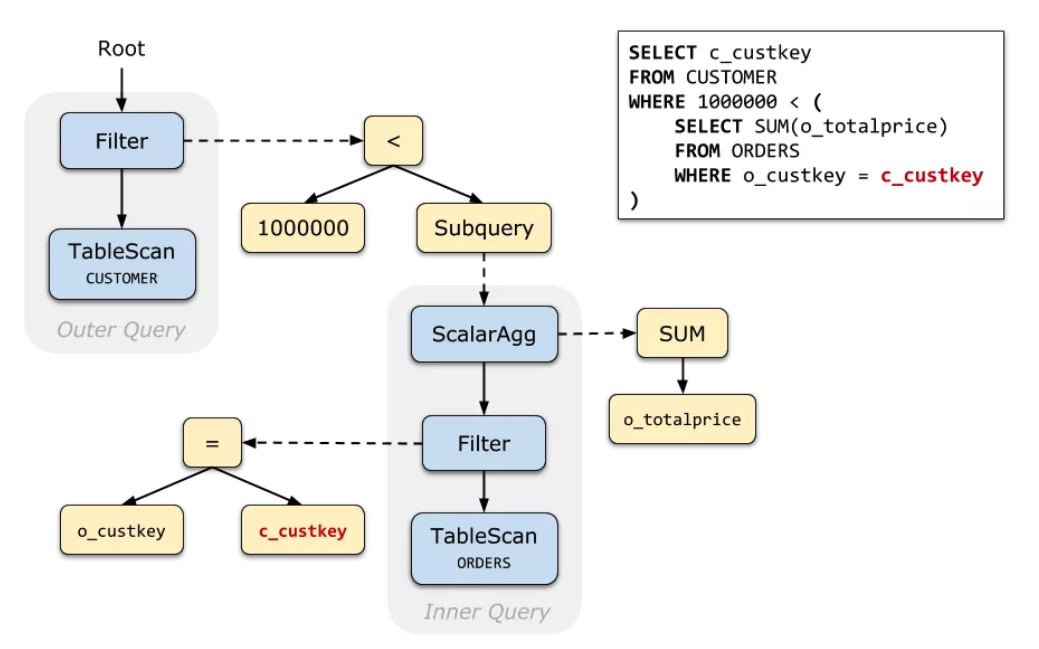
# 子查询

## 概述

根据SQL标准，将一个查询块嵌套进一个表达式中，我们就得到了一个子查询。

关系型数据库中的查询块，通常由关系代数算子组成的树状计划来表达。而每个算子中的表达式与外层的查询树的关系是暧昧不清的。



查询树中的表达式藏着另一棵查询树，本质在于通过表达式来描述查询树间的关系。比如说自定义函数、比较运算符、逻辑运算符。由于表达式类型繁多且过于复杂，很难直接通过传统的关系代数抽象模拟子查询与主查询间的关系。这让子查询成为关系型数据库中实现的难点之一。

既然子查询的实现对数据库来说如此复杂，执行的效率也未必高，为何还要实现这种SQL语法呢？除了SQL标准以外还有其它原因吗？

... WHERE AGE > ALL(SELECT AGE ...)... WHERE SALARY <= ANY(SELECT SALARY ...)

将各个查询树理解为一个数据集，通过JOIN来描述数据间的交并集关系在部分场景下是非常晦涩的。但观察以上SQL，AGE >ALL和SLARY<ANY的子查询描述非常接近自然语言，能大大简化用户构建复杂查询的过程。因此从某种意义上来讲，可以将子查询理解为一种历史悠长的SQL语法糖。

一言以蔽之，子查询就是一个把复杂留给数据库，把简单送给用户的经典案例。

当一个查询是另一个查询的条件时，称之为子查询。

子查询和连接一样，提供了使用单个查询访问多个表中的数据的方法。子查询可以使用在SELECT、INSERT、UPDATE和DELETE语句中，使它们能够利用子查询返回的结果。

注意：子查询不仅可以使用在WHERE子句中，还经常使用在SELECT子句中。这种情况下，在子查询中要返回的单一值经常是聚合分析的结果。

需要注意的是，子查询必须包含括号。

子查询与主查询之间并不是水平关系，而是从属关系。这就意味着不论使用哪种类型的子查询，都必须确保不能改变主查询的完整性。通常情况下，优化器都会将子查询合并到主查询中，以便产生更优质的执行计划。这里可能采用嵌套循环、排序合并或哈希连接等方式。

## 特点

### 优点

通常来讲，使用子查询的好处如下：

子查询允许结构化的查询，这样就可以把一个查询语句的每个部分隔开。

子查询提供了另一种方法来执行有些需要复杂的JOIN和UNION来实现的操作。

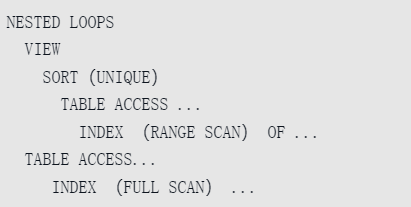
在许多人看来，子查询可读性较高。而实际上，这也是子查询的由来。

### 缺点

## 处理方式

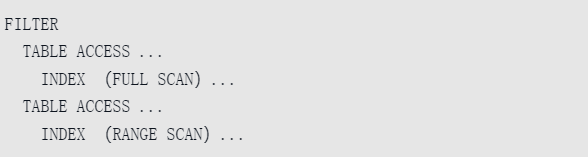
在合并之后，可能有两种处理方式：一种是子查询优先，一种是主查询优先。 **1、子查询优先**

如果子查询与主查询的表连接方式是优先执行子查询，并将其执行结果提供给主查询的嵌套循环连接，那么优化器将优先执行子查询，并通过对结果进行唯一排序SOR（UNIQUE），再与主查询进行连接。在排序合并连接和哈希连接中，也是这样处理的。通常可以看到类似下面的执行计划。



1. **主查询优先**

如果将主查询的执行结果作为外侧循环来使用，而把子查询作为内侧循环来使用。此时采用在内侧循环中第一行被连接成功之后就立刻结束内侧循环的方式。这种处理方式所制定的策略就是前面在嵌套循环中提到的FILTER。通常可以看到类似下面的执行计划。



## 分类

### 按照谓词分类

#### 单值子查询

一个子查询如果产生一个单纯的数据，该子查询就如同一个常量，那么就可以像使用常量一样使用它。在实际应用中，我们经常要求子查询只返回一个值，这样就可以将一列值和单个子查询返回值进行比较，这时，可以使用等于（=）、不等于（<>）、大于（>）、小于（<）、大于等于（>=）、小于等于（<=）等运算符。

注：**在SELECT子句中使用子查询时，子查询必须返回单值**。

#### 多值子查询

一个子查询除了可以产生一个单一值外，也可以产生一个关系，该关系可以包含若干元组。SQL提供了若干对于关系的操作符，并产生一个布尔型的结果，这些操作符主要用在子查询的结果关系上，它主要包括：IN、EXISTS、SOME（ANY）、ALL、UNIQUE等。

##### IN

##### EXISTS

##### SOME（ANY）

##### ALL

##### UNIQUE

### 按照语法分类

#### 独立子查询

常用于in、not in中，语法特点是子查询与外部查询完全可以独立运行。语法意义上的含义是主表谓词对应的范围筛选，比如下面的示例。



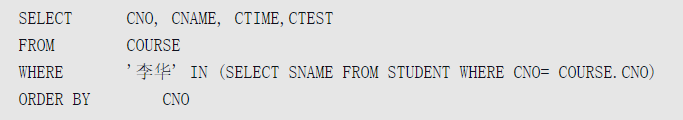
#### 相关子查询

常用于exists、not exists中，当然in、not in也可以。它的语法特点是相互包含，外表的信息被子查询引用，子查询嵌套在外部查询中。语法意义上的含义是存在性判断，比如下面的示例。

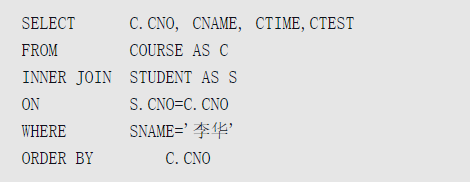


##### 使用IN引入相关子查询

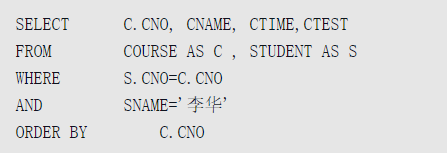
查询COURSE表中李华同学修过的课程的课程号、课程名称、学时和考试时间信息。实例代码如下：



上述代码在执行过程中，DBMS每从COURSE表取出一条记录，都要执行一次子查询。因此，相关子查询具有较高的处理要求，在应用中应尽量避免使用。例如，我们可以采用表的连接实现上面的查询操作，代码如下：

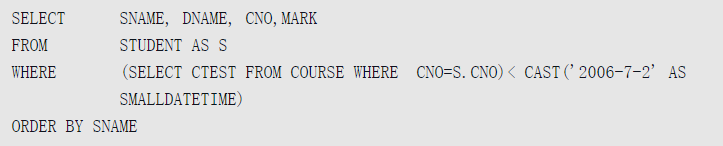


当然，上面表的连接也可以采用带WHERE子句的多表的SELECT语句，代码如下：



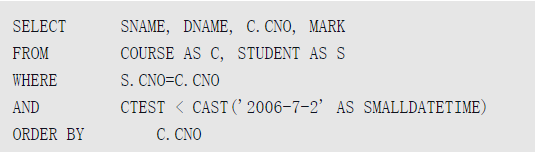
##### 比较运算符引入相关子查询

在STUEDENT表中查询学生的姓名、所在系，课程号及成绩，要求查询的课程的考试时间在2006年7月2日之前。实例代码如下：



比较运算符要求其比较的数据具有相同的数据类型，而STUDENT表中CTEST字段的数据类型为SMALLDATETIME，因此我们需要使用CAST运算式将时间字符串“2006-7-2”转换为SMALLDATETIME型。

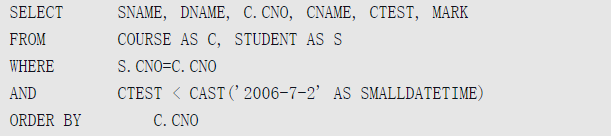
同样，我们可以采用INNER JOIN连接表或者SELECT语句的方法实现上面的查询操作。这里采用带WHERE子句的多表的SELECT语句的实现，代码如下：



可见，二者实现的功能是完全相同的。

与采用相关子查询的方式相比，采用表连接的方式对数据的查询更加灵活。因为，如果采用相关子查询，所能查询的列只能是主查询中FROM子句中的列，不能查询其它列；而采用多表连接的方式，则不存在这个问题，可查询连接的任意表中的列。

在STUEDENT表和COURSE表中查询学生的姓名、所在系，课程号、课程名、考试时间及成绩，要求查询的课程的考试时间在2006年7月2日之前。实例代码如下：

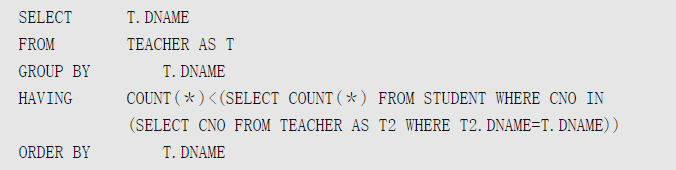


可见，使用多表连接的方式，我们只要在SELECT子句中添加COURSE表中的几个字段，即可得到想要的结果；而如果采用相关子查询的方式，则还需要将COURSE表添加到主查询的FROM子句中，这显然失去了相关子查询的意义。

##### 在HAVING子句中使用相关子查询

在SQL中，DBMS使用WHERE子句中的搜索条件来过滤查询结果表中不想要的行，使用HAVING子句中的搜索条件删除那些不想要的组。

在TEACHER表中查询其包含的系的名称，要求这些系中，各系包含的教师数目要少于STUDENT表中修过该系的教师开设课程的学生的人次。假如TEACHER表中，A系有m个教师，而在STUDENT表中有n人次修过这m个教师开设的课程，如果n>m，则A系就符合查询条件。实例代码如下：



#### 标量子查询

常用于结果集不大，子查询访问非常高效的情况。希望针对每个外部查询的结果，查询其他表、视图等信息。语法的特点是每行匹配结果都是单行单列。一般使用相关标量子查询居多。语法意义上如果匹配不到，则为空。优化这种查询多改为Outer Join，注意连接条件是否为空，比如下面的示例。

### 按照语义划分

#### 标量子查询

标量子查询（ Scalar Subquery ）输出一行值。

普通算子输出的二维表（行与列），类似于向量；与之对应称只输出一行一列的子查询为标量。

SELECT c\_custkey

FROM CUSTOMER

WHERE 1000000 < (

SELECT SUM(o\_totalprice)

FROM ORDERS

WHERE o\_custkey = c\_custkey

)

#### 集合比较

集合比较（ Quantified Comparision ）：

ALL（全部满足）

ANY（只要满足一个）

SELECT c\_name

FROM CUSTOMER

WHERE c\_nationkey <> ALL (SELECT s\_nationkey FROM SUPPLIER)

#### 存在性测试

存在性测试（ Existential Test ）：

EXISTS（存在）

NOT EXISTS（不存在）

SELECT c\_custkey

FROM CUSTOMER

WHERE c\_nationkey = 86 AND EXISTS(

SELECT \* FROM ORDERS

WHERE o\_custkey = c\_custkey

)

### 按照逻辑运算分类

语义上的划分更易理解，但数据库会基于逻辑运算的角度抽象出另一种划分方式：

#### Scalar

Scalar：俗称标量子查询，对外输出一行

#### Semi

Semi：(EXISTS, IN, ANY) 可拆解为析取的逻辑关系。

例如AGE IN (SELECT AGE)子查询可表达为a.age=b.age[0] OR a.age=b.age[1] OR ... OR a.age=b.age[n]

#### Anti-Semi

Anti-Semi：(NOT EXISTS, NOT IN, ALL) 可拆解为合取的逻辑关系。

例如AGE >ALL (SELECT AGE)子查询可表达为a.age>b.age[0] AND a.age>b.age[1] AND ... AND a.age>b.age[n]

### 嵌套子查询

大多数情况下，子查询出现在查询的HAVING子句或WHERE子句中。子查询自身也可以包含WHERE子句和或HAVING子句，同样，子查询也可以出现在其他子查询中。位于其他子查询内的子查询被称为嵌套的子查询。

### 树查询

## 使用场景

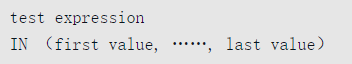
使用子查询可以避免由于子查询中的数据产生的重复

使用子查询更符合语意，更好理解

# 原理

## IN子查询

IN运算符允许根据一行记录中是否有一列包括在一系列值之中而选择该行。语法如下。



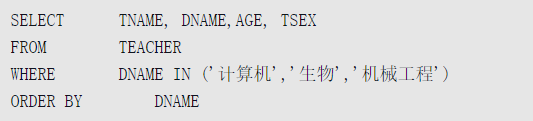
所有的条件在IN运算符后面罗列，并以括号“()”包括起来，条件中间用逗号分开。当testexpression处于括号中列出的一系列值之中时，IN运算符求值为True。

与IN运算符相对的还有NOT IN运算符，其判别结果刚好与IN运算符相反，即当test expression不处于括号中列出的一系列值之中时，NOT IN运算符求值为True。

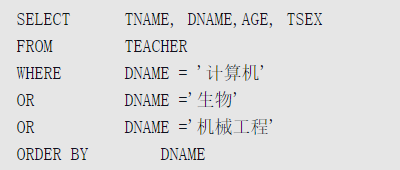
### IN子查询与OR运算符

**在大多数情况下，OR运算符与IN运算符可以实现相同的功能**。

例如：



等价于：



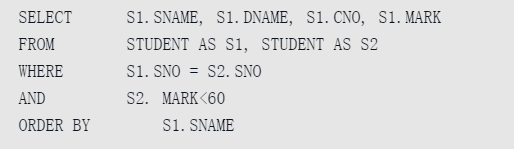
我们可以发现二者结果完全相同。那么为什么要提出IN运算符呢？这是因为与OR运算符相比，IN运算符具有以下两个优点。

• 当选择的条件很多时，采用IN运算符就显得很简便，只需在括号内用逗号间隔依次罗列即可，运行效率比OR运算符要高。

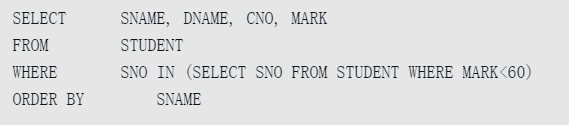
• 使用IN运算符，其后面所列的条件可以是另一条SELECT语句，即子查询。

### IN子查询与自连接

可以使用IN实现自连接：



采用IN子查询的方式如下：



分析一下代码的执行过程。

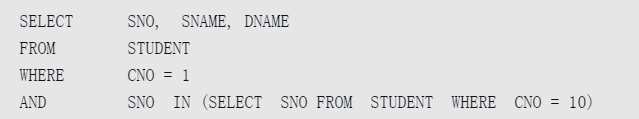
1. 执行一个查询时，DBMS首先处理最里面的子查询。系统首先执行“SELECT SNO FROMSTUDENT WHERE MARK<60”语句，生成STUDENT表中成绩小于60的学生的学号SNO集。
2. DBMS一次处理STUDENT表中的一行记录，并且将每行记录中的SNO列值与子查询结果集中的SNO值进行比较。
3. 如果在系统子查询结果集中找到了与正在处理的记录中的SNO值相匹配的值，WHERE子句求值为True，DBMS则将该记录的相关信息归入结果表。
4. 如果在子查询结果集中没有发现与记录的SNO值相匹配的值，WHERE子句求值为False，DBMS转而去处理STUDENT表中的下一行而舍弃该行数据。

### IN子查询与集合交/差集

在STUDENT表中查询修过1号或者10号课程的学生的学号、姓名及所在系信息。

分析：该实例实际上可理解为修过1号课程的学生的相关信息与修过10号课程的学生的相关信息的交集。

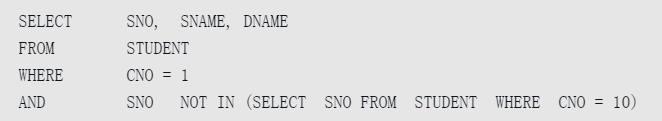
实例代码如下：



在STUDENT表中查询修过1号课程但是没有修10号课程的学生的学号、姓名及所在系信息。

分析：该实例实际上可理解为修过1号课程的学生的相关信息与修过10号课程的学生的相关信息的差集。

实例代码如下：



注意：由运算符IN引入的子查询返回的列值，既可以来自主查询的表，也可以来自其他表。SQL对查询的唯一要求就是它必须返回单一列的数据值，并且其数据类型必须与IN前面表达式的数据类型相兼容。

## EXIST子查询

EXISTS是一个非常强大的谓词，它允许数据库高效地检查指定查询是否产生某些行。通常EXISTS的输入是一个子查询，并关联到外部查询，但这不是必须的。根据子查询是否返回行，该谓词返回TRUE或FALSE。与其他谓词和逻辑表达式不同的是，无论输入子查询是否返回行，EXISTS都不会返回UNKNOWN。如果子查询的过滤器为某行返回UNKNOWN，则表示该行不返回，因此，这个UNKNOWN被认为是FALSE。

注意：尽管通常不建议在SQL语句中使用\*，因为可能会引起一些问题的产生，但是在EXIST子查询中\*可以放心地使用。EXISTS只关心行是否存在，而不会去取各列的值。

有些DBA有过一些其他数据库的使用经验，在其他数据库中可能存在这样“幽默”的优化定理，就是将IN语句改写为EXISTS，这样SQL查询的效率更高。据我所知，的确曾有过这种说法，这可能是因为当时优化器还不是很稳定和足够优秀。目前在绝大多数的情况下，IN和EXISTS都具有相同的执行计划。但是要注意的是，NOT IN和NOT EXISTS具有非常不同的执行计划。

### IN和EXIST区别

mysql中的in语句是把外表和内表作hash 连接，而exists语句是对外表作loop循环，每次loop循环再对内表进行查询。一直大家都认为exists比in语句的效率要高，这种说法其实是不准确的。这个是要区分环境的。

1、如果查询的两个表大小相当，那么用in和exists差别不大。

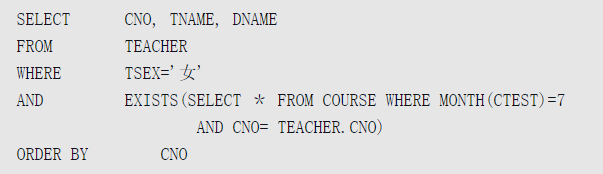
2、如果两个表中一个较小，一个是大表，则子查询表大的用exists，子查询表小的用in。

3、not in 和not exists如果查询语句使用了not in 那么内外表都进行全表扫描，没有用到索引；而not extsts的子查询依然能用到表上的索引。所以无论那个表大，用not exists都比not in要快。

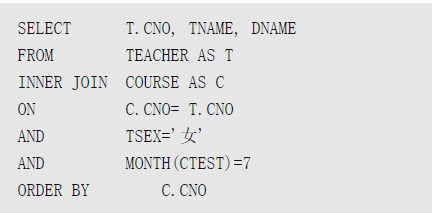
### EXISTS子查询与交集

在TEACHER表中查询女教师开设的、6月份考试的课程号、开课教师的姓名以及教师所在的系。分析：实际上我们要查询的课程为TEACHER表中女教师开设的课程与COURSE表中7月考试的课程的交集。

实例代码如下：



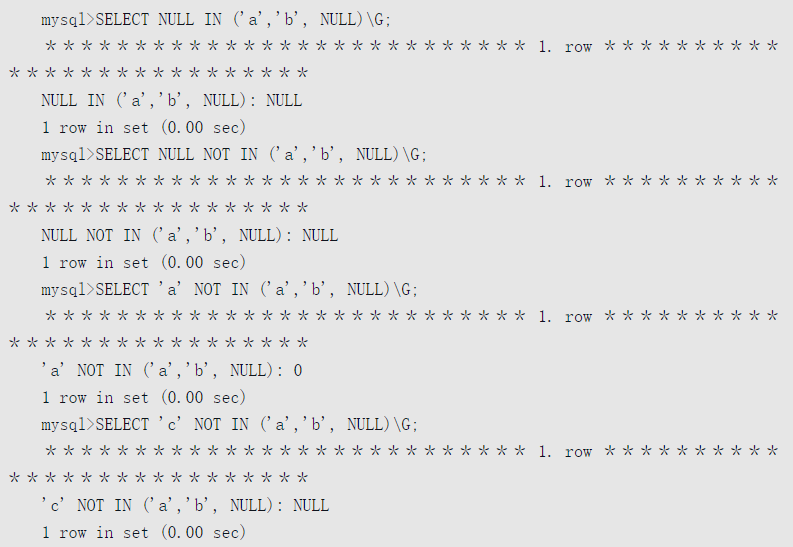
也可以采用INNER JOIN方式，代码如下：



### NOT EXISTS

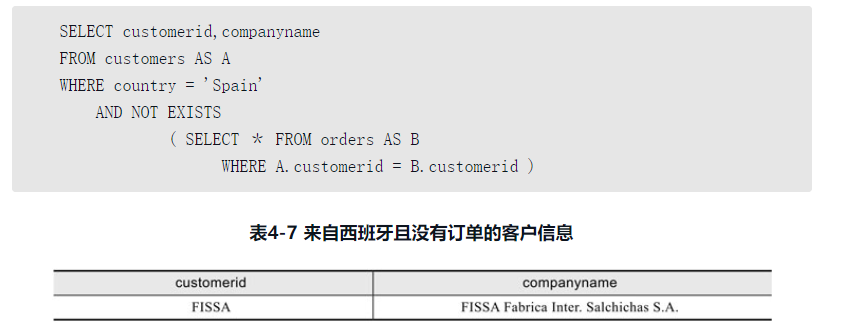
EXISTS与IN的一个小区别体现在对三值逻辑的判断上。EXISTS总是返回TRUE或FALSE，而对于IN，除了TRUE、FALSE值外，还有可能对NULL值返回UNKNOWN。但是**在过滤器中，UNKNOWN的处理方式与FALSE相同，因此使用IN与使用EXISTS一样， SQL优化器会选择相同的执行计划**。

但是输入列表中包含NULL值时，NOT EXISTS和NOT IN之间的差异就表现得非常明显了。输入列表中包含NULL值时，IN总是返回TRUE和UNKNOWN，因此NOT IN总是返回NOT TRUE和NOT UNKNOWN，即FALSE和UNKNOWN。我们来看下面的例子：



IN和NOT IN的返回值都是显而易见的。NULL IN('a','b',NULL)返回的是NULL，因为对NULL值进行比较返回的是UNKNOWN状态。最后， 'c'NOT IN('a','b',NULL)的结果可能出乎一些人的意料，其返回的是NULL。因为之前已经说**对于包含NULL值的NOT IN来说，其总是返回FALSE和UNKNOWN，而对于NOT EXISTS，其总是返回TRUE和FALSE。这就是NOT EXISTS和NOT IN的最大区别**。

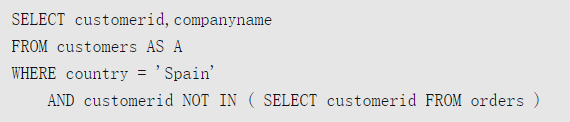
例如，我们要返回来自西班牙且没有订单的客户信息，下面是使用NOT EXISTS谓词的解决方案，生成的结果如表4-7所示。



该查询的执行计划如图4-8所示。



该查询和EXISTS的执行计划并没有什么不同，首先过滤来自西班牙的消费者，然后再匹配相关子查询。接着我们再用NOT IN来解决这个问题，其返回和NOT EXISTS相同的结果。该查询的过程如下：



再来看SQL语句的执行计划，如图4-9所示。

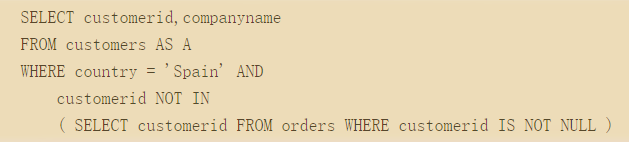


虽然NOT IN和NOT EXISTS产生相同的结果，但是执行计划却发生了一些细微的改变。在NOT IN中，相关子查询中的type列变为index\_subquery，而在NOT EXISTS中， type列和EXISTS查询一样，都是ref。

对于NOT EXIST和NOT IN，虽然执行计划不同，但是返回的结果是相同的。这是因为orders表中不存在customerid为NULL的行。若人为地插入以下数据，再来比较NOT EXISTS和NOT IN之间的区别：



再次运行NOT EXISTS和NOT IN查询，就会发现NOT EXISTS依旧返回之前的结果，但是NOT IN查询返回空集合，这是因为orders表中存在customerid为NULL的行。所以NOT IN的查询返回的是FALSE和UNKNOWN，而不是TRUE，从而导致我们找不到需要的数据。因此对于使用NOT IN的子查询，可以在子查询中先过滤掉NULL值，如：



测试完这些查询，执行下面的语句来移除customerid为NULL的行。



注：对于NOT IN和NOT EXISTS是否可以等价转换，需要看数据中是否含有NULL值，如果没有NULL值则可以直接转换，否则需要在子查询中增加过滤条件。

## NOT运算符

NOT运算符用于对搜索条件的布尔值求反。本节将讨论NOT运算符的使用、应用场合及其与<>运算符的区别。

注：**NULL值进行取反，结果仍是NULL**。

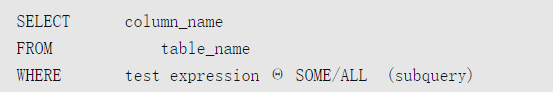
对于简单的条件查询，NOT运算符与<>运算符的功能几乎没有什么区别，那么NOT运算符的优势体现在哪里呢？答案是它可以与其他运算符组合使用，这一点是<>运算符所不能实现的。

注：MySQL数据库系统不支持NOT运算符。在MySQL数据库系统中，NOT运算符只能被用在EXISTS运算符的前面，形如NOT EXISTS。

## SOME/ANY/ALL子查询

只要我们使用了SQL比较运算符（等于=、不等于<>、大于>、小于<、大于等于>=、小于等于<=）中的一个来比较两个表达式的值，那么运算符前后的表达式都必须为单一值。正如10.1节所介绍的那样，只有当子查询返回值为单值时，才可使用子查询作为比较判式的表达式之一。

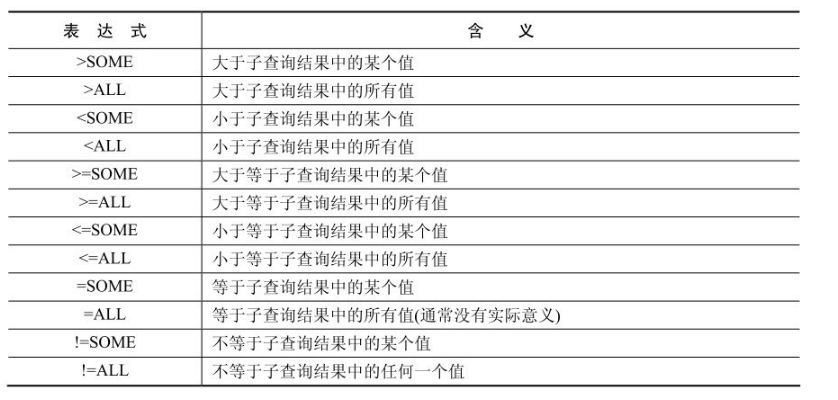
而数量词SOME、ANY和ALL则允许使用比较运算符将单值与子查询返回的值加以比较，这里的子查询返回的结果可以是多行的。语法如下：



test expression可以是实际值、列名、表达式或是另一个返回单一值的子查询。Θ为比较运算符，可以是6种SQL比较运算符中的任意一种。我们如果采用SOME关键字，则表示表达式的值至少与子查询结果中的一个值相比，满足比较运算符Θ；而如果采用ALL关键字，则表示表达式的值与子查询结果中所有的值相比，都满足比较运算符Θ。

**注：**关键字SOME和关键字ANY具有完全相同的功能。

SOME/ALL关键字与比较运算符的联合使用：



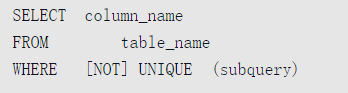
SOME和ALL与聚集函数的对应关系：



## UNIQUE

UNIQUE运算符用来测试集合是否存在重复元组。与EXISTS判式相似，它总是与子查询结合使用，而且只要子查询结果中没有重复记录，UNIQUE判式的值就为True；如果子查询的结果表中有重复的记录，那么UNIQUE判式的值为False。

简单的UNIQUE子查询语法可表示如下：



这里需要说明的是，只要子查询subquery中返回的结果表存在重复记录，则UNIQUE (subquery)就返回True，WHERE子句的结果也就为True；当然，如果使用NOT UNIQUE判式，当子查询返回的结果有重复的记录行时，它返回True。

要使UNIQUE判式有意义，我们应该在子查询中建立搜索条件，以匹配子查询连接起来的两个表中的值。

# 子查询优化

SQL作为一种声明式的语言，仅仅描述了它需要什么样的数据，具体怎么操作则还要看数据库自己的发挥。查询优化技术发展到如今（2021），子查询必然会导致性能下降的说法已经过于武断了。

关联子查询的本意为外表每一行数据与子查询集合数据的运算。当外表数据行过多时，这一嵌套的过程将不可避免导致性能低下。因此子查询优化很重要的一步就是去关联化（unnesting），如今的去关联技术已经日趋成熟，HyPer 2015年就宣称自己能够《Unnesting Arbitrary Queries》。

针对子查询，优化器支持了多种优化策略。

Oracle查询转换功能主要有**启发式（基于规则）**查询转换以及**基于Cost的查询转换**两种，针对子查询主要有Subquery Unnest、Push Subquery等。

查询转换的目的是转化为Join（包括Semi、Anti Join等），充分利用索引、Join技术等高效访问方式提高效率。

**如果子查询不能unnest（启发式），可以选择把子查询转换为Inline View（基于Cost）；如果都不可以，那么子查询就会最后执行，可能会看到类似Filter的操作**。

## 边界

将子查询进行逻辑拆解后，并不代表它就是一个普通的表达式。关系型数据库的表达式计算过程中，函数会尽量以行数据作为输入输出。

对集合进行处理的函数通常会抽象为一个新的算子，例如聚合函数会单独用Agg算子处理。同样的道理子查询也应该抽象为一个单独的算子。算子与表达式切割时的边界划分问题也遵循相同的逻辑。

数据库引擎一般不会将复杂的数据集运算混淆到表达式计算中，从实现的复杂度考虑，也会尽量明确集合运算与行运算的边界。



1、Scalar：由于它的输出仅有一行，子查询的边界可以划分到查询块上。

2、SEMI/ANTI：查询块可以输出多行, 只有配合子查询的 ANY，ALL 表达式才能将输出限制为一行 boolean 值。因此边界必须包含子查询表达式前面的入参。





最后基于子查询本身是否有关联项，还可以分为关联子查询及非关联子查询。凡是非关联子查询本质上都可以视做为一个常量，而关联子查询处理时必须要考虑子查询内外层的数据关系。对这层数据关系的处理便是子查询优化技术的重点。

## 去关联化（Unnesting）

由于处理的是集合数据，子查询应该从表达式中剥离出来，以算子的方式展示在执行计划中。这个指代子查询的算子一般称之为Apply。

Apply这个名字来源于LISP：指一类特殊的函数，入参是一个参数列表的集合，返回值是对应的一个结果列表集合。从关系代数数据库语义上看与关联子查询的嵌套执行过程类似。最早由微软SQL Server论文提出。

将子查询转为Apply算子后，关联项还残留在子查询的查询树内，所以没有办法直接将Apply以Join的方式处理。

因此，子查询优化通常最重要的一步是去关联化。

SELECT c\_custkey

FROM CUSTOMER

WHERE 1000000 < (

SELECT SUM(o\_totalprice)

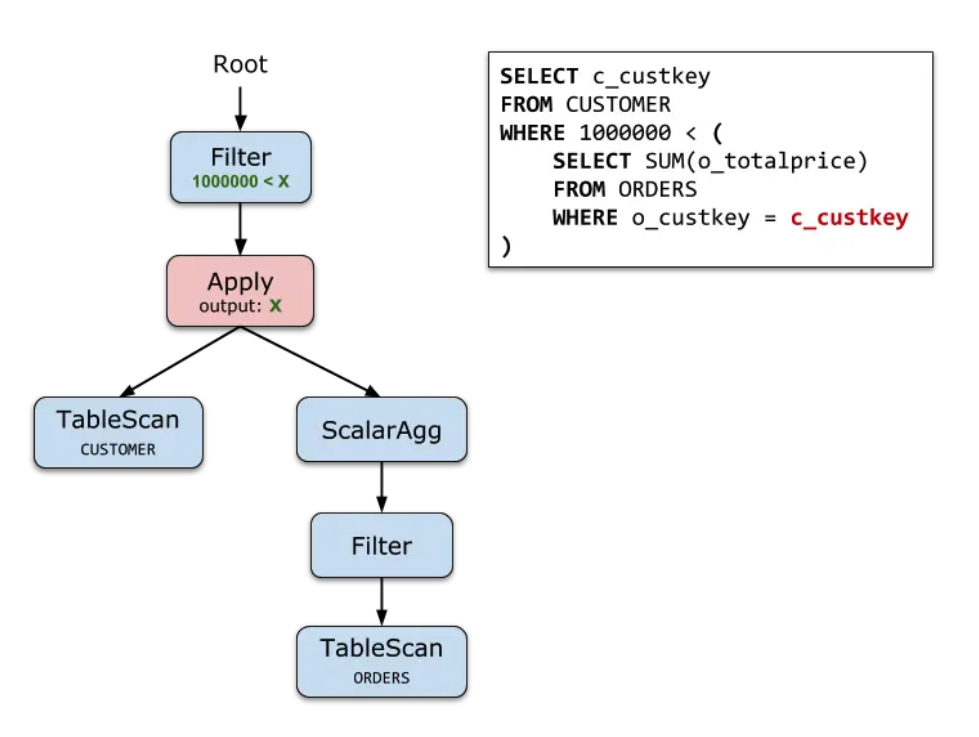
FROM ORDERS

WHERE o\_custkey = c\_custkey

)

我们以上面SQL为例，直观地感受一下，为什么说关联子查询的去关联化是十分必要的。

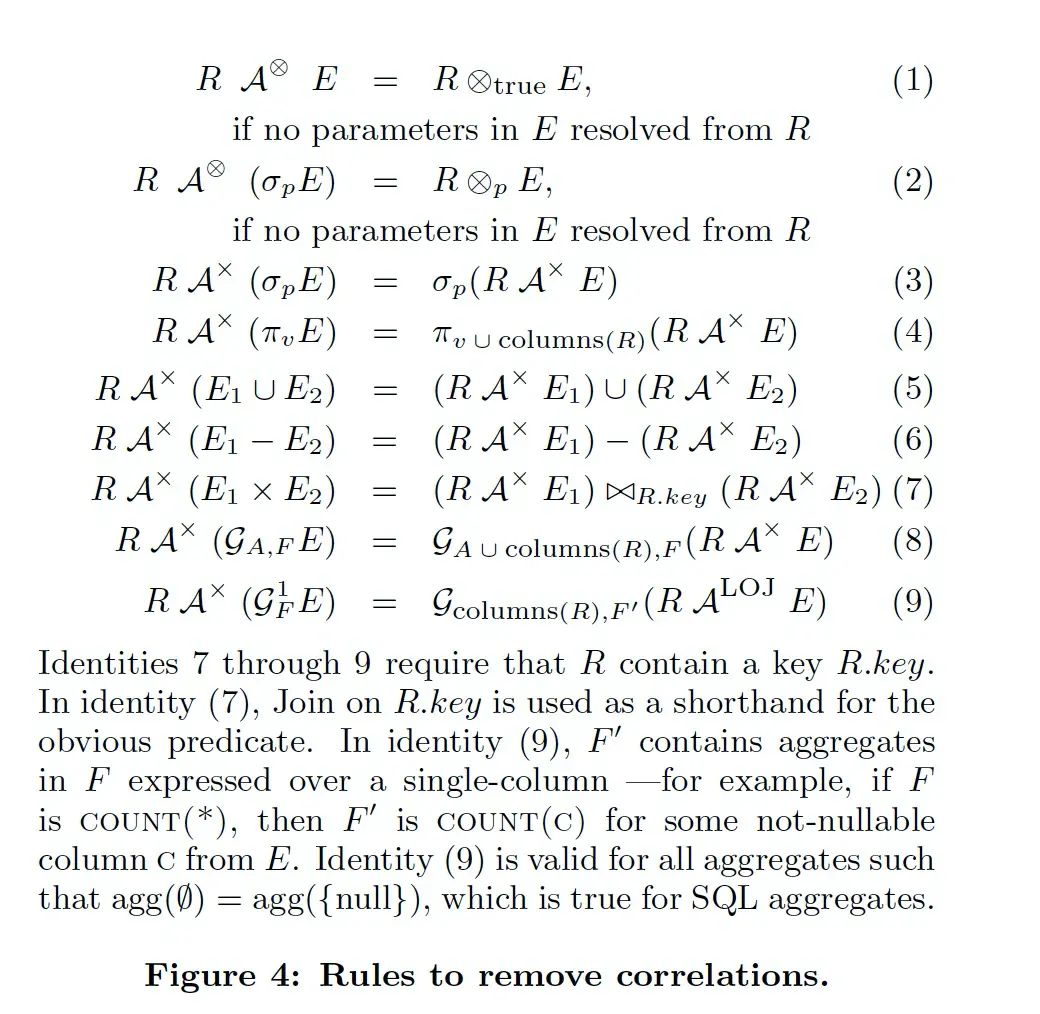
下面是未经去关联化的原始查询计划（Relation Tree）。实际执行时，查询计划执行器（Executor）在执行到Apply时，针对每一行数据，都要去执行一遍Apply右侧的查询树。这样通过嵌套的方式处理关联子查询，处理耗时会随数据量增长呈直线上升状态，如果是多层子查询嵌套，耗时呈指数级上升也不奇怪。为了避免长耗时带来的糟糕体验，必须要将子查询去关联化。



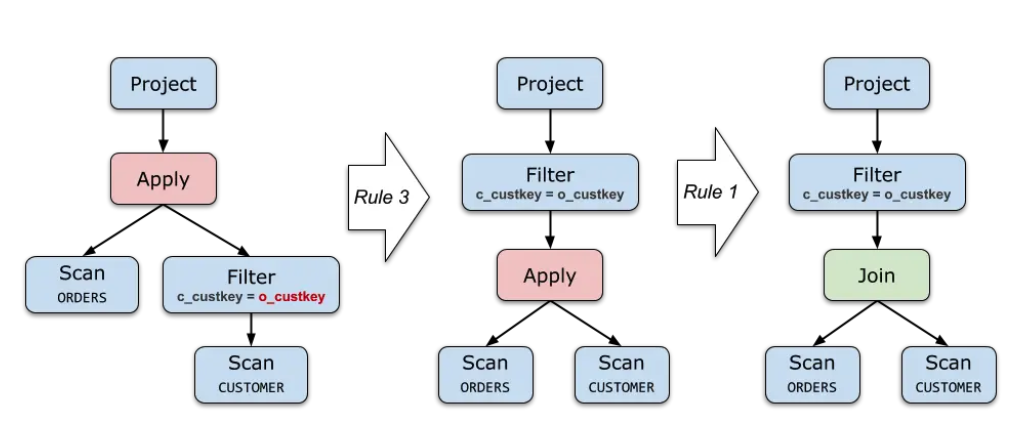
### 基于规则去关联化（Unnesting base on rule）

上世纪八九十年代，SQL标准拓展了子查询的存在范围。掀起了一股研究如何去关联化的热潮，当时研究的主流之一便是基于规则的去关联化。

在01年发布的《Orthogonal Optimization of Subqueries and Aggregation》论文是基于规则去关联化的集大成作，其中总结了9条转换规则：



下图中的两次转换分别对应规则3与规则1。



示例中的转换过程是有前提条件的，Join与Filter间的关系契合AND逻辑运算符是关键。

1. 子查询作为表达式，在Filter中起到过滤的作用
2. 执行计划中从下到上起到过滤作用的节点（例如Filter和Join），上下叠加的逻辑关系是AND
3. 子查询转为SemiJoin后，与上层Filter节点叠加，不会破坏与Filter中抽离后的表达式间的AND关系，但会破坏OR关系（比如下面展示的复杂例子）

基于规则的转换无法处理所有模式的子查询，比如以下几个例子：

// 复杂示例1 disjunction 中的子查询

SELECT \* FROM

supplier a

WHERE

s\_address IN (select s\_name from supplier where a.s\_phone=s\_phone)

OR s\_name='1'

// 复杂示例2 子查询包含聚合与非等值关联项

SELECT \* FROM T1

WHERE AGE >

(SELECT AVG(AGE) FROM T2 WHERE T1.NAME!=NAME)

// 复杂示例3 多层嵌套子查询

...

(SELECT \* FROM T1

WHERE ID IN

(SELECT ID FROM T2 WHERE T3.NAME=NAME))

...

### Magic Set去关联化（Unnesting base on magic）

Magic Set是一种非常古老的数据处理技术，最早应用在演绎数据库（Deductive Database）中。如今在关系型数据库中子查询去关联化上也发挥着很重要的作用。

96年DB2《Cost-Based Optimization for Magic: Algebra and Implementation》将Magic Set作为一个关系代数算子引入到CBO中。15年HyPer基于此发展出了自己的side-ways information passing optimization技术，用于处理所有类型子查询的Unnesting。HyPer官方网站的执行计划展示中，将类似的算子命名为Magic。

考虑这样的一条 SQL

SELECT s.name, e.course

FROM students s, exams e

WHERE s.id=e.sid AND

(s.major = ’CS’ or s.major = ’Games Eng’) AND

e.grade >= (SELECT avg(e2.grade)+1 --one grade worse

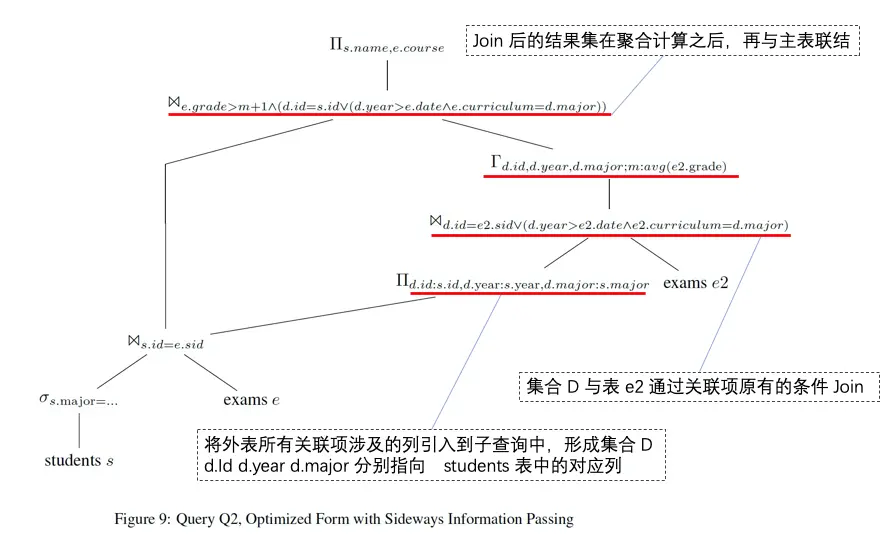
FROM exams e2 --than the average grade

WHERE s.id=e2.sid OR --of exams taken by

(e2.curriculum=s.major AND --him/her or taken

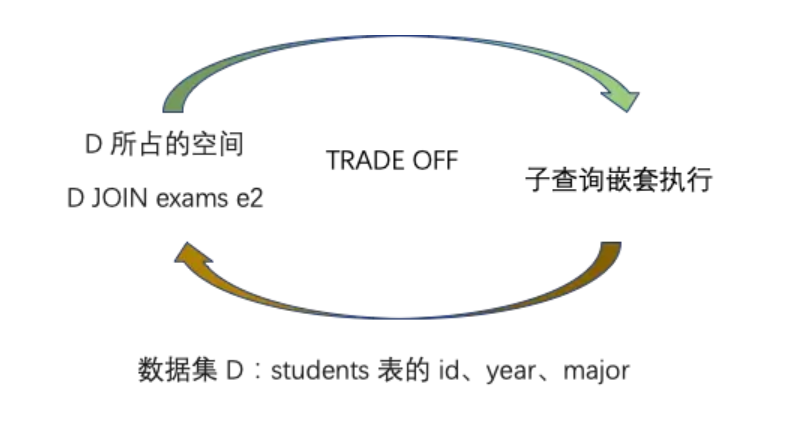
s.year>e2.date)) --by elder peers

这个SQL的Unnesting主要难在有非等值关联项s.year>e2.date，这将导致无法在关联项所在的filter层避免Join计算。



△图例来自HyPer的论文《Unnesting Arbitrary Queries》

在HyPer的论文中，尝试将关联项涉及列的数据copy一份，引入到子查询内部。用Join计算来替换关联项，从而实现了去关联化。



从实现上看，是以一部分空间和额外的Join计算为代价，实现了子查询的去关联化。

### Semi Join算子衍生

数据库通常会利Semi Join算子簇表达去关联化后的子查询。这里会遇到另一个问题，Apply与Semi Join的关系代数定义无法完全等价。

semi join ：“半连接”，意味着仅输出一张表的列，另一张表的列不向上层输出

// 复杂示例1 disjunction 中的子查询

SELECT \* FROM

supplier a

WHERE

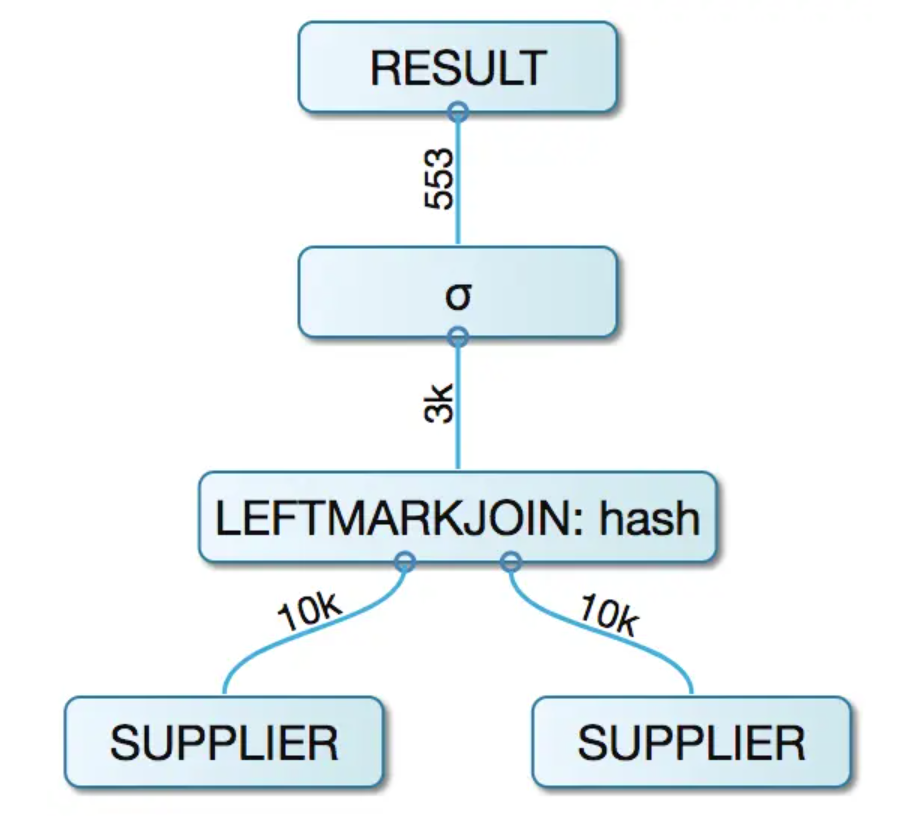
s\_address IN (select s\_name from supplier where a.s\_phone=s\_phone)

OR s\_name='1'

考虑以上SQL，子查询处于OR表达式之间，这会导致无法将其简单的转为SemiJoin，因为Filter与SemiJoin叠加的过滤关系与OR相违。

Mark Join：除了输出连接数据以外，还会保留一个mark位置，用来标记这一行的连接结果(TRUE/FALSE/NULL)。

针对这个场景，HyPer引入了Mark Join替换SemiJoin。



在Mark Join上层的Filter中，会形成markvalue OR sname='1'的表达式。通过增加一列输出的方式避免与OR语义的违背。

采用Mark机制，让Join多输出了一列，不但破坏了Join的关系代数含义，执行层也需要做相应的大规模改造。但它除了能解决上面示例中的OR子查询以外，对Project、Scalar类子查询也可支持，这种做法很有借鉴意义。

截至HyPer 2015年的论文《Unnesting Arbitrary Queries》发出为止，传统的数据库厂商如SQL Server 、Oracle都不支持Project中的非Scalar子查询，对于复杂子查询的Unnesting支持也十分有限，HyPer作为第一个号称可以Unnesting所有子查询的数据库，确实是一个非常激进的实践者。

## 解嵌套子查询

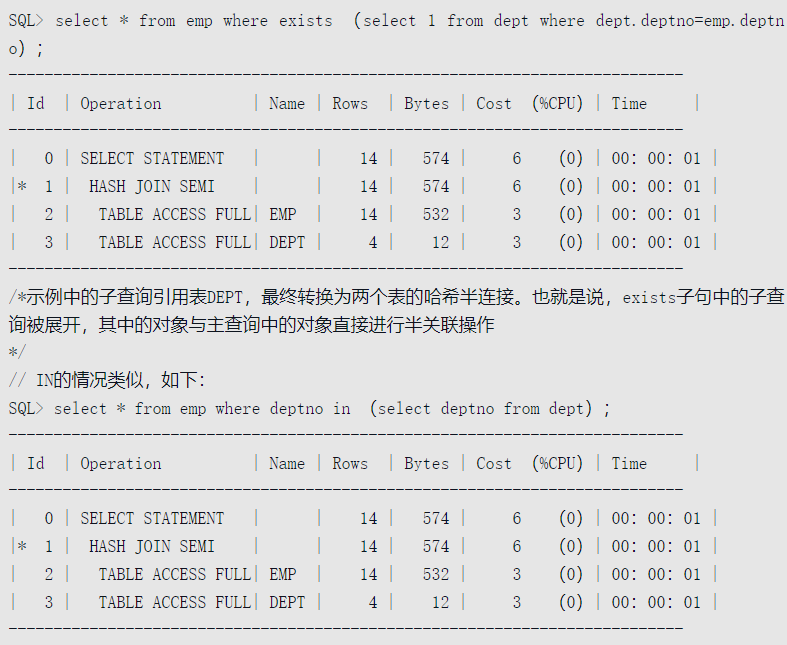
解嵌套子查询是指在对存在嵌套子查询的复杂语句进行优化时，查询转换器会尝试将子查询展开，使得其中的表能与主查询中的表关联，从而获得更优的执行计划。部分子查询反嵌套属于启发式查询转换，部分属于基于代价的转换。

系统中存在一个参数来控制解嵌套子查询——\_unnest\_subquery。参数\_unnest\_subquery在8i中的默认设置是false，从9i开始其默认设置是true。然而9i在非嵌套时不考虑成本。只有在10g中才开始考虑两种不同选择的成本，并选取成本较低的方式。当从8i升级到9i时，可能想阻塞某些查询的非嵌套。利用子查询中的no\_unnest提示可以完成这一点。在8i和9i中，如果star\_transformation\_enabled=true，则非嵌套时被禁用（即使用了提示）。在11g环境下还受优化器参数\_optimizer\_unnest\_all\_subqueries控制。此外，提示UNNEST/NO\_UNNEST可以控制是否进行解嵌套。

下面我们通过几个示例看看解嵌套子查询。

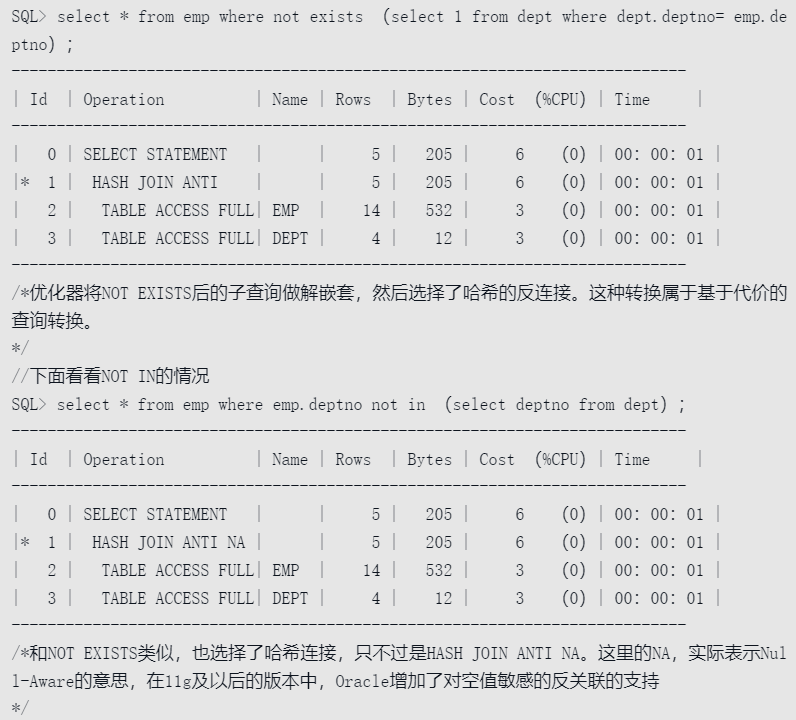
### IN/EXISTS转换为SEMI JOIN

1. IN/EXISTS转换为SEMI JOIN：



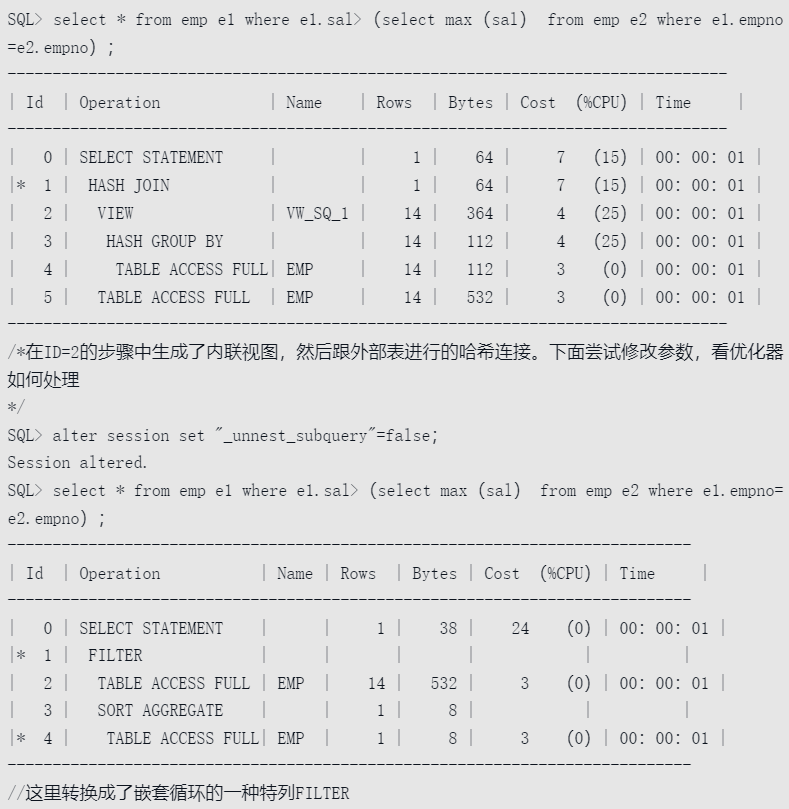
### IN/EXISTS转换为ANTI JOIN

1. IN/EXISTS转换为ANTI JOIN：



### 关联子查询的解嵌套（内联视图）

1. 关联子查询的解嵌套：在对于关联子查询的解嵌套过程中，会将子查询构造出一个内联视图，并将内联视图与主查询的表进行关联。这个操作可以通过参数\_unnest\_subquery来控制。这种转换属于启发式查询转换。



## 子查询转换/重写

Oracle 2009年发表的《Enhanced subquery optimizations in Oracle》向大家展示了数量繁多的子查询重写优化。这些技术基于TPC-H中的子查询做了针对性的优化，估计是Oracle的程序员在TPC-H打榜时写的。

这些重写技术对参数的提取和推导有很高要求。随便列一下其中的内容：

### 子查询合并

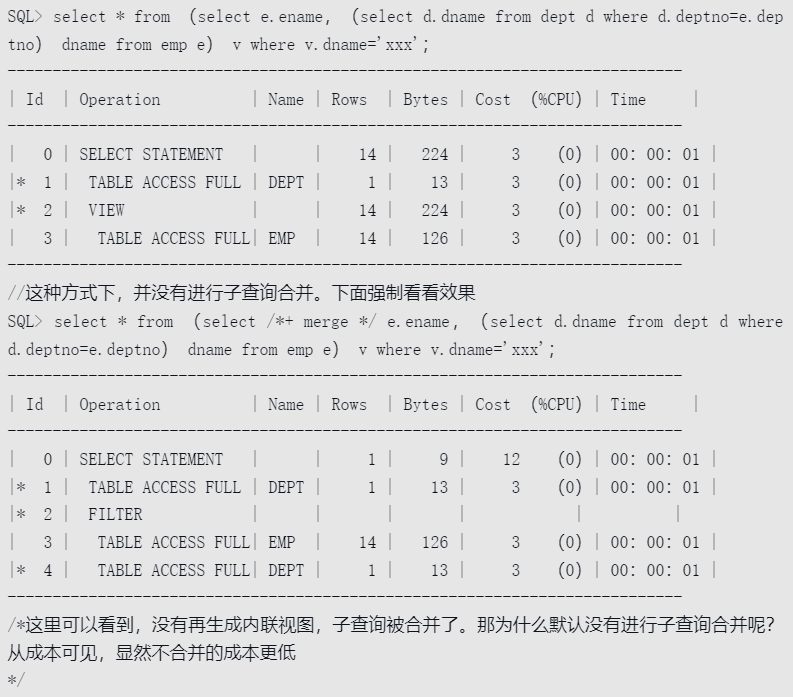
子查询合并是指优化器不再单独为子查询生成执行计划，而是将子查询合并到主查询中，最终为合并后的结果生成一个最优的执行计划。可以通过参数\_simple\_view\_merging或者提示MERGE/NO\_MERGE来控制是否开启、关闭子查询合并。

根据子查询的复杂程度，子查询可分为简单子查询、复杂子查询。

所谓简单子查询，是指可以简单将子查询字段投影到外部的情况。对于这种情况，优化器采取的是启发式策略，即满足条件下就行合并。

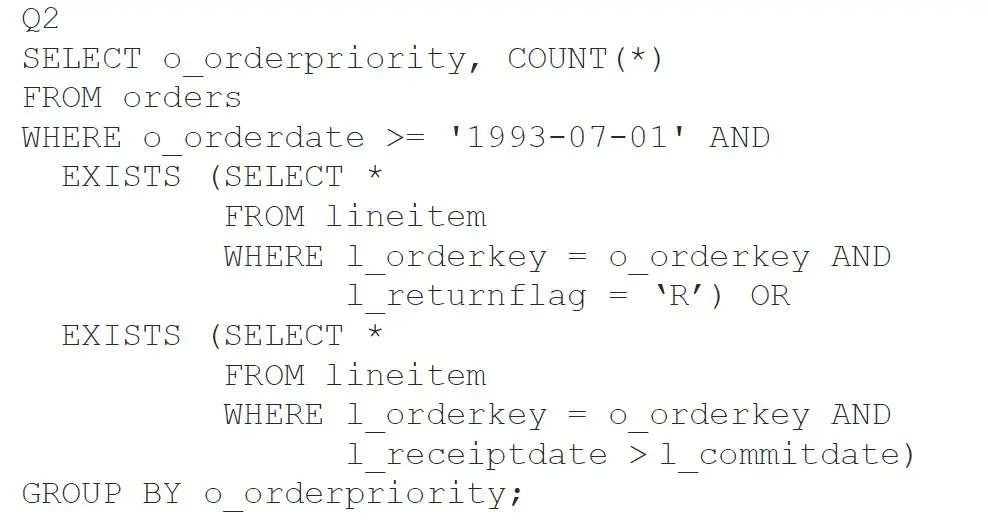
而复杂子查询是指存在分组行数的情况。针对这种情况，优化器采取的是基于代价的策略，最终是否转换取决于成本。当然还有一些子查询是无法进行合并的。

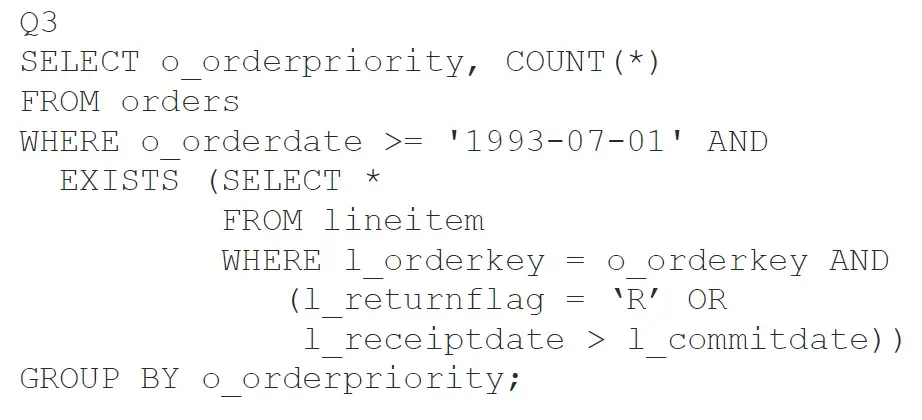
下面通过几个示例看一下。

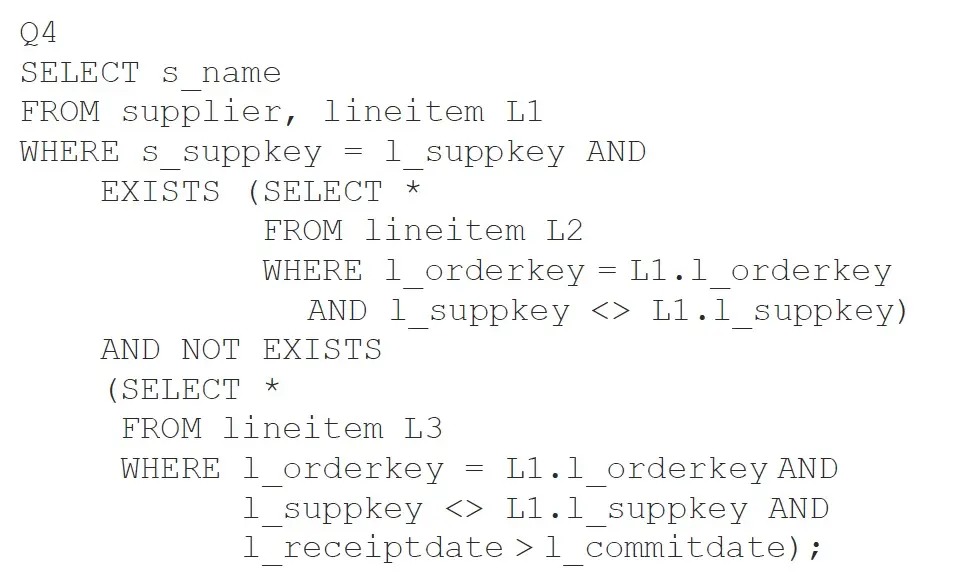


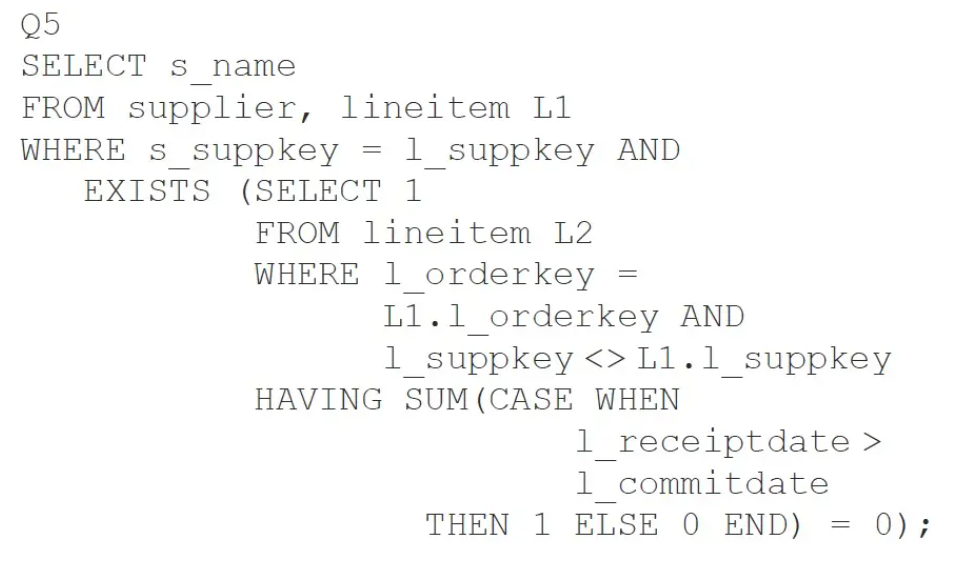
#### 子查询间的合并

将多个类似子查询合并的技术，如下图的Q2转Q3，Q4转Q5



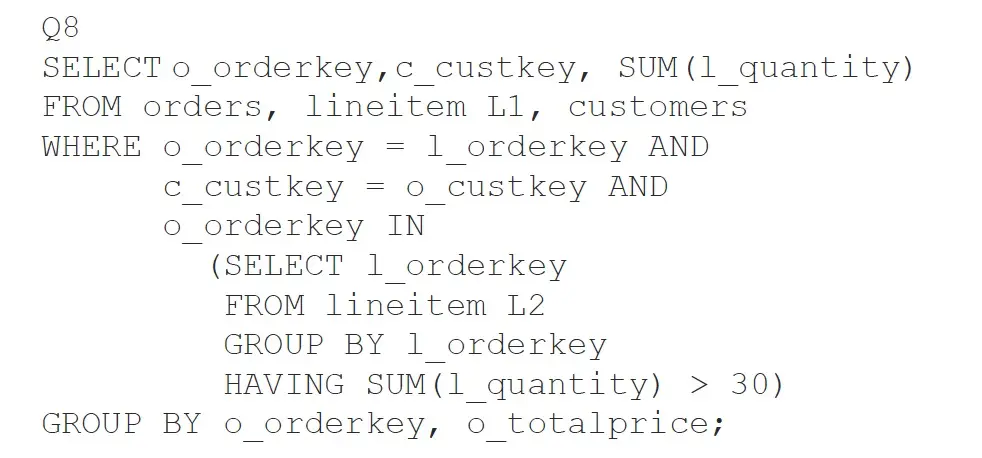


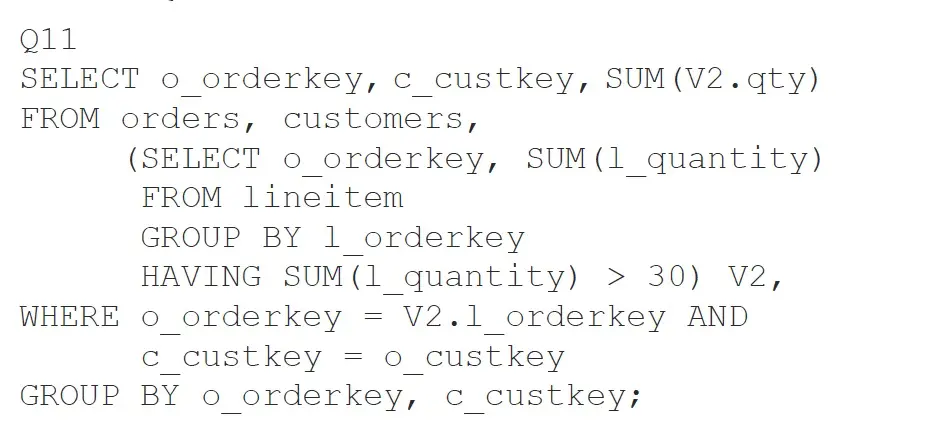




#### 子查询与主表合并

子查询除了与类似的子查询合并以外，还可以与主表合并。例如 Q8 转 Q11





### 窗口函数优化

最早IBM在《WinMagic : Subquery Elimination Using Window Aggregation》 一文中提出了基于窗口函数的子查询优化技巧。而Oracle也将窗口函数改写作为其子查询重写技术的代表之一。

窗口函数优化：查询改写（RBO）阶段触发。**在满足一定条件时，将带agg 的连接关系转化为窗口函数**。是为了避免全表扫描而应用的连接（或子查询）重写技术。

简单讲就是带Agg的子查询被重写成了窗口函数。重写的前提是外部查询块包含所有子查询的表以及过滤条件。例如：

Q1:

SELECT T1.X

FROM T1,T2

WHERE T1.y = T2.y and

T2.name='ming' and

T2.z relop (SELECT AGG(T2.w)

FROM T2

WHERE T2.y = T1.y and

T2.name='ming');

外部查询块有T1和T2两张表，包含了子查询中所有的查询表(T2)；并且同时也包含了所有的过滤条件（T2.name='ming'）。relop是关系操作符的简写。

满足以上条件之后就可以将Q1重写为Q2：

Q2:

SELECT T1.x

FROM T1,(

SELECT AGG (T2.w) OVER (PARTITION BY y) AS win\_agg,

z,

y

FROM T2

WHERE T2.name='ming' ) V

WHERE T1.y=V.y and

V.z relop win\_agg

如果T1与T2的连接是无损连接（lossless join）的话，Q1可以转为Q3：

Q3:

SELECT V.x

FROM (

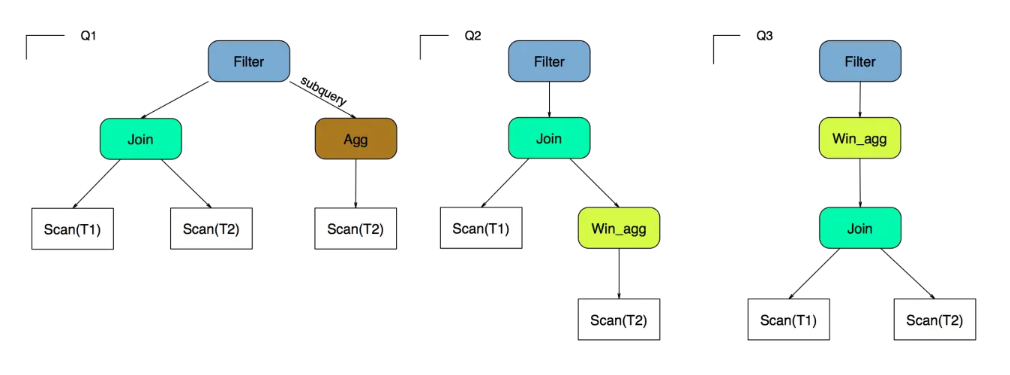
SELECT T1.x, T2.z,

AGG (T2.w) OVER (PARTITION BY T2.y) AS win\_agg

FROM T1, T2

WHERE T1.y = T2.y ) V

WHERE V.z relop win\_agg



关于lossless join这里不做展开讨论，实际上Q2和Q3之间的区别可以理解为Agg与Join的reorder。

在将Q1重写为Q2之后，会在CBO中根据Cost判断是否需要转化为Q3的形式。这里举个具体的例子Q4（TPC-H Q17）：

Q4:

SELECT sum(l\_extendedprice) / 7.0 AS avg\_yearly

FROM

lineitem,

part

WHERE

p\_partkey = l\_partkey

AND p\_brand = 'Brand#23'

AND p\_container = 'MED BOX'

AND l\_quantity < (

SELECT 0.2 \* avg(`l\_quantity`)

FROM

lineitem

WHERE

l\_partkey = p\_partkey

);

在50G的场景下，各个plan的计算量如下所示：



原始执行计划重写为窗口函数后，明显可以看到减少了一次10^8级别的扫描。

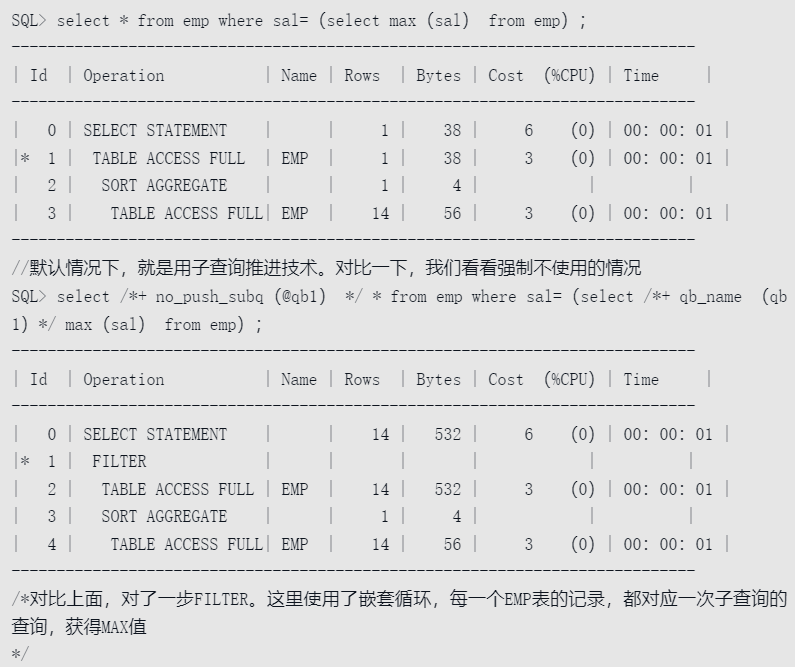
对比重写为窗口函数的两个执行计划，从处理的行数可以看到Plan2窗口函数的Cost要小很多。但要转成Plan2的情况需要一些前提条件，类似Agg与Join reorder的判断。在lossless join的场景下是可以直接转为Plan2的，Q4 part与lineitem之间是外键连接，因此是符合这个条件。

另外Plan2的Join利用Batch Key Access算法，相当于对lineitem表进行10^4次索引扫描，Cost极低。

窗口函数优化对TPC-H Q2和Q17都有显著的RT提升效果，在分布式数据库中，实测窗口函数+ BKA优化对Q17有近百倍提升。

## 子查询推进

**子查询推进是一项对未能合并或者反嵌套的子查询优化的补充优化技术**。这一技术是在9.2版本引入的。通常情况下，未能合并或者反嵌套的子查询的子计划会被放置在整个查询计划的最后执行，而子查询推进使得子查询能够提前被评估，使之可以出现在整体执行计划较早的步骤中，从而获得更优的执行计划。可以通过PUSH\_SUBQ/NO\_PUSH\_SUBQ来控制。

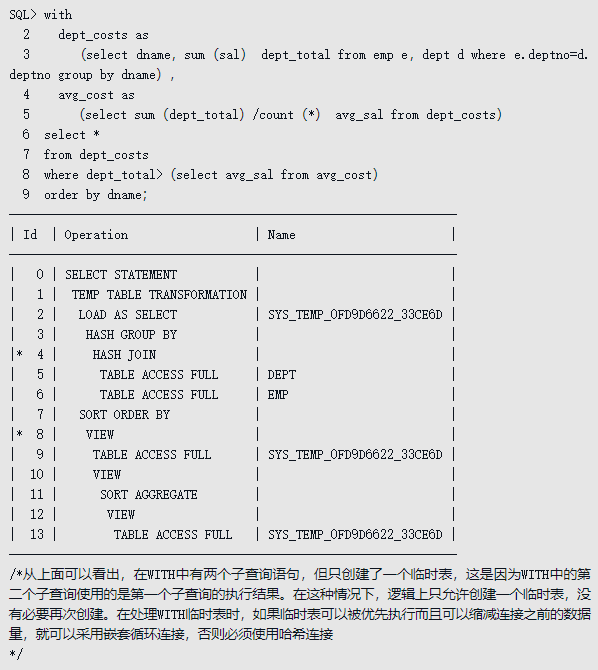


## 子查询分解

所谓子查询分解，是指由WITH创建的复杂查询语句存储在临时表中，按照与一般表相同的方式使用该临时表的功能。

从概念上来看它与嵌套视图比较类似，但各自有其优缺点。优点在于子查询如果被多次引用，使用嵌套视图就需要被执行多次，尤其在海量数据中满足条件的结果非常少得情况下，两者差别很明显。

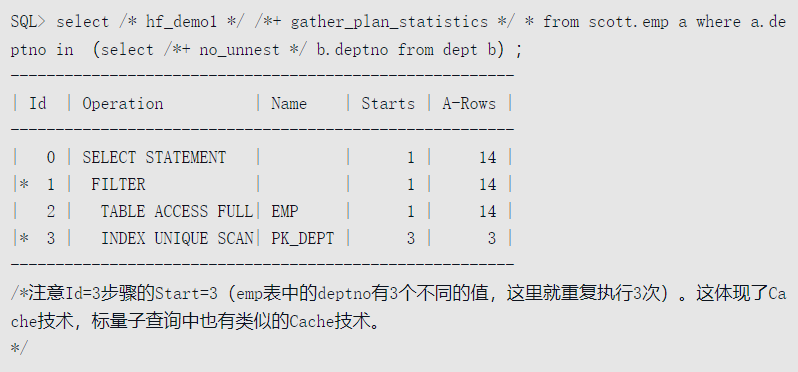
使用WITH子查询的优点就在于其复杂查询语句只需要执行一次，但结果可以在同一个查询语句中被多次使用。缺点是使用WITH子查询，由于不允许执行查询语句变形，所以无效的情况也比较多。尤其是WITH中的查询语句所创建的临时表无法拥有索引，当其查询结果的数据量比较大的时候，很可能会影响执行效率。



## 子查询缓存

针对某些子查询操作，优化器可以将子查询的结果进行缓存，避免重复读取。这一特性在FILTER型的子查询或标量子查询中都能观察到。

示例：



## 执行陷阱

查询优化并不是子查询技术的全部，从实现角度分析，执行过程中的陷阱更加防不胜防。

### Count

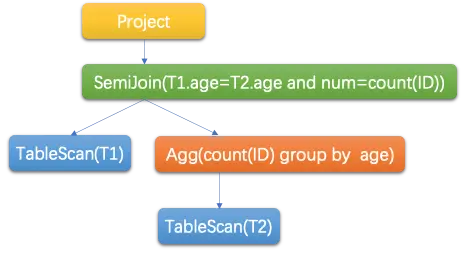
Count陷阱主要存在于没有group by的Count子查询中。考虑以下SQL：

SELECT \* FROM T1

WHERE T1.NUM = (SELECT COUNT(ID) FROM T2

WHERE T1.AGE=AGE)

如果将其转为以下的执行树，当T2的某age数量为0时，SemiJoin的连接会因为T1.age=T2.age表达式结果为Null，从而无法输出正确的结果。本质上的原因是由于COUNT聚合函数的特殊性导致。想要解决必须要Join节点输出Null行，类似Left类型的特性。



其它的聚合函数也有些不同的问题，比如说>ALL类子查询无法直接转化为>MAX。因为在空结果集的状态下，>ALL返回TRUE，而>MAX返回FALSE。

### Null-Aware Anti Join

观察以下两个子查询，想要输出User表中不同名或不同年龄的人：

// SQL1

SELECT \* FROM USER T1

WHERE AGE NOT IN (SELECT AGE FROM USER T2

WHERE T1.NAME=T2.NAME);

// SQL2

SELECT \* FROM USER T1

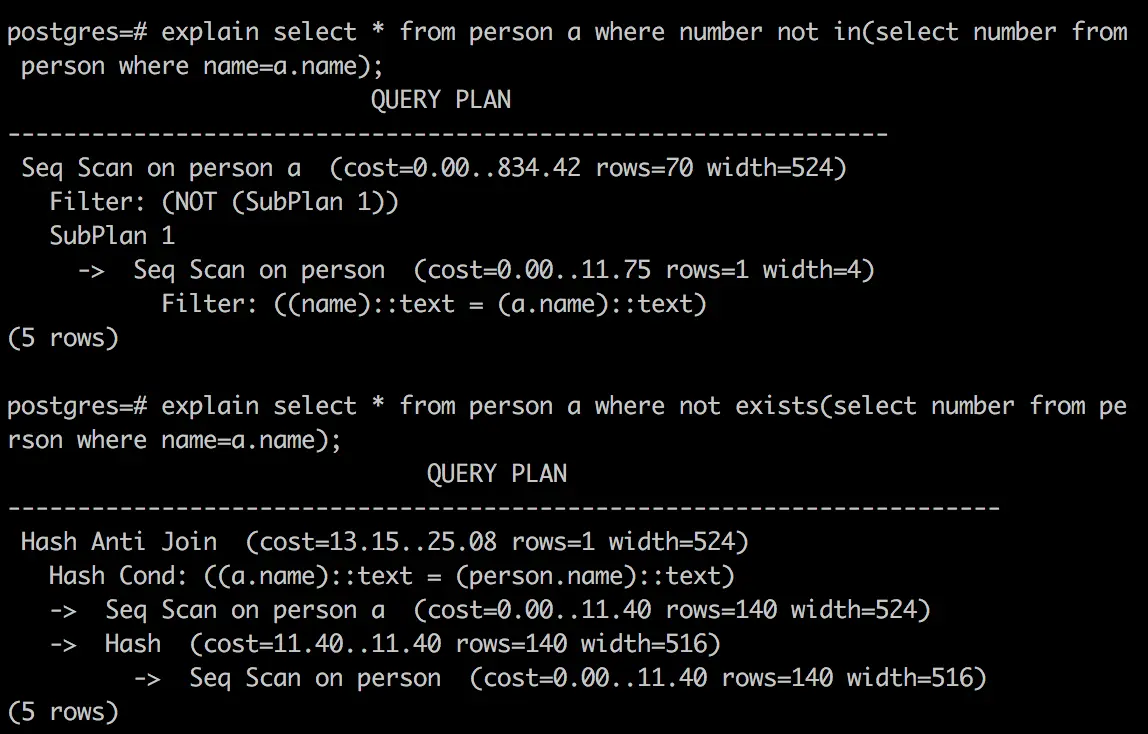
WHERE NOT EXISTS (SELECT 1 FROM USER T2

WHERE T1.NAME=T2.NAME AND T1.AGE=AGE);

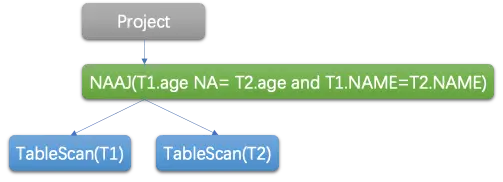
表面看上去两个子查询是等价的，实际上假如USER表中有一行age为NULL的数据，SQL1不会输出，SQL2会将其输出。

NOT EXISTS等价于Anti Join，Anti Join的执行器在处理on条件时，如果执行结果为null，则认为没有匹配。问题在于，not in的子查询的操作数也转成了on条件之一，与关联项通过and连接。这时就会出现多输出的正确性问题。

那么这种情况下，可以选择不将NOT IN子查询转为连接，例如PG在处理NOT IN子查询保留了原来的处理方式。



但还有一种选择，Oracle在《Enhanced Subquery Optimizations inOracle》一文中提到，他们采用了一种新的算子，Null-Aware Anti Join（NAAJ）来处理。还是以SQL1为例，NAAJ的处理算法如下：



如果T2是空集，则返回所有行

如果T2的Age有任意一行为Null，则不返回任何行

如果T1的Age在某行值为Null，则不返回该行

对于T1的每一行，如果NA条件执行为Null或TRUE，则不返回该行；否则返回。

**注意：**

NAAJ中的NA条件是经过反转的，NOT IN -> NA= >ALL  ->  NA<=

# 子查询特殊问题

## 空值问题

首先值得关注的问题是，在NOT IN子查询中，如果子查询列有空值存在，则整个查询都不会有结果。这可能是跟主观逻辑上感觉不同，但数据库就是这样处理的。因此，在开发过程中，需要注意这一点。

第二个值得关注的是，在11g之前，如果主表和子表的对应列未同时有NOT NULL约束，或都未加ISNOT NULL限制，则Oracle会走FILTER。11g有新的ANTI NA（NULL AWARE）优化，可以正常对子查询进行UNNEST。

## OR问题

对含有OR的Anti Join或Semi Join，注意有FILTER的情况。如果FILTER影响效率，可以通过改写为UNION、UNION ALL、AND等逻辑条件进行优化。优化的关键要看FILTER满足条件的次数。

## [NOT]IN/EXISTS问题

1. **IN/EXISTS**

从原理来讲，IN操作是先进行子查询操作，再进行主查询操作。EXISTS操作是先进行主查询操作，再到子查询中进行过滤。

IN操作相当于对inner table执行一个带有distinct的子查询语句，然后得到的查询结果集再与outertable进行连接，当然连接的方式和索引的使用仍然等同于普通的两表连接。EXISTS操作相当于对outer table进行全表扫描，用从中检索到的每一行与inner table做循环匹配输出相应的符合条件的结果，其主要开销是对outer table的全表扫描（full scan），而连接方式是nested loop方式。

**当子查询表数据量巨大且索引情况不好（大量重复值等），则不宜使用产生对子查询的distinct检索而导致系统开支巨大的IN操作；反之当外部表数据量巨大（不受索引影响）而子查询表数据较少且索引良好时，不宜使用引起外部表全表扫描的EXISTS操作。如果限制性强的条件在子查询，一般建议使用IN操作。如果限制性强的条件在主查询，则使用EXISTS操作。**

1. **NOT IN/EXISTS**

在子查询中，NOT IN子句将执行一个内部的排序和合并。无论在哪种情况下，NOT IN都是最低效的（因为它对子查询中的表执行了一个全表遍历）。**为了避免使用NOT IN，可以把它改写成外连接（Outer Joins）或NOT EXISTS**。

# 应用

## 多例过滤

子查询匹配两个值，机MySQL独有的多例过滤方式：

SELECT a.user\_name,b.timestr,kills

FROM user1 a

JOIN user\_kills b ON a.id = b.user\_id

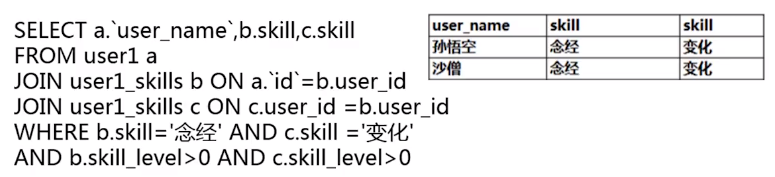
WHERE (b.user\_id,b.kills) IN (

SELECT user\_id,MAX(kills) FROM user\_kills GROUP BY user\_id

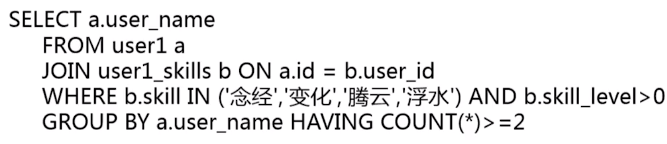
);

## 同属性多值过滤

### JOIN方式实现



### GROUP BY方式实现



## 多属性查询

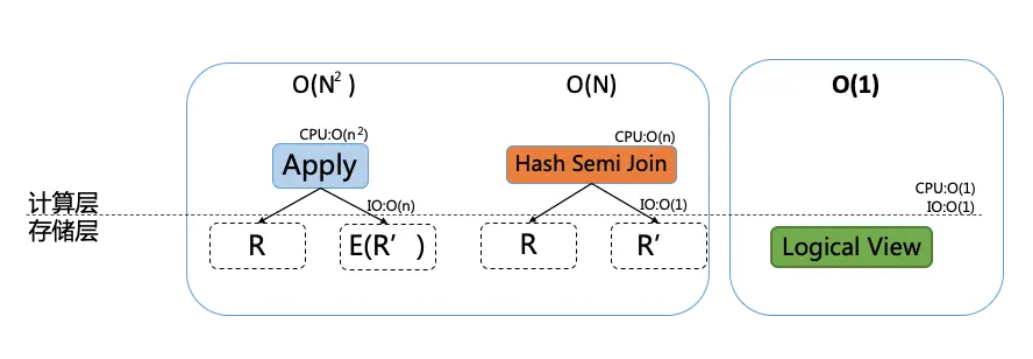
# 分布式数据库子查询

分布式数据库的优势在于拥有更多的计算和存储资源可供调用，但由于数据分布在不同的节点，节点间的数据传输 IO 很容易成为性能瓶颈。执行子查询时尤其要考虑如何扬长避短。

利用集群的计算性能

减少网络 IO 开销

## 去关联化更加重要



子查询的嵌套执行会导致网络IO随数据量上涨。除了执行缓慢以外，还会导致整体的资源消耗变大，系统容量变低。

去关联化除了可以将O(N^2) 的时间复杂度消除为O(N) ，可以避免大量的网络IO开销。

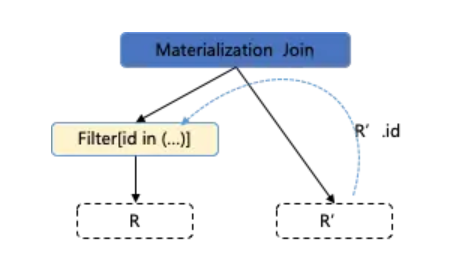
如果数据的分布满足条件，子查询还可以整体下推到存储节点进行计算，充分利用了集群的计算性能以外，也避免了大量Scan数据在网络层的传输。

所以说在分布式系统中，如何将更多的子查询转为连接执行尤为重要。

注：转为连接操作才有可能下推到存储节点计算，否则需要在上层做计算了。

## 物化的不同运用

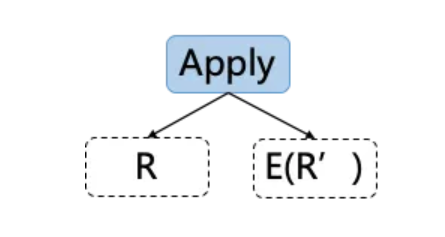
物化的思路除了可以用于关联项消除以外，还可以用于削减连接计算中的网络传输数据。



如图所示当连接一端的数量量较小时，可以将其全量或部分捞出（Semi可以部分，Anti必须全量），作为常量代入到另一侧的执行计划中去处理。

## Apply的PreFilter

即便是无法转为连接的子查询，依然可以通过子查询的逻辑特性来削减网络 IO 的数据量。



1、SEMI：表达式通过OR连接

假设子查询的condition为E，Semi类型的Apply算子的子查询一侧都可以增加以下过滤条件：

E OR (E IS NULL)对于E1 OR E2 OR E3 ...来讲En为FALSE的表达式是可以忽略的：

E OR (E IS NULL)

对于E1 OR E2 OR E3 ... 来讲

En为FALSE的表达式是可以忽略的

2、ANTI：表达式通过AND连接

如果是 anti 类型，则可以增加以下过滤条件

// E' = NOT EE' OR (E' IS NOT NULL)       对于 E1 AND E2 AND E3 ... 来讲En 为 TRUE 的表达式是可以忽略的

// E' = NOT E

E' OR (E' IS NOT NULL)

对于 E1 AND E2 AND E3 ... 来讲

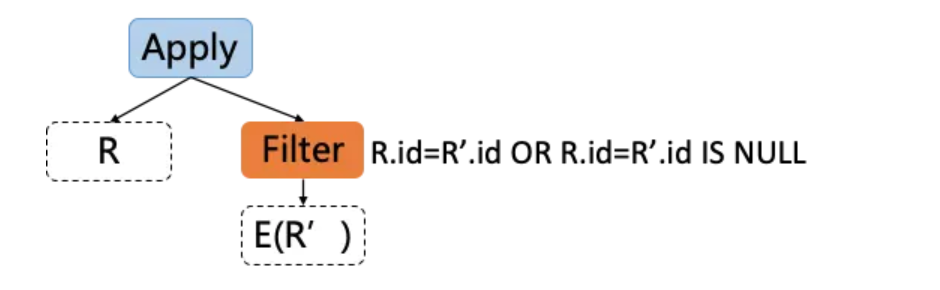
En 为 TRUE 的表达式是可以忽略的

分别举例：

SELECT \* FRMO R

WHERE ID IN (SELECT ID FROM R'

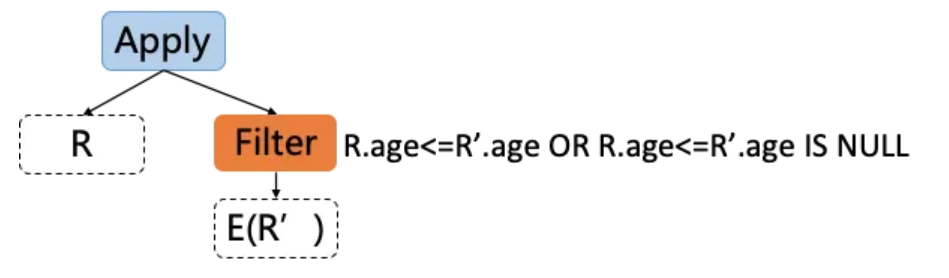
...)



SELECT \* FRMO R

WHERE AGE>ALL(SELECT AGE FROM R'

...)



注意：Anti下推的表达式需要反转

通过这样的方式，可以在子节点提前过滤掉不需要的数据。即便在空集的情况下，这个filter下推也是成立的。

## Cache

Cache缓存：在无法转化为连接的场景下，关联子查询的嵌套执行会导致极大的性能开销。尤其在分布式场景下还会随数据量增长而导致更多IO。在多层嵌套的场景下，几千的数据量就会导致分钟级的查询耗时。合理的使用缓存可以极大避免数据量增长带来的IO消耗，从而让子查询的原语义执行也可以飞快。CBO可以根据数据量判断有必要的话，在执行层增设一层Cache，从而将网络IO降级为内存。

Cache是永不过时的优化技术，在Apply中合理的利用缓存具有化腐朽为神奇的效果。

举个最简单的多层 Apply SQL：

SELECT \*

FROM T1

WHERE 100>

(SELECT col1

FROM T2

WHERE T1.pk = T2.pk

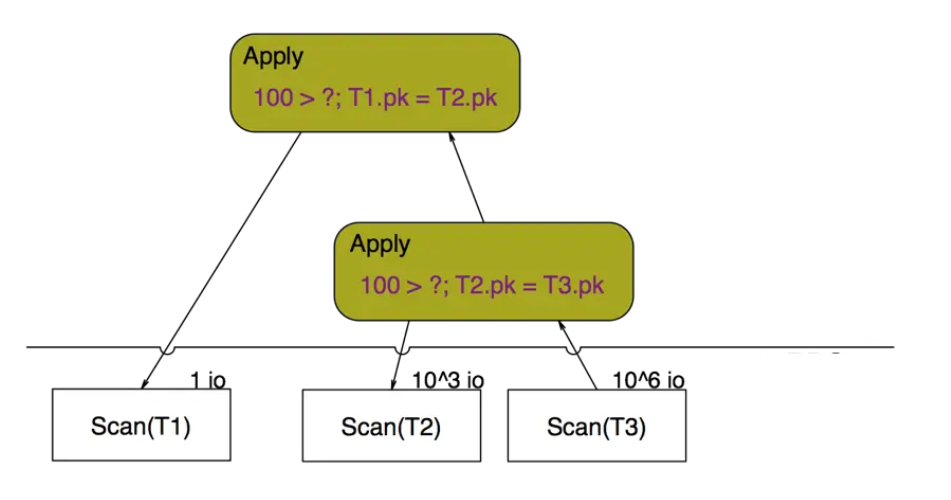
AND 100 >

(SELECT col2

FROM T3

WHERE T2.pk = T3.pk))

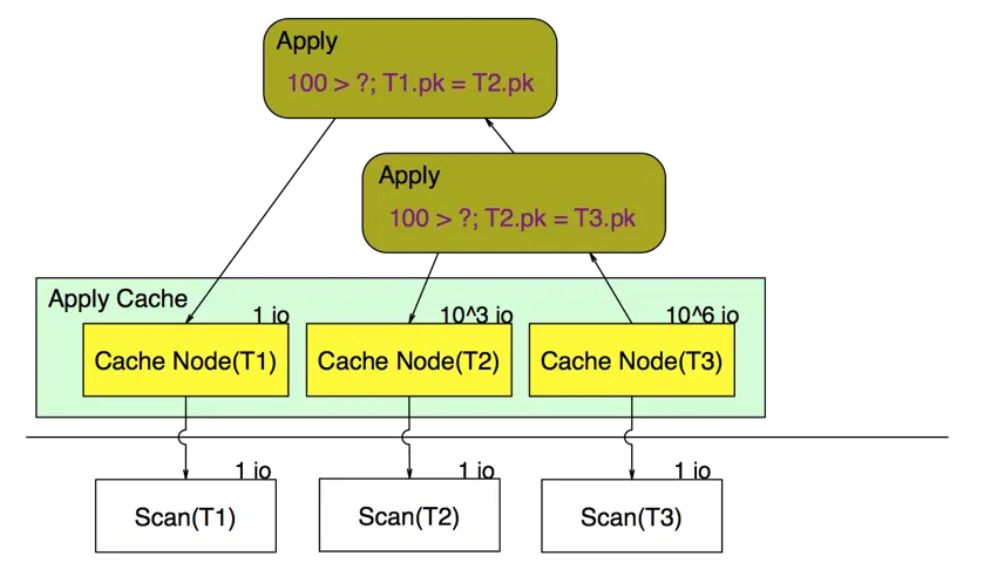
假设T1，T2，T3都为1000行数据，先看一下Apply的执行过程：



T1表作为最外层的主表，只需要扫描一次。扫出的数据有1000行，所以Apply会往T2这边扫扫描1000次；同理，每次扫描T2都需要扫1000次T3。最终T2的扫描次数是1000次，T3的扫描次数是10^6次。

这意味着在千行的数据量场景下，多层的Apply会导致十万乃至百万级的网络IO次数。即便Apply是预期的慢查询，但这种Cost是不可接受的。

由此引入 Cache 最主要的目的在于减少网络IO次数：



如上图所示，虽然到Cache Node的IO次数没有变化，但IO的级别从网络降级到了内存。网络上的IO三张表各自仅进行一次。内存的存取速度与网络天壤之别，这里不再放具体SQL RT数据支撑。