写给运营同学和初学者的 Sql 入门教程

Preface

为什么以《写给运营同学和初学者的 Sql 入门教程》为题？

这原本是给一位关系要好的运营同学定制的 Sql 教程。在饿了么，总部运营的同学在排查、跟踪线上问题和做运营决策的时候，除了通过运营管理系统查询信息和依赖数据分析师给出的分析数据，常常也需要直接从数据库管理台通过写 Sql 的方式获取更细致、更实时的业务数据，并基于这些数据进行一些及时的分析，从而更快的给出运营方案。在这样的背景下，Sql 已经越来越成为我们运营同学的一项必备技能。网上有很多 Sql 教程（e.g. w3school），我也翻阅过一些运营同学桌上的 Sql 纸质书，这些教程都很好，但普遍侧重介绍语法，很多很多的语法，配以简短的 demo。作为老司机的 reference book 很赞，可是对于刚入门甚至还没有入门的学习者，就未免困难了一点。再回顾运营同学的使用场景，大多数情况下是根据一些已有条件做一些简单的查询，偶尔会有一些相对复杂的查询，比如对查询结果做一些聚合、分组、排序，或是同时查询两三张数据表，除此以外，建表、建索引、修改表字段、修改字段值等等这些操作，在运营同学的日常工作中基本是不会遇到的。

基于以上种种原因，写了这篇教程，初衷是能够帮助这位好朋友以更高的 ROI 入门 Sql。下面是写这篇教程时的一些考量：

1. 从数据库、数据表的基础概念以及最简单的 Sql 语法开始，做一些必要的铺垫，后续的每一章再逐步增加难度；
2. 只介绍查询语法（更准确的说，是最常用的查询语法），将它们的用法和套路解释清楚，不涉及日常工作暂时还用不到的建表、修改表结构等等操作，避免铺天盖地的语法一个接一个反而干扰了学习的节奏；
3. 通过穿插一些小测验、小温习，及时检验对知识点的理解和复习已经学习过的知识点；
4. 结合一些业务场景的 demo 帮助理解；
5. 结尾提供一章快速复习，既是复习，也可以自测还有哪些知识点没有掌握到位；

建议所有阅读教程的同学，都尝试搭建一套自己的数据库服务（建议安装 MySQL），对教程中的 demo 多做一些练习，不论是示例、小测验还是小温习里面的 Sql 语句，都不妨亲自执行一下，这也是一种很好的帮助你熟悉语法的方式。当然搭建自己的数据库会是一个不小的挑战，写作这篇教程的时候，我在自己的 VPS 上安装了 MySQL(MariaDB) 并提供了一个连接脚本（隐藏了连接 MySQL 的过程）给朋友使用，但是这种方式并不适合推广到所有人。具体的安装和使用方式，不在本教程的叙述范围内，所以…运营妹子们可以求助下熟悉的研发同学，汉子们嘛..

1. 数据创建脚本-通过该脚本导入 demo 数据到 MySQL 中
2. 链接 VPS 上 MySQL 的脚本-基本原理是建立了一个 ssh tunnel，在 tunnel 里和远端的 MySQL 通信，但实际上本地建 MySQL 服务更方便，仅供参考

by 多肉

Introduction

其实 Sql 并没有那么难。Sql 是帮助你和关系型数据库交互的一套语法，主要支持的操作有 4 类：

1. DQL：其实就是数据查询操作，通过特定的语法查询到数据库里你想要的数据结果，并且展示出来；
2. DML：数据修改操作，包括更新某个数据字段、向某张表里面插入一条新的数据等等；
3. DDL：数据定义操作，比如创建一张新的表或是创建一个新的索引（索引是啥？我们后面专门聊一聊它）；
4. DCL：数据授权操作，比如授权特定的人可以查询某张特定的表；

听起来挺吓人的对吧，但实际上 DML、DDL、DCL 这 3 类操作在日常的运营工作中几乎都不会用到，经常会使用到的呐其实是第一种，也就是数据查询操作（DQL）。Sql 基本的查询语法也比较简单，那么难在哪里呢？我猜想难在学习了基本语法之后，不知道怎么应用到实际的 Case 上。在接下来的内容里，我将以一些十分接近现实的众包运营 Case 为例，逐一解释最基本的 Sql 查询语法并且分析如何将它应用到具体的场景上。

1 预备知识

好的吧，吹了一波牛还是逃不过需要介绍一些最基础的东西，但是我保证这是整篇教程中最枯燥的部分，后面就会有趣很多。

1.1 数据库和数据表

为了更简单的理解这两个概念以及他们之间的关系，可以这么类比：

1. 数据表：就是一张表格，想象一张你经常在使用的 Excel 表，有行有列，每一行就是一条数据，每一列对应了这条数据的某一个具体的字段，当然这张表还有一个表名。数据表也是如此，只是字段名、表名不像 Excel 表格那样好理解，比如 Excel 表格里面某一列的字段名可能叫骑手 id，而对应到数据表里面可能就叫做 rider_id，Excel 的表名可能叫骑手基本信息表，而对应到数据表的表名则可能叫 tb_rider；
2. 数据库：就是集中管理一批相关的表格的地方。你可以把它理解成是一个文件夹。平时你可能会创建一个名叫众包业务的文件夹，然后将众包运营相关的 Excel 表格都放在这个文件夹里面。数据库也是如此，比如我们会有一个库名叫做 crowd 的数据库，里面可能存放了 tb_rider、tb_order 等和骑手、运单相关的多张数据表；

所以，“关系型数据库”的概念很吓唬人，但其实道理很简单，就是列和列之间有一定的联系，整合在一起就是一条有意义的数据，将这些数据归纳起来就构成了一张表，而将一批有关联的表一同管理起来就得到了一个数据库。

1.2 最基本的 Sql 查询语法

最基本的 Sql 查询语法其实就一个：

```
SELECT 列名（或者*，表示所有列）FROM 表名 WHERE 筛选条件;
```

> B.T.W 注意 SELECT...FROM...WHERE...; 语句结尾的这个分号，在标准 Sql 语法中这个分号是必要的

让我们按照 FROM、WHERE、SELECT 的顺序理解一下这个语法：

1. FROM 表名：顾名思义，就是从表名指定的这张表格中；
2. WHERE 筛选条件：意思是“当满足筛选条件”的时候；
3. SELECT 列名：意思是选择出这些记录，并且展示指定的列名；

串联起来便是，从 FROM 后面指定的数据表中，筛选出满足 WHERE 后面指定条件的数据，并且展示 SELECT 后指定的这几列字段。是不是很简单呐？不过好像抽象了一点。所以我们来看几个具体的超简单的例子。假设我们有一张学生数学期末考试成绩表，数据表长下面这样，表名叫作 tb_stu_math_score。

id(自增主键)	name(学生姓名)	number(学号)	grade(年级)	class(班级)	score(得分)
1	柯南	010201	1	2	100
2	小哀	010202	1	2	100
3	光彦	010203	1	2	98
4	步美	010204	1	2	95
5	元太	010205	1	2	59

让我们试着理解一下下面几个查询语句：

```
[1] SELECT name FROM tb_stu_math_score WHERE score >= 95;
```

从 tb_stu_math_score 表中挑选出得分大于 95 分的学生姓名，得到的结果显而易见：

name
柯南
小哀
光彦
步美

```
[2] SELECT name, number FROM tb_stu_math_score WHERE score < 60;
```

从 tb_stu_math_score 表中挑选出得分小于 60 分的学生姓名，得到的结果是：

name	number
元太	010205

```
[3] SELECT * FROM tb_stu_math_score WHERE score = 100;
```

从 tb_stu_math_score 表中挑选出得分为 100 分学生的所有信息（注意 SELECT 后面的 * 符号，表示所有字段），得到的结果是：

id	name	number	grade	class	score
1	柯南	010201	1	2	100
2	小哀	010202	1	2	100

小测验

看看下面这些 Sql 查询语句你是不是知道是什么含义并且知道查询结果是什么了呢？

```
1. SELECT name, grade, class, score FROM tb_stu_math_score WHERE number = "010201";
2. SELECT * FROM tb_stu_math_score WHERE name = " 小哀";
3. SELECT id, score FROM tb_stu_math_score WHERE number = "010202";
```

2 更进一步

刚刚我们学习了 Sql 查询的最最最最基础的语法，但是相信我，所有的 Sql 查询几乎都长这个样子，所以理解了这个最基础的语法结构，后面学习起来就轻松多了。接下来让我通过一些例子，扩展这个基础语法，教你一些更加高级的 Sql 查询操作。不过首先，我们还是要看一下接下来我们的范例数据表长啥样。

假设我们有一张骑手数据表，表名叫作 tb_rider，还有一张运单数据表，表名叫作 tb_order，这两张表分别长下面这个样子。

[1] 骑手数据表：tb_rider

id	name	real_name_certify_state	level	level_city	is_deleted	created_at	updated_at
1	Stark	2	3	1	0	2017-01-01 22:00:19	2018-01-01 06:40:01
2	Banner	2	3	9	0	2017-04-28 12:01:19	2018-01-01 06:40:01
3	Rogers	2	2	1	0	2017-04-10 17:24:01	2018-01-01 06:40:01
4	Thor	1	0	1	0	2017-12-31 23:10:39	2018-01-01 06:40:01
5	Natasha	2	1	1	0	2017-02-11 15:03:13	2018-01-01 06:40:01
6	Barton	2	1	9	0	2017-02-11 15:04:19	2018-01-01 06:40:01
7	Coulson	2	3	9	0	2017-01-03 23:00:22	2018-01-01 06:40:01
8	Coulson	1	0	2	0	2017-01-05 10:10:23	2018-01-01 06:40:01

字段含义：

1. id：自增主键。又是一个听起来很吓人的名字，但实际含义很简单。“自增”的意思是，每次在这张数据表中创建一条新记录的时候，数据库都会在上一个 id 值的基础上自动加上一个固定的步长（默认就是 +1）作为新记录的 id 值。而所谓“主键”，就是能够在一张数据表中唯一标识一条记录的字段，因为每条记录的 id 都不一样，所以它是主键。这个字段可以是为了单纯的标识数据的唯一性，也可以具有一些业务含义，比如这里的 id 就同时也是骑手的账号 id；
2. name：骑手姓名；
3. real_name_certify_state：实名认证状态：1-认证中，2-认证通过，3-认证失败；
4. level：骑手等级，3-金牌，2-银牌，1-铜牌，0-普通；
5. level_city：等级城市；
6. is_deleted：这条数据是否有效，在大多数产线相关的数据表中，都会有这样的标记字段。0-未删除（表示有效），1-已删除（表示无效）；
7. created_at：这条数据的创建时间，也是所有产线数据表中都必须有的字段；
8. updated_at：这条数据最新一次被更新的时间，同样是所有产线数据表中都必须有的字段；

[2] 运单数据表：tb_order

id	order_id	order_state	rider_id	rider_name	grabbed_time	created_at	updated_at
1	300000201712300001	40	1	Stark	2017-12-30 12:34:55	2017-12-30 12:34:17	2017-12-30 12:39:30
2	300000201712300002	40	1	Stark	2017-12-30 12:34:56	2017-12-30 12:34:18	2017-12-30 12:44:27
3	300000201712300003	40	2	Banner	2017-12-30 13:23:12	2017-12-30 13:20:02	2017-12-30 13:54:09
4	300000201712300004	40	5	Natasha	2017-12-30 13:35:03	2017-12-30 13:34:19	2017-12-30 14:03:17
5	300000201712300005	40	1	Stark	2017-12-30 16:01:22	2017-12-30 16:01:03	2017-12-30 16:08:21
6	300000201712300006	40	3	Rogers	2017-12-30 16:10:45	2017-12-30 16:08:57	2017-12-30 16:34:27
7	300000201712310001	20	6	Barton	2017-12-31 09:12:57	2017-12-31 09:12:07	2017-12-31 09:20:35
8	300000201712310002	80	7	Coulson	2017-12-31 09:15:01	2017-12-31 09:10:33	2017-12-31 09:20:17
9	300000201712310003	80	2	Banner	2017-12-31 09:20:17	2017-12-31 09:18:10	2017-12-31 09:22:24
10	300000201712310004	20	3	Rogers	2017-12-31 10:37:33	2017-12-31 10:34:01	2017-12-31 10:38:09
11	300000201712310005	10	0		1970-01-01 00:00:00	2017-12-31 19:29:02	2017-12-31 19:29:02
12	300000201712310006	10	0		1970-01-01 00:00:00	2017-12-31 19:29:27	2017-12-31 19:29:27
13	300000201712310007	10	0		1970-01-01 00:00:00	2017-12-31 19:30:01	2017-12-31 19:30:01

字段含义：

1. id：自增主键。呐，这里的 id 就是单纯的主键作用，没有其他的业务含义；
2. order_id：运单号，业务层面上运单的唯一标识；
3. order_state：运单当前的状态。10-待抢单，20-待到店，80-待取餐，40-已送达；
4. rider_id：抢单的骑手 id，还未被抢的运单这个字段是默认值 0；
5. rider_name：抢单的骑手姓名，还未被抢的运单这个字段是默认值空字符；
6. grabbed_time：抢单时间，还未被抢的运单这个字段是默认的 “1970-01-01 00:00:00”(这是一个特殊的时间，有兴趣的话可以搜索关键词：时间戳)；
7. created_at：这条数据的创建时间，也是所有产线数据表中都必须有的字段；
8. updated_at：这条数据最新一次被更新的时间，同样是所有产线数据表中都必须有的字段；

小温习

试着理解看看下面这几条 Sql 的含义以及返回的数据结果吧？

```
1. SELECT name, real_name_certify_state FROM tb_rider WHERE level = 3;
2. SELECT * FROM tb_order WHERE rider_id = 1;
3. SELECT rider_id, rider_name, order_id, grabbed_time FROM tb_order
   WHERE order_state = 40;
```

2.1 IN 操作

场景：线下反馈了一批骑手说自己理应是上海的金牌，但是牌级是普通或者展示的是金牌却没有享受到上海的金牌活动，你已经知道了这几个分别是 id=(2, 4, 7) 的骑手，想排查一下他们的等级更新情况。

这时你可以选择像这样一条一条的查询，像之前我们介绍的那样：

```
1. SELECT name, real_name_certify_state, level, level_city FROM tb_rider WHERE id=2;
2. SELECT name, real_name_certify_state, level, level_city FROM tb_rider WHERE id=4;
3. SELECT name, real_name_certify_state, level, level_city FROM tb_rider WHERE id=7;
```

这样当然可以达到目的，但是只有两三个骑手的时候还勉强可以操作，如果有几十个骑手这样查起来就太费劲了。这时候我们可以使用 IN 这个语法。

```
SELECT name, real_name_certify_state, level, level_city FROM tb_rider WHERE id IN(2, 4, 7);
```

很简单的对吧？但我们还是来简单理解一下，WHERE id IN(2, 4, 7) 的意思就是筛选 id 字段的值在 2, 4, 7 这几个值当中的记录，执行这条 Sql 语句你就会得到下面这样的结果。

name	real_name_certify_state	level	level_city
Banner	2	3	9
Thor	1	0	1
Coulson	2	3	9

于是你会发现，Thor 这个骑手因为他没有通过实名认证所以肯定评不上金牌，Banner 和 Coulson 两位骑手虽然都是金牌骑手，但是等级城市却是福州，所以享受不到上海金牌的活动。

那如果不知道骑手 id，只知道骑手的名字怎么办？也可以使用 IN 查询，只是这时候筛选的条件变成了 name，取值范围也变成了 “Banner”，“Thor”，“Coulson”。就像这样。

```
SELECT name, real_name_certify_state, level, level_city FROM tb_rider
WHERE name IN("Banner", "Thor", "Coulson");
```

于是你顺利的得到了以下的结果。

name	real_name_certify_state	level	level_city
Banner	2	3	9
Thor	1	0	1
Coulson	2	3	9
Coulson	1	0	2

Oops! 居然有两个 Coulson！

这就是在实际应用中要特别注意的地方了：> 当你使用类似骑手 id 这种被设计为唯一值的字段作为查询依据时，返回的结果也是唯一的，而当你使用类似骑手姓名这类字段作为查询依据时，就有可能出现上面这种情况。这时候你就需要依赖更多的信息来判断，哪一条才是你真正想要的。所以能够用明确的字段作为查询依据时就要尽可能的使用。

2.2 关系运算符：AND 和 OR

最常用的关系运算符有两个 AND 和 OR，用来连接多个筛选条件。顾名思义，AND 就是“并且”的意思，也就是同时满足 AND 前后两个筛选条件；OR 就是“或者”的意思，也就是满足 OR 前后任何一个筛选条件。有点抽象了对不对，我们看一个具体的例子。

场景：假设你想要看看 2017-02-01(包括 2017-02-01 当天）到 2017-06-01(不包括 2017-06-01 当天）期间注册的骑手所有信息。

注册时间对应到数据上就是骑手信息的创建时间（created_at），换句话说，就是查询 tb_rider``表中创建时间处于 2017-02-01 到 2017-06-01 之间的数据。那这样的 Sql 应该怎么写呢，这时我们就可以用到 AND“‘。

```
SELECT * FROM tb_rider WHERE created_at >= "2017-02-01 00:00:00"
AND created_at < "2017-06-01 00:00:00";
```

> B.T.W 注意因为包括 2017-02-01 当天，而不包括 2017-06-01 当天，所以前者是 >=，而后者是 <。

让我们再来推广一下。假设现在的场景变成：想看一看 2017-02-01(包括当天）之前，或者 2017-06-01(包括当天）之后注册的骑手所有信息。我们应该怎么写这个 Sql 呢？既然是或的关系，我们就应该使用 OR 了。

```
SELECT * FROM tb_rider WHERE created_at <= "2017-02-01 00:00:00"
OR created_at >= "2017-06-01 00:00:00";
```

> B.T.W 注意这里既包括了 2017-02-01 当天，又包括了 2017-06-01 当天，所以前者是 <=，后者是 >=。

当然啦，AND 和 OR 这样的关系运算符，不仅仅能够连接前后两个筛选条件，也可以通过使用若干个 AND 和 OR 连接多个不同的筛选条件。比如：想要看看 2017-02-01(包括 2017-02-01 当天）到 2017-06-01(不包括 2017-06-01 当天）期间注册的且当前是金牌等级的骑手所有信息，那么我们可以这么写。

```
SELECT * FROM tb_rider
WHERE created_at >= "2017-02-01 00:00:00"
AND created_at < "2017-06-01 00:00:00"
AND level = 3;
```

2.3 排序：ORDER BY

让我们先小小的复习一下上面学到的知识点，有一个这样的场景：

> 我们打算看一下 Stark 这位骑手，在 2017-12-30 当天抢单且当前状态为已完成的运单号和运单的创建时间。

如何写这个 Sql 呢？先思考 3s…1…2…3，看看是否和你想的一样。

```
SELECT order_id, created_at FROM tb_order
WHERE rider_id = 1
AND grabbed_time >= "2017-12-30 00:00:00"
AND grabbed_time < "2017-12-31 00:00:00"
AND order_state = 40;
```

如果你没有写对，没关系，让我们来分析一下：

1. Stark 这位骑手的骑手 id 是 1，所以我们的第一个筛选条件为 rider_id = 1；
2. 因为我们要看 2017-12-30 当天抢单的运单 id，确定了我们的第二个筛选条件是抢单时间，对应的是 grabbed_time 这个字段，而 2017-12-30 当天，实际上指的就是 2017-12-30 00:00:00(包括）到 2017-12-31 00:00:00(不包括）这段时间；
3. 最后是已完成这个条件，order_state 字段标识了运单状态，因此我们的筛选条件是 order_state = 40；

执行这个语句，我们得到了下面这样的结果。

order_id	created_at
300000201712300001	2017-12-30 12:34:17
300000201712300002	2017-12-30 12:34:18
300000201712300005	2017-12-30 16:01:03

有点美中不足，我想按照运单的创建时间倒序排序把最近创建的运单排在最前面，这时候就可以使用 ORDER BY 语法了。

```
SELECT order_id, created_at FROM tb_order
WHERE rider_id = 1
AND grabbed_time >= "2017-12-30 00:00:00"
AND grabbed_time < "2017-12-31 00:00:00"
AND order_state = 40
ORDER BY created_at DESC;
```

让我们再来理解一下，DESC 是“递减”的意思，与之对应的是 ASC 递增。ORDER BY created_at DESC 的含义是，按照（BY)created_at 字段值递减（DESC）的顺序对查询结果排序（ORDER)。于是我们得到如下的结果。

order_id	created_at
300000201712300005	2017-12-30 16:01:03
300000201712300002	2017-12-30 12:34:18
300000201712300001	2017-12-30 12:34:17

> B.T.W 在现实场景中有时候查询结果的集合会很大（例如几百行、几千行），但是我们只想看其中前 10 行的数据，这时候我们可以使用 LIMIT 语法。例如这里我们可以使用 LIMIT 语法仅仅展示前两行查询结果：SELECT order_id, created_at FROM tb_order WHERE rider_id = 1 AND grabbed_time >= “2017-12-30 00:00:00” AND grabbed_time < “2017-12-31 00:00:00” AND order_state = 40 ORDER BY created_at DESC LIMIT 2;

我们再来看一个更加复杂的场景：假设想要查询 2017-12-30 和 2017-12-31 两天所有运单的所有信息，并先按照骑手 id 递增，再按运单状态递减的顺序排序展示。还是先思考一会儿。

这时的 Sql 类似长这样。

```
SELECT * FROM tb_order
WHERE created_at >= "2017-12-30 00:00:00"
```

```
AND created_at < "2018-01-01 00:00:00"
ORDER BY rider_id ASC, order_state DESC;
```

如果前面的每个知识点都理解了，这里应该就只对“先按照骑手 id 递增，再按运单状态递减的顺序排序展示”有所疑惑。实际上我们不仅可以对一个字段排序，还可以把多个字段作为排序的依据，而且不同字段上的排序规则（递增/递减）可以不同。但排序是有优先级的，比如这里，只有当 rider_id 字段的值都相同无法区分顺序时，才会对相同 rider_id 的这几条数据再按照 order_state 字段的值进行排序。举例来说，rider_id = 2 且 order_state = 80 的数据，也依然不可能排在 rider_id = 1 且 order_state = 40 的数据前面。

执行这条 Sql 语句，将得到的结果如下。

id	order_id	order_state	rider_id	rider_name	grabbed_time	created_at	updated_at
11	300000201712310005	10	0		1970-01-01 00:00:00	2017-12-31 19:29:02	2017-12-31 19:29:02
12	300000201712310006	10	0		1970-01-01 00:00:00	2017-12-31 19:29:27	2017-12-31 19:29:27
13	300000201712310007	10	0		1970-01-01 00:00:00	2017-12-31 19:30:01	2017-12-31 19:30:01
1	300000201712300001	40	1	Stark	2017-12-30 12:34:55	2017-12-30 12:34:17	2017-12-30 12:39:30
2	300000201712300002	40	1	Stark	2017-12-30 12:34:56	2017-12-30 12:34:18	2017-12-30 12:44:27
5	300000201712300005	40	1	Stark	2017-12-30 16:01:22	2017-12-30 16:01:03	2017-12-30 16:08:21
9	300000201712310003	80	2	Banner	2017-12-31 09:20:17	2017-12-31 09:18:10	2017-12-31 09:22:24
3	300000201712300003	40	2	Banner	2017-12-30 13:23:12	2017-12-30 13:20:02	2017-12-30 13:54:09
6	300000201712300006	40	3	Rogers	2017-12-30 16:10:45	2017-12-30 16:08:57	2017-12-30 16:34:27
10	300000201712310004	20	3	Rogers	2017-12-31 10:37:33	2017-12-31 10:34:01	2017-12-31 10:38:09
4	300000201712300004	40	5	Natasha	2017-12-30 13:35:03	2017-12-30 13:34:19	2017-12-30 14:03:17
7	300000201712310001	20	6	Barton	2017-12-31 09:12:57	2017-12-31 09:12:07	2017-12-31 09:20:35
8	300000201712310002	80	7	Coulson	2017-12-31 09:15:01	2017-12-31 09:10:33	2017-12-31 09:20:17

这个部分相对有一点难，可以多对比着例子理解一下。

3 高级一点的话题

进入到这个部分，说明之前的内容你基本都已经掌握了，在日常运营的操作中有 30% 左右的场景都可以使用前面讲述的这些知识点解决（当然会有个熟能生巧的过程）。这个部分，我将继续介绍几个更加高级、当然也更加有难度的 Sql 技能，当你结束这一部分的学习并且熟练掌握这些技能的时候，你会发现绝大部分需要通过查数据来确认的场景你都可以胜任。因为这个章节的内容本身难度又大了些，如果再对着一张复杂的表就更加难以关注重点，因此我们精简一下表结构，只保留一些必要的字段。新的 tb_order 表如下。

id	order_id	order_state	rider_id	rider_name	merchant_customer_distance	created_at
1	300000201712300001	40	1	Stark	2.5	2017-12-30 12:34:17
2	300000201712300002	40	1	Stark	1.8	2017-12-30 12:34:18
3	300000201712300003	40	2	Banner	1.8	2017-12-30 13:20:02
4	300000201712300004	40	5	Natasha	2.7	2017-12-30 13:34:19
5	300000201712300005	40	1	Stark	1.2	2017-12-30 16:01:03
6	300000201712300006	40	3	Rogers	0.5	2017-12-30 16:08:57
7	300000201712310001	20	6	Barton	1.3	2017-12-31 09:12:07
8	300000201712310002	80	7	Coulson	2.9	2017-12-31 09:10:33
9	300000201712310003	80	2	Banner	0.7	2017-12-31 09:18:10
10	300000201712310004	20	3	Rogers	2.2	2017-12-31 10:34:01
11	300000201712310005	10	0		0.3	2017-12-31 19:29:02
12	300000201712310006	10	0		1.3	2017-12-31 19:29:27
13	300000201712310007	10	0		3.0	2017-12-31 19:30:01

新增的列： merchant_customer_distance：配送距离（商家到用户的直线距离），单位是千米（km）。

3.1 聚合函数：COUNT, SUM, AVG

千万别被聚合函数这个名字吓唬到，可以简单的理解为对数据进行一些加工处理，让我们先来分别看一下这几个聚合函数的基本定义。

1. COUNT：对查询结果集合中特定的列进行计数；
2. SUM：对查询结果的某个字段进行求和；
3. AVG：就是 average 的意思，对查询结果的某个字段计算平均值；

让我们分别来看几个具体的例子。

[1] 场景：查询 2017-12-30 这一天，骑手 Stark 的所有完成单（状态为 40）总量

你可以这样来写这个 Sql。

```
SELECT COUNT(id) FROM tb_order WHERE rider_id = 1
AND order_state = 40 AND created_at >= "2017-12-30 00:00:00"
AND created_at < "2017-12-31 00:00:00";
```

到这里你应该已经能够很好的理解 WHERE...AND...AND... 这部分的含义，我们就不再过多的讨论这个部分（对自己要有信心！试着理解先自己理解一下）。

让我们重点来看一下 COUNT(id) 这部分的含义。其实很简单，就是对 id 这一列进行计数。连起来看这段 Sql，意思就是：从 tb_order 这张表中（FROM tb_order）筛选（WHERE）骑手 id 为 1(rider_id = 1）且运单状态为已完成（order_state = 40）且创建时间大于等于 2017 年 12 月 30 日（created_at >= "2017-12-30 00:00:00）且创建时间小于 2017 年 12 月 31 日（created_at < "2017-12-31 00:00:00）的数据，并且按照 id 这列对返回的结果集合进行计数。

我们看到 tb_order 这张表中,2017-12-30 当天由骑手 Stark 配送且状态是已完成的运单分别是 300000201712300001、300000201712300002、300000201712300005 这几个运单号的运单，对应的自增 id 分别是 id=[1, 2, 5]，所以对 id 这一列进行计数得到的结果是 3。所以我们得到的查询结果如下表。

COUNT(id)
3

有时候你仅仅是想查一下满足某个条件的记录的总行数，而并非想对某个特定的列进行计数，这时就可以使用 COUNT(*) 语法。比如上面的这个 Sql 也可以写成下面这个样子。

```
SELECT COUNT(*) FROM tb_order WHERE rider_id = 1
AND order_state = 40 AND created_at >= "2017-12-30 00:00:00"
AND created_at < "2017-12-31 00:00:00";
```

因为返回的结果有三行，所以我们会得到下表的结果。

COUNT(*)
3

看起来 COUNT(列) 和 COUNT(*) 是完全等价的？有些特定的场景下的确如此，这里需要补充一下 COUNT 的两个小脾气。

1. COUNT 不会自动去重；
2. COUNT 在某一条查询结果中，用来计数的那一列的值为“空”时，这条记录不进行计数；

    B.T.W 注意这里的“空”指的是 <null>，而不是某一列没有展示出任何值就是空，这是一个相对技术的概念，当前不理解可以先跳过

有一点晕是吗？不着急，我们来看两个例子。假设有两张表，很简单的表，长下面这样。

示例表 1：tb_sample_1

id	name
1	Stark
2	Stark
3	Coulson
4	Natasha
5	Stark

示例表 2：tb_sample_2

id	name
1	Stark
2	Stark
3	<null>
4	<null>
5	Natasha
6	Coulson

我们下猜一猜下面几条 Sql 的执行结果分别是什么？

```
1. SELECT COUNT(id) FROM tb_sample_1;
2. SELECT COUNT(*) FROM tb_sample_1;
3. SELECT COUNT(name) FROM tb_sample_1;
4. SELECT COUNT(name) FROM tb_sample_2;
```

    B.T.W 当 SELECT...FROM...WHERE... 语句中的 WHERE... 部分被省略时，表示查询表中的所有数据（不对数据进行筛选）。

让我们逐一分析一下。

```
1. SELECT COUNT(id) FROM tb_sample_1;
```

这条 Sql 没有太多可以分析的，因为 tb_sample_1 表中 id 字段的取值范围是 id=[1, 2, 3, 4, 5]，共 5 个，所以我们得到的结果如下。

COUNT(id)
5

```
2. SELECT COUNT(*) FROM tb_sample_1;
```

这条 Sql 也没有太多需要分析的，因为 COUNT(*) 的含义是计算查询结果的总行数，tb_sample_1 共 5 行数据，所以我们得到的结果如下。

COUNT(*)
5

```
3. SELECT COUNT(name) FROM tb_sample_1;
```

这条 Sql 里面我们对 name 这一列进行计数，tb_sample_1 表中包含 3 个 Stark，1 个 Coulson 和 1 个 Natasha，因为 COUNT 不进行自动去重，因此结果是 5=3(Stark)+1(Coulson)+1(Natasha)，如下表。

COUNT(name)
5

```
4. SELECT COUNT(name) FROM tb_sample_2;
```

这条 Sql 语句我们还是对 name 这一列进行计数，tb_sample_2 表中包含 2 个 Stark，1 个 Coulson，1 个 Natasha 以及 2 个 <null>，由于 COUNT 不去重因此 2 个 Stark 都会被计数，但 COUNT 不会对值为“空”的结果进行计数，因此两个 <null> 都会被忽略。所以最终的结果为 4=2(Stark)+1(Coulson)+1(Natasha)，如下表。

COUNT(name)
4

[2] 场景：查询 Stark 这名骑手的累计配送里程

让我们先定义一下累计配送里程：骑手所有配送完成单的配送距离（商家到用户的直线距离）之和。

这里的关键词是求和，所以我们要用到 SUM 这个聚合函数。对字段求和的意思是把返回的结果集合中该字段的值累加起来。让我们看下这个场景的 Sql 怎么写。

```
SELECT SUM(merchant_customer_distance) FROM tb_order
WHERE rider_id = 1 AND order_state = 40;
```

让我们来分析一下这条语句，FROM tb_order WHERE rider_id = 1 AND order_state = 40 已经比较好理解了，就是从 tb_order 表中筛选出骑手 id 为 1 且配送状态为 40 的记录。而这里的 SUM(merchant_customer_distance) 的含义，就是对前面的条件筛选出的数据结果中的 merchant_customer_distance 列的值进行求和。根据骑手 id 和配送状态筛选出的记录分别为 id=(1, 2, 5)，对应的 merchant_customer_distance 的值分别为 merchant_customer_distance=(2.5, 1.8, 1.2)，求和结果为 5.5=2.5+1.8+1.2，如下表。

SUM(merchant_customer_distance)
5.5

[3] 场景：查询 Stark 这名骑手的平均配送里程

同样的，让我们先来定义一下平均配送里程：骑手所有完成单的配送距离（商家到用户的直线距离）之和除以总的完成单量。

基于 SUM 的经验和前面的“预告”，不难想到这次我们会用到 AVG 这个聚合函数。对字段求平均值的意思是，把结果集合中该字段的值累加起来再除以结果总行数。AVG 帮我们自动完成了“做除法”的动作，所以 Sql 的长相和上一个场景的 SUM 是如出一辙的。

```
SELECT AVG(merchant_customer_distance) FROM tb_order
WHERE rider_id = 1 AND order_state = 40;
```

根据骑手 id 和配送状态筛选出的记录分别为 id=(1, 2, 5)，对应的 merchant_customer_distance 的值分别为 merchant_customer_distance=(2.5, 1.8, 1.2)，求平均值的结果为 1.83=(2.5+1.8+1.2) / 3，如下表。

AVG(merchant_customer_distance)
1.83

写在 3.1 节的最后：

对着这几个场景学习下来，不知道你感觉怎么样吖？是否觉得这几个聚合函数本身还蛮简单的，或者也有可能会觉得一下子灌输了很多知识点有点费劲呢？其实聚合函数有它复杂的一面，我们上面看的这些 Case 都是比较简单的使用方式。但是千万不要担心，一方面是因为运营工作中遇到的绝大多数场景都不会比这些示例 Case 更复杂，另一方面是不鼓励过于复杂的使用这些聚合函数，因为查询的逻辑越是复杂就越是难以“预测”查询的结果，Sql 并不是一个适合表达“逻辑”的语言，如果对数据的再加工逻辑很多，就应该考虑像分析师提需求或者学习更加利于表达逻辑的其他编程语言。

其次要说的就是多给自己些信心，同时也要多一点耐心。Sql 虽然不同于 Python、Java 这样的通用编成语言，除了语法还杂糅着一套体系化的编程概念、设计哲学，但是初次上手的时候还是会感觉到有些吃力的。但是只要多去理解几遍示例、多自己写一写，特别是在之后遇到实际工作中真实场景的时候自己思考如何转化为 Sql、多实践、多回顾分析，很快就会在潜移默化中掌握它，要相信熟能生巧。

接下来的 3.2、3.3 节，我会继续介绍两个实用的 Sql 语法，以及如何将它们和聚合函数结合使用，会更有难度一些。

3.2 对查询结果去重：DISTINCT 语法

DISTINCT 语法顾名思义就是对某一列的值进行去重，让我们首先来回顾一下 3.1 节中 COUNT 的其中一个例子。

这个例子使用的是 tb_sample_1 这张表，这张表很简单，让我再把它贴出来。

id	name
1	Stark
2	Stark
3	Coulson
4	Natasha
5	Stark

对应的，我们想要回顾的这条 Sql 语句也很简单。

```
SELECT COUNT(name) FROM tb_sample_1;
```

前面我们已经分析过这条 Sql：对 name 这列进行计数，有 3 个 Stark，1 个 Coulson，1 个 Natasha，所以得到最终的结果如下表。

COUNT(name)
5

可是有的时候，我们不想对相同的名字进行重复计数，当有多个相同的名字时只计数一次。这时候就可以使用到 DISTINCT 语法。

```
SELECT COUNT(DISTINCT name) FROM tb_sample_1;
```

对比上一条 Sql 只是增加了一个 DISTINCT 关键字，其实理解起来呢也不用把它想的太复杂啦：COUNT(DISTINCT name) 就是对去重后的 name 进行计数。tb_sample_1 中有 3 个 Stark，但是 3 个 Stark 是重复的，使用 DISTINCT 语法后只会被计算一次，另外还有 1 个 Coulson 和一个 Natasha，所以得到的结果如下表。

COUNT(DISTINCT name)
3

DISTINCT 语法可以单独使用，这时就是它本身的意思，对某列的值进行去重。但是相比之下，更常见的是像上面的例子一样和 COUNT 这个聚合函数一起使用，这样就可以对去重后的结果进行计数。

3.3 将查询数据分组：GROUP BY 语法

前面我们基于 tb_order 这张表讲解了很多 Sql 的语法知识，让我们再来回忆一下这张表的容颜。

id	order_id	order_state	rider_id	rider_name	merchant_customer_distance	created_at
1	300000201712300001	40	1	Stark	2.5	2017-12-30 12:34:17
2	300000201712300002	40	1	Stark	1.8	2017-12-30 12:34:18
3	300000201712300003	40	2	Banner	1.8	2017-12-30 13:20:02
4	300000201712300004	40	5	Natasha	2.7	2017-12-30 13:34:19
5	300000201712300005	40	1	Stark	1.2	2017-12-30 16:01:03
6	300000201712300006	40	3	Rogers	0.5	2017-12-30 16:08:57
7	300000201712310001	20	6	Barton	1.3	2017-12-31 09:12:07
8	300000201712310002	80	7	Coulson	2.9	2017-12-31 09:10:33
9	300000201712310003	80	2	Banner	0.7	2017-12-31 09:18:10
10	300000201712310004	20	3	Rogers	2.2	2017-12-31 10:34:01
11	300000201712310005	10	0		0.3	2017-12-31 19:29:02
12	300000201712310006	10	0		1.3	2017-12-31 19:29:27
13	300000201712310007	10	0		3.0	2017-12-31 19:30:01

温故而知新！先来出几道题目复习一下前面所学的 Sql 知识。

复习题 1：试着写出以下几个场景对应的 Sql 语句

1. 查询 2017-12-30 当天创建的运单，状态为已完成且配送距离大于等于 2 公里的总单量；
2. 查询 2017-12-30 当天创建且状态为已完成的所有运单的平均配送距离；
3. 查询 2017-12-30 当天完成过配送任务（至少配送完成 1 单）的骑手总人数；

复习题 2：试着理解以下几条 Sql 的含义并且写出查询的结果

```
1. SELECT COUNT(order_id) FROM tb_order WHERE order_state = 40
   AND merchant_customer_distance >= 2.0 AND created_at >= "2017-12-30 00:00:00"
   AND created_at < "2017-12-31 00:00:00";
2. SELECT AVG(merchant_customer_distance) FROM tb_order WHERE order_state = 40
   AND created_at >= "2017-12-30 00:00:00" AND created_at < "2017-12-31 00:00:00";
3. SELECT COUNT(DISTINCT rider_id) FROM tb_order WHERE order_state = 40
   AND created_at >= "2017-12-30 00:00:00" AND created_at < "2017-12-31 00:00:00";
```

聪明的你是否发现复习题 2 就是复习题 1 的答案呢？如果还没有发现，没关系，再回过头来多分析几遍，Practice Makes Perfect 绝对是真理。不过复习这几个例子可不仅仅是为了复习哦，让我们在 1、2 两个场景的基础下扩展一下，讲解新的知识点。思考下面这两个场景。

1. 查询 2017-12-30 当天每个参与跑单的骑手各自的完成单总量；
2. 查询 2017-12-30 当天每个参与跑单骑手的完成单平均配送距离；

首先分析一下这里的场景 1。“2017-12-30 当天” 这个条件不难转化为 created_at >= '2017-12-30 00:00:00' AND created_at < '2017-12-31 00:00:00', “完成单” 不难转化为 order_state = 40, 由于要计算运单的 “总量” 我们也不难想到可以对 order_id 进行 COUNT 操作。那么如何分组到每个骑手身上呢？这时候就要用到 GROUP BY 了。

```
SELECT COUNT(order_id) FROM tb_order WHERE order_state = 40
AND created_at >= "2017-12-30 00:00:00" AND created_at < "2017-12-31 00:00:00"
GROUP BY rider_id;
```

注意这里执行顺序是先按照 WHERE 条件进行筛选, 然后根据骑手 id 进行分组 (GROUP BY), 最后再对每个分组按照运单号进行计数。因此我们可以得到下表的结果。

COUNT(order_id)
3
1
1
1

好像有哪里不对？结果中看不到对应的骑手吖！不着急, 我们稍微修改下刚才的 Sql, 将骑手 id、骑手姓名这 2 列展示出来就可以了。

```
SELECT rider_id, rider_name, COUNT(order_id)
FROM tb_order WHERE order_state = 40
AND created_at >= "2017-12-30 00:00:00"
AND created_at < "2017-12-31 00:00:00"
GROUP BY rider_id;
```

我们得到如下表的结果。

rider_id	rider_name	COUNT(order_id)
1	Stark	3
2	Banner	1
5	Natasha	1
3	Rogers	1

这样是不是就清晰多了。

再来分析场景 2。有了前面的例子, “2017-12-30 当天”、“完成单” 这两个条件应该是已经得心应手、信手拈来了, “平均配送距离” 问题也不大, 可以转化为 AVG(merchant_customer_distance)。那么如何分组到每个骑手身上呢？还是通过 GROUP BY 语法。我们的 Sql 长成下面这个样子。

```
SELECT rider_id, rider_name, AVG(merchant_customer_distance)
FROM tb_order WHERE order_state = 40
AND created_at >= "2017-12-30 00:00:00"
AND created_at < "2017-12-31 00:00:00"
GROUP BY rider_id;
```

得到如下表的结果。

rider_id	rider_name	AVG(merchant_customer_distance)
1	Stark	1.83
2	Banner	1.8
5	Natasha	2.7
3	Rogers	0.5

还是需要特别提一下这里的执行顺序, 首先执行的是 WHERE 条件筛选, 然后对筛选出的数据结果根据骑手 id 进行分组, 最后再对每个分组中的数据进行 merchant_customer_distance 列的求平均值。

3.4 聚合函数的好搭档：HAVING 语法

HAVING 语法的含义类似于 WHERE, 当我们使用 HAVING 的时候一般遵循 HAVING 筛选条件的语法结构。你可能会问啦, 既然和 WHERE 语法含义差不多、使用方式又很类似, 那干嘛还要凭空多个 HAVING 语法出来呢？原因就在于聚合函数。WHERE 语法是不能和聚合函数一起使用的, 但有些时候我们却需要依赖聚合函数的计算结果作为筛选条件。让我们看一下 3.3 节中场景 2 这个例子。

场景 2：查询 2017-12-30 当天每个参与跑单骑手的完成单平均配送距离。

通过前面我们的分析, 得到这样的 Sql。

```
SELECT rider_id, rider_name, AVG(merchant_customer_distance)
FROM tb_order WHERE order_state = 40
AND created_at >= "2017-12-30 00:00:00"
```

```
AND created_at < "2017-12-31 00:00:00"
GROUP BY rider_id;
```

我们在场景 2 的基础上再扩展一下。

扩展的场景 2：查询 2017-12-30 当天每个参与跑单骑手的完成单平均配送距离，并筛选出其中平均配送距离超过 1.5km 的数据。

我们得到这样的 Sql 结果。

```
SELECT rider_id, rider_name, AVG(merchant_customer_distance)
FROM tb_order WHERE order_state = 40
AND created_at >= "2017-12-30 00:00:00"
AND created_at < "2017-12-31 00:00:00"
GROUP BY rider_id
HAVING AVG(merchant_customer_distance) > 1.5;
```

比较一下不难发现，变化仅仅是末尾多了 HAVING AVG(merchant_customer_distance) > 1.5 这条子句。让我们分析看看。SELECT ... FROM ... WHERE ... 和之前的用法并没有变化，GROUP BY rider_id 将 SELECT 的结果根据 rider_id 进行分组，分组完成后 HAVING AVG(merchant_customer_distance) > 1.5 语句对每一组的 merchant_customer_distance 字段值求取平均数，并且将平均数大于 1.5 的结果筛选出来，作为返回结果。

执行这条 Sql 我们得到结果。

rider_id	rider_name	AVG(merchant_customer_distance)
1	Stark	1.83
2	Banner	1.8
5	Natasha	2.7

Rogers 这位骑手（骑手 id=3）因为平均配送距离为 0.5，不满足 HAVING 语句指定的“平均配送距离大于 1.5km”的筛选条件，所以没有在我们的查询结果中。

4 有点超纲的话题

4.1 字段类型

类型这个词此刻你听起来可能还是很陌生的，但其实在计算机科学领域，类型是一个非常基础而且广泛存在的概念，几乎每一种编程语言都有自己的类型系统。

> B.T.W 在二进制中每一个 0 或者 1 被称作一个比特位，所以 32 位是指一段二进制数据中 0 和 1 的个数加在一起共有 32 个，例如 00000000000000000000000000000001 表示一个 32 位二进制数，0000000000000001 表示一个 16 位二进制数。

[1] 为什么要定义类型的概念？

关于为什么要有类型这个概念，我呐有一个“不成熟”的理解：编程语言作为人和机器交互的一种工具，人类对数据有人类逻辑上的理解，当我们看到 2903 的时候我们会认为这是个整数，当我们看到 1031.2903 的时候我们会认为这是个小数。而机器在处理数据或者存取数据的时候，是无差别的按照比特位进行二进制运算或者读写的。人类很难做到直接用二进制输入计算机，当然也不能接受计算机直接以二进制的形式输出结果。设想一下，如果某天咱们想用一下电脑上的计算器，计算个 1+1=2，但是我们没有类型，我们需要理解机器是如何处理二进制的，那么就可能需要输入 00000000000000000000000000000001 + 00000000000000000000000000000001，而得到的结果也是二进制 00000000000000000000000000000010，这得多累人呐。有了类型就轻松多了，通过定义数据的类型，根据类型的约定，计算机就知道如何将这个 1 转化为二进制（包括：应该转化为 16 位、32 位还是 64 位的二进制，对这段二进制数据进行操作的时候，应该把它看作整数还是浮点数等等），而返回结果的时候也就知道如何将二进制的 00000000000000000000000000000010 转化为我们能够理解的整数 2

编程语言的类型其实就是人与机器约定好的，去理解和操作数据的一套规则。

总而言之，在机器的眼里，无论是对数据进行何种操作，它看到的都是一串一串由 0 和 1 构成的东西，称呼这种东西有专门的术语，叫作“字节流”或者“二进制流“。

让我们再一起看一个例子。假设要处理这样的一段二进制流：00000000100111011000001111010111，这段二进制流可以表示很多东西，要明确它的含义，就需要明确它的类型，比如下面这两种不同的类型，这段流表示的内容就完全不同。

1. 如果我们把这段二进制流看作是 32 位整型，那么它代表的是 10322903 这个整数；
2. 如果我们把这段二进制流看作是 2 个 16 位整型（前 16 位 0000000010011101 表示一个整型，后 16 位 1000001111010111 表示一个整型），那么它分别代表的是 157 和 33751 这两个整数；

我知道你此刻对为何转换为 32 位整型是 10322903？为何看作 2 个 16 位整型转换后是 157 和 33751？还有着很多疑惑。但是关于二进制和十进制的转换方法呢，在这里就不做展开了，如果你很感兴趣、很想知道可以再单独给你讲这个方法。讲上面的这些，最主要的还是希望你明白，定义“类型”的概念，根本上是在人机交互的过程中提供了一种机制，赋予无差别的二进制流一定的语义。

还是太抽象了对不对？没关系，我们再来举个栗子。

前面我们在预备知识这一章中使用到了 tb_stu_math_score 这张表，为了不让你辛苦的再翻回去，我们再贴一下这张表的内容啦。

id(自增主键)	name(学生姓名)	number(学号)	grade(年级)	class(班级)	score(得分)
1	柯南	010201	1	2	100
2	小哀	010202	1	2	100
3	光彦	010203	1	2	98
4	步美	010204	1	2	95
5	元太	010205	1	2	59

也写过类似下面这条 Sql 语句。

```
SELECT score FROM tb_stu_math_score WHERE id=1;
```

这条 Sql 语句非常非常的简单，现在我们已经知道它会返回第一行数据 score 这一列的值，结果长下面这样。

```
score
100
```

让我们分析一下获取这个结果的整个流程，帮助你理解一下，类型是如何发挥作用的。

1. 这条 Sql 语句会被执行，根据主键 id 找到对应的行，进而获取到这一行、score 这列的值；
2. 但是我们上面说到，计算机的存储是基于二进制的，所以获取到的 score 值是类似于 00000000000000000000000001100100 这样的二进制流；
3. 这时候根据 tb_stu_math_score 表的定义,score 这一列被定义为整型,于是将二进制流转化为一个整型数,经过进制转换得到 00000000000000000000000001100100 对应的整型值为 100，于是我们看到的结果就是 100。

实际上反过来也非常类似，当我们向这张表中写入数据时，例如写入的 score 列的值为 100。因为存储基于二进制，根据表的定义，score 列的类型为整型，于是将值 100 按照整型转换为对应的二进制流 00000000000000000000000001100100，并且写入到库中。

[2] Sql 的主要数据类型有哪些？

Sql 中常常接触的数据类型主要包括几类。

1 整型

1. tinyint：用来表示很小很小的整数，比如常常用它作为 is_deleted、is_valid 这些字段的字段类型，因为这两个字段表示该条记录是否有效，只存在两个值分别是 0 和 1；
2. smallint：比 tinyint 稍微大一点点的整型，可以表示更大一点的整数，比如 200、404、401 这样的整数值；
3. int：常用的整型，可以用来表示比较大的整数，比如 10322(事实上 int 可以表示的整数范围远远比这个大)；
4. bigint：用来表示非常大的整数，比如大多数表的自增 id 就会使用这个类型，可以表示类似 10322903 这样非常大的整数（事实上 bigint 可以表示的整数范围远远比这个要大)；

2 浮点型

1. decimal：可以表示非常准确的小数，比如经纬度；

3 字符串类型

1. char：固定长度的字符串；
2. varchar：可变长度的字符串；

这里固定长度和可变长度指的是数据库中的存储形式，因为这部分的内容其实有些超出了这个教程的范围，我们不过多的解释这里的区别。一般在我们实际的应用中 varchar 用的更多一些。它们都表示类似于"very glad to meet u，Huohuo!" 这样的一串字符，当然也可以是中文" 敲开心认识你，火火！"。

4 日期类型

1. date：表示一个日期，只包含日期部分，不包含时间，比如当前日期“2018-01-23”；
2. datetime：表示一个日期，同时包含日期部分和时间部分，比如当前日期“2018-01-23 03:01:43”；

我们在这里只是简单的介绍了几种 Sql 中常见的字段类型，并没有很深入的去解释它们的原理、差异以及一些其他的数据类型，咱们不着急去学习那些“高大上”的内容，先理解这些类型的含义。

[3] 怎么知道一张表中每一列的类型是什么？

第 1 种方式是使用 DESC 表名命令，例如我们想看一下之前提到的 tb_rider 表的每一列字段类型，就可以执行命令 DESC tb_rider，得到下面的结果。

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	<null>	auto_increment
name	varchar(32)	NO			
real_name_certify_state	int(11)	NO		0	
is_deleted	tinyint(4)	NO		0	
created_at	datetime	NO	MUL	CURRENT_TIMESTAMP	
updated_at	datetime	NO	MUL	CURRENT_TIMESTAMP	on update CURRENT_TIMESTAMP
level	tinyint(4)	NO		0	
level_city	varchar(32)	NO			

注意这里的第一列表示字段名称，第二列 Type 则表示对应字段的字段类型。比如 id 字段，是一个 int 类型。

第二种方式是使用 SHOW CREATE TABLE 表名命令，例如 SHOW CREATE TABLE tb_rider，得到下面的结果。

```
CREATE TABLE `tb_rider` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(32) NOT NULL DEFAULT '' COMMENT ' 姓名',
  `real_name_certify_state` int(11) NOT NULL DEFAULT '0' COMMENT ' 身份证认证状态',
  `is_deleted` tinyint(4) NOT NULL DEFAULT '0' COMMENT ' 该用户是否还存在. 0: 不存在, 1: 存在',
  `created_at` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT ' 创建时间',
  `updated_at` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT ' 更新时间',
  `level` tinyint(4) NOT NULL DEFAULT '0' COMMENT ' 骑手等级：0 普通 1 铜牌 2 银牌 3 金牌',
  `level_city` varchar(32) NOT NULL DEFAULT '' COMMENT ' 配送员等级城市',
  PRIMARY KEY (`id`),
  KEY `ix_created_at` (`created_at`),
  KEY `ix_updated_at` (`updated_at`)
) ENGINE=InnoDB AUTO_INCREMENT=9 DEFAULT CHARSET=utf8 COMMENT=' 配送员信息';
```

我们以

```
`name` varchar(32) NOT NULL DEFAULT '' COMMENT ' 姓名'
```

来解释一下这里的语句。

1. name 是字段名（列名）；
2. varchar 表示字段类型为字符串；
3. NOT NULL 表示这个字段不能为空，为空的意思是没有指定任何值给这个字段（注意不等价于空字符串）；
4. “DEFAULT '' 表示如果没有指定这个字段的值则使用空字符串作为默认值；
5. COMMENT ' 姓名' 是对这个字段的备注，表示这个字段的业务含义，只用做展示；

4.2 索引

索引绝对算得上是关系型数据库中最关键同时也是最有难度的话题。即便是经验丰富的研发同学，也经常会踩到索引的坑。不过我们这里介绍索引，只是为了更好的服务于查询，我会尽可能避免牵扯进一些复杂的概念和底层原理。

[1] 什么是索引？

那么到底什么是索引呢？你可以把数据库理解为一本很厚的书（假设有 10 万页），书中的内容就是数据库里的数据，那么索引就是书的目录。假设你从来没有阅读过这本书，此刻你想要阅读书的第 7 章第 2 小节。如果没有目录，你可能需要翻阅整本书找到你要阅读的内容。但是在有目录的情况下，你就只需要先查一下目录找到对应的页码，然后直接翻到那一页就能看到你想看的内容了。索引也是类似的，首先查询索引找到目标数据的位置，再从特定的位置读取出数据的内容。

如何设计索引，是设计数据库表的时候考虑的关键点之一。索引一般由表中的某一列或者某几列构成，一旦设置某一列为索引，那么之后每次在往表中写入数据的时候，都会更新这一列到索引中去。事实上，索引在技术层面是比较复杂的，涉及到磁盘 I/O、B 树、优化器（Optimizer）等很多技术概念，不过我们先不去深究这些。

[2] 为什么索引很重要，它有什么用？

索引之所以重要，最主要的原因是能够大大提高查询的速度。上面我们举了书的例子，当这本书的页数足够大的时候（假设有 2000 万页），如果没有目录，想要查阅其中的某一章节的内容，那几乎就是天方夜谭了。数据库也是如此，当表中的数据只有几行或者几十行、几百行的时候，有没有索引其实差别不大，但是当表中的数据非常非常多的时候（比如众包的运单表，2000 万 + 行），如果没有索引，要找到某一条目标数据，查询的速度就会非常非常非常的慢。

[3] 如何使用索引？

要使用索引非常简单，只需要在 WHERE 条件中使用到索引列作为查询条件，让我们举个例子。

id	order_id	order_state	rider_id	rider_name	merchant_customer_distance	created_at
1	300000201712300001	40	1	Stark	2.5	2017-12-30 12:34:17
2	300000201712300002	40	1	Stark	1.8	2017-12-30 12:34:18
3	300000201712300003	40	2	Banner	1.8	2017-12-30 13:20:02
4	300000201712300004	40	5	Natasha	2.7	2017-12-30 13:34:19
5	300000201712300005	40	1	Stark	1.2	2017-12-30 16:01:03
6	300000201712300006	40	3	Rogers	0.5	2017-12-30 16:08:57
7	300000201712310001	20	6	Barton	1.3	2017-12-31 09:12:07
8	300000201712310002	80	7	Coulson	2.9	2017-12-31 09:10:33
9	300000201712310003	80	2	Banner	0.7	2017-12-31 09:18:10
10	300000201712310004	20	3	Rogers	2.2	2017-12-31 10:34:01
11	300000201712310005	10	0		0.3	2017-12-31 19:29:02
12	300000201712310006	10	0		1.3	2017-12-31 19:29:27
13	300000201712310007	10	0		3.0	2017-12-31 19:30:01

还是这张 tb_order 表，假设这张数据表中 order_id 是索引列，那么当我们以 order_id 作为查询条件时，我们就利用了索引，比如下面这条 Sql。

```
SELECT * FROM tb_order WHERE order_id = 300000201712310007;
```

当然啦，类似的使用 order_id 作为查询条件的 Sql 也都会利用到索引，看看你是否都理解下面两条 Sql 语句的含义。

```
1. SELECT * FROM tb_order
   WHERE order_id IN (300000201712310007, 300000201712310006)
   AND order_state = 40;
```

```
2. SELECT order_id, order_state FROM tb_order
   WHERE order_id >= 300000201712300001
   AND order_id <= 300000201712300006
   AND order_state = 40;
```

那么如果一张表里面不止一列是索引，而在查询的 Sql 中这些索引列都作为了 WHERE 语句的查询条件，会使用哪个列作为索引还是都使用？假设 tb_order 表中 order_id 和 rider_id 两列都是索引列，那么下面这条 Sql 语句会使用哪个作为索引呢？

```
SELECT * FROM tb_order
WHERE order_id >= 300000201712310001
AND order_id <= 300000201712310007
AND rider_id > 0;
```

答案是不确定的。使用哪个索引，甚至是否使用索引，从根本上来说是由优化器（Optimizer）决定的，它会分析多个索引的优劣，以及使用索引和不使用索引的优劣，然后选择最优的方式执行查询。这部分话题就太过复杂了，这里不做展开。尽管有优化器（Optimizer）的存在，但是对于我们的查询来说，能够使用明确的索引字段作为查询条件的，就应该尽可能使用索引字段。

[4] 索引的类型、如何确定表中的哪些列是索引列？

还记得字段类型一节中提到的 DESC 表名和 SHOW CREATE TABLE 表名语法吗？前面我们将这两个语法用在了 tb_rider 表上，这一节让我们看一看 tb_order 表。

首先是 DESC tb_order，我们会得到下面的结果。

Field	Type	Null	Key	Default	Extra
id	bigint(20)	NO	PRI	<null>	auto_increment
order_id	bigint(20)	NO	UNI	0	
rider_id	int(11)	NO		0	
rider_name	varchar(100)	NO			
order_state	tinyint(4)	NO		0	
is_deleted	tinyint(4)	NO		0	
grabbed_time	timestamp	NO		CURRENT_TIMESTAMP	
merchant_customer_distance	decimal(10,2)	NO		0.00	
created_at	datetime	NO	MUL	CURRENT_TIMESTAMP	
updated_at	datetime	NO	MUL	CURRENT_TIMESTAMP	on update CURRENT_TIMESTAMP

之前我们关注的是 Type 这一项，这里让我们关注 Key 这一项。我们看到有些列对应的 Key 是空的，这就表示这一列（或者叫这个字段）不是索引列（或者叫索引字段）。但 id、order_id、created_at 和 updated_at 这几列对应的 Key 均是有值的，这说明这几列都是索引列。但这几列 Key 的值又各不相同，这是为啥呐？这是以内索引也分为不同的类型，让我们逐个来解释一下。

1. PRI：是 primary 的缩写，标记这一列为主键，主键的概念我们在一开始的时候有介绍过，就是用来唯一标识表中每一行数据的索引；
2. UNI：是 unique 的缩写，顾名思义就是唯一的意思，被设置为 UNI KEY 的列，不允许出现重复的值，如果尝试向表中插入这一列的值完全相同的两行数据，则会引发报错。我猜你肯定会觉得疑惑，那 UNI KEY 和 PRI KEY 有啥区别？首先是这两种类型的索引在实现上是有区别的（这一点咱们不深究，涉及到了数据库底层对索引的实现），其次 PRI KEY 更多的是数据库层面的语义，仅仅是描述数据的唯一性，而 UNI KEY 则更多是业务层面的语义，比如说这里的 order_id 字段，因为业务上不能存在两个运单号完全相同的运单，所以需要把 order_id 这一列设置为 UNI KEY；
3. MUL：是 multiple 的缩写，表示这一列是被设置为一个普通索引。之所以叫做 multiple，是因为此时可能这一列单独作为索引，也可能这一列和其他标记为 MUL 的列共同构成了一个索引（这种由多列共同构成的索引被叫作复合索引）；

现在我们还处在 Sql 以及数据库知识（是的，除了 Sql，我还偷偷介绍了一些数据库原理）学习的初级阶段，所以让我们知道这写差异，但是不着急去把这些搞得一清二楚，它们都是索引，只要合理使用，都可以帮助我们加快 Sql 查询的效率。

另一种识别表中索引列的方法就是通过 SHOW CREATE TABLE 表名命令，比如 SHOW CREATE TABLE tb_order，我们得到下面的结果。

```
CREATE TABLE `tb_order` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT ' 对外不提供，内部使用',
  `order_id` bigint(20) NOT NULL DEFAULT '0' COMMENT ' 运单的跟踪号（可以对外提供）',
  `rider_id` int(11) NOT NULL DEFAULT '0' COMMENT ' 配送员 id',
  `rider_name` varchar(100) NOT NULL DEFAULT '' COMMENT ' 配送员名字',
  `order_state` tinyint(4) NOT NULL DEFAULT '0' COMMENT ' 配送状态',
  `is_deleted` tinyint(4) NOT NULL DEFAULT '0',
  `grabbed_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT ' 抢单时间',
  `merchant_customer_distance` decimal(10,2) NOT NULL DEFAULT '0.00' COMMENT ' 商铺到顾客步行距离',
  `created_at` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `updated_at` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`),
  UNIQUE KEY `uk_order_id` (`order_id`),
  KEY `ix_created_at` (`created_at`),
  KEY `ix_updated_at` (`updated_at`)
) ENGINE=InnoDB AUTO_INCREMENT=14 DEFAULT CHARSET=utf8 COMMENT=' 配送单';
```

看到末尾几行的 PRIMARY KEY、UNIQUE KEY 和 KEY 了吗，它们就对应于 DESC tb_order 结果中的 PRI、UNI 和 MUL，分别标识主键索引、唯一索引和普通索引。每一行括号内的字段就表示对应的索引列。

4.3 JOIN 语法家族

我尝试了好几种解释清楚 JOIN 语法的方法（JOIN 语法的确有些复杂），始终不能让我自己满意，最终决定还是从一个例子开始。让我们首先看一张新的表，建表语句长下面这样。

```
CREATE TABLE `tb_grab_order_limit` (
  `id` BIGINT(20) NOT NULL AUTO_INCREMENT COMMENT ' 自增主键',
  `rider_id` BIGINT(20) NOT NULL DEFAULT 0  COMMENT ' 骑手 id',
  `order_grab_limit` INT(11) NOT NULL DEFAULT '0' COMMENT ' 接单上限',
  `is_deleted` TINYINT NOT NULL DEFAULT 0 COMMENT ' 该记录是否被删除',
  `created_at` DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT ' 创建时间',
  `updated_at` DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT ' 更新时间',
  PRIMARY KEY(`id`),
  KEY `ix_rider_id` (`rider_id`),
  KEY `ix_created_at` (`created_at`),
  KEY `ix_updated_at` (`updated_at`)
) ENGINE = InnoDB DEFAULT CHARSET=utf8 comment=" 自定义骑手接单上限表";
```

小温习

参考上面的建表语句尝试回答下面这几个问题。

1. 这张表的表名是什么？
2. order_grab_limit 这个字段的含义是什么？
3. 这张表的主键索引是什么？有几个唯一索引、几个普通索引？

没错！这就是自定义骑手接单上限表。描述了某一个骑手（rider_id）对应的他的接单上限（order_grab_limit）。表中的数据如下。

id	rider_id	order_grab_limit	is_deleted	created_at	updated_at
1	1	11	0	2018-02-25 17:22:03	2018-02-25 17:22:03
2	2	9	0	2018-02-25 17:22:21	2018-02-25 17:22:21
3	4	9	0	2018-02-25 17:22:31	2018-02-25 17:22:31
4	6	7	0	2018-02-25 17:22:39	2018-02-25 17:22:39
5	10	8	0	2018-02-25 17:22:46	2018-02-25 17:22:46

再让我们回顾一下前面反复用到的 tb_rider 表。

id	name	real_name_certify_state	level	level_city	is_deleted	created_at	updated_at
1	Stark	2	3	1	0	2017-01-01 22:00:19	2018-01-01 06:40:01
2	Banner	2	3	9	0	2017-04-28 12:01:19	2018-01-01 06:40:01
3	Rogers	2	2	1	0	2017-04-10 17:24:01	2018-01-01 06:40:01
4	Thor	1	0	1	0	2017-12-31 23:10:39	2018-01-01 06:40:01
5	Natasha	2	1	1	0	2017-02-11 15:03:13	2018-01-01 06:40:01
6	Barton	2	1	9	0	2017-02-11 15:04:19	2018-01-01 06:40:01
7	Coulson	2	3	9	0	2017-01-03 23:00:22	2018-01-01 06:40:01
8	Coulson	1	0	2	0	2017-01-05 10:10:23	2018-01-01 06:40:01

(终于铺垫完啦！)

[1] 从 LEFT JOIN 开始

以这两张表为基础，设想一个场景：假设要查询 tb_rider 表中所有骑手对应的自定义接单上限。我们的 Sql 应该怎么写呢？

思路 1：先查出 tb_rider 表中所有骑手 id，再根据这些骑手 id 作为查询条件，通过前面学习过的 IN 语法从 tb_grab_order_limit 表中查询出所对应的自定义接单上限的记录。

```
SELECT id FROM tb_rider;
```

和

```
SELECT rider_id, order_grab_limit FROM tb_grab_order_limit
WHERE rider_id IN (1, 2, 3, 4, 5, 6, 7, 8);
```

思路 1 显然是个 Bad idea。但是思路 1 诠释了解决这个查询问题的基本要点。

1. 我们最终想要的数据是需要结合 tb_rider 和 tb_grab_order_limit 两张表共同得出的；
2. 关联这两张数据表的条件是骑手 id；
3. 因为查询的要求是：tb_rider 表中所有骑手，因此应该以 tb_rider 表中的骑手 id 作为查询参考集合；
4. 不是所有 tb_rider 表中的骑手都配置了自定义接单上限，思路 1 的查询方案存在一个缺点，就是我们需要根据查询结果，在逻辑上做一个转换得知哪些骑手没有配置自定义接单上限（ 不在返回结果中的骑手）；

思路 2：基于这几个要点我们可以使用 LEFT JOIN 语法，下面是对应的 Sql 语句。

```
SELECT tb_rider.id, tb_grab_order_limit.order_grab_limit
FROM tb_rider LEFT JOIN tb_grab_order_limit
ON tb_rider.id = tb_grab_order_limit.rider_id;
```

这里先介绍一下 JOIN 语法的基本结构：表 1 (INNER/LEFT/RIGHT/FULL) JOIN 表 2 ON 表 1. 列 1 = 表 2. 列 2。JOIN 关键字前后连接的是两张需要关联查询的数据表，ON 关键字后面跟着关联的条件。一共有四种类型的 JOIN，他们分别是 INNER JOIN、LEFT JOIN、RIGHT JOIN 和 FULL JOIN。以例子中的 LEFT JOIN 为例，表 1 LEFT JOIN 表 2 ON 表 1. 列 1 = 表 2. 列 2 的含义是，遍历表 1 中的列 1 的值，如果表 2 中列 2 的值有和它相等的则展示对应的记录，如果没有表 2. 列 2 和表 1. 列 1 相等，则展示为 null。

思路 2 的例子中，tb_rider LEFT JOIN tb_grab_order_limit ON tb_rider.id = tb_grab_order_limit.rider_id 的含义是，遍历 tb_rider 表中 id 这一列 (tb_rider 表的 id 字段业务含义就是骑手 id) 的值，寻找 tb_grab_order_limit 表中 rider_id 列的值和它相等的记录，如果不存在则是 null。

我们还看到 SELECT 语句的内容和我们之前使用的很类似，但又稍微有点不一样，都是表名. 列名的书写形式。其实这主要是指明了字段所属的表，因为 JOIN 的两张数据表中可能存在的相同名称的列，例如 tb_rider 表和 tb_grab_order_limit 表都有 id 字段，但含义截然不同，这样写更加明确。

最终思路 2 的结果如下。

id	order_grab_limit
1	11
2	9
4	9
6	7
7	<null>
8	<null>
5	<null>
3	<null>

我们看到骑手 id=(7, 8, 5, 3) 的几个骑手没有配置自定义的接单上限，但因为是 LEFT JOIN，他们仍然会展示在查询结果中，不过因为没有接单上限的记录，order_grab_limit 的结果为 null。

让我们再回头看一下表名. 列名这个写法。如果思路 2 中的 Sql 改成下面这样，返回结果会变成什么呢？

```
SELECT tb_grab_order_limit.rider_id, tb_grab_order_limit.order_grab_limit
FROM tb_rider LEFT JOIN tb_grab_order_limit
ON tb_rider.id = tb_grab_order_limit.rider_id;
```

让我们来分析一下。我们知道 LEFT JOIN 的返回结果集合是以它左侧连接的数据表决定的，所以结果集仍然包含 8 条记录，但是骑手 id=(7, 8, 5, 3) 这个骑手没有对应的接单上限的配置，因此当我们展示这几个骑手的 tb_grab_order_limit.rider_id 列的值的时候，类似于 tb_grab_order_limit.order_grab_limit，也是 null。因此结果是下面这样。

rider_id	order_grab_limit
1	11
2	9
4	9
6	7
<null>	<null>
<null>	<null>
<null>	<null>
<null>	<null>

如果你还是不太明白，然我们在 SELECT 的时候，加上 tb_rider.id，或许有助于理解。

```
SELECT tb_rider.id, tb_grab_order_limit.rider_id, tb_grab_order_limit.order_grab_limit
FROM tb_rider LEFT JOIN tb_grab_order_limit
ON tb_rider.id = tb_grab_order_limit.rider_id;
```

结果是。

id	rider_id	order_grab_limit
1	1	11
2	2	9
4	4	9
6	6	7
7	<null>	<null>
8	<null>	<null>
5	<null>	<null>
3	<null>	<null>

[2] LEFT JOIN 的姊妹篇：RIGHT JOIN

前面我们知道 LEFT JOIN 是以连接的左侧表作为查询的结果集的依据，RIGHT JOIN 则是以连接的右侧表作为依据。让我们考虑另一个场景：假设想要查询所有设置了自定义接单上限的骑手姓名。应该如何写这个 Sql 呢？

先在聪明的大脑里思考几分钟。此时你需要类比 LEFT JOIN，需要理解上一段内容讲述的 LEFT JOIN 知识点，可能需要回到上一段再看一看示例 Sql 语句以及对应的结果。没关系，一开始学习的时候慢慢来。

答案是这样的。

```
SELECT tb_grab_order_limit.rider_id, tb_rider.name
FROM tb_rider RIGHT JOIN tb_grab_order_limit
ON tb_rider.id = tb_grab_order_limit.rider_id;
```

对应的查询结果则是。

rider_id	name
1	Stark
2	Banner
4	Thor
6	Barton
10	<null>

如果这个结果和你脑海中思考的结果不一样，不要着急，让我们再来解释一下。RIGHT JOIN 是以连接的右侧表为依据，而 tb_grab_order_limit 中的骑手 id=(1, 2, 4, 6, 10)，其中骑手 id 为 10 的骑手在 tb_rider 表中是没有的，所以 name 为 null。

小测验

尝试下将上面的这条 Sql 语句改写成 LEFT JOIN 吧（要求得到相同的查询结果）？

[3] 一丝不苟的 INNER JOIN

之所以叫“一丝不苟”的 INNER JOIN，是因为 INNER JOIN 是非常严格的关联查询，换句话说，必须是根据 JOIN 条件两张表中存在匹配记录的才作为结果集返回。让我们回顾下 [1] 中 LEFT JOIN 的 Sql。

```
SELECT tb_rider.id, tb_grab_order_limit.order_grab_limit
FROM tb_rider LEFT JOIN tb_grab_order_limit
ON tb_rider.id = tb_grab_order_limit.rider_id;
```

它的返回结果是。

id	order_grab_limit
1	11
2	9
4	9
6	7
7	<null>
8	<null>
5	<null>
3	<null>

如果我们将 LEFT JOIN 改为 INNER JOIN 呐？修改后的 Sql 像这样。

```
SELECT tb_rider.id, tb_grab_order_limit.order_grab_limit
FROM tb_rider INNER JOIN tb_grab_order_limit
ON tb_rider.id = tb_grab_order_limit.rider_id;
```

这时返回的查询结果变成了。

id	order_grab_limit
1	11
2	9
4	9
6	7

这是因为 INNER JOIN 会遍历连接一侧的表，根据 ON 后的连接条件，和连接另一侧的表进行比较，只有两张表中存在匹配的记录才会作为结果集返回。例如这里，它会遍历 tb_rider 表中 id 字段的值，并且去 tb_grab_order_limit 表中寻找 rider_id 与之匹配的记录，如果找到则作为结果返回。

> B.T.W INNER JOIN 和 JOIN 是等价的，换句话说，表 1 INNER JOIN 表 2 ON... 和表 1 JOIN 表 2 ON... 是完全等价的。

小测验

猜测一下下面的这条 Sql 语句的返回结果是什么？

```
SELECT tb_rider.id, tb_grab_order_limit.order_grab_limit
FROM tb_grab_order_limit INNER JOIN tb_rider
ON tb_grab_order_limit.rider_id = tb_rider.id;
```

提示：这里交换了一下 INNER JOIN 连接的两张表的位置，根据 INNER JOIN 的特性，查询结果会有影响嘛？

[4] 心大的 FULL JOIN

FULL JOIN 其实并不在乎匹配与否，而是将连接的两张表中所有的行都返回，如果有匹配的则返回匹配的结果，如果没有匹配则哪张表中缺失则对应的将当前这条记录标记为 null。看一个例子就明白啦！

```
SELECT tb_rider.id, tb_rider.name, tb_grab_order_limit.rider_id, tb_grab_order_limit.order_grab_limit
FROM tb_rider FULL JOIN tb_grab_order_limit ON tb_rider.id = tb_grab_order_limit.rider_id;
```

这条 Sql 语句的查询结果是这样的。

id	name	rider_id	order_grab_limit
1	Stark	1	11
2	Banner	2	9
4	Thor	4	9
6	Barton	6	7
3	Rogers	<null>	<null>
5	Natasha	<null>	<null>
7	Coulson	<null>	<null>
8	Coulson	<null>	<null>
<null>	<null>	10	10

可以看到 tb_rider 表中骑手 id=(3, 5, 7, 8) 的骑手在 tb_grab_order_limit 表中没有匹配的记录，而 tb_grab_order_limit 表中骑手 id=(10) 的骑手在 tb_rider 表中没有匹配记录，但是它们都作为结果集返回了。只不过缺失 tb_grab_order_limit 记录的，rider_id 和 order_grab_limit 字段值为 null，而缺失 tb_rider 记录的，id 和 name 字段的值为 null。

事实上，绝大多数情况下，FULL JOIN 都不会被用到。而且在一些数据库管理系统中，例如 MySql(我们的线上环境主要使用的就是 MySql)，是不支持 FULL JOIN 语法的。对于上面的查询语句，需要使用一些技巧通过 LEFT JOIN、RIGHT JOIN 以及 UNION(这篇教程中我们不讨论 UNION 语法哦) 语法的组合来实现同样效果的查询。

```
SELECT tb_rider.id, tb_rider.name, tb_grab_order_limit.rider_id, tb_grab_order_limit.order_grab_limit
FROM tb_rider LEFT JOIN tb_grab_order_limit ON tb_rider.id = tb_grab_order_limit.rider_id
UNION
SELECT tb_rider.id, tb_rider.name, tb_grab_order_limit.rider_id, tb_grab_order_limit.rider_id
FROM tb_rider RIGHT JOIN tb_grab_order_limit ON tb_rider.id = tb_grab_order_limit.rider_id
WHERE tb_rider.id IS null;
```

这已经超出了这篇教程的讨论范围啦！如果想要挑战一下自己，以下是一些提示。

1. UNION 连接两条 SELECT 语句，作用是将两个 SELECT 语句的查询结果取交集；
2. 第 2 条 SELECT 语句中的 WHERE tb_rider.id IS null 是为了对存在匹配的数据记录去重（否则 UNION 之后会有重复的结果）；
3. WHERE 语句是在 RIGHT JOIN 之后，UNION 之前执行的；

试着在这两条提示下理解一下这条 Sql 语句，如果能够弄明白这条语句是如何等价于 FULL JOIN 的，那么说明你对 JOIN 家族的语法已经基本掌握啦。如果暂时还不能弄得非常明白也没关系，多看一看例子，多写一写实践一下，慢慢就会明白啦。

题外话

从上面的讲解我们了解到 JOIN 的四种用法，总结一下。

1. INNER JOIN 关键字在两张表中都有匹配的值的时候返回匹配的行；
2. LEFT JOIN 关键字从左表返回所有的行，即使在右表中没有匹配的行；
3. RIGHT JOIN 关键字从右表返回所有的行，即使在左表中没有匹配的行；
4. FULL JOIN 关键字从左表和右表那里返回所有行，即使右表的行在左表中没有匹配或者左表的行在右表中没有匹配，这些行也会返回；

不过这些都是刻板的文字总结，让我们换个视角总结一下这集中 JOIN 语法。

离散数学中在讨论集合论的时候介绍过“韦恩图”的概念，它清楚的描述了数据集合之间的关系。而 JOIN 的这 4 种操作也正好对应了 4 种集合运算，下面的这张图（Figure 1）很清楚的描述了这种关系。

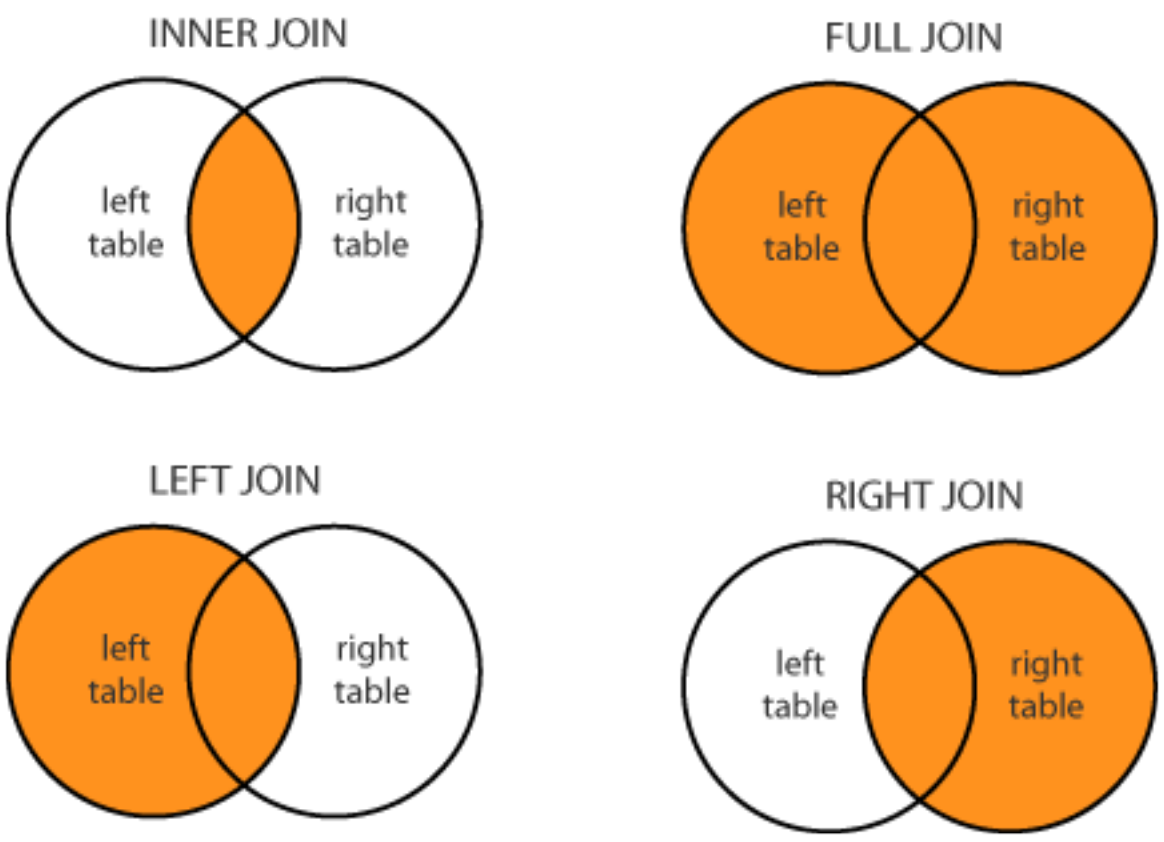

Figure 1: Mathematical Principle of Sql Join Syntax

4.4 嵌套的 SELECT 语法

再来看一下讲述 LEFT JOIN 的开始，我们提到的那个例子：查询 tb_rider 表中所有骑手对应的自定义接单上限。当时我们首先提出了思路 1，是分为 2 个步骤的。

```
SELECT id FROM tb_rider;
```

和

```
SELECT rider_id, order_grab_limit FROM tb_grab_order_limit
WHERE rider_id IN (1, 2, 3, 4, 5, 6, 7, 8);
```

我们说这个思路不好，这是显然的，因为在现实场景中往往数据集合都很大（例如这里的 rider_id 在现实中可能是成百上千甚至成千上万个），思路本身没有问题但无法操作执行。所以在 4.3 节我们选择通过 JOIN 语法来实现同样的查询。那是不是思路 1 就真的只能是个纸上谈兵的思路了呢？当然不是啦！我们还可以使用嵌套的 SELECT 语句，就像这样。

```
SELECT rider_id, order_grab_limit FROM tb_grab_order_limit
WHERE rider_id IN (SELECT id FROM tb_rider);
```

这个写法非常好理解，WHERE rider_id IN (SELECT id FROM tb_rider) 首先执行括号中的语句 SELECT id FROM tb_rider，然后执行 IN 筛选，就是我们的思路 1 描述的那样。于是得到下面的结果。

rider_id	order_grab_limit
1	11
2	9
4	9
6	7

复习题

回想一下上面的结果和以下哪条 Sql 语句的执行结果是一致的呢？为什么是一致的，为什么和其他的不一致？

```
1. SELECT tb_rider.id, tb_grab_order_limit.order_grab_limit
   FROM tb_rider LEFT JOIN tb_grab_order_limit
   ON tb_rider.id = tb_grab_order_limit.rider_id;
2. SELECT tb_grab_order_limit.rider_id, tb_rider.name
   FROM tb_rider RIGHT JOIN tb_grab_order_limit
   ON tb_rider.id = tb_grab_order_limit.rider_id;
3. SELECT tb_rider.id, tb_grab_order_limit.order_grab_limit
   FROM tb_rider INNER JOIN tb_grab_order_limit
   ON tb_rider.id = tb_grab_order_limit.rider_id;
4. SELECT tb_rider.id, tb_grab_order_limit.order_grab_limit
   FROM tb_rider FULL JOIN tb_grab_order_limit
   ON tb_rider.id = tb_grab_order_limit.rider_id;
```

小测验

思考一下以下这个场景，看看能否写出它对应的 Sql 语句？

场景：筛选出所有通过实名认证（real_name_certify_state=2）的金牌（level=3）骑手（tb_rider 表），在 2017-12-30 当天（created_at >= xxx AND created_at < yyy）所跑运单（tb_order 表）的运单号（order_id）。

想一想有几种写法呢？

5 闯关答题：快速复习

前面的几个段落我们学习了 Sql 查询中最常用，而且特别好用的语法知识，让我们简单总结一下。

1. 数据库、数据表的概念；
2. 最基本的 Sql 查询结构；
3. IN 查询和逻辑操作语法（AND/OR）；
4. 对查询结果进行排序和 LIMIT 语法；
5. 聚合函数（COUNT/AVG/SUM）和 DISTINCT 语法；
6. 对查询结果分组（GROUP BY）；
7. 对聚合函数的结果进行筛选的 HAVING 语法；
8. 字段类型和索引的概念和作用；
9. JOIN 语法的一家子（LEFT JOIN/RIGHT JOIN/INNER JOIN/FULL JOIN）；
10. 嵌套的 SELECT 语法；

学习了这么多知识点，实在是太腻害了！给自己点赞！

但是（凡事都有个但是）…

想要把这些知识点融会贯通，灵活应用到现实工作中更多变、更复杂的查询场景，仅仅是“学会”是不够的，还需要更多的“练习”和“回味”。

这个部分我设计了一个“闯关答题”项目，通过思考和回答这些闯关题，帮助你更好的掌握上面提到的知识点。

先来看一下答题将要用到的数据表。

[1] 商品数据表：tb_product

id	product_id	name	price
1	1001	iPad Pro 10.5 64G WLAN	4888
2	1002	Macbook Pro 2017 13.3 i5/8G/256GB	13888
3	1003	iPhone X 64G	8388

建表语句：

```
CREATE TABLE `tb_product` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT ' 自增主键',
  `product_id` bigint(20) NOT NULL DEFAULT '0' COMMENT ' 商品 id',
  `name` varchar(100) NOT NULL DEFAULT '' COMMENT ' 商品名称',
  `price` int(11) NOT NULL DEFAULT '0' COMMENT ' 商品价格',
  PRIMARY KEY (`id`),
  UNIQUE KEY `uk_product_id` (`product_id`)
) ENGINE=InnoDB AUTO_INCREMENT=14 DEFAULT CHARSET=utf8 COMMENT=' 商品信息表';
```

字段含义：

1. id：自增主键；
2. product_id：商品 id；
3. name：商品名称；
4. price：商品单价，单位是元；

[2] 用户数据表：tb_customer

id	customer_id	name	gender	balance
1	NO100001	火火	女	18888
2	NO100002	拨泼抹	女	9000
3	NO100003	艾桥	男	7990
4	NO100004	水娃	女	8388

建表语句：

```
CREATE TABLE `tb_customer` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT ' 自增主键',
  `customer_id` varchar(100) NOT NULL DEFAULT '' COMMENT ' 用户 id',
  `name` varchar(100) NOT NULL DEFAULT '' COMMENT ' 用户姓名',
  `gender` varchar(30) NOT NULL DEFAULT '' COMMENT ' 用户性别',
  `balance` int(11) NOT NULL DEFAULT '0' COMMENT ' 账户余额',
  PRIMARY KEY (`id`),
  UNIQUE KEY `uk_customer_id` (`customer_id`)
) ENGINE=InnoDB AUTO_INCREMENT=14 DEFAULT CHARSET=utf8 COMMENT=' 用户信息表';
```

字段含义：

1. id：自增主键；
2. customer_id：用户 id；
3. name：用户姓名；
4. gender：用户的性别；
5. balance：用户当前的可用账户余额，单位是元；

[3] 订单数据表：tb_order

id	order_id	customer_id	product_id	quantity
1	NUM1000301	NO100001	1001	1
2	NUM1000302	NO100001	1002	2
3	NUM1000303	NO100002	1002	2
4	NUM1000304	NO100003	1002	1
5	NUM1000305	NO100001	1003	1

建表语句：

```
CREATE TABLE `tb_order` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT ' 自增主键',
  `order_id` varchar(100) NOT NULL DEFAULT '' COMMENT ' 订单 id',
  `customer_id` varchar(100) NOT NULL DEFAULT '0' COMMENT ' 用户 id',
  `product_id` bigint(20) NOT NULL DEFAULT '0' COMMENT ' 商品 id',
  `quantity` int(11) NOT NULL DEFAULT '0' COMMENT ' 商品价格',
  PRIMARY KEY (`id`),
  UNIQUE KEY `uk_order_id` (`order_id`)
) ENGINE=InnoDB AUTO_INCREMENT=14 DEFAULT CHARSET=utf8 COMMENT=' 订单数据表';
```

字段含义：

1. id：自增主键；
2. order_id：订单号；
3. customer_id：下单用户 id；
4. product_id：购买的商品 id；
5. quantity：购买的数量；

了解完需要用到表结构，我们就要开始答题啦！

第一关：查询账户余额大于 1 万元的用户 id 和姓名？

Answer:

```
SELECT customer_id, name FROM tb_customer WHERE balance > 10000;
```

customer_id	name
NO100001	火火

第二关：查询账户余额小于 1 万元且性别为女生的用户姓名？

Answer:

```
SELECT name FROM tb_customer WHERE balance < 10000 AND gender=" 女";
```

name
拨泼抹
水娃

第三关：查询用户 id 为 NO100001 和 NO100002 的用户，所有购买记录的订单号？

Hint：IN

Answer:

```
SELECT order_id FROM tb_order WHERE customer_id IN ("NO100001", "NO100002");
```

order_id
NUM1000301
NUM1000302
NUM1000303
NUM1000305

第四关：查询用户 id 为 NO100001、NO100002 两位用户所有的购买记录（所有字段），要求按照优先以商品 id 递增、其次以订单号递减的规则展示数据？

Hint：IN、ORDER BY

Answer:

```
SELECT * FROM tb_order WHERE customer_id IN ("NO100001", "NO100002")
ORDER BY product_id ASC, order_id DESC;
```

id	order_id	customer_id	product_id	quantity
1	NUM1000301	NO100001	1001	1
3	NUM1000303	NO100002	1002	2
2	NUM1000302	NO100001	1002	2
5	NUM1000305	NO100001	1003	1

第五关：查询性别为女生的用户总数？

Hint：COUNT

Answer:

```
SELECT COUNT(customer_id) FROM tb_customer WHERE gender=" 女";
```

COUNT(customer_id)
3

第六关：查询 NO100001、NO100002、NO100003 三位用户各自购买商品的总数（不区分商品类型），输出购买商品件数大于等于 2 件的用户 id 以及他们对应购买的商品总数？

Warning：“购买商品的总数”和上一关“女生用户的总数”，这两个“总数”一样吗？

Hint：IN、SUM、HAVING

Answer:

```
SELECT customer_id, SUM(quantity) FROM tb_order
WHERE customer_id IN ("NO100001", "NO100002", "NO100003")
GROUP BY customer id
HAVING SUM(quantity) >= 2;
```

customer_id	SUM(quantity)
NO100001	4
NO100002	2

第七关：查询 NO100001、NO100002、NO100003 三位用户各自购买商品的总数（不区分商品类型），输出购买总数前两名的用户 id 以及他们对应购买的商品总数？

Hint：IN、SUM、ORDER BY、LIMIT

Answer:

```
SELECT customer_id, SUM(quantity) FROM tb_order
WHERE customer_id IN ("NO100001", "NO100002", "NO100003")
GROUP BY customer_id
ORDER BY SUM(quantity) DESC
LIMIT 2;
```

customer_id	SUM(quantity)
NO100001	4
NO100002	2

第八关：查询所有用户各自购买商品的总数（不区分商品类型），输出购买商品件数大于等于 2 件的用户 id 以及他们对应购买的商品总数？要求给出至少两种写法。

Warning：注意是“所有用户”，不是所有的用户都购买了商品

Hint：关联查询有哪些方法？

Answer:

写法一：嵌套的 SELECT

```
SELECT customer_id, SUM(quantity) FROM tb_order
WHERE customer_id IN (SELECT customer_id FROM tb_customer)
GROUP BY customer_id
HAVING SUM(quantity) >= 2;
```

customer_id	SUM(quantity)
NO100001	4
NO100002	2

写法二：使用 LEFT JOIN 语法

```
SELECT tb_customer.customer_id, SUM(tb_order.quantity) FROM tb_customer
LEFT JOIN tb_order ON tb_customer.customer_id = tb_order.customer_id
GROUP BY tb_customer.customer_id
HAVING SUM(tb_order.quantity) >= 2;
```

customer_id	SUM(tb_order.quantity)
NO100001	4
NO100002	2

第九关：查询所有用户各自购买商品的总数（不区分商品类型），输出购买总数前两名的用户 id 以及他们对应购买的商品总数？要求给出至少两种写法。

Hint：关联查询有哪些方法？

Answer:

写法一：嵌套的 SELECT

```
SELECT customer_id, SUM(quantity) FROM tb_order
WHERE customer_id IN (SELECT customer_id FROM tb_customer)
GROUP BY customer_id
ORDER BY SUM(quantity) DESC
LIMIT 2;
```

customer_id	SUM(quantity)
NO100001	4
NO100002	2

写法二：使用 LEFT JOIN 语法

```
SELECT tb_customer.customer_id, SUM(tb_order.quantity) FROM tb_customer
LEFT JOIN tb_order ON tb_customer.customer_id = tb_order.customer_id
GROUP BY tb_customer.customer_id
ORDER BY SUM(tb_order.quantity) DESC
LIMIT 2;
```

customer_id	SUM(tb_order.quantity)
NO100001	4
NO100002	2

第十关：以下哪几条 Sql 语句使用到了索引？分别是哪些字段上的索引？是什么类型的索引？

```
1. SELECT name FROM tb_customer WHERE customer_id = 1001;
2. SELECT product_id, name FROM tb_product WHERE price > 5000;
3. SELECT order_id, customer_id, product_id FROM tb_order
   WHERE order_id = "NUM1000302" AND customer_id = "NO100001"
   AND product_id = "1002";
4. SELECT order_id FROM tb_order WHERE id > 2;
```

Hint：索引

Answer:

sql 序号	是否使用到索引	索引所在字段	索引类型
1	是	customer_id	UNIQUE KEY
2	否	-	-
3	是	order_id	UNIQUE KEY
4	是	id	PRIMARY KEY