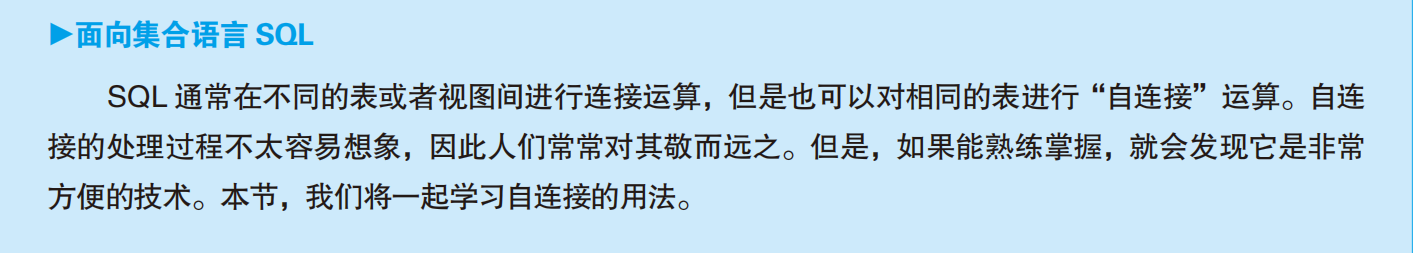
**必知必会的窗口函数四\_自连接的用法**

在上次的文章中，我们了解了窗口函数的基本知识，以及一些常见的窗口函数的使用方法，具体文章可以从以下连接中进行学习

**必知必会的窗口函数三：常见的窗口函数**

本次文章我们来学习一下SQL中的自连接技术，自连接（self join）是SQL语句中经常要用的连接方式，使用自连接可以将自身表的一个镜像当作另一个表来对待，从而能够得到一些特殊的数据。



1. **可重排列**

假设现在我们有一张Products表如下所示：



语句创建语句如下所示：

**CREATE** **TABLE** products

(**name** **VARCHAR**(16) **PRIMARY** **KEY**,

price **INTEGER** **NOT** **NULL**);

**INSERT** **INTO** products **VALUES**('苹果', 50);

**INSERT** **INTO** products **VALUES**('橘子', 100);

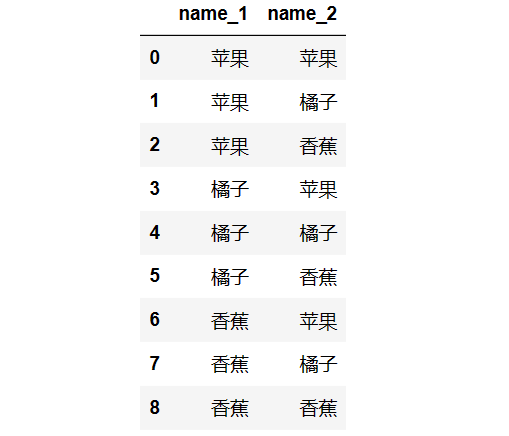
**INSERT** **INTO** products **VALUES**('香蕉', 80)

那么基于此表进行有序对的生成代码我们可以这样实现：  
-- 用于获取可重排列的 SQL 语句

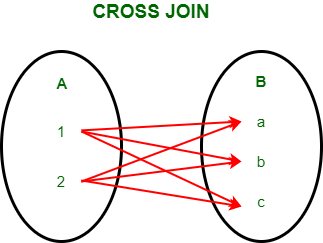
**SELECT** *P1*.name **AS** *name\_1*, *P2*.name **AS** *name\_2*

**FROM** Products *P1*, Products *P2*;

实现结果如下所示：



从结果上来看数量正好是3的平方，从结果上也有诸如（苹果,苹果）这样的数据对，这样来看正好是一个笛卡尔集。也是一种交叉连接（假设有两个表A和B）：



为了过滤掉相同的数据对，我们可以加上过滤条件：

**SELECT** *p1*.name **AS** *name\_1*, *p2*.name **AS** *name\_2*

**FROM** products **AS** *p1*, products **AS** *p2*

**WHERE** *p1*.name <> *p2*.name

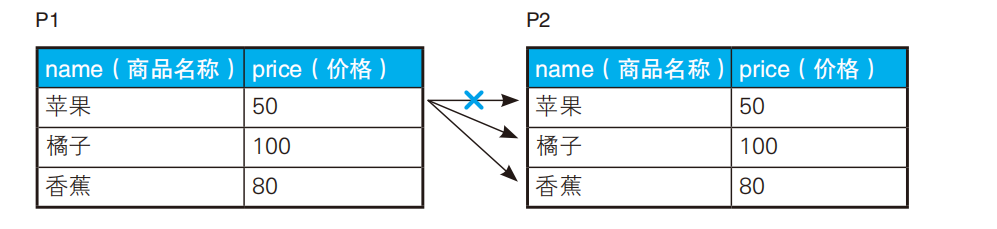
或者显示的使用连接方式：

**SELECT** *p1*.name **AS** *name\_1*, *p2*.name **AS** *name\_2*

**FROM** products **AS** *p1* **inner** **join** products **AS** *p2*

**ON** *p1*.name <> *p2*.name

也就是说强制使得下述关系包含在我们的数据处理过程中：



那么使用Pandas来实现上述的自连接，我们的代码可以编写如下所示：

query = """select \* from Products"""  
Products = pd.read\_sql\_query(query, engine)   
*# 将name列转换为一个DataFrame*name\_df = Products["name"].to\_frame()  
*# 创建一个公共的键列，值都为0*name\_df["key"] = 0  
*# 用merge函数进行交叉连接，on参数指定公共的键列，how参数指定连接方式为outer*cross\_join = name\_df.merge(name\_df, on="key", how="outer").rename(columns={"name\_x": "name\_1", "name\_y": "name\_2"})  
*# 删除结果表中的键列*cross\_join = cross\_join[['name\_1','name\_2']]  
*# 查看结果表*cross\_join

上述Python代码输出的也是9个值，如果想进行过滤可以使用以下代码：

query = """select \* from Products"""  
Products = pd.read\_sql\_query(query, engine)   
*# 将name列转换为一个DataFrame*name\_df = Products["name"].to\_frame()  
*# 创建一个公共的键列，值都为0*name\_df["key"] = 0  
*# 用merge函数进行交叉连接，on参数指定公共的键列，how参数指定连接方式为outer*cross\_join = name\_df.merge(name\_df, on="key", how="outer").rename(columns={"name\_x": "name\_1", "name\_y": "name\_2"})  
*# 用query函数筛选出结果表中name\_1和name\_2不相等的行*inner\_join = cross\_join.query("name\_1 != name\_2")  
*# 查看结果表*inner\_join

与之不同的是我们对结果集加上了**name\_1 != name\_2**进行过滤实现了上述要求，但是我们也可以更换成以下方法：

**cross\_join.drop\_duplicates(inplace=True)**

大家有什么其他好的方法也可以说出来以下学习一下。

通过上述的案例我们得出一个结论，相同的表的自连接和不同表间的普通连接并没有什么区别，自连接里的“自”这个词感觉也没什么太大的意义。

当然了我们还可以使用其他符号进行过滤：

**SELECT** *P1*.name **AS** *name\_1*, *P2*.name **AS** *name\_2*

**FROM** Products *P1*, Products *P2*

**WHERE** *P1*.name > *P2*.name;

这里补充一下，PostgreSQL 会逐个字符地比较它们的Unicode值，从而来判断它们的大小。

1. **删除重复行**

在关系数据库的世界里，重复行和NULL值一样，都很让人烦恼，我们经常会想很多办法来排除掉重复行。

假设我们创建表的语句如下：

**CREATE** **TABLE** products\_other

(**name** **VARCHAR**(16),

price **INTEGER** **NOT** **NULL**);

**INSERT** **INTO** products\_other **VALUES**('苹果', 50);

**INSERT** **INTO** products\_other **VALUES**('橘子', 100);

**INSERT** **INTO** products\_other **VALUES**('橘子', 100);

**INSERT** **INTO** products\_other **VALUES**('橘子', 100);

**INSERT** **INTO** products\_other **VALUES**('香蕉', 80)

可以看出（橘子,100）是一个重复的数据，我们打算将它删除。

《SQL进阶指南》中使用的是Oracle的rowid进行删除实验的，这里我们使用之前学习到的窗口函数来进行实验：

**WITH** **temp** **AS**

(**SELECT** \*, **ROW\_NUMBER**() **OVER** (**PARTITION** **BY** **name**, price **ORDER** **BY** ctid) **AS** rn

**FROM** products\_other)

**DELETE** **FROM** **temp** **WHERE** rn > 1;

[rowid 是 Oracle 中的一个特殊列，表示行的物理位置1](https://www.postgresql.org/docs/current/sql-expressions.html" \t "https://copilot.microsoft.com/_blank)[。PostgreSQL 中的类似概念是 ctid ，它表示行在表中的块号和偏移量](https://blog.csdn.net/weixin_39540651/article/details/106097373" \t "https://copilot.microsoft.com/_blank)。

这里需要注意的是在pg中索引扫描是先查询到数据的ctid，然后根据ctid去得到相应的数据。

但是ctid和oracle中的rowid并不完全相同，ctid是PG表中的系统字段，表示数据行在它所在表内的物理位置。ctid的字段类型是oid。但是VACUUM FULL操作之后，经过回收数据块内的空闲空间，数据行在块内的物理位置会发生移动，即ctid会发生变化。**大家可以实验一下是怎样变化的。**

当然了我们可以使用group by结合子查询来去重：

**DELETE** **FROM** products\_other

**WHERE** ctid **NOT** **IN** (

**SELECT** **min**(ctid)

**FROM** products\_other

**GROUP** **BY** name, price

);

这样我们就完成了重复数据的删除操作，那么在Pandas中我们怎么删除重复数据呢？在pandas中删除重复的数据使用drop\_duplicates()这个方法，该方法定义如下：

****DataFrame.drop\_duplicates(subset=None, keep=‘first’, inplace=False)****

subset：默认为None，需要标记重复的标签或标签序列

keep：默认为‘first’，如何标记重复标签

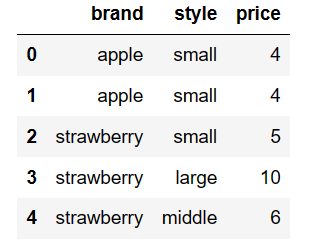
（1）first：将除第一次出现以外的重复数据标记为True

（2）last：将除最后一次出现以外的重复数据标记为True

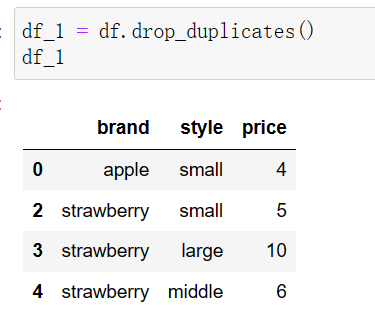
（3）False：将所有重复的项都标记为True（会删除所有重复数据）

Inplace:默认Fasle，不替换原始的数据

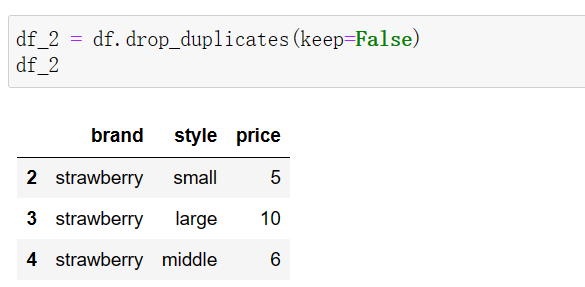
假设我们的数据如下:



那么不使用任何参数的时候当每一条记录的每一个列名的数据是一致的时候才是重复的：



那么当参数keep=False的时候：



可以看出删除的是所有的重复的数据。

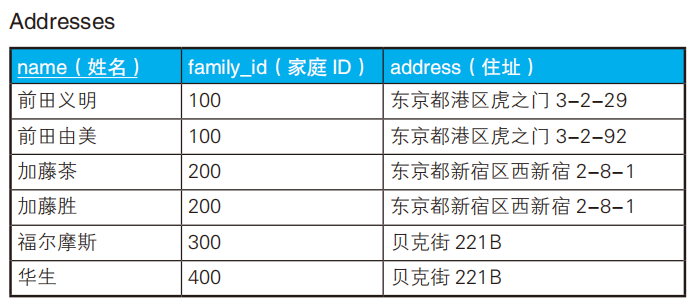
那么当参数keep=’first’的时候：



**这里出一个小题目，上述数据根据三列都是一样的情况下判定是重复的数据，那么如何指定任何几列来判断数据的重复，从而进行去重的操作呢？**

1. **查找局部不一致的列**

假设有下面这样一张住址表，主键是人名，同一家人家庭ID一样，找出是同一家人但住址却不同的记录。



这个问题不难，我们来看一下：

**CREATE** **TABLE** addresses

(**name** **VARCHAR**(32),

family\_id **INTEGER**,

address **VARCHAR**(32),

**PRIMARY** **KEY**(**name**, family\_id));

**INSERT** **INTO** addresses **VALUES**('前田义明', '100', '东京都港区虎之门3-2-29');

**INSERT** **INTO** addresses **VALUES**('前田由美', '100', '东京都港区虎之门3-2-92');

**INSERT** **INTO** addresses **VALUES**('加藤茶', '200', '东京都新宿区西新宿2-8-1');

**INSERT** **INTO** addresses **VALUES**('加藤胜', '200', '东京都新宿区西新宿2-8-1');

**INSERT** **INTO** addresses **VALUES**('福尔摩斯', '300', '贝克街221B');

**INSERT** **INTO** addresses **VALUES**('华生', '400', '贝克街221B');

我们可以使用自连接相关的语法写出这样的查询语句：

**SELECT** **DISTINCT** *A1*.***name***, *A1*.***address***

**FROM** ***Addresses*** *A1*, ***Addresses*** *A2*

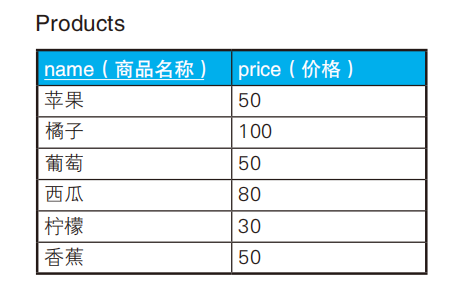
**WHERE** *A1*.***family\_id*** = *A2*.***family\_id***

**AND** *A1*.***address*** <> *A2*.***address*** ;

这里很简单就不在使用Pandas来实现了。

1. **排序**

在之前的窗口函数中我们介绍了很多的排序函数，如RANK(),DENSE\_RANK()。比如我们来进行排序，假设表如下：



由于之前已经创建了该表，只需更新数据即可：

**DELETE** **FROM** products;

**INSERT** **INTO** products **VALUES**('苹果', 50);

**INSERT** **INTO** products **VALUES**('橘子', 100);

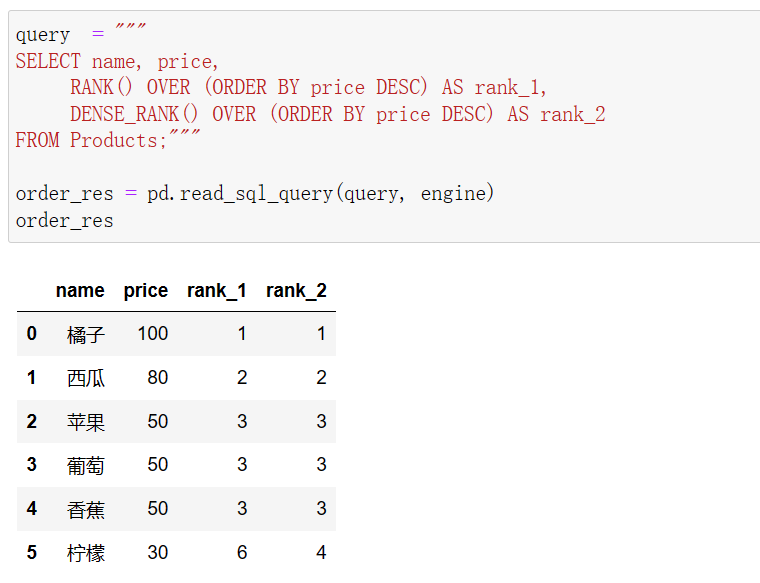
**INSERT** **INTO** products **VALUES**('葡萄', 50);

**INSERT** **INTO** products **VALUES**('西瓜', 80);

**INSERT** **INTO** products **VALUES**('柠檬', 30);

**INSERT** **INTO** products **VALUES**('香蕉', 50);

那么使用窗口函数对price进行排序的方法和结果如下：



可以看出RANK()函数的排序结果是跳跃的，而DENSE\_RANK()的排序结果不是。

这里就自然存在一个想法，我们自己怎么实现这个排序函数呢，毕竟在有一些数据库中这些函数是没有的。比如目前的MySQL函数中是么有这个RANK()函数的。

从排序结果来看，我们思考这个问题，实际上排序的序号就是price对应的在所有price中的序号：



有了这个思想，我们来实现就比较容易一些：

**SELECT** *P1*.name,

*P1*.price,

(**SELECT** **COUNT**(*P2*.price)

**FROM** Products *P2*

**WHERE** *P2*.price > *P1*.price) + 1 **AS** *rank\_1*

**FROM** Products *P1*

**ORDER** **BY** ***rank\_1***;

子查询统计的就是比相应的price值大的数量，就如上述的表格的展示的一样。结果如下：



当然了，我们实现的是RANK()函数的排序，中间的结果是跳跃的，那么如何实现不跳跃的排序呢？

我们就只需要将比较的元素的个数从全量6个缩减到4个即可，这个比较简单，我们对集合的元素进行去重即可，我们对p2.price进行去重**distinct** P2.price，实现的脚本如下所示：

**SELECT** *P1*.name,

*P1*.price,

(**SELECT** **COUNT**(**distinct** P2.price)

**FROM** Products *P2*

**WHERE** *P2*.price > *P1*.price) + 1 **AS** *rank\_1*

**FROM** Products *P1*

**ORDER** **BY** ***rank\_1***;

结果就是意料之中的输出了：



那么使用内连接的实现方式可以写出这样的代码：

**SELECT** *P1*.name,

**MAX**(*P1*.price) **AS** *price*,

**COUNT**(*P2*.name) +1 **AS** *rank\_1*

**FROM** Products *P1* **INNER** **JOIN** Products *P2*

**ON** *P1*.price < *P2*.price

**GROUP** **BY** *P1*.name

**ORDER** **BY** ***rank\_1***;



咦？结果怎么少了一个呢？原因很简单内连接的时候，因为找不到比橘子价格100还要大的元素的，所以就出现这样的结果了。

如果更改成left join也能得到预期的结果：

**SELECT** *P1*.name,

**MAX**(*P1*.price) **AS** *price*,

**COUNT**(*P2*.name) +1 **AS** *rank\_1*

**FROM** Products *P1* **left** **join** Products *P2*

**ON** *P1*.price < *P2*.price

**GROUP** **BY** *P1*.name

**ORDER** **BY** ***rank\_1***;

自连接的这种方式在理解上要稍微不是那么顺畅一些，不知道大家的感受是怎样的。

1. **排序Pandas实现**

使用pandas来实现上述排序很简单，如果使用pandas自带的函数来实现就直接写出如下的代码：

**import** pandas **as** pd

# 读取准备

**data** = {

'name': ['苹果', '橘子', '葡萄', '西瓜', '柠檬', '香蕉'],

'price': [50, 100, 50, 80, 30, 50]

}

df = pd.DataFrame(**data**)

# 使用 **rank**() 函数进行排序，参数默认min,也可以设置为 first或 dense

df['rk'] = df['price'].**rank**(**method**='min')

# df['rk'] = df['price'].**rank**(method='dense')

df\_sorted = df.sort\_values('rk')

df\_sorted

这里的输出结果正好对应上述的RANK()函数的实现，主要使用的是Pandas中的rank函数，通过传入不同的函数完成不同的功能。

依然，我们可以不借助相关的函数来实现，我们使用自己的方式来实现这个需求，代码如下：

**import** pandas **as** pd

# 数据准备

df = pd.DataFrame({

'name': ['苹果', '橘子', '葡萄', '西瓜', '柠檬', '香蕉'],

'price': [50, 100, 50, 80, 30, 50]

})

# 使用apply函数计算每个水果的价格在所有产品价格中的排名

df['rank'] = df['price'].apply(lambda x: (df['price'] > x).**sum**() + 1)

# df['dense\_rank'] = df['price'].apply(lambda x: (df['price'].**unique**() > x).**sum**() + 1)

# 按照排名排序

df = df.sort\_values('rank')

# df = df.sort\_values('dense\_rank')

df

上述代码(df['price'] > x).**sum**()和(df['price'].**unique**() > x).**sum**()是整个实现的关键点，这个思路还是集合的一种思想，大家好好体会。

1. **总结**

本次推文我们学习了自连接的一些知识。自连接是 不亚于 CASE 表达式的重要技术，请一定熟练掌握。我们需要注意的是与多表之间进行的普通连接相比，自连接的性能开销更大（特别是与非等值连接结合使用的时候），因此用于自连接的列推荐使用主键或者在相关列上建立索引。

自连接的开销为什么大，我这里引用LLM的给的结果：

（1）数据库自连接涉及的环节较多，包括数据在内存中的存储、数据在网络中的传输、数据库对数据的处理等，这些环节都需要消耗一定的计算资源。

（2）数据库自连接涉及到大量的数据操作，如数据的筛选、匹配、连接等，这些操作都需要消耗大量的计算资源。

（3）数据库自连接过程中，需要进行大量的数据转换和格式化操作，这些操作也需要消耗一定的计算资源。

（4）数据库自连接过程中，需要进行大量的数据索引和排序操作，这些操作需要消耗大量的计算资源。

（5）数据库自连接过程中，需要进行大量的数据并发控制和事务处理操作，这些操作也需要消耗一定的计算资源。

总之，数据库自连接开销大的原因是多方面的，包括数据处理、数据转换、数据索引、数据并发控制和事务处理等。为了减少数据库自连接的开销，可以采取一些优化措施，如优化数据库查询语句、建立有效的索引、采用缓存技术等。

1. **参考文献**
2. MICK-《SQL进阶教程》第2版
3. Joe Celko-《SQL权威指南（第4版）》

<https://www.runoob.com/postgresql/postgresql-datetime.html>

<https://learnsql.com/blog/sql-window-functions-rows-clause/>

<https://www.zhihu.com/question/596882478?utm_id=0>

<https://zhuanlan.zhihu.com/p/351822793>

<https://zhuanlan.zhihu.com/p/622993495>

<https://zhuanlan.zhihu.com/p/260182391>

