文档图片归OldGuo所有，禁止转载。

第九章 OldGuo 师徒班-MySQL存储引擎

9.1 存储引擎介绍

相当于Linux 中的文件系统，和“磁盘”。负责IO的控制（磁盘资源、内存资源、对象资源）。

9.2 不同存储引擎特性差异

```
db01 [test]>show engines;
```

```
percona  MariaDB等产品可能会用到的存储引擎。
Tokudb特性：
Additional features unique to TokuDB include:
Up to 25x Data Compression
Fast Inserts
Eliminates Slave Lag with Read Free Replication
Hot Schema Changes
Hot Index Creation - TokuDB tables support insertions, deletions and queries
with no down time while indexes are being added to that table
Hot column addition, deletion, expansion, and rename - TokuDB tables support
insertions, deletions and queries without down-time when an alter table adds,
deletes, expands, or renames columns
On-line Backup
或者：
Myrocks
rocksdb ---> LSM TREE   ---> TiDB

参考内容：
https://mariadb.com/kb/en/installing-tokudb/
https://www.percona.com/doc/percona-server/5.7/tokudb/tokudb_installation.html
https://www.jianshu.com/p/898d2e4bd3a7

监控类（zabbix），历史数据类，边缘业务数据存储需求可以考虑。其他推荐：MongoDB HBASE。
```

9.3 InnoDB 存储引擎核心特性介绍

```
MVCC          ：  多版本并发控制
聚簇索引      ：  用来组织存储数据和优化查询
支持事务      ：  数据最终一致提供保证
支持行级锁    ：  并发控制
外键          ：  多表之间的数据一致一致性
多缓冲区支持
```

```
自适应Hash索引： AHI
复制中支持高级特性。
备份恢复： 支持热备。
自动故障恢复：CR  Crash Recovery
双写机制 ： DWB  Double Write Buffer

MyISAM和InnoDB的区别？
```

9.4 存储引擎管理

9.4.1 查询支持的存储引擎

```
查询所有可用的存储引擎种类
mysql> show engines;

查询、设置默认存储引擎
mysql> select @@default_storage_engine;
+--------------------------+
| @@default_storage_engine |
+--------------------------+
| InnoDB                   |
+--------------------------+
1 row in set (0.00 sec)
vim /etc/my.cnf
default_storage_engine=InnoDB
```

9.4.2 查看某张表的存储引擎

```
 mysql> show create table xta;
查询系统中所有业务表的存储引擎信息
 mysql> select
 table_schema,
 table_name ,
 engine
 from information_schema.tables
 where table_schema not in
('sys','mysql','information_schema','performance_schema');

 巡检需求： 将业务表中所有非InnoDB查询出来
mysql> select table_schema,table_name,engine from information_schema.tables
where table_schema not in
('mysql','sys','information_schema','performance_schema') and engine !='innodb';
```

9.4.2 创建表设定存储引擎

```
 mysql> create table xxx (id int) engine=innodb charset=utf8mb4;
```

9.4.2 修改已有表的存储引擎

```
 mysql> alter table world.xxx engine=innodb;
```

9.4.2 将所有的非InnoDB引擎的表查询出来，批量修改为InnoDB

```
客户案例：将历史遗留下的非InnoDB表进行处理
1.查询所有非InnoDB表
mysql> select table_schema,table_name ,engine
  from information_schema.tables
  where
  table_schema not in ('sys','mysql','information_schema','performance_schema')
  and engine !='innodb';
2. 备份所有非InnoDB表
select concat("mysqldump -uroot -p123 ",table_schema," ",table_name,"
>/tmp/",table_schema,"_",table_name,".sql")
  from information_schema.tables
  where
  table_schema not in ('sys','mysql','information_schema','performance_schema')
  and engine !='innodb';
3. 修改存储引擎
mysql> select concat("alter table ",table_schema,".",table_name,"
engine=innodb;")   from information_schema.tables     where    table_schema not
in ('sys','mysql','information_schema','performance_schema')    and engine
!='innodb' into outfile '/tmp/a.sql';
 mysql> source /tmp/a.sql
```

9.4.5 碎片情况

```
查询碎片：
information_schema.tables;  ---> DATA_FREE

alter table world.xxx engine=innodb ALGORITHM=COPY;
analyze table world.city;

转储表：
create table t1_bak like t1;
insert into t1_bak selewct * from t1;
drop table t1 ;
rename table t1_bak to t1;

mysqldump 导出  导入。

或者工具
pt-os
ghost

注意：
    1. 最好是空窗期做
    2. 准备double的存储空间 tmpdir
    3. 整理碎片只对 InnoDB 独立表空间方式有效。
```

9.5 InnoDB体系结构---线程和内存结构详解(X)

9.5.1 线程结构

```
# 1. 前台线程（连接层）
show processlist ;  show full processlist;
select * from information_schema.processlist ;

# 2. 后台线程（Server\Engine）
mysql> select * from performance_schema.threads\G

说明： 如何查询到连接线程和SQL线程关系

select * from information_schema.processlist ;   ---> ID=10
select * from performance_schema.threads where processlist_id=10\G
select * from performance_schema.events_statements_history where thread_id=?
```

Master Thread

```
a. 控制刷新脏页到磁盘（CKPT）
b. 控制日志缓冲刷新到磁盘（log buffer ---> redo）
c. undo页回收
d. 合并插入缓冲(change buffer)
e. 控制IO刷新数量

说明：
参数innodb_io_capacity表示每秒刷新脏页的数量，默认为200。
innodb_max_dirty_pages_pct设置出发刷盘的脏页百分比，即当脏页占到缓冲区数据达到这个百分比时，就会刷新innodb_io_capacity个脏页到磁盘。

参数innodb_adaptive_flushing = ON（自适应地刷新），该值影响每秒刷新脏页的数量。原来的刷新规则是：脏页在缓冲池所占的比例小于innodb_max_dirty_pages_pct时，不刷新脏页；大于innodb_max_dirty_pages_pct时，刷新100个脏页。
随着innodb_adaptive_flushing参数的引入，InnoDB存储引擎会通过一个名为buf_flush_get_desired_flush_rate的函数来判断需要刷新脏页最合适的数量。粗略地翻阅源代码后发现buf_flush_get_desired_flush_rate通过判断产生重做日志（redo log）的速度来决定最合适的刷新脏页数量。因此，当脏页的比例小于innodb_max_dirty_pages_pct时，也会刷新一定量的脏页。
```

IO Thread

```
在InnoDB存储引擎中大量使用Async IO来处理写IO请求,IO Thread的工作主要是负责这些IO请求的回调处理。
写线程和读线程分别由innodb_write_threads和innodb_read_threads参数控制，默认都为4。
```

Purge Thread

```
事务在提交之前，通过undolog(回滚日志)记录事务开始之前的状态，当事务被提交后，undolog便不再需要，因此需要Purge Thread线程来回收已经使用并分配的undo页。可以在配置文件中添加innodb_purge_threads=1来开启独立的Purge Thread，等号后边控制该线程数量，默认为4个。
```

Page Cleaner Thread

```
InnoDB 1.2.X版本以上引入，脏页刷新，减轻master的工作，提高性能。
查看方法：
    客户端版本：
    mysql -V
    server 版本：
    select @@version;
    engine 版本：
    SELECT * FROM information_schema.plugins;
    SELECT @@innodb_version;
```

MySQL版本	Innodb引擎版本
5.1.x	1.0.x版本(官方称为InnoDB Plugin)
5.5.x	1.1.x版本
5.6.x	1.2.x版本

9.4.2 内存结构（ In-Memory Structures)

> 图片出处： https://dev.mysql.com/doc/refman/8.0/en/innodb-architecture.html

InnoDB Buffer Pool(IBP)

```
作用：
    用来缓冲、缓存，MySQL的数据页（data page ）和索引页、UNDO。MySQL中最大的、最重要的内存区域。
管理：
    查询：
    > select @@innodb_buffer_pool_size;
    > select @@innodb_buffer_pool_instances;
    默认大小： 128M
    生产建议： 物理内存的：50-75%。
    在线设置：

    mysql> set global innodb_buffer_pool_size = 48318382080;

    重新登录mysql生效。
    永久设置：
    vim /etc/my.cnf
    #添加参数
    innodb_buffer_pool_size=256M
    show global status like '%innodb%wait%';

产生不够用的情景有哪些？
1. 设置太小。
2. 大事务。
3. ckpt触发不及时。
4. IO比较慢。
5. 查询语句优化的不好。


内部结构：参考官方文档
```

出处：https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool.html
1. 采用LRU：最近最少使用原则。
2. LRU链表：“热端”（5/8）+“冷端” （3/8）
3. 新入内存的page，会被放在冷端的头部。
4. 如果page被访问，会被调取到热端的头部。

data buffer

磁盘上加载数据页缓冲区域

Change buffer

比如insert，update，delete 数据。

对于聚簇索引会立即更新。

对于不在buffer pool的辅助索引需要更新，不是实时更新的。

在InnoDB 内存结构中，加入了insert buffer，现在版本叫change buffer。

Change buffer 功能是临时缓冲辅助索引需要的数据更新。

当我们需要查询新insert 的数据，会在内存中进行merge(合并)操作，此时辅助索引就是最新的。

The innodb_change_buffer_max_size variable permits configuring the maximum size of the change buffer as a percentage of the total size of the buffer pool. By default, innodb_change_buffer_max_size is set to 25. The maximum setting is 50. Consider increasing innodb_change_buffer_max_size on a MySQL server with heavy insert, update, and delete activity, where change buffer merging does not keep pace with new change buffer entries, causing the change buffer to reach its maximum size limit.
Consider decreasing innodb_change_buffer_max_size on a MySQL server with static data used for reporting, or if the change buffer consumes too much of the memory space shared with the buffer pool, causing pages to age out of the buffer pool sooner than desired.
Test different settings with a representative workload to determine an optimal configuration. The innodb_change_buffer_max_size setting is dynamic, which permits modifying the setting without restarting the server.

AHI 自适应hash索引

MySQL的InnoDB引擎，能够创建只有Btree。
AHI作用：
自动评估"热"的内存索引page,生成HASH索引表。
帮助InnoDB快速读取索引页。加快索引读取的效果。
相当与索引的索引。

metux 闩锁争用。

Log Buffer（redo buffer)

作用： 用来缓冲 redo log日志信息。
管理 ：

```
        查询：
        mysql> select @@innodb_log_buffer_size;
        默认大小：16M
        生产建议：和innodb_log_file_size有关，1-2倍

        设置方式 ：
        vim /etc/my.cnf
        innodb_log_buffer_size=33554432

24C 96G
innodb_log_file_size=2G
innodb_log_files_in_group=3
innodb_log_buffer_size=1G

重启生效：
[root@db01 data]# /etc/init.d/mysqld restart
show global status like '%innodb%log%';
```

9.5 InnoDB体系结构---物理存储结构详解

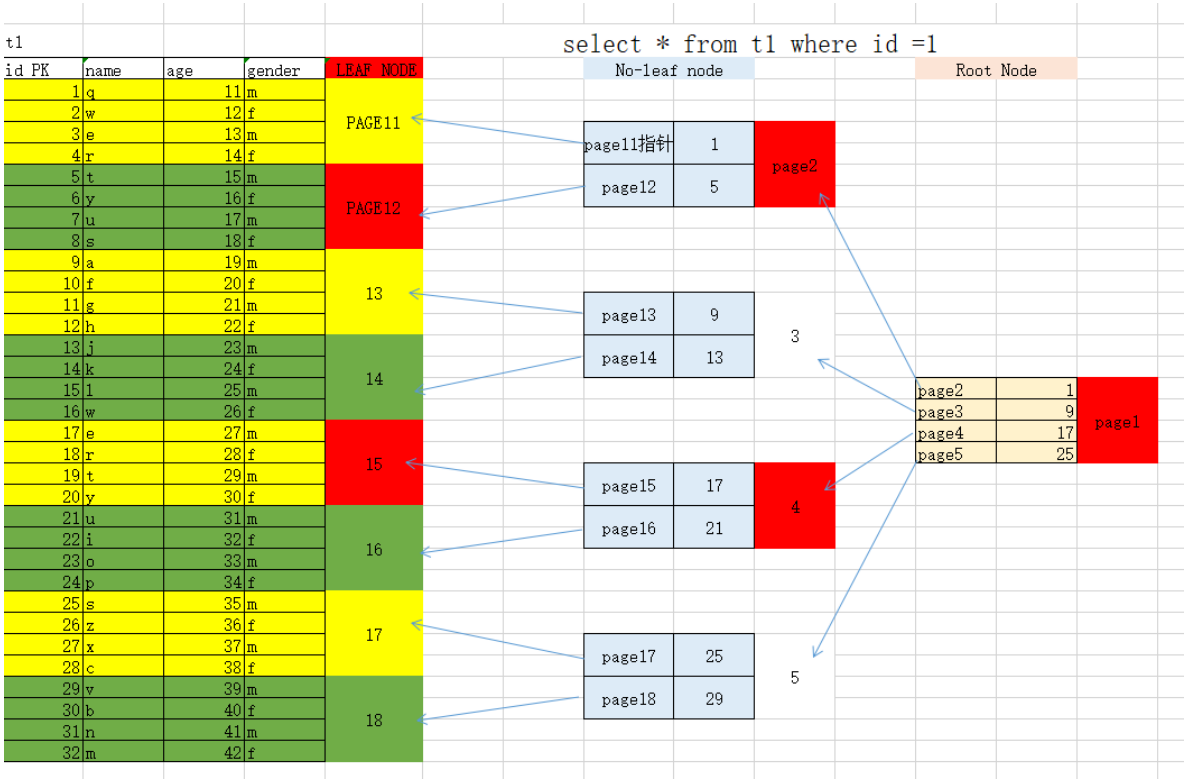
9.5.1 表

```
8.0 以前 InnoDB表：
ibd  ：  数据和索引
frm  :  存私有的数据字典信息
ibdataN：系统的数据字典信息

8.0 在数据字典改变
ibd：数据和索引+ 冗余的SDI私有数据字典信息
取消了 ibdata中的系统数据字典信息。

mysql.ibd  ---> 整个系统的系统数据
sdi序列化的数据字典----> 每个表的表空间自行管理json格式的私有数据字典信息，用来替换frm的。
```

9.5.2 索引

聚簇索引

```
InnoDB表，永远存在的一类索引。
构建前提：
        1. pk的列会自动作为聚簇索引。
        2. 没有pk ,会选择非空UK。
        3. 都没有，生成隐藏的ROW_ID(6字节)
功能：
       1. IOT，索引组织表。
       2. 针对ID条件查询快速找到记录。
结构简述：
       leaf ：  按照ID顺序逻辑上有序存储的数据行。在同一个区内的数据页使用物理连续。
       non-leaf:
              root: 下层节点的ID范围+指针
              内部节点：也可以理解为枝节点,下层节点的ID范围+指针
```




```
insert    33,s,18,m
```

辅助索引

结构简述：如图

leaf ： “提取” 辅助索引(name)列值+ID列值，根据辅助列(name)值排序，生成叶子节点。

non-leaf:

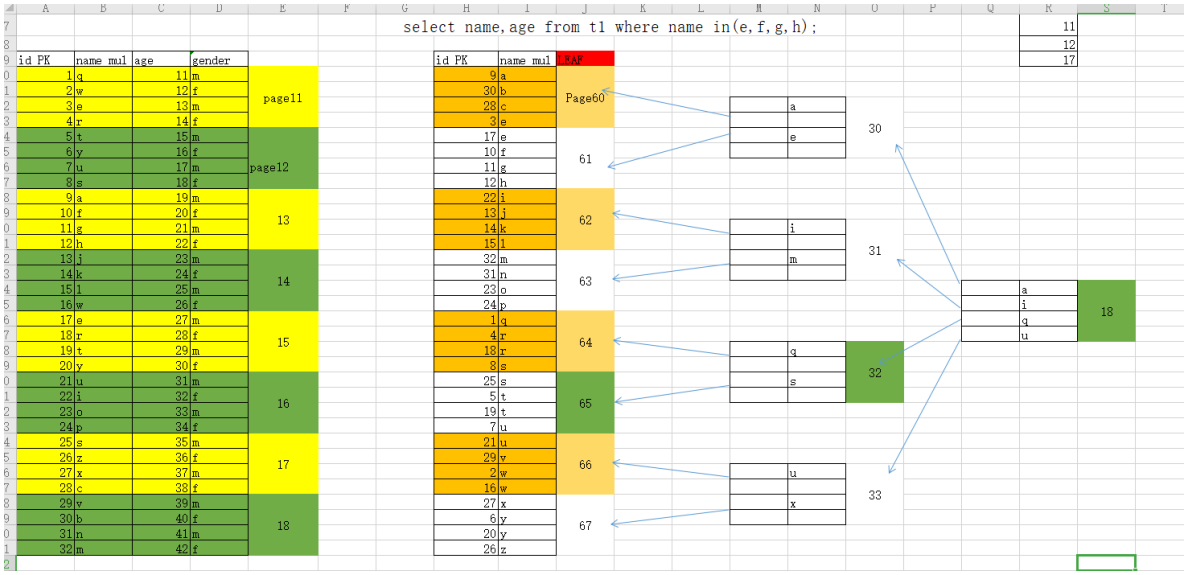
root: 下层节点的name范围+指针

内部节点：也可以理解为枝节点,下层节点的name范围+指针

查询过程：

索引覆盖 : `select id,name from t1 where name='x';`

回表查询 : `select * from t1 where name='x';`



优化器的算法

```
ICP   MRR  SNLJ  BNLJ  BKA   HASH JOIN（8.0.18）
```

9.5.3 表空间

```
system tablespace
file per table tablespace
undo tablespace
temp tablespace
```

System Tablespace(共享表空间)

```
## 存储方式
ibdata1~ibdataN, 5.5版本默认的表空间类型。

##  ibdata1共享表空间在各个版本的变化
5.5版本：
    系统相关：（全局）数据字典信息（表基本结构信息、状态、系统参数、属性..）、UNDO回滚信息(记录撤销操作)、Double Write buffer信息、临时表信息、change buffer
    用户数据：  表数据行、表的索引数据

5.6版本：共享表空间只存储于系统数据，把用户数据独立了。
    系统相关：（全局）数据字典信息、UNDO回滚信息、Double Write信息、临时表信息、change buffer

5.7版本：在5.6基础上，把临时表独立出来，UNDO也可以设定为独立
    系统相关：（全局）数据字典信息、UNDO回滚信息、Double Write信息、change buffer

8.0.19版本：在5.7的基础上将UNDO回滚信息默认独立，数据字典不再集中存储了。
        系统相关：Double Write信息、change buffer

8.0.20版本:在之前版本基础上，独立 Double Write信息
系统相关：change buffer

https://dev.mysql.com/doc/refman/5.7/en/innodb-architecture.html
https://dev.mysql.com/doc/refman/8.0/en/innodb-architecture.html
```

总结： 对于InnoDB(8.0)表来讲，例如 city表
city.ibd
mysql.ibd
undo
ibdata
redo

共享表空间管理
扩容共享表空间
```
mysql> select @@innodb_data_file_path;
+-------------------------+
| @@innodb_data_file_path |
+-------------------------+
| ibdata1:12M:autoextend  |
+-------------------------+
1 row in set (0.00 sec)

mysql> select @@innodb_autoextend_increment;
+--------------------------------+
| @@innodb_autoextend_increment |
+--------------------------------+
|                             64 |
+--------------------------------+
1 row in set (0.00 sec)
```

参数用途：ibdata1文件，默认初始大小12M，不够用会自动扩展，默认每次扩展64M

设置方式：
错误的方法：
```
innodb_data_file_path=ibdata1:12M;ibdata2:100M;ibdata3:100M:autoextend
```

重启数据库报错，查看日志文件
```
vim /data/3306/data/db01.err
###################
[ERROR] InnoDB: The innodb_system data file './ibdata1' is of a different size
4864 pages (rounded down to MB) than the 768 pages specified in the .cnf file!
###################
```

实际大小：
4864*16K/1024=76M
my.cnf文件设置：
768*16K/1024=12M

正确的方法：
先查看实际大小：
```
[root@db01 data]# ls -lh ibdata1
-rw-r----- 1 mysql mysql 76M May  6 17:11 ibdata1
```

配置文件设定为和实际大小一致：
```
innodb_data_file_path=ibdata1:76M;ibdata2:100M;ibdata3:100M:autoextend
```

模拟在初始化时设置共享表空间（生产建议）
5.7 中建议：设置共享表空间2-3个，大小建议512M或者1G，最后一个定制为自动扩展。
8.0 中建议：设置1-2个就ok，大小建议512M或者1G

清理数据
```
[root@db01 data]# /etc/init.d/mysqld stop
```

```
[root@db01 data]# rm -rf /data/3306/data/*

[root@db01 data]# vim /etc/my.cnf
# 修改
innodb_data_file_path=ibdata1:100M;ibdata2:100M;ibdata3:100M:autoextend

# 重新初始化
[root@db01 data]# mysqld --initialize-insecure --user=mysql --
basedir=/data/app/mysql --datadir=/data/3306/data

# 重启数据库生效
[root@db01 data]# /etc/init.d/mysqld start
```

File-Per-Table Tablespaces

```
# 介绍
5.6版本中，针对用户数据，单独的存储管理。存储表的数据行和索引。
通过参数控制：
mysql> select @@innodb_file_per_table;
+-------------------------+
| @@innodb_file_per_table |
+-------------------------+
|                       1 |
+-------------------------+

测试： 共享表空间存储用户数据
mysql> set global innodb_file_per_table=0;

# 利用独立表空间进行快速数据迁移

源端:3306/test/t100w  -----> 目标端：3307/test/t100w

1. 锁定源端t100w表
mysql> flush tables test.t100w with read lock;
mysql> show create table test.t100w;
CREATE TABLE `t100w` (
  `id` int(11) DEFAULT NULL,
  `num` int(11) DEFAULT NULL,
  `k1` char(2) DEFAULT NULL,
  `k2` char(4) DEFAULT NULL,
  `dt` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;


2. 目标端创建test库和t100w空表
mysql> create database test charset=utf8mb4;
CREATE TABLE `t100w` (
  `id` int(11) DEFAULT NULL,
  `num` int(11) DEFAULT NULL,
  `k1` char(2) DEFAULT NULL,
  `k2` char(4) DEFAULT NULL,
  `dt` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

3.  单独删除空的表空间文件
mysql> alter table test.t100w discard tablespace;
```

```
4. 拷贝源端ibd文件到目标端目录，并设置权限
[root@db01 test]# cp /data/3306/data/test/t100w.ibd /data/3307/data/test/
[root@db01 test]# chown -R mysql.mysql /data/*

5. 导入表空间
mysql> alter table test.t100w import tablespace;
mysql> select count(*) from test.t100w;
+----------+
| count(*) |
+----------+
|  1000000 |

6. 解锁源端数据表
mysql> unlock tables;


作业： 将3306实例的test.t100w  ---> 3307实例test.t100w


真是的生产故障场景：
fsck
我们也在用 confluence jira

C   丢失了
J   113张 frm  ibd


python ./setup.py build
python ./setup.py install
mysqlfrm -diagnostic /usr/local/mysql/data/test/new_table.frm



案例1：同学由于不可抗力因素，导致只剩下test库下的ibd 和 frm文件了。（5.6版本）。没有备份

案例2：同学将ibdata1（5.7版本）误rm掉了。导致只剩下test库下的ibd 和 frm文件了。备份坏的。

test业务库： 200多张表。

痛点： 1. 表太多   2. 建表语句



在8.0没有frm文件，建表语句怎么查呢？
[root@db world]# ibd2sdi city.ibd
```

Undo tablespace

```
说明： 撤销日志，回滚日志。

1. 作用： 用来作撤销工作。
```

2. 存储位置： 5.7版本，默认存储在共享表空间中（ibdataN）。8.0版本以后默认就是独立的（undo_001-undo_002）。

3. 生产建议： 5.7版本后，将undo手工进行独立。

4. undo 表空间管理
4.1 如何查看undo的配置参数
```
SELECT @@innodb_undo_tablespaces;  ---->3-5个     #打开独立undo模式，并设置undo的个数。
SELECT @@innodb_max_undo_log_size;                #undo日志的大小，默认1G。
SELECT @@innodb_undo_log_truncate;                #开启undo自动回收的机制（undo_purge）。
SELECT @@innodb_purge_rseg_truncate_frequency;    #触发自动回收的条件，单位是检测次数。
```

4.2 配置undo表空间
```
#########官方文档说明#############
Important
The number of undo tablespaces can only be configured
when initializing a MySQL instance and is fixed for the life of the instance.
###################################

[root@db01 tmp]# /etc/init.d/mysqld stop
[root@db01 tmp]# rm -rf /data/3306/data/*

vim /etc/my.cnf
# 添加参数
innodb_undo_tablespaces=3
innodb_max_undo_log_size=128M
innodb_undo_log_truncate=ON
innodb_purge_rseg_truncate_frequency=32

# 重新初始化数据库生效
[root@db01 data]# mysqld --initialize-insecure --user=mysql --
basedir=/data/app/mysql --datadir=/data/3306/data

# 启动数据库
[root@db01 data]# /etc/init.d/mysqld start
[root@db01 data]# ll /data/3306/data/undo00*
-rw-r----- 1 mysql mysql 10485760 May  7 15:39 /data/3306/data/undo001
-rw-r----- 1 mysql mysql 10485760 May  7 15:39 /data/3306/data/undo002
-rw-r----- 1 mysql mysql 10485760 May  7 15:39 /data/3306/data/undo003

# 注： 8.0 undo表空间与5.7稍有区别，可参考：
https://dev.mysql.com/doc/refman/8.0/en/innodb-undo-tablespaces.html
1. 添加UNDO
CREATE UNDO TABLESPACE oldguo ADD DATAFILE 'oldguo.ibu';
2. 查看
SELECT TABLESPACE_NAME, FILE_NAME FROM INFORMATION_SCHEMA.FILES    WHERE
FILE_TYPE LIKE 'UNDO LOG';

3. 删除undo
ALTER UNDO TABLESPACE  oldguo  SET INACTIVE;
DROP UNDO TABLESPACE oldguo;
SELECT TABLESPACE_NAME, FILE_NAME FROM INFORMATION_SCHEMA.FILES    WHERE
FILE_TYPE LIKE 'UNDO LOG';

说明： 关于UNDO回收策略
SELECT @@innodb_purge_rseg_truncate_frequency;
```

```
select * from INNODB_TABLESPACES where ROW_FORMAT='UNDO'\G
```

tmp tablespace

```
1. 作用： 存储临时表。
2. 管理：
innodb_temp_data_file_path=ibtmp1:128M;ibtmp2:128M:autoextend:max:500M
重启生效。
3. 建议数据初始化之前设定好，一般2-3个，大小512M-1G。
```

9.5.4 段

```
表段（分区表除外）
undo段
```

9.5.5 区extent（簇）

```
一个区默认64个连续数据页。默认值是1M空间。
区也可以被称之为“簇”。也是聚簇索引中的分配单元，通常也是read-ahead的单元。
```

9.5.6 页 page

```
默认16KB。
```

通用结构

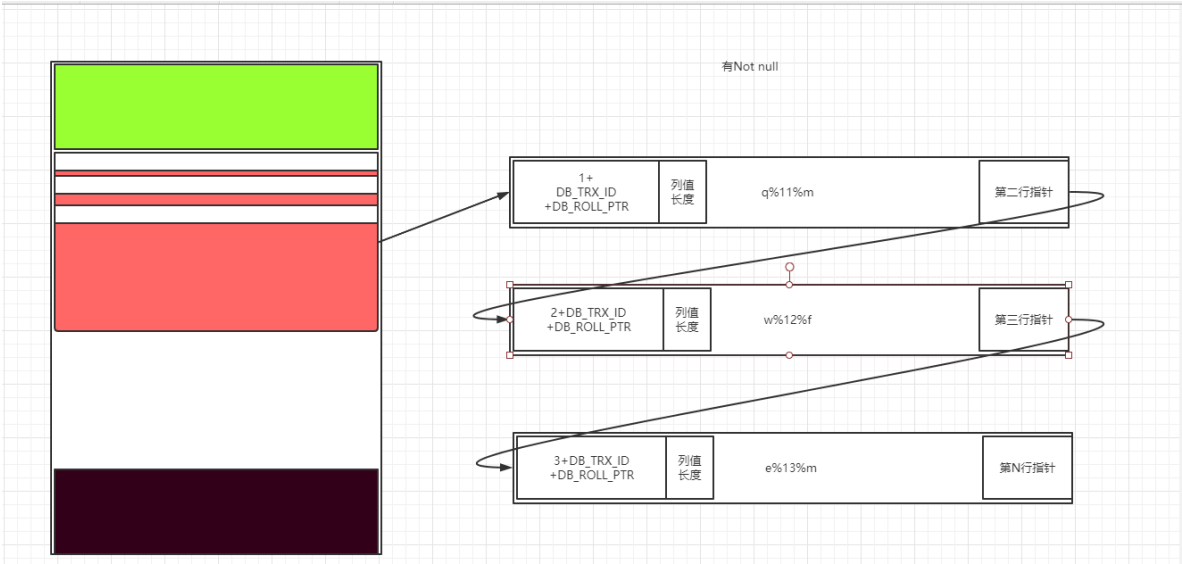
```
Fil Header
Page Header
The Infimum and Supremum Records

Page Directory
Fil Trailer
```

> 如何计算索引树的高度?

数据部分

```
User Records
Free Space
```

9.5.7 行格式



MySQL目前有4种行格式：Redundant、Compact、Dynamic、Compressed

Version 5.6 已经默认使用 Compact
Version 5.7+ 默认使用Dynamic

结构说明：

1、存储单元为页（page），16KB（16384B）
2、每页至少需要存两条数据
3、每条记录都会有记录头
4、每条记录3个隐藏列（rowId，transactionId，rollPointer）
5、记录按照聚簇索引组织存储

Compact：
变长字段(记录的长度)列表 + NULL列表 + 记录头信息 + 列值
变长字段（记录的长度）列表：采用1-2个字节来表示一个字段的长度
记录头信息：使用5个字节来表示，主要包含：该记录是否被删除，记录类型，下一条记录的相对偏移量；
隐藏列： rowId，transactionId，rollPointer

Dynamic：
与Compact行格式很像，差异在于页面溢出的处理上；

Compressed：
在于Dynamic使用了压缩算法；

页溢出：
因为每页16KB，至少存储两行，所以每行大概有8KB的数据；抛开记录头信息等，大致每列超过768B就会产生页溢出；

Compact：
1、会将溢出的数据单独放入一个页；外加20B存储额外页的信息（plus the 20-byte pointer to the externally stored part）
2、索引可以使用前768B

768B ---> utf8 varchar(255)

Dynamic：
1、如果页溢出，则使用20B存储整个列信息(列数据都存储在溢出页上)（with the clustered index record containing only a 20-byte pointer to the overflow page）
2、可以使用前3072B字符的索引（--innodb-large-prefix决定）

```
3072B  ---> utf8   varchar(1022)
3072B  ---> utf8mb4   varchar(766)


建议：
    5.7+ 版本Dynamic。
    建议，大字段不要存储到MySQL 核心业务表中。
    或者非得用，建议将大列做hash值运算，然后单独存储一列，每次查询按照hash值列进行查询。

结论：
    Compact：768字节
    Dynamic：3072字节
    建立索引时：列值长度不能超过以上字节数。
```

page dir slot介绍

```
页目录。存储着2-N个SLOT（槽），2字节
作用：  从数据页中，快速的找到想要的record（记录） 。
slot 分为了三种：
    最小 ：  1个最小记录
    最大 ：  1-8个记录
    普通 ：  4-8个记录
```

9.5.4 DWB（double write bufffer)

```
作用：
MySQL，最小IO单元page（16KB），OS中最小的IO单元是block（4KB）
为了防止出现以下问题：
mysqld process crash in the middle of a page write
DWB每次1M，2次写完。数据页再刷盘。

补充：
DWB 对于写入性能确实有影响。

UPS + 高可用架构
```

9.5.5 REDO日志

```
- what?
• DML(8.0+ DDL)操作导致的页面变化，均需要记录Redo日志；

• 大部分为物理日志；

- when?
• 在页面修改完成之后，在脏页刷出磁盘之前，写入Redo日志；
• 日志先行，日志一定比数据页先写回磁盘（WAL）；

• 聚簇索引/二级索引/Undo页面修改，均需要记录Redo日志；
```

```
1. 作用: 记录数据页的变化。实现“前进”的功能。WAL（write ahead log），MySQL保证redo优先于数据写入磁盘。
```

```
2. 存储位置： 数据路径下，进行轮序覆盖记录日志
ib_logfile0  48M
ib_logfile1  48M
3. 管理：
3.1 查询redo log文件配置
mysql> show variables like '%innodb_log_file%';
+---------------------------+----------+
| Variable_name             | Value    |
+---------------------------+----------+
| innodb_log_file_size      | 50331648 |
| innodb_log_files_in_group | 2        |
+---------------------------+----------+
3.2 设置
生产建议：
    大小： 512M-4G
    组数： 2-4组
vim  /etc/my.cnf
# 添加参数：
innodb_log_file_size=100M
innodb_log_files_in_group=3

#重启生效
[root@db01 data]# /etc/init.d/mysqld restart
[root@db01 data]# ll /data/3306/data/ib_logfile*
-rw-r----- 1 mysql mysql 104857600 May  7 16:17 /data/3306/data/ib_logfile0
-rw-r----- 1 mysql mysql 104857600 May  7 16:17 /data/3306/data/ib_logfile1
-rw-r----- 1 mysql mysql 104857600 May  7 16:17 /data/3306/data/ib_logfile2
[root@db01 data]#
```

9.5.6 UNDO日志

```
- what?
• DML(8.0+ DDL)操作导致的数据记录变化，均需要将记录的前镜像写入Undo日志；
• 逻辑日志；
- when?
• DML(8.0 + DDL) 操作修改聚簇索引前，记录Undo日志(Undo日志，先于Redo日志。)；

• 二级索引记录的修改，不记录Undo日志；

• 注意：Undo页面的修改，同样需要记录Redo日志；
```

9.5.7 磁盘结构与内存结构工作关系

LSN（日志序列号）

```
定义说明：
    LSN（log sequence number）日志序列号，5.6.3之后占用8字节，LSN主要用于发生crash时对数据进行recovery，LSN是一个一直递增的整型数字，表示事务写入到日志的字节总量。
    LSN不仅只存在于重做日志中，在每个数据页头部也会有对应的LSN号，该LSN记录当前页最后一次修改的LSN号，用于在recovery时对比重做日志LSN号决定是否对该页进行恢复数据。前面说的checkpoint也是有LSN号记录的，LSN号串联起一个事务开始到恢复的过程。

查看lsn:
```

```
   show engine innodb status\G
   Log sequence number 2687274848548
   Log flushed up to 2687274848516
   Pages flushed up to 2687273963960
   Last checkpoint at 2687273963960
```

简单说明:

```
   Log sequence number: 当前系统最大的LSN号
   log flushed up to:当前已经写入redo日志文件的LSN
   pages flushed up to：已经将更改写入脏页的lsn号
   Last checkpoint at就是系统最后一次刷新buffer pool脏中页数据到磁盘的checkpoint
```

以上4个LSN是递减的:

```
    LSN1>=LSN2>=LSN3>=LSN4.
```

内容:

```
    每个数据页有LSN，重做日志有LSN，checkpoint有LSN。
```

checkpoint

sharp checkpoint：
完全检查点，数据库正常干净关闭时，会触发把所有的脏页都写入到磁盘上(这时候logfile的日志就没用了，脏页已经写到磁盘上了)。

fuzzy checkpoint：
模糊检查点，部分页写入磁盘。

1. master thread checkpoint
差不多以每秒或每十秒的速度从缓冲池的脏页列表中刷新一定比例的页回磁盘，这个过程是异步的，不会阻塞用户查询。

```
show variables like '%io_cap%';
+------------------------+-------+
| Variable_name          | Value |
+------------------------+-------+
| innodb_io_capacity     | 200   |
| innodb_io_capacity_max | 2000  |

PCI-E  2000-3000  4000-6000
flash  5000-8000  10000-16000

2、flush_lru_list checkpoint
mysql> show variables like '%lru%depth';
+----------------------+-------+
| Variable_name         | Value |
+----------------------+-------+
| innodb_lru_scan_depth | 1024  |
+----------------------+-------+
1 row in set (0.01 sec)

参数innodb_lru_scan_depth控制lru列表中可用页的数量，默认是1024。

3、 async/sync flush checkpoint
log file快满了，会批量的触发数据页回写，这个事件触发的时候又分为异步和同步，不可被覆盖的
redolog占log file的比值：75%--->异步、90%--->同步。
```

指的是重做日志文件不可用的情况，这时需要强制将一些页刷新回磁盘，而此时脏页是从脏页列表中选取的。若将已经写入到重做日志的LSN记为redo_lsn，将已经刷新回磁盘最新页的LSN记为checkpoint_lsn，则可定义：

```
checkpoint_age = redo_lsn - checkpoint_lsn
```

再定义以下的变量：

```
async_water_mark = 75% * total_redo_log_file_size
sync_water_mark = 90% * total_redo_log_file_size
```

若每个重做日志文件的大小为1GB，并且定义了两个重做日志文件，则重做日志文件的总大小为2GB。那么async_water_mark=1.5GB，sync_water_mark=1.8GB。则：
当checkpoint_age<async_water_mark时，不需要刷新任何脏页到磁盘；
当async_water_mark<checkpoint_age<sync_water_mark时触发Async Flush，从Flush列表中刷新足够的脏页回磁盘，使得刷新后满足checkpoint_age<async_water_mark；
checkpoint_age>sync_water_mark这种情况一般很少发生，除非设置的重做日志文件太小，并且在进行类似LOAD DATA的BULK INSERT操作。此时触发Sync Flush操作，从Flush列表中刷新足够的脏页回磁盘，使得刷新后满足checkpoint_age<async_water_mark。

4、dirty page too much checkpoint
脏页太多检查点，为了保证buffer pool的空间可用性的一个检查点。

```
mysql> show global status like 'Innodb_buffer_pool_pages%t%';
+---------------------------------+-------+
| Variable_name                   | Value |
+---------------------------------+-------+
| Innodb_buffer_pool_pages_data   | 2964  |
| Innodb_buffer_pool_pages_dirty | 0     |
| Innodb_buffer_pool_pages_total | 8191  |
+---------------------------------+-------+
3 rows in set (0.00 sec)

mysql> show global status like '%wait_free';
+------------------------------+-------+
| Variable_name                | Value |
+------------------------------+-------+
| Innodb_buffer_pool_wait_free | 0     |
+------------------------------+-------+
1 row in set (0.00 sec)
```

1、Innodb_buffer_pool_pages_dirty/Innodb_buffer_pool_pages_total：表示脏页在buffer的占比
2、Innodb_buffer_pool_wait_free：如果>0，说明出现性能负载，buffer pool中没有干净可用块

2、脏页控制参数

```
mysql> show variables like '%dirty%pct%';
+---------------------------------+-----------+
| Variable_name                   | Value     |
+---------------------------------+-----------+
| innodb_max_dirty_pages_pct      | 75.000000 |
| innodb_max_dirty_pages_pct_lwm | 0.000000  |
+---------------------------------+-----------+
2 rows in set (0.01 sec)
```

1、默认是脏页占比75%的时候，就会触发刷盘，将脏页写入磁盘，腾出内存空间。建议不调，调太低的话，io压力就会很大，但是崩溃恢复就很快；
2、lwm：low water mark低水位线，刷盘到该低水位线就不写脏页了，0也就是不限制。

CR(Crash Recovery)

```
redo : 重做日志  innodb log buffer , ib_logfileN
undo : 回滚日志
LSN  :  LOG  Seq  NO
page : 数据页，page_no , Page_lsn(ckpt)
行记录：头部。DB_TRX_ID  , DB_Roll_PTR
IBP:  InnoDB Buffer Pool ,缓冲数据页。
DP: dirty page ,内存中发生变化的数据页，没有被写到磁盘的那些。
```

```
redo前滚 构造脏页
undo 回滚未提交事务
```

9.5.8 其他结构-ib_buffer_pool

```
作用
缓冲和缓存，用来做“热”（经常查询或修改）数据页，减少物理IO。
当关闭数据库的时候，缓冲和缓存会失效。
5.7版本中，MySQL正常关闭时，会将内存的热数据存放（流方式）至ib_buffer_pool。下次重启直接读取ib_buffer_pool加载到内存中。
```

9.6 InnoDB 事务详解 *

9.6.1 介绍

```
事务：Transaction （交易）。 伴随着交易类的业务出现的概念（工作模式）
交易？
    物换物，等价交换。
    货币换物，等价交换。
    虚拟货币换物（虚拟物品），等价交换。

现实生活中怎么保证交易“和谐” ，法律、道德等规则约束。
数据库中为了保证线上交易的“和谐”，加入了“事务”工作机制。
```

9.6.2 事务控制语句

```
 标准(显示)的事务控制语句
# 开启事务
begin;  start  transaction;
# 提交事务
commit;
# 回滚事务
rollback;

注意：
事务生命周期中，只能使用DML语句（select、update、delete、insert）

# 事务的生命周期演示：
mysql> use world
```

```
mysql> begin;

mysql> delete from city where id=1;
mysql> update city set countrycode='CHN' where id=2;


mysql> commit;
```

隐式提交：
begin
a
b

begin

SET AUTOCOMMIT = 1

导致提交的非事务语句：
DDL语句： （ALTER、CREATE 和 DROP）
DCL语句： （GRANT、REVOKE 和 SET PASSWORD）
锁定语句：（LOCK TABLES 和 UNLOCK TABLES）
导致隐式提交的语句示例：
TRUNCATE TABLE
LOAD DATA INFILE
SELECT FOR UPDATE

隐式回滚
会话窗口被关闭。
数据库关闭 。
出现事务冲突（死锁）。

9.6.3 事务的ACID

A: 原子性
不可再分性：一个事务生命周期中的DML语句，要么全成功要么全失败，不可以出现中间状态。
主要通过：undo保证的。

C：一致性
事务发生前，中，后，数据都最终保持一致。
CR + DWB

I：隔离性
事务操作数据行的时候，不会受到其他事务的影响。
读写隔离： 隔离级别、MVCC
写写隔离： 锁、隔离级别

D: 持久性
一但事务提交，永久生效（落盘）。
redo保证 ckpt。

9.6.4 隔离级别

作用
实现事务工作期间的“读”的隔离
读？ ----》 数据页（记录）的读

```
6.4.2 级别类型
mysql> select @@transaction_isolation;


# RU ： READ-UNCOMMITTED 读未提交
可以读取到事务未提交的数据。隔离性差，会出现脏读（当前内存读），不可重复读，幻读问题


# RC ： READ-COMMITTED   读已提交（可以用）：
可以读取到事务已提交的数据。隔离性一般，不会出现脏读问题，但是会出现不可重复读，幻读问题

# RR ： REPEATABLE-READ  可重复读（默认）  ：
防止脏读（当前内存读），不可重复读，幻读问题。需要配合锁机制来避免幻读。

# SE ： SERIALIZABLE     可串行化
结论： 隔离性越高，事务的并发读就越差。


结论： 一般情况下RC就够用了。
select  for  update

start transaction with consistance snapshot
```

9.7 InnoDB 锁机制

9.7.1 锁的介绍

```
锁机制：innodb中主要是写的隔离
    作用：保护并发访问资源。
```

9.7.2 锁的类型（大面）

```
保护的资源分类：

内存资源层次
    latch（闩锁）：rwlock、mutex，主要保护内存资源
    buffer pool    log buffer

Server层：
    MDL： Metadata_lock,元数据(DDL操作)
    lock table t1 read ;
    mysqldump、XBK（PBK）：备份非InnoDB数据时，触发FTWRL全局锁表（Global）。

engine层：
    row lock：InnoDB 默认锁粒度，加锁方式都是在索引加锁的。
            record lock     ：  记录锁，在索引锁定。RC级别record lock。
            gap lock        ：  间隙锁，在索引间隙加锁。RR级别存在。防止幻读。
            next key lock   ：  下一键锁， GAP+Record。    RR级别存在。防止幻读。
# 功能性上：
    IS   ：  select * from t1 lock in shared mode;
```

```
S    :  读锁。
IX   :  意向排他锁。表上添加的。 select * from t1 for update;
X    :  排他锁，写锁。
```

附录： 锁兼容性表

	X	IX	S	IS
X	Conflict	Conflict	Conflict	Conflict
IX	Conflict	Compatible	Conflict	Compatible
S	Conflict	Conflict	Compatible	Compatible
IS	Conflict	Compatible	Compatible	Compatible

附录：MDL细分

属性	含义	范围/对象
GLOBAL	全局锁	范围
COMMIT	提交保护锁	范围
SCHEMA	库锁	对象
TABLE	表锁	对象
FUNCTION	函数锁	对象
PROCEDURE	存储过程锁	对象
TRIGGER	触发器锁	对象
EVENT	事件锁	对象

MDL按锁住的对象来分类，可以分为global，commit，schema，table，function，procedure，trigger，event，这些对象发生锁等待时，我们在show full processlist可以分别看到如下等待信息。

```
        Waiting for global read lock            ---->  flush table with read
lock
        Waiting  for commit lock
        Waiting  for schema metadata lock
        Waiting for table metadata lock         ---->  DDL
        Waiting  for stored function metadata  lock
        Waiting  for stored procedure metadata lock
        Waiting for trigger  metadata lock
        Waiting  for event  metadata lock
```

按照锁的持有时间分类

属性	含义
MDL_STATEMENT	从语句开始执行时获取，到语句执行结束时释放。
MDL_TRANSACTION	在一个事务中涉及所有表获取MDL，一直到事务commit或者rollback(线程中终清理)才释放。
MDL_EXPLICIT	需要MDL_context::release_lock()显式释放。语句或者事务结束,也仍然持有，如Lock table, flush .. with lock语句等。

按照操作的类型和对象分类

属性	含义	示例
MDL_INTENTION_EXCLUSIVE(IX)	意向排他锁用于global和commit的加锁。	truncate table t1; insert into t1 values(3,'oldguo');会加如下锁 (GLOBAL,MDL_STATEMENT,MDL_INTENTION_EXCLUSIVE) (SCHEMA,MDL_TRANSACTION,MDL_INTENTION_EXCLUSIVE)
MDL_SHARED(S)	只访问元数据 比如表结构，不访问数据。	set golbal_read_only =on 加锁 (GLOBAL，MDL_EXPLICIT，MDL_SHARED)
MDL_SHARED_HIGH_PRIO(SH)	用于访问information_schema表，不涉及数据。	select * from information_schema.tables; show create table xx; desc xxx; 会加如下锁： (TABLE,MDL_TRANSACTION,MDL_SHARED_HIGH_PRIO)
MDL_SHARED_READ(SR)	访问表结构并且读表数据	select * from t1; lock table t1 read; 会加如下锁: (TABLE，MDL_TRANSACTION，MDL_SHARE_READ)
MDL_SHARED_WRITE(SW)	访问表结构并且写表数据	insert/update/delete/select .. for update 会加如下锁： (TABLE，MDL_TRANSACTION，MDL_SHARE_WRITE)
MDL_SHARED_UPGRADABLE(SU)	是mysql5.6引入的新的metadata lock, 在alter table/create index/drop index会加该锁;为了online ddl才引入的。特点是允许DML，防止DDL；	（TABLE，MDL_TRANSACTION， MDL_SHARED_UPGRADABLE)
MDL_SHARED_NO_WRITE(SNW)	可升级锁，访问表结构并且读写表数据，并且禁止其它事务写。	alter table t1 modify c bigint; (非onlineddl) (TABLE，MDL_TRANSACTION，MDL_SHARED_NO_WRITE)
MDL_SHARED_NO_READ_WRITE(SNRW)	可升级锁，访问表结构并且读写表数据，并且禁止其它事务读写。	lock table t1 write; 加锁 (TABLE,MDL_TRANSACTION,MDL_SHARED_NO_READ_WRITE
MDL_EXCLUSIVE(X)	防止其他线程读写元数据	CREATE/DROP/RENAME TABLE，online DDL在rename阶段也持有X锁 (TABLE，MDL_TRANSACTION，MDL_EXCLUSIVE)

几种典型语句的加(释放)锁流程

> select语句操作MDL锁流程:
>
> 1) Opening tables阶段，加共享锁
> a) 加MDL_INTENTION_EXCLUSIVE锁
> b) 加MDL_SHARED_READ锁
>
> 2) 事务提交阶段，释放MDL锁
> a) 释放MDL_INTENTION_EXCLUSIVE锁
> b) 释放MDL_SHARED_READ锁

> DML语句操作MDL锁流程
>
> 1) Opening tables阶段，加共享锁
> a) 加MDL_INTENTION_EXCLUSIVE锁
> b) 加MDL_SHARED_WRITE锁
>
> /////engine
>
> 2) 事务提交阶段，释放MDL锁
> a) 释放MDL_INTENTION_EXCLUSIVE锁
> b) 释放MDL_SHARED_WRITE锁

> alter操作MDL锁流程(copy方式)
> 1) Opening tables阶段，加共享锁
> a) 加MDL_INTENTION_EXCLUSIVE锁
> b) 加MDL_SHARED_UPGRADABLE锁，升级到MDL_SHARED_NO_WRITE锁
>
> 2) 操作数据，copy data，流程如下：
> a) 创建临时表tmp，重定义tmp为修改后的表结构
> b) 从原表读取数据插入到tmp表
>
> 3) 将MDL_SHARED_NO_WRITE读锁升级到MDL_EXCLUSIVE锁
> a) 删除原表，将tmp重命名为原表名
>
> 4) 提交阶段，释放MDL锁
> a) 释放MDL_INTENTION_EXCLUSIVE锁
> b) 释放MDL_EXCLUSIVE锁

> Online DDL-- inplace
>
> 第一阶段： Prepare阶段
> 创建新的临时frm文件(与InnoDB无关)
> 持有EXCLUSIVE-MDL锁，禁止读写
> 根据alter类型，确定执行方式(copy,online-rebuild,online-norebuild)
> 假如是Add Index，则选择online-norebuild即INPLACE方式
> 更新数据字典的内存对象
> 分配row_log对象记录增量(仅rebuild类型需要)
> 生成新的临时ibd文件(仅rebuild类型需要)
>
> 第二阶段： ddl执行阶段
> 降级EXCLUSIVE-MDL锁，允许读写
> 扫描old_table的聚集索引每一条记录rec
> 遍历新表的聚集索引和二级索引，逐一处理
> 根据rec构造对应的索引项
> 将构造索引项插入sort_buffer块排序
> 将sort_buffer块更新到新的索引上
> 记录ddl执行过程中产生的增量(仅rebuild类型需要)
> 重放row_log中的操作到新索引上(no-rebuild数据是在原表上更新的)
> 重放row_log间产生dml操作append到row_log最后一个Block
>
> 第三阶段： commit阶段
> 当前Block为row_log最后一个时，禁止读写，升级到EXCLUSIVE-MDL锁
> 重做row_log中最后一部分增量
> 更新innodb的数据字典表
> 提交事务(刷事务的redo日志)
> 修改统计信息
> rename临时idb文件，frm文件
> 变更完成

问题： select 会不会阻塞其他操作?

> mysql> select * from performance_schema.metadata_locks\
>
> show processlist

> select * from performance_schema.threads
>
> select * from performance_schema.events_statements_current;
>
> select * from performance_schema.events_statements_history

InnoDB engine级别锁：

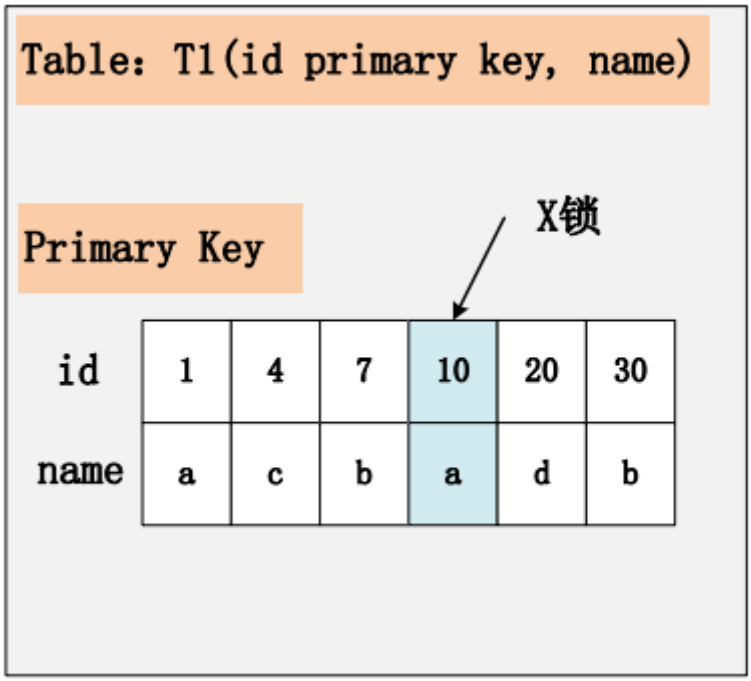
> a. 锁定对象都是基于索引进行“加锁”
>
> b. 分类：
>
> record lock （记录锁）j
>
> GAP (间隙锁)： 两个记录之间的缝隙 （a,j)
>
> NKL（下一键锁）Next key lock ： GAP+record lock 并集 ， (a,j]
>
> c. 不同隔离级别加锁行为
>
> RC：大部分是记录锁。在有外键时可能有所出入。
>
> RR : 加锁的粒度是NKL
>
> d. 在RR级别下的加锁细节
>
> 原则 1：加锁的基本单位是 next-key lock。并且next-key lock 是前开后闭区间。(5,10]
> 原则 2：查找过程中访问到的索引才会加锁。
> 原则 3：索引上的等值查询，给唯一索引加锁的时候，next-key lock 退化为行锁。
>
> 注意的点： 前提是这个数据在表中有
>
> 原则 4：索引上的等值查询，向右遍历时且最后一个值不满足等值条件的时候，next-key lock 退化为间隙锁。
>
> 8019之前bug: 唯一索引上的范围查询会访问到不满足条件的第一个值为止。

9.7.3 不同情况下的锁处理

RC级别：

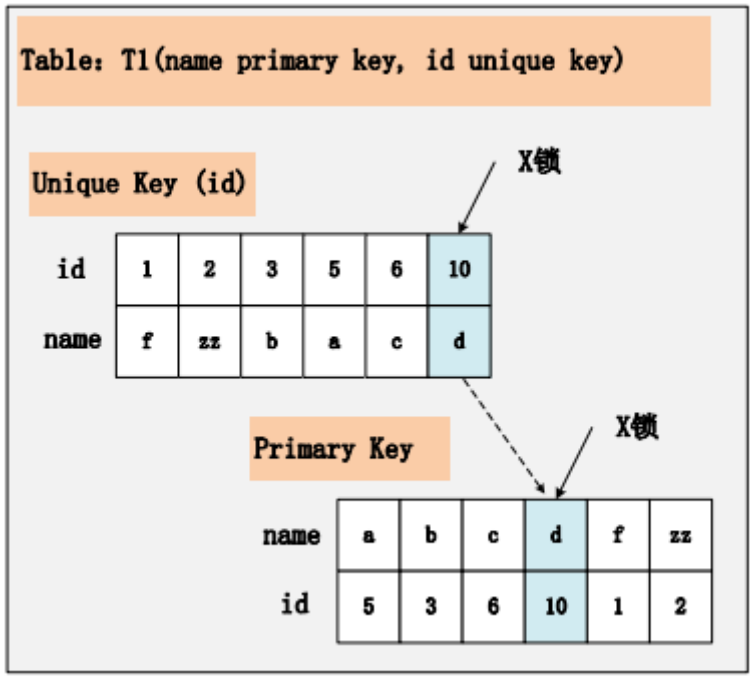
情景一：id 主键+RC

> 这个组合，是最简单，最容易分析的组合。id 是主键，Read Committed 隔离级别，给定 SQL：delete from t1 where id = 10; 只需要将主键上，id = 10 的记录加上 X 锁即可。如下图所示：



情景二：id 唯一索引+RC

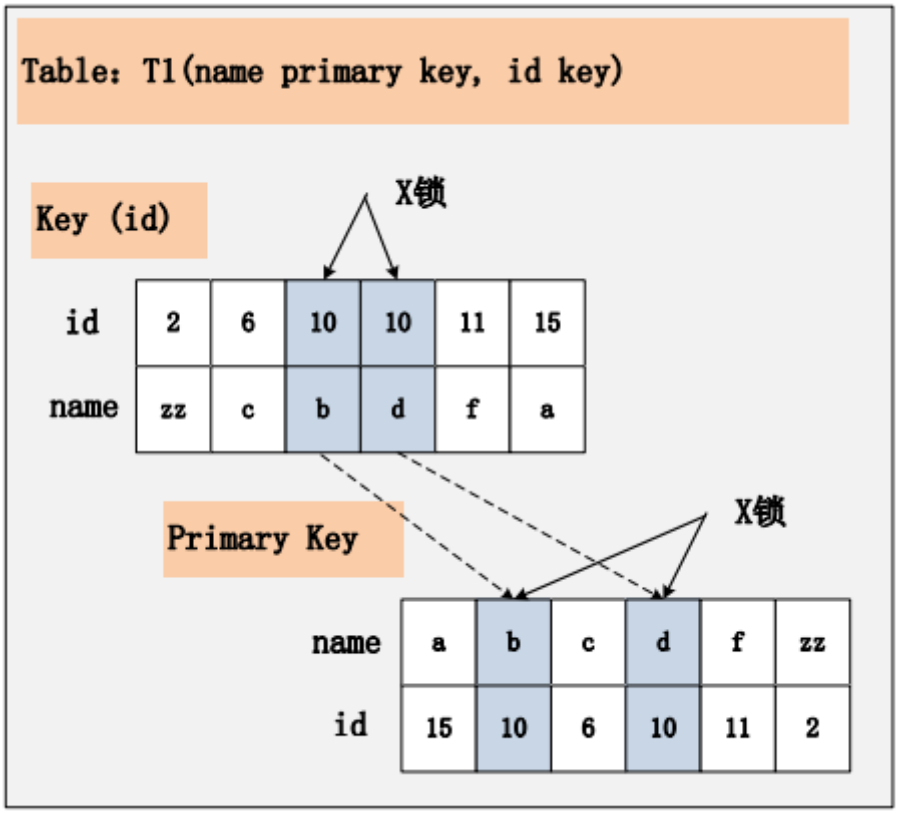
> 这个组合，id 不是主键，而是一个 Unique 的二级索引键值。那么在 RC 隔离级别下，delete from t1 where id = 10; 需要加什么锁呢？见下图：



> 若 id 列是 unique 列，其上有 unique 索引。那么 SQL 需要加两个 X 锁，一个对应于 id unique 索引上的 id = 10 的记录，另一把锁对应于聚簇索引上的[name='d',id=10]的记录。

情景三：id 非唯一索引+RC

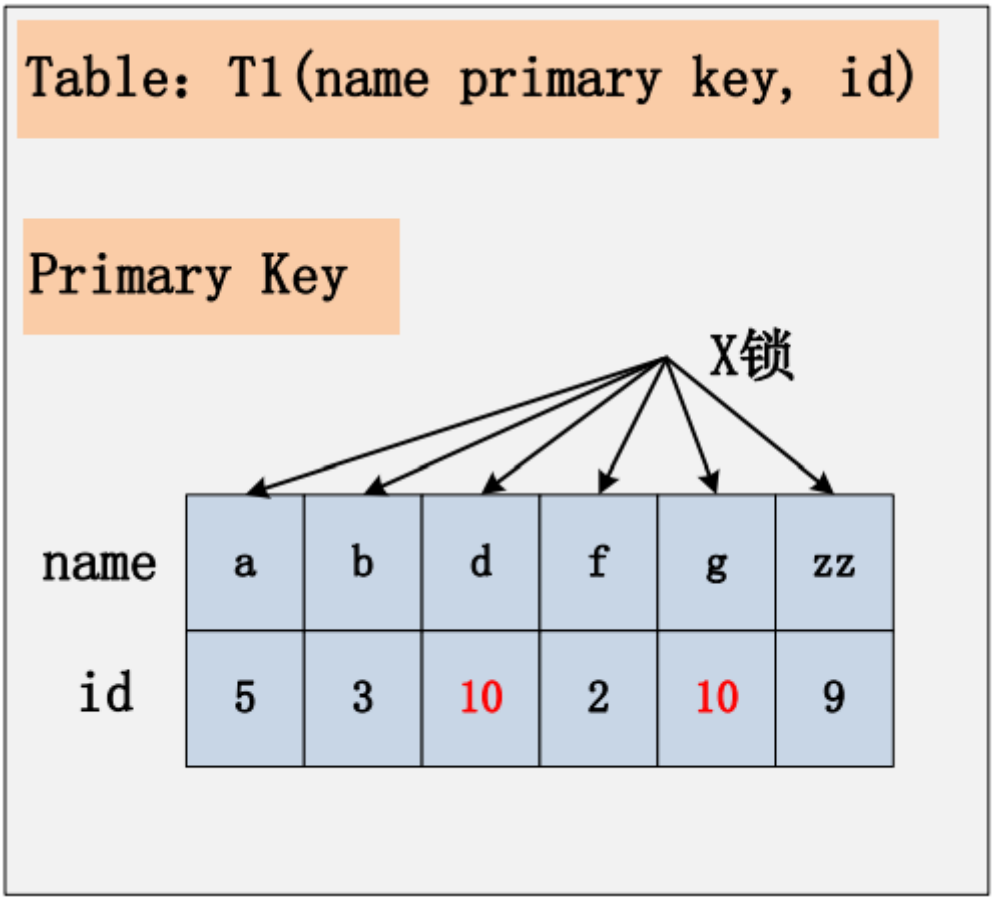
> 相对于组合一、二，组合三又发生了变化，隔离级别仍旧是 RC 不变，但是 id 列上的约束又降低了，id 列不再唯一，只有一个普通的索引。假设 delete from t1 where id = 10; 语句，仍旧选择 id 列上的索引进行过滤 where 条件，那么此时会持有哪些锁？同样见下图：



> 若 id 列上有非唯一索引，那么对应的所有满足 SQL 查询条件的记录，都会被加锁。同时，这些记录在主键索引上的记录，也会被加锁。

情景四：id 无索引+RC

> 相对于前面三个组合，这是一个比较特殊的情况。id 列上没有索引，where id = 10;这个过滤条件，没法通过索引进行过滤，那么只能走全表扫描做过滤。对应于这个组合，SQL 会加什么锁？或者是换句话说，全表扫描时，会加什么锁？这个答案也有很多：有人说会在表上加 X 锁；有人说会将聚簇索引上，选择出来的 id = 10;的记录加上 X 锁。那么实际情况呢？请看下图：



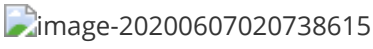
> 若 id 列上没有索引，SQL 会走聚簇索引的全扫描进行过滤，由于过滤是由 MySQL Server 层面进行的。因此每条记录，无论是否满足条件，都会被加上 X 锁。但是，为了效率考量，MySQL 做了优化，对于不满足条件的记录，会在判断后放锁，最终持有的，是满足条件的记录上的锁，但是不满足条件的记录上的加锁/放锁动作不会省略

RR级别加锁原则：

```
原则 1：加锁的基本单位是 next-key lock。并且next-key lock 是前开后闭区间。(5,10]
原则 2：查找过程中访问到的索引才会加锁。
原则 3：索引上的等值查询，给唯一索引加锁的时候，next-key lock 退化为行锁。
原则 4：索引上的等值查询，向右遍历时且最后一个值不满足等值条件的时候，next-key lock 退化为间隙锁。
8019之前bug：唯一索引上的范围查询会访问到不满足条件的第一个值为止。
```

情景五：id 主键+RR

```
id 列是主键列，Repeatable Read 隔离级别，针对 delete from t1 where id = 10; 这条SQL，加锁与组合一：[id 主键，Read Committed]一致。
```



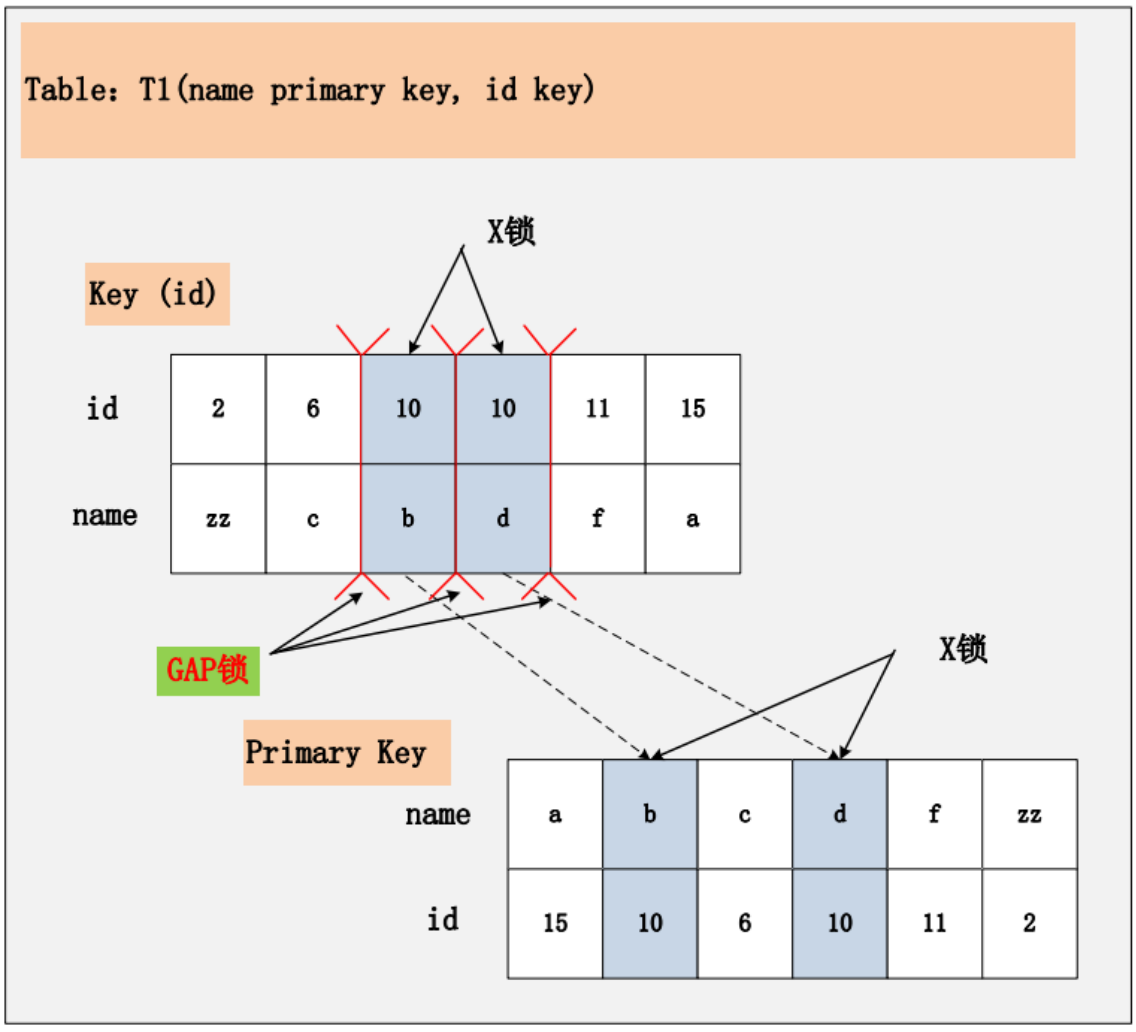
情景六：id 唯一索引+RR

```
delete
from t1 where id = 10;
```

image-20200607020809789

情景七：id 非唯一索引+RR

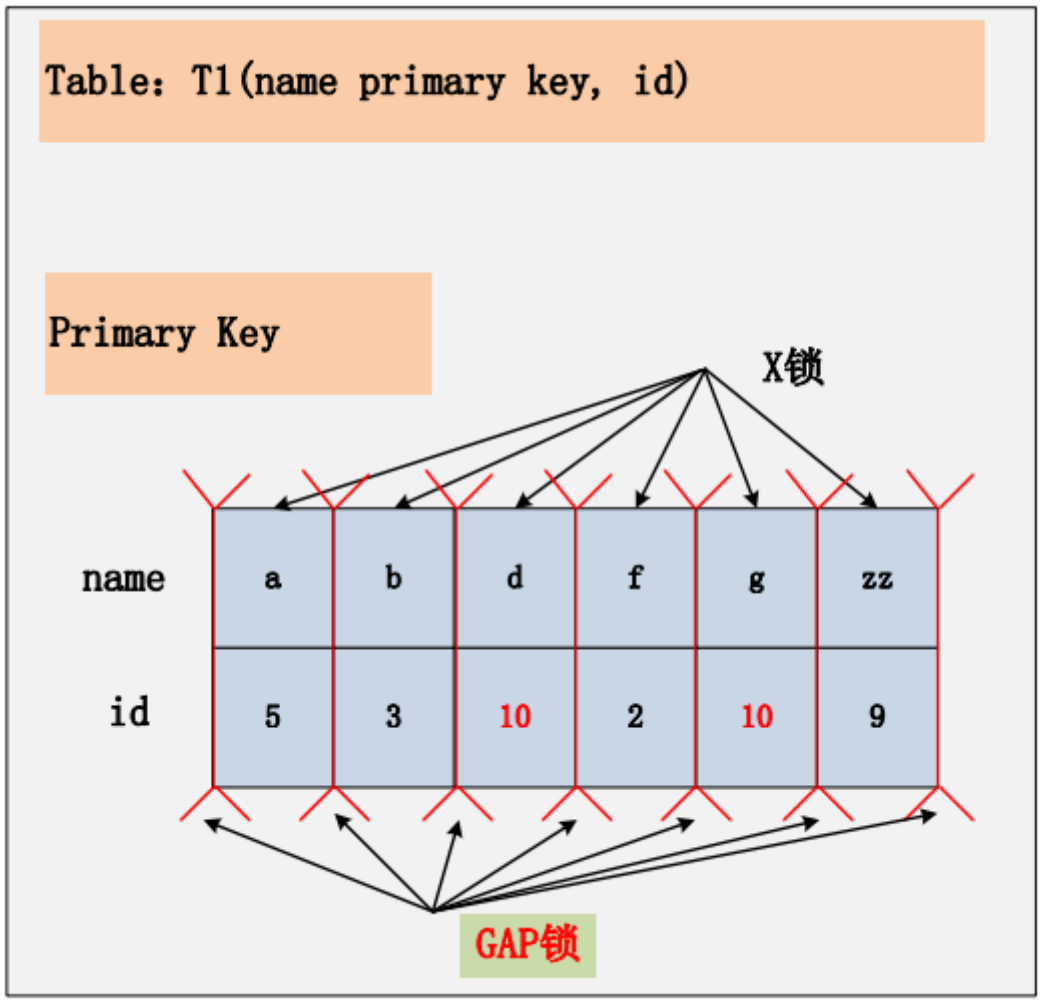
Repeatable Read 隔离级别，id 上有一个非唯一索引，执行 delete from t1 where id = 10; 假设选择 id 列上的索引进行条件过滤，最后的加锁行为，是怎么样的呢？同样看下面这幅图：



Repeatable Read 隔离级别下，id 列上有一个非唯一索引，对应 SQL：delete from t1 where id = 10; 首先，通过 id 索引定位到第一条满足查询条件的记录，加记录上的 X 锁，加 GAP 上的 GAP 锁，然后加主键聚簇索引上的记录 X 锁，然后返回；然后读取下一条，重复进行。直至进行到第一条不满足条件的记录[11,f]，此时，不需要加记录 X 锁，但是仍旧需要加 GAP 锁，最后返回结束。

情景八：id 无索引+RR

Repeatable Read 隔离级别下的最后一种情况，id 列上没有索引。此时 SQL：delete from t1 where id = 10; 没有其他的路径可以选择，只能进行全表扫描。最终的加锁情况，如下图所示：



在 Repeatable Read 隔离级别下，如果进行全表扫描的当前读，那么会锁上表中的所有记录，同时会锁上聚簇索引内的所有 GAP，杜绝所有的并发 更新/删除/插入 操作。当然，也可以通过触发 semi-consistent read，来缓解加锁开销与并发影响，但是 semi-consistent read 本身也会带来其他问题，不建议使用。

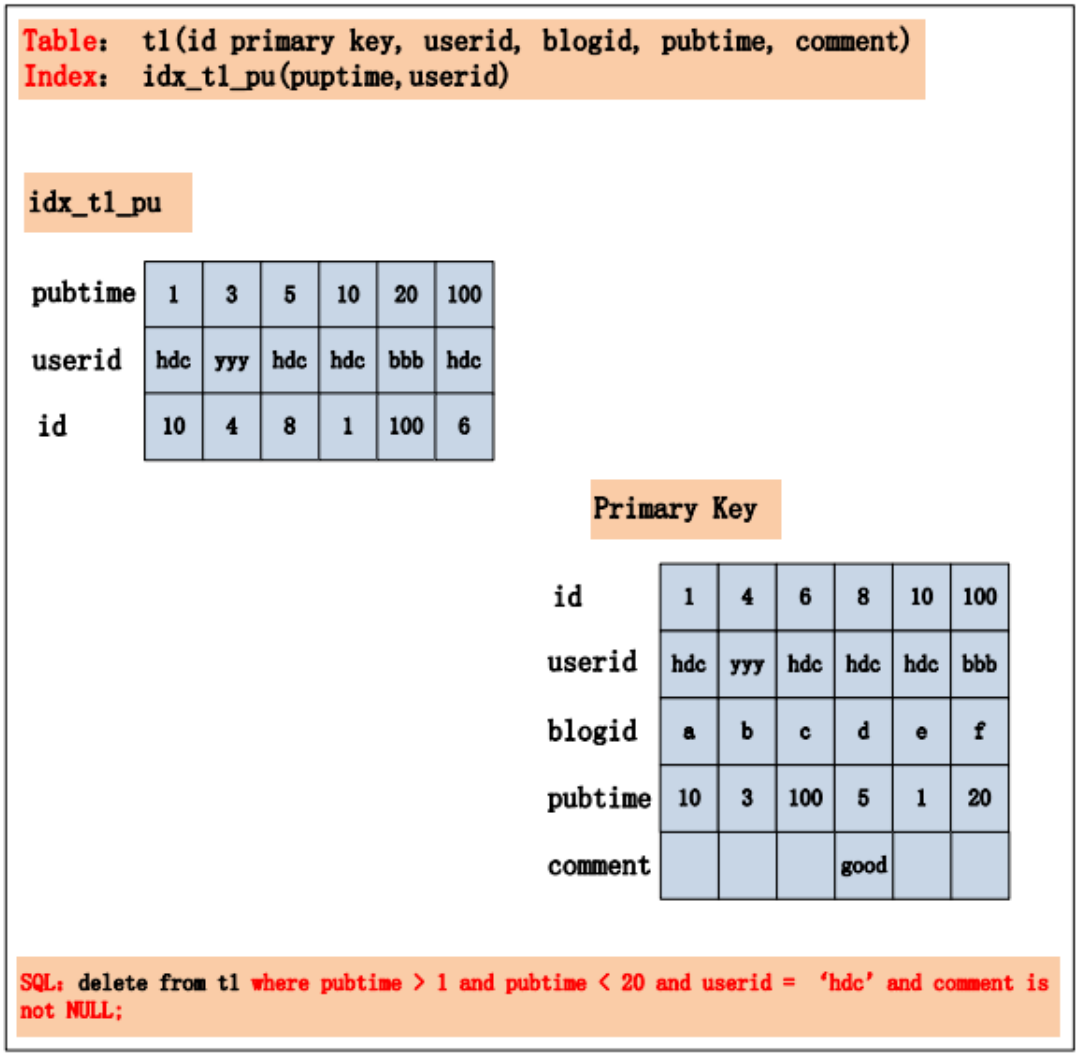
情景九：Serializable

针对前面提到的简单的 SQL，最后一个情况：Serializable 隔离级别。对于 SQL2：delete from t1 where id = 10; 来说，Serializable 隔离级别与 Repeatable Read 隔离级别完全一致，因此不
做介绍。
Serializable 隔离级别，影响的是 SQL1：select * from t1 where id = 10; 这条 SQL，在 RC，RR
隔离级别下，都是快照读，不加锁。但是在 Serializable 隔离级别，SQL1 会加读锁，也就是说快照读不复存在，MVCC 并发控制降级为 Lock-Based CC。
结论：在 MySQL/InnoDB 中，所谓的读不加锁，并不适用于所有的情况，而是隔离级别相关的。Serializable 隔离级别，读不加锁就不再成立，所有的读操作，都是当前读。

####

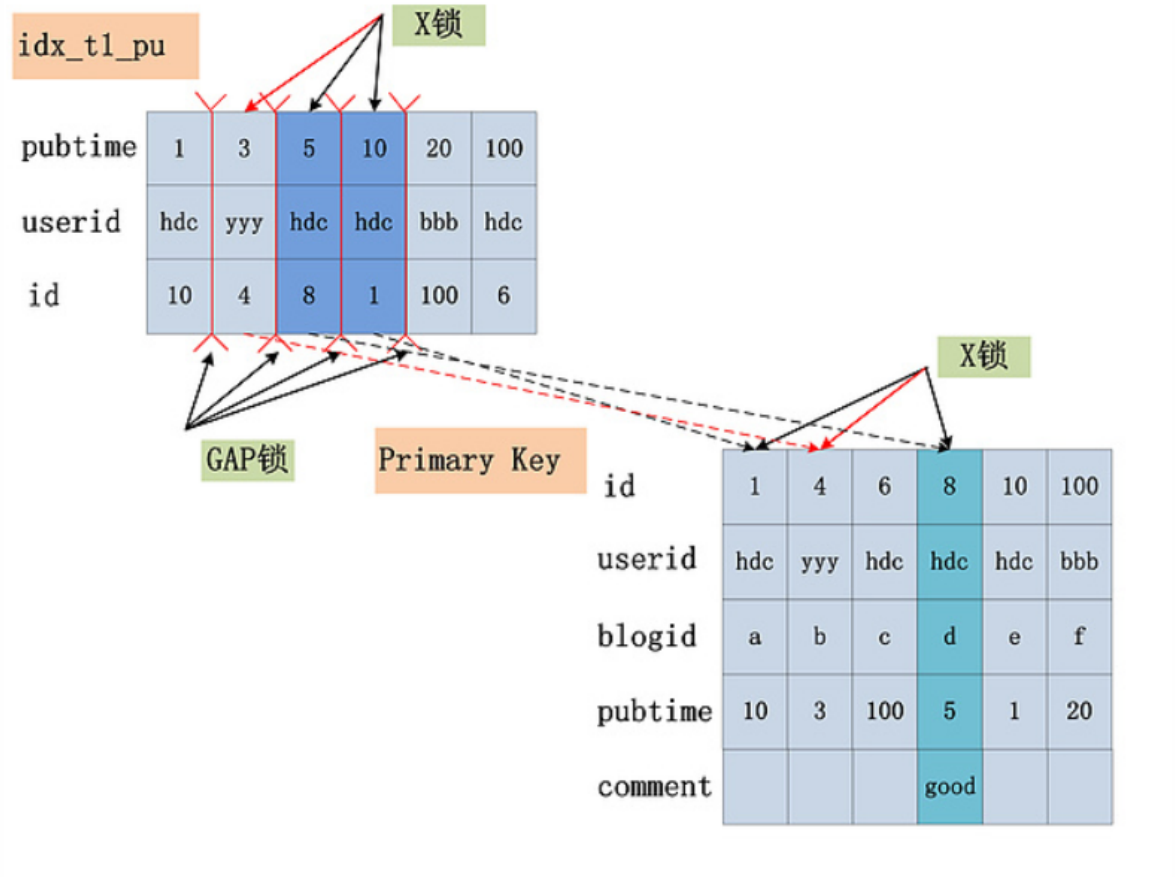
扩展作业：

如图中的 SQL，会加什么锁？假定在 Repeatable Read 隔离级别下.同时，假设 SQL 走的是 idx_t1_pu 索引。



在 Repeatable Read 隔离级别下，针对一个复杂的 SQL，首先需要提取其 where 条件。
Index Key 确定的范围，需要加上 GAP 锁；Index Filter 过滤条件，视 MySQL 版本是否支持 ICP，
若支持 ICP，则不满足 Index Filter 的记录，不加 X 锁，否则需要 X 锁；Table Filter 过滤条件，
无论是否满足，都需要加 X 锁。

Table: t1(id primary key, userid, blogid, pubtime, comment)
Index: idx_t1_pu(puptime,userid)



SQL: delete from t1 where pubtime > 1 and pubtime < 20 and userid = 'hdc' and comment is not NULL;

事务操作

```
# 自动提交机制
默认情况下，执行的任一条语句（select  update delete insert ），都会以一个独立事务存在，自动begin  commit。除非手工begin或者start transaction
mysql> select @@autocommit;

# 标准事务控制语句
begin;
commit;
rollback;
```

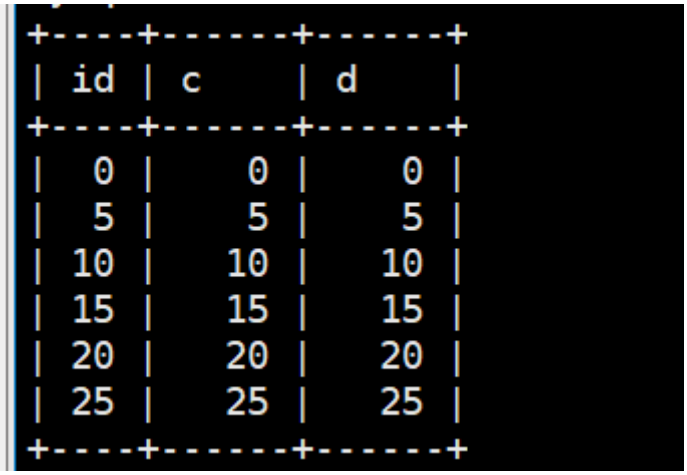
隔离级别操作

```
mysql> select @@transaction_isolation;
mysql> set global transaction_isolation='READ-COMMITTED';

脏读 、 不可重复读 、 幻读
RC级别的问题读：
不可重复读 、 幻读

RR级别
防止不可重复读？ MVCC 一致性快照读
配合GAP锁，防止幻读。
```

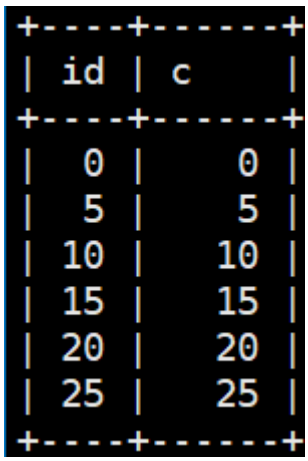
RR级别锁实践检验:

```
CREATE TABLE `t` (
  `id` int(11) NOT NULL,
  `c` int(11) DEFAULT NULL,
  `d` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `c` (`c`)
) ENGINE=InnoDB;

insert into t values
(0,0,0),
(5,5,5),
(10,10,10),
(15,15,15),
(20,20,20),
(25,25,25);
```

case1:



id	c	d
0	0	0
5	5	5
10	10	10
15	15	15
20	20	20
25	25	25



id	c
0	0
5	5
10	10
15	15
20	20
25	25

T1	T2	T3
update t set d=d+1 where id=7;		
	insert into t values(8,8,8); --???	
		update t set d=d+1 where id=10;??

case2：

T1	T2	T3
select id from t where c=5 lock in shard mode		
	update t set d=d+1 where id=5;	
		insert into t values(7,7,7);

case3 :

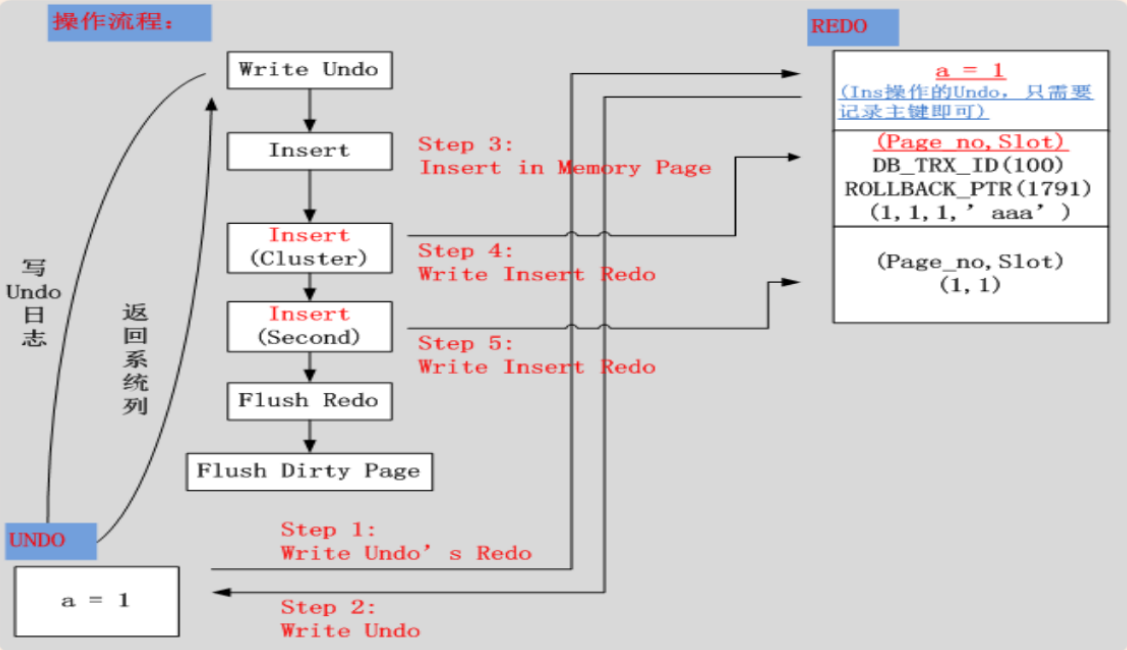
T1	T2	T3
update t set d=d+1 where id>12 and id<17		
	insert into t values(11,11,11) ---block	insert into t values(21,21,21); ---ok

9.8 扩展--InnoDB Undo、 Redo在i，u，d操作时的如何工作?

9.8.1 INSERT：

- Undo
- 将插入记录的主键值，写入Undo；
- Redo
- 将[space_id，page_no，完整插入记录，系统列，...]写入Redo；
- space_id，page_no 组合代表了日志操作的页面；

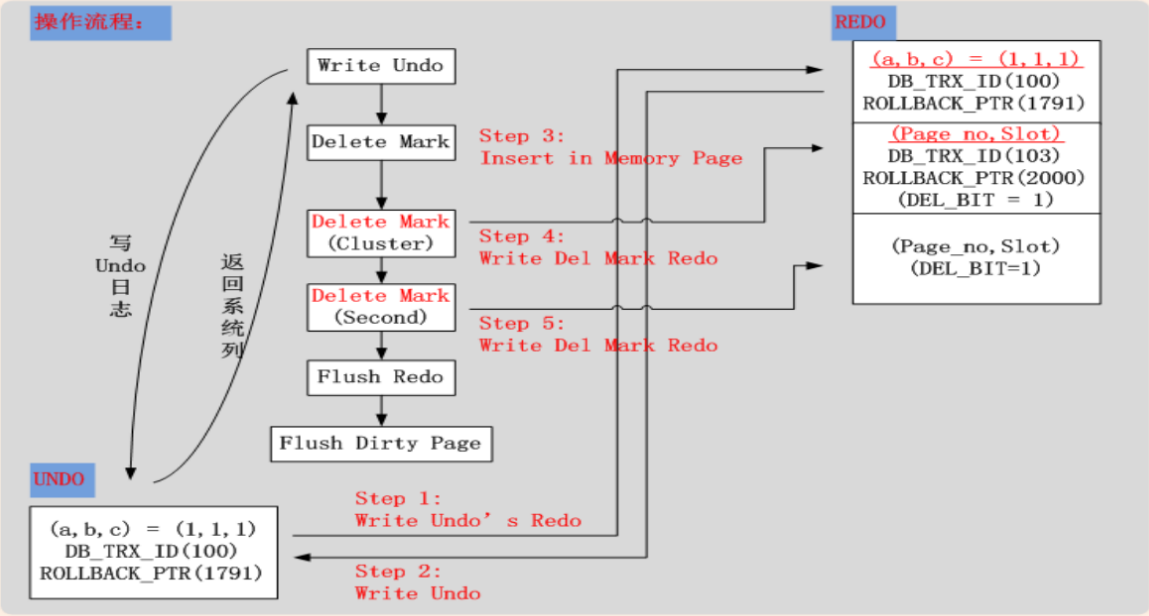
```
Create table t1 (a int primary key, b int, c int, d varchar(200))engine=innodb;
Create index idx_t1_bc on t1 (b, c);

Insert into t1 values (1,1,1,' aaa' );
```



9.8.2 DELETE

- Undo
- 1. Delete，在InnoDB内部为Delete Mark操作，将记录上标识Delete_Bit，而不删除记录；
- 2. 将当前记录的系统列写入Undo (DB_TRX_ID, ROLLBACK_PTR, ...)；
- 3. 将当前记录的主键列写入Undo；
- 4. 将当前记录的所有索引列写入Undo (why? for what?)；
- 5. 将Undo Page的修改，写入Redo；
- Redo
- 将[space_id，page_no，系统列，记录在页面中的Slot，...]写入Redo；

```
Delete from t1 where a = 1;
```



9.8.3 UPDATE

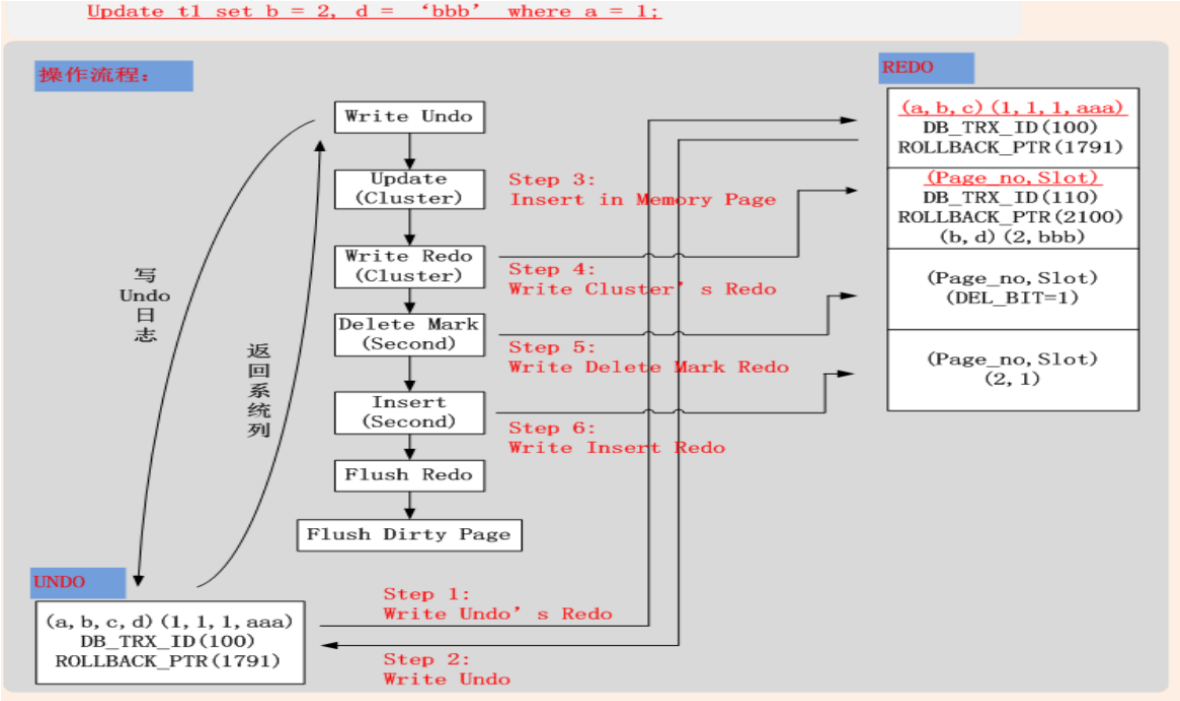
情况一：Update(未修改聚簇索引键值，属性列长度未变化)

- Undo (聚簇索引)
- 1. 将当前记录的系统列写入Undo (DB_TRX_ID, ROLLBACK_PTR, ...);
- 2. 将当前记录的主键列写入Undo;
- 3. 将当前Update列的前镜像写入Undo;
- 4. 若Update列中包含二级索引列，则将二级索引其他未修改列写入Undo;
- 5. 将Undo页面的修改，写入Redo;
- Redo
- 进行In Place Update，记录Update Redo日志(聚簇索引);
- 若更新列包含二级索引列，二级索引肯定不能进行In Place Update，记录Delete Mark + Insert Redo日志;

```
update t1 set b=2 ,d='bbb' where a=1

pk : a
MUL: b,c
```

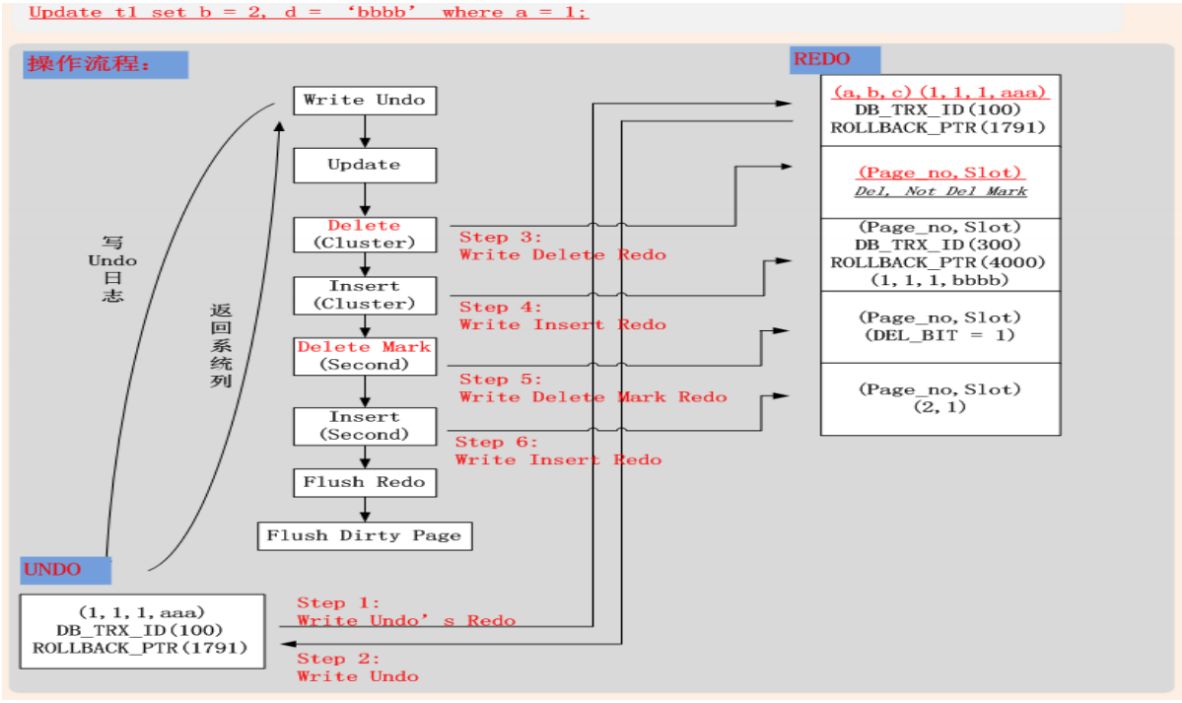
a(pK)	b	c	d
1	2	1	aaa-->bbbb

b	c	a
2	1	1



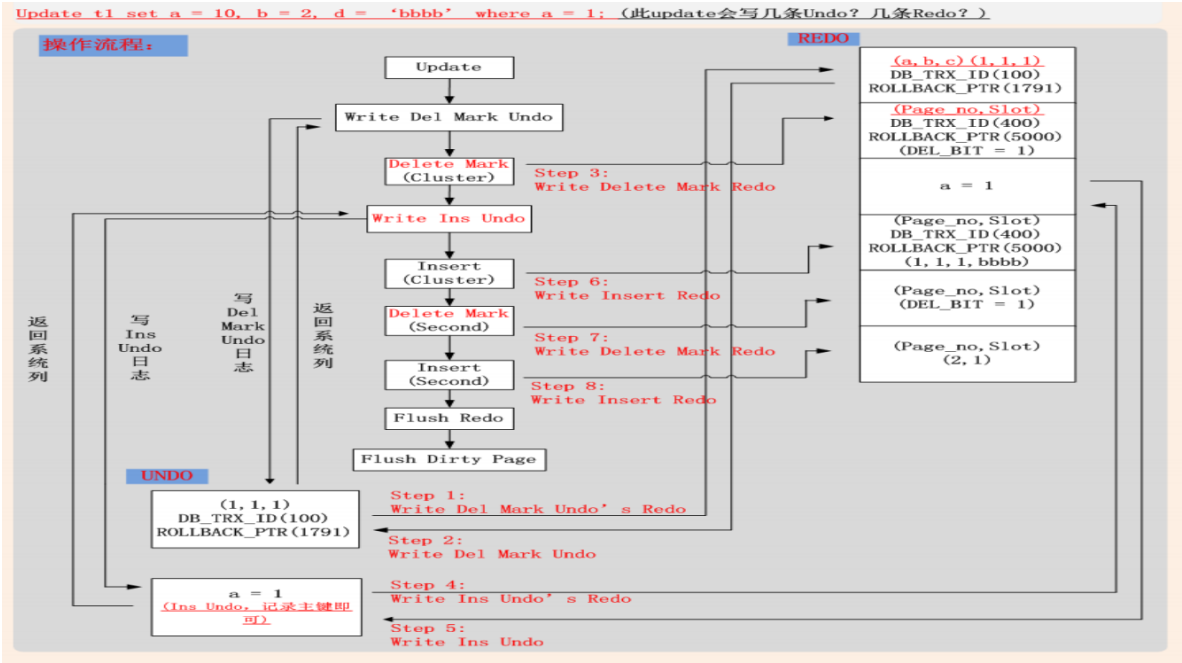
情况二： Update(未修改聚簇索引键值，属性列长度发生变化)

- Undo (聚簇索引)
- 1. 将当前记录的系统列写入Undo (DB_TRX_ID, ROLLBACK_PTR, ...);
- 2. 将当前记录的主键列写入Undo;
- 3. 将当前Update列的前镜像写入Undo;
- 4. 若Update列中包含二级索引列，则将二级索引其他未修改列写入Undo;
- 5. 将Undo页面的修改，写入Redo;
- Redo
- 不可进行In Place Update，记录Delete + Insert Redo日志(聚簇索引);
- 若更新列包含二级索引列，二级索引肯定不能进行In Place Update，记录Delete Mark + Insert Redo日志;



情况三：Update(修改聚簇索引键值)

- Undo（聚簇索引）
  - 1. 不可进行In Place Update。Update = Delete Mark + Insert；
  - 2. 对原有记录进行Delete Mark操作，写入Delete Mark操作Undo；
  - 3. 将新纪录插入聚簇索引，写入Insert操作Undo；
  - 4. 将Undo页面的修改，写入Redo；
- Redo
  - 不可进行In Place Update，记录Delete Mark + Insert Redo日志(聚簇索引)；
  - 若更新列包含二级索引列，二级索引肯定不能进行In Place Update，记录Delete Mark + Insert Redo日志；



9.10 MVCC 多版本并发控制

MVCC ： 多版本并发控制
功能： 通过UNDO生成多版本的”快照“。非锁定读取。
乐观锁： 乐观。
悲观锁： 悲观。
每个事务操作都要经历两个阶段：
1. MVCC采用乐观锁机制，实现非锁定读取。
2. 在RC级别下，事务中可以立即读取到其他事务commit过的readview
3. 在RR级别下，事务中从第一次查询开始，生成一个一致性readview，直到事务结束。

创建ReadView
- 获取kernel_mutex
- 遍历trx_sys的trx_list链表，获取所有活跃事务，创建ReadView
- Read Committed
- 语句开始，创建ReadView
- Repeatable Read
- 事务开始，创建ReadView

9.11 InnoDB 核心参数调整及状态监控

```
InnoDB Buffer Pool：
innodb_buffer_pool_size
innodb_buffer_pool_chunk_size
innodb_buffer_pool_instances
innodb_dedicated_server

show global status like '%innodb%wait%';
要具体真对每个服务器配置，进行压测，看平均IO延时，然后取得当前最佳值。
具体观测指标： hash_table_locks。

redo-log：
innodb_log_buffer_size=33554432
innodb_log_file_size
innodb_log_files_in_group
innodb_log_group_home_dir

重启生效：
[root@db01 data]# /etc/init.d/mysqld restart
show global status like '%innodb%log%';

change buffer:
```

```
innodb_change_buffer_max_size

表空间：
innodb_data_file_path=ibdata1:100M;ibdata2:100M;ibdata3:100M:autoextend
innodb_file_per_table=1
innodb_undo_tablespaces;  ---->3-5个     #打开独立undo模式，并设置undo的个数。
innodb_max_undo_log_size;                #undo日志的大小，默认1G。
innodb_undo_log_truncate;                #开启undo自动回收的机制（undo_purge）。
innodb_purge_rseg_truncate_frequency;    #触发自动回收的条件，单位是检测次数。
innodb_undo_directory
innodb_temp_data_file_path=ibtmp1:12M;ibtmp2:128M:autoextend:max:500M

innodb_io_capacity
innodb_io_capacity_max


innodb_max_dirty_pages_pct
transaction_isolation=RC\RR
innodb_lock_wait_timeout=10
```

9.13 扩展:Mini-TRansaction(MTR)

- 定义
- mini-transaction不属于事务；InnoDB内部使用
- 对于InnoDB内所有page的访问(I/U/D/S)，都需要mini-transaction支持
- 功能
- 访问page，对page加latch (只读访问：S latch；写访问：X latch)
- 修改page，写redo日志 (mtr本地缓存)
- page操作结束，提交mini-transaction (非事务提交)
- 将redo日志写入log buffer
- 将脏页加入Flush List链表
- 释放页面上的 S/X latch
- 总结
- mini-transaction，保证单page操作的原子性(读/写单一page)
- mini-transaction，保证多pages操作的原子性(索引SMO/记录链出，多pages访问的原子性)