<高性能 MySQL(第 2 版)中文版-读书笔记>

作者: 宋先旺(315224416)

红色倾斜: 表示疑问 *#?问题内容*

蓝色倾斜: 表示解答(带下划线) *#.问题答案*

橙色加粗: 表示强调

其他定义:

/表示普通文本中的"或", |表示命令中的"或"; &表示"并且"

<label:"文本">表示内部标签引用

<import:"文本">表示外部文件引用

<url:"文本">表示外链

1，事务隔离级别

通过 **set [global|session] transaction isolation level** [隔离级别]进行相关设置；可以通过 global 与 session 上的 **tx_isolation** 变量获取当前的隔离级别；另外需要注意的是 global 级别的设置对已有的连接无效

1.1，隔离级别

Read uncommitted 可以读取其他事务尚未提交的数据(脏读)；**read committed** 可以读取其他事务已提交的数据(不可重复读)；**repeatable read** 可重复读；**serializable** 串行读(具有隐式的 lock in share mode)，其中 read uncommitted 与 serializable 不支持 Mvcc 特性，因为它们始终读取最新的版本

1.2，可重复读原则被违背的问题

在 **repeatable read** 隔离级别下不同的客户端分别同时开启线程 AB；表 t1(InnoDB)[x(int primary key auto)，y(int)][[1,2],[2,3]];

Start transaction；(A)

Start transaction；(B);

Select * from t1 where x=1 for update；(A)

Select * from t1 x=1;(B)

读取的是被(A)修改前的原始记录，打印[1,2]

Select * from t1 where x=1 for update；(B) 发生阻塞

Update t1 set y=200 where x=1；(A);

Commit；(A)；(A) 此时(B)得到独占锁解除解除阻塞状态

打印[1,200] (B)

#?为什么在使用显式锁的时候会出现不可重复读?

#.Mvcc 对于 lock in share mode 以及 for update 是无效的

1.3，幻读与不可重复读

不可重复读：比如中午去食堂吃饭，你好不容易找到了一个位置，丢下一本课本占位，接着去打饭(你认为会带回这个位置还是你的)，可是回来的时候你发现你占座的位置被人给坐了，record lock 可以解决这个问题

幻读：你找了一个没人的角落坐下，起身去打饭，准备回来独享清闲，可是回来看见旁边坐了一个恐龙妹，严重干扰你的食欲；gap lock 间隙锁解决了这个问题(mvcc 也同样可以处理这个问题)

Repeatable read 使用了 next-key lock(gap lock+record lock)所以不存在不可重复读以及幻读的问题

2，MySQL 中的锁

2.1，锁的使用方式(显式锁与隐式锁)

隐式锁 SQL 执行完成之后自动释放，而**显式锁**(for update/lock in share mode)则在事务提交或回滚之时释放；for update/lock in share mode 会读取最新的数据从而有可能出现幻读(见<label:"1.2，可重复读原则被违背的问题">)

Update，delete 操作的记录等同于加上了 select...for update 显式锁，所以他们也能"看见"最新的数据而并不理会快照

2.2，锁的类型

有 **X**，**S**，**IX**，**IS** 四种类型，分别为独占锁，共享锁，意向独占锁，意向共享锁

X 锁为独占锁，只有获取它的线程可以读取或修改被锁定的资源；S 锁为共享锁，可以被多个线程同时获取(读取资源)，但是不能在其他线程还没有释放 S 锁的情况下修改资

源(可能导致死锁)，innodb 对死锁的处理方式为回滚持有 X 锁最少的事务

#?IX 与 IS 锁在 MySQL 上是怎么实现的?

2.3，锁的行为

Record lock(行锁)：单个记录上的锁(支持较高的并发但开销比表锁大)，它是通过给索引项加锁实现的，意味着只有通过索引条件检索数据才会使用行级锁

Gap lock(间隙锁)：锁定一个范围但不锁定记录本身，防止幻读

Next-key lock：Gap lock+Record lock 锁定一个范围以及记录本身(区间为(left，right])

悲观锁：被操作数据从开始读取到回写完成之前数据始终处于加锁状态

乐观锁：读取数据时将数据的当前版本一并读出然后释放锁定，写入时将之前读取出的 version 加 1，如果大于数据当前的实际版本号则写入，否则视为过期数据予以拒绝，较之与悲观锁，乐观锁避免了长事务中的加锁开销，但是它依赖与系统逻辑从而有可能导致不遵循这一逻辑的第三方应用写入脏数据

需要注意的有两点：当查询结果超过总记录数一定的比例时 describe 的 type 为 ALL 并使用表锁；是否加锁以及加何种锁在 InnoDB 中取决与具体的隔离级别

2.4，关于锁的注意事项

在一条水平标尺中存在一定数量的 P 点(记录)以及 N 点(P 点之间的间隙)

如果在线程 T1 中使用"显式锁选定"或更新/删除一个范围或一行不存在的记录(select(X or S)/update(X)/delete(X))，那么会形成一个区间 L，L 的某一侧位于 N 点则自动向外扩充至距离最近的 P 点，最终可能形成一个区间(-，Pn]/(Pn1，Pnx]/(Pn1，+]/(-，+)，在线程 T2 中，在区间 L 内的 N 点不能插入任何数据，可以任意删除，锁定，更新其中的任何 N 点；对于 P 点的修改/删除/更新则完全取决于 T1 锁采用的锁(X or S)

关于锁的更多信息请参见<label:"18，锁的调试">以及<url:"http://www.mysqlops.com/2012/05/19/locks_in_innodb.html">

3，多版本并发事务控制(Mvcc)

在 MySQL 中每一个事务均用一个自增且唯一的 ID 表示，每个事务均会从特定的快照中读取数据；在 **Mvcc** 模式下，每一行数据均有两个隐含字段：创建它的事务版本与删除它的事务版本，一个特定的事务只能"看见"创建版本早于或等于自身"并且没有删除标记"或"有删除标记"但删除版本晚于自身事务 ID 的行

Mvcc 只所以具有更好的并发性能是因为它使用了**非锁定的读**且忽略 X 锁阻塞而直接读取回滚段中的内容(for update 与 lock in share mode 显式锁除外)

(1)，InnoDB 在只支持 read committed 与 repeatable commit(set [global|session] transaction isolation level 隔离级别)隔离模式下的多版本并发控制，因为 read uncommitted 始终读取行的最新版本，而 serializable 则强制事务操作排序(lock in share mode)也是读取最新版本

(2)，update/delete 某条记录的时候会把改动前的记录放入回归段(更新失败时恢复的依据)，同时也为版本号早于当前事务的其他事务提供查询一致性的保证；update 操作事实上创建了一个新行并写入自己的事务 ID(创建版本)，同时会修改原始数据的**删除版本**为自己的 ID，生成新行而不是直接修改原数据行是为了防止其他事务看到的是修改后的数据从而违背了可重复读的原则

4，MySQL 存储引擎

4.1，存储引擎摘要

MySQL 的表锁与索引一样由 MySQL 服务器实现，而其他锁由存储引擎实现；可以使用 show full processlist 查询各线程的查询状态(包括锁定状态)，**show engines** 可

以查看当前数据库系统对存储引擎的支持情况

服务器变量 **table_locks_immediate** 表示可以立即获取锁的查询次数，**table_locks_waited** 表示不能立即获取锁的次数(如果它很高建议检查服务器)，但是从 MySQL5.5 起所有对表的访问都需要获取新引入的 metadata lock，当被阻塞者的被 X 锁阻塞时 table_locks_waited 不会被加 1，因为被阻塞者还处于获取 metadata lock 的阶段

4.2，MyISAM 引擎(非事务)

(1)，MyISAM 读锁在特定的情况下允许并发插入，取决于一个参数 **concurrent_insert**(0/1/2:never/auto/always)，如果设置为 2(推荐值)则应定期 **optimize** 表；从并发插入的角度来看，MyISAM 可以适用于大规模插入/读取的情况，但是不能适应大规模更新/读取的情况(R/W 互不兼容)；同时它也支持延迟数据写入(insert delayed into)来提高客户端的效率

支持配置 **delay_key_write** 延迟索引写入(缓存在**键缓冲区**中)，可以为不同的 myisam 表设置多个键缓冲区，但是服务器崩溃会造成索引损坏；同时键缓冲区也存在互斥锁，因此可以考虑为修改频繁的表单独建立键缓冲区

短时间需要写入大量数据时可以通过 alter table ... **Disable**/**enable** keys 暂时禁用/启用索引，数据完成后可以通过排序来重新构建索引，但是对唯一索引无效，因为插入数据时需要载入唯一索引并检查新插入行的唯一性

(2)，分为三个文件.frm，.myi，.myd；有三种表格式：静态表(不包含 varchar，text，blob，出现故障容易恢复)，动态表(难以恢复占用空间少)，压缩表(row_format 选项控制)，关于表的详细信息可以通过 **show table status**(like ***)查看；在损坏时可以通过 **repair table** 修复表

(3)，支持全文索引(fulltext)以及压缩索引(pack_keys)，每个表最大索引数 64(每

索引最大 16 列)

(4)，支持使用 **myisampack** 工具生成压缩且只读的表，支持 repair table 修复表

4.3，InnoDB 引擎(事务)

(1)，InnoDB 的表文件与 MyISAM 有一些差异，InnoDB 没有 MyISAM 那样的 myi 与 myd 文件，取而代之的是共享表空间文件；**innodb_data_file_path** 配置为 innodb_data_file_path=datafile_spec1;spec2....，spec 格式为"存储路径:增长的步长[M|G]"，[:**autoextended**:max:最大容量]只能用来描述最后一个文件；也可以设置 **innodb_file_per_table** 为 ON 从而为每个表指定单独的表空间(*.ibd 索引与数据文件)

(2)，支持行级锁，间隙锁与**外键**(外键也在索引的基础上实现)以及 Mvcc，同时 InnoDB 支持聚集索引，不能自行选择聚集列而是按"主键->unique 列->系统自动生成隐含主键"的顺序进行自动选择

同时 NDB 以及 xtraDB 引擎也支持事务

4.4，Memory 引擎(非事务)

Memory 表支持 B-Tree 以及 Hash 索引，同时也支持延迟插入；不支持 blob 以及 text 类型的字段，对于 varchar 则隐式转换为 char 类型；可以通过 **max_heap_table_size** 设置 Memory 表可以增长到的最大大小，超过此大小则将会报错

临时表与 Memory 表有一些差别，临时表所使用的存储引擎取决于 **default-storage-engine** 的设置，它超过 **tmp_table_size** 大小时会转换为磁盘上的临时表；临时表通常由用户创建或在查询中自动生成

4.5，Federated 引擎(远程&非事务)

此引擎链接远程表需要在[MySQLD]节点加入 **Federated** 开启对它的支持，**show engines** 可以获取其开启状态；它支持 CRUD 索引但不支持查询缓存以及 DDL 操作

它在客户端服务器只有一个*.frm 文件，数据存储在远程服务器；创建方式为 create table t(***) engine=Federated connection="MySQL://用户名:密码@IP 地址:端口/数据库/表"

4.6，Merge 引擎

Merge引擎支持一组结构完全相同的MyISAM表的聚合且可以执行CRUD操作；假设有MyISAM表t1与t2，那么创建合并表create table t3(***) engine=**Merge union**=(t1，t2) **insert_method**=(first弟一个表，last最后一个表，no不允许插入)

合并表有很多缺点：不支持全文索引(即使基础表定义了)，只能使用相同的基础表，不能维持单值约束(primary/unique)，一旦唯一键或主键查询成功，那么就会立即终止查询而忽略其他的表

主要优点：可以处理单表大小超过 OS 的限制，在日志系统中多个小表在清理陈旧数据时具有速度优势

4.7, infobright 面向列引擎

其在数十 TB 数据级别时仍然可以良好工作

4.8, archive 引擎

仅仅支持 insert/select，mysql5.1 前不支持索引，数据保存到磁盘前会使用 zlib 算法进行压缩，特别适合日志类应用，支持快速的 insert，虽然不支持事务但是在批量操作完成前对其他线程不可见

4.9，本章总结

通常来说需要事务就选择 innodb/xtradb/ndb，需要记录日志就选择 myisam/archive，超大数据量(10TB+)选择 infobrightdb 或 tokudb

(1)，事务提交时会先修改内存中的数据形成脏页，脏页会在之后的某个时机刷新

到磁盘，然后再把修改行为追加到事务日志中(因为是顺序 I/O 所以比较快)，如事务已持久化到日志中但脏页并未写入到磁盘时发生崩溃，重启时存储引擎能自动恢复这部分被修改的数据

#?这里的恢复是指 undo(撤销)还是 rebo(重做)?

(2)，回滚不能撤销对非事务表的改动并且表定义语句(DDL)与锁表等操作会导致事务隐含提交，也不能回滚 DDL 操作；一条针对非事务表的 SQL 被中断可能导致部分数据被修改而部分数据没有得到更改

(3)，转储 alter table [表] engine=[要改变的引擎](对于 InnoDB 可达到整理碎片的目的)，事实上经过测试 InnoDB 表也可以使用 optimize 进行整理；事实上这个方法会产生一张新表并从原表导入数据，在导入数据完成前原表会被读锁占用

(4)，**check table** 检查表，**repair table** 修复表，**optimize table** 优化表碎片

(5)，对延时插入的支持 insert delayed(MyISAM/Memory/Archive)，InnoDB 也存在插入缓冲区的但是不能使用 delayed

5，基准测试与性能分析

5.1，概述

(1)，基准测试与性能分析分为"**集成式**"(Web,code,SQL server)与"**单组件式**"(仅 SQL server)；基准测试是测试系统能够达到的整体性能，而性能分析则解释为什么是这种性能

(2)，测试常用指标：**特定单位内事务处理量(TPS/QPS)**，**响应时间百分比(最大响应时间意义不大，因为测试时间越长，最大响应时间可能也越长)**，**扩展性**(连接数量变化/数据库大小变化/硬件调整，即通过硬件提升是否能提高其响应能力)，**并发性测试**(Threads_running 可以参考并发度)

TPS(每秒传输的事务个数)=(com_commit+com_rollback)/uptime(TPM 每分钟事务数)，对于 MyISAM 则使用 **QPS**(每秒查询量)=questions/uptime

#?如何获取特定时间范围内完成的各个事务的响应时间?

#.tcprstat 工具可以查看平均/最大/最小相应时间

(3)，并发性测试存在的问题：Web 服务器的处理能力与数据库服务器的处理能力不一定是对等的，不要透过 Web 等中间服务器测试数据库的能力；并发度通常理解为 MySQL 可以在单位时间内处理多少个链接请求以及响应时间是否在规定的时间内

5.2，基准测试需要注意的主要问题

(1)，测试环境与真实环境不一致(软硬件配置/网络环境)，还有一种特殊的情况既磁盘碎片

(2)，数据与用户行为没有参照真实环境，或者没有考虑用户行为的在不同时间段内的差异(访问频率/访问顺序等/访问热点)，可以开启查询日志获取用户真实的行为，但是要注意区分是多用户还是单用户的

(3)，没有考虑到数据库服务器暖机(如生成查询缓存)，操作系统的文件缓存

(4)，测试结果不具有普遍性，这个主要是受测试次数的影响，可能某个特定的时间点很快但其他时候很慢

5.3，测试工具

(1)，集成测试工具 sysbench，ab.exe，http_load，load runder....，需要注意的是集成测试工具的测试结果并不总能真实的反应数据库的处理能力，因为 Web 服务器的处理能力并不总是与数据库服务器的处理能力对等的

(2)，单组件测试工具(mysqlslap/sysbench/database test suite/sql-bench/super smack/tcprstat)

5.4，性能分析

5.4.1，概述

需要注意性能分析本身也有开销，可以在特定的存储引擎中使用 insert delayed into 加快处理速度<空闲时写入表>，因为日志的统计不需要具有实时性；在 PHP 中对 MySQL 的性能分析可以通过继承 mysqli 类重写 query 方法来实现(也可以使用 Php 中也可以使用 **xhprof** 以及 **ifp** 工具)，同时 mysql 也提供了企业监控器工具

性能并不意味着 cpu 占用率的降低，例如老版本的 innodb 升级到新版后出现的 cpu 占用率提升是因为对资源的利用效率更高了

5.4.2，MySQL 查询日志

5.4.2.1，普通日志

在 my.ini 的[MySQLD]节中添加 **log**="路径/文件名"，它会记录下所有查询(包括语法错误未能执行的)；因为普通日志是在 SQL 解析前记录的，所以只包含 SQL 本身的信息而不包含查询相关信息

5.4.2.2，慢速日志

在 my.ini 的[MySQLD]节中添加

log-slow-queries="D:/MySQL/MySQL Server 5.5/logs/sleep_query.log"

long_query_time=1 #阀值，为 0 表示捕获所有查询

MySQL5.2.1 之前 long_query_time 只支持以秒为单位，需要打补丁才支持设置为毫秒，之后的版本则不受此限制；可以使用 **mysqldumpslow** 工具对慢速日志进行分析，它可以按记录次数/查询时间/返回行数等约束条件进行分析

慢速日志提供了很多详细的信息如：是否命中查询缓存/是否全表扫描/

是否使用临时表；需要注意的是性能问题并不总是慢速 SQL 导致的，这是结果而不是原因，可能是大量不超过阀值的 SQL 导致另外一些 SQL 超时的，具体如下:

(1)，执行查询时发生表锁定

(2)，正在进行系统备份导致整体 I/O 缓慢

(3)，没有生成缓存

(4)，其他查询

5.4.3，分析查询计划

Describe select * from (select * from t1) as s1 inner join (select * from t2) as x2 on x1.x=x2.x

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	t1	ALL	NULL	NULL	NULL	NULL	3	Using where
3	DEPENDENT SUBQUERY	t3	ALL	NULL	NULL	NULL	NULL	1	Using where
2	DEPENDENT SUBQUERY	t2	unique_subquery	PRIMARY	PRIMARY	4	func	1	Using index

Describe 是 **explain** 的同义词，添加 **extended** 可以让服务器分析查询计划的时候保留下重建后的查询(通过 show warnings 查看)，**partitions** 可以查看特定的查询使用了那些分区

更多信息请参见<label:"附录 A: Describe">或<import:"Explain_语法详细解析.pdf">

5.4.4，监控服务器状态

show [**global**|**session**] **status** [like ***]监控服务器状态(global/session:所有连接/当前连接)，为了避免受到之前运行的干扰请使用**flush status**清零；大概作用如下:

(1)，查看之前的查询创建了多少临时表(子查询)或临时文件(created_tmp_disk_tables，Created_tmp_tables...)

(2), 查看服务器往来流量

(3), 查看服务器运行信息

(4), 定位其他性能瓶颈

更多信息请参见<import:"Show Status 中文对照表.xls">

5.4.5, profiling 工具

show profile 展示最后一条查询的生命周期内各个阶段的耗时情况, 必须先使用 **set profiling=1** 开启这个特性; show profile CPU for query [x], x 为一个数字, 通过 show profiles(注意末尾有 S)得到查询 ID

Show profiles

Query_ID	Duration	Query
1	0.00046825	SHOW WARNINGS
2	0.02401900	select * from t2 LIMIT 0, 1000
3	0.00748925	SELECT TABLE_CATALOG AS TABLE_CAT, TABLE_SCHEMA AS TABLE_SCHEM,
4	0.01541800	SELECT TABLE_CATALOG AS TABLE_CAT, TABLE_SCHEMA AS TABLE_SCHEM,

Show profile

Status	Duration
starting	0.000024
checking permissions	0.000006
Opening tables	0.004256
System lock	0.000018
init	0.000080
optimizing	0.000012
statistics	0.000015
preparing	0.000014
executing	0.000194
checking permissions	0.000102
checking permissions	0.003579

Show profile CPU for query 1

Status	Duration	CPU_user	CPU_system
starting	0.000102	0.000000	0.000000
checking permissions	0.000008	0.000000	0.000000
Opening tables	0.013032	0.000000	0.000000
System lock	0.000026	0.000000	0.000000
init	0.000050	0.000000	0.000000
optimizing	0.000007	0.000000	0.000000
statistics	0.000020	0.000000	0.000000
preparing	0.000007	0.000000	0.000000
executing	0.000002	0.000000	0.000000
Sending data	0.010191	0.000000	0.000000
end	0.000065	0.000000	0.000000

5.4.6，其他性能分析手段

其他一些性能分析手段包括 Web 日志，服务器嗅探器等

5.4.7，本章总结

(1)，变量；全局变量 global，@@global；会话变量 session，@@session，@@lobal，local；用户变量只能使用一个@否则会被视为系统变量(不存在则发生错误)

访问 select @@是先检查是否存在会话变量，如不存在则查找 global

(2)，**benchmark**(重复次数，表达式)测试特定表达式执行指定的次数需要的时间，建议使用 **sql_no_cache** 防止命中缓存

6，架构优化与索引

6.1，数据类型

应选择尽可能简单且更小的数据类型以减少"磁盘/CPU/内存/索引"的开销；对于索引来说应尽可能使用简单较小的数据类型，同时避免 NULL

(1)，NULL 不会出现在索引的叶子节点上，无论对索引列插入多少 NULL 值.myi 文件的大小均不会发生变化，但是需要注意如果复合索引中的某一列不为空，那么索引节点是存储了其他列的 NULL 值的；如果字段允许为 NULL 那么会多使用一个额外的字节表示字

段是否为 NULL

#?为什么 NULL 对于索引具有负面影响？为什么经过测试 MySQL 是可以为 NULL 的 is NULL 或 is not NULL 条件使用索引的(5.5.24-0ubuntu0.12.04.1-log)?

*#.NULL 不会出现在索引的叶子节点上"这样的表述是错误的，NULL 对于数据库效率的负面影响在于如果字段允许为 NULL 则会使用**额外的一个字节**来存储字段值是否为 NULL，索引扫描的时候会先检查字段值是否为 NULL，非空再读取实际的值，这样会有 cpu 与内存的开销*

(2)，整数，可以使用 x tinyint **Unsigned**(适用于所有数值类型)的形式强制只能存储正整数并且改变所能存储的值的范围(tinyint[-128:127]->Unsigned->[0:255])；int(1)/int(11)这样的形式只是对 MySQL 客户端等交互工具的一种显示的字符个数，对于存储和计算的数据来说没有任何影响

(3)，实数，float/double/decimal[总位数，小数点后位数]，分别占用 4/8/16 个字节；由于 float 与 double 存在计算误差，所以应使用更高精度的 decimal 类型进行货币计算

(4)，char 与 varchar;

char 最大长度为 255，它在存储时会自动添加空格占位符，取出时则去掉末尾的所有空格；而 varchar 只有在 4.1 之前的版本才会这么做；char 与 varchar 的限定单位为字符数而不是字节数，对于汉字占有几个字符取决于具体过程中使用的编码

varchar 比较节省存储空间(磁盘或内存)但会导致碎片(数据非连续存储会导致更新出现性能问题)且会使用 1-2 个字节存储值的长度信息，varchar 的使用时机为字段的最大值长度与平均长度存在巨大的长度差异且其不被频繁更新(请区别插入)为不同长度的时候

通常情况下 char 拥有更快的速度与更大的空间占用，具体使用 char 还是 varchar

取决与对 CPU 占用与存储空间占用的倾向；但这并不意味这可以定义一个非常大的 varchar 值，因为查询的时候 MySQL 会根据 varchar 定义的长度申请内存块；经常变化但长度相对固定的数据使用 char，其余的尽量用 varchar

对于部分存储引擎来说 char 与 varchar 并没有差别，比如 Memory 会自动转换 varchar 为 char

(5)，text/blob，更大的数据可以使用 **longtext/longblob** 类型，blob 表示与字符集无关的二进制存储

在达到一定的长度上限的时候会在磁盘上单独存储这些数据，每个具体的行存储了 1-4 个字节(估计是用来存储内容的地址)；因为 text/blob 列一般都比较大，对他们的操作(检索/排序等)会一下子加大结果集的体积从而很容易超出 **tmp_table_size**(使用磁盘临时表的阀值)的限制，因此应尽量避免在索引中使用 text/blob，一定要使用也可以使用类似于 order by substring 这样的方法限制 text 或 blob 的大小避免其超过 tmp_table_size 的阀值；另外可以考虑对它们使用局部索引，因为它们不支持全列索引

如查询中需要使用**隐式临时表**且用到 blob/text，因为 memory 不支持 blob/text 所以会使用基于磁盘的 myisam 临时表导致性能损失，解决方法则为使用 substring

#?隐式临时表与 create temporary table 创建的表有什么不同?

(6)，enum 类型；对于取值可以固定在一个列表中的字段来说(如学历:小学/初中/高中/大学)，使用 enum 比使用 char 拥有更高的效率，因为 enum 将字符串转换成数字(select enum 列+1 from xxx)存储并且按数字排序而不是字符串排序(比较时按字符串比较)，但是有一定的映射转换开销

如果 enum 会被频繁变更，那么就必须考虑 alter table 带来的阻塞

(7)，set 类型(最多 64 个成员)；create table t1(y **set('a', 'b', 'c', 'd')**);insert

into t1 values('a,b'),('d');select * from t1 **find_in_set**('b', y); set 数据类型非常适用于使用 ACL 的情况，判断用户是否具有相应的权限；set 存储时保存为数字，但是比较时采用字符串方式

总结，尽可能的使用小的数据类型并且尽量使用数字存储数据，比如：IP 地址可通过 **inet_aton**/**inet_ntoa** 函数"还原/转换"成数字存储，始终非负的数值字段也可以使用 **Unsigned** 尽量使用更小的数值类型

多表联接查询最好不要超过 12 张

6.2，索引

MySQL 的索引与行级锁一样由具体的存储引擎实现而不是服务器层实现，索引只所以快是因为最大程度上避免了全表扫描以及减少锁定

6.2.1，B-Tree 索引

不同的存储引擎会使用不同的算法：InnoDB 使用 B+，NDB 使用 T-Tree

B-：非叶子节点最多能有两个儿子(只有一个根节点)，左边的子节点存储小于自身关键字的值，右边的子节点存储大于自身关键字的值；B-的检索性能类似于二分，但是优于二分的是插入，删除节点不需要大量移动内存中的数据，为了防止 B-在大量的操作中变形，通常会引入额外的平衡算法；检索有可能在非子节点结束；特定的关键字在树中只出现一次

B+(使用了稀疏索引)：B-的变体，非叶子节点的指针树于关键字个数相同，非叶子节点的指针 P[i]存储的值为 K[i]-k[i]+1，所有的叶子节点处于同一个链表上，P[i]不仅有指向子树的指针，也有指向兄弟节点 P[i+-1]的指针

更新关于 B 树的信息参见<import:"B-TREE 索引.doc">

(1)，B-Tree 适用于**范围**/**排序**(索引节点已排序)/**全键值**(等于或不等于)/**最左**

前缀(复合索引[a,b,c]分解为三个索引[a]，[a,b],[a,b,c])/**列前缀检索**(比如 like 'XXX%')类型的查询

在复合索引[a，b，c]中，如果某一列使用了范围查询，那么其右侧的列则无法通过索引优化查找，如 a = 10 and b like "a%" and c=12 这时候只有前两列可以使用索引

因为**查询优化器是基于开销的**，所以对于复合索引[a，b，c]在特定的情况下比如需要取出的列均属于同一复合索引且不包含额外的其他列的情况下可以使用列的任意组合(a=X or c=X/b=X or c=X)或 c like '%X'这样违背列前缀原则的表达式的情况下依然可以使用索引加速查询，只要 MySQL 自认为开销是合理的；但是需要尽量避免出现这样的表达式；更多信息请参见<import:"MySQL 索引使用笔记.doc">

(2)，**覆盖索引**(仅 B-Tree 支持；经验证 Memory/Falcon 不支持覆盖索引)；它避免了对原始行的访问即要查询的数据或条件在索引中可以直接获取，限制条件为需要取出的列必须位于同一索引

(3)，**合并索引**(index_merge)；在多个不同且独立的索引上进行并行扫描然后合并结果；5.0 版本之前的 MySQL 不支持这个特性(可变通为多个 union 分散索引查找)从而导致了全表扫描；合并索引的缺点为并行查询与合并操作可能会导致相对较高的 CPU 负载；合并索引有三种：交集(or)/并集(and)/排序并集；如果因为某些条件出现了 range 查找也可以通过 **use index**(k1，k2)强制使用 **index_merge**

#?排序并集是什么意思?

6.2.2，Hash 索引

目前只有 Memory 与 NDB 支持 Hash 索引，InnoDB 则支持通过 **innodb_adaptive_hash_index**(on/off)非人工干预的自适应的 Hash 索引(为引用频繁的索引)；另

外 Hash 索引不支持索引覆盖

Hash 索引使用特定的算法为一列或多列(复合索引)生成一个唯一的值(内部存储为数字)，所以它只适用于全键值查找(in 或不等于)，对于多个行的哈希碰撞将会在碰撞区间内使用链表查找，因而不适用于太多碰撞的列

自定义哈希索引：新建一个列，存放某个字段的 Hash 值然后为这个列定义索引(在存储引擎不支持 Hash 索引的情况下模拟 Hash 索引)，这通常用在列值过长的情况下建立 btree 索引时索引过大，从而通过 hash 算法将较长的值变为较短的值；hash 函数的选择上建议不要使用 sha1/md5/crc32，前两者生成的值较长，后者会存在大量碰撞；使用 hash 索引时必须携带 hash 值以及原值防止碰撞，如 username_md5 = md5(username) and username="xxx"

6.2.3，索引优化

(1)，隔离列(防止覆盖索引失效)

索引列直接参与表达式的算术计算(+/-/concat 等)或传入的值与索引列使用不一致的校对规则则会禁用索引覆盖甚至导致索引无法被查询使用，如下:

索引 x char(255) **character set** utf8 **collate** utf8_general_ci

Select * from 表名 where x+1=2; //x 为索引列

Select * from 表名 where concat(x, '.')='aaa';

Select * from 表名 where x=**_latin1**"传入值" **collate** latin1_german1_ci

(2)，为前缀建立索引

在满足业务要求的情况下可以只为列的前 x 位建立索引，它可以显著的减小索引文件的大小；**索引选择性=count(distinct substring(column, 1, x))/count(*))**，这

个值越接近 **count(distinct column)/count(*)**越能精准定位到需要的记录，需要在选择性与索引文件大小之间权衡；同时前缀索引也赋予了 blog/text 等不支持索引的列的索引功能

6.2.4，聚集索引与非聚集索引

聚集索引**仅InnoDB/SolidDB支持**，在每个表上只能有一个且在MySQL中不能人工选择聚集列；聚集列选择顺序：主键->非空(not NULL)且unique的索引列->聚集表自动生成的隐藏主键(6个字节的指针)

聚集索引优点：读取时进一步减少I/O(主索引叶子节点处即数据行)；聚集索引缺点：对聚集列的更新/删除或插入成本很大；非聚集索引列变大，因为非主索引的叶子节点存储的**不是行指针而是主键值**；对非主索引的检索导致**二次查找**，如表create table t1(x int primary key, y int, key(y))执行select * from t1 where y=12则等同于select * from t1 where x in(select x from t1 where y=12，直接查找聚集列则不会有这个问题

聚集列的选择一定要注意，否则会造成大量的数据需要移动，通常使用的auto_increment是一个不错的选择，但是要注意auto_inrement锁，即上界竞争：select max(auto_increment column) from table for update，新版的MySQL减轻了这个锁的冲突，auto列必须是在索引列上，可以重复相同的值，如不指定则新列增长为max(auto)+1

非聚集索按插入顺序分布，优点是更新插入速度快，访问速度则较聚集索引慢(比聚集索引二次查找快)；聚集索引与非聚集索引的差别在于，非聚集索引的索引节点除存储是是**索引值与行的物理地址**，而聚集索引除了主索引外其他索引的索引叶子节点处存储的是**索引值与主键值**

6.2.5, 使用索引加速排序(order by)

如果有多个列, 那么必须严格按照复合索引的顺序(不能像普通索引那样可以中间跳过列或使用不同的索引)并且使用一致的排序方向才可以; 如果跳过了复合索引中的列, 那么被跳过的列必须在where中以常数(即相等)的形式出现;

加入存在表t1[w, x, y, z], 复合索引(x, y, z)

索引排序的条件(using index):

(1), select x from t1 order by x asc, y asc, z asc; 不解释

(2), select x from t1 where y='ddd' order by x asc, z asc; 可以使用索引, 虽然order by中没有第二列y, 但是where中使用了y形成了最左前缀

(3), select x from t1 where x>'ddd' order by x asc, y asc; 可以使用索引, 因为虽然在where中对x进行了范围查找, 但是在order by中还是形成了最左前缀

非索引排序的条件(using filesort):

(1), select x from t1 order by x asc, y desc; 排序方向不一致

(2), select x from t1 order by x asc, y asc, z asc, w asc; 使用了不在复合索引中的列w(或其他索引)

(3), select x from t1 order by x asc, z asc; 跳过了中间的列

(4), select x from t1 where x>'abd' order by y asc, z asc; 跳过的列使用了范围条件

(5), select x from t1 where x='abd' and y in('fds', 'gasd') order by z asc; 同(4)

文件排序(use filesort)是没有用到索引加速排序的查询的统称, 并不是真正一定使用了文件; 这时MySQL使用自己的快速排序算序法在内存中(**sort_buffer_size**每线

程独立使用)中进行排序；如果结果集太大以致超过sort_buffer_size，则将磁盘上的数据分块后排序(**Created_tmp_files**会增加)，最后通过合并各块从而生成最后结果

文件排序又分为**双路排序**与**单路排序**，主要取决于返回字段列的定义长度是否超过了**max_length_for_sort_data**设定的阀值，如未超过则使用单路排序；双路排序先取出排序字段与行指针，排序后再根据行指针读取其余的字段，这样的话造成第二次的随机访问；单路排序(MySQL4.1后支持)是读取所有的符合条件的行的所有列然后再排序，没有二次查找，缺点是内存空间占用会比较大

如果采用单路排序，对**sort_buffer_size**的要求会变大(需要提取的字段都在缓存中)，也可以设置**tmpdir**指向到更快速的设备以加速临时文件的访问速度

如果数据来源为一个以上的表，那么还会使用**using temporary**保存中间数据

6.2.6，压缩索引(仅 MyISAM 支持)

(create table ...**pack_keys**=0|1|default，0 不压缩，1 压缩，default 只压缩char与varchar列)；压缩原理为：如果前三个值分别为"hello jack"与"hello,jim"，"hello,lilei" 那么压缩"hello "，存储为"hello jack","6jim","6lilei"，这样会使索引文件变小，但会使CPU负载增加(权衡CPU负载与I/O负载)；而且还导致一个问题就是不能进行二分查找从而只能顺序查找，因为每一个前缀压缩的值都取决与之前的值，每个关键字需要一个额外的字节只是前一个关键字中有多少字节与下一个关键字相同

6.2.7，多余索引与重复索引

(1)，重复索引

create table t1(x int primary key, unique(x), key (x))，因为MySQL对于**primary key**/**unique**是使用索引来实现的(**foregin**也是使用索引实现)，所以相当于建

立了三个不同的索引，SQL语句create table t1(x char(20), key(x), key(x))也是如此，查询优化器会逐个检查它们的使用成本从而降低了效率并且还会存在多个索引的维护成本

(2)，多余索引

(A,B)与(A)，在某些情况下即使不需要针对B进行查找也可以为B建立基于A的复合索引，具体原因参见索引覆盖；如存在多余索引**I1[A，B]**/**I2[A]**，那么在某些只需要使用索引I2的地方可能会使用I1，因为查询优化器**按照索引建立次序**检查索引，当它认为某个索引的开销最优的时候会忽略余下的索引，即使后面的索引是最优的；可以使用**ignore index**(keylist)禁止使用特定的索引/**use index**(使用特定的索引)/**force index**(同use index)控制索引的使用规则；由于开销的原因某些可以走索引的查询会直接使用表扫描，force index较之use index提高了这个阀值

6.2.8，索引与锁定

并非所有符合索引的条件都会通过索引优化查询，如果需要返回的行与总行数值比超过一定的范围并且不能使用索引覆盖(使用索引优化查询并不等同于覆盖索引)的情况下，会执行全表扫描并使用表锁

6.2.9，更新索引信息

查询分析器会调用存储引擎API来估算范围内的**数据量**与数据基数性(每个键值有多少数据)以了解索引的分布情况；如果这个数据不准确会使查询优化器做出错误的优化决定，使用**analyze table**更新索引信息(存储引擎内部也会定期自动调用)；**optimize talbe**优化索引以及数据碎片，对于不支持optimize的引擎alter table xx engine=xxx也就有与optimize一样的效果，但是需要注意这种方式会先创建一个新表，然后复制原表数据到新表，此时原表是有读锁存在的，手工导出数据并重新导入就不会有这个问题

6.3，本章总结

统计数据的并发问题，比如要统计网站的点击量，可以通过把累加分布到多条统计行然后通过sum来减少更新时的行锁等待；select * from t1 where x=ceil(rand()*100)，中可能筛选出多条x值不同的记录，**rand**函数与其他函数不同，在对每条记录进行比较时都会重新调用

在MySQL中，alter table会新建一个需要的目的表，然后从源表中导入数据(这中间会耗费大量的时间)，然后删除源表；alter table...modify column就是这样的行为；alter column就不会有这样低效的操作过程；需要注意的是change column，alter column，modify column具有不同的行为

可以谨慎的使用交换.frm方法低成本的修改表结构，先使用create table ... Like .. 新建一张结构相同的表，变更(通常是增加或修改)新表的结构，然后用新表的.frm覆盖原表的frm，再此期间需要做flush tables with read lock/unlock tables方式数据写入

7，查询优化

7.1，查询优化摘要

(1)，限制返回的行与列的数量，提取全部列可能导致索引无效以及大量的网络传输

(2)，MySQL开销指标：需要检查的行数(比如join)以及返回的行数，执行时间；如发现返回少量的数据需检查大量的行时要考虑查询设计是否合理

(3)，周期性的清除陈旧数据并在服务器空闲时汇总表

(4)，分解查询，将复杂的操作拆分成简单的操作并且前置部分操作(比如使用前端排序，联接，分组等...)

7.2，查询执行周期

发送查询到服务器->服务器检查是否有缓存(返回结果前检查了权限)->解析优化查询并生成执行计划->调用存储引擎api执行查询->返回查询结果

MySQL的通信协议是基于半双工的，因此在服务器数据发送完成之前，当前客户端无法与服务器进行通讯并且服务器不会释放数据资源的锁定，使用**sql_buffer_result**提示符以及临时表模式的视图(**Algorithm=temptable**)可把结果集放入临时表并提前释放锁定

查询包含了很多特定的状态(show processlist)；休眠/查询/锁定/分析和统计(检查索引分布/优化查询)/复制到磁盘临时表/排序/发送数据等；需要注意的是，MySQL查询优化器生成的执行计划是一个数据结构而不是字节码

7.3，查询优化流程

7.3.1，解析器与预处理器;

解析器生成解析树并检查各个标识符是否有效并且在适当的位置上(**语法检查**)，预处理器检查资源是否存在并且是否具有访问权限以及是否存在语义问题(**语义及权限检查**)

7.3.2，查询优化器(基于开销)

查询优化器根据**表或索引的页**，**索引的基数性**(count(distinct substring(column，1，x))/count(column))，**键或行的长度**以及**键的分布情况**等综合数据来判断一条查询的最优开销，但并不总是最优的，因为它假设所有数据都需要从磁盘获取；show status like '**last_query_cost**'(查询优化器评估这条查询大约需要的随机读取数)可以查看最近一条查询的开销；但是索引的统计信息不是精确的并且不同的存储引擎有不同的统计策略(如archive这样的引擎根本就不保存统计信息)，必要的时候使用**analyze table**更新这个信息

MySQL对于查询的优化只考虑SQL语句本身，而忽略**并发因素/缓存/用户函数的执行成本**等；查询优化器不会依次检查所有方案(见<label:"6.2.7，多余索引与重复索引">)；当需要检查的行数太大的时候，MySQL可能会放弃索引优化而直接扫描物理行，需要特别注意的是子查询生成的结果集是没有索引的

(1)，静态优化(仅执行一次)

静态优化(编译时优化)是根据解析树(不考虑具体的参数值)来进行优化；根据代数等量转换法则把复杂的表达式简化成相等的更精简的表达式；可以通过**describe extended+show warnings**查看重建后的查询

(2)，动态优化(每次都要重新评估)

动态优化(运行时优化)；考虑上下文/参数值以及索引分布情况，每次都会重新检查优化方案(需要注意的是统计信息不是绝对精确的)

优化类型:

(1)，联接中的表重新排序

MySQL会自动调整表的顺序已减少需要读取的行数(可以参考**describe** rows属性)，如需自行调整请使用**straight join**

(2)，外联结转换为内联接；如select * from t1 left join t2 using(iu) where t2.iu is not NULL可以转换为等价的inner join格式，与具体的查询条件有关系

(3)，代数等价简化法则，移出不可能的限制条件并且简化表达式，如(a<b and b=c) and a=5转换为b>5 and b=c and a=5，5=5 and a=5简化为a=5

(4)，等价传递法则，Select * from t1 inner join t2 using(x) where t1.x=1中x对所有表都是等价的，这时候会条件会变为t1.x=1 and t2.x=1，可以通过show warnings查看；需要注意的是union并没有这个特性

(5)，覆盖索引

(6)，子查询优化，简化为索引查找而不是多个独立的查询；*#?是怎么优化的?*

(7)，早期终止(Impossible WHERE)，在列上应用了不可能的条件比如x定义为not NULL 但是查询使用is NULL；那么在优化阶段就提前终止并返回结果而不是继续查询

(8)，in排序，很多其他数据库是把in转换成多个or，但是MySQL是把in作为一个列表使用二分查找(较大的list会引发性能问题)

(9)，优化count/max/min；对于索引列的min与max会使用(**Select tables optimized away**通过索引直接一次定位)，对于count的优化只有MyISAM可以使用Select tables optimized away(MyISAM已保存总行数)，优化后会从执行计划中移除这个表并使用一个常量来代替它；与MyIsam不同的是InnoDB的count为using index(需要走索引才知道总行数)所以无法使用**Select tables optimized away**优化

(10)，评估和简化常量表达式(type:const)；重复执行不会发生值改变的表达式如year(now())(但包含类似rand函数的表达式除外)或通过primary与unique**能且只能找到一行数据**的select(包括min/max)均会被简化为一个常量，余下的部分将可以直接使用这些常量(ref:**const**)

7.4，查询优化器以及它的限制

7.4.1，非关联子查询

子查询只运行一次，与外部查询无关;

7.4.2，关联的子查询(dependent subquery)

标准句式select * from t1 where exists(select * from t2 where t2.x=t1.x)，外部查询的会把每一条结果传递到内部查询重新执行一次；MySQL不能在子查询中

读取而外部查询中删除或更新同一个表的原因也正在于此(想象一下在同一个数组的双重循环的内部循环中改变数组的情况)，可以通过update t1 set *** where ID in (select ID from (select ID from t1) as t)的方法规避这个问题

7.4.3，优化的限制

(1)，union的限制；MySQL不会利用外部的查询条件应用到union的各个子查询，如果没有all，那么服务器会自动使用distinct

(2)，合并索引优化(index_merge)，见<label:"6.2.1，B-Tree索引">

(3)，相等专递，对于应用了in的列，如果MySQL检测到这个列与该条查询中其他表的某些列相等，则MySQL会复制in的结果集到其他表，在结果集过大的时候会导致执行与优化的效率问题；*#?什么是相等传递且为什么是复制而不是引用呢?*

7.5，优化特定类型的查询

7.5.1，count 的优化

count的优化(可以使用索引覆盖)；传入*表示忽略所有列的值统计表的行数，如使用具体的列则为此列为NULL的行不会被统计到；MyISAM自身保存了总行数，而InnoDB则需要扫描表，目前测试MyISAM为**Select tables optimized away**，而InnoDB仍然使用**using index**

对于带where的统计有一种比较取巧的优化方案，select count(*) from t1 where x>5可能会扫描很多行，可以优化为select (select count(*) from t1)-count(*) from t1 where x<=5这样就只需要扫描5行数据(在有索引的情况下)

7.5.2，优化 group 与 distinct

Group by 比 order by 多了排序后的分组操作，可以使用 asc[默认]/desc 控制分组数据的排序方向，使用 **group by column order by NULL** 则节省分组的排序开

销

在不能使用索引的情况下分组会扫描整个表，将得到的数据按分组顺序连续填充到创建的临时表(**using temporary**)，然后应用聚集函数(如果有)，最后排序后输出;但是即使在应用索引(using index)的情况下仍然可能出使用文件排序(using filesort)的情况，这是因为虽然可以在索引中取出所有的字段但group额外的排序操作(order)违反了索引排序的原则，见<label:"6.2.5，使用索引加速排序(order by)">

#?什么是松散索引扫描(Using index for group-by，167页)以及紧凑索引(using index)扫描？对于复合索引[a，b，c]，看书上的解释似乎它们的差别在于松散索引要求的组合可以为group by [a]，[a，b]，[a，b，c]；而紧凑索引则可以在group中跳过列，只要被跳过的列在where中以常量的形式出现?

其他:

关于group by a，b，c，d **with rollup**(对列顺序敏感)；除了常规的行为之外，with rollup具有这样的功能，(a，b，c，d)，除了最后一列(常规分组)外，分别再分组，(a)，(a，b)，(a，b,c)，如(select country，provice，city，sum(point) as total from sales group by country，provice，city with rollup):

country	provice	city	total
中国	四川	成都	3852
中国	四川	雅安	30
中国	四川	(Null)	3932
中国	江苏	南京	52
中国	江苏	(Null)	52
中国	湖北	武昌	77
中国	湖北	襄樊	12
中国	湖北	(Null)	89
中国	(Null)	(Null)	4073
美国	纽约州	曼哈顿区	1000000
美国	纽约州	(Null)	1000000
美国	(Null)	(Null)	1000000
(Null)	(Null)	(Null)	1004073

Distinct作为group的一种特殊形式，可以参考group的优化方法，同时它也会支持；查询select distinct t1.a from t1，t2 where t1.a=t2.a会在发现t2中的第一个t1.a时停止开始下一轮操作而不是继续读取

7.5.3，使用临时表的条件

(1)，**sql_buffer_result**强制将结果集存放到临时表以尽快释放锁；*#?实际测试为什么并未导致Created_tmp_tables增加?*

(2)，**sql_small_result**，*#?是否必须与group by以及distinct一起使用?*

(3)，from中的子查询

更多信息参见<url:"http://dev.MySQL.com/doc/refman/5.6/en/internal-temporary-tables.html">

7.5.4，使用临时文件的条件

#?除了不合理的order之外还有什么情况会使用临时文件?

7.5.5，优化 limit

对于不能走索引且偏移量很大的分段，比如limit 100000，20会产生100020行数据并丢弃前100000行；变通的方法是select t1.* from t1 inner join (select x fro

m t1 limit 10000, 20) as t2 using(x)使用**索引偏移**来代替物理行偏移；如果在页面中存在一个连续递增并且唯一且可以预知的值，可以考虑直接使用范围定位

7.6，查询优化修饰符

(1)，**high_priroity**与**low_priority**；high_priority可以应用于select与insert语句，它让SQL跳过队列中的其他SQL优先执行；low_priority则相反，如果有其他语句访问数据，那么它就自动排到最后(适用于select，insert，delete，update，prelace)

但是它们只在有表锁的存储引擎上有效，其他比如InnoDB这样细粒度锁或者支持并发控制的引擎上无效；即使对于MyISAM也应该谨慎的使用(会禁止并发插入)，它仅仅修改了队列

(2)，**Delayed**延迟插入；操作结果会被放入缓冲区，不能得到**last_insert_id**()；插入速度较快但只有部分引擎支持(memory/myisam/archive)，InnoDB自带了写入缓冲区

(3)，**straight_join**，应用于join强制按顺序联接表；它的原则为让MySQL访问尽可能少的行

(4)，**sql_small_result**告诉group或distinct查询结果集较小，使用索引过的临时表排序；**sql_big_result**则表示结果集较大，使用磁盘上的临时表排序

(5)，**sql_buffer_result**，告诉服务器把结果集保存在临时表中并尽快释放表锁，因为服务器没有发送完数据的时候不会释放表锁，它会减少锁冲突，但是会增加服务器内存开销

(6)，**sql_cache**与**sql_no_cache**，控制是否将结果放入缓存；它们是否生效取决与**query_cache_type**(0/1/2)的设置

(7)，**sql_calc_found_rows**与**found_rows**()；统计一个select在没有limit的情况下应该返回多少行；经测试在实际使用过程中效率不会比另外使用count统计高

(8)，**ignore index/use index/force index**；use/force index两者作用类似，但后者对是否使用全表扫描具有更高的阀值

(9)，**optimizer_search_depth**(控制优化器检查执行计划的搜索深度)，**optimizer_prune_level**(让优化器根据要检查行的数量跳过某些查询计划)

7.7，用户变量以及表达式执行顺序

使用用户变量的缺点：禁止在查询缓存或需要文字常量或标识的地方(表名/列名/limit)处使用，5.0之前区分大小写，定义时最好有默认值

需要注意在应用用户变量是要注意select/where/order处于查询的不同阶段，表达式执行顺序为为from(执行一次)->where(执行多次)->group(多次)->having->order(多次)->select(多次)；*#?实际执行是不是这个顺序且如何验证?*

8，MySQL 高级特性

8.1，MySQL 查询缓存

很多数据库会缓存查询计划从而使后续查询跳过**优化与解析**的阶段，MySQL则更进一步缓存了执行阶段；流程为：检查是否为select语句->是否具有缓存(区分大小写且全文本Hash匹配并检查了用户权限)->解析优化执行(标注是否可被缓存)->缓存结果并发送

8.1.1，查询缓存的限制条件以及开销

不能缓存包含了**用户自定义函数**以及其他**非确定结果的函数**(current_user，connection_ID等)，**存储过程**(包括存储过程里的独立查询)，**变量**，**临时表**(create temp table或临时表模式的视图)，**显式锁定**，MySQL**系统数据库**中的表以及具有**列级权限**的表以及超过**query_cache_limit**大小的结果集；缓存的粒度为客户端发送的完整select，因此查询内的子查询不会被单独缓存

缓存的开销；查询前需要检查缓存，保存缓存带来的开销(同时需要清理之前的缓

存)；对表的结构或内容的任何更改将无条件失效涉及到该表的缓存；在事务中对表的改变即使对外部不可见(即使有Mvcc)也会导致所有引用该表的缓存失效且关于该表的数据无法被缓存，直到事务被提交为止，因此**长期运行的事务会降低命中率**；另外5.1之前prepare也不会被缓存；对于**MyISAM**表可以使用**query_cache_wlock_invalidate**控制当表被write锁定时是否允许其他线程读取涉及到该表的缓存(0允许)

对于大数据量的查询缓存来说，让它失效会是一个比较长的过程，并且会产生一个全局锁阻塞其他线程对查询缓存的访问；检查查询是否命中以及是否有查询失效的时候也会存在全局锁

#?全局锁是否如x锁那样是一个独占锁?

8.1.2，缓存如何使用内存

查询缓存内存池(**query_cache_size**)由MySQL预申请并管理，系统保留40KB用于分配数据结构，剩余的空间会在缓存结果时被分成大于或等于**query_cache_min_res_unit**大小的块，具体大小由MySQL进行预估，因为MySQL不能预先知道结果集的大小而是产生一行发送一行，如果不够则继续申请新快(分配速度相对较慢所以会在分配次数与空间使用率上进行平衡)，多余的部分则裁剪放入空闲空间，但是如果裁剪后的空间小于**query_cache_min_res_unit**则不能重新分配使用(碎片)，需要使用**flush query cache**进行整理

8.1.3，缓存的应用场景

适用与产生结果需要很长的时间，并且结果集相对较小的查询；命中率**qcache_hits/(qcache_hits+com_select)**，可以通过**qcache_lowmem_prunes**获取多少查询因为内存不足而失效(为缓存新查询而移除的查询缓存数)；命中率不是一个恒定的指标，对于开销很大的查询，即使极低的命中率也是值得的

Com_select(未命中的查询)=**qcache_inserts**(加入到缓存中的查询数量)+**qcache_not_cached**(缓存未打开或其他原因导致的不能缓存的查询数量)+queries with errors found by during columns/rights check

Select查询总数=qcache_hits(命中的)+com_select+queries with errors found by parser, 也可以通过**questions**服务器状态获得

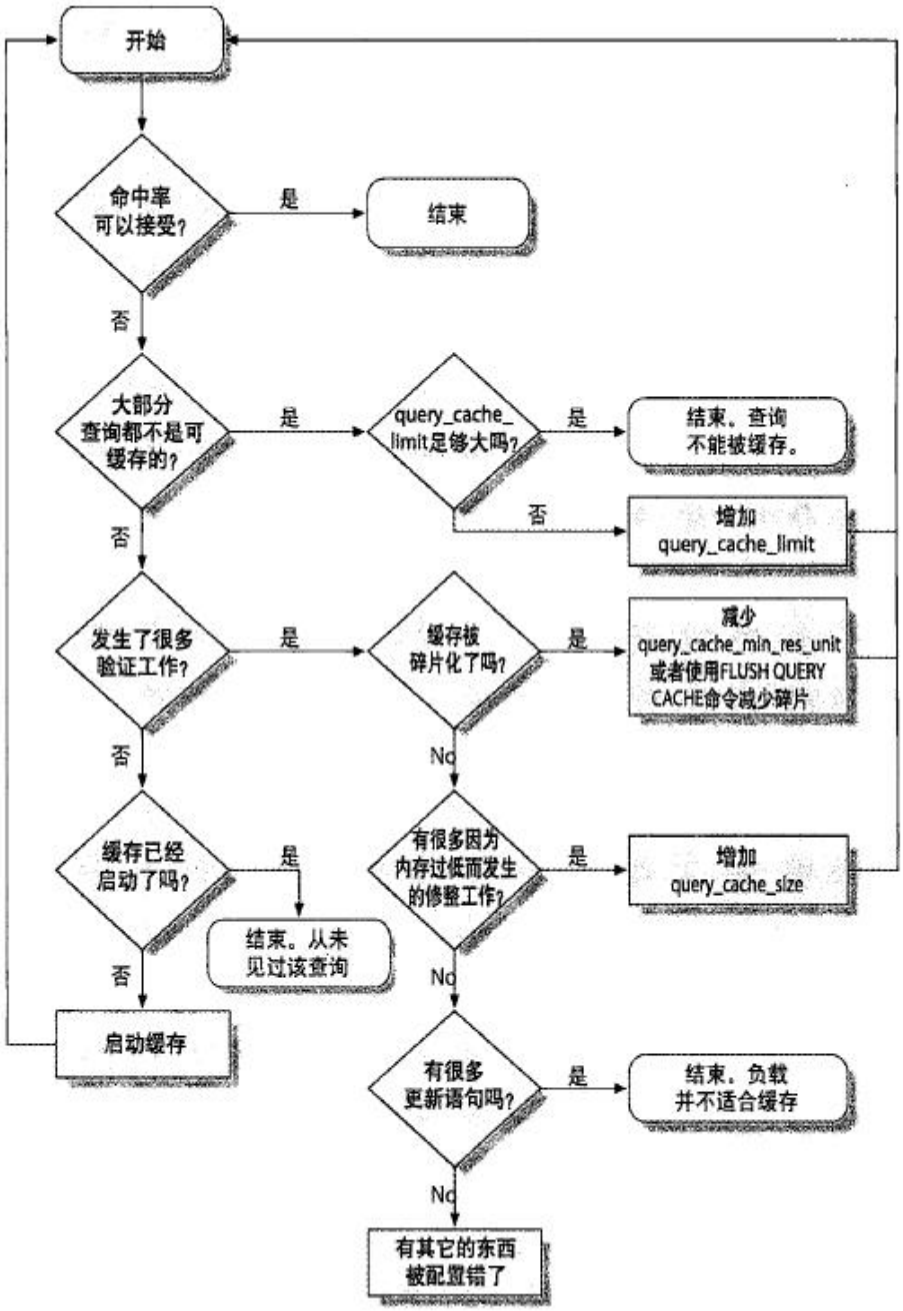
消极因素：结果集太大或语句不符合要求(**qcache_not_cached**记录)，第一次查询，查询被移除或失效(过期或内存的原因)，没有开启查询缓存，重复性查询太少，内存不足或数据频繁改变

8.1.4，缓存的优化(全局变量)

query_cache_type([1]ON/[0]OFF/[2]demand根据sql_cache)，**query_cache_size**(目前推荐为250MB)，query_cache_min_res_unit(最小的块大小)，**query_cache_limit**(可以缓存的最大结果集)，query_cache_wlock_invalidate(在MyISAM中是否禁止其他线程读取被写锁定的表的缓存数据)

8.1.5，减少碎片

需要平衡query_cache_min_res_unit的设置，**结果集的平均大小**=(query_cache_size(总大小)-qcache_free_memory(剩余大小))/qcache_queries_in_cache(已缓存的查询数量)；qcache_free_blocks为碎片数量，可以使用flush query cache(锁定所有对缓存的访问)整理碎片或**reset query cache**清除所有缓存(会导致qcache_lowmem_prunes增加)



8.1.6，InnoDB 中的查询缓存(190 页)

4.1中的事务完全禁止使用缓存，之后的版本中InnoDB会对所操作的每一个表向服务器询问一个特定的事务是否可以读写查询缓存

每个表在InnoDB的数据字典中都有一个对应的事务ID；*#?是引用该表的事务ID的最大值还是所有事务ID的最大值?*

有两种情况会导致锁定查询缓存不可用：1，特定的事务发现它所引用的表在I

nnoDB数据字典中记录的事务ID计数器大于自己的ID；2，特定的事务对表进行了for update或修改数据的操作，在它提交之前其他事务无法读取/写入特定表的缓存；*#?这两点似乎有点冲突?*

8.1.7，通用缓存优化方案

(1)，将一个大表拆分成多个小表，它可以减小特定数据集缓存失效的机率

(2)，批量操作(更新/删除/插入)，比如insert into xx values(...),(...)或insert into xx select * from t1减少失效的次数，另外一个优点是还可以减少索引更新的次数

(3)，失效大型缓存时会导致服务器挂起，建议不要设置query_cache_size超过250MB

(4)，通过**query_cache_type**关闭缓存的时候同时把**query_cache_size**清0，因为它不会自动释放已经缓存的对象

8.2，在 MySQL 中存储代码

存储过程与存储函数只能在定义它们时的当前库中使用，并且它们均是通过 **SQL SECURITY** 设定执行时使用创建者/调用者的权限；它们只能使用局部变量(不带@)而不能使用用户变量(带@)做为参数，但是@可以在其内部直接被访问

8.2.1，存储过程

主要优点：减少发布开销并且便于维护，减小客户端发送的数据量，可以访问存储过程授权访问特定的表，缓存了执行计划(*#?不能在多个连接中共享是否属实?*)，集中商业逻辑并保证业务的完整性；主要缺点：执行计划的缓存是基于单个链接的(可以使用持久连接)，优化器不能估计存储过程的开销，目前还不能使用**deterministic**(给定的参数下返回值是固定的)优化重复调用

主要特性：不能在里面使用use变换数据库(有一个隐含use打开过程所属的

库)，可以返回多个结果集(需要客户端支持)，不能使用load data infile

主要语法:

(1)，**declare** 错误名 **condition** for [sqlstate err_code|mysql_err_code]，declare [continue|exit|undo(undo没有被实现)] **handler** for [SQLstate err_code|mysql_err_code...] begin sql_smt end；声明错误与错误处理(声明可以合并到错误处理中)，**if then** ...elseif then...else...endif，**case when** ..then ...else...end case，case ...when...then...else...end case，**loop**...end loop(跳出语句是leave支持repeat/begin/while/loop)，**iterate**再次循环(支持while/repeat/loop等同于PHP中的continue)，**while** ...do ...end while

Declare必须定义在存储过程的所有语句之前；自定义变量与condition不分先后但必须定义在cursor之前，cursor必须定义在handler之前

(2)，declare cur **cursor** for sql_smt(没有引号)，open/close cur(打开/关闭光标)，repeat 循环体 until 条件变量(为真终止，定义错误02000检测终止并设置变量为真) end repeat，**fetch** cur **into** v1，v2，...(从光标中解析数据)

8.2.2，存储函数

create function(**只能是in参数**) returns 返回值的类型以及长度 begin ...end，函数体必须有返回值且不能直接输出结果；show [function|procedure] status where db='数据库名'可以查看定义在特定库上的函数或存储过程

8.2.3，触发器

触发器(**create trigger** 触发器名字 触发器时机[before|after] 事件 **on** 表名 for each row begin ...end)；MySQL只支持行级触发器，即总是针对结果集的每一行操作而不是整体，遍历第一行的时候**row_count**函数1；在一张表上最多可以定义同样的触

发器两个(before/after)但是触发器名在数据库中必须唯一，old对象是只写的，new是读写的(在前置触发器中new.自增ID始终为1)；在触发器中禁止向客户端返回数据；**show trigers**查看当前库中的触发器

触发器的问题主要是它的"透明性"增加了排查问题的难度，Before->run->after是一个整体的事务操作，中间任意一部的错误都会导致流程终止，但是如果使用了非事务引擎则不能被回滚

触发器可以被使用合并算法(Merge)的视图上的操作间接触发

8.2.4，事件

MySQL全局变量**event_scheduler**设定是否开启事件(on/off)

8.2.5，高级注释

/*!**版本号**[换行]代码或注释[换行]*/，大于特定的版本才执行其中的内容

8.2.6，游标

MySQL游标的结果保存在临时表中，操作是next指针；尽量不要让结果集太大以致超过**tmp_table_size**，否则会变成在磁盘上创建的临时表

8.2.7，准备语句

准备语句(**prepare**:生命周期为单个链接)；在服务器上保存了部分执行计划，在多次查询的情况下可以减少网络发送的数据量，参数以二进制的形式发送而不是ascii形式发送；通过**deallocate prepare** smt销毁准备语句，不销毁可能导致内存泄漏；缺点为不能使用5.0之前的查询缓存，对于只使用一次的准备语句开销不划算，不能在存储函数以及触发器中使用，不能在标识符的位置使用?替代符(比如表名，数据库名以及系统关键字)

在实际测试中发现如果在存储过程中不能直接Execute smt using 传入的参数，必须使用set @v=传入的参数，然后using @v

Prepare声明的语句必须是一个字面量或变量且不能由函数直接返回，可以采用这样的方式**拼接SQL**即：set @sql=concat('create table ', @tablename, '(x int)'), prepare smt from @sql的方式

8.2.8，视图

MySQL针对视图的查询有两种算法(**Algorithm**)：合并算法(**Merge**)与临时表算法(**TEMPTABLE**)，如果指定了无法被使用算法则产生一个warnings且视图类型自动变为**UNDEFINED**，对于UNDEFINED类型的视图在实际操作时MySQL会自动选择合适的算法

合并算法(支持CRUD以及索引)中的行于原始表中的行具有一一对应的关系：创建视图create view t_view as select * from t1 where x>500，执行查询select * from t_view where x<1000，重建后为合并查询select * from t1 where x>500 and x<1000

临时表算法能更快的释放表锁但不支持CRUD操作，也意味着他不支持索引以及触发器；它的数据与原始行不能保持一一对应的关系(group/distinct/union/count等原因导致)：创建视图create view t_view as select distinct * from t1 where x>500，执行查询select * from t_view where x<1000，重建后的查询为select * from t_view(**TEMPTABLE**) where x<1000

MySQL并不支持物化视图以及索引视图，物化视图把数据存储在一个看不见的表中，然后周期性的从原始表更新数据；物化视图和索引视图可以通过创建临时表模拟；索引视图是建立在物化视图的基础上，它具有自己独立的索引，更新基础表的同时也会更新视图中的索引

8.3，MySQL 字符集

字符集表明了二进制位到字符之前的对应关系(用来存储)，而校对规则是字符的比较方法(给定的字符集至少有一套校对规则)，只有基于字符的值才有字符集(列值，字面量，表达式的结果，用户变量，不包含数值型)； 可以通过**show character set**查看服务器支持的字符集，**show collation**(支持where过滤)查看支持的校对规则(_ci不区分大小写/_cs区分大小写/_bin二元简单的比较二进制码).

MySQL的字符集有两类: **创建对象**时的默认设置(服务器->数据库->表->列依次继承)与**连接时**(服务器与客户端沟通)的设置

创建对象时的设置:

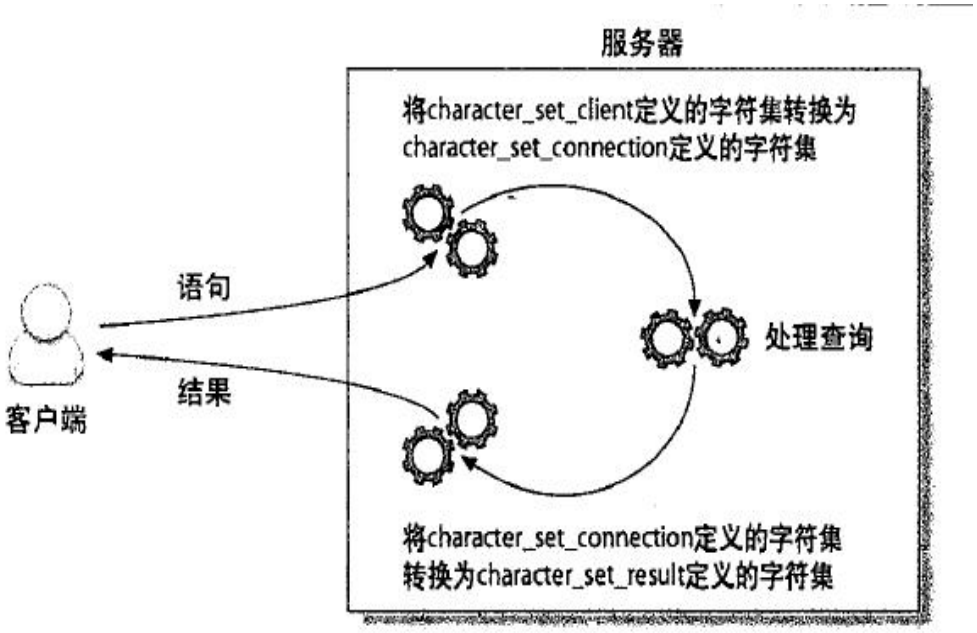
(1)，服务器; **default-character-set**与**default-collation**在5.5中已被**character-set-server**与**collation-server**代替(会被创建的数据库继承)

(2)，**character_set_database**/**collation_database**分别用来指示当前数据库的字符集与校验规则(更改数据库时自动变化)

(3)，使用方法; **character set** 字符集 **collate** 校验规则(数据库，表，列)

连接时的设置:

(1)，**character_set_client**(客户端发送查询的字符集); **character_set_connection**/**collation_connection**表示收到的数据转需要换成的字符集与校验规则，存储时需要转换为实际列存储需要的字符集，发送的时候反向这个过程，需要注意的是不同编码转换可能丢失数据造成乱码; _字符集"字符串" collate 校验方式 这样的字符串不转换); **character_set_results**(返回结果集时使用的字符集)，**set names** xxx等价于同时设置@@character_set_client,@@character_set_results=xxx,@@character_set_connection=xxx;



产生乱码一般有几个原因：客户端所采用的编码与character_set_results不一致

(2)，改变校对规则(引介词)；变量：变量名 collate[binary速记符] 字符集；常量：_utf8"值" collate utf8_general_ci(简单字符串默认使用@@character_set_connection/@@collation_connection)；它仅仅是一个申明符号，不改变字符串的值；

charset(xxx)函数返回参数的字符集，collation函数返回参数的校验规则；在有多个不同字符集的多项式中，确定要使用的校验规则可以通过coercibility函数返回一个表示特定子项可压缩性的值，在多项式中使用压缩性最低的子项的校验规则(如两个项的校对规则不同但可压缩性一致则发生错误)．同时可以使用convert(value using charset)函数转换字符集，对于select..outfile，如果不使用convert就无法设置数据数据的字符集

对于传入参数然后返回值的简单函数(lower/upper)返回值与输入值字符集一样，对于合并了多个不同字符集的参数并返回单个值的函数，返回值的字符集请参照"可压缩性"

改变参数的校验规则可能会导致索引失效(不同校验规则的列做jion操作的时候也有这个问题)，因为索引本身是基于校对规则排序的；不一致的校对规则也存在翻译成本

不同的字符集对于存储空间具有不同的要求，如果不需要在数据库中处理字符集(数据库只是简单的存储)，那么可以使用binary存储数据并用另一个列表明它的字符集

#?为什么数据库所有的地方设置为utf8，如果列为utf8在命令行需要且必须设置@@character_set_client为gb2312，但是列为gb2312或latin1时@@character_set_client必须与列字符集一样才能正确插入数据?

#?character_set_connection/collation_connection与列的字符集以及校对规则的关系是什么?

8.4，全文搜索(索引)

目前仅MyISAM支持(char/varchar/text)；全文索引是一个B树结构，第一层是关键字，然后第二层为一个由包含特定关键字的文档的指针锁构成的列表；停用词ft_stopword_file以及单词长度超过ft_min_word_len的词会被忽略，停用词也包括哪些虽未明确指定但是在超过一半的行中都存在的单词，MySQL会优化同一个查询内相同的match..against防止重复执行

Create table t1(bID int primary key, btitle varchar(255), bcontent text, **fulltext**(btitle, bcontent)) engine=MyISAM创建一个全文索引，**match**(column1, column2) **against**(keyword)如果用在select中则返回浮点数表示相关性(为0则表示不相关)，在where中则选择出符合条件的记录(会自动按相关性排序，因为全文索引与普通索引互斥，所以额外的order可能会导致文件排序)，如果要提高某个列在全文搜索中的重要度，则需要建立额外的索引；全文索引如果包含多个列必须作为一个整体使用

布尔搜索(比较慢)；match(column) against('key1 key2' in bollean mode); key含有它的行排名较高，~key含有它的行排名较低，+key行必须含有它，-key行不能含有它，key*以key打头单词的行排名较高

全文搜索的局限与变通方式；全文搜索的排名与关键字出现的频率有关，与位置无关；修改有一百个单子的文本引起的索引操作是100次而不是一次(引发大量碎片)；大数据量文本引发的性能问题；全文搜索不能使用索引覆盖(它的工作方式导致的)；在where表达式中，全文搜索是有限执行的，其次才是其他所以，比如match(column) against(key) and column，即使column上有普通索引可用，它也会先使用全文索引；group会导致严重的性能问题，因为会有大量的匹配数据；*#?为什么实际测试的时候不能检索出值?*

8.5，外键约束(目前仅 InnoDB 支持)

(1)，需要的条件，被引用的列必须有索引(值可以重复)，引用列的索引是自动创建的，引用列的索引不能被drop index删除；引用列与被引用列的定义应完全一致；从某种程度来说，外键也可以保持和事务一样的一致性

(2)，**constraint** 指定子表索引名 **foreign key**(引用列) **references** 父表(被引用列) [on delete ...][on update [restrict(禁止删除或更新父表)|cascade(自动删除或更新子表)|set NULL(父表删除或更新设置子表列为NULL)|no action(等同于restrict)]]，外键级联引发的操作不会使用触发器

8.6，分区表

分区表(所有引擎支持)特性通过**have_partitioning**变量开启，它根据分区函数确定数据行与分区的映射关系

重要的特点：可以快速定位数据所在的分区从而防止冗余检查与访问以尽可能防止全表扫描的情况发生，删除数据速度快(**drop partition**)，分区数据可以分布到多个硬盘(**data/index directory**)；对于sum/count这样的聚集函数，查询可以在多个分区**并行执行**最后汇总；如果存在唯一索引则**分区键必须是唯一索引的一部分**，如果有多个唯一索引，那么分区间必须取自它们的并集

分区的删除方式为**alter table**...**drop partition**...**add partition**分区名(分区/子分区/删除/合并分区等)，describe **partitions** select****可以用来查看查询访问了哪些分区，如果在where操作中对分区的列做了额外的操作则不能使用到分区的特性，与不能使用索引覆盖的情况类似

(1)，**range分区**(按数值所在的区间确定分区)；create table t1(ID int primary key auto_increment.....) partition by range(ID)(partition p0 values less than(6小于6的)，partition p1 values less than(11).....)，对于没有明确定义的最大值，可以使用一个额外分区 less than maxvalue；也可以只取列的一部分

(2)，**list分区**；partition by list(ID)(partition p0 values in (1,3,5)，partition p1 values in(2,4,6))，插入未在列表中出现的ID会发生错误

(3)，**Hash分区**；partition by Hash(列) partitions 分区数量(希望数据平均分布在分区中)；大量数据插入或更新的时候Hash函数有一定的性能开销

(4)，**Key分区**；差不多等同于Hash分区，但是它不能使用用户自定义的表达式

另外还可以在分区下再创建子分区

8.7，分布式事务 XA(InnoDB 支持)

分布式事务分为内部XA与外部**XA**，内部XA跨存储引擎，外部XA跨服务器，XA事务与普通事务(start transaction)是完全互斥的，比如显式回滚或提交

XA事务需要一个协调员(全局事务管理器**TM**)，第一阶段它会通知所有的参与者**RM**(资源管理器负责局部事务的回滚与提交等操作)准备提交事务，第二阶段当它收到所有参与者均就绪的信号时，才发送消息让所有的参与者真正提交数据

Xa start 唯一标识符(是一个字符串)----------进入active状态

标准SQL操作

Xa end 唯一标识符----------进入idle状态(空闲)

Xa prepare 唯一标识符----------进入parpared状态(准备)

Xa commit|rollback 唯一标识符----------提交事务

Xa recover 则返回了处于parpared状态的事务的信息

因为要求客户端能够理解XA的含义，所以目前只有connector/j5.0.0以上的版本直接支持了分布式事务

9，优化服务器设置

摘要：MySQL的默认配置只适用于简单的运行于查询操作，通过修改配置文件大约可以获得最多三倍的性能提升，其后的改动性能提升就不那么明显；如果需要更高的性能则需要检查整个应用程序以及服务器体系的架构与分布是否合理；MySQL实际使用的配置文件可参考**MySQLD --help**(搜索default options)

9.1，变量的作用域与动态性

变量分为**系统变量**与**用户变量**，系统变量分为**服务器级**与**会话级**，会话级变量继承自服务器级变量并且作用域为当前会话；对服务器级变量的动态改变(动态变量)其作用域等于当前服务器实例的生存期的长度并且这些调整针对已有的连接无效；为会话级变量赋值为**default**可恢复其默认值(对应的全局变量值)，但是对服务器变量要慎用(有可能不是服务器启动时的值)

用户变量有两种，session级(一个@)作用与当前会话的生存周期，它可以直接在存储过程与存储函数内访问而不必作为参数传入；局部变量由**declare**定义且(不带@)作用域为**begin**...**end**，同时只能定义在begin...end之内

9.2，配置调优

调优应该先是基于查询层面的(比如建立索引/分表等)，然后才进行配置文件的调

整，对于每一个调整都必须进行严格的基准测试；数据库的优化不是一个一劳永逸的过程，而是根据库容以及硬件等条件的变化而不断循环的

9.2.1，内存使用优化

MySQL锁能使用的内存一般取决与操作系统以及其API的限制；32位linux的限制为单进程使用内存的上限为2.5-2.7GB(超过此值会导致MySQL崩溃)；因为系统API一次不能分配2GB以上的内存，所以就不能为**innodb_buffer_pool_size**分配大于此值的内存，但64位系统仍然具有一个相对宽松的限制

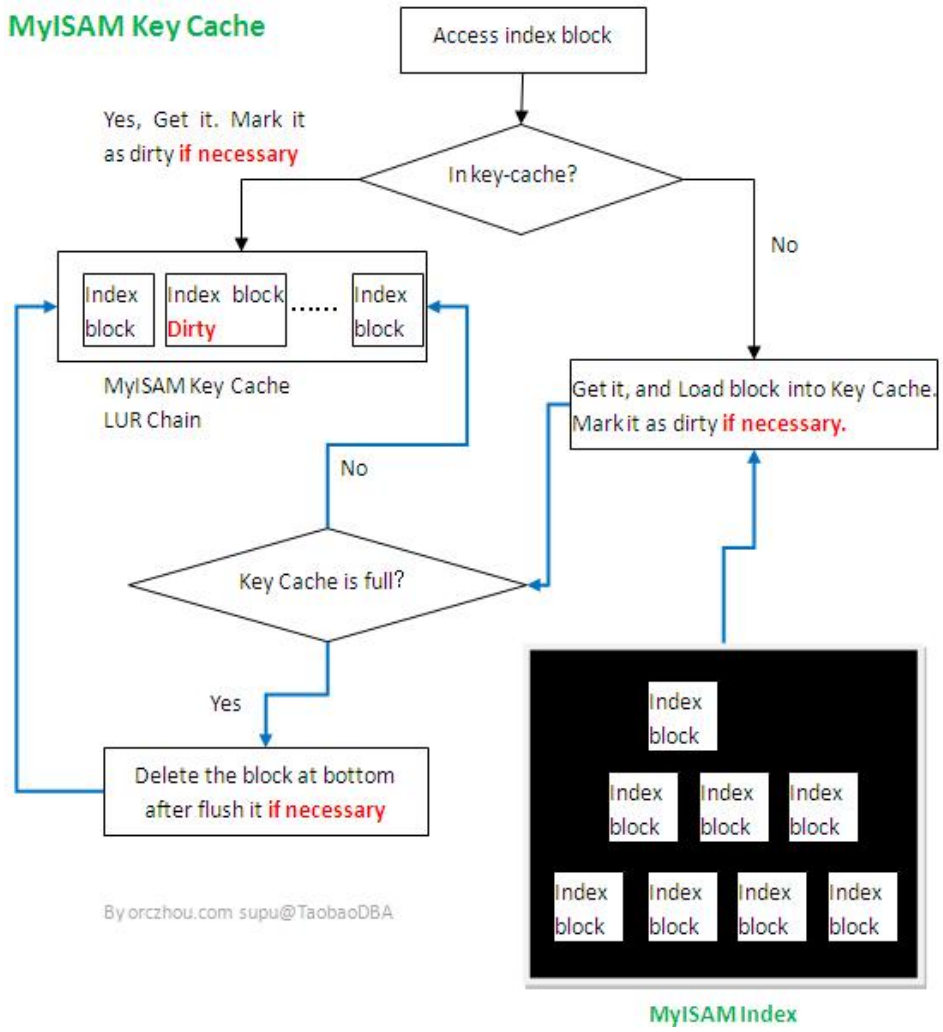
(1)，连接使用的内存；MySQL连接在单纯的开启状态下只用了很小的内存(排序/临时表等需要较大的内存)；可以通过**thread_cache_szie**缓存一定数量的线程(新连接可以复用这些线程)，可以参照**threads_created**状态的值进行设置；

(2)，操作系统保留内存；在内存使用超过一定的阀值时操作系统会创建并使用磁盘虚拟内存(视特定的OS)

(3)，为缓存分配内存；主要有几个方面：MyISAM数据缓存(只能使用OS原生缓存)以及键缓存，InnoDB缓存池，查询缓存，排序缓存，Memory引擎需要的内存等；如果服务器只使用MyISAM引擎则可以通过**skip-innodb**关闭它

#?对于这些缓存的内存开销是否受到OS对进程使用内存大小的限制?

(4)，MyISAM缓存



MyISAM只缓存索引而数据交由OS缓存，即使没有MyISAM表也应为服务器自身分配约32MB的空间，因为系统表使用了该存储引擎

通过**key_buffer_size**设置缓存更多的索引数据，与之相对应的是**key_read_requests**/**key_reads**分别代表了从缓存以及从磁盘读取的数据块的数量，**key_write_requests**/**key_writes**则分别表示写入到键缓存以及磁盘的次数，如果写入到磁盘的次数过于频繁则应考虑增大key_buffer_size的设置；**key_blocks_unused**/**key_blocks_used**则分别代表了键缓冲区内未使用与已使用的块数量

键数据块的大小通常为树节点的大小(取决于与关键字大小)，如**key_cache_block_size**设置不当可能会导致**写入排队**，如操作系统页(分配单元大小)大小为4KB，而键数据块设置为1KB，那么：MySQL请求1KB的数据->OS读取全部4KB数据返回1KB->删除缓

存->MySQL要求OS回写1KB磁盘->OS读取4KB到内存中然后修改其中1KB然后写入磁盘，如键数据块与页大小相同则此布操作可以避免，因为簇是文件系统读写存储的最小单位；

#?键数据块是不是一个类似与簇那样的最小操作单位的概念？配置成簇大小的整数倍是否可以提高效率？读取时4KB数据已经在内存中，为什么会写的时候还要再读一次?

多个命名的键缓冲区使用**set global** key_buffer_1.**key_buffer_size**=xxx创建，使用**cache index** 表1... **in** key_buffer_1保存索引到特定的缓冲区然后**load index into cache** 表1...预加载索引到缓存，也可以通过**init_file**=*.SQL自动在服务器启动时处理(每个命令一行且不能包含注释)；使用多个键缓冲区是因为当缓冲区被更新时会阻塞其他线程，因此要按表读取/更新频率设置不同的缓冲区

#?delay_key_write与key_buffer_size是什么关系？如果delay_key_write设置为off，那么对与key_buffer_size有什么影响?

*#.delay_key_write为on表示在数据**发生变动**时MyISAM只将数据写入磁盘(也可以作为创建MyISAM表是的参数)，索引则在表关闭时写入；key_buffer_size是控制如何加快对**索引的访问**的，但是如果缓冲区满并且有新的索引块需要从磁盘读取到缓冲区则按照本节开头的图例所列的方式处理*

9.2.2，InnoDB 缓冲池

可以通过**innodb_buffer_pool_size**设置缓冲池的大小，它保存了索引以及行数据，自适应Hash索引(**innodb_adaptive_hash_index**)，锁以及其他内部结构，同时也保存了插入缓冲区(合并更多写入然后顺序执行)；对于InnoDB的暖机不同于**MyISAM**的**load index**...**in**...或**cache index into cache**...而是使用全表或全索引扫描加载索引与数据

官方建议将60-80%的物理内存分配给InnoDB缓冲池，但是这80%之外InnoDB还需要8%的正常维护开销以及自适应Hash索引开启但没有正常关闭服务器而重新启动时12%的恢复开销，建议不要超过50GB，因为在执行检查点或插入缓存合并的时候会变慢且并发会因为锁定而减少

#?为什么会导致插入缓存合并的时候变慢，不是可以通过innodb_max_dirty_pages_pct设置允许的脏页比例么？为什么设置过大会导致大量的锁定?

innodb_max_dirty_pages_pct(0-100)设置脏页的最大百分比(推荐75-80)，如脏页超过这个比例则有一个后台线程周期性的线程将数据**刷写到磁盘**；设置较大的值会导致启动与关闭时间变长，另外修改它的值不会立即发生效果，其他资料见<import:"InnoDB页结构浅析.doc">

9.2.3，线程缓存

系统最多可以缓存**thread_cache_size**(每线程大概占用256KB)个线程；当新连接到达时如果**threads_cached**中有空闲的线程则移出一条给新连接使用同时threads_cached的值减一，连接关闭后线程被回收到缓存中直到下一个新链接到来为止(缓存空间不够时直接销毁)；**threads_connected**为当前正在被连接使用的线程数(受max_connections的限制)，**threads_created**是因为缓存中没有可使用的线程而新创建的线程数(如果有空闲线程则直接使用而不是重新创建)，**threads_cached**可以查看缓存中空闲的线程数量

9.2.4，表缓存(MyISAM)

表缓存保存了文件的句柄而不用重复打开表，它分为两个部分：打开表的缓存(**table_open_cache**)与表定义的缓存(**table_definition_cache**)；有几个重要的参数：**opened_tables**已打开表的数量，**open_files_limit**为OS允许MySQL实例打开文件的实际数量，如被耗尽服务器将会拒绝连接；具体的设置应为**max_connections***N，N为查询中一

条SQL可能使用的表数量的最大值，另外还需要为临时表与文件保留一些额外的句柄

表缓存满而又有一个新线程需要打开一个不在缓存中的表时(LRU算法)，实际表句柄缓存数量大于table_open_cache时(*#?因为LRU的存在难道会缓存超过table_open_cache数量的表?*)，**flush tables**时会从缓存中移除表句柄即关闭所有打开的表；需要注意的是对于在同一个查询中被引用两次的表也要使用两个句柄，初始的时候有三个句柄：一个索引句柄(可共享)与两个数据句柄

关于句柄的详细定义参见<url:"http://baike.baidu.com/view/194921.htm">

9.2.5，InnoDB 字典

InnoDB有自己的**表缓存**，当表打开的时候会向字典添加一个约4KB(也可能更大)的且表关闭时不会被删除与释放的对象(10万个表或分区大概占用1GB内存)，它为表的打开与操作统计数据；与MyISAM相比InnoDB不会一直在表中保存统计数据而是服务器启动的时候重新计算

#?计算的是什么数据？Innodb表是如何被关闭的?

#.flush tables会关闭表，对于MyISAM则为从table_open_cache中移除一个条目

如果使用**innodb_file_per_table**为表指定不同的空间，那么还受到**innodb_open_files**参数的限制

9.3，MySQL I/O 调优

9.3.1，摘要

一些参数控制了MySQL如何将数据同步到硬盘，通常它对性能有很大的影响；保证数据被立即写入磁盘的代价是很高的

9.3.2，MyISAM I/O 调优

delay_key_write(on/off)参数可以控制是否启用延迟索引写入，如开启则导致因数据改变而发生的索引改变在表关闭(flush tables)时方写入磁盘，否则在数据写入时立即对磁盘索引做出相应的改变;它有一些缺点：异常关闭导致索引损坏，大量写入的合并会导致关闭表的时间延长且未刷新的数据快不会给从磁盘读取的数据块腾出空间而导致查询停止

#?如果开启delay_key_write那么索引数据在表关闭前会一直保存在key_buffer_size中从而有可能超过键缓冲区的大小，这时候是怎么处理的？对key_buffer_size自身的LRU机制有何种影响？见<label:"9.2.1，内存使用优化">第4部分

对于因此而损坏的表可以通过**myisam_recover_options**变量设置恢复方式;可以通过**myisam_use_mmap**变量启用数据文件的**内存映射**直接访问OS的页面缓存而避免了代价较高的系统调用

9.3.3，InnoDB I/O 调优

(1)，事务日志(ib_logfile*)

当事务提交时先更新内存中的数据(产生脏页)，然后再更新事务日志(也是断电后恢复的依据)，如二进制日志开启则会先更新二进制日志再提交事务；因为大量的事务提交会导致大量随机的I/O所以使用事务日志变随机I/O为顺序I/O；事务日志的大小是固定的，后台线程会从顶部循环到底部合并操作然后写入到特定的表

#?innodb_max_dirty_pages_pct表示脏页达到一定的比例才刷新到磁盘，这里是指刷新到事务日志还是真正的表空间?

日志文件的大小与数量由**innodb_log_file_size**与**innodb_log_files_in_group**控制，如日志文件太小会导致事务更新因为等待空闲空间而导致大量的等待状态或频

繁的检查点；如果日志文件太大则有可能导致关闭时间以及意外关闭时的恢复时间变长；检查点可以由多个因素触发：日志文件满，flush表，InnoDB内部定时器；通常情况下的优化之需要调整日志文件的大小与日志缓冲区的大小，通常**innodb_log_files_in_group**保持为2(因为正在被后台线程处理的日志文件不能被写入)

innodb_log_buffer_size可以用来设置日志缓冲区大小，如果没有text/blob等大数据官方建议1-8M；**innodb_flush_log_at_trx_commit**控制日志缓冲如何刷新以及刷新频率，有这样几个值：0表示每秒钟刷新一次缓冲区中的数据到日志文件以及刷新日志文件中的数据到磁盘但忽略事务提交动作(可能丢失事务)，1(**建议**)每次提交事务时刷新缓冲区数据到日志文件并刷新日志文件到磁盘(不会遗失任何事务但有可能OS假冒刷新)，2(**速度较快**)每次提交事务时刷新缓冲区数据到日志文件但不刷新磁盘，每秒钟也会刷新一次日志文件数据到磁盘，因为线程调度问题也可能超过1秒或更长的时间；需要注意的是因为OS以及磁盘有自己的缓存机制(建议关闭磁盘缓冲区)；**innodb_flush_method**可以设置InnoDB与文件系统的交互方式

#?这里的刷新特性来看，一个在线运行的服务器即使事务量很小，也会有很高的频率需要刷新日志文件中的数据到磁盘？请问调整日志文件大小还有什么意义？innodb_max_dirty_pages_pct的设置还有什么意义？如果掉电，日志缓冲区中的内容会不会丢失?

#?数据是从事务日志刷新到磁盘还是从内存直接刷新到磁盘？从innodb_max_dirty_pages_pct的角度看，数据应该是从内存直接刷新到磁盘，但是为什么说事务可以合并操作变随机I/O为顺序I/O?

9.3.4，InnoDB 表空间

共享表空间实际上是一个通过**innodb_data_file_path**设置的跨越了一个或

多个表空间文件的虚拟文件系统；如果配置使用了多个表空间，那么会在一个写满后再使用下一个表空间；如果开启了**innodb_file_per_table**(ON)则每个InnoDB表使用独立的文件，但仍然需要为其他撤销日志与系统数据定义共享表空间(可以关闭自动延伸)

InnoDB也支持将数据存放到未格式化的原始分区以提高访问效率，缺点是不能使用系统命令访问这些文件；*#?这点如何做到?*

旧数据与表空间：大量未提交的数据会导致很多老的数据在表空间中无法被清理而直接耗尽表空间，因为未提交的事务还需要这些数据且清理进程是单线程的而清理速度跟不上；*#?未提交的数据不是存在与InnoDB缓冲池与事务日志里面的吗?*

Show engine InnoDB status;

Trx ID counter 11B601

Purge done for trx's n:o < 11B48F undo n:o < 0

描述了总事务量与已清理的事务，如果差值太大的话，建议通过一些手段对MySQL减速(比如限制连接数)，也可以**innodb_max_purge_lag**(未清理的事务量)为大于0的数字，超过这个值自动延迟更新事务

双写缓冲是为了防止页面因为意外而被部分写入，在共享表空间分配了一个可存储100个页面的空间，当**页面缓冲池**清写到磁盘时会先写入双写缓冲区，然后再写入到真正的地方；在某些文件系统如**ZFS**已经做了这样的工作则可通过**innodb_doublewrite**关闭此特性；*#?改动不是先写入事务日志再写入磁盘的吗？这里为什么又从缓冲池直接写到磁盘了?*

9.3.5，其他 I/O 调优

sync_binlog为0则由OS控制何时刷新二进制日志到磁盘；如果大于0，那么就表示在刷新到磁盘前可以有多少写入(N个事务)，为0或越大的数字能见效开销但是可能

在发生意外时造成大量事务丢失；**expire_logs_days**可自动删除老的二进制日志

9.4，并发调优

9.4.1，MyISAM 并发调优

对于并发插入可以参考章节<label:"4.2，MyISAM引擎(非事务)">的第一部分，也可以开启**delay_key_write**索引延迟写入(合并后写入)，见<label:"9.3.2，MyISAM I/O调优">

9.4.2，InnoDB 并发调优

InnoDB使用自己的线程调度来控制对内核的访问，**innodb_thread_concurrency**用来确定这个数量，为0则不检查；如果数量满则通过**innodb_thread_sleep_delay**确定线程在这之后(微秒)重试，如果还是不能进入则排入等待线程的队列并将控制权交给操作系统；*#?与最大连接数的设置有什么关系？进入InnoDB核心后能干什么？*

9.5，基于工作的负载调优

首先要熟悉服务器的基本情况以及运行的业务，可以通过innotop工具监控服务器性能指标

9.5.1，优化 text/blob 负载

针对text/blog的优化，可以通过substring将其转换成varchar以减小其大小从而减少临时表的生成，如果不能避免，可以改变tmpdir降临时表生成到高速设备(比如内存虚拟盘--tmpfs)；在InnoDB引擎上可以考虑增加缓冲区的大小，在InnoDB上如果text/blob超过**768字节**则会单独使用一个页面来存储它

9.5.2，检测服务器状态变量

主要是一些表达负面的参数，比如创建了多少临时表，意外中断的连接数等

9.6，每连接设置调优

如果不能确认，不要改动全局性的连接设置，否则有可能造成大量的资源浪费；基于单个连接的变量有**sort_buffer_size**，**tmp_table_size**等

10，操作系统与硬件优化

MySQL性能瓶颈通常有两种：CPU瓶颈与I/O瓶颈

10.1，CPU 的选择

早期MySQL架构对于多CPU或多核心的支持存在问题，单个查询/连接只能使用一个CPU(或核心)，但多个并行互不冲突的查询可以使用不同的CPU，所以应视情况选择CPU，之后的MySQL版本可能已改善对多CPU或核心的支持；*#?257页提到来自多个链接的查询可以并行使用多CPU，是否特定的链接由始至终只能使用一个CPU或一个核心?*

由于32位CPU寻址的限制，单个**key_buffer_size**的大小不能超过4GB(虽然可以变通设置多个键缓冲区)，但在64位OS与CPU上可以更好的使用超过4GB的内存

10.2，平衡内存与磁盘资源

存储介质的速度由快到慢分别为：CPU寄存器->CPU缓存->内存->磁盘，更快速的设备应该用来存储最热的那一部分数据，良好的数据库缓存设置是优于OS缓存机制的，因为它更能准确的把握业务特点

10.2.1，随机/顺序 I/O 与数据写入

主要差别在与索引的顺序与实际的key顺序是否一致，因而顺序读写更有效率；从而可知缓存需要随机I/O的数据更能大幅度提高I/O效率；同时MySQL也会进行I/O合并变同步的随机I/O为异步的顺序I/O

#?I/O合并的详细含义是什么?

10.2.2，工作集与缓存单元

工作集是指所有数据中被经常使用的那一部分，缓存单元是指存储引擎所能操作的最小数据单位；对于InnoDB来说，页面大小为16KB，那么即使只需要取得100字节的数据也会完成的读取两个数据页从而占用32KB的内存空间(数据页+索引页)，*#?InnoDB的索引不是与数据一起存放的吗？为什么还有索引页与数据页之分？写入100字节到磁盘的时候是否也需要整页写入？*

Falcon(同时支持行缓存与页缓存)的缓存单元则为行而不是页面，因此对于分布比较广泛的的随机访问具有很快的速度

10.2.3，为服务器选择存储硬件

硬盘有几个主要性能指标：容量，传输速度，访问时间(转速，寻道时间...)，多个硬盘可以分散I/O的压力，但是具体还受限于具体的存储引擎，比如MyISAM更新添加表锁就限制了并行更新但是读不受影响；另外可以使用RAID磁盘阵列提高I/O容量，如下：

级别	特点	冗余	需要的磁盘	较快读取	较快写入
RAID 0	便宜、快速、危险	否	N	是	是
RAID 1	快速读取、简单、安全	是	通常为 2	是	否
RAID 5	在安全、速度和开销之间进行了折中	是	N + 1	是	和其他因素相关
RAID 10	昂贵、快速、安全	是	2N	是	是
RAID 50	适用于极多的数据	是	2 (N+1)	是	是

除了RAID之外还有**SAN**(存储区域网络)与**NAS**(网络附加存储)等方案，但是速度均比RAID慢

通常来说使用多个磁盘卷可以提高I/O效率从而减少I/O等待，可以通过MySQL的一些配置将数据分散存储在不同的磁盘上，例如：InnoDB可以定义分区表，表空间不同部分的存储位置；MySQL可以定义临时文件(tmpdir)，日志文件的位置等

10.3，网络配置

反向域名解析(**skip_name_resolve**)是一个很耗时的过程，它可能导致大量的用

户处于**unauthenticated user**状态(IP转换为域名以确定帐户权限)

主要是因为MySQL收到客户端请求的时候只能获取其IP地址，为了更好的匹配MySQL.user表中host为域名的部分，需要进行IP到域名的转换，关闭它会导致MySQL.user中host使用域名的帐户无法获得适当的权限

#?修改服务器上的hosts文件是否可以缓解解析带来的负面影响？MySQL5.5.21为什么找不到host_cache_size?

另外需要注意的是网络的带宽与网络的长度(客户端与服务器之间)，在内部网络的合理延迟估算公式可以参考：同交换机≈0.1ms，同机房≈0.2ms，同城机房≈0.5~1ms，远距离≈(两地线路距离/100)ms；对于通常所说的千兆网卡来说满载流量为1024Mbit/8=128MByte(1B=8b)，这里需要注意Mb与MB的差别

10.4，选择操作系统

首先是稳定性，然后其次是系统性能以及可维护性；其次应匹配合适的版本，例如在64为系统上安装64位MySQL而不是它的32位版本；另外还要注意文件系统的选择

需要注意操作系统以及文件系统的一些独特的特性，将OS的功能尽量关闭至应用所需的最低限度

11，MySQL 中的复制

11.1，摘要

MySQL提供了两种复制方案：基于语句(**statement**)的复制与基于行(**row**)的复制(基于行的复制会占用更多的带宽)，另外还有一种混合模式**mixed**；主要解决三个问题：故障切换，测试MySQL升级以及负载平衡

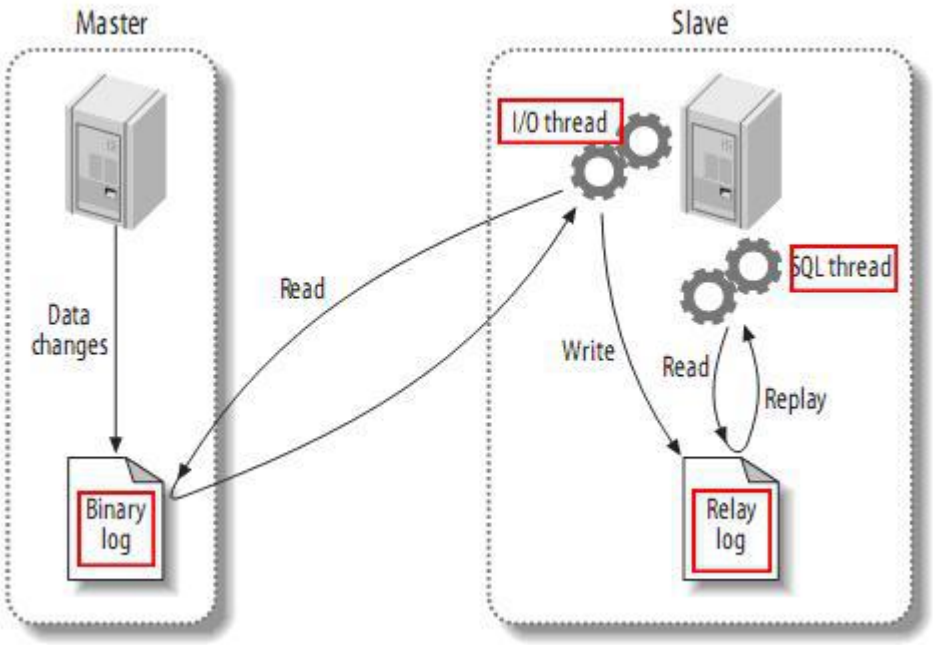
对于复制来说通常支持旧版本复制到新版本，而新版本复制到旧版本则可能因为旧版本服务器无法理解新版服务器的特性而失败

二进制事件包含了一系列事务组，每个事务组下有若干事件，因此一条SQL至少对应三个事件**begin**&**query**&**commit**

11.2，复制如何工作

(1)，主服务器写入数据更改到二进制日志；(2)，从服务器I/O slave thread复制主服务器二进制日志到自己的中继日志；(3)，从服务器SQL thread重放中继日志，应用更改到本地数据

在主服务器的事务被提交前会先串行(事务数据在二进制日志中不会交叉)更新二进制日志(二进制事件)再提交事务，然后从服务器开启一个**I/O slave thread**建立一个到主服务器的普通连接读取二进制事件并写入本地中继日志，同时从服务器会启动一个二进制转储线程(**SQL thread**)读取并重放中继日志中的事件，这两个thread在从服务器均为show processlist可见，**I/O slave thread**主服务器可见；**SQL thread**在重放完毕后会进入休眠直到其被新的二进制事件唤醒；如图:



需要注意的是原本在主服务器上可以并行更新(由不同客户端发起)的操作在从服务器依然是串行化(单线程)的，这是一个很重要的瓶颈；因为**SQL thread**通常能赶上**I/O s**

lave thread，中继日志一般存在与OS缓存中，所以只具有很小的开销

11.3，创建复制

有三个步骤：在主服务器建立复制帐号，配置主/从服务器，连接主服务器并开始工作

主服务器上用于复制的帐号必须具有**replication slave**(基本)的权限，除此之外不再需要其他权限；如果还需要监视复制则必须加入**replication client**(如show [master|slave] status)权限

11.3.1，主从服务器的设置

(1)，主服务器(**show master status**)

log-bin=MySQL-bin #开启二进制日志

server-id= #所有服务器中唯一的数字

binlog-do-db=db_name #仅当db_name为当前库时的SQL操作才计入二进制日志，如不指定表示记录所有数据库的操作

binlog_ignore_db=db_name #忽略的数据库

(2)，从服务器(**show slave status**)

log-bin=MySQL-bin #开启二进制日志

server-id= #所有服务器中唯一的数字

mater-host=

master-port=

master-user=

master-password=

master-connect-retry=

replicate-do-db=

对于新版MySQL 5.5来说从服务器的配置已由**change master to**命令代替(server-id仍然写在配置文件中)并生成**master.info**文件;

11.3.2, 启动复制(含非全新服务器)

可以使用**change master to**设置连接到主服务器的选项(旧版仍然设置my.ini), 也可以通过**show** [**slave**|**master**] **status**查看主从服务器的工作状态; 使用**start**/**stop slave**开始/停止从服务器

对于已经存在数据的服务器如果要做主从, 首先要保证主从服务器的数据完全一致, 然后配置主从的各项参数; 从服务器只重放主服务器二进制日志开启后的事件而不会自动复制之前的数据

出于保证主从服务器数据一致性的考虑, 需要开启**read_only**(on/off)临时禁止客户端更新服务器数据, 但是它对具有supper权限帐户以及replicate thread是无效的; **skip_networking**(on/off)则可以控制服务器只接受本地访问; 同时也要密切注意缓冲区中尚未刷新的数据导致的问题

11.3.3, 推荐的复制参数设置

sync_binlog表示发生特定次数的提交后刷新日志缓冲区中的数据到磁盘(仅适用于二进制日志), 为0则表示由系统控制, 但通常设置为1; 主要防止服务器崩溃后二进制日志损坏或丢失信息

relay_log_space_limit可以保证SQL thread线程如果落后I/O slave thread太多的情况下从服务器磁盘不被中继日志占满, 如果超过其设置的值则暂停读取二进制日志

11.4，高级复制以及原理

binlog_format可以设置MySQL复制的类型，有三个值：STATEMENT，ROW，MIXED(自动选择)

11.4.1，基于命令的复制

通过记录主服务器上执行的改变数据的SQL来完成该工作；优点在于一个占用带宽小(被操作的对象往往比操作它的语句大的多)且日志文件也小的多；缺点在于有些命令无法被正确复制(如current_user/uuid/now/connection_id等不确定因素)且可能存在部分SQL重新执行的代价很高，而且会在从服务器重复执行SQL的解析执行工作；可以使用**mysqlbinlog**工具查看其内容(对二进制日志与中继日志均有效)

11.4.2，基于行的复制

二进制中存储了被更改的数据，日志文件较基于命令的二进制日志文件大，占用带宽较大但它会从一些执行代价很高的SQL中获益；可以使用**mysqlbinlog**工具但较之基于语句的复制方式相比缺点是对旧版的兼容性较差

11.4.3，复制所依赖的文件

对于主从服务器的二进制与中继日志，均存在一个名为***.index**的文本文件，内容为*.000001，*.000002.....

在从服务器存在一个由change master生成且删除后不会被自动重新生成的名为**master.info**的文本文件，它包含了指示从服务器如何连接到主服务器以及I/Q slave thread已读取位置等信息

relay-log.info，记录了从服务器当前使用的中继日志以及其偏移量，如果删除则可能导致从服务器忘记当前位置而发生重复复制

为了防止磁盘被二进制日志文件占满，可以通过**expire_logs_days**设置主服

务器将超过特定天数的二进制日志删除. *#?中继日志是否只能手动删除?*

11.4.4, 转发复制事件

log_slave_updates在开启(on)的情况下从服务器会将自己收到的事件写入本地的二进制日志从而使从服务器可以成为其他服务器的主服务器; 因为server-id必须是全局唯一的, 所以**log_slave_updates**开启不会发生无限循环, 在环形或其他结构中服务器会忽略I/O slave thread读取到的且原始数据源为自己的事件

11.4.5, 复制过滤器

过滤器可以存在于两个位置: 主服务器写入二进制日志时以从服务器重放中继日志时; 在二进制日志上主要是**binlog_do_db**/**binlog_ignore_db**; 适当的设置可以减少日志文件大小以及网络负载

binlog_do_db声明如**当前数据库**为其设置的值, 那么接下来所有能产生二进制事件(如create database/show...不产生二进制事件)的操作无论被操作的资源是否在当前库中均写入二进制日志, 这样有可能导致在主库对其他数据库的操作在slave重放时发生资源不存在的错误从而导致复制终止; 如果当前库不为**binlog_do_db**的设置, 那么对binlog_do_db所指定库的任何操作均不计入二进制日志

binlog_ignore_db同样具有上面所说的问题从而导致对它的操作有可能写入二进制日志, 因此还应该在从服务器做相应的过滤

对于从服务器的配置项在**SQL thread**重放时发生作用, 如下:

Replicate_Do_DB: 设定需要复制的数据库, 多个DB用逗号分隔

Replicate_Ignore_DB: 设定可以忽略的数据库.

Replicate_Do_Table: 设定需要复制的Table

Replicate_Ignore_Table: 设定可以忽略的Table

Replicate_Wild_Do_Table: 功能同Replicate_Do_Table，但可以带通配符来进行设置。

Replicate_Wild_Ignore_Table:功能同Replicate_Do_Table，功能同Replicate_Ignore_Table，可以带通配符

#?为什么说binlog_do_db与binlog_ignore_db的设置会使按时间点的复制变得不可能?

11.5，复制的拓扑结构

MySQL目前不支持一个从服务器拥有多个主服务器，它的拓扑结构可以在这个原则下任意构建

(1)，主动/被动模式下的M-M

主动模式为两台均可写的服务器构建而成的M-M架构；但存在更新问题，如两个服务器同时重放中继日志但值又不一样时(最终交换了值造成数据不同步)，可以通过**auto_increment_increment**(auto步长)与**auto_increment_offset**(auto的起点)来解决M-M结构插入时的ID冲突问题，也可以使用auto+server-id多主键或md5等方式，如果要规避更新问题则需要在应用层面限制特定ID的数据只能在特定的服务器被更新

被动模式中同一时刻只有一台服务器可写；有两种方法：第一种是通过在需要只读的服务器上设置**sql_log_bin**设为0从而禁止将自己的操作记录到二进制日志中；另外一种是杀掉主写服务器的SQL thread，这样可以保证主读服务器的操作对自己是暂时透明的

(2)，带从服务器的M-M结构；M1->S1，M2->S2，M1<->M2

(3)，环形结构，M1->M2->M3->M1，可能存在同时更新的问题

(4)，金字塔结构；利于负载的分布，缺点如环形一样一旦某个节点故障会影响多个服务器

(5)，分发结构

如果一个主服务器对应多个从服务器，那么每个从服务器均会单独请求并要求主服务器发送二进制事件而不会彼此共享资源，对于特别大的二进制事件来说会给主服务器造成严重的负担

采用M->C->(S1，S2...)或类似的金字塔结构可以有效处理这个问题，对于分发服务器C应采用**Blackhole**引擎来提高这一过程的效率，Blackhole引擎不记录任何数据；需要注意主服务器新建的非Blackhole引擎的表导致的负面影响

11.6，定制复制的解决方案

选择性复制：主服务器存储所有数据，各个从服务器只存储特定的部分，与水平分区类似

分离功能：针对在线事务处理**OLTP**与在线支付分析**OLAP**使用复制体系中不同的服务器

旧数据归档：主要是需要清理主服务器但从服务器不需要做相应的清理：可以暂时在主服务器设置**sql_log_bin**为0，也可以在被复制的库之外执行清理命令

定制全文搜索：可以为主从服务器上相同的表使用不同的存储引擎

日志服务器：二进制日志与具体的服务器无关，因此可以复制并任意重放病可以用于崩溃后的恢复；*#?如果只有中继日志，如何恢复?*

11.7，复制的管理与维护

11.7.1，监控复制

在主服务器使用**show [master|salve] status**查看主从服务器的基本信息(非supper用户需要replication client权限)，**show [master|binary] logs**则给出了二进制日志文件列表以及对应的大小；**purge master logs** [to '*.000002'| before '日期']

删除指定"日志或日期"之前的二进制日志；使用**show binlog events** [in '*.000002' from 位置]显示二进制日志中的事件；通过**show slave hosts**则可以在主服务器列出注册的从服务器清单

11.7.2，测量从服务器延迟

从服务器状态值**seconds_behind_master**表示SQL thread落后于I/O slave thread的秒数，如果主从服务器之间的网络状况良好，它也可以表示从服务器落后于主服务器的秒数；否则可以使用Maatkit工具套件进行相关的维护

11.7.3，改变主服务器

(1)，计划中的改变

停止主服务器的写入(**read_only**或**flush tables with read lock**)->等待从服务器SQL thread重放完毕->验证数据是否一致(maatkit工具包可以)->stop slave/设置master_host为空/reset slave.....

Reset master/slave：前者用于清除所有的二进制日志文件以及相应的索引文件并开启一个全新的二进制日志；后者则为删除从服务器的**master.info**以及所有中继日志相关文件，需要注意的是reset salve对于配置在my.*中的内容是无效的；reset master后需要重新在slave指定日志文件与复制位置

可以使用**master_pos_wait**函数确认从服务器是否到达主服务器给定的位置，它挂起特定的时间直至从服务器到达预定的位置，返回值为从开始挂起到挂起结束期间执行的事务数，如果为NULL则应检查从服务器的复制相关线程状态是否正常，为-1则表示超时

(2)，非计划的改变

主服务器发生崩溃时，应选择一个pos为最新的从服务器提升为主服务器(需l

og_slave_updates开启)，然后等待所有从服务器完成重放操作；其余部分参考本节第一部分

对于在主服务器上没有被发送到从服务器的二进制事件(binlog_format:statement)，可以使用**mysqlbinglog**工具处理；或者直接复制二进制日志到其他正常的服务器

#?M:pos:100->(s1:pos:100，s2:pos:55)，它们互相具有不一致的MySQL版本，如M彻底崩溃，升级s1为新主服务器并使s2成为它的从服务器，因为s1与s2版本不一致导致相同的二进制事件位置不一致，如果为s2确定s1的偏移量？见311页

11.8，复制故障的解决方案

11.8.1，MySQL 主从服务器故障或日志文件损坏

对于主服务器，如果磁盘没有备份电池则可以设置**sync_binlog**为1保证在发生意外时尽可能少的丢失事务，否则可能会发生从服务器找不到事件编号的情况(丢失的二进制事件无法被恢复)，同时对**innodb_flush_log_at_trx_commit**的不当设置也有可能造成这种情况

#?317页二进制日志的几种崩溃类型是怎么回事？1，数据改变且事件有效；2，数据改变但事件无效；3，数据已跳过且/或事件长度错误；4，一些偏移已经损坏或重写，或者偏移已经发生了改变且下一个时间位于错误的偏移

从服务器意外关闭时如果**master.info**或**relay-log.info**没有保存到磁盘，那么可能从之前已处理的POS开始从而导致它重新执行一些二进制事件；对于InnoDB表可以通过它在恢复时写入的错误日志获取正确的位置

主服务器二进制日志损坏或SQL在从服务器执行时发生故障会导致从服务器停止复制，这时可以通过在从服务器运行**set global sql_slave_skip_counter**=N多次跳

过N个(建议为1)事件组直到找到正确的位置为止(注意是事件组)，必须先停止复制执行skip操作后重新启动slave并使用show slave status检查是否正常

对于从服务器中继日志的损坏则可以重新指定**master_log_file**以及相应的**master_log_pos**

11.8.2，过大的复制延迟

可以配置MySQL从服务器减少磁盘操作(包括减少二进制日志所包含的内容)或改善网络环境来处理此问题

对于耗时较大的如汇总等操作，可以选用一台特殊的服务器(M<->M->(s1,s2,...))来进行计算，然后将结果写入到主服务器从而使其他从服务器避免了重复运算

因为串行化写入与重放的原因导致复制的效率很低，必要时可以使用导入/导出数据来代替复制，如不需要将导入操作记录到二进制日志最好临时禁用**sql_log_bin**以加快导入速度

对于SQL thread的优化可以使用第三方程序在它读取特定的中继日志之前将符合条件的SQL转换为select语句执行，如中继日志SQL update t1 set ** where ID=5可转化为select * t1 where ID=5提前载入数据到内存，但是不能超过当前重放点太多，否则可能因为数据太多造成缓存失效；但是它也有一些限制：MyISAM中select与update，delete的表锁，工作集如果小于或等于内存大小则无必要进行热身

#?对于SQL thread数据的预热具体如何实现，目前没有找到相关的介绍?

11.8.3，过大的包以及受限制的带宽

max_allowed_packet用于设置MySQL服务器允许接收的最大的数据包(sql语句)的长度，主从服务器不匹配可能会导致问题，对于网络方面的改善可以通过**slave_compressed_protocol**设置传输的信息使用zlib引擎进行压缩，但是这样会有一定的CPU开

销

数据包的大小为发送至服务器的单条SQL的大小或服务器返回客户端的单一行的大小；当发送的包大小超过max_allowed_packet时，服务器会发送"信息包过大"的错误并关闭连接，而可能造成客户端"丢失与MySQL服务器的连接"

11.8.4，磁盘空间限制

可以通过**expire_logs_days**以及**relay_log_space_limit**，前者删除过期的日志，后者则为当中继日志大小超过此值时I/O slave thread先停止工作等待SQL thread处理并删除部分中继日志

12，伸缩性与高可用性

性能：表示系统当前对请求的响应能力；**负载能力**：是指当在当前基础上访问突然增大到N后系统的处理水平仍然可以处在一个比较合理的位置的能力，N越大表示负载能力越好；**伸缩性**：系统是否可以通过增加服务器或其他资源提升系统的处理能力；**可用性**：非故障时间所占的比例；**容错性**：当一定数量的服务器故障时应具有继续提供原有服务一部分获全部功能的能力

12.1，MySQL 的可伸缩性

通常来说有三种方式：向上扩展(使用更加强大的服务器)；向外扩展(业务切割/数据分块/读写分离)；回缩(归档并清理老旧的数据)

12.1.1，向上扩展

在规划之前应该预估系统的处理峰值以及结合应用自身的特点进行规划，增加设备前应确定当前系统是否已优化到极致再考虑是否增加新的硬件或其他资源；升级硬件时应考虑MySQL自身以及OS的限制

12.1.2，向外扩展

对于向外扩展主要有按**读写分割**/**业务功能分割**/**数据分块**三类，这三类不是绝对界限分明的，可以考虑按具体情况混合使用它们；但是对于被分块的数据来说目前还没有一个透明且通用的抽象层可以很好的粘合它们，数据分块知识点如下:

(1)，选择分割键

可以根据一个或多个键的值来决定数据块的位置；分块的目标是最常用的查询应能尽可能的确定需要访问的块；对于多个键的组合分块如键(A，B)即可针对它们的组合分块，也可以分别为每个键分块(冗余)，这一切均取决于具体的查询条件

(2)，跨块的查询

通常并发的向各个节点发出请求然后合并返回值，也可以使用汇总表，对于不支持并发请求的语言可以采用第三方服务或控件处理；对于数据的一致性可以考虑通过XA分布式事务来保证(需要注意如果块分布在多个服务器那么外键将不可用)

应设计一个抽象层来透明化应用程序到数据源之间的通信，它应该包含**创建**/**合并数据块**&**数据迁移管理**&**事务管理**&**选择分块**&**并发查询**并合并等功能；目前可以使用**MySQL proxy**工具实现这样的功能

(3)，数据块与节点

通常没必要为一个节点只分配一个数据块，因为这样要么因为数据块过小造成节点资源浪费以及大量的查询需要跨分块，要么为了避免节点资源浪费而导致数据块过大从而难以进行alter/optimize等常规维护

(4)，固定分配/动态分配

固定分配中行的位置依赖于特定的分配函数(Hash/mod...)，在某些分配模式下当数据块的数量发生变化的时候可能需要移动已有数据块中的数据且固定分配无法保证

各数据块的大小一致

动态分配则依靠一个字典确定键与数据块的映射关系，通过这个字典可以确定数据的位置从而尽量避免了跨分块的查询，因为它可以用很低的成本在数据块之间迁移特定的数据；如果数据量很大可能导致字典过大的情况，这时候可以考虑考虑混合固定/动态这两种分配方式，即key->(Hash)->map[0，1，2...]->(dynamics)->block[0，1，2...]

(5)，显式分配

显式分配既在一个字段中同时保存了分区编号与行编号；假设**分区编号**为x，**行编号**为y，**位移数**为b，位移后的**结果**为r；那么应得到公式(x<<b)+y=r，x=r>>b，y=r&~(x<<b)；这里的r作为最终的行标识存储在数据库中；需要注意的是b的长度必须足以容纳下转换成二进制后的y并且保证数据不会因为位移而失真

#?因为插入数据之前需要使用行编号与分区号，但是未插入时怎么获取行号？是否需要一个带auto_increment的表进行行号的分配？

#?向左位移的最大数超过多少时可能导致左边的数据被丢弃？

(6)，生成全局ID

可以使用MySQL自带的**uuid**或**uuid_short**函数或第三方uuid生成器；各服务设置不同的**auto_increment_increment**与**auto_increment_offset**；也可以使用使用一个带**auto_increment**的表进行分配；多列主键(自增ID+服务器ID或数据块ID)

(7)，NDB以及Federated

使用NDB引擎以及Federated引擎进行向外扩展请参考相应资料

12.1.3，回缩

回缩时需要防止大规模的归档操作与正常的操作之间的锁竞争，应按照深度优先的原则防止孤立数据的产生，删除之前应保证数据已正确存放到归档位置，设计适当的

解归档机制

对归档数据与活跃数据之间设置合理的间隔可以提高系统效率；InnoDB使用页级缓存，采用隔离措施可以保证缓存中的数据页中尽可能存放热度较高的数据；对于已经采用了行级缓存的Falcon引擎来说依然可以受益，Falcon每次对一页做索引，如果混合了冷数据就会造成一定程度的负面影响

也可以使用分区表来实现冷热数据的隔离或为热数据所在的节点使用更快的处理设备

#?如何判断数据的活跃程度？InnoDB使用页级缓存是否意味着它的缓存以及存取是以页为单位的(即使只有其中一条数据是真正需要的)?

12.2，负载平衡

负载平衡有五个主要指标：**可伸缩性**，**可用性**，**高效性**(通过路由控制引导使用恰当的软硬件资源)，**透明性**(客户端只"看到"负载平衡器而不用关心后端的架构)，**一致性**

12.2.1，直接连接服务器(绕过平衡器)

当应用需要访问一组完全对等可替换的服务器时，使用中央化的负载平衡是很好的选择，但是当应用需要自行判断数据来源时就必须与具体的服务器直接连接

对于master/slave结构来说最大的问题在于复制延迟，在必要的情况下对于不接受旧数据的查询可以直接在主服务器完成，也可以根据从服务器的迟滞程度来决定，或者在session层则可以设置一个标志来决定需要读取的服务器，用户提交数据时记录一个时间戳，根据这个时间戳的早晚决定是否读取主服务器，如果提交数据在10秒前，但从服务器5秒前已更新则可以读取从服务器(show master/slave status可以得到相关数据)

使用动态DNS与虚拟IP(Lvs支持)也是一个选择，如果从服务器能跟上主服务器则将读操作的域名与虚拟IP只想从服务器，否则指向主服务器

12.2.2，负载平衡器以及算法

MySQL负载平衡器分为软件与硬件的(软件的如**MySQL proxy**)，如果平衡器不知道后端各服务器的真实负载，则可能造成负载不均衡；平衡器还需要具有将适当的查询指向特定服务器的功能，尽可能操作已经预热的数据；另外增加服务器后需要调整负载平衡器的连接池大小否则有可能导致新增的服务器没有访问，因为新增服务器是"冷"的，所以应该提前将它作为其他活动服务器的从服务器运行一段时间

对于所有的服务器来说应多余的设置一些**max_connections**，在其他服务器崩溃时维持系统的正常运行

负载平衡的常用算法有：随机，轮询，最少连接优先，最快连接优先，散列化以及权重

12.2.3，主服务与多台从服务器的负载平衡

可以按照OLTP/OLAP等用途让不同的用户使用不同的从服务器；如果主服务器还没有对数据进行分区，那么可以在分发服务器采用带**触发器**的**Blackhole**表按一定的规则插入数据到不同的分区，然后配置从服务器进行有选择的复制

12.3，高可用性

高可用需要解决的问题是提供一定的数据冗余性在部分服务器发生故障时能进行在线替换，挑战是如何快速起可靠的完成这一过程

12.3.1，高可用规划

应该尽量发现和处理系统中的薄弱环节，可以通过一个公式来确定它的先后顺序即：**风险承担系数=部件故障概率*故障损失，**但也需要考虑发生故障后的恢复时间

12.3.2，冗余系统

冗余使用**SAN**(存储区域网络)当服务器宕机时由另外的服务器接驳这个文件

系统从而快速恢复服务(慎用难以修复的MyISAM表)；也可以使用**DRBD**技术复制磁盘架构

MySQL的第三方补丁(<url:"http://code.google.com/p/google-mysql-tools/">)提供了**半同步复制**的功能既：主机开启一个事务后直到有一个从服务器收到事件才会结束；对于**SolidDB**引擎而言，目前可以使用soid information technology公司提交的高可用技术，在这个方案下从服务器的写是多线程的且不会落后于主服务器

12.3.3，故障转移与恢复

除了常规的DNS与IP切换还有很多故障转移与恢复方法

Google提供了M-M复制的一系列管理工具来监控服务器状态(<url:"http://code.google.com/p/mysql-master-master/">)，当它发现服务器故障时会把IP指向另外的服务器；它也可以用于M-S以及M-M下的一个或多个从服务器

很多时候应用本身是最先获知服务器故障的，可以在应用中集成自动的故障处理方案，或系统异常时自动通知系统管理员

13，应用层面的优化

在系统中出现性能瓶颈的时候，并非问题的根源总在数据库服务器，有时候不恰当的应用设计与配置也会导致这一问题

13.1，检查应用程序

检查各应用程序服务器的配置以及资源使用率->应用是否获取了多余的数据或执行了多余的查询->当前工作是否交给数据库完成更有效率(反之亦然)->是否采用了正确的SQL写法->是否有必要建立到数据库的连接->负载均衡算法是否合理->是否使用了连接池->是否及时关闭或释放了不用的连接

13.2，Web 服务器层面的优化

使用轻量级的Web服务器如**lighttpd**或**nginx**代替apache处理静态内容并尽量使

它最小化；**squid**可以缓存之前响应的页面从而减小后端服务器的压力；开启**Gzip**压缩从而节省网络带宽；对于apache来说使用长连接是非常低效率的行为，可以使用lighttpd与squid作为它的前端

13.3，内容缓存

内容缓存的关键点在于：需要缓存的内容，如何缓存以及缓存过期；通常情况下缓存越靠近客户端就越有效率

对于需要被缓存的列表其中的每条记录的信息量很大，此时如果直接缓存这个列表那么缓存资源的利用率是非常低的，因为其他列表或许也包含了部分冗余的数据；这时候建立先缓存列表中所有记录的唯一标识，然后再按唯一标识分别缓存具体的记录

13.4，替代 MySQL

对于某些特定的操作，如果MySQL不能完成或不能很快速的完成，那么可以考虑适应替代品；如使用**sphinx**代理MyISAM做全文检索或在应用程序中处理数据的联接/分组/排序等操作

14，备份与还原

备份的主要目的有几个方面：灾难还原，恢复被用户正常删除的数据，审查某个时间点的数据以及测试

14.1，需要权衡的事项

对数据丢失的承受能力，通常来说承受能力越大越容易备份；如果承受能力非常小或完全不允许丢失数据，那么就需要一些特殊的手段如存放二进制日志到**SAN**或**DRBD**

使用**逻辑备份**还是**裸备份**；逻辑备份即导出数据文件是MySQL可识别的SQL或分隔符形式，它具有对服务器的版本以及存储引擎兼容性强的特点，导出数据的过程中如果磁盘损坏仍然有一定的机会导出正确的数据(因为部分数据可以在内存中获取)，缺点是速度较

慢以及导入时服务器的工作量较大

裸备份即直接复制数据文件，它的速度较快，导入的时候工作量也很小，对于InnoDB表应关闭服务器后再备份；对于MyISAM表可以通过**show table status**获得最后更新时间(Update_time)，而对于InnoDB则只能通过添加触发器来确定

14.2，需要备份什么

二进制日志/事务日志/中继日志以及相关索引文件，触发器/存储过程/存储函数(因为它们存在于MySQL系统数据库)，服务器配置文件，工具脚本等

14.3，备份方式

14.3.1，完全备份

即备份所有数据，这种备份以及恢复方式较为直观，但是备份时间较长且存在大量的冗余数据，备份过程对现有的系统有比较大的影响，在备份过程中有可能有新的数据被插入或修改(需要防止)

14.3.2，差异备份与增量备份

差异备份与增量备份的主要差别在于差异备份是备份自上一次完整备份之后的所有数据变动，而增量备份是备份自上一次任意备份以来的变动；定期进行完整备份是必要的，可以避免时间跨度太大的情况下漫长的恢复过程

MySQL二进制日志是最典型的增量备份方式，可以在备份时使用**flush logs**生成下一个新的二进制日志文件从而很容易的与之前的备份进行区分；也可以通过为所有记录增加一个timestamp来确定哪里数据需要备份

14.4，存储引擎与一致性

14.4.1，数据一致性

即要防止一个或多个具有关联关系的表在备份的过程中其他客户端又更新了

数据从而导致备份后的数据不完整，这通常需要**lock tables**或**flush tables with read lock**进行锁定；对于InnoDB因为具有Mvcc特性在特定的隔离级别下(如repeatable)可以规避这一问题(备份过程在必须在一个事务中)，但是如果备份发生在两个业务上有关联的事务之间时就必须要么合并事务要么锁定表

14.4.2，文件一致性

lock tables与**flush tables**的组合对于保证文件一致性有特殊的用处；对于MyISAM引擎，这个操作组合可以保证锁定表并将内存中还未写入的数据全部保存到磁盘；但是对于InnoDB引擎，这个组合既不能保证表不能被其他线程(自身的维护线程)写入，也不能保证所有内存中的数据都被立即刷新到磁盘，只能通过show engine InnoDB status进行确认

#?对于innodb表是否lock tables无法阻止MySQL后台线程对表的写入？

14.5，M-S 下的备份

在常规情况下无法防止主库误操作所带来的损失，因为无法预知从库落后于主库的程度，可以通过mk-slave-delay(maatkit工具集)工具实现MySQL的**延迟复制**，当主库发生误操作时有足够的时间发现并跳过误操作语句

14.6，备份二进制日志

二进制日志不仅为复制提供支持，同时也是一种不错的增量备份的方式；它包含了时间发生的时间，源服务器ID，特定记录的偏移量，执行特定事件的线程ID等信息，需要注意的是exec_time这个信息是不可信赖的

即使存在二进制日志，定期的完全备份仍然是减小恢复时间的必要手段；清除日志应配置MySQL的**expire_logs_days**参数，直接删除会导致*.index与磁盘文件不同步

14.7，数据备份

14.7.1，逻辑备份

mysqldump可以很好的完成数据的逻辑备份，但是它对于整库导出的时候存在文件过大导致编辑器无法打开以及在这种情况下无法只恢复特定的表的问题

可以使用**select outfile**/**load data**通过导出/导入定界符文件来规避mysqldump的上述缺陷，并且在同样的表上定界符文件的大小与导入/导出性能是优于mysqldump的

因为mysqldump是单进程导入/导出的，所以效率很低，可以使用mk-parallel-dump/mk-parallel-restore(Maatkit工具集)进行并行操作

14.7.2，文件系统快照

文件系统快照用户快速的在线备份与恢复，主要有两种实现方式：写时复制(**copy on write**)在复制一个对象时并不真正复制数据而是在快照中引用原有的位置并标注copy on write为1，当原有的数据块发生变动时才真正复制原有数据到新的地址；I/O冲定向(**I/O redirect**)请参见相关文档

linux系统上可以使用**LVM**技术实现它

14.8，备份还原

还原数据时应防止其他客户端对数据库的访问，可以使用**flush tables**，**read_only**，**skip_networking**或其他手段关闭连接通道

14.8.1，裸文件还原

对于MyISAM表比较简单，通常可以直接覆盖，如果表正被打开则可以使用**lock tables**与**flush tables**关闭它，服务器能立即发现这些变化

而对于InnoDB表则需要关闭服务器，覆盖时必须保证事务日志文件与表空间

文件是匹配的；如果**innodb_file_per_table**处于打开状态则数据与索引位于*.ibd中

所有工作完成准备重启服务器之间检查应检查MySQL服务器对被还原的文件是否具有适当的访问权限

14.8.2，逻辑备份还原

还原时如果不希望还原过程被二进制日志记录则应关闭**sql_log_bin**选项，同事应避免在一个事务中导入大量的数据从而导致它产生并写入一个非常大的回滚段

source以及load data(等同mysqlimport工具)针对不同的数据格式进行还原操作(SQL文件与定界符文件)，需要注意的是source过程中如果发生错误并不会导致还原任务的终止

14.8.3，基于时间点的还原

如果在一次完整的备份后发生一次或多次误操作并且中间又夹杂着正常操作，那么可以先读取这个完整备份，然后使用**mysqlbinlog**的**--start-position**或**--start-datetime**跳过误操作的部分，也可以按时间范围(--start-datetime)进行操作

14.8.4，基于延迟的还原

当主服务器发生的误操作还未到达从服务器时，先停止从服务器，然后设置**start slave until**让从服务器复制到特定的位置时停止，然后从服务器设置**sql_slave_skip_counter**跳过误操作部分

14.8.5，InnoDB 还原(结构损坏)

InnoDB的健壮性依赖无缓冲的I/O以及fsync调用，数据在真正写入到物理设备之前发生的服务器故障会导致数据损坏；损坏主要有以下三种类型:

(1)，二级索引损坏

可以通过optimize进行修复，也可以使用select outfile与load data进行表

重建

(2)，集群索引损坏

通常可以设置不同的**innodb_force_recovery**参数来导出表，如果导出过程导致InnoDB崩溃则需要缩小导出的范围；innodb_force_recovery参数控制InnoDB在启动与做常规操作时的服务器行为

(3)，系统结构损坏

除了可以参考innodb_force_recovery参数的设置外，如果InnoDB损坏到MySQL无法启动的程度，可以参考使用<url:"http://code.google.com/p/innodb-tools/">中的工具

14.9，备份工具

除了常规的mysqldump工具，对于不同的引擎还有不同的第三方工具可用；对于MyISAM表可以使用**mysqlhotcopy**；对于InnoDB表可以使用**ibbackup**/**xtrabackup**，它不用停止MySQL&设置锁以及中断常规的数据库活动；LVM也是一个很好的选择

mk-parallel-dump/restore，它相当于对mysqldump的多线程封装，也能导出定界符格式的文件

15，服务器安全

15.1，MySQL 帐户认证体系

MySQL的授权不仅仅凭借用户名与密码，还包括了访问的来源；权限按授予对象的不同可以分为两种：与对象有关的(表创建，列的读取权限等)，与对象无关的如：flush表，关闭服务器以及执行其他管理任务

15.1.1，授权表

它是一组存在于MySQL系统数据库中的MyISAM表；user(记录基本认证以

及全局权限)，host(与来源相关的权限)，db(数据库级权限)，tables_priv(表级权限)，columns_priv(列级权限)，procs_priv(存储过程与函数的权限)

是否授于权限会根据这个顺序user->db(host)->tables_priv-> columns_priv->procs_priv，在第一个发现Y(允许)的地方停止搜索并授权，否则拒绝操作；因此对于外部使用的帐户在user表中相应的权限设置为N是必要的；对于没有明确的授权默认处理方式均为拒绝

Host表不会被**grant**/**revoke**影响(需要时手动修改)，当db表中特定的记录host字段为空时则执行db.db=host.db and db.priv=host.priv=Y，如通过则允许反之则拒绝；需要注意的是如果db.priv=N则直接拒绝而不会执行之前的联接

15.1.2，权限分配与查看

主要命令有**show grants** [**for** 用户名]查看权限(默认当前用户)，**grant**/**revoke**授予/取消权限，**create**/**drop user**创建/删除用户，**flush privileges**刷新权限；分配权限时可以对@右边的部分使用通配符从而进行批量授权，但是必须使用引号包含@右边的部分

对于information_schema系统数据库，MySQL对自动处理用户对它的访问权限，更改它的访问权限可能导致异常

15.1.3，不同版本之间的差异

从MySQL4.1开始引入了更安全的新密码算法(以*开头)，新密码为41位比老密码多出25位；**secure_auth**参数控制是否接受使用旧格式密码的帐户的认证请求；对于使用旧加密算法的客户端可以使用old_password函数设置密码

MySQL5.0引入了存储程序以及相应控制权限的支持，**SQL SECURITY** {**DEFINER**|**INVOKER**}可以在创建存储程序时指定其执行时使用的身份；不仅要确保特定的用

户具有可以调用存储程序的权限(**Execute**)，也要保证对被间接访问的表具有适当的权限，对于视图以及触发器等也是如此

对于触发器则可以分配trigger权限. *#?似乎有点冗余，即使不分配trigger权限只要当前用户对触发器操作的表有适当的权限即可反之即使分配了trigger权限如果没有对它所操作的表的权限也是不能访问的，为什么存在这个权限?*

15.1.4，权限对性能的影响

主要影响在于如果权限太多或太细则会导致的一定的权限核对成本以及查询缓存不会保存引用了具有列级权限的表的数据，有鉴于此全局权限(user表记录)是最快的;同时也要防止反向域名解析所带来的负面影响

15.1.5，常见问题以及处理方法

对于某些比较严格的应用场景，可能不允许用户在数据库中执行特定的DDL操作(如create temporary)但是又需要放开对临时表的所有权限，这时候可以另外建立一个数据库并授予用户在其中的操作权限

MySQL允许没有密码的用户，可以在[client]中加入一行password(但是需要注意某些客户端会忽略它)，还可以设置**sql_mode**为**no_auto_create_user**方式防止**grant**自动创建没有密码的用户(除非同时指定了密码)

注意安全信息的泄露；如果用户有权访问的数据库中存在Federated表(即使用户对库只有insert权限)，它也可以通过**show create table**来查看**Federated**表的远程地址以及密码；同样对于MySQL系统数据库应严防普通用户对它进行任何形式的访问

Super权限的用户在服务器上的操作不受read_only的限制；需要注意的是当服务器的连接数达到max_connections时，仍然有一个max_connections+1的连接通道可以供super权限的帐户使用

使用通配符批量授权，如grant select on `bac%`.*(_必须写成_)就能对所有以bac开头的数据库进行授权(`符号是必须的)，但是无法批量匹配表以及用户名

废弃的权限；使用**drop user**删除特定的用户会自动清理相关的权限表(host表除外)，但是删除相应的资源则不会执行这一过程；可能会导致一个新建且创建后从未进行任何授权操作的资源对某些用户应用了之前的授权策略

匿名用户所带来的影响；匿名用户在user表中的表现形式为空字符串的user字段，如果相关资源权限表(db，tables_priv等)没有明确的指定user，那么即表示所有用户(包括匿名用户)均具有这些权限

15.2，服务器安全

主要是需要定期检查服务器更新，配置合理的权限以及不要用特权帐号运行MySQL

15.3，网络安全

15.3.1，访问来源控制

如果应用与MySQL在同一台服务器上，那么就可以在配置文件的[**MySQLD**]节点中加入**skip_networking**来拒绝本机之外的网络连接(在从服务器上设置不会影响复制)，也可以使用**bind-address**设置只允许来源为特定IP的连接

15.3.2，加密传输

可以使用VPN连接MySQL，VPN通过建立一个隧道对传输的数据进行了加密；MySQL原生支持ssl(have_openssl参数控制)，可以使用grant require进行相关的设置，但是它会增加一些开销；如果客户端是为linux或unix服务器，那么使用**SSH隧道技术**也是一个很好的选择，在客户端机器执行SSH -N -f -l 4406:服务器IP:3306，然后就可以使用MySQL -h 127.0.0.1进行连接

15.3.3，自动屏蔽客户端

存在一个参数**max_connect_errors**控制允许控制来自特定主机的错误连接数，达到上限后拒绝连接并将错误写入错误日志，**aborted_connects**变量可以观察失败的连接数；重启或**flush hosts**后重置这个计数器

15.3.4，数据加密

MySQL有大量的加密函数可用：**encode**/**decode**，sha1，md5，aes_encrypt/aes_decrypt，password(需要注意password函数存在版本差异)以及old_password等

对于文件系统的加密是必要的，它对MySQL是透明的，但是它有一定的系统开销；也可以在应用层实现加密

16，服务器状态

MySQL的状态大体分为两种，**系统变量**(show variables)与**系统状态**(show status)，它们均存在两个层级即global与session且都可以从**information_schema**系统数据库中相关表中获取;相关信息 需要注意部分MySQL计数器在达到最大值后会归零

16.1，线程与连接相关信息

connections(试图连接到MySQL服务器的连接数，无论是否成功)，**max_used_connections**(服务器启动后同时使用的最大连接数)，**threads_connected**(当前打开的连接数量)，**threads_created**(创建了多少线程用来处理连接)，**threads_cached**(缓存内的线程数量)，**Threads_running**(激活连接数)

Aborted_clients(连接没有被正确关闭的次数)，**Aborted_connects**(试图连接到服务器但连接失败的数量)

Bytes_received(从所有客户段收到的字节数)，**Bytes_sent**(发送给所有客户端的字节数)

Slow_launch_threads(创建连接时间超过此值的连接数)

16.2，二进制日志相关状态

binlog_cache_size设置事务日志缓冲区大小，可以根据Binlog_cache_use/Binlog_cache_disk_use的状态进行参考设置，前者表示缓存中的事物数量，后者为有多少事务因为binlog_cache_size已满而使用临时文件存放

16.3，SQL 命令计数器

questions记录了服务器上执行的SQL的行数(包括管理语句)而不是客户端发送SQL的条数，各类型的select主要通过show status like 'select%'获取；Select_full_join为没有使用索引的联接的数量，Select_full_range_join在引用表中使用范围搜索的联接数

服务器也提供了一组状态用来标识特定操作被执行的次数：com_delete，com_update，com_select，com_insert，com_commit，com_rollback等

16.4，临时文件与临时表

created_tmp_disk_tables服务器在硬盘上自动创建的临时表数量，Created_tmp_files服务器创建的临时文件的数量，Created_tmp_tables创建的临时表数量

16.5，handler 操作

主要通过show status like 'handler%'分析服务器做的最多的操作是哪些，如Handler_commit，Handler_delete，Handler_rollback等

16.6，MyISAM 索引键缓冲区

Show status like 'key%'

16.7，MyISAM 文件描述符

Show status like 'open%'

16.8，查询缓存

Show status like 'qcache%'，**com_select**为未命中缓存的select数量，**qcache_hits**为已命中查询缓存的数量

16.9，表锁定

Table_locks_immediate能立即获得表锁的次数，**Table_locks_waited**不能立即获得表锁的次数，见<label:"4.1，存储引擎摘要">

16.10，show engine InnoDB status

它主要展示InnoDB引擎内部的信息，有部分信息会在间隔一段时间后清零，可以使用innotop工具分析并监控InnoDB

Latest foreign key error会标注服务器发生的外键错误；**latest detected deadlook**用来标注服务器出现的死锁；**transactions**包含了InnoDB事务信息的概要

File I/O节点显示I/O helper线程的状态，至少有四个：插入缓冲区线程(将记录从插入缓冲合并到表空间)，日志线程(负责异步日志的刷新)，读线程(预测并读取InnoDB需要的数据)，写线程(刷新脏缓冲区)

INSERT BUFFER AND ADAPTIVE Hash INDEX节点展示插入缓冲区以及自适应Hash索引的状态

Log节点展示已写入日志文件的总字节数以及最近一次检查点

Buffer pool and Memory节点显示InnoDB申请的内存总数，缓冲池的总大小，空闲页的数量，脏页数量以及缓冲池的命中率

16.11，information_schema 系统数据库

它保存了MySQL中各个数据库以及相关**表**/**视图**/**触发器**等的完整**模式信息**以及系统状态与变量，show系列命令是对其中元数据的再封装

17，MySQL 相关工具

17.1，监控工具

Nagios，zenoos，hyperic hq，opennms，innotop等

17.2，分析工具

HackMySQL，maatkit，

17.3，辅助工具

mk-archiver清除于归档，mk-find用于数据库与表，mk-parallel-dump/restore多线程逻辑备份/恢复，mk-show-grants排除&分类&排序grant，mk-slave-delay延迟复制，mk-slave-prefetch预热从服务器中继日志，mk-table-checksum验证一个或多个服务器数据是否一致，mk-table-sync发现表之间的差异并生成解析它们的最小SQL命令集

18，锁的调试

MySQL服务器级锁主要有表锁，全局锁(**flush tables with read lock**)，名称锁(重命名或删除表)与字符串锁(get_lock函数)，另外一些关于锁的信息参见<label:"2，MySQL中的锁">

18.1，表锁与全局读锁

表锁分为显式/隐式表锁，显式表锁通常由**lock tables read**/**write**触发，而隐式锁由MySQL自动添加与释放(可通过**sleep**函数测试)；当一个线程等待其他线程释放表锁的时候会看到show processlist中state为**Waiting for table [level lock|metadata lock]**(*#?书上说的state为locked为何没有出现?*)，分别为read与write锁；可以使用**mysqladmin**工具同时加入**debug**参数查找谁持有表锁

全局读锁由**flush tables with read lock**创建，对其他线程它的等待会在state显示**Waiting for global read lock**；与表锁不同的是在执行它之前必须释放已有的表

锁否则会发生错误，而申请新的表锁时会自动释放之前的表锁

18.2，名称锁

它也属于表锁的一种，在重命名或删除表时出现；等待它的线程的state为**waiting for table**，可以通过**show open tables**查看哪些表(注意不是线程)持有名称锁

18.3，字符串锁

使用select **get_lock**(字符串名称，秒)获取一个锁，一个用户在同一时刻不能拥有一把以上的用户锁，申请新的用户锁时会自动释放以前的用户锁；被阻塞的线程state为user lock；目前还没有办法获知谁拥有某个用户锁

18.4，存储引擎中的锁等待

18.4.1，InnoDB 锁等待

除了show processlist，也可以通过**show engine InnoDB status**查看transactions节点了解谁在等待锁，但是无法查看谁持有锁，只能通过多次运行show engine InnoDB status观察哪些长时间运行的事务；MySQL提供了一组**监控器**，如下

innodb_monitor/**innodb_lock_monitor**(间隔16秒)前者是对show engine InnoDB status的封装，而后者具有更多的关于锁的信息但每个事务最多只能打印其所持有的10个锁；**innodb_table_monitor**/**innodb_tablespace_monitor**(间隔64秒)前者包含表的结构以及内部信息，后者则包含表空间(不包含表独立空间)的信息

监控器定时运行并输出信息到错误日志且会影响，删除特定的表即可停止监控器，开启监控器后show engine InnoDB status也会有额外的输出；重启服务器后需要删除并重新建立监控器

Innotop的**lock**模式也可以查看事务持有的锁

18.4.2，Falcon 锁等待

大约在MySQL6.0时可以在**information.Falcon_transaction**中找到它的信息

附录 A: Describe

它对于存储过程/函数，触发器是无效的；对于内存排序与临时文件它也无法区分而统一显示为filesort；同时它也是不太精确

(1)，ID列

SQL查询序列号(较大的ID先执行，如果一样则从上到下的顺序)，在select_type为union result的情况下ID为空，因为对临时表的使用不会出现在原始中，而DERIVED derived及subquery代表的是子查询而不是临时表(尽管它也有可能使用了临时表)

(2)，select_type列

Simple：简单查询

Primary：最外面的select查询

Subquery/Derived：非from/from中的子查询

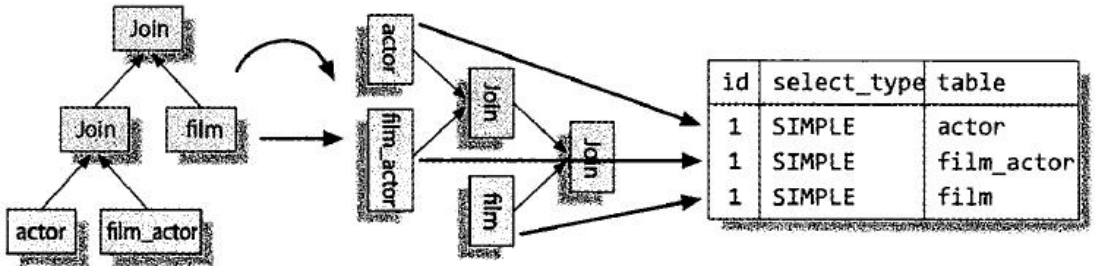
Union：在union中第一个select之外的select语句块(注意是union级别的查询而不是子查询)

Dependent：可能与union或subquery结合，表示依赖外部的查询

Union result：各union子句结果集所在的临时表

(3)，table列

表示查询访问的表，通过它可以观察到实际的执行顺序，对于联接操作则如下图:



From语句中子查询table的表现形式为derivedN，这里的N表示这个子查询的数据源来自ID为N的那一行(N与ID有对应关系)

如果查询由一连串的union构成，那么在union result行的table为<unionN1，N2...>，这里的n同样是对应其他行的ID

(4)，type列

All：全表扫描

Index/range：全索引扫描/有限制的索引扫描

Ref/eq_ref：前者表示对于传入的值(常数或前一个/组表的结果集)可能存在多个匹配；后者表示对于传入的值最多只有一个匹配，它只能出现在对primary与unique索引的查询上；ref有一个变种ref_or_NULL用来表示查询条件中同时需要查询是否在NULL的情况(x=56 or x is NULL)，它会进行二次搜索即第一次找关键字，第二次匹配NULL

Const/system：对主键或其他唯一索引进行查询时出现，这一行的列值会被查询优化器视为一个常数；system则表示在之前的前提上这个表中仅有一行

NULL：对于设置为not NULL列的is NULL查找，甚至不需要查找索引就可以得到需要的结果

Unique/index_subquery：对子查询使用的算法

index_merge：索引合并

(5)，possible_keys/key

分别表示可能使用的索引以及最后MySQL实际使用的索引

(6)，key_len

表示MySQL在查询中使用索引的字节数；如果字段没有定义为not NULL，那么无论对定长还是变长数据均有一个额外的字节表示字段是否为NULL，对于变长数据还需要额外的两个字节存储字段的实际长度；同时key_len也受到字符集的影响：latin1中一个字符一个字节，gbk一个字符两个字节，utf8一个字符三个字节

通过这个长度可以判断查询优化器使用了索引的哪些部分，需要注意的是key_len的长度取决于定义时的长度而不是实际长度

(7)，ref列

对于当前行所对应的表，使用那一列或常数结合与key一起在当前表中进行查找

(8)，rows列

为了找到最终所需的行而大约需要读取的行数，这是一个估计值

(9)，filtered列

仅对describe添加extended时出现

(10)，extra列

Using filesort：文件排序

Using temporary：需要创建临时表容纳结果