并删除原表
5. rename中间表为原表
6. 刷新数据字典，并释放锁

使用工具： **online-schema-change** ， 是percona推出的一个针对mysql在线ddl的工具。percona是一个mysql分支维护公司，专门提供mysql技术服务的。

46.count(列名)、count(1)和count(*)有什么区别?

进行统计操作时,count中的统计条件可以三种选择:

```
EXPLAIN  SELECT COUNT(*) FROM user;


EXPLAIN  SELECT COUNT(列名) FROM user;


EXPLAIN  SELECT COUNT(1) FROM user;
```

执行效果上：

- count(*) 包括了所有的列,在统计时 不会忽略列值为null的数据。
- count(1) 用1表示代码行,在统计时,不会忽略列值为null的数据。
- count(列名)在统计时,会忽略列值为空的数据,就是说某个字段的值为null时不统计。

执行效率上：

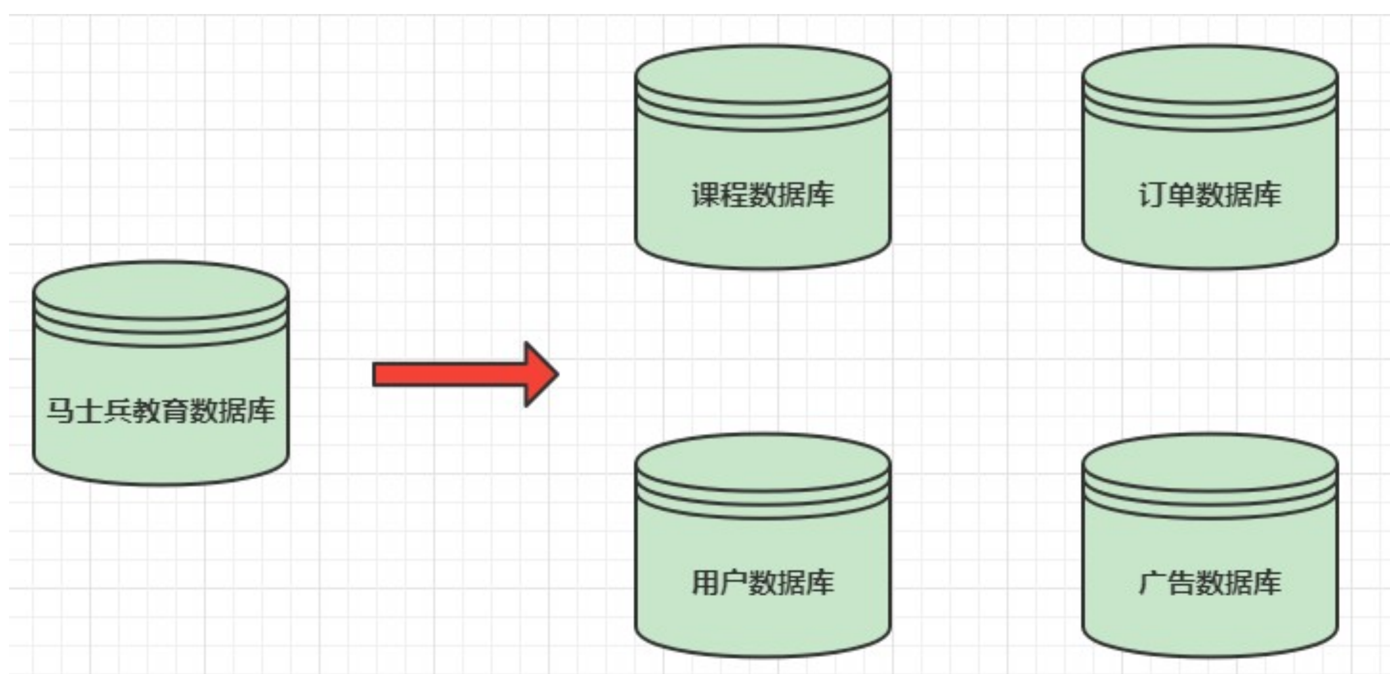
- InnoDB引擎：count（字段) < count(1) = count(*)
    - InnoDB通过遍历最小的可用二级索引来处理select count(*) 语句，除非索引或优化器提示指示优化器使用不同的索引。如果二级索引不存在，则通过扫描聚集索引来处理。
    - InnoDB已同样的方式处理count(1)和count(*)
- MyISAM引擎：count（字段) < count(1) <= count(*)
    - MyISAM存储了数据的准确行数，使用 `count(*)`会直接读取该行数， 只有当第一列定义为NOT NULL时，count（1），才会执行该操作，所以优先选择 `count(*)`
- count(列名) 会遍历整个表，但不同的是，它会先获取列，然后判断是否为空，然后累加，因此count(列名)性能不如前两者。

注意：count(*)，这是SQL92 定义的标准统计行数的语法，跟数据库无关，与NULL也无关。而count(列名) 是统计列值数量，不计NULL，相同列值算一个。

47.什么是分库分表？什么时候进行分库分表?

什么是分库分表

简单来说，就是指通过某种特定的条件，将我们存放在同一个数据库中的数据分散存放到多个数据库（主机）上面，以达到分散单台设备负载的效果。



- 分库分表解决的问题

  分库分表的目的是为了解决由于数据量过大而导致数据库性能降低的问题，将原来单体服务的数据库进行拆分.将数据大表拆分成若干数据表组成，使得单一数据库、单一数据表的数据量变小，从而达到提升数据库性能的目的。

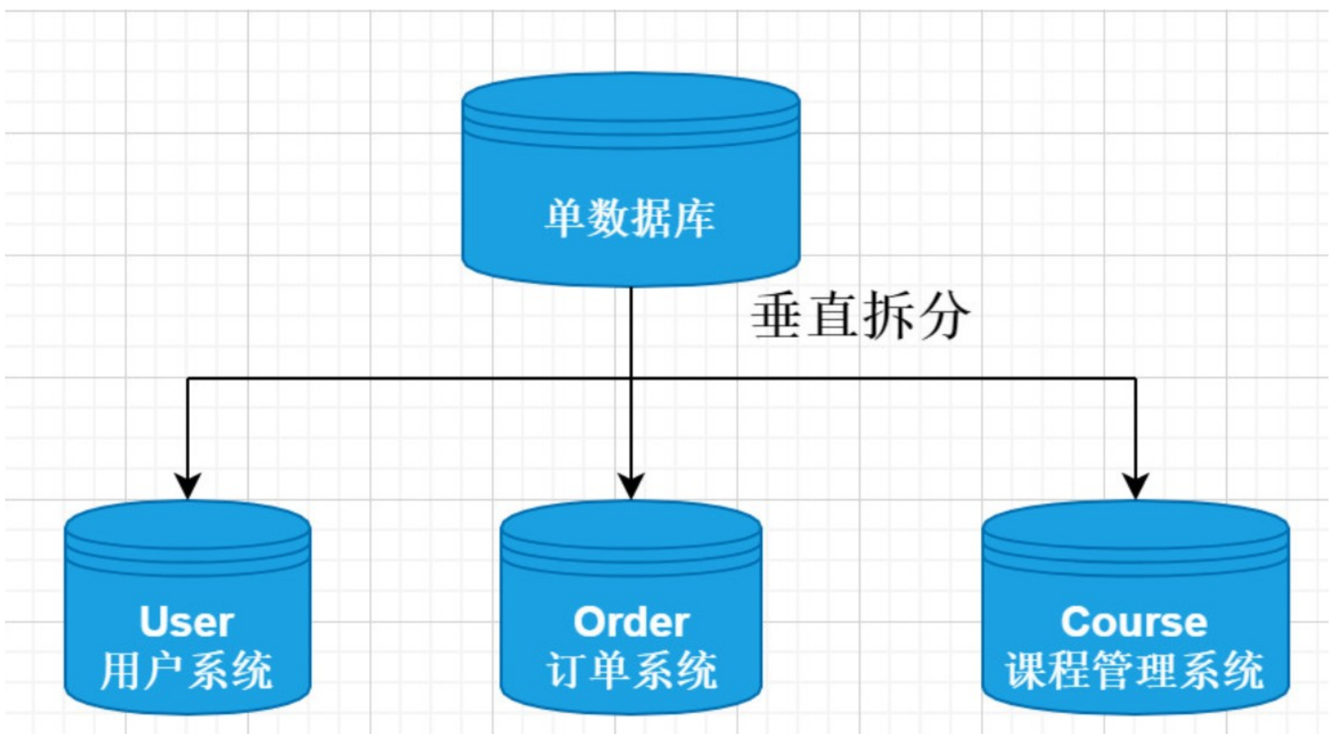
- 什么情况下需要分库分表
  - **单机存储容量遇到瓶颈.**
  - **连接数,处理能力达到上限.**

> 注意:
>
> 分库分表之前,要根据项目的实际情况 确定我们的数据量是不是够大,并发量是不是够大,来决定是否分库分表.
>
> 数据量不够就不要分表,单表数据量超过1000万或100G的时候, 速度就会变慢(官方测试),

分库分表包括： 垂直分库、垂直分表、水平分库、水平分表 四种方式。

垂直分库

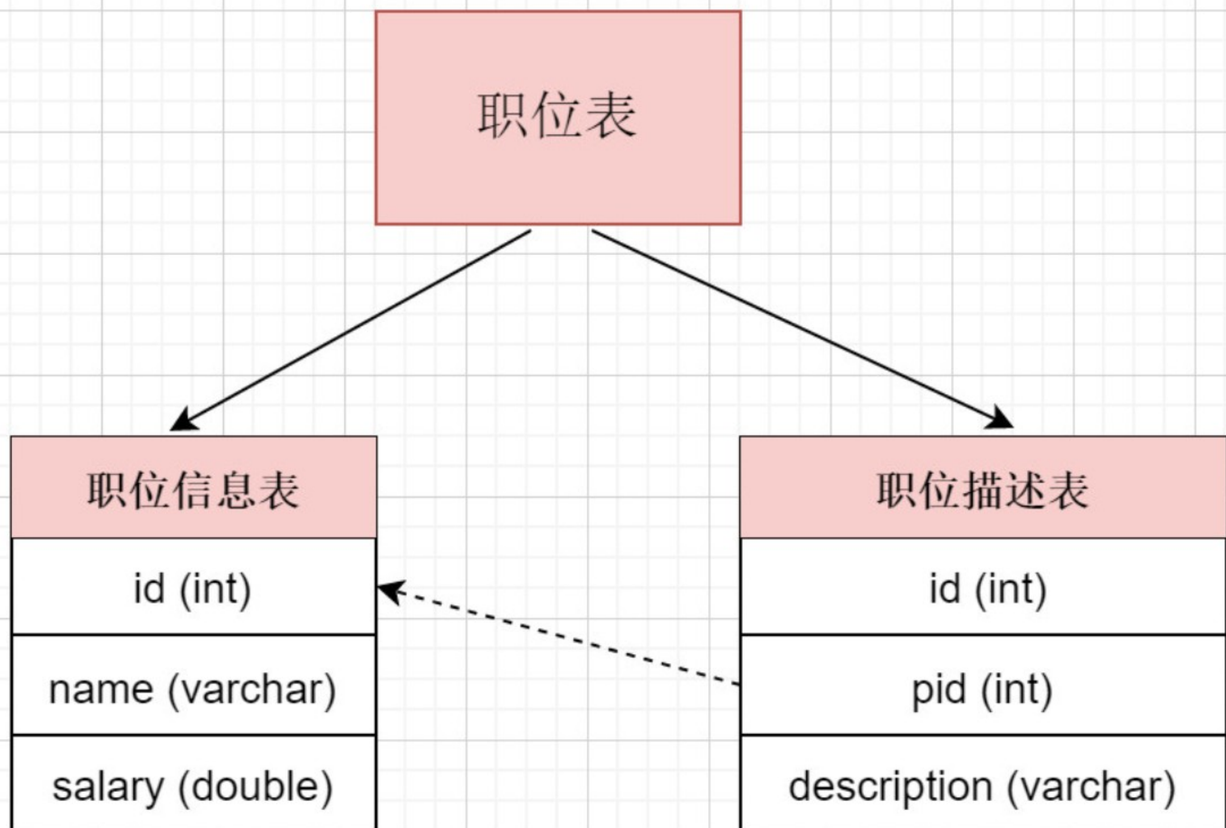
- 数据库中不同的表对应着不同的业务，垂直切分是指按照业务的不同将表进行分类,分布到不同的数据库上面
    - 将数据库部署在不同服务器上，从而达到多个服务器共同分摊压力的效果



垂直分表

表中字段太多且包含大字段的时候，在查询时对数据库的IO、内存会受到影响，同时更新数据时，产生的binlog文件会很大，MySQL在主从同步时也会有延迟的风险

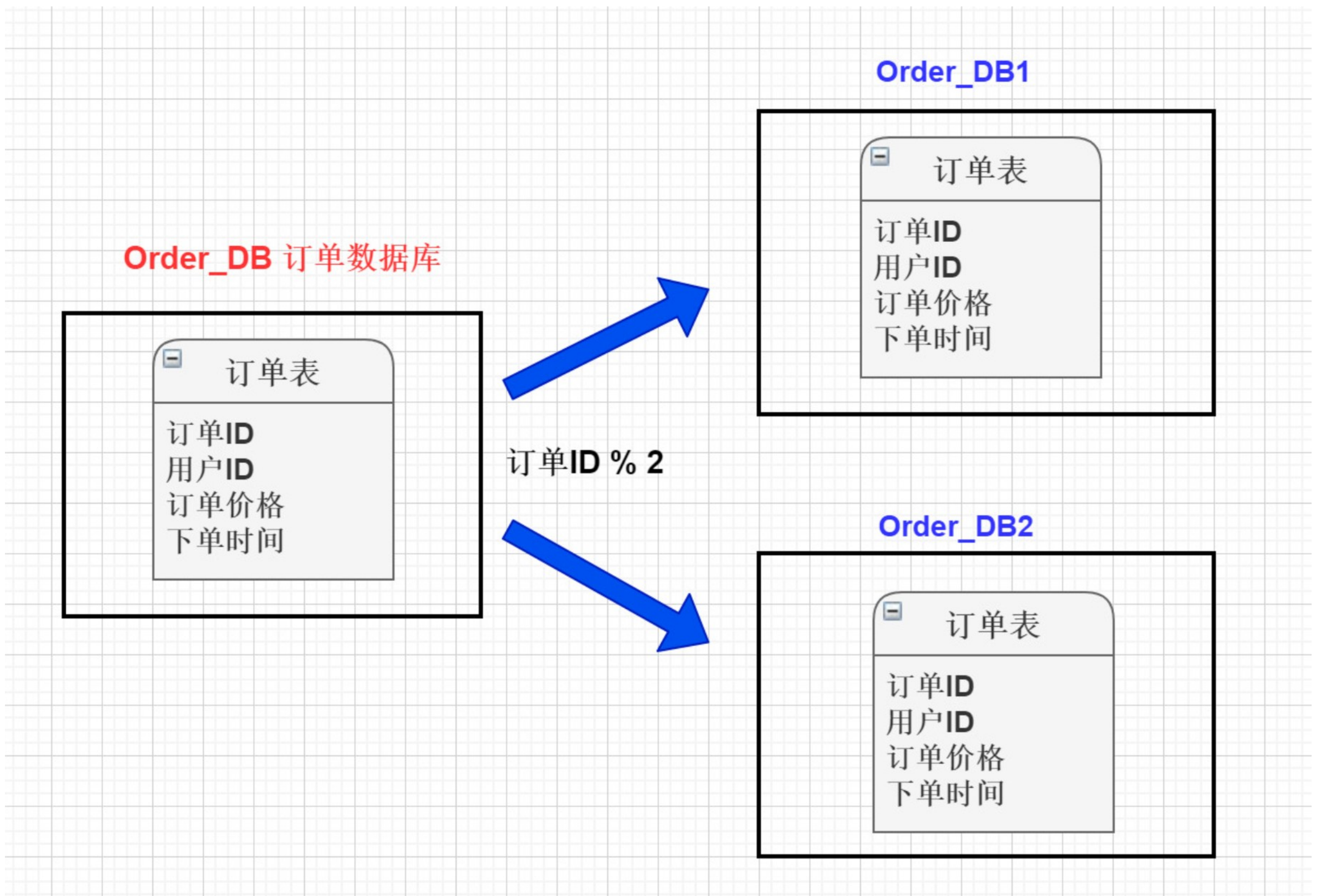
- 将一个表按照字段分成多表，每个表存储其中一部分字段。
- 对职位表进行垂直拆分, 将职位基本信息放在一张表, 将职位描述信息存放在另一张表



- 垂直拆分带来的一些提升
  - 解决业务层面的耦合，业务清晰
  - 能对不同业务的数据进行分级管理、维护、监控、扩展等
  - 高并发场景下，垂直分库一定程度的提高访问性能
- 垂直拆分没有彻底解决单表数据量过大的问题

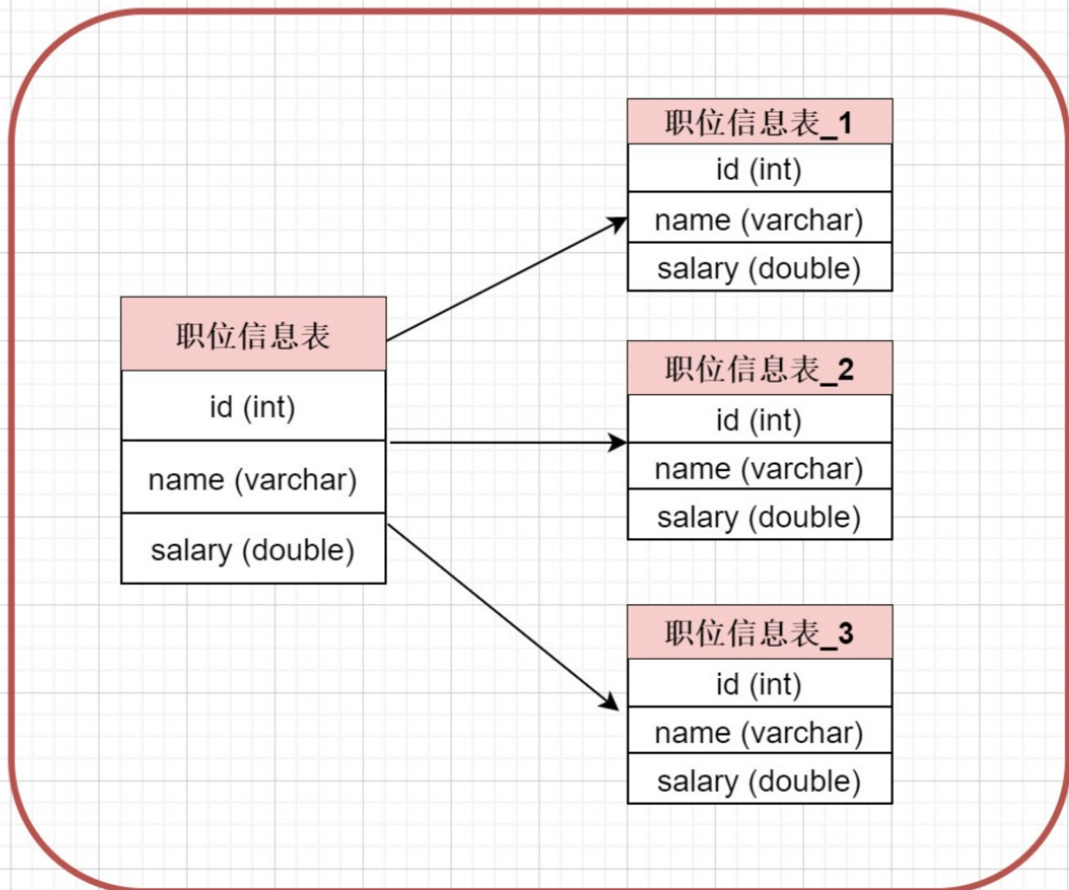
水平分库

- 将单张表的数据切分到多个服务器上去，每个服务器具有相应的库与表，只是表中数据集合不同。水平分库分表能够有效的缓解单机和单库的性能瓶颈和压力，突破IO、连接数、硬件资源等的瓶颈.
- 简单讲就是根据表中的数据的逻辑关系，将同一个表中的数据按照某种条件拆分到多台数据库（主机）上面, 例如将订单表 按照id是奇数还是偶数, 分别存储在不同的库中。



水平分表

- 针对数据量巨大的单张表（比如订单表），按照规则把一张表的数据切分到多张表里面去。 但是这些表还是在同一个库中，所以库级别的数据库操作还是有IO瓶颈。



- 总结
  - **垂直分表**: 将一个表按照字段分成多表，每个表存储其中一部分字段。
  - **垂直分库**: 根据表的业务不同,分别存放在不同的库中,这些库分别部署在不同的服务器.
  - **水平分库**: 把一张表的数据按照一定规则,分配到**不同的数据库**,每一个库只有这张表的部分数据.
  - **水平分表**: 把一张表的数据按照一定规则,分配到**同一个数据库的多张表中**,每个表只有这个表的部分数据.

48.说说 MySQL 的主从复制?

主从复制的用途

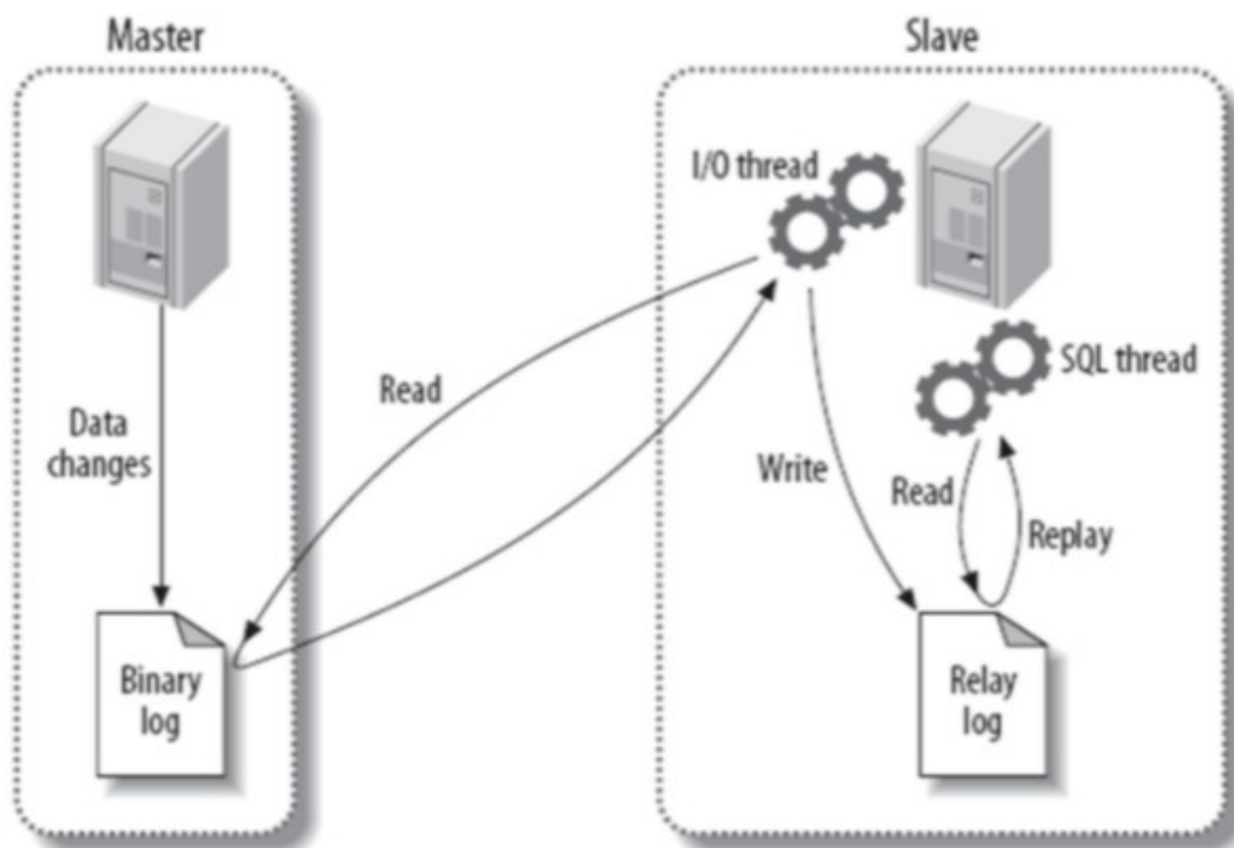
- 实时灾备，用于故障切换
- 读写分离，提供查询服务
- 备份，避免影响业务

主从部署必要条件

- 主库开启binlog日志（设置log-bin参数）
- 主从server-id不同
- 从库服务器能连通主库

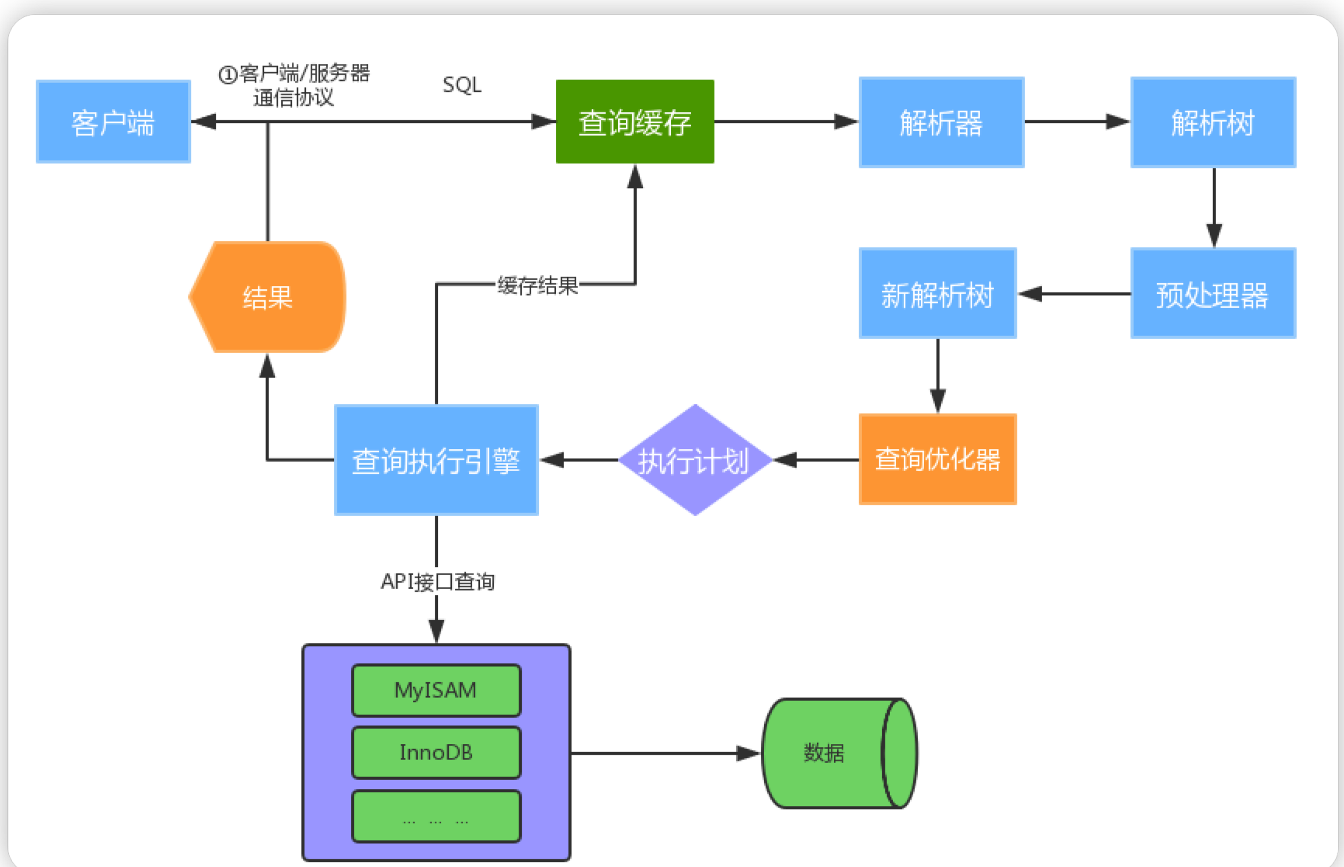
主从复制的原理

- Mysql 中有一种日志叫做 bin 日志（二进制日志）。这个日志会记录下所有修改了数据库的SQL 语句（insert,update,delete,create/alter/drop table, grant 等等）。
- 主从复制的原理其实就是把主服务器上的 bin 日志复制到从服务器上执行一遍，这样从服务器上的数据就和主服务器上的数据相同了。



1. 主库db的更新事件(update、insert、delete)被写到binlog
2. 主库创建一个binlog dump thread，把binlog的内容发送到从库
3. 从库启动并发起连接，连接到主库
4. 从库启动之后，创建一个I/O线程，读取主库传过来的binlog内容并写入到relay log
5. 从库启动之后，创建一个SQL线程，从relay log里面读取内容，执行读取到的更新事件，将更新内容写入到slave的db

49. 说一下 MySQL 执行一条查询语句的内部执行过程？



- ①建立连接（Connectors&Connection Pool），通过客户端/服务器通信协议与MySQL建立连接。MySQL 客户端与服务端的通信方式是 “ 半双工 ”。对于每一个 MySQL 的连接，时刻都有一个线程状态来标识这个连接正在做什么。
  通讯机制：
  - 全双工：能同时发送和接收数据，例如平时打电话。
  - 半双工：指的某一时刻，要么发送数据，要么接收数据，不能同时。例如早期对讲机
  - 单工：只能发送数据或只能接收数据。例如单行道

  线程状态：
  show processlist; //查看用户正在运行的线程信息，root用户能查看所有线程，其他用户只能看自己的
  - id：线程ID，可以使用kill xx；
  - user：启动这个线程的用户
  - Host：发送请求的客户端的IP和端口号
  - db：当前命令在哪个库执行
  - Command：该线程正在执行的操作命令
    - Create DB：正在创建库操作
    - Drop DB：正在删除库操作
    - Execute：正在执行一个PreparedStatement
    - Close Stmt：正在关闭一个PreparedStatement
    - Query：正在执行一个语句
    - Sleep：正在等待客户端发送语句
    - Quit：正在退出
    - Shutdown：正在关闭服务器
  - Time：表示该线程处于当前状态的时间，单位是秒

    - State：线程状态
        - Updating：正在搜索匹配记录，进行修改
        - Sleeping：正在等待客户端发送新请求
        - Starting：正在执行请求处理
        - Checking table：正在检查数据表
        - Closing table : 正在将表中数据刷新到磁盘中
        - Locked：被其他查询锁住了记录
        - Sending Data：正在处理Select查询，同时将结果发送给客户端
    - Info：一般记录线程执行的语句，默认显示前100个字符。想查看完整的使用show full processlist;

- ②查询缓存（Cache&Buffer），这是MySQL的一个可优化查询的地方，如果开启了查询缓存且在查询缓存过程中查询到完全相同的SQL语句，则将查询结果直接返回给客户端；如果没有开启查询缓存或者没有查询到完全相同的 SQL 语句则会由解析器进行语法语义解析，并生成“解析树”。
    - 缓存Select查询的结果和SQL语句
    - 执行Select查询时，先查询缓存，判断是否存在可用的记录集，要求是否完全相同（包括参数值），这样才会匹配缓存数据命中。
    - 即使开启查询缓存，以下SQL也不能缓存
        - 查询语句使用SQL_NO_CACHE
        - 查询的结果大于query_cache_limit设置
        - 查询中有一些不确定的参数，比如now()
    - show variables like '%query_cache%'; //查看查询缓存是否启用，空间大小，限制等
    - show status like 'Qcache%'; //查看更详细的缓存参数，可用缓存空间，缓存块，缓存多少等

- ③解析器（Parser）将客户端发送的SQL进行语法解析，生成"解析树"。预处理器根据一些MySQL规则进一步检查“解析树”是否合法，例如这里将检查数据表和数据列是否存在，还会解析名字和别名，看看它们是否有歧义，最后生成新的“解析树”。
- ④查询优化器（Optimizer）根据“解析树”生成最优的执行计划。MySQL使用很多优化策略生成最优的执行计划，可以分为两类：静态优化（编译时优化）、动态优化（运行时优化）。
  - 等价变换策略
    - 5=5 and a>5 改成 a > 5
    - a < b and a=5 改成b>5 and a=5
    - 基于联合索引，调整条件位置等
  - 优化count、min、max等函数
    - InnoDB引擎min函数只需要找索引最左边
    - InnoDB引擎max函数只需要找索引最右边
    - MyISAM引擎count(*)，不需要计算，直接返回
  - 提前终止查询
    - 使用了limit查询，获取limit所需的数据，就不在继续遍历后面数据
  - in的优化
    - MySQL对in查询，会先进行排序，再采用二分法查找数据。比如where id in (2,1,3)，变成 in (1,2,3)
- ⑤查询执行引擎负责执行 SQL 语句，此时查询执行引擎会根据 SQL 语句中表的存储引擎类型，以及对应的API接口与底层存储引擎缓存或者物理文件的交互，得到查询结果并返回给客户端。若开启用查询缓存，这时会将SQL 语句和结果完整地保存到查询缓存（Cache&Buffer）中，以后若有相同的 SQL 语句执行则直接返回结果。
  - 如果开启了查询缓存，先将查询结果做缓存操作

    - 返回结果过多，采用增量模式返回

50.Mysql内部支持缓存查询吗？

使用缓存的好处：当MySQL接收到客户端的查询SQL之后，仅仅只需要对其进行相应的权限验证之后，就会通过Query Cache来查找结果，甚至都不需要经过Optimizer模块进行执行计划的分析优化，更不需要发生任何存储引擎的交互.

mysql5.7支持内部缓存，8.0之后已废弃

mysql缓存的限制

1. mysql基本没有手段灵活的管理缓存失效和生效，尤其对于频繁更新的表
2. SQL必须完全一致才会导致cache命中
3. 为了节省内存空间，太大的result set不会被cache (< query_cache_limit)；
4. MySQL缓存在分库分表环境下是不起作用的；
5. 执行SQL里有触发器,自定义函数时，MySQL缓存也是不起作用的；
6. 在表的结构或数据发生改变时，基于该表相关cache立即全部失效。

替代方案

- 应用层组织缓存，最简单的是使用redis，ehcached等