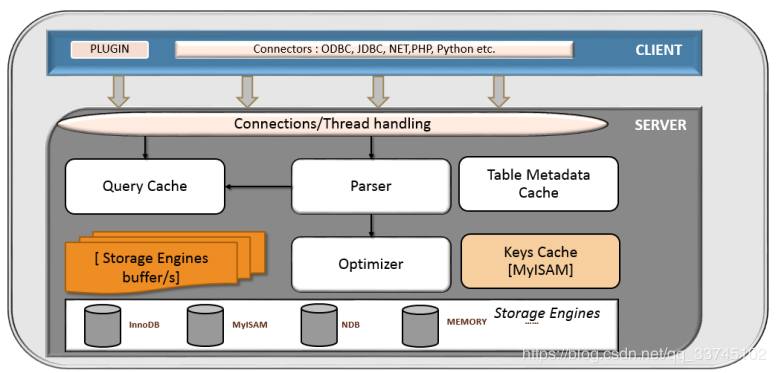
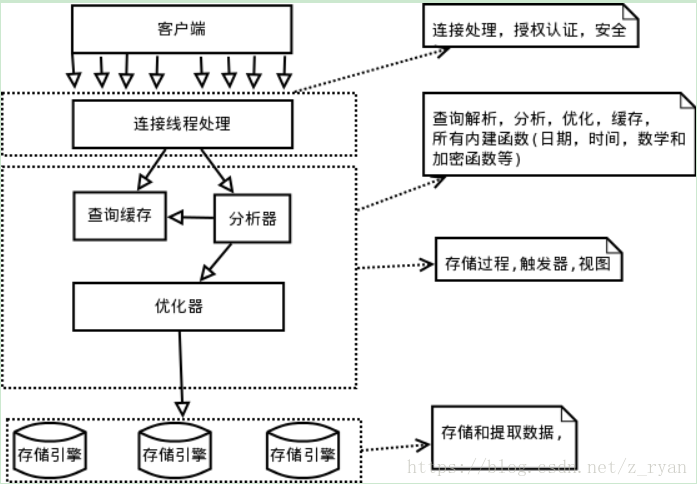
# mysql服务架构

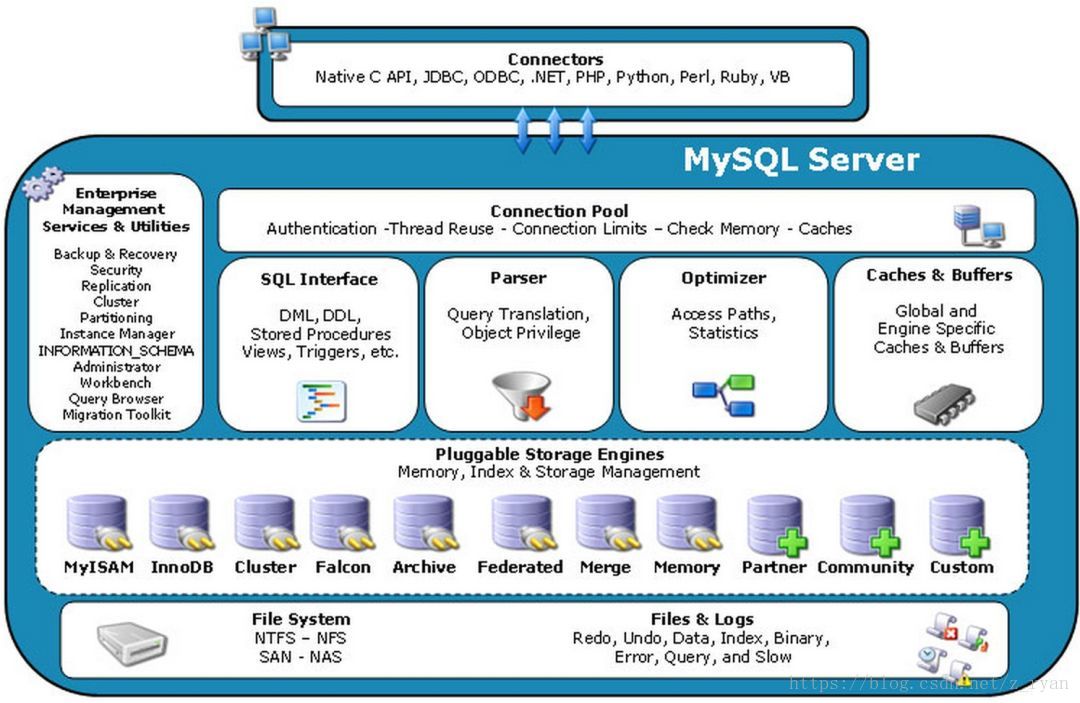
## 逻辑架构（英文）



## 逻辑架构（中文）



## 逻辑架构（详细）



* **概念解析：**

1. Connectors

　　指的是不同语言中与SQL的交互。

1. Connection Pool

管理缓冲用户连接，线程处理等需要缓存的需求。负责监听对 MySQL Server 的各种请求，接收连接请求，转发所有连接请求到线程管理模块。每一个连接上 MySQL Server 的客户端请求都会被分配（或创建）一个连接线程为其单独服务。而连接线程的主要工作就是负责 MySQL Server 与客户端的通信，接受客户端的命令请求，传递 Server 端的结果信息等。线程管理模块则负责管理维护这些连接线程。包括线程的创建，线程的 cache 等。

1. Management Serveices & Utilities

　　系统管理和控制工具。

1. SQL Interface

　　接受用户的SQL命令，并且返回用户需要查询的结果。

1. Parser

　　SQL命令传递到解析器的时候会被解析器验证和解析。解析器是由Lex和YACC实现的，是一个很长的脚本。在 MySQL中我们习惯将所有 Client 端发送给 Server 端的命令都称为 query ，在 MySQL Server 里面，连接线程接收到客户端的一个 Query 后，会直接将该 query 传递给专门负责将各种 Query 进行分类然后转发给各个对应的处理模块。

主要功能：**生成语法树**

　　a 、 将SQL语句进行语义和语法的分析，分解成数据结构，然后按照不同的操作类型进行分类，然后做出针对性的转发到后续步骤，以后SQL语句的传递和处理就是基于这个结构的；

　　b、 如果在分解构成中遇到错误，那么就说明这个sql语句是不合理的。

1. Optimizer

　　查询优化器：SQL语句在查询之前会使用查询优化器对查询进行优化。就是优化客户端请求query，根据客户端请求的 query 语句，和数据库中的一些统计信息，在一系列算法的基础上进行分析，得出一个最优的策略，告诉后面的程序如何取得这个 query 语句的结果。一般2种方式：**规则优化、成本优化**

　　使用的是“选取-投影-联接”策略进行查询：

用一个例子就可以理解： select uid,name from user where gender = 1;

这个select 查询先根据where 语句进行选取，而不是先将表全部查询出来以后再进行gender过滤；然后根据uid和name进行属性投影，而不是将属性全部取出以后再进行过滤。最后将这两个查询条件联接起来生成最终查询结果。

1. Cache和Buffer

　　查询缓存：主要功能是将客户端提交 给MySQL 的 Select 类 query 请求的返回结果集 cache 到内存中，与该 query 的一个 hash 值 做一个对应。该 query 所取数据的基表发生任何数据的变化之后， MySQL 会自动使该 query 的Cache 失效。在读写比例非常高的应用系统中， Query Cache 对性能的提高是非常显著的。当然它对内存的消耗也是非常大的。**但是在8.0之后取消缓存区**

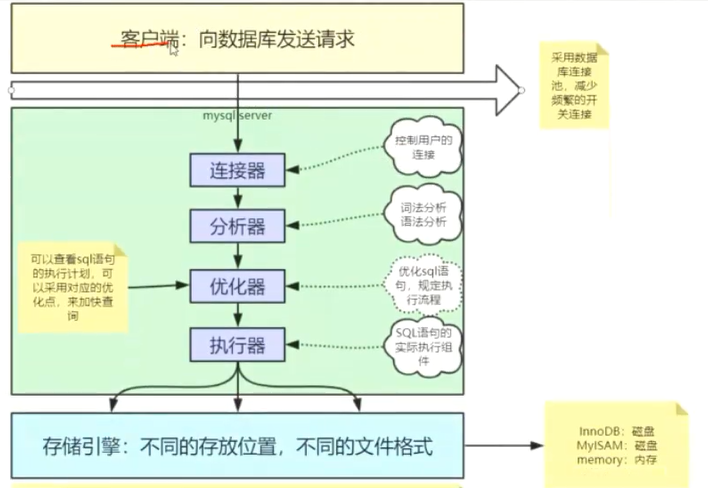
　　如果查询缓存有命中的查询结果，查询语句就可以直接去查询缓存中取数据。这个缓存机制是由一系列小缓存组成的。比如表缓存，记录缓存，key缓存，权限缓存等。

1. 存储引擎接口

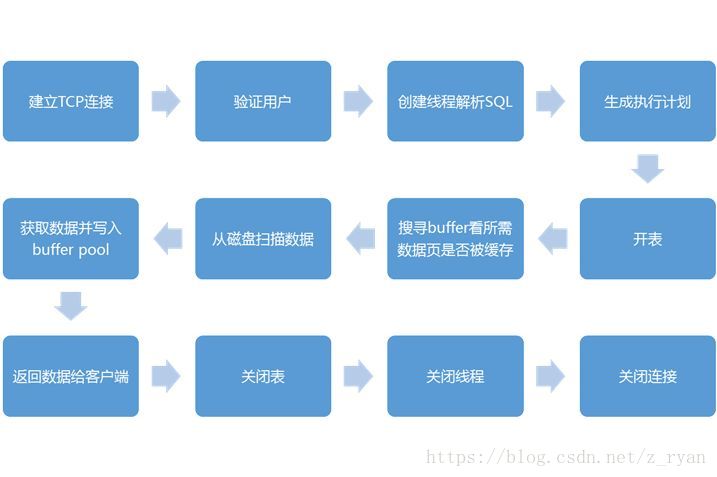
　　MySQL区别于其他数据库的最重要的特点就是其插件式的表存储引擎。MySQL插件式的存储引擎架构提供了一系列标准的管理和服务支持，这些标准与存储引擎本身无关，可能是每个数据库系统本身都必需的，如SQL分析器和优化器等，而存储引擎是底层物理结构的实现，每个存储引擎开发者都可以按照自己的意愿来进行开发。

**注意：存储引擎是基于表的，而不是数据库。**

## 逻辑示意图



## 数据库工作流程



* **流程描述：**

**在这里，我从数据库架构的三个层面分别介绍数据库的工作流程：**

**最上层：客户端连接**

　　1、连接处理：客户端同数据库服务层建立TCP连接，连接管理模块会建立连接，并请求一个连接线程。如果连接池中有空闲的连接线程，则分配给这个连接，如果没有，在没有超过最大连接数的情况下，创建新的连接线程负责这个客户端。

　　２、授权认证：在真正的操作之前，还需要调用用户模块进行授权检查，来验证用户是否有权限。通过后，方才提供服务，连接线程开始接收并处理来自客户端的SQL语句。

**第二层：核心服务**

　　1、连接线程接收到SQL语句之后，将语句交给SQL语句解析模块进行语法分析和语义分析。（**生成语法树**）

　　2、如果是一个查询语句，则可以先看查询缓存中是否有结果，如果有结果可以直接返回给客户端。

　　3、如果查询缓存中没有结果，就需要真的查询数据库引擎层了，于是发给SQL优化器，进行查询的优化。如果是表变更，则分别交给insert、update、delete、create、alter处理模块进行处理。

**第三层：数据库引擎层（IO限制，select \* 不推荐写）**

　　1、打开表，如果需要的话获取相应的锁。

　　2、先查询缓存页中有没有相应的数据，如果有则可以直接返回，如果没有就要从磁盘上去读取。

　　3、当在磁盘中找到相应的数据之后，则会加载到缓存中来，从而使得后面的查询更加高效，由于内存有限，多采用变通的LRU表来管理缓存页，保证缓存的都是经常访问的数据。

最后，获取数据后返回给客户端，关闭连接，释放连接线程。

## 基于服务架构，要注意

1. 由于mysql的执行器，最终是交给存储引擎来查询结果，存储引擎其实就是与磁盘就行IO交互，获取数据，而相对于内存读写来说，IO操作是非常慢的，因此我可以通过**减少IO操作次数、减少每次获取文件的大小来提高IO效率**，因此非必要情况下，**不要写select \* from table**
2. 基于逻辑架构图来说，我们针对mysql的优化，应该针对**分析器、优化器这2块进行的**

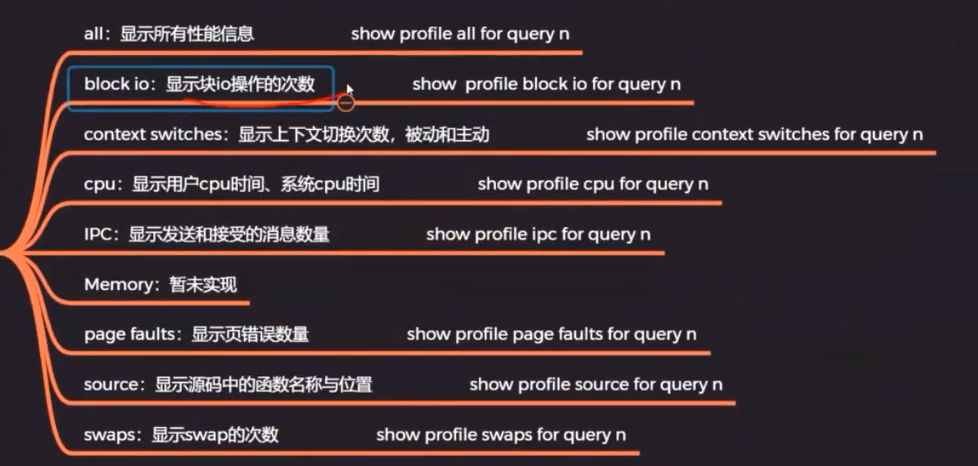
# mysql监控-profile查看执行时间

## show profile

官网地址 SQL Statements -> Database Administration statements -> show statements -> show profile Statement

1. 设置 SET profiling = 1;
2. show profiles; //显示查询过sql的总的执行时间
3. show PROFILE; //默认显示最近查询的执行时间明细
4. SHOW PROFILE FOR QUERY 1; //指定查询ID,在show profiles;可以看到 查询ID

## show profile 具体参数



## show profile 注意

官网中提到，show profile 可能在将来将要被替换掉，暂时8.0及以前的可以放心使用。

**官网原话：**

**The SHOW PROFILE and SHOW PROFILES statements are deprecated and will be removed in a future MySQL release. Use the Performance Schema instead; see Section 26.19.1, “Query Profiling Using Performance Schema”.**

# MySQL监控-Performance Schema

## 文档地址

官网地址：MySQL Performance Schema

https://dev.mysql.com/doc/refman/5.6/en/performance-schema.html

mysql5.7是默认开启Performance Schema

## performance\_schema的介绍

**Mysql的performance schema用于监控MySql server在一个较低级别的运行过程中的资源消耗、资源等待的情况。**

* **特点如下：**

1. 提供了一种在数据库运行时实时检查server的内部执行情况的方法，performance schema数据库中的表使用performance\_schema存储引擎、该数据库主要关注数据库运行工程中的性能相关的数据，与information\_schema不同，information\_schema主要关注server运行过程中的元数据信息。
2. performance\_schema通过监视server的事件来实现监控server内部运行情况，**事件**就是server内部活动中所做的任何事情及对应的事件消耗，利用这些信息来判断server中的相关资源消耗在哪里？一般来说，事件可以是函数调用、操作系统的等待、SQL语句执行的阶段（如sql语句执行过程中的parsing或sorting阶段）或者整个SQL语句与SQL语句集合。事件的采集可以方便的提供sever的相关存储引擎对磁盘文件、表IO、表锁等资源的同步调用信息。
3. performance\_schema中的事件与写入二进制日中的事件（描述数据修改的events）、事件计划调度程序（这是一种存储程序）的事件不同。performance\_schema中的事件记录的是server执行某些活动对某些资源的消耗、耗时、这些活动执行的次数等情况。
4. performance\_shema中事件只记录在本地server的

performance\_schema中，其下的这些表中数据发生变化时不会被写入binlog中，也不会通过复制机制被复制到其他server中。

1. 当前活跃事件、历史事件和事件摘要相关的表中记录的信息。能提供某个事件的执行次数、使用时长。进而可用于分析某个特定线程、特定对象（如mutex或file）相关联的活动。
2. performance\_schema存储引擎使用sever源码中的“检测点”来实现事件数据的搜集。对于performance\_schema实现机制本身的代码没有相关的单独线程来检测，这与其他功能（如复制或时间计划程序）不同。
3. 收集的事件数据存储在performance\_schema数据的表中，这些表可以使用select语句查询，也可以使用sql语句更新performance\_schema数据库中的表记录（如动态修改performance\_schema的setup\_\*开头的几个配置表、但要注意：配置表的更改会立即生效，这会影响数据收集）
4. performance\_schema的表中的数据不会持久化在磁盘中，而是保存在内存中，一旦服务器重启，这些数据会丢失（包括配置表中内的整个performance\_schema下的所有数据）。
5. mysql支持的所有平台中事件监控功能都可用，单不同平台中用于统计事件开销的计时器类型可能会有所差异。

## performance schema入门

* **在mysql的5.7 版本中，性能模式是默认开启的，如果想要显式的关闭的话需要修改配置文件，不能直接进行修改，会报错Variable ‘performance\_schema’ is a read only variable.**

1. **查看performance\_schema的属性**

|  |
| --- |
| mysql> SHOW VARIABLES LIKE 'performance\_schema'; |

1. **在配置文件中修改performance\_schema的属性，on表示开启，off表示关闭**

|  |
| --- |
| [mysqld]  performance\_schema=ON |

1. **切换数据库**

|  |
| --- |
| use performance\_schema; |

1. **查看当前数据库的所有表，会看到很多表存储相关新**

|  |
| --- |
| **show tables;** |

1. **可以查看表结构**

|  |
| --- |
| **show create table setup\_consumers;** |

* **基本概念：**

1. **instruments:**生成者，用于采集mysql中各种的操作产生的事件信息，对应配置表中的配置项我们可以为监控采集配置项；
2. **consumers**:消费者，对应的消费者表用于存储来自instruments采集的数据，对应配置表中的配置项我们可以称为消费存储配置项。

## performance\_schema表的分类

**performance\_schema库下的表可以按照监视不同的纬度分组**

* 语句事件记录表，记录了语句事件信息

1. 当前语句事件表：events\_statements\_current
2. 历史语句事件表：events\_statements\_history
3. 长语句历史事件表：events\_statements\_history\_long
4. 聚合后摘要表：summary,其中summary表还可以根据账号（account）、主机（host）、程序（program）、线程（thread）、用户（user）、全局（global）在过滤细分

|  |
| --- |
| show tables like '%statement%'; |

* 等待事件记录表，与语句事件类型的相关记录类似

|  |
| --- |
| show tables like '%wait%'; |

* 阶段事件记录表，记录语句执行的极端事件的表

|  |
| --- |
| show tables like '%stage%'; |

* 监控文件系统层调用的表

|  |
| --- |
| show tables like '%file%'; |

* 监控内存使用的表

|  |
| --- |
| show tables like '%memory%'; |

* 动态对performance\_schema进行配置的配置表

|  |
| --- |
| show tables like '%setup%'; |

## performance\_schema的简单配置与使用

### 简单配置

**数据库刚刚初始化并启动时，并非所有instruments（事件采集项，在采集项的配置表中每一项都一个开关字段，或为YES,或为NO）和consumers（与采集项类似，也有一个对应的事件类型保存表配置项，为YES就表示对应的表保存性能数据，为NO就表示对应的表不保存性能数据）都启用了，所以默认不会搜集所有的事件，可能你需要检测的事件并没有打开，需要进行设置，可以使用如下两个语句打开对应的instruments和consumers（行计数可能会因MySQL版本而异）。**

* **打开等待事件的采集器配置开关，修改setup\_instruments表中对应的采集配置项：**

|  |
| --- |
| **UPDATE setup\_instruments SET ENABLED = 'YES', TIMED = 'YES'**  **WHERE name like 'wait%';** |

* **打开等待事件的消费者配置开关，修改setup\_consumers**

|  |
| --- |
| **UPDATE setup\_consumers SET ENABLED = 'YES' WHERE name like '%wait%';** |

* **当配置完成之后，可以查看当前sever正在做什么，可以通过events\_waits\_current表来得知，该表每个线程只包含一行数据，用于显示每个线程的最新监视事件**

|  |
| --- |
| **select \* from events\_waits\_current;** |

### 等待事件记录表使用

#### \_current表解释

* **events\_waits\_current重点字段解释**

**14 //thread\_id 线程**

**12 //event\_id 事件**

**12 //end\_event\_id 结束事件**

**wait/synch/mutex/innodb/buf\_dblwr\_mutex //event\_name事件名字**

**buf0dblwr.cc:898 //source 这个校验代码在那个源文件中及行号**

**93753488864139500 //timer\_start 开始时间**

**93753488864429695 //timer\_end 结束时间**

**290195 //timer\_wait 耗时**

**291405928 //object\_instance\_begin**

**lock //操作类型**

**这条语句表达的意思是：线程14 正在等待buf\_dblwr\_mutex锁，等待耗时290195**

* **注意：\_current表中每个线程只能保留一条记录，当线程完成工作，该表中不会再记录改线程的事件信息。**

#### \_history表解释

* \_history表中记录每个线程应该执行完成的事件信息，但每个线程的事件信息只会记录10条，再多就会被覆盖。
* \_history\_long 表中记录所有线程的事件信息，但总记录数量是10000，超过就会被覆盖掉

|  |
| --- |
| SELECT thread\_id,event\_id,event\_name,timer\_wait FROM  events\_waits\_history ORDER BY thread\_id LIMIT 21; |

#### summary表解释

* summary表提供所有事件的汇总信息，该组中的表以不同的方式汇总事件数据（如：按用户，按主机，按线程等），例如：要查看哪些instruments占用最多的时间，可以通过对

events\_waits\_summary\_global\_by\_event\_name表的count\_star或

sum\_timer\_wait列进行查询（这两列对时间的记录数执行count(\*)、时间记录的timer\_wait列执行sum(timer\_wait)统计而来）

|  |
| --- |
| SELECT EVENT\_NAME,COUNT\_STAR FROM  events\_waits\_summary\_global\_by\_event\_name ORDER BY COUNT\_STAR DESC LIMIT 10; |

#### instance表解释

* instance表记录哪些类型的对象会被检测，这些对象在被server使用时，在该表中将会产生一条事件记录，例如：file\_instance表列出文件IO操作及其关联文件名

|  |
| --- |
| select \* from file\_instances limit 20; |

## 常用配置项的参数说明

### 启动选项

1. performance\_scheam\_consumer\_events\_statements\_current=TRUE

是否在mysql server 启动时候就开启events\_statements\_current表的记录功能（该表记录当前的语句事件信息），启动之后可以再

setup\_consumers表中使用UPDATE语句进行动态更新setup\_consumers配置表中的events\_statements\_current配置项，默认值为TRUE

1. performance\_schema\_consumer\_events\_stages\_history =TRUE

与（1）中的选项类似，但该选项是用于配置是否记录语句事件短历史信息，默认为TRUE

1. performance\_schema\_consumer\_events\_stages\_history\_long=FALSE

配置长历史信息，默认FALSE

1. 除了statement（语句）事件之外，还支持wait（等待）事件、state（阶段）事件、transaction（事物）事件，他们与statement事件一样都有3个启动项分别设置，但这些等待事件默认未启用，如果需要mysql server启动时一同启动，通常需要写进my.cnf配置文件中
2. performance\_schema\_consumer\_global\_instrumentation=TRUE

是否在mysql server 启动时开启全局表（如：mutex\_instance、rwlock\_instances、cond\_instances、file\_instance、user、hostsaccounts、socket\_summary\_by\_event\_name、file\_summary\_by\_instance等大部分的全局对象计数统计和事件汇总统计信息表）的记录功能，启动之后也可以在setup\_counsumers表中使用UPDATE语句进行动态更新全局配置项，默认为TRUE

1. performance\_schema\_consumer\_statements\_disgest=TRUE

默认为TRUE,可以在setup\_consumers表中使用update局域动态更新digest配置项

1. performance\_schema\_consumer\_thread\_instrumentation=TRUE

是否在mysql server 启动时开启

1. events\_xxx\_summary\_by\_yyy\_by\_event\_name

默认true

1. performance\_schema\_instrument[=name]

是否在mysql server启动时启用默写采集策略，由于instruments配置项多个，所有改配置项支持key-value模式，还支持%进行通配，如下：

[=name]可以指定instruments名称，也可以使用通配符（前缀+通配符），也可以使用%代表所有的instruments

## performance\_schema实践操作

### 哪类的sql语句执行最多？

|  |
| --- |
| select DIGEST\_TEXT,COUNT\_STAR,FIRST\_SEEN,LAST\_SEEN from  events\_statements\_summary\_by\_digest ORDER BY COUNT\_STAR DESC |

### 哪些sql的平均响应时间最多？

|  |
| --- |
| select DIGEST\_TEXT,AVG\_TIMER\_WAIT from  events\_statements\_summary\_by\_digest ORDER BY AVG\_TIMER\_WAIT DESC |

### 哪些sql排序记录数最多

|  |
| --- |
| select DIGEST\_TEXT,SUM\_SORT\_ROWS from  events\_statements\_summary\_by\_digest ORDER BY SUM\_SORT\_ROWS DESC |

### 哪些sql扫描记录最多

|  |
| --- |
| select DIGEST\_TEXT,SUM\_ROWS\_EXAMINED from  events\_statements\_summary\_by\_digest ORDER BY  SUM\_ROWS\_EXAMINED DESC |

### 哪些sql使用临时表最多？

|  |
| --- |
| SELECT  DIGEST\_TEXT,  SUM\_CREATED\_TMP\_TABLES,  SUM\_CREATED\_TMP\_DISK\_TABLES  FROM  events\_statements\_summary\_by\_digest  ORDER BY  SUM\_CREATED\_TMP\_TABLES DESC |

### 哪些表返回结果最多

|  |
| --- |
| SELECT  DIGEST\_TEXT,  SUM\_ROWS\_SENT  FROM  events\_statements\_summary\_by\_digest  ORDER BY  SUM\_ROWS\_SENT DESC |

### 那个表物理IO最多

|  |
| --- |
| SELECT  FILE\_NAME,  EVENT\_NAME,  SUM\_NUMBER\_OF\_BYTES\_READ,  SUM\_NUMBER\_OF\_BYTES\_WRITE SUM\_ROWS\_SENT  FROM  file\_summary\_by\_instance  ORDER BY  SUM\_NUMBER\_OF\_BYTES\_READ + SUM\_NUMBER\_OF\_BYTES\_WRITE DESC |

### 那个表逻辑IO最多？

|  |
| --- |
| SELECT  OBJECT\_NAME,  COUNT\_READ,  COUNT\_WRITE,  COUNT\_FETCH,  SUM\_TIMER\_WAIT  FROM  table\_io\_waits\_summary\_by\_table  ORDER BY  SUM\_TIMER\_WAIT DESC |

### 那个索引访问最多？

|  |
| --- |
| SELECT  OBJECT\_NAME,  INDEX\_NAME,  COUNT\_FETCH,  COUNT\_INSERT,  COUNT\_UPDATE,  COUNT\_DELETE  FROM  table\_io\_waits\_summary\_by\_index\_usage  ORDER BY  SUM\_TIMER\_WAIT DESC |

### 那个索引从来没有用过？

|  |
| --- |
| SELECT  OBJECT\_SCHEMA,OBJECT\_NAME,INDEX\_NAME  FROM  table\_io\_waits\_summary\_by\_index\_usage  WHERE  INDEX\_NAME IS NOT NULL  AND COUNT\_STAR = 0  AND OBJECT\_SCHEMA <> 'mysql'  ORDER BY OBJECT\_SCHEMA,OBJECT\_NAME |

### 那个事件消耗事件最多？

|  |
| --- |
| SELECT  EVENT\_NAME,COUNT\_STAR,SUM\_TIMER\_WAIT,AVG\_TIMER\_WAIT  FROM  events\_waits\_summary\_global\_by\_event\_name  WHERE  EVENT\_NAME != 'idle'  ORDER BY SUM\_TIMER\_WAIT DESC |

### 剖析某条sql的执行情况，包括statement信息,stage,wait信息

|  |
| --- |
| SELECT  EVENT\_ID,SQL\_TEXT  FROM  events\_statements\_history  WHERE  SQL\_TEXT LIKE '%count(\*)%' |

### 查看每个阶段的时间消耗

|  |
| --- |
| SELECT  EVENT\_ID,EVENT\_NAME,SOURCE,TIMER\_END - TIMER\_START  FROM  events\_stages\_history\_long  WHERE  NESTING\_EVENT\_ID = 1553 |

### 查看每个阶段的锁等待情况

|  |
| --- |
| SELECT  EVENT\_ID,EVENT\_NAME,SOURCE,TIMER\_WAIT,  OBJECT\_NAME,INDEX\_NAME,OPERATION,NESTING\_EVENT\_ID  FROM  events\_waits\_history\_long  WHERE  NESTING\_EVENT\_ID = 1553 |

# 查看mysql连接信息

官网地址：sql statements -> database administration -> show statements -> show processlist

|  |
| --- |
| show PROCESSLIST; |

## show processlist参数解释

|  |  |
| --- | --- |
| 列 | 描述 |
| **id** | **ID标识，要kill一个语句的时候很有用** |
| **user** | **当前连接用户** |
| **host** | **显示这个连接从哪个ip的哪个端口上发出** |
| **db** | **数据库名** |
| **command** | **连接状态，一般是休眠（sleep），查询（query），连接（connect）,是针对这次操作的概括性** |
| **time** | **连接持续时间，单位是秒** |
| **state** | **显示当前sql语句的状态，是针对command操作的具体执行状态（步骤）的描述** |
| **info** | **显示这个sql语句** |

## command详解

表格罗列主要的状态，详细请参考官网

<https://dev.mysql.com/doc/refman/5.7/en/thread-commands.html>

|  |  |
| --- | --- |
| 状态 | 描述 |
| **sleep** | **线程正在等待客户端发送新的请求** |
| **query** | **线程正在执行查询或正在将结果发送给客户端** |
| **locked** | **在mysql的服务层，该线程正在等待表锁** |
| **analyzing and statistics** | **线程正在收集存储引擎的统计信息，并生成查询的执行计划** |
| **coping to tmptable** | **线程正在执行查询，并且将其结果集赋值到一个临时表中** |
| **sorting result** | **线程正在对结果集进行排序** |
| **sending data** | **线程可能在多个状态之间传送数据，或在生成结果集或者向客户端返回结果** |

## state的具体含义

|  |  |
| --- | --- |
| 状态 | 描述 |
| **Checking table** | **正在检查数据表（这是自动的）。** |
| Closing tables | 正在将表中修改的数据刷新到磁盘中，同时正在关闭已经用完的表。这是一个很快的操作，如果不是这样的话，就应该确认磁盘空间是否已经满了或者磁盘是否正处于重负中。 |
| Connect Out | 复制从服务器正在连接主服务器。 |
| Copying to tmp table on disk | 由于临时结果集大于tmp\_table\_size，正在将临时表从内存存储转为磁盘存储以此节省内存。 |
| Creating tmp table | 正在创建临时表以存放部分查询结果。 |
| deleting from main table | 服务器正在执行多表删除中的第一部分，刚删除第一个表。 |
| deleting from reference tables | 服务器正在执行多表删除中的第二部分，正在删除其他表的记录。 |
| Flushing tables | 正在执行FLUSH TABLES，等待其他线程关闭数据表。 |
| Killed | 发送了一个kill请求给某线程，那么这个线程将会检查kill标志位，同时会放弃下一个kill请求。MySQL会在每次的主循环中检查kill标志位，不过有些情况下该线程可能会过一小段才能死掉。如果该线程程被其他线程锁住了，那么kill请求会在锁释放时马上生效。 |
| Locked | 被其他查询锁住了。 |
| Sending data | 正在处理SELECT查询的记录，同时正在把结果发送给客户端。 |
| Sorting for group | 正在为GROUP BY做排序。 |
| Sorting for order | 正在为ORDER BY做排序。 |
| Opening tables | 这个过程应该会很快，除非受到其他因素的干扰。例如，在执ALTER TABLE或LOCK TABLE语句行完以前，数据表无法被其他线程打开。正尝试打开一个表。 |
| Removing duplicates | 正在执行一个SELECT DISTINCT方式的查询，但是MySQL无法在前一个阶段优化掉那些重复的记录。因此，MySQL需要再次去掉重复的记录，然后再把结果发送给客户端。 |
| Reopen table | 获得了对一个表的锁，但是必须在表结构修改之后才能获得这个锁。已经释放锁，关闭数据表，正尝试重新打开数据表。 |
| Repair by sorting | 修复指令正在排序以创建索引。 |
| Repair with keycache | 修复指令正在利用索引缓存一个一个地创建新索引。它会比Repair by sorting慢些。 |
| Searching rows for update | 正在讲符合条件的记录找出来以备更新。它必须在UPDATE要修改相关的记录之前就完成了。 |
| Sleeping | 正在等待客户端发送新请求. |
| System lock | 正在等待取得一个外部的系统锁。如果当前没有运行多个mysqld服务器同时请求同一个表，那么可以通过增加--skip-external-locking参数来禁止外部系统锁。 |
| Upgrading lock | INSERT DELAYED正在尝试取得一个锁表以插入新记录。 |
| Updating | 正在搜索匹配的记录，并且修改它们。 |
| User Lock | 正在等待GET\_LOCK()。 |
| Waiting for tables | 该线程得到通知，数据表结构已经被修改了，需要重新打开数据表以取得新的结构。然后，为了能的重新打开数据表，必须等到所有其他线程关闭这个表。以下几种情况下会产生这个通知：FLUSH TABLES tbl\_name, ALTER TABLE, RENAME TABLE, REPAIR TABLE, ANALYZE TABLE,或OPTIMIZE TABLE。 |
| waiting for handler insert | INSERT DELAYED已经处理完了所有待处理的插入操作，正在等待新的请求。 |

## show status 详解

# schema与数据库类型优化

## 数据类型的优化

### 更小的通常更好

1. 在设计表时，应该尽量的确认数据的长度，用合适的数据理性，例如：存储性别：0/1，在mysql中最好使用TINYINT，因为在mysql中TINYINT占用1 byte，而INT占用4 bytes。具体数据类型如下：

[**https://www.runoob.com/mysql/mysql-data-types.html**](https://www.runoob.com/mysql/mysql-data-types.html)

**结论：测试2000条数据，设计2张表，但是字段类型不一样，插入同样的数据，结果是.ibd文件一样大，但是占用的空间不一样大。**

1. .frm和.ibd（innodb）

.frm表示创建表的表结构

.ibd表示表的数据文件

### 简单就好

* **注释：**

**简单的数据类型的操作通常需要更少的CPU周期，例如：**

1. **整型比字符串操作代价更低，因为字符集和校对规则是字符集比较比整型更复杂**
2. **使用mysql自建类型而不是字符串来存储日期和时间**
3. **用整型存储IP地址**

* **案例1：**

创建两张相同的表，改变日期的数据类型，查看SQL语句的执行速度

select \* from emp where hiredate='1980-12-17';

select \* from emp2 where hiredate='1980-12-17';

通过show profile，查看总执行时间：

* **案例2：**

mysql内置函数IP转数值

select inet\_aton('10.4.123.38');

mysql内置函数数值转IP

select inet\_ntoa('168065830');

### 尽量避免使用null

* **注释：**

如果查询中包含可为null的列，对mysql来说很难优化，因为可为null的列使得索引、索引统计和值比较都更加复杂，坦白来说，通常情况下null的列改为not null带来的性能提升比较小，所以没有必要将所有的表的schema进行修改，但是应该尽量避免设计成为null的列，例如默认值是‘’是可以的

### 实施细则

#### 整数类型：

可以使用的几种整数类型：TINYINT、SMALLINT、MEDIUMINT、INT或INTEGER、BIGINT，分别使用8、16、24、32、64位存储空间。尽量使用满足需求的最小数据类型。

#### 字符串类型：

varchar根据实际内容长度保存数据

操作系统存储空间是按照4K(页)块进行存储的，例如：我有一个5.0K的数据存储在硬盘中，会显示，实际大小是5.0K,占用空间是8.0K

1. 使用最小的符合需求的长度
2. varchar(n) n 小于等于255使用额外一个字节报错长度，n>255使用额外2个字节保存长度
3. varchar(5)与varchar(255)保存同样的内容，硬盘存储空间相同，但内存空间占用不同，是指定的大小（**这对于排序或者临时表，这些内容都需要通过内存来实现作业会产生比较大的不利影响，所以如果某些字段会涉及到文件排序或者基于磁盘的临时表时，分配VARCHAR数据类型时仍然不能够太过于慷慨。还是要评估实际需要的长度，然后选择一个最长的字段来设置字符长度。如果为了考虑冗余，可以留10%左右的字符长度。**）

**例如：**MySQL建立索引时如果没有限制索引的大小，索引长度会默认采用的该字段的长度，也就是说varchar(100)建立的索引存储大小要比varchar(10)建立索引存储大小大的多，加载索引使用的内存也更多。

1. varchar在mysql5.6之前变更长度，或者从255以下变更到255以上时，都会导致锁表
2. 应用场景：

* 存储长度波动较大的数据，如：文章，有的会很短，有的会很长；
* 字符串很少更新的场景，每次更新都会重新计算并使用额外存储空间保存长度
* 适合保存多字节字符，如：汉字、特殊字符等

#### 字符类型

char固定长度的字符串

1. 最大长度：255
2. 会自动删除末尾的空格（查询）
3. 检索效率、写效率会比varchar高，以空间换时间
4. 应用场景：

* 存储长度波动不大的数据，如md5摘要
* 存储短字符串，经常更新的字符串

#### BLOB和TEXT

MySQL把每个BLOB和TEXT值当做一个独立的对象处理，两者都是为了存储很大的数据而设计的字符串类型，分别采用二进制和字符方式存储。一般不会用，不适合大数量存储

#### datetime

1. 占用8个字节
2. 与时区无关、数据底层时区配置，对datetime无效
3. 可保存到毫秒
4. 可存储时间范围大
5. 不要使用字符串存储日期类型，占用空间大，损失日期类型的函数便捷性

#### timestamp（较多，可以跨时区）

1. 占用4个字节
2. 时间范围：1970-01-01到2038-01-19
3. 精确到秒
4. 采用整型存储
5. 依赖数据设置的时区
6. 自动更新timestamp

#### date

1. 占用的字节比使用字符串、datetime、int存储要少，只需3个字节
2. 使用date类型还可以利用日期时间函数进行日期之间的计算
3. date类型用于保存1000-01-01到9999-12-31之间的日期

#### 枚举

有时可以使用枚举代替常用的字符串类型，mysql存储枚举类型会非常紧凑，会根据列表值得数据压缩到一个或者2个字节中，mysql在内部会将每个值在列表中位置保存为整数，并且在表的.frm文件中保存“数字-字符串”的映射关系的查找表

|  |
| --- |
| create table enum\_test(e enum('fish','apple','dog') not null);  insert into enum\_test(e) values('fish'),('dog'),('apple');  select \* from enum\_test where e=1; |

#### 特殊类型数据

人们经常使用varchar(15)来存储ip地址，然而，它的被子是32位无符号整数，不是字符串，可以使用INET\_ATON()和INET\_NTOA函数在这2中表示之间转换

|  |
| --- |
| mysql内置函数IP转数值  select inet\_aton('10.4.123.38');  mysql内置函数数值转IP  select inet\_ntoa('168065830'); |

## 合理使用范式和反范式

* **注释：**

**三范式是为了解决数据冗余的**

## 适当的拆分

当我们的表中存在类似于 TEXT 或者是很大的 VARCHAR类型的大字段的时候，如果我们大部分访问这张表的时候都不需要这个字段，我们就该义无反顾的将其拆分到另外的独立表中，以减少常用数据所占用的存储空间。这样做的一个明显好处就是每个数据块中可以存储的数据条数可以大大增加，既减少物理 IO 次数，也能大大提高内存中的缓存命中率。

**分库、分表：**

* **垂直拆分**
* **水平拆分**

### 范式

#### 优点

1. 范式化的更新通常比反范式要快
2. 当数据较好的范式化后，很少或者没有重复的数据
3. 范式化的数据比较小，可以放在内存中，操作比较快

#### 缺点

1. 通常需要进行关联

### 反范式

#### 优点

1. 所有的数据都在同一张表中，可以避免关联
2. 可以设计有效的索引；

#### 缺点

1. 表格内的冗余较多，删除数据时候会造成表有些有用的信息丢失

### 范式和反范式应用举例

1. 在一个网站实例中，这个网站，允许用户发送消息，并且一些用户是付费用户。现在想查看付费用户最近的10条信息。  在user表和message表中都存储用户类型(account\_type)而不用完全的反范式化。这避免了完全反范式化的插入和删除问题，因为即使没有消息的时候也绝不会丢失用户的信息。这样也不会把user\_message表搞得太大，有利于高效地获取数据。
2. 另一个从父表冗余一些数据到子表的理由是排序的需要。
3. 缓存衍生值也是有用的。如果需要显示每个用户发了多少消息（类似论坛的），可以每次执行一个昂贵的自查询来计算并显示它；也可以在user表中建一个num\_messages列，每当用户发新消息时更新这个值。
4. 用户、订单、商品、订单商品表

**范式：**

****

**反范式：**

****

## 主键的选择

### 代理主键

与业务无关的，无意义的数字序列

### 自然主键

事物属性中的自然唯一标识

### 推荐使用代理主键

1. 它们不与业务耦合，因此更容易维护
2. 一个大多数表，最好是全部表，通用的键策略能够减少需要编写的源码数量，减少系统的总体拥有成本

## 字符集的选择

**通常我们在存储中文的时候，我们一般用utf-8 mb4**

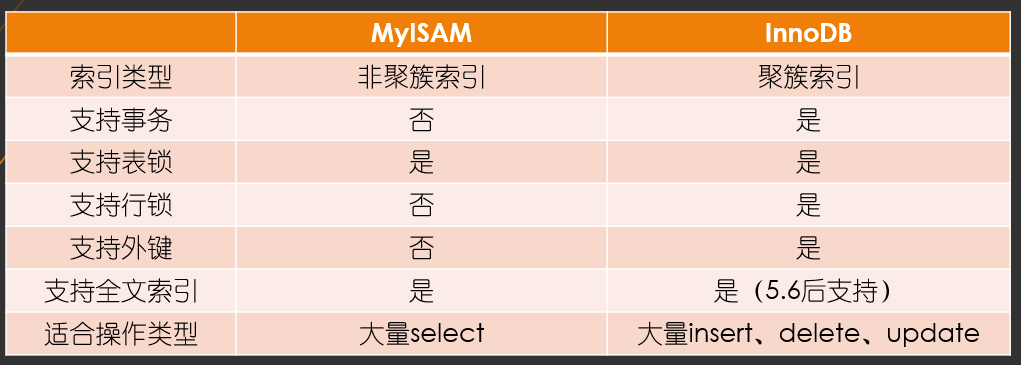
**字符集直接决定了数据在MySQL中的存储编码方式，由于同样的内容使用不同字符集表示所占用的空间大小会有较大的差异，所以通过使用合适的字符集，可以帮助我们尽可能减少数据量，进而减少IO操作次数。**

1. 纯拉丁字符能表示的内容，没必要选择 latin1 之外的其他字符编码，因为这会节省大量的存储空间。
2. 如果我们可以确定不需要存放多种语言，就没必要非得使用UTF8或者其他UNICODE字符类型，这回造成大量的存储空间浪费。
3. MySQL的数据类型可以精确到字段，所以当我们需要大型数据库中存放多字节数据的时候，可以通过对不同表不同字段使用不同的数据类型来较大程度减小数据存储量，进而降低 IO 操作次数并提高缓存命中率。

## 存储引擎的选择

**因此在做读写分离时，读库，应该使用MyISAM存储引擎**

**存储引擎：数据文件的组织形式**



## 适当的冗余

1. 被频繁引用且只能通过 Join 2张(或者更多)大表的方式才能得到的独立小字段。
2. 这样的场景由于每次Join仅仅只是为了取得某个小字段的值，Join到的记录又大，会造成大量不必要的 IO，完全可以通过空间换取时间的方式来优化。不过，冗余的同时需要确保数据的一致性不会遭到破坏，确保更新的同时冗余字段也被更新。

# 执行计划

直接看.md文档



# 通过索引优化

## 索引的基本概念

1. **mysql数据存储的本质**

其实mysql的数据本质上是存储在物理服务器上的具体的文件里，如果要查找某个数据，通常情况下，我们可以遍历服务器上的每个文件中的每个字符，进行对比，来查找自己的数据。而为了加快查找速度，我们最好的办法，就是要知道我们的数据存放在那个文件里，并且在这个文件中的offset(偏移量)，然后通过seek跳过偏移量，直接查找出对应的数据，这些信息就是存储在索引中。因此使用索引，是能大幅度提高查询效率的。

1. **索引解决的问题**

通过（1）提出快速查找数据的需求，我们认为索引应该解决的问题是：1.**通过索引能快速知道数据在那个文件中；2.通过索引我们知道数据这个文件中的偏移量。**

1. **索引的数据结构选择：**
   1. **hash表的索引格式：**

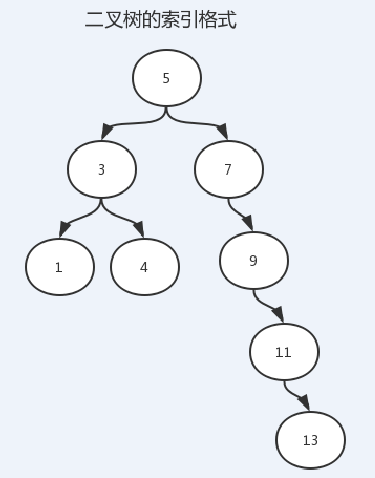
**释义：**hash表的存储，是基于数据的存储的，通过对hashcode的取模运算，存入到对应的下表的数据中，如果取模值相同，值进行链表存储。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |

**缺点：**

1. **利用hash存储的话需要将所有的索引数据加载到内存中，比较耗费内存（memory存储引擎，就是hash）**
2. **如果所有的查询都是等值查询，那个hash确实很快，但是在企业或者实际工作中范围查询的数据更多，而不是等值查询，因此hash不合适。**
   1. **二叉树的索引格式**

**释义：二叉树要插入子节点时，会判断值是否大于自己本身的值，如果大于则插入右边的节点，否则插入左侧的节点。**



**缺点：会因为数的深度过深而造成io次数变多，影响数据读取的效率**

* 1. **平衡二叉树**

**释义：平衡二叉树是为了解决子树长度过长，而导致的io次数变多的问题，因此平衡二叉树的定义：最短的子树与最长的子树的高度差不会超过1，否则会进行旋转调整（左旋、右旋）**

**缺点：轮转调整会非常耗时，因此在索引在插入、删除时，会消耗效率，同时树也会很深。**

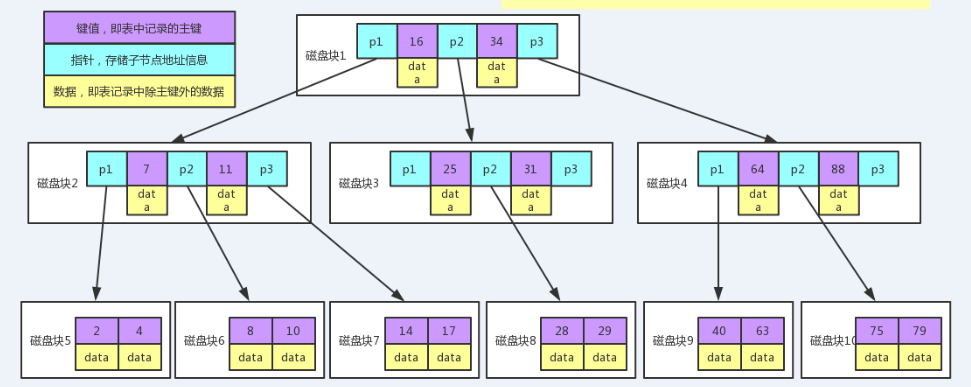
* 1. **红黑树**

**释义：红黑树是avl数的一个变种，通过颜色控制轮转次数，从而平衡插入、删除的效率，而损失了查询效率，红黑树满足2个条件：**

1. **最长子树的最大长度 不能 超过 最短 子树长度的 2倍**
2. **最长子树的黑树的个数，与最短子树的 黑树个数相同**

**缺点：无论是红黑树或者平衡二叉树，最终都有2个分支，从而会导致树的长度过深，导致io次数变多**

* 1. **B树**

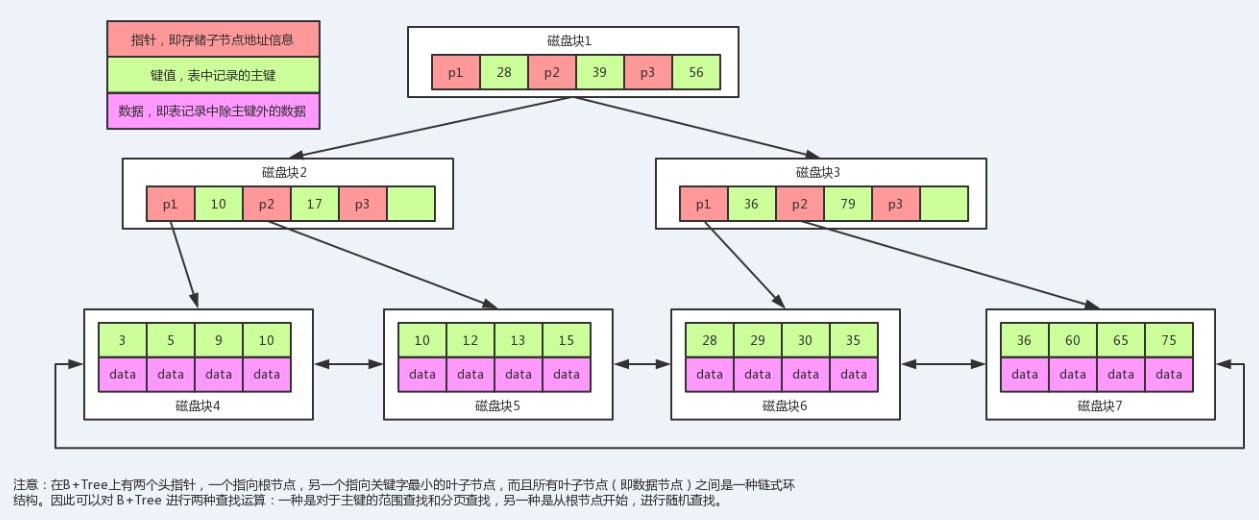


**释义：每个磁盘块都会存储许多key值，形成一个一个的区间段，并且key节点是包含【键值、指向子节点的地址信息的指针、数据，即表中除主键的其他数据】**

**缺点：因为key节点中存储的数据信息，导致每个磁盘块存储的key值得数量会减少，因此导致树的深度加深，导致io次数增加；**

**有点：有肯能在非叶子节点，查找出想要的数据**

* 1. **B+树**



**三层的b+tree可以支持千万级别的数据查找**

**释义：B+树是在B树的基础上做了一种优化，变化如下：**

1. **B+Tree每个节点可以包含更多的节点（key）这样做的原因有两个，第一是为了降低数的高度，第二是将数据范围变为更多个区间，区间越多，数据检索越快**
2. **非叶子节点至存储key(主键值、子节点的地址信息/指针)，叶子节点存储key和数据**
3. **叶子节点两两指针相互连接（符合磁盘的预读性），顺序查询性能更高**

## InnoDB 和 myISAM索引存储区别

InnoDB:索引数据与数据库数据时存储在同一文件中，即B+Tree的叶子节点就是数据库表中对应的一条数据。

MyISAM:索引数据与数据库数据分别存储在2个文件中，即B+Tree的叶子节点是存储的数据的物理磁盘的地址。

## 索引的优点

1. 大大减少了服务器需要扫描的数据量;
2. 帮助服务器避免排序和临时表;

在mysql中：（1）若查询条件中没有走索引，或者是走的普通索引，结果都会按照id主键从小到大排序；（2）若查询条件走联合索引，则结果会按照联合索引的从左到右的字段先后顺序排序。

1. 将随机io变成顺序io.

## 索引的用处

1. 快速查找匹配WHERE子句的行。
2. 从consideration中消除行,如果可以在多个索引之间进行选择，mysql通常会使用找到最少行的索引;
3. 如果表具有多列索引，则优化器可以使用索引的任何最左前缀来查找行;
4. 当有表连接的时候，从其他表检索行数据
5. 查找特定索引列的min或max值;
6. 如果排序或分组时在可用索引的最左前缀上完成的，则对表进行排序和分组
7. 在某些情况下，可以优化查询以检索值而无需查询数据行

## 索引的分类

1. 主键索引
2. 唯一索引
3. 普通索引
4. 全文索引
5. 组合索引

## 索引匹配方式

**注意：官网中，提供一些演示数据，我们根据这些数据操作官网给的例子**

**mysql官网 –> documentaion –> mysql server –> mysql 5.7 reference manual –> more –> example databases**

**具体地址：**<https://dev.mysql.com/doc/index-other.html>

### 全值匹配

**释义：全值匹配指的是和索引中的所有列进行匹配**

**案例：staffs有联合索引：index idx\_nap` (`name`,`age`,`pos`)**

**执行计划：**

**explain select \* from staffs where name = 'July' and age = '23' and pos = 'dev';**

### 匹配最左前缀

释义：只匹配前面的几列

执行计划：**staffs有联合索引：index idx\_nap` (`name`,`age`,`pos`)**

explain select \* from staffs where name = 'July' and age = '23';

explain select \* from staffs where name = 'July';

### 匹配列前缀

释义：可以匹配某一列的值的开头部分

执行计划：**staffs有联合索引：index idx\_nap` (`name`,`age`,`pos`)**

explain select \* from staffs where name like 'J%';

**注意：模糊查询时，%在前面是无法命中索引，因为违反了匹配列前缀的原则**

explain select \* from staffs where name like '%y';

### 匹配范围值

释义：可以查找某一个范围的数据

执行计划：**staffs有联合索引：index idx\_nap` (`name`,`age`,`pos`)**

explain select \* from staffs where name > 'Mary';

### 精确匹配某一列并范围匹配另外一列

释义：可以查询第一列的全部和第二列的部分，但是无法同时匹配1、3列

执行计划：**staffs有联合索引：index idx\_nap` (`name`,`age`,`pos`)**

explain select \* from staffs where name = 'July' and age > 25;

解释：因为执行计划type range,并且ref null

执行计划：**staffs有联合索引：index idx\_nap` (`name`,`age`,`pos`)**

explain select \* from staffs where name = 'July' and pos > 25;

解释：此时，执行计划type ref，并且ref const(常量)，所以只命中name索引

### 只访问索引的查询

释义：查询的时候只需要访问索引，不需要访问数据行，本质上就是覆盖索引

执行计划：**staffs有联合索引：index idx\_nap` (`name`,`age`,`pos`)**

explain select name,age,pos from staffs where name = 'July' and age = 25 and pos = 'dev';

解释：执行计划中extra Using index,即使用了索引覆盖。

## 哈希索引

### 基本知识

1. 基于哈希表的实现，只有精确匹配索引所有列的查询才有效
2. 在mysql中，只有memory的存储引擎显式支持哈希索引
3. 哈希索引自身只需存储对应的hash值，所以索引的结构十分紧凑，这让哈希索引查找的速度非常快

### 哈希索引限制

1. 哈希索引只包含哈希值和行指针，而不存储字段值，索引不能使用索引中的值来避免读取行
2. 哈希索引数据并不是按照索引值顺序存储的，所以无法进行排序
3. 哈希索引不支持部分列匹配查找，哈希索引是使用索引列的全部内容来计算哈希值
4. 哈希索引支持等值比较查询，也不支持任何范围查询
5. 访问哈希索引的数据非常快，除非有很多哈希冲突，当出现哈希冲突的时候，存储引擎必须遍历链表中的所有行指针，逐行进行比较，直到找到所有符合条件的行
6. 哈希冲突比较多的话，维护的代价也会很高

### 注意：hash冲突

**hash冲突**：如果是hash算法比较简单，就是取模运算（%）,那么key值只能低位的后四位参与运算，这样取模的值，会有可能大量重复，导致存入数据中同一位置，最后此位置就会形成很长的链表。这就是hash冲突；

**解决思路：**编写优秀的hash算法，例如hashMap的hash算法，使用了扰动函数，目的就是让key的高位也参加运算。

### 案例

**名称解释：CRC** [循环冗余校验](https://baike.baidu.com/item/%E5%BE%AA%E7%8E%AF%E5%86%97%E4%BD%99%E6%A0%A1%E9%AA%8C/3219009),CRC32，32位循环冗余校验码

**案例描述：**

当需要存储大量的URL，并且根据URL进行搜索查找，如果使用B+树，存储的内容就会很大

**解决方案**：使用mysql的CRC32将url生成对应的整数进行存储，并为此字段创建索引，这样整数类型的所有效率要比长字符串类型的索引高

select id from url where url=""

也可以利用将url使用CRC32做哈希，可以使用以下查询方式：

select id fom url where url="" and url\_crc=CRC32("")

此查询性能较高原因是使用体积很小的索引来完成查找

## 组合索引

**释义：**当包含多个列作为索引，需要注意的是正确的顺序依赖于该索引的查询，同时需要考虑如何更好的满足排序和分组的需要

**案例：建立组合索引a,b,c**



图中最后一行，错了

## 聚簇索引与非聚簇索引

### 聚簇索引

**释义：**不是单独的索引类型，而是一种数据存储方式，指的是数据行跟相邻的键值紧凑的存储在一起（在B+Tree中，叶子节点存储的是key值、data），我们**可以这样判断：索引跟数据存储在一个文件中**

**优点：**

1. 可以把相关数据保存在一起；
2. 数据访问更快，因为索引和数据保存在同一个树中；
3. 使用覆盖索引扫描的查询可以直接使用页节点中的主键值

**缺点：**

1. 聚簇数据最大限度地提高了IO密集型应用的性能，如果数据全部在内存，那么聚簇索引就没有什么优势
2. 插入速度严重依赖于插入顺序，按照主键的顺序插入是最快的方式
3. 更新聚簇索引列的代价很高，因为会强制将每个被更新的行移动到新的位置
4. 基于聚簇索引的表在插入新行，或者主键被更新导致需要移动行的时候，可能面临页分裂的问题
5. 聚簇索引可能导致全表扫描变慢，尤其是行比较稀疏，或者由于页分裂导致数据存储不连续的时候

**相关名词解释**

**页分裂：**因为每个磁盘块存储的空间是一定的，是提前分配好的。如果在insert数据时，没有按照索引的顺序进行插入，则有可能将插入的数据放到已经存储满的磁盘块中，这样，就会将原来的磁盘块，分裂成2个跟原来一样的磁盘块，并且增加了一层 B+Tree。会影响查询效率。

**页合并：**当数据删除时，如果发现原来有经过页分裂形成的多个磁盘块，则进行磁盘块合并

### 非聚簇索引

**释义：**数据文件跟索引文件分开存放

### 关于数据迁移索引处理的案例

**场景描述：**

公司需要大数量迁移至新的数据库。

**原因分析：**

因为B+Tree的索引的特点，会导致索引改动非常麻烦、耗时

**解决方案：**

在导入数据前，关闭表的索引，导入成功后，在打开索引

关闭：ALTER TABLE `test` DISABLE KEYS ;

开启：ALTER TABLE `test` ENABLE KEYS;

## 覆盖索引

### 基本介绍

1. 如果一个索引包含所有需要查询的字段的值，我们称之为覆盖索引
2. 不是所有类型的索引都可以称为覆盖索引，覆盖索引必须要存储索引列的值
3. 不同的存储实现覆盖索引的方式不同，不是所有的引擎都支持覆盖索引，memory不支持覆盖索引

### 优势

1. 索引条目通常远小于数据行大小，如果只需要读取索引，那么mysql就会极大的较少数据访问量（减少IO次数）
2. 因为索引是按照列值顺序存储的，所以对于IO密集型的范围查询会比随机从磁盘读取每一行数据的IO要少的多（IO密集型，操作IO频繁）
3. 一些存储引擎如MYISAM在内存中只缓存索引，数据则依赖于操作系统来缓存，因此要访问数据需要一次系统调用，这可能会导致严重的性能问题
4. 由于INNODB的聚簇索引，覆盖索引对INNODB表特别有用

### 案例



## 优化小细节

### 尽量不适用表达式当where条件

当使用索引列进行查询的时候尽量不要使用表达式，把计算放到业务层而不是数据库层

### 尽量使用主键查询

尽量使用主键查询，而不是其他索引，因此主键查询不会触发回表查询

### 使用前缀索引：

有时候需要索引很长的字符串，这会让索引变的大且慢，通常情况下可以使用某个列开始的部分字符串，这样大大的节约索引空间，从而提高索引效率，但这会降低索引的选择性，**索引的选择性**是指不重复的索引值和数据表记录总数的比值，范围从1/#T到1之间。索引的选择性越高则查询效率越高，因为选择性更高的索引可以让mysql在查找的时候过滤掉更多的行。

一般情况下某个列前缀的选择性也是足够高的，足以满足查询的性能，**但是对应BLOB,TEXT,VARCHAR类型的列**，必须要使用前缀索引，因为mysql不允许索引这些列的完整长度，使用该方法的诀窍在于要选择足够长的前缀以保证较高的选择性，通过又不能太长。

案例：

文档优化思路：如果索引字段特别长，那么我们可以通过将索引字段前缀进行分组，与原来做对比，找出列前缀分组的差异性与源字段的差异性基本差不多，就可以使用前缀做索引。

--还可以通过另外一种方式来计算完整列的选择性，可以看到当前缀长度到达7之后，再增加前缀长度，选择性提升的幅度已经很小了

select count(distinct left(city,3))/count(\*) as sel3,

count(distinct left(city,4))/count(\*) as sel4,

count(distinct left(city,5))/count(\*) as sel5,

count(distinct left(city,6))/count(\*) as sel6,

count(distinct left(city,7))/count(\*) as sel7,

count(distinct left(city,8))/count(\*) as sel8

from citydemo;

--计算完成之后可以创建前缀索引

alter table citydemo add key(city(7));

**名字解释：OLAP系统，例如：电商系统，存储10年的历史数据，我要对历史数据，进行统计分析，以便为决策提供数据依据。**

**名词解释：show index from cityDemo;查询结果会出现一个属性字段Cardinality(基数)，表示在数据表中，这个索引大概有几个不重复的值。**

### 使用索引扫描来排序

**排序的效率问题：**如果查看执行计划（extra）出现filesort（文件排序），则mysql会将查询的记录全部加载到内存中，进行指定字段的排序，会消耗效率；因为索引本来就是有顺序的，如果使用索引进行排序，效率会很快，但是如果在查询过程中使用了索引覆盖，则效率会更快，因为本身不需要回表查询了。

**索引失效：**

**注意：有可能表中已经做了索引，并且查询的时候使用 age > 20 范围查询，如果执行计划显示没使用索引排序，则有可能rows(预估需要扫描的行)太大了。**

1. 如果表中是一个联合索引，并且查询的第一个条件是 范围查询，则后面的where条件将不会用到索引，排序也不会用到。
2. 如果排序字段是多个，并且排序的方式不同，则无法用到索引扫描。
3. **如果查询需要关联多张表，则只有当orderby子句引用的字段全部为第一张表时，才能使用索引做排序。**
4. ****

### 推荐使用in

union all,in,or都能够使用索引，但是推荐使用in

### exist用法：

内部的子查询true,则外部查询就会保留此条记录。注意，外部查询不能使用exists的子查询的字段。

select \* from emp e where exists(select deptno from dept d where(deptno = 20 or deptno = 30) and e.deptno = d.deptno);

### 范围列可以用到索引

范围条件是：<、<=、>、>=、between；

范围列可以用到索引，但是范围列后面的列无法用到索引，索引最多用于一个范围列。

### 强制类型转换会全表扫描

例如：

//phone vachar(),因此不会使用索引

explain select \* from user where phone=13800001234;

//会使用索引

explain select \* from user where phone='13800001234';

### 索引列选择

1. 尽量使用长度较小的字段当索引列，否则可以使用列前缀做索引
2. 更新十分频繁，数据区分度不高的字段上不宜建立索引：原因如下：

* 更新会变更B+树，更新频繁的字段建议索引会大大降低数据库性能；
* 类似于性别这类区分不大的属性，建立索引是没有意义的，不能有效的过滤数据；
* 一般区分度在80%以上的时候就可以建立索引，区分度可以使用 **count(distinct(列名))/count(\*) 来计算**。

1. 创建索引的列，不允许为null，可能会得到不符合预期的结果
2. 索引列最好能被用到排序，并且最好能被用到覆盖索引。

### join表时索引要求

当需要进行表连接的时候，最好不要超过三张表，因为需要join的字段，数据类型必须一致。

最好的方式：小表为驱动表，大表为非驱动表，且关联字段在大表中是主键索引。

### 能使用limit的时候尽量使用limit

mysql中使用条件查询时，他会对数据块中的数据遍历比较是否满足where条件，如果你明确知道自己的要查询出的数据条数。就使用limit，会省略掉多余的判断

### 单表索引建议控制在5个以内

根据官网说明：

下表通过显示将各种字符串值存储到CHAR（4）和VARCHAR（4）列中的结果（假设列使用单字节字符集，如latin1），说明了CHAR和VARCHAR之间的区别。

**因此，如果索引数过多，导致索引文件变大，io次数变多。还会增加插入和修改的维护成本。**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Value** | **CHAR(4)** | **Storage Required** | **VARCHAR(4)** | **Storage Required** |
| '' | '    ' | 4 bytes | '' | 1 byte |
| 'ab' | 'ab  ' | 4 bytes | 'ab' | 3 bytes |
| 'abcd' | 'abcd' | 4 bytes | 'abcd' | 5 bytes |
| 'abcdefgh' | 'abcd' | 4 bytes | 'abcd' | 5 bytes |

### 单索引字段数不允许超过5个（组合索引）

因为：组合索引，有最左匹配原则，如果字段个数过多，会导致查询时where条件限制很多。并且也会增加插入、删除、修改的维护成本。

### 创建索引的时候应该避免以下错误概念

1. 索引越多越好
2. 过早优化，在不了解系统的情况下进行优化

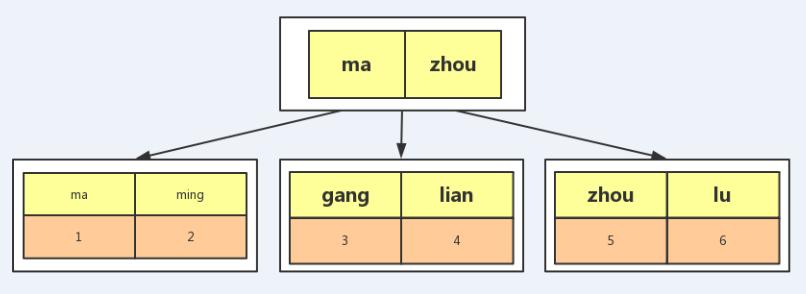
## 面试技术名词

### 回表

**注意：**在一张表中，最好创建主键，如果不创建，mysql也会自己默认创建6位的主键，索引一般会默认使用主键形成B+Tree;

**回表：在一张表中，我为普通列创建了普通索引，此时这个索引的B+Tree如下：**

**即叶子节点是存储是主键，最后查找出主键，通过主键的B+Tree来查找最后的数据。**



### 覆盖索引

就是你要查找的值，正好是索引的列，即不用查找叶子节点，直接返回查询结果，这就是覆盖索引。

第一种情况：你直接用主键当where条件，直接查找主键B+Tree,返回值

第二种情况：你用普通索引当where条件，直接在普通索引的叶子节点就可以返回查询结果，不用回表了。

### 最左匹配

如果在实际的业务中，发现某几个字段，经常联合出现进行查询，则考虑需要创建联合索引，例如：将name、age创建联合索引，查询语句如下：

select \* from emp where name = ? and age = ?,则会使用联合索引

select \* from emp where name = ? 则会使用name索引

select \* from emp where age = ? 则不会命中任何索引

**案例：**

select \* from emp where name = ? and age = ?

select \* from emp where name = ?

select \* from emp where age = ?

**针对上面的查询语句，我们应该创建如下索引：**

**name、age的联合索引，age的单列索引，因为相比较name,age的占用磁盘空间会更小，有可能减少io索引次数。**

### 索引下推

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

**博客地址：https://www.cnblogs.com/Chenjiabing/p/12600926.html**

联合索引【name、age】

查询语句：select \* from emp where name='' and age = '';

老版本的mysql: 先从B+Tree中取出索引的符合name的数据，然后返回server层，在进行age过滤。

新版本的mysql:使用索引下推，在B+Tree取出数据的时候，直接进行age过滤，这样IO检索的数据量就会减少，在给server层，就会提高效率

### 谈谈sql优化的经验（自己总结）

优化思路：

1. 首先排除外界因素，比如说,是否网络延迟，查询耗时是不是每次都是那么慢？
2. 查找查询慢的原因：
3. 查看sql执行计划，**如果是简单查询**，看看type是不是ALL,如果是ALL,统计查询条件添加适合的索引；**如果复查查询**，先根据id的规则，从内层向外层**拆分成简单的sql**进行优化，**然后再看具体的查询类型，如果是union / union all ，是否能转换成in查询，如果是关联查询，看看关联条件是否使用索引，是否驱动表的关联行数进行限制，如果这些都不行，简单计算下关联字段总大小是否超过mysql的join\_buffer\_size大小，能不能调整join\_buffer\_size大小，来优化关联效率。另外如果索引没有问题，看看索引字段的长度、基数、是否有异常的情况，如果长度特别长，看看能不能使用前缀进行索引匹配。另外看看执行计划中extra信息，是否是文件排序等等**
4. 查看mysql的连接信息 show processlist

主要是关注command的信息以及state的信息，看看这次查询操作，那什么地方消耗的大小时间。如果是在表锁这块消耗了大量时间，那就查看这张表的update操作是否合规，是否使用索引进行更新等等。

1. 如果这些都没有问题，那就看看mysql物理服务器的内存使用情况咋样，是不是表数据量太大导致io检索太慢，是否物理内存不够导致io检查慢等等其他原因吧

# 索引监控（暂时没啥用，后续补充）

执行命令：

show status like 'Handler\_read%';

参数解释：

|  |  |  |
| --- | --- | --- |
| **参数** | **解释** | **备注** |
| Handler\_read\_first | 读取索引第一个条目的次数 |  |
| Handler\_read\_key | 通过index获取数据的次数 | 关注 |
| Handler\_read\_last | 读取索引最后一个条目的次数 |  |
| Handler\_read\_next | 通过索引读取下一条数据的次数 |  |
| Handler\_read\_prev | 通过索引读取上一条数据的次数 |  |
| Handler\_read\_rnd | 从固定位置读取数据的次数 |  |
| Handler\_read\_rnd\_next | 从数据节点读取下一条数据的次数 | 关注 |

# 索引优化案例



# 查询优化

## 定义

在编写快速的查询之前，需要清楚一点，真正重要的是响应时间，而且要知道在整个SQL语句的执行过程中每个步骤都花费了多长时间，要知道哪些步骤是拖垮执行效率的关键步骤，想要做到这点，必须要知道查询的生命周期，然后进行优化，不同的应用场景有不同的优化方式，不要一概而论，具体情况具体分析，

可以借助：show PROFILE显示各部分执行的消耗的时间。

**查询优化的步骤：查询慢的原因、优化数据访问、执行过程优化、优化特定类型的查询，以下会详细讲解**

## 查询慢的原因

1. 网络
2. CPU
3. IO
4. 上下文切换
5. 系统调用
6. 生成统计信息
7. 锁等待时间

## 优化数据访问

* **查询性能低下的主要原因是访问的数据太多，某些查询不可避免的需要筛选大量的数据，我们可以通过减少访问数据量的方式进行优化**：

1. 确认应用程序是否在检索大量超过需要的数据
2. 确认mysql服务器层是否在分析大量超过需要的数据行
3. 案例：在sakila数据中表rental有16044条数据，我想查询16000-16005这5条数据。

explain select \* from rental limit 10000,5;

在执行计划中，发现rows=16008,这明显不合理，需要优化。

* **是否向数据库请求了不需要的数据**

1. **查询不需要的记录：**

我们常常会误以为mysql会只返回需要的数据，实际上mysql却是先返回全部结果再进行计算，在日常的开发习惯中，经常是先用select语句查询大量的结果，然后获取前面的N行后关闭结果集。

优化方式是在查询后面添加limit

1. **多表关联时返回全部列**
2. **总是取出全部列**

在公司的企业需求中，禁止使用select \*,虽然这种方式能够简化开发，但是会影响查询的性能，所以尽量不要使用

1. **重复查询相同的数据**

如果需要不断的重复执行相同的查询，且每次返回完全相同的数据，因此，基于这样的应用场景，我们可以将这部分数据缓存起来，这样的话能够提高查询效率

## 执行过程优化

### 查询缓存

* 在mysql8.0之后的版本中，取消了查询缓存，原因是，查询缓存一单表中的数据发生变化了，就会失效，即命中缓存的 几率不是很大。

尽管如此，在mysql5中的某些情况是可以使用的，例如字典表等。

* 在解析一个查询语句之前，如果查询缓存是打开的，那么mysql会优先检查这个查询是否命中查询缓存中的数据，如果查询恰好命中了查询缓存，那么会在返回结果之前会检查用户权限，如果权限没有问题，那么mysql会跳过所有的阶段，就直接从缓存中拿到结果并返回给客户端

### 查询优化处理

#### 语法解析器和预处理（概念）

* 在Apache有开源的产品来做mysql的语法解析。
* mysql通过关键字将SQL语句进行解析，并生成一颗解析树，mysql解析器将使用mysql语法规则验证和解析查询，例如验证使用使用了错误的关键字或者顺序是否正确等等，预处理器会进一步检查解析树是否合法，例如表名和列名是否存在，是否有歧义，还会验证权限等等

#### 查询优化器

##### 释义

当语法树没有问题之后，相应的要由优化器将其转成执行计划，一条查询语句可以使用非常多的执行方式，最后都可以得到对应的结果，但是不同的执行方式带来的效率是不同的，优化器的最主要目的就是要选择最有效的执行计划

mysql使用的是基于成本的优化器，在优化的时候会尝试预测一个查询使用某种查询计划时候的成本，并选择其中成本最小的一个

优化器

##### 优化成本计算

select count(\*) from film\_actor;

show status like 'last\_query\_cost';

可以看到这条查询语句大概需要做1104个数据页才能找到对应的数据，这是经过一系列的统计信息计算来的。

影响计算的因素：

1. 每个表或者索引的页面个数
2. 索引的基数
3. 索引和数据行的长度
4. 索引的分布情况

##### mysql选择错误的执行计划原因

* 统计信息不准确

InnoDB因为其mvcc的架构，并不能维护一个数据表的行数的精确统计信息

* 执行计划的成本估算不等同于实际执行的成本

有时候某个执行计划虽然需要读取更多的页面，但是他的成本却更小，因为如果这些页面都是顺序读或者这些页面都已经在内存中的话，那么它的访问成本将很小，mysql层面并不知道哪些页面在内存中，哪些在磁盘，所以查询之际执行过程中到底需要多少次IO是无法得知的

* mysql的最优可能跟你想的不一样

mysql的优化是基于成本模型的优化，但是有可能不是最快的优化

* mysql不考虑其他并发执行的查询
* mysql不会考虑不受其控制的操作成本

执行存储过程或者用户自定义函数的成本

##### 优化器的优化策略

mysql对查询的静态优化只需要一次，但对动态优化在每次执行时都需要重新评估

* 静态优化

直接对解析树进行分析，并完成优化

* 动态优化

动态优化与查询的上下文有关，也可能跟取值、索引对应的行数有关

##### 优化器的优化类型

1. 重新定义关联表的顺序

数据表的关联并不总是按照在查询中指定的顺序进行，决定关联顺序时优化器很重要的功能

1. 将外连接转化成内连接，内连接的效率要高于外连接
2. 使用等价变换规则，mysql可以使用一些等价变化来简化并规划表达式
3. 优化count(),min(),max()

索引和列是否可以为空通常可以帮助mysql优化这类表达式：例如，要找到某一列的最小值，只需要查询索引的最左端的记录即可，不需要全文扫描比较

1. 预估并转化为常数表达式，当mysql检测到一个表达式可以转化为常数的时候，就会一直把该表达式作为常数进行处理

explain select film.film\_id,film\_actor.actor\_id from film inner join film\_actor using(film\_id) where film.film\_id = 1

1. 索引覆盖扫描，当索引中的列包含所有查询中需要使用的列的时候，可以使用覆盖索引
2. 子查询优化

mysql在某些情况下可以将子查询转换一种效率更高的形式，从而减少多个查询多次对数据进行访问，例如将经常查询的数据放入到缓存中

1. 等值传播

如果两个列的值通过等式关联，那么mysql能够把其中一个列的where条件传递到另一个上：

explain select film.film\_id from film inner join film\_actor using(film\_id

) where film.film\_id > 500;

这里使用film\_id字段进行等值关联，film\_id这个列不仅适用于film表而且适用于film\_actor表

explain select film.film\_id from film inner join film\_actor using(film\_id

) where film.film\_id > 500 and film\_actor.film\_id > 500;

#### 关联查询

1. **join的多种方式**

注释：在不适用index进行表关联时，mysql会使用join\_buffer缓存关联列。有N个join，就会有N-1的join buffer 缓冲区，注意mysql的join\_buffer\_size是可以设置的，但是大小，应该根据硬件内存的大小综合考虑。

查看join\_buffer\_size大小： show variables like '%join\_buffer%';

注意：如果是大表join大表，可以通过过滤一些无效数据，从而减少数据集进行优化。

left join 中的 and 是在表连接前过滤A表或B表里哪些记录符合连接条件，同时会兼顾left join 还是right join ，即假如是左连接的话，如果左边的某些记录不符合连接条件，那么他不进行连接，但是仍然留在结果集中（此时右边的连接结果为null）。on条件实在生成临时表时使用的条件，它不管on中的条件是否为真，都会返回左边表中的记录。

**2. 注意：**

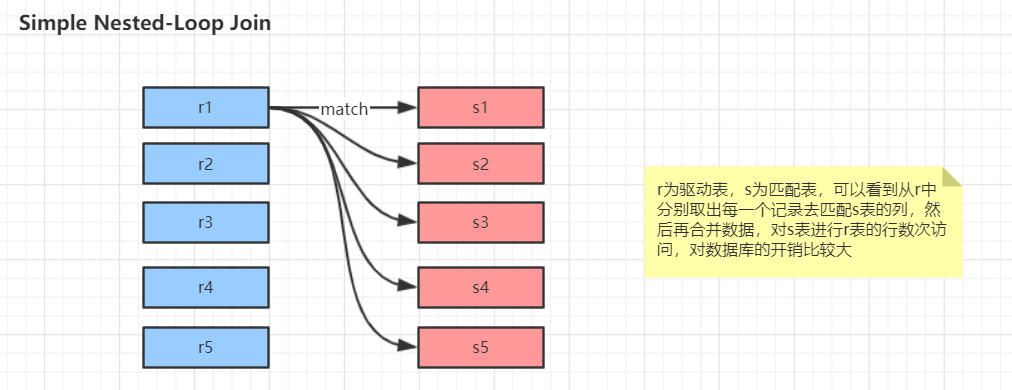
（1）Join Buffer会缓存所有参与查询的列而不是只有Join的列。

（2）可以通过调整join\_buffer\_size缓存大小

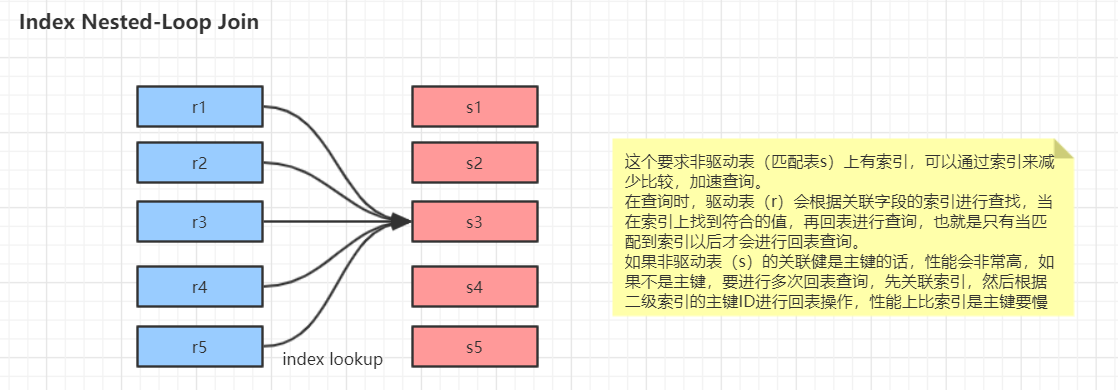
（3）join\_buffer\_size的默认值是256K，join\_buffer\_size的最大值在MySQL 5.1.22版本前是4G-1，而之后的版本才能在64位操作系统下申请大于4G的Join Buffer空间。

（4）使用Block Nested-Loop Join算法需要开启优化器管理配置的optimizer\_switch的设置block\_nested\_loop为on，默认为开启。

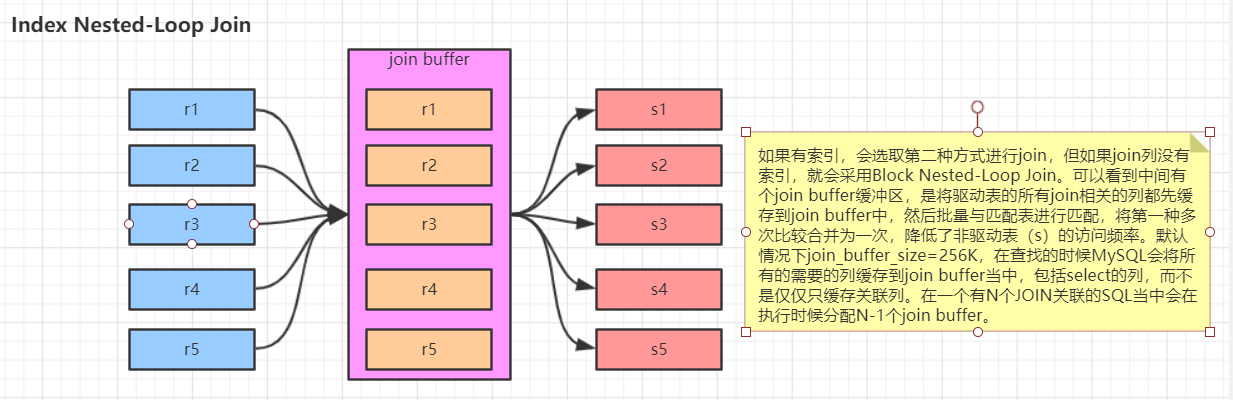
Simple Nested-Loop Join



Index Nested-Loop Join



Block Nested-Loop Join



#### 关联查询案例

**案例分析：**

基于关联查询的原理，我们可以有这样的认知，如果驱动表为小表的话，我们关联查询轮询遍历的次数就会越少。因此我们希望关联查询时，小表作为驱动表，而这个动作，通常mysql的优化器会帮助我们来处理。但是有时候，优化器做出错误的判断，我们也可以使用关键字，规定，我们写的sql中，左表为驱动表。

1. 查看不同的顺序执行方式对查询性能的影响：

EXPLAIN SELECT

film.film\_id,

film.title,

film.release\_year,

actor.actor\_id,

actor.first\_name,

actor.last\_name

FROM

film

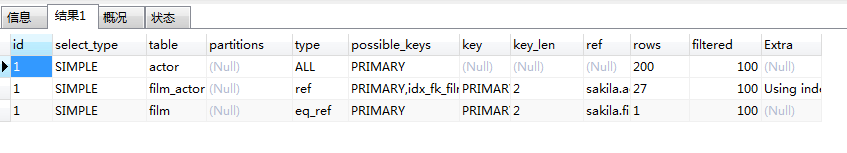
INNER JOIN film\_actor USING (film\_id)

INNER JOIN actor USING (actor\_id);

**执行计划截图：actor=200,film\_actor=5462,film=1000,从执行计划看出，rows = 200 + 27 + 1 = 228行**

**查看执行的成本：**

**show status like 'last\_query\_cost';**



1. 使用关键字，强制执行驱动表

EXPLAIN SELECT STRAIGHT\_JOIN

film.film\_id,

film.title,

film.release\_year,

actor.actor\_id,

actor.first\_name,

actor.last\_name

FROM

film

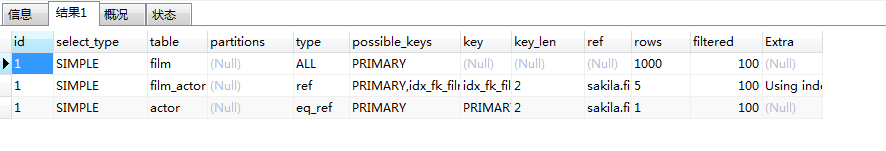
INNER JOIN film\_actor USING (film\_id)

INNER JOIN actor USING (actor\_id);

**执行计划截图：actor=200,film\_actor=5462,film=1000,从执行计划看出，rows = 1000 + 5 + 1 = 1006行,比第一个多出好多行。**

**查看执行的成本：**

**show status like 'last\_query\_cost';**



#### 排序优化

##### 释义

无论如何排序都是一个成本很高的操作，所以从性能的角度出发，应该尽可能避免排序或者尽可能避免对大量数据进行排序。

**推荐使用利用索引进行排序，但是当不能使用索引的时候，**mysql就需要自己进行排序，如果数据量小则再内存中进行，如果数据量大就需要使用磁盘，mysql中称之为filesort。

如果需要排序的数据量小于排序缓冲区(show variables like '%sort\_buffer\_size%';),mysql使用内存进行快速排序操作，如果内存不够排序，那么mysql就会先将树分块，对每个独立的块使用快速排序进行排序，并将各个块的排序结果存放再磁盘上，然后将各个排好序的块进行合并，最后返回排序结果

##### 两次传输排序

第一次数据读取是将需要排序的字段读取出来，然后进行排序，第二次是将排好序的结果按照需要去读取数据行。

这种方式效率比较低，原因是第二次读取数据的时候因为已经排好序，需要去读取所有记录而此时更多的是随机IO，读取数据成本会比较高

两次传输的优势，在排序的时候存储尽可能少的数据，让排序缓冲区可以尽可能多的容纳行数来进行排序操作

##### 单次传输排序

先读取查询所需要的所有列，然后再根据给定列进行排序，最后直接返回排序结果，此方式只需要一次顺序IO读取所有的数据，而无须任何的随机IO，问题在于查询的列特别多的时候，会占用大量的存储空间，无法存储大量的数据

##### 参数调优

当需要排序的列的总大小超过max\_length\_for\_sort\_data定义的字节，mysql会选择双次排序，反之使用单次排序，当然，用户可以设置此参数的值来选择排序的方式

## 优化特定类型的查询

### 优化count()查询

1. count()作用

count()是特殊的函数，有两种不同的作用，一种是某个列值的数量，也可以统计行数

1. 纠正错误说法：

count(1),count(\*),count(索引字段),count(非索引字段)，不考虑null的情况下，效率是一样的。从执行计划，以及查询成本都能证明这一结论。不过写常量能简单点而已。

1. 总有人认为myisam的count函数比较快，这是有前提条件的，只有没有任何where条件的count(\*)才是比较快的。因为myisam内部会统计插入的行数，如果不带where条件，会直接返回该值。
2. 使用近似值

在某些应用场景中，不需要完全精确的值，可以参考使用近似值来代替，比如可以使用explain来获取近似的值

其实在很多OLAP的应用中，需要计算某一个列值的基数，有一个计算近似值的算法叫hyperloglog。

1. 更复杂的优化

一般情况下，count()需要扫描大量的行才能获取精确的数据，其实很难优化，在实际操作的时候可以考虑使用索引覆盖扫描，或者增加汇总表，或者增加外部缓存系统。

### 优化关联查询

1. 确保on或者using子句中的列上有索引，在创建索引的时候就要考虑到关联的顺序：

当表A和表B使用列C关联的时候，如果优化器的关联顺序是B、A，那么就不需要再B表的对应列上建上索引，没有用到的索引只会带来额外的负担，一般情况下来说，只需要在关联顺序中的第二个表的相应列上创建索引

1. 确保任何的groupby和order by中的表达式只涉及到一个表中的列，这样mysql才有可能使用索引来优化这个过程：

### 优化子查询

子查询的优化最重要的优化建议是尽可能使用关联查询代替

原因：子查询，会生成临时表，当子查询的结果集太大的话，一个会造成内存压力，另外一个如果结果集过大的话，会将数据存入磁盘中，也会导致IO操作。

而关联查询，也会创建临时表，只不过此时的临时表是存储join之后的数据，会进行一些on条件的过滤，相比较而言，结果集会小一些。

### 优化limit分页

优化此类查询的最简单的办法就是尽可能地使用覆盖索引，而不是查询所有的列

1. 例如：执行计划：rows=1000

select film\_id,description from film order by title limit 50,5

1. 例如：执行计划：rows=55+1+55=111

EXPLAIN SELECT film.film\_id,film.description

FROM film

INNER JOIN (SELECT film\_id FROM film ORDER BY title LIMIT 50,5) AS lim USING (film\_id);

### 优化union查询

除非确实需要服务器消除重复的行，否则一定要使用union all，因此没有all关键字，mysql会在查询的时候给临时表加上distinct的关键字，这个操作的代价很高

### 推荐使用用户自定义变量

#### 释义

用户自定义变量是一个容易被遗忘的mysql特性，但是如果能够用好，在某些场景下可以写出非常高效的查询语句，在查询中混合使用过程化和关系话逻辑的时候，自定义变量会非常有用。

用户自定义变量是一个用来存储内容的临时容器，在连接mysql的整个过程中都存在。

#### 自定义变量的使用

1. set @one :=1
2. set @min\_actor :=(select min(actor\_id) from actor)
3. set @last\_week :=current\_date-interval 1 week;

#### 自定义变量的限制

1. 无法使用查询缓存
2. 不能在使用常量或者标识符的地方使用自定义变量，例如表名、列名或者limit子句
3. 用户自定义变量的生命周期是在一个连接中有效，所以不能用它们来做连接间的通信
4. 不能显式地声明自定义变量地类型
5. mysql优化器在某些场景下可能会将这些变量优化掉，这可能导致代码不按预想地方式运行
6. 赋值符号：=的优先级非常低，所以在使用赋值表达式的时候应该明确的使用括号
7. 使用未定义变量不会产生任何语法错误

#### 自定义变量的使用案例

**案例一：优化排名语句**

1. 在给一个变量赋值的同时使用这个变量，实现行号,但是一定要事先定义set@rownum:=0

select actor\_id,@rownum:=@rownum+1 as rownum from actor limit 10;

1. 查询获取演过最多电影的前10名演员，然后根据出演电影次数做一个排名

select actor\_id,count(\*) as cnt from film\_actor group by actor\_id order by cnt desc limit 10;

**案例二：避免重新查询刚刚更新的数据：当需要高效的更新一条记录的时间戳，同时希望查询当前记录中存放的时间戳是什么？**

1. **正常的解决方案，是更新完，在查询一遍**

**update film\_actor set last\_update=now() where actor\_id =1;**

**select last\_update from film\_actor where actor\_id =1;**

1. **可以使用自定义变量记录更新的时间，直接用变量查询**

**update film\_actor set last\_update=now() where actor\_id =1 and @now:=now();**

**select @now;**

**案例三：确定取值的顺序（在赋值和读取变量的时候可能是在查询的不同阶段）**

1. **因为where和select在查询的不同阶段执行，所以看到查询到两条记录，这不符合预期**

**set @rownum:=0;**

**select actor\_id,@rownum:=@rownum+1 as cnt from actor where @rownum<=1;**

1. **当引入了orde;r by之后，发现打印出了全部结果，这是因为order by引入了文件排序，而where条件是在文件排序操作之前取值的**

**set @rownum:=0;**

**select actor\_id,@rownum:=@rownum+1 as cnt from actor where @rownum<=1 order by first\_name**

1. **解决这个问题的关键在于让变量的赋值和取值发生在执行查询的同一阶段**

**set @rownum:=0;**

**select actor\_id,@rownum as cnt from actor where (@rownum:=@rownum+1)<=1;**

# 表的分区

## 分区表的应用场景

1. 表非常大以至于无法全部都放在内存中，或者只在表的最后部分有热点数据，其他均是历史数据
2. 分区表的数据更容易维护

* 批量删除大量数据可以使用清除整个分区的方式
* 对一个独立分区进行优化、检查、修复等操作

1. 分区表的数据可以分布在不同的物理设备上，从而高效地利用多个硬件设备
2. 可以使用分区表来避免某些特殊的瓶颈

* innodb的单个索引的互斥访问
* ext3文件系统的inode锁竞争

1. 可以备份和恢复独立的分区

## 分区表的好处

1. 查询

通过分区字段可以快速定位到数据所在分区，从而大大缩小了数据的查询范围，同时在底层，mysql会为每个物理子表重新组织索引树，因此在物理子表使用索引查询会更快，因为数量小，索引树的层次会随之减少，从而导致索引覆盖、回表等操作，效率会更高一些。

1. 删除、修改

当某些分区存放的都是历史数据，我们可以很轻松的清理掉这个分区的所有数据，同时我们在删除时，如果where条件包含分区字段、索引字段，比普通的删除也会更快些，同时，如果你删除的数据没有到索引列，但用到了分区列，这样，只会锁住某几个物理子表的。但这些都是建立在大数量的基础上。

1. 新增

对于新增而言，如果是大数量没有分区前，新增一条数据，会重新维护索引树，并且索引树越大，维护起来的成本越高，并且有可能出现页分裂、页合并的现象，当分区之后，每个物理子表的索引树会随之数据量的减小而变的简单，这样在插入时，会减少维护索引树的代价。

## 分区表的限制

1. 一个表最多只能有1024个分区，在5.7版本的时候可以支持8196个分区
2. 在早期的mysql中，分区表达式必须是整数或者是返回整数的表达式，在mysql5.5中，某些场景可以直接使用列来进行分区
3. 如果分区字段中有主键或者唯一索引的列，那么所有主键列和唯一索引列都必须包含进来
4. 分区表无法使用外键约束

## 分区表的原理



## 查看分区数据量

SELECT PARTITION\_NAME,TABLE\_ROWS

FROM INFORMATION\_SCHEMA.PARTITIONS

WHERE TABLE\_NAME = 'table\_name';

## 分区的类型

### 范围分区

**根据列值在给定范围内将行分配给分区。**

1. **创建带分区的表，下面建表的语句是按照store\_id来进行分区的，指定了4个分区：**

**在当前的建表语句中可以看到，store\_id的值在1-5的在p0分区，6-10的在p1分区，11-15的在p3分区，16-20的在p4分区，但是如果插入超过20的值就会报错，因为mysql不知道将数据放在哪个分区，可以通过maxvalue表示始终大于等于最大可能整数值的整数值**

CREATE TABLE employees (

id INT NOT NULL,

fname VARCHAR(30),

lname VARCHAR(30),

hired DATE NOT NULL DEFAULT '1970-01-01',

separated DATE NOT NULL DEFAULT '9999-12-31',

job\_code INT NOT NULL,

store\_id INT NOT NULL

)

PARTITION BY RANGE (store\_id) (

PARTITION p0 VALUES LESS THAN (6),

PARTITION p1 VALUES LESS THAN (11),

PARTITION p2 VALUES LESS THAN (16),

PARTITION p3 VALUES LESS THAN MAXVALUE

);

1. **可以使用date类型进行分区：如虚妄根据每个员工离开公司的年份进行划分，如year(separated)**

CREATE TABLE employees (

id INT NOT NULL,

fname VARCHAR(30),

lname VARCHAR(30),

hired DATE NOT NULL DEFAULT '1970-01-01',

separated DATE NOT NULL DEFAULT '9999-12-31',

job\_code INT,

store\_id INT

)

PARTITION BY RANGE ( YEAR(separated) ) (

PARTITION p0 VALUES LESS THAN (1991),

PARTITION p1 VALUES LESS THAN (1996),

PARTITION p2 VALUES LESS THAN (2001),

PARTITION p3 VALUES LESS THAN MAXVALUE

);

1. **可以使用函数根据range的值来对表进行分区，如timestampunix\_timestamp()**

**timestamp不允许使用任何其他涉及值的表达式**

CREATE TABLE quarterly\_report\_status (

report\_id INT NOT NULL,

report\_status VARCHAR(20) NOT NULL,

report\_updated TIMESTAMP NOT NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP

)

PARTITION BY RANGE ( UNIX\_TIMESTAMP(report\_updated) ) (

PARTITION p0 VALUES LESS THAN ( UNIX\_TIMESTAMP('2008-01-01 00:00:00') ),

PARTITION p1 VALUES LESS THAN ( UNIX\_TIMESTAMP('2008-04-01 00:00:00') ),

PARTITION p2 VALUES LESS THAN ( UNIX\_TIMESTAMP('2008-07-01 00:00:00') ),

PARTITION p3 VALUES LESS THAN ( UNIX\_TIMESTAMP('2008-10-01 00:00:00') ),

PARTITION p4 VALUES LESS THAN ( UNIX\_TIMESTAMP('2009-01-01 00:00:00') ),

PARTITION p5 VALUES LESS THAN ( UNIX\_TIMESTAMP('2009-04-01 00:00:00') ),

PARTITION p6 VALUES LESS THAN ( UNIX\_TIMESTAMP('2009-07-01 00:00:00') ),

PARTITION p7 VALUES LESS THAN ( UNIX\_TIMESTAMP('2009-10-01 00:00:00') ),

PARTITION p8 VALUES LESS THAN ( UNIX\_TIMESTAMP('2010-01-01 00:00:00') ),

PARTITION p9 VALUES LESS THAN (MAXVALUE)

);

1. **基于时间间隔的分区方案，在mysql5.7中，可以基于范围或事件间隔实现分区方案，有两种选择**
   1. **基于范围的分区，对于分区表达式，可以使用操作函数基于date、time、或者datatime列来返回一个整数值**

CREATE TABLE members (

firstname VARCHAR(25) NOT NULL,

lastname VARCHAR(25) NOT NULL,

username VARCHAR(16) NOT NULL,

email VARCHAR(35),

joined DATE NOT NULL

)

PARTITION BY RANGE( YEAR(joined) ) (

PARTITION p0 VALUES LESS THAN (1960),

PARTITION p1 VALUES LESS THAN (1970),

PARTITION p2 VALUES LESS THAN (1980),

PARTITION p3 VALUES LESS THAN (1990),

PARTITION p4 VALUES LESS THAN MAXVALUE

);

CREATE TABLE quarterly\_report\_status (

report\_id INT NOT NULL,

report\_status VARCHAR(20) NOT NULL,

report\_updated TIMESTAMP NOT NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP

)

PARTITION BY RANGE ( UNIX\_TIMESTAMP(report\_updated) ) (

PARTITION p0 VALUES LESS THAN ( UNIX\_TIMESTAMP('2008-01-01 00:00:00') ),

PARTITION p1 VALUES LESS THAN ( UNIX\_TIMESTAMP('2008-04-01 00:00:00') ),

PARTITION p2 VALUES LESS THAN ( UNIX\_TIMESTAMP('2008-07-01 00:00:00') ),

PARTITION p3 VALUES LESS THAN ( UNIX\_TIMESTAMP('2008-10-01 00:00:00') ),

PARTITION p4 VALUES LESS THAN ( UNIX\_TIMESTAMP('2009-01-01 00:00:00') ),

PARTITION p5 VALUES LESS THAN ( UNIX\_TIMESTAMP('2009-04-01 00:00:00') ),

PARTITION p6 VALUES LESS THAN ( UNIX\_TIMESTAMP('2009-07-01 00:00:00') ),

PARTITION p7 VALUES LESS THAN ( UNIX\_TIMESTAMP('2009-10-01 00:00:00') ),

PARTITION p8 VALUES LESS THAN ( UNIX\_TIMESTAMP('2010-01-01 00:00:00') ),

PARTITION p9 VALUES LESS THAN (MAXVALUE)

);

* 1. **基于范围列的分区，使用date或者datatime列作为分区列**

CREATE TABLE members (

firstname VARCHAR(25) NOT NULL,

lastname VARCHAR(25) NOT NULL,

username VARCHAR(16) NOT NULL,

email VARCHAR(35),

joined DATE NOT NULL

)

PARTITION BY RANGE COLUMNS(joined) (

PARTITION p0 VALUES LESS THAN ('1960-01-01'),

PARTITION p1 VALUES LESS THAN ('1970-01-01'),

PARTITION p2 VALUES LESS THAN ('1980-01-01'),

PARTITION p3 VALUES LESS THAN ('1990-01-01'),

PARTITION p4 VALUES LESS THAN MAXVALUE

);

### 列表分区

类似于按range分区，区别在于list分区是基于列值匹配一个离散值集合中的某个值来进行选择

CREATE TABLE employees (

id INT NOT NULL,

fname VARCHAR (30),

lname VARCHAR (30),

hired DATE NOT NULL DEFAULT '1970-01-01',

separated DATE NOT NULL DEFAULT '9999-12-31',

job\_code INT,

store\_id INT

) PARTITION BY LIST (store\_id)(

PARTITION pNorth VALUES IN (3, 5, 6, 9, 17),

PARTITION pEast VALUES IN (1, 2, 10, 11, 19, 20),

PARTITION pWest VALUES IN (4, 12, 13, 14, 18),

PARTITION pCentral VALUES IN (7, 8, 15, 16)

);

### 列分区

mysql从5.5开始支持column分区，可以认为是range和list的升级版，在5.5之后，可以使用column分区替代range和list，但是column分区只接受普通列不接受表达式

 CREATE TABLE `list\_c` (

 `c1` int(11) DEFAULT NULL,

 `c2` int(11) DEFAULT NULL

) ENGINE=InnoDB DEFAULT CHARSET=latin1

PARTITION BY RANGE COLUMNS(c1)

(PARTITION p0 VALUES LESS THAN (5) ENGINE = InnoDB,

 PARTITION p1 VALUES LESS THAN (10) ENGINE = InnoDB)

 CREATE TABLE `list\_c` (

 `c1` int(11) DEFAULT NULL,

 `c2` int(11) DEFAULT NULL,

 `c3` char(20) DEFAULT NULL

) ENGINE=InnoDB DEFAULT CHARSET=latin1

PARTITION BY RANGE COLUMNS(c1,c3)

(PARTITION p0 VALUES LESS THAN (5,'aaa') ENGINE = InnoDB,

 PARTITION p1 VALUES LESS THAN (10,'bbb') ENGINE = InnoDB)

 CREATE TABLE `list\_c` (

 `c1` int(11) DEFAULT NULL,

 `c2` int(11) DEFAULT NULL,

 `c3` char(20) DEFAULT NULL

) ENGINE=InnoDB DEFAULT CHARSET=latin1

PARTITION BY LIST COLUMNS(c3)

(PARTITION p0 VALUES IN ('aaa') ENGINE = InnoDB,

 PARTITION p1 VALUES IN ('bbb') ENGINE = InnoDB)

### hash分区

基于用户定义的表达式的返回值来进行选择的分区，该表达式使用将要插入到表中的这些行的列值进行计算。这个函数可以包含myql中有效的、产生非负整数值的任何表达式

CREATE TABLE employees (

id INT NOT NULL,

fname VARCHAR(30),

lname VARCHAR(30),

hired DATE NOT NULL DEFAULT '1970-01-01',

separated DATE NOT NULL DEFAULT '9999-12-31',

job\_code INT,

store\_id INT

)

PARTITION BY HASH(store\_id)

PARTITIONS 4;

CREATE TABLE employees (

id INT NOT NULL,

fname VARCHAR(30),

lname VARCHAR(30),

hired DATE NOT NULL DEFAULT '1970-01-01',

separated DATE NOT NULL DEFAULT '9999-12-31',

job\_code INT,

store\_id INT

)

PARTITION BY LINEAR HASH(YEAR(hired))

PARTITIONS 4;

### key分区

类似于hash分区，区别在于key分区只支持一列或多列，且mysql服务器提供其自身的哈希函数，必须有一列或多列包含整数值

CREATE TABLE tk (

col1 INT NOT NULL,

col2 CHAR(5),

col3 DATE

)

PARTITION BY LINEAR KEY (col1)

PARTITIONS 3;

### 子分区

在分区的基础之上，再进行分区后存储

CREATE TABLE `t\_partition\_by\_subpart`

(

`id` INT AUTO\_INCREMENT,

`sName` VARCHAR(10) NOT NULL,

`sAge` INT(2) UNSIGNED ZEROFILL NOT NULL,

`sAddr` VARCHAR(20) DEFAULT NULL,

`sGrade` INT(2) NOT NULL,

`sStuId` INT(8) DEFAULT NULL,

`sSex` INT(1) UNSIGNED DEFAULT NULL,

PRIMARY KEY (`id`, `sGrade`)

) ENGINE = INNODB

PARTITION BY RANGE(id)

SUBPARTITION BY HASH(sGrade) SUBPARTITIONS 2

(

PARTITION p0 VALUES LESS THAN(5),

PARTITION p1 VALUES LESS THAN(10),

PARTITION p2 VALUES LESS THAN(15)

);

## 如何使用分区表

1. 释义：

如果需要从非常大的表中查询出某一段时间的记录，而这张表中包含很多年的历史数据，数据是按照时间排序的，此时应该如何查询数据呢？

因为数据量巨大，肯定不能在每次查询的时候都扫描全表。考虑到索引在空间和维护上的消耗，也不希望使用索引，即使使用索引，会发现会产生大量的碎片，还会产生大量的随机IO，但是当数据量超大的时候，索引也就无法起作用了，此时可以考虑使用分区来进行解决

1. 全量扫描数据，不要任何索引：

使用简单的分区方式存放表，不要任何索引，根据分区规则大致定位需要的数据为止，通过使用where条件将需要的数据限制在少数分区中，这种策略适用于以正常的方式访问大量数据

1. 索引数据，并分离热点

如果数据有明显的热点，而且除了这部分数据，其他数据很少被访问到，那么可以将这部分热点数据单独放在一个分区中，让这个分区的数据能够有机会都缓存在内存中，这样查询就可以只访问一个很小的分区表，能够使用索引，也能够有效的使用缓存

## 在使用分区需注意的问题

1. null值会使分区过滤无效，因此分区字段最好 设置非空
2. 分区列和索引列不匹配，会导致查询无法进行分区过滤（不理解）
3. 选择分区的成本可能很高

如前所述分区有很多类型，不同类型分区的实现方式也不同，所以它们的性能也各不相同。尤其是范围分区，对于回答“这一行 属于哪个分区”、“这些符合查询条件的行在哪些分区”这样的问题的成本可能会非常高，因为服务器需要扫描所有的分区定义的 列表来找到正确的答案。类似这样的线性搜索的效率不高，所以随着分区数的增长，成本会越来越高。

我们所实际碰到的类似这样的最糟牒的一次问题是按行写入大量数据的时候。每写入一行数据到范围分区的表时，都需要扫描分区 定义列表来找到合适的目标分区。可以通过限制分区的数量来缓解此问题，根据实践经验，对大多数系统来说，100个左右的分区是 没有问题的。其他的分区类型，比如键分区和哈希分区，则没有这样的问题。

1. 打开并锁住所有底层表的成本可能很高

当查询访问分区表的时候，MySQL需要打开井锁住所有的底层表，这是分区表的另一个开销。这个操作在分区过滤之前发生，所以无 法通过分区过滤降低此开销，并且该开销也和分区类型无关，会影响所有的查询。这一点对一些本身操作非常快的查询，比如根据 主键查找单行，会带来明显的额外开销。可以用批量操作的方式来降低单个操作的此类开销，例如使用批量插入或者 LOAD DATA INFILE、一次删除多行数据，等等。当然同时还是需要限制分区的个数

1. 维护分区的成本可能很高

某些分区维护操作的速度会非常快，例如新增或者删除分区（当删除一个大分区可能会很慢，不过这是另一回事）。而有些操作， 例如重组分区或者类似ALTER语句的操作：这类操作需要复制数据。重组分区的原理与ALTER类似，先创建一个临时的分区，然后将 数据复制到其中，最后再删除原分区。

# 服务器参数设置

## general通用配置

1. 数据文件存放的目录

datadir=/var/lib/mysql

1. mysql.socket表示server和client在同一台服务器，并且使用localhost进行连接，就会使用socket进行连接

socket=/var/lib/mysql/mysql.sock

1. 存储mysql的pid

pid\_file=/var/lib/mysql/mysql.pid

1. mysql服务的端口号

port=3306

1. mysql存储引擎

default\_storage\_engine=InnoDB

1. 当忘记mysql的用户名密码的时候，可以在mysql配置文件中配置该参数，跳过权限表验证，不需要密码即可登录mysql

skip-grant-tables

## character字符集配置

1. 客户端数据的字符集

character\_set\_client

1. mysql处理客户端发来的信息时，会把这些数据转换成连接的字符集格式

character\_set\_connection

1. mysql发送给客户端的结果集所用的字符集

character\_set\_results

1. 数据库默认的字符集

character\_set\_database

1. mysql server的默认字符集

character\_set\_server

## connection 连接配置

1. mysql的最大连接数，如果数据库的并发连接请求比较大，应该调高该值

max\_connections

1. 限制每个用户的连接个数

max\_user\_connections

1. mysql能够暂存的连接数量，当mysql的线程在一个很短时间内得到非常多的连接请求时，就会起作用，如果mysql的连接数量达到max\_connections时，新的请求会被存储在堆栈中，以等待某一个连接释放资源，如果等待连接的数量超过back\_log,则不再接受连接资源

back\_log

1. mysql在关闭一个非交互的连接之前需要等待的时长

wait\_timeout

1. 关闭一个交互连接之前需要等待的秒数

interactive\_timeout

## log 日志配置

1. 指定错误日志文件名称，用于记录当mysqld启动和停止时，以及服务器在运行中发生任何严重错误时的相关信息

log\_error

1. 指定二进制日志文件名称，用于记录对数据造成更改的所有查询语句

log\_bin

1. 指定将更新记录到二进制日志的数据库，其他所有没有显式指定的数据库更新将忽略，不记录在日志中

binlog\_do\_db

1. 指定不将更新记录到二进制日志的数据库

binlog\_ignore\_db

1. 指定多少次写日志后同步磁盘

sync\_binlog

1. 是否开启查询日志记录

general\_log

1. 指定查询日志文件名，用于记录所有的查询语句

general\_log\_file

1. 是否开启慢查询日志记录

slow\_query\_log

1. 指定慢查询日志文件名称，用于记录耗时比较长的查询语句

slow\_query\_log\_file

1. 设置慢查询的时间，超过这个时间的查询语句才会记录日志

long\_query\_time

1. 是否将管理语句写入慢查询日志

log\_slow\_admin\_statements

## cache 缓存配置

### key\_buffer\_size

索引缓存区的大小（只对myisam表起作用）

### query cache

* **query\_cache\_size：查询缓存的大小，未来版本被删除**

|  |  |
| --- | --- |
| **查询命令** | **作用** |
| show status like '%Qcache%' | 查看缓存的相关属性 |
| show status like '%Qcache\_free\_blocks%' | 缓存中相邻内存块的个数，如果值比较大，那么查询缓存中碎片比较多 |
| show status like '% Qcache\_free\_memory%' | 查询缓存中剩余的内存大小 |
| show status like '%Qcache\_hits%' | 表示有多少此命中缓存 |
| show status like '%Qcache\_inserts%' | 表示多少次未命中而插入 |
| show status like '%Qcache\_lowmen\_prunes%' | 多少条query因为内存不足而被移除cache |
| show status like '%Qcache\_queries\_in\_cache%' | 当前cache中缓存的query数量 |
| show status like '%Qcache\_total\_blocks%' | 当前cache中block的数量 |

* **query\_cache\_limit：超出此大小的查询将不被缓存**
* **query\_cache\_min\_res\_unit：缓存块最小大小**
* **query\_cache\_type：缓存类型，决定缓存什么样的查询**

|  |  |
| --- | --- |
| **结果** | **含义** |
| 0 | 表示禁用 |
| 1 | 表示将缓存所有结果，除非sql语句中使用sql\_no\_cache禁用查询缓存 |
| 2 | 表示只缓存select语句中通过sql\_cache指定需要缓存的查询 |

### sort\_buffer\_size

每个需要排序的线程分派该大小的缓冲区

### max\_allowed\_packet=32M

限制server接受的数据包大小

### join\_buffer\_size=2M

表示关联缓存的大小

### thread\_cache\_size

* 释义：thread\_cache\_size

服务器线程缓存，这个值表示可以重新利用保存再缓存中的线程数量，当断开连接时，那么客户端的线程将被放到缓存中以响应下一个客户而不是销毁，如果线程重新被请求，那么请求将从缓存中读取，如果缓存中是空的或者是新的请求，这个线程将被重新请求，那么这个线程将被重新创建，如果有很多新的线程，增加这个值即可

* 相关参数：

|  |  |
| --- | --- |
| **结果** | **含义** |
| show status like '%Thread\_cached%' | 代表当前此时此刻线程缓存中有多少空闲线程 |
| show status like '%Threads\_connected%' | 代表当前已建立连接的数量 |
| show status like '%Threads\_created%' | 代表最近一次服务启动，已创建现成的数量，如果该值比较大，那么服务器会一直再创建线程 |
| show status like '%Threads\_running%'; | 代表当前激活的线程数 |

## INNODB 存储引擎

1. show variables like '%innodb\_buffer\_pool\_size%';

该参数指定大小的内存来缓冲数据和索引，最大可以设置为物理内存的80%

1. show variables like '%innodb\_flush\_log\_at\_trx\_commit%';

主要控制innodb将log buffer中的数据写入日志文件并flush磁盘的时间点，值分别为0，1，2

1. show variables like '%innodb\_thread\_concurrency%';

设置innodb线程的并发数，默认为0表示不受限制，如果要设置建议跟服务器的cpu核心数一致或者是cpu核心数的两倍

1. show variables like '%innodb\_log\_buffer\_size%';

此参数确定日志文件所用的内存大小，以M为单位

1. show variables like '%innodb\_log\_file\_size%';

此参数确定数据日志文件的大小，以M为单位

1. show variables like '%innodb\_log\_files\_in\_group%';

以循环方式将日志文件写到多个文件中

1. show variables like '%read\_buffer\_size%';

mysql读入缓冲区大小，对表进行顺序扫描的请求将分配到一个读入缓冲区

1. show variables like '%read\_rnd\_buffer\_size%';

mysql随机读的缓冲区大小

1. show variables like '%innodb\_file\_per\_table%';

此参数确定为每张表分配一个新的文件

# mysql事物

文章地址：https://www.cnblogs.com/kismetv/p/10331633.html

## 概念

1. MySQL中服务器层不管理事务，事务是由存储引擎实现的
2. 事务（Transaction）：

是访问和更新数据库的程序执行单元；事务中可能包含一个或多个sql语句，这些语句要么都执行，要么都不执行。

1. ACID特性：

* 原子性（Atomicity，或称不可分割性）
* 一致性（Consistency）
* 隔离性（Isolation）
* 持久性（Durability）

按照严格的标准，只有同时满足ACID特性才是事务；但是在各大数据库厂商的实现中，真正满足ACID的事务少之又少。例如MySQL的NDB Cluster事务不满足持久性和隔离性；InnoDB默认事务隔离级别是可重复读，不满足隔离性；Oracle默认的事务隔离级别为READ COMMITTED，不满足隔离性……因此与其说ACID是事务必须满足的条件，不如说它们是衡量事务的四个维度。

## 原子性

1. 原子性是指一个事务是一个不可分割的工作单位，其中的操作要么都做，要么都不做；如果事务中一个sql语句执行失败，则已执行的语句也必须回滚，数据库退回到事务前的状态。
2. 实现原理：undo log

## 持久性

1. 持久性是指事务一旦提交，它对数据库的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。
2. 实现原理：redo log

## 隔离性

1. 与原子性、持久性侧重于研究事务本身不同，隔离性研究的是不同事务之间的相互影响。隔离性是指，事务内部的操作与其他事务是隔离的，并发执行的各个事务之间不能互相干扰。严格的隔离性，对应了事务隔离级别中的Serializable (可串行化)，但实际应用中出于性能方面的考虑很少会使用可串行化。
2. 隔离性追求的是并发情形下事务之间互不干扰。简单起见，我们主要考虑最简单的读操作和写操作(加锁读等特殊读操作会特殊说明)，那么隔离性的探讨，主要可以分为两个方面：

(一个事务)写操作对(另一个事务)写操作的影响：锁机制保证隔离性

(一个事务)写操作对(另一个事务)读操作的影响：MVCC保证隔离性

1. 脏读:

当前事务(A)中可以读到其他事务(B)未提交的数据（脏数据），这种现象是脏读。举例如下（以账户余额表为例）：

|  |  |  |
| --- | --- | --- |
| 时间 | 事物A | 事物B |
| T1 | 开始事物 | 开始事物 |
| T2 |  | 修改【张三】余额100变为200 |
| T3 | 查询张三余额200【脏读】 |  |
| T4 |  | 提交事务 |

1. 不可重复读：

在事务A中先后两次读取同一个数据，两次读取的结果不一样，这种现象称为不可重复读。脏读与不可重复读的区别在于：前者读到的是其他事务未提交的数据，后者读到的是其他事务已提交的数据。举例如下：

|  |  |  |
| --- | --- | --- |
| 时间 | 事物A | 事物B |
| T1 | 开始事物 | 开始事物 |
| T2 | 查询张三余额100 |  |
| T3 |  | 修改【张三】余额100变为200 |
| T4 |  | 提交事务 |
| T5 | 查询张三余额200【不可重复读】 |  |

1. 幻读

在事务A中按照某个条件先后两次查询数据库，两次查询结果的条数不同，这种现象称为幻读。不可重复读与幻读的区别可以通俗的理解为：前者是数据变了，后者是数据的行数变了。举例如下：

|  |  |  |
| --- | --- | --- |
| 时间 | 事物A | 事物B |
| T1 | 开始事物 | 开始事物 |
| T2 | 查询0<id<5之间的账户：  张三 100 |  |
| T3 |  | 插入数据 李四 200 |
| T4 |  | 提交事务 |
| T5 | 查询0<id<5之间的账户：  张三 100  李四 200 【幻读】 |  |

1. 事务隔离级别

mysql默认事物隔离级别：repeatable read(可重复度)

|  |  |  |  |
| --- | --- | --- | --- |
| 隔离级别 | 脏读 | 不可重复读 | 幻读 |
| read uncommitted读未提交 | 可能 | 可能 | 可能 |
| read committed读可提交 | 不可能 | 可能 | 可能 |
| repeatable read可重复度 | 不可能 | 不可能 | 可能 |
| serialize可串行化 | 不可能 | 不可能 | 不可能 |

概括来说，InnoDB实现的RR，通过锁机制（包含next-key lock）、MVCC（不加锁情况下，包括数据的隐藏列、基于undo log的版本链、ReadView）等，实现了一定程度的隔离性，可以满足大多数场景的需要。不过需要说明的是，RR虽然避免了幻读问题，但是毕竟不是Serializable，不能保证完全的隔离

## 一致性

1. 概念：

一致性是指事务执行结束后，数据库的完整性约束没有被破坏，事务执行的前后都是合法的数据状态。数据库的完整性约束包括但不限于：实体完整性（如行的主键存在且唯一）、列完整性（如字段的类型、大小、长度要符合要求）、外键约束、用户自定义完整性（如转账前后，两个账户余额的和应该不变）

1. 实现：

可以说，一致性是事务追求的最终目标：前面提到的原子性、持久性和隔离性，都是为了保证数据库状态的一致性。此外，除了数据库层面的保障，一致性的实现也需要应用层面进行保障。

实现一致性的措施包括：

1. 保证原子性、持久性和隔离性，如果这些特性无法保证，事务的一致性也无法保证
2. 数据库本身提供保障，例如不允许向整形列插入字符串值、字符串长度不能超过列的限制等
3. 应用层面进行保障，例如如果转账操作只扣除转账者的余额，而没有增加接收者的余额，无论数据库实现的多么完美，也无法保证状态的一致

## 总结

下面总结一下ACID特性及其实现原理：

1. 原子性：语句要么全执行，要么全不执行，是事务最核心的特性，事务本身就是以原子性来定义的；实现主要基于undo log
2. 持久性：保证事务提交后不会因为宕机等原因导致数据丢失；实现主要基于redo log
3. 隔离性：保证事务执行尽可能不受其他事务影响；InnoDB默认的隔离级别是RR，RR的实现主要基于锁机制（包含next-key lock）、MVCC（包括数据的隐藏列、基于undo log的版本链、ReadView）
4. 一致性：事务追求的最终目标，一致性的实现既需要数据库层面的保障，也需要应用层面的保障