

MySQL数据库优化攻略

- 目录
- 1、数据库优化简单介绍
 - 优化的目的
 - 优化方面
- 2、SQL语句优化
 - 慢查日志以及相关分析工具mysqldumpslow
 - 通过explain查询和分析SQL的执行计划
 - Count和Max的优化
 - 子查询的优化
 - Group by的优化
 - Limit查询的优化
- 3、索引优化
 - 如何选择合适的列建立索引
 - 索引优化SQL的方法
 - 索引维护的方法
- 4、数据库表结构优化
 - 选择合适的数据类型
 - 数据库表的范式化优化
 - 数据库表的反范式化优化
 - 数据库表的垂直拆分
 - 数据库表的水平拆分
- 5、系统级配置优化
 - 数据库系统配置优化
 - MySQL配置文件优化
 - 第三方配置工具使用
- 6、服务器硬件优化

数据库优化的目的

- 增加数据库稳定性
- 增加数据库处理的效率，减少应用相应的时间
- 增加用户体验
- 避免出现页面访问错误
-

数据库优化的几个方面



数据库优化的几个方面

- 结构良好的SQL和有效且合适的索引。
- 表结构的设计，根据数据库范式来设置数据表结构，简洁明了的设计，减少冗余，有益于SQL查询的写法。
- 对系统配置优化，MySQL大部分运行在Linux上且基于文件的，我们在设置方面做一些优化，比如打开文件数设置等
- 选择适合数据库的CPU（越多不一定越好），更多的内存、更快的

SQL以及索引优化

演示数据库说明：

使用MySQL提供的sakila示例数据库，可以通过以下URL地址获取

<http://dev.mysql.com/doc/index-other.html>

Example Databases

Title	Download DB	HTML Setup Guide	PDF Setup Guide
employee data (large dataset, includes data and test/verification suite)	GitHub	View	US Ltr A4
world database	Gzip Zip	View	US Ltr A4
world_x database	TGZ Zip	View	US Ltr A4
sakila database	TGZ Zip	View	US Ltr A4
menagerie database	TGZ Zip		

如何发现有问题的SQL

- 使用MySQL慢查日志对效率有问题的SQL进行监控
 - `show variables like '%slow_query_log%';`
 - `set global slow_query_log=on;`

 - `show variables like '%log_queries_not_using_indexes%';`
 - `set global log_queries_not_using_indexes=on;`

 - `set global long_query_time=1;`

慢查日志的存储格式

```
# User@Host: root[root] @ [192.168.50.201] Id: 37028
# Query_time: 10.000659 Lock_time: 0.000000 Rows_sent: 1 Rows_examined: 0
use sakila;
SET timestamp=1590646927;
select SLEEP(10);
```

慢查日志所包含的内容

执行SQL的主机信息

```
# User@Host: root[root] @ [192.168.50.201] Id: 37028
```

SQL执行的相关信息

```
# Query_time: 10.000659 Lock_time: 0.000000 Rows_sent: 1 Rows_examined: 0  
use sakila;
```

SQL执行的时间

```
SET timestamp=1590646927;
```

SQL的具体内容

```
select SLEEP(10);
```

慢查日志分析工具

- mysqldumpslow

```
root@localhost:/# mysqldumpslow -h
Option h requires an argument
ERROR: bad option

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

Parse and summarize the MySQL slow query log. Options are

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

-v            verbose
-d            debug
-s ORDER      what to sort by (al, at, ar, c, l, r, t), 'at' is default
               al: average lock time
```

慢查日志分析工具

- 使用mysqldumpslow 获取查询最慢n条记录的SQL

mysqldumpslow -t n

Eg: mysqldumpslow -t 3 localhost-slow.log

```
root@localhost:/var/lib/mysql# mysqldumpslow -t 3 localhost-slow.log

Reading mysql slow query log from localhost-slow.log
Count: 1  Time=10.00s (10s)  Lock=0.00s (0s)  Rows=1.0 (1), root[root]@[192.168.50.201]
  select SLEEP(N)

Count: 1  Time=1.34s (1s)  Lock=0.00s (0s)  Rows=1.0 (1), root[root]@[192.168.3.9]
  select sleep(N.N)

Count: 6  Time=0.10s (0s)  Lock=0.00s (0s)  Rows=3737.3 (22424), root[root]@[127.0.0.1]
  SELECT
  t.id,
  t.parent_id
```

慢查日志分析工具

- 使用mysqldumpslow 获取返回记录集最多的10个SQL
 - `mysqldumpslow -s r -t 10 localhost-slow.log`
- 使用mysqldumpslow 获取访问次数最多的10个SQL
 - `mysqldumpslow -s c -t 10 localhost-slow.log`
- 使用mysqldumpslow 获取按照时间排序的前10条里面含有左连接的查询语句
 - `mysqldumpslow -s t -t 10 -g "left join" localhost-slow.log`

使用explain查询SQL的执行计划

- explain select customer_id,first_name,last_name from customer;

```
mysql> explain select customer_id,first_name,last_name from customer;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table      | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE      | customer   | NULL       | ALL  | NULL         | NULL | NULL    | NULL | 599  | 100.00  | NULL  |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

使用explain查询SQL的执行计划

- Explain返回各列的含义
- table: 显示这一行的数据是关于哪张数据库表的
- Type : 显示了连接使用了何种类型，从最好到最差的连接类型为 const、eq_reg、ref、range、index和all
- Possible keys: 显示可能应用在这张表中的索引、如果为空，表示没有可能的索引
- Key: 实际使用的索引，如果为null，则表示没有使用索引
- ken_len: 使用的索引的长度，在不损失精度的情况下，长度越短越好
- Ref:显示索引的哪一列被使用了，如果可能的话，是一个常数
- Rows: mysql认为的用来返回请求数据，所必须要检查的行数，

使用explain查询SQL的执行计划

- Explain返回各列的含义
- Extra列需要注意的返回值
 - Using filesort: 看到这个的时候，查询就需要优化了，用了文件排序的方式，mysql需要进行步骤来发现如何对返回的值进行排序，他根据连接类型以及存储排序键值和匹配条件的全部行的行指针来排序全部行。
 - Using temporary, 看到这个的时候，表明mysql需要创建一个临时的表来存储结果，这通常发生在对不同的列集合进行order by上。

发生上述情况是使用到了外部的文件或者临时表进行了辅助查询，需要特别注意。

Count和Max的优化语句

- 查询最后支付时间：优化Max（）函数
- `select max(payment_date) from payment;`

```
mysql> explain select max(payment_date) from payment;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | payment | NULL | ALL | NULL | NULL | NULL | NULL | 16086 | 100.00 | NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

Count和Max的优化语句

- alter table payment drop index idx_paydate;
- create index idx_paydate on payment(payment_date);

```
mysql> explain select max(payment_date) from payment;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE      | NULL | NULL       | NULL | NULL          | NULL | NULL    | NULL | NULL | NULL     | Select tables optimized away |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

Count和Max的优化语句

- 优化count（）函数
- 首先说一下count（1）、count（*）、count（某列）的区别
- 以一张临时表tst2为例、对应的结果如下。

```
mysql> select * from tst2;
+-----+-----+
| id | value |
+-----+-----+
| 1 | tst1 |
| 2 | tst2 |
| 3 | NULL |
+-----+-----+
3 rows in set (0.00 sec)
```

```
mysql> select count(*),count(1),count(id),count(value) from tst2;
+-----+-----+-----+-----+
| count(*) | count(1) | count(id) | count(value) |
+-----+-----+-----+-----+
|          3 |          3 |          3 |          2 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

Count和Max的优化语句

- truncate table tst1;
- call batchInsert(500000);
- create index idx_status on tst1(status);

```
mysql> SELECT count(status) from tst1;
+-----+
| count(status) |
+-----+
|          500000 |
+-----+
1 row in set (0.16 sec)
```

```
mysql> SELECT count(content) from tst1;
+-----+
| count(content) |
+-----+
|          500000 |
+-----+
1 row in set (0.47 sec)
```

```
mysql> explain select count(content) from tst1;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tst1	NULL	ALL	NULL	NULL	NULL	NULL	490416	100.00	NULL

```
1 row in set, 1 warning (0.00 sec)
```

```
mysql> explain select count(status) from tst1;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tst1	NULL	index	NULL	idx_status	2	NULL	490416	100.00	Using index

```
1 row in set, 1 warning (0.00 sec)
```

Count和Max的优化语句

- 利用count的特性，在一条SQL中查出不同分级电影的数量
- `select count(rating='G' or null) as 'G',`
- `count(rating='PG' or null) as 'PG',`
- `count(rating='PG-13' or null) as 'PG-13',`
- `count(rating='R' or null) as 'PG',`
- `count(rating='NC-17' or null) as 'NC-17'`
- `from film;`

```
+-----+-----+-----+-----+-----+
| G     | PG    | PG-13 | PG    | NC-17 |
+-----+-----+-----+-----+-----+
| 178   | 194   | 223   | 195   | 210   |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

子查询的优化

通常情况下需要把子查询优化为join查询，但在优化是要注意是否有一对多的关系。要注意重复数据

```
explain select title,release_year,LENGTH
from film
where film_id in(
select film_id from film_actor where actor_id in
(SELECT actor_id from actor where first_name = 'sandra'))
```

Group by优化

优化groupby查询

```
explain select actor.first_name,actor.last_name,count(*)  
from film_actor  
INNER JOIN actor USING(actor_id)  
GROUP BY film_actor.actor_id
```

Group by优化

优化groupby查询：优化后

```
explain select actor.first_name,actor.last_name,c.cnt  
from actor
```

```
INNER JOIN (select actor_id,count(*) as cnt from film_actor  
GROUP BY actor_id) as c USING(actor_id);
```

Limit优化

- Limit用于分页处理，经常和order by语句配合使用，因此大多数时候会使用filesort，会造成IO问题
- explain select film_id,description from film ORDER BY title LIMIT 50,5;

```
mysql> explain select film_id,description from film ORDER BY title LIMIT 50,5;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | film | NULL | ALL | NULL | NULL | NULL | NULL | 1000 | 100.00 | Using filesort |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

Limit优化

优化情形一：使用有索引的列，或者主键进行order by操作

```
explain select film_id,description from film ORDER BY film_id  
LIMIT 50,5;
```

优化情形二：记录上次返回的主键，在下次查询是使用主键过滤

```
select film_id,description from film where film_id>55 and  
film_id<=60
```

使用该方式有个问题，需要辅助查询的列是连续的

索引优化

- 如何选择合适列建立索引？
 - 1、在where、group by、order by等从句经常出现的列建立索引
 - 2、索引字段越小越好
 - 3、离散度大的列放到联合索引的前面
 - `Select * from payment where staff_id=2 and customer_id=580`
 - Index (staff_id,customer_id) 好? 还是index (customer_id,staff_id) 好?
 - 要取决于staff_id和customer_id哪个列的离散度大, 该例子中customer_id的离散度明显大于staff_id, 所以采用customer_id在前面较好。
 - `select count(DISTINCT staff_id),count(DISTINCT customer_id) from payment;`

索引优化

- 联合索引为什么离散度大的列放前面好？
 - 这里涉及两个知识点：最左前缀，列基数cardinality
 - 最左前缀，可以理解为在复合索引中，越是左边越好
 - 因为分别对A,B,C三个列建联合索引index，实际上建立3个索引，每个索引都包含A
A,B,C
A,B
A
 - 列基数，是指它所容纳的所有非重复值的个数，可以理解为 distinct（某列）后的结果，列基数cardinality越大，重复值越少，需要建索引，因为如果某列的值都是重复的就不是特别有必要建索引了。

索引优化

以payment表为例子，在staff_id, customer_id建立联合索引

```
show index from payment;
```

```
alter table payment drop index idx_staff_customer;
```

查看对应列的列基数

```
select count(DISTINCT staff_id),count(DISTINCT customer_id) from payment;
```

```
mysql> select count(DISTINCT staff_id),count(DISTINCT customer_id) from payment;
+-----+-----+
| count(DISTINCT staff_id) | count(DISTINCT customer_id) |
+-----+-----+
|                2 |                599 |
+-----+-----+
1 row in set (0.01 sec)
```

索引优化

- create index idx_staff_customer on payment(staff_id,customer_id);

```
mysql> show index from payment;
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment
payment	0	PRIMARY	1	payment_id	A	16086		NULL	NULL	BTREE		
payment	1	idx_staff_customer	1	staff_id	A	2		NULL	NULL	BTREE		
payment	1	idx_staff_customer	2	customer_id	A	1198		NULL	NULL	BTREE		

```
3 rows in set (0.00 sec)
```

- create index idx_customer_staff on payment(customer_id,staff_id);

```
mysql> show index from payment;
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment
payment	0	PRIMARY	1	payment_id	A	16086		NULL	NULL	BTREE		
payment	1	idx_staff_customer	1	customer_id	A	599		NULL	NULL	BTREE		
payment	1	idx_staff_customer	2	staff_id	A	1198		NULL	NULL	BTREE		

```
3 rows in set (0.01 sec)
```

索引优化

最左原则示例

create index idx_customer_amount_staff on payment(customer_id,amount,staff_id);

```
mysql> show index from payment;
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_t
payment	0	PRIMARY	1	payment_id	A	16086				BTREE
payment	1	idx_staff_customer	1	customer_id	A	599				BTREE
payment	1	idx_staff_customer	2	amount	A	4812				BTREE
payment	1	idx_staff_customer	3	staff_id	A	7424				BTREE

explain Select * from payment where customer_id=580 and amount=2.99 and staff_id=2;

explain Select * from payment where customer_id=580 and amount=2.99;

explain Select * from payment where customer_id=580;

```
mysql> explain Select * from payment where customer_id=580 and amount=2.99 and staff_id=2;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	payment	NULL	ref	idx_staff_customer	idx_staff_customer	6	const,const,const	5	100.00	NULL

1 row in set, 1 warning (0.00 sec)

索引优化

最左原则示例

explain select * from payment where customer_id=580 and staff_id=2;

```
mysql> explain select * from payment where customer_id=580 and staff_id=2;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	payment	NULL	ref	idx_staff_customer	idx_staff_customer	2	const	27	10.00	Using index condition

1 row in set, 1 warning (0.00 sec)

```
mysql> select count(*) from payment where customer_id=580 and staff_id=2;
```

count(*)
13

1 row in set (0.00 sec)

select count(*) from payment where customer_id=580 and amount in (select distinct amount from payment);

```
mysql> select count(*) from payment where customer_id=580 and amount in (select distinct amount from payment);
```

count(*)
27

索引优化

最左原则示例

```
explain Select * from payment where staff_id=2 and amount=2.99;
```

```
explain Select * from payment where amount=2.99;
```

```
explain Select * from payment where staff_id=2;
```

```
mysql> explain Select * from payment where staff_id=2 and amount=2.99;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	payment	NULL	ALL	NULL	NULL	NULL	NULL	16086	1.00	Using where

```
1 row in set, 1 warning (0.00 sec)
```

所以要根据实际情况，将离散值比较大，列基数比较大，常作为查询条件的字段，放在联合索引的左边，并且索引的优化也不是一劳永逸的，要根据数据的变化以及查询场景的变化，适时的进行相应的调整。

索引优化

- 索引的维护和优化：重复以及冗余索引
 - 重复索引是指相同的列以相同的顺序建立同类型的索引，例如下表中主键和ID列上的索引就是重复索引

```
create table tst4(  
  id int not null primary key,  
  name varchar(10) not null,  
  title varchar(50) not null,  
  unique(id)  
)engine=innodb;
```

索引是不是越多也好呢？

通常认为索引会提高查询效率，但是会降低写入效率。

但是实际情况过多的索引不但会影响写入效率，有时候也会影响查询效率。

所以我们不但要学会增加合适的索引，也要适时的删除不合适的索引等。

索引优化

- 索引的维护和优化：重复以及冗余索引
 - 冗余索引是指多个索引的前缀列是相同的，或者在联合索引中包含了主键的索引，例如下面这个例子，

```
create table tst4(  
  id int not null primary key,  
  name varchar(10) not null,  
  title varchar(50) not null,  
  index idx_name_title (name,title),  
  Index idx_name (name)  
)engine=innodb;
```
 - 如果创建了索引 (a,b)，再创建索引 (a) 就是冗余索引，因为这只是前面一个索引的前缀索引，因此 (a,b) 也可以当作(a)来使用，但是 (b,a) 就不是冗余索引，索引(b)也不是，因为b不是索引 (a,b) 的最左前缀列。

索引优化

- 索引的维护和优化：重复以及冗余索引
 - 冗余索引是指多个索引的前缀列是相同的，或者在联合索引中包含了主键的索引，例如下面这个例子，

```
create table tst4(  
  id int not null primary key,  
  name varchar(10) not null,  
  title varchar(50) not null,  
  key(name,id)  
)engine=innodb;
```
 - 由于InnoDB的特性，会将主键添加到每一个索引后面，所以一种情况是将一个索引扩展为(A,ID),其中ID是主键，因为对于InnoDB来说主键已经包含在二级索引中了，所以这也是冗余的。

索引优化

- 重复以及冗余索引示例表创建语句

```
drop table tst4;  
create table tst4(  
id int not null primary key,  
name varchar(10) not null,  
title varchar(50) not null,  
index idx_name_title (name,title),  
Index idx_name (name),  
Index idx_name_id(name,id)  
)engine=innodb;
```

索引优化

- 索引的维护和优化：查找重复以及冗余索引

```
select
sta1.TABLE_SCHEMA,
sta1.TABLE_NAME,
sta1.INDEX_NAME,
sta2.INDEX_NAME,
sta1.COLUMN_NAME
from information_schema.STATISTICS as sta1
join information_schema.STATISTICS as sta2
on sta1.TABLE_SCHEMA=sta2.TABLE_SCHEMA
and sta1.TABLE_NAME=sta2.TABLE_NAME
and sta1.SEQ_IN_INDEX=sta2.SEQ_IN_INDEX
and sta1.COLUMN_NAME=sta2.COLUMN_NAME
where sta1.SEQ_IN_INDEX=1
and sta1.INDEX_NAME<>sta2.INDEX_NAME;
```

- 这个语句只能作为一个比较简单的判断方法查询前缀包含索引，并不能查找出所有的重复以及冗余索引，比如说类包含了主键的索引等。

索引优化

- 索引的维护和优化：查找重复以及冗余索引

- pt-duplicate-key-checker --help

pt-duplicate-key-checker /

-u 用户/

-p 密码/

-h IP地址

```
# #####
# sakila.tst4
# #####
# idx_name is a left-prefix of idx_name_title
# Key definitions:
#   KEY `idx_name` (`name`),
#   KEY `idx_name_title` (`name`,`title`),
# Column types:
#   `name` varchar(10) not null
#   `title` varchar(50) not null
# To remove this duplicate index, execute:
ALTER TABLE `sakila`.`tst4` DROP INDEX `idx_name`;

# Key idx_name_id ends with a prefix of the clustered index
# Key definitions:
#   KEY `idx_name_id` (`name`,`id`)
#   PRIMARY KEY (`id`),
# Column types:
#   `name` varchar(10) not null
#   `id` int(11) not null
# To shorten this duplicate clustered index, execute:
ALTER TABLE `sakila`.`tst4` DROP INDEX `idx_name_id`, ADD INDEX `idx_name_id` (`name`);
```

索引优化

- 索引的维护和优化：查找长期未使用的索引

```
select object_schema,  
object_name,  
index_name,  
b.`table_rows`  
from performance_schema.table_io_waits_summary_by_index_usage a  
join information_schema.tables b  
on a.`object_schema`=b.`table_schema`  
and a.`object_name`=b.`table_name`  
where index_name is not null  
and count_star=0  
order by object_schema,object_name;
```

- 可以作为辅助查询方法，查找长期未使用的索引，需要优化时根据实际情况具体分析。

数据库结构的优化

- 选择合适的数据类型
 - 数据类型选择，重要在于合适，如何选定选择的数据类型是否合适
 - 1、使用可存在数据的最小的数据类型
 - 2、使用简单的数据类型，int要比varchar类型在mysql处理上简单
 - 3、尽可能的使用not null的定义字段
 - 4、尽量少使用text、blob等类型，必须要使用的时候最好考虑使用附加表的方式进行处理。

数据库表结构优化

- 选择合适的数据类型
 - 使用int来储存日期格式，利用from_unixtime(),unix_timestamp()两个函数来进行转换
 - create table tst5(id int auto_increment not null,timestr int,primary key(id));
 - Insert into tst5(timestr) values (unix_timestamp('2020-06-10 22:22:22'));
 - Select from_unixtime(timestr) from tst5;

数据库表结构优化

- 选择合适的数据类型
 - 使用bigInt来存储IP地址，利用INET_ATON (),INET_NTOA ()两个函数来进行转换
 - create table tst6(id int auto_increment not null, IPaddress BIGINT,primary key(id));
 - Insert into tst6(IPaddress) values (INET_ATON('192.168.0.1'));
 - Select INET_NTOA(IPaddress) from tst6;

数据库结构优化

- 表的范式化和反范式化

- 范式化是指数据库设计的规范，一般来说符合第三设计范式的要求即可，也就是要求数据表中不能存在非关键字段对任意候选关键字段的传递依赖（在2NF基础上消除传递依赖）。

商品名称	价格	重量	有效期	分类	分类描述
可乐	3.00	250ml	2014.6	饮料	碳酸饮料
北冰洋	3.00	250ml	2014.7	饮料	碳酸饮料

- 存在以下传递依赖关系：
 - 商品名称 → 商品分类 → 分类描述
 - 存在非关键字段“分类描述”对关键字段“商品名称”的传递依赖。

数据库结构优化

- 表的范式化和反范式化
 - 不符合3NF存在表可能存在如下问题：
 - 1、数据冗余
 - 分类描述对于分类是相同的，分类描述会有很大的数据冗余
 - 2、数据的插入异常、更新异常和删除异常等。
 - 比如说如果我们删除了所有的饮料类别的数据，那么饮料这个分类以及相应的分类描述就没地方保存了。更新也是如此，如果要更新某个分类的描述，需要更新所有该分类数据的分类描述。

数据库结构优化

- 表的范式和反范式化

商品名称	价格	重量	有效期	分类	分类描述
可乐	3.00	250ml	2014.6	酒水饮料	碳酸饮料
苹果	8.00	500g		生鲜食品	水果



商品名称	价格	重量	有效期
可乐	3.00	250ml	2014.6
苹果	8.00	500g	

分类	分类描述
酒水饮料	碳酸饮料
生鲜食品	水果

分类	商品名称
酒水饮料	可乐
生鲜食品	苹果

数据库结构优化

- 表的范式化和反范式化
 - 反范式化是指为了查询效率的考虑把原本符合范式的表适当的增加冗余，用以达到优化查询效率的目的，通过空间以及更新效率等操作换取查询效率。

用户表	用户ID	姓名	电话	地址	邮编
订单表	订单ID	用户ID	下单时间	支付类型	订单状态
订单商品表	订单ID	商品ID	商品数量	商品价格	
商品表	商品ID	名称	描述	过期时间	

数据库结构优化

- 表的范式和反范式化
 - 例如查询订单信息

```
Select b.用户名,b.电话,b.地址,a.订单ID,  
       SUM(c.商品价格*c.商品数量) as 订单价格  
From 订单表 a  
Join 用户表 b on a.用户ID=b.用户ID  
Join 订单商品表 c on c.订单ID=b.订单ID  
Group by b.用户名,b.电话,b.地址,a.订单ID
```

数据库结构优化

- 表的范式化和反范式化
 - 反范式化是指为了查询效率的考虑把原本符合范式的表适当的增加冗余，用以达到优化查询效率的目的，通过空间以及更新效率等操作换取查询效率。

用户表

用户ID	姓名	电话	地址	邮编
------	----	----	----	----

订单表

订单ID	用户ID	下单时间	支付类型	订单状态	订单价格	用户名	电话	地址
------	------	------	------	------	------	-----	----	----

订单商品表

订单ID	商品ID	商品数量	商品价格
------	------	------	------

商品表

商品ID	名称	描述	过期时间
------	----	----	------

数据库结构优化

- 表的范式和反范式化
 - 例如查询订单信息

Select

a.用户名,

a.电话,

a.地址,

a.订单ID,

a.订单价格

From 订单表 a

数据库结构优化

- 表的垂直拆分
 - 所谓垂直拆分，就是把原来一个很多列的表拆分成多个表，解决表的宽度问题，通常表的垂直拆分可以按照以下原则进行
 - 1、把不常用的字段单独存放到一个表中
 - 2、把大字段独立存放到一个表中
 - 3、把经常组合使用的列放到一个表中

数据库结构优化

- 表的垂直拆分

```
CREATE TABLE `film` (  
  `film_id` smallint(5) unsigned NOT NULL,  
  `title` varchar(255) NOT NULL,  
  `description` text,  
  `release_year` year(4) DEFAULT NULL,  
  `language_id` tinyint(3) unsigned NOT NULL,  
  `original_language_id` tinyint(3) unsigned DEFAULT NULL,  
  `rental_duration` tinyint(3) unsigned NOT NULL DEFAULT '3',  
  `rental_rate` decimal(4,2) NOT NULL DEFAULT '4.99',  
  `length` smallint(5) unsigned DEFAULT NULL,  
  `replacement_cost` decimal(5,2) NOT NULL DEFAULT '19.99',  
  `rating` enum('G','PG','PG-13','R','NC-17') DEFAULT 'G',  
  `special_features` set('Trailers','Commentaries','Deleted Scenes','Behind the Scenes') DEFAULT  
  NULL,  
  `last_update` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE  
  CURRENT_TIMESTAMP,  
  PRIMARY KEY (`film_id`)  
) ENGINE=InnoDB
```

数据库结构优化

- 表的垂直拆分

```
CREATE TABLE `film` (  
  `film_id` smallint(5) unsigned NOT NULL,  
  `release_year` year(4) DEFAULT NULL,  
  `language_id` tinyint(3) unsigned NOT NULL,  
  `original_language_id` tinyint(3) unsigned DEFAULT NULL,  
  `rental_duration` tinyint(3) unsigned NOT NULL DEFAULT '3',  
  `rental_rate` decimal(4,2) NOT NULL DEFAULT '4.99',  
  `length` smallint(5) unsigned DEFAULT NULL,  
  `replacement_cost` decimal(5,2) NOT NULL DEFAULT '19.99',  
  `rating` enum('G','PG','PG-13','R','NC-17') DEFAULT 'G',  
  `special_features` set('Trailers','Commentaries','Deleted Scenes','Behind the Scenes')  
  DEFAULT NULL,  
  `last_update` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE  
  CURRENT_TIMESTAMP,  
  PRIMARY KEY (`film_id`)  
) ENGINE=InnoDB
```

数据库结构优化

- 表的垂直拆分

```
CREATE TABLE `film_file_ext` (  
  `film_id` smallint(5) unsigned NOT NULL,  
  `title` varchar(255) NOT NULL,  
  `description` text,  
  PRIMARY KEY (`film_id`)  
) ENGINE=InnoDB
```

数据库结构优化

- 表的水平拆分
 - 表的水平拆分主要是为了解决单表的数据量过大的问题，水平拆分的表，每一个表的结构是完全一致的，

数据库分区

- 查看mysql是否支持分区需要使用命令：
• `show variables like %partition%'`;

如果`have_partition_engine`的值为YES，代表和数据库支持分区。

如果empty，则说明当前mysql的版本就不支持分区

如果你使用mysql的5.6以及以后版本，同样会出现empty的结果。

5.6以及后续版本依然支持分区，只不过将上面的验证方式修改为`show plugins;` 这里会显示所有插件，如果有：

`partition ACTIVE STORAGE ENGINE GPL`插件则表明支持分区。

```
PERFORMANCE_SCHEMA | ACTIVE | STORAGE ENGINE | NULL | GPL
ARCHIVE             | ACTIVE | STORAGE ENGINE | NULL | GPL
BLACKHOLE           | ACTIVE | STORAGE ENGINE | NULL | GPL
FEDERATED           | DISABLED | STORAGE ENGINE | NULL | GPL
partition           | ACTIVE | STORAGE ENGINE | NULL | GPL
ngram               | ACTIVE | FTPARSER       | NULL | GPL
+-----+-----+-----+-----+
44 rows in set (0.00 sec)
```

数据库分区

- 水平分区（根据列属性按行分）
举个简单例子：一个包含十年发票记录的表可以被分区为十个不同的分区，每个分区包含的是其中一年的记录。
- 常用的分区模式
 - **Range**（范围）
 - **Hash**（哈希）
 - **Key**（键值）
 - **List**（预定义列表）
 - **Composite**（复合模式）

数据库分区

- **Range（范围）** – 这种模式允许将数据划分不同范围。例如可以将一个表通过年份划分成三个分区，80年代（1980's）的数据，90年代（1990's）的数据以及任何在2000年（包括2000年）后的数据。
- **Hash（哈希）** – 这种模式允许DBA通过对表的一个或多个列的Hash Key进行计算，最后通过这个Hash码不同数值对应的数据区域进行分区。例如DBA可以建立一个对表主键进行分区的表。
- **Key（键值）** – Hash模式的一种延伸，这里的Hash Key是MySQL系统产生的。
- **List（预定义列表）** – 这种模式允许系统通过DBA定义的列表的值所对应的行数据进行分割。例如：DBA建立了一个横跨三个分区的表，分别根据2004年2005年和2006年值所对应的数据。
- **Composite（复合模式）** - 是以上模式的组合使用而已。例如在初始化已经进行了Range范围分区的表上，我们可以对其中一个分区再进行hash哈希分区。

数据库分区

- 例如创建分区表

```
CREATE TABLE part_table (
```

```
  c1 int default NULL,
```

```
  c2 varchar(30) default NULL,
```

```
  c3 date default NULL
```

```
) engine=INNODB
```

```
  PARTITION BY RANGE (year(c3)) (PARTITION p0 VALUES LESS THAN (1995),
```

```
  PARTITION p1 VALUES LESS THAN (1996) , PARTITION p2 VALUES LESS THAN (1997) ,
```

```
  PARTITION p3 VALUES LESS THAN (1998) , PARTITION p4 VALUES LESS THAN (1999) ,
```

```
  PARTITION p5 VALUES LESS THAN (2000) , PARTITION p6 VALUES LESS THAN (2001) ,
```

```
  PARTITION p7 VALUES LESS THAN (2002) , PARTITION p8 VALUES LESS THAN (2003) ,
```

```
  PARTITION p9 VALUES LESS THAN (2004) , PARTITION p10 VALUES LESS THAN (2010),
```

```
  PARTITION p11 VALUES LESS THAN MAXVALUE );
```

数据库分区

- 例如创建非分区表

```
create table part_no_table (  
    c1 int(11) default NULL,  
    c2 varchar(30) default NULL,  
    c3 date default NULL  
    ) engine=INNODB;
```

批量往part_table插入数据，然后再将part_table的数据插入part_no_table

```
show create procedure load_part_table;
```

```
insert into part_no_table select * from part_table;
```

```
SELECT * from part_table LIMIT 10;
```

数据库分区

查看SQL执行计划

explain select count(*) from part_no_table where c3 > date('1995-01-01') and c3 < date ('1995-12-31');

```
mysql> explain select count(*) from part_no_table where c3 > date('1995-01-01') and c3 < date ('1995-12-31');
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table      | partitions | type | possible_keys | key | key_len | ref | rows  | filtered | Extra      |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE      | part_no_table | NULL       | ALL | NULL          | NULL | NULL    | NULL | 276360 | 11.11 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

explain select count(*) from part_table where c3 > date('1995-01-01') and c3 < date ('1995-12-31');

```
mysql> explain select count(*) from part_table where c3 > date('1995-01-01') and c3 < date ('1995-12-31');
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table      | partitions | type | possible_keys | key | key_len | ref | rows  | filtered | Extra      |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE      | part_table | p1         | ALL | NULL          | NULL | NULL    | NULL | 27887 | 11.11 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

数据库分区

- 增加筛选日起范围

explain select count(*) from part_no_table where c3 > date ('1995-01-01') and c3 < date ('1997-12-31');

```
mysql> explain select count(*) from part_no_table where c3 > date ('1995-01-01') and c3 < date ('1997-12-31');
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	part_no_table	NULL	ALL	NULL	NULL	NULL	NULL	276360	11.11	Using where

```
1 row in set, 1 warning (0.00 sec)
```

explain select count(*) from part_table where c3 > date ('1995-01-01') and c3 < date ('1997-12-31');

```
mysql> explain select count(*) from part_table where c3 > date ('1995-01-01') and c3 < date ('1997-12-31');
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	part_table	p1,p2,p3	ALL	NULL	NULL	NULL	NULL	83134	11.11	Using where

```
1 row in set, 1 warning (0.00 sec)
```

数据库分区

对比分区和未分区的文件以及索引大小

```
SELECT TABLE_NAME, CONCAT(TRUNCATE(data_length/1024/1024,2),' MB') AS
data_size,
CONCAT(TRUNCATE(index_length/1024/1024,2),' MB') AS index_size
FROM information_schema.tables WHERE TABLE_SCHEMA = 'sakila'
GROUP BY TABLE_NAME
ORDER BY data_length DESC;
```

```
mysql> SELECT TABLE_NAME, CONCAT(TRUNCATE(data_length/1024/1024,2),' MB') AS data_size,
-> CONCAT(TRUNCATE(index_length/1024/1024,2),' MB') AS index_size
-> FROM information_schema.tables WHERE TABLE_SCHEMA = 'sakila'
-> GROUP BY TABLE_NAME
-> ORDER BY data_length DESC;
```

TABLE_NAME	data_size	index_size
tst1	148.65 MB	5.51 MB
part_no_table	15.51 MB	0.00 MB
part_table	15.18 MB	0.00 MB
tst3	6.51 MB	3.51 MB
payment	1.51 MB	4.50 MB

数据库分区

对比分区和未分区的文件以及索引大小

```
create index idx_c3 on part_no_table (c3);
```

```
create index idx_c3 on part_table (c3);
```

```
mysql> SELECT TABLE_NAME, CONCAT(TRUNCATE(data_length/1024/1024,2),
LE_SCHEMA = 'sakila' GROUP BY TABLE_NAME ORDER BY data_length DESC
+-----+-----+-----+
| TABLE_NAME          | data_size | index_size |
+-----+-----+-----+
| tst1                  | 148.65 MB | 5.51 MB    |
| part_no_table         | 15.51 MB  | 5.51 MB    |
| part_table           | 15.18 MB  | 4.40 MB    |
| tst3                  | 6.51 MB   | 3.51 MB    |
| payment              | 1.51 MB   | 4.50 MB    |
| rental               | 1.51 MB   | 1.14 MB    |
| film                  | 0.18 MB   | 0.07 MB    |
```

建议方法：

分区和未分区占用文件空间大致相同（数据和索引文件）

如果查询语句中有未建立索引字段，分区时间远远优于未分区时间

如果查询语句中字段建立了索引，分区和未分区的差别缩小，分区略优于未分区。

对于大数据量，建议使用分区功能。

数据库分区

查看表分区相关信息语句

```
select partition_name part,partition_expression expr,partition_description  
descr,table_rows from information_schema.partitions where table_schema =  
schema() and table_name='part_table';
```

```
mysql> select partition_name part,partition_expression expr,  
'  
';  
+-----+-----+-----+-----+  
| part | expr      | descr  | table_rows |  
+-----+-----+-----+-----+  
| p0   | year(c3)  | 1995   |          0 |  
| p1   | year(c3)  | 1996   |        27887 |  
| p2   | year(c3)  | 1997   |        27879 |  
| p3   | year(c3)  | 1998   |        27368 |  
| p4   | year(c3)  | 1999   |        27289 |  
| p5   | year(c3)  | 2000   |        27932 |  
| p6   | year(c3)  | 2001   |        27215 |  
| p7   | year(c3)  | 2002   |        27177 |  
| p8   | year(c3)  | 2003   |        27995 |  
| p9   | year(c3)  | 2004   |        27976 |  
| p10  | year(c3)  | 2010   |        27917 |  
| p11  | year(c3)  | MAXVALUE |          0 |  
+-----+-----+-----+-----+  
rows in set (0.00 sec)
```

数据库分区

```
CREATE TABLE users (  
    uid INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,  
    name VARCHAR(30) NOT NULL DEFAULT "",  
    email VARCHAR(30) NOT NULL DEFAULT ""  
)  
PARTITION BY RANGE (uid) (  
    PARTITION p0 VALUES LESS THAN (3000000)  
    DATA DIRECTORY = '/data0/data'  
    INDEX DIRECTORY = '/data1/idx',  
  
    PARTITION p1 VALUES LESS THAN (6000000)  
    DATA DIRECTORY = '/data2/data'  
    INDEX DIRECTORY = '/data3/idx',  
  
    PARTITION p2 VALUES LESS THAN (9000000)  
    DATA DIRECTORY = '/data4/data'  
    INDEX DIRECTORY = '/data5/idx',  
  
    PARTITION p3 VALUES LESS THAN MAXVALUE DATA DIRECTORY = '/data6/data'  
    INDEX DIRECTORY = '/data7/idx'
```

RANGE 类型

在这里，将用户表分成4个分区，以每300万条记录为界限，每个分区都有自己独立的数据、索引文件的存放目录，与此同时，这些目录所在的物理磁盘分区也可以是完全独立的，可以提高磁盘IO吞吐量。

数据库分区

```
CREATE TABLE category (  
    cid INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,  
    name VARCHAR(30) NOT NULL DEFAULT ''  
)  
PARTITION BY LIST (cid) (  
    PARTITION p0 VALUES IN (0,4,8,12)  
    DATA DIRECTORY = '/data0/data'  
    INDEX DIRECTORY = '/data1/idx',  
  
    PARTITION p1 VALUES IN (1,5,9,13)  
    DATA DIRECTORY = '/data2/data'  
    INDEX DIRECTORY = '/data3/idx',  
  
    PARTITION p2 VALUES IN (2,6,10,14)  
    DATA DIRECTORY = '/data4/data'  
    INDEX DIRECTORY = '/data5/idx',  
  
    PARTITION p3 VALUES IN (3,7,11,15)  
    DATA DIRECTORY = '/data6/data'  
    INDEX DIRECTORY = '/data7/idx'
```

LIST 类型

分成4个区，根据List条件作为判断，数据文件和索引文件单独存放。

数据库分区

- 子分区

子分区是针对 RANGE/LIST 类型的分区表中每个分区的再次分割。再次分割可以是 HASH/KEY 等类型。

```
CREATE TABLE users (  
    uid INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,  
    name VARCHAR(30) NOT NULL DEFAULT "",  
    email VARCHAR(30) NOT NULL DEFAULT ""  
)  
PARTITION BY RANGE (uid) SUBPARTITION BY HASH (uid % 4) SUBPARTITIONS 2(  
    PARTITION p0 VALUES LESS THAN (3000000)  
    DATA DIRECTORY = '/data0/data'  
    INDEX DIRECTORY = '/data1/idx',  
  
    PARTITION p1 VALUES LESS THAN (6000000)  
    DATA DIRECTORY = '/data2/data'  
    INDEX DIRECTORY = '/data3/idx'  
);
```

数据库分区

- 删除分区

- `ALTER TABLE users DROP PARTITION p0;` #删除分区 p0

- 重建分区

- **RANGE** 分区重建

- `ALTER TABLE users REORGANIZE PARTITION p0,p1 INTO (PARTITION p0 VALUES LESS THAN (6000000));` #将原来的 p0,p1 分区合并起来，放到新的 p0 分区中。

- **LIST** 分区重建

- `ALTER TABLE users REORGANIZE PARTITION p0,p1 INTO (PARTITION p0 VALUES IN(0,1,4,5,8,9,12,13));` #将原来的 p0,p1 分区合并起来，放到新的 p0 分区中。

- **HASH/KEY** 分区重建

- `ALTER TABLE users REORGANIZE PARTITION COALESCE PARTITION 2;` #用 REORGANIZE 方式重建分区的数量变成2，在这里数量只能减少不能增加。想要增加可以用 ADD PARTITION 方法。

数据库分区

- 删除分区

- ALTER TABLE users DROP PARTITION p0; #删除分区 p0

- 重建分区

- **RANGE** 分区重建

- ALTER TABLE users REORGANIZE PARTITION p0,p1 INTO (PARTITION p0 VALUES LESS THAN (6000000)); #将原来的 p0,p1 分区合并起来，放到新的 p0 分区中。

- **LIST** 分区重建

- ALTER TABLE users REORGANIZE PARTITION p0,p1 INTO (PARTITION p0 VALUES IN(0,1,4,5,8,9,12,13));#将原来的 p0,p1 分区合并起来，放到新的 p0 分区中。

- **HASH/KEY** 分区重建

- ALTER TABLE users REORGANIZE PARTITION COALESCE PARTITION 2; #用 REORGANIZE 方式重建分区的数量变成2，在这里数量只能减少不能增加。想要增加可以用 ADD PARTITION 方法。

数据库分区

- 新增分区

- 新增 **RANGE** 分区

- ALTER TABLE category ADD PARTITION (PARTITION p4 VALUES IN (16,17,18,19) DATA DIRECTORY = '/data8/data' INDEX DIRECTORY = '/data9/idx');

- 新增 **HASH/KEY** 分区

- ALTER TABLE users ADD PARTITION PARTITIONS 8; #将分区总数扩展到8个。

- 给已有的表加上分区

```
alter table results partition by RANGE (month(time))  
(  
PARTITION p0 VALUES LESS THAN (3),  
PARTITION p1 VALUES LESS THAN (6),  
PARTITION p2 VALUES LESS THAN (9),  
PARTITION p3 VALUES LESS THAN (12)  
);
```

系统配置优化

- 操作系统配置优化

- 数据库是基于操作系统的，目前大多数的mysql都是安装在linux系统之上
- 所以对于操作系统的一些参数配置也会影响到数据库的性能，下面就简单列出一些常用的系统配置。
- 网络方面的配置，需要修改etc/sysctl.conf文件
- net.ipv4.tcp_max_syn_backlog = 65535
- 减少断开连接是，资源回收
- net.ipv4.tcp_max_tw_buckets = 8000
- net.ipv4.tcp_tw_reuse = 1
- net.ipv4.tcp_tw_recycle = 1
- net.ipv4.tcp_fin_timeout = 10

系统配置优化

- 操作系统配置优化

- 数据库是基于操作系统的，目前大多数的mysql都是安装在linux系统之上
- 所以对于操作系统的一些参数配置也会影响到数据库的性能，下面就简单列出一些常用的系统配置。
- 打开文件数的限制，使用
 - `ulimit -a`查看参数的值。可以修改
 - `/etc/security/limits.conf`文件，增加如下内容，用以修改打开文件数量的限制
 - `soft nfile 65535`
 - `hard nfile 65535`

`soft nproc` : 单个用户可用的最大进程数量(超过会警告);

`hard nproc`: 单个用户可用的最大进程数量(超过会报错);

`soft nfile` : 可打开的文件描述符的最大数(超过会警告);

`hard nfile` : 可打开的文件描述符的最大数(超过会报错);

- 其他的一些包括文件格式ext4、ext3、xfs等。

系统配置优化

- MySQL配置文件
 - Mysql可以通过启动时指定配置参数和使用配置文件两种方式进行配置，在大多数情况下我们使用位于/etc/my.cnf或是/etc/mysql/my.cnf位置的配置文件，window系统则是位置C:/windows/my.ini文件，MySQL配置文件查找循序可以通过以下方法获得。
 - `mysql --help | grep 'my.cnf'`
 - 如果有多个配置文件时，后面的配置文件配置项会覆盖掉前面的配置项。

系统配置优化

- MySQL配置文件
 - Innodb_buffer_pool_size
 - 非常重要的一个参数，用于配置innodb的缓冲池，如果系统中只有innodb表，则推荐配置为总内存的75%
 - `select ENGINE,ROUND(SUM(data_length+index_length)/1024/1024,1) as "Total MB" from information_schema.`TABLES` where table_schema not in("information_schema","performance_schema") GROUP BY ENGINE;`
- 允许的情况下 $\text{Innodb_buffer_pool_size} \gg \text{Total MB}$

系统配置优化

- MySQL配置文件
 - InnoDB_buffer_pool_instances
 - Mysql5.5以后版本增加的参数。可以控制缓冲池的个数，默认情况下只有一个缓冲池。
 - 我们可以把缓冲池分成多份，比如缓冲池是独占使用的，多个缓冲池可以增加并发性，减少单个缓冲池可能产生的阻塞情况等。

系统配置优化

- MySQL配置文件
 - InnoDB_log_buffer_size
 - InnoDB log 缓冲的大小，由于日志每秒都会刷新，一般不用设置太大值。

系统配置优化

- MySQL配置文件

- Innodb_flush_log_at_trx_commit

- 对innodb的IO效率影响比较大，默认值为1，可以设置0、1、2

- 值为0：提交事务的时候，不立即把 redo log buffer 里的数据刷入磁盘文件的，而是依靠 InnoDB 的主线程每秒执行一次刷新到磁盘。此时可能你提交事务了，结果 mysql 宕机了，然后此时内存里的数据全部丢失。

- 值为1：提交事务的时候，就必须把 redo log 从内存刷入到磁盘文件里去，只要事务提交成功，那么 redo log 就必然在磁盘里了。注意，因为操作系统的“延迟写”特性，此时的刷入只是写到了操作系统的缓冲区中，因此执行同步操作才能保证一定持久化到了硬盘中。

- 值为2：提交事务的时候，把 redo 日志写入磁盘文件对应的 os cache 缓存里去，而不是直接进入磁盘文件，可能 1 秒后才会把 os cache 里的数据写入到磁盘文件里去。

- 可以看到，只有1才能真正地保证事务的持久性，但是由于刷新操作 fsync() 是阻塞的，直到完成后才返回，我们知道写磁盘的速度是很慢的，因此 MySQL 的性能会明显地下降。如果不在乎事务丢失，0和2能获得更高的性能。

系统配置优化

- MySQL配置文件

- innodb_read_io_threads
- innodb_write_io_threads

- 以上参数决定了innodb读写的IO进程数，默认值为4

- 假如CPU是2颗8核的，那么可以设置：

```
innodb_read_io_threads = 8  
innodb_write_io_threads = 8
```

- 如果数据库的读操作比写操作多，那么可以设置：

```
innodb_read_io_threads = 10  
innodb_write_io_threads = 6
```

- 也就是说，你可以根据情况加以设置。

- 注意

这两个参数不支持动态改变，需要把该参数加入my.cnf里，修改完后重启MySQL服务，允许值的范围是1~64。

系统配置优化

- MySQL配置文件
 - InnoDB_file_per_table
- 在有了innodb_file_per_table参数后innodb可以把每个表的数据单独保存。单独保存有两方面的优势一个是方便管理，二个是提长性能。

服务器硬件优化

- 如何选择cpu
 - 是选择单核更快的CPU，还是选择多核CPU？
 - MySQL有一些工作只能使用一个核心进行运行，此时更快的单核心可以获得更快的处理速度。
 - 合适的CPU核心数量，例如5.5版本数据库，超过32核之后，数据库性能反而有所下降。

服务器硬件优化

- Disk IO优化
 - 使用RAID等
 - 使用RAID0将多个磁盘当成一个磁盘使用，提高IO效率。
 - 包括其他的RAID1， RAID5、 RAID10等方式。
- 使用更好的存储设备，固态硬盘等。

数据库优化的一些忠告

- 优化有风险，修改需谨慎。
- 优化不总是对一个单纯的环境进行，还很可能是一个复杂的已投产的系统。
- 优化手段本来就有很大的风险，我们尽量把风险识别到和预见到。
- 对于优化来说解决问题所带来的问题,控制在可接受的范围内才是可接受的优化成果。
- 对于优化而言，保持现状或出现更差的情况都是失败。
- 稳定性和业务可持续性,通常比性能更重要。
- 优化工作,是由业务需要驱使的！！！！
- 在数据库优化上有两个主要方面：即安全与性能。
 - 安全 ---> 数据可持续性
 - 性能 ---> 数据的高性能访问