2016-07-05 MySQL 安装 1

安装

1. 解压 tar zxf mysql-5.7.13-linux-glibc2.5-x86_64.tar.gz
2. cd mysql-5.7.13-linux-glibc2.5-x86_64
3. ls support-files/ 下面全是一些可执行脚本
4. mv mysql-5.7.13-linux-glibc2.5-x86_64 /usr/local/
5. 创建快捷连接 ln -s mysql-5.7.13-linux-glibc2.5-x86_64 mysql
6. groupadd mysql
7. useradd mysql -g mysql
8. chown -R root:mysql .（为了让 MySQL 数据库只让 root 和 mysql 用户使用）
9. 修改 /etc/my.cnf 文件
10. bin/mysqld --initialize（初始化一些 MySQL 的系统表）
11. cp support-files/mysql.server /etc/init.d/mysqld
12. 启动 mysql 服务器 /etc/init.d/mysqld start
13. 登录 mysql bin/mysql -p"**error.log 中的临时密码**"
14. 添加 MYSQL_HOME 到 PATH

目标

熟练安装 mysql-5.7

2016-07-07 MySQL 安装 2

MySQL 官方推荐的安装：

```
groupadd mysql
useradd -r -g mysql -s /bin/false mysql
cd /usr/local
tar zxvf /path/to/mysql-VERSION-OS.tar.gz
ln -s full-path-to-mysql-VERSION-OS mysql
cd mysql
mkdir mysql-files
chmod 750 mysql-files
chown -R mysql .
chgrp -R mysql .
bin/mysql_install_db --user=mysql    # Before MySQL 5.7.6
bin/mysqld --initialize --user=mysql # MySQL 5.7.6 and up
bin/mysql_ssl_rsa_setup              # MySQL 5.7.6 and up
chown -R root .
chown -R mysql data mysql-files
bin/mysqld_safe --user=mysql &
command is optional
cp support-files/mysql.server /etc/init.d/mysql.server
```

目录：mysql-files　作用？

官方推荐的安装方式的和其它安装方式的区别：是否开启 SSL

连接

mysql 支持的连接方式：

- SOCKET　本地（MySQL 服务端和 MySQL 客户端在同一台机器）
- TCP/IP　网络

SOCKET

SOCKET 连接：mysql -u[root] -p[password]
免密码登录：在/etc/my.cnf 中在[client]下配置 user 和 password
如下：

```
[client]
user = root
password = Abc001@123
```

TCP/IP

TCP/IP 连接：mysql -h[host] -u[user] -p[password] -P[port]

bind_address：一般绑定在内网 ip 上，如 192.168.1.101

SSL 连接
开启 SSL：
1. bin/mysql_ssl_rsa_setup
2. cd /mysql_data/data
3. chown -R mysql:mysql *.pem （赋予新生成的文件 mysql 组下的 mysql 用户的权限）
4. 使用 mysql 客户端通过 TCP/IP 连接登录时，默认开启 ssl
5. mysql -h[host] -u[user] -p[password] -P[port] **(--ssl=0 关闭)**

升级

mysql 升级策略： in-place upgrade（原地更新）
1. /etc/init.d/mysqld stop
2. cd /usr/local/mysql
3. unlink mysql 【原始指向 ln -s /root/software/mysql-5.6.10 mysql】
4. ln -s /root/software/mysql-5.7.13 mysql 【软链重新指向】
5. cd /mysql_data/data && mysql_upgrade -s -uroot -p

升级可能遇到的问题：
系统文件还是 mysql-5.6，需要使用 mysql_upgrade -s -u -p 更新系统库，不更新数据文件，因为数据文件都是兼容的。
不加 -s 时，会对 data_dir 下所有的文件进行升级重建，这是没有必要的。
具体指：只对 **mysql 和 performance_schema 文件夹下的系统库** 进行升级

多实例安装

- 在一台服务器上安装多个 MySQL 实例
- 充分利用硬件资源（提高资源利用率，降低成本）
- 通过 mysqld_multi 程序安装

类似于一台服务器上登录多个不同的帐号
通常用于游戏数据库
和普通安装的区别：

	mysql1	mysql2	mysql3
port	3306	3307	3308
data_dir	/mysql_data/data1	/mysql_data /data2	/mysql_data /data3
socket	/tmp/mysql.sock1	/tmp/mysql.sock2	/tmp/mysql.sock3

运行在不同的进程上

配置文件修改

vi /etc/my.cnf
在文件末尾添加

```
[mysqld_multi]
mysqld      = /usr/local/mysql/bin/mysqld_safe #【用于启动 mysqld_multi】
mysqladmin = /usr/local/mysql/bin/mysqladmin #【用于关闭 mysqld_multi】
log=   /usr/local/mysql/mysqld_multi.log
[mysqld1]
port = 3307
server-id = 10001
datadir= /mysql_data/data1/data
socket = /mysql_data/data1/tmp/mysql1.sock

[mysqld2]
port = 3308
server-id = 10002
datadir= /mysql_data/data2/data
socket = /mysql_data/data2/tmp/mysql2.sock

[mysqld3]
port = 3309
server-id = 10003
datadir= /mysql_data/data3/data
socket = /mysql_data/data3/tmp/mysql3.sock

[mysqld4]
port = 3310
server-id = 10004
datadir= /mysql_data/data4/data
socket = /mysql_data/data4/tmp/mysql4.sock
```

配置完成后可以通过命令 mysqld_multi report 查看
查看的主要是 my.cnf 标签中[mysqld_multi]标签下的[mysql1]、[mysql2]、[mysql3]、、、、

查看配置文件加载顺序（命令 **mysql --help**）：
【在 Default options are read from the following files in the given order 后面】
linux
1. /etc/my.cnf
2. /etc/mysql/my.cnf
3. /usr/local/mysql/etc/my.cnf
4. ~/.my.cnf

windows
1. C:\Windows\my.ini
2. C:\Windows\my.cnf
3. C:\my.ini
4. C:\my.cnf
5. C:\software\mysql-5.7.10-winx64\my.ini
6. C:\software\mysql-5.7.10-winx64\my.cnf

初始化实例

需要特别指定 datadir 因为 mysql 不支持共享数据
需要分别初始化：
/usr/local/mysql/bin/mysqld --initialize --datadir=/mysql_data/data1
/usr/local/mysql/bin/mysqld --initialize --datadir=/mysql_data/data2
/usr/local/mysql/bin/mysqld --initialize --datadir=/mysql_data/data3
/usr/local/mysql/bin/mysqld --initialize --datadir=/mysql_data/data4

MySQL 的实例不共享任何数据，区别于 Oracle 的共享表空间等

[mysqld1]、[mysqld2]、[mysqld3]、[mysqld4]的所有配置都会继承上面配置文件中的[mysqld]和[mysqld-5.7]

server-id 也需要不一致【安装主从的时候，才需要配置】

关闭多实例

mysqld_multi stop n 【关闭第 n 个】
mysqld_multi stop 关闭所有的多实例

注意

mysqld_multi 可以启动多个版本的 MySQL
在[mysqldn]下配置
basedir = /usr/local/mysql56

mysqld_safe 守护进程

是每个 mysqld 的守护进程

在/etc/init.d/mysqld 脚本中，实际通过 mysqld_safe 进程来启动 mysqld 进程

如果 mysqld 进程 down 了，当 mysqld_safe 检测到 mysqld 被关闭之后，会自动重启 mysqld 服务

生产环境启动哪个好？即：是否需要守护进程？
在高可用环境下，一般不需要立即重启。
物理机：mysqld_safe
云环境：mysqld

innodb_buffer_pool_size 是物理机内存的最大大小，所有实例的内存大小总和一般不高于物理机的 70%。

mysqladmin 管理命令

可以根据 -P port 或者 -S /m_data/data1/tmp/mysql.sock

如：mysqladmin -uroot -p shutdown -S /mysql_data/data1/tmp/mysql1.sock

目标

1. 安装 mysqld_multi
2. 思考/etc/init.d/mysqld stop 实现原理

2016-07-12 权限管理

用户权限管理

grant

grant 语法：

```
GRANT

    priv_type [(column_list)]
      [, priv_type [(column_list)]] ...
    ON [object_type] priv_level
    TO user_specification [, user_specification] ...
    [REQUIRE {NONE | tls_option [[AND] tls_option] ...}]
    [WITH {GRANT OPTION | resource_option} ...]

GRANT PROXY ON user_specification
    TO user_specification [, user_specification] ...
    [WITH GRANT OPTION]

object_type: {
    TABLE
  | FUNCTION
  | PROCEDURE
}

priv_level: {
    *
  | *.*
```

```
  | db_name.*
  | db_name.tbl_name
  | tbl_name
  | db_name.routine_name
}

user_specification:
    user [ auth_option ]

auth_option: {     # Before MySQL 5.7.6
    IDENTIFIED BY 'auth_string'
  | IDENTIFIED BY PASSWORD 'hash_string'
  | IDENTIFIED WITH auth_plugin
  | IDENTIFIED WITH auth_plugin AS 'hash_string'
}

auth_option: {     # As of MySQL 5.7.6
    IDENTIFIED BY 'auth_string'
  | IDENTIFIED BY PASSWORD 'hash_string'
  | IDENTIFIED WITH auth_plugin
  | IDENTIFIED WITH auth_plugin BY 'auth_string'
  | IDENTIFIED WITH auth_plugin AS 'hash_string'
}

tls_option: {
    SSL
  | X509
  | CIPHER 'cipher'
  | ISSUER 'issuer'
  | SUBJECT 'subject'
}

resource_option: {
  | MAX_QUERIES_PER_HOUR count
  | MAX_UPDATES_PER_HOUR count
  | MAX_CONNECTIONS_PER_HOUR count
  | MAX_USER_CONNECTIONS count
}
```

mysql 库下：
user：查看对所有库的权限
db：查看对指定库的权限

MAX_CONNECTIONS_PER_HOUR：每小时可以连接多少次
MAX_USER_CONNECTIONS：同时可以有多少个用户连接

示例：给报表用户添加权限
grant
    select
on
    wp.wp_report
to
    'report'@'192.168.1.%' identified by '123'
with
    max_queries_per_hour 3
    max_connections 1

revoke

revoke 语法

```
REVOKE

    priv_type [(column_list)]
      [, priv_type [(column_list)]] ...
    ON [object_type] priv_level
    FROM user [, user] ...

REVOKE ALL PRIVILEGES, GRANT OPTION
    FROM user [, user] ...

REVOKE PROXY ON user
    FROM user [, user] ...
```

revoke 只是删除权限，但是不删除用户
revoke all on *.* from 'senior_dba'@'127.0.0.1'

所有的用户都有登录权限（即：usage 权限）

删除用户：drop user
flush privileges;刷新权限【通过修改 mysql 下的 user 表和 db 表的时候，才需要这么做】
通过 grant revoke 不需要刷新权限

mysql5.6 和 mysql5.7 关于密码保存的区别：

5.6 保存在 password 字段

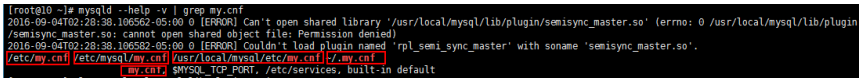
5.7 保存在 authentication_string 字段

配置

升级：mysql_upgrade -s 只升级原数据表

mysql 参数覆盖问题：

mysqld --help -v | grep my.cnf



mysql 会依次启动 my.cnf 文件，按照参数替换原则进行加载，并且后面的参数会替换后面的文件

顺序如图：

1. /etc/my.cnf
2. /etc/mysql/my.cnf
3. /usr/local/mysql/my.cnf 【mysql 安装文件所在目录】
4. ~/my.cnf 【用户目录】
5. my.cnf 【当前文件夹下】

PATH 配置：PATH=/usr/local/mysql/bin:$PATH 【需要把高版本的 mysql 放在最前面，是为了防止其它目录下的低版本 mysql 客户端干扰】

最大连接数：max_connections【默认是 100】

错误 error 1040：Too many connections

当线上环境：max_connections 被打爆时（连接数太多），如何解决？通常来说，此时 CPU 已经达到 100%，一般情况下：重启解决

1. 写一个监控程序，长连接到 mysql，一直不退出
2. 使用线程池【mysql 内部的线程池与一般应用中的连接池不同】

在线修改参数：set **session|global** max_connections = 1;

开启线程池（thread_pool）

开启线程池之后，可以开启额外的参数：

extra_port：额外的端口，不受 max_connections 参数限制

连接池：快速的建立连接
线程池：mysql 内部用于限制连接

2016-07-13 数据类型和函数

数据类型

int

unsigned/signed

是否有符号

不建议使用 unsigned
原因：数据容易溢出

zerofilll

zerofilll 只是一个显示出来的值，不改变数据本身的大小。
设置为 zerofilll 后，默认的数据类型是： int unsigned zerofilll（无符号类型）

auto_increment

- 自增
- 要求**自增列必须是键值**（主键、唯一键、普通键）
- 每张表只有一个

在自增的列上添加的值为 **NULL** 或者 **0** 都表示是自增
重启之后的 auto_increment 的值是最大的主键值：select **max(pk) + step as auto** from tb1;
这个值不是持久化的
如何使用？
建议将自增的主键设置为 bigint（long）

数字类型

- 单精度：FLOAT
- 双精度：DOUBLE
- 高精度：DECIMAL

格式：
FLOAT(M,D) / DOUBLE(M,D) / DECIMAL(M,D) 表示：显示 M 位整数，D 位小数

当：create table t1(a decimal(5,2))
插入：insert into t1 values(999.999); 插入失败。
报错：out of range values for column

如果 **sql_mode** 不是严格类型：可以插入，但是显示的值是：999.99，忽略了最后一位。
mysql5.6 是宽松模式
mysql5.7 是严格模式 sql_mode = strict_trans_tables

字符串类型

CHAR(N)	定长字符	字符	是	255
VARCHAR(N)	变长字符	字符	是	16384
BINARY(N)	定长二进制字节	字节	否	255
VARBINARY(N)	变长二进制字节	字节	否	16384
TINYBLOB	二进制大对象	字节	否	256
BLOB	二进制大对象	字节	否	16K
MEDIUMBLOB	二进制大对象	字节	否	16M
LONGBLOB	二进制大对象	字节	否	4G
TINYTEXT	大对象	字节	是	256
TEXT	大对象	字节	是	16K
MEDIUMTEXT	大对象	字节	是	16M
LONGTEXT	大对象	字节	是	4G

存储字符

定长 CHAR(N)
变长 VARCHAR(N)
N 表示字符个数

存储字节

字节 BINARY(N)
字符 VARBINARY(N)
N 表示字节个数

insert into tb1 values(**0xE68892,0x62**);
转成 16 进制，保存到数据库，可以较少的出现字符集兼容问题。

只有 char 和 varchar 是字符的，其它的都是字节。16383 字节

BLOB 一般用于保存图像、视频等。
但是一般情况下，图像和视频都存在专门的图片和视频服务器中。

BLOB 和 TEXT 通常只能使用前缀索引，大小由参数 **max_sort_length** 控制。通常使用全文索引，而不是 b+树索引。

字符集

character set

常见：**utf8mb4（推荐）**、utf8、gbk、gb18030…。这些都算是变长字符集

查看 MySQL 支持的字符集：show character set;

// 配置文件中设置
character_set_server=utf8mb4

Collation

MySQL 支持的 Collation：show collation;

字符的比较规则。

set of rules for comparing characters in a character set。

默认的比较规则：**utf8mb4_general_ci**，不区分大小写。
如果需要区分大小写：**utf8mb4_bin**

在默认的 collation 下，比较大小时，会截断字符后的空格，但是前面的空格不会截断。

控制参数：
collation_connection
collation_database
collation_server

一般密码区分大小写。

函数

hex()

将对应字符转为 16 进制

2016-07-14 数据类型和函数

collation

MySQL 默认不区分大小写
- 不区分大小写：utf8mb4_general_ci
- 区分大小写：utf8mb4_bin
- set NAMES utf8 COLLATE utf8mb4_general_ci;
- set NAMES utf8 COLLATE utf8mb4_bin;
- 不区分大小写模式下：只会截断后段的空格，不会截断前段的空格。

函数

Duplicate key

sysdate()和 now()函数不一致。
sysdate():返回执行到函数的时间
now():返回执行 select 的时间

内置函数不走索引
例如：查询 2016 年 7 月 14 号的所有订单(a 不走索引，b 走索引)
a:select * from orders where date_format(order_time,'%Y%m%d')='20160714';
b:select * from orders where order_time >= '2016-07-14' and order_time < '2016-07-15';

求出员工的下一个生日是什么时间？使用 **date_add**、**date_diff** 函数。employees.employees

employees 库

需要从官网下载
需要把 employees.sql 中的 storage_engine 改成 default_storage_engine【5.6 以后的参数】

导入命令：mysql < employees.sql

问题：求出员工的最近一次生日？

length() 返回的是：字节长度，与字符集有关。
char_length() 返回的是：字符长度。
left(var,len)：截取字符串的左边 len 位。
mid(var,startIndex,len)：从 start 开始截取 len 位。
instr(a,b)：b 字符出现在 a 中的位置。

JSON

schema free

好处：
- 插入数据时，可以检查数据的有效性
- 查询性能高（能达到文档数据库的性能）
- 支持部分属性索引

MySQL5.7 支持函数索引，可以在 JSON 上快速的索引。

JSON 类型相关的函数：
json_append

json_contains_key
json_extract
json_merge
json_remove
json_replace
json_search
json_set
json_test_parse
json_valid
json_array_insert
json_unquote:去除 json 的引号

```
create table json_tb(
    id bigint primary key auto_increment,
    data json
)

insert into json_tb(data) values('{"name":"Tom","age":"12"}'),('{"name":"John","age":"13"}');
```

根据 json 列查询

```
select * from json_tb where json_unquote(json_extract(data,"$.name")) = 'Tom';
```

创建函数索引
虚拟列：不占用存储空间

1.创建虚拟列
2.对虚拟列添加索引

```
alter table json_tb add column name varchar(128) as
(json_unquote(json_extract(data,"$.name"))) virtual;
alter table json_tb add index idx_name(name);
```

json 虚拟列 create table 时，添加虚拟列。

```
create table test1(
    id bigint primary key auto_increment,
    data json,
    name varchar(32) as (json_unquote(json_extract(data,"$.name"))) virtual
)
```

有 json 列对应的虚拟列类型时，插入数据

```
insert into tb(id,json_col) values(null,'{}');
```

不能指定虚拟列

BIT

不要使用 bit 数据类型。数据丢失后，无法恢复！

底层使用 int 实现。

2016-07-16

1.表结构关系
    视图是一个虚表
    反引号 `` 作用：避免关键字冲突,如下：
    create table a (`int` int);

    建表语句中的 data directory 是指数据存储目录，默认位置是 datadir

    创建临时表：create temporary table a (a int);
    临时表用处：只对当前会话可见，对其它会话不可见。
    临时表保存在：ibtmp1(5.7 临时表空间)

    当临时表和用户表同名时，优先使用临时表返回数据。

    一般在使用存储过程或者游标时，会用到临时表

2.外键约束
    foreign key
3.alter table

2016-07-19 分区表和 SQL 语句

分区表

注意：分区通常是针对日期字段进行

特殊的数据类型：NULL

NULL != NULL
NULL **IS[NOT] NULL**
NULL != “”（空字符串）

MySQL 中把 NULL 值看作是负无穷。如果分区的列是 NULL 值，则这条记录会插入到第一个分区。

分区类型

range list hash key 分区条件必须是 INT 类型
columns(5.5)

建议所有的字段类型都是 NOT NULL
建议分区键都是 **NOT NULL**

range

```
create table t_range(
    id int primary key
) engine=innodb
partition by range(id)(
    partition p0 values less than (10),
    partition p1 values less than (20)
);
```

物理存储：有多个 idb 等数据文件
frm 是表空间文件。
插入“10”时，插入在 p1 分区。
验证：通过 explain select * from tb where id = 10; 可以查看到 partition 分区信息。

插入 id 大于 20 时，报错：Table has no partition for value 30。

分区字段必须是整型的。
如下图：
    p01 分区实际存放的是 1984 年的数据，
    p19 分区实际存放的是 2002 年及以后的年份。

```
ALTER TABLE titles
partition by range (year(from_date))
(
    partition p01 values less than (1985),
    partition p02 values less than (1986),
    partition p03 values less than (1987),
    partition p04 values less than (1988),
    partition p05 values less than (1989),
    partition p06 values less than (1990),
    partition p07 values less than (1991),
    partition p08 values less than (1992),
    partition p09 values less than (1993),
    partition p10 values less than (1994),
    partition p11 values less than (1995),
    partition p12 values less than (1996),
    partition p13 values less than (1997),
    partition p14 values less than (1998),
    partition p15 values less than (1999),
    partition p16 values less than (2000),
    partition p17 values less than (2001),
    partition p18 values less than (2002),
    partition p19 values less than (MAXVALUE)
);
```

分区时才有 MAXVALUE

list

```
create table t_list(
    id int,
    b int
) engine=innodb
partition by list(b)(
    partition p0 values in(1,3,5,7,9),
    partition p1 values in(0,2,4,6,8)
);
```

类似 range 分区

hash

```
create table t_hash(
    a int,
    b datetime
) engine=innodb
partition by hash(year(b))
partitions 4; 相对平均的存放到 4 个分区
```

key

```
create table t_key(
    a int,
    b datetime
) engine=innodb
partition by key(b)
partitions 4;
```

MySQL 会自动对 key 进行 hash 或者 md5 计算出一个 INT 类型。

如果需要每周产生一个分区，则分区函数应该如何使用？

columns

```
create table t_columns(
    a int,
    b datetime
) engine=innodb
partition by range [list] columns(b)(
    partition p0 values less than ('2009-01-01'),
    partition p1 values less than ('2010-01-01')
);
```

子分区

在分区的基础上再进行分区
也称为复合分区
只支持在 range 和 list 的分区上再进行 hash 或者 key 的子分区

```
create table ts(
    a int,
    b datetime
)engine=innodb
partition by range(year(b))
subpartition by hash(to_days(b))
subpartitions 2(
    partition p0 values less than(1990),
    partition p1 values less than(2000),
    partition p2 values less than(MAXVALUE)
)
```

先按照 range 分区分为 3 个分区，然后每个分区又有 2 个子分区。

分区是否对性能有提升？
主要通过查询方式来判断是否有性能提升。

MySQL 的分区是局部分区：即所有的索引只在当前分区里面。

```
create table t1(
    col1 int not null,
    col2 int not null
    unique key(col1)
)engine=innodb
partition by hash(col2) partitions 4;  将会创建失败。
```

分区键必须是唯一索引的一部分。

```
use information_schema;
select * from partitions; //  记录了所有的分区表的信息。
```

MySQL5.6 默认看不到 partitions，需要 explain partitions select * from a;
MySQL5.7 默认已经能看到 partitions。

SQL 语句

语法

SQL 执行顺序
(8) **SELECT**
(9) **DISTINCT** <select_list>
(1) **FROM** <left_table>
(3) <join_type> **JOIN** <right_table>
(2) **ON** <join_condition>
(4) **WHERE** <where_condition>
(5) **GROUP BY** <group_by_list>
(6) **WITH** {CUBE | ROLLUP}
(7) **HAVING** <having_condition>
(10) **ORDER BY** <order_by_list>
(11) **LIMIT** <limit_number>

LIMIT

当 LIMIT 很大的时候，效率会很慢。
分页时：推荐使用 select * from tb where pk > ? limit offset,limit

ORDER BY

MySQL 中，如果不加 order by，则会随机取 n 条记录，不一定按照主键取。
如何根据 order by 的字段进行分页？
即：如何优化分页查询
select emp_no,first_name,last_name from employees order by last_name limit 10;

order by 1,2 … 其实是对应第一个字段、第二个字段、…

排序需要使用的参数：sort_buffer_size 是一个会话级别的参数
查看当前会话排序相关的状态值：**show status like 'sort';**

2016-07-21 SQL 语句

语法

group by

group by 后一般会加上一个分组的函数（sum，count，avg，…）

having 对 group by 分组之后，使用分组条件对数据进行过滤。

where 过滤的是非聚合的结果，即查询的记录
having 过滤的是聚合后的结果。

group by 一般会产生临时表，因此需要调高临时表参数，默认是 128K

设置会话级别的变量：**tmp_table_size**

count(*) 与 count(1) 的区别？
答：没有什么区别！
只有 count 具体字段时，才有区别，count 具体字段时，只会返回该字段的数量，不包括 NULL 值，count(*)则会包含 NULL 值

sql_mode：only_full_group_by 需要设置为这样。

否则会出现如下的错误：
create table a(userid int,price int,date datetime);

insert into a values(1,10,'2017-01-01 12:00:00');
insert into a values(2,10,'2017-01-01 12:00:00');
insert into a values(3,10,'2017-01-01 12:00:00');
insert into a values(2,10,'2017-01-01 12:00:00');
insert into a values(1,10,'2017-01-01 12:00:00');

select userid,sum(price),date from a group by a;

这条 sql 实际是错误的，在 sql_mode 没有设置 only_full_group_by 时，不抱错，但是设置之后，就报错。因此强烈需要把这个模式打开。
如何求出分组之后的第一个时间呢？或者求出前几条

函数：group_concat

```
GROUP_CONCAT([DISTINCT] expr [,expr ...]
            [ORDER BY {unsigned_integer | col_name | expr}
                [ASC | DESC] [,col_name ...]]
            [SEPARATOR str_val])
```

SQL 举例：
select userid,group_concat(price order by date desc separator '-') price_list from c1 group by userid;

having

表示对**聚合（如：group by）**之后的表达式进行过滤

where 和 having 的区别？
having 之后的字段，必须在 select 子句中

join

包括 inner join 和 outer join。其中 inner 和 outer 可以省略

inner join

select a.*,b.* from a,b where a.x = b.y
select * from a join b on a.x = b.y
这两种写法没有什么不同。
出现 join on 语法的目的是为了支持外连接。

select * from a,b where a.x = b.y and b.y > 1 先做关联还是先做过滤呢？
一般情况下：根据关联条件中**过滤度高的条件进行过滤**。

下面两条 SQL 的区别：
select * from a inner join b on (a.x = b.y and b.y > 1)
select * from a inner join b on a.x = b.y where b.y > 1

执行结果一致
过滤条件既可以写在 on 后，也可以写在 where 后

outer join

保留表中的数据必须返回，不存在的数据，以 NULL 值表示。

left outer join

左表是保留表

取出在 a 表中存在，但是在 b 表中不存在的数据。
select * from a left join b on a.x = b.y where b.y is NULL;

right outer join

右表是保留表

2016-07-23 子查询

分类

普通子查询

operand comparison_operator **ANY** (subquery)
operand comparison_operator **ALL** (subquery)
operand comparison_operator **SOME** (subquery)
operand **IN** (subquery)

相关子查询

引用了外部查询列的子查询

使用

IN

select s1 from t1 where s1 = ANY (select s1 from t2);
select s1 from t1 where s1 IN (select s1 from t2);

某个列 in 某个子查询

NOT IN

例子：在 A 中，但是不在 B 中
select a.x from left join (select distinct y from b) c on a.x = c.y

NOT IN
只会返回 0 和 NULL 都表示 FALSE

EXISTS

只返回 TRUE、FALSE

exists 就是一个相关子查询，必须要和外部表进行关联。

和 IN 的区别？
select * from a where a.x in (select b.y from b);
select * from a exists (select * from b where a.x = b.y);

分页优化

select * from employees order by birth_date limit 3000000,10;
优化：
alter table employees add index idx_birth_date(**birth_date,emp_no**);

select * from employees where
(birth_date,emp_no) > ('1952-02-02',217446)
order by birth_date,emp_no limit 3000000,10;
效果：只能进行上一页、下一页，不能直接进行跳页

2016-07-26 多表关联

分为 inner join 和 outer join

下面两条 SQL 的区别是什么呢？
1. select * from x join y on x.a = y.a;
2. select * from x where a in (select a from y);
答：1 可能有重复的 a 值，2 中没有重复的
如果需要改写则：select distinct x.* from x join y on x.a = y.a;

行号问题：
set @a:=0;
select @a:=@a+1 as row_num, tb.* from tb limit 10;
推荐：
select @a:=@a+1 as rownum,e.* from employees e,(select @a:=0) a limit 10;
所有 select 不带 order by 时，都表示随机取，都表示随机从集合中取多少条记录，因此记录不一定是排序的。

使用子查询方式获取行号
select emp_no,(select count(1) from employees t2 where t2.emp_no <=t1.emp_no) as row_num from employees t1 order by emp_no limit 5;

2016-07-28 DML、触发器

Prepare SQL
1. PS Cache
2. 防范 SQL 注入 【MySQL 会做属性转换】
3. 动态查询
使用：
set @sql = 'select * from tb where id = ?';
set @id = 12345;
prepare stmt from @sql;
execute stmt using @a;
deallocate prepare stmt;

DML 语句

1. null 值在分区表的最小的分区中
2. insert 语法

```
insert into a values(1),(2),(3);
// 遇到重复的值就更新
// 只会更新主键的值，其它的值不变
insert into t(x) values() on duplicate key update x = x + 1;

replace into t(a,b) values(11,22);

insert into t(a,b) select a,b from t1;

insert into t(a,b) set a = 'a1',b = 'b1'
```

3. delete

```
delete from tb

使用 not in 的时候，如果子句中有 null 值，需要排除 null
select * from tb1 where tb1.a not in (1,2,null)
```

4. update

```
tb:
a   b   c
1   10  100
2   20  200
3   30  300
```

update tb set a = a + 1,b = a + 10 where a = 2;

更新之后：a = 3, b = 13

replace 引起的问题

replace 实际进行的是 insert into 操作

唯一索引的问题：可能会删除多余的列

例子：

```
CREATE TABLE `a11` (
  `a` int(11) NOT NULL,
  `b` int(11) DEFAULT NULL,
  PRIMARY KEY (`a`),
  UNIQUE KEY `b` (`b`)
)
```

insert into a11 values(1,2),(2,3),(3,4);　　此时 select * 共有 3 条记录

replace into a11 values(2,4);　　此时 select * 只有 2 条记录

原因：

replace 时，发现主键列 a 已经有 2，有重复，会先删除记录(2,3)；

再次执行 insert into a11 发现唯一索引列 b，已经有记录 4，则会删除记录(3,4)

最后再执行插入(2,4) 最终 select * 只剩下 2 条记录

replace 好处：反复执行的结果是一致的

重复执行时是幂等的

5. 存储过程

命令行下，需要先使用 delimiter // （修改默认的结束符";" 为"//"）

```
create procedure proc_n(in/out a type) // in 可以省略，但是 out 不能省略
begin
    // 过程体
end;
```

// 创建临时表 MySQL5.7 创建以后，有个文件 ibtmp1，用于存储临时表，实际的表空间保存在 tmpdir 中，即 *.frm 文件

create temporary table a (a int);

// 跟 order by 时的临时表是不一样的

1.insert ->

2.尽量缩小 select 的范围之后再 join 其它表

3.不要对大范围的结果集进行 order by

4.存储过程（缺点：不支持一些分布式数据库中间件，MySQL 对存储过程的支持不好，导致存储过程的性能不高）

5.自定义函数

1：必须要有返回值

2：每条记录都会执行一次函数。每一行记录都会执行一次 func 函数[select * from table where column = func(val)]

6.触发器

```
create trigger tg_name
before[after] insert[update,delete,replace] on tb_name
for each row
begin
    trigger_body
end;
```

NEW.col 更新以后的列值

OLD.clo 更新以前的列值（只读）

注意：replace 会触发 delete 触发器和 insert 触发器

DML 数据操作语句(select,insert,update,delete)

DDL 数据定义语句(drop[truncate] table)

DCL 数据控制语句(grant)

5.6-：同一个类型的触发器只有 1 个

5.7：允许多个同一类型的触发器

触发器

触发器格式

```
create trigger tg_name
before[after] insert[update,delete,replace] on tb_name
for each row
begin
    trigger_body
end;
```

NEW.col 更新以后的列值
OLD.clo 更新以前的列值（只读）

注意：replace 会触发 delete 触发器和 insert 触发器

DML 数据操作语句(select,insert,update,delete)
DDL 数据定义语句(drop[truncate] table)
DCL 数据控制语句(grant)

5.6-：同一个类型的触发器只有 1 个
5.7：允许多个同一类型的触发器
一条 SQL 和触发器在整个一个事务之中

好处：可以封装逻辑（比如：记录修改日志等）

触发器高级应用—物化视图

视图是一张虚表，每次使用时，都需要计算一遍虚表
创建视图：create view v_name as select * from tb join ** on **;

物化视图（MySQL 中不存在的，是 Oracle 的概念。MySQL 中可以用虚表加触发器来模拟实现）
主要用来做统计

示例：

```
create table order(
  id primary key auto_increment,
  prod varchar(32),
  price decimal(8,2),
  cnt int
)
```

```
CREATE TABLE `orders_mv` (
  `prod_name` varchar(32) NOT NULL,
  `price_sum` decimal(8,2) NOT NULL,
  `amount_sum` int(11) NOT NULL,
  `price_avg` float NOT NULL,
  `orders_cnt` int(11) NOT NULL,
  UNIQUE KEY `prod_name` (`prod_name`)
)
```

场景：

从 MySQL 抽取数据到 Oracle（MySQL 和 Oracle 的数据格式不一样）

从 MySQL 拉 200 张表到 Oracle 中存 200 张表，同时在 Oracle 中建立 40 个物化视图，可以很简单的进行数据导入。

重要的概念：ETL（Extract Transform Load）

2016-07-30 存储过程、存储函数、游标

triggers 存放位置：information_schema 库的 routines 表
mysql 删除用户时，不会删除该用户已经定义的
删除用户时，注意该用户创建的级联对象。比如该用户创建的存储过程等，需要为其它用户添加访问的权限，否则都无法访问。

游标

例子：
create table orders(empId int,ordermonth date,qty int)

empid	ordermonth	thismonth	total	avg
1	1996-07	121	121	121.00
1	1996-08	247	368	184.00
1	1996-09	255	623	207.67
1	1996-10	143	766	191.50
1	1996-11	318	1084	216.80
1	1996-12	536	1620	270.00
...	...	...		
2	1996-07	50	50	50.00
2	1996-08-	94	144	72.00
2	1996-09	137	281	93.67
2	1996-10-	248	529	132.25
2	1996-11	237	766	153.20
...	...	...		

```
select
    a.empId,a.ordermonth,a.qty as thismonth
    sum(b.qty) as total,
    cast(avg(b.qty) as decimal(5,2)) as avg
from orders a
inner join orders b
    on a.empId = b.empId
```

2016-08-02 存储引擎 1

官方：
MyISAM
InnoDB 建议只用这个
Memory
Federated
CSV
Archive
第三方
TokuDB
InfoBright 列式存储引擎

Spider

Oracle 已经不再对其它存储引擎进行开发，只开发 InnoDB

MyISAM

会创建 3 个文件
.frm　表空间文件
.myd　数据文件
.myi　索引文件

命令：mysqlfrm　*.frm
修复错误数据表

MyISAM 调优：
参数：key_buffer_size 只缓存索引，不缓存数据。
MyISAM 自己不管理内存，由操作系统控制内存分配
Oracle，InnoDB，SQLServer 等都是由数据库来控制内存分配

并且 /etc/my.cnf 中参数 skip-myisam 参数无效。MyISAM 存储引擎无法被跳过

Memory

可以使用 Redis 代替

表锁设计，全内存的存储引擎，数据库启动后数据丢失，不支持事务

支持哈希索引

Memory 存储引擎无法禁用：需要使用到临时表时，MySQL 默认会使用 Memory 存储引

擎创建表

调优参数：max_heap_table_size = 16M 临时表可以使用的内存的大小

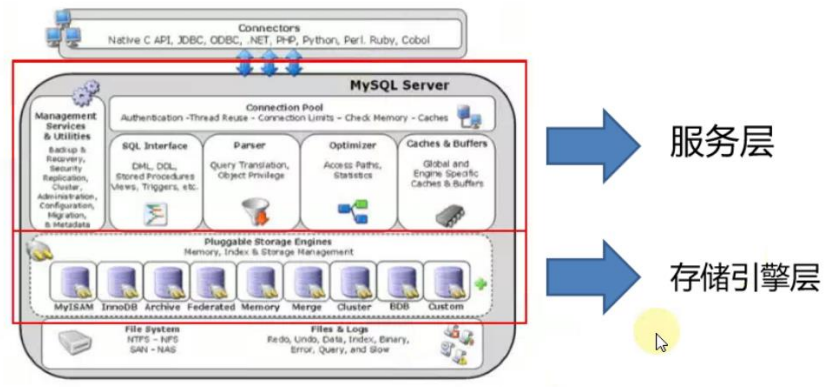
CSV

无法被禁用
基于逗号“，”进行分割。
例如：mysql 库的 slow_log 表。

```
            Table: slow_log
Create Table: CREATE TABLE `slow_log` (
  `start_time` timestamp(6) NOT NULL DEFAULT CURRENT_TIMESTAMP(6) ON UPDATE CURRENT_TIMESTAMP(6),
  `user_host` mediumtext NOT NULL,
  `query_time` time(6) NOT NULL,
  `lock_time` time(6) NOT NULL,
  `rows_sent` int(11) NOT NULL,
  `rows_examined` int(11) NOT NULL,
  `db` varchar(512) NOT NULL,
  `last_insert_id` int(11) NOT NULL,
  `insert_id` int(11) NOT NULL,
  `server_id` int(10) unsigned NOT NULL,
  `sql_text` mediumblob NOT NULL,
  `thread_id` bigint(21) unsigned NOT NULL
) ENGINE=CSV DEFAULT CHARSET=utf8 COMMENT='Slow log'
```

作业：
不建议使用 InnoDB 以为的存储引擎。
怎么查询哪些表是其它存储引擎（所有的非 InnoDB 表）？

2016-08-04 存储引擎 2



上层：SQL 解析，SQL 优化等
下层：存储引擎层（大部分情况只使用 InnoDB）

Federated 存储引擎

网游类的公司用的较多：跨实例之间访问但是量较少

- 允许本地访问远程 MySQL 数据库表中的数据
- 本地不存储任何数据文件
- 类似于 Oracle 中的透明网关
  仅支持 MySQL=>MySQL，不支持异构数据库（MySQL !=> Oracle）
- 默认不开启
  启用方式：[mysqld]federated

不启用存储引擎：skip-[存储引擎名]

存储引擎是面向表的

```
create table t1(
      a int,
      b varchar(10)
)   engine = federated
connection = 'mysql://username:password@hostname:port/database/tablename'
```

connection 格式:**schema://user_name[:password]@host_name[:port]/db_name/tb_name**

是以表为单位，只能以表名结尾。

获取数据库中所有非 InnoDB 存储引擎的表：

```
SELECT table_schema,table_name,engine FROM TABLES
WHERE engine <> 'InnoDB' AND
table_schema NOT IN
('mysql','information_schema','performance_schema');
```

修改存储引擎：alter table db.t1 engine = innodb

MySQL 体系结构

数据库：文件

数据库实例：进程

使用进程来操作文件

架构模式
MySQL：单进程多线程（类似 SQL Server）
Oracle：多进程

MySQL 中 Database 等同于 Schema
create database t1 == create schema t1

instance ->db = schema->table -> view

配置文件
datadir （数据文件目录）=

1. cd datadir/
2. mkdir db1 // 创建数据库 db1
3. chown –R mysql:mysql db1/ 授权，否则 MySQL 无法创建表

cd /datadir/tb1
tb1.ibd // 数据文件
tb1.frm // 表结构定义文件，每张表对应一个

MySQL 配置文件 my.cnf
MySQL 配置文件默认的执行顺序可用命令 mysqld –help –v | grep my.cnf 查看
MySQL 的配置文件遵循参数覆盖原则

```
!include /home/mydir/my.cnf       # 可以包含其它文件中的配置
[client]

[mysqld]
log_error = /home/mysql/log/error.log        # MySQL 错误文件
# MySQL 错误文件级别 MySQL 5.7 已经废弃
# 0：errors；1：errors、warnings；2：all
log_warnings = 2
[mysqld-5.7]
# MySQL 5.7 使用这个配置来指定日志级别
# 1：errors；2：errors、warnings；3：errors、warnings、notes（default）
log_error_verbosity = 3
[mysqld-5.6]    # 针对某个版本生效

[mysqldump]

[mysqladmin]
```

```
[mysqld_multi] # MySQL 多实例安装
[mysqld1]
[mysqld2]
```

mysqlfrm 恢复损坏数据

1. 根据 tb1.frm 获得建表信息
2. 根据 tb1.ibd 获得数据信息 （根据一些工具）

慢查询日志

开启慢查询日志：

slow_query_log = 1 #启用慢查询日志

slow_query_log_file = /home/mysql/log/slow.log #慢查询日志位置

long_query_time = 0.2 #慢查询时间（可以使用小数，单位是秒）超过 200 毫秒的就记录为慢查询

log_queries_not_using_indexes = 1 # 认为没有走索引的 sql 也是慢查询

min_examined_rows_limit = 100 #至少扫描 100 行的查询，才会进入慢查询日志

慢查询日志文件改名之后，需要使用如下命令：

mysql > flush logs;

2016-08-06 文件结构

慢查询日志

参数	说明	版本说明
slow_query_log	是否开启慢查询日志	
slow_query_log_file	慢查询日志文件名	
long_query_time	指定慢查询阈值	5.5 毫秒支持
min_examined_row_limit	扫描记录少于该值的SQL不记录到慢查询日志	
log-queries-not-using-indexes	将没有使用索引的SQL记录到慢查询日志	
log_throttle_queries_not_using_indexes	限制每分钟记录没有使用索引SQL语句的次数	5.6
log-slow-admin-statement	记录管理操作，如ALTER/ANALYZE TABLE	
log_output	慢查询日志的格式，{FILE|TABLE|NONE}	5.5
log_slow_slave_statements	在从服务器上开启慢查询日志	
log_timestamps	写入时区信息	5.7

slow_query 不记录获取锁的时间，只记录 sql 执行的时间

log_output：慢查询日志保存的目的地是哪里
    FILE（默认）
    TABLE
    NONE
可以同时存 2 个 写法：“TABLE,FILE”
set global log_output = ‘TABLE,FILE'

如果存表：备份的时候，需要将 slow_log 表排除掉

log_slow_admin_statements： 是否记录一些管理命令

校验表数据
check table tb1;
analyze table tb1;

通用日志（general_log）（1：开启）

可以当审计日志来使用

开启之后，性能下降明显

审计需要使用：audit 插件

MySQL 中的索引与维护

NULL != NULL

primary key (id)
unique key (name)
key idx_no_pl

什么是索引？

b+tree

二分查找法 binary search

1. 叶子节点保存了所有已经排序过的数据
2. 页内数据有序
3. 查询类似于二分查找法
4. 所有的有序是指 **逻辑有序（叶子节点有序，并且页内节点也是有序）**

root page 只有 1 层
non leaf page 可以有多层
leaf page 只有 1 层

```
create table t1 ( a int, b int ,c int, primary key(a),key idx_b(b));
select a from t1 where b =

主键 a 对应的值是：(a,b,c)
普通索引 idx_a 对应的值：(b,a)
```

页节点大小 innodb_page_size = 8k 推荐使用 16k

如何计算 non leaf page 层数（高度）？

2016-08-09 索引 1

索引介绍

对数据库表中的（一列或多列）键值进行排序的一种数据结构。

是用来提升查询效率的一种方法，但是插入性能可能就越慢。

插入慢的原因：需要对索引进行排序

可以对多个键值（列）创建组合索引。

索引类型

primary key，unique key，key（index）

MySQL 中一张表中只能有一个主键：唯一且非空

primary key 和 unique key 的区别：

- 主键：唯一且非空，并且一张表只能有一个主键
- 唯一键：唯一但可以为空，一张表可以有多个唯一键

注意：MySQL 中可以对唯一键添加多个 NULL 值，在 MySQL 中 NULL 不等于 NULL

unique key 和 unique index 是一样的。

创建规则：区分度大

比如：不建议在性别列上添加索引。区分度太低（只有 2 个值）

索引实现

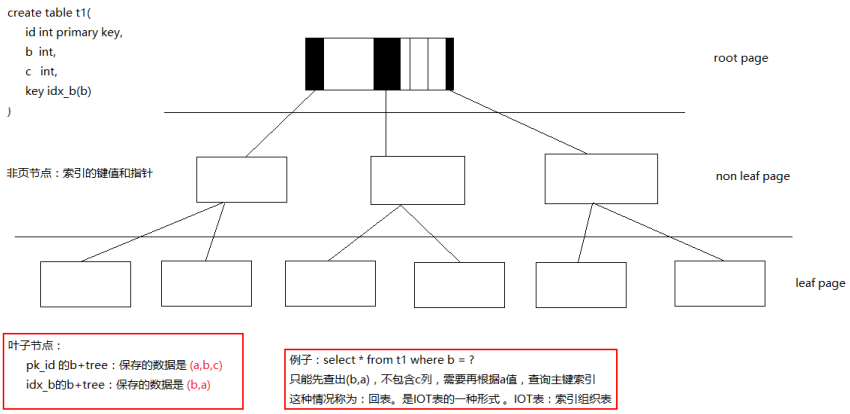
BINARY SEARCH（二分查找法）
对已经排序完的数据进行查询，速度非常快。

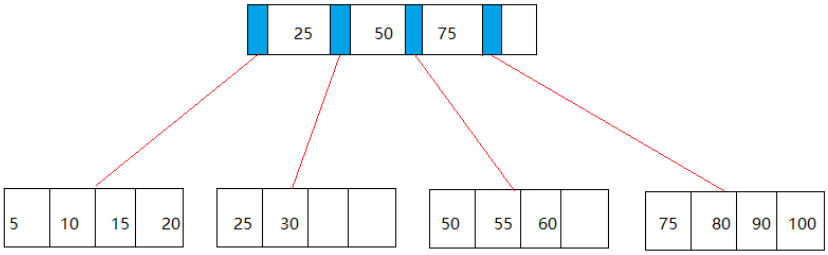
B+ Tree1

有几个索引就有几颗 B+ Tree
所有的 b+tree 都是面向于页的：page 或者 block
在 InnoDB 中每个页的大小是 16K，Oracle 是 8K

1. 叶子节点保存所有已经排序的数据（页与页之间）
2. 页内数据有序（页子节点内部）
3. 查找类似二分查找法
4. 所有的有序是逻辑有序（页与页和页内都死逻辑有序）

有序的原因：每个页上有个指针：next_key 指向下一个页
查找：
通过 B+Tree 索引并不能找到一个给定键值的具体行。
B+Tree 只能找到被查找数据所在的页，然后数据库把页读入到内存，再在内存中进行查找，最后得到要查找的数据。





叶子节点和非页节点

非页节点：只存放需要排序的键值和一个指针
叶子节点：存放需要排序的键值和其它的值

- 主键：pk 值以及其它所有的 column（一整行记录）
- 普通：需要排序的键值和主键值

架构

MySQL 使用**索引组织表**的架构，数据是存放在索引里面的。
即：所有数据都是存放在主键索引的叶子节点上。

IOT 表：数据保存在主键索引里。
堆表：索引和数据是分开存储的，B+tree 不记录任何数据。

索引区别

聚集索引：就算主键索引。

MySQL：表的所有数据是保存在主键索引的**叶子节点**上

二级索引（普通索引即非聚集索引）的叶子节点存放的是键值和主键值。

二级索引：通过二级索引查找不在本索引上的列的值时，需要根据主键再进行一次查找，这种查找被称为书签查找，也被称为回表。

书签查找：先找到主键值，再进行一次查询

B+ Tree2

在完全不创建索引的情况下，MySQL 会隐式的创建自增主键。

B+ tree 高度为 1 的时候：root 节点就是叶子节点

聚集索引：主键索引（包含了主键值和其它所有列的值）
非聚集索引（二级索引）：非主键索引（只包含索引的键值和主键值）

高度计算-聚集索引

一个页大小：16K（配置的，默认是 16K）
一行记录的平均大小：400 字节
（怎么查看一行记录的平均大小：show table status like ‘tb_name’中的 Avg_row_length。计算方式： 一个页的大小/一个页中的总记录数 **page / total**）
一个页能存放多少记录？大约 16K/400 = 40 条

非页节点保存的索引的键值和指针
非页节点的大小：16K

mysql 中的一个 pointer（指针）大小固定为：6 字节

假设一个索引的键值是 bigint =>8 字节
那么一个非页节点能存放多少条（key,pointer）？ => 16K/(8+6) = 1000

高度为 1 时：叶子节点[leaf page]就是根节点[root page]

B+ tree 高度为 1，那么能存放多少条记录？ => 40（40）
B+ tree 高度为 2，那么能存放多少条记录？ => 40*1000（4W）
B+ tree 高度为 3，那么能存放多少条记录？ => 40*1000*1000（4000W）
B+ tree 高度为 3，那么能存放多少条记录？ => 40*1000*1000*1000（400E）

1000W 的查询速度和 4000W 的是一样的

I/O 次数计算

高度越低，IO 次数

页就算最小的 IO 单位

SELECT * FROM t WHERE PK = ?

当 B+ tree 高度为 3 => 需要找 3 个页（page）=> 需要 3 次 IO

普通硬盘 => 100 IOPS => 0.04 sec
SSD 硬盘 => 10000 IOPS => 0.0004 sec

通过索引来定位记录

查询方式

- 点查询（point select）【PK，UK，KEY】
  SELECT * FROM t1 WHERE PK = ? SSD 的时间：0.0004 sec – 0.04 之间
- 范围查询（range select）
  - BETWEEN ** AND **，key >= ? and key <= ?
  - SELECT * FROM t1 WHERE key > ? and key < ?

  主要看查询的条数：通过 range 定位边界是很快的，但如果条数很多，则需要更多的时间，因为记录中保存的 next_pointer 是指针，指向下一个页。即：**链表**
- 表查询（scan select）

高度计算-二级索引

```
CREATE TABLE t1 (
    id bigint,
    name varchar(32),
    mobile varchar(32),
```

```
    primary key (id),
    key idx_name_mobile(name,mobile)
)
```

二级索引叶子节点：(key,PK)，非页节点：(key，pointer)

假设一个二级索引(name,mobile) => 64 字节

一个非页节点能存放多少条记录？ => 16K/(64 + 6) = 200 条

B+ tree 高度为 1，那么能存放多少条记录？ => 16K/(64 + 8) = 200 （索引值和主键值）
B+ tree 高度为 2，那么能存放多少条记录？ =>200 * 16K / (64 + 6) = 4w
B+ tree 高度为 3，那么能存放多少条记录？ =>200 * 200 * 200 = 800W
B+ tree 高度为 4，那么能存放多少条记录？ =>200 * 800W = 16E

根据二级索引进行查询：select * from t1 where key = ?
当 B+Tree 高度为 3 时：需要 3 + 3 = 6 次 IO
select * from t1 where (key >= ? and key <=?) => IO 次数*2 【每一条记录都需要回表】

什么时候二级索引的 IO 不会乘以 2？
索引覆盖到所需要的列时：select pk from t1 where key = ?

堆表与索引组织表的区别

索引组织表：IOT 表
堆表：heap 表

索引组织表：适合互联网

	索引组织表	堆表
查询速度	pk 比 key 查询速度快	pk 和 key 查询速度一样快
更新记录大小	一条记录更新，如果 PK 值不更新，则二级索引不需要更新	一条记录更新，可能所有的记录都需要更新 优化：行迁移【每个页专门预留部分空行】
range 查询	range、order by 较快【顺序查询】	
二级索引的查询	需要额外的一次回表	不需要回表

例如：

```
CREATE TABLE t1(
    a int,b int,c int,d int,
    primary key (a),
    key idx_b(b)
)
```

update c ,d 时，索引 b 是不需要更新的

索引表

主要涉及到表拆分
示例：

```
CREATE TABLE user_info(
    userId bigint auto_increment,
    username varchar(32),
    registdate datetime,
    email varchar(50),
    primary key (userId),
    unique key uk_name(username),
    key idx_registdate(registdate)
);
```

这一张表可以拆分成以下 3 张表：

```
create table UserInfo(
    userId bigint auto_increment,
    username varchar(32),
    registdate datetime,
    email varchar(50),
    primary key(userId)
)

create table idx_username_constraint(
    username varchar(32),
    primary key(username)
)
create table idx_username(
    userId bigint not null,
    username varchar(32),
    primary key(username,userId)
)
create table idx_registdate(
    userId bigint not null,
```

```
    registdate datetime,
    primary key(userId,registdate)
)
```

当 innodb_page_size = 4K 的时候

当一行记录平均大小为：400 字节
一页能存放多少条记录？4K/400 = 10 rows/page

当主键是 bigint 时，主键占用 8 个字节

那么一个非页节点能存放多少个（key,pointer）？ 4K/(8+6) = 250 条

B+Tree 高度为 1，能存放多少条记录？ => 10
B+Tree 高度为 2，能存放多少条记录？ => 10 * 250 = 2500
B+Tree 高度为 3，能存放多少条记录？ => 2 * 250 * 250 =12.5 万
B+Tree 高度为 4，能存放多少条记录？ => 2 * 250 * 250 * 250 = 3125 万

当一行记录平均大小为：2K 字节时

B+Tree 高度为 1，能存放多少条记录？ => 2
B+Tree 高度为 2，能存放多少条记录？ => 2 * 250 = 500
B+Tree 高度为 3，能存放多少条记录？ => 10 * 250 * 250 = 625000 = 62.5 万
B+Tree 高度为 4，能存放多少条记录？ => 10 * 250 * 250 * 250 = 156250000 = 1.6 亿

B+Tree Split（分裂）

分裂原理：
todo

在 InnoDB 中，每次分裂的代价非常大，会用一个 X-Lock（排它锁），锁住整棵树。此时，整棵树都无法访问。
所有操作都是串行，需要等待树分裂完成。
S-Lock：共享锁。

这里的锁是 Latch，会永远等待，没有超时，对内存中的树进行保护。
Lock 是对记录进行锁，而 Latch 是对内存中的对象（B+Tree）进行锁。

数据库中存在 Latch 和 Lock
Latch 会一直阻塞，并且没有超时

Lock 会一直阻塞，但是有超时

分裂时 MVCC 也无效，因为 MVCC 是基于 Lock 的

MySQL 架构是：Updating In Place，因此不能使用 CopyOnWrite

MySQL 性能关键点：B+Tree 分裂

如果 DML 操作，特别是更新或者插入比较频繁的话，一旦 B+Tree 分裂，则并发能力非常弱。

MySQL 5.7 解决了 B+Tree 更新问题，原理是什么？

数据库没有一个最优的方案，即没有最优的模式。

当 B+tree 分裂时，会对整个索引树加一把排他锁，无法绕过。

页越小对索引的影响越大。

当更新数据很大时，并发性能很差

当单行记录很大的时候，可以适当调整每页大小（不要太小）

淘宝单条记录大小是 1.5K，单页大小是 16K。

对数据库调优，不要有固定的模式，需要根据行业来进行。

没有讲 B+Tree 的操作。。。需要结合 Explain 使用

索引维护

添加索引的时候，一般不会直接使用 alter table add index ***
alter table 的大部分操作已经支持 online ddl
小于 20 万记录：alter table
大于 20 万记录：pt online schema change

在线表结构修改

MySQL 大部分 alter table 已经支持 online ddl

alter table tb1 add index idx_a(a,b)

pt-online-schema-change

软件：*percona-toolkit*

pt-online-schema-change –alter " add column b " D = db1 , t = tb1 –S /tmp/mysql.sock –execute

原理（查看通用日志或查看源码）：

查看慢查询日志后得到：

- create table tb1_new like tb1;
- create triggers
  1. create trigger del after delete on tb1 for each row delete from tb1_new where tb1_new.id = OLD.id
  2. create trigger upd after update on tb1 for each row replace into tb1_new values(new.id,new.**,new.**)
  3. create trigger ins after insert on tb1 for each row replace into tb1_new values(new.id,new.*,new.*)
- insert into tb1_new select * from tb1 where id >=3 and id <=1002 lock in share mode

索引碎片问题。。。

2016-08-11 索引 2

复合索引

How to use B+ Tree Index？

Cardinality（基数）

在选择性高的列上创建索引

不是一个精确的值，是通过抽样算法，计算出来的

The count of none unique record。记录的是某列中不重复的记录的总和
High selectivity。高选择性（区分度越高越好）

查看表中的索引（两张方案）：

```
1.  USE dbt3;SHOW INDEX FROM `lineitem`;

2.  SELECT * FROM information_schema.STATISTICS WHERE TABLE_SCHEMA = 'dbt3'
AND TABLE_NAME = 'lineitem';
```

如何求出哪个索引的选择度是比较低的？

information_schema
如何查询不合理的索引？
information_schema.statistics;

```
SELECT
    CONCAT(
        s.TABLE_SCHEMA,
        '.',
        s.TABLE_NAME
    ) 名字,
    t.TABLE_ROWS  总行数,
    s.CARDINALITY  区分列行数,
    IFNULL(s.CARDINALITY, 0) / t.TABLE_ROWS  区分度
FROM
    information_schema.STATISTICS s
JOIN information_schema.`TABLES` t ON s.TABLE_SCHEMA = t.TABLE_SCHEMA
AND s.TABLE_NAME = t.TABLE_NAME
AND s.TABLE_SCHEMA NOT IN ('mysql', 'sys')
```

Compound Index（复合索引）

多个列组成的索引

例子 1：
index idx_a_b(a,b) on tb1
能用到：
select * from tb1 where a = ?
select * from tb1 where a = ? and b = ? （idx_a_b(a,b)和 idx_b_a(b,a)都可以）
不能用到：
select * from tb1 where b = ?
排序：
select * from tb1 where a = ? order by b（只能建 idx_a,b(a,b)的复合索引）
此时已经对(a,b)排过序，不需要再对 b 进行排序一次

Index Coverage

例子 2：
index idx_a_b_c(a,b,c) on tb2
select * from tb2 where a = ? and c = ? order by b //可以用到索引，并且不需要再进行排序。

原理：在索引中(a,b,c)中 a,b,c 列已经排过序了，其中(a,b)列也已经排过序了
1. 先执行 a = ?
2. 再执行 order by b
3. 最后执行 c = ?

例子记录如下：
a,b,c,d,e
(1,2,8,1,1),
(1,3,4,2,3),
(1,3,6,1,1),
(1,4,1,2,3),
(1,4,2,2,3)
但是如果 order by c 的话，还需要再进行一次排序

注意： 二级索引是包含主键的，建立二级索引的时候，最好把主键显示的带进去，并且放到二级索引的最后。

只要不需要回表的查询，都可以称为索引覆盖(Covering index)。

创建索引 idx_a_b_c(A,B,C)时，实际是按照 A、B、C 升序排列的。
只要排序的方向不一致，都会出现 **filesort**
例如：

```
select * from tb where a = ? order by b desc,c asc
select * from tb where a = ? order by b asc,c desc
```

都会出现 filesort

MySQL5.7 如何解决呢？

使用函数索引，创建虚拟列
只能是数字

```
create table t1(
  id int,
  a varchar(32),
  b int,
  b1 int as (-b) virtual,
  c int,
  c1 int as (-c) virtual
)
```

函数索引【MySQL5.7】

alter table tb1 add column as (function(column)) **virtual/stored**

虚拟列
virtual 不占用任何索引空间
stored 占用索引空间

函数索引：在虚拟列上建立的索引

Index With Included Column

例子：

```
create table tb1(
    id int primary key auto_increment,
    name varchar(32),
    email varchar(32),
    mobile int,
    unique key uk_name(name)
)
```

索引表

索引倾斜

select * from tb where status = 1

SQL JOIN - 算法

nested_loop join

simple nested_loop join

通常来说即：笛卡尔积，复杂度：**$O(n^2)$ 永远不会使用**

index nested_loop join

基于索引的潜逃查询，推荐
select * from tb1, tb2 where tb1.x = tb2.y
通常来说：如果 tb1 作为驱动表时，优化器将直接扫描 tb2 上的索引 y，然后根据 tb1.x 来进行过滤。

对于 inner join：扫描成本就是外表行数，因此需要驱动表越小，优化器就倾向于选择小表作为驱动表
如：select * from tb1,tb2 where tb1.x = tb2.y;
如果 tb1 有 10w 条记录，tb2 有 100w 条记录，
则优化器倾向于选择小表（tb1）作为驱动表。

即：索引通常需要创建在被驱动表上

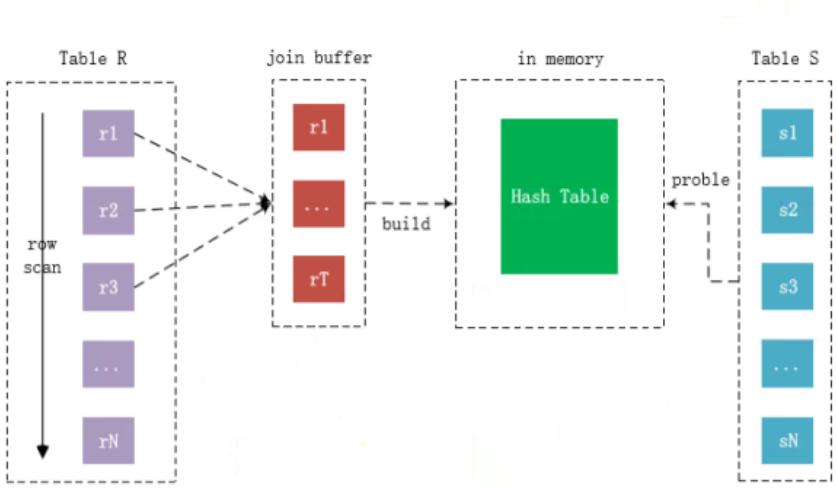
block nested_loop join（都没有使用到索引时，才会用到）

基于块的驱动算法。
加入内存（join_buffer_size）用空间换时间
join_buffer_size：会话级别参数

```
For each tuple r in R do
        store used columns as p from R in join buffer
        For each tuple s in S do
                If p and s satisfy the join condition
                        Then output the tuple <p,s>
```

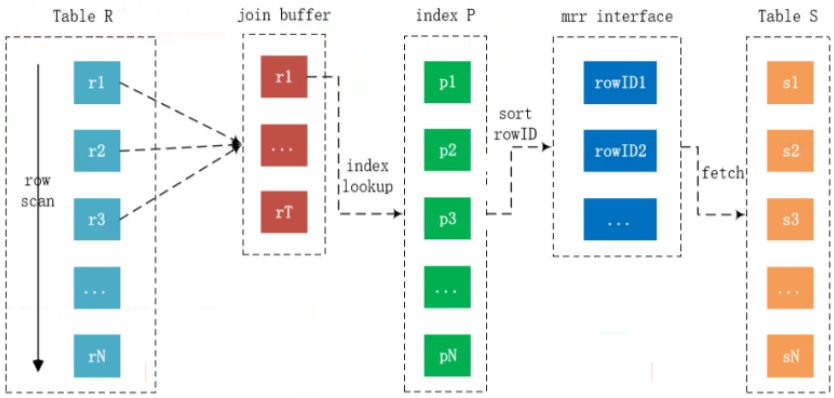
classic_hash join

只能等值查询，不需要回表，具体算法如下：



bached key access join（BKA）

用来解决：如果需要关联的列是二级索引的话，对需要回表的列进行了排序操作。
BKA 算法：



回表、不回表

假设 tb1 有 150w 条记录，其中 a 有 50w 条，且 a 列是二级索引。
select * from tb1 where a = 'abc';
需要回表时：需要 50w * 3（b+tree 高度为 3）次 IO（随机 IO）
不需要回表时（扫描聚集索引）：150w/(16k/100) = 1w 次左右（顺序 IO）

怎么优化呢？（MRR）

MySQL 优化器是最差的部分，并且不支持 hash join。

Explain 命令

大部分情况下：
id 不同的，从下往上看
id 相同的，从上往下看
（id 相等，通常来说是外部表和内部表进行关联）

Column	JSON Name	Meaning
id	select_id	The SELECT identifier
select_type	None	The SELECT type
table	table_name	The table for the output row
partitions	partitions	The matching partitions
type	access_type	The join type
possible_keys	possible_keys	The possible indexes to choose
key	key	The index actually chosen
key_len	key_length	The length of the chosen key
ref	ref	The columns compared to the index
rows	rows	Estimate of rows to be examined
filtered	filtered	Percentage of rows filtered by table condition
Extra	None	Additional information

Explain输出

列	含义
id	执行计划的id标志
select_type	SELECT的类型
table	输出记录的表
partitions	符合的分区，[PARTITIONS]
type	JOIN的类型
possible_keys	优化器可能使用到的索引
key	优化器实际选择的索引
key_len	使用索引的字节长度
ref	进行比较的索引列
rows	优化器预估的记录数量
filtered	根据条件过滤得到记录的百分比 [EXTENDED]
Extra	额外的显示选项

```
mysql > explain partitions select * from
employees where emp_no=20000\G
*************************** 1. row
***************************
           id: 1
  select_type: SIMPLE
        table: employees
   partitions: NULL
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 1
        Extra: NULL
1 row in set (0.00 sec)
```

MySQL 的 IN 子查询会自动去重（通过物化表实现，新增一个表，表中的字段添加唯一索引）
IN 的子查询改写成 join：
select * from tb1 where tb1.a in (select tb2.b from tb2)
select tb1.* from tb1,(select distinct tb2.b from tb2) tb22 where tb1.a = tb22.b

显示 SQL 语句的执行计划

语法：
EXPLAIN | DESCRIBE | DESC
[explain_type]
explainable_stmt

```
explain_type:{
        partitions |
        extended |
        format = {
                json |
                traditional
        }
}

explainable_stmt:{

}
```

MySQL 的优化是基于 CBO 的

查看上一次查询的成本：show status like 'last_query_cost';

query_cost:查询成本


```
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "1154002.20"
    },
    "table": {
      "table_name": "lineitem",
      "access_type": "ALL",
      "rows_examined_per_scan": 5509211,
      "rows_produced_per_join": 5509211,
      "filtered": "100.00",
      "cost_info": {
        "read_cost": "52160.00",
        "eval_cost": "1101842.20",
        "prefix_cost": "1154002.20",
        "data_read_per_join": "756M"
      },
      "used_columns": [
        "l_orderkey",
        "l_partkey",
        "l_suppkey",
        "l_linenumber",
        "l_quantity",
        "l_extendedprice",
        "l_discount",
        "l_tax",
        "l_returnflag",
        "l_linestatus",
        "l_shipDATE",
        "l_commitDATE",
        "l_receiptDATE",
        "l_shipinstruct",
        "l_shipmode",
        "l_comment"
      ]
    }
  }
}
```

select_type:
simple:没有使用聚合 union 和 sub query 子查询

full table scan 其实也是索引扫描（pk 扫描），聚集索引。
全表扫描不一定比走索引慢

2016-08-13 Explain

cardinality 计算

如何找到表中最不合适的索引？

一般：cardinality / table_rows < = 10%

函数：group_concat

```
SELECT CONCAT(s.TABLE_SCHEMA,'.',s.table_name) AS TABLE_NAME,
index_name,
GROUP_CONCAT(CARDINALITY ORDER BY SEQ_IN_INDEX) cardinality,
MIN(IFNULL(CARDINALITY/table_rows,0)) SELECTIVITY
FROM STATISTICS s,
TABLES t
WHERE s.TABLE_NAME = t.TABLE_NAME
AND s.table_schema = t.table_schema AND t.table_rows >= 100
AND s.table_schema NOT IN
('mysql','performance_schema','information_schema','sys')
GROUP BY s.table_schema,TABLE_NAME,index_name
ORDER BY SELECTIVITY ;
```

```
SELECT
    CONCAT(t.TABLE_SCHEMA,'.',t.TABLE_NAME) table_name,INDEX_NAME, CARDINALITY,
    TABLE_ROWS, CARDINALITY/TABLE_ROWS AS SELECTIVITY
FROM
    information_schema.TABLES t,
 (
   SELECT table_schema,table_name,index_name,cardinality
   FROM information_schema.STATISTICS
   WHERE seq_in_index=1
```

```
 ) s
WHERE
      t.table_schema = s.table_schema
           AND t.table_name = s.table_name AND t.table_rows != 0
           AND                t.table_schema                NOT                IN
( 'mysql','performance_schema','information_schema','sys');
```

全文索引

select * from tb where name like '%Tom%';

B+Tree 为什么不能前缀匹配呢？
关键字没有整个排序

全文索引：会对整个列进行分词，然后把各个分词保存到 B+Tree 树索引中

alter table add fulltext index idx_name(name)

坏处：
1.一张表只能添加一个全文索引
2.不支持 online ddl

select * from tb where match(title,body) against ('Tomcat' in natural language mode);

select * from tb where match(title,body) against ('Tomcat' in boolean mode);

地理位置索引

Explain 返回列

通常来说：执行计划的 id 越大越先执行，id 相同时，自上往下关联

特例如 union：执行计划的 id 越小越先执行

select * from a join b 和 select * from b join a 有什么区别？

```
SELECT * FROM part WHERE p_partkey IN
( SELECT l_partkey FROM lineitem
WHERE l_shipdate BETWEEN '1997-01-01' AND '1997-01-10' )
ORDER BY p_retailprice DESC LIMIT 10;
```

```
SELECT * FROM part a,
( SELECT distinct l_partkey FROM lineitem
WHERE l_shipdate BETWEEN '1997-01-01' AND '1997-02-01' ) b
WHERE a.p_partkey = b.l_partkey
ORDER BY p_retailprice DESC LIMIT 10;
```

通常来说：小表做驱动表

SQL 优化--JOIN 算法

最简单的算法：双层 for 循环

MySQL 只支持：index nested_loop join
具体算法：

```
for each row r in R do
    lookup r in S index
        if found s == r
            then output the tuple<r,s>
```

Explain 输出-select_type

select_type Value	JSON Name	Meaning
SIMPLE	None	Simple SELECT (not using UNION or subqueries)
PRIMARY	None	Outermost SELECT
UNION	None	Second or later SELECT statement in a UNION
DEPENDENT UNION	dependent (true)	Second or later SELECT statement in a UNION, dependent on outer query
UNION RESULT	union_result	Result of a UNION.
SUBQUERY	None	First SELECT in subquery
DEPENDENT SUBQUERY	dependent (true)	First SELECT in subquery, dependent on outer query
DERIVED	None	Derived table SELECT (subquery in FROM clause)
MATERIALIZED	materialized_from_subquery	Materialized subquery
UNCACHEABLE SUBQUERY	cacheable (false)	A subquery for which the result cannot be cached and must be re-evaluated for each row of the outer query
UNCACHEABLE UNION	cacheable (false)	The second or later select in a UNION that belongs to an uncacheable subquery (see UNCACHEABLE SUBQUERY)

simple

简单查询，一般是单表

union

联合查询：
select * from t1 union select * from t2;

dependency subquery

依赖子查询

```
selct
	emp_no,
	dept_no,
	(select count(1) from dept_emp t2 where t1.emp_no = t2.emp_no) as row_num
from
	dept_emp t1;
```

exists 大部分都是 dependency subquery

materialized

重要：in 会自动去重

derived

Explain 输出-type

const：主键扫描
eq_ref：唯一索引进行关联
ref：普通索引关联
fulltext：全文索引
ref_or_null：和 ref 类似
range：范围查询【between ** and **】
index：二级索引
all：全表扫描

Explain 输出-key_len

索引使用的长度
计算 key_len：
1. 主键：int => 4，bigint => 8
2. 复合索引
   - 可以为 null 时，需要 1 个字节记录 null 的标记值，not null 时，不需要记录。
   - 字符集不同时，需要根据不同的字符集计算。如 varchar(10)：10 表示可以存 10 个字符，如果字符集是 utf8，则需要 10 * 3，如果字符集是 utf8mb4，则需要 10 * 4

explain select * from employees order by birth_date,emp_no limit 1000,30
优化器开关：**optimizer_switch**

二级索引中包含了主键索引
MySQL5.5 版本及以前的版本中，默认是不包含的
MySQL5.6 版本及后续的版本中，默认都包含了
查看方式：show variables like 'optimizer_switch'\G
开关：use_index_extensions=on

行号问题

使用如下方式
select @a:=@a+1 as row_num,emp_no from employees,(select @a:=0) a;
不是使用子查询

如果根据 key 需要返回的数据很多，超过了全表数据的 80%，则优化器倾向于使用聚集索引进行查询（主键索引）

主键：顺序扫描
key：随机扫描（还需要回表）

顺序扫描比随机扫描快 100 倍以上。

Covering Index 索引覆盖

Index Coverage

idx_a_b(a,b) 是复合索引

select ***count(1)*** from buy_log where buy_date < '2011-02-01' and buy_date >= '2011-01-01'

二级索引中包含索引
优化器倾向于使用小表作为驱动表
当返回数据过多时，优化器倾向使用全表扫描

```
SELECT CONCAT(s.TABLE_SCHEMA,'.',s.table_name) AS TABLE_NAME,
index_name,
GROUP_CONCAT(CARDINALITY ORDER BY SEQ_IN_INDEX) cardinality,
MIN(IFNULL(CARDINALITY/table_rows,0)) SELECTIVITY
FROM STATISTICS s,
TABLES t
WHERE s.TABLE_NAME = t.TABLE_NAME
AND s.table_schema = t.table_schema AND t.table_rows >= 100
AND s.table_schema NOT IN
( 'mysql','performance_schema','information_schema','sys')
GROUP BY s.table_schema,TABLE_NAME,index_name
ORDER BY SELECTIVITY ;
```

2016-08-16 系统监控

MySQL 不是一个 CPU 密集型的应用

常用命令

一般使用下面的命令进行监控

top => CPU

iostat => IO

vmstat =>

sar =>

ps =>

dstat =>

系统层监控

CPU 性能指标

CPU 长时间大于 80%的话，说明 CPU 是瓶颈

top

查看 CPU 的运行情况

max_connection

哪种异常情况使 MySQL 实例所有的 CPU 到达 100%？=> 阻塞
例子：select * from tb where a = ？ 这条 SQL 不走索引，并且执行时间非常慢。
CPU 达到 100%，同时伴随着 max_connections 全部用完。

解决：
第一时间查看 slow.log，修改 SQL。
修改 max_connection 无效（只是暂时有效）。set global max_connection = 4000

正确 MySQL 的 CPU 100% 处理的顺序：

1. 关闭应用
2. KILL 掉 MySQL 进程
3. 为该查询添加索引
4. 重启应用

如果**只重启 MySQL，然后再添加索引**容易出现如下问题
waiting for table metadata lock（元数据锁）

原因：

1. 线程 1 执行 SQL（select * from t1 where a = ?）
   线程 2 执行 alter table t1 add index idx_a(a)
2. 线程 2 会等待线程 1 执行，造成 waiting for table metadata lock

online ddl

其它线程可读可写，但是当其它线程在进行读写的时候，不能执行 online ddl

online ddl：

执行 ddl 操作的时候，可以进行读写操作
执行读写操作的时候，不能进行 online ddl

只有某 1 个 MySQL 实例的 CPU 达到 100% ？
某个 CPU 上正在运行大 SQL

目前为止：一个 connection 只运行在 1 个 CPU 上

如何查看哪个线程使用的 CPU 最多？

top –H 看到的是全局的信息

MySQL 是单进程多线程的架构

top 命令返回值含义

%us 用户进程的 cpu 开销
%sy 内核使用的 cpu 的开销
%wa 等待 IO 操作所需要的 CPU 时间

%id CPU 空闲的百分比
%ni 花费在改变进程的执行顺序和优先级的 CPU 百分比

通常来说：$us > $sy

硬盘性能指标

iostat

每个扇区的大小 512 字节

r/s：每秒完成的读 I/O 设备次数
w/s：每秒完成的写 I/O 设备次数
rsec/s：每秒读取的扇区数。
wsec/s：每秒写的扇区数
rkB/s：
wkB/s：
avgqu-sz：平均 IO 队列的长度【这个是真正的瓶颈】
    HDD：2~3
    SSD ：20~3
await：平均每次设备 I/O 操作的等待时间（毫秒）
svctm：平均每次设备 I/O 操作的服务时间（毫秒）
%util：一秒中有多少时间用于 I/O 操作，即被 I/O 消耗的 CPU 的百分比

- disk属性值说明：
  - rrqm/s: 每秒进行 merge 的读操作数目。即 rmerge/s
  - wrqm/s: 每秒进行 merge 的写操作数目。即 wmerge/s
  - r/s: 每秒完成的读 I/O 设备次数。即 rio/s
  - w/s: 每秒完成的写 I/O 设备次数。即 wio/s
  - rsec/s: 每秒读扇区数。即 rsect/s
  - wsec/s: 每秒写扇区数。即 wsect/s
  - rkB/s: 每秒读K字节数。是 rsect/s 的一半，因为每扇区大小为512字节。
  - wkB/s: 每秒写K字节数。是 wsect/s 的一半。
  - avgrq-sz: 平均每次设备I/O操作的数据大小 (扇区)。
  - avgqu-sz: 平均I/O队列长度。
    - HDD：2~3
    - SSD：20~30
  - await: 平均每次设备I/O操作的等待时间 (毫秒)。
  - svctm: 平均每次设备I/O操作的服务时间 (毫秒)。
  - %util: 一秒中有百分之多少的时间用于 I/O 操作，即被io消耗的cpu百分比

使用系统库：performance_schema
表：threads

MySQL 5.6 的 performance_schema 库中的表 threads 中，没有列 **THREAD_OS_ID**

I/O 使用率到达 100%，但是并没有用户连接到 MySQL 服务器，为什么？
解决：

1. 使用 iotop 查看是否是 MySQL 用户导致？
2. 如果是 MySQL
   - 可以通过 performance_schema. threads. **THREAD_OS_ID** 查看具体的进程号【MySQL 5.7】
   - 通过 pstack 查看
3. 也可能是 io_ibuf_thread 线程在做合并

关键是结合`performance_schema`.`threads`

MySQL 层监控

命令：show global status 一般用来采集数据

查看负载情况

查看 MySQL 每秒钟的 select 次数
mysqladmin extended-status –r –i 1
| grep “Com_select|Com_insert|Com_delete|Com_update”

2016-08-18 磁盘

介绍

IOPS：I/O PER SECOND
1 次 IO 的大小：16K
访问方式：顺序访问和随机访问
磁盘分类：HDD（机械硬盘）、SSD（固态硬盘）

HDD

类似人，是有损耗的。
转速就代表性能
常见转速：5400 转/分钟、7200 转/分钟、10000 转/分钟、15000 转/分钟
随机访问最大 IOPS：250 次/S
顺序访问：100M/S
随机访问：16K * 250 = 4M/S
顺序比随机访问快的原理：随机访问每次都需要重新寻址，而顺序访问只需要第一次寻址

优化器倾向于使用聚集索引全表扫描而不使用二级索引的原因：
顺序访问比随机访问速度快。
单色需要满足要求：
1. 查询的是二级索引
2. 返回的数据量大于 20%

提升 IOPS 性能

1. 通过 RAID 技术

功耗非常高
RAID0、RAID1、RAID5、RAID10
磁盘阵列

2. 通过购买共享存储设备
3. 但是提升非常有限

SSD

磁盘调度算法：deadline（推荐）或者 noop

参数：
innodb_flush_neighbors = 0　　脏页刷新算法（默认只刷新脏页，不刷新脏页所在的区）
innodb_log_file_size = 8G　　磁盘性能越高可以设置越大（重做日志大小）

未来当磁盘不再是瓶颈的时候，数据存储算法可能是瓶颈（如：不再是 B+Tree 等）

一般只选择 **英特尔 SSD**

硬盘界的特斯拉

Flash 闪存 + 中央控制器（类似 CPU）

用电设备
由 Flash Memory 组成
没有读写磁头
IOPS 高
50000+ IOPS
读写速度非对称

选择

- PCIE or SATA/SAS
- SATA/SAS 一遇安装和升级
- SATA/SAS 与 PCIE 的性能差距逐渐减小
- PCIE 的性能很少有应用可以完全使用
- 优先选择 SATA/SAS 接口的 SSD

品牌

- Intel
- FusionIO（希捷）
- 宝存

操作系统和文件系统

文件系统

xfs/ext4
noatime
nobarrier

操作系统

mount –o noatime,nobarrier /dev/sdb1/data　提升 5%

noatime：不记录更新时间
nobarrier：只写入 cache，不实际写入 SSD

磁盘调度算法：deadline
echo deadline > /sys/block/sda/queue/scheduler 提升 30%

查看磁盘性能
iostat -xm 10　每 10s 刷新一次信息

合并读	合并写	读	写	读带宽	写带宽	每秒读块大小	平均队列深度	服务时间
rrqm/s	wrqm/s	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	svctm

数据库层面

磁盘调度算法设置：

- deadline 或 noop

InnoDB 存储引擎参数设置：

- innodb_flush_neighbors = 0
- innodb_log_file_size = 8G（至少 4G）

希望数据库的状态：平稳

块设备存储

DataBase（数据库系统）：16K
FileSystem（文件系统）：4K
Disk（扇区）：512B

2016-08-23 基准测试

文件操作 api
c 语言
fread
fwrite

// 从文件 filename 的 offset 位置开始读取 length 个字节
f = fopen("filename",offset,length)

fread(filename,offset,length)
fwrite(filename,offset,length,data)

fwrite 表示写到 pageCache,并没有写到文件对应的句柄上。
必须显示调用
fsync(filename) 才能刷新到磁盘

数据库：offset 都是 16K*N，length 都是 16K

direct io

innodb_flush_method = o_direct // 并不会提升速度

fsync(file)将会更新数据的 metadata

设置 O_DIRECT，则数据库的内存完全由数据库服务器来控制，不由操作系统控制。

sysbench

sysbeanch –test=fileio help

磁盘层面的读写比例
数据库层面的读写比例

IOPS 计算
每秒：28MB/s　=> 28 * 1024K/16K(每页大小) = 2000IOPS

sysbench

ldconfig　/usr/local/mysql/lib/

准备阶段：
sysbench --test=fileio --file-num=4 --file-block-size=16384 --file-total-size=1G prepare
测试截断：
sysbench --test=fileio --file-num=4 --file-block-size=16384 --file-total-size=1G --file-test-mode=**rndrd** --file-extra-flags=direct --max-time=300 --max-requests=0 --num-threads=16 --report-interval=1 run
清除文件：
sysbench clean

参数：--rw-ratio=3:2 读写比例

file-test-mode
随机读：rndrd
顺序读：seqrd
随机写：rndwr
顺序写：seqwr

磁盘刷新算法：
innodb_flush_method=o_direct
表示每次写，直接写文件，不写操作系统缓存，没有必要做 fsync 了。

2016-08-25 数据库基准测试

QPS & TPS

query per second
transaction per second

QPS

服务器查询总次数：show global status like 'Questions';
服务器启动到现在的时间：show global status like 'Uptime';
QPS = Questions/ Uptime

TPS

数据库监控系统

1. 备份
2. 热表
3. SQL 排行榜（执行次数最多的 SQL）
4. 慢 SQL

测试工具

sysbench

偏向互联网，主要是根据主键查询。

官方使用

tpcc

国际标准的测试方法，测试电商的，压力比较大，会有锁机制

专门针对联机交易处理系统（OLTP 系统）的规范

主要包含：
- 新订单：
- 支付：
- 发货：

ldconfig /usr/local/mysql/lib/

mysqlslap

秒杀优化
线程池
thread_handling = pool-of-thread

2016-08-27 测试与备份

备份类型

热备

在线备份
对应用无影响

冷备

备份数据文件
需要停机
备份 datadir 文件下下所有文件

温备

在线备份
只能读，不能写

MySQL 热备工具

所有的热备只能使用 InnoDB 存储引擎
其它存储引擎只能是冷备

ibbackup
xtrabackup
mysqldump

物理备份（备份文件）、逻辑备份（备份数据库内容）

逻辑备份：导出的结果都是 sql 语句

物理备份

在讲 InnoDB 时，再讲

逻辑备份

MySQLdump 恢复与备份
mysqldump [options] --single-transaction database [tables]

mysqldump [options] --single-transaction --databases [options]DB1[DB2 DB3]
mysqldump [options] --single-transaction --all-database

single-transaction：保证数据一致性，所有的备份数据，都是备份时间点开始时的数据

备份时，可读可写。只有很小的锁，可以忽略不计。

增量备份是通过二进制文件来实现的，不是命令

不推荐使用 source 命令导入 SQL 文件，会输出很多结果。source 使用远程连接
推荐使用小于号（<） mysql < test.sql

mysqlpump
mysqlpump 支持并行备份
支持显示进度条，参数：--watch-progress
解决了逻辑备份（mysqldump）导出比较慢的问题

导入还是单线程

并行复制参数：--default-parallelism= 10（开启 10 个线程进行复制）

mydumper

mydumper 导入和导出都比较快

2016-08-30 二进制日志

参数：
sync_binlog=1
innodb_support_xa=1
同时开启这两个参数来保证二进制日志和 InnoDB 存储引擎数据文件的同步

导入和导出

MySQL 逻辑备份工具：mysqldump mysqlpump mydumper
一定要加上参数：single-transaction
一致性的备份：
MySQL 备份不锁表

mysqldump -w 备份一张表，并且可以带条件

table t1(a int,b varchar(31))

mysqldump db1 t1 -w “a>3”

mydumper
-t --thread 多线程导出（即使只有 1 张表，也可以多线程导出）
mysqlpump -t 也可以多线程导出，但是只能基于表

导出的数据：
表结构 SQL 和数据 SQL 分开
官方的 mysqlpump 只有一个文件

--trx-consistency-only 数据一致性
-D --daemon 后台进行

导入

mysqldump 导入 select * into outfile ‘a.txt’ from b; 默认存放在 test 库中

load data infile ‘a.txt’ into table b;
（不做数据解析，效率比 insert 高非常多）

复制

MySQL 复制实现：逻辑复制，Oracle 复制实现：物理复制（通常被这么称，但是其实是通过物理和逻辑复制）

	MySQL	Oracle	SQL Server
复制类型	逻辑复制	物理逻辑复制	物理逻辑复制
优点	灵活	复制速度非常快	复制速度非常快
缺点	配置不当容易出错	要求物理数据严格一致	要求物理数据严格一致

Oracle 不仅数据严格一致，而且主从服务器上数据块都严格一致
物理复制：大小一致，且块内容一致
逻辑复制：只要求逻辑结果一致即可，并不要求物理上底层的块也一致

逻辑复制
记录每次逻辑操作
主从数据库要求可以不一致

物理逻辑复制
记录每次对于数据页的操作
主从数据库严格一致

MySQL 复制基于二进制日志文件，只能写一份，不能写多份
Oracle 基于重做日志文件

二进制日志--逻辑复制的基础

复制参数：log_bin = /mysql_data/replication/bin.log
格式参数：binlog_format = row

	Statement	Row	Mixed
说明	记录操作的 SQL 语句	记录操作的每一行数据	混合模式
优点	易于理解	数据一致性高，可 flashback	结合
缺点	不支持不确定的sql语句(如uuid)	每张表一定要求主键	
线上使用	不推荐	推荐	不推荐

二进制日志内容

由各种类型的 event 组成

日志文件是追加写

show master status;

show binlog events in 'bin.000012';
每个 event 大小：pos 相减

statement 格式

statement：binlog 中会显示每一条输入的 sql

row 格式

看不到任何 sql 内容
记录的是每一行记录的变化，可以保证主从数据最终一致
只能通过 mysqlbinlog -vv bin.000001

缺点：如果列很多，则记录内容非常大
可通过参数：binlog_row_image 设置
默认值：**FULL**，记录所有的操作
MINIMAL，最小记录

是通过主键来进行操作

只记录修改的值，不记录具体的 sql

Rows_query event 建议打开
binlog_rows_query_log_events = 1 （ 0：关闭）
记录 binlog 中引起哪些行变化的 SQL 语句，但是被注释

对性能的影响，几乎可以忽略不计。

记录的是每一条记录，发生逻辑变化的内容，如果没有逻辑变化，就不会有日志产生

row 格式记录的全项，可 flashback（闪回）

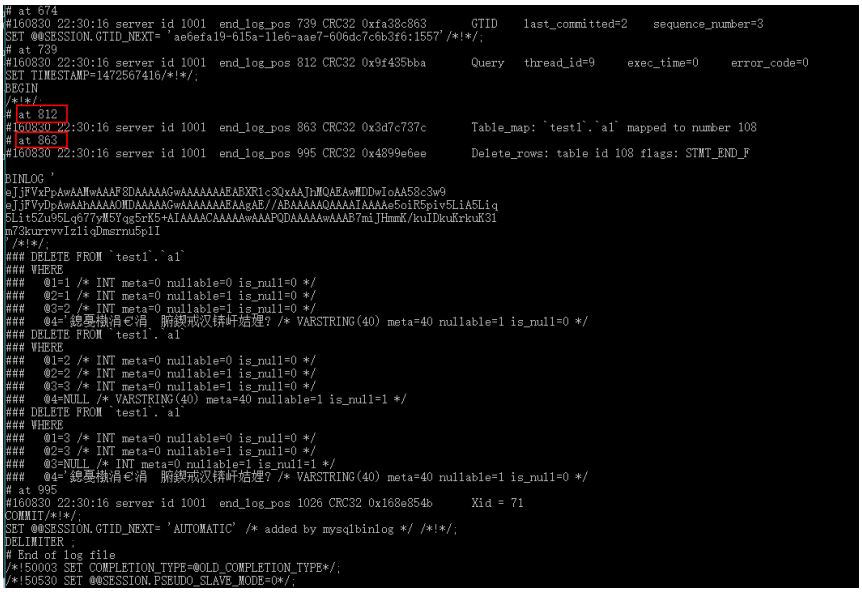
官方并没有实现：flashback 功能。

问题：怎么做增量备份？
通过 binlog
一般一周一个全备

做法：

mysqlbinlog
--start-position=12
--stop-position=123

误操作：
可使用 flashback 进行恢复



目标：备份脚本

熟悉 mysqlbinlog 命令以及其参数

2016-09-03 复制 1

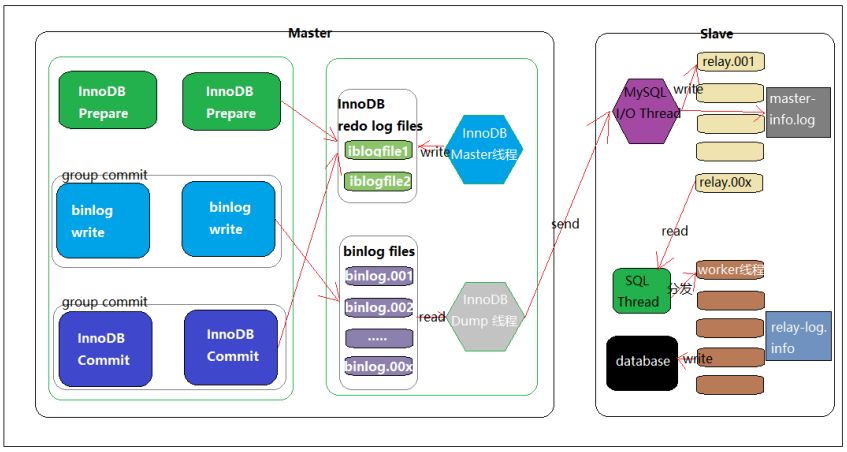
查看 binlog

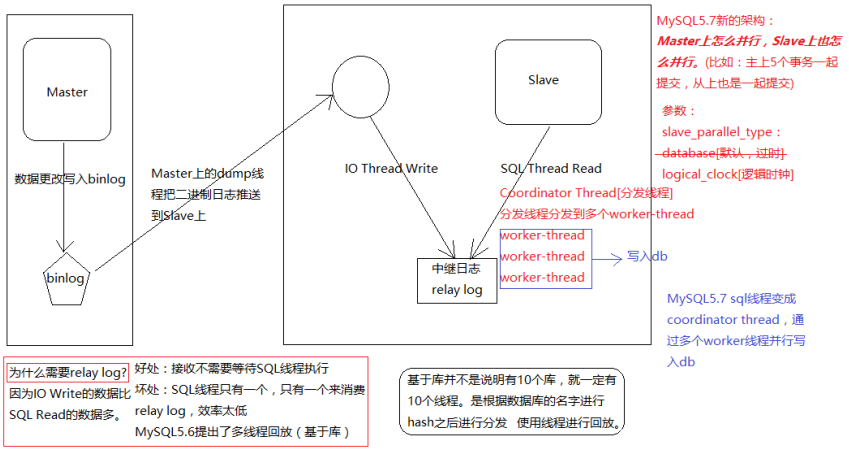
怎么找一张表，在 bin_log 日志中的操作？
mysqlbinlog bin.000003 -vv |grep "mysql" -A 2
在 bin_log 中查找针对表“mysql”的操作

查看 binlog 中的数据
mysqlbinlog bin.0000021 -vv --base64-output=decode-rows |grep “mysql”
--start-position=123
--stop-position=456
--start-datetime=12:00:00
--stop-datetime=13:00:00

MySQL 的复制基于二进制日志

复制原理





问题：master 上产生的日志是多个线程同时写日志，但是 slave 上只有一个 sql 线程来消费日志。

5.6 版本：提供了多线程的回放操作，但是是基于库的多线程。比如：有 3 个库，则只有 3 个线程在同时回放。
但是也不是一对一的

有多少个线程在回放：show variables like ‘slave_parallel_workers’;

5.7 版本：根据 master 上怎么并行，slave 上也怎么并行。
show variables like ‘slave_parallel_type’;

set global slave_parallel_type = ‘**logical_clock**’
复制延时从 8 小时，减少到 0

set global slave_parallel_workers = 0
set global slave_parallel_workers = 1

这两个都表示是单线程，但是有区别。

复制搭建

官方备份：mysqldump

1. server-id master 和 slave 的 server-id 不能重复

2. create user 'rpl'@'%' identified by '123';【创建复制用户】
3. grant replication slave on *.* to 'rpl'@'%'【创建复制权限】
4. 在 slave 上需要添加：set global read_only = 1【只读】

备份方法：
mysqldump --single-transaction --master-data=1 -B db1 > db1.sql
db1.sql 中注意：
change master to log_file='bin.000034',master_log_pos=1234;

切换命令：
change master to mster_log_file='bin.12345',master_log_pos=1235,
master_host='192.168.1.101',master_user='repl',master_password='repl';

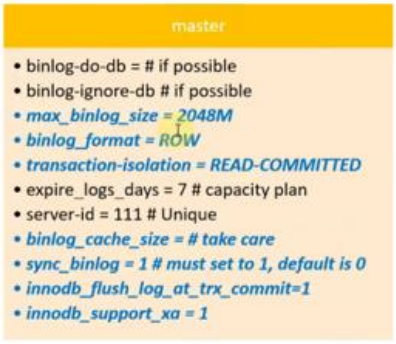
mysql-5.7 参数： super_read_only = 1
5.7 在做 master、slave 切换时，需要同时关闭 read_only 和 super_read_only

级联复制
master - slave -slave
缺点：延时变大（一次操作经过两次网络请求）
优点：跨机房，跨数据中心的场景下应用比较多

通常来说：mysql 都是内网 ip，当跨机房复制时，需要有外网 ip

5.7 版本之前有 bug，如果 slave 有很多的话，会导致 master 性能下降

2016-09-06 复制 2



relay-log-recovery = 1 # I/O thread crash safe 【不管数据库怎么 down，复制不会出错】

- 正常情况下 master、slave 数据一致
- master crash，然后 master 重启成功
- slave crash,然后 slave 重启成功

relay-info.log

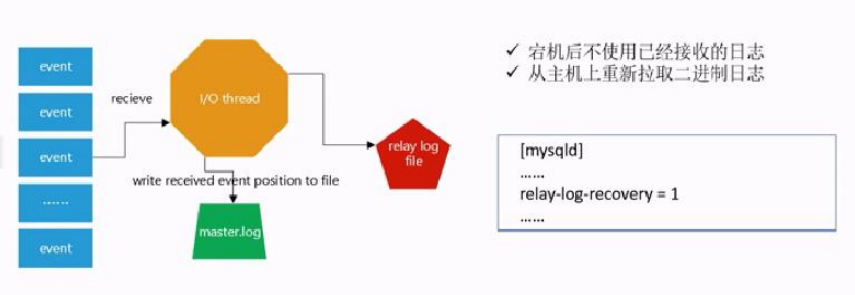
SQL 线程高可用

如何解决 SQL 线程的 thread crash safe？
通过添加到数据表

[mysqld-5.6]
relay_log_info_repository = table

I/O 线程高可用

参数：sync_master_info



如何解决 I/O 线程的 thread crash safe ?

所有的 mysql 复制都是以 event 为依据的。

二进制日志有各种各样的 event 组成。
查看 events：show binlog events limit 5;

[mysqld]
relay_log_recovery = 1

GTID

组成

- Global Transaction ID
- uuid + transactionId

show variables like 'server_uuid';
server_uuid 保存在 datadir 文件夹下的 auto.cnf 文件中

auto.cnf

```
[auto]
server-uuid=ae6efa19-615a-11e6-aae7-606dc7c6b3f6
```

注意：
复制数据库文件时，需要删除这个文件。

意义

1. 全局唯一
2. 更容易的进行 failover 操作
3. slave 可以直接连上新的 master

配置

2016-09-08 InnoDB

InnoDB PHYSICAL STORAGE

表空间

Leaf node segment
Non-Leaf node segment
Rollback segment

表空间-区

区是最小的空间申请单位
区的大小固定是 1M
16K 64 个页（1024/16）
8k 128 个页（1024/8 ）

4K 256 个页（1024/4 ）
通常一次申请 4 个区的大小
B+Tree 的叶子节点和非叶节点存储在不同的 segment

表空间-页

页
- 是最小的 I/O 单位

普通用户表
- 默认每个页 16K
- --innodb_page_size(from 5.6)
- 只能基于数据库（Oracle 可以基于表来设置 page_size）

压缩表
- 基于页的压缩
- 每个表的页大小可以不同

启用压缩：row_format=compressed key_block_size=8

通常来说压缩到一半（8K）就可以了。

ALTER TABLE tb1 ROW_FORMAT=COMPRESSED, KEY_BLOCK_SIZE=4

KEY_BLOCK_SIZE=4 表示压缩的页大小为 4K
启用压缩：实际内存中还是 16K 的数据，InnoDB 对数据进行压缩，然后刷进磁盘，存储时，按照每个页 4K 进行存储。

压缩变快的原因：每次 I/O 传输的大小变小。从 16K 的传输降低到 4K 的传输。

对数据进行存取时：都会先在内存中进行解压和压缩

压缩是否有好处？
压缩比例并不是越小越好

透明页压缩（目前只有两种压缩算法，mysql5.7 开始）
create table tb1(a int) compression = “lz4”
create table tb2(a int) compression = “zlib”

锁超时和死锁是两个概念。

2016-09-10 InnoDB 页压缩与物理存储结构

表空间-页

透明页压缩：不需要关心如何设置页的大小

使用的是文件系统的压缩

原理：每个页依然是 16K，每次写这个页的时候，做了压缩，比如从 16k 压缩到 8k，剩余的空间填充 8k 个 0，写文件系统时，调用文件系统的**空洞特性**，实际只传输 8k 的数据。

即：对每个页（16K）进行压缩，然后对压缩后没有使用到的空间使用 0 来填充

空洞特性：PUNCH HOLE
create table bb(a int) compression='lz4';

InnoDB: **Punch hole not supported by the file system** or the tablespace page size is not large enough. Compression disabled

目前只有编辑安装才支持。

```
2016-09-08T12:20:13.560933-05:00 0 [Note] /usr/local/mysql/bin/mysqld (mysqld 5.7.13-log) starting as process 3962 ...
2016-09-08T12:20:13.564665-05:00 0 [Note] InnoDB: PUNCH HOLE support not available
2016-09-08T12:20:13.564686-05:00 0 [Note] InnoDB: Mutexes and rw_locks use GCC atomic builtins
2016-09-08T12:20:13.564689-05:00 0 [Note] InnoDB: Uses event mutexes
2016-09-08T12:20:13.564691-05:00 0 [Note] InnoDB: GCC builtin __sync_synchronize() is used for memory barrier
2016-09-08T12:20:13.564694-05:00 0 [Note] InnoDB: Compressed tables use zlib 1.2.3
2016-09-08T12:20:13.564696-05:00 0 [Note] InnoDB: Using Linux native AIO
```

测试工具
电商：tpcc
社交：linkbench

怎么判断页压缩的压缩比？
读取*.ibd 文件，随机读取几个页的数据，压缩后跟之前的页大小进行对比

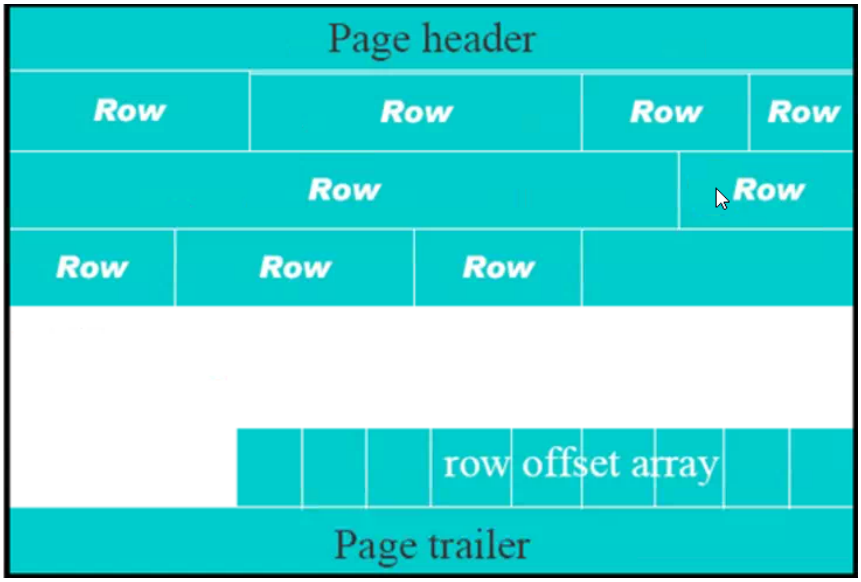
information_schema.INNODB_CMP;
这张表是用来查看旧的压缩算法的效果。

information_schema.INNODB_CMP_PER_INDEX
参数：innodb_cmp_per_index_enabled，可以查看每张表的压缩信息。

表空间-记录

记录格式参数：ROW_FORMAT
COMPACT（5.7 以前默认）
COMPRESSED（支持压缩）
DYNAMIC（5.7 以后默认）

5.7 新的参数：innodb_default_row_format



Page Header：

前一个页的指针（prev）

后一个页的指针（next）

校验码

Page Trailer：

校验码

当两个校验码一致，则认为这页的数据是干净的、完整的。

所有的记录都存放在 Row 中。

B+树索引只能快速定位到**记录所在的页**

row offset array 记录的是排序后的记录的 row_id 的指针

select * from tb1 where pk = 10000

(1) 查找到这条记录所在的页

(2) 使用二分查找法，从 row offset array 中查到该条记录

- InnoDB存储引擎是索引组织表
  - Index Organized Table
  - 与Oracle IOT表类似
  - 适合OLTP应用
- 叶子节点存放所有数据
  - 索引即数据
  - 数据即索引
- 聚集索引记录存在以下系统列
  - rowid： B+树索引键值
  - trx id： 6字节
  - roll pointer： 7字节

自动创建 6 个字节的主键列，并且是自增的
每条记录最多多 3 个列：

- rowid：6 字节（B+树索引键值【如果没有主键列，则会自动创建自增主键】）
- trx id ：6 字节 （事务 id）
- rollback pointer：7 字节 （回滚指针）

trx id 和 rollback pointer 是为了事务而实现的事务 id 以及回滚指针

二级索引只有：键值和主键值
聚集索引：所有数据和事务 id 和回滚指针

Compact 记录格式：
变长字段列表+NULL 标志位+record header +col1+col2+…

标记记录被删除：将 record header 中的标记位置为删除

```
create table tb1(
    t1 varchar(10),
    t2 varchar(10),
    t3 char(10),
    t4 varchar(10)
) engine=innodb row_format=compact
```

同一个数据库中：所有的没有创建主键的列都使用同一个 ROW_ID
即：多张表使用同一个 ROW_ID 列

CHAR 与 VARCHAR 的区别？

- CHAR 最大到 255 个字符，VARCHAR 最大到 16K-1（16383）个字符
- CHAR 定长，VARCHAR 变长

Dynamic 记录格式
优化大对象记录的存储

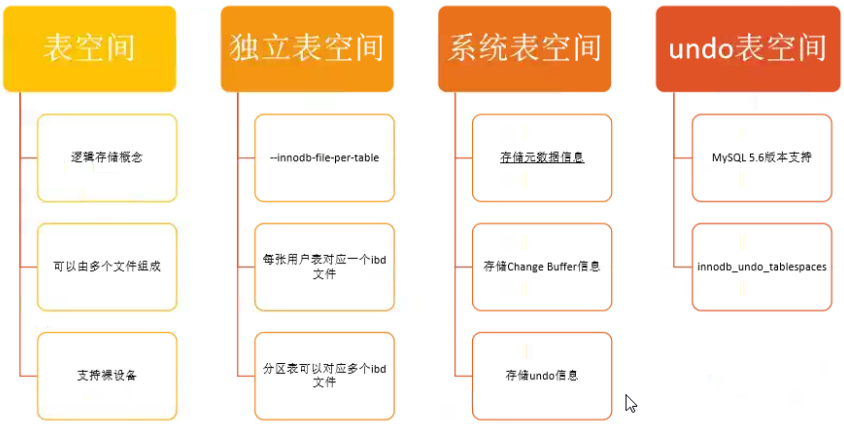
原则：
一个页能存放的记录越多，则性能越优

InnoDB 存储引擎表空间文件

全局共享的表空间：**ibdata1**
参数：innodb_file_per_table【是否是每个表一个表空间】

压缩：compressed

InnoDB 存储引擎文件--表空间



2016-09-13 InnoDB 内存体系结构

后台线程

即：InnoDB Background Thread

查询线程信息：
select * from performance_schema.threads;

- IO Thread
1. Insert buffer thread
2. Log thread
3. Read thread
   发起异步读，只有在预读的时候，才需要使用。
4. Write thread
   异步写线程，建议增大。
- Master Thread
- Latch monitor thread
- Purge thread【MySQL5.5+】回收 undo

随机预读和顺序预读：
innodb_random_read_ahead
innodb_read_ahead_threshold

thread/innodb/io_ibuf_thread
thread/innodb/io_log_thread

thread/innodb/buf_dump_thread dump LRU 列表中的数据到磁盘的线程
thread/innodb/page_cleaner_thread 刷新脏页的线程数
    包含 4 个子线程：thread/innodb/srv_worker_thread

thread/innodb/ srv_master_thread 刷新重做日志

page_cleaner_thread ：脏页刷新线程，最终也会调用 IO Thread。所有的脏页写入都是 IO_Write_Thread 执行，并且是异步的
5.6 只有 1 个
5.7 可以设置为多个。参数：**innodb_page_cleaners** = 12

脏页刷新根据 innodb_io_capacity 参数来决定。
每秒刷新 innodb_io_capacity * 5%

每 10 秒刷新 innodb_io_capacity 个这么多

刷新上线：innodb_io_capacity_max

IO_THREAD

innodb_read_io_thread

发起异步读，只有在预读的时候，才需要使用。

innodb_write_io_thread

InnoDB 中所有脏页的写入都使用 innodb_write_io_thread，所有脏页的写入都是异步的。

innodb_write_io_threads 建议调大。

select 顺序
1.先从 buffer_pool 中读取
2.再从磁盘读取
3.放入 buffer_pool

如果页被修改了，则需要刷回磁盘，此时使用的是异步写。

通常情况下：
innodb_purge_threads 回收 undo，建议调大
innodb_page_cleanners 脏页回收，建议调大
io_write_thread 异步写线程，建议调大

MASTER_THREAD

master_thread 作用：**刷新脏页**，**刷新重做日志**（InnoDB 独有的重做日志）
内部有多个循环组成：main_loop background_loop flush_loop suspend_loop 等
main_loop 主要功能：分为每秒和每 10 秒的操作
每 1 秒的操作：

- 日志刷新到磁盘，即使这个事务还没有提交（总是刷新 redo log buffer）
- 合并插入缓存（可能）判断：如果每秒钟的 IO 操作小于 5 次，则发生
- 至多刷新 100 个缓冲池中的脏页到磁盘（可能）由 **innodb_max_dirty_pages_pct** 判断
- 如果没有用户活动，则切回 background_loop（可能）

每 10 秒的操作：

- 刷新 100 个脏页（可能）判断过去 10 秒内的 IO 操作是否超过 200 次
- 合并至多 5 个插入缓冲（总是）
- 将日志刷新到磁盘（总是）
- 删除无用的 undo 页（总是）判断当前事务系统中被删除的行是否可以删除，如果可以，则立即删除（对表的 update、delete 操作，原先的行只是被标记为删除）
- 刷新 100 个或 10%个脏页到磁盘（总是）判断缓冲池中脏页数量是否超过 70%，如果超过，则刷新 100 个，如果小于 70%，则只需刷新 10%的脏页到磁盘

刷新脏页

参数 **innodb_io_capacity**

innodb_io_capacity：决定脏页刷新吞吐量。
查看线上数据库的读写比例 IOPS，根据比例计算。

刷新脏页的原则

每 1 秒钟刷新 **5% * innodb_io_capacity** 个脏页，
每 10 秒刷新 **innodb_io_capacity** 个脏页。

默认值：200（在 HDD 硬盘下）
如果是 SSD，则需要调大

MySQL5.6 开始刷新脏页已经在使用 page_cleaner_thread 中刷新，
master_thread 只用来刷新重做日志。

如果脏页达到 innodb_max_dirty_pages_pct，则不管是 1 秒还是 10 秒，则全量刷新一次。
默认值是 75%。即：如果脏页数量达到 75%，则全量刷新。

innodb_io_capacity 可以在线调。

刷新重做日志

- Master Thread 每一秒将重做日志刷新到重做日志文件
- 每个事务提交时将会重做日志刷新到重做日志文件
- 当重做日志缓冲池剩余空间小于 1/2 时，重做日志缓冲刷新到重做日志文件

每 1 秒刷新重做日志到磁盘

pager less

Purge Thread

刷新未使用的 undo page
事务被提交之后，其使用的 undo log 可能不再需要，因此，使用 Pruge Thread 来回收已经使用并分配的 undo 页。

以此来减轻 Master Thread 的压力.

参数:innodb_purge_threads = 4

Page Cleaner Thread

专门用来刷新脏页到磁盘，减轻 Master Thread 的压力。

查看线程状态

命令：iotop -u mysql 查看 io 最大的 mysql 线程的 io 使用率

show engine innodb status;

使用 pager less 可以按行读取。

pending 值较大的话，说明磁盘有负载。

主线程
Process ID=12345 Main thread ID = 123424234

InnoDB Buffer Pool

缓冲池

读取：首先将从磁盘读到的页放在缓冲池中（称：将页 FIX 在缓冲池中），下一次读取相同的页时，首先判断该页是否在缓冲池中。
修改：对于数据库的修改操作，则先修改缓冲池中的页，然后以一定的频率刷新到磁盘上。这里的刷新利用 checkpoint 机制。

缓冲池中的数据：索引页、数据页、undo 页、插入缓冲、自适应哈希索引、锁信息，数据字典信息。

InnoDB 内存大小（建议：当前机器内存的 70%左右）
参数：innodb_buffer_pool_size

只要是在磁盘上，所有的数据都会被以页的形式被读入缓冲池，所以缓冲池越大越好。

当内存大于数据库时，此时相当于所有的数据都在内存中。

建议打开：log_queries_not_using_indexes 在错误日志中记录没有使用索引的 SQL

InnoDB Buffer Pool 最多的数据就是 data

默认共有 innodb_buffer_pool_size/innodb_page_size 个页

缓冲池管理-LRU List、Free List、Flush List

通过 LRU 算法来进行管理。
Buffer Pool
  Free List
  LRU List
    LRU
    unzip_LRU
  Flush List
    根据 oldest_lsn 进行排序。
  Flush List 中的页是在 LRU List 中的，Flush List 中只包含了所有的脏页

整个 Buffer Pool 的大小 = Free List + LRU List

LRU List 如何进行管理？

- 最近最少使用的算法
- midpoint LRU

默认情况下：第一次被读到的数据，被放到中间。
默认情况下，需要被放到 midpoint 上等待 1 秒中。
参数：innodb_old_blocks_time = 1000
--innodb_old_blocks_pct
--innodb_old_blocks_time

information_schema.INNODB_BUFFER_PAGE_LRU
space：每张表的 id，page_number：某个页

统计热点数据和热点索引：在 sys 库中
select * from schema_table_statistics;
select * from schema_index_statistics;

有多少个 buffer_pool 实例（设置为 CPU 的数量）
innodb_buffer_pool_instances 会平分整个 buffer_pool
为什么要设置多个 buffer_pool？
类似 ConcurrentHashmap 的锁拆分

当页从 LRU 列表的 OLD 部分加入到 NEW 部分时，发生 page made young，因为 innodb_old_block_time 的设置而导致页没有从 old 部分移动到 new 部分的操作称为 page not made young。

重做日志缓冲

redo log buffer

额外的内存池

在 InnoDB 存储引擎中，对内存的管理是通过一种“内存堆（heap）”的方式进行的

2016-09-20 InnoDB Buffer Pool

回顾

BUFFER_POOL 设置

innodb_buffer_pool_size 缓冲池总大小
innodb_buffer_pool_instances 一共有多少个缓冲池？buffer_pool

为什么需要多个 buffer_pool_instances？
每个 buffer_pool 都有 1 个 latch
多个 buffer_pool_instance 就会有多个 latch
打散之后，可以减少锁的竞争，提升性能。

通常设置为 CPU 的个数。

设置过大会引起过多的上下文切换。

BUFFER_POOL 组成

- FREE LIST 空闲列表
  数据库启动时，按照 buffer_pool 的总大小，按照 16K 一个页，把数据放入 FreeList 中。
- LRU LIST
- FRUSH LIST （保存脏页的指针，标记某个页是脏页）
- unzip_LRU（非 16K 的页）

为了避免查询页时扫描 LRU List，还为每个 buffer pool instance 维护了一个 page hash，通过 space id 和 page no 可以直接定位到 该 page。
一般情况下，当我们需要读入一个 page 时，首先根据 space id 和 page no 找到对应的 buffer pool instance，然后查询 page hash，如果 page hash 中没有，则标识需要从磁盘读取。在读磁盘潜首先我们为即将读入内存的数据页分配一个空闲的 block。当 Free List 上存在空闲的 block 时，可以直接从 FreeList 上摘取；如果没有就需要从 unzip_lru 中或者从 LRUList 上驱逐 page。
遵循的策略：

1. 首先尝试从 UNZIP_LRU 上驱逐解压页
2. 如果没有，再尝试从 LRU 链表上驱逐 Page
3. 如果还无法从 LRU 上获取空闲的 block，用户线程就会参与刷新脏页，尝试做一次 single page flush，单独从 LRU 上刷掉一个脏页，然后再重试。

读取一个页的时候，会从 FreeList 中读取一个页，保存到 LRU List、

如果页变脏，则会放入 FLUSH LIST 中，但是放进去的只是一个指针，标记某个页是脏页。并不真实保存脏页的数据。

lru list 中既有脏页，也有干净的页。

整个 buffer_pool_size = free list + lru list

LRU List 的管理

使用最少使用算法
使用 midpoint 机制 默认是 3/8

前 5/8 是 new，后 3/8 是 old。3/8 的位置被称为 midpoint

读取一条记录默认是放在 midpoint 位置，当数据被再次读的时候，才放进 new page 中

需要全表扫描时，可以通过如下做法，防止 LRU 列表被污染。
解决办法 1：

```
1.  set global innodb_old_blocks_time = 1000;
2.  select * from tb1; //  一些查询操作
3.  set global innodb_old_blocks_time = 0 ;
```

解决办法 2：
扫描操作放到 slave 上。

参数：**innodb_old_blocks_time** 防止 LRU LIST 被污染

buffer_pool
两张表用来查看缓冲池中页的状态。不推荐线上使用。
information_schema 库中 innodb_buffer_pool_stats、**INNODB_BUFFER_PAGE_LRU（space,pageno)**

重要的概念：SPACE、PAGE_NUMBER

SPACEE：表空间对应的 id 号。共享表空间是从 0 开始，新建的表从 1，2，3。。。开始自增。
PAGE_NUMBER：一个表空间都是以页的大小来管理的，第一个页就是 0，第二个就是 1，。。。

可以通过(space,page_number)：可以知道是哪个表的哪个页。

如何管理 unzip_LRU？

使用伙伴算法进行内存管理，例如申请 4K 的缓冲池时：
1. 检查 4KB 的 unzip_LRU 列表是否有可用的空闲页
2. 如有，则直接使用
3. 检查是否有 8KB 的 unzip_LRU 列表
4. 如有，则将该页分裂成 2 个 4KB 的页，存放到 4KB 的 unzip_LRU 列表中
5. 从 LRU 列表中申请 16KB 的页，将页分为 1 个 8KB 的页和 2 个 4KB 的页，分别放在对应的 unzip_LRU 列表中.

可以通过 information_schema.***INNODB_BUFFER_PAGE_LRU***来观察 unzip_LRU 列表中的页。

预热

使得数据库快速恢复到运行状态
- 数据库重启
- 数据库服务器宕机

启动之后，一般 buffer_pool 是空的，都不在 FREE LIST 中，都在磁盘中。

需要将数据从磁盘读取到 buffer_pool。但是速度一般较慢

预热方法

mysql5.6 以前

- select count(1) from tb force index(pk);
- select count(1) from tb force index(primary);

mysql5.6 以后，使用如下参数

innodb_buffer_pool_dump_now

innodb_buffer_pool_dump_at_shutdown（数据库关闭的时候发生一次 dump）

innodb_buffer_pool_load_now（立即装载 dump 文件）

innodb_buffer_pool_load_at_startup （启动时，装载 dump 文件）

innodb_buffer_pool_load_abort

innodb_buffer_pool_filename

dump 出来的文件，是一个文本文件，保存的是**(space,page_number)**

默认保存在 data 文件夹下 filename = ib_buffer_pool

数据库启动的时候会读取这个文件，把文件中对应的页，读取到 buffer_pool 中，启动完毕后，预热完成。

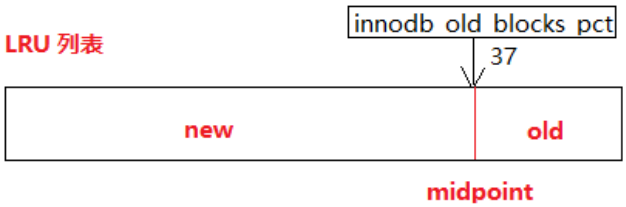
加快预热效率的方法：对文件进行排序（即：对 space 和 pageno 进行排序）

预热策略

将 LRU 链表 dump 出来

mysql5.7 参数

innodb_buffer_pool_dump_pct，参数如图：一般只需要导出 new 区的 LRU 百分比就可以。



innodb_buffer_pool_dump_at_shutdown 关闭时，讲 buffer_pool 中的热点数据 dump 出来，文件内容是（spaceNo,pageNo）；

innodb_buffer_pool_load_at_startup 启动时，加载 dump 的文件（对 dump 出来的文件进行排序，可以加快 startup 的速度）

checkpoint

原理和作用

将脏页刷新到磁盘
缩短数据库的恢复时间
LSN（log sequence number）是一个累加的值。
LSN 表示每个日志页的版本号，保存在 page header 中。

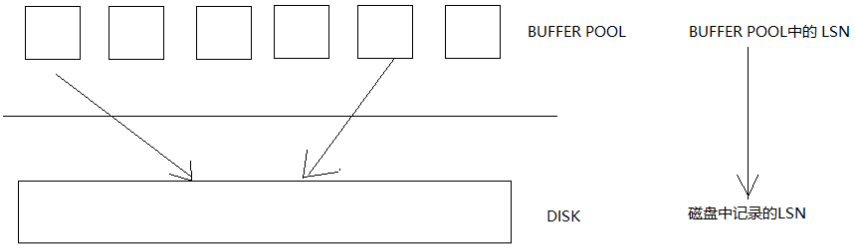
当页发生变化之后，就会有 LSN。
lsn 保存在 page header 中

当页被修改了，LSN 就会发生改变。此时会出现 buffer_pool 中的 LSN 和磁盘中的 LSN 不一致。

不一致时，会通过 checkpoint 技术，将 LSN 刷新到一致。

oldest_modification：修改时
newest_modification：刷新时

checkpoint 的作用：缩短恢复的时间。



脏页刷新指，将**buffer pool中的数据刷新到磁盘。**
刷新时，如果buffer pool中的LSN最大值是13000，当刷新到10000时，数据库宕机了，则磁盘中保存的最大LSN是10000；
恢复时只需要从buffer pool中LSN= 10000的位置开始恢复即可，大大减少了恢复的时间。

Sharp Checkpoint

innodb_fast_shutdown ｛ 1 : 0 ｝：关闭数据库的时候是否刷新脏页
1：只刷新数据的脏页，不刷新 insert buffer 的脏页。
简单的说：当数据库关闭是会刷新。
缺点：刷新时，会阻塞整个系统

Fuzzy Checkpoint

哪些场景会刷新脏页呢？

将部分脏页刷新回磁盘
每次刷新只刷新一点脏页

参数：innodb_io_capacity 每秒钟刷新多少脏页（默认是 5%）。

Master Thread Checkpoint

每 1 秒刷新 **innodb_io_capacity * 5%**个脏页，
每 10 秒刷新 **innodb_io_capacity** 个脏页

从 FLUSH_LIST【脏页列表】中刷新

LRU_LIST Checkpoint

LRU LIST 中能被淘汰的页必定是干净的页。

从 LRU 列表中刷新
为什么需要从 LRU LIST 中刷新呢？
Free List 在不断的使用过程中，会不断减少，直到为 0；此时所有的页都存在于 LRU LIST 中。

当再来一次查询时，需要从 LRU LIST 中淘汰掉一部分数据，但是如果这部分数据是脏页，那就不能淘汰，需要将这部分脏页刷新到磁盘，以便于让 LRU LIST 中的 oldest page 中全部都是干净的页，然后再淘汰掉最后的干净的页。

此时出现的情况被称为：LRU LIST 不够用

引入新的参数：**innodb_lru_scan_depth**

当 Free List 为空，并且 LRU LIST 已满，需要从 LRU LIST 中淘汰数据时，并不是刷新一个。

InnoDB 会批量查询个 **innodb_lru_scan_depth** 页，如果发现页是脏页，则把这么多个脏页全部刷新。
以保证 LRU LIST 的尾端都是干净的页。

innodb_lru_scan_depth 是基于 buffer_pool_instance，即是每个 buffer_pool_instance 的参数。

Async/Sync Flush Checkpoint

重做日志文件不可用的情况下，这时需要强制将一些页刷回磁盘，此时脏页是从 flush lru list 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
async_water_mark = 75% * total_redo_log_size
sync_water_mark = 90% * total_redo_log_size

即：innodb_log_file_size

- 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，出发 Async Flush，从 Flush 列表中刷新足够的脏页回磁盘，使得刷新后的 checkpoint_age < async_water_mark
- 当 checkpoint_age > async_water_mark 时，触发 Sync Flush，从 Flush 列表中刷新足够的脏页回磁盘，是的刷新后满足 checkpoint_age < async_water_mark

目的：保证重做日志的循环使用。在 Page Cleaner Thread 中生效。
参数：innodb_max_dirty_pages_pct 控制刷新的数量。
默认：75% 表示：当缓冲池中脏页的数量占据 75%时，强制进行 checkpoint，刷新一部分脏页到磁盘。

buffer_pool 压缩

压缩方式

compression（透明页压缩）

每个页的大小都是 16K

一个压缩的页在内存中可能有两个：压缩的页和解压的页

解压只做一次。

key_block_size

mysql 到底用了多少内存，即当发生 OOM 时，该如何处理？
需要开启 performance_schema
可以从 sys 库中查看。。。

管理

BUFFER_POOL 中既有 4k 的页，又有 8k 的页，还有 16k 的页，该如何管理呢？

伙伴分配算法。

将一个 16K 的页分配为两个 8K 的页，或者 4 个 4K 的页。

show engine innodb status\G

LRU len：未压缩的 LRU LIST 中有多少个页，即 LRU 长度
unzip_LRU：压缩的 LRU LIST 中有多少个页，即压缩的 LRU 长度

BUG

performance_schema 内存泄漏问题。

从 sys 库中查看：memory_global_by_current_bytes;
SQL：
select event_name,current_alloc from sys.memory_global_by_current_bytes limit 10;

MySQL 默认对内存的统计：只统计 performance_schema。
打开监控：
use performance_schema;
show tables like 'setup_instruments';

update setup_instruments set enabled = '**YES**';

打开的监控越多，就会占用越多的内存，对性能有影响。

2016-09-22 InnoDB 存储引擎特性

DoubleWrite

目的

数据写入的可靠性

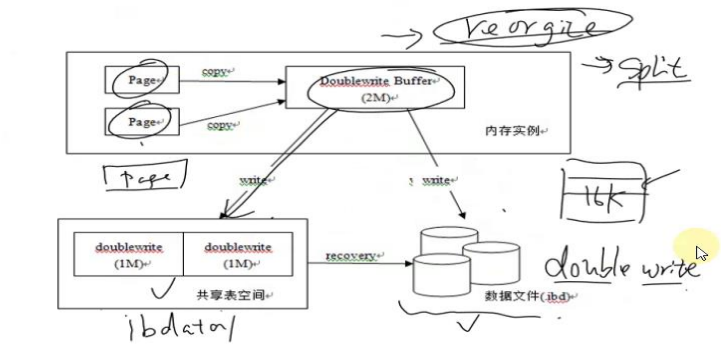
解决的问题

partial write

- 16K 只写入了 4K，6K，8K 的情况
- 不可以通过 redo log 进行恢复
- Like Oracle media corrupt

先 doublewrite 写入 ibdata1，然后再写如 *.idb 文件
doublewrite 由两部分组成：内存中的 doublewrite buffer （2MB），磁盘上共享表空间中连续的 128 个页（2 个区 extent，2MB）
在对缓冲池中的脏页进行刷新时，并不直接写磁盘，而是通过 memcpy 将脏页先复制到内存中的 doublewrite buffer 中，之后通过 doublewrite buffer 再分两次，每次 1MB 的顺序写入共享表空间的物理磁盘上，然后调用 fsync，同步磁盘。之后再将 doublewrite buffer 中的页写入各个表空间文件中。

DOUBLEWRITE



页在进行更新的时候，不是写入到表空间文件（.idb 文件）,而是写入 double write buffer 中，先把 128 个页（2MB）写入共享表空间，然后再写入对应的表空间文件。
支持原子写的磁盘，可以关闭 doublewrite
atomic write

redo 恢复页的前提是：页是干净的，通过判断 page header 和 page tailer 来判断页是否干净。

zfs 是 CopyOnWrite 写时复制
mysql 是 update in place 原地更新

show global status like 'innodb_dblwr%';
Innodb_dblwr_pages_written dbw 已写（可以用来统计生产环境中写入的量）
Innodb_dblwr_writes dbw 实写

show global status like 'Innodb_buffer_pool_pages_flushed';
当前从缓冲池中刷新到磁盘页的数量。

CopyOnWrite 不会发生 partial write 问题，因为原来的页总是干净的
double write buffer 存在于共享表空间中

Insert/Change Buffer

全局只有一颗 Insert Buffer B+树，存放在共享表空间中，即：ibdata1 中。
通过.ibd 文件恢复数据后，还需要 repaire table 操作。

- 提高辅助索引的插入（删除、修改）性能
- none unique secondary index

二级索引一般是无序的。

change buffer 参数：innodb_change_buffer_max_size
默认值：25 表示最多使用 1/4 的缓冲池内存空间，该参数最大值为 50。

实现：空间换时间
1.先判断二级索引是否在缓冲池中，如果在，则直接插入
2.先将二级索引放入到 Insert Buffer（也是一颗 b+tree，最大 2k）中

非页节点存放查询的 search_key（9 字节）：space（4 字节） 、marker、 offset（4 字节）
每张表有一个唯一的 space_id
marker：兼容老版本的 Insert Buffer
offset：页所在的偏移量

Adaptive Hash Index （自适应 Hash 索引）

记录本身是无序的，通过记录之间的指针串联起来。

通过二叉查找法在 row offset array 中查找。

只会对热点区的数据创建 hash 索引。时间复杂度是 O(1)
怎么判断是否是热点数据？
1. 索引是否被访问了 17 次
2. 索引中某个页是否被访问了至少 100 次
3. 对索引中的页访问的模式是相同的
- idx_a_b(a,b)
- where a = ?
- where a = ? and b = ?
4. 仅支持点查询

建议关掉，原因：实现过于复杂，会导致资源竞争过高。
参数：
innodb_adaptive_hash_index
innodb_adaptive_hash_index_parts 锁拆分

Async IO

主要的优势是进行 IO Merge。
如 :访问(spage,page_no)为(8,6),(8,7),(8,8)时, sync IO 需要进行 3 次 IO 操作(每次访问 16K),而 AIO 会进行 IO Merge,从(8,6)开始直接访问 48K 的页。
参数：innodb_use_native_aio

Flush neighbor page

试图刷新 当前页 所在区 中的所有脏页

对传统机械磁盘有效

ssd 建议关闭。
参数：innodb_flush_neighbors
　　0：只刷新自己的页
缺点：刷新太频繁。

double write 保证页写入的可靠性。
（实现了**原子写**或**用 CopyOnWrite 实现**的磁盘，可以关闭）
change buffer 提升二级索引的修改效率
adaptive hash index 提升点查询的性能（将热点页中的索引创建 hash 索引）
flush neighbor page 刷新当前 脏页（dirty page）时，是否刷新临界页。

checkpoint 保存在哪里呢？
保存在 ib_logfile 中

mysql 的重做日志，不会归档，Why？
归档的目的：用来做恢复
mysql 通过 binlog 来进行恢复。

什么是 checkpoint？
mysql 有个 PageCleaner Thread 来刷新脏页，
LRU 怎么刷，根据参数：innodb_lru_scan_depth 来刷新

2016-09-24 事务

Transaction 事务

A 原子性　一个事务中的所有操作，要么全部成功，要么全部失败
　　redo
C 一致性　数据的关系，不随事务的提交而改变
　　undo
I 隔离性

lock
D 持久性　事务一旦提交，就会持久保存在文件中
redo
undo

什么是一致性？
在事务过程中，数据的关系是一致的
什么是隔离性？
一个事务所做的修改，对其它事务是不可见的

开启事务语句

```
START TRANSACTION

    [transaction_characteristic [, transaction_characteristic] ...]

transaction_characteristic:
    WITH CONSISTENT SNAPSHOT
  | READ WRITE
  | READ ONLY

BEGIN [WORK]
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
SET autocommit = {0 | 1}
```

事务类型
平事务 Flat Transaction【begin[start transaction]; do sth; commit[rollback];】
带有保存点的事务
链事务
嵌套事务 Nested Transaction【不支持】
分布式事务
长事务

sysbench 一组事务包括了 18 条 SQL 语句
tpcc 分为 4 个类型

平事务

开始：begin/start;
提交：commit/rollback;

带有保存点的事务：
begin
action1;
action2;
savepoint　action1;
rollback to　action1;
other;
...
commit

链式事务

begin;
commit and chain; 这句话类似与包含：commit;begin;

分布式事务

在分布式环境下运行的扁平事务。
每个节点都要复合 ACID
比如：持卡人从招行转账 1000 到中行。

分布式事务的问题？
1.性能较差
原因：
- 操作都是串行
- 需要多发一次 prepare 请求

2.prepare 完成之后，有的 commit 成功，有的 commt 失败，怎么办呢？
这种事务被称为悬挂事务，需要人工干预，手动处理。
(1). xa start 'tx1';
(2). insert into t values();
　　delete ***
　　update ***
(3). xa end 'tx1';

(4). xa prepare 'tx1';
(5). xa commit 'tx1';

如果步骤 4 执行完成，但是数据库宕机或者重启，则事务会一直悬挂，需要人工介入解决。
重启执行之后执行 **xa recover**;

显式事务和隐式事务

显式事务：
begin;
insert/update/delete
commit;
隐式事务：
autocommit = 1 默认情况下，自动提交
insert/delete/update

事务的持久性实现

begin;
commit;

事务 commit 之后，会把日志从 log buffer 刷新到 redo log file 中，发生一次 fsync。此时只有日志落盘，页并不一定已经落盘。
事务提交之后，事务修改的页，不一定已经刷新到磁盘，页没有落盘。
如果发生宕机，InnoDB 会从 redo log file 中恢复时，通过 checkpoint 恢复到指定的 LSN。

checkpoint 保存在 log buffer 前 2k 的字节中。

如果是一个长事务，则事务的中间过程中已经慢慢的刷新到重做日志中，即在事务提交的过程中已经开始写磁盘了，在最终 commit 的时候，只落了最后一点内容。

写日志的条件是什么？
Master Thread
刷新脏页：MySQL5.7 由专门的 page cleaner 线程执行，可以为多个线程。
参数：innodb_io_capacity。
刷新日志：

- 每 1S 进行一次脏页刷新，也意味着每 1S 将内存中的 log buffer 刷新到 redo log file 中
- redo log buffer 使用大于 1/2 时刷新
- 事务提交时刷新

innodb_flush_log_at_trx_commit = {0 | 1 | 2}
默认是 1，表示每次 commit 的时候，一定写日志。
设置为 0，表示事务提交的时候，不强制写日志，日志的写入，由 master thread 控制。会丢失 1S 的数据。
设置为 2，表示

日志

InnoDB 有两种非常重要的日志：undo log 和 redo log。

undo log 用来保证事务的原子性和 InnoDB 的 MVCC

REDO LOG

redo log 用来保证事务的持久性。
redo log buffer 参数：innodb_log_buffer 通常 8MB 够了

redo log 文件以 ib_logfile[number]命名
redo log file 参数：
innodb_log_files_in_group 配置为多少个 redo log
innodb_log_file_size 每个 redo log 大小
innodb_log_group_home_dir redo 日志目录

total redo log = innodb_log_files_in_group * innodb_log_file_size

开启 binlog 之后，MySQL 的事务提交机制：
1.prepare redo log（fsync）
2.write binlog（fsync）
3.commit redo log（fsync）这一步写入操作系统缓存即可，不需要 fsync 到磁盘
recover 过程：
1.scan binlog _ > txid list -> hash table(binlogTxIdList)
2.scan redo log -> txid list (redoTxIdList)
3.判断 redoTxIdList 中的 txId 是否在 binlogTxIdList 中，若存在就 commit，否则就 rollback

UNDO LOG

InnoDB 存储引擎对 undo 的管理同样采用段的方式。rollback segment，每个回滚段记录了 1024 个 undo log segment，而在每个 undo log segment 段中进行 undo 页的申请。
innodb_undo_logs=128 个 rollback segment。
代替参数：innodb_rollback_segments
事务在 undo log segment 分配页并写入 undo log 的过程中，同样需要写入重做日志。当事

务提交时，InnoDB 存储引擎会做以下两件事儿：
1. 将 undo log 放入列表中，以供之后的 purge 操作
2. 判断 undo log 所在的页是否可以重用，若可以则分配给下一个事务使用

是否可以删除 undo log 和 undo log 所在的页，由 purge 线程来判断

show engine innodb status 中：
History list length 表示 undo log 的数量

undo log 格式

insert undo log：由 insert 操作产生的 undo log，因为 insert 操作只对事务本身可见，对其它事务不可见，故 undo log 可以在事务提交之后直接删除，不需要 purge 操作。

update undo log：由 update、delete 操作产生的 undo log。该 undo log 可能需要 MVCC 机制，因此不能在事务提交时就进行删除。提交时放入到 undo log 链表，等待 purge 线程进行最后的删除。

Purge

用于完成最终的 update、delete 操作。
一个页上允许多个 undo log 存在。
histroy list 按照事务提交的顺序存储 undo log，先提交的事务，总在尾端。
参数：**innodb_purge_batch_size** 设置每次 purge 操作需要清理的 undo page 数量。
参数：**innodb_max_purge_lag** 控制 history list 的长度，若长度大于该值，则会延迟 DML 操作，默认为 0，表示不限制。当大于 0 时，延缓 DML 操作，延缓算法为：
delay = ((length(history list) - innodb_max_purge_lag) * 10) - 5
参数：**innodb_max_purge_lag_delay** 控制 delay 的最大毫秒数
当根据 innodb_max_purge_lag 计算的 delay 大于此值时，设置为此值。

LSN （log sequence number）

LSN 表示事务写入重做日志的字节的总量，单位是字节。
LSN 存在于 page、redo log block、checkpoint

页中的 LSN：在每个页的头部，有一个值：FIL_PAGE_LSN，记录了该页的 LSN。在页中，LSN 表示该页最后刷新时，LSN 的大小。因为重做日志记录的是每个页的日志，因此页中的

LSN 用来判断页是否需要进行恢复操作。

恢复过程：
1.先扫 checkpoint 位置
2.再看页中的 lsn 是否需要恢复

undo log
参数：

innodb_max_undo_log_size	undo log 最大大小
innodb_undo_directory	undo log 目录
innodb_undo_log_truncate	
innodb_undo_logs	指定回滚段的大小
innodb_undo_tablespaces	指定有多少个 undo log 文件

插入 10000 万条记录，分批插入的原因？
减少复制延时。
begin

insert into …. (10000 次)

commit

参数：innodb_flush_log_at_trx_commit 控制是否每次事务提交，都刷新日志。

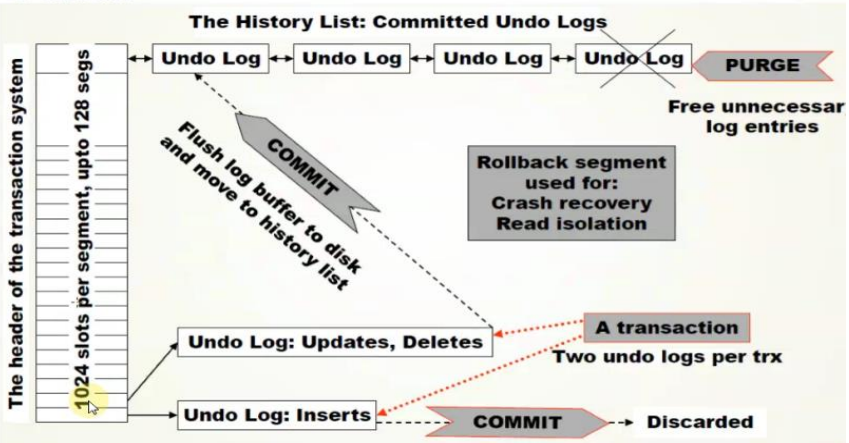
分批提交：减少主从延时。

回滚：数据都在 undo 里

mysql 通过 binlog 实现复制有延时，oracle 通过 redo 实现复制不会有延时

不在 mysql 中执行 alter table 原因却用 pt-online-schema-change（复制机制）？
pt-osc 通过新表和触发器机制，可以实时将数据同步到 Slave，不会有数据延时。因为 pt-osc 是复制机制，不是原地更新的机制。
主从延迟会小，但是执行效率比较差。

PURGE

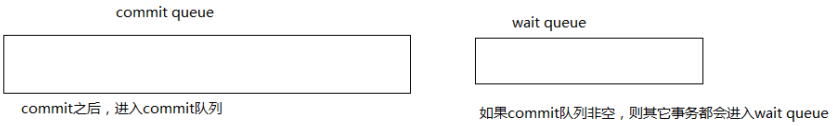


2016-09-27 组提交

group commit
组事务提交：prepare -> binlog -> 存储引擎日志
事务提交时，进行两个操作：

1. 修改内存中事务对应的信息，将日志写入重做日志缓冲
2. 调用 fsync 确保日志都从重做日志写入磁盘

group commit 合并多个事务为一个事务，进行一次 fsync 操作。



首先按照顺序将事务放入一个队列中，队列的第一个事务称为 leader，其它事务称为 follower，leader 控制着 follower 的行为。
步骤：

- flush 阶段：将每个事务的二进制写入内存中。
- sync 阶段 :将内存中的二进制日志刷新到磁盘，若队列中有多个事务，那么仅一次 fsync 操作就完成了二进制日志的写入。
- commit 阶段：leader 根据顺序调用存储引擎层事务的提交。

数据库省的都是 IO，写文件只能是串行的。

一个事务提交，需要做的事情：fsync redo log file 一次

组提交：1 次 IO 提交了多个事务。比如：1 次 fsync 就将 3 个事务写入到重做日志文件中。

参数：binlog_group_commit_sync_delay　控制 Flush 阶段中等待的时间，即使前一组事务已经提交，当前的事务也不会立即进入 Sync 阶段，而是至少等待这么长的时间。如果写入或者更新操作特别多，则比较适合。
参数：binlog_group_commit_sync_no_delay_count 等待多少个一起提交
默认都是 0，由 InnoDB 自己控制

影响：总体的 qps 可能会下降

打开 group_commit，使用 sysbench 测试性能提升？

什么是 logical_clock？为什么可以并行复制？
多线程回放。
并行复制的前提：没有冲突
flush binary logs; // 生成一个新的二进制日志文件

last_committed = 12 在哪个组中提交的
sequence_number = 120

在 master 上看 last_committed，看一组里有多少事务。

MySQL5.7 新推出的**多线程复制（并行复制）**
参数：
slave_parallel_type = **LOGICAL_CLOCK**
slave_parallel_workers = 4 （4 个线程）

原来是单线程
IO Thread -> SQL Thread

从机：SQL 线程->cord ->SQL 线程
->SQL 线程

set global binlog_group_commit_sync_delay = 5;

如果多线程复制延时比较大，则需要调整 logical_clock 的数量

事务的提交由 redo 控制。

参数 :**innodb_flush_log_at_trx_commit** = { 0 | 1 | 2 } 是不是在事务提交的时候，进行 fsync。
1：默认
0：不在事务每次提交时立即刷新 redo，由 InnoDB Master Thread 每隔 1 秒刷新一次。，缺点，数据丢失，是否在事务提交时，fsync 到 redo 文件。
2：写入 innodb log buffer

mysqldump 出来后
mysqldump 导出的数据中的 insert 会做合并，即：insert into t values**(),(),(),()**…
mysqldump --extended-insert 导出的数据不会合并。

MySQL5.7 基于表的复制
参数：binlog_order_commits = ON

exec master log info position 当前从机执行到 master 哪个位置

redo log 每 1 秒中刷新一次，
bin log 只在事务提交完成之后才写一次日志。

参数：
binlog_cache_size 会话级别参数。
每个事务提交都会产生二进制日志。

2016-10-08 锁

隔离级别（my.cnf）

transaction_isolation = READ-COMMITTED
MySQL 默认是 REPEATABLE-READ

默认情况下：
read-uncommitted 能读取到未提交的数据
read-committed 解决了脏读，未解决不可重复读、幻读
repeatable-read 解决不可重复读，未解决幻读【**MySQL 在这个隔离级别下已解决**】
serializable 解决所有

脏读 dirty read 【读取到未提交的数据】
不可重复读 unrepeatable-read 【在一个事务中，两次读取的数据不一致。一条记录】
幻读 phantom read 【两次读到的结果集的数量不一致】

可以在 slave 上把事务隔离级别调为 read-uncommit。

MySQL 在 repeatable-read 解决了幻读问题。

ACID
事务的隔离性：一个事务所做的修改，对其它事务不可见。

在不同的隔离级别下，锁的机制是不一样的。

锁的概念

latch（闩锁）

互斥锁（mutex）
读写锁（rw-lock）

latch 和 **lock** 在数据库中是完全不同的概念

	lock	latch
对象	事务	线程（多个线程访问同一个变量）
保护	数据库内容	内存中的数据结构（如：全局变量等）
持续时间	整个事务过程	临界资源（必须使用完立马释放）
模式	行锁、表锁、意向锁	读写锁、互斥锁、互斥量等
死锁	通过 waits-for graph、timeout 等机制进行死锁检测与处理	没有死锁检测和处理机制。
存在于	Lock Manager 的哈希表中	每个数据结构的对象中

命令：show engine innodb mutex; 查看锁信息

锁的类型

行级共享锁：S
行级排它锁：X
意向共享锁：IS

意向排它锁：IX
自增锁　　：AI

意向锁

设计目的：在一个事务中揭示下一行被请求的锁类型

揭示下一层级请求的锁类型，不关心当前层级。
IS：事务想要获取一张表中某几行的数据
IX：事务想要获取一张表中
InnoDB 中的意向锁是**表锁**。

MySQL 中的锁实际是锁一颗树。

排它锁：select * from a where a = a for update;
共享锁：select * from a where a = a lock in share mode;

InnoDB 中，没有数据库级别的锁，没有记录级别的锁。
意向锁都是加在表级别

什么时候对表加共享锁、排它锁？

alter table a add index idx_b(b);　a => **S lock** (MySQL5.5 之前)

S 锁：alter table 时添加
X 锁：不支持在表级别添加排它锁

自增锁

在事务提交之前释放
innodb_autoinc_lock_mode = {0 | 1 | 2}
0：传统方式
1：默认参数　（在 SQL 语句执行完之后，才释放锁）
　　simple inserts 并发
　　bulk insert 传统方式　（insert into a select **** 可以保证批量插入生成的 id 是连续）
2：所有自增都以并发方式（在键自增之后，就释放锁）
　　同一 sql 语句自增 id 可以不连续
　　row-based binlog

InnoDB 自增列必须被定义为一个 KEY
MySQL 自增实现：
　　启动时：select max(auto_inc_col) + 1 as max_auto_inc from tb for update;

但是并不持久化。

导致的问题：重启之后，会重新插入已经被删除的数据。

锁的查询

命令：show engine innodb status
参数：innodb_status_output_locks 是否可以看到锁
begin; // 如果不加这个，默认是自动提交的，就无法看到锁信息了。
select * from a where a = 1 for update;
锁信息如图：

```
------------
TRANSACTIONS
------------
Trx id counter 7945
Purge done for trx's n:o < 7449 undo n:o < 0 state: running but idle
History list length 34
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 421106375310048, not started
0 lock struct(s), heap size 1136, 0 row lock(s)
---TRANSACTION 7944, ACTIVE 21 sec
2 lock struct(s), heap size 1136, 1 row lock(s)
MySQL thread id 4, OS thread handle 139630227162880, query id 22 localhost root cleaning up
TABLE LOCK table `test1`.`user` trx id 7944 lock mode IX
RECORD LOCKS space id 39 page no 3 n bits 72 index PRIMARY of table `test1`.`user` trx id 7944 lock_mode X locks rec but not gap
Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
 0: len 8; hex 8000000000000001; asc         ;;
 1: len 6; hex 0000000001b9; asc       ;;
 2: len 7; hex a900000024011d; asc     $  ;;
 3: len 30; hex 0002002200120003001500040000c19000c1c006167656e616d6502313805; asc    "              agename 18 ; (total 35 bytes);
```

通过 sys 库的 **innodb_lock_waits** 视图可以查看锁信息。但是不能知道阻塞当前事务的 SQL 语句。

实际是通过 information_schema 库下的表：
innodb_trx
innodb_locks
innodb_lock_waits

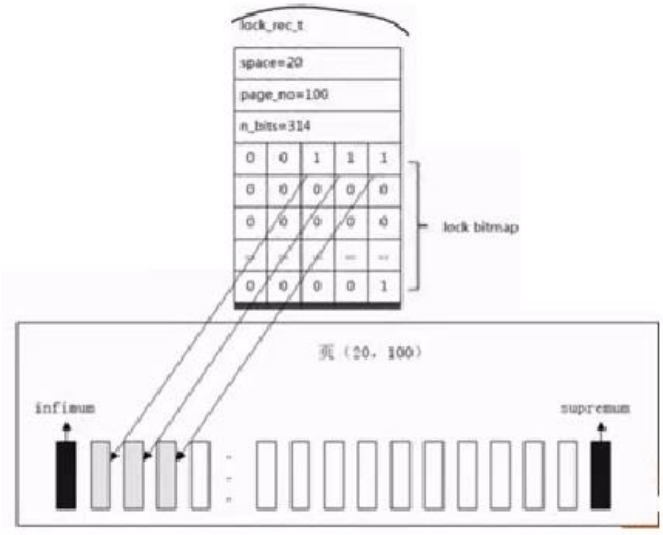
锁与并发

表锁如何获取（不是 innodb 层的锁，是 metadata lock 【mdl 也是从上往下加锁】）？
lock table tb1 write;
lock table tb1 read;
什么时候加表锁呢？ddl 的时候！

MySQL 的锁是怎么管理的？

用位图的方式来进行管理，lock bitmap

show engine innodb status 中的 heap no 是什么？表示在位图中的第几位。
每个页一个锁对象，保存在 bitmap 中，实际存储的是**页的 space,page**



2016-10-11 锁的算法 1

阿里云上关于锁的介绍：*https://yq.aliyun.com/articles/4270*

锁的类型

- RECORD LOCK *lock_mode X locks rec but not gap*
  单个行记录上的锁
- GAP LOCK *lock_mode X*
  锁定一个范围，不包括记录本身
- NEXT-KEY LOCK *lock_mode X locks gap before rec*
  锁定一个范围，并且锁定记录本身（锁定前一条记录到当前记录本身）
- **锁住的是索引**
- 插入意向锁（*lock_mode X locks gap before rec insertion*）
  gap lock

提高并发插入性能
只阻塞当前事务的插入，不阻塞其他事务在当前的范围插入

比如：select * from t where pk = 10
可能存在 3 种锁：
1. Record Lock
   锁住 *13*
2. Gap Lock
   锁住范围 *(11 - 13)* 不包括 13
3. Next-Key Lock
   锁住范围 *(11 - 13]* 包括 13

LOCK_ORDINARY

RC 只是在用户层面不存在 GAP 锁了，在其它层面还是有的。

锁的管理
使用位图来进行管理，且每个页 100 个字节的锁大小。

加锁策略

与隔离级别有关
repeatable-read：Next-Key Lock 和 Gap Lock
read-committed：Record Lock

REPREATABLE-READ

为什么在 *REPEATABLE-READ* 下加 **Next-Key Lock** ？
为了防止在同一个事务中出现幻读（事务中，两次查询返回**不同的结果集**）。

锁信息 **500 lock struct(s) ，heap size 342123，50000 row lock(s)**
表示：**499 个页锁（page），1 个表锁（table）**，锁住 50000 条记录。

MySQL 在默认的隔离级别（repreatable-read）下通过解决幻读已经达到了完整的隔离性

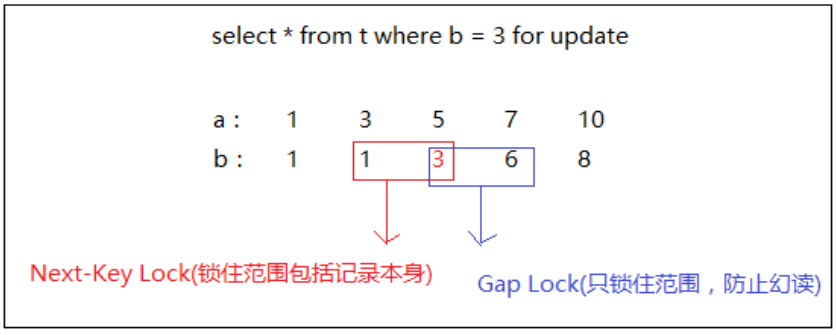
```
CREATE TABLE z ( a INT, b INT, PRIMARY KEY(a), KEY(b) );
INSERT INTO z SELECT 1,1;
INSERT INTO z SELECT 3,1;
INSERT INTO z SELECT 5,3;
INSERT INTO z SELECT 7,6;
INSERT INTO z SELECT 10,8;
```

session1：select * from z where b = 3 for update （排它锁 **X 锁**）
session2：

select * from z where a = 5 lock in share mode; 不能（主键是 Record Lock）
insert into z values(2,1); 可以
insert into z values(4,2); 不能
insert into z values(4,3); 不能
insert into z values(6,4); 不能
insert into z values(6,6); 不能（why？）
insert into z values(6,7); 可以（why？）
insert into z values(8,6); 可以

在 R-R 隔离级别下：
session1 中：加锁分析：
a：1 3 **5** 7 10
b：1 1 **3** 6 8

b（普通索引）：Next-Key Lock (1,3]
Gap Lock (3,6) 目的：解决**幻读**
a（主键索引）：Record Lock



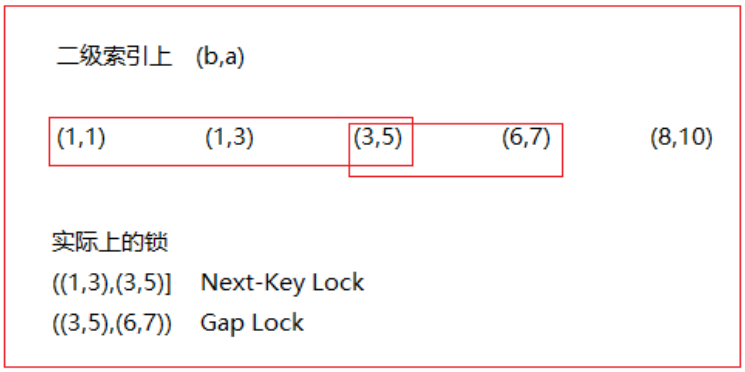
READ-COMMITTED

在 R-C 下：不需要解决幻读，都是 Record Lock

1.技能是什么？
2.擅长什么？
3.处理过什么问题？

2016-10-13 锁的算法 2

为什么 (6,6) 不能插入呢？
实际上二级索引加锁时，需要加上主键值。



实际锁定的范围是：
((1,3),(1,5)] Next-Key Lock
((3,5),(6,7)) Gap Lock
Oracle 不占用锁
InnoDB 是通过位图来记录锁
没有锁升级（Like Oracle）
【SQL Server 有锁升级】

Oracle 中的锁没有任何开销，锁的标记是存放在记录上。
InnoDB 是通过位图的方式来记录锁

锁定读（加锁）

锁定排它读：select * from t where pk = ? for update;
锁定共享读：select * from t where pk = ? lock in share mode;

非锁定读（MVCC）

consistent non-blocking read

多版本并发控制

即：**select 后不加 for update 或 lock in share mode**

select * from t where key = ? 读不加锁。

MVCC 实现原理：

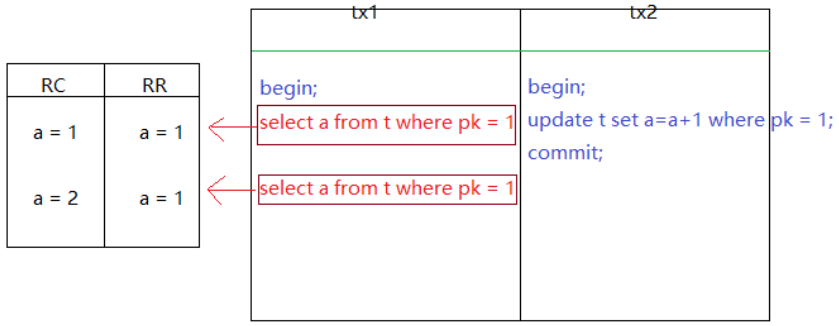
1. 发起对一条记录的读取，首先判断这条记录是否加锁，如果发现记录被加锁，则表示记录对当前的事务是不可见的。
2. 每条记录上都有 undo 指针（回滚指针），可以找到对应的回滚段，可以构造这条记录的前一个版本，则返回前一个版本的记录。

MVCC 利用 undo 实现对一条一条的记录进行回滚。

注意：每条记录上都有一个回滚指针（undo，7 个字节）和事务 id

与 Oracle 不同的是，Oracle 的 MVCC 永远读取到记录的最新版本，不支持 REPEATABLE-READ 的隔离级别。

在 READ-COMMITTED 隔离级别下：发生不可重复读。
在 REPEATABLE-READ 隔离级别下：解决不可重复读。



一条记录只有 1 个 undo 指针。

怎么知道一条记录是否对当前事务可见呢？
每条记录上还有一个事务 id，用来表示当前记录的可见性。
如果读到一条记录，发现记录的事务 id 比当前的事务 id 小，并且当前事务是活跃的，则表示记录是不可见的。

事务 id：用来判断记录是否是锁定的

RC 和 RR 创建 ReadView 的时机不同

ReadView 会创建新的事务 id

ReadView 是用来判断记录的可见性的。

RC 每次查询都会创建 ReadView
RR 只在第一次读的时候创建 ReadView

事务提交之后 **undo 指针**不会被删除，但是 undo 可能会被删除。

undo 什么时候会被删除呢？
没有其它事务引用当前 undo log 的时候，才会被删除，这个删除的过程叫 purge。

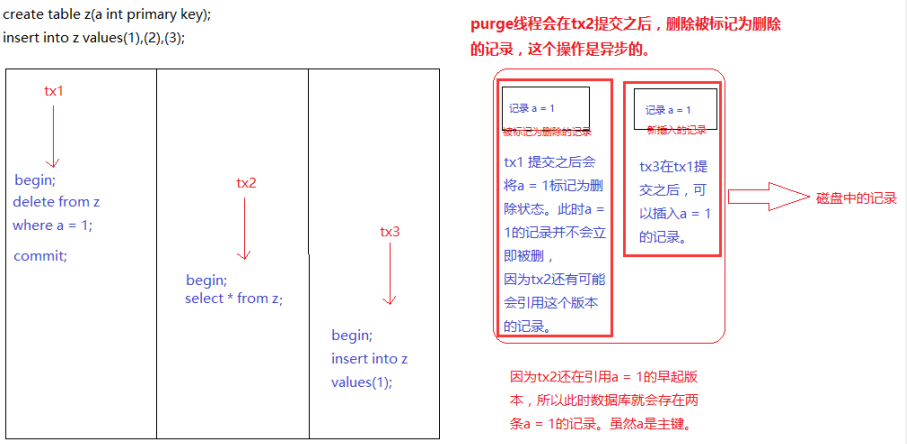
唯一索引问题

唯一索引中的数据都是唯一的吗？

删除一条记录是：并不是直接删除，而是标记删除。
为什么是标记删除呢？和 MVCC 有关
只有当数据存在时，才能通过 MVCC 来读取前一个版本，如果数据直接被删除，则无法看到前一个版本的数据。

实际上：update 和 delete 都不是真实的将数据删除或修改。

purge thread 回收 undo，真正删除记录。
如下图：



参数：
innodb_status_output

innodb_status_output_locks：打印锁信息

死锁

两个或两个以上的事务在执行过程中
因争夺资源而造成的一种互相等待的现象
AB - BA
解决死锁：
超时： innodb_lock_timeout
wait-for graph
自动死锁检测

死锁检测算法更改为非递归
死锁后，选择**回滚 undo 量最少的事务**。

锁超时

错误：ERROR 1205
参数：innodb_lock_wait_timeout （单位秒，会话级别）
锁超时之后，并不会直接回滚，需要手动回滚。

死锁

错误：ERROR 1213
参数：innodb_print_all_deadlock
    0：不打印
    1：打印所有的死锁信息到错误日志中
死锁会回滚当前事务。

为什么死锁会回滚，锁超时却处于中间状态？
死锁需要释放资源，锁超时可以重新获取资源。

购物车死锁

简单的说：库存减 1
begin;
update stock = stock -1 where skuId = 100;
update stock = stock -1 where skuId = 200;
commit;

会发生死锁：

	user1	user2	user3
	update stock = stock-1 where skuId = 100;	update stock = stock-1 where skuId = 200;	update stock = stock-1 where skuId = 300;
	update stock = stock-1 where skuId = 200;	update stock = stock-1 where skuId = 300;	update stock = stock-1 where skuId = 100;
点击结算时商品的顺序	skuId = 100 skuId = 200	skuId = 200 skuId = 300	skuId = 300 skuId = 100

解决方案：更新时，按照 skuId 排序。
原因：死锁是出现 AB 等待 BA 问题，进行排序之后，可以避免这种问题。

但是引来了第二个问题
出现了**由于锁等待导致的锁超时，会产生大量的等待。**

此时需要对应用程序做限流。
如何做限流呢？
应用层：

使用消息队列。
过滤 ip。

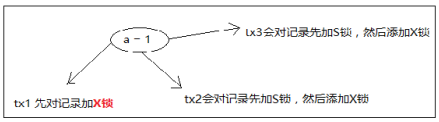
数据库层面：启动线程池
参数：thread_handling = pool-of-threads

唯一索引死锁

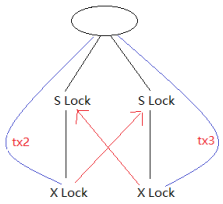
create table v (a int primary key);
insert into v values(1);

S Lock 是为了保证唯一性约束
delete from v where a = 1;
insert into v values(1);
insert into v values(1);

tx1:	tx2:	tx3:
begin; delete from z where a = 1;		
	begin; insert into z values(1);	begin; insert into z values(1);
commit;	DeadLock	Insert success



tx1在事务未提交时，当新的事务tx2，tx3在插入时，会尝试对记录添加S锁（保证记录的唯一性）。
tx1事务提交之后，tx2,tx3会对记录添加X锁。



tx2会等待tx3的S Lock
tx3会等待tx2的S Lock

死锁发生

MySQL 会在唯一索引锁等待的时候，对记录添加 S Lock（共享锁），以保证记录的唯一性。

添加索引：
老版本加 S Lock
新版本不加锁（online ddl）

2016-10-15 锁的算法 3

死锁

唯一索引引起的死锁

官方死锁示例：http://dev.mysql.com/doc/refman/5.7/en/innodb-locks-set.html

REPEATABLE-READ 隔离性

MySQL 在 RR 隔离级别下，实现了 2.99 的隔离性。
在这个例子中，没有实现可重复读。

```
create table t (a int ,b int ,primary key(a));
insert into t values(1,1),(2,1);
```

tx1	tx2
begin; select * from t where a = 1;	begin; update t set b = b + 1 where a = 1; commit;
select * from t where a = 1 for update; commit;	

批注 [cm1]: 此时查询的结果是：(1,1)。
原因：读不加锁

批注 [cm2]: 此时查询的结果是：(1,2)

在分布式事务下：一般需要设置隔离级别为：serializable。
在分布式事务情况下：可能存在悬挂事务，否则感知不到。

一致性读--READVIEW

一致性读以 ReadView 为基础实现。

所有的表都是一致性的读，即：读到的都是同一时间点的数据。

每次读都会创建 ReadView，会带着当前版本的 tx_id，只能看到比当前 tx_id 小的数据。

ReadView 是以第一条 SQL 语句开始的时间来创建的，事务 id 也是从第一条 SQL 开始时创

建的。

MySQL 是如何实现一致性读的呢？
每行记录上都有一个事务 Id

怎么在 begin 的时候，就发起 ReadView 呢？
start transaction with consistent snapshot；只在 REPEATABLE-READ 下生效。

好处：做备份。

备份

备份类型

热备（Hot Backup）

在线备份
对应用基本无影响

冷备（Cold Backup）

备份数据文件
需要停机
备份 datadir

温备（Warm Backup）

在线备份
对应用影响大
通常加一个读锁

备份工具

逻辑备份

保存的是 sql 语句。

mysqldump

mysqldump --**single-transaction** --master-data=1
--single-transaction 备份开始时的数据
-A --all-databases
-B db1 aaa test1
--master-data=1
> backup.sql
MySQL general_log 可以用来记录数据库的所有操作。

通过打开 general_log 和 log_output='table' 可以查看 mysqldump 原理。
逻辑备份原理（**备份开始的时间点的数据**）：

FLUSH TABLES;
FLUSH TABLES WITH READ LOCK; 获取实例级别的读锁（**非锁定读**）
SET SESSION TRANSACTION ISOLATION LEVEL **REPEATABLE READ**; 设置隔离级别
START TRANSACTION **WITH CONSISTENT SNAPSHOT**; 快照读（只读取当前时间的快照）
SHOW VARIABLES LIKE 'gtid_mode'; GTID 机制是否打开
SELECT @@GLOBAL.GTID_EXECUTED; 获取备份文件的位置
SHOW MASTER STATUS; 获取初始的备份文件
UNLOCK TABLES; 释放读锁
具体步骤：

1. FLUSH TABLES（关闭表）
2. FLUSH TABLES WITH READ LOCK（获取实例级别的读锁（**非锁定读**））
   /*!40100 SET @@SQL_MODE='' */
   /*!40103 SET TIME_ZONE='+00:00' */
3. SET SESSION TRANSACTION ISOLATION LEVEL **REPEATABLE READ**
4. START TRANSACTION /*!40100 WITH CONSISTENT SNAPSHOT */ （快照读）
5. SHOW VARIABLES LIKE 'gtid_mode'
6. SELECT @@GLOBAL.GTID_EXECUTED
7. UNLOCK TABLES
   SELECT LOGFILE_GROUP_NAME, FILE_NAME, TOTAL_EXTENTS, INITIAL_SIZ
   SELECT DISTINCT TABLESPACE_NAME, FILE_NAME, LOGFILE_GROUP_NAME,
   SHOW VARIABLES LIKE 'ndbinfo_version'
   test1
   SHOW CREATE DATABASE IF NOT EXISTS `test1`
8. SAVEPOINT sp
   show tables
   show table status like 'a1'
   SET SQL_QUOTE_SHOW_CREATE=1
   SET SESSION character_set_results = 'binary'
   show create table `a1`
   SET SESSION character_set_results = 'utf8'

```
    ◦◦◦◦◦◦◦
9.  ROLLBACK TO SAVEPOINT sp
10. RELEASE SAVEPOINT sp
```

savepoint 干嘛的？是相关元数据锁的
系统表如何备份？

mysqlpump

MySQL 官方的针对表级别的多线程并行备份工具。即：多线程备份多个表，针对某一个表，还是单线程。

mydumper【推荐】

github【https://github.com/maxbube/mydumper】

多线程备份工具。
针对表级别备份。

metadata：记录 GTID 位置。
每张表产生两个文件：
    表结构文件 tb1-schema.sql
    数据文件 tb1.sql

命令：
mydumper -t 4 -r 1000 --trx-consistency-only -B db1 -T tb1

-t 线程数
-r 每次导出行数
--trx-consistency-only 一致性读
-B 数据库
-T 表

如何保证多个线程导出时仍然是一致性读呢？

FLUSH TABLES WITH READ LOCK;

查看备份逻辑的步骤：
set global log_output = 'table';
set global general_log = 1;

mydumper -t 4 -r 1000 --trx-consistency-only -B db1 -T tb1

select * from mysql.general_log limit;

READ-COMMITTED 也是快照读，但是读取到的都是最新的快照的数据。

并行导出原理：
1.主线程 flush tables with read lock
2.其它线程切换 tx_isolation 为 repeatable-read
3.start transaction with consistent snapshot;
4.单表进行并行备份时，必须要有主键。实际是根据主键进行分片备份。

2016-10-18 备份原理-物理备份

RR 看到的是所有表，在当前时间的一致性读。创建 ReviewView 的时候，是当前读的时间。

物理备份
好处：恢复速度非常快
1.怎么操作
2.怎么实现
3.区别

备份工具

ibbackup 官方备份工具、收费
xtrabackup 开源、免费
mysqldump 官方

xtrabackup 备份与恢复

包含两个命令：xtrabackup、innobackup

备份用法

```
innobackupex
    --database=test      # 备份的数据库
    --compress --compress-threads=8   #启用压缩，压缩的线程数
```

```
--stream=xbstream[tar] --parallel=4   #  启用流的方式进行。tar 会慢很多
-uroot -p --socket=/tmp/mysql.sock
./ > backup.xbstream   #目录
```

恢复用法

```
xbstream
-x <backup.xbstream
-C tmp #解压到哪个目录

#  启动压缩之后，需要使用 qpress 解压
for f in `find ./ -iname "*\.qp"`; do qpress -dT2 $f $(dirname   $f) && rm -f   $f ; done

然后执行
innobackupex --copy-back   /path/to/backup-dir
innobackupex --apply-log    /path/to/backup-dir
```

xtrabackup 原理

1. 记录当前 redo 日志的 LSN
2. copy 表空间文件，同时 copy 当前产生的 redo 日志
3. flush table with read lock。是为了记录当前 filename，pos，gtid
4. copy redo 日志
5. backup 完成

与 mysqldump 和 mydumper 的区别？

1. flush tables with read lock; 记录当前 filename，pos，gtid
2. start transaction with consistent snapshot;
3. unlock tables;
4. select * from table1,table2…

最大的区别是：
xtrabackup 记录的是备份结束时的数据，
mysqldump 和 mydumper 记录的是备份开始时的数据，需要额外的二进制日志的回放

xtrabackup 直接备份的是文件

透明表空间传输

alter table t discard tablespace;
flush tables t from export;
copy t.idb,t.cfg to source machine
unlock tables;
alter table t import tablespace;

独立表空间的导入和导出

增量备份的原理？
基于页的修改

如何基于二进制日志做增量备份
flush binary logs; # 产生一个新的 bin.log
只需要备份这个 bin.log 之前的日志即可

物理备份步骤：

复制高级特性

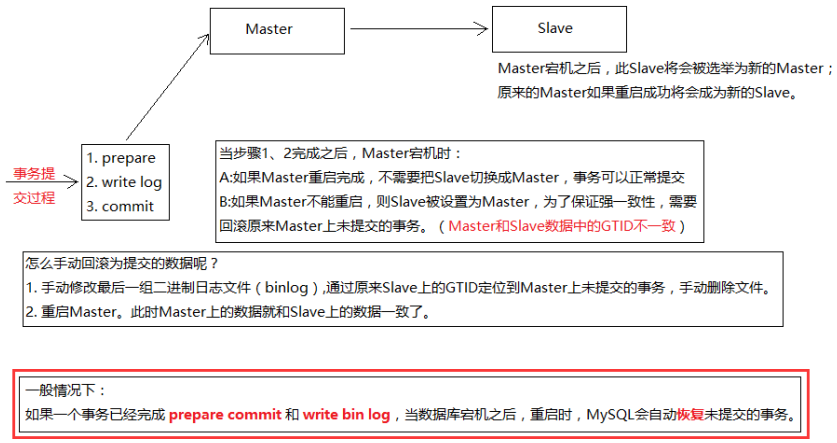
配置方式

异步复制
默认就是异步复制

半同步复制
semi_sync_master_enabled
semi_sync_slave_enabled
semi_sync_master_timeout

无损复制
semi_sync_master_wait_point=after_sync
semi_sync_master_wait_for_slave_count = 1

复制原理



(1) innodb prepare;
(2) write binary log; *# 无损复制，在这一步之后发送日志*
(3) innodb commit; *# 半同步复制，在这一步完成之后发送日志*

半同步复制【MySQL5.5】

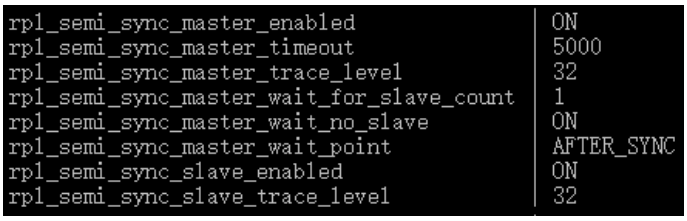
MySQL semi-sync replication

保证在第三步时，至少有 1 个 Slave 接收到数据

无损复制【MySQL5.7】

保证在第二步时，至少有 1 个 Slave 接收到数据

无损复制参数：



rpl_semi_sync_master_enabled	ON
rpl_semi_sync_master_timeout	5000
rpl_semi_sync_master_trace_level	32
rpl_semi_sync_master_wait_for_slave_count	1
rpl_semi_sync_master_wait_no_slave	ON
rpl_semi_sync_master_wait_point	AFTER_SYNC
rpl_semi_sync_slave_enabled	ON
rpl_semi_sync_slave_trace_level	32

参数：rpl_semi_sync_master_wait_for_slave_count 保证有几个 Slave 接收到二进制日志

半同步复制和无损复制的区别？为什么无损复制不会有数据丢失？

如：Master -> Slave
insert into t values 1;
在半同步复制下（第三步之后发送数据，此时 Master 已经提交）
如果在第三步提交后，但还没发送到 Slave 之前，master 宕机，此时，1 这条记录就丢失了。
在无损复制下（在第二步之后发送数据，此时 Master 还没有提交）
如果在第二步 Master 发生 crash，则数据已经发送到 Slave 上了，不会造成数据丢失。

问题：如果 Slave 已经接收到数据，但是 Master 还没有提交时，Master 宕机，此时会不会多一条数据呢？
不会，在 prepare 和 write log 完成之后，表示事务已经提交，如果 Master 启动成功，则会自动 commit 这条事务。

强一致的高可用？
出现问题的步骤：
(1) Master 上已经完成了 prepare commit；write binary log;
(2) Master 上发送到 Slave 上时，日志还没有发送到 Slave 时，Master 宕机
(3) Slave 被选举为 Master
(4) 宕机的 Master 又重启成功
(5) 此时就需要 rollback 掉宕机的 Master 上已经 prepare 的事务

怎么回滚？删掉对应的二进制日志
truncate 掉最后的日志文件。
在 Slave 上找到最新的 gtid（即同步到的最新的日志）
在 Master 上找到大于 Slave 上的 gtid 的日志，把这些日志都删掉就可以了。
实际是找到对应的 filename 和 position，通过 truncate 函数，截断 bin.log 就可以了。

需要在 Master 上安装 agent，需要在 Master 每次启动时，找到 Slave 上最新的 gtid，判断是否已经完全同步，如果没有，则删掉大于这个 gtid 的日志。

数据库底层实际是组提交，如果需要删除，则一般需要删除 10 左右的事务

无损复制，实际上利用组提交来减少网络传输，变相的提高了传输的速度。

如果启动无损复制之后，但是所有的 Slave 宕机都宕机了，会发生 Master 上的所有操作都会 hang 住，即无法接收新的日志。
此时会根据参数
超时时间：rpl_semi_sync_master_timeout
超时之后自动转为异步复制，在后期如果又重新连上之后，又会自动变为无损复制。

如果要做强一致，则需要把这个参数设置的非常大，防止出现网络抖动而引发的无损复制自动变为异步复制。

通过 pt-heartbeta 工具可以查看是否 hang 住。

多源复制

复制搭建：(要求每个示例上的 server-id 不一样)

1.备份（master3306 上）
    mysqldump --single-transaction --master-data=1 -A >fullbackup.sql
2.恢复（slave3307 上）
    mysqlslave < fullbackup.sql
3. master3306 上
    create user rpl@'%' identified by '123'
4. master3306 上
    grant replaction slave on *.* to rpl@'%'
5. slave3307 上
    change master to master_log_file = 'binlog.00001',master_log_pos=123,
    master_host='127.0.0.1',master_port=3306,
    master_user='rpl',master_password='123'
6. start slave

master 上：show slave hosts;
slave 上：show slave status

如果在从机上执行 create database db1 失败了；

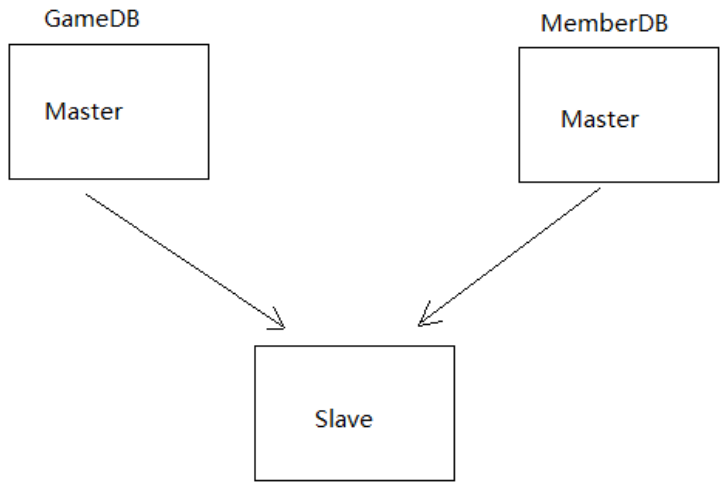
可以 set global sql_slave_skip_counter=1;跳过当前的 event
然后再 start slave;此时就又开始同步了。

2016-10-20 MySQL 复制

无损复制性能

无损复制参数：semi_sync_master_wait_point = **after_sync**

multi source replaction 多源复制
用途：业务合并，类似于数据仓库



在游戏中合区时，会发生合库操作。

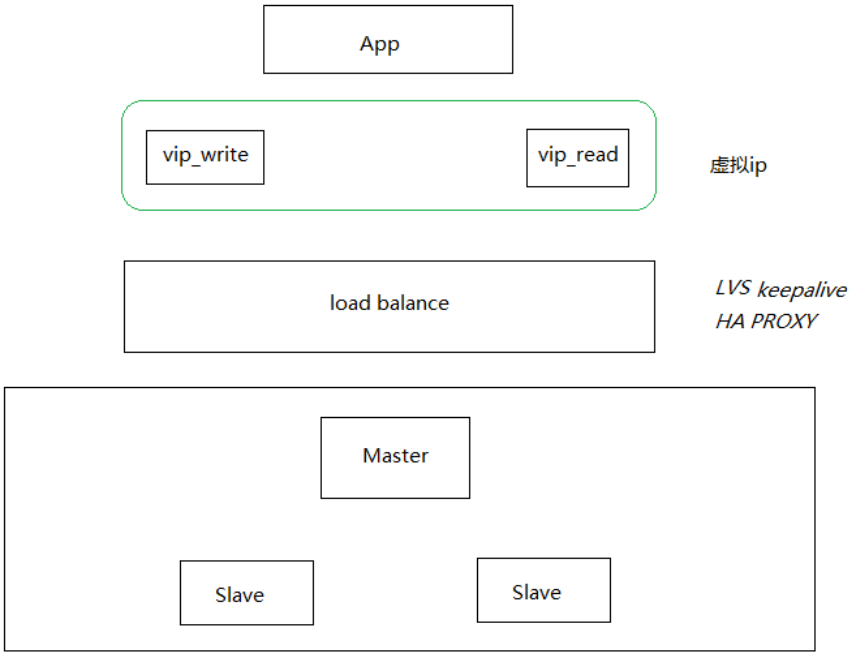
如果没有这种，怎么办？

使用 Federated 存储引擎。(或者使用 ETL)

2016-10-25 高可用

2016-10-27 高可用

读写分离方案



在服务器安装 keepalived 软件

MHA

是一个高可用软件

实时监控数据库状态，如果一台数据库宕机了，它会自动创建主从关系

前提：MHA 的机器可以免密码 ssh 登录到服务器。

原理：
    ssh 192.168.1.101 登录服务器
    ifconfig eth0:0 xxxx 设置虚拟 ip

1. 使用 lvs 搭建负载均衡
2. 使用 mysql router 搭建负载均衡
3. 查看视频

2016-11-05 分布式数据库

分布式数据库的特点

优点：
可用性提高
可扩展性提升
某些情况下吞吐率提升
缺点：
依赖中间件
sql 语句支持不足
运维复杂度提高
某些情况下性能下降巨大

分布式数据库-Shard

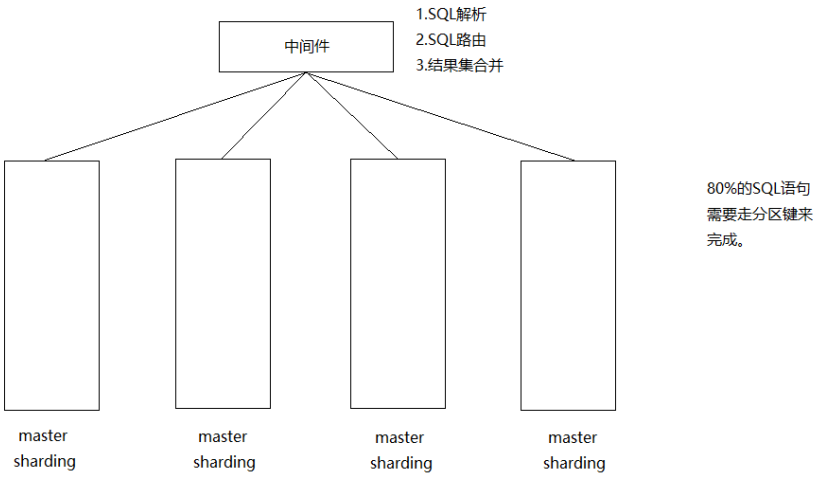
shard 和 partition 是类似的概念

分区键的选择

分区键

```
create table employees(
    id bigint not null auto_increment,
    name varchar(32),
    primary key pk_id(id)
```

)
partition by hash(id)
partitions 4



原则 1：如果不能选择有效的分区键，这张表就不要 shard，作为全局表。如：库存表
原则 2：如果业务选择不出有效的分区键，那也不需要 shard

例如电商：userId
用户表：users
订单表：orders
订单明细表：order_line

库存表：stock
下单之后如何减库存呢？
stock 专门放在一个全局的实例中。

减去库存时，变成一个分布式事务。

begin;
insert into orders;
insert into order_line;
update stock set stock = stock -1; // 这里会发生分布式事务
commit;

一般都没有实现分布式事务。
把减库存的操作 insert 到 mq 中

会有一个对账系统，如果发现某个订单在支付 30 分钟之后，还没有下单成功，会把这个订单放到人工处理的地方。

最后达到 **最终一致**。

分区键选择 userId
针对的都是同一个用户的信息

买家的所有数据都可以根据 userId 分片，之后落在同一个

快递行业：orderId

SQL 查询相对来说比较简单，不能跨分片。

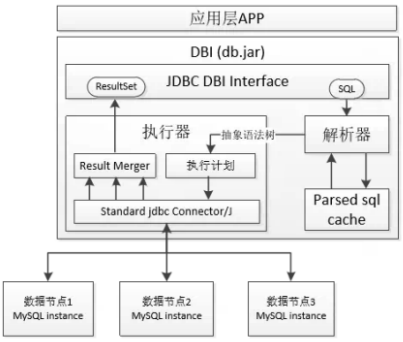
中间件产品

MySQL Proxy
MariaDB MaxScale
网易 DDB/淘宝 TDDL
MyCat / Cobar

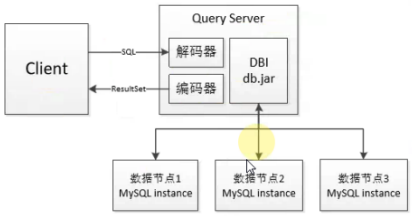
DDB——DBI模式

优点：
- 性能较好

缺点：
- 仅支持Java语言
- DDB版本升级需要应用层配合



DDB——QS模式



优点：
- 支持各类编程语言
- Query Server水平可扩展
- 更好地运维与监控
- 平滑易升级

缺点：
- 性能较DBI模式差
  - 但是可以通过多个QS来提升性能

模式：DBI 模式
实现方式：重写 jdbc

优点：性能较好
缺点：仅支持 Java 语言，版本升级需要应用层配合

模式：QS 模式
client 发送 SQL 到 QueryServer，然后下发到 MySQL 实例
通信协议都是 MySQL 协议，并且 QueryServer 是无状态的，即 QueryServer 可以集群化部署。
缺点：性能较差（多连一次 QueryServer 服务器，多了一次网络转发）
优点：通信协议是 MySQL 协议，客户端可以像连接 MySQL 一样，连接 QueryServer。

2016-11-08 优化

内存

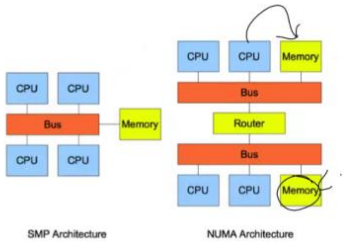
关键是 **NUMA 和透明大页**

什么是 crash-save ?

什么是 NUMA ?

软硬件设置——内存

- NUMA
  - Non-Uniform Memory Access
  - 非一致存储访问结构



NUMA的内存分配策略有四种：
1. default：总是在本地节点分配
2. bind：强制分配到指定节点上
3. interleave：在所有节点或者指定的节点上交织分配
4. preferred：在指定节点上分配，失败则在其他节点上分配

swap
free

下载和使用 numa-ctl 工具
修改 /etc/init.d/mysqld 启动文件

```
    if test -x $bindir/mysqld_safe
    then
      # Give extra arguments to mysqld with the my.cnf file. This script
      # may be overwritten at next upgrade.
      echo "Use numactl to start MySQL"
      numactl --interleave all $bindir/mysqld_safe --datadir="$datadir" --pid-f
ll 2>&1 &
      wait_for_pid created "$!" "$mysqld_pid_file_path"; return_value=$?

      # Make lock for RedHat / SuSE
      if test -w "$lockdir"
      then
        touch "$lock_file_path"
      fi

      exit $return_value
    else
```

numa-ctl 控制每个进程的内存分配策略

numactl --interleave all

建议在数据库应用中关闭**透明大页**

磁盘调度算法：deadline
/sys/block/vdb/

网卡

mount　noatime nobarrier
网卡：qps 在 3 万以上，才会出现
网卡软中断：
top 去看的话，在一个 cpu 上 soft 会达到 100%，负载非常不均衡。

解决方案：
启动网卡多队列
set_irq_affinity.sh[来自 Google]
service iqrbalance stop

RAID 卡

BBU
建议打开
RAID 缓存
write backup【断电时，会使用电池，将内存中的数据写入磁盘】

其它

透明页压缩