

当你能走得更远看得更多的时候，才会遇到真正的同路人，或许才能一生为伴和终身为友。而且越是时间长久的陪伴，就越是要保持距离，一条路也不非要事事都问，天天都讲，有人偶尔迷途也不必着急去找，谁都得独自成长，这才是爱。别小看了生活里的这些距离，我们有多少情感最终都是败在距离上。利益与伤害，烦恼与怨言，吵闹与挣扎，无非都在这个距离上折腾，远了生出不满，近里又生出矛盾。都说距离产生美，其实，那就是彼此尊重能够随时退一步的海阔天空。人性有贪欲，常常控制不住喜欢得寸进尺，太进了更容易变成占便宜，管闲事，好暧昧，多猜忌，如果不用距离来加以节制，那接下来就顺理成章的又卖乖了。于是，亲人心寒，爱人反目，朋友疏远，同事与陌生人之间就更不要说了。适当的距离是我们表达爱的最佳方式，爱不是枷锁更不是手段，人与人之间要用爱来沟通，但别用爱来说事较真。而距离，给我们提供了这样一个空间，让里面既有自己又有别人，合作可以愉快，相处很是轻松，帮助也常会帮到点子上，不是八卦和瞎起哄。没有距离的相处才是一种自私的表现，因为是你只想着自己才会忽略别人的感受。如果你说那是爱，自私的爱又能走多远？当我们最终在爱里失去了别人，又因此失去了自己，在痛苦里最终失去了善良，又因此失去了世界的时候，你就会明白，距离原来是爱的翅膀，而用距离来节制爱，才是最恰当的爱护与情谊。不必靠得太近，还要有各自的生活，也不会离得太远，只有一个转身的距离。最好的关系，是亲近地保持关距离，包括父母，子女，夫妻，朋友，同事，等等。让我们保持自己的意愿和生活方式，同时尊重他人的这一切，终究还是要留点空间才能各得其所，过上自己想要的生活

**SQL**

Summary

● Database Basics

● SQL Basic

○ Select/from/where/like/group by/order by/having

● SQL Intermediate

○ Join/distinct/union/coalesce/temp table/with

● SQL Advanced

○ Self join/case when/if/Over partition by/date function

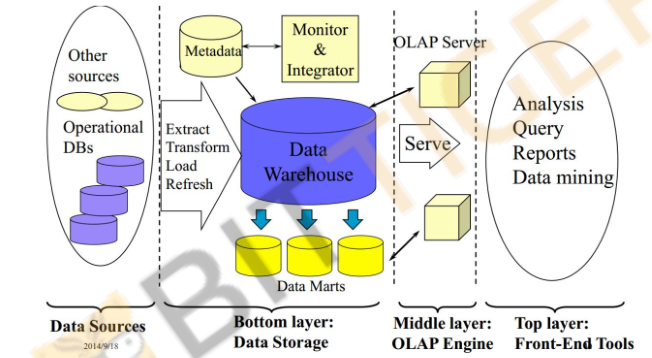
● SQL Interview

○ Type of questions, how to approach

● Coding Practice

**Database Basics**

Data system high-level view



从左到右：

**Data Source**: operational data, 例如客户和sales的互动，什么时候开始，什么时候有个contract——所有这些data都要经过处理——ETL，进入**data warehouse**. 数据仓库可能是很多的database，一般什么是一个database？——MySQL可能是一个database，所有的data都存在MySQL这个系统里面，打开MySQL的客户端就可以去调用这些data。还有别的database，比如一些文档，图片，有文档的db，图片有图片的db，可以理解为Data Marts就是一个一个的DB，data warehouse就是把它们整合到一起。Data warehouse不只是把data放在这里就可以了，还有不同的data之间怎么联系，如何从一个database流到另一个database也是在data warehouse中进行的。比如在facebook中数据的流动：每十分钟产生的数据会被放到一个叫scooba的database中，这个db的好处是特别快，只有几分钟的延迟data就能进去了。坏处是存储量提别小，没几个小时就得把数据导出，不能长期存储。Hive这个db比较慢，有很长的延迟，但是存储量很大，读取相对也比较方便。每隔几个小时，scooba中的数据就会经过一个pipeline被导到Hive中，这些都是在data warehouse中进行的。

当data warehouse中存储的数据很多了以后，往前进行有一个**分析使用**的层面，比如拿SQL去**query**这个data，或者用data直接生成一个**report** dashboard，这些都是前端的**front end tools**。在data和使用层之间有一个**OLAP Engine**，相当于一个**connection/server**，把数据联系起来。在公司中一般由data engineer架好的。SQL和所有data相关的使用都是在前端。

后端、底层部分的ETL和OLAP-建server都是由data engineer来做的。

ETL(Extract Transform Load-数据的变形，整理和清洁): mostly owned by data engineer

**前端：DS/BA**: 理解你的warehouse中有什么data set，understand how to use data warehouse, own front end tools

JD中关于DS BA要求的对data warehouse有经验应该是指对**每个data base**知道怎么使用以及知道schima——最重要的。后端的部分太底层，要由engineer的工作完成，不一定要求BA做。

DB有：MySQL,Hive,Mango DB, Hadoop Hive——可能会expect你知道这些。

**SQL Introduction**

Standard language for querying and manipulating data

**S**tructured **Q**uery **L**anguage

是一种语言而不特指某个软件，很多的DB都可以用sql来query。比如：

PostgreSQL, SQLite, **MySQL**, MSSQL, **Oracle ——这些都是r**elational **d**ata**b**ase **m**anagement **s**ystem**s** (RDBMSes)\*

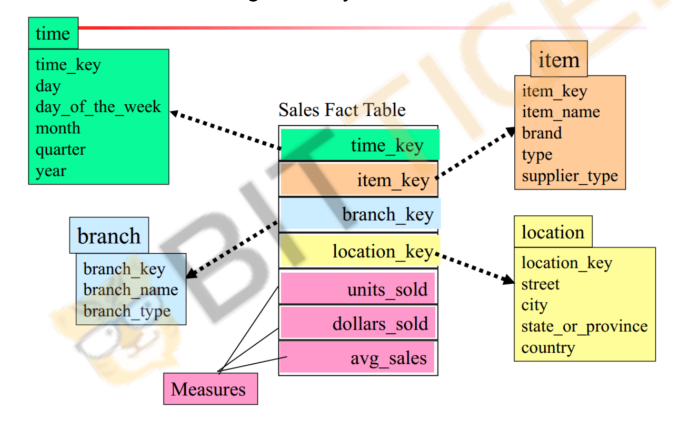
SQL, TSQL = the actual languages used by the aforementioned上述的 database systems. Although (in theory) SQL is standardised,有basic版本，每个db共用，但也有一些function，更advanced的extention各个db中不一样。 in practice each vendor has their own "extensions" (variations) on the language - which means that SQL written for MySQL may not work under Sybase, for example.

**RDBMS**

Relational database management system **关系型**数据库管理系统

Schima的一个例子

Sales的一个table：

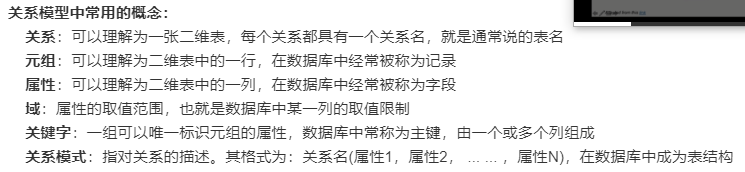


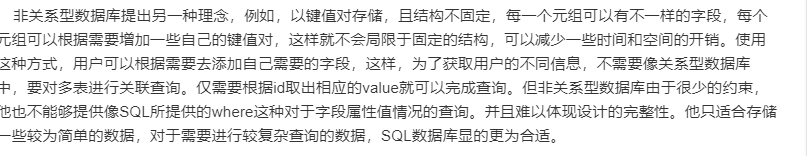
中间的每个column又可以连接到其他table，比如关于时间的细节的table，比如关于item的具体信息的table。

面试可能让你design一个schema。把需要重复利用的信息单独放在一个table中，要用的时候可以用一个key来join两个table。如果把item table的信息也都放在sales table中，那别的table要用到item信息时还得到sales table中来查，太不方便。

Primary key:每一行的unique ID,比如student id。在sales fact table中前四个column的combination是它的primary key。

Foreign key：别的table的primary key，可以通过它，把这两个table join起来，但不是本table的primmary key。





关系型数据库的最大特点就是事务的一致性：传统的关系型数据库读写操作都是事务的，具有ACID的特点，这个特性使得关系型数据库可以用于几乎所有对一致性有要求的系统中，如典型的银行系统。

    但是，在网页应用中，尤其是SNS应用中，一致性却不是显得那么重要，用户A看到的内容和用户B看到同一用户C内容更新不一致是可以容忍的，或者说，两个人看到同一好友的数据更新的时间差那么几秒是可以容忍的，因此，关系型数据库的最大特点在这里已经无用武之地，起码不是那么重要了。

---------------------

关系数据库的另一个特点就是其具有固定的表结构，因此，其扩展性极差，而在SNS中，系统的升级，功能的增加，往往意味着数据结构巨大变动，这一点关系型数据库也难以应付，需要新的结构化数据存储。

    于是，非关系型数据库应运而生，由于不可能用一种数据结构化存储应付所有的新的需求，因此，非关系型数据库严格上不是一种数据库，应该是一种数据结构化存储方法的集合。

    必须强调的是，数据的持久存储，尤其是海量数据的持久存储，还是需要一种关系数据库这员老将。

---------------------

**E-R diagram**

1976 proposed by Peter Chen

● ER diagram is widely used in **database design（即schema）**

● **Represent conceptual level** of a database system

● **Describe** things and their relationships **in high level**

Basic concepts

● **Entity set** – **an abstraction of similar things**, e.g. cars, students: An entity set contains many entities

● **Attributes**: **common properties** of the entities in a entity sets

● Relationship – specify the **relations among entities from two or more entity sets**



**SQL vs. NoSQL**

NoSQL = Not Only SQL

**SQL** databases are **table based databases-每一行有固定数目的columns**

**NoSQL** databases are **document based, key-value pairs, graph** databases or wide-column stores.列的数目不固定，每个元组可以根据自己的需要增减键值对。

- Hadoop/Hbase (Wide-column)

- MongoDB (Document Based)

- Neo4J (Graph)

Graph database存的是点&线，线用两端的点的坐标表示。是这么存储的。如果你去visualize这个database，可以看到不同的点连起来。能看到数据点间关系的visualization。比如connection，人和人之间的关系。

NoSQL Example (Json, document based)

{

"id":"ew12-res2-234e-544f",

"title":"post title",

"date":"2016-01-01",

"body":"this is an awesome post stored on NoSQL",

"createdBy":User,

"images":["http://myfirstimage.png","http://mysecondimage.png"],

"videos":[

{"url":"http://myfirstvideo.mp4", "title":"The first video"},

{"url":"http://mysecondvideo.mp4", "title":"The second video"}

],

"audios":[

{"url":"http://myfirstaudio.mp3", "title":"The first audio"},

{"url":"http://mysecondaudio.mp3", "title":"The second audio"}

]

}

**SQL Data Types**

Atomic types

● Characters文字: CHAR(20), VARCHAR(50)

不同database中的名字不太一样，在SQL中VARCHAR,括号中加个数字表示

它的长度。Hive中是另外的词。好像是CHAR( )?

● Numbers: INT, BIGINT, SMALLINT, FLOAT 也是每个db略有不同

● Others: MONEY, DATETIME, …

不会复合考察知识点

● 要不只写code

● 要不只问知识点

大部分时候要写code

常考面试题

1. What is the difference between primary key and foreign key?

Ans：primary key is this table’s unique identifier for each tuple/row, foreign is not the primary key for this table, it is another table’s primary key, but we can use it to join this two table.

2. How would you define the schema of table X?

Ans：根据面试的具体情况而定。先定义primary key, e.g. student ID. Then adds some attributes, and maybe we should create other tables to store some information about the major and we could use the major key to join the student info table with the major table.

\*\*\*\*学习网站\*\*\*\*

W3School

<https://www.w3schools.com/sql/>

ModeAnalytics

<https://community.modeanalytics.com/sql/tutorial/introduction-to-sql/>

**SQL Basic Syntax**

**SQL Syntax**

SELECT column-names

FROM table-name

WHERE condition (OPTIONAL)

GROUP BY group-name (OPTIONAL)

HAVING condition after aggregated (OPTIONAL)

ORDER BY sort-order [ASC/DESC] (OPTIONAL)

关键词的顺序不能变

Practice Link: <https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_all>

SQL Syntax: SELECT, FROM, WHERE

**Selection：**

SELECT Price, Category

FROM Product

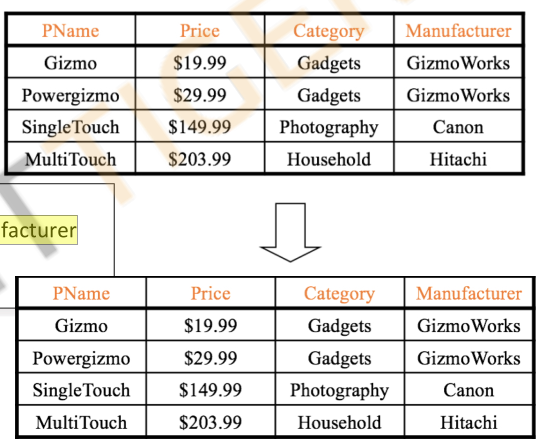
You can use \* to select all columns

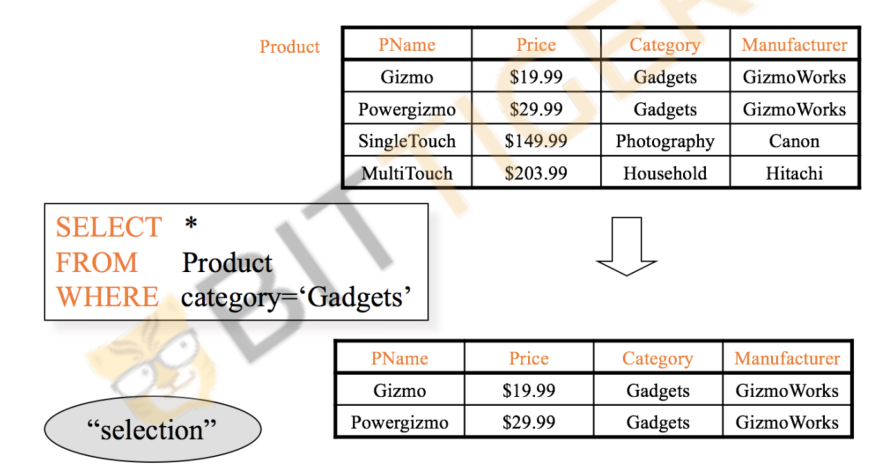
SELECT \*

FROM Product

SELECT PName, Price, Category, Manufacturer

FROM Product

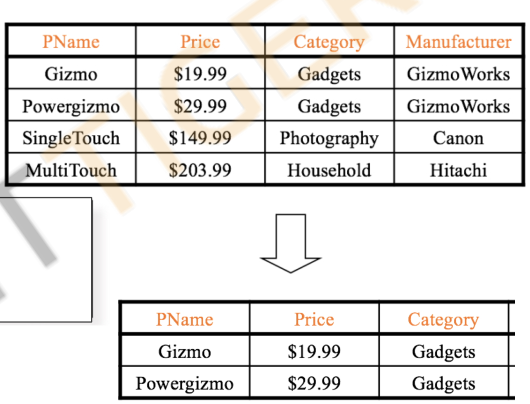




SELECT PName, Price, Category

FROM Product

WHERE Category = ‘Gadgets’



select是对列筛选：要哪些列

Where是对行筛选：有哪些行符合条件

SELECT NOTES

SELECT

● \*

● a1,a2,a3

● A1 **as** b1, a2 as b2——重命名

● a1 + a2 as b1, a1-a1\*a2/a3 as b2——**运算**的话**必须要进行重命名**，不重命名会起个COL3之类的；如果hive中没对col命名会report一个error

● **DISTINCT** a1, a2, a3——**选出unique values，写在select之后，**对所指定的列的combination做distinct。

● Top 10 (SQL server), e.g. **select top 10** .... **LIMIT** (MySQL)——Limit写在query的最后面

● Aggregate function (later)

WHERE NOTES

WHERE +condition

● >, <, =

● !=, <>

● LIKE——相似

● BETWEEN——指定range，范围。常用来选择日期 e.g. where date between 2018/1/1 and 2019/1/1

● IN——column在某个范围内。 e.g. where last\_name in (w,l,z)——常用来选出某个列名中的元素属于某个list中的所有data， 还经常用于sub-query

● IS NULL (refer to website)——是空值就返回。或者IS NOT NULL。经常用于join之后判断是否是空值。

在where中也可以进行逻辑判断： 都是和其他条件相结合的，用and or来连接许多conditions

● AND/OR

● NOT

Like and Wildcards Examples

寻找“类似的”情况

常加%，**%**可以**代表任意长度的任意字符**。但是**相对位置是严格**按照规定寻找的。

● Different between Like “%plus”, Like “plus%” and Like “%plus%”?

● Like “pl\_s”

**\_特指一个字符，长度固定是1**

**有时会问%和\_的区别：\_是一个，%任意多个。**

**Where xxcol like ‘%xx%’**

SQL Syntax: AGGREGATE FUNCTIONS

**MIN** returns the smallest value in a given column

**MAX** returns the largest value in a given column

**SUM** returns the sum of the numeric values in a given column

**AVG** returns the average value of a given column

**COUNT** returns the total **number of values in a given column——不计入missing value**

**COUNT(\*)** returns the **number of rows** in a table——**计入missing value**，只要有一行就会算一次

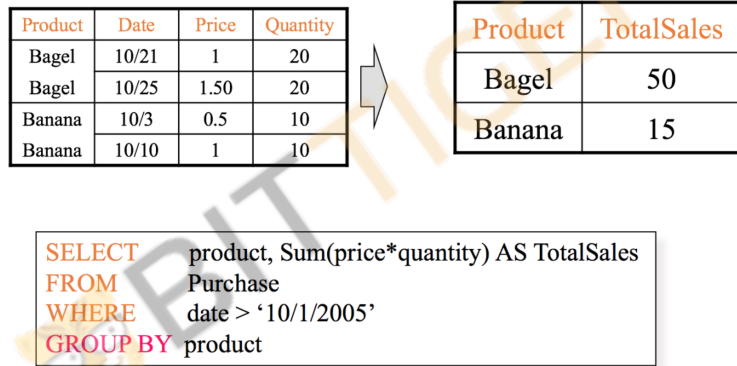
**COUNT(1)** = COUNT(\*)

所有的aggregate function一般都需要group by来对应，就是你要知道需要return的范围是啥。

即，对哪些组（分组依据是啥）分别做sum/min/max/avg/count？

如果直接 select min(sales) from sales table，会返回整个table中最小的sales。有时不想要整个column的最小值，而想看每个部门谁的sales最少，这时就要用到GROUP BY。

SQL Syntax: GROUP BY

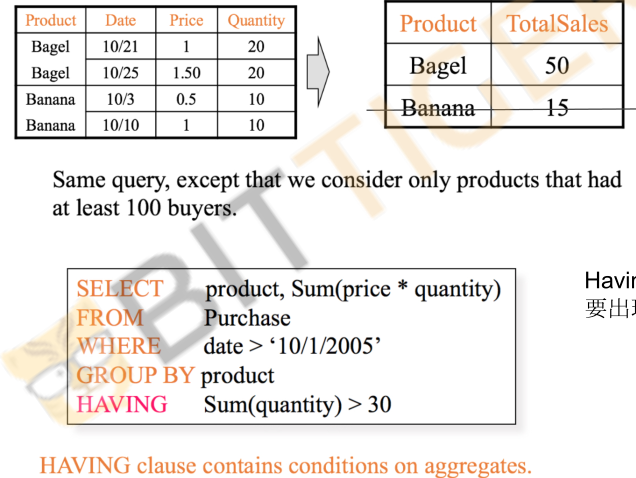


Select AGGREGATE function后也要重命名。

要得到**每个**不同的**产品**的TotalSales就**GROUP BY product**，想对哪一列的不同类别分别做某种aggregation就对该列group by。如果去掉group by就变成对该列一整列做aggregation。这里就是对整个table范围的total sales求sum。只得到一个总值。

SQL Syntax: HAVING

HAVING是GROUP BY之后的condition，区别于**where**的是where中的condition**直接作用于table 中的行**，以选出符合条件的行；而**HAVING**是先做一个aggregation，对**aggregate之后的值做条件限制**。



HAVING和WHERE里的condition不一定要出现在select里（相当于filter）

因为select语句中已经重命名新的列名，



这里能否写HAVING TotalSales > 30?

答案：不能。Select中的重命名发生在整个query的最后，**进行HAVING时，重命名as TotalSales还没有发生**，会找不到TotalSales而报错. 所以**HAVING中必须要用raw data写全（不能用重命名代替）**：Sum(price\*quantity) > 30.这也是面试中常犯的错误。

最常考的SQL知识点：

WHERE和HAVING的区别

Where 是一个约束声明，使用Where约束来自数据库的数据，对**raw data**进行操作，Where是在结果返回之前起作用的，**Where中不能使用聚合函数。**

Having是一个过滤声明，是在查询**返回结果集以后对查询结果进行的过滤操作（对聚合之后的column进行过滤）**，**在Having中必须使用聚合函数**。HAVING必须要发生在group by之后，先要有group by才有HAVING。HAVING中聚合范围是group by指定的。在查询过程中聚合语句(sum,min,max,avg,count)要比having子句优先执行。（先where挑出几行想要的raw data，然后group by聚合,最后having过滤结果）

而where子句在查询过程中执行优先级高于聚合语句（先选出来再对这些满足条件的做聚合）

面试中也会有一些陷阱，让你写一些filter，要注意什么时候该用where，什么时候该用having。

---------------------

如何用R跑sql

<https://www.r-bloggers.com/make-r-speak-sql-with-sqldf/>

可以在R中，SASS中，公司客户端中以及SQL server中跑SQL

在R中install tool搜sqldf

package ‘sqldf’ successfully unpacked and MD5 sums checked

The downloaded binary packages are in

C:\Users\hp\AppData\Local\Temp\RtmpARIwTJ\downloaded\_packages

跑这段代码，将刚刚讲syntax用的product表存在R中，存成一个data frame

> pname = c('Gizmo','Powergizmo','SingleTouch',

+ 'MultiTouch','DoubleTouch')

> price = c(19.99,29.99,149.99,203.99, 19.99)

> category = c('Gadgets','Gadgets','Photography',

+ 'Household','Photography')

> manufacturer = c('GizmoWorks','GizmoWorks','Canon',

+ 'Hitachi','Canon')

> product = data.frame(PName = pname, Price = price,

+ Category = category,

+ Manufacturer = manufacturer)

> product

PName Price Category Manufacturer

1 Gizmo 19.99 Gadgets GizmoWorks

2 Powergizmo 29.99 Gadgets GizmoWorks

3 SingleTouch 149.99 Photography Canon

4 MultiTouch 203.99 Household Hitachi

5 DoubleTouch 19.99 Photography Canon

在R中写SQL语句就是在sqldf中写：例如sqldf(“select \* from product limit 3”)。 limit 3是MySQL中的写法，意思是只显示前三行，用于preview。

Basic practice:

#### basic

# 1, get product name, price, category name, manufacturer for products price > 30, display by price high to low

Order by默认的排序：从小到大。

> sqldf("select \* from product where price > 30 order by price DESC") 大小写无所谓

PName Price Category Manufacturer

1 MultiTouch 203.99 Household Hitachi

2 SingleTouch 149.99 Photography Canon

# 2, get product name (PName) and price for products name that contains 'touch'

sqldf("select PName,price

+ from product

+ where PName like '%Touch%'")

PName Price

1 SingleTouch 149.99

2 MultiTouch 203.99

3 DoubleTouch 19.99

> sqldf("select PName,price

+ from product

+ where PName like '%touch%'")

PName Price

1 SingleTouch 149.99

2 MultiTouch 203.99

3 DoubleTouch 19.99

R中的SQL：Like中的内容大小写不敏感，但有的语言不行，尽量exactly和原写法一直，把大小写区分出来

在column name前加upper( ) or UPPER( )可以在输出结果中将该列名全部字母变成大写：

> sqldf("select **upper**(PName),price——注意这里没给它重命名，

+ from product

+ where **upper**(PName) like '%TOUCH%'")

**upper(PName)** Price——结果就会返回列名为upper(PName)

1 SINGLETOUCH 149.99

2 MULTITOUCH 203.99

3 DOUBLETOUCH 19.99

重命名后：

> sqldf("select upper(PName) as PName,price

+ from product

+ where upper(PName) like '%TOUCH%'")

PName Price

1 SINGLETOUCH 149.99

2 MULTITOUCH 203.99

3 DOUBLETOUCH 19.99

小写是LOWER( )

在使用LIKE( )进行比较时，有时我们不确定数据库中的相关字段是大写还是小写，就会使用upper()或lower() function先把该列中的全部行的全部字母统一转换成大写或小写，再比较

# 3, get category name whose average price > $30——考察having和where

> sqldf("select **C**ategory from product

+ group by **c**ategory

+ having **AVG**(price) > 30")

Category

1 Household

2 Photography

发现大小写都能跑：Select语句里面写的列名也不用和table中的列名完全大小写一致。大小写不敏感

# 4, get category name for the most expensive product

不建议的写法：

> ##not recommended:subquery-complicated and can only give you the max result,

> ## what if I want the two most expensive categories?

> sqldf("select category from product

+ where price = (select max(price) from product)")

Category

1 Household

> sqldf("select category from product

+ order by price desc

+ limit 1")

Category

1 Household

如果是要前两个最大的，在**MySQL中写limit 2**，SQL Server中一般写select **top 2** category from product order by price desc

**Limit**永远放在最后，return的是“limit”前面的那些语句返回的值的limit，是**对处理后的结果（比如aggregate后的结果）做前n行的切割**。不像where那样对raw data操作。

常考面试题：

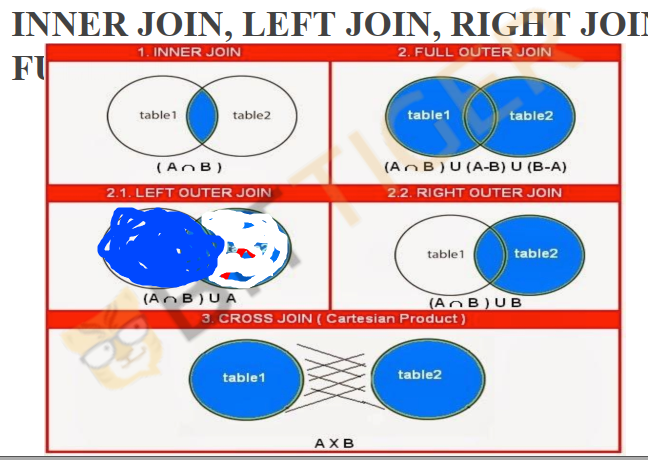
1. What is the difference between “WHERE” and “HAVING”?

2. SQL最简单的coding题，比如考察group by的使用。

**SQL Intermediate Syntax**

Intermediate的问题大部分是各种join，学会join就能解决80%-90%的sql问题，一般不会问特别advanced的问题。

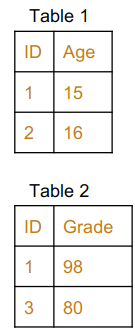
**JOIN**



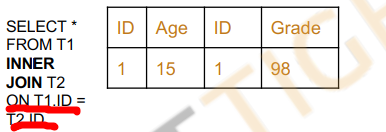
**Left** join: **取left table所有内容**，两个table中共有的行会存在，（只有left table有而）right table没有的行会显示missing

Cross join把A和B中元素两两相乘，结果会非常大，用的很少。

例子：



Inner join: 只取共有部分。INNER JOIN可以简写成JOIN，只写JOIN就默认是做inner join

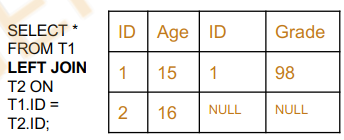


Join左边的table就是left table

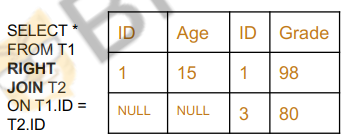
On就是condition，按照哪一列来join

Select跟之前一样，要返回的column是什么select什么

Left join：也可以写成LEFT OUTER JOIN

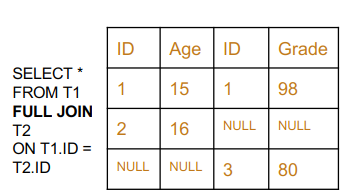


Right join：



Full join: 把left right table所有的行都放在一起，原来任何表有值的部分依然有值，没有对应数据的就missing

等同于 FULL OUTER JOIN



* 需要注意的一点：因为join的时候经常有重复的列名，所以可以用table1.列名，table2.列名来区分。

在select时可以用T1.xxx来选择表1的某一列, 或者用 T1.\*来选择T1的所有列。join之后还是可以选T1.XXX,即原table的名字可以直接用

* 常常需要：先join，再进行**筛选**：

比如

SELECT T1.ID

FROM Table1 AS T1 **LEFT JOIN** T2 ON T1.ID

= T2.ID

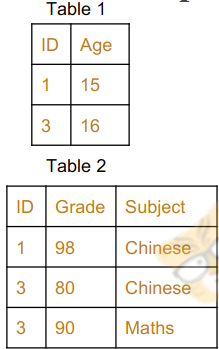
**Where** t2.ID IS NULL

先 **LEFT JOIN ID**再筛选出t2.ID IS **NOT** NULL的部分相当于**inner join**。但通常不会这样做，通常我们会想要找出left join过后**在T1里面但不在T2里面**的。就是Where t2.ID **IS NULL**

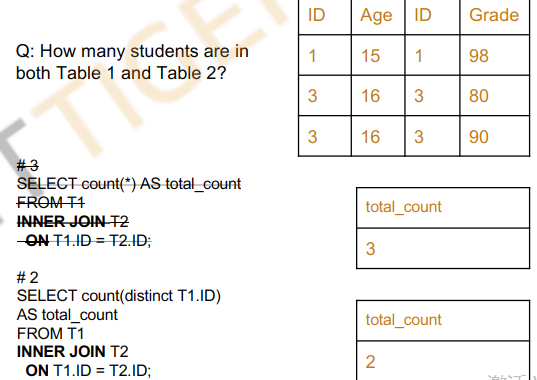
在join的时候可以对table的名字重命名，有时table的名字很长，不想重复写，就直接叫T1，重命名的方式可以写Table1 as T1,也可以不写as，直接把新名字跟在老名字后面：e.g. Table1 T1, user u, spend s。在SQL中所有的重命名的AS是可以省略的。

* 平时工作中写join多过于写subquery。

**DISTINCT**

****

**Common ‘Mistakes’: duplicate created unexpectedly**



一个学生可能考了多门科目，导致表2中同个ID重复出现，在与表1通过ID (inner) join时就会自动duplicate相关的record，这样在计算sum，或者计算count等时会产生错误。这时就要用到**distinct**

* Distinct——join之后常用的trick。

The **SELECT DISTINCT** statement is used to return only distinct (different) values.

SELECT DISTINCT column\_name1,column\_name2

FROM table\_name;

**SELECT COUNT(DISTINCT column\_name)**

**FROM table\_name**

* Count Distinct一般比较慢，如何不使用DISTINCT来数出distinct values？——Count distinct without using distinct

**使用group by:对某列group by就相当于自动对该列做deduplicate**

**写group by时可以按照column在SELECT后的相对位置来简写，group by 1就代表对SELECT后的第1个column分组**

Select count(\*)

From

(SELECT customer\_id

FROM table

**GROUP BY 1**);

相当于>> select count(distinct customer\_id) from table

**UNION and UNION ALL**

The UNION operator is used to combine the result-set of two or more SELECT statements.

The UNION operator selects only distinct values by default. **To allow duplicate values, use the ALL** keyword with UNION.

SELECT column\_name1,column\_name2 FROM table1

UNION (ALL)

SELECT column\_name1,column\_name2 FROM table2;

只要上下两个SELECT的column的数目是一样的，**类型是一样的**，就可以把它们UNION起来。

Union（叠）起来的两个column的类型必须是一样的，column name可以不一样。一般都会重命名成一样的名字:

SELECT column\_name\_A as column\_name FROM table1

UNION (ALL)

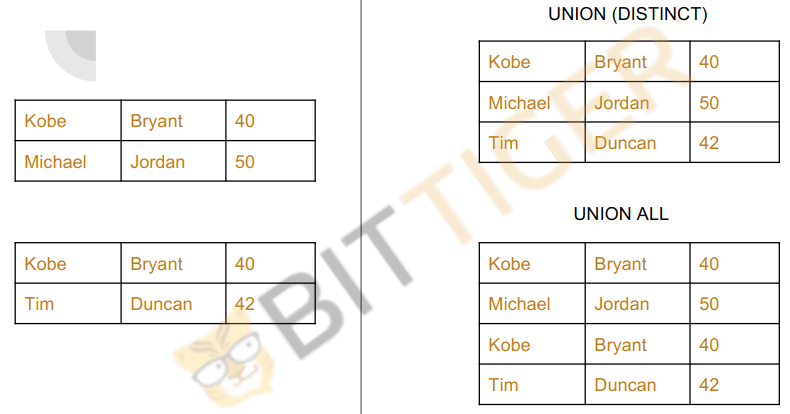
SELECT column\_name\_B as column\_name FROM table2;

Interview question: What is the difference between UNION and JOIN?

JOIN是横向，拼接列和列

UNION是纵向，拼接行和行（列数不变，只是增加行数）

例子：



一般什么时候会用union？同类信息但存在了不同的table，用union把它们合在一起。

比如一个table中是第一个班的data，table2是第二个班的data。为了得到所有学生的信息就

Select student\_id, gender from table1

UNION

Select student\_id, gender from table2

或者有时同样一个table，想把数据重复几遍，比如先选出所有的，把它定义成一个category叫做ALL，

再选出male的，再选出female的，再合成一个表，也就是说同一个table可以自己跟自己UNION。

**COALESCE**

**面试常考**

**Set default value if missing 用于数据补齐**

（相当于快速地做个丢失数据处理，如果放着很多missing后续很难拿来用。）missing在SQL中就是NULL。

有一些值，当它missing，根据常识我们知道它应该是0（或者别的imputation），就用coalesce补起来。

SELECT **COALESCE( s.spend, 0)** AS spend——s表中的spend里，如果是missing的value，让它等于0

FROM user u

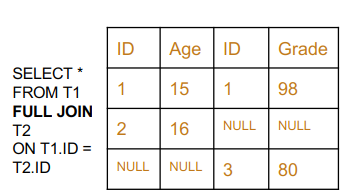
LEFT OUTER JOIN spend s

ON u.tsid = s.tsid

——把user表和spend表join起来发现很多user没有花过钱（有些user表里的ID在spend表里没有），就会显示missing value。没有花过钱就是spend = 0.就用COALESCE把missing value set为0.

还有一种情况是 **COALESCE****(T1.ID, T2.ID)**，把两列combine起来。（很常用）

比如full join之后，有的ID在T1中，有的ID在T2中，我想combine 两个表中的ID，作为join后的新表的unique identifier（primary key）。那就必须每行都有，不能是missing value。——combine these two columns of IDs，只要在某个表中出现了ID就把这个ID拿出来。 那就需要用到COALESCE(T1.ID, T2.ID)——T1里面有的ID就用T1的ID，如果T1里面的ID是missing value，就去找T2的ID，把T2的ID补到T1 ID missing的位置（ID 3 in this example），这样我们就可以create a new column called combined ID,把所有的ID放在一个列中。



还有一些时候，公司给的data不是特别clean，比如Amazon，用户可以在登录账号以前就做一些浏览和很多别的操作。这时我们就会发现data中的很多user id是missing的，但是可能有cookie id（他当时用的浏览器的cookie id）。那我们就可以用COALESCE(user\_id,cookie\_id)。当用户登录账号时就用user\_id,当用户没有登录账号时就用cookie\_id代替。

COALESCE可以嵌套。比如前两个ID都没有的话，再用别的column比如IP address之类的来代替ID。COALESCE(COALESCE(user\_id,cookie\_id),IP\_address)

在面试中，因为需要各种join，**join之后经常出现各种missing，希望你（用COALESCE）把它补齐**。

COALESCE可以用在SELECT里面也可以用在WHERE里面

**Creating CTE with ‘WITH’**

To make queries **reusable**, you can specifies a **temporary named result set**, known as a **common table expression** (CTE) in WITH statement. A CTE **can refer to itself and previously defined CTE**, cannot forward reference.

CTE的好处就是你前面定义了一次之后，后面可以反复调用，不需要在主query中再写好几遍。特别是在UNION ALL或SELF JOIN之后CTE table可能要用到好几次，用CTE非常简便。当需要写特别长的sub-query时，写WITH statement更清晰。

一个CTE跟下一个CTE之间：**后面的可以引用前面的**。可以定义好几个CTE，中间用逗号隔开，最后一个CTE后直接跟主query（不再需要用逗号隔开）。CTE里的column names可以不写。

WITH cte1 (column1, column2, ...) AS (SELECT ...),

cte2 (column1, column2, …) AS (SELECT …)——注意没有分号或逗号

SELECT ...

e.x.WITH Sales\_CTE (SalesPersonID, SalesOrderID) AS——WITH后跟你给这个CTE起的名字（CTE中有哪些column，optional）

( SELECT SalesPersonID,

SalesOrderID

FROM Sales.SalesOrderHeader

WHERE SalesPersonID IS NOT NULL )——**AS后加()括住CTE内容**，后面就可以写主query，主query之前不用逗号分隔

SELECT SalesPersonID,

COUNT(SalesOrderID) AS TotalSales

FROM Sales\_CTE——可以直接FROM CTE，调用

GROUP BY 1;

**TEMP TABLE**

另外一种简化query的方法。它跟CTE都可以把sub-query单独写出来，让你的query更易读。

The **SELECT INTO** statement copies data from one table and inserts it into a new table.

Temp table is much more preferred than sub-query in terms of readability.

SELECT column\_name(s)

INTO newtable [IN externaldb]——newtable就是temp table的名字

FROM table1;——最后要加个分号表示结束

Temp table和CTE都可以达到类似的作用，就是可以简化query但是它们两个有一些不同。不同在于：

Temp table是一个table，**真实的建立了一个名为new table的table**。你**可以在本session的任意地方调用这个table**。每次调用temp table只是去读取它，并不会每次都重新创建它。但是这个**table只存在到本次session结束**。当你关掉这个session再重新打开，之前创建的temp table就不存在了，要重新create。Session结束就是你关掉了SQL server/ R studio-这个session就结束了，当你重新打开server就会是一个新的session。或者当你share code，只要换一台机器，就不是同一个session了，你创建过的temp table就没了，需要重新run。对比正常的完整的table，它们被创建出来后**会一直存在于SQL SERVER上**，用户A和用户B都可以从自己的机器上直接调用这个table，是一个permanent的table。Temp table我create了只能我用，别人要用就是一个different session了，要自己去创建。

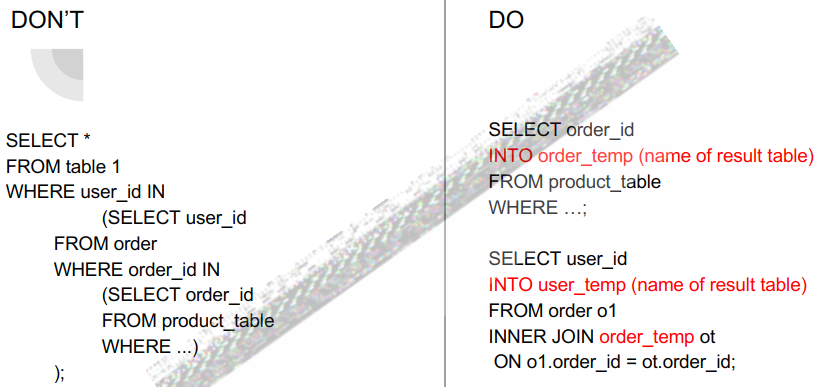
CTE相当于一个快捷键，它并没有create一个真实的table，它只是create了一个**指向**，机器看到sales\_CTE就去**run那一段对应的query**，每次看到sales\_CTE都要去重新run它括号中的那段（没有create出现成的table放在那等着你用）。而且**CTE是存在于query里面的**，属于query的一部分，写完CTE直接跟主query，没有用分号隔开，直到主query结束才有分号。**CTE只能在它所在的这个主query里面用**，当这个query的分号结束，我写了下一个query，那**下一个query就不能用这个sales\_CTE了，需要重新定义一个新的CTE**。

**该用哪一个？看你需要重复利用多少次**。这两个各有各的好处，sales\_CTE节省了空间，不需要真的去存储一个table，创造新table的读和写需要花费时间。一般来讲，如果你的**相关data只会被调用一次，那CTE更快**，就没必要专门创建一个table再去读这个table。但是有的情况下你**这个table需要用到好多次，而且是在好几个query里都需要用到**，不想每次都要重新create CTE，那就**一次性写个Temp table**，后面的query就会比较快了。

面试中用with比较多，因为query也就run一遍，WITH就能达到目的。With（CTE的建立）（和CTE的调用）是在一个query里面的，temp table建立的时候是单独的query，要用分号单独把它隔开，用的时候又是另一个query（至少会有两个query）。

Interview Question:

What is the difference between a view table and a temp table?



左边的是sub-query中嵌套又一层sub-query，不推荐，因为难读。

右边的是先把左边最内层的query result创建一个名为order\_temp的temp table，把左边的中间那层query创建成一个名为user\_temp的temp table，这个select user-id来自于order table和order\_temp的inner join，join on 两个table的order Id. 右边这个inner join和左边的sub query是等价的，因为where order\_id in order temp table中的order\_id就相当于order这个table和order\_temp这个table的inner join。

左边：IN的操作就是排除order表中的order\_Id不在IN后面的list中的那些行，只保留后面的list中有的

右边：INNER JOIN就是保留两者（order表和list）的overlap的部分，list就是order\_temp。所以order在order\_temp中不存在的order\_id对应的行就会被去掉。

所以两者表达的意思一样。JOIN和IN都可以达到这种筛选的作用。一般何时prefer用IN何时prefer用JOIN？

——看你要IN的list的长短。

如果order\_temp，即要筛选的list特别长，一般用JOIN比较好。用IN的话要一个个到后面的list中匹配，比较慢。

如果IN后面的list比较短，那就用IN比较好。

Sub-query并不限制只能用原来的table，这个没啥关系，sub-query就是指query里套着query，query的判断依据就是只要出现了一套SELECT...FROM...就是一个新的query。Sub-query里的table可以是任何table，叫做sub-query只是因为嵌套，加了括号，括号里面又有query。Sub-query的括号后面可以加个as，给这个sub-query一个名字，后面要用到这个sub-query时就可以直接写这个名字。

SQL中所有的重命名的AS都是可以省略的。

比如SELECT column\_name as cln——对列名重命名的as

FROM table1 as T1——对table名重命名的as

还有对sub-query重命名的as

什么时候不会省略：CTE时，WITH cte1 **AS** (SELECT...), 省略了会非常难懂，有AS在会很清楚你在定义一个CTE，AS后面就是CTE的内容。（Good practice就是有定义的部分尽量都写清楚AS什么什么，只有在table名重命名时省略掉也很清楚，面试时不放心尽量都写上。）

常考面试题——各种JOIN

1. What is the difference between **“INNER JOIN” and “LEFT JOIN”**?

——**第一轮面试HR面试**经常会问的问题。HR可能也不太会写，但她会问你这种理论性的问题，**问你有什么区别**。

1. What is the difference between **“UNION” and “UNION ALL”**?

Or what is the difference between **“JOIN” and “UNION”**?

3. **SELECT COUNT(DISTINCT id)**; What is the **more efficient** way to achieve count distinct ids?

A: select count(1) from (select id from table1 **group by** 1);

Group by is more efficient than count(distinct xx).group by比select distinct更快

Intermediate这部分主要考察怎么用LEFT/INNER JOIN，一般为SQL最普遍的coding题，会了JOIN之后就能解决80%的面试题。但难度从简单到困难都有。

Coding练习：#### intermediate——来自当时上课Piazza上出的每日一题。相当于LeetCode的medium难度。

1.# 给一个flight的table，有depature city和 arrival city，**求unique的不论顺序的组合**。比如 depature, arrival

# A B

# B A

# 结果只出现 A B。

flight = data.frame(departure = c('la','la','ny','ny','sf','sf'),

arrival = c('ny','sf','la','sf','aus','ny'))

一般这种题给的时间是15-20分钟，一旦有了思路就会比较简单了。

思路：出现了A B,如何筛选出B A？怎么知道是重复了？

创建一个新的table，以arrival作为departure，以departure作为arrival。

这道题的两种方法：method1——table自己和自己INNER JOIN选出不要的，然后NOT IN不要的。

method 2——table自己和自己LEFT JOIN然后直接选出不重复的以及重复的其中一组。为什么要LEFT JOIN?就是因为LEFT JOIN可以保留原表，不丢失信息，看得很清楚要哪些，不要哪些，方便写WHERE筛选

别人的答案：先挑出我们要去掉的那些行，然后再select \* from flight where xxx **NOT IN** (不要的)

SELECT distinct \*

FROM flight

WHERE (departure,arrival) not in

(

SELECT A.\*

FROM flight AS A

INNER JOIN flight AS B

ON A.departure = B.arrival AND

A.arrival = B.departure

WHERE A.departure > A.arrival

)

拆解：

首先看下not in 后面的括号中的结果：

找出配对的那些行程：

> sqldf("

+ SELECT **\***

+ FROM flight AS A

+ INNER JOIN flight AS B

+ ON A.departure = B.arrival AND

+ A.arrival = B.departure

+

+ ")

departure arrival departure..3 arrival..4

1 la ny ny la

2 ny la la ny

3 ny sf sf ny

4 sf ny ny sf

把选出的范围改成A.\*，只要原表中的那两列，再加上WHERE筛选条件——A B和B A这两行里一定有一行的departure > arrival，用这个条件可以只留下A B和B A的其中一行。

> sqldf("

+ SELECT A.\*

+ FROM flight AS A

+ INNER JOIN flight AS B

+ ON A.departure = B.arrival AND

+ A.arrival = B.departure

+ WHERE A.departure > A.arrival

+ ")

departure arrival

1 ny la

2 sf ny

本来的flight table长这样：

> flight

departure arrival

1 la ny

2 la sf

**3 ny la**

4 ny sf

5 sf aus

**6**  **sf ny**

3 和 6 是我们要去掉的，刚才已经选出来这两行了，接下来只需要去掉它们——NOT IN

SELECT \*

FROM flight

WHERE (departure,arrival) not in

(刚才那一串)

**最终结果：**

> sqldf("SELECT \* FROM flight

+ WHERE (departure,arrival) NOT IN

+ (SELECT A.\*

+ FROM flight AS A

+ INNER JOIN flight AS B

+ ON A.departure = B.arrival AND

+ A.arrival = B.departure

+ WHERE A.departure > A.arrival)")

departure arrival

1 la ny

2 la sf

3 ny sf

4 sf aus

但是NOT IN一般比较慢，尤其是当table比较大的时候。可以尝试不同的方式：

我自己的思路是**用left join**，这样有重复的行会按照F1.d=F2.a & F1.a=F2.d匹配出来（显示没有NULL），这时光看右侧表就可以知道如何筛选ABBA的其中之一（e.g.D > A），没有重复的会显示NULL，然后用WHERE筛选出F2表中IS NULL的 + A B和B A中的其中一行（比如同样限制departure > arrival）（DAAD重复的话只选其中一组，DAAD不重复的话直接入选）。

先看下LEFT JOIN的结果：注意ON的条件，要按照DAAD的回路来JOIN ON，老师在这里没有AND，跑出一大串结果，很乱，没在规定时间内答出来。

sqldf("SELECT \*

+ FROM flight as F1

+ LEFT JOIN flight as F2

+ ON F1.departure = F2.arrival

+ AND F1.arrival = F2.departure")

departure arrival departure..3 arrival..4

1 la ny ny la

2 la sf <NA> <NA>

3 ny la la ny

4 ny sf sf ny

5 sf aus <NA> <NA>

6 sf ny ny sf

> sqldf("SELECT F1.\*

+ FROM flight as F1

+ LEFT JOIN flight as F2

+ ON F1.departure = F2.arrival

+ AND F1.arrival = F2.departure

+ WHERE F2.departure > F2.arrival

+ OR F2.departure IS NULL")

departure arrival

1 la ny

2 la sf

3 ny sf

4 sf aus

例题2：#### 三个ins table.——Facebook原题，Facebook常用的题型

经典的几个table join——join时要找出那些mutual column。

t1 = data.frame(user\_name = c('LBJ','Curry','Brady'),

sports\_category = c('basketball','basketball','football'))

t2 = data.frame(user\_id = c(11,22,33,44,55,66,77,88,99),

user\_name = c('LBJ','Curry','Brady','fan4','fan5','fan6',

'fan7','fan8','fan9'))

t3 = data.frame(user\_id = c(11,11,22,22,22,33,33,33,33),

user\_id\_following = c(22,44,11,55,66,77,88,99,44))

# t1(user\_name, sports\_category) t1中只有celebrity(名人)运动员。pk=user\_name

# t2(user\_id, user\_name, registration\_date)

# t2中是所有人的用户信息，包括celebrity和普通人，且不会出现celebrity和普通人重名的情况（重要假设）。pk=user\_id

# t3(user\_id, user\_id\_following, follow\_date)

# 用户follow信息，注意user\_id\_following中包括celebrity和普通人

# 计算每个category有多少人follow

我的思路：要求的“每个category有多少人follow”这个信息是否全了？不全的话要怎么获得？——有关following的信息user\_id\_following在表3中，而sports\_category在表1中，表1-t1(user\_name, sports\_category)和表3-t3(user\_id, user\_id\_following)无法直接join,需要通过 t2(user\_id, user\_name)来连接。所以要JOIN这三个表。

老师讲的：如果在面试，怎么回答——要计算每个category有多少人follow，我需要两个info——category info and follower info。category info is in t1 and follower info is in t3. So I need to combine t1 and t3. But t1 and t3 have no common column so I need t2 as a bridge to connect t1 and t3. Now I will begin with select first and I will leave it blank for now.（写个select然后放着最后写，[一般写完from和join用select\*看一下新表长啥样，都有哪些列，需不需要整理整合，才能知道最后我要的列是啥，才能具体写select哪些列]，先写后面的，FROM啥JOIN啥和啥）....边写边讲。可以每次做题的时候给自己讲练习面试场景。

我自己的尝试：

> ## join the firt two tables

> sqldf("SELECT t2.user\_id,COALESCE(t2.user\_name,t1.user\_name) AS user\_name,t1.sports\_category

+ FROM t2 LEFT JOIN t1

+ ON t1.user\_name = t2.user\_name;")

user\_id user\_name sports\_category

1 11 LBJ basketball

2 22 Curry basketball

3 33 Brady football

4 44 fan4 <NA>

5 55 fan5 <NA>

6 66 fan6 <NA>

7 77 fan7 <NA>

8 88 fan8 <NA>

9 99 fan9 <NA>

这里遇到一个问题，就是想对这个新的join后的table重命名，但无论是在SELECT xxx 后加AS new\_table\_name,还是在下一行写INTO new\_table\_name来做个temp table，都会报错。不知道是不是因为R中不支持这么多功能，而且发现了 RIGHT and FULL OUTER JOINs are not currently supported in R。最后尝试了写成CTE放在主query里，it finally works。

#join the three tables

> sqldf("WITH comb12\_CTE AS

+ (SELECT t2.user\_id,COALESCE(t2.user\_name,t1.user\_name) AS user\_name,t1.sports\_category

+ FROM t2 LEFT JOIN t1

+ ON t1.user\_name = t2.user\_name)

+ SELECT **\***

+ FROM t3 LEFT JOIN comb12\_CTE

+ ON t3.user\_id = comb12\_CTE.user\_id

+

+ ")

user\_id user\_id\_following user\_id..3 user\_name sports\_category

1 11 22 11 LBJ basketball

2 11 44 11 LBJ basketball

3 22 11 22 Curry basketball

4 22 55 22 Curry basketball

5 22 66 22 Curry basketball

6 33 77 33 Brady football

7 33 88 33 Brady football

8 33 99 33 Brady football

9 33 44 33 Brady football

至此我们已经能看到每一类sports里都有user\_id\_following这个列与之对应，count一下following就好了。只需要把刚刚代码中SELECT的内容改成COUNT。

最终结果：

> sqldf("WITH comb12\_CTE AS

+ (SELECT t2.user\_id,COALESCE(t2.user\_name,t1.user\_name) AS user\_name,t1.sports\_category

+ FROM t2 LEFT JOIN t1

+ ON t1.user\_name = t2.user\_name)

+ **SELECT sports\_category,COUNT(DISTINCT user\_id\_following) AS num\_user\_following**

+ FROM t3 LEFT JOIN comb12\_CTE

+ ON t3.user\_id = comb12\_CTE.user\_id

+ GROUP BY 1

+ ")

sports\_category num\_user\_following

1 basketball 5

2 football 4

在SQL中，**可以并排连着很多join**，不需要把它变成sub-query结合成一个表再join，可以直接join完一个紧接着join。

> ##other student's answer: inner join three tables directly

> sqldf("SELECT sports\_category,COUNT(user\_id\_following) AS num\_follower

+ FROM t3 JOIN t2

+ ON t3.user\_id = t2.user\_id

+ JOIN t1

+ ON t1.user\_name = t2.user\_name

+ GROUP BY 1")

sports\_category num\_follower

1 basketball 5

2 football 4

原来不用写CTE，**直接连着join**就可以实现多表join。

如果你一开始写的都是join（INNER JOIN）面试官中间可能就会提示性/纠正性的问，假如还有一个名人，他对应的sports\_category没有fan following，你用inner join就会把这个category直接去掉了还怎么统计？没人follow就是follower为0个，要保存成0就要全部改成**left join**，就算没粉丝，T1中的**category也不会被去掉**。那最后统计时也要加上**COALESCE(count(user\_id\_following),0)**，把本来user\_id\_following是NULL(所以count of follower也会是NULL)的category记为0个follower。——**考虑要全面**！！这个题里因为我们没发现有哪个**category对应的所需信息即user\_id\_following为NULL**，就没有考虑用0做imputation和**处理missing value的情况**，但是如果数据量大真的要考虑全，这种情况很有可能出现。

面试时没有人expect你一下全写对，一般先跟面试官讨论下我的思路大概是啥，（可以初步写点东西看看结果都长啥样，能否对进行下一步有一些提醒和帮助）。面试官如果觉得你的思路差不多是对的,on the right track再下笔。反正不要沉默，不要闷头在那儿自己想，这样他不知道你在干嘛，交谈也无法进行下去。如果你有问题有点不清楚的也及时和面试官讨论，**有些confused的要去verify实际题目想表达的意思**，比如这个题可能有人就不确定t3中到底是谁follow谁？Join的时候是join on user\_id还是user\_id\_following？(因为join是要join on celebrity 的id但最终count是count fans id，如果搞反了就全错了)。这些都提前问清楚再操作，以免自己乱判断导致一错到底，在错误的方向上白费努力。

**写code之前要跟面试官讨论的思路：要达到某个目的，你需要哪些信息？这些信息分别要从哪里获得？为了获得这些信息你想干嘛？讲完这个框架再慢慢完善细节。保证你和你的面试官在same page：你知道他在问啥，他知道你在干嘛。**

当你写完一个基本版本后可能面试官会进一步提问，假如想要一个更fancy点的功能，比如某类sports没有粉丝就展示为0，那就再改成别的或加一点别的，现场改就行。

面试现场考SQL时，你的code可以出错，出错后你能否**自己尽快debug**也是一个重要的能力的展示。能在现场Figure out你的code为啥不对。

注意，写了group by之后其他地方一定要aggregation，不然肯定有报错。只要写了group by，其他地方出现的column（尤其是select后紧跟着的）都得是aggregation，要么就是group by里的内容，（by后面的那个column可以单独出现）。

例题三：

member = data.frame(member\_id = c(11,11,22,22,33,44),

email = c('x11\_1@gmail', 'x11\_2@gmail',

'x22\_1@gmail','x22\_2@gmail',

'x33@gmail','x44@gmail'))

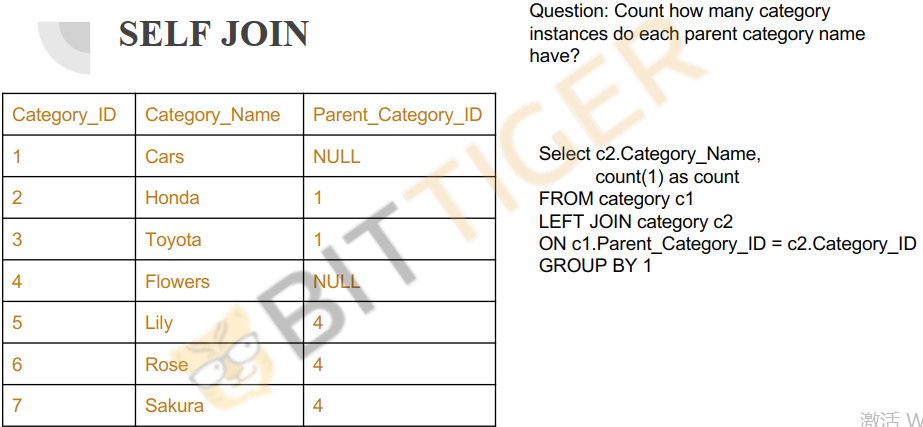
# member\_id, email\_address

# 一个member 可能有2个email address

# output：memeber id，email1，email2

**SQL Advanced**

**1.Self-join**



其实之前的例题已经用过了，就是join，只不过是一个table**跟它自己join**，没什么fancy的。但是面试中可能会问到，会考，就单独拿出来加强印象。平时也经常会用到self-join.

#simple question: count how many category instances do each parent category id have?

sqldf("Select Parent\_Category\_ID,

count(category\_Name) as count

FROM category

GROUP BY 1")

我自己的思路：

这道题要Count how many**category instance**s do each **parent category name** have，我需要parent category name和category instance处在不同的列才能count和group by，或者说（老师的说法）如何用parent category name代替Parent\_Category\_ID.要把自己的表中的某列信息拆分成不同的列来操作（比如count）时就要用到self-join了。

SELECT c2.Category\_Name,COUNT(c1.Category\_Name)

FROM category c1

JOIN category c2

ON c1.Parent\_Category\_ID = c2.Category\_ID

WHERE c2.Category\_Name IS NOT NULL

GROUP BY c2.Category\_Name

答案很正确

但是细节上，我发现join完不用写WHERE这一句，因为**INNER JOIN的时候自动去掉了所on的列中的null对应的行，（**这里因为Parent\_Category\_ID的第1行和第4行是NULL，JOIN的时候就会去掉这两行，）只留下（两个被join的表共有的行，即）Category\_ID为23567的行。

> sqldf("SELECT \*

+ FROM category c1

+ JOIN category c2

+ ON c1.Parent\_Category\_ID = c2.Category\_ID")

Category\_ID Category\_Name Parent\_Category\_ID Category\_ID..4 Category\_Name..5

1 2 Honda 1 1 Cars

2 3 Toyota 1 1 Cars

3 5 Lily 4 4 Flowers

4 6 Rose 4 4 Flowers

5 7 Sakura 4 4 Flowers

Parent\_Category\_ID..6

1 NULL

2 NULL

3 NULL

4 NULL

5 NULL

对比join和left join:left join会保留表格的所有行，让左表完全是原表的原样

> sqldf("SELECT \*

+ FROM category c1

+ **LEFT JOIN** category c2

+ ON c1.Parent\_Category\_ID = c2.Category\_ID")

Category\_ID Category\_Name Parent\_Category\_ID Category\_ID..4 Category\_Name..5

1 1 Cars NULL NA <NA>

2 2 Honda 1 1 Cars

3 3 Toyota 1 1 Cars

4 4 Flowers NULL NA <NA>

5 5 Lily 4 4 Flowers

6 6 Rose 4 4 Flowers

7 7 Sakura 4 4 Flowers

Parent\_Category\_ID..6

1 <NA>

2 NULL

3 NULL

4 <NA>

5 NULL

6 NULL

7 NULL

count(1) and count(字段)

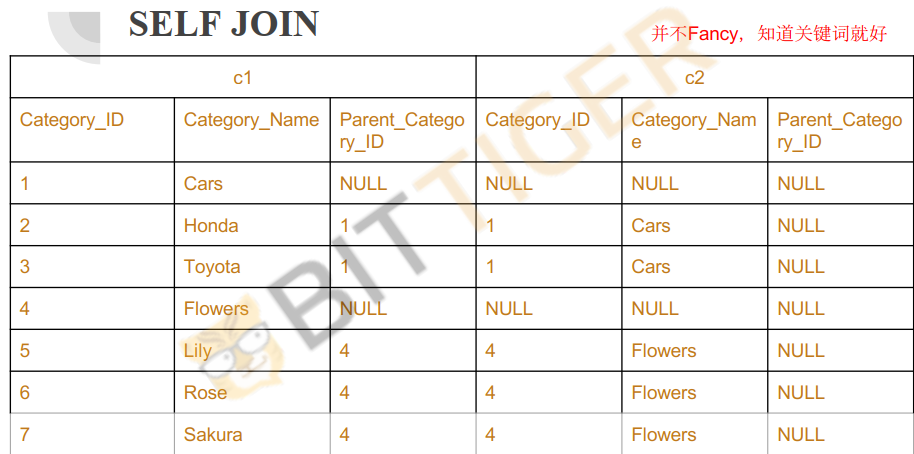
两者的主要区别是

（1） count(1) 会统计表中的所有的记录数，包含字段为null 的记录。

（2） count(字段) 会统计该字段在表中出现的次数，忽略字段为null 的情况。即不统计字段为null 的记录。

经过对比还发现，count c1的category name group by c2的category name，和count c2的category name group by c2的category name结果是一样的，都能达到count的目的。

所以用inner join的话可以去掉WHERE c2.Category\_Name IS NOT NULL，用LEFT JOIN的话如果是COUNT（列名）也不用加WHERE c2.Category\_Name IS NOT NULL，如果是用COUNT（1），因为会数着NULL，这时可以加上WHERE c2.Category\_Name IS NOT NULL。



别的同学的答案——

这样**直接FROM两个表**，最终结果没错，但这么做会对先这两个表做**full outer join**，交叉相乘得到一个非常大的表，然后再执行WHERE来filter出需要的行。很低效率，工作中很少用，面试时写这样直接FROM两个table的会被认为是非常不efficienct的答案，虽然结果一样，但是**平时尽量不要这样写**，**建议写成self-join**. 这道题LEFT JOIN和JOIN都可以。

> sqldf("SELECT c2.Category\_Name,count(1) as count

+ **FROM category c1, category c2**

+ WHERE c1.Parent\_Category\_ID = c2.Category\_ID

+ GROUP BY 1")

Category\_Name count

1 Cars 2

2 Flowers 3

另一种given answer：和我自己的答案一模一样

SELECT t2.Category\_ID, COUNT(1) AS count

FROM category AS t1

INNER JOIN category AS t2

ON t1.Parent\_Category\_ID = t2.Category\_ID

WHERE t2.Category\_ID IS NOT NULL

GROUP BY 1; self-join练习题：

# get **employee ids** whose salary is higher than his manager

employee\_salary = data.frame(employee\_id = c(11,22,33,44,55,66,77),

salary = c(60000, 55000, 130000, 70000, 50000, 90000, 100000),

manager\_id = c(33, 33, 77, 44, 44, 44, 77),

department = c('legal','legal','legal','sales','sales','sales','executive'))

employee\_id salary manager\_id department

1 11 60000 33 legal

2 22 55000 33 legal

3 33 130000 77 legal

4 44 70000 44 sales

5 55 50000 44 sales

6 66 90000 44 sales

7 77 100000 77 executive

当你看到面试时只有一个table，又不能直接可以得出结果，一般就是要用self-join了。

> sqldf("SELECT \*

+ FROM employee\_salary e1

+ LEFT JOIN employee\_salary e2

+ ON e1.manager\_id = e2.employee\_id")

employee\_id salary manager\_id department employee\_id..5 salary..6 manager\_id..7

1 11 60000 33 legal 33 130000 77

2 22 55000 33 legal 33 130000 77

3 33 130000 77 legal 77 100000 77

4 44 70000 44 sales 44 70000 44

5 55 50000 44 sales 44 70000 44

6 66 90000 44 sales 44 70000 44

7 77 100000 77 executive 77 100000 77

department..8

1 legal

2 legal

3 executive

4 sales

5 sales

6 sales

7 executive

SELECT employee\_id,e1.salary,manager\_id,e2.salary,

FROM employee\_salary e1

LEFT JOIN employee\_salary e2

ON e1.manager\_id = e2.employee\_id

WHERE e1.salary > e2.salary

> sqldf("SELECT e1.employee\_id,e1.salary,e2.employee\_id as manager,e2.salary as manager\_salary

+ FROM employee\_salary e1

+ LEFT JOIN employee\_salary e2

+ ON e1.manager\_id = e2.employee\_id

+ WHERE e1.salary > e2.salary")

employee\_id salary manager manager\_salary

1 33 130000 77 1e+05

2 66 90000 44 7e+04

这道题中只需要get employee ids，所以SELECT后面那一串列可以在最终结果中去掉，只留下e1.employee\_id

这道题也可以用inner join，因为所需的信息既要在左边又要在右边，两边都要存在才行，因为要找到employee salary（左） & his managers’ salary（右）。如果有NULL，也不符合e1.salary > e2.salary，也会被filter掉，所以用LEFT JOIN和INNER JOIN均可。

————————————————————————————————————————

# get the top 2 salaries within each department <https://sangei.iteye.com/blog/2359584>

——以下都是自己的分析，老师cover的简单做法见WINDOW FUNCTION那一part

> sqldf("SELECT e1.salary,e1.department,e1.employee\_id

+ FROM employee\_salary e1

+ LEFT JOIN employee\_salary e2

+ ON e1.department = e2.department AND e1.salary < e2.salary

+ GROUP BY e1.department,e1.salary,e1.employee\_id

+ HAVING COUNT(e2.employee\_id)<2

+ ORDER BY e1.department,e1.salary DESC

+ ")

salary department employee\_id

1 100000 executive 77

2 130000 legal 33

3 60000 legal 11

4 90000 sales 66

5 70000 sales 44

这段代码是照着网上的self-join写出来的，当不能用window function时的硬核解决办法。

精彩之处在于join的条件(把比自己大的挑出来放在右表)和group by+having的组合运用来count出“top n”；最后order

By多列的组合来对每个department中的salary做排序。

拆解：先看下按照上述ON的条件LEFT JOIN得到的表格长啥样

> sqldf("SELECT \*

+ FROM employee\_salary e1

+ LEFT JOIN employee\_salary e2

+ ON **e1.department = e2.department AND e1.salary < e2.salary**")

employee\_id salary manager\_id department employee\_id..5 salary..6 manager\_id..7 department..8

1 11 60000 33 legal 33 130000 77 legal

2 22 55000 33 legal 11 60000 33 legal

3 22 55000 33 legal 33 130000 77 legal

4 33 130000 77 legal NA NA NA <NA>

5 44 70000 44 sales 66 90000 44 sales

6 55 50000 44 sales 44 70000 44 sales

7 55 50000 44 sales 66 90000 44 sales

8 66 90000 44 sales NA NA NA <NA>

9 77 100000 77 executive NA NA NA <NA>

打草稿，根据该LEFT JOIN的规则画一画，就大体理解它做了什么了。

根据这个表，我们把 比一条记录（一行）的salary高的同department中的记录选出来放在右表中。在该department中已经是工资最高的人对应的右表为空（无敌），比他工资高的只有一个人的就对应一条记录，比他工资高的有几个人就会在左表中copy出几条他自己的记录，用以一一匹配右表中比他工资高的同事。

因为我们要选每个部门中工资最高的前两个人，首先要选出**右表为NULL的那些人，他们的工资在相应部门中是最高的**。接下来再选出**只有一个人比他高的，也就是第二的**。就是说这些人的**id在左表中只出现过一次**。为了验证这个筛选方法正确与否，可以用草稿纸手算一下expect的结果是什么。根据手算，expect选出legal部门中的33和11，sales部门中的66和44，executive部门中的77.这些记录的确**只在左表中出现了一次**，剩余记录都在左表中出现了不止一次。（我们的目标也就是去掉22和55，而这俩都重复出现了多行。）

用GROUP BY左表的前三列来去重，筛选条件是 COUNT(e2.employee\_id)<2，也就是说表一的**前三列所确定的某行 在右表中只能出现过1次或0次**。（右表意味着比左表大的人组成的表，count(e2.某列)<2意味着只能有1个人比左表大或0个人比左表大，当有1个人比左表大时选出的是该部门的第2，当有0个人比左表大时选出的是该部门的第1，这就是top2，真的曲折。。。）用这样的方法我们就成功的去掉了22和55，因为他俩都在右表中各自对应有两条记录COUNT(e2.employee\_id)=2了。

> ##intermediate result2 of group by and having--remove 22 and 55,which with 2 records in right table each

> sqldf("SELECT \*

+ FROM employee\_salary e1

+ LEFT JOIN employee\_salary e2

+ ON e1.department = e2.department AND e1.salary < e2.salary

+ GROUP BY e1.department,e1.salary,e1.employee\_id

+ HAVING COUNT(e2.employee\_id)<2")

employee\_id salary manager\_id department employee\_id..5 salary..6 manager\_id..7 department..8

1 77 100000 77 executive NA NA NA <NA>

2 11 60000 33 legal 33 130000 77 legal

3 33 130000 77 legal NA NA NA <NA>

4 44 70000 44 sales 66 90000 44 sales

5 66 90000 44 sales NA NA NA <NA>

到这里我们已经选出了符合条件的所有人，但我们需要在每个部门内有序排列，让每个部门的第一高于第二。所以最后加一句，对最终结果排个序。ORDER BY e1.department,e1.salary DESC。这里要注意的是不能只对e.salary这一列排序，这样将得到不分部门的全局排序，而我们要的是**每个部门内部的薪水**的**由高到低排列**，所以**order by的是department和salary这两列的结合（注意顺序）**。department在前就是先对department做个排列（按照首字母顺序），关键是**这么做能把同个department的人放在一起**。Salary在后，就是**在满足department已经符合顺序的基础上，salary要符合DESC**的顺序——这样做就实现了**在每个department内部对salary排序而非全局排序**。

最后，我们只选左表的相关列并调整顺序，把最重要的放在前面。加上ORDER BY。最终结果：

sqldf("SELECT e1.salary,e1.department,e1.employee\_id

+ FROM employee\_salary e1

+ LEFT JOIN employee\_salary e2

+ ON e1.department = e2.department AND e1.salary < e2.salary

+ GROUP BY e1.department,e1.salary,e1.employee\_id

+ HAVING COUNT(e2.employee\_id)<2

+ ORDER BY e1.department,e1.salary DESC

+ ")

salary department employee\_id

1 100000 executive 77

2 130000 legal 33

3 60000 legal 11

4 90000 sales 66

5 70000 sales 44

————————————————————————————————————————————

还可以用union实现,（因为是不同部门的result的UNION，一定是DISTINCT VALUES，也不会出现duplicate的行，所以用UNION就可以，可以不用UNION ALL）

employee\_id salary manager\_id department

1 11 60000 33 legal

2 22 55000 33 legal

3 33 130000 77 legal

4 44 70000 44 sales

5 55 50000 44 sales

6 66 90000 44 sales

7 77 100000 77 executive

SELECT employee\_id,salary,department

FROM employee\_salary

WHERE department = legal

ORDER BY salary DESC

LIMIT 2

UNION

SELECT employee\_id,salary,department

FROM employee\_salary

WHERE department = sales

ORDER BY salary DESC

LIMIT 2

SELECT employee\_id,salary,department

FROM employee\_salary

WHERE department = executive

ORDER BY salary DESC

LIMIT 2

Error in result\_create(conn@ptr, statement) :

ORDER BY clause should come after UNION ALL not before

**2.CASE WHEN**

用于在SQL中define一个condition。

**SELECT CASE (****"column\_name")**

**WHEN "value1" THEN "result1"#——当这个column的value是value1就return result 1**

**WHEN "value2" THEN "result2"**

**...**

**[ELSE “NOT”]**

**END**#——一定要加END，表示我的CASE WHEN写完了。END后可以加个AS new\_column\_name

**FROM "table\_name";** ##——用CASE WHEN define一个新的column

这种写法就是如果你的condition只是on one column，所有的condition都是在看你所写的"column\_name"怎样怎样，

就可以CASE (“column\_name”) WHEN value怎样 THEN result怎样。如果你的condition比较复杂，另外一种写法是：

**SELECT CASE**

**WHEN condition1 THEN “result1”**

**WHEN condition2 THEN “result2”**

**...**

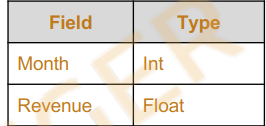
**[ELSE “NOT”]**

**END**

**FROM “table name”**

其中的condition可以是任何条件，比如像之前的写法中的”column1 = value1” or “column 1 > column 2”etc.

例子：根据month定义Quarter：



SELECT CASE

WHEN month in (1,2,3) THEN ‘Q1’

WHEN month in (4,5,6) THEN ‘Q2’

WHEN month in (7,8,9) THEN ‘Q3’

WHEN month in (10,11,12) THEN ‘Q4’

(ELSE “N/A”) #it is optional, but could be a common mistake，有时忘记定义ELSE会发现你漏掉了一些情况，在CASE WHEN

##中没有cover所有的情况，那没有被定义的情况出现时结果会自动变为missing value，就是NULL。

END AS Quarter #一定要记得写END表示这段CASE WHEN写完了

, SUM(Revenue) as sum\_rev

FROM table

GROUP BY 1

**值得注意的一点**：**CASE WHEN中的conditions是从上到下evaluate的**，比如

CASE WHEN x > 10 THEN “10+”

CASE WHEN x > 5 THEN “5+”

....

如果被判断的数值是15，就会返回10+，后面的直接skip掉不会再看。如果不满足x>10，才会看下一个condition是否满足。所以不用担心如果x>10，是否同时满足两个condition？**不会同时满足两个condition**，它是从上往下看的，**如果满足第一个，就会划为第一种情况，不会再往下看了。**

**CASE WHEN**在面试中挺**常用**的，因为经常要定义一个category，判断什么什么condition，都是用CASE WHEN。

**CASE WHEN可以写任意多个条件**（最少一个condition）。当只有一个condition时，可以写

SELECT CASE

WHEN condition THEN “value if yes”

ELSE “value if no”

END as new\_column\_name

FROM “table name”

也可以写成IF。即，**IF是CASE WHEN的一种特殊情况的简便写法**。**当只有一个condition时，可以用IF来简写**。此时它们的作用一样。

**3.IF**

SELECT IF( condition, value if yes, value if no)#——如果不写 value if no，不满足condition时会返回NULL。

FROM "table\_name";

**Can replace CASE WHEN when there is only one condition**

SELECT IF(month in (11, 12),

‘Holiday\_Season’,

‘Other’) **as seson**

, SUM(Revenue) as sum\_rev

FROM table

GROUP BY 1

例题1：在Facebook和Linkedin等有connection的社交网络的公司的面试中很常见。属于较难的题。

# define relationship as ‘mutual friend','follow'(col1 follow col2),'fans(col2 follow col1)'

follow = data.frame(user\_id = c(11,11,22,22,22,33,33,33,33),

user\_id\_following = c(22,44,11,55,66,77,88,99,44))

user\_id user\_id\_following

1 11 22

2 11 44

3 22 11

4 22 55

5 22 66

6 33 77

7 33 88

8 33 99

9 33 44

不光要用CASE WHEN，还要用到SELF JOIN。因为这里只有一个table,要找到两个user之间的关系，就要想到self join。

因为两个col其实都是id，只是一个follow着另外一个，所以就用f1的其中一个col和f2的另外一个col来join。这样Join以后可以看到全部user之间谁follow谁，又被谁follow的交叉关系。

这道题比较tricky，做题之前要先想清楚，我想return一个什么样的structure。15-20min要写出来。

> sqldf("SELECT \*

+ FROM follow f1

+ LEFT JOIN follow f2

+ ON f1.user\_id\_following = f2.user\_id")

user\_id user\_id\_following user\_id..3 user\_id\_following..4

1 11 22 22 11

2 11 22 22 55

3 11 22 22 66

4 11 44 NA NA

5 22 11 11 22

6 22 11 11 44

7 22 55 NA NA

8 22 66 NA NA

9 33 77 NA NA

10 33 88 NA NA

11 33 99 NA NA

12 33 44 NA NA

首先left join自己，再根据这个中间结果，写各种判断条件。

My final answer:

> sqldf("SELECT DISTINCT CASE

+ WHEN f1.user\_id\_following = f2.user\_id

+ AND f1.user\_id = f2.user\_id\_following THEN 'mututal friend'

+ WHEN f2.user\_id IS NULL THEN 'fans'

+ WHEN f1.user\_id = f2.user\_id\_following

+ AND f1.user\_id\_following != f2.user\_id THEN 'follow'

+ WHEN f1.user\_id\_following = f2.user\_id

+ AND f1.user\_id != f2.user\_id\_following THEN 'fans'

+ END AS relationship,f1.user\_id,f1.user\_id\_following

+ FROM follow f1

+ LEFT JOIN follow f2

+ ON f1.user\_id\_following = f2.user\_id

+ ")

relationship user\_id user\_id\_following

1 mututal friend 11 22

2 fans 11 22

3 fans 11 44

4 mututal friend 22 11

5 fans 22 11

6 fans 22 55

7 fans 22 66

8 fans 33 77

9 fans 33 88

10 fans 33 99

11 fans 33 44

微调：

> sqldf("SELECT DISTINCT CASE

+ WHEN f1.user\_id\_following = f2.user\_id

+ AND f1.user\_id = f2.user\_id\_following THEN 'mututal friend'

+ WHEN f1.user\_id = f2.user\_id\_following THEN 'follow'

+ WHEN f1.user\_id\_following = f2.user\_id OR f2.user\_id IS NULL THEN 'fans'

+ END AS relationship,f1.user\_id,f1.user\_id\_following

+ FROM follow f1

+ LEFT JOIN follow f2

+ ON f1.user\_id\_following = f2.user\_id

+ ")

relationship user\_id user\_id\_following

1 mututal friend 11 22

2 fans 11 22

3 fans 11 44

4 mututal friend 22 11

5 fans 22 11

6 fans 22 55

7 fans 22 66

8 fans 33 77

9 fans 33 88

10 fans 33 99

11 fans 33 44

对比LEFT JOIN后的表格检查正确与否：

user\_id user\_id\_following user\_id..3 user\_id\_following..4

1 11 22 22 11

2 11 22 22 55

3 11 22 22 66

4 11 44 NA NA

5 22 11 11 22

6 22 11 11 44

7 22 55 NA NA

8 22 66 NA NA

9 33 77 NA NA

10 33 88 NA NA

11 33 99 NA NA

12 33 44 NA NA

> #given ans:

> sqldf("SELECT DISTINCT CASE

+ WHEN f1.user\_id = f2.user\_id\_following THEN 'mututal friend'

+ ELSE 'one way follower'

+ END AS relationship,f1.user\_id,f1.user\_id\_following

+ FROM follow f1

+ LEFT JOIN follow f2

+ ON f1.user\_id\_following = f2.user\_id

+ WHERE f1.user\_id < f1.user\_id\_following

+ GROUP BY 1,2,3

+ ")

和我的答案的区别：

* 因为只有两种结果，所以判断完mutual即可用ELSE概括剩下的所有情况，不用再罗列condition。
* 因为JOIN ON中间两列相等，那么在判断mutual时，它的condition就只需要头尾两列相等。（只要头尾两列相等一定满足mutual，剩余的情况要么NULL，要么不相等，都不是mutual）
* Where筛选出了11 22 和 22 11 中的其中一种情况，它们都是mutual friend，和顺序无关，属于一种组合而非排列。**经常用WHERE筛选，只留下无关顺序的组合中的一种情况**。
* 我认为这里没有必要加GROUP BY 1,2,3因为GROUP BY所有列和原结果没区别。

relationship user\_id user\_id\_following

1 mututal friend 11 22

2 one way follower 11 22

3 one way follower 11 44

4 one way follower 22 55

5 one way follower 22 66

6 one way follower 33 44

7 one way follower 33 77

8 one way follower 33 88

9 one way follower 33 99

为了解决结果中的11 22既是mutual又是one way的情况，有一个ANY( ) function,可以套在case when对mutual的condition上，表示只要有任意一行满足条件就返回mutual，但是这个在R这里可能用不了。也可以用min( ) function，套在整个CASE WHEN外面，配合最后的GROUP BY 1,2，表示当一对value1,value2有多种result时只返回最小的那个。

#modified ans：

SELECT f1.user\_id as u1, f1.user\_id\_following as follow1,

MIN(CASE WHEN f1.user\_id = f2.user\_id\_following THEN ‘mutual friend’

ELSE ‘one way follower’ END) as relationship # aggregation on group by col1,col2，对combination of col1,col2做聚合。

FROM follow f1

LEFT JOIN follow f2

ON f1.user\_id\_following = f2.user\_id

WHERE f1.user\_id < f1.user\_id\_following #去掉“11 22和22 11”中的“22 11”，只要不同组合，不要不同排列

GROUP BY 1,2 #当col1，col2的combination有行和行之间重复时，relationship取CASE WHEN中的最小值，即mutual friend。

也就是说用min可以使那些既满足mutual friend，又满足one way follow的user pair的最终结果是mutual friend，使那些只满足one way follow的user pair的最终结果还是one way follow。单词的大小是看首字母的字母顺序来比较的，因为m在o的前面，所以m < o。

例题2：LinkedIn面试原题——较难，但是是真实的面试题。主要考点是self-join（找有共同好友的user pairs）；**group by（deduplicate）**；**left join（和原表对比**，看中间结果是否已经在原表中，只取不在原表中的）

Question:How would you design a system to recommend 2nd degree friends to user?

Table:Friend

UID1 UID2

A B

A C

B A

B C

B D

C A

D C

像Facebook中的People You May Know那样，给每个用户推荐他可能认识的2nd degree friends.希望给出的是一个这样的dataframe: uid1(user itself),2nd\_degree\_friend(PYMK)。

思考：1. What kind of users do you want to recommend

A: Have mutual friend but not friend

2. How would you like to show your result?

A: Pair of users. With duplicates?

我的思路：先找到谁和谁可能是secondary friends（他俩有共同好友）；再找出其中已经是好友的，把已经是好友的去掉。

# (1)left join to see mutual friends

> sqldf("SELECT \*

+ FROM friend f1

+ LEFT JOIN friend f2

+ ON f1.uid2 = f2.uid1")

uid1 uid2 uid1..3 uid2..4

1 A B B A.

2 A B B C

3 A B B D

4 A C C A

5 B A A B

6 B A A C.

7 B C C A

8 B D D C

9 C A A B

10 C A A C

11 D C C A

老师在这里使用的是INNER JOIN，这个没差别，因为这个表ON的条件既在左表中又在右表中，LEFT JOIN出来的结果也没有NULL，和INNER JOIN的是一样的，取两表共用的行。老师讲的是因为我**要找的是有mutual friend的pairs，就是要两表有overlapped的才保留**，所以**用INNER JOIN**就可以了。另一个区别：老师最后加了**GROUP BY f1.uid1,f2.uid2，**这个最后结果只有9行，上面我的11行结果中加“.”的是老师的9行结果中没有的。这个是因为我们要看有共同好友的pairs就是**取出join的结果的col1和col4，group by这两行的combination就是对这两行取出的pairs去重**。（老师讲的是col1和col4的pairs会重复出现很多次，why？因为每次他俩有个共同好友，他俩这一对就会出现一次。）意思就是不care他们的共同好友是哪几个，只要有any共同好友就可以取出来这一对把它们定义为可能的secondary friend，group by起到了对这对好友**deduplicate**的作用。妙！

> #extract user pairs with mutual friends-i.e.col1 and col4

> sqldf("SELECT f1.uid1,f2.uid2

+ FROM friend f1

+ LEFT JOIN friend f2

+ ON f1.uid2 = f2.uid1

+ WHERE f1.uid1 != f2.uid2")

uid1 uid2

1 A C

2 A D

3 B C

4 B A

5 B C

6 C B

7 D A

Then I found that there are duplicate “B C”s. So use DISTINCT to remove duplicate.

> #extract user pairs with mutual friends

> sqldf("SELECT **DISTINCT** f1.uid1,f2.uid2

+ FROM friend f1

+ LEFT JOIN friend f2

+ ON f1.uid2 = f2.uid1

+ WHERE f1.uid1 != f2.uid2")

uid1 uid2

1 A C

2 A D

3 B C

4 B A

5 C B

6 D A

到这一步和老师的结果是一样的。但我是先join，得到所有有共同好友的情况（包括自己和自己有共同好友），再用WHERE筛选排除掉“自己和自己有共同好友”的情况；老师是在join on后面就写f1.uid2 = f2.uid1 and f1.uid1 != f2.uid2，在join时排除掉。

Finally, combine the above operations:remove the existed friends from the extracted pairs with mutual friends.从有共同好友的user pairs中去掉已经是好友的，即已经在friend table中的。我把选择有共同好友的user pairs的这段query放入一个CTE中，方便后面判断它们在不在原表中时调用。WHERE中的条件应该是把extracted user pairs**正着和反着**都检查过**不在friend table中**才行。（针对这个问题老师居然直接改了raw table，强行加入了互为好友的反排列，比如她加入了C B, C D等，不过也不排除确实记错题了。）还剩一个最后的小问题就是不会对”ADDA”去重，因为我觉得用之前的col1<col2的方法，在这里或许能得到正确结果，但并不普遍适用，万一你的结果是A D, B A, D A, C B，那用这个方法也会去掉不重复的B A和C B。。。

和老师的另一个写法上的区别是**我用CTE和NOT IN；**而老师是用**括号把上一步**extract user pairs with mutual friends**括起来**，括号外紧跟**重命名为second**，然后**用这个second和原表left join**, second left join friend f3 on second.uid1 = f3.uid1 and second.uid2 = f3.uid2(**col1和col1，col2和col2对应相等**) ，最后**用where选出join后右表为null的列**（右表为null就是那些没在原表中得以匹配的，也就是左表中有而原friend表中没有的，也就是还不是朋友的。达到了这一步的筛选目的。）最后得到一样的ADDA。——也是一个好办法！

#which user pairs in the above results are not friend of each other?

> sqldf("WITH mutual\_CTE (u1,u2) AS

+ (SELECT DISTINCT f1.uid1,f2.uid2

+ FROM friend f1

+ LEFT JOIN friend f2

+ ON f1.uid2 = f2.uid1

+ WHERE f1.uid1 != f2.uid2)

+ SELECT u1,u2

+ FROM mutual\_CTE

+ WHERE (u1,u2) NOT IN friend AND (u2,u1) NOT IN friend")

u1 u2

1 A D

2 D A

收获 **A TRICK**：**当我们想把(已经存在某个表中的)data去掉的时候，可以用一个left join, 然后用“where右表.xxx is null”,只保留那些没join上的，把能join（匹配）上的都去掉。**

1. **Over Partition By ——WINDOW FUNCTION**

**背！**

FUNCTION() OVER (PATITION BY x ORDER BY y ASC/DESC)

Ex. RANK() OVER (PARTITION BY x ORDER BY y DESC)

Apply rank() function

OVER- for the set of rows….

PARTITION BY x – with the same value of x

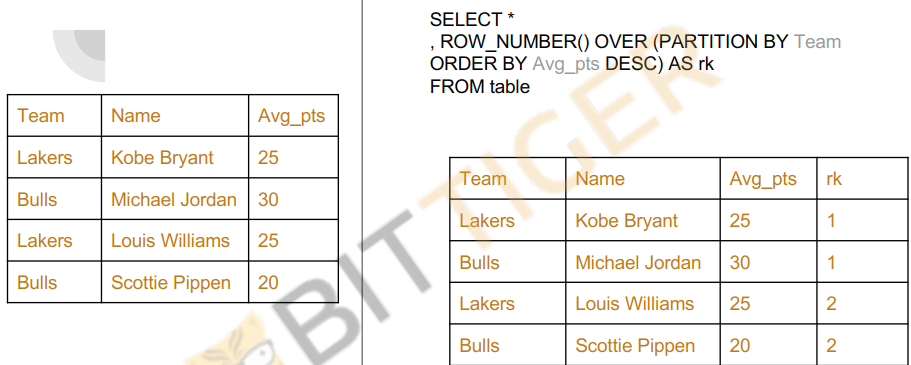
ORDER BY y DESC – in descending order of y

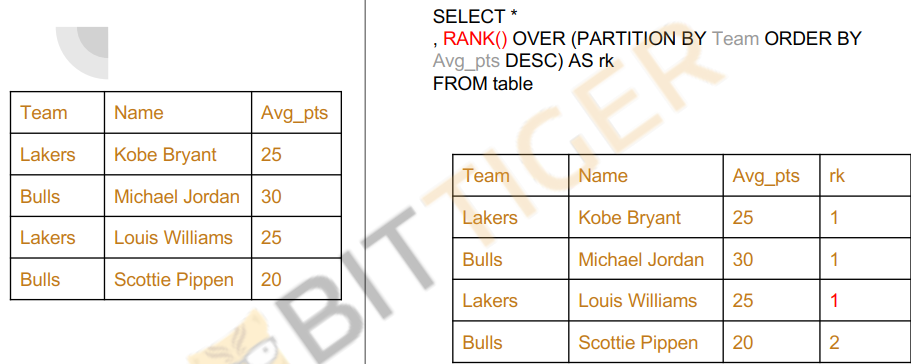
Frequently used functions:

RANK(), ROW\_NUMBER() AND DENSE\_RANK()

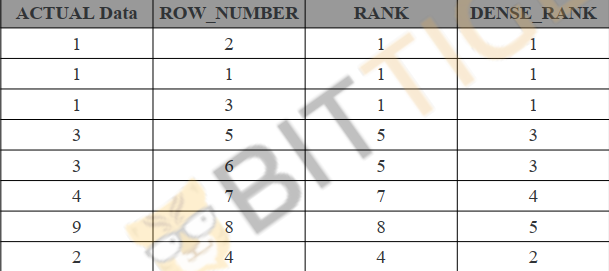
Not available in MySQL，only available in SQL Server AND Oracle。R中无法运行。所以有的面试题不让你用Window Function，也有的面试题专门考Window Function。担心MySQL中写不了达不到训练效果？WINDOW FUNCTION不用怎么训练，因为没啥逻辑上的东西，其实一点也不难。不像JOIN需要细想想，需要大量训练逻辑。Over partition by只要你能记住它怎么写就好。对这个函数，死记即可！

例如：





这三个函数的区别：



ROW\_NUMBER( )也是排序，但当几个数**value相同时**，对这几个数的排序**会随机assign一个排名**。比如1 1 1就会排成第1，第2，第3或者第3第2第1或者第3第1第3等等。然后下一个value的排序就会往下数，是第4。ROW\_NUMBER( )的结果不会出现重复。

RANK( )会重复，1 1 1——并列第一，RANK给出的排名就是第1，第1，第1。下一个大/小的value比如2的RANK就会从第4排起（因为前面的并列第一已经占了3个排名那就没有第2第3了,下一个直接是4），排名结果数字会跳跃。

DENSE\_RANK( )也会重复，1 1 1——并列第一，RANK给出的排名就是第1，第1，第1。下一个大/小的value比如2的RANK就会从第2排起，排名结果数字不跳跃，紧挨着，所以叫DENSE.不管怎么样结果都会有1 2 3名。

例题：WINDOW FUNCTION最常考的例题，(就是之前不太会的那个，第一题第二问，)刷LeetCode会看到很常见。（每个班的前几名的成绩等等。）

这道题不难，只要你知道WINDOW FUNCTION，写起来很straight foward.只要记住这句话syntax写法就行。

有的公司会在第一轮面。LinkedIn第一轮HR面时就面了这道题。

# get the **top 2** salaries within **each department**

> employee\_salary = data.frame(employee\_id = c(11,22,33,44,55,66,77),

+ salary = c(60000, 55000, 130000, 70000, 50000, 90000, 100000),

+ manager\_id = c(33, 33, 77, 44, 44, 44, 77),

+ department = c('legal','legal','legal','sales','sales','sales','executive'))

> employee\_salary

employee\_id salary manager\_id department

1 11 60000 33 legal

2 22 55000 33 legal

3 33 130000 77 legal

4 44 70000 44 sales

5 55 50000 44 sales

6 66 90000 44 sales

7 77 100000 77 executive

> #用WINDOW FUNCTION解决，我的写法：我还没写全因为我现在选出的是每个department中的salary的全部排名，还没选出top 2.

> sqldf("SELECT employee\_id,salary,department,

+ **ROW\_NUMBER()** OVER (PARTITION BY department ORDER BY salary DESC) AS salary\_rank

+ FROM employee\_salary e1

+ ")

WINDOW FUNCTION在R中也不能运行，这里只是记录一下code。

老师的写法：**把WINDOW FUNCTION放在WHERE中，让排名****<=2，就能选出前两名**。

SELECT employee\_id,salary,department

FROM employee\_salary

WHERE **RANK()** OVER (PARTITION BY department ORDER BY salary DESC) <=2

#对每个department做排序；具体是以salary为排序基准；因为要的是top 2 salary，就要逆序排列，从大到小；让rank <=2

#就拿到了top 2 salaries (in each department)。这里用RANK()还是ROW\_NUMBER()都可以。**正序排列时可以写asc也可以啥都不写。TOP N就是让rank或者row\_number <=N.**

**5.Date Time functions**

时间函数每个语言的表达很不一样。比如有的语言中是DATEADD( ),有的语言中是DATE\_ADD( ),所以考的不是很多。很少有公司面试时会专门抠这个syntax的细节。工作中写的时候可以现查，比如prosto和HIVE的具体syntax就很不一样。

**面试有时让你写伪代码，大概意思和功能要写对**。

* 在SQL中可以做日期的加减法：

比如对某个date加上/减去几天，

**DATE\_ADD(date, INTERVAL expr unit)** (Oracle)

E.x. SELECT DATE\_ADD('2010-12-31 23:59:59', -> INTERVAL 1 DAY); 或1 MINUTE，1 HOUR等时间单位

* 计算两个日期的差：帮你算中间间隔几天

**DATEDIFF(expr1,expr2)**

1. x. SELECT DATEDIFF('2007-12-31 23:59:59','2007-12-30’);

比如有的考试题让你算 过去7天内的啥啥，那你就要加一个WHERE DATEDIFF(xxx) < = 7

* 看一个日期是星期几（面试中几乎没见过，工作中比较常用）

DAYOFWEEK(date)

E.x. SELECT DAYOFWEEK('2007-02-03’);

* 对日期转换格式（面试中几乎没考过，工作中会用）

DATE\_FORMAT(date,format)

1. x. SELECT DATE\_FORMAT('2009-10-04 22:23:00', '%W %M %Y'); -> 'Sunday October 2009'

Use Oracle syntax as example <https://docs.oracle.com/cd/E17952_01/mysql-5.6-en/date-and-time-functions.html>

1. **Other advanced topics**

一般面试不会考

* Query Optimization (JOIN or Sub-query?)

——General practice is if your table is large, then using join is highly recommended.除非sub-query return的特别小可以用一下sub-query。

* Functions? (Other PL)?

Self defined functions——面试肯定不考，工作中要用，每个公司每个language有一些自己定义的functions。

* Different Data format? **CAST**

——有不同的data format时可能需要做类型转换，比如JOIN之前需要把**数据类型转换**成两边一样的。比如

CAST big int AS character.

* Different Timezone?

——有时不同时区也需要转换，但都是用的时候再查，面试一般不考这种。

————————————————————————————————————————————

**练习题**

**LeetCode|CareerCup**

**Careercup的题都是各个公司面经，有答案和讨论**

<https://www.careercup.com/page?pid=sql-interview-questions>

SQL\_practice2中1、5题简单可做，比较常见，会讲。

第2题比较难。第4题后半部分比较难。

第3题超级难，属于拔高题，有兴趣自己研究，可能不会讲。

**SQL Interview**

**SQL interview questions**

1. SQL coding (**90%**) sometimes **within a case study** -- Understand problems + Coding

2. Knowledge question (sometimes embedded in coding). -- 小公司或entry level position或电话面试可能问

SQL怎么面？大公司的话，第一轮tech interview至少会问一个SQL问题。在Onsite interview时会有一轮专门面SQL，或是两轮Technical interview中各有一部分SQL。整个Interview从头到尾至少会面两个SQL题。小公司不一定。如果面试的是BA的职位，或者DS，只要是这种做数据比较多的至少会面两个SQL题。90%的情况是在case study里面，不直接考你SQL题，通常给你一个情景，一个case，给你一些data，问你怎么分析这个data，怎么做什么什么事情。当问到某个具体情况的时候才会让你写个SQL。||还有一种情况是直接问你knowledge question，比如问你case when是什么，或者直接问where 和having有什么区别，或者问你Left join和inner join有什么区别，小公司喜欢问，因为case study准备起来比较麻烦。或者是Entry level的position，要求没有那么高，他会直接问你，看你懂不懂sql最基本的东西。还有像HR，第一轮面试时问你基本知识点。

基本上准备coding题就可以了，如果coding题过关，知识题也不会差到哪去。准备一下常见的一些知识题就可以了。不用背特别多的sql 数据库的东西，这些考得倒不是很多，比如，如何insert，怎么alter一个table，怎么给一个table加一些备注等等，面data engineer可能会问，DA/BA/DS不怎么会问这种纯知识性的问题。

**Important Evaluation Factors:**

**● Logic (the key!!!!)**

逻辑是最关键的，特别是Join，各种不同的Join,在考察的过程中它主要是看你逻辑是否清晰，这是最关键的。

● Understand real problem & translate to a data problem

在一个case study中要能理解真实的问题，知道这是一个什么样的情景，公司的什么data（回顾LinkedIn面试题，用户互相follow的data），然后你再把这个data translate到一个SQL的问题。这种understand，有时它给的题目就是一个business的问题，比如我想找出一个月之内表现最好的一个部门——那你就要去想，“表现最好的部门”是什么意思？sales最高？还是说增长最快？这种问你你要自己去把它translate到一个data的problem。

还有题目会问你，如果我想做一个事情,需要什么样的data？然后再写sql。比如你要给用户recommend他的朋友听过的歌，那这个data应该保存成什么格式？那我可能要想，朋友听过的歌——保存成，a user, a song, how many friends of him/her have been heard of?还是一个user和另外一个user，他们共同听过的歌 etc。你要先去define这个data的structure，你有几个column，每个column的意思是什么？后面才会说，按照你这个思路，我这有两个tables，如何写sql得到我想要的data。Understand a real problem->define a schema->translate to a data problem.

● **Communication**

BA/DS最重要的技能之一，考察贯穿整个interview的全程，在SQL面试过程中也在考察你的交流。**要边写边交流**，解释你的query。如果一直有解释你的思路，保持良好的communication，就算query有错或者最后没写出来，communication也会是加分项。

Not very important but good to have:

● Format——越清晰越好，不要写一长串。

● Bug Free——最好没错，可以直接跑通。但如果（能通过面试官提示的哪里不太对）或自己改正过来也可以。

● Query optimization——有的人写的query跑起来比较慢，如果**时间允许你可以和面试官说我的query虽然是对的但是可能还能优化一下提高效率**，如果能自己优化代码，也是一个加分项。

**How to approach SQL interview questions**

**Step 1**, make sure you are 100% clear what is the question.

● “Give me the distribution of app acquisition over time”——他想让你得到的是什么样的data？如果建立一些column，你的output是什么样的columns？

当你在面试中遇到这样的一个问题，**你首先要问一些问题去clarify**，

比如你要问，什么是app acquisition? 可能会告诉你number of users downloaded this app, 或者number of users sign up/register

问Time frame是多久？可能会告诉你，看data，我的data覆盖了有多久的那我就看多久的。

问domestic还是global的？可能会告诉你，不管，就是看overall的

问time是by day还是by week？可能会告诉你，by day。

当得到一些具体的要求以后，再思考**你的output table应该写成什么样？**

答案——ouotput table包括两个列比较好：day of download,number of downloads on that day.

按什么排列比较好? 因为要看over time，还是按天排列比较好，而不是按下载数的asc/desc排序，可以看到数 据随 着时间的变化。

distribution是看cumulative的还是non-cumulative的？还是单看每天的绝对值（不累计）比较容易看到趋势。

● “Give me the performance of YoY revenue growth”

这部分是大家很容易忽略的，本应该在这里花大量时间来clarify，要明确地知道，最后想要的结果大概长什么样。脑海中对要得到的结果要大概有框架，有清楚的概念，不然写着写着就乱了。

Ask **clarifying** question if you are not sure! (A lot of time you are supposed to)

**Step 2**, build the join in head, and if too many, build temp table.

拿到问题想得到的结果，先想一下我需不需要写个JOIN?我**需要哪些INFO**？如果需要写JOIN，需要哪些data?比如在之前的练习题中，我需要category name 和 how many category instances in the corresponding category name?那么就要想，category name**从哪里来**？

category instances从哪里来？

脑海中大概想好要做的self join。

如果有很多的join要做，或者join很复杂，可以用temp table或者CTE。

**Step 3**, write the SQL (better if you can at the same time **talk through it**)

当你大概知道了需要用什么和什么JOIN之后（on什么），你可以开始落笔写这个query。（为什么不要一上来就写，当你还没有清晰的逻辑时很容易写错，浪费时间而且写了又擦可能会打乱思路。）

在第2和第3步，想和写的过程中，要talk through it。要和面试官保持交流。“要得到某个东西，我需要table1的xx information 和 table2 的 xxx information，所以我需要对这两个table进行xx join。我认为要做left join，因为我需要table1 所有的行和table2共有的行。我觉得要ON id来join...”

纸上写，白板写和电脑写都有可能。如果是电话面试，可能会给你一个链接，google doc之类的，real time的写。

On site通常是白板上写。平时都在电脑上练习，建议**去面试前特别是onsite面试前，找一个白板，在白板上试一下**，因为有的人竖过来就突然不会写了。（让面试官给你找张纸比较wierd，你先在纸上写，面试官也看不到你在写些什么，不能很好的communicate你写的过程。）如果平时只习惯在纸上写，面试之前一定要竖过来在白板上练习一下。

**Step 4**, double check (table prefix, distinct, left/inner/right join, etc.). Think about ways to optimize it

从上到下读一遍，小的细节是否写错？检查一些东西，比如table prefix——a.id/b.id 这种也很容易写错，从哪个表中选出的对应的行？是不是要写distinct，如果需要是否忘记写？Join，group by,这些syntax的顺序有没有写错？

有没有更优化，效率更高的写法？

总结步骤：**想**，问题问什么，想得到什么——**想**，需要哪些数据，要不要**join**，join on什么，想差不多后**讲**——**写**，边写边**讲**——**检查**细节，补充优化。

**What to notice**

Take your time! (A bit slow is fine, quick but buggy code ~= nothing)

不要太着急，不要一开始就下笔写。先问几个问题，然后要想好再慢慢写，尽量避免写了又擦，显得你逻辑特混乱。

写完之后还要double check，看结果是不是对的。

还有人埋头写，完全不会交流讨论，其实面试官对你最后的结果并不是那么看重，还是看重你的thinking process，要清晰地展示你的思路。也能帮助你自己理顺思路&争取时间。如果你跑偏了，也能给你提示把你拉回来，不然如果在错的方向上全写完了可能也没时间纠正了。

● Clarify, summarize

● Fine writing

● Double check when you finish

Make sure you talk through your thinking process

● Help yourself to think

● Help interviewer to understand and help you if you are going in a wrong direction

对着同学/镜子在白板上练习，至少一次（体验还是很不一样的）

**Steps in Syntax**

1. FROM

2. JOIN

3. WHERE

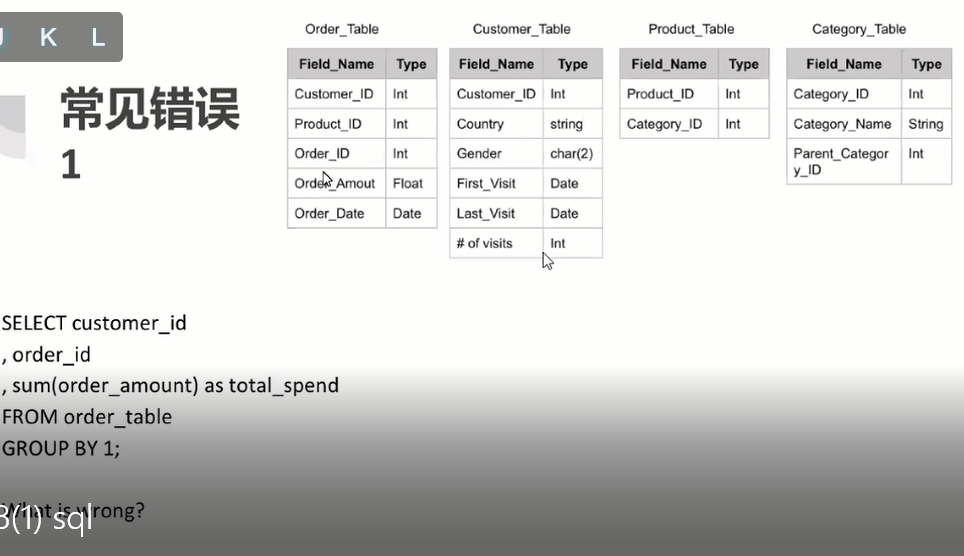
4. SELECT, GROUP BY, HAVING

5. ORDER BY

写的顺序和呈现出来的顺序其实不一样，一般先写的是FROM，因为你最先知道要从哪个table开始，从这里提取数据，然后把JOIN写上。然后再想是不是要filter掉什么东西，加上WHERE.基本上join的部分写完之后才会加上SELECT，GROUP BY这些。相当于**从中间往上下两头写**。最后才加HAVING（having 是aggregate之后的）, ORDER BY（最后最后看是否需要排个序）这种。

一上来就写SELECT还没概念，不知道具体名字（万一后面改了），也不知道列名在哪（join完在哪个table）。有可能你先把SELECT写了，后面join完名字/位置变了，上面却忘记改了，就会出错。先把SELECT空下。

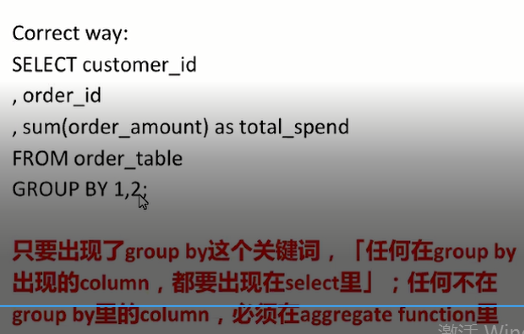
**SQL常见的错误**



我们可以看出这个query想要做的就是想知道**每个用户在每一单中**总共花了多少钱。（Order\_amount应该指的是钱数。）这个query的错误在哪里？

错在group by。

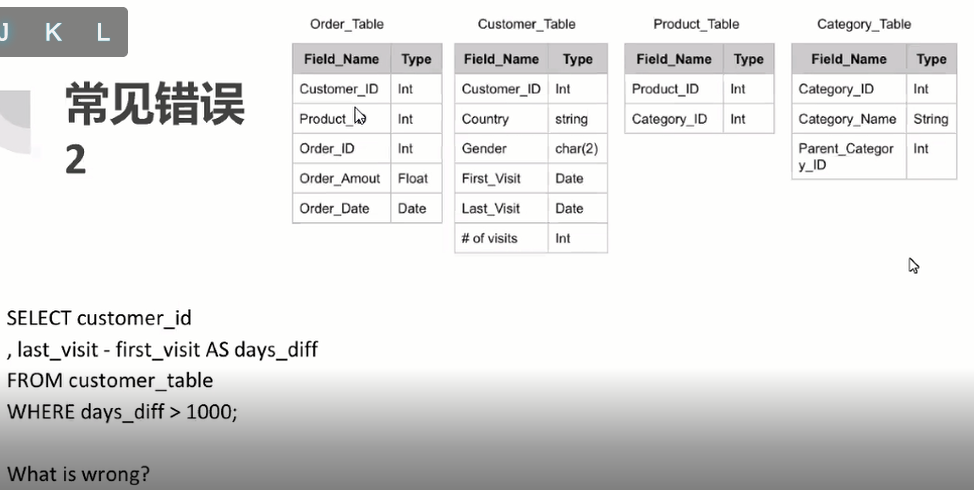
应该group by 1,2



任何不在group by中的column都必须出现在aggregation function（Sum count max min）中。

比如这个例子，select 了三列，只aggregate（sum）了第三列，那么就**要把没有放在aggregation function 中的前两列都放在group by**中。所以先写FROM, 再写select 和 group by，这两个一起写就不容易出错，你知道自己select了什么cols,就会知道该group by什么cols。（不然早早写完select，等到group by都忘了前面有啥）

如果这里要看的是一个customer所有的order的total amount，那就把select中的order\_id去掉。最后group by 1.

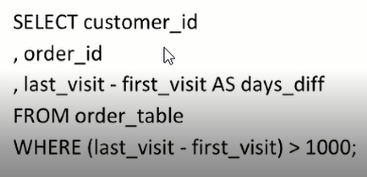


一开始我以为他这么写是想要找出第一次visit后超过1000天没来过的customer\_id

但是这么想不对，因为你怎么知道这个customer中间来没来过？只是数据提供的是第一次登陆的时间和最后一次登录的时间，中间可能来过很多次，这里没显示而已。所以这个query可能想看的就是user retention。**有哪些超过1000天还在使用的老客户。留存时间大于1000天的。**

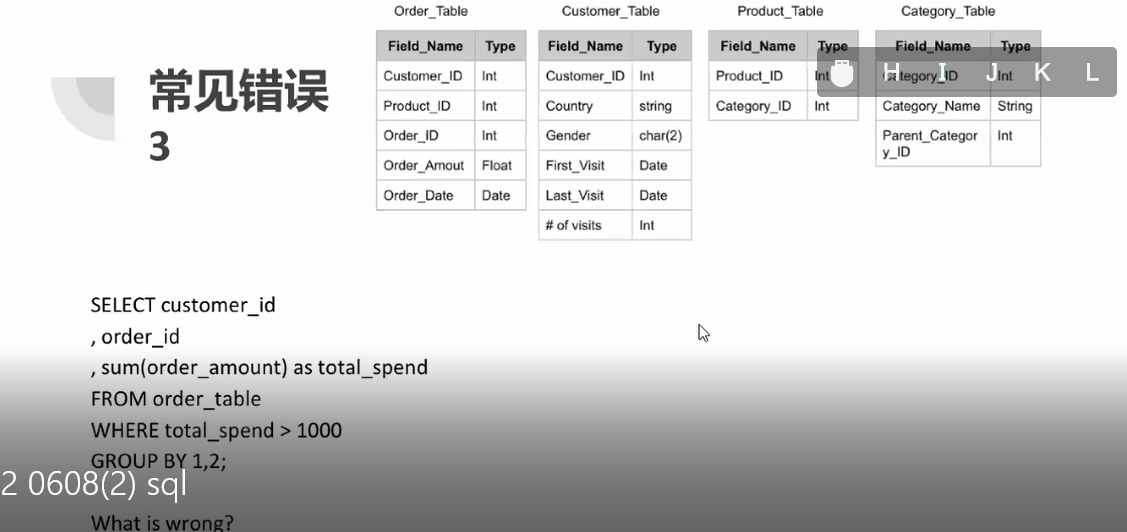
他这么写的错误是——**WHERE是作用在raw data上**的，raw data上并没有days\_diff这个column。因为days\_diff是在SELECT这里定义的重命名，重命名发生在最后，当执行WHERE时，它就是从 customer\_table这个raw data中寻找，这时重命名还没发生。所以会报错。

正确写法：WHERE中要使用raw data。



这也是为什么最好先写FROM和WHERE,后写SELECE,这样就不容易把SELECT里定义的column拿来用。

在WHERE中不能像GROUP BY中那样用1,2代替SELECT中的column，因为同理，在执行WHERE的时候，还没有执行SELECT.它并不知道你说的这个1是指哪个col。

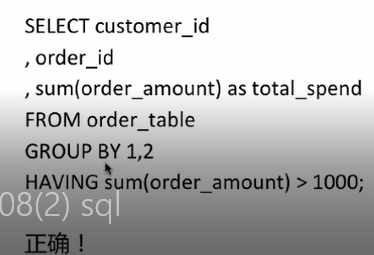


这个写法要做什么：找出购物花费超过1000刀的customer\_id和order\_id。

错误：WHERE这一句。不光是WHERE后面用了SELECT中重命名的total\_spend,就算你改成WHERE sum(order\_amount) > 1000也不对，因为这里是**对aggregation写条件，要用HAVING**.HAVING——condition after aggregated

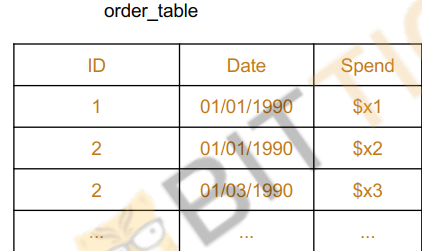
WHERE是作用在一行一行的raw data上的，这种**做过任何aggregation的都不能写在WHERE中，要用HAVING**。

并且**还要注意顺序，HAVING要放在GROUP BY的后面**。



**Coding Practice**

SQL question 1: **Get 1000th highest spend value**



R中数据table generation code：

id = seq(1:100)

date = sort(rep(c('01/01/1990','01/02/1990','01/03/1990','01/04/1990','01/05/1990'), 20))

s = rnorm(100, 50, 10)

order\_table = data.frame('id'=id, 'date'=date,'spend'=s)

产生的order table：

> order\_table

id date spend

1 1 01/01/1990 47.15422

2 2 01/01/1990 51.19077

3 3 01/01/1990 44.06002

4 4 01/01/1990 41.70763

5 5 01/01/1990 31.11890

6 6 01/01/1990 38.96892

7 7 01/01/1990 50.30811

8 8 01/01/1990 48.90105

9 9 01/01/1990 60.46756

10 10 01/01/1990 47.81280

11 11 01/01/1990 63.16865

12 12 01/01/1990 40.20380

13 13 01/01/1990 35.81952

14 14 01/01/1990 36.60253

15 15 01/01/1990 41.55693

16 16 01/01/1990 44.20529

17 17 01/01/1990 59.28385

18 18 01/01/1990 67.76497

19 19 01/01/1990 65.91701

20 20 01/01/1990 46.79196

21 21 01/02/1990 55.24342

22 22 01/02/1990 63.56394

23 23 01/02/1990 53.05408

24 24 01/02/1990 52.76977

25 25 01/02/1990 40.81369

26 26 01/02/1990 52.62974

27 27 01/02/1990 51.85354

28 28 01/02/1990 55.61598

29 29 01/02/1990 53.38123

30 30 01/02/1990 48.35190

31 31 01/02/1990 47.91731

32 32 01/02/1990 25.98856

33 33 01/02/1990 66.16666

34 34 01/02/1990 55.23364

35 35 01/02/1990 52.36989

36 36 01/02/1990 49.89957

37 37 01/02/1990 45.01514

38 38 01/02/1990 39.70236

39 39 01/02/1990 48.64075

40 40 01/02/1990 45.70131

41 41 01/03/1990 48.80593

42 42 01/03/1990 37.50866

43 43 01/03/1990 46.79797

44 44 01/03/1990 58.16678

45 45 01/03/1990 44.99466

46 46 01/03/1990 51.67120

47 47 01/03/1990 59.02782

48 48 01/03/1990 40.08830

49 49 01/03/1990 48.51319

50 50 01/03/1990 53.20168

51 51 01/03/1990 65.89476

52 52 01/03/1990 55.16083

53 53 01/03/1990 52.11558

54 54 01/03/1990 48.65703

55 55 01/03/1990 47.19470

56 56 01/03/1990 46.91967

57 57 01/03/1990 62.84631

58 58 01/03/1990 61.37439

59 59 01/03/1990 65.95360

60 60 01/03/1990 55.63600

61 61 01/04/1990 55.27313

62 62 01/04/1990 35.89967

63 63 01/04/1990 62.70995

64 64 01/04/1990 50.96257

65 65 01/04/1990 53.97818

66 66 01/04/1990 48.96520

67 67 01/04/1990 64.16811

68 68 01/04/1990 64.09404

69 69 01/04/1990 47.00670

70 70 01/04/1990 58.01699

71 71 01/04/1990 47.36950

72 72 01/04/1990 34.15296

73 73 01/04/1990 55.02462

74 74 01/04/1990 42.93431

75 75 01/04/1990 54.51452

76 76 01/04/1990 46.06684

77 77 01/04/1990 53.56469

78 78 01/04/1990 60.75661

79 79 01/04/1990 62.03813

80 80 01/04/1990 55.20928

81 81 01/05/1990 50.31095

82 82 01/05/1990 58.50361

83 83 01/05/1990 47.78713

84 84 01/05/1990 47.43105

85 85 01/05/1990 46.44097

86 86 01/05/1990 55.98110

87 87 01/05/1990 44.83213

88 88 01/05/1990 52.84965

89 89 01/05/1990 29.57704

90 90 01/05/1990 61.99300

91 91 01/05/1990 42.95699

92 92 01/05/1990 42.34838

93 93 01/05/1990 60.63262

94 94 01/05/1990 61.61862

95 95 01/05/1990 54.25647

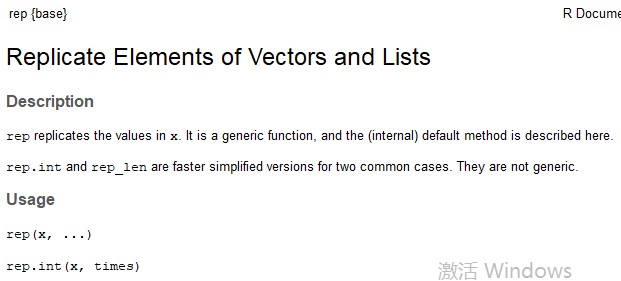
96 96 01/05/1990 54.18078

97 97 01/05/1990 44.77359

98 98 01/05/1990 36.30928

99 99 01/05/1990 49.58421

100 100 01/05/1990 41.88143



rep(x, times = 1, length.out = NA, each = 1)

rnorm(n, mean = 0, sd = 1)

如果先让你找出(所有order中)最高的spend：

SELECT MAX(spend),id

FROM order\_table

~~GROUP BY 2,3~~

因为要找出全部order中最贵的spend，不需要group by id，group by id的话会找出每个ID中最贵的spend。

SELECT MAX(spend),id

FROM order\_table

这个也不对，因为SELECT后面的所有col要不就全都是aggregation，要么就对那些不是aggregation的col做group by。

**没法mix一个raw data的column和一个aggregate的column，还不进行group by的。**

这里要么就写

SELECT MAX(spend),id

FROM order\_table

GROUP BY id

要么就把id去掉，不SELECT

SELECT MAX(spend)

FROM order\_table

这两种写法在语法上都对了，那么你根据题目要求，应该写哪种？第二种√。这里不需要进行group by。

正确答案1：SELECT MAX(spend) FROM order\_table

正确答案2：SELECT \* FROM order\_table ORDER BY spend DESC LIMIT 1

**Get 1000th highest spend value**

SELECT \* FROM order\_table ORDER BY spend DESC LIMIT 1000

这个写法不对，这个返回的是前1000个。而题目要的是第1000个。

TOP 和 LIMIT应该不能mix（别人的创意写法应该不对）

我的写法：用CTE把最高的前1000行写成一个单独的表再选择其中最低的就是第1000。

也可以直接用sub-query把前1000行括起来。

WITH CTE\_1000 AS (SELECT \* FROM order\_table ORDER BY spend DESC LIMIT 1000)

SELECT \*

FROM CTE\_1000

ORDER BY spend ASC

LIMIT 1

R中数据只有100行所以返回第10个最大的。

#first 10 highest

> sqldf("SELECT \* FROM order\_table ORDER BY spend DESC LIMIT 10")

id date spend

1 18 01/01/1990 67.76497

2 33 01/02/1990 66.16666

3 59 01/03/1990 65.95360

4 19 01/01/1990 65.91701

5 51 01/03/1990 65.89476

6 67 01/04/1990 64.16811

7 68 01/04/1990 64.09404

8 22 01/02/1990 63.56394

9 11 01/01/1990 63.16865

10 57 01/03/1990 62.84631

> ## the 10th highest

> sqldf("WITH CTE\_10 AS (SELECT \* FROM order\_table ORDER BY spend DESC LIMIT 10)

+ SELECT \*

+ FROM CTE\_10

+ ORDER BY spend ASC

+ LIMIT 1")

id date spend

1 57 01/03/1990 62.84631

GIVEN ANSs：

1) Use TOP / LIMIT keyword

SELECT \*

FROM(

SELECT Spend

FROM order\_table

ORDER BY Spend DESC

LIMIT 1000)

ORDER BY Spend LIMIT 1;

用TOP的话就是把它放在上面，SELECT TOP 1000 spend FROM...后面一样（不用LIMIT）。

What can be better?——format建议**一个关键词占一行**，而不要整个写得非常长一行，不容易读，易乱。让人不想读。

括号中的subquery要空格。缩进。能清晰看到sub-query

关键词大写，table相关的小写。

SELECT spend

FROM (

SELECT spend

FROM order\_table

ORDER BY 1 DESC

LIMIT 1000

)

ORDER BY 1

LIMIT 1;

1. Use rank() over keyword 用WINDOW FUNCTION

SELECT \*

FROM (SELECT \*

, RANK() OVER (ORDER BY Spend DESC)——这里如果想看每个ID的spend就写partition by，想看所有的就不写，直接ORDER BY。因为定义rk是在subquery中，所以运行外层的WHERE时，FROM的source中已有rk这个column了，就可以直接调用rk。

AS rk

FROM order\_table)

**WHERE rk = 1000**;

Any problems with this code?

What if you have records with **same spending**?

用Rank()可能出现的问题：并列的情况可能会跳过**，有的rank可能就不存在**，可能从1 1 1直接跳到3.所以就可能没有第1000这个rank，比如直接999->1003，**找不到rank = 1000的情况**，会报错。这种情况下应该用row\_number( ).

2) Use row\_number() over keyword

SELECT spend

FROM (

SELECT spend

, row\_number() over (order by spend desc) as spend\_rank

FROM order\_table)

WHERE spend\_rank = 1000;

Need to clarify with interviewer.

**用row\_number() 才能保证肯定有spend\_rank = 1000的那一行**,因为row\_number是把并列的值随机assign一个排名，保证不会有重复排名，所以一定会有1000.那为什么不用dense\_rank( )？Dense\_rank( )也一定会有1000，不会跳过一个排名。但是dense\_rank()给你的rk = 1000的值可能并不是真正的order上的第1000名，因为如果前面有重复的，本该占据一个排名的，但dense\_rank把所有重复的都算在一个排名里，几个人占一个坑。并且下一个排名也不浪费也会占满。rk = 1000可能order上>1000了。用dense\_rank=1000返回的是第1000个unique高的值，而不是第1000个高的值，这跟我们想要的答案就不一样了。

“选出第1000个高的值”——用row\_number

“选出第1000个unique高的值”——用Dense\_rank

Rank（）和row\_number()这两个函数，坑和萝卜的数量相等，但rank()会有空坑，当同样大小的萝卜被放到一个坑里的时候，后面就会空一个坑。row\_number( )不空坑，一个萝卜一个坑，就算俩萝卜一样大，也不往一个坑里放，拆散了随机排个先后。

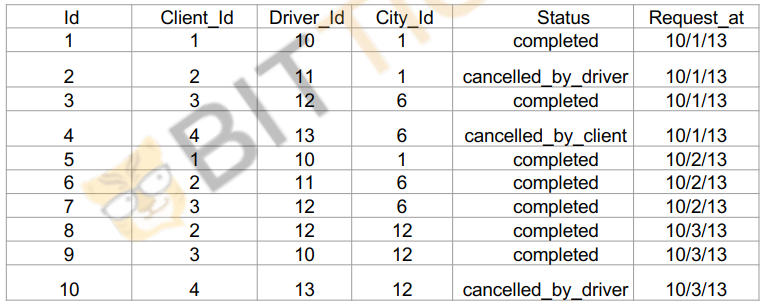
Dense\_rank( )坑和萝卜的数量可能不相等。坑可能少于萝卜。不会空坑。相同大小的萝卜就都放一个坑里，但后面紧接着的坑也不会空下。接着放下一个排名的萝卜。

SQL question 2：属于case study

给情景，给real data，根据这个data解决real问题

The Trips table holds all taxi trips. Each trip has a unique Id, while Client\_Id and Driver\_Id are both foreign keys to the Users\_Id at the Users table. Status could be (‘completed’, ‘cancelled\_by\_driver’, ‘cancelled\_by\_client’).

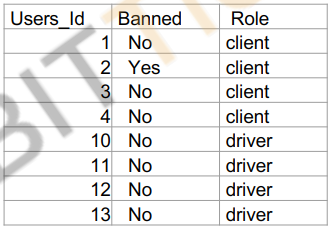
Table 1: trips



The Users table holds all users. Each user has an unique Users\_Id, and Role is one of (‘client’, ‘driver’, ‘partner’).

Write a SQL query to **find the cancellation rate** of **requests made by unbanned clients for each day**

Table 2: Users



* 先思考：要用到哪些信息？（需要用到哪些列，分别在哪个表里，要写JOIN吗，写的话，什么join）

要用到client\_id,status,request\_at,|| users\_id,banned. 信息在两个表中，需要写join

Left join和inner join都可以。因为表二的描述是The Users table holds all users.基本上table1中有的user\_id,table2中也都会有，所以可以用inner join。但是用LEFT JOIN也不会算错，保险起见可以也保留一下左表全表。但用INNER JOIN比较快。

SELECT COUNT(t.id)

FROM trips t

JOIN Users u

ON t.Client\_id = u.user\_id

画了一下JOIN后的中间结果，发现**表2的作用只是额外匹配上一堆client是否被banned的状态信息**，表1 JOIN表2并不会减少行数，只是把表一变宽，每一行都多出了client有无banned的信息（用client\_id join就是看表一中列出的client是否被banned，如果用driver\_id join就是看表一中列出的driver是否被banned）表一中的任何信息都不会被filter掉，比如cancelled by driver的记录依然在JOIN后的结果中。——这个最好在脑海中想清楚，对后面怎么计数至关重要。

* 然后思考：怎么计算cancellation rate?—— Number of Cancelled trips By clients/total number of trips of each day（面试的时候讲出来，你的思路。Talk through it）

计数每天有多少trips：

SELECT COUNT(1)

FROM trips

GROUP BY request\_at

计数有多少trips被unbanned clients取消掉：

SELECT SUM(IF(status = ‘cancelled\_by\_client’ AND banned = ‘No’,1,0))

FROM trips t

JOIN Users u

ON t.Client\_id = u.users\_id

GROUP BY request\_at

分子分母一起：

SELECT SUM(IF(status = ‘cancelled\_by\_client’ AND banned = ‘No’,1,0))/COUNT(1) AS cancellation\_rate

FROM trips t

JOIN users u

ON t.client\_id = u.users\_id

GROUP BY request\_at

计算完cancellation rate之后记得把日期也select上，人家要的是每天的cancellation rate。要能看到哪一天对应多少cancel。

给最终结果加上日期： MY FINAL ANS:

SELECT request\_at AS day,

SUM(IF(status = ‘cancelled\_by\_client’ AND banned = ‘No’,1,0))/COUNT(1) AS cancellation\_rate

FROM trips t

JOIN users u

ON t.client\_id = u.users\_id

GROUP BY request\_at

不过R里面好像不能跑：

Error in result\_create(conn@ptr, statement) : no such function: IF

如果能跑，返回的应该是0.还是不对。这里有个little trick——分子是SUM的结果，是一个整数。分母是COUNT的结果，也是一个整数。SQL中两个整数相除，返回的还应该是整数。比如本该cancellation rate是0.5，要强制转换成整数类型，只取整数部分（去掉小数部分），最后就变成了0. 这时怎么处理？把分子中SUM的内容变成1.0,0.0，这样分子就会是小数类型，结果就可以是小数类型了。

答案更正：

SELECT request\_at AS day,

SUM(IF(status = ‘cancelled\_by\_client’ AND banned = ‘No’,**1.0,0.0**))/COUNT(1) AS cancellation\_rate

FROM trips t

JOIN users u

ON t.client\_id = u.users\_id

GROUP BY request\_at

TSQL是整除，可以用CAST(COUNT(1) AS double)——把整数类型转换成小数类型。其他软件可以直接除两个整数得到小数。

其实只有分子需要看JOIN的结果因为分子的计算对banned的状态有condition。分母应该是每一天所有order，只看左表就够了。但是也可以看JOIN后的结果,因为左表的按日期group by的行数计数和JOIN后的表按日期group by的行数计数是一样的（毕竟两个表的行数没有区别。JOIN只是把左表横向变宽。）所以对分母来说COUNT这两个表中的谁都行。为了统一和分子用一个表，就用JOIN后的计数。（小疑问：cancelled by driver算不算在分母里？我目前觉得应该算）

错误思路：一开始我把分子分母分开算，分子是把两个表JOIN后用WHERE来筛选出 t.status = ‘cancelled\_by\_client’ AND u.banned = ‘No’，计数符合条件的t.id的行数。而分母是单看左表，计数t.id的行数，（两者都需要group by request\_at，按日期分类。）这样没法直接定义cancellation rate.

分子需要用到WHERE来限制条件；分母要对总体计数就不能限制任何条件。不能把它们放在一个query里，仿佛陷入了僵局。

~~SELECT COUNT(t.id) as numerator~~

~~FROM trips t~~

~~JOIN Users u~~

~~ON t.Client\_id = u.user\_id~~

~~WHERE~~ ~~t.status = ‘cancelled\_by\_client’~~ **~~AND u.banned = ‘No’~~**

~~GROUP BY request\_at~~

~~SELECT COUNT(t.id) as denominator~~

~~FROM trips t~~

~~GROUP BY request\_at~~

最理想的写法是让分子分母在每个关键字下都是统一的，（FROM一样的data source）这样才能把它们统一写在一个SELECT中。如何统一？关键是要去掉计算分子时想用到的WHERE.计算分子只能用COUNT（符合条件的行）吗？只有WHERE能加condition吗？是否可能**把条件上移至SELECT中，让分子分母FROM的部分统一**呢？答案是可以的。**用IF就能实现把condition放在SELECT中**。

各退一步，分母本没有那么多要求，不需要计数JOIN后的表格，但是用JOIN后的表格结果也一样。

分子可以先不用WHERE来筛选符合条件的行，而是直接FROM JOIN后的原始表格，在SELECT中通过IF计数符合条件的那些。记住这个神操作——**SELECT SUM(IF(condition,1,0)) .把value if yes设成1，value if no设成0，IF外面套一层SUM，就能实现在SELECT中，对符合条件的记录进行计数操作。（**如果对计数结果的数据类型有要求，可以调整value if yes/no的数据类型。比如如果你想要SUM的结果为小数就写成**SUM(IF(condition,1.0,0.0))）**如果想用COUNT就不能1,0，因为0也会被计数。**SELECT COUNT(IF(condition,id,NULL))**——如果想用COUNT就把value if yes设成如果条件满足，你想计数的那一列的列名；value if no设成NULL。可以不写，自动就是NULL。COUNT函数在COUNT列明时不会把NULL计入。这种情况习惯用SUM比较多。

同学出现的问题：

**WHERE u.banned = ‘No’**这个条件也可以用AND加在ON后面，表示**在INNER JOIN的时候就把banned = ‘Yes’ filter掉**了。有时可以直接用 ON xxx **AND xxx**来取代WHERE.——用AND代替WHERE只有在INNER JOIN中可以吗? 在有的SERVER里，INNER,LEFT,RIGHT JOIN都可以用，但有的server是只有LEFT可以用。比较tricky，不放心的话还是改成WHERE.但是INNER JOIN都可以用。所以在INNER JOIN时一般会改成AND。

SQL中WHERE=什么的判断语句写一个等号即可，不像python和R中判断语句要写两个等号。

当判断字符串是否是xxx时记得加引号。

写subquery——where id not in (select.......) 最好只在**not in后面的这个list非常短的时候写sub-query**，不然不efficient，会比较慢。

* 对题目Write a SQL query to find the **cancellation rate of requests made by unbanned clients** for each day的理解不同会写出不同的答案：

老师给的答案是这样的：

SELECT request\_at AS day,

SUM(IF(**status like 'cancel%'**, 1.0, 0.0))/COUNT(1) AS cancellation\_rate

FROM trips t

JOIN users u

ON t.client\_id = u.users\_id

AND u.role = 'client'

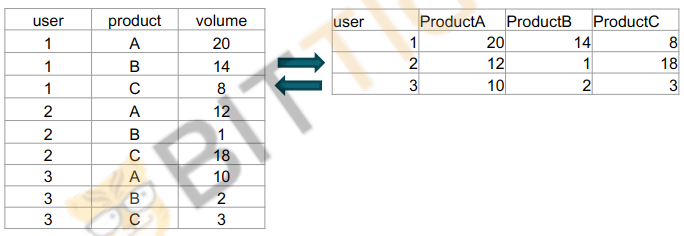
**AND u.banned = 'No'**

GROUP BY request\_at

她对题目要求的理解是：求在那些**由unbanned clients发起的request中，被cancelled掉的trips。**（可能被client cancel也可能是被driver cancel,所以**IF中的逻辑判断是like语句**——只要有”cancel啥啥”就计数，不管是谁cancel的都要计入）；所有由unbanned clients发起的trips是分母，不看那些被banned的client request的trips。所以JOIN的时候就filter掉了banned clients的相关records。**这些到底怎么理解一定要在写之前和面试官confirm**。

SQL question 5:

Can you use SQL to transpose a table without pivot/unpivot function?



SQL中有个FUNCTION，叫做transpose(),还有个FUNCTION叫做pivot().也就是说SQL里有直接做transpose的function。

这个题就是让你不要用pivot/unpivot function来实现这个功能。Interesting question.在面试题题库中比较常见。

* 先看怎么从左边到右边？假如你在面试，先不要下笔写，先想想用什么function可以实现？从3个column变成4个column？这是一个SQL中特别常见，常用的trick，从很多行的table变成一个很多列的table：

——长表变宽表，要根据（某些column上的）condition定义一个（多个）新的column，要**用CASE WHEN或IF来增加column**。

SELECT user,

CASE (“product”) WHEN “A” THEN volume END AS ProductA,

CASE (“product”) WHEN “B” THEN volume END AS ProductB,

CASE (“product”) WHEN “C” THEN volume END AS ProductC

FROM table

> p = rep(c('A','B','C'),3)

> u = c(1,1,1,2,2,2,3,3,3)

> v = sample(0:20,9)

> d = data.frame('user'=u,'product'=p,'volume'=v)

> d

user product volume

1 1 A 15

2 1 B 7

3 1 C 16

4 2 A 11

5 2 B 6

6 2 C 17

7 3 A 19

8 3 B 3

9 3 C 2

> sqldf("SELECT user,

+ CASE(product) WHEN 'A' THEN volume END AS ProductA,

+ CASE(product) WHEN 'B' THEN volume END AS ProductB,

+ CASE(product) WHEN 'C' THEN volume END AS ProductC

+ FROM d")

user ProductA ProductB ProductC

1 1 15 NA NA

2 1 NA 7 NA

3 1 NA NA 16

4 2 11 NA NA

5 2 NA 6 NA

6 2 NA NA 17

7 3 19 NA NA

8 3 NA 3 NA

9 3 NA NA 2

或者加上ELSE 0

> #WIRH else

> sqldf("SELECT user,

+ CASE(product) WHEN 'A' THEN volume ELSE 0 END AS ProductA,

+ CASE(product) WHEN 'B' THEN volume ELSE 0 END AS ProductB,

+ CASE(product) WHEN 'C' THEN volume ELSE 0 END AS ProductC

+ FROM d")

user ProductA ProductB ProductC

1 1 15 0 0

2 1 0 7 0

3 1 0 0 16

4 2 11 0 0

5 2 0 6 0

6 2 0 0 17

7 3 19 0 0

8 3 0 3 0

9 3 0 0 2

这时经常出现的一个问题是，在长表转换成宽表的中间过程会出现很多0(如果写了ELSE 0的话)。这时就需要做aggregation。比如用MAX或者SUM来取出实际的数，去掉多余的0.把同个用户对应的很多列变成一列——GROUP BY user

> # aggregate to remove the 0s

> sqldf("SELECT user,

+ MAX(CASE(product) WHEN 'A' THEN volume ELSE 0 END) AS ProductA,

+ MAX(CASE(product) WHEN 'B' THEN volume ELSE 0 END) AS ProductB,

+ MAX(CASE(product) WHEN 'C' THEN volume ELSE 0 END) AS ProductC

+ FROM d

+ GROUP BY user")

user ProductA ProductB ProductC

1 1 15 7 16

2 2 11 6 17

3 3 19 3 2

同学的问题：select后面加括号里面又写select。Select里面不能加sub-query.sub-query只能加在FROM后面。

* 右表变到左表怎么转换？

#wide table to long table:UNION ALL

> sqldf("

+ SELECT user,'A' as product, ProductA as volume FROM d2

+ UNION ALL

+ SELECT user,'B' as product, ProductB as volume FROM d2

+ UNION ALL

+ SELECT user,'C' as product, ProductC as volume FROM d2

+ ORDER BY user,product")

user product volume

1 1 A 15

2 1 B 7

3 1 C 16

4 2 A 11

5 2 B 6

6 2 C 17

7 3 A 19

8 3 B 3

9 3 C 2