# DBA的工作

首先了解业务，弄清数据的使用和流向

关键点：**数据存储，数据流**

数据存储：表示数据的使用，这批数据如何利用才能更安全，准确，高效。 如何规划数据结构和落地架构。

数据流：（有效性）数据的生命周期，当处于不同生命周期的数据，也应该有不同的流向。（归档，历史数据 如何处理等）

# SQL审核规范

## 1.审核和执行需要分开

（一个任务两个动作）。禁止审核并执行所有代码。如果版本发布比较紧急，请邮件电话说明原因，并承担以后可能会造成影响的风险。

## 2.表使用何种存储引擎，字符集，自增设置。

## 3.默认B-tree索引，无特殊情况，不建议自建using hash。

## 4.表及字段名称明确并添加comment注释。

## 5.字段类型尽量简短

（tinyint不用int,smallint不用int,int不用bigint,varchar。有很多项目组习惯用varchar，char（其实完全可以用tingint,int代替的）。varchar值在符合承载值内越小越好，即分配真正需要的空间。并且尽量设置保持一样。

## 6.字段尽量禁止使用default null。

使用not null. 需要给字段设置一个初始默认值）。-- 防止索引失效，防止造成统计偏差。

## 7.添加必要索引

聚合索引一般都是需要的并且索引尽量设置简短，索引名称规范，统一。例如索引名称命名为：idx\_filedname 。一般建议一张表字段最好不超过20个，索引最好不要超过7个。

## 8.DDL,DML代码需分开

1)DDL文件夹内每张表一个文件。命名格式为alter\_tablename\_01.sql，alter\_tablename\_02.sql  或者create\_tablename\_03.sql  ...

2)DML文件夹内文件命名为tablename\_01.sql，tablename\_02.sql  。代码开头use schema;+ begin 结尾处添加COMMIT;

3)格式

use schema;

begin;

insert into tablename ....;

update tablename set .. ;

commit;

## 9.代码提交规范

代码提交文本文件必须保存格式为utf8 ，并且提交DBM时，需要附加readme.txt文件（文件内容包括：操作流程和注意事项 ，代码执行顺序，本次提交代码主要功能和目的,对应接口负责人）文件。

## 10.慎用BLOB、TEXT

超过255长度的VARCHAR列类型；如有必要，考虑对大字段表做关联分离。

理由：当InnoDB的文件格式（innodb\_file\_format）设置为Antelope，并且行格式为COMPACT 或 REDUNDANT 时，BLOB、TEXT或者长VARCHAR列只会将其前768字节存储在聚集索引页中。（最大768字节的作用是便于创建前缀索引/prefix index），其余更多的内容存储在额外的page里，哪怕只是多了一个字节。因此，所有列长度越短越好。在off-page中存储的BLOB、TEXT或者长VARCHAR列的page是独享的，不能共享。因此强烈不建议在一个表中使用多个长列。

## 合理利用json字段

保证了JSON数据类型的强校验，JSON数据列会自动校验存入此列的内容是否符合JSON格式，非正常格式则报错，而varchar类型和text等类型本身是不存在这种机制的。

MySQL同时提供了一组操作JSON类型数据的内置函数。

更优化的存储格式，存储在JSON列中的JSON数据会被转成内部特定的存储格式，允许快速读取。

可以基于JSON格式的特征支持修改特定的键值。(即不需要把整条内容拿出来放到程序中遍历然后寻找替换再塞回去，MySQL内置的函数允许你通过一条SQL语句就能搞定)

# 开发重点关注

## 1.项目设计前做好规划

1)业务如果复杂，考虑使用分库（耦合度高放置一个库内，耦合度低，采用分库）

2)设计表时对于需求字段考虑周全。（提前考虑所需必要字段。例如add\_time,updatetime时间列，绝大多数情况下是需要的。基本配置表例外）。

理由：mysql5.5版本前不支持在线DDL。 5.6支持在线DDL (这其中也分为索引，和字段。对于添加修改索引5.6版本可以“基本”做到在线DDL。5.6版本对于在线添加字段其实是和5.5一样。做不到！依然维持两倍存储空间，即需创建临时表。可读不可写，并发度低的本质。)

## 2.避免使用select \* 查询

## 3.频繁更新的表字段优化设计

对于频繁更新的表字段是否应该这么做，如果必须进行update操作，尽量更新的字段越少越好。并且频繁更新的字段最好非索引。理由：更新操作会增加索引和统计信息的维护成本。

## 4.一个表在设计时，提前考虑必要索引，了解索引如何使用（和程序代码密切相关）

## 5.在设计时，尽量保证一张表存在必要列

为什么？ mysql存储引擎在工作时需要在服务器层与存储引擎层之间通过行缓冲格式拷贝数据，然后再服务器层将缓冲内容解码成各个列。从行缓冲中将编码过的列转换成行数据结构的操作代价是非常高的。

## 6.一张表应该存在时间字段。

如：add\_time,update\_time。优点：对于查询统计，监控等是有优势的。并且可以有效利用二级索引，提高缓存命中，从而提高查询性能。

## 7.用整行保存ip 字段。而不是varchar

mysql> select inet\_aton('61.183.81.130');

+----------------------------+

| inet\_aton('61.183.81.130') |

+----------------------------+

|                 1035424130 |

+----------------------------+

1 row in set (0.01 sec)

mysql> select inet\_ntoa(1035424130);

+-----------------------+

| inet\_ntoa(1035424130) |

+-----------------------+

| 61.183.81.130         |

+-----------------------+

1 row in set (0.00 sec)

## 8.代码查询尽量保证所需必要字段。

查询字段过多的影响： **无法使用覆盖索引。-- 避免二次扫描。**

## 9.学会"拆分"表

存储需要的字段，相对查询更新频率不高或者存储量较大字段能否考虑拆分为新表关联

建议：对于完全的范式化和完全的反范式，以及在两者中折中的考虑。研发同事可以了解范式，反范式，混合范式之间的区别和各自优势，合理利用在项目中。

|  |  |  |
| --- | --- | --- |
|  | 优点 | 缺点 |
| 范式化模型 | 因为数据没有冗余，更新容易 | 当查询涉及很多关联模型（join）时，会导致查询性能低下 |
| 反范式化模型 | 数据冗余将带来很好的读取性能（因为不需要join关联多表，而且通常反范式模型很少做更新操作） | 需要维护冗余数据 |

## 10.学会预见“未来“

### 1.说明

对于有预见的表数据增长较大的，要在设计上很严格，是否可以考虑在上线前就使用分区。使用何种分区比较合理，需要和DBA商量，了解业务代码选择分区方式。

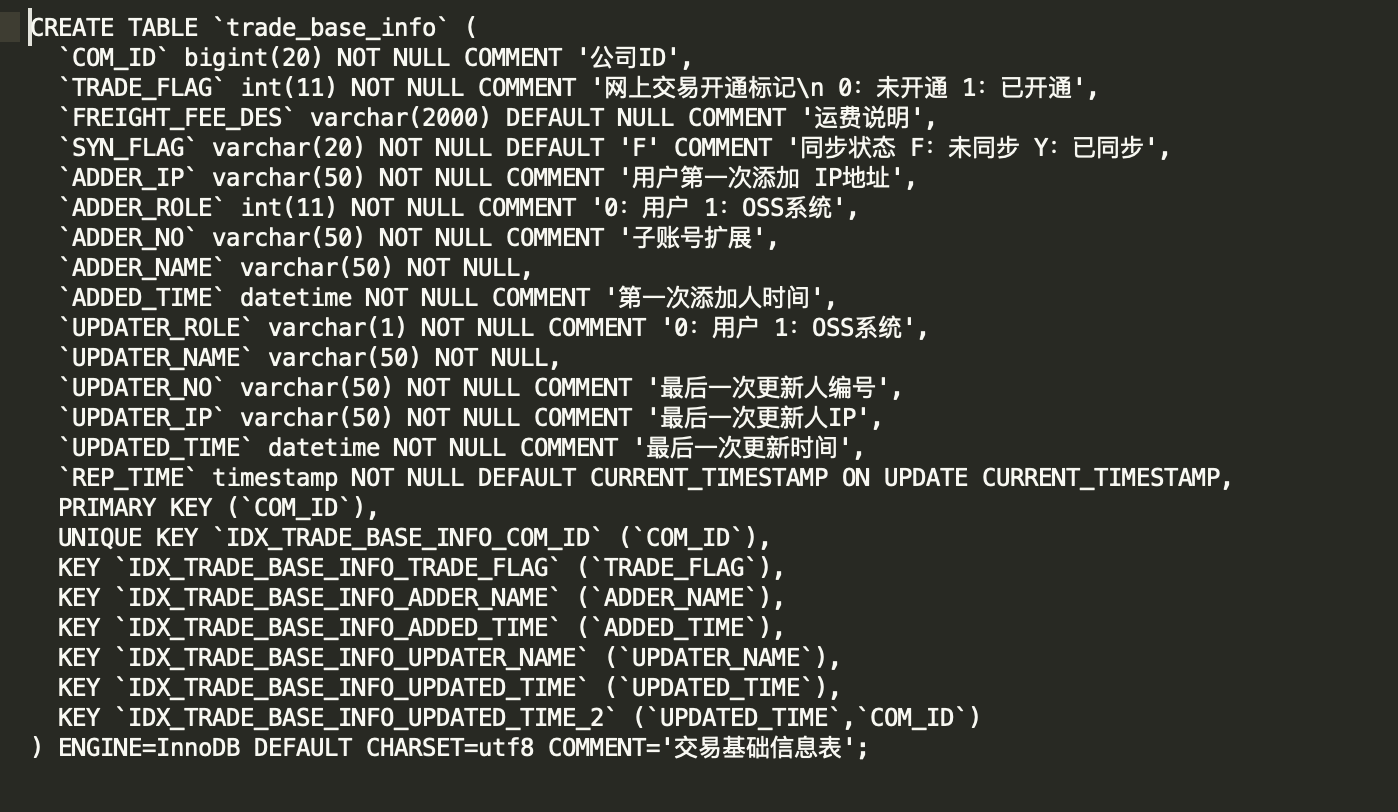
### 2.优势

分区最大的优点就是优化器可以根据分区函数来过滤一些分区。根据粗粒度索引的优势，通过分区过滤通常可以让查询扫描更少的数据。（在某些场景下）

### 3.原则：

即便在创建分区时可以使用表达式，（5.5版本可以直接对列分区）但是在查询时却只能根据列来过滤分区。

## 11.大家来找茬



# 三优原则

表结构设计优化，索引优化，查询性能优化

## 1.索引优化

### 1.索引的好处：

a.索引大大减少服务器需要扫描的数据量

b.索引可以帮助服务器避免排序和临时表

c.索引可以将随机I/0变为顺序I/O

### 2.索引分类

1.B-tree 索引

**(按照顺序存储数据)，后面单独介绍索引与算法。**

2.哈希索引

哈希索引基于哈希表实现。只有精确匹配索引的所有列的查询才有效。对于每一行数据，存储引擎都会对所有的索引列计算一个哈希吗，哈希码是一个较小值，并且不同键值的行计算出来的哈希码不一样。哈希索引将所有的哈希码存储在索引中，同时在哈希表中保存指向每个数据行的指针。

**哈希索引的限制：**

a.哈希索引只包含哈希值和行指针而不存储字段值。不能使用索引中的值来避免读取行。

b.哈希索引数据不是按照索引值顺序存储的，索引无法用于排序。

c.哈希索引不支持部分索引列的匹配查找。因为哈希索引始终是使用索引列的全部内容来计算哈希值的。

d.哈希索引只支持等值比较，包括：=,in(),<=>,也不支持范围查询 如where price>100

e.哈希无法做到真正意义上的唯一性。虽然访问哈希索引的数据非常快，但是当有很多哈希冲突时，存储引擎必须遍历链表中所有行的指针，逐行进行比较，直到找到符合条件的行。

案例分享：

**select id from url where url="http://www.baidu.com"**and url\_crc=CRC32("http://www.baidu.com");

-- 使用简单哈希函数CRC32 ,强加密哈希函数 SHA1(),MD5()

create table dohash(

id int unsigned not null auto\_increment,

url varchar(255) not null,

url\_crc int unsigned not null default 0,

primary key (id));

delimiter //

create trigger t\_crc\_hash\_insert before insert on dohash for each row begin

set new.url\_crc=CRC32(new.url);

end;

//

delimiter //

create trigger t\_crc\_hash\_update before update on dohash for each row begin

set new.url\_crc=CRC32(new.url);

end;

//

delimiter ;

mysql> insert into dohash (url) values ('https://www.baidu.com');

Query OK, 1 row affected (0.06 sec)

mysql> select \* from dohash;

+----+-----------------------+------------+

| id | url                   | url\_crc    |

+----+-----------------------+------------+

|  1 | https://www.baidu.com | 3010065587 |

+----+-----------------------+------------+

1 row in set (0.00 sec)

mysql> update dohash set url='http://mysql.com' where id=1;

Query OK, 1 row affected (0.00 sec)

Rows matched: 1  Changed: 1  Warnings: 0

mysql> select \* from dohash;

+----+------------------+------------+

| id | url              | url\_crc    |

+----+------------------+------------+

|  1 | http://mysql.com | 2999451278 |

+----+------------------+------------+

1 row in set (0.00 sec)

alter table add index `url\_idx` (`url`);

mysql> explain select \* from dohash where url='http://mysql.com';

+----+-------------+--------+------+---------------+---------+---------+-------+------+-------------+

| id | select\_type | table  | type | possible\_keys | key     | key\_len | ref   | rows | Extra       |

+----+-------------+--------+------+---------------+---------+---------+-------+------+-------------+

|  1 | SIMPLE      | dohash | ref  | url\_idx       | url\_idx | 767     | const |    1 | Using where |

+----+-------------+--------+------+---------------+---------+---------+-------+------+-------------+

1 row in set (0.00 sec)

mysql> alter table dohash add index `urlcrc\_idx` (`url\_crc`);

Query OK, 0 rows affected (0.06 sec)

Records: 0  Duplicates: 0  Warnings: 0

mysql> explain select \* from dohash where url\_crc=crc32('http://mysql.com');

+----+-------------+--------+------+---------------+------------+---------+-------+------+-------------+

| id | select\_type | table  | type | possible\_keys | key        | key\_len | ref   | rows | Extra       |

+----+-------------+--------+------+---------------+------------+---------+-------+------+-------------+

|  1 | SIMPLE      | dohash | ref  | urlcrc\_idx    | urlcrc\_idx | 4       | const |    1 | Using where |

+----+-------------+--------+------+---------------+------------+---------+-------+------+-------------+

1 row in set (0.00 sec)

3.全文索引

a.全文索引是一种特殊类型索引，查找的是文本中的关键词，而不是直接比较索引中的值。

b.match against

### 3.索引创建的原则

**在有更多不同值的列上创建索引的选择性会更好。（切记）**

传统经验：不应该在选择性低的列上创建索引。但是！情况并非绝对。

为了能够匹配最左前缀，适当的时候可以在代码中对查询条件新增 and field in ('m','f'),这样并不会过滤任何行，不影响搜索结果。但是匹配了最左前缀索引。

但是这种技巧不能滥用，否则会带来更多的麻烦，造成查询性能降低。因为每增加一个 in()条件，优化器做的都将以指数的形式增加。

### 4.高性能索引策略

1.索引列不能是表达式的一部分，也不能是函数的参数

select user\_id from user where user\_id+1=7;

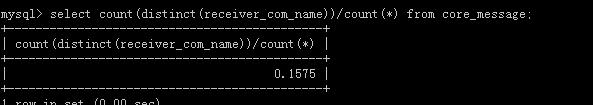
select ... from user\_login where to\_days(current\_date) - to\_days(date\_cols) <10;

**2.前缀索引和索引的选择性 :**

**缺点：mysql无法使用前缀索引做order by ,group by ,也无法做覆盖扫描**

**使用场景：很长的十六进制id,对城市等编码code做前缀索引。已达到节约索引空间，提高索引效率的目的**

 技巧：前缀的选择性接近于完整列的选择性。



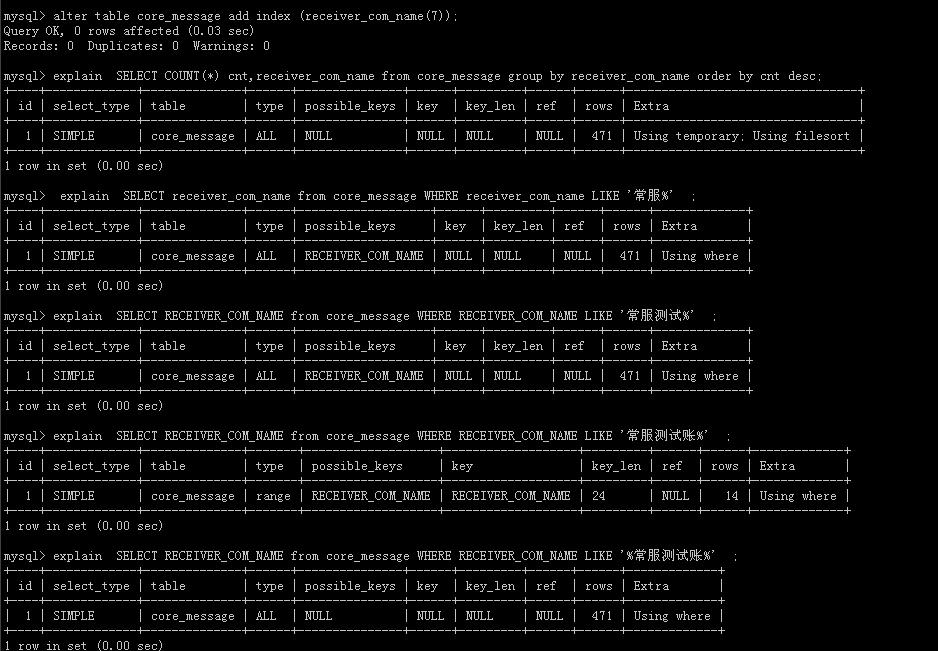
select count(distinct left(reciever\_com\_name,4))/count(\*) as sel4,

count(distinct left(reciever\_com\_name,5))/count(\*) as sel5,

count(distinct left(reciever\_com\_name,6))/count(\*) as sel6,

count(distinct left(reciever\_com\_name,7))/count(\*) as sel7

from core\_message;



**3.创建多列索引**

**误区：为每个检索的列创建单列索引**

**select film\_id,act\_id from film\_act where act\_id=1 or film\_id=2;    (单列索引)**

**(5.0) select film\_id,act\_id from film\_act where act\_id=1 union all select film\_id,act\_id from film\_act where film\_id=2;  --这样可以做到。**

**(5.0+) select film\_id,act\_id from film\_act where act\_id=1 or film\_id=2;  (idx\_act\_film\_id)**

**认知**

**1.当出现服务器对多个索引做相交操作（AND），通常意味着需要一个包含所有相关列的索引，而不是单列索引**

**2.当服务器需要对多个索引做联合操作（or）,通常需要耗费大量的CPU,内存资源在算法的缓存，排序，合并操作上。特别是当其中索引的选择性不高，需要合并扫描返回大量数据时。**

**3.更重要的是，优化器不会把这些成本计算到“查询成本”，优化器只关心随机页面读取，这会使得查询成本被低估。**

**4.索引字段顺序的合理选择 (基数选择算法)**

**select \* from wp\_comments where comment\_post\_ID =203 and comment\_parent =0;**

**mysql> select sum(comment\_post\_ID=203),sum(comment\_parent=0) from wp\_comments;**

**+--------------------------+-----------------------+**

**| sum(comment\_post\_ID=203) | sum(comment\_parent=0) |**

**+--------------------------+-----------------------+**

**|                       21 |                    2400 0|**

**+--------------------------+-----------------------+**

**-- 基数选择算法**

**select count(distinct(comment\_post\_ID)/count(\*)) as poet\_select,**

**count(distinct(comment\_parent)/count(\*)) as parent\_select,count(\*) from wp\_comments;**

**5.学会使用覆盖索引**

**概念：如果一个索引包含（或者覆盖）所有需要查询的字段的值，即覆盖索引**

**原理：Innodb的二级索引在叶子节点中保存了行的主键值，所以如果二级主键能够覆盖查询，则可以避免对主键索引的二次查询**

**mysql> alter table core\_product\_description\_search  modify column product\_desc  varchar(1000);**

**mysql> alter table core\_product\_description\_search add index `com\_desc\_idx` (`com\_id`,`product\_desc`);**

**KEY `com\_desc\_idx` (`COM\_ID`,`PRODUCT\_DESC`(255))  --查看表结构发现！**

**6.使用延迟关联：延迟对列的访问**

**mysql> explain select \* from core\_product\_description\_search join (select PRODUCT\_ID from core\_product\_description\_search where com\_id=5304 and PRODUCT\_DESC like '%d%') as t1  on (t1.PRODUCT\_ID=core\_product\_description\_search.PRODUCT\_ID);**

**+----+-------------+---------------------------------+--------+---------------------------------------------------------+--------------+---------+---------------+------+-------------+**

**| id | select\_type | table                           | type   | possible\_keys                                           | key          | key\_len | ref           | rows | Extra       |**

**+----+-------------+---------------------------------+--------+---------------------------------------------------------+--------------+---------+---------------+------+-------------+**

**|  1 | PRIMARY     | <derived2>                      | ALL    | NULL                                                    | NULL         | NULL    | NULL          |  169 |             |**

**|  1 | PRIMARY     | core\_product\_description\_search | eq\_ref | PRIMARY                                                 | PRIMARY      | 8       | t1.PRODUCT\_ID |    1 |             |**

**|  2 | DERIVED     | core\_product\_description\_search | ref    | CORE\_PRODUCT\_DESCRIPTION\_SEARCH\_COM\_ID\_IDX,com\_desc\_idx | com\_desc\_idx | 8       |               |  169 | Using where |**

**+----+-------------+---------------------------------+--------+---------------------------------------------------------+--------------+---------+---------------+------+-------------+**

**7.避免多个范围条件：**

1.对于范围条件查询，mysql无法再使用范围列后面的其他索引列了，但是对于“多个等值条件查询”，就没有这个限制。

select .... from test where **sex in ('m','f') and country in ('zhengzhou','nanjing') and last\_time >date\_sub(now(),interval 7 day)**and age between 20 and 30 ;

-- "in"为等值查询

-- ,">","<",between 为范围查询

## 2.查询性能优化

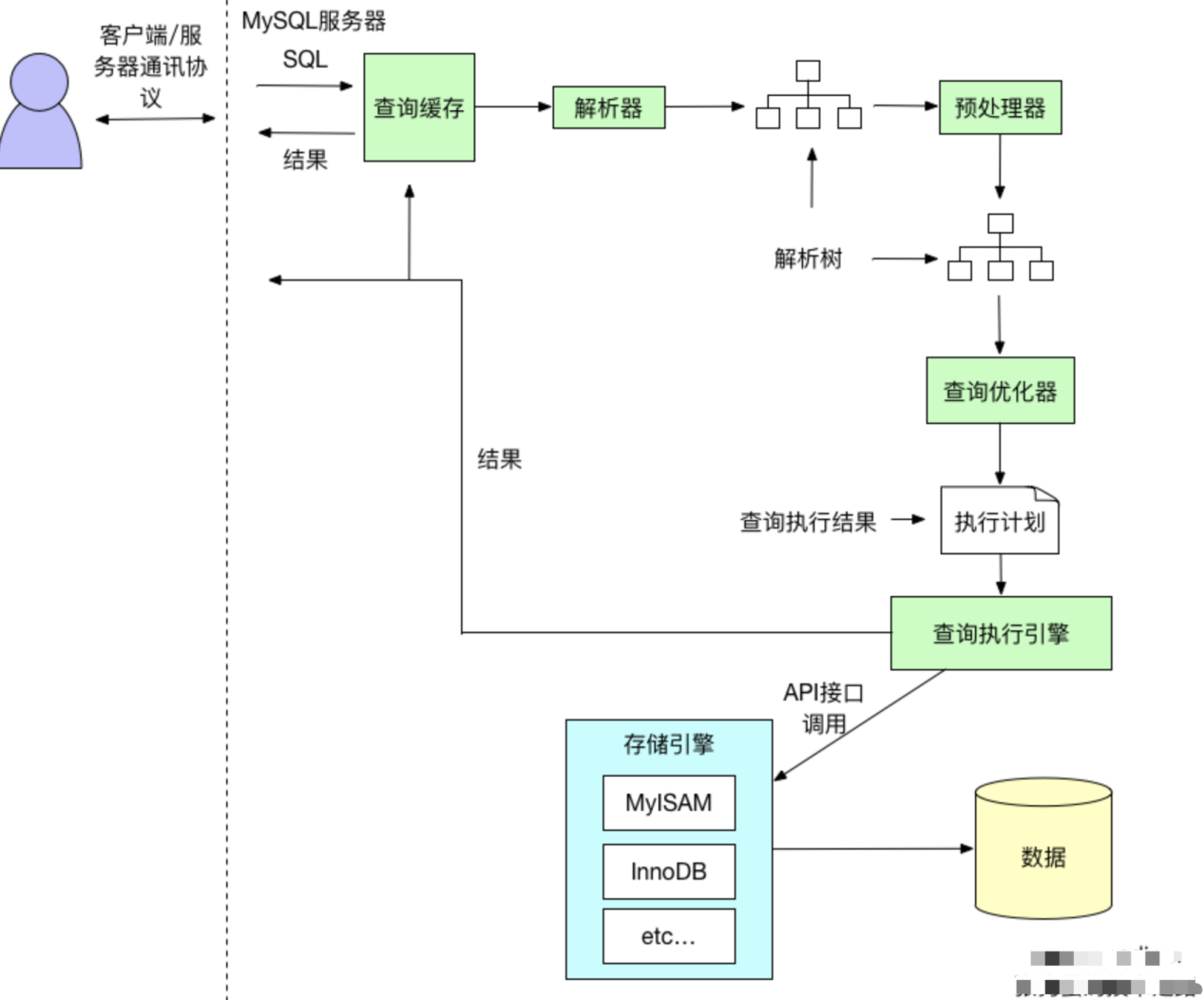
### 1.查询性能开销

响应时间 (RT)（服务时间，排队时间） 包括：网络，CPU计算，生成统计信息，执行计划，锁等待等操作

扫描行数

返回行数

SQL执行过程：



### 2.优化数据访问

1.是否请求了不需要的数据？ select \* from ......即有效数据的访问。

1)查询不需要的记录

2)多表关联返回全部列

3)总是列出全部列

2.扫描行数和返回行数

做一个关联查询时，服务器必须扫描多行才能生成结果集中的一行。

### 3.重构，切分查询

案例：delete from message where created\_time <date\_sub(now(),interval 3 month);   -- limit 10000

分解关联查询 (  让缓存效率更高，减少锁竞争，减少冗余记录查询，查询本身效率提升)

select \* from tag

join tag\_post on tag\_post.tag\_id = tag.id

join post on tag\_post.post\_id = post.id

where tag.tag='mysql';

>>>> select \* from tag where tag='mysql';

>>>> select \* from tag\_post where tag\_id=1234;

>>>> select \* from post where post.id in (123,456,567,9018);

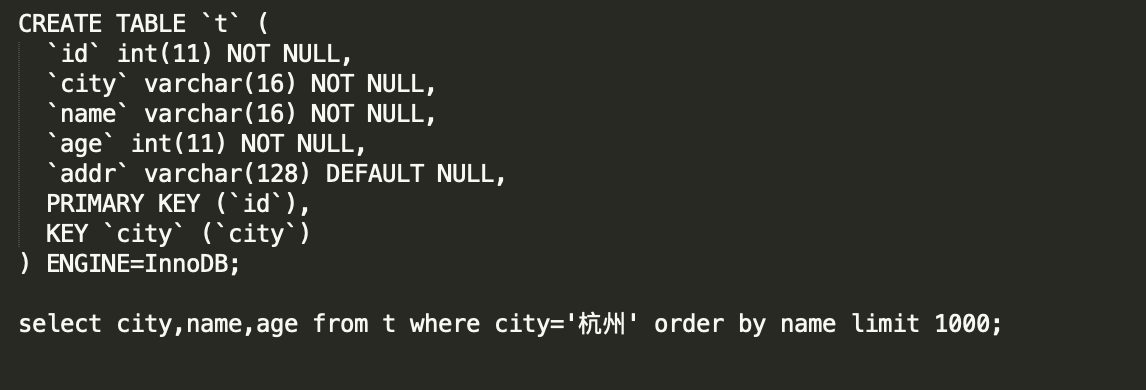
### 4.利用查询缓存

理解：通过对一个大小写敏感的哈希查找实现。 缓存命中 查询缓存是否有必要开启，需要通过MySQL统计信息的捕获观察，开发代码实现查询变更频率 。

### 优化排序

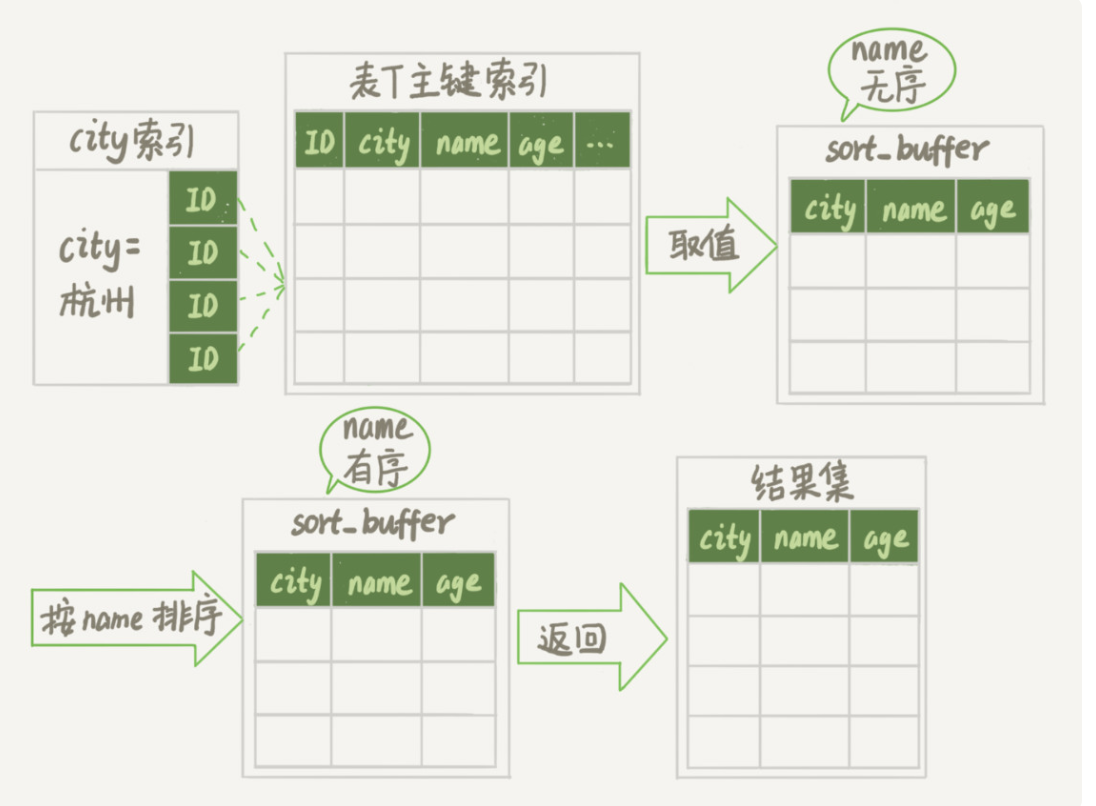
排序算法：

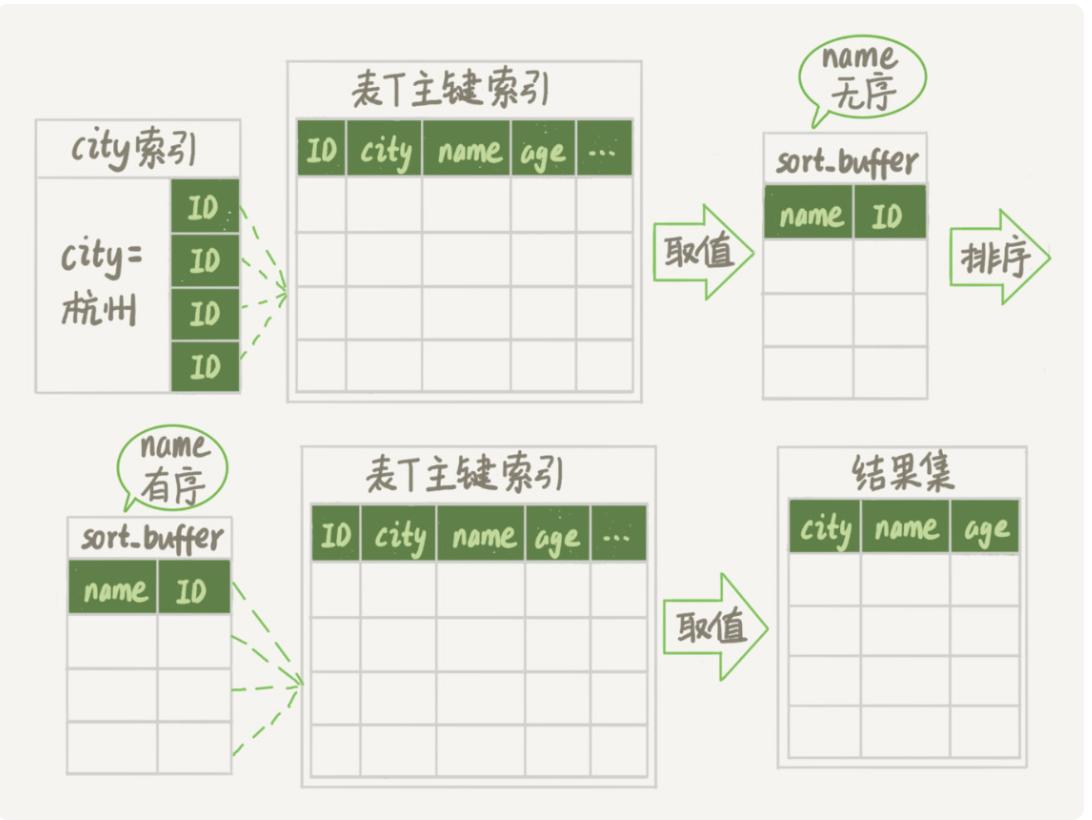
全字段排序，rowid排序

老版本：读取行指针和需要排序字段对其排序，然后根据排序结果读取所需要的所有数据行

新版本：先读取查询所需的所有列，然后根据给定列进行排序，最后直接返回排序结果 （避免二次读取）

全字段排序算法 VS Rowid排序算法





使用文件排序对小数据集会很快，但是如果一个查询匹配的结果有上百万行怎么办？

1.对于选择性非常低的列，可以增加一些特殊索引来排序：例如创建 （sex,rating）查询 select cols1 from profiles where sex='M' order by rating limit 100000,20;

2.select cols1,clos2, from profiles where sex='M' order by rating limit 100000,10; 对于偏移量增加，mysql需要花费大量时间扫描来丢弃数据，怎么办？  限制用户能够翻页的数量。（不可为一个办法）

反范式化，预先计算，缓存，这些都是解决此类查询性能的策略，但是还有一个比较好的策略>>>>>延迟关联

select <cols1> from profiles inner join (select  <primary key cols1> from profiles where x.sex='M' order by rating limit 100000,10) as x using (<primary key cols1>);

 总结：在开发系统的时候，你总是不可避免地会使用到 order by 语句。心里要清楚每个语句的排序逻辑是怎么实现的，还要能够分析出在最坏情况下，每个语句的执行对系统资源的消耗，不犯低级错误。

### 6.关联子查询

select \* from film where film\_id in (select film\_id from film\_actor where actor\_id=1);

select group\_cancat(film\_id) from film\_actor where actor\_id=1;   -- result:1,3,13,15,122,182,199,1000,1999

select \* from film where film\_id in (1,3,13,15,122,182,199,1000,1999);

先对film表进行全表扫描，然后根据返回的film\_id逐个执行子查询。 如果外层film表非常大，查询性能非常糟糕！

select \* from film inner join film\_actor using (film\_id) where actor\_id=1;

select \* from film where exists (select \* from film\_actor where actor\_id=1 and film\_actor.film\_id=film.film\_id ); --性能非常差

mysql> select count(\*) from ams\_bidlog where exists (select \* from ams\_car where flag=1 and ams\_car.carId=ams\_bidlog.auctionCarId );

+----------+

| count(\*) |

+----------+

|  3403054 |

+----------+

1 row in set (45.24 sec)

mysql> select sql\_no\_cache count(\*) from ams\_bidlog where auctionCarId in (select carId from ams\_car where flag=1);

+----------+

| count(\*) |

+----------+

|  3403054 |

+----------+

1 row in set (1.31 sec)

### 7.UNION的限制

(select first\_name,last\_name from actor order by last\_name)

union all

(select first\_name,last\_name from customer order by last\_name) limit 20;

(select first\_name,last\_name from actor order by last\_name limit 20)

union all

(select first\_name,last\_name from customer order by last\_name limit 20)

limit 20;

### 8. 特定类型查询优化

a.最大值，最小值优化

select min(actor\_id) from actor where first\_name='mysql';  -- 前提first\_name非索引

select actor\_id from actor using index (pimary) where first\_name='mysql' limit 1;

b.count()查询

理解概念：统计某个列值数量 （要求列值为非空，不会统计null值），统计行数

select count(\*) from city where id>5;

select (select count(\*) from city) - count(\*) from city where id<=5;

mysql> select count(\*) from aucs.ams\_bidlog where id >1000;

+----------+

| count(\*) |

+----------+

| 15321094 |

+----------+

1 row in set (22.93 sec)

mysql> select (select count(\*) from aucs.ams\_bidlog) - count(\*) from aucs.ams\_bidlog where id<=1000;

+---------------------------------------------------+

| (select count(\*) from aucs.ams\_bidlog) - count(\*) |

+---------------------------------------------------+

|15321094 |

+---------------------------------------------------+

1 row in set (2.22 sec)

-------------------------------------------------------------------------------------------------------

select sum(if(color='blue',1,0)) as blue,sum(if (color='red',1,0)) as red from items;

select count(color='blue' or null) as  blue,count(color='red' or null ) as red from items;

group by ,distinct优化

select actor.first\_name,actor.last\_name,count(\*) from film\_actor inner join actor using(actor\_id) group by actor.first\_name,actor.last\_name;

select actor.first\_name,actor.last\_name,count(\*) from film\_actor inner join actor using(actor\_id) group by film\_actor.actor\_id;

### 9.limit分页

select film\_id,film\_name,description from film order by title limit 10000,50;

select film.film\_id,film\_name,film.description from film inner join (select film\_id from film order by title limit 10000,50) as lim using (film\_id);

### 10.同表更新

update tb1 as outer\_tb1 set cnt=(select count(\*) from tb1 as inner\_tb1 where inner\_tb1.type= outer\_tb1.type );

update tb1 as a  inner join (select type,count(\*) cnt from tb1 group by type) as b using (type) set a.cnt=b.cnt;