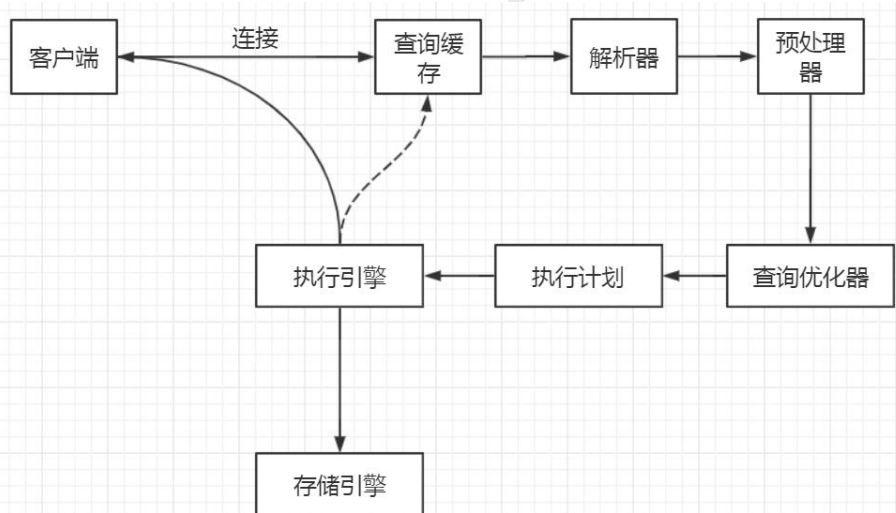


# MySQL 优化思路与工具——青山

## 1 优化思路



## 2 连接——配置优化

第一个环节是客户端连接到服务端，连接这一块有可能会出现什么样的性能问题？有可能是服务端连接数不够导致应用程序获取不到连接。比如报了一个 Mysql: error 1040: Too many connections 的错误。

可以从两个方面来解决连接数不够的问题：

1、从服务端来说，我们可以增加服务端的可用连接数。

如果有多个应用或者很多请求同时访问数据库，连接数不够的时候，我们可以：

(1) 修改配置参数增加可用连接数，修改 max\_connections 的大小：

```
show variables like 'max_connections'; -- 修改最大连接数，当有多个应用连接的时候
```

(2) 或者，或者及时释放不活动的连接。交互式和非交互式的客户端的默认超时时间

间都是 28800 秒，8 小时，我们可以把这个值调小。

```
show global variables like 'wait_timeout'; --及时释放不活动的连接，注意不要释放连接池还在使用的连接
```

2、从客户端来说，可以减少从服务端获取的连接数，如果我们想要不是每一次执行 SQL 都创建一个新的连接，可以引入连接池，实现连接的重用。

## 3 缓存——架构优化

### 3.1 缓存

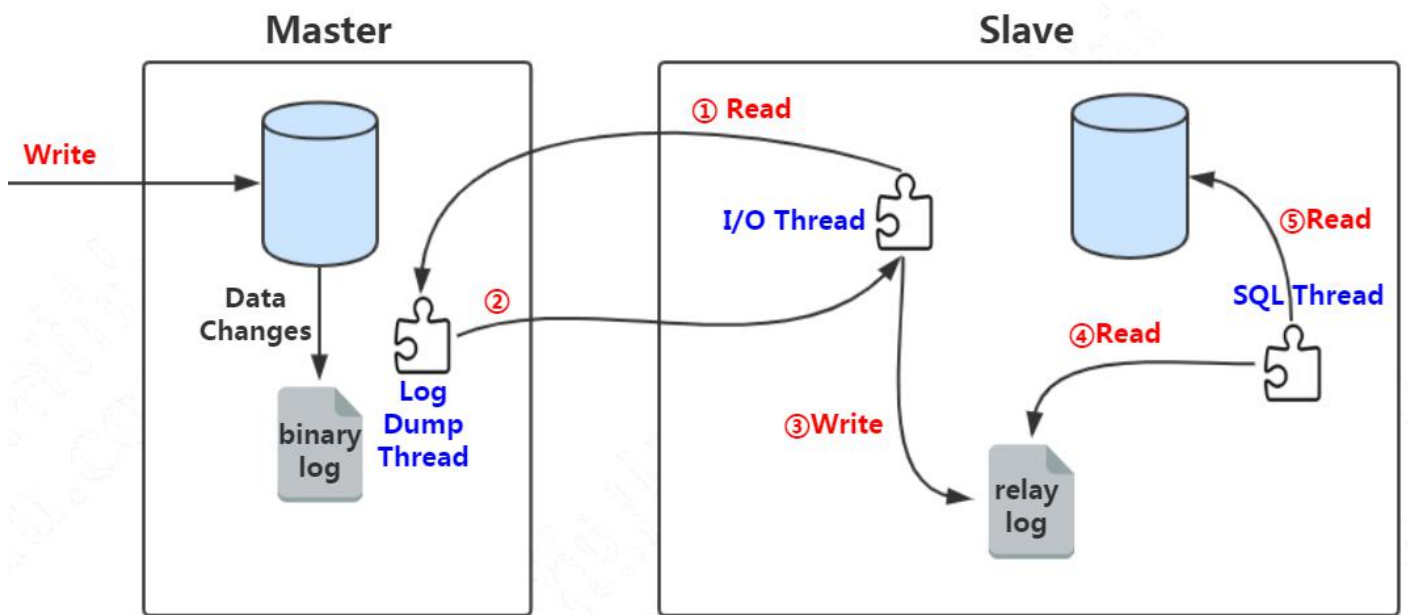
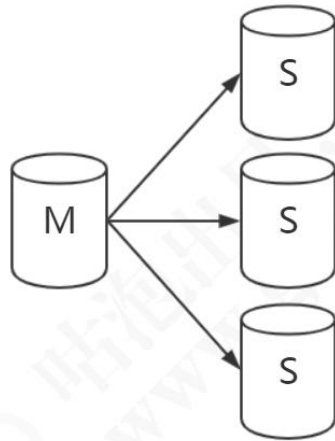
在应用系统的并发数非常大的情况下，如果没有缓存，会造成两个问题：一方面是会给数据库带来很大的压力。另一方面，从应用的层面来说，操作数据的速度也会受到影响。

我们可以用第三方的缓存服务来解决这个问题，例如 Redis。

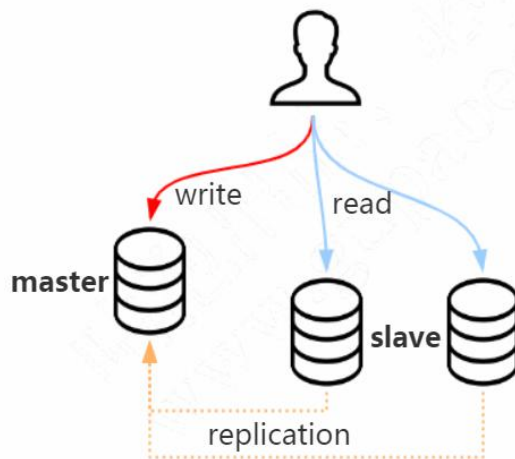


### 3.2 集群，主从复制

如果单台数据库服务满足不了访问需求，那我们可以做数据库的集群方案。



做了主从复制的方案之后，我们只把数据写入 master 节点，而读的请求可以分担到 slave 节点。我们把这种方案叫做读写分离。

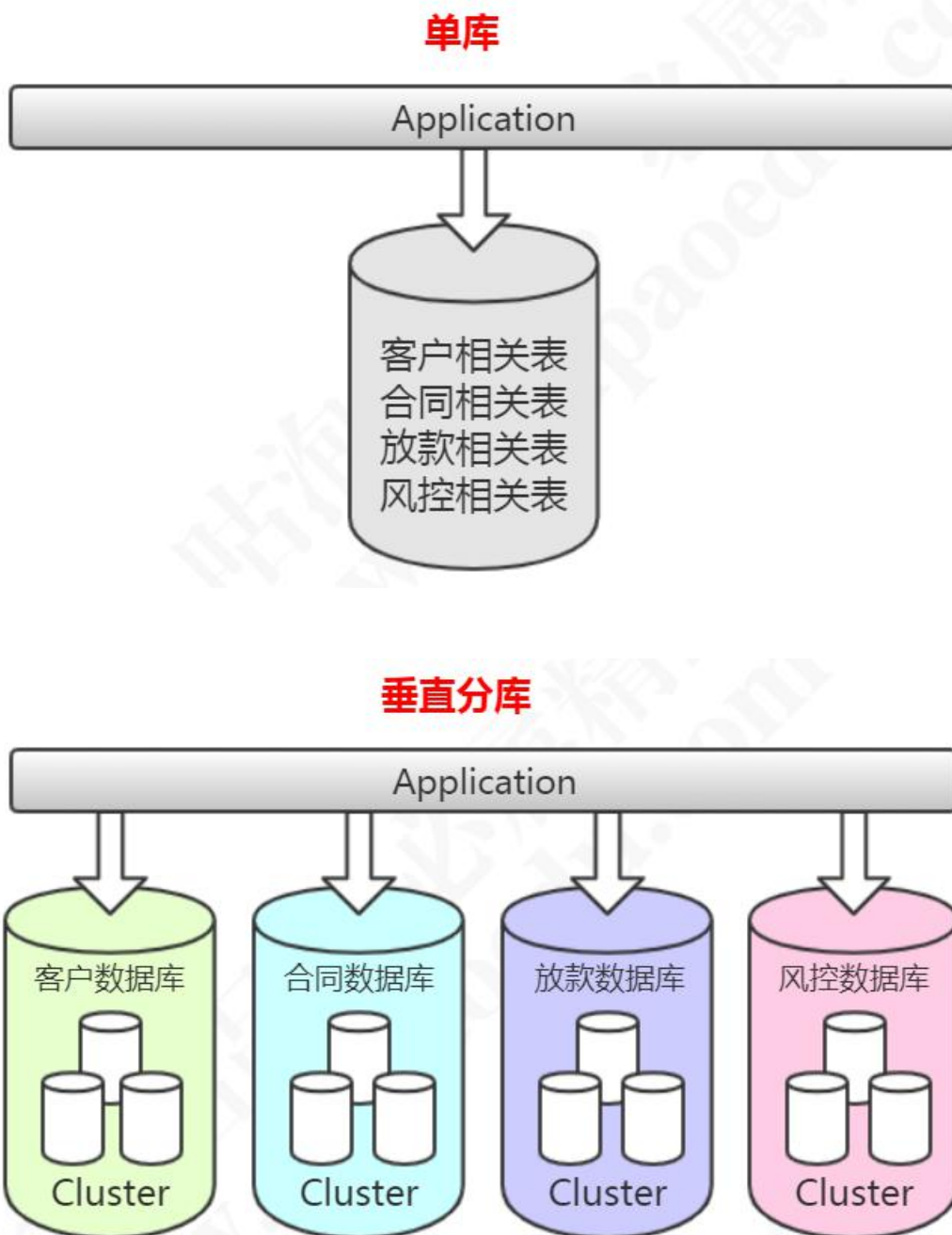


读写分离可以一定程度低减轻数据库服务器的访问压力，但是需要特别注意主从数据一致性的问题。

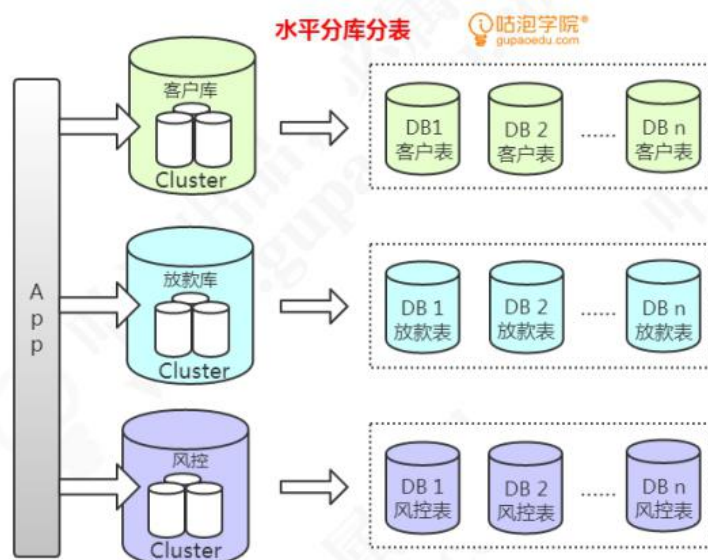
### 3.3 分库分表

垂直分库，减少并发压力。水平分表，解决存储瓶颈。

垂直分库的做法，把一个数据库按照业务拆分成不同的数据库：



水平分库分表的做法，把单张表的数据按照一定的规则分布到多个数据库。



## 4 优化器——SQL 语句分析与优化

我们的服务层每天执行了这么多 SQL 语句，它怎么知道哪些 SQL 语句比较慢呢？

第一步，我们要把 SQL 执行情况记录下来。

### 4.1 慢查询日志 slow query log

<https://dev.mysql.com/doc/refman/5.7/en/slow-query-log.html>

#### 4.1.1 打开慢日志开关

因为开启慢查询日志是有代价的（跟 bin log、optimizer-trace 一样），所以它默认是关闭的：

```
show variables like 'slow_query%';
```

Variable_name	Value
slow_query_log	ON
slow_query_log_file	/var/lib/mysql/localhost-slow.log

除了这个开关，还有一个参数，控制执行超过多长时间的 SQL 才记录到慢日志，默

认是 10 秒。如果改成 0 秒的话就是记录所有的 SQL。

```
show variables like '%long_query%';
```

可以直接动态修改参数（重启后失效）。

```
set @@global.slow_query_log=1; -- 1 开启，0 关闭，重启后失效  
set @@global.long_query_time=3; -- mysql 默认的慢查询时间是 10 秒，另开一个窗口后才会查到最新值
```

```
show variables like '%long_query%';  
show variables like '%slow_query%';
```

或者修改配置文件 my.cnf。

以下配置定义了慢查询日志的开关、慢查询的时间、日志文件的存放路径。

```
slow_query_log = ON  
long_query_time=2  
slow_query_log_file = /var/lib/mysql/localhost-slow.log
```

模拟慢查询：

```
select sleep(10);
```

查询 user\_innodb 表的 500 万数据（没有索引）。

```
SELECT * FROM `user_innodb` where phone = '136';
```

## 4.1.2 慢日志分析

### 1、日志内容

```
show global status like 'slow_queries'; -- 查看有多少慢查询  
show variables like '%slow_query%'; -- 获取慢日志目录
```



```
cat /var/lib/mysql/localhost-slow.log
```

```
# Time: 2019-12-28T12:35:52.281391Z
# User@Host: root[root] @ [192.168.8.1] Id: 64
# Query_time: 6.524766 Lock_time: 0.000145 Rows_sent: 1 Rows_examined: 5000000
SET timestamp=1577536552;
select * from user_innodb where name='青山';
```

## 2、mysqldumpslow

<https://dev.mysql.com/doc/refman/5.7/en/mysqldumpslow.html>

MySQL 提供了 `mysqldumpslow` 的工具，在 MySQL 的 `bin` 目录下。

```
mysqldumpslow --help
```

例如：查询用时最多的 10 条慢 SQL：

```
mysqldumpslow -s t -t 10 -g 'select' /var/lib/mysql/localhost-slow.log
```

```
Reading mysql slow query log from /var/lib/mysql/localhost-slow.log
Count: 1 Time=25.26s (25s) Lock=0.00s (0s) Rows=5000000.0 (5000000
92.168.8.1]
SELECT * FROM `user_innodb`

Count: 1 Time=20.87s (20s) Lock=0.00s (0s) Rows=2499866.0 (2499866
92.168.8.1]
SELECT * FROM `user_innodb` where gender = N

Count: 2 Time=9.33s (18s) Lock=0.00s (0s) Rows=1.0 (2), root[root]
select * from user_innodb where name='S'
```

Count 代表这个 SQL 执行了多少次；

Time 代表执行的时间，括号里面是累计时间；

Lock 表示锁定的时间，括号是累计；

Rows 表示返回的记录数，括号是累计。

## 4.2 SHOW PROFILE

<https://dev.mysql.com/doc/refman/5.7/en/show-profile.html>

SHOW PROFILE 是谷歌高级架构师 Jeremy Cole 贡献给 MySQL 社区的, 可以查看 SQL 语句执行的时候使用的资源, 比如 CPU、IO 的消耗情况。

在 SQL 中输入 help profile 可以得到详细的帮助信息。

### 4.2.1 查看是否开启

```
select @@profiling;  
set @@profiling=1;
```

### 4.2.2 查看 profile 统计

(命令最后带一个 s)

```
show profiles;
```

Query_ID	Duration	Query
23	0.004409	SHOW STATUS
24	0.003016	show variables like 'slow_
25	0.00428275	SHOW STATUS
26	0.00255275	SELECT QUERY_ID, SUM(C
27	0.00231375	SELECT STATE AS `状态`, F
28	0.00102425	SET PROFILING=1

查看最后一个 SQL 的执行详细信息, 从中找出耗时较多的环节 (没有 s)。

```
show profile;
```



Status	Duration
starting	4.2E-5
checking permission	9E-6
Opening tables	1.2E-5
init	3.2E-5
System lock	6E-6
optimizing	3E-6
optimizing	3E-6
statistics	8E-6
preparing	7E-6
statistics	4E-6

6.2E-5，小数点左移 5 位，代表 0.000062 秒。

也可以根据 ID 查看执行详细信息，在后面带上 for query + ID。

```
show profile for query 1;
```

#### 4.2.3 其他系统命令

分析 Server 层的运行信息，可以用 show status。

show status 服务器运行状态

说明：<https://dev.mysql.com/doc/refman/5.7/en/show-status.html>

详细参数：<https://dev.mysql.com/doc/refman/5.7/en/server-status-variables.html>

SHOW STATUS 用于查看 MySQL 服务器运行状态（重启后会清空）。

```
SHOW GLOBAL STATUS;
```

可以用 like 带通配符过滤，例如查看 select 语句的执行次数。

```
SHOW GLOBAL STATUS LIKE 'com_select'; -- 查看 select 次数
```

show processlist 运行线程

如果要分析服务层的连接信息，可以用 show processlist:

<https://dev.mysql.com/doc/refman/5.7/en/show-processlist.html>

```
show processlist;
```

这是很重要的一个命令，用于显示用户运行线程。

如果说其中的某个线程有问题，可以根据 id 号 kill 线程。

也可以查表，效果一样：

```
select * from information_schema.processlist;
```

```
mysql> show processlist;
+----+-----+-----+-----+-----+-----+-----+-----+
| Id | User | Host                | db    | Command | Time | State | Info                |
+----+-----+-----+-----+-----+-----+-----+-----+
|  2 | root | 192.168.8.1:8858    | NULL  | Sleep   | 3111 |      | NULL                |
|  3 | root | 192.168.8.1:8859    | gupao | Sleep   | 3084 |      | NULL                |
|  5 | root | 192.168.8.1:8861    | gupao | Sleep   | 2828 |      | NULL                |
|  6 | root | localhost           | NULL  | Query   |  0   | starting | show processlist  |
+----+-----+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

列	含义
Id	线程的唯一标志，可以根据它 kill 线程
User	启动这个线程的用户，普通用户只能看到自己的线程
Host	哪个 IP 端口发起的连接
db	操作的数据库
Command	线程的命令 <a href="https://dev.mysql.com/doc/refman/5.7/en/thread-commands.html">https://dev.mysql.com/doc/refman/5.7/en/thread-commands.html</a>
Time	操作持续时间，单位秒
State	线程状态，比如查询可能有 copying to tmp table, Sorting result, Sending data <a href="https://dev.mysql.com/doc/refman/5.7/en/general-thread-states.html">https://dev.mysql.com/doc/refman/5.7/en/general-thread-states.html</a>
Info	SQL 语句的前 100 个字符，如果要查看完整的 SQL 语句，用 SHOW FULL PROCESSLIST

show engine 存储引擎运行信息

<https://dev.mysql.com/doc/refman/5.7/en/show-engine.html>

<https://dev.mysql.com/doc/refman/5.7/en/innodb-standard-monitor.html>

show engine 用来显示存储引擎的当前运行信息，包括事务持有的表锁、行锁信息；

事务的锁等待情况；线程信号量等待；文件 IO 请求；buffer pool 统计信息。

例如查看 InnoDB：

```
show engine innodb status;
```

-----

现在我们已经知道哪些 SQL 慢了，为什么慢呢？慢在哪里？

MySQL 提供了一个执行计划的工具（在架构中我们有讲到，优化器最终生成的就是一个执行计划），其他数据库，例如 Oracle 也有类似的功能。

通过 EXPLAIN 我们可以模拟优化器执行 SQL 查询语句的过程，来知道 MySQL 是怎么处理一条 SQL 语句的。通过这种方式我们可以分析语句或者表的性能瓶颈。

## 4.3 EXPLAIN 执行计划

官方链接：<https://dev.mysql.com/doc/refman/5.7/en/explain-output.html>

我们先创建三张表。一张课程表，一张老师表，一张老师联系方式表（没有任何索引）。

```
DROP TABLE IF EXISTS course;
CREATE TABLE `course` (
  `cid` int(3) DEFAULT NULL,
  `cname` varchar(20) DEFAULT NULL,
  `tid` int(3) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

```
DROP TABLE IF EXISTS teacher;
CREATE TABLE `teacher` (
  `tid` int(3) DEFAULT NULL,
  `tname` varchar(20) DEFAULT NULL,
```

```
`tcid` int(3) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

```
DROP TABLE IF EXISTS teacher_contact;
CREATE TABLE `teacher_contact` (
  `tcid` int(3) DEFAULT NULL,
  `phone` varchar(200) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

```
INSERT INTO `course` VALUES ('1', 'mysql', '1');
INSERT INTO `course` VALUES ('2', 'jvm', '1');
INSERT INTO `course` VALUES ('3', 'juc', '2');
INSERT INTO `course` VALUES ('4', 'spring', '3');
```

```
INSERT INTO `teacher` VALUES ('1', 'qingshan', '1');
INSERT INTO `teacher` VALUES ('2', 'jack', '2');
INSERT INTO `teacher` VALUES ('3', 'mic', '3');
```

```
INSERT INTO `teacher_contact` VALUES ('1', '13688888888');
INSERT INTO `teacher_contact` VALUES ('2', '18166669999');
INSERT INTO `teacher_contact` VALUES ('3', '17722225555');
```

#### 4.3.1 id

id 是查询序列编号，每张表都是单独访问的，一个 SELECT 就会有一个序号。

id 值不同

id 值不同的时候，先查询 id 值大的（先大后小）。

-- 查询 mysql 课程的老师手机号

```
EXPLAIN SELECT tc.phone
FROM teacher_contact tc
WHERE tcid = (
  SELECT tcid
FROM teacher t
WHERE t.tid = (
  SELECT c.tid
FROM course c
WHERE c.cname = 'mysql'
```

```
)
);
```

查询顺序：course c——teacher t——teacher\_contact tc。

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	tc	(Null)	ALL	(Null)	(Null (Null)		(Nu	3	33.33	Using where
2	SUBQUERY	t	(Null)	ALL	(Null)	(Null (Null)		(Nu	6	16.67	Using where
3	SUBQUERY	c	(Null)	ALL	(Null)	(Null (Null)		(Nu	4	25	Using where

先查课程表，再查老师表，最后查老师联系方式表。子查询只能以这种方式进行，只有拿到内层的结果之后才能进行外层的查询。

id 值相同（从上往下）

-- 查询课程 ID 为 2，或者联系表 ID 为 3 的老师

EXPLAIN

SELECT t.tname,c.cname,tc.phone

FROM teacher t, course c, teacher\_contact tc

WHERE t.tid = c.tid

AND t.tcrid = tc.tcrid

AND (c.cid = 2

OR tc.tcrid = 3);

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t	(Null)	ALL	(Null)	(Null (Null)		(Nu	3	100	(Null)
1	SIMPLE	c	(Null)	ALL	(Null)	(Null (Null)		(Nu	4	25	Using where; Using join buffer
1	SIMPLE	tc	(Null)	ALL	(Null)	(Null (Null)		(Nu	3	33.33	Using where; Using join buffer

id 值相同时，表的查询顺序是**从上往下**顺序执行。例如这次查询的 id 都是 1（说明子查询被优化器转换成了连接查询），查询的顺序是 teacher t（3 条）——course c（4 条）——teacher\_contact tc（3 条）。

在连接查询中，先查询的叫做驱动表，后查询的叫做被驱动表，我们肯定要把小表放在前面查询，因为它的中间结果最少。

既有相同也有不同

如果 ID 有相同也有不同，就是 ID 不同的**先大后小**，ID 相同的**从上往下**。

#### 4.3.2 select type 查询类型

**这里并没有列举全部**（其它：DEPENDENT UNION、DEPENDENT SUBQUERY、MATERIALIZED、UNCACHEABLE SUBQUERY、UNCACHEABLE UNION）。

下面列举了一些常见的查询类型：

##### SIMPLE

简单查询，不包含子查询和关联查询 union。

```
EXPLAIN SELECT * FROM teacher;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	<b>SIMPLE</b>	teacher	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	100	(Null)

再看一个包含子查询的案例：

```
-- 查询 mysql 课程的老师手机号
EXPLAIN SELECT tc.phone
FROM teacher_contact tc
WHERE tcid = (
  SELECT tcid
  FROM teacher t
  WHERE t.tid = (
    SELECT c.tid
    FROM course c
    WHERE c.cname = 'mysql'
  )
);
```



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	tc	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	33.33	Using where
2	SUBQUERY	t	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	33.33	Using where
3	SUBQUERY	c	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	4	25	Using where

## PRIMARY

子查询 SQL 语句中的主查询，也就是最外面的那层查询。

## SUBQUERY

子查询中所有的内层查询都是 SUBQUERY 类型的。

## DERIVED

派生查询，表示在得到最终查询结果之前会用到临时表。例如：

```
-- 查询 ID 为 1 或 2 的老师教授的课程
EXPLAIN SELECT cr.cname
FROM (
  SELECT * FROM course WHERE tid = 1
  UNION
  SELECT * FROM course WHERE tid = 2
) cr;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	<derived2>	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	4	100	(Null)
2	DERIVED	course	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	4	25	Using where
3	UNION	course	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	4	25	Using where
(N)	UNION RESULT	<union2,3>	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	Using temporary

对于关联查询，先执行右边的 table（UNION），再执行左边的 table，类型是 DERIVED。

## UNION

用到了 UNION 查询（UNION 会用到内部的临时表）。同上例。

UNION ALL 不需要去重，因此不用临时表。

## UNION RESULT

主要是显示哪些表之间存在 UNION 查询。<union2,3>代表 id=2 和 id=3 的查询存在 UNION。同上例。

```
EXPLAIN SELECT cr.cname
FROM (
SELECT * FROM course WHERE tid = 1
UNION ALL
SELECT * FROM course WHERE tid = 2
) cr;
```

### 4.3.3 type 访问方法

<https://dev.mysql.com/doc/refman/5.7/en/explain-output.html#explain-join-types>

所有的连接类型中，上面的最好，越往下越差。

在常用的链接类型中：system > const > eq\_ref > ref > range > index > all

这里并没有列举全部（其他：fulltext、ref\_or\_null、index\_merger、unique\_subquery、index\_subquery）。

以上访问类型除了 all，都能用到索引。

#### const

主键索引或者唯一索引与常数进行等值匹配，只能查到一条数据的 SQL。

```

DROP TABLE IF EXISTS single_data;
CREATE TABLE single_data(
  id int(3) PRIMARY KEY,
  content varchar(20)
);
insert into single_data values(1,'a');

EXPLAIN SELECT * FROM single_data a where id = 1;

```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered
1	SIMPLE	a	(Null)	const	PRIMARY	PRIMARY	4	const	1	100

## system

system 是 const 的一种特例，只有一行满足条件，对于 MyISAM、Memory 的表，只查询到一条记录，也是 system。

例如：只有一条数据的系统表。

```
EXPLAIN SELECT * FROM mysql.proxies_priv;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered
1	SIMPLE	proxies_priv	(Null)	system	(Null)	(Null (Null))	(Null (Null))	(Null)	1	100

## eq\_ref

通常出现在多表的 join 查询，被驱动表通过唯一性索引（UNIQUE 或 PRIMARY KEY）进行访问，此时被驱动表的访问方式就是 eq\_ref。

eq\_ref 是除 const 之外最好的访问类型。

先删除 teacher 表中多余的数据，teacher\_contact 有 3 条数据，teacher 表有 3 条数据。

```

DELETE FROM teacher where tid in (4,5,6);
commit;

```

-- 备份

```
INSERT INTO `teacher` VALUES (4, 'james', 4);
INSERT INTO `teacher` VALUES (5, 'tom', 5);
INSERT INTO `teacher` VALUES (6, 'seven', 6);
commit;
```

为 teacher\_contact 表的 tcid (第一个字段) 创建主键索引。

```
-- ALTER TABLE teacher_contact DROP PRIMARY KEY;
ALTER TABLE teacher_contact ADD PRIMARY KEY(tcid);
```

为 teacher 表的 tcid (第三个字段) 创建普通索引。

```
-- ALTER TABLE teacher DROP INDEX idx_tcid;
ALTER TABLE teacher ADD INDEX idx_tcid (tcid);
```

执行以下 SQL 语句：

```
select t.tcid from teacher t,teacher_contact tc where t.tcid = tc.tcid;
```

tcid
1
2
3

此时的执行计划 (teacher\_contact 表是 eq\_ref)：

select_type	table	partitions	type	possible_keys	key	key_len	ref	rows
SIMPLE	t	(Null)	index	idx_tcid	idx_tcid	5	(Null)	3
SIMPLE	tc	(Null)	eq_ref	PRIMARY	PRIMARY	4	gupao	1

**小结：**

以上三种 system, const, eq\_ref, 都是可遇而不可求的，基本上很难优化到这个状态。

ref

查询用到了非唯一性索引。

例如：使用 tcid 上的普通索引查询：

```
explain SELECT * FROM teacher where tcid = 3;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered
1	SIMPLE	teacher	(Null)	ref	idx_tid	idx_tid	5	const	1	100

## range

对索引进行范围扫描。

如果 where 后面是 between and 或 <或 > 或 >= 或 <=或 in 这些, type 类型就为 range。

不走索引一定是全表扫描 (ALL)，所以先加上普通索引。

```
-- ALTER TABLE teacher DROP INDEX idx_tid;
ALTER TABLE teacher ADD INDEX idx_tid (tid);
```

执行范围查询（字段上有普通索引）：

```
EXPLAIN SELECT * FROM teacher t WHERE t.tid <3;
-- 或
EXPLAIN SELECT * FROM teacher t WHERE tid BETWEEN 1 AND 2;
```

id	select_type	table	partitions	type	possible_keys	key	key_len
1	SIMPLE	t	(Null)	range	idx_tid	idx_tid	5

IN 查询也是 range（字段有主键索引）

```
EXPLAIN SELECT * FROM teacher_contact t WHERE tcid in (1,2,3);
```

id	select_type	table	partitions	type	possible_keys	key	key_len
1	SIMPLE	t	(Null)	range	PRIMARY	PRIMARY	4

## index

Full Index Scan，查询全部索引中的数据（比不走索引要快）。

```
EXPLAIN SELECT tid FROM teacher;
```

id	select_type	table	partitions	type	possible_keys	key	key_len
1	SIMPLE	teacher	(Null)	index	(Null)	idx_tid	5

## ALL

Full Table Scan，如果没有索引或者没有用到索引，type 就是 ALL。代表全表扫描。

### 小结：

一般来说，需要保证查询至少达到 range 级别，最好能达到 ref。

ALL（全表扫描）和 index（查询全部索引）都是需要优化的。

### 4. 3. 4 possible\_key、key

可能用到的索引和实际用到的索引。如果是 NULL 就代表没有用到索引。

possible\_key 可以有一个或者多个，比如查询多个字段上都有索引，或者一个字段同时有单列索引和联合索引。

能用到的索引并不是越多越好。可能用到索引不代表一定用到索引。

如果通过分析发现没有用到索引，就要检查 SQL 或者创建索引。

### 4. 3. 5 key\_len

索引的长度（使用的字节数）。跟索引字段的类型、长度有关。

表上有联合索引：KEY `comidx\_name\_phone` (`name`,`phone`)



```
explain select * from user_innodb where name ='青山';
```

key\_len = 1023, 为什么不是  $255 + 11 = 266$  呢?

这里的索引只用到了 name 字段, utf8mb4 编码 1 个字符 4 个字节。所以是  $255 * 4 = 1020$ 。使用变长字段 varchar 需要额外增加 2 个字节, 允许 NULL 需要额外增加 1 个字节。一共是 1023。

#### 4.3.6 rows

MySQL 认为扫描多少行 (数据或者索引) 才能返回请求的数据, 是一个预估值。一般来说行数越少越好。

#### 4.3.7 filtered

这个字段表示存储引擎返回的数据在 server 层过滤后, 剩下多少满足查询的记录数量的比例, 它是一个百分比。

如果比例很低, 说明存储引擎层返回的数据需要经过大量过滤, 这个是会消耗性能的, 需要关注。

#### 4.3.8 ref

使用哪个列或者常数和索引一起从表中筛选数据, 可以参考一下。

#### 4.3.9 Extra

执行计划给出的额外的信息说明。

using index

属于覆盖索引的情况, 不需要回表。

```
EXPLAIN SELECT tid FROM teacher ;
```

using where

使用了 where 过滤，表示存储引擎返回的记录并不是所有的都满足查询条件，需要在 server 层进行过滤（跟是否使用索引没有关系）。

```
EXPLAIN select * from user_innodb where phone ='13866667777';
```

select_type	table	type	possible key	key_len	ref	rows	filtered	Extra
SIMPLE	user_innodb	ALL	(Null)	(Null)	(Null)	4646619	10	Using where

using Index Condition

索引下推，昨天的课已经讲过了。

using filesort

不能使用索引来排序，用到了额外的排序（跟磁盘或文件没有关系）。需要优化。

（复合索引的前提）

```
ALTER TABLE user_innodb DROP INDEX comidx_name_phone;
ALTER TABLE user_innodb add INDEX comidx_name_phone (name,phone);
```

执行 SQL:

```
EXPLAIN select * from user_innodb where name ='青山' order by id;
```

（order by id 引起）

select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
SIMPLE	user_innodb	ref	comidx_name_phone	comidx_1023	const	1	100	Using index condition;	Using filesort

using temporary

在查询的时候，需要做去重、排序之类的工作的时候，可能会用到临时表。

举几个例子：

### 1、distinct 非索引列

```
EXPLAIN select DISTINCT(tid) from teacher t;
```

### 2、group by 非索引列

```
EXPLAIN select tname from teacher group by tname;
```

### 3、使用join 的时候，group 任意列

```
EXPLAIN select t.tid from teacher t join course c on t.tid = c.tid group by t.tid;
```

Using Temporary 需要优化，例如创建复合索引。

总结一下：

模拟优化器执行 SQL 查询语句的过程，来知道 MySQL 是怎么处理一条 SQL 语句的。

通过这种方式我们可以分析语句或者表的性能瓶颈。

如果需要具体的 cost 信息，可以用：

EXPLAIN FORMAT=JSON。

如果觉得 EXPLAIN 还不够详细，可以用开启 optimizer trace。

## 4.4 SQL 与索引优化

SQL 语句的优化的目标，大部分时候都是用到索引。

对于每一种具体的 SQL，也有相应的优化方案，官网：

- ▼ Optimization
  - Optimization Overview
  - ▼ Optimizing SQL Statements
    - ▼ Optimizing SELECT Statements

## 5 存储引擎

### 5.1 存储引擎的选择

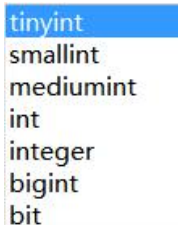
为不同的业务表选择不同的存储引擎, 例如: 查询插入操作多的业务表, 用 MyISAM。临时数据用 Memeroy。常规的并发大更新多的表用 InnoDB。

### 5.2 字段定义

原则: 使用可以正确存储数据的最小数据类型。

为每一列选择合适的字段类型。

#### 5.2.1 整数类型



tinyint  
smallint  
mediumint  
int  
integer  
bigint  
bit

INT 有 8 种类型, 不同的类型的最大存储范围是不一样的。

性别? 用 TINYINT, 因为 ENUM 也是整数存储。

#### 5.2.2 字符类型

变长情况下, varchar 更节省空间, 但是对于 varchar 字段, 需要一个字节来记录长度。

固定长度的用 char, 不要用 varchar。

### 5.2.3 不要用外键、触发器、视图

降低了可读性；

影响数据库性能，应该把计算的事情交给程序，数据库专心做存储；

数据的完整性应该在程序中检查。

### 5.2.4 大文件存储

不要用数据库存储图片（比如 base64 编码）或者大文件；

把文件放在 NAS 上，数据库只需要存储 URI（相对路径），在应用中配置 NAS 服务器地址。

### 5.2.5 表拆分或字段冗余

将不常用的字段拆分出去，避免列数过多和数据量过大。

比如在业务系统中，要记录所有接收和发送的消息，这个消息是 XML 格式的，用 blob 或者 text 存储，用来追踪和判断重复，可以建立一张表专门用来存储报文。

## 6 总结：优化体系

所以，如果在面试的时候再问到这个问题“你会从哪些维度来优化数据库”，你会怎么回答？



除了对于代码、SQL 语句、表定义、架构、配置优化之外，业务层面的优化也不能

忽视。举两个例子：

1) 在某一年的双十一，为什么会做一个充值到余额宝和余额有奖金的活动，例如充 300 送 50？

因为使用余额或者余额宝付款是记录本地或者内部数据库，而使用银行卡付款，需要调用接口，操作内部数据库肯定更快。

2) 在去年的双十一，为什么在凌晨禁止查询今天之外的账单？

这是一种降级措施，用来保证当前最核心的业务。

3) 最近几年的双十一，为什么提前个把星期就已经有双十一当天的价格了？

预售分流。

在应用层面同样有很多其他的方案来优化，达到尽量减轻数据库的压力的目的，比如限流，或者引入 MQ 削峰，等等等等。

为什么同样用 MySQL，有的公司可以抗住百万千万级别的并发，而有的公司几百个并发都扛不住，关键在于怎么用。所以，用数据库慢，不代表数据库本身慢，有的时候还要往上层去优化。

当然，如果关系型数据库解决不了的问题，我们可能需要用到搜索引擎或者大数据的方案了，并不是所有的数据都要放到关系型数据库存储。

这是优化的层次，如果说遇到的一个具体的慢 SQL 的问题，我们又应该怎么去优化呢？比如有人给你发来一条 SQL，你应该怎么分析？

一个案例：

一个 sum 语句性能提升 3 倍的优化案例

<https://gper.club/articles/7e7e7f7ff4g5egc2g63>



## 一、分析查询基本情况

1、涉及到表结构，字段的索引情况、每张表的数据量、查询的业务含义。

这个非常重要，因为有的时候你会发现 SQL 根本没必要这么写，或者表设计是有问题的。

## 二、找出慢的原因

1、查看执行计划，分析 SQL 的执行情况，了解表访问顺序、访问类型、索引、扫描行数等信息。

2、如果总体的时间很长，不确定哪一个因素影响最大，通过条件的增减，顺序的调整，找出引起查询慢的主要原因，不断地尝试验证。

找到原因：比如是没有走索引引起的，还是关联查询引起的，还是 order by 引起的。

找到原因之后：

## 三、对症下药

1、创建索引或者联合索引

2、改写 SQL，这里需要平时积累经验，例如：

1) 使用小表驱动大表

2) 用 join 来代替子查询

3) not exist 转换为 left join IS NULL

4) or 改成 union

4) 使用 UNION ALL 代替 UNION，如果结果集允许重复的话

5) 大偏移的 limit，先过滤再排序。

如果 SQL 本身解决不了了，就要上升到表结构和架构了。

3、表结构（冗余、拆分、not null 等）、架构优化。

4、业务层的优化，必须条件是否必要。

如果没有思路，调优就是抓瞎，肯定没有任何头绪。

赶集网 mysql 开发 36 军规

<https://gper.club/articles/7e7e7f7ff4g5agc2g66>

转载 58 到家 MySQL 军规升级版

<https://gper.club/articles/7e7e7f7ff4g5agc2g67>