MySQL 优化原理 (2)

如果有同学看完上一篇关于 MySQL 文章，文末留有两个很开放的问题，如有兴趣可以在脑袋里想想。本文也会试着回答这两个问题，希望能给你一些参考。现在可以思考一个问题，如果数据量非常大的情况下，您根据业务选择了合适的字段，精心设计了表和索引，还仔细的检查了所有的 SQL，并确认已经没什么问题，但性能仍然不能满足您的要求，该怎么办呢？还有其他优化策略吗？答案是肯定的。接下来继续和您讨论一些常用的 MySQL 高级特性以及其背后的工作原理。

分区表

合理的使用索引可以极大提升 MySQL 的查询性能，但如果单表数据量达到一定的程度，索引就无法起作用，因为在数据量超大的情况下，除非覆盖索引，因回表查询会产生大量的随机 I/O，数据库的响应时间可能会达到不可接受的程度。而且索引维护（磁盘空间、I/O操作）的代价也会非常大。

因此，当单表数据量达到一定程度时（在 MySQL4.x 时代，MyISAM 存储引擎业内公认的性能拐点是 500W 行，MySQL5.x 时代的性能拐点则为 1KW ~ 2KW 行级别，具体需根据实际情况测试），为了提升性能，最为常用的方法就是分表。分表的策略可以是垂直拆分（比如：不同订单状态的订单拆分到不同的表），也可以是水平拆分（比如：按月将订单拆分到不同表）。但总的来说，分表可以看作是从业务角度来解决大数据量问题，它在一定程度上可以提升性能，但也大大提升了编码的复杂度，有过这种经历的同学可能深有体会。

在业务层分表大大增加了编码的复杂程度，而且处理数据库的相关代码会大量散落在应用各处，维护困难。那是否可以将分表的逻辑抽象出来，统一处理，这样业务层就不用关心底层是否分表，只需要专注在业务即可。答案当然是肯定的，目前有非常多的数据库中间件都可以屏蔽分表后的细节，让业务层像查询单表一样查询分表后的数据。如果再将抽象的逻辑下移到数据库的服务层，就是我们今天要讲的分区表。

分区可以看作是从技术层面解决大数据问题的有效方法，简单的理解，可以认为是 MySQL 底层帮我们实现分表，分区表是一个独立的逻辑表，底层由多个物理子表组成。存储引擎管理分区的各个底层表和管理普通表一样（所有底层表必须使用相同的存储引擎），分区表的索引也是在各个底层表上各自加上一个完全相同的索引。从存储引擎的角度来看，底层表和普通表没有任何不同，存储引擎也无须知道。在执行查询时，优化器会根据分区的定义过滤那些没有我们需要数据的分区，这样查询就无需扫描所有分区，只需要查找包含需要数据的分区就可以了。

更好的理解分区表，我们从一个示例入手：一张订单表，数据量大概有 10TB，如何设计才能使性能达到最优？

首先可以肯定的是，因为数据量巨大，肯定不能走全表扫描。使用索引的话，你会发现数据并不是按照想要的方式聚集，而且会产生大量的碎片，最终会导致一个查询产生成千上万的随机 I/O，应用随之僵死。所以需要选择一些更粗粒度并且消耗更少的方式来检索数据。比如先根据索引找到一大块数据，然后再在这块数据上顺序扫描。

这正是分区要做的事情，理解分区时还可以将其当作索引的最初形态，以代价非常小的方式定位到需要的数据在哪一片“区域”，在这片“区域”中，你可以顺序扫描，可以建索引，还可以将数据都缓存在内存中。因为分区无须额外的数据结构记录每个分区有哪些数据，所以其代价非常低。只需要一个简单的表达式就可以表达每个分区存放的是什么数据。

对表分区，可以在创建表时，使用如下语句：

SQL | 复制代码

```sql
CREATE TABLE sales {
    order_date DATETIME NOT NULL
    -- other columns
} ENGINE=InnoDB PARTITION BY RANGE(YEAR(order_date)) (
    PARTITION p_2014 VALUES LESS THAN (2014),
    PARTITION p_2015 VALUES LESS THAN (2015)
    PARTITION p_2016 VALUES LESS THAN (2016)
    PARTITION p_2017 VALUES LESS THAN (2017)
    PARTITION p_catchall VALUES LESS THAN MAXVALUE
)
```

分区子句中可以使用各种函数，但表达式的返回值必须是一个确定的整数，且不能是一个常数。MySQL还支持一些其他分区，比如键值、哈希、列表分区，但在生产环境中很少见到。在MySQL5.5以后可以使用

RANGE COLUMNS类型分区，这样即使是基于时间分区，也无需再将其转化成一个整数。

接下来简单看下分区表上的各种操作逻辑：

- `SELECT`：当查询一个分区表时，分区层先**打开并锁住所有的底层表**，优化器先判断是否可以过滤部分分区，然后在调用对应的存储引擎接口访问各个分区的数据
- `INSERT`：当插入一条记录时，分区层先**打开并锁住所有的底层表**，然后确定哪个分区接收这条记录，再将记录写入对应的底层表，`DELETE` 操作与其类似
- `UPDATE`：当更新一条数据时，分区层先**打开并锁住所有的底层表**，然后确定数据对应的分区，然后取出数据并更新，再判断更新后的数据应该存放到哪个分区，最后对底层表进行写入操作，并对原数据所在的底层表进行删除操作

有些操作是支持条件过滤的。例如，当删除一条记录时，MySQL 需要先找到这条记录，如果 `WHERE` 条件恰好和分区表达式匹配，就可以将所有不包含这条记录的分区都过滤掉，这对 `UPDATE` 语句同样有效。如果是 `INSERT` 操作，本身就只命中一个分区，其他分区都会被过滤。

虽然每个操作都会“先打开并锁住所有的底层表”，但这并不是说分区表在处理过程中是锁住全表的。如果存储引擎能够自己实现行级锁，例如 InnoDB，则会在分区层释放对应表锁。这个加锁和解锁的操作过程与普通 InnoDB 上的查询类似。

在使用分区表时，为了保证大数据量的可扩展性，一般有两个策略：

- 全量扫描数据，不用索引。即只要能够根据 WHERE 条件将需要查询的数据限制在少数分区中，效率是不错的
- 索引数据，分离热点。如果数据有明显的“热点”，而且除了这部分数据，其他数据很少被访问到，那么可以将这部分热点数据单独存放在一个分区中，让这个分区的数据能够有机会都缓存在内存中。这样查询就可以值访问一个很小的分区表，能够使用索引，也能够有效的利用缓存。

分区表的优点是优化器可以根据分区函数来过滤一些分区，但很重要的一点是要在 `WHERE` 条件中带入分区列，有时候即使看似多余的也要带上，这样就可以让优化器能够过滤掉无须访问的分区，如果没有这些条件，MySQL 就需要让对应的存储引擎访问这个表的所有分区，如果表非常大的话，就可能会非常慢。

上面两个分区策略基于两个非常重要的前提：查询都能够过滤掉很多额外的分区、分区本身并不会带来很多额外的代价。而这两个前提在某些场景下是有问题的，比如：

1、**NULL值会使分区过滤无效**

假设按照 `PARTITION BY RANGE YEAR(order_date)` 分区，那么所有 `order_date` 为NULL或者非法值时，记录都会被存放到第一个分区。所以 `WHERE order_date BETWEEN '2017-05-01' AND ‘2017-05-31’`，这个查询会检查两个分区，而不是我们认为的 2017 年这个分区（会额外的

检查第一个分区），是因为 `YEAR()` 在接收非法值时会返回 NULL。如果第一个分区的数据量非常大，而且使用全表扫描的策略时，代价会非常大。为了解决这个问题，我们可以创建一个无用的分区，比如：`PARTITION p_null values less than (0)`。如果插入的数据都是有效的话，第一个分区就是空的。

> 在MySQL5.5以后就不需要这个技巧了，因为可以直接使用列本身而不是基于列的函数进行分区：`PARTITION BY RANGE COLUMNS(order_date)`。直接使用这个语法可避免这个问题。

2、**分区列和索引列不匹配**

当分区列和索引列不匹配时，可能会导致查询无法进行分区过滤，除非每个查询条件中都包含分区列。假设在列 a 上定义了索引，而在列 b 上进行分区。因为每个分区都有其独立的索引，所以在扫描列 b 上的索引就需要扫描每一个分区内对应的索引，当然这种速度不会太慢，但是能够跳过不匹配的分区肯定会更好。这个问题看起来很容易避免，但需要注意一种情况就是，关联查询。如果分区表是关联顺序的第2张表，并且关联使用的索引与分区条件并不匹配，那么关联时对第一张表中符合条件的每一行都需要访问并搜索第二张表的所有分区（关联查询原理，请参考前一篇文章）

3、**选择分区的成本可能很高**

分区有很多种类型，不同类型的分区实现方式也不同，所以它们的性能也不尽相同，尤其是范围分区，在确认这一行属于哪个分区时会扫描所有的分区定义，这样的线性扫描效率并不高，所以随着分区数的增长，成本会越来越高。特别是在批量插入数据时，由于每条记录在插入前，都需要确认其属于哪一个分区，如果分区数太大，会造成插入性能的急剧下降。因此有必要限制分区数量，但也不用太过担心，对于大多数系统，100 个左右的分区是没有问题的。

4、**打开并锁住所有底层表的成本在某些时候会很高**

前面说过，打开并锁住所有底层表并不会对性能有太大的影响，但在某些情况下，比如只需要查询主键，那么锁住的成本相对于主键的查询来说，成本就略高。

5、**维护分区的成本可能会很高**

新增和删除分区的速度都很快，但是修改分区会造成数据的复制，这与 `ALTER TABLE` 的原理类似，需要先创建一个历史分区，然后将数据复制到其中，最后删除原分区。因此，设计数据库时，考虑业务的增长需要，合理的创建分区表是一个非常好的习惯。在 MySQL5.6 以后的版本可以使用 `ALTER TABLE EXCHAGE PARTITION` 语句来修改分区，其性能会有很大提升。

分区表还有一些其他限制，比如所有的底层表必须使用相同的存储引擎，某些存储引擎也不支持分区。分区一般应用于一台服务器上，但一台服务器的物理资源总是有限的，当数据达到这个极限时，即使分区，性能也可能会很低，所以这个时候分库是必须的。但不管是分区、分库还是分表，它们的思想都是一样的，大家可以好好体会下。

视图

对于一些关联表的复杂查询，使用视图有时候会大大简化问题，因此在许多场合下都可以看到视图的身影，但视图真如我们所想那样简单吗？它和直接使用 `JOIN` 的 SQL 语句有何区别？视图背后的原理又了解多少？

视图本身是一个虚拟表，不存放任何数据，查询视图的数据集由其他表生成。MySQL 底层通过两种算法来实现视图：临时表算法（TEMPTABLE）和合并算法（MERGE）。所谓临时表算法就是将 SELECT 语句的结果存放到临时表中，当需要访问视图的时候，直接访问这个临时表即可。而合并算法则是重写包含视图的查询，将视图定义的 SQL 直接包含进查询 SQL 中。通过两个简单的示例来体会两个算法的差异，创建如下视图：

SQL | 复制代码

```sql
-- 视图的作用是查询未支付订单
CREATE VIEW unpay_order AS
SELECT * FROM sales WHERE status = 'new'
WITH CHECK OPTION;   // 其作用下文会讲
```

现要从未支付订单中查询购买者为 `csc` 的订单，可以使用如下查询：

SQL | 复制代码

```sql
-- 查询购买者为csc且未支付的订单
SELECT order_id,order_amount,buyer FROM unpay_order WHERE buyer = 'csc';
```

使用临时表来模拟视图：

SQL | 复制代码

```sql
CREATE TEMPORARY TABLE tmp_order_unpay AS SELECT * FROM sales WHERE status = 'new';
SELECT order_id,order_amount,buyer FROM tmp_order_unpay WHERE buyer = 'csc';
```

使用合并算法将视图定义的 SQL 合并进查询 SQL 后的样子：

SQL | 复制代码

```sql
SELECT order_id,order_amount,buyer FROM sales WHERE status = 'new' AND buyer = 'csc';
```

MySQL 可以嵌套定义视图，即在一个视图上在定义另一个视图，可以在 `EXPLAN EXTENDED` 之后使用 `SHOW WARNINGS` 来查看使用视图的查询重写后的结果。如果采用临时表算法实现的视图，`EXPLA`

`IN` 中会显示为派生表（`DERIVED`），注意 `EXPLAIN` 时需要实际执行并产生临时表，所以有可能会很慢。

明显地，临时表上没有任何索引，而且优化器也很难优化临时表上的查询，因此，如有可能，尽量使用合并算法会有更好的性能。那么问题来了：合并算法（类似于直接查询）有更好的性能，为什么还要使用视图?

首先视图可以简化应用上层的操作，让应用更专注于其所关心的数据。其次，视图能够对敏感数据提供安全保护，比如：对不同的用户定义不同的视图，可以使敏感数据不出现在不应该看到这些数据的用户视图上；也可以使用视图实现基于列的权限控制，而不需要真正的在数据库中创建列权限。再者，视图可以方便系统运维，比如：在重构 schema 的时候使用视图，使得在修改视图底层表结构的时候，应用代码还可以继续运行不报错。

基于此，使用视图其实更多的是基于业务或者维护成本上的考虑，其本身并不会对性能提升有多大作用（注意：此处只是基于MySQL考虑，其他关系性数据库中视图可能会有更好的性能，比如 `ORACLE` 和 `MS SQL SERVER` 都支持物化视图，它们都比 MySQL 视图有更好的性能）。而且使用临时表算法实现的视图，在某些时候性能可能会非常糟糕，比如：

SQL | 复制代码

```sql
-- 视图的作用是统计每日支出金额，DATE('2017-06-15 12:00:23') = 2017-06-15
CREATE VIEW cost_per_day AS
SELECT DATE(create_time) AS date,SUM(cost) AS cost FROM costs GROUP BY date
;
```

现要统计每日的收入与支出，有类似于上面的收入表，可以使用如下 SQL：

SQL | 复制代码

```sql
SELECT c.date,c.cost,s.amount
FROM cost_per_day AS c
JOIN sale_per_day AS s USING(date)
WHERE date BETWEEN '2017-06-01' AND '2017-06-30'
```

这个查询中，MySQL 先执行视图的 SQL，生成临时表，然后再将 `sale_per_day` 表和临时表进行关联。这里 `WHERE` 字句中的 `BETWEEN` 条件并不能下推到视图中，因而视图在创建时，会将所有的数据放到临时表中，而不是一个月数据，并且这个临时表也不会有索引。

当然这个示例中的临时表数据不会太大，毕竟日期的数量不会太多，但仍然要考虑生成临时表的性能（如果 costs 表数据过大，`GROUP BY` 有可能会比较慢）。而且本示例中索引也不是问题，通过上一篇我们知道，如果 MySQL 将临时表作为关联顺序中的第一张表，仍然可以使用 `sale_per_day` 中的索引。

但如果是对两个视图做关联的话，优化器就没有任何索引可以使用，这时就需要严格测试应用的性能是否满足需求。

我们很少会在实际业务场景中去更新视图，因此印象中，视图是不能更新的。但实际上，在某些情况下，视图是可以更新的。**可更新视图**是指通过更新这个视图来更新视图涉及的相关表，只要指定了合适的条件，就可以更新、删除甚至是向视图中插入数据。通过上文的了解，不难推断出：更新视图的实质就是更新视图关联的表，将创建视图的 `WHERE` 子句转化为 `UPDATE` 语句的 `WHERE` 子句，只有使用合并算法的视图才能更新，并且更新的列必须来自同一个表中。回顾上文创建视图的SQL语句，其中有一句：`WITH CHECK OPTION`，其作用就是表示通过视图更新的行，都必须符合视图本身的 `WHERE` 条件定义，不能更新视图定义列以外的列，否则就会抛出 `check option failed` 错误。

视图还有一个容易造成误解的地方：“对于一些简单的查询，视图会使用合并算法，而对于一些比较复杂的查询，视图就会使用临时表算法”。但实际上，视图的实现算法是视图本身的属性决定的，跟作用在视图上的 SQL 没有任何关系。那什么时候视图采用临时表算法，什么时候采用合并算法呢？一般来说，只要原表记录和视图中的记录无法建立一一映射的关系时，MySQL 都将使用临时表算法来实现视图。比如创建视图的 SQL 中包含 `GROUP BY`、`DISTINCT`、`UNION`、聚合函数、子查询的时候，视图都将采用临时表算法（这些规则在以后的版本中，可能会发生改变，具体请参考官方手册）。

相比于其它关系型数据库的视图，MySQL 的视图在功能上会弱很多，比如 `ORACLE` 和 `MS SQL SERVER` 都支持物化视图。**物化视图**是指将视图结果数据存放在一个可以查询的表中，并定期从原始表中刷新数据到这张表中，这张表和普通物理表一样，可以创建索引、主键约束等等，性能相比于临时表会有质的提升。但遗憾的是 MySQL 目前并不支持物化视图，当然 MySQL 也不支持在视图中创建索引。

存储过程与触发器

回到第二个问题，有非常多的人在分享时都会抛出这样一个观点：尽可能不要使用存储过程，存储过程非常不容易维护，也会增加使用成本，应该把业务逻辑放到客户端。既然客户端都能干这些事，那为什么还要存储过程？

如果有深入了解过存储过程，就会发现存储过程并没有大家描述的那么不堪。我曾经经历过一些重度使用存储过程的产品，依赖到什么程度呢？就这么说吧，上层的应用基本上只处理交互与动效的逻辑，所有的业务逻辑，甚至是参数的校验均在存储过程中实现。曾经有出现过一个超大的存储过程，其文件大小达到惊人的 80K，可想而知，其业务逻辑有多么复杂。在大多数人眼中，这样的技术架构简直有点不可理喻，但实际上这款产品非常成功。

其成功的原因在一定程度上得益于存储过程的优点，由于业务层代码没有任何侵入业务的代码，在不改变前端展示效果的同时，可以非常快速的修复 BUG、开发新功能。由于这款产品需要部署在客户的私有环境上，快速响应客户的需求就变得尤为重要，正是得益于这种架构，可以在客户出现问题或者提出新需求时，快速响应，极端情况下，我们可以在1小时内修复客户遇到的问题。正是这种快速响应机制，让我们获得大量的客户。

当然存储过程还有其他的优点，比如，可以非常方便的加密存储过程代码，而不用担心应用部署到私有环境造成源代码泄露、可以像调试其他应用程序一样调试存储过程、可以设定存储过程的使用权限来保证数据安全等等。一切都非常美好，但如果我们的产品是基于 `MS SQL SERVER` 实现的，其可以通过 `T-SQL` 非常方便的实现复杂的业务逻辑。你可以把 `T-SQL` 看做是一门编程语言，其包含 `SQL` 的所有功能，还具备流程控制、批处理、定时任务等能力，你甚至可以用其来解析 XML 数据。关于 `T-SQL` 的更多信息可以参考 MSDN，主流的关系型数据库目前只有 `MS SQL SERVER` 支持 `T-SQL`，因此，MySQL 并不具备上文描述的一些能力，比如，MySQL 的存储过程调试非常不方便（当然可以通过付费软件来获得很好的支持）。

除此之外，MySQL 存储过程还有一些其他的限制：

- 优化器无法评估存储过程的执行成本
- 每个连接都有独立的存储过程执行计划缓存，如果有多个连接需要调用同一个存储过程，将会浪费缓存空间来缓存相同的执行计划

因此，在 MySQL 中使用存储过程并不是一个太好策略，特别是在一些大数据、高并发的场景下，将复杂的逻辑交给上层应用实现，可以非常方便的扩展已有资源以便获得更高的计算能力。而且对于熟悉的编程语言，其可读性会比存储过程更好一些，也更加灵活。不过，在某些场景下，如果存储过程比其他实现会快很多，并且是一些较小的操作，可以适当考虑使用存储过程。

和存储过程类似的，还有触发器，触发器可以让你在执行 `INSERT`、`UPDATE` 和 `DELETE` 时，执行一些特定的操作。在 MySQL 中可以选择在 SQL 执行之前触发还是在 SQL 执行后触发。触发器一般用于实现一些强制的限制，这些限制如果在应用程序中实现会让业务代码变得非常复杂，而且它也可以减少客户端与服务器之间的通信。MySQL 触发器的实现非常简单，所以功能非常有限，如果你在其他数据库产品中已经重度依赖触发器，那么在使用 MySQL 触发器时候需要注意，因为 MySQL 触发器的表现和预想的不一致。

首先对一张表的每一个事件，最多只能定义一个触发器，而且它只支持“基于行的触发”，也就是触发器始终是针对一条记录的，而不是针对整个 SQL 语句。如果是批量更新的话，效率可能会很低。其次，触发器可以掩盖服务器本质工作，一个简单的 SQL 语句背后，因为触发器，可能包含了很多看不见的工作。再者，触发器出现问题时很难排查。最后，触发器并不一定能保证原子性，比如 `MyISAM` 引擎下触发器执行失败了，也不能回滚。在 `InnoDB` 表上的触发器是在同一个事务中执行完成的，所以它们的执行是原子的，原操作和触发器操作会同时失败或者成功。

虽然触发器有这么多限制，但它仍有适用的场景，比如，当你需要记录 MySQL 数据的变更日志，这时触发器就非常方便了。

外键约束

目前在大多数互联网项目，特别是在大数据的场景下，已经不建议使用外键了，主要是考虑到外键的使用成本：

- 外键通常要求每次修改数据时都要在另外一张表中执行一次查找操作。在 InnoDB 存储引擎中会强制外键使用索引，但在大数据的情况下，仍然不能忽略外键检查带来的开销，特别是当外键的选择性很低时，会导致一个非常大且选择性低的索引。
- 如果向子表中插入一条记录，外键约束会让 InnoDB 检查对应的父表的记录，也就需要对父表对应记录进行加锁操作，来确保这条记录不会在这个事务完成之时就被删除了。这会导致额外的锁等待，甚至会导致一些死锁。
- 高并发场景下，数据库很容易成为性能瓶颈，自然而然的就希望数据库可以水平扩展，这时就需要把数据的一致性控制放到应用层，也就是让应用服务器可以承担压力，这种情况下，数据库层面就不能使用外键。

因此，当不用过多考虑数据库的性问题时，比如一些内部项目或传统行业项目（其使用人数有限，而且数据量一般不会太大），使用外键是一个不错的选择，毕竟想要确保相关表始终有一致的数据，使用外键要比在应用程序中检查一致性方便简单许多，此外，外键在相关数据的删除和更新操作上也会比在应用中要高效。

绑定变量

可能大家看到“绑定变量”这个词时，会有一点陌生，换个说法可能会熟悉一些：`prepared statement` 。绑定变量的SQL，使用问号标记可以接收参数的位置，当真正需要执行具体查询的时候，则使用具体的数值代替这些问号，比如：

SQL | 复制代码

```sql
SELECT order_no, order_amount FROM sales WHERE order_status = ? and buyer = ?
```

为什么要使用绑定变量？总所周知的原因是可以预先编译，减少 SQL 注入的风险，除了这些呢？

当创建一个绑定变量 SQL 时，客户端向服务器发送了一个 SQL 语句原型，服务器收到这个 SQL 语句的框架后，解析并存储这个 SQL 语句的部分执行计划，返回给客户端一个 SQL 语句处理句柄，从此以后，客户端通过向服务器发送各个问号的取值和这个句柄来执行一个具体查询，这样就可以更高效地执行大量重复语句，因为：

- 服务器只需要解析一次 SQL 语句
- 服务器某些优化器的优化工作也只需要做一次，因为 MySQL 会缓存部分执行计划
- 通信中仅仅发送的是参数，而不是整个语句，网络开销也会更小，而且以二进制发送参数和句柄要比发送 ASCII 文本的效率更高

需要注意的是，MySQL 并不是总能缓存执行计划，如果某些执行计划需要根据参入的参数来计算时，MySQL 就无法缓存这部分执行计划。

使用绑定变量的最大陷阱是：你知道其原理，但不知道它是如何实现的。有时候，很难解释如下3种绑定变量类型之间的区别：

1. 客户端模拟的绑定变量：客户端的驱动程序接收一个带参数的 SQL，再将参数的值带入其中，最后将完整的查询发送到服务器。
2. 服务器绑定变量：客户端使用特殊的二进制协议将带参数的 SQL 语句发送到服务器端，然后使用二进制协议将具体的参数值发送给服务器并执行。
3. SQL 接口的绑定变量：客户端先发送一个带参数的 SQL 语句到服务器端，这类似于使用 `prepared` 的 SQL 语句，然后发送设置的参数，最后在发送 `execute` 指令来执行 SQL，所有这些都是用普通的文本传输协议。

比如某些不支持预编译的 JDBC 驱动，在调用 `connection.prepareStatement(sql)` 时，并不会把 SQL 语句发送给数据库做预处理，而是等到调用 `executeQuery` 方法时才把整个语句发送到服务器，这种方式就类似于第1种情况。因此，在程序中使用绑定变量时，理解你使用的驱动通过哪种方式来实现就显得很有必要。延伸开来说，对于自己使用的框架、开源工具，不应仅仅停留在会使用这个层面，有时间可以深入了解其原理和实现，不然有可能被骗了都不知道哦。

用户自定义函数

MySQL 本身内置了非常多的函数，比如 `SUM` 、 `COUNT` 、 `AVG` 等等，可实际应用中，我们常常需要更多。大多数情况下，更强大的功能都是在应用层面实现，但实际上 MySQL 也提供了机会让我们可以去扩展 MySQL 函数，这就是用户自定义函数（`user-defined function`），也称为： `UDF` 。需要注意 `UDF` 与存储过程和通过 SQL 创建函数的区别，存储过程只能使用 SQL 来编写，而 `UDF` 没有这个限制，可以使用支持 C 语言调用约定的任何编程语言来实现。
`UDF` 必须事先编译好并动态链接到服务器上，这种平台相关性使得 `UDF` 在很多方面都很强大， `UDF` 速度非常快，而且可以访问大量操作系统功能，还可以使用大量库函数。如果需要一个 MySQL 不支持的统计聚合函数，并且无法使用存储过程来实现，而且还想不同的语言都可以调用，那么 `UDF` 是不错的选择，至少不需要每种语言都来实现相同的逻辑。

所谓能力越大，责任也就越大， `UDF` 中的一个错误可能直接让服务器崩溃，甚至扰乱服务器的内存和数据，因此，使用时需要注意其潜在的风险。在 MySQL 版本升级时也需要注意，因为你可能需要重新编译或者修改这些 `UDF` ，以便让它们能在新版本中工作。

这里有一个简单的示例来展示如何创建 `UDF` ：将结果集转化为 JSON，具体的代码请参考：lib_mysqludf_json。

```sql
-- 1、首先使用c语言实现功能
-- 2、编译
-- 这里省略第1、2步，实现并编译成.so
-- 3、使用SQL创建函数
drop function json_array;
create function json_array returns string soname 'lib_mysqludf_json.so';
-- 4、使用函数
select json_array(
            customer_id
          ,   first_name
          ,   last_name
          ,   last_update
          ) as customer
from   customer
where  customer_id =1;
-- 5、得到的结果如下：
+-----------------------------------------+
| customer                                |
+-----------------------------------------+
| [1,"MARY","SMITH","2006-02-15 04:57:20"] |
+-----------------------------------------+
```

其大致的实现流程：使用 C 语言实现逻辑 -> 编译成 `.so` 文件 -> 创建函数 -> 使用函数。`UDF` 在实际工作中可能很少使用，但作为开发者的我们，了解这么一款强大的工具，在解决棘手问题时，也让我们有了更多的选择。

字符集

最后说说字符集。

关于字符集大多数人的第一印象可能就是：数据库字符集尽量使用 `UTF8`，因为 `UTF8` 字符集是目前最适合于实现多种不同字符集之间的转换的字符集，可以最大程度上避免乱码问题，也可以方便以后的数据迁移。But why?

字符集是指一种从**二进制编码到某类字符符号的映射**，可以参考如何使用一个字节来表示英文字母。校对规则是指一组**用于某个字符集的排序规则**，即采用何种规则对某类字符进行排序。MySQL 每一类编码字符都有其对应的字符集和校对规则。MySQL 对各种字符集的支持都非常完善，但同时也带来一些复杂性，某些场景下甚至会有一些性能牺牲。

一种字符集可能对应多种校对规则，且都有一个默认校对规则，那在 MySQL 中是如何使用字符集的？在 MySQL 中可以通过两种方式设置字符集：创建对象时设置默认值、客户端与服务器通信时显式设置。

MySQL 采用“阶梯”式的方式来设定字符集默认值，每个数据库，每张表都有自己的默认值，它们逐层继承，最终最靠底层的默认设置将影响你创建的对象。比如，创建数据库时，将根据服务器上的 `character_set_server` 来设置数据库的默认字符集，同样的道理，根据 `database` 的字符集来指定库中所有表的字符集……不管是对数据库，还是表和列，只有当它们没有显式指定字符集时，默认字符集才会起作用。

当客户端与服务器通信时，它们可以使用不同的字符集，这时候服务器将进行必要的转换工作。当客户端向服务器发送请求时，数据以 `character_set_client` 设置的字符集进行编码；而当服务器收到客户端的 SQL 或者数据时，会按照 `character_set_connection` 设置的字符集进行转换；当服务器将要进行增删改查等操作前会再次将数据转换成 `character_set_database`（数据库采用的字符集，没有单独配置即使用默认配置，具体参考上文），最后当服务器返回数据或者错误信息时，则将数据按 `character_set_result` 设置的字符集进行编码。服务器端可以使用 `SET CHARACTER SET` 来改变上面的配置，客户端也可以根据对应的 API 来改变字符集配置。客户端和服务器端都使用正确的字符集才能避免在通信中出现问题。

那如何选择字符集？

在考虑使用何种字符集时，最主要的衡量因素是存储的内容，在能够满足存储内容的前提下，尽量使用较小的字符集。因为更小的字符集意味着更少空间占用、以及更高的网络传输效率，也间接提高了系统的性能。如果存储的内容是英文字符等拉丁语系字符的话，那么使用默认的 `latin1` 字符集完全没有问题，如果需要存储汉字、俄文、阿拉伯语等非拉丁语系字符，则建议使用 `UTF8` 字符集。当然不同字符在使用 `UTF8` 字符集所占用的空间是不同的，比如英文字符在 `UTF8` 字符集中只使用一个字节，而一个汉字则占用 3 个字节。

除了字符集，校对规则也是我们需要考虑的问题。对于校对规则，一般来说只需要考虑是否以大小写敏感的方式比较字符串或者是否用字符串编码的二进制来比较大小，其对应的校对规则的后缀分别是 `_cs`、`_ci` 和 `_bin`。大小写敏感和二进制校对规则的不同之处在于，二进制校对规则直接使用字符的字节进行比较，而大小写敏感的校对规则在多字节字符集时，如德语，有更复杂的比较规则。举个简单的例子，`UTF8` 字符集对应校对规则有三种：

- `utf8_bin` 将字符串中的每一个字符用二进制数据存储，区分大小写
- `utf8_general_ci` 不区分大小写，`ci` 为 `case insensitive` 的缩写，即大小写不敏感
- `utf8_general_cs` 区分大小写，`cs` 为 `case sensitive` 的缩写，即大小写敏感

比如，创建一张表，使用 `UTF8` 编码，且大小写敏感时，可以使用如下语句：

```sql
CREATE TABLE sales (
    order_no VARCHAR(32) NOT NULL PRIMARY KEY,
    order_amount INT NOT NULL DEFAULT 0,
    ......
) ENGINE=InnoDB COLLATE=utf8_general_cs;
```

因此，在项目中直接使用 `UTF8` 字符集是完全没有问题的，但需要记住的是不要在一个数据库中使用多个不同的字符集，不同字符集之间的不兼容问题很难缠。有时候，看起来一切正常，但是当某个特殊字符出现时，一切操作都会出错，而且你很难发现错误的原因。

字符集对数据库的性能有影响吗?

某些字符集和校对规则可能会需要多个的 CPU 操作，可能会消耗更多的内存和存储空间，这点在前文已经说过。特别是在同一个数据库中使用不同的字符集，造成的影响可能会更大。

不同字符集和校对规则之间的转换可能会带来额外的系统开销，比如，数据表 `sales` 在 `buyer` 字段上有索引，则可以加速下面的 `ORDER BY` 操作：

```sql
SELECT order_no,order_amount FROM sales ORDER BY buyer;
```

只有当 SQL 查询中排序要求的字符集与服务器数据的字符集相同时，才能使用索引进行排序。你可能会说，这不是废话吗？其实不然，MySQL 是可以单独指定排序时使用的校对规则的，比如：

```sql
-- 你说，这不是吃饱了撑的吗？我觉得也是，也许会有其适用的场景吧
-- 这时候就不能使用索引排序呢，只能使用文件排序
SELECT order_no,order_amount FROM sales ORDER BY buyer COLLATE utf8_bin;
```

当使用两个字符集不同的列来关联两张表时，MySQL 会尝试转换其中一个列的字符集。这和在数据列外面封装一个函数一样，会让 MySQL 无法使用这个列上的索引。关于 MySQL 字符集还有一些坑，但在实际应用场景中遇到的字符集问题，其实不是特别的多，所以就此打住。

结语

MySQL 还有一些其他高级特性，但在大多数场景下我们很少会使用，因此这里也没有讨论，但多了解一些总是好的，至少在需要的时候，你知道有这样一个东西。我们非常多的人，**总是会认为自己所学的知识就像碎片一样不成体系，又找不到解决办法，那你有没有想过也许是碎片不够多的缘故？点太少，自然不**

能连接成线，线太少，自然不能结成网。因而，没有其他办法，**保持好奇心、多学习、多积累，量变总有一天会质变**，写在这儿，与大家共勉吧。

前面我写的一些文章里面会有提到过，架构设计是一种平衡的艺术，其实质应该是一种妥协，是对现有资源的一种妥协。有时候我们会不自觉的陷入某一个点，比如，为了追求数据的扩展性，很多人一上来就开始分库分表，然后把应用搞得非常复杂，到最后表里还没有装满数据，项目就已经死了。所以在资源有限或者未来还不可知的情况下，尽量使用数据库、语言本身的特性来完成相应的工作，是不是会更好一点。解决大数据问题，也不只是分库分表，你还应该还可以想到分区；有些业务即使在分布式环境下也不一定非要在业务层完成，合理使用存储过程和触发器，也许会让你更轻松.....

最后，本文所讨论的知识点均出自《高性能 MySQL》，强烈建议大家读一读这本书。

参考资料

[1] Baron Scbwartz 等著；宁海元 周振兴等译；高性能MySQL（第三版）；电子工业出版社， 2013