课程目标

掌握 MySQL 数据库优化的层次和思路

掌握 MySQL 数据库优化的工具。

内容定位

适合学习了 MySQL 架构、MySQL 索引、MySQL 事务的同学。

集中答疑链接：https://gper.club/articles/7e7e7f7ff7g55gc9g6b

1 优化思路

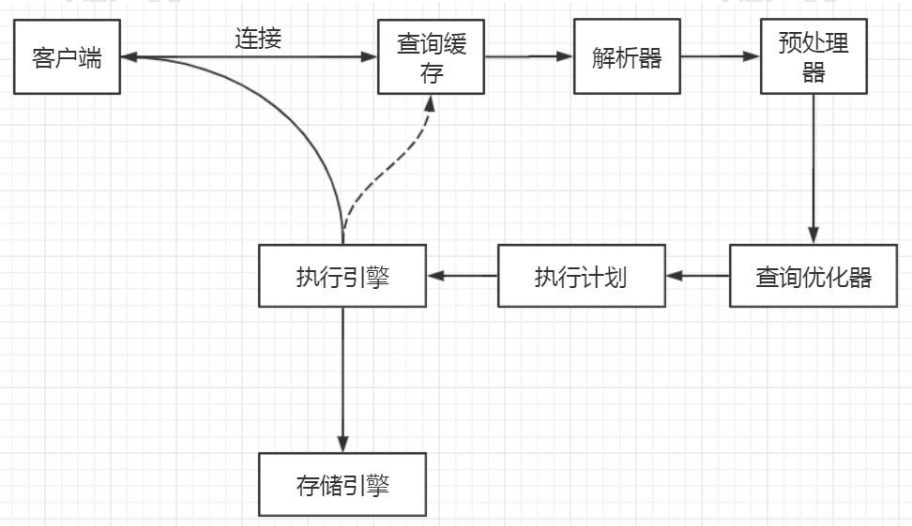
作为架构师或者开发人员，说到数据库性能优化，你的思路是什么样的？

或者具体一点，如果在面试的时候遇到这个问题：你会从哪些维度来优化数据库，你会怎么回答？

我们在第一节课开始的时候讲了，这四节课的目标是为了让大家建立数据库的知识体系，和正确的调优的思路。

我们说到性能调优，大部分时候想要实现的目标是让我们的查询更快。一个查询的动作又是由很多个环节组成的，每个环节都会消耗时间，我们在第一节课讲 SQL 语句的执行流程的时候已经分析过了。

我们要减少查询所消耗的时间，就要从每一个环节入手。



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。

我们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，应该怎么做？

这个时候我们可以引入连接池，实现连接的重用。

我们可以在哪些层面使用连接池？ORM 层面（MyBatis 自带了一个连接池）；或者使用专用的连接池工具（阿里的 Druid、Spring Boot 2.x 版本默认的连接池 Hikari、老牌的 DBCP 和 C3P0）。

当客户端改成从连接池获取连接之后，连接池的大小应该怎么设置呢？大家可能会有一个误解，觉得连接池的最大连接数越大越好，这样在高并发的情况下客户端可以获取的连接数更多，不需要排队。

实际情况并不是这样。连接池并不是越大越好，只要维护一定数量大小的连接池，其他的客户端排队等待获取连接就可以了。有的时候连接池越大，效率反而越低。

Druid 的默认最大连接池大小是 8。Hikari 的默认最大连接池大小是 10。

为什么默认值都是这么小呢？

在 Hikari 的 github 文档中 ,给出了一个 PostgreSQL 数据库建议的设置连接池大小的公式：

https://github.com/brettwooldridge/HikariCP/wiki/About-Pool-Sizing

它的建议是机器核数乘以 2 加 1。也就是说，4 核的机器，连接池维护 9 个连接就够了。这个公式从一定程度上来说对其他数据库也是适用的。这里面还有一个减少连接池大小实现提升并发度和吞吐量的案例。

为什么有的情况下，减少连接数反而会提升吞吐量呢？为什么建议设置的连接池大小要跟 CPU 的核数相关呢？

每一个连接，服务端都需要创建一个线程去处理它。连接数越多，服务端创建的线

程数就会越多。

问题：CPU 是怎么同时执行远远超过它的核数大小的任务的？时间片。上下文切换。

而 CPU 的核数是有限的，频繁的上下文切换会造成比较大的性能开销。

我们这里说到了从数据库配置的层面去优化数据库。不管是数据库本身的配置，还是安装这个数据库服务的操作系统的配置，对于配置进行优化，最终的目标都是为了更好地发挥硬件本身的性能，包括 CPU、内存、磁盘、网络。

在不同的硬件环境下 ,操作系统和 MySQL 的参数的配置是不同的 ,没有标准的配置。

在我们这几天的课程里面也接触了很多的 MySQL 和 InnoDB 的配置参数 ,包括各种开关和数值的配置，大多数参数都提供了一个默认值，比如默认的 buffer_pool_size，默认的页大小，InnoDB 并发线程数等等。

这些默认配置可以满足大部分情况的需求，除非有特殊的需求，在清楚参数的含义的情况下再去修改它。修改配置的工作一般由专业的 DBA 完成。

至于硬件本身的选择，比如使用固态硬盘，搭建磁盘阵列，选择特定的 CPU 型号这些，更不是我们开发人员关注的重点，这个我们就不做过多的介绍了。

如果想要了解一些特定的参数的含义，官网有一份系统的参数列表可以参考：

https://dev.mysql.com/doc/refman/5.7/en/server-system-variables.html

除了合理设置服务端的连接数和客户端的连接池大小之外，我们还有哪些减少客户端跟数据库服务端的连接数的方案呢？

我们可以引入缓存。

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。

运行独立的缓存服务，属于架构层面的优化。

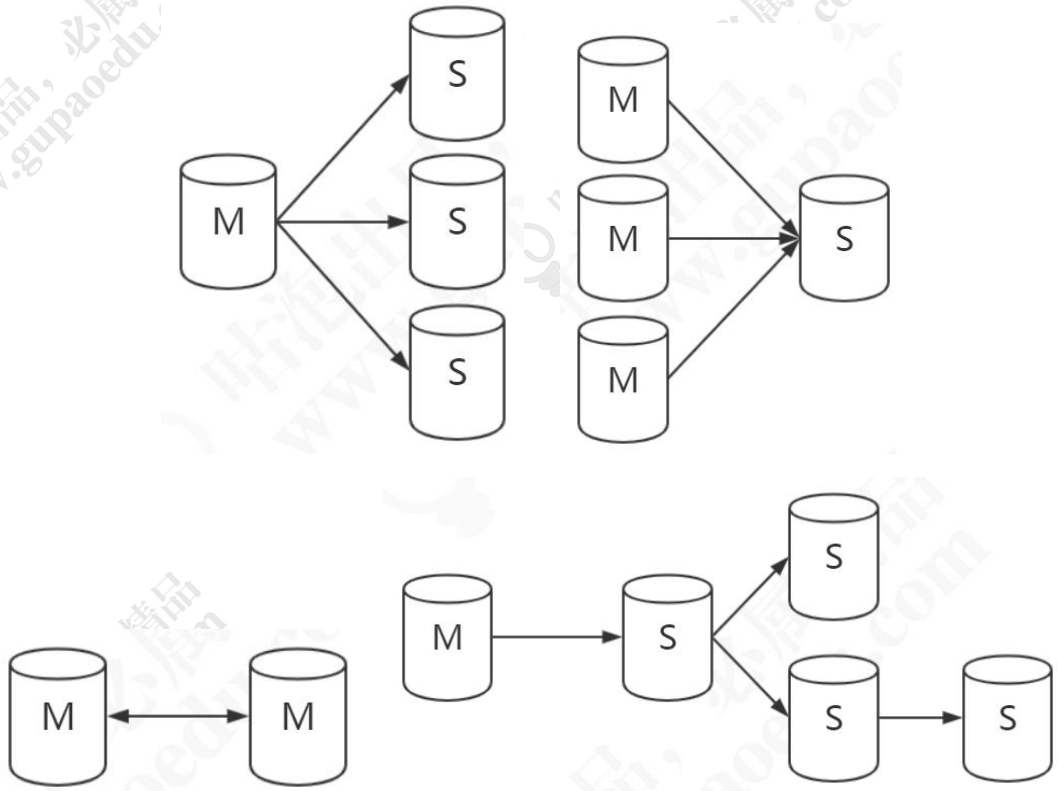
为了减少单台数据库服务器的读写压力，在架构层面我们还可以做其他哪些优化措施？

3.2 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。

集群的话必然会面临一个问题，就是不同的节点之间数据一致性的问题。如果同时读写多台数据库节点，怎么让所有的节点数据保持一致？

这个时候我们需要用到复制技术（replication），被复制的节点称为 master，复制的节点称为 slave。slave 本身也可以作为其他节点的数据来源，这个叫做级联复制。



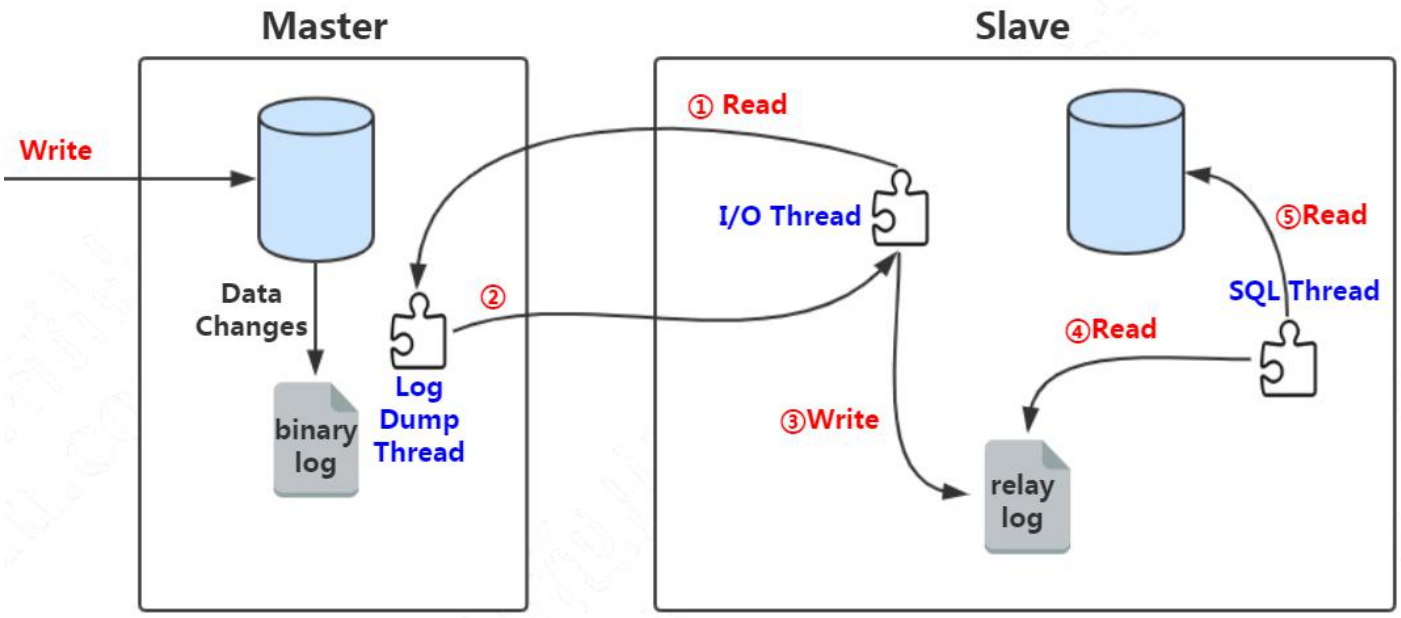
主从复制是怎么实现的呢？更新语句会记录 binlog，它是一种逻辑日志。

有了这个 binlog，从服务器会获取主服务器的 binlog 文件，然后解析里面的 SQL 语句，在从服务器上面执行一遍，保持主从的数据一致。

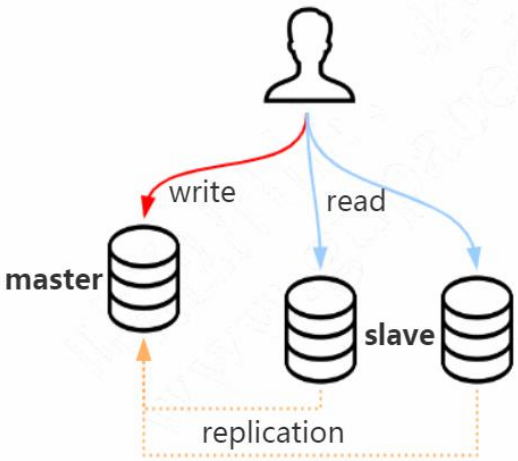
这里面涉及到三个线程，连接到 master 获取 binlog，并且解析 binlog 写入中继日志，这个线程叫做 I/O 线程。

Master 节点上有一个 log dump 线程，是用来发送 binlog 给 slave 的。

从库的 SQL 线程，是用来读取 relay log，把数据写入到数据库的。



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。如果我们在 master 写入了，马上到 slave 查询，而这个时候 slave 的数据还没有同步过来，怎么办？

所以，基于主从复制的原理，我们需要弄明白，主从复制到底慢在哪里？

3.2.1 单线程

在早期的 MySQL 中，slave 的 SQL 线程是单线程。master 可以支持 SQL 语句的并

行执行，配置了多少的最大连接数就是最多同时多少个 SQL 并行执行。

而 slave 的 SQL 却只能单线程排队执行，在主库并发量很大的情况下，同步数据肯定会出现延迟。

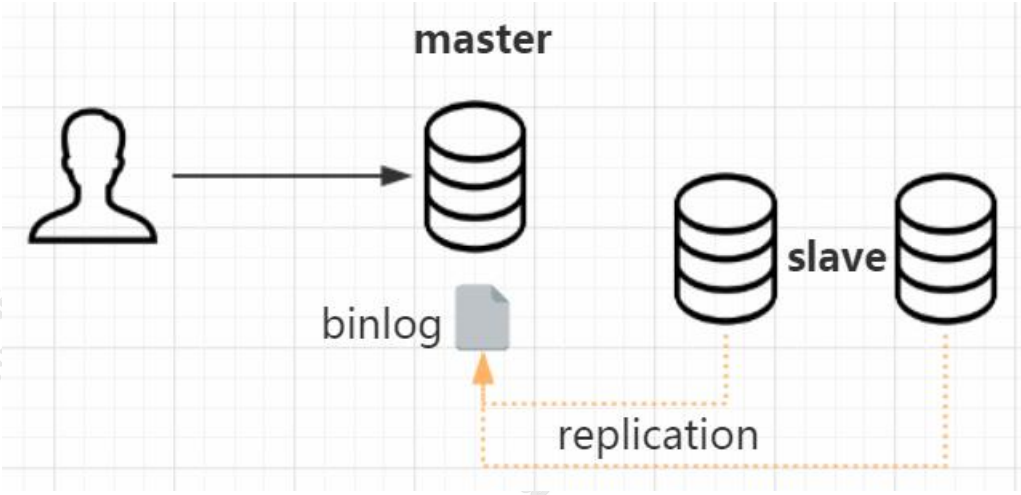
为什么从库上的 SQL Thread 不能并行执行呢？举个例子，主库执行了多条 SQL 语句，首先用户发表了一条评论，然后修改了内容，最后把这条评论删除了。这三条语句在从库上的执行顺序肯定是不能颠倒的。

```
insert into user_comments (10000009,'nice');
update user_comments set content ='very good' where id =10000009;
delete from user_comments where id =10000009;
```

怎么解决这个问题呢？怎么减少主从复制的延迟？

3.2.2 异步与全同步

首先我们需要知道，在主从复制的过程中，MySQL 默认是异步复制的。也就是说，对于主节点来说，写入 binlog，事务结束，就返回给客户端了。对于 slave 来说，接收到 binlog，就完事儿了，master 不关心 slave 的数据有没有写入成功。



如果要减少延迟，是不是可以等待全部从库的事务执行完毕，才返回给客户端呢？这样的方式叫做全同步复制。从库写完数据，主库才返会给客户端。

这种方式虽然可以保证在读之前，数据已经同步成功了，但是带来的副作用大家应该能想到，事务执行的时间会变长，它会导致 master 节点性能下降。

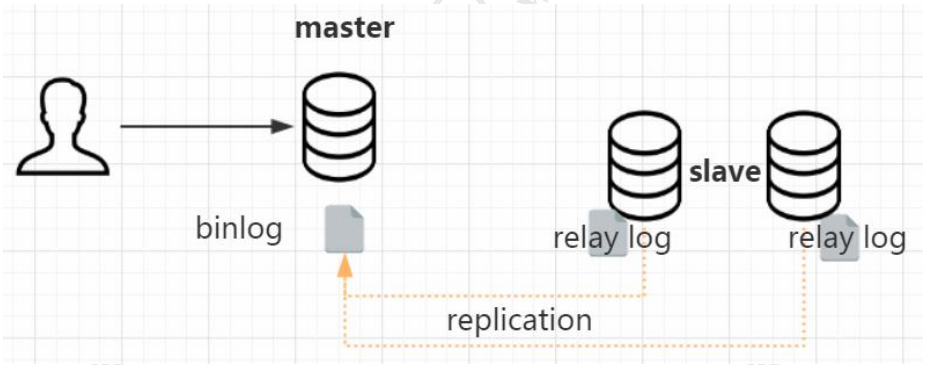
有没有更好的办法呢？既减少 slave 写入的延迟，又不会明显增加 master 返回给客户端的时间？

3.2.3 半同步复制

介于异步复制和全同步复制之间，还有一种半同步复制的方式。

半同步复制是什么样的呢？

主库在执行完客户端提交的事务后不是立刻返回给客户端，而是等待至少一个从库接收到 binlog 并写到 relay log 中才返回给客户端。master 不会等待很长的时间，但是返回给客户端的时候，数据就即将写入成功了，因为它只剩最后一步了：就是读取 relay log，写入从库。



如果我们要在数据库里面用半同步复制，必须安装一个插件，这个是谷歌的一位工程师贡献的。这个插件在 mysql 的插件目录下已经有提供：

```
cd /usr/lib64/mysql/plugin/
```

主库和从库是不同的插件，安装之后需要启用：

```
-- 主库执行
INSTALL PLUGIN rpl_semi_sync_master SONAME 'semisync_master.so';
```

```
set global rpl_semi_sync_master_enabled=1;
show variables like '%semi_sync%';

-- 从库执行
INSTALL PLUGIN rpl_semi_sync_slave SONAME 'semisync_slave.so';
set global rpl_semi_sync_slave_enabled=1;
show global variables like '%semi%';
```

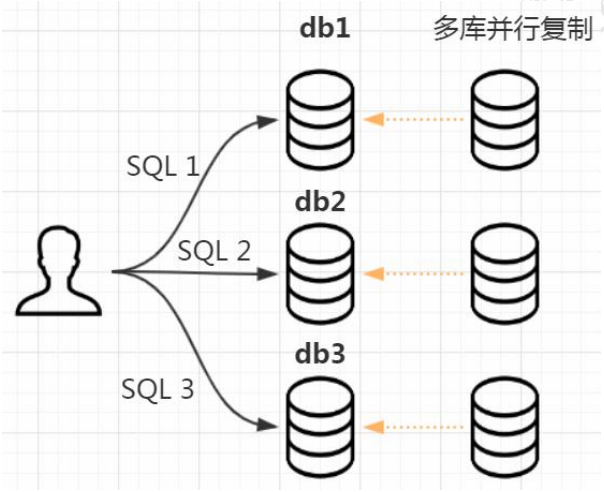
相对于异步复制，半同步复制提高了数据的安全性，同时它也造成了一定程度的延迟，它需要等待一个 slave 写入中继日志，这里多了一个网络交互的过程，所以，半同步复制最好在低延时的网络中使用。

这个是从主库和从库连接的角度，来保证 slave 数据的写入。

另一个思路，如果要减少主从同步的延迟，减少 SQL 执行造成的等待的时间，那有没有办法在从库上，让多个 SQL 语句可以并行执行，而不是排队执行呢？

3.2.4 多库并行复制

怎么实现并行复制呢？设想一下，如果 3 条语句是在三个数据库执行，操作各自的数据库，是不是肯定不会产生并发的问题呢？执行的顺序也没有要求。当然是，所以如果是操作三个数据库，这三个数据库的从库的 SQL 线程可以并发执行。这是 MySQL 5.6 版本里面支持的多库并行复制。



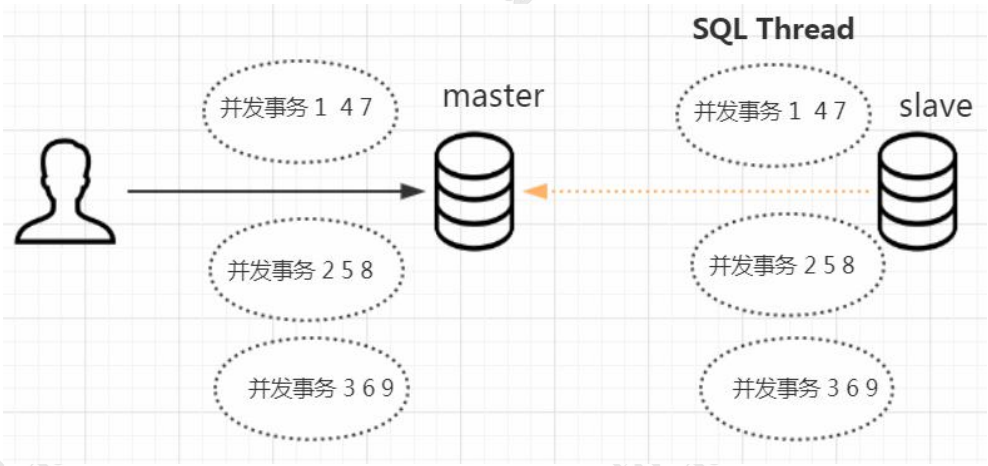
但是在大部分的情况下，我们都是单库多表的情况，在一个数据库里面怎么实现并行复制呢？或者说，我们知道，数据库本身就是支持多个事务同时操作的；为什么这些事务在主库上面可以并行执行，却不会出现问题呢？

因为他们本身就是互相不干扰的，比如这些事务是操作不同的表，或者操作不同的行，不存在资源的竞争和数据的干扰。那在主库上并行执行的事务，在从库上肯定也是可以并行执行，是不是？比如在 master 上有三个事务同时分别操作三张表，这三个事务是不是在 slave 上面也可以并行执行呢？

3.2.5 异步复制之 GTID 复制

https://dev.mysql.com/doc/refman/5.7/en/replication-gtids.html

所以，我们可以把那些在主库上并行执行的事务，分为一个组，并且给他们编号，这一个组的事务在从库上面也可以并行执行。这个编号，我们把它叫做 GTID（Global Transaction Identifiers），这种主从复制的方式，我们把它叫做基于 GTID 的复制。



如果我们要使用 GTID 复制，我们可以通过修改配置参数打开它，默认是关闭的：

```
show global variables like 'gtid_mode';
```

无论是优化 master 和 slave 的连接方式，还是让从库可以并行执行 SQL，都是从数据库的层面去解决主从复制延迟的问题。

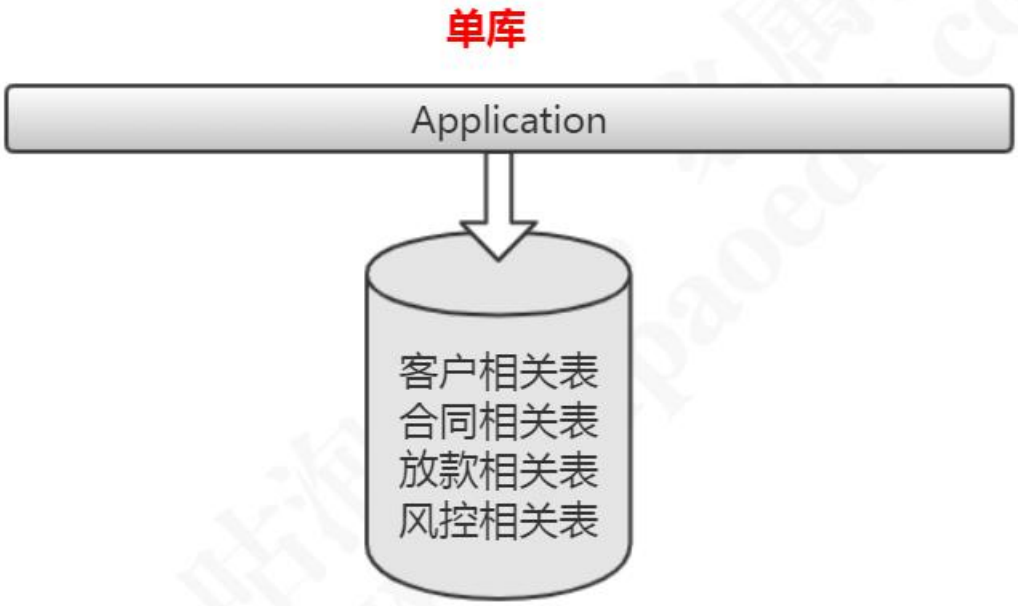
除了数据库本身的层面之外，在应用层面，我们也有一些减少主从同步延迟的方法。

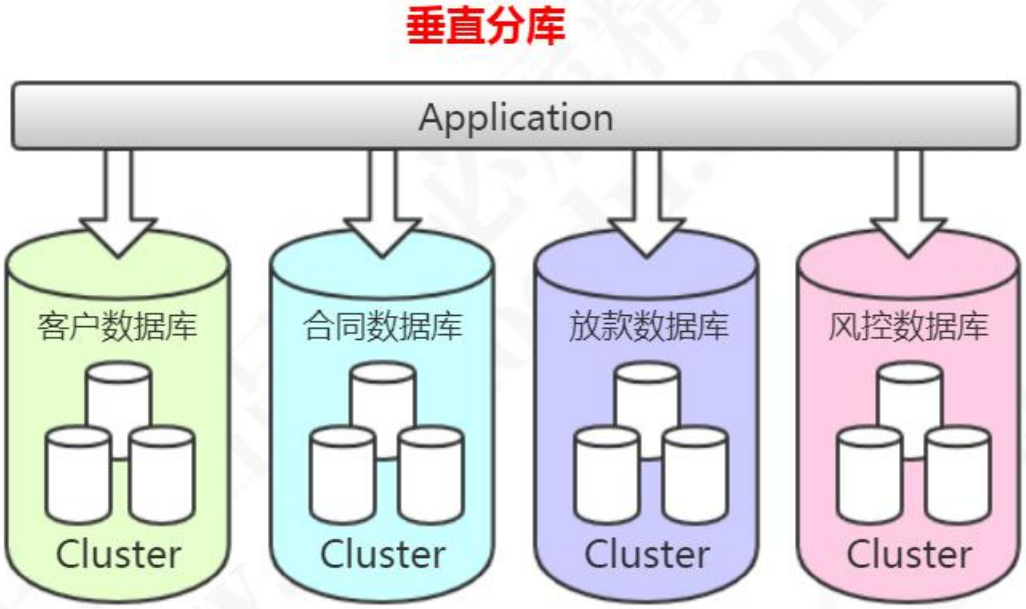
我们在做了主从复制之后，如果单个 master 节点或者单张表存储的数据过大的时候，比如一张表有上亿的数据，单表的查询性能还是会下降，我们要进一步对单台数据库节点的数据分型拆分，这个就是分库分表。

3.3 分库分表

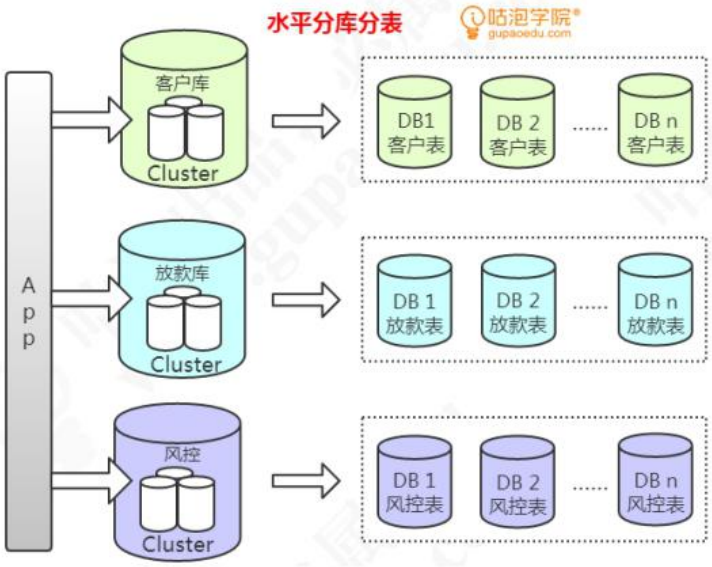
垂直分库，减少并发压力。水平分表，解决存储瓶颈。

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。



通过主从或者分库分表可以减少单个数据库节点的访问压力和存储压力，达到提升数据库性能的目的，但是如果 master 节点挂了，怎么办？

所以，高可用（High Available）也是高性能的基础。

3.4 高可用方案

https://dev.mysql.com/doc/mysql-ha-scalability/en/ha-overview.html

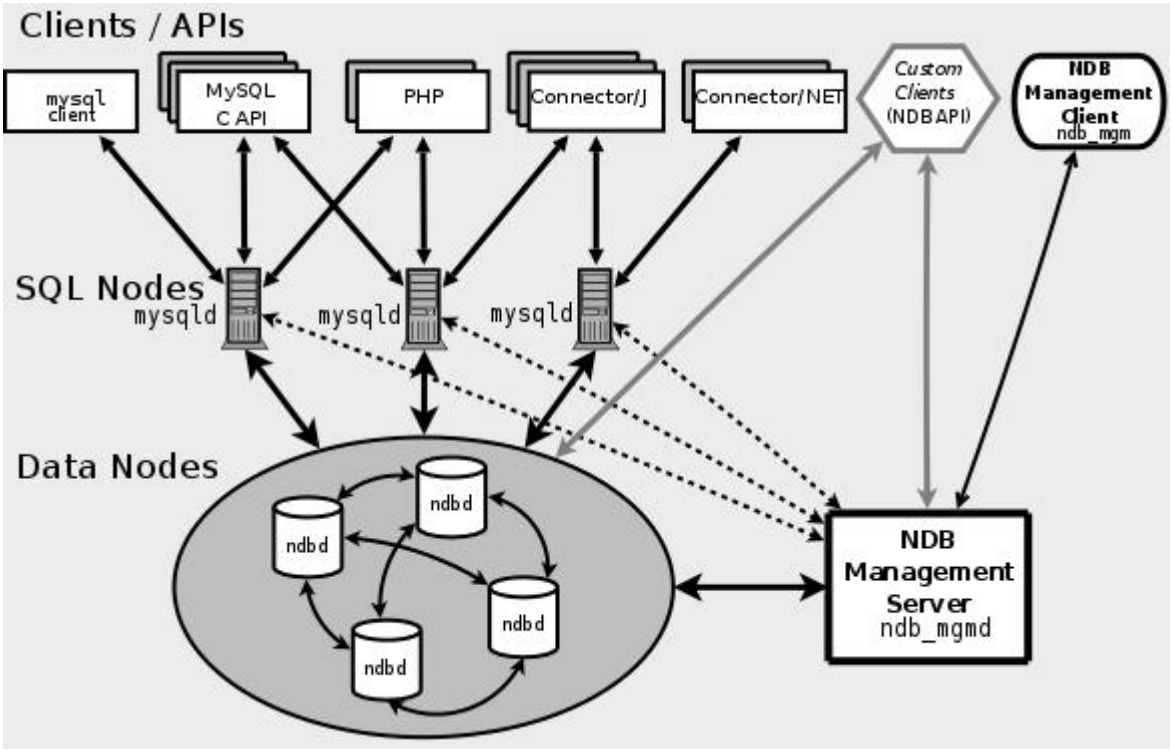
3.4.1 主从复制

传统的 HAProxy + keepalived 的方案，基于主从复制。

3.4.2 **NDB Cluster**

https://dev.mysql.com/doc/mysql-cluster-excerpt/5.7/en/mysql-cluster-overview.html

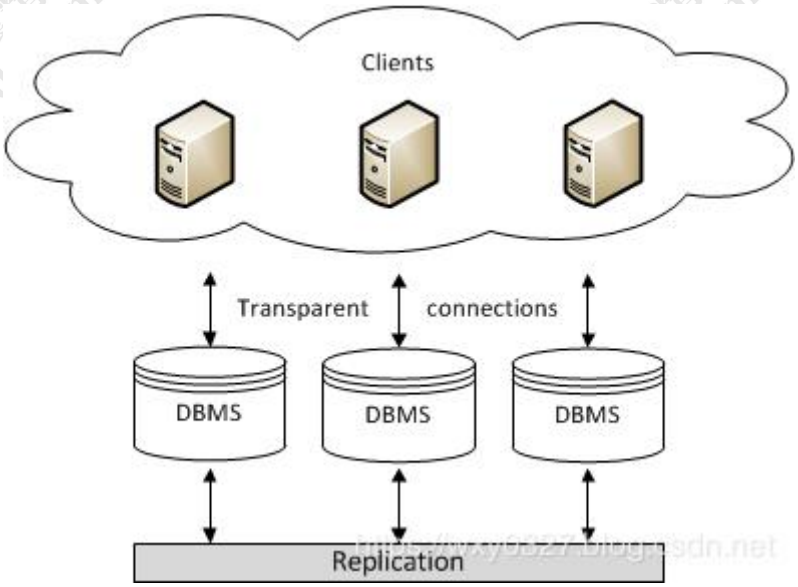
基于 NDB 集群存储引擎的 MySQL Cluster。



3.4.3 **Galera**

https://galeracluster.com/

一种多主同步复制的集群方案。



3.4.4 MHA/MMM

https://tech.meituan.com/2017/06/29/database-availability-architecture.html

MMM（Master-Master replication manager for MySQL），一种多主的高可用架构，是一个日本人开发的，像美团这样的公司早期也有大量使用 MMM。

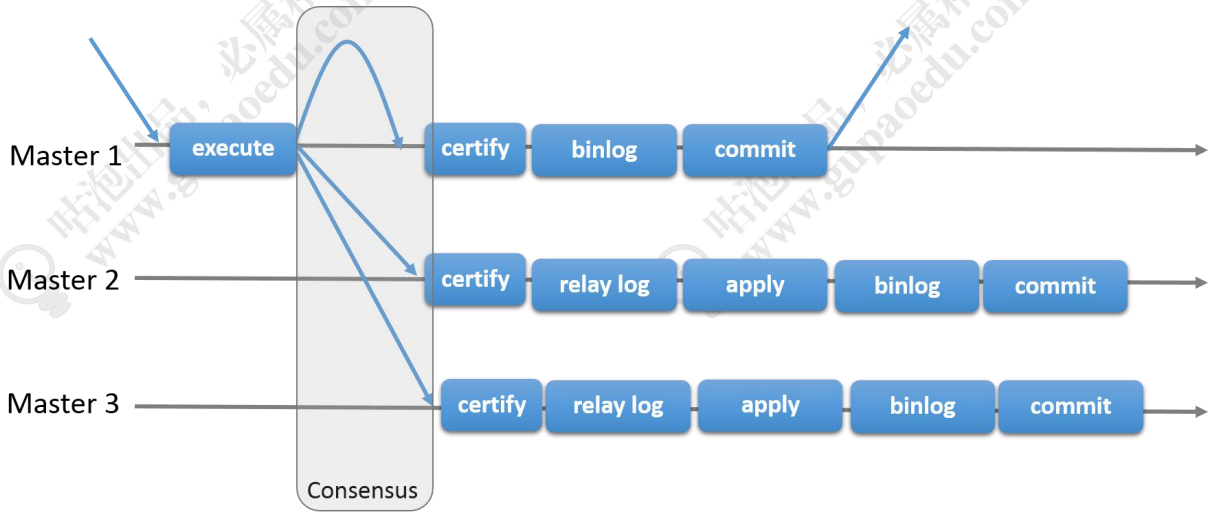
MHA（MySQL Master High Available）。

MMM 和 MHA 都是对外提供一个虚拟 IP，并且监控主节点和从节点，当主节点发生故障的时候，需要把一个从节点提升为主节点，并且把从节点里面比主节点缺少的数据补上，把 VIP 指向新的主节点。

3.4.5 MGR

https://dev.mysql.com/doc/refman/5.7/en/group-replication.html
https://dev.mysql.com/doc/refman/5.7/en/mysql-cluster.html

MySQL 5.7.17 版本推出的 InnoDB Cluster，也叫 MySQL Group Replicatioin（MGR），这个套件里面包括了 mysql shell 和 mysql-route。



总结一下：

高可用 HA 方案需要解决的问题都是当一个 master 节点宕机的时候，如何提升一个数据最新的 slave 成为 master。如果同时运行多个 master，又必须要解决 master 之间数据复制，以及对于客户端来说连接路由的问题。

不同的方案，实施难度不一样，运维管理的成本也不一样。

以上是架构层面的优化，可以用缓存，主从，分库分表。

第三个环节：

解析器，词法和语法分析，主要保证语句的正确性，语句不出错就没问题。由 Sever 自己处理，跳过。

第四步：优化器

4 优化器——SQL 语句分析与优化

优化器就是对我们的 SQL 语句进行分析，生成执行计划。

问题：在我们做项目的时候，有时会收到 DBA 的邮件，里面列出了我们项目上几个耗时比较长的查询语句，让我们去优化，这些语句是从哪里来的呢？

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_fil	/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。

```
show variables like '%slow_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（检查是不是没有索引）。

```
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';
```

有了慢查询日志，怎么去分析统计呢？比如 SQL 语句的出现的慢查询次数最多，平均每次执行了多久？

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 20 条慢 SQL：

```
mysqldumpslow -s t -t 20 -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。

除了慢查询日志之外，还有一个 SHOW PROFILE 工具可以使用。

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(D
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;
```

除了慢日志和 show profile，如果要分析出当前数据库中执行的慢的 SQL，还可以通过查看运行线程状态和服务器运行信息、存储引擎信息来分析。

4.2.3 其他系统命令

show processlist 运行线程

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。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 https://dev.mysql.com/doc/refman/5.7/en/thread-commands.html
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 https://dev.mysql.com/doc/refman/5.7/en/general-thread-states.html
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 status 服务器运行状态

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

SHOW STATUS 用于查看 MySQL 服务器运行状态（重启后会清空），有 session 和 global 两种作用域，格式：参数-值。

可以用 like 带通配符过滤。

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

show engine 存储引擎运行信息

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

```
show engine innodb status;
```

如果需要将监控信息输出到错误信息 error log 中（15 秒钟一次），可以开启输出。

```
show variables like 'innodb_status_output%';
-- 开启输出：
SET GLOBAL innodb_status_output=ON;
SET GLOBAL innodb_status_output_locks=ON;
```

我们现在已经知道了这么多分析服务器状态、存储引擎状态、线程运行信息的命令，如果让你去写一个数据库监控系统，你会怎么做？

其实很多开源的慢查询日志监控工具，他们的原理其实也都是读取的系统的变量和状态。

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。

explain 可以分析 update、delete、insert 么？

MySQL 5.6.3 以前只能分析 SELECT; MySQL5.6.3 以后就可以分析 update、delete、insert 了。

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');
```

explain 的结果有很多的字段，我们详细地分析一下。

先确认一下环境：

```
select version();
show variables like '%engine%';
```

4.3.1 id

id 是查询序列编号。

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

```
-- 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.tcid = tc.tcid
AND (c.cid = 2
OR tc.tcid = 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，查询的顺序是 teacher t（3 条）——course c（4 条）——teacher_contact tc（3 条）。

teacher 表插入 3 条数据后：

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

-- （备份）恢复语句
DELETE FROM teacher where tid in (4,5,6);
COMMIT;
```

id 也都是 1，但是从上往下查询顺序变成了 :teacher_contact tc(3 条)——teacher t（6 条）——course c（4 条）。

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tc	(Null)	ALL	(Null)	(Null	(Null)	(Nu	3	100	(Null)
1	SIMPLE	t	(Null)	ALL	(Null)	(Null	(Null)	(Nu	6	16.67	Using where; Using join buffer
1	SIMPLE	c	(Null)	ALL	(Null)	(Null	(Null)	(Nu	4	25	Using where; Using join buffer

为什么数据量不同的时候顺序会发生变化呢？这个是由笛卡尔积决定的。

举例：假如有 a、b、c 三张表，分别有 2、3、4 条数据，如果做三张表的联合查询，当查询顺序是 a→b→c 的时候，它的笛卡尔积是：2*3*4=6*4=24。如果查询顺序是 c→b→a，它的笛卡尔积是 4*3*2=12*2=24。

因为 MySQL 要把查询的结果，包括中间结果和最终结果都保存到内存，所以 MySQL 会优先选择中间结果数据量比较小的顺序进行查询。所以最终联表查询的顺序是 a→b→c。这个就是为什么 teacher 表插入数据以后查询顺序会发生变化。

（小标驱动大表的思想）

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，不包含关联查询 union。

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

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	teacher	(Null)	ALL	(Null)	(Null	(Null)	(Nu	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)	(Nu	3	33.33	Using where
2	SUBQUERY	t	(Null)	ALL	(Null)	(Null	(Null)	(Nu	3	33.33	Using where
3	SUBQUERY	c	(Null)	ALL	(Null)	(Null	(Null)	(Nu	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

衍生查询，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)	(Nu	4	100	(Null)
2	DERIVED	course	(Null)	ALL	(Null)	(Null	(Null)	(Nu	4	25	Using where
3	UNION	course	(Null)	ALL	(Null)	(Null	(Null)	(Nu	4	25	Using where
(Nu	UNION RESULT	<union2,3>	(Null)	ALL	(Null)	(Null	(Null)	(Nu	(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。同上例。

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。

4.3.3 type 连接类型

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，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。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)	(Nul	1	100

eq_ref

通常出现在多表的 join 查询，表示对于前表的每一个结果，都只能匹配到后表的一行结果。一般是唯一性索引的查询（UNIQUE 或 PRIMARY KEY）。

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_tcic	5	(Null)	3
SIMPLE	tc	(Null)	eq_ref	PRIMARY	PRIMAI	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，或者关联操作只使用了索引的最左前缀。

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

索引范围扫描。

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。

NULL

不用访问表或者索引就能得到结果，例如：

```
EXPLAIN select 1 from dual where 1=1;
```

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，可能用到索引不代表一定用到索引。

反过来，possible_key 为空，key 可能有值吗？

表上创建联合索引：

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

执行计划（改成 select name 也能用到索引）：

```
explain select phone from user_innodb where phone='126';
```

id	select_type	table	type	possible_keys	key
1	SIMPLE	user_innodb	index	(Null)	comidx_name_phone

结论：是有可能的（这里是覆盖索引的情况）。

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。

4.3.6 rows

MySQL 认为扫描多少行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。

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。

4.3.9 Extra

执行计划给出的额外的信息说明。

using index

用到了覆盖索引，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)	(Nul	4646619	10	Using where

Using index condition（索引条件下推）

索引下推，在第二节课中已经讲解过了。

https://dev.mysql.com/doc/refman/5.7/en/index-condition-pushdown-optimization.html

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);
```

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

（order by id 引起）

select_type	table	p	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

用到了临时表。例如（以下不是全部的情况）：

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;
```

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。

分析出问题之后，就是对 SQL 语句的具体优化。

比如怎么用到索引，怎么减少锁的阻塞等待，在前面两次课已经讲过。

4.4 SQL 与索引优化

当我们的 SQL 语句比较复杂，有多个关联和子查询的时候，就要分析 SQL 语句有没有改写的方法。

举个简单的例子，一模一样的数据：

```
-- 大偏移量的 limit
select * from user_innodb limit 900000,10;

-- 改成先过滤 ID，再 limit
SELECT * FROM user_innodb WHERE id >= 900000 LIMIT 10;
```

对于具体的 SQL 语句的优化，MySQL 官网也提供了很多建议，这个是我们在分析具体的 SQL 语句的时候需要注意的，也是大家在以后的工作里面要去慢慢地积累的（这里我们就不一一地分析了）。

https://dev.mysql.com/doc/refman/5.7/en/optimization.html

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 分区或者分表

分区不推荐。

交易历史表：在年底为下一年度建立 12 个分区，每个月一个分区。

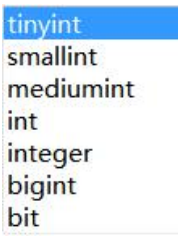
渠道交易表：分成当日表；当月表；历史表，历史表再做分区。

5.3 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：

5.3.1 整数类型



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 也是整型存储。

5.3.2 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。

5.3.3 非空

非空字段尽量定义成 NOT NULL，提供默认值，或者使用特殊值、空串代替 null。

NULL 类型的存储、优化、使用都会存在问题。

5.3.4 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。

5.3.5 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。

5.3.6 表拆分

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

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。举几个例子：

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（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，为什么提前一个多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，有的公司可以扛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。

集中答疑链接：https://gper.club/articles/7e7e7f7ff7g55gc9g6b

作者：咕泡学院-青山老师

最后更新时间：2020 年 1 月 5 日 22:54:57