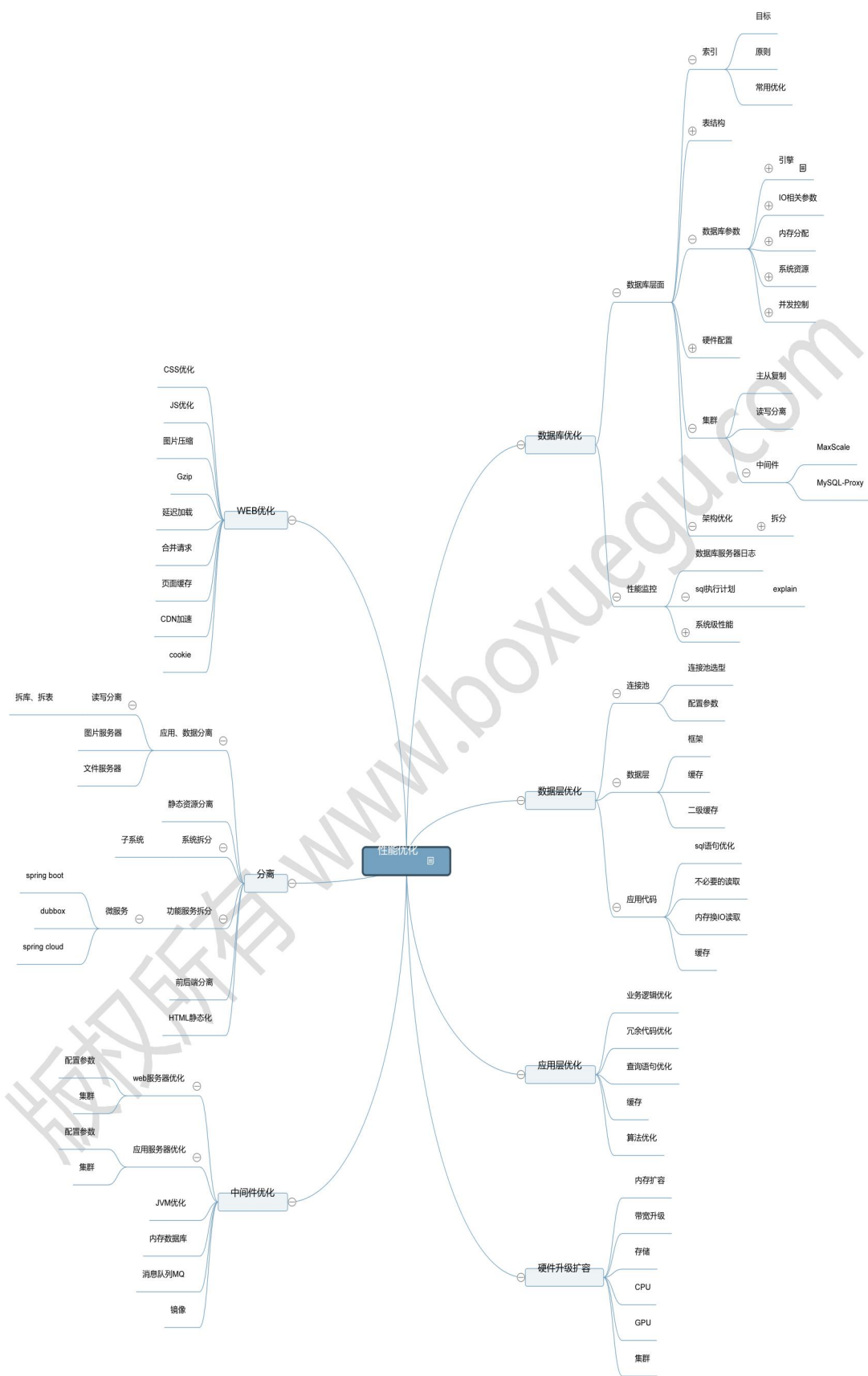


JAVAE 性能优化系列课程

性能优化概览

作为一个程序员，性能优化是常有的事情，性能优化的概念很广，不管是桌面应用还是 web 应用还是 APP 应用，不管是前端还是后端，不管是单点应用还是分布式系统，都会涉及到性能调优的问题。性能优化的几点建议和原则：

- 1、适度优化，切忌过度优化
- 2、先优化最大瓶颈，事半功倍
- 3、依据数据而不是凭空猜测
- 4、性能优化是持久战，道高一尺魔高一丈
- 5、深入理解业务



第一章 MySQL 数据库优化

1.1. 准备工作

1.1.1. 慢查询日志

当查询超过一定的时间没有返回结果的时候，才会记录到慢查询日志中。默认不开启。

采样的时候手工开启。可以帮助我们找出执行慢的 SQL 语句

查看慢 SQL 日志是否启用（on 表示启用）：

```
show variables like 'slow_query_log';
```

查看执行慢于多少秒的 SQL 会记录到日志文件中

```
show variables like 'long_query_time';
```

可以使用模糊搜索，查看所有含有 query 的变量信息

```
show variables like '%query%';
```

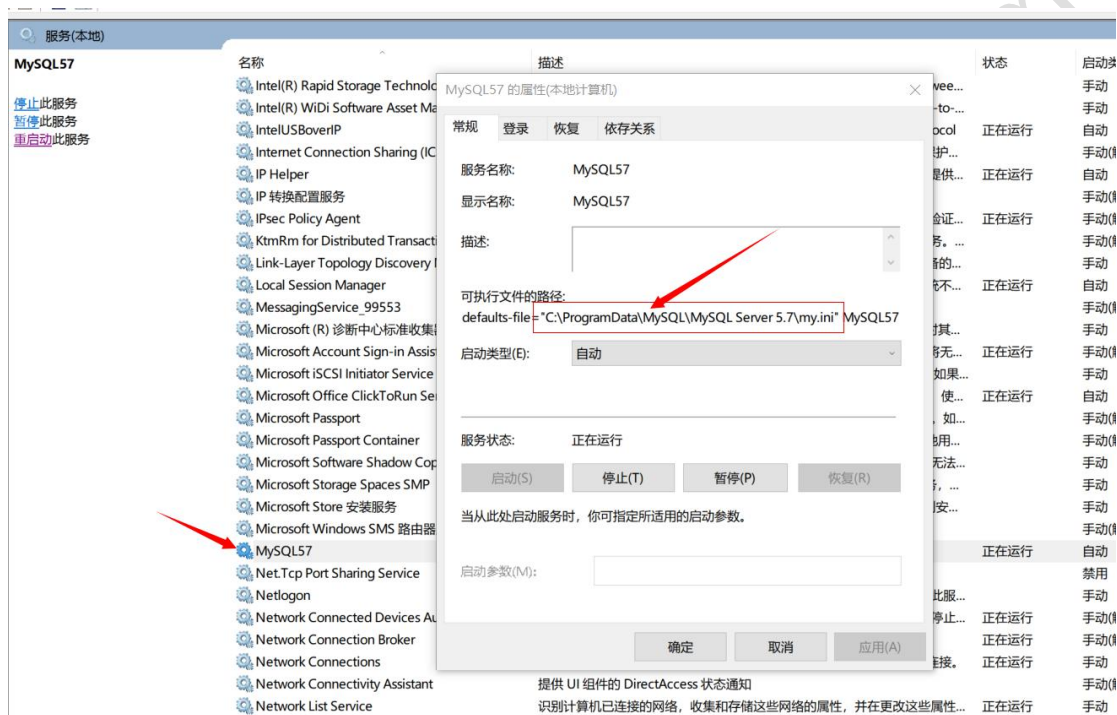
```
1 show variables like '%query%';
```

信息	结果1	概况	状态
Variable_name	Value		
binlog_rows_query_log_events	OFF		
ft_query_expansion_limit	20		
have_query_cache	YES		
long_query_time	10.000000		
query_alloc_block_size	8192		
query_cache_limit	1048576		
query_cache_min_res_unit	4096		
query_cache_size	0		
query_cache_type	OFF		
query_cache_wlock_invalidate	OFF		
query_prealloc_size	8192		
slow_query_log	ON		
slow_query_log_file	DESKTOP-XFSQRD-slow.log		

1.1.2. 修改 mysql 配置参数

my.ini (Linux 下文件名为 my.cnf)，查找到[mysqld]区段，增加日志的配置。

Windows 下路径一般为 C:\ProgramData\MySQL\MySQL Server 5.7\my.ini"，可以在启动参数中查看使用的是那个配置文件。



常用的参数详解：

```
#--是否开启慢查询日志
slow_query_log=1
# --指定保存路径及文件名，默认为数据文件目录，
slow_query_log_file="bxg_mysql_slow.log"
# --指定多少秒返回查询的结果为慢查询
long_query_time=1
# --记录所有没有使用到索引的查询语句
log_queries_not_using_indexes=1
#--记录那些由于查找了多于 1000 次而引发的慢查询
min_examined_row_limit=1000
# --记录那些慢的 optimize table, analyze table 和 alter table 语句
log_slow_admin_statements=1
#--记录由 Slave 所产生的慢查询
log_slow_slave_statements=1
```

datadir=C:/ProgramData/MySQL/MySQL Server 5.7\Data --数据文件目录

注意：修改以下参数，需要重新启动数据库服务才会生效。

1.1.3. 命令行修改慢查询配置

命令行修改配置方式不需要重启即可生效，但重启之后会自动失效。

```

set global slow_query_log=1;
set global slow_query_log_file='bxg_mysql_slow.log';
set long_query_time=1;
set global log_queries_not_using_indexes=1;
set global min_examined_row_limit=1000;
set global log_slow_admin_statements=1;
set global log_slow_slave_statements=1;
    
```

其他参数可通过以下命令查阅：

```

show variables like '%query%';
show variables like '%slow%';
    
```

1.1.4. 慢日志格式

时间、主机信息、执行信息、执行时间、执行内容

```

# Time: 2019-03-08T12:40:26.772710Z
# User@Host: root[root] @ localhost [::1] Id: 12
# Query time: 0.096832 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0
SET timestamp=1552048826;
INSERT INTO lianjia(city,title,price,perprice,
refprice,buildtime,communityname,area,recordid,huxing,louceng,
jianzhumianji,huxingjiegou,taoneimianji,jianzhuleixing,
chaoxiang,jianzhujiegou,zhuangxiu,tihubili,gongnuanfangshi,dianti,chanquanianxian,
guapaishijian,jiaoyiquanshu,shangcijiyaoyi,yongtu,nianxian,chanquansuoshu,diyaxinx,i,fangbenbeijian,fangxiebianma
) VALUES
('gz','此房子望小区园林，位置方便，户型舒适','310万','37256元/平米','首付及贷款情况请咨询经纪人','2006年建/板楼','南国奥林匹克花园','番禺;汉溪','108400264287','2室2厅1厨1卫','中楼层 (共4层)','83.21m','错层','73.6m','东
西','钢筋混凝土','精装','一梯两户','','无','70年','2019-02-21','商品房','2013-04-09','普通住宅','满五年','共有','有抵押 67万
客户偿还','已上传房本照片','');
    
```

1.1.5. 查询缓存

Query Cache 会缓存 select 查询，安装时默认是开启的，但是如果对表进行 INSERT, UPDATE, DELETE, TRUNCATE, ALTER TABLE, DROP TABLE, or DROP DATABASE 等操作时，之前的缓存会无效并且删除。这样一定程度上也会影响我们数据库的性能。所以对一些频繁的变动表的情况开启缓存是不明智的。还有一种情况我们测试数据库性能的时候也要关闭缓存，避免缓存对我们测试数据的影响。

```
show VARIABLES like '%cache%';
```

Variable_name	Value
innodb_disable_sort_file_cache	OFF
innodb_ft_cache_size	8000000
innodb_ft_result_cache_limit	2000000000
innodb_ft_total_cache_size	640000000
key_cache_age_threshold	300
key_cache_block_size	1024
key_cache_division_limit	100
max_binlog_cache_size	18446744073709547520
max_binlog_stmt_cache_size	18446744073709547520
metadata_locks_cache_size	1024
query_cache_limit	1048576
query_cache_min_res_unit	4096
query_cache_size	0
query_cache_type	OFF
query_cache_wlock_invalidate	OFF
stored_program_cache	256
table_definition_cache	1400
table_open_cache	2000
table_open_cache_instances	16
thread_cache_size	10

查看缓存命中情况

```

select count(*) FROM test;
select count(*) FROM test;
show status like '%qcache%';
    
```

信息	结果1	结果2	结果3	概况	状态
Variable_name	Value				
Qcache_free_blocks	1				
Qcache_free_memory	31438688				
Qcache_hits	3				
Qcache_inserts	1				
Qcache_lowmem_prunes	0				
Qcache_not_cached	18				
Qcache_queries_in_cache	1				
Qcache_total_blocks	4				

关闭缓存有两种放法，一种临时的，一种永久的。临时的直接在命令行执行

```
set global query_cache_size=0;
```

set global query_cache_type=0; --如果配置文件中为关闭缓存的话，不能通过命令开启缓存

永久的修改配置文件 my.cnf ,添加下面的配置即可。

query_cache_type=0

query_cache_size=0

另外，我们还可以通过 sql_no_cache 关键字在 sql 语句中直接禁用缓存，在开启缓存

的情况下我们对 sql 语句做一些改动

Select sql_no_cache count(*) from pythonlearn.lianjia; -- 不缓存

Select sql_cache count(*) from pythonlearn.lianjia; -- 缓存(也可以不加，默认缓存已经开启了)

1.1.6. 准备测试数据

创建测试表

-- 用户表

```
CREATE TABLE `person` (  
  `id` bigint(20) unsigned NOT NULL,  
  `fname` varchar(100) NOT NULL,  
  `lname` varchar(100) NOT NULL,  
  `age` tinyint(3) unsigned NOT NULL,  
  `sex` tinyint(1) unsigned NOT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=INNODB DEFAULT CHARSET=utf8
```

--用户部门表

```
CREATE TABLE `department` (  
  `id` bigint(20) unsigned NOT NULL,  
  `department` varchar(100) NOT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=INNODB DEFAULT CHARSET=utf8
```

-- 用户住址表

```
CREATE TABLE `address` (  
  `id` bigint(20) unsigned NOT NULL,  
  `address` varchar(100) NOT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=INNODB DEFAULT CHARSET=utf8
```

创建存储过程，用于批量添加测试数据

delimiter \$\$

drop procedure if exists generate;

CREATE DEFINER='root'@'localhost' PROCEDURE `generate`(IN num INT)

BEGIN


```
DECLARE chars VARCHAR(100) DEFAULT
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789';
DECLARE fname VARCHAR(10) DEFAULT '';
DECLARE lname VARCHAR(25) DEFAULT '';
DECLARE id int UNSIGNED;
DECLARE len int;
set id=1;
DELETE from person;
WHILE id <= num DO
    set len = FLOOR(1 + RAND()*10);
    set fname = '';
    WHILE len > 0 DO
        SET fname = CONCAT(fname,substring(chars,FLOOR(1 + RAND()*62),1));
        SET len = len - 1;
    END WHILE;
    set len = FLOOR(1+RAND()*25);
    set lname = '';
    WHILE len > 0 DO
        SET lname = CONCAT(lname,SUBSTR(chars,FLOOR(1 + RAND()*62),1));
        SET len = len - 1;
    END WHILE;
    INSERT into person VALUES (id,fname,lname, FLOOR(RAND()*100),
FLOOR(RAND()*2));
    set id = id + 1;
END WHILE;
END $$

delimiter $$
drop procedure if exists genDepAdd;
CREATE DEFINER='root'@'localhost' PROCEDURE `genDepAdd`(IN num INT)
BEGIN
    DECLARE chars VARCHAR(100) DEFAULT '行政技术研发财务人事开发公关推广营销咨询客服
运营测试';
    DECLARE chars2 VARCHAR(100) DEFAULT '北京上海青岛重庆成都安徽福建浙江杭州深圳温
州内蒙古天津河北西安三期';
    DECLARE depart VARCHAR(10) DEFAULT '';
    DECLARE address VARCHAR(25) DEFAULT '';
    DECLARE id int UNSIGNED;
    DECLARE len int;
    set id=1;
    WHILE id <= num DO
        set len = FLOOR(2 + RAND()*2);
        set depart = '';
        WHILE len > 0 DO
```



```
SET depart = CONCAT(depart,substring(chars,FLOOR(1 + RAND()*26),1));
SET len = len - 1;
END WHILE;
set depart=CONCAT(depart,'部');

set len = FLOOR(6+RAND()*18);
set address = "";
WHILE len > 0 DO
    SET address = CONCAT(address,SUBSTR(chars2,FLOOR(1 + RAND()*33),1));
    SET len = len - 1;
END WHILE;
INSERT into department VALUES (id,depart);
INSERT into address VALUES (id,address);
set id = id + 1;
END WHILE;
END $$
```

为了提高速度，可以暂停事务。测试添加 100 万随机数据，大概 600s 左右时间。

```
-- 停掉事务
set autocommit = 0;
-- 调用存储过程
call generate(1000000);
-- call genDepAdd(1000000);
-- 重启事务
set autocommit = 1;
```



```
[SQL]-- 停掉事务
set autocommit = 0;
受影响的行: 0
时间: 0.001s

[SQL]
-- 调用存储过程
call generate(1000000);
受影响的行: 1
时间: 601.388s

[SQL]
-- 重启事务
set autocommit = 1;
受影响的行: 0
时间: 3.566s
```

对比 MyIsam：当创建表时选择 MyIsam 格式，插入数据会很慢，仅仅 3000 条数据就需要 2 分钟的时间，由此可见 MyIsam 和 InnoDB 的差距还是很大的。另外在执行过程中

可以发现 Mysam 插入的数据可以在表中实时看到，而 InnoDB 做了事务最终一次提交，所以数据不能实时看到，只有存储过程全部执行完成后才可以看到数据。



1.2. 常用工具

1.2.1. 分析工具

1.2.1.1. Mysqldumpslow

mysqldumpslow 是 mysql 自带的用来分析慢查询的工具，基于 perl 开发。

Windows 下需要下载安装 perl 编译器，下载地址：

<http://pan.baidu.com/s/1i3GLKAp>

参考：https://www.cnblogs.com/moss_tan_jun/p/8025504.html

C:\Program Files\MySQL\MySQL Server 5.6\bin>perl mysqldumpslow.pl --help

```

Usage: mysqldumpslow [ OPTS... ] [ LOGS... ]

Parse and summarize the MySQL slow query log. Options are

--verbose      verbose
--debug        debug
--help         write this text to standard output

-v            verbose
-d            debug
-s ORDER      what to sort by (al, at, ar, c, l, r, t), 'at' is default
               al: average lock time
               ar: average rows sent
               at: average query time
               c: count
               l: lock time
               r: rows sent
               t: query time
-r            reverse the sort order (largest last instead of first)
-t NUM        just show the top n queries
-a            don't abstract all numbers to N and strings to 'S'
-n NUM        abstract numbers with at least n digits within names
-g PATTERN    grep: only consider stmts that include this string
-h HOSTNAME    hostname of db server for *-slow.log filename (can be wildcard),
               default is '*', i.e. match all
-i NAME        name of server instance (if using mysql.server startup script)
-l            don't subtract lock time from total time
    
```

```

perl mysqldumpslow.pl -r -s c -a -t 3 "C:\ProgramData\MySQL\MySQL Server
5.7\Data\bxxg_mysql_slow.log"
    
```

```

c:\Program Files\MySQL\MySQL Server 5.7\bin>perl mysqldumpslow.pl -r -s c -a -t 3 "C:\ProgramData\MySQL\MySQL S
erver 5.7\Data\bxxg_mysql_slow.log"

Reading mysql slow query log from C:\ProgramData\MySQL\MySQL Server 5.7\Data\bxxg_mysql_slow.log
Count: 1 Time=0.00s (0s) Lock=0.00s (0s) Rows=0.0 (0), 0users@0hosts
Time: 2019-03-12T13:14:51.568098Z
# User@Host: root[root] @ localhost [127.0.0.1] Id: 30
# Query_time: 0.000000 Lock_time: 0.000000 Rows_sent: 15 Rows_examined: 316
SET timestamp=1552396491;
SELECT QUERY_ID, SUM(DURATION) AS SUM_DURATION FROM INFORMATION_SCHEMA.PROFILING GROUP BY QUERY_ID

Count: 1 Time=0.00s (0s) Lock=0.00s (0s) Rows=0.0 (0), 0users@0hosts
Time: 2019-03-12T13:22:00.158574Z
# User@Host: root[root] @ localhost [127.0.0.1] Id: 30
# Query_time: 0.017569 Lock_time: 0.000000 Rows_sent: 100 Rows_examined: 10100
SET timestamp=1552396920;
select * from pythonlearn.lianjia limit 10000,100

Count: 1 Time=0.00s (0s) Lock=0.00s (0s) Rows=0.0 (0), 0users@0hosts
Time: 2019-03-12T13:19:02.454108Z
# User@Host: root[root] @ localhost [127.0.0.1] Id: 30
# Query_time: 0.001956 Lock_time: 0.000000 Rows_sent: 16 Rows_examined: 355
SET timestamp=1552396742;
SELECT STATE AS `锁状态`, ROUND(SUM(DURATION),7) AS `锁时间`, CONCAT(ROUND(SUM(DURATION)/0.001218*100,3), '%
') AS `锁时间占比` FROM INFORMATION_SCHEMA.PROFILING WHERE QUERY_ID=222 GROUP BY STATE ORDER BY SEQ
    
```

Count: 4 (执行了多少次)

Time=375.01s (每次执行的时间) (1500s) (一共执行了多少时间)

Lock=0.00s (0s) (等待锁的时间)

Rows=10200.3 (每次返回的记录数) (40801) (总共返回的记录数), username[password]@[10.194.172.41]

1.2.1.2. mysqlsla

Mysqlsla 是 daniel-nichter 用 perl 写的一个脚本，专门用于处理分析 Mysql 的日志

而存在。通过 Mysql 的日志主要分为：General log, slow log, binary log 三种。通 过

query 日志，我们可以分析业务的逻辑，业务特点。通过 slow log，我们可以找到服务器的瓶颈。通过 binary log，我们可以恢复数据。Mysqsla 可以处理其中的任意日志。

参考：<https://yq.aliyun.com/articles/59260>

1.2.1.3. pt-query-digest

pt-query-digest 是用于分析 mysql 慢查询的一个工具，它可以分析 binlog、General log、slowlog，也可以通过 SHOWPROCESSLIST 或者通过 tcpdump 抓取的 MySQL 协议数据来进行分析。可以把分析结果输出到文件中，分析过程是先对查询语句的条件进行参数化，然后对参数化以后的查询进行分组统计，统计出各查询的执行时间、次数、占比等，可以借助分析结果找出问题进行优化。

参考：<https://blog.csdn.net/seteor/article/details/24017913>

1.2.2. EXPLAIN 执行计划

1.2.2.1. 用法

1. EXPLAIN SELECT

经常使用的方式，查看 sql 的执行计划

2. EXPLAIN EXTENDED SELECT

将执行计划"反编译"成 SELECT 语句，运行 SHOW WARNINGS，可得到被 MySQL 优化器优化后的查询语句。

3. EXPLAIN PARTITIONS SELECT

用于分区表的 EXPLAIN 生成 QEP 的信息，用来查看索引是否正在被使用，并且输出其使用的索引的信息。

EXPLAIN SELECT id, fname, lname FROM person WHERE lname='x8RJWmQX' AND id in (select id from person where id BETWEEN 0 and 6000);

信息	结果1	概况	状态								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	person	(Null)	range	PRIMARY	PRIMARY	8	(Null)	11522	10	Using where
1	SIMPLE	person	(Null)	eq_ref	PRIMARY	PRIMARY	8	course11		100	Using index

1.2.2.2. 字段类别

字段	说明
Id	查询中执行 select 子句或操作表的顺序
Select_type	所使用的 SELECT 查询类型，包括以下常见类型： SIMPLE、PRIMARY、SUBQUERY、UNION、DERIVED、UNION RESULT、DEPENDENT、 UNCACHEABLE
table	所使用的的数据表的名字
type	表示 MySQL 在表中找到所需行的方式，又称“访问类型”。取值按优劣排序 为 NULL > system > const > eq_ref > ref > fulltext > ref_or_null > index_merge > unique_subquery > index_subquery > range > index > ALL
Possible_keys	可能使用哪个索引在表中找到记录
key	实际使用的索引
Key_len	索引中使用的字节数
ref	显示索引的哪一列被使用了
rows	估算的找到所需的记录所需要读取的行数
filtered	通过条件过滤出的行数的百分比估计值
extra	包含不适合在其他列中显示但十分重要的额外信息

1.2.2.3. id

包含一组数字，表示查询中执行 select 子句或操作表的顺序，id 相同执行顺序由上至下。如果是子查询，id 的序号会递增，id 值越大优先级越高，越先被执行

1.2.2.4. select_type

所使用的 SELECT 查询类型，包括以下常见类型：

- SIMPLE：表示为简单的 SELECT，查询中不包含子查询或者 UNION
- PRIMARY：查询中若包含任何复杂的子部分，最外层查询则被标记为 PRIMARY
- SUBQUERY：在 SELECT 或 WHERE 列表中包含了子查询，该子查询被标记为 SUBQUERY

d. UNION：表连接中的第二个或后面的 select 语句，若第二个 SELECT 出现在 UNION 之后，则被标记为 UNION。

e. DERIVED：DERIVED（衍生）用来表示包含在 from 子句中的子查询的 select。若 UNION 包含在 FROM 子句的子查询中，外层 SELECT 将被标记为 DERIVED。mysql 会递

归执行并将结果放到一个临时表中。服务器内部称为“派生表”，因为该临时表是从子查询中派生出来的

f.UNION RESULT：从 UNION 表获取结果的 SELECT 被标记为 UNION RESULT

g.DEPENDENT：意味着 select 依赖于外层查询中发现的数据。

h.UNCACHEABLE：意味着 select 中的某些特性阻止结果被缓存于一个 item_cache 中。

1.2.2.5. table

所使用的的数据表的名字，他们按被读取的先后顺序排列。

1.2.2.6. type

表示 MySQL 在表中找到所需行的方式，又称“访问类型”。取值按优劣排序为 NULL>system > const > eq_ref > ref > fulltext > ref_or_null > index_merge > unique_subquery > index_subquery > range > index > ALL。一般来说，得保证查询至少达到 range 级别，最好能达到 ref。

a.ALL：Full Table Scan 全表扫描，MySQL 将遍历全表以找到匹配的行。

b.index：Full Index Scan 全索引扫描，index 与 ALL 区别为 index 类型只遍历索引树

c. range：索引范围扫描，对索引的扫描开始于某一点，返回匹配值域的行。显而易见的索引范围扫描是带有 between 或者 where 子句里带有 <, > 查询。当 mysql 使用索引去查找一系列值时，例如 IN() 和 OR 列表，也会显示 range（范围扫描），当然性能上面是有差异的。

d. ref_or_null：该联接类型如同 ref，但是添加了 MySQL 可以专门搜索包含 NULL 值的行。

e. index_merge：该联接类型表示使用了索引合并优化方法。

f. `unique_subquery`: 该类型替换了下面形式的 IN 子查询的 `ref: value IN (SELECT primary_key FROM single_table WHERE some_expr)`。 `unique_subquery` 是一个索引查找函数,可以完全替换子查询,效率更高。

g. `index_subquery`: 该联接类型类似于 `unique_subquery`。可以替换 IN 子查询,但只适合下列形式的子查询中的非唯一索引: `value IN (SELECT key_column FROM single_table WHERE some_expr)`

h.`ref`: 就是连接程序无法根据键值只取得一条记录,使用索引的最左前缀或者索引不是 `primary key` 或 `unique` 索引的情况。当根据键值只查询到少数几条匹配的记录时,这就是一个不错的连接类型。

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

j.`const`、`system`: 当 MySQL 对查询某部分进行优化,并转换为一个常量时,使用这些类型访问。如将主键置于 `where` 列表中,MySQL 就能将该查询转换为一个常量。

注: `system` 是 `const` 类型的特例,当查询的表只有一行的情况下,使用 `system`

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

`explain select * from address where id = (select min(id) from person);`

信息	结果1	概况	状态								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	address	(Null)	const	PRIMARY	PRIMAR 8		const	1	100	(Null)
2	SUBQUERY	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	Select tables optimized away

1.2.2.7. possible_keys

指出 MySQL 能使用哪个索引在表中找到记录,查询涉及到的字段上若存在索引,则该索引将被列出,但不一定被查询使用

1.2.2.8. key

显示 MySQL 在查询中实际使用的索引，若没有使用索引，显示为 NULL

1.2.2.9. key_len

表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度（key_len 显示的值为索引字段的最大可能长度，并非实际使用长度。如果键是 NULL,则长度为 NULL。

1.2.2.10. ref

显示索引的哪一列被使用了，有时候会是一个常量：表示哪些列或常量被用于用于查找索引列上的值，可能值为库.表.字段、常量、null。

1.2.2.11. rows

MySQL 根据表统计信息及索引选用情况，估算的找到所需的记录所需要读取的行数。

1.2.2.12. filtered

显示了通过条件过滤出的行数的百分比估计值。

1.2.2.13. extra

包含不适合在其他列中显示但十分重要的额外信息，提供了与关联操作有关的信息，没有则什么都不写。

a.Using index：该值表示相应的 select 操作中使用了覆盖索引（Covering Index）。MySQL 可以利用索引返回 select 列表中的字段，而不必根据索引再次读取数据文件包含所有满足查询需要的数据的索引称为覆盖索引（Covering Index）。注意：如果要使用覆盖索引，一定要注意 select 列表中只取出需要的列，不可 select *，因为如果将所有字段一起做索引会导致索引文件过大，查询性能下降。

b.Using where：表示 mysql 服务器将在存储引擎检索行后再进行过滤。许多 where 条件里涉及索引中的列，当（并且如果）它读取索引时，就能被存储引擎检验，因此不是

所有带 where 字句的查询都会显示 "Using where"。有时 "Using where" 的出现就是一个暗示：查询可受益与不同的索引。

c. Using temporary：表示 MySQL 需要使用临时表来存储结果集，常见于排序和分组查询。这个值表示使用了内部临时(基于内存的)表。一个查询可能用到多个临时表。有很多原因都会导致 MySQL 在执行查询期间创建临时表。两个常见的原因是在来自不同表的上使用了 DISTINCT, 或者使用了不同的 ORDER BY 和 GROUP BY 列。可以强制指定一个临时表使用基于磁盘的 MyISAM 存储引擎。这样做的原因主要有两个：1) 内部临时表占用的空间超过 $\min(\text{tmp_table_size}, \text{max_heap_table_size})$ 系统变量的限制；2) 使用了 TEXT/BLOB 列。

d. Using filesort：MySQL 中无法利用索引完成的排序操作称为“文件排序”

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

f. Impossible where：这个值强调了 where 语句会导致没有符合条件的行。

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

i. Index merges：当 MySQL 决定要在一个给定的表上使用超过一个索引的时候，就会出现以下格式中的一个，详细说明使用的索引以及合并的类型。

Using sort_union(...)

Using union(...)

Using intersect(...)

1. 2. 2. 14. 小结

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

情况

1.2.3. Profiling 的使用

要想优化一条 Query，就须要清楚这条 Query 的性能瓶颈到底在哪里，是消耗的 CPU 计算太多，还是需要的 IO 操作太多？要想能够清楚地了解这些信息，可以通过 Query Profiler 功能得到。

Query Profiler 是 MYSQL 自带的一种 query 诊断分析工具，通过它可以分析出一条 SQL 语句的性能瓶颈在什么地方。通常我们是使用的 explain,以及 slow query log 都无法做到精确分析,但是 Query Profiler 却可以定位出一条 SQL 语句执行的各种资源消耗情况,比如 CPU, IO 等,以及该 SQL 执行所耗费的时间等。

用法

- (1) 通过执行“set profiling”命令，可以开启关闭 QueryProfiler 功能

```
mysql> SET global profiling=on;
```

- (2) 查看相关变量

```
show VARIABLES like '%profiling%';
```

- (3) 设置保存数量默认 15 条，最大值为 100

```
mysql> set profiling_history_size=100;
```

(4) 在开启 Query Profiler 功能之后，MySQL 就会自动记录所有执行的 Query 的 profile 信息，下面执行 n 条 Query 作为测试

```
select * from person limit 10000,100;
```

(3) 获取当前系统中保存的多个 Query 的 profile 的概要信息

```
mysql> show profiles;
```

信息	结果1	概况	状态
Query_ID	Duration	Query	
490	0.00307825	SELECT QUERY_ID, SUM(DURATION) AS SUM_DURATION FROM INFORMATION_SCHEMA.PROFILING GROUP BY QUERY_ID	
491	0.0025865	SELECT STATE AS `状态`, ROUND(SUM(DURATION),7) AS `期间`, CONCAT(ROUND(SUM(DURATION)/0.018484*100,3), '%') AS `百分比`	
492	0.0002385	SELECT * FROM `pythonlearn`.`lianjia` LIMIT 0	
493	0.0277085	SHOW COLUMNS FROM `pythonlearn`.`lianjia`	
494	0.00014	SET PROFILING=1	
495	0.002325	SHOW STATUS	
496	0.0020545	SHOW STATUS	
497	0.017557	select * from pythonlearn.lianjia limit 10000,100	
498	0.00118075	SHOW STATUS	
499	0.0041345	SELECT QUERY_ID, SUM(DURATION) AS SUM_DURATION FROM INFORMATION_SCHEMA.PROFILING GROUP BY QUERY_ID	
500	0.003488	SELECT STATE AS `状态`, ROUND(SUM(DURATION),7) AS `期间`, CONCAT(ROUND(SUM(DURATION)/0.017560*100,3), '%') AS `百分比`	
501	0.00024775	SELECT * FROM `pythonlearn`.`lianjia` LIMIT 0	
502	0.0005785	SHOW COLUMNS FROM `pythonlearn`.`lianjia`	
503	0.00021275	SET PROFILING=1	
504	0.00182775	SHOW STATUS	
505	0.00097525	SHOW STATUS	

(4) 针对单个 Query 获取详细的 profile 信息。

可以根据概要信息中的 Query_ID 来获取某个 Query 在执行过程中详细的 profile 信息。例如查看 cpu 和 io 的详细信息

```
show profile cpu,block io for query 501;
```

```
5 show profile cpu,block io for query 501;
6
```

信息	结果1	概况	状态
----	-----	----	----

Status	Duration	CPU_user	CPU_system	Block_ops_in	Block_ops_out
starting	4.8E-5	0	0	(Null)	(Null)
checking permissions	5E-6	0	0	(Null)	(Null)
Opening tables	1.2E-5	0	0	(Null)	(Null)
init	1.8E-5	0	0	(Null)	(Null)
System lock	7E-6	0	0	(Null)	(Null)
optimizing	3E-6	0	0	(Null)	(Null)
executing	1.2E-5	0	0	(Null)	(Null)
end	2E-6	0	0	(Null)	(Null)
query end	5E-6	0	0	(Null)	(Null)
closing tables	1.7E-5	0	0	(Null)	(Null)
freeing items	7.3E-5	0	0	(Null)	(Null)
cleaning up	4.6E-5	0	0	(Null)	(Null)

show profile ALL for query 501;

ALL : 显示所有信息

|BLOCK IO : 块设备 IO 输入输出次数

|CONTEXT SWITCHES: 上下文切换相关开销

|CPU: 用户和系统的 CPU 使用情况

|IPC: 显示发送和接收消息的相关消耗

|MEMORY: 内存消耗情况 (该版本 is not currently implemented)

|PAGE FAULTS: 显示主要和次要页面故障相关的开销

|SOURCE: 显示和 Source_function,Source_file,Source_line 相关的开销信息

|SWAPS: 显示交换次数相关的开销

注意: profiling 被应用在每一个会话中, 当前会话关闭后, profiling 统计的信息将丢失。

1.2.4. last_query_cost

查上一个查询的代价，而且它是 io_cost 和 cpu_cost 的开销总和，它通常也是我们评价一个查询的执行效率的一个常用指标。last_query_cost 对于简单的查询可以精确的得到计算，但于包含子查询或 union 的复杂查询值是 0。

```
show status like 'last_query_cost';
```

1.2.5. timestampdiff 查看执行时间

这种方法有一点要注意，就是三条 sql 语句要尽量连一起执行，不然误差太大，根本不准。

```
set @d=now();
```

```
select id from person where lname='x8RJWmQX';
```

```
select timestampdiff(second,@d,now());
```

如果是用命令行来执行的话，有一点要注意，就是在 select timestampdiff(second,@d,now());后面，一定要多 copy 一个空行，不然最后一个 sql 要你自己按回车执行，这样就不准了。

1.2.6. 第三方工具查看执行时间

第三方 MySQL 客户端工具都自带 sql 执行时间显示功能，如 navicat、sqlyog 等等。

1.2.7. 数据库连接进程列表

```
show processlist;
```

```
8 show processlist;
```

id	User	Host	db	Command	Time	State	Info
3	root	localhost:56188	course1	Query	0	starting	show processlist
4	root	localhost:56932	course1	Sleep	17		(Null)
7	root	localhost:58058	course1	Sleep	19679		(Null)
8	root	localhost:58068	course1	Sleep	9036		(Null)
11	root	localhost:58827	(Null)	Sleep	5948		(Null)
12	root	localhost:59021	course1	Sleep	7109		(Null)
13	root	localhost:59157	pythonlearn	Sleep	18866		(Null)
16	root	localhost:59216	pythonlearn	Sleep	16571		(Null)
19	root	localhost:58604	course1	Sleep	9144		(Null)
23	root	localhost:60332	course1	Sleep	7924		(Null)
24	root	localhost:63151	course1	Sleep	5936		(Null)

1.3. 数据库引擎

1.3.1. 引擎介绍

MySQL 中的数据用各种不同的技术存储在文件（或者内存）中。这些技术中的每一种技术都使用不同的存储机制、索引技巧、锁定水平并且最终提供广泛的不同的功能和能力。通过选择不同的技术，你能够获得额外的速度或者功能，从而改善你的应用的整体功能。

这些不同的技术以及配套的相关功能在 MySQL 中被称作存储引擎(也称作表类型)。MySQL 默认配置了许多不同的存储引擎，可以预先设置或者在 MySQL 服务器中启用。你可以选择适用于服务器、数据库和表格的存储引擎，以便在选择如何存储你的信息、如何检索这些信息以及你需要你的数据结合什么性能和功能的时候为你提供最大的灵活性。

1.3.1.1. 引擎定义

数据库引擎是用于存储、处理和保护数据的核心服务。利用数据库引擎可控制访问权限并快速处理事务，从而满足企业内大多数需要处理大量数据的应用程序的要求。使用数据库引擎创建用于联机事务处理或联机分析处理数据的关系数据库。这包括创建用于存储数据的表和用于查看、管理和保护数据安全的数据对象（如索引、视图和存储过程）。

1.3.1.2. 引擎作用

- 1) 设计并创建数据库以保存系统所需的关系或 XML 文档。

- 2) 实现系统以访问和更改数据库中存储的数据。包括实现网站或使用数据的应用程序，还包括生成使用 SQL Server 工具和实用工具以使用数据的过程。
- 3) 为单位或客户部署实现的系统。
- 4) 提供日常管理支持以优化数据库的性能。

1.3.1.3. 引擎种类

存储引擎主要有： 1. MyIsam , 2. InnoDB, 3. Memory, 4. Blackhole, 5. CSV, 6. Performance_Schema, 7. Archive, 8. Federated , 9 Mrg_Myisam。在实际工作中用的比较多的是 MyIsam 和 InnoDB, 而 InnoDB 更因为各方面的优越性更是获得了更多的青睐。尤其是 MySQL 8.0 版本发布后，其中具有重大意义的是官方废弃了 MyISAM 存储引擎。

参考：<https://www.cnblogs.com/sunsky303/p/8274586.html>

查看当前数据库支持的引擎可以用一下语句：

show ENGINES;

Engine	Support	Comment	Transactions	XA	Savepoints
InnoDB	DEFAULT	Supports transactions, row-level locking	YES	YES	YES
MRG_MYISAM	YES	Collection of identical MyISAM tables	NO	NO	NO
MEMORY	YES	Hash based, stored in memory	NO	NO	NO
BLACKHOLE	YES	/dev/null storage engine	NO	NO	NO
MyISAM	YES	MyISAM storage engine	NO	NO	NO
CSV	YES	CSV storage engine	NO	NO	NO
ARCHIVE	YES	Archive storage engine	NO	NO	NO
PERFORMANCE_SCHEMA	YES	Performance Schema	NO	NO	NO
FEDERATED	NO	Federated MySQL storage engine	(Null)	(Null)	(Null)

1.3.1.4. 修改数据库引擎

方式一 修改配置文件 my.ini:

在[mysqld]后面添加 default-storage-engine=InnoDB，重启服务，数据库默认的引擎修改为 InnoDB。

方式二 在建表的时候指定

```
create table mytbl(  
    id int primary key,  
    name varchar(50)  
)ENGINE=MyISAM;
```

方式三 建表后更改

```
alter table person2 ENGINE =InnoDB;
```

1.3.1.5. 查看当前数据库引擎

方式一

```
show table status from table_name;
```

方式二

```
show create table table_name
```

方式三

使用第三方数据库管理工具。

1.3.2. InnoDB

1.3.2.1. 定义

InnoDB 是一个事务型的存储引擎，有行级锁定和外键约束。

InnoDB 引擎提供了对数据库 ACID 事务（原子性 Atomicity、一致性 Consistency、隔离性 Isolation、持久性 Durability）的支持，并且实现了 SQL 标准的四种隔离级别，关于数据库事务与其隔离级别的内容请见数据库事务与其隔离级别这类型的文章。该引擎还提供了行级锁和外键约束，它的设计目标是处理大容量数据库系统，它本身其实就是基于 MySQL 后台的完整数据库系统，MySQL 运行时 InnoDB 会在内存中建立缓冲池，用于缓冲数据和索引。但是该引擎不支持 FULLTEXT 类型的索引，而且它没有保存表的行数，当 `SELECT COUNT(*) FROM TABLE` 时需要扫描全表。当需要使用数据库事务时，该引擎当然是首选。由于锁的粒度更小，写操作不会锁定全表，所以在并发较高时，使用 InnoDB 引擎会提升效率。但是使用行级锁也不是绝对的，如果在执行一个 SQL 语句时 MySQL 不能确定要扫描的范围，InnoDB 表同样会锁全表。

-- 这个就是 select 锁表的一种，不明确主键。增删改查都可能会导致锁全表。

```
SELECT * FROM products WHERE name='Mouse' FOR UPDATE;
```

1.3.2.2. 适用场景

- 1) 经常更新的表，适合处理多重并发的更新请求。
- 2) 支持事务。
- 3) 可以从灾难中恢复（通过 bin-log 日志等）。
- 4) 外键约束。只有他支持外键。
- 5) 支持自动增加列属性 auto_increment。

1.3.2.3. 官方对 InnoDB 的讲解

1) InnoDB 给 MySQL 提供了具有提交、回滚和崩溃恢复能力的事务安全（ACID 兼容）存储引擎。

2) InnoDB 锁定在行级并且也在 SELECT 语句提供一个 Oracle 风格一致的非锁定读，这些特色增加了多用户部署和性能。没有在 InnoDB 中扩大锁定的需要，因为在 InnoDB 中行级锁定适合非常小的空间。

3) InnoDB 也支持 FOREIGN KEY 强制。在 SQL 查询中，你可以自由地将 InnoDB 类型的表与其它 MySQL 的表的类型混合起来，甚至在同一个查询中也可以混合。

4) InnoDB 是为处理巨大数据量时的最大性能设计，它的 CPU 效率可能是任何其它基于磁盘的关系数据库引擎所不能匹敌的。

- 5) InnoDB 被用来在众多需要高性能的大型数据库站点上产生。

1.3.3. MyIsam

1.3.3.1. 定义

MyIASM 没有提供对数据库事务的支持，也不支持行级锁和外键，因此当 INSERT(插入)或 UPDATE(更新)数据时即写操作需要锁定整个表，效率便会低一些。

MyIsam 存储引擎独立于操作系统，也就是可以在 windows 上使用，也可以比较简单的将数据转移到 linux 操作系统上去。

意味着：引擎在创建表的时候，会创建三个文件，一个是.frm 文件用于存储表的定义，一个是.MYD 文件用于存储表的数据，另一个是.MYI 文件，存储的是索引。操作系统对大文件的操作是比较慢的，这样将表分为三个文件，那么.MYD 这个文件单独来存放数据自然可以优化数据库的查询等操作。有索引管理和字段管理。MyISAM 还使用一种表格锁定的机制，来优化多个并发的读写操作，其代价是你需要经常运行 OPTIMIZE TABLE 命令，来恢复被更新机制所浪费的空间。

1.3.3.2. 适用场景

- 1) 不支持事务的设计，但是并不代表着有事务操作的项目不能用 MyIsam 存储引擎，可以在 service 层进行根据自己的业务需求进行相应的控制。
- 2) 不支持外键的表设计。
- 3) 查询速度很快，如果数据库 insert 和 update 的操作比较多的话比较适用。
- 4) 整天 对表进行加锁的场景。
- 5) MyISAM 极度强调快速读取操作。
- 6) MyIASM 中存储了表的行数，于是 SELECT COUNT(*) FROM TABLE 时只需要直接读取已经保存好的值而不需要进行全表扫描。如果表的读操作远远多于写操作且不需要数据库事务的支持，那么 MyIASM 也是很好的选择。

1.3.3.3. 特性

ISAM 执行读取操作的速度很快，而且不占用大量的内存和存储资源。

在设计之初就预想数据组织成有固定长度的记录，按顺序存储的。ISAM 是一种静态索引结构。

1.3.3.4. 缺点

1.它不支持事务处理

2.也不能够容错。如果你的硬盘崩溃了，那么数据文件就无法恢复了。如果你正在把 ISAM 用在关键任务应用程序里，那就必须经常备份你所有的实时数据，通过其复制特性，MYSQL 能够支持这样的备份应用程序。

1.3.4. Memory (也叫 HEAP) 堆内存

1.3.4.1. 定义

使用存在内存中的内容来创建表。每个 MEMORY 表只实际对应一个磁盘文件。MEMORY 类型的表访问非常得快，因为它的数据是放在内存中的，并且默认使用 HASH 索引。

但是一旦服务关闭，表中的数据就会丢失掉。HEAP 允许只驻留在内存里的临时表格。驻留在内存里让 HEAP 要比 ISAM 和 MYISAM 都快，但是它所管理的数据是不稳定的，而且如果在关机之前没有进行保存，那么所有的数据都会丢失。在数据行被删除的时候，HEAP 也不会浪费大量的空间。HEAP 表格在你需要使用 SELECT 表达式来选择和操控数据的时候非常有用。

1.3.4.2. 适用场景

1) 那些内容变化不频繁的代码表，或者作为统计操作的中间结果表，便于高效地堆中间结果进行分析并得到最终的统计结果。

2) 目标数据比较小，而且非常频繁的进行访问，在内存中存放数据，如果太大的数据会造成内存溢出。可以通过参数 max_heap_table_size 控制 Memory 表的大小，限制 Memory 表的最大的大小。

3) 数据是临时的，而且必须立即可用得到，那么就可以放在内存中。

4) 存储在 Memory 表中的数据如果突然间丢失的话也没有太大的关系。

注意：Memory 同时支持散列索引和 B 树索引，B 树索引可以使用部分查询和通配查询，也可以使用 <、> 和 >= 等操作符方便数据挖掘，散列索引相等的比较快但是对于范围的比较慢很多。

1.3.4.3. 特性要求

1) 要求存储的数据是数据长度不变的格式，比如，Blob 和 Text 类型的数据不可用（长度不固定的）。

2) 要记住，在用完表格之后就删除表格。

1.3.5. Mrg_Myisam（分表的一种方式-水平分表）

1.3.5.1. 定义

是一个相同的可以被当作一个来用的 MyISAM 表的集合。“相同”意味着所有表同样的列和索引信息。

也就是说，他将 MyIsam 引擎的多个表聚合起来，但是他的内部没有数据，真正的数据依然是 MyIsam 引擎的表中，但是可以直接进行查询、删除更新等操作。

比如：我们可能会遇到这样的问题，同一种类的数据会根据数据的时间分为多个表，如果这时候进行查询的话，就会比较麻烦，Merge 可以直接将多个表聚合成一个表统一查询，然后再删除 Merge 表（删除的是定义），原来的数据不会受影响。

1.3.6. Blackhole（黑洞引擎）

1.3.6.1. 定义

任何写入到此引擎的数据均会被丢弃掉，不做实际存储；Select 语句的内容永远是空。他会丢弃所有的插入的数据，服务器会记录下 Blackhole 表的日志，所以可以用于复制数据到备份数据库。

1.3.6.2. 使用场景

- 1) 验证 dump file 语法的正确性
- 2) 以使用 blackhole 引擎来检测 binlog 功能所需要的额外负载

```
CREATE TABLE `Blackhole` (  
  `id` bigint(20) unsigned NOT NULL,  
  `fname` varchar(100) NOT NULL,  
  `lname` varchar(100) NOT NULL,  
  `age` tinyint(3) unsigned NOT NULL,  
  `sex` tinyint(1) unsigned NOT NULL,  
  PRIMARY KEY (`id`)
```

```
) ENGINE=Blackhole DEFAULT CHARSET=utf8
```

1.3.7. Innodb 对比 mylsam

1.3.7.1. 事务

MyISAM 类型不支持事务处理等高级处理，而 InnoDB 类型支持，提供事务支持已经外部键等高级数据库功能。

InnoDB 表的行锁也不是绝对的，假如在执行一个 SQL 语句时 MySQL 不能确定要扫描的范围，InnoDB 表同样会锁全表，例如 `updatetable set num=1 where name like "a%"`，就是说在不确定的范围时，InnoDB 还是会锁表的。

1.3.7.2. 性能

以前版本中 MyISAM 类型的表强调的是性能，其执行数度比 InnoDB 类型更快。但现在 InnoDB 在多方面的性能已经赶上活超过了 Mysqlam。

1.3.7.3. 行数保存

InnoDB 中不保存表的具体行数，也就是说，执行 `select count() fromtable` 时，InnoDB 要扫描一遍整个表来计算有多少行，但是 MyISAM 只要简单的读出保存好的行数即可。注意的是，当 `count()` 语句包含 `where` 条件时，两种表的操作是一样的。

1.3.7.4. 索引存储

对于 AUTO_INCREMENT 类型的字段，InnoDB 中必须包含只有该字段的索引，但是在 MyISAM 表中，可以和其他字段一起建立联合索引。

MyISAM 支持全文索引 (FULLTEXT)、压缩索引，InnoDB 不支持

MyISAM 的索引和数据是分开的，并且索引是有压缩的，内存使用率就对应提高了不少。能加载更多索引，而 InnoDB 是索引和数据是紧密捆绑的，没有使用压缩从而会造成 InnoDB 比 MyISAM 体积庞大不小。

InnoDB 存储引擎被完全与 MySQL 服务器整合，InnoDB 存储引擎为在主内存中缓存数据和索引而维持它自己的缓冲池。InnoDB 存储它的表 & 索引在一个表空间中，表空间可以包含数个文件（或原始磁盘分区）。这与 MyISAM 表不同，比如在 MyISAM 表中每个表被存在分离的文件中。InnoDB 表可以是任何尺寸，即使在文件尺寸被限制为 2GB 的操作系统上。

1.3.7.5. 服务器数据备份

InnoDB 必须导出 SQL 来备份，LOAD TABLE FROM MASTER 操作对 InnoDB 是不起作用的，解决方法是首先把 InnoDB 表改成 MyISAM 表，导入数据后再改成 InnoDB 表，但是对于使用的额外的 InnoDB 特性(例如外键)的表不适用。

而且 MyISAM 应对错误编码导致的数据恢复速度快。MyISAM 的数据是以文件的形式存储，所以在跨平台的数据转移中会很方便。在备份和恢复时可单独针对某个表进行操作。

InnoDB 是拷贝数据文件、备份 binlog，或者用 mysqldump，在数据量达到几十 G 的时候就相对痛苦了。

1.3.7.6. 锁的支持

MyISAM 只支持表锁。InnoDB 支持表锁、行锁 行锁大幅度提高了多用户并发操作的新能。但是 InnoDB 的行锁，只是在 WHERE 的主键是有效的，非主键的 WHERE 都会锁全表的。

1.3.7.7. 使用建议

以下两点必须使用 InnoDB：

- 1) 可靠性高或者要求事务处理，则使用 InnoDB。这个是必须的。
- 2) 表更新和查询都相当的频繁，并且表锁定的机会比较大的情况指定 InnoDB 数据引擎的创建。

对比之下，MyISAM 的使用场景：

- 1) 做很多 count 的计算的。如一些日志，调查的业务表。
- 2) 插入修改不频繁，查询非常频繁的。

MySQL 能够允许你在表这一层应用数据库引擎，所以你可以只对需要事务处理的表格来进行性能优化，而把不需要事务处理的表格交给更加轻便的 MyISAM 引擎。对于 MySQL 而言，灵活性才是关键。

1.4. sql 语句优化

1.4.1. 常用 sql 优化建议

1.4.1.1. 避免 SELECT *

从数据库里读出越多的数据，那么查询就会变得越慢。并且如果你的数据库服务器和 WEB 服务器是两台独立的服务器的话，这还会增加网络传输的负载。

```
select * from person where lname='x8RJWmQX';
```

```
select id from person where lname='x8RJWmQX';
```

```
mysql> select * from person where lname='x8RJWmQX';
```

id	fname	lname	age	sex
5012	POomue	x8RJWmQX	91	0

```
1 row in set (0.38 sec)
```

```
mysql> select id from person where lname='x8RJWmQX';
```

id
5012

```
1 row in set (0.29 sec)
```

1.4.1.2. 避免在 where 子句中使用!=或<>操作符

应尽量避免在 where 子句中使用!=或<>操作符，否则引擎放弃使用索引而进行全表扫描。

```
EXPLAIN select * from person where fname != 'sss';
```

信息	结果1	概况	状态								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	person	(Null)	ALL	index_fname	(Null)	(Null)	(Null)	997528	95.74	Using whe

1.4.1.3. 尽量避免全表扫描

对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。

1.4.1.4. 用 UNION 来代替 OR

采用 OR 语句：

```
select * from person where fname = 'LVc1oJjd' or fname = 'bjRdlVo';
```

采用 UNION 语句，返回的结果同上面的一样，但是速度要快些：

```
select * from person where fname = 'LVc1oJjd'
```

Union

```
select * from person where fname='bjRdlVo';
```

分别对这两个 sql 进行 explain 分析：

OR 语句的结果

20

信息	结果1	结果2	概况	状态							
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	person	(Null)	range	index_fname	index fname	303	(Null)	2	100	Using index condition

UNION 语句的结果

信息	结果1	结果2	概况	状态							
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	person	(Null)	ref	index_fname	index_fname	303	const	1	100	(Null)
2	UNION	person	(Null)	ref	index_fname	index_fname	303	const	1	100	(Null)
(Null)	UNION RESULT	<union1,2	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	Using temporary table

我们来比较下重要指标，发现主要差别是 type 和 ref 这两项。type 显示的是访问类型，是较为重要的一个指标，结果值从好到坏依次是：

system > const > eq_ref > ref > fulltext > ref_or_null > index_merge >

unique_subquery > index_subquery > range > index > ALL

UNION 语句的 type 值为 一般为 ref，OR 语句的 type 值为 range，可以看到这是一个很明显的差距。

UNION 语句的 ref 值为 const，OR 语句的 type 值为 null，const 表示是常量值引用，非常快。

这两项的差距就说明了 UNION 要优于 OR，从我们的直观感觉上也可以理解，虽然这两个方式都用到了索引，但 UNION 是用一个明确的值到索引中查找，目标非常明确，OR 需要对比两个值，目标相对要模糊一些，所以 OR 在恍惚中落后了。

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
708	PRIMARY	person	(Null)	ref	index_fname	index_fname	303	const	1	100	(Null)
709	UNION	person	(Null)	ref	index_fname	index_fname	303	const	1	100	(Null)
710	UNION RESULT	<union1,2	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	Using temporary table

1.4.1.5. like 语句避免前置百分号

前置百分号会导致索引失效

```
select * from person where fname like '%LVc1o%';
```

24. EXPLAIN select * from person where fname like 'LVc1o%' union select * from person where fname like 'hiRd'

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	person	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	997528	11.11	Using whe

下面走索引

```
select * from person where fname like 'LVc1o%';
```

24. EXPLAIN select * from person where fname like 'LVc1o%' union select * from person where fname like 'hiRd'

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	person	(Null)	range	index_fname	index_fn 303		(Null)	4		100 Using indi

1.4.1.6. 避免 where 子句中使用参数

如果在 where 子句中使用参数，也会导致全表扫描。因为 SQL 只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。

然而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入

项。如下面语句将进行全表扫描：

```
select id from t where num=@num
```

可以改为强制查询使用索引：

```
select id from t with(index(索引名)) where num=@num
```

1.4.1.7. 避免在 where 子句中对字段进行表达式操作

应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：

```
select id from t where num/2=100
```

应改为：

```
select id from t where num=100*2
```

1.4.1.8. 避免在 where 子句中对字段进行函数操作

应尽量避免在 where 子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：

```
select id from t where substring(name,1,3)='abc' --name 以 abc 开头的 id
```

```
select id from t where datediff(day,createdate,'2005-11-30')=0 --'2005-11-30'生成的 id
```

应改为：

```
select id from t where name like 'abc%'
```

```
select id from t where createdate>='2005-11-30' and createdate<'2005-12-1'
```

1.4.1.9. 避免无意义查询

不要写一些没有意义的查询，如需要生成一个空表结构：

```
select col1,col2 into #t from t where 1=0
```

这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样：

```
create table #t(...)
```

1.4.1.10. 用 exists 代替 in

很多时候用 exists 代替 in 是一个好的选择：

```
select num from a where num in(select num from b)
```

用下面的语句替换：

```
select num from a where exists(select 1 from b where num=a.num)
```

1.4.1.11. 尽量使用数字型字段

尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

1.4.1.12. 使用 varchar/nvarchar 代替 char/nchar

尽可能的使用 varchar/nvarchar 代替 char/nchar，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

1.4.1.13. 大临时表使用 select into 代替 create table

在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log，以提高速度；如果数据量不大，为了缓和系统表的资源，应先 create table，然后 insert。

1.4.1.14. 临时表先 truncate table，然后 drop table

如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table，然后 drop table，这样可以避免系统表的较长时间锁定。

1.4.1.15. 存储过程使用 SET NOCOUNT ON

在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON，在结束时设置 SET NOCOUNT OFF。无需在执行存储过程和触发器的每个语句后向客户端发送 DONEINPROC 消息。

1.4.1.16. 避免向客户端返回大数据量

尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

1.4.1.17. 避免在 where 子句中对字段进行 null 值判断

应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描。如：

```
select id from t where num is null
```

可以在 num 上设置默认值 0，确保表中 num 列没有 null 值，然后这样查询：

```
select id from t where num=0
```


在 Mysql5.7 版本中该条建议已经不用考虑了，因为 null 判断也能使用索引了。

```

14 SHOW PROFILE ALL FOR QUERY 303;
15
16 EXPLAIN select * from person where fname is null;
17
18

```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	person	(Null)	ref	index_fname	index_fn 303		const	1		100 Using index condition

1.4.2. Join 语句的优化

1.4.2.1. 尽可能减少 Join 语句中 Nested Loop 的循环总次数

最有效的办法是让驱动表的结果集尽可能地小，“永远用小结果集驱动大结果集”。

比如，当两个表（表 A 和表 B）Join 时，如果表 A 通过 WHERE 条件过滤后有 10 条记录，而表 B 有 20 条记录。如果选择表 A 作为驱动表，也就是被驱动表的结果集为 20，那么我们通过 Join 条件对被驱动表（表 B）的比较过滤就会进行 10 次。反之，如果选择表 B 作为驱动表，则须要进行 20 次对表 A 的比较过滤。

1.4.2.2. 优先优化 Nested Loop 的内层循环

不仅在数据库的 Join 中应该这样做，实际上在优化程序语言时也有类似的优化原则。

内层循环是循环中执行次数最多的，每次循环节约很少的资源，就能在整个循环中节约很多的资源

1.4.2.3. 保证 Join 语句中被驱动表的 Join 条件字段已经被索引

其目的正是基于上面两点的考虑，只有让被驱动表的 Join 条件字段被索引了，才能保证循环中每次查询都能够消耗较少的资源，这也正是内层循环的实际优化方法

1.4.2.4. 不要太吝惜 Join Buffer 的设置

当无法保证被驱动表的 Join 条件字段被索引且内存资源充足时，不要太吝惜 Join Buffer 的设置。在 Join 是 All、Index、range 或 index_merge 类型的特殊情况下，Join Buffer

才能派上用场。在这种情况下，Join Buffer 的大小将对整个 Join 语句的消耗起到非常关键的作用

1.4.3. GROUP BY 关键字优化

1、group by 实质是先排序后分组，遵照索引的最佳左前缀。

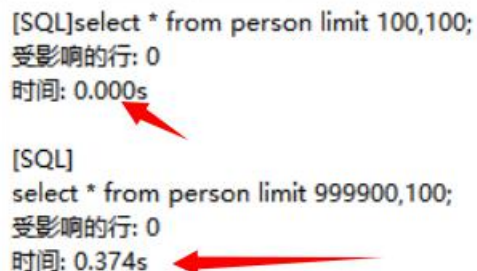
2、当无法使用索引列，增大 max_length_for_sort_data 参数的设置 + 增大 sort_buffer_size 参数的设置

3、where 高于 having，能写在 where 限定的条件就不要去 having 去限定了。

1.4.4. 大数据量的分页优化

使用 limit 进行分页，翻到 10000 多页后效率低。原因在于 limit offset 会逐行查找，是先查询再跳过。

select * from person limit 999900,100; -- 慢了，大概需要 0.4 秒多



```
[SQL]select * from person limit 100,100;
受影响的行: 0
时间: 0.000s

[SQL]
select * from person limit 999900,100;
受影响的行: 0
时间: 0.374s
```

1.4.4.1. 从业务逻辑优化

不允许翻过 100 页，例如百度一般可以翻到 70 页左右

1.4.4.2. 技术优化方法一

select * from person where id>999900 limit 100;

```
[SQL]
select * from person limit 999900,100;
受影响的行: 0
时间: 0.414s
```

```
[SQL]
select * from person where id>999900 limit 100;
受影响的行: 0
时间: 0.001s
```

这样就非常快，0.001s 左右，因为使用了 id 索引

但这样用有前提，id 是连续的，中间的数据不能删，否则 id 为 999900 的并不是第 999900 个记录。

1. 4. 4. 3. 技术优化方法二

如果必须用 limit offset 查询，就用延迟关联

```
select id from person limit 999900 ,100;
```

这样只查询 id 列，实现了索引覆盖，就会很快

```
select p.* from person p inner join (select id from person limit 999900 ,100) as tmp on
p.id=tmp.id;
```

通过内连接再获取分页后每条记录的详细信息

```
[SQL]
select * from person limit 999900,100;
受影响的行: 0
时间: 0.364s
```

```
[SQL]
select * from person where id>999900 limit 100;
受影响的行: 0
时间: 0.001s
```

```
[SQL]
select p.* from person p inner join (select id from person limit 999900 ,100) as tmp on p.id=tmp.id;
受影响的行: 0
时间: 0.225s
```

1. 4. 5. 优化更须要优化的 Query

这个问题须要从对整个系统的影响来考虑。哪个 Query 的优化能给系统整体带来更大的收益，就更须要优化。一般来说，高并发低消耗的影响 > 低并发高消耗

假设有一个 Query 每小时执行 10000 次，每次需要 20 个 IO，而另外一个 Query 每小时执行 10 次，每次需要 20000 个 IO。

(1) 通过 IO 消耗来分析

两个 Query 每小时所消耗的 IO 总数目是一样的，都是 200000 IO/小时

假设优化第一个 Query，从 20 个 IO 降低到 18 个 IO，也就是降低了 2 个 IO，则节省了 $2 \times 10000 = 20000$ (IO/小时)

而如果希望通过优化第二个 Query 达到相同的效果，必须要让每个 Query 减少 $20000 / 10 = 2000$ IO

可以看出第一个 Query 节省 2 个 IO 即可达到第二个 Query 节省 2000 个 IO 相同的效果

(2) 通过 CPU 消耗来分析

原理和上面一样，只要让第一个 Query 节省一小块资源，就可以让整个系统节省出一大块资源，尤其是在排序、分组这些对 CPU 消耗比较多的操作中更加明显

(3) 从对整个系统的影响来分析

一个频繁执行的高并发 Query 的危险性比一个低并发的 Query 要大很多，当一个低并发的 Query 执行计划有误时，所带来的影响只是该 Query 请求者的体验会变差，对整体系统的影响并不会特别突出，但是，如果一个高并发的 Query 执行计划有误，它带来的后果很可能就是灾难性的。

1.5. 索引优化

1.5.1. 什么是索引？为什么要建立索引？

索引用于快速找出在某个列中有一特定值的行，不使用索引 MySQL 必须从第一条记录开始读完整个表，直到找出相关的行，表越大查询数据所花费的时间就越多，如果表中

查询的列有一个索引，MySQL 能够快速到达一个位置去搜索数据文件，而不必查看所有数据，那么将会节省很大一部分时间。

例如：有一张 person 表，其中有 2W 条记录，记录着 2W 个人的信息。有一个 Phone 的字段记录每个人的电话号码，现在想要查询出电话号码为 xxxx 的人的信息。

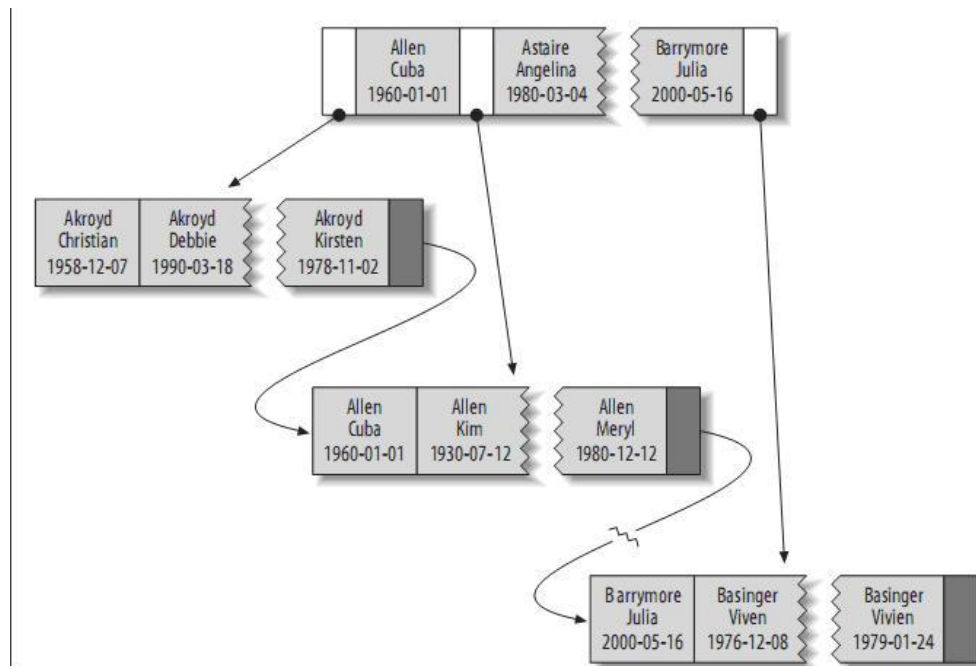
如果没有索引，那么将从表中第一条记录一条条往下遍历，直到找到该条信息为止。

如果有了索引，那么会将该 Phone 字段，通过一定的方法进行存储，好让查询该字段上的信息时，能够快速找到对应的数据，而不必在遍历 2W 条数据了。其中 MySQL 中的索引的存储类型有两种：BTREE、HASH。也就是用树或者 Hash 值来存储该字段，要知道其中详细是如何查找的，需要一定的算法知识了。

1.5.1.1. B-Tree

B-Tree 索引，它是目前关系型数据库中查找数据最为常用和有效的索引，大多数存储引擎都支持这种索引。使用 B-Tree 这个术语，是因为 MySQL 在 CREATE TABLE 或其它语句中使用了这个关键字，但实际上不同的存储引擎可能使用不同的数据结构，比如 InnoDB 就是使用的 B+Tree。

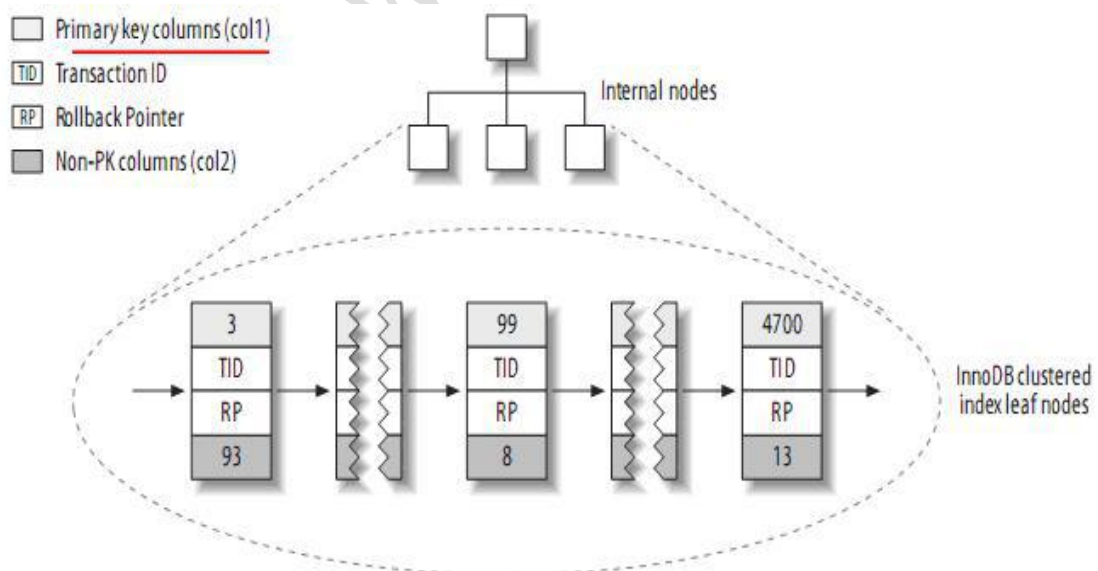
B+Tree 中的 B 是指 balance，意为平衡。需要注意的是，B+树索引并不能找到一个给定键值的具体行，它找到的只是被查找数据行所在的页，接着数据库会把页读入到内存，再在内存中进行查找，最后得到要查找的数据。



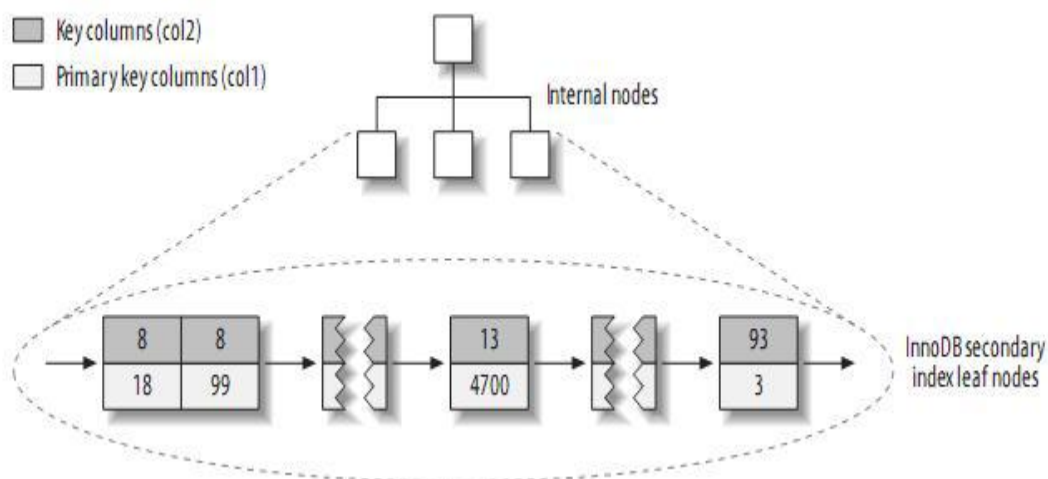
1.5.1.2. InnoDB 聚簇索引 (clustered index)

聚簇索引保证关键字的值相近的元组存储的物理位置也相同（所以字符串类型不宜建立聚簇索引，特别是随机字符串，会使得系统进行大量的移动操作），且一个表只能有一个聚簇索引。因为由存储引擎实现索引，所以，并不是所有的引擎都支持聚簇索引。

聚簇索引：



二级索引：



1.5.2. 索引的优点和缺点

1.5.2.1. 优点

- 1、所有的 MySQL 列类型(字段类型)都可以被索引，也就是可以给任意字段设置索引
- 2、大大加快数据的查询速度

1.5.2.2. 缺点

- 1、创建索引和维护索引要耗费时间，并且随着数据量的增加所耗费的时间也会增加
- 2、索引也需要占空间，我们知道数据表中的数据也会有最大上线设置的，如果我们有大量的索引，索引文件可能会比数据文件更快达到上线值
- 3、当对表中的数据进行增加、删除、修改时，索引也需要动态的维护，降低了数据的维护速度。

1.5.3. 使用原则

通过上面说的优点和缺点，我们应该可以知道，并不是每个字段度设置索引就好，也不是索引越多越好，而是需要自己合理的使用。

1.5.3.1. 并不是所有索引对查询都有效

并不是所有索引对查询都有效，SQL 是根据表中数据来进行查询优化的，当索引列有大量数据重复时，SQL 查询可能不会去利用索引，如一表中有字段 sex，male、female 几乎各一半，那么即使在 sex 上建了索引也对查询效率起不了作用。

1.5.3.2. 索引并不是越多越好

索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数较好不要超过 6 个，若太多则应考虑一些不常使用到的列上建的索引是否有必要。

1.5.3.3. 避免更新聚簇索引数据列

应尽可能的避免更新 clustered 索引数据列，mysql 默认的 clustered 索引为主键，因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。

1.5.3.4. 经常更新的表就避免对其进行过多的索引

对经常更新的表就避免对其进行过多的索引，对经常用于查询的字段应该创建索引。

1.5.3.5. 数据量小的表最好不要使用索引

数据量小的表最好不要使用索引，因为由于数据较少，可能查询全部数据花费的时间比遍历索引的时间还要短，索引就可能不会产生优化效果。

1.5.3.6. 避免在不同值少的列上加索引

在一不同值少的列上(字段上)不要建立索引，比如在学生表的"性别"字段上只有男，女两个不同值。相反的，在一个字段上不同值较多可以根据需要建立索引。

1.5.3.7. 根据业务需求建立索引

索引的建立要根据业务特点进行，不能凭空想象的设置索引。经常作为查询条件的列才有建立索引的必要性。

1.5.4. 索引的分类

索引是在存储引擎中实现的，也就是说不同的存储引擎，会使用不同的索引。MyISAM 和 InnoDB 存储引擎：只支持 B TREE 索引，也就是说默认使用 B TREE，不能够更换，MySQL5.7 中 InnoDB 可以支持 HASH 索引；MEMORY/HEAP 存储引擎：支持 HASH 和 B TREE 索引。索引可划分为单列索引（其中包括普通索引、唯一索引、主键索引）、组合索引、全文索引、空间索引，其中单列索引是一个索引只包含单个列，但一个表中可以有多个单列索引。

1.5.4.1. 普通索引

MySQL 中基本索引类型，没有什么限制，允许在定义索引的列中插入重复值和空值，纯粹为了查询数据更快一点。

Index(xx) 或者 key(xx)

1.5.4.2. 唯一索引

索引列中的值必须是唯一的，但是允许为空值，

UNIQUE INDEX UniqIdx(xx)

1.5.4.3. 主键索引

是一种特殊的唯一索引，不允许有空值。

PRIMARY KEY(id)

1.5.4.4. 组合索引

在表中的多个字段组合上创建的索引，只有在查询条件中使用了这些字段的左边字段时，索引才会被使用，使用组合索引时遵循最左前缀集合。

INDEX MultIdx(id,name,age)

由 id、name 和 age3 个字段构成的索引，索引行中就按 id/name/age 的顺序存放，索引可以索引下面字段组合(id, name, age)、(id, name)或者(id)。如果要查询的字段不构成索引最左面的前缀，那么就不会是用索引，比如，age 或者 (name, age) 组合就不会使用索引查询

1.5.4.5. 全文索引

全文索引，只有在 MyISAM 引擎上才能使用，只能在 CHAR,VARCHAR,TEXT 类型字段上使用全文索引。全文索引就是在一堆文字中，通过其中的某个关键字等，就能找到该字段所属的记录行，比如有"你是个大牛，神人 ..." 通过大牛，可能就可以找到该条记录。

这里说的是可能，因为全文索引的使用涉及了很多细节，我们只需要知道这个大概意思。

```
FULLTEXT INDEX FullTxtIdx(info)
SELECT * FROM t4 WHERE MATCH(info) AGAINST('gorlr');
```

1.5.4.6. 空间索引

只有在 MyISAM 引擎上才能使用，空间索引是对空间数据类型的字段建立的索引，MySQL 中的空间数据类型有四种，GEOMETRY、POINT、LINESTRING、POLYGON。

在创建空间索引时，使用 SPATIAL 关键字。

创建空间索引的列，必须将其声明为 NOT NULL。。

```
SPATIAL INDEX spatIdx(g)
```

1.5.5. 索引优化口诀

全值匹配我最爱，最左前缀要遵守；

带头大哥不能死，中间兄弟不能断；

索引列上少计算，范围之后全失效；

Like 百分写最右，覆盖索引不写星；

不等空值还有 or，索引失效要少用；

VAR 引号不可丢，SQL 高级也不难！

参考：<https://blog.csdn.net/zjy15203167987/article/details/81812370>

参考：<https://www.jianshu.com/p/d5b2f645d657>

1.5.6. 覆盖索引

如果索引包含满足查询的所有数据，就称为覆盖索引。覆盖索引是一种非常强大的工具，能大大提高查询性能。只需要读取索引而不用读取数据有以下一些优点：

- (1) 索引项通常比记录要小，所以 MySQL 访问更少的数据；
- (2) 索引都按值的大小顺序存储，相对于随机访问记录，需要更少的 I/O；
- (3) 大多数数据引擎能更好的缓存索引。比如 MyISAM 只缓存索引。
- (4) 覆盖索引对于 InnoDB 表尤其有用，因为 InnoDB 使用聚集索引组织数据，如果

二级索引中包含查询所需的数据，就不再需要在聚集索引中查找了。

覆盖索引不能是任何索引，只有 B-TREE 索引存储相应的值。而且不同的存储引擎实现覆盖索引的方式都不同，并不是所有存储引擎都支持覆盖索引(Memory 和 Falcon 就不支持)。

对于索引覆盖查询(index-covered query)，使用 EXPLAIN 时，可以在 Extra 一列中看到“Using index”。

1.5.6.1. 场景

产品中有一张图片表，数据量将近 100 万条，有一条相关的查询语句，由于执行频次较高，想针对此语句进行优化。表结构很简单，主要字段：

user_id 用户 ID
picname 图片名称
smallimg 小图名称

一个用户会有多条图片记录，现在有一个根据 user_id 建立的索引：uid，查询语句也很简单。取得某用户的图片集合

```
select picname, smallimg from pics where user_id = xxx;
```

1.5.6.2. 优化前

执行查询语句（为了查看真实执行时间，强制不使用缓存）

```
select SQL_NO_CACHE picname, smallimg from pics where user_id=17853;
```

执行了 10 次，平均耗时在 40ms 左右。使用 explain 进行分析

```
explain select SQL_NO_CACHE picname, smallimg from pics where user_id=17853
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	pics	ref	uid	<u>uid</u>	5	const	5618	<u>NULL</u>

使用了 user_id 的索引，并且是 const 常数查找，表示性能已经很好了

1.5.6.3. 优化后

因为这个语句太简单，sql 本身没有什么优化空间，就考虑了索引。修改索引结构，建立一个(user_id,picname,smallimg)的联合索引：uid_pic。重新执行 10 次，平均耗时降到了 30ms 左右。使用 explain 进行分析

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	pics	ref	uid,uid_pic	<u>uid_pic</u>	5	const	11780	<u>Using index</u>

看到使用的索引变成了刚刚建立的联合索引，并且 Extra 部分显示使用了'Using Index'

1.5.6.4. 总结

'Using Index'的意思是“覆盖索引”，它是使上面 sql 性能提升的关键。一个包含查询所需字段的索引称为“覆盖索引”，MySQL 只需要通过索引就可以返回查询所需要的数据，而不必在查到索引之后进行回表操作，减少 IO，提高了效率。

例如上面的 sql，查询条件是 user_id，可以使用联合索引，要查询的字段是 picname smallimg，这两个字段也在联合索引中，这就实现了“覆盖索引”，可以根据这个联合索引一次性完成查询工作，所以提升了性能

1.5.7. InnoDB 行锁优化建议

InnoDB 存储引擎由于实现了行级锁定，虽然在锁定机制的实现方面带来的性能损耗可能比表级锁定要更高一些，但是在整体并发处理能力方面是要远远优于 MyISAM 的表级锁定的。当系统并发量较高的时候，InnoDB 的整体性能和 MyISAM 相比就会有比较明显的优势了。但是当我们使用不当的时候，可能会让 InnoDB 的整体性能表现不仅不比 MyISAM 高，甚至可能会更差。

建议：

- (1) 尽可能让所有的数据检索都通过索引来完成，从而避免 InnoDB 因为无法通过索引键加锁而升级为表级锁定
- (2) 合理设计索引，让 InnoDB 在索引键上面加锁的时候尽可能准确，尽可能地缩小锁定范围，避免造成不必要的锁定而影响其他 Query 的执行
- (3) 尽可能减少基于范围的数据检索过滤条件，避免因间隙锁带来的负面影响而锁定了不该锁定的记录
- (4) 尽量控制事务的大小，减少锁定的资源量和锁定时间长度
- (5) 在业务环境允许的情况下，尽量使用较低级别的事务隔离，以减少 MySQL 因为实现事务隔离级别所带来的附加成本。

1.6. 表结构优化

1.6.1. 永远为每张表设置一个 ID

我们应该为数据库里的每张表都设置一个 ID 做为其主键，而且最好的是一个 INT 型的(推荐使用 UNSIGNED)，并设置上自动增加的 AUTO_INCREMENT 标志。

就算是你 `users` 表有一个主键叫 `"email"` 的字段，你也别让它成为主键。使用 `VARCHAR` 类型来当主键会使用得性能下降。另外，在你的程序中，你应该使用表的 ID 来构造你的数据结构。

而且，在 MySQL 数据引擎下，还有一些操作需要使用主键，在这些情况下，主键的性能和设置变得非常重要，比如，集群，分区.....

在这里，只有一个情况是例外，那就是“关联表”的“外键”，也就是说，这个表的主键，通过若干个别的表的主键构成。我们把这个情况叫做“外键”。比如：有一个“学生表”有学生的 ID，有一个“课程表”有课程 ID，那么，“成绩表”就是“关联表”了，其关联了学生表和课程表，在成绩表中，学生 ID 和课程 ID 叫“外键”其共同组成主键。

1.6.2. 有限值字段使用 ENUM 而不是 VARCHAR

ENUM 类型是非常快和紧凑的。在实际上，其保存的是 `TINYINT`，但其外表上显示为字符串。这样一来，用这个字段来做一些选项列表变得相当的完美。

如果你有一个字段，比如“性别”，“国家”，“民族”，“状态”或“部门”，你知道这些字段的取值是有限而且固定的，那么，你应该使用 `ENUM` 而不是 `VARCHAR`。

MySQL 也有一个“建议”(见第十条)告诉你怎么去重新组织你的表结构。当你有一个 `VARCHAR` 字段时，这个建议会告诉你把其改成 `ENUM` 类型。使用 `PROCEDURE ANALYSE()` 你可以得到相关的建议。

1.6.3. 固定长度的表会更快

如果表中的所有字段都是“固定长度”的，整个表会被认为是 `"static"` 或 `"fixed-length"`。例如，表中没有如下类型的字段：`VARCHAR`，`TEXT`，`BLOB`。只要你包括了其中一个这些字段，那么这个表就不是“固定长度静态表”了，这样，MySQL 引擎会用另一种方法来处理。

固定长度的表会提高性能，因为 MySQL 搜寻得会更快一些，因为这些固定的长度是很容易计算下一个数据的偏移量的，所以读取的自然也会很快。而如果字段不是定长的，那么，每一次要找下一条的话，需要程序找到主键。

并且，固定长度的表也更容易被缓存和重建。不过，唯一的副作用是，固定长度的字段会浪费一些空间，因为定长的字段无论你用不用，他都是要分配那么多的空间。

使用“垂直分割”技术(见下一条)，你可以分割你的表成为两个一个是定长的，一个则是不定长的。

1.6.4. 尽可能的使用 NOT NULL

除非你有一个很特别的原因去使用 NULL 值，你应该总是让你的字段保持 NOT NULL。这看起来好像有点争议，请往下看。

首先，问问你自己“Empty”和“NULL”有多大的区别(如果是 INT，那就是 0 和 NULL)? 如果你觉得它们之间没有什么区别，那么你就不要使用 NULL。不要以为 NULL 不需要空间，其需要额外的空间，并且，在你进行比较的时候，你的程序会更复杂。当然，这里并不是说你就不能使用 NULL 了，现实情况是很复杂的，依然会有些情况下，你需要使用 NULL 值。

1.7. 数据库参数优化

1.7.1. 最佳实践

1.7.1.1. 使用 InnoDB 存储引擎

InnoDB 引擎已经在多方面超越了 MyISAM 引擎，没有特殊需求的情况下建议选择 InnoDB 引擎。

1.7.1.2. 让 InnoDB 使用全部内存

innodb_buffer_pool_size 参数指定了 InnoDB 可以使用的内存总量。

建议设置为物理内存的 80%，因为要给操作系统留有空间。

如果你的内存是 32GB，可以设置为大约 25GB

`innodb_buffer_pool_size = 25600M`

注意：

(1) 如果值小于 1GB，说明真的应该升级服务器了

(2) 如果内存特别大，例如 200gb，就不必给操作系统留 20% 了，因为 OS 用不了 40gb。

1.7.1.3. 让 InnoDB 多实例

`innodb_buffer_pool_size` 的值大于 1G 时，`innodb_buffer_pool_instances` 会把 InnoDB 的缓存池划分成多个实例。

多个缓冲池的好处：

多个线程同时访问缓冲池时可能会遇到瓶颈，而多个缓冲池则可以最小化这个冲突

官方建议的 buffer 数量：每个 buffer pool 实例至少要 1G

例如内存为 32GB，`innodb_buffer_pool_size` 为 25GB，那么合适的方案就是

$25600M / 24 = 1.06GB$

`innodb_buffer_pool_instances = 24`

1.7.1.4. 加大 `max_length_for_sort_data` 参数的设置

在 MySQL 中，排序算法分为两种，一是只加载排序字段到内存，排序完成后再到表中取其他字段，二是加载所有需要的字段到内存，显然第二种节省了 IO 操作，所以更快。

决定使用哪种算法是通过参数 `max_length_for_sort_data` 来决定的，当所有返回字段的最大长度小于这个参数值时，MySQL 就会选择第二种算法，反之使用第一种。所以，如果有充足的内存让 MySQL 存放须要返回的非排序字段，就可以加大这个参数的值来让 MySQL 选择第二种排序算法。

当内存不是很充裕时，不能简单地通过强行加大上面的参数来强迫 MySQL 去使用高效算法，否则可能会造成 MySQL 不得不将数据分成很多段，然后进行排序，这样可能会得不偿失，此时就须要去掉不必要的返回字段，让返回结果长度适应 `max_length_for_sort_data` 参数的限制。

1.7.1.5. 增大 `sort_buffer_size` 参数设置

增大 `sort_buffer_size` 并不是为了让 MySQL 选择第二种排序算法，而是为了让 MySQL 尽量减少在排序过程中对须要排序的数据进行分段，因为分段会造成 MySQL 不得不使用临时表来进行交换排序。

1.7.1.6. 打开查询缓存

充分利用 Mysql 的查询缓存机制，业务中很多 SQL 都会重复执行的，当然现在很多数据层框架中也有缓存功能，但数据层框架中的缓存属于应用级别的，在数据被外部更新时会导致缓存数据过期问题。

1.7.2. 案例

发现网站页面打开非常慢，对处理过程简单记录了一下。

1.7.2.1. 找问题

首先登录服务器使用 `top` 查看当前进程信息，发现排名第一的是 `mysql`，占用 `cpu` 达到了 100% 以上，这就明确了是 `mysql` 的问题。

登录 `mysql`，使用 `show processlist` 查看下当前执行状态，发现了大量 `LOCK` 操作，也有多个 `Copying to tmp table` 的操作，说明有 `sql` 出现了问题，操作过于复杂，对临时表使用频繁，把其他操作阻塞了。

1.7.2.2. 解决思路

找到了问题后，把处理方向确定为检查和修改配置、`sql` 优化。

1.7.2.3. 修改 mysql 配置

临时表

既然涉及到了临时表，就先查看下目前临时表的信息

查看临时表的使用状态

```
show global status like 'created_tmp%';
```

发现 created_tmp_disk_tables 值过高，需要增加此值。

再看一下现在临时表的大小

```
show variables like '%tmp_table_size';
```

在现在值的基础上增加一些，重新设置临时表大小

线程缓存数

看当前线程情况

```
show global status like 'Thread%';
```

发现 threads_created 的值过大，表明 MySQL 服务器一直在创建线程

查看当前值

```
show variables like 'thread_cache_size';
```

此参数需要调高

打开表数量

查看打开表的情况

```
show global status like 'open%tables%';
```

发现 opened_tables 数量过大，说明 table_cache 的值可能太小。

查看当前值

```
show variables like 'table_cache';
```

此参数需要调高

最大连接数

查看当前允许的最大连接数

show variables like 'max_connections';

查看服务器连接数的峰值

show global status like 'Max_used_connections';

峰值还没到最大限制，不需要修改

join buffer 和 sort buffer

查看现有值

SELECT @@sort_buffer_size;

SELECT @@join_buffer_size;

是默认值，需要修改

修改配置

确定了要修改的参数后，修改 my.cnf，例如

table_cache = 64

sort_buffer_size = 8M

join_buffer_size = 4M

thread_cache_size = 300

thread_concurrency = 8

tmp_table_size = 246M

1.7.2.4. sql 优化

从 show processlist 结果集中找出主要的复杂语句，对其进行 explain 和 profile

分析，进行索引优化，把复杂的 sql 根据业务拆分为多个小的 sql。

1.8. 主从复制

MySQL 主从复制是指数据可以从一个 MySQL 数据库服务器主节点复制到一个或多个从节点。MySQL 默认采用异步复制方式，这样从节点不用一直访问主服务器来更新自己的数据，数据的更新可以在远程连接上进行，从节点可以复制主数据库中的所有数据库或者特定的数据库，或者特定的表。

1.8.1. 目的

1.8.1.1. 数据同步备份

主库 master 发生故障后，可以马上切换到从库 slave，降低服务风险。

1.8.1.2. 读写分离

可以把写操作放在 master，读取操作放在 slave，减轻单一数据库的操作压力

1.8.1.3. 高可用 HA

随着系统中业务访问量的增大，如果是单机部署数据库，就会导致 I/O 访问频率过高。

有了主从复制，增加多个数据存储节点，将负载分布在多个从节点上，降低单机磁盘 I/O 访问的频率，提高单个机器的 I/O 性能。

1.8.2. 基本原理

master 记录下自己的操作日志，授权从服务器可以读取操作日志，slave 会开启两个线程。

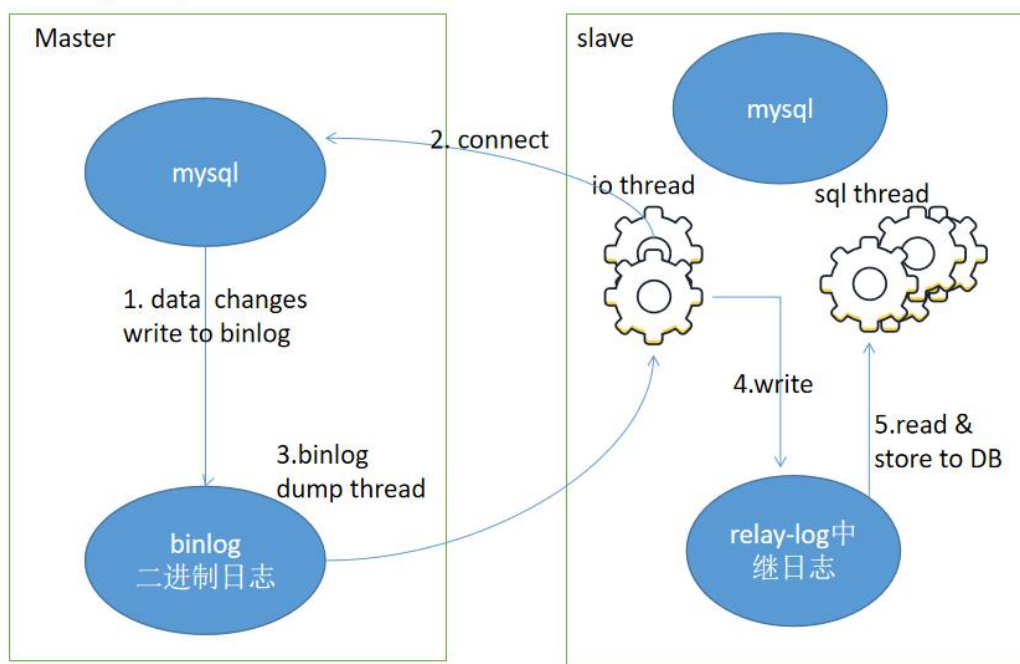
IO 线程

负责连接 master 连接成功后，睡眠并等待 master 产生新的事件，有新的就保存到自己的中继日志中，中继日志通常位于操作系统的缓存中，所以开销很小。

sql 进程

负责执行中继日志中的 sql 操作，这样 slave 的内容就和 master 的一致了。

1.8.3. 执行步骤



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

1.8.4. 配置方式

1.8.4.1. 前提条件

停止对 master 数据库的操作，把 master 中的数据库全部导入到 slave，使两边数据库完全一致。

1.8.4.2. 配置 master

1. 修改 master 的配置文件，使用二进制日志，指定 server-id，重启服务。目的是让各自都有了自己的唯一标示，并以二进制文件格式进行交流。Centos 中路径为 /etc/my.cnf。

[mysqld]

Log_bin=mysql-bin //[必须]启用二进制日志

server-id=10//[必须]服务器唯一 ID，默认是 1，一般取 IP 最后一段

配置完成后需要重启 mysqlserver 才能生效。

systemctl restart mysqld

2. 创建授权用户

登陆主服务器 mysql 命令行，创建一个用于从服务器复制的用户。

mysql -uroot -p

mysql>grant replication slave on *.* to 'root'@'%' identified by '123456';

"*.*"表示对所有库的所有操作，"%"表示所有客户端都可能连，也可用具体客户端 IP

代替，如 192.168.33.11，加强安全。

3. 记录 master 状态信息

查看二进制日志文件名，及最新位置。让 slave 知道用哪个用户信息访问 master，知道读取哪个日志文件，及从哪儿开始读。

mysql>show master status;

```
mysql> show master status;
+-----+-----+-----+-----+-----+
| File           | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+-----+-----+-----+-----+-----+
| mysql-bin.000001 |      437 |              |                  |                  |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

其中 file、position 字段需要记录下值，mysql-bin.000001 是用于主从复制的文件名，

437 是日志文件内的最新位置。

1.8.4.3. 配置 slave

1. 修改配置文件 my.cnf，使用二进制日志，指定 server-id，重新启动服务。

[mysqld]

Log_bin=mysql-bin

Server_id=11

2. 将 slave 指向 master

登陆从服务器 mysql 命令行，使用之前创建的用户和 master 的日志文件及其位置。

slave 中使用被授权用户信息及日志文件信息，进行指向 master。这时已经建立了和 master 的联系，明确了从哪儿读取日志文件。

```
mysql> change master to  
master_host='192.168.33.10',master_user='root',master_password='123456',master_log_file='mysql-bin.00  
0001',master_log_pos=437;  
//注意不要断开，“437”无单引号。
```

3. 启动 slave

执行启动 slave 的命令，开始主从复制

```
mysql> start slave;
```

4. 查看 slave 状态

```
mysql> show slave status\G;
```

结果中有两个重要数据项：

1) Slave_IO_Running: Yes IO 线程状态，必须 YES

2) Slave_SQL_Running: Yes SQL 线程状态，必须 YES

常见的问题是 SQL 线程没有正常工作 Slave_SQL_Running: No

通常是两边的数据库不是完全对应的，需要确保 master 上的库及到目前为止的最新记录都复制到 slave 上了。

1.8.4.4. 验证测试

当 IO 线程和 SQL 线程都正常后，到 master 中随意测试下插入、修改、删除操作，同时到 slave 中检查。

master 执行以下命令：

```
Create database mastertest;  
Use mastertest;  
CREATE TABLE `test` (  
  `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,  
  `name` varchar(100) NOT NULL,  
  PRIMARY KEY (`id`)
```

```
) ENGINE=InnoDB DEFAULT CHARSET=utf8;  
Insert into test(name) values('boxuegu1test');  
Insert into test(name) values('boxuegu2test');  
Insert into test(name) values('boxuegu3test');
```

slave 下验证

```
show databases;  
Use mastertest;  
Show tables;  
Select * from test;
```

1.8.5. 使用 Mysql Utilities 快速搭建主从复制

Mysql Utilities 是一个 Mysql 的工具箱（基于 python），里面有不少好用的工具，其中的 mysqlreplicate 命令，可以让我们通过一个命令就能快速配置好主从复制环境。

1.8.5.1. Mysql Utilities 下载地址

<http://dev.mysql.com/downloads/utilities/1.5.html>

1.8.5.2. Mysql Utilities 文档

<http://dev.mysql.com/doc/mysql-utilities/1.6/en/utis-overview.html>

1.8.5.3. 用法：

```
mysqlreplicate \  
--master=root:111111@192.168.31.168:3306 \  
--slave=root:111111@192.168.31.101:3306 \  
--rpl-user=replutil:111111
```

--master 指定主库的连接信息

--slave 指定从库的连接信息

--rpl-user 指定用于复制的用户信息，这个用户需要提前在 master 上创建好，例如：

```
grant ALL PRIVILEGES on *.* to replutil@"192.168.31.101" Identified by "111111";
```

注意，创建用户时，其中的从库 IP 要明确，不要用 '%'。

可以看到，总共只需要两步：

(1) master 上创建用于复制的用户

(2) 执行 mysqlreplicate 命令

1.8.5.2.几秒钟就完成了主从配置。

1.8.5.4. 安装方法

下载解压 Mysql Utilities，进入解压后的目录，执行编译安装命令

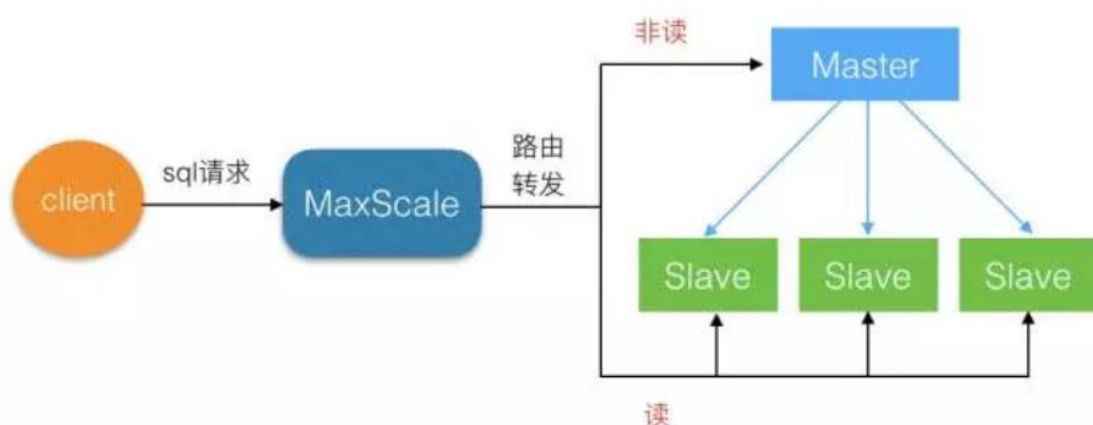
```
python ./setup.py build
```

```
python ./setup.py install
```

执行完成后，就可以使用其中的工具命令了

1.9. 读写分离中间件 MaxScale

配置好了 Mysql 的主从复制结构后，我们希望实现读写分离，把读操作分散到从服务器中，并且对多个从服务器能实现负载均衡。读写分离和负载均衡是 Mysql 集群的基础需求，MaxScale 就可以帮着我们方便的实现这些功能。



1.9.1. MaxScale 的基础构成

MaxScale 是 Mysql 的兄弟公司 MariaDB 开发的，现在已经发展得非常成熟。

MaxScale 是插件式结构，允许用户开发适合自己的插件。

MaxScale 目前提供的插件功能分为 5 类

1.9.1.1. 认证插件

提供了登录认证功能，MaxScale 会读取并缓存数据库中 user 表中的信息，当有连接进来时，先从缓存信息中进行验证，如果没有此用户，会从后端数据库中更新信息，再次进行验证

1.9.1.2. 协议插件

包括客户端连接协议，和连接数据库的协议。

1.9.1.3. 路由插件

决定如何把客户端的请求转发给后端数据库服务器，读写分离和负载均衡的功能就是由这个模块实现的。

1.9.1.4. 监控插件

对各个数据库服务器进行监控，例如发现某个数据库服务器响应很慢，那么就不向其转发请求了。

1.9.1.5. 日志和过滤插件

提供简单的数据库防火墙功能，可以对 SQL 进行过滤和容错。

1.9.2. 配置方式

1.9.2.1. 配置一主二从集群环境

准备 3 台服务器，安装 Mysql，配置一主二从的复制结构。主从复制的配置过程参加上一节内容。

1.9.2.2. 安装 MaxScale

最好在另一台服务器上安装，如果资源不足，可以和某个 Mysql 放在一起。

MaxScale 的下载地址，最新版本 2.3.4，实例中使用的是 1.4.5：

<https://downloads.mariadb.com/files/MaxScale>

根据自己的服务器选择合适的安装包，以 centos 7 为例安装步骤如下：

```
yum install libaio.x86_64 libaio-devel.x86_64 novacom-server.x86_64 libedit -y
```

```
wget  
https://downloads.mariadb.com/MaxScale/1.4.5/centos/7/x86_64/maxscale-1.4.5-1.centos.7.x8  
6_64.rpm  
rpm -ivh maxscale-1.4.5-1.centos.7.x86_64.rpm
```

1.9.2.3. 配置 MaxScale

在开始配置之前，需要在 master 中为 MaxScale 创建两个用户，用于监控模块和路由模块：

创建监控用户

```
create user 'maxmon'@'%' identified by '123456';  
grant replication slave,replication client on *.* to 'maxmon'@'%';
```

创建路由用户

```
create user 'maxrou'@'%' identified by '123456';  
grant select on mysql.* to 'maxrou'@'%';  
flush privileges;
```

用户创建完成后，开始配置

```
vi /etc/maxscale.cnf
```

找到 [server1] 部分，修改其中的 address 和 port，指向 master 的 IP 和端口

复制 2 次 [server1] 的整块儿内容，改为 [server2] 与 [server3]，同样修改其中的

address 和 port，分别指向 slave1 和 slave2

```
[server1]  
type=server  
address=192.168.33.10  
port=3306  
protocol=MySQLBackend  
  
[server2]  
type=server  
address=192.168.33.11  
port=3306  
protocol=MySQLBackend  
  
[server3]  
type=server  
address=192.168.33.12  
port=3306  
protocol=MySQLBackend
```

找到 [MySQL Monitor] 部分，修改 servers 为 server1,server2,server3，修改 user 和 passwd 为之前创建的监控用户的信息 (maxmon,123456)

```
[MySQL Monitor]
type=monitor
module=mysqlmon
servers=server1,server2,server3
user=maxmon
passwd=123456
#259D29C04694136AB1DC3E0149573202
monitor_interval=10000
detect_stale_master=true
```

找到 [Read-Write Service] 部分，修改 servers 为 server1,server2,server3，修改 user 和 passwd 为之前创建的路由用户的信息 (maxrou,123456)

```
#[Read-Only Service]
#type=service
#router=readconroute
#servers=server1
#user=myuser
#passwd=mypwd
#router_options=slave

# ReadWriteSplit documentation:
# https://github.com/mariadb-corporation/MaxScale/blob/master/Documentation/Routers/ReadWriteSplit.md

[Read-Write Service]
type=service
router=readwritesplit
servers=server1,server2,server3
user=maxrou
passwd=123456
#259D29C04694136AB1DC3E0149573202
max_slave_connections=100%
```

由于我们使用了 [Read-Write Service]，需要删除另一个服务 [Read-Only Service]，删除其整块儿内容即可，底部还有一个[Read-Only Client]也需要删除。

配置完成，保存并退出编辑器。

1.9.2.4. 启动 MaxScale

执行启动命令

```
maxscale --config=/etc/maxscale.cnf
```

查看 MaxScale 的响应端口是否已经就绪

```
netstat -ntlp
```

```
[root@da843624f82c ~]# netstat -ntlp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       User        Inode        PID/Program name
tcp        0      0 0.0.0.0:4006             0.0.0.0:*               LISTEN      0           36518        114/maxscale
tcp        0      0 0.0.0.0:6603             0.0.0.0:*               LISTEN      0           36521        114/maxscale
```


4006 是 Read-Write Listener 使用的端口，用于连接 MaxScale

6603 是 MaxAdmin Listener 使用的端口，用于 MaxScale 管理器

登录 MaxScale 管理器，查看一下数据库连接状态，默认的用户名和密码是

admin/mariadb

maxadmin --user=admin --password=mariadb

MaxScale> list servers

```
[root@localhost etc]# maxadmin --user=admin --password=mariadb
MaxScale> list servers
Servers.
```

Server	Address	Port	Connections	Status
server1	192.168.33.10	3306	0	Master, Running
server2	127.0.0.1	3306	0	Slave, Running
server3	192.168.33.12	3306	0	Slave, Running

可以看到，MaxScale 已经连接到了 master 和 slave

1.9.2.5. 测试

先在 master 上创建一个测试用户

```
create user 'rtest'@'%' identified by '111111';
grant ALL PRIVILEGES on *.* to 'rtest'@'%';
```

使用 Mysql 客户端到连接 MaxScale

```
mysql -urtest -p'111111' -h'192.168.33.11' -P4006
```

执行查看数据库服务器名的操作来知道当前实际所在的数据库

```
Select @@hostname;
start transaction;
Select @@hostname;
roolback;
Select @@hostname;
```

```
mysql> select @@hostname;
+-----+
| @@hostname |
+-----+
| Slave2      |
+-----+
1 row in set (0.00 sec)

mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select @@hostname;
+-----+
| @@hostname |
+-----+
| Master      |
+-----+
1 row in set (0.00 sec)

mysql> rollback;
Query OK, 0 rows affected (0.00 sec)

mysql> select @@hostname;
+-----+
| @@hostname |
+-----+
| Slave2      |
+-----+
1 row in set (0.00 sec)
```

开启事务后，就自动路由到了 master，普通的查询操作，是在 slave 上。

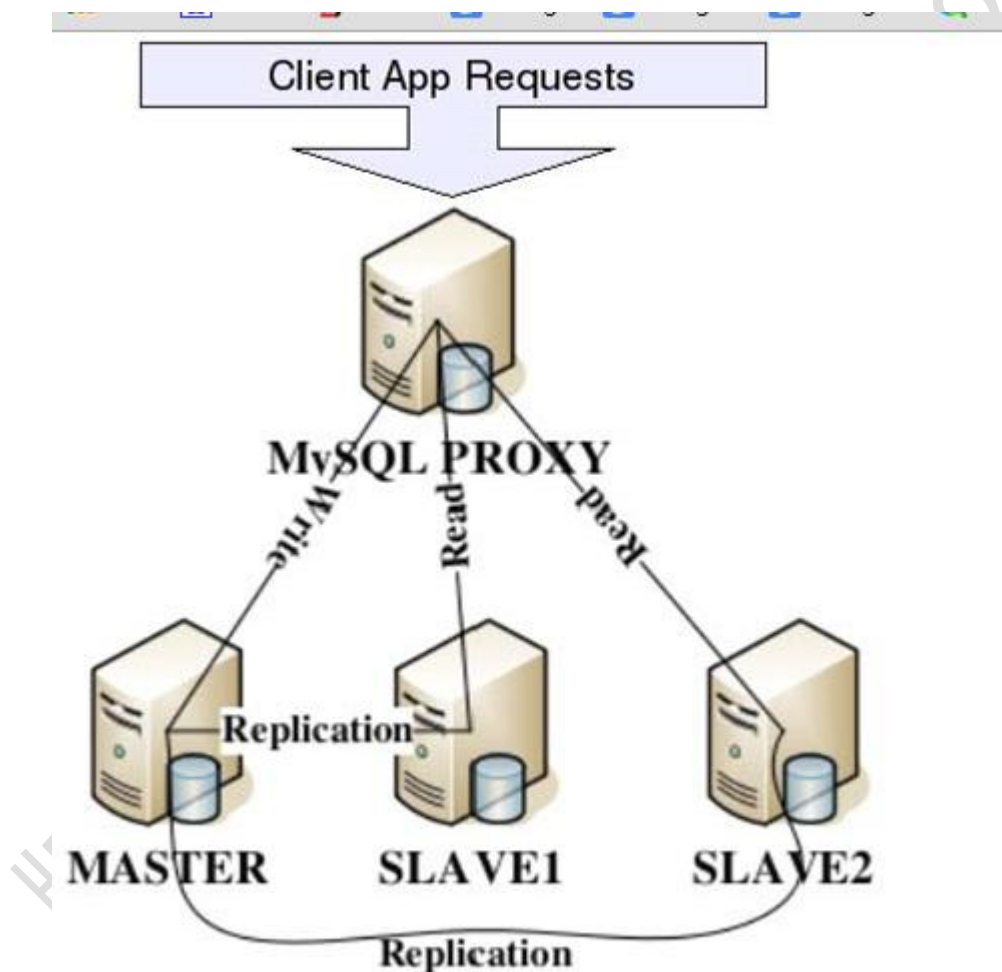
1. 10. 读写分离中间件 MySQL-Proxy

可参考：<https://www.cnblogs.com/luckcs/articles/2543607.html>

MySQLProxy 是 MySQL 官方提供的一个数据库代理层产品，和 MySQLServer 一样，相同是一个基于 GPL 开源协议的开源产品。可用来监视、分析或者传输他们之间的通讯信息，具备的功能主要有连接路由、Query 分析、Query 过滤和修改、负载均衡以及主要的 HA 机制等。

实际上 MySQLProxy 本身并不具有上述全部的这些功能，而是提供了实现上述功能的基础。要实现这些功能，还须要通过我们自行编写 LUA 脚本来实现。

MySQLProxy 实际上是在 client 请求与 MySQLServer 之间建立了一个连接池。全部 client 请求都是发向 MySQLProxy，然后经由 MySQLProxy 进行对应的分析。推断出是读操作还是写操作，分发至对应的 MySQLServer 上。对于多节点 Slave 集群，也能够起到负载均衡的效果。以下是 MySQLProxy 的基本架构图：



通过上面的架构简图我们能够非常清晰的看出 MySQLProxy 在实际应用中所处的位置，以及能做的基本事情。

1.11. 数据库切分 (Sharding)

当数据量达到一定程度时，我们处于性能考虑就需要将我们存放在同一个数据库中的数据分散存放到多个数据库（主机）上面，以达到分散单台设备负载的效果。数据的切分同一时候还能够提高系统的总体可用性，由于单台设备 Crash 之后。仅仅有总体数据的某部分不可用，而不是全部的数据。

数据的切分（Sharding）依据其切分规则的类型，能够分为垂直切分、水平切分、联合切分模式。

1.11.1. 数据的垂直切分

1.11.1.1. 简介

一种是依照不同的表（或者 Schema）来切分到不同的数据库（主机）之上，这样的切能够称之为数据的垂直（纵向）切分。

垂直切分的最大特点就是规则简单，实施也更为方便，尤其适合各业务之间的耦合度非常低。相互影响非常小，业务逻辑非常清晰的系统。在这样的系统中，能够非常容易做到将不同业务模块所使用的表拆分到不同的数据库中。依据不同的表来进行拆分。对应用程序的影响也更小，拆分规则也会比较简单清晰。

一个架构设计较好的应用系统。其总体功能肯定是由非常多个功能模块所组成的。而每一个功能模块所须要的数据对应到数据库中就是一个或者多个表。而在架构设计中，各个功能模块相互之间的交互点越统一越少，系统的耦合度就越低，系统各个模块的维护性以及扩展性也就越好。这样的系统实现数据的垂直切分也就越容易，不同功能模块的数据存放于不同的数据库主机中，能够非常容易就避免掉跨数据库的 Join 存在。

当然很多情况下系统的耦合度没有那么低，我们就必须依据实际的应用场景进行评估权衡。决定是迁就应用程序将须要 Join 的表的相关某快都存放在同一个数据库中，还是让

应用程序做很多其它的事情，也就是程序全然通过模块接口取得不同数据库中的数据，然后在程序中完毕 Join 操作。

1. 11. 1. 2. 案例分析

系统功能能够基本分为四个功能模块：用户，群组消息，相册以及事件。分别对应为例如以下这些表：

1. 用户模块表：user,user_profile,user_group,user_photo_album
2. 群组讨论表：groups,group_message,group_message_content,top_message
3. 相册相关表：photo,photo_album,photo_album_relation,photo_comment
4. 事件信息表：event

初略一看，没有哪一个模块能够脱离其它模块独立存在，模块与模块之间都存在着关系。莫非无法切分？

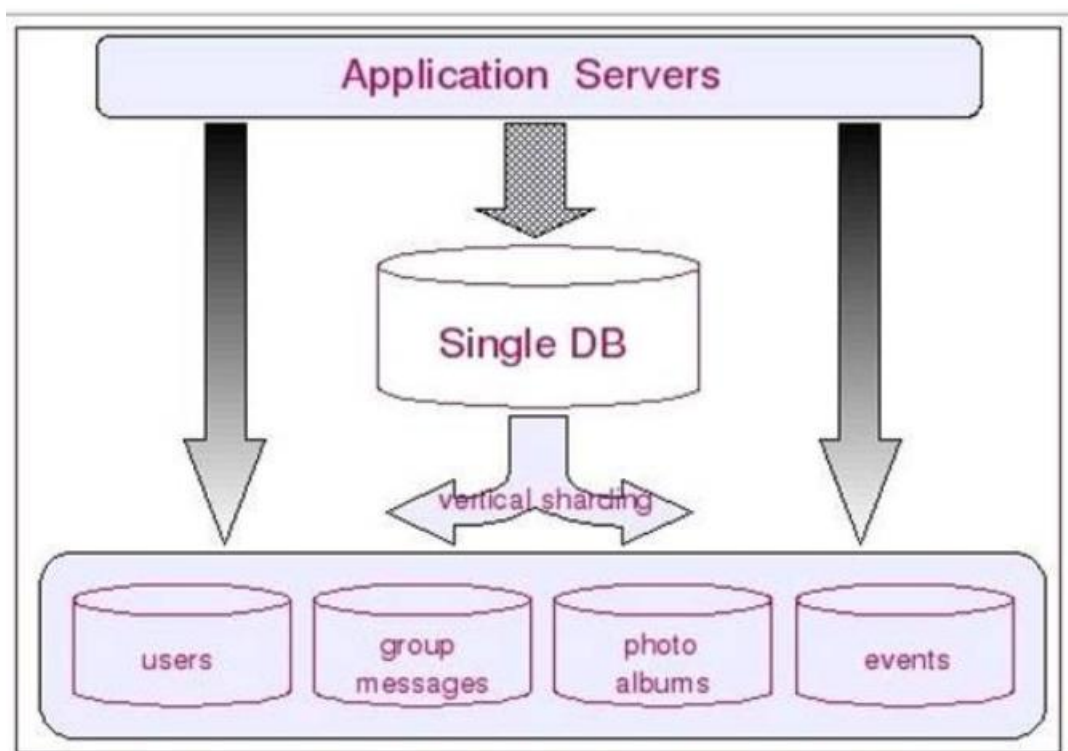
当然不是，我们再略微深入分析一下，能够发现，尽管各个模块所使用的表之间都有关联，可是关联关系还算比较清晰，也比较简单。

◆ 群组讨论模块和用户模块之间主要存在通过用户或者是群组关系来进行关联。一般关联的时候都会是通过用户的 id 或者 nick_name 以及 group 的 id 来进行关联。通过模块之间的接口实现不会带来太多麻烦。

◆ 相册模块仅仅与用户模块存在通过用户的关联。这两个模块之间的关联基本就有通过用户 id 关联的内容。简单清晰，接口明白；

◆ 事件模块与各个模块可能都有关联，可是都仅仅关注其各个模块中对象的 ID 信息，相同能够做到非常容易分拆。

所以。我们第一步能够将数据库依照功能模块相关的表进行一次垂直拆分。每一个模块所涉及的表单独到一个数据库中，模块与模块之间的表关联都在应用系统端通过接口来处理。例如以下图所看到的：



通过这样的垂直切分之后。之前仅仅能通过一个数据库来提供的服务。就被分拆成四个数据库来提供服务，服务能力自然是添加几倍了。

1.11.1.3. 垂直切分的长处

- ◆ 数据库的拆分简单明了，拆分规则明白；
- ◆ 应用程序模块清晰明白，整合容易。
- ◆ 数据维护方便易行，容易定位。

1.11.1.4. 垂直切分的缺点

- ◆ 部分表关联无法在数据库级别完毕。须要在程序中完毕。
- ◆ 对于访问极其频繁且数据量超大的表仍然存在性能平静，不一定能满足要求。
- ◆ 事务处理相对更为复杂；

- ◆ 切分达到一定程度之后，扩展性会遇到限制；
- ◆ 过读切分可能会带来系统过渡复杂而难以维护。

1.11.2. 数据的水平切分

1.11.2.1. 简介

依据表中的数据的逻辑关系，将同一个表中的数据依照某种条件拆分到多台数据库（主机）上面，这样的切分称之为数据的水平（横向）切分。一般来说，简单的水平切分主要是将某个访问极其平庸的表再依照某个字段的某种规则来分散到多个表之中。每一个表中包括一部分数据。

水平切分于垂直切分相比。相对来说略微复杂一些。由于要将同一个表中的不同数据拆分到不同的数据库中，对于应用程序来说，拆分规则本身就较依据表名来拆分更为复杂，后期的数据维护也会更为复杂一些。

如依据某个数字类型字段基于特定数目取模，某个时间类型字段的范围，或者是某个字符类型字段的 hash 值。假设整个系统中大部分核心表都能够通过某个字段来进行关联。那这个字段自然是一个进行水平分区的首选了。

一般来说，像如今互联网非常火爆的 Web2.0 类型的站点。基本上大部分数据都能够通过会员用户信息关联上，可能非常多核心表都非常适合通过会员 ID 来进行数据的水平切分。切分之后基本上不会出现各个库之间的交互。

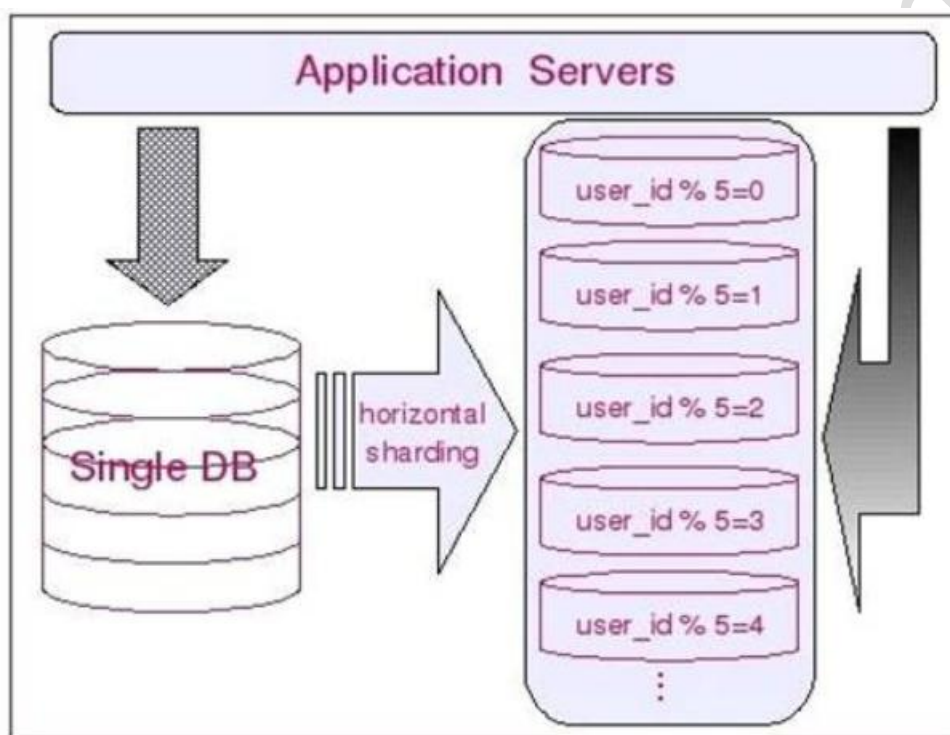
1.11.2.2. 案例分析

全部数据都是和用户关联的。那么我们就能够依据用户来进行水平拆分，将不同用户的数据切分到不同的数据库中。当然，唯一有点差别的是用户模块中的 groups 表和用户没有直接关系。所以 groups 不能依据用户来进行水平拆分。对于这样的特殊情况下的表，我们全然能够独立出来。单独放在一个独立的数据库中。

所以，对于我们的演示样例数据库来说，大部分的表都能够依据用户 ID 来进行水平的切分。不同用户相关的数据进行切分之后存放在不同的数据库中。如将全部用户 ID 通过 5 取模然后分别存放于五个不同的数据库中。

每一个和用户 ID 关联上的表都能够这样切分。这样，基本上每一个用户相关的数据。都在同一个数据库中，即使是须要关联，也能够非常简单的关联上。

我们能够通过下图来更为直观的展示水平切分相关信息：



1. 11. 2. 3. 水平切分的长处

- ◆ 表关联基本能够在数据库端全部完毕；
- ◆ 不会存在某些超大型数据量和高负载的表遇到瓶颈的问题；
- ◆ 应用程序端总体架构修改相对较少；
- ◆ 事务处理相对简单；
- ◆ 仅仅要切分规则能够定义好。基本上较难遇到扩展性限制；

1. 11. 2. 4. 水平切分的缺点

- ◆ 切分规则相对更为复杂，非常难抽象出一个能够满足整个数据库的切分规则；
- ◆ 后期数据的维护难度有所添加，人为手工定位数据更困难；
- ◆ 应用系统各模块耦合度较高，可能会对后面数据的迁移拆分造成一定的困难。

1.11.3. 联合切分

1.11.3.1. 简介

当我们某个（或者某些）表的数据量和访问量特别的大，通过垂直切分将其放在独立的设备上后仍然无法满足性能要求，这时候我们就必须将垂直切分和水平切分相结合。先垂直切分，然后再水平切分，才能解决这样的超大型表的性能问题。

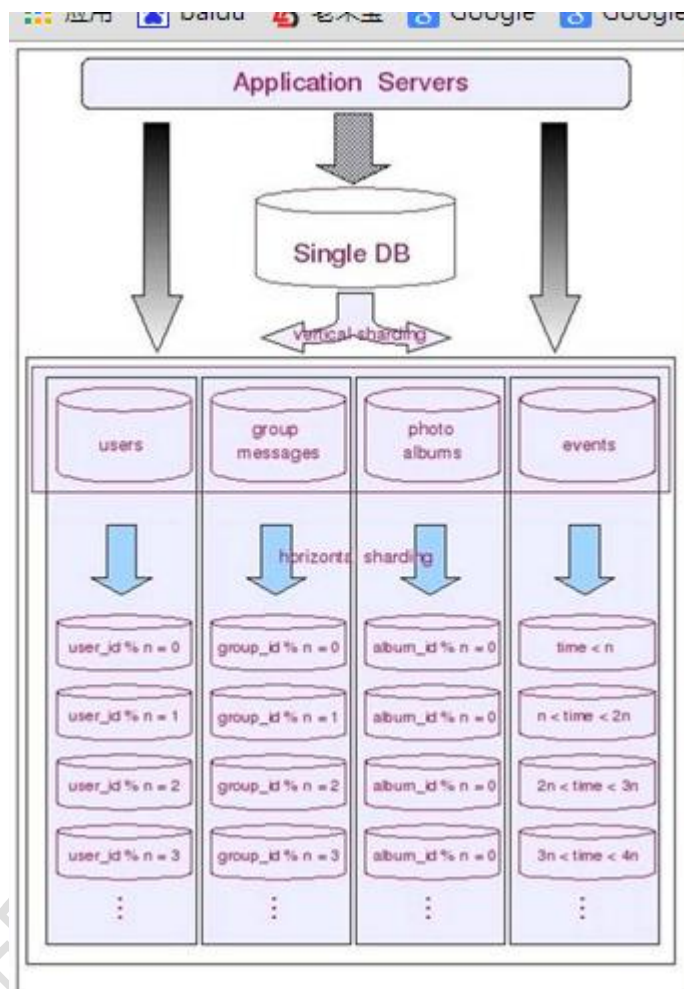
每一个应用系统的负载都是一步一步增长上来的，在开始遇到性能瓶颈的时候，大多数架构师和 DBA 都会选择先进行数据的垂直拆分，由于这样的成本最小。最符合这个时期所追求的最大投入产出比。然而随着业务的不断扩张，系统负载的持续增长，在系统稳定一段时期之后，经过了垂直拆分之后的数据库集群可能又再一次不堪重负，遇到了性能瓶颈。这时候我们就必须要通过数据的水平切分的优势，来解决这里所遇到的问题。并且，我们全然不必在使用数据水平切分的时候，推倒之前进行数据垂直切分的成果，而是在其基础上利用水平切分的优势来避开垂直切分的弊端。而水平拆分的弊端（规则难以统一）也已经被之前的垂直切分解决掉了，让水平拆分能够进行的得心应手。

1.11.3.2. 案例分析

假设在最开始。我们进行了数据的垂直切分，然而随着业务的不断增长，数据库系统遇到了瓶颈，我们选择重构数据库集群的架构。怎样重构？我们选择了在垂直切分的基础上再进行水平拆分。在经历过垂直拆分后的各个数据库集群中的每一个都仅仅有一个功能模块。而每一个功能模块中的全部表基本上都会与某个字段进行关联。如用户模块全部都能够通过用户 ID 进行切分，群组讨论模块则都通过群组 ID 来切分。相册模块则依据相册

ID 来进行切分。最后的事件通知信息表考虑到数据的时效性（仅仅会访问近期某个事件段的信息），则考虑按时间来切分。

下图展示了切分后的整个架构：



实际上，在非常多大型的应用系统中，垂直切分和水平切这两种数据的切分方法基本上都是并存的。并且经常在不断的交替进行，以不断的添加系统的扩展能力。我们在应对不同的应用场景的时候，也须要充分考虑到这两种切分方法各自的局限，以及各自的优势。在不同的时期（负载压力）使用不同的结合方式。

1.11.3.3. 联合切分的长处

- ◆ 能够充分利用垂直切分和水平切分各自的优势而避免各自的缺陷；

- ◆ 让系统扩展性得到最大化提升。

1.11.3.4. 联合切分的缺点

- ◆ 数据库系统架构比较复杂。维护难度更大。
- ◆ 应用程序架构也相对更复杂；

1.11.4. 数据切分及整合方案

我们已经非常清晰了通过数据库的数据切分能够极大的提高系统的扩展性，可是数据库中的数据在经过垂直和（或）水平切分被存放在不同的数据库主机之后，应用系统面临的最大的问题就是怎样来让这些数据源得到较好的整合。

数据的整合非常难依靠数据库本身来达到这个效果，尽管 MySQL 存在 Federated 存储引擎，能够解决部分相似的问题。可是在实际应用场景中却非常难较好的运用。那我们该怎样来整合这些分散在各个 MySQL 主机上面的数据源呢？

总的来说，存在两种解决思路：

1. 在每一个应用程序模块中配置管理自己须要的一个（或者多个）数据源。直接访问各个数据库，在模块内完毕数据的整合；

2. 通过中间代理层来统一管理全部的数据源。后端数据库集群对前端应用程序透明；

可能 90%以上的人在面对上面这两种解决思路的时候都会倾向于选择第二种，尤其是系统不断变得庞大复杂的时候，尽管短期内须要付出的成本可能会相对更大一些，可是对整个系统的扩展性来说，是非常有帮助的。所以，对于第一种解决思路我这里就不准备过多的分析，以下我重点分析一下在另外一种解决思路中的一些解决方式。

1.11.4.1. 自行开发中间代理层

通过自行开发中间代理层能够最大程度的应对自身应用的特点，最大化的定制非常多个性化需求，在面对变化的时候也能够灵活的应对。当然，选择自行开发享受让个性化定

制最大化的乐趣的同一时候，自然也须要投入很多其它的成本来进行前期研发以及后期的持续升级改进工作，并且本身的技术门槛可能也比简单的 Web 应用要更高一些。

1. 11. 4. 2. 利用 MySQLProxy 实现数据切分及整合

基于 MySQLProxy 自行编写 LUA 脚本实现数据切分相关代理功能。

1. 11. 4. 3. 利用 Amoeba 实现数据切分及整合

Amoeba 是一个基于 Java 开发的，专注于解决分布式数据库数据源整合 Proxy 程序的开源框架，基于 GPL3 开源协议。眼下 Amoeba 已经具有 Query 路由、Query 过滤、读写分离、负载均衡以及 HA 机制等相关内容。

Amoeba 主要解决的以下几个问题：

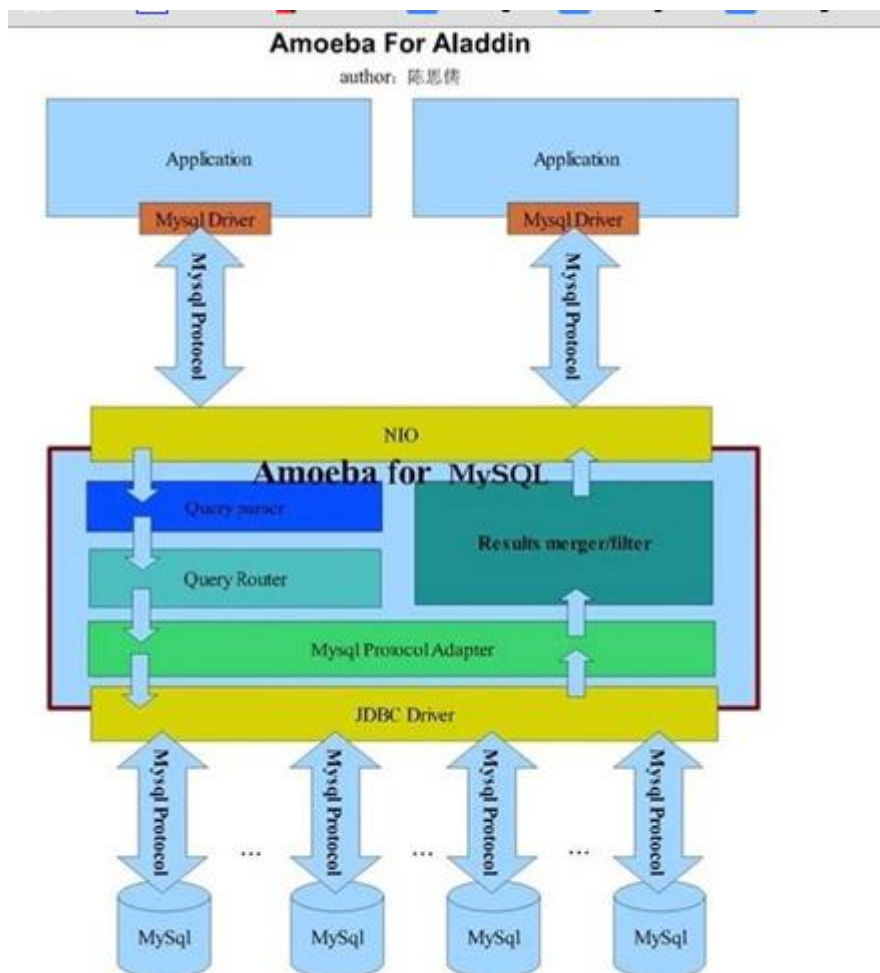
1. 数据切分后复杂数据源整合；
2. 提供数据切分规则并降低数据切分规则给数据库带来的影响。
3. 降低数据库与 client 的连接数。
4. 读写分离路由；

我们能够看出，Amoeba 所做的事情，正好就是我们通过数据切分来提升数据库的扩展性所须要的。

Amoeba 并非一个代理层的 Proxy 程序，而是一个开发数据库代理层 Proxy 程序的开发框架，眼下基于 Amoeba 所开发的 Proxy 程序有 AmoebaForMySQL 和 AmoebaForAladin 两个。

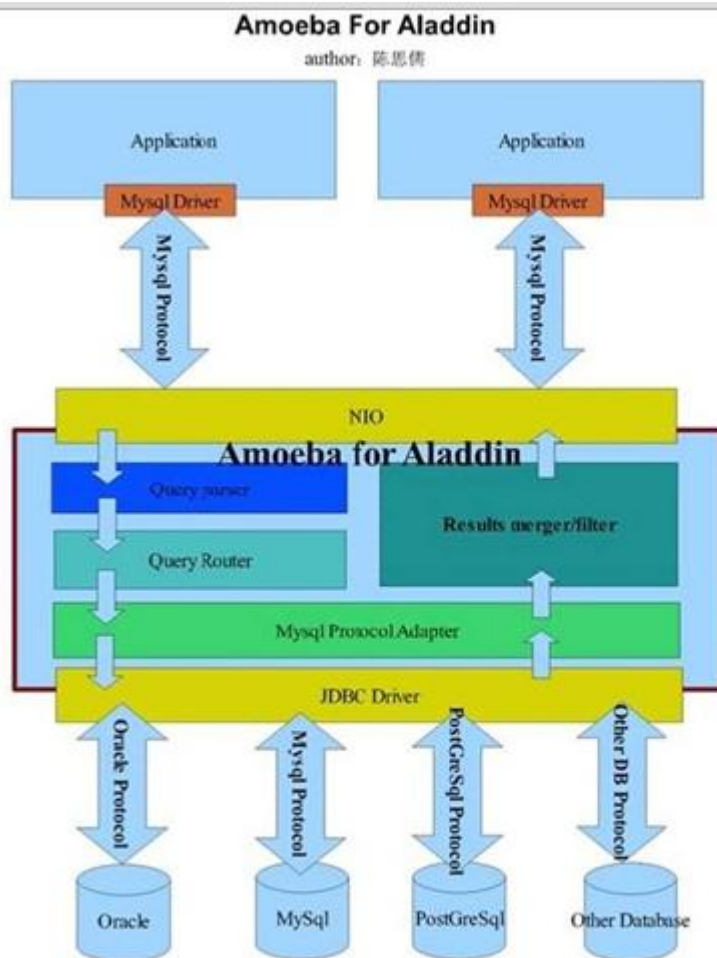
AmoebaForMySQL 主要是专门针对 MySQL 数据库的解决方式，前端应用程序请求的协议以及后端连接的数据源数据库都必须是 MySQL。对于 client 的不论什么应用程序来说，AmoebaForMySQL 和一个 MySQL 数据库没有什么差别。不论什么使用 MySQL 协

议的 client 请求，都能够被 AmoebaForMySQL 解析并进行对应的处理。下如能够告诉我们要们 AmoebaForMySQL 的架构信息（出自 Amoeba 开发人员博客）：



AmoebaForAladin 则是一个适用更为广泛，功能更为强大的 Proxy 程序。

它能够同一时候连接不同数据库的数据源为前端应用程序提供服务，可是仅仅接受符合 MySQL 协议的 client 应用程序请求。也就是说，仅仅要前端应用程序通过 MySQL 协议连接上来之后，AmoebaForAladin 会自己主动分析 Query 语句，依据 Query 语句中所请求的数据来自己主动识别出该所 Query 的数据源是在什么类型数据库的哪一个物理主机上面。下图展示了 AmoebaForAladin 的架构细节（出自 Amoeba 开发人员博客）：



乍一看，两者好像全然一样。细看之后才会发现两者主要的差别仅在于通过 MySQLProtocolAdapter 处理之后。依据分析结果推断出数据源数据库。然后选择特定的 JDBC 驱动和对应协议连接后端数据库。

事实上通过上面两个架构图大家可能也已经发现了 Amoeba 的特点了，他仅仅仅仅是一个开发框架。我们除了选择他已经提供的 ForMySQL 和 ForAladin 这两款产品之外。还能够基于自身的需求进行对应的二次开发。得到更适应我们自己应用特点的 Proxy 程序。

当对于使用 MySQL 数据库来说。不论是 AmoebaForMySQL 还是 AmoebaForAladin 都能够非常好的使用。当然，考虑到不论什么一个系统越是复杂，其性能肯定就会有一定

的损失，维护成本自然也会相对更高一些。所以，对于仅仅须要使用 MySQL 数据库的时候，我还是建议使用 AmoebaForMySQL。

AmoebaForMySQL 的使用非常简单，全部的配置文件都是标准的 XML 文件，总共同拥有四个配置文件。分别为：

- ◆ amoeba.xml：主配置文件，配置全部数据源以及 Amoeba 自身的参数设置。
- ◆ rule.xml：配置全部 Query 路由规则的信息。
- ◆ functionMap.xml：配置用于解析 Query 中的函数所对应的 Java 实现类；
- ◆ rullFunctionMap.xml：配置路由规则中须要使用到的特定函数的实现类；

假设您的规则不是太复杂，基本上仅须要使用到上面四个配置文件里的前面两个就可完毕全部工作。Proxy 程序经常使用的功能如读写分离。负载均衡等配置都在 amoeba.xml 中进行。此外。Amoeba 已经支持了实现数据的垂直切分和水平切分的自己主动路由。路由规则能够在 rule.xml 进行设置。

眼下 Amoeba 少有欠缺的主要就是其在线管理功能以及对事务的支持了，以前在与相关开发人员的沟通过程中提出过相关的建议，希望能够提供一个能够进行在线维护管理的命令行管理工具，方便在线维护使用，得到的反馈是管理专门的管理模块已经纳入开发日程了。另外在事务支持方面临时还是 Amoeba 无法做到的，即使 client 应用在提交给 Amoeba 的请求是包括事务信息的，Amoeba 也会忽略事务相关信息。当然，在经过不断完好之后，我相信事务支持肯定是 Amoeba 重点考虑添加的 feature。

关于 Amoeba 更为具体的用法读者朋友能够通过 Amoeba 开发人员博客 (<http://amoeba.sf.net>) 上面提供的使用手册获取，这里就不再细述了。

1. 11. 4. 4. 利用 HiveDB 实现数据切分及整合

和前面的 MySQLProxy 以及 Amoeba 一样, HiveDB 相同是一个基于 Java 针对 MySQL 数据库的提供数据切分及整合的开源框架, 仅仅是眼下的 HiveDB 仅仅支持数据的水平切分。

主要解决大数据量下数据库的扩展性及数据的高性能访问问题, 同一时候支持数据的冗余及主要的 HA 机制。

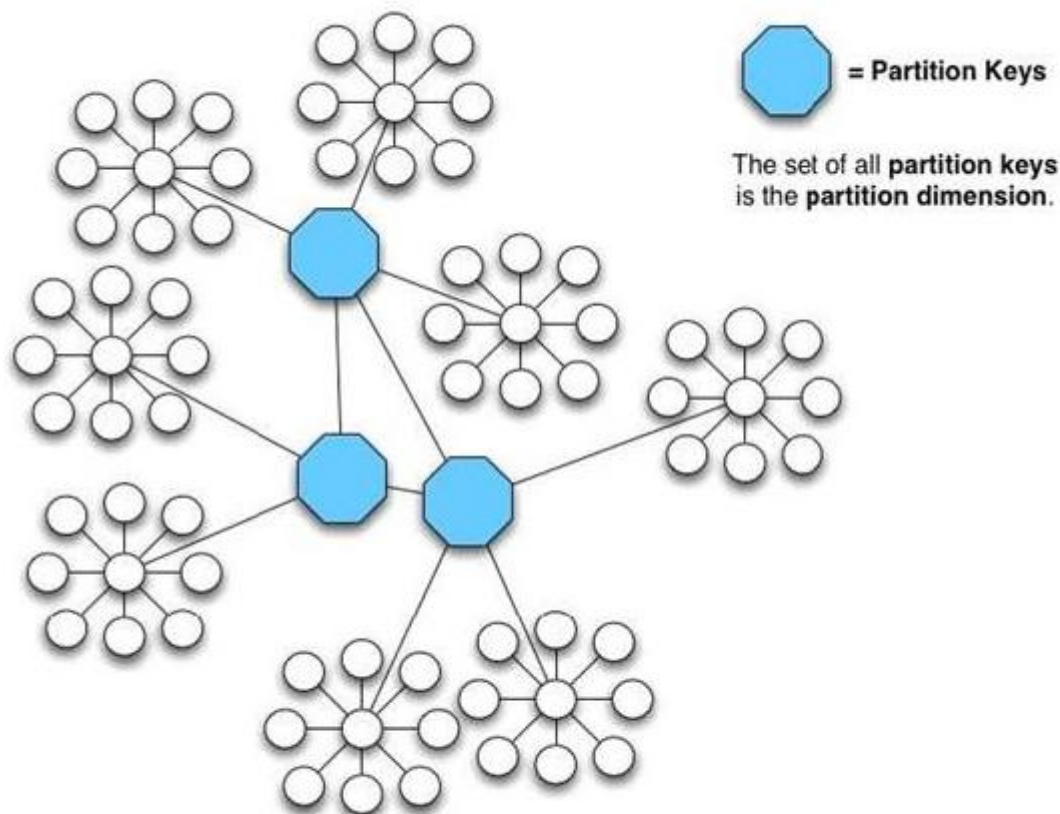
HiveDB 的实现机制与 MySQLProxy 和 Amoeba 有一定的差异, 他并非借助 MySQL 的 Replication 功能来实现数据的冗余, 而是自行实现了数据冗余机制, 而其底层主要是基于 HibernateShards 来实现的数据切分工作。

在 HiveDB 中, 通过用户自己定义的各种 Partitionkeys (事实上就是制定数据切分规则), 将数据分散到多个 MySQLServer 中。在访问的时候。在执行 Query 请求的时候。会自己主动分析过滤条件, 并行从多个 MySQLServer 中读取数据, 并合并结果集返回给 client 应用程序。

单纯从功能方面来讲, HiveDB 可能并不如 MySQLProxy 和 Amoeba 那样强大, 可是其数据切分的思路与前面二者并无本质差异。此外, HiveDB 并不仅仅是一个开源爱好者所共享的内容, 而是存在商业公司支持的开源项目。

以下是 HiveDB 官方站点上面一章图片, 描写叙述了 HiveDB 怎样来组织数据的基本信息, 尽管不能具体的表现出太多架构方面的信息, 可是也基本能够展示出其在数据切分方面独特的一面了。

Easily Partitioned Data Set



1. 11. 4. 5. mycat 数据整合

参见：<http://www.songwie.com/articlelist/11>

1. 11. 4. 6. 其它实现数据切分及整合的解决方案

除了上面介绍的几个数据切分及整合的总体解决方案之外，还存在非常多其它相同提供了数据切分与整合的解决方案。如基于 MySQLProxy 的基础上做了进一步扩展的 HSCALE，通过 Rails 构建的 SpockProxy。以及基于 Pathon 的 Pyshards 等等。

无论大家选择使用哪一种解决方案，总体设计思路基本上都不应该会有什么变化。那就是通过数据的垂直和水平切分，增强数据库的总体服务能力，让应用系统的总体扩展能力尽可能的提升，扩展方式尽可能的便捷。

仅仅要我们通过中间层 Proxy 应用程序较好的攻克了数据切分和数据源整合问题。那么数据库的线性扩展能力将非常容易做到像我们的应用程序一样方便。仅仅须要通过加入便宜的 PCServer，就可以线性添加数据库集群的总体服务能力，让数据库不再轻易成为应用系统的性能瓶颈。

1. 11. 5. 数据切分与整合可能存在的问题

在实施数据切分方案之前，有些可能存在的问题我们还是须要做一些分析的。

一般来说，我们可能遇到的问题主要会有以下几点：

- ◆ 引入分布式事务的问题。
- ◆ 跨节点 Join 的问题；
- ◆ 跨节点合并排序分页问题。

1. 11. 5. 1. 引入分布式事务的问题

一旦数据进行切分被分别存放在多个 MySQLServer 中之后，无论我们的切分规则设计的多么的完美（实际上并不存在完美的切分规则），都可能造成之前的某些事务所涉及到的数据已经不在同一个 MySQLServer 中了。

在这样的场景下，假设我们的应用程序仍然依照老的解决方式，那么势必须要引入分布式事务来解决。分布式事务本身对于系统资源的消耗就是非常大的，性能本身也并非太高，并且引入分布式事务本身在异常处理方面就会带来较多比较难控制的因素。

首先须要考虑的一件事情就是：是否数据库是唯一一个能够解决事务的地方呢？事实上并非这样的，我们全然能够结合数据库以及应用程序两者来共同解决。各个数据库解决自己身上的事务，然后通过应用程序来控制多个数据库上面的事务。将一个跨多个数据库的分布式事务分拆成多个仅处于单个数据库上面的小事务，并通过应用程序来总控各个小事务。

当然，这样作的要求就是我们的应用程序必须要有足够的健壮性。当然也会给应用程序带来一些技术难度。

1.11.5.2. 跨节点 Join 的问题

上面介绍了可能引入分布式事务的问题，如今我们再看看须要跨节点 Join 的问题。

数据切分之后可能会造成有些老的 Join 语句无法继续使用。由于 Join 使用的数据源可能被切分到多个 MySQLServer 中了。

怎么办？这个问题从 MySQL 数据库角度来看，假设非得在数据库端来直接解决的话，恐怕仅仅能通过 MySQL 一种特殊的存储引擎 Federated 来攻克了。Federated 存储引擎是 MySQL 解决相似于 Oracle 的 DBLink 之类问题的解决方式。

和 OracleDBLink 的主要差别在于 Federated 会保存一份远端表结构的定义信息在本地。乍一看，Federated 确实是解决跨节点 Join 非常好的解决方式。可是我们还应该清晰一点，那就似乎假设远端的表结构发生了变更，本地的表定义信息是不会跟着发生对应变化的。假设在更新远端表结构的时候并没有更新本地的 Federated 表定义信息。就非常可能造成 Query 执行出错，无法得到正确的结果。

对待这类问题，我还是推荐通过应用程序来进行处理，先在驱动表所在的 MySQLServer 中取出对应的驱动结果集。然后依据驱动结果集再到被驱动表所在的 MySQLServer 中取出对应的数据。可能非常多读者朋友会觉得这样做对性能会产生一定的影响，是的，确实是对性能有一定的负面影响，可是除了此法，基本上没有太多其它更好的解决的方法了。

并且，由于数据库通过较好的扩展之后，每台 MySQLServer 的负载就能够得到较好的控制。单纯针对单条 Query 来说，其响应时间可能比不切分之前要提高一些，所以性能方面所带来的负面影响也并非太大。更何况。相似于这样的须要跨节点 Join 的需求也并非

太多。相对于总体性能而言，可能也仅仅是非常小一部分而已。所以为了总体性能的考虑，偶尔牺牲那么一点点。事实上是值得的。毕竟系统优化本身就是存在非常多取舍和平衡的过程。

1.11.5.3. 跨节点合并排序分页问题

一旦进行了数据的水平切分之后，可能就并不仅仅有跨节点 Join 无法正常执行，有些排序分页的 Query 语句的数据源可能也会被切分到多个节点。这样造成的直接后果就是这些排序分页 Query 无法继续正常执行。事实上这和跨节点 Join 是一个道理。数据源存在于多个节点上，要通过一个 Query 来解决，就和跨节点 Join 是一样的操作。相同 Federated 也能够部分解决。当然存在的风险也一样。

还是相同的问题，怎么办？我相同仍然继续建议通过应用程序来解决。

怎样解决？解决的思路大体上和跨节点 Join 的解决相似，可是有一点和跨节点 Join 不太一样。Join 非常多时候都有一个驱动与被驱动的关系。所以 Join 本身涉及到的多个表之间的数据读取一般都会存在一个顺序关系。可是排序分页就不太一样了，排序分页的数据源基本上能够说是一个表（或者一个结果集）。本身并不存在一个顺序关系，所以在从多个数据源取数据的过程是全然能够并行的。

这样排序分页数据的取数效率我们能够做的比跨库 Join 更高。所以带来的性能损失相对的要更小，在有些情况下可能比在原来未进行数据切分的数据库中效率更高了。

1.11.5.4. 小结

当然，不论是跨节点 Join 还是跨节点排序分页。都会使我们的应用 server 消耗很多其它的资源，尤其是内存资源，由于我们在读取访问以及合并结果集的这个过程须要比原来处理很多其它的数据。

事实上全然不是这样，首先应用程序由于其特殊性。能够非常容易做到非常好的扩展性，可是数据库就不一样。必须借助非常多其它的方式才干做到扩展。并且在这个扩展过程中，非常难避免带来有些原来在集中式数据库中能够解决但被切分开成一个数据库集群之后就成为一个难题的情况。

要想让系统总体得到最大限度的扩展，我们仅仅能让应用程序做很多其它的事情来解决数据库集群无法较好解决的问题。

1. 12. 硬件及操作系统层相关优化

1. 12. 1. CPU 相关

在服务器的 BIOS 设置中，可调整下面的几个配置，目的是发挥 CPU 最大性能，或者避免经典的 NUMA 问题：

- 1、选择 Performance Per Watt Optimized(DAPC)模式，发挥 CPU 最大性能，跑 DB 这种通常需要高运算量的服务就不要考虑节电了；
- 2、关闭 C1E 和 C States 等选项，目的也是为了提升 CPU 效率；
- 3、Memory Frequency（内存频率）选择 Maximum Performance（最佳性能）；
- 4、内存设置菜单中，启用 Node Interleaving，避免 NUMA 问题；

1. 12. 2. 磁盘 I/O 相关

下面几个是按照 IOPS 性能提升的幅度排序，对于磁盘 I/O 可优化的一些措施：

- 1、使用 SSD 或者 PCIe SSD 设备，至少获得数百倍甚至万倍的 IOPS 提升；
- 2、购置阵列卡同时配备 CACHE 及 BBU 模块，可明显提升 IOPS（主要是指机械盘，SSD 或 PCIe SSD 除外。同时需要定期检查 CACHE 及 BBU 模块的健康状况，确保意外时不至于丢失数据）；

3、有阵列卡时，设置阵列写策略为 WB，甚至 FORCE WB（若有双电保护，或对数据安全性要求不是特别高的话），严禁使用 WT 策略。并且闭阵列预读策略，基本上是鸡肋，用处不大；

4、尽可能选用 RAID-10，而非 RAID-5；

5、使用机械盘的话，尽可能选择高转速的，例如选用 15KRPM，而不是 7.2KRPM 的盘。

1.12.3. 文件系统层优化

在文件系统层，下面几个措施可明显提升 IOPS 性能：

1、使用 deadline/noop 这两种 I/O 调度器，千万别用 cfq（它不适合跑 DB 类服务）；

2、使用 xfs 文件系统，千万别用 ext3；ext4 勉强可用，但业务量很大的话，则一定要用 xfs；

3、文件系统 mount 参数中增加：noatime, nodiratime, nobarrier 几个选项（nobarrier 是 xfs 文件系统特有的）；

1.12.4. 其他内核参数优化

针对关键内核参数设定合适的值，目的是为了减少 swap 的倾向，并且让内存和磁盘 I/O 不会出现大幅波动，导致瞬间波峰负载：

1、将 vm.swappiness 设置为 5-10 左右即可，甚至设置为 0（RHEL 7 以上则慎重设置为 0，除非你允许 OOM kill 发生），以降低使用 SWAP 的机会；

2、将 vm.dirty_background_ratio 设置为 5-10，将 vm.dirty_ratio 设置为它的两倍左右，以确保能持续将脏数据刷新到磁盘，避免瞬间 I/O 写，产生严重等待（和 MySQL 中的 innodb_max_dirty_pages_pct 类似）；

3、将 net.ipv4.tcp_tw_recycle、net.ipv4.tcp_tw_reuse 都设置为 1，减少 TIME_WAIT，
提高 TCP 效率。

1. 12. 5. 硬件升级

1. 12. 5. 1. 升级内存

1. 12. 5. 2. 增加 CPU

1. 12. 5. 3. 存储升级

1. 12. 5. 4. 带宽升级

版权所有 www.boxuegu.com