**查询转换**

云和恩墨(北京)信息技术有限公司

技术顾问 燕鑫

http://www.enmotech.com

**文档控制：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **序** | **版本号** | **更改人** | **日期** | **备注** |
| 1 | 1.0版 | 燕鑫 | 2018-05-25 | 初始版本 |
|  |  |  |  |  |
|  |  |  |  |  |

目录

[1. 查询转换的介绍 - 5 -](#_Toc518067369)

[2. 查询转换的分类 - 5 -](#_Toc518067370)

[2.1 子查询展开(Subquery Unnesting) - 5 -](#_Toc518067371)

[2.1.1 SINGLE-ROW Subquery - 5 -](#_Toc518067372)

[2.1.2 将子查询展开 - 8 -](#_Toc518067373)

[2.1.3 将子查询转换为内嵌视图 - 16 -](#_Toc518067374)

[2.1.4 对于子查询展开成本的反思 - 20 -](#_Toc518067375)

[2.1.5 子查询展开总结 - 23 -](#_Toc518067376)

[2.2 视图合并（View Merging） - 23 -](#_Toc518067377)

[2.2.1 简单视图合并 - 23 -](#_Toc518067378)

[2.2.2 外连接视图合并 - 27 -](#_Toc518067379)

[2.2.3 复杂视图合并 - 36 -](#_Toc518067380)

[2.2.4 视图合并总结 - 47 -](#_Toc518067381)

[2.3 ORACLE对IN的处理 - 47 -](#_Toc518067382)

[2.3.1 IN-List Iterator - 47 -](#_Toc518067383)

[2.3.2 IN-List Expansion/OR Expansion - 49 -](#_Toc518067384)

[2.3.3 IN-List Filter - 55 -](#_Toc518067385)

[2.3.4 对in子查询展开 - 55 -](#_Toc518067386)

[2.3.5 ORACLE对IN的处理的总结 - 55 -](#_Toc518067387)

[2.4 连接谓词推入（Join Predicate Pushdown） - 56 -](#_Toc518067388)

[2.4.1 可merge但是no merge的情况 - 57 -](#_Toc518067389)

[2.4.2 没法merge的情况 - 60 -](#_Toc518067390)

[2.4.3 连接谓词推入总结 - 63 -](#_Toc518067391)

[2.5 连接因式分解（Join Factorization） - 64 -](#_Toc518067392)

[2.5.1 单独的UNION ALL sql语句 - 64 -](#_Toc518067393)

[2.5.2 带UNION ALL的视图 - 69 -](#_Toc518067394)

[2.5.3 连接因式分解总结 - 72 -](#_Toc518067395)

[2.6 表扩展（Table Expansion） - 72 -](#_Toc518067396)

[2.6.1 表扩展的介绍 - 72 -](#_Toc518067397)

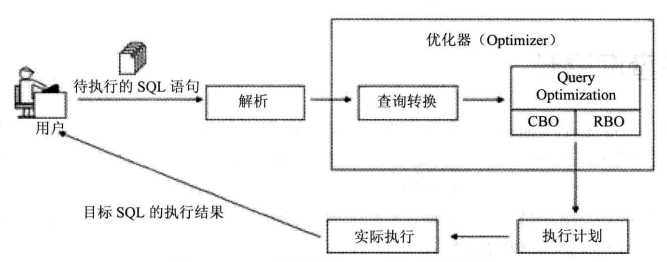
[2.6.2 表扩展+连接因式分解 - 75 -](#_Toc518067398)

[2.6.3 表扩展总结 - 78 -](#_Toc518067399)

[2.7 表移除（Table Elimination） - 78 -](#_Toc518067400)

[3. 查询转换总结 - 79 -](#_Toc518067401)

# 查询转换的介绍

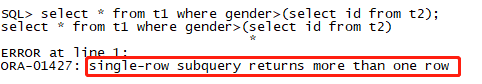


# 查询转换的分类

## 子查询展开(Subquery Unnesting)

### SINGLE-ROW Subquery

single-row的意思就是子查询前是=,<>,>=,>,<,<=，因为子查询这时候只返回一行，如果返回的是多行，会报错，这里就演示一个：



所以将这类子查询划分为single-row subquery。

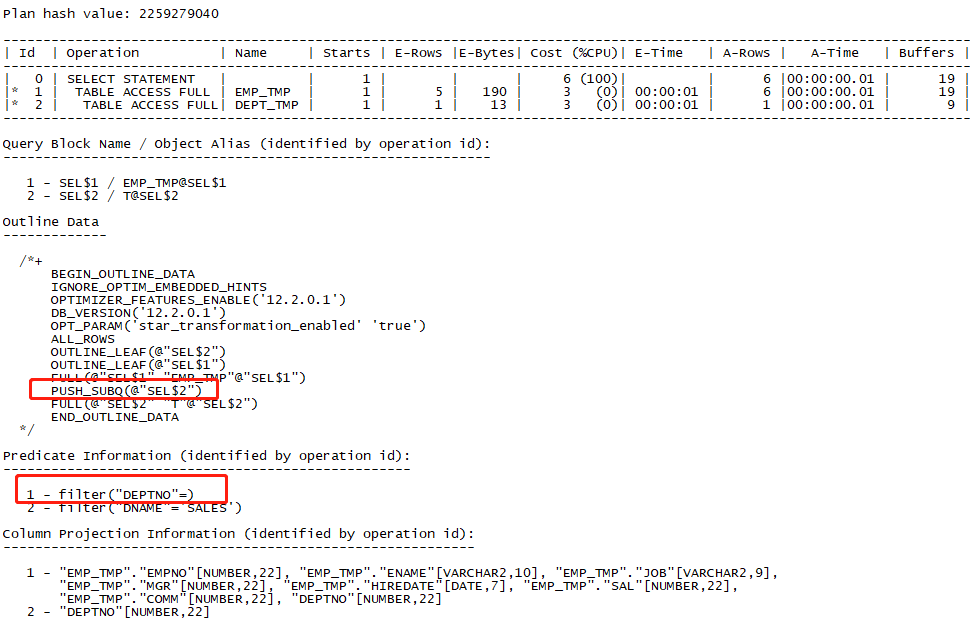
之所以单独列出来，是因为我个人并不认为这是真正意义上的子查询展开，而且也不需要子查询展开。我们直接来看下例子：

SQL> select \* from emp\_tmp where deptno=(select deptno from dept\_tmp where dname='SALES');

SQL> select /\*+LEADING(@"SEL$5DA710D3" "DEPT"@"SEL$2" "EMP"@"SEL$1") USE\_NL(@"SEL$5DA710D3" "EMP"@"SEL$1")\*/ \* from emp\_tmp where deptno=(select deptno from dept\_tmp t where dname='SALES');

SQL> select \* from emp\_tmp where deptno=(select /\*+no\_unnest\*/ deptno from dept\_tmp t where dname='SALES');

这三条sql的执行计划是一样的：

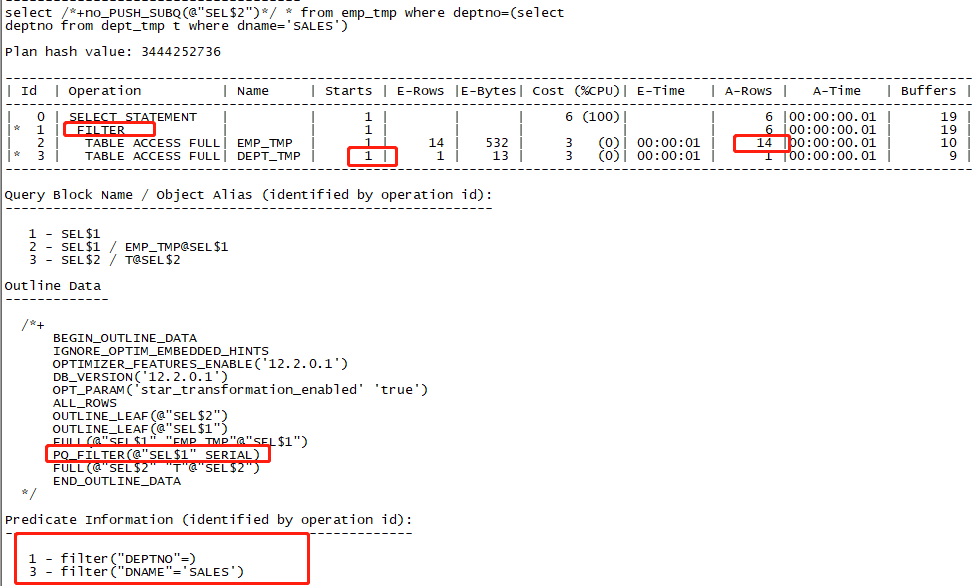


之所以把这三条放在一起比较，也是想说明我的一个观点，我并不认为single-row subquery属于子查询展开的范畴。

事实上，这是另一种手段：push\_subq。这个相关的原理还没弄清楚，但是，如果想让这里子查询单独先执行可以：

SQL> select /\*+no\_PUSH\_SUBQ(@"SEL$2")\*/ \* from emp\_tmp where deptno=(select deptno from dept\_tmp t where dname='SALES');

我们可以看到如下执行计划：



这里很有趣，filter被驱动表的循环次数应是驱动表按条件过滤后的结果集的连接列的唯一值个数，但事实上我们这里看到id=3的starts是1。

而且我们注意比较这俩个执行计划，cost是相同的，buffers是相同的，只是过滤的位置不一样，push\_subq的效果是让子查询直接成为了主表的子步骤。

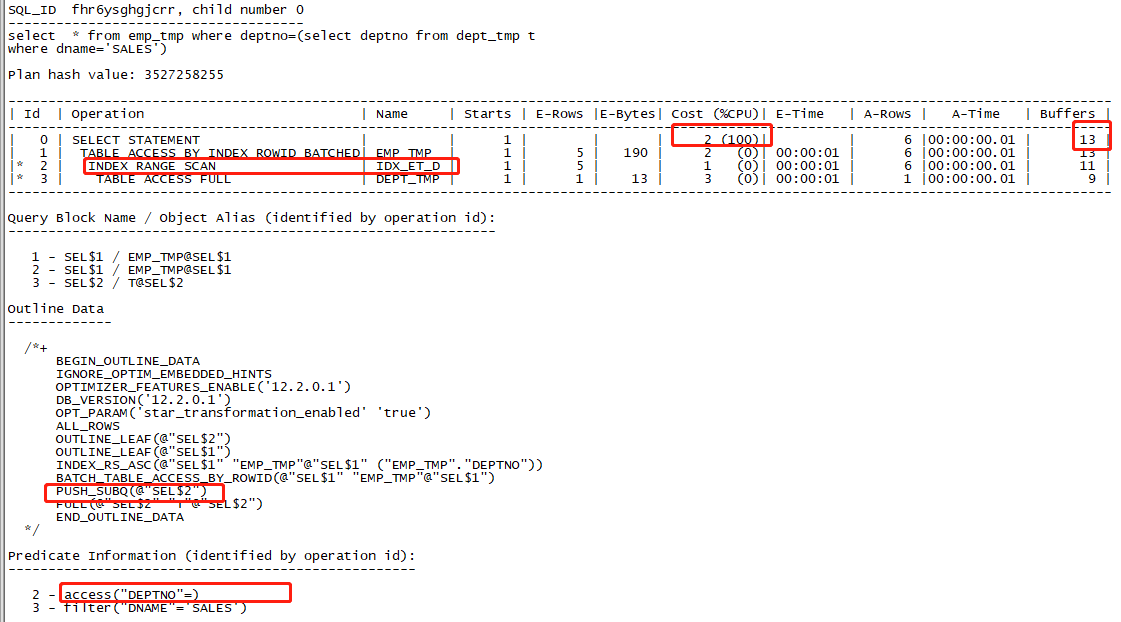
其实这很好理解，对于这种特殊的子查询，其实怎么走都是把子查询作为了一个值来看待。所以filter只是表示这个子查询是没先执行的。并没有真正的去把子查询循环3次（emp\_tmp的deptno列的基数）。

所以，nl也是没有意义的，即使驱动结果集不是子查询，循环只走一次，所以nl hint是无效的。

所以这里也就不深究这种single row子查询了。

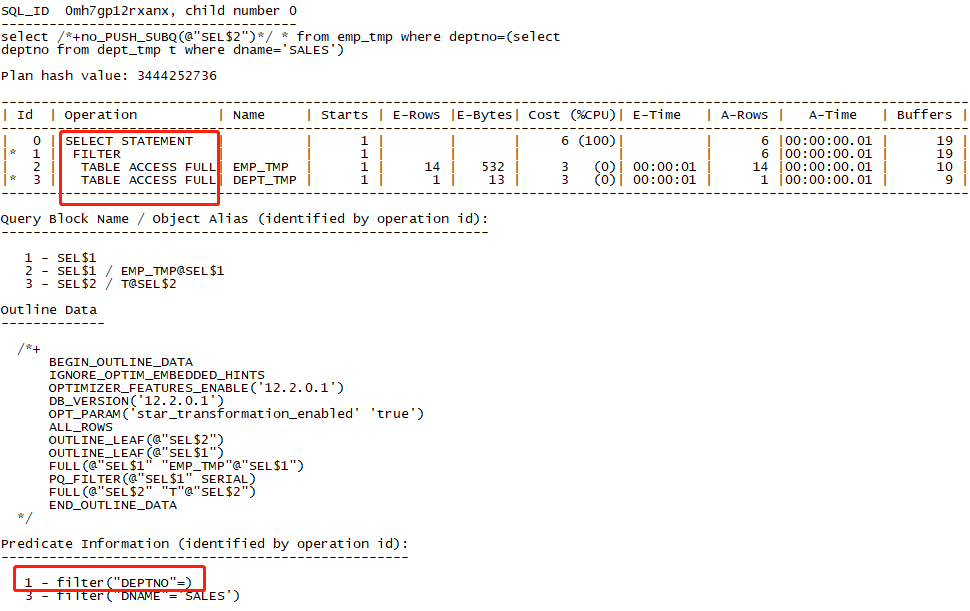
遇到这种子查询，优化就很简单：这就是个过滤条件，把它当普通过滤条件来优化就可以了。

显然，对于这条sql我们应该给emp\_tmp的deptno列加索引：



看到cost降了，逻辑读降了，步骤2的谓词信息也变成了access。

再看一下no\_push\_subq：



这就能看出push\_subq的效果了，目前给我的感觉，就是把子查询当作一个值来传递给主表。后续再进一步研究。

#### Push\_subq的进一步研究

先说结论，目前可以确定有如下俩点：

1. push\_subq的作用，就是将子查询作为主表的一个过滤条件，执行计划的效果就是将其作为主表的一个子步骤。
2. Push\_subq的hint只有在子查询无法展开时（是那种一点展开的办法都没有的），才可能产生作用。

下面我们看第二个结论的实验。

对于子查询可展开的情况，进行push\_subq是没用的，无论你是选择手动让其不展开还是展开。

因为push\_subq本来正常是出现single-row unnesting中的，这本来就是一种长这一副子查询不展开的脸的子查询展开。所以在能左右子查询展开或者不展开的话，就没办法使用这种push\_subq的操作，因为怎么操作都是矛盾的。

现在我们来看实验，我们该实验是与2.1.2的第一个实验对应的。

我们在2.1.2中可以看到，push\_subq不起作用，那么想让它起作用（当然这里需要改变sql语义了），我们加个条件：

select e.empno

from emp\_tmp e

where e.empno = 7369

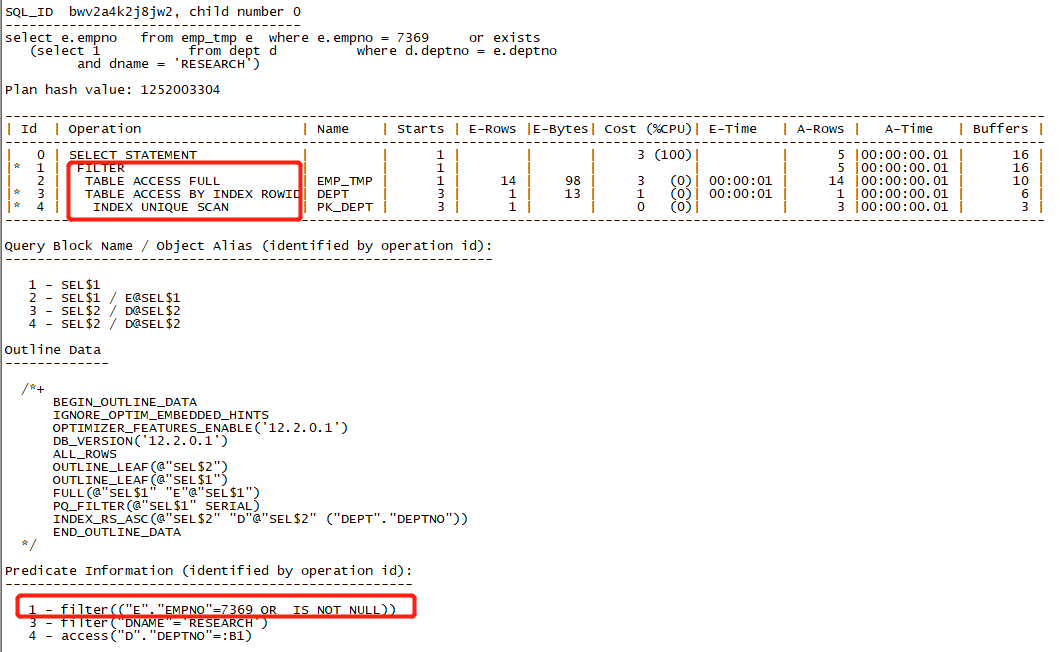
or exists

(select 1

from dept d

where d.deptno = e.deptno

and dname = 'RESEARCH');



我们看到，这个exists是真的展不开，同时我们还发现id=1的谓语信息是：filter(("E"."EMPNO"=7369 OR IS NOT NULL))。对于这一点，会涉及另一个问题，即使OR扩展，所以这一点在OR扩展中说，我们先说push\_subq：

select e.empno

from emp\_tmp e

where e.empno = 7369

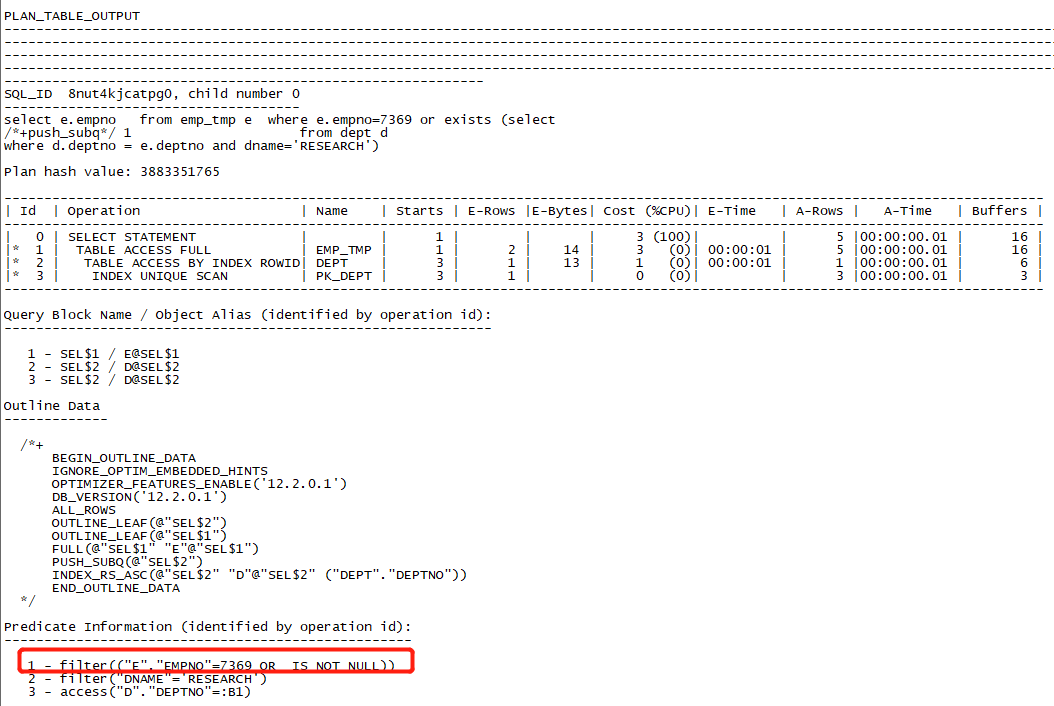
or exists

(select /\*+push\_subq\*/ 1

from dept d

where d.deptno = e.deptno

and dname = 'RESEARCH');



我们看到，现在执行计划的意思就是，CBO先把子查询里面的查询执行了，然后把这一步作为一个主表查询的过滤条件（子步骤）传给主表。

其实我们注意到，俩种执行计划的buffers都是16，所以这种调整几乎没有意义。但是有些情况还是有一点意义：

select e.empno, e.sal , d.loc

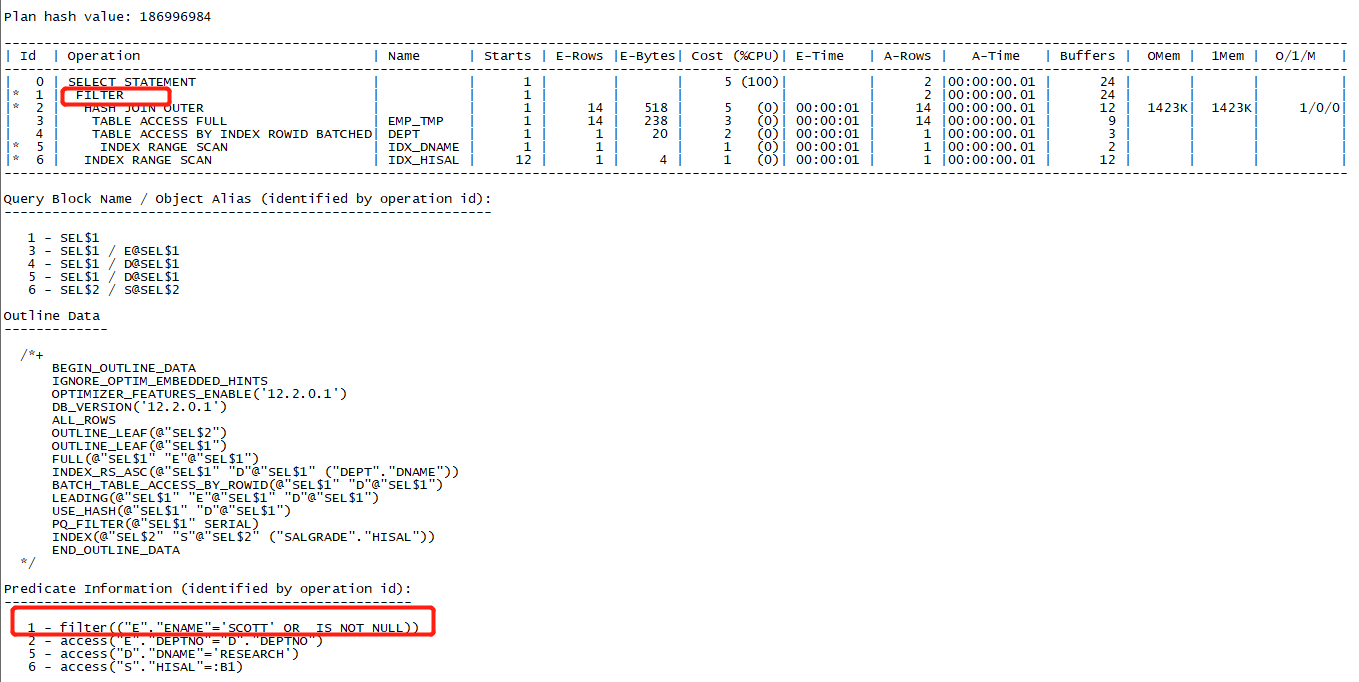
from emp\_tmp e, dept d

where e.deptno = d.deptno(+)

and d.dname(+) = 'RESEARCH'

and (e.ename = 'SCOTT' or exists

(select hisal from salgrade s where s.hisal = e.sal));



我们看这个执行计划，filter是最后做的，这个filter我个人认为，是做了俩件事，然后oracle把它写成了一个：

1. 只有过滤的作用，也就是只有一个孩子的filter，这是对于ename为SCOTT的操作；
2. 高级NL，也就是对于子查询不展开的filter，是有一对孩子的。

现在的顺序其实是外连接，然后对结果集过滤，那我现在就想先对主表过滤，再做外连接咋办？用push\_subq，让其成为一个主表的过滤条件：

select e.empno, e.sal, d.loc

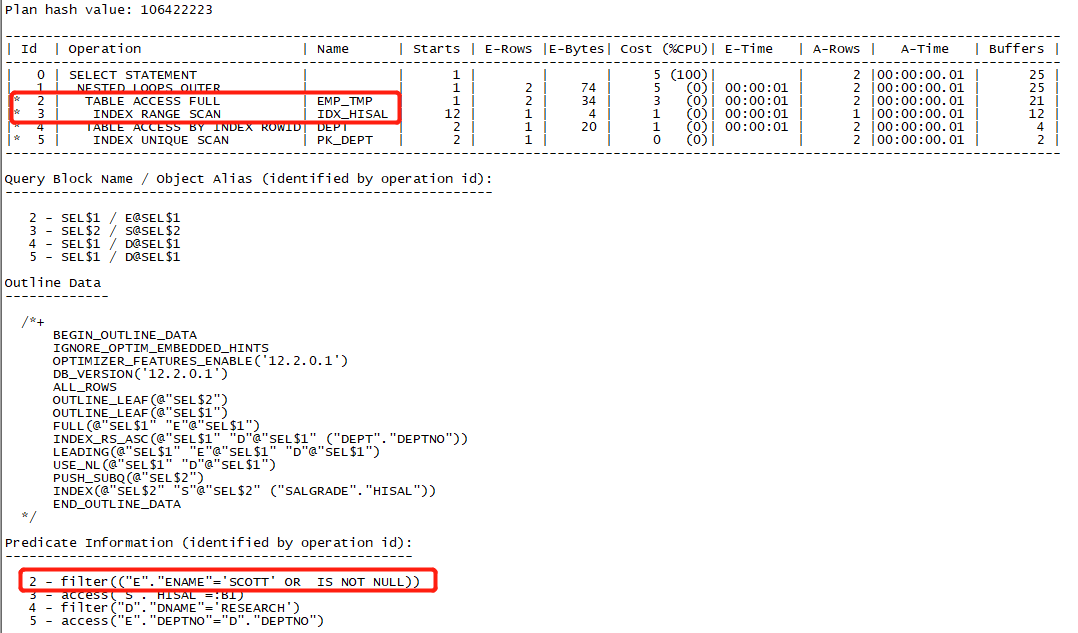
from emp\_tmp e, dept d

where e.deptno = d.deptno(+)

and d.dname(+) = 'RESEARCH'

and (e.ename = 'SCOTT' or exists

(select /\*+push\_subq\*/hisal from salgrade s where s.hisal = e.sal));



我们看到，确实如我们所愿了，但是buffer还涨了，这个原因是因为dept表还不够大，如果dept表再大点儿，使用push\_subq的效率就肯定高了。

但是对于这样的sql，最优的优化方式显然不是这样，因为cbo智商有限，所以这样的sql需要手改，把or改成union（约等于手动or扩展，之所以说是约等于，2.3.2.1说），这样sql就该展开子查询展开子查询，该走ename列上的索引走ename列的索引。具体我们2.3.2.1说。

### 将子查询展开

除了上述的single-row子查询以外:

In

Exists

Not in

Not exists

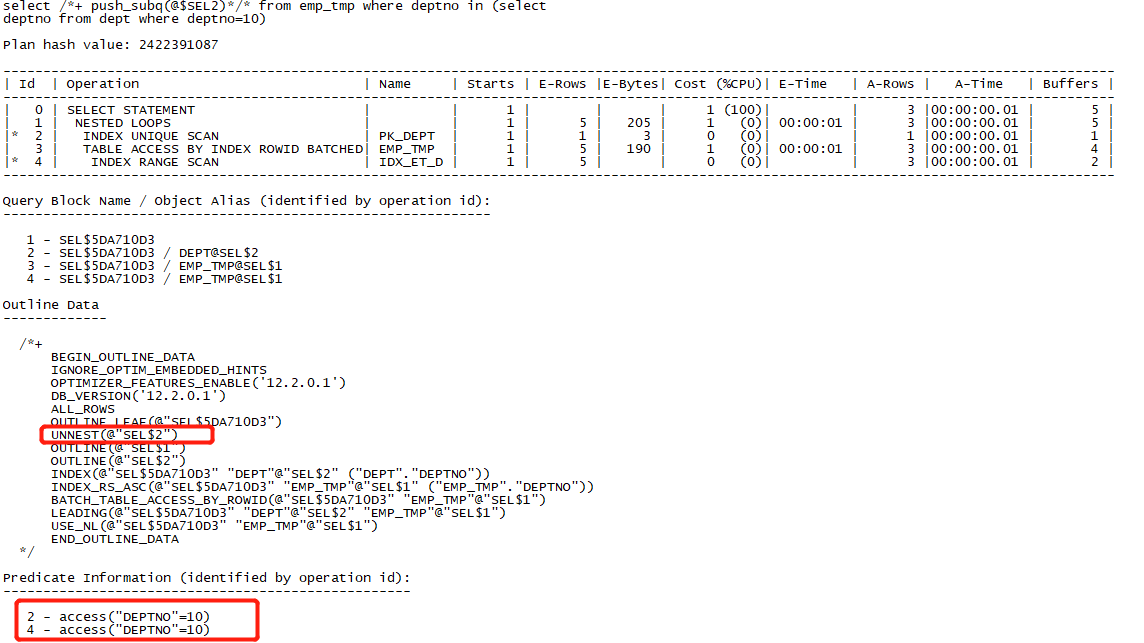
ALL

ANY

都是返回结果可以包含多条记录。

那如果能确定返回的结果只有一条的情况下，会和single row subquery执行计划一样么？

SQL> select /\*+ push\_subq(@$SEL2)\*/\* from emp\_tmp where deptno in (select deptno from dept where deptno=10);



我们看到根本不吃这一套，还是使用了unnest，由此，说明俩个事儿

1. 我又觉得push\_subq也算是一种子查询展开，就像盖总说的，这应该是对unnest的一种补充，不重要了，后面再慢慢研究push\_subq。
2. 就算是明知道in后面的子查询只会返回一行结果集，仍然在处理方式上不同于single-row子查询。

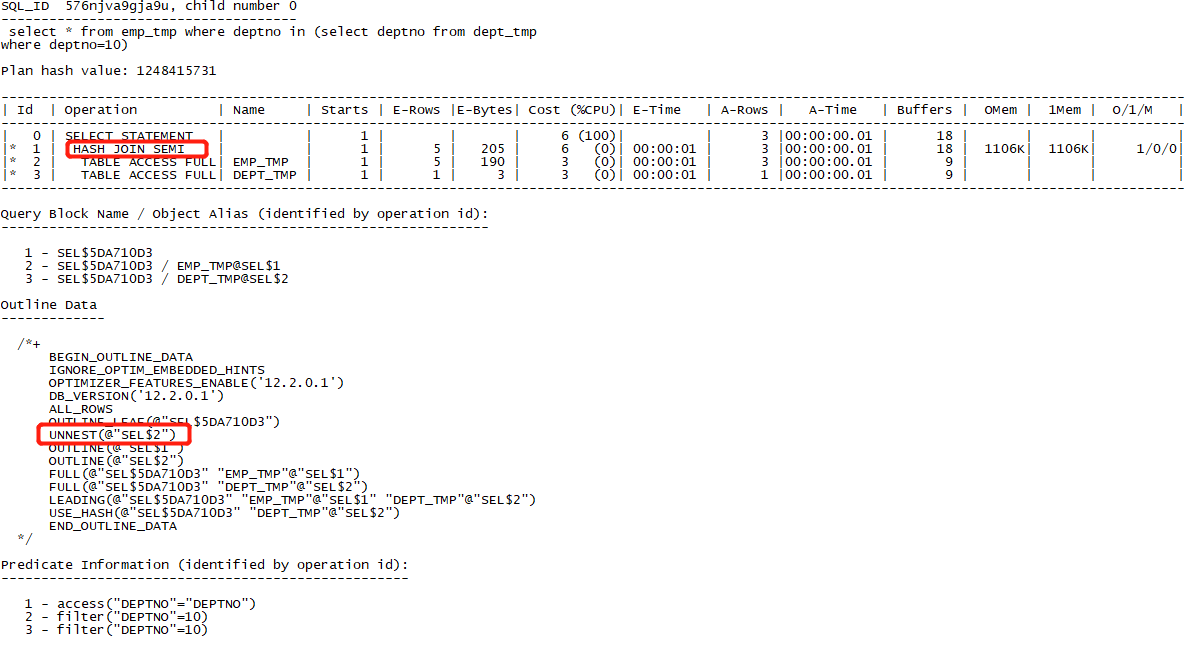
接下来我们来看下，各种子查询展开的情况。

#### 半连接

##### 正常的semi展开

SQL> select \* from emp\_tmp where deptno in (select deptno from dept\_tmp where deptno=10);

这里我事先把emp\_tmp的deptno列上的索引删了。看下执行计划：



看到hash join semi的字样，这里注意，只要执行计划中看到semi就说明子查询一定展开了！看到outline部分，也是这么描述的。

这里子查询相当于展开成为:

select emp\_tmp.\*

from emp\_tmp, dept

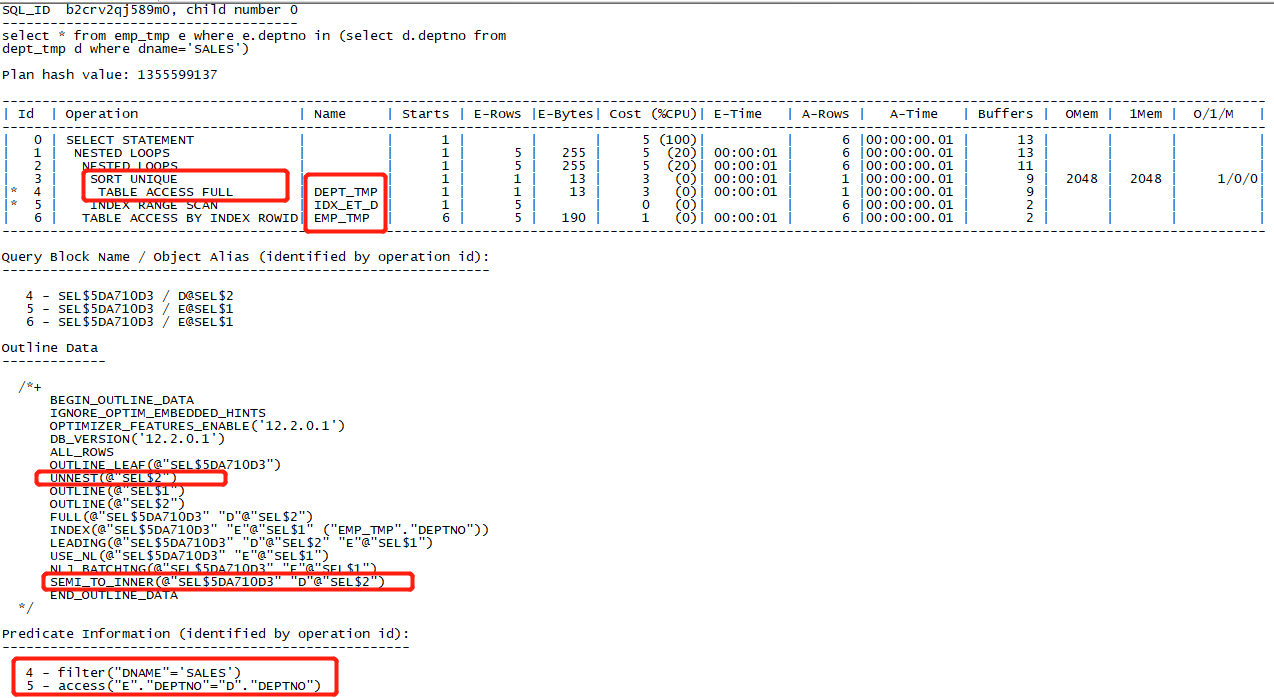
where emp\_tmp.deptno semi= dept\_tmp.deptno

and dept\_tmp.deptno = 10;

##### 主表连接列有索引

下面我们来看一下如果主表上的连接列有索引，是什么情况：

SQL> select \* from emp\_tmp e where e.deptno in (select d.deptno from dept\_tmp d where dname='SALES');



这里我们有俩个发现：

1. 子查询的表变成了驱动表,看上去像是个内连接
2. 这里没有semi字样，但是在outline中发现了semi\_to\_inner

我们一步步来看到底发生了什么。

首先仍旧是子查询正常展开：

select emp\_tmp.\*

from emp\_tmp, dept

where emp\_tmp.deptno semi= dept\_tmp.deptno

and dname='SALES';

然后把这个semi消掉：

select emp\_tmp.\*

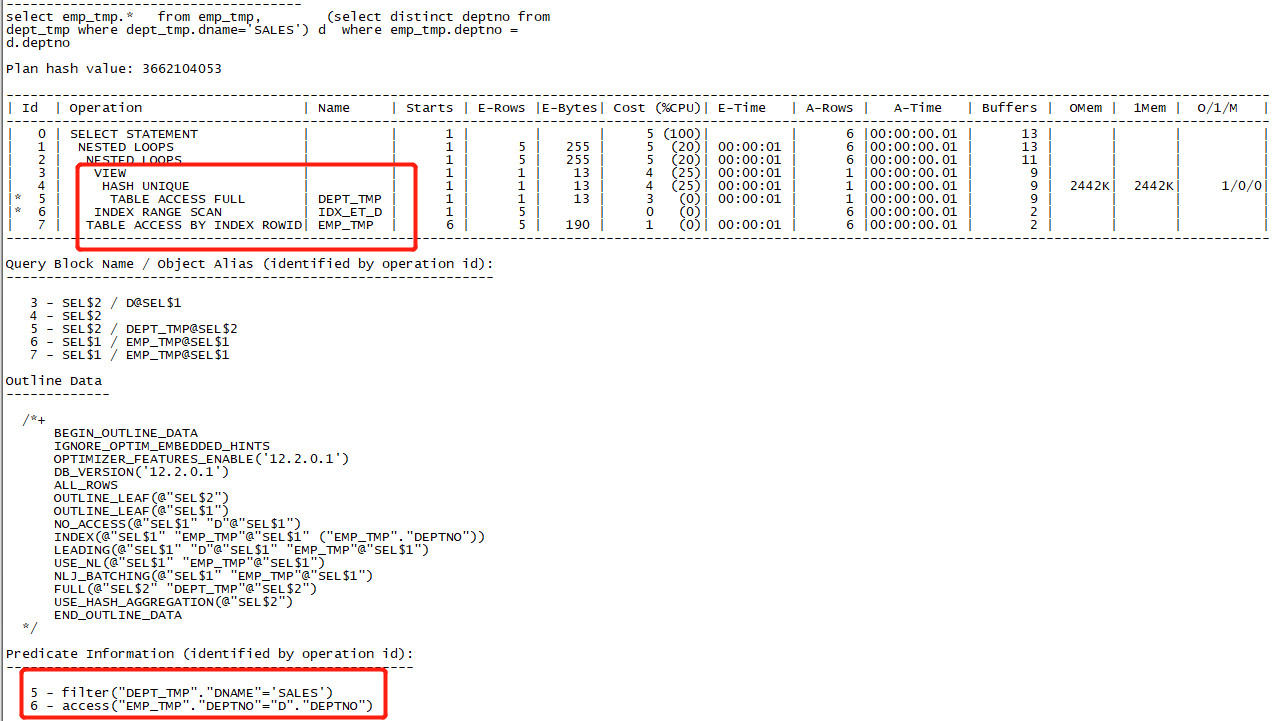
from emp\_tmp,

(select distinct deptno from dept\_tmp where dept\_tmp.dname='SALES') d

where emp\_tmp.deptno = d.deptno;

现在原sql就被转化成了emp\_tmp与表dept\_tmp过滤后去重的结果集进行内连接了。

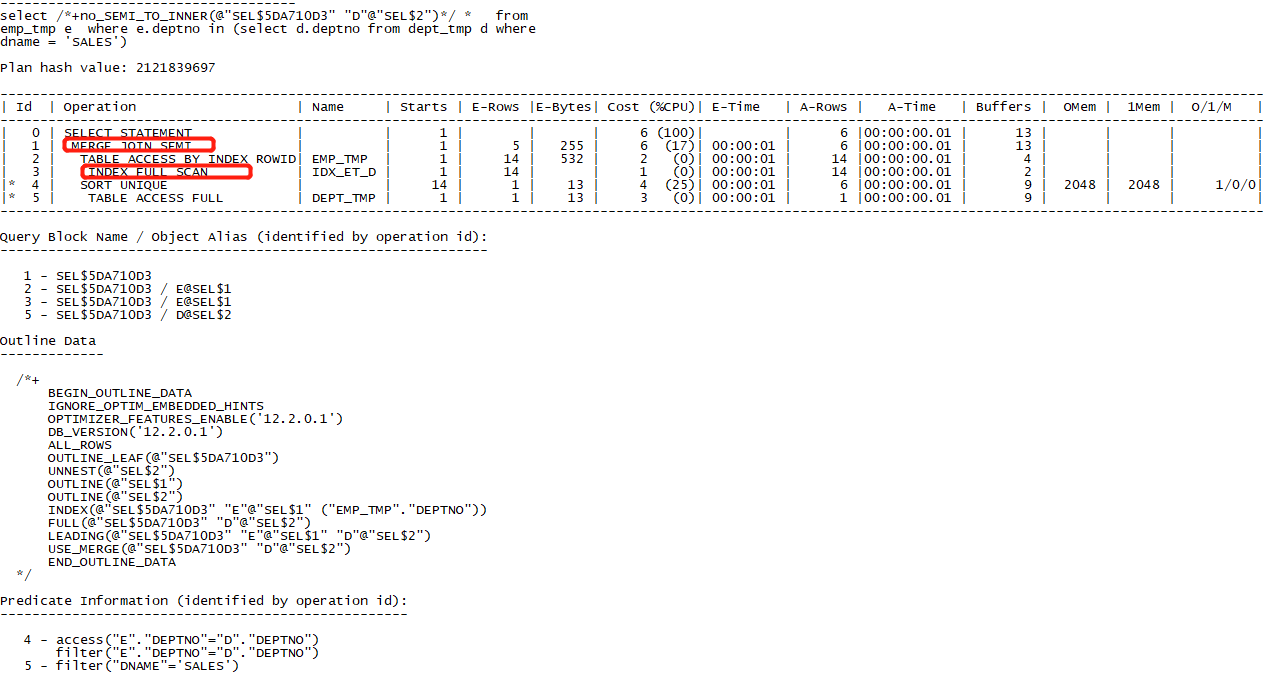
我们来执行一下：



可以看到，执行计划几乎是一样的，注意执行计划中的这个view是没有名字的哦。

从上面的实验，其实也解释了semi\_to\_inner的含义和原理，也就是半连接转内连接的原理。

下面我们看下no\_semi\_to\_inner的效果：

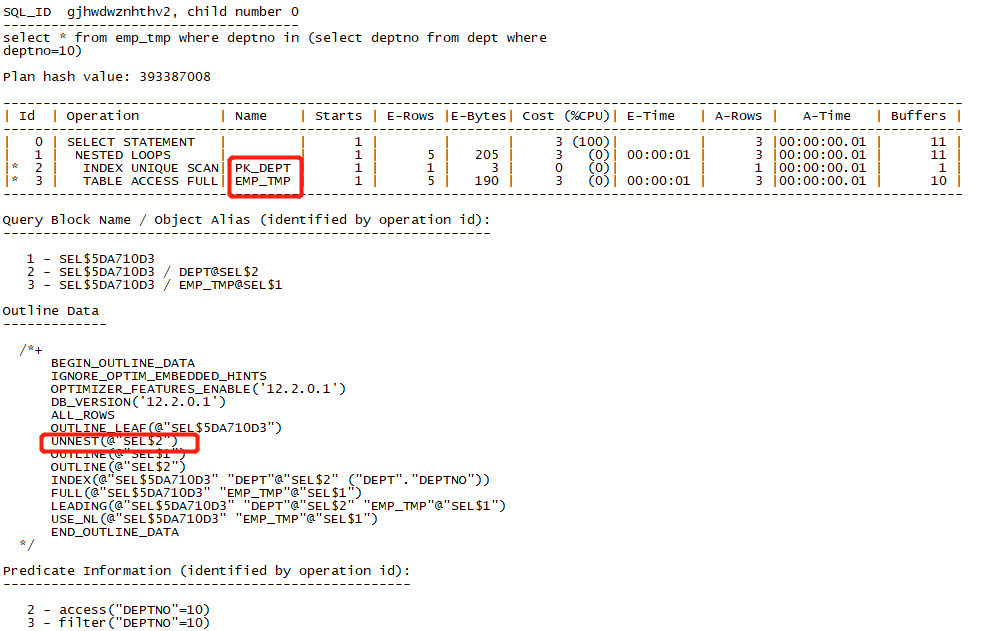


因为还要走semi半连接，所以emp必须为主表，所以就不能真正使用到索引的筛选特性，这里其实是用了该索引的有序性，就不赘述了。

##### 从表的连接列有唯一索引

下面我们再来做一个查询，这次使用dept表，看下从表的连接列有主键是什么效果：

SQL> select \* from emp\_tmp where deptno in (select deptno from dept where deptno=10);



这里我们看到，驱动表变了，不再是主表了，从outline部分，我们看到子查询被展开了，这到底是怎么做到的呢。

首先，把子查询展开：

select emp\_tmp.\* from emp\_tmp,dept where emp\_tmp.deptno semi= dept.deptno and dept.deptno=10;

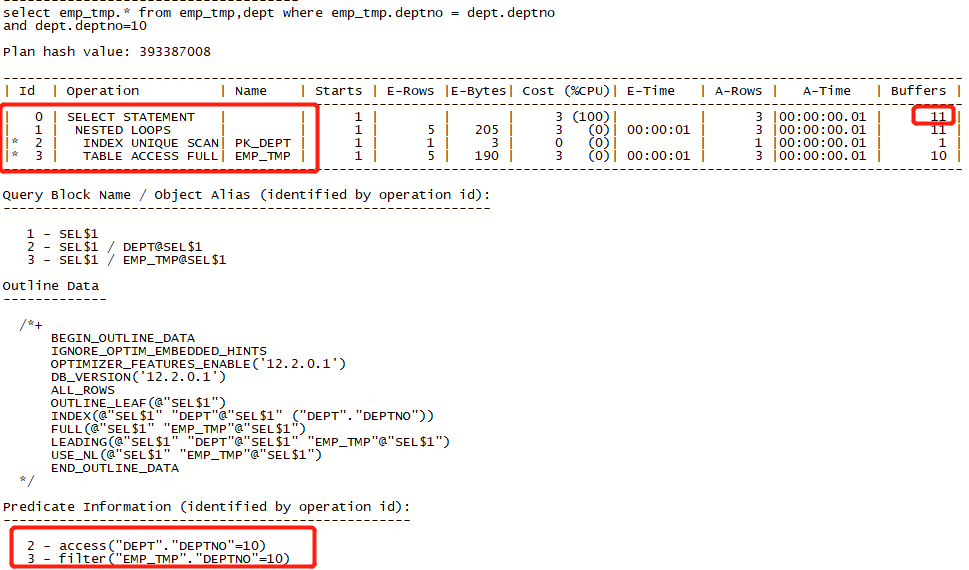
这里semi=其实可以把表达式表示为：

select emp\_tmp.\* from emp\_tmp,(按deptno 去重后的dept） where emp\_tmp.deptno = dept.deptno and dept.deptno=10;

注意，虽然这里表象是把半连接变成了内连接，但这是emp\_tmp与去重后的dept的内连接，也就是说现在还是emp\_tmp与dept的半连接。

但是，deptno是dept的主键，那对于这条指定了deptno的sql，去重这一步就是没有必要的，所以这条sql最终是：

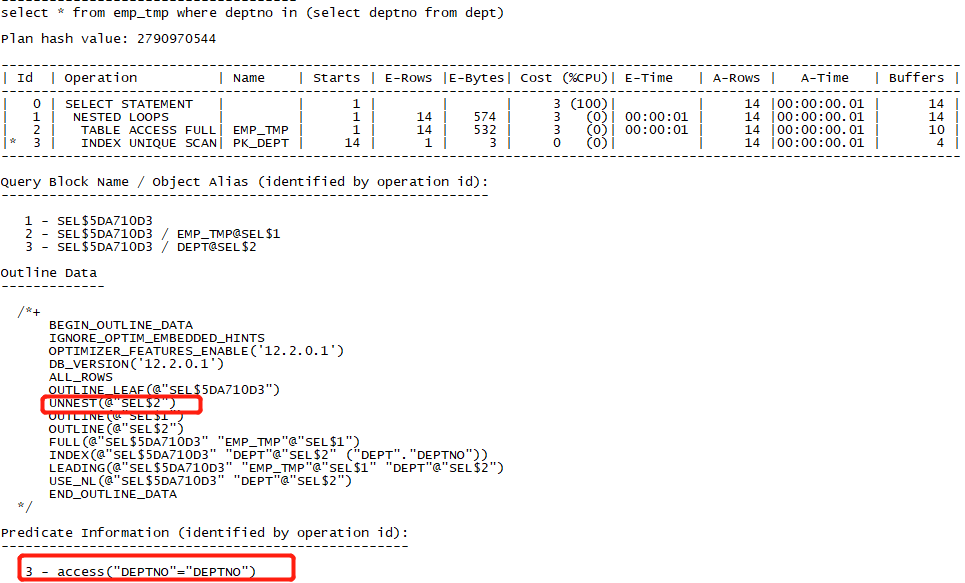
select emp\_tmp.\* from emp\_tmp,dept where emp\_tmp.deptno = dept.deptno and dept.deptno=10;



我们看到执行计划与前面是一模一样，buffers都一样。但是为什么我们在上一个执行计划中没有看到outline中又semit\_to\_inner，即半连接转内连接。

我们再来看一条sql，这次我们把对从表主键的等值过滤条件去掉：

SQL> select \* from emp\_tmp where deptno in (select deptno from dept);



我们可以看到，依然没有semi\_to\_inner，这就说明了如下结论：

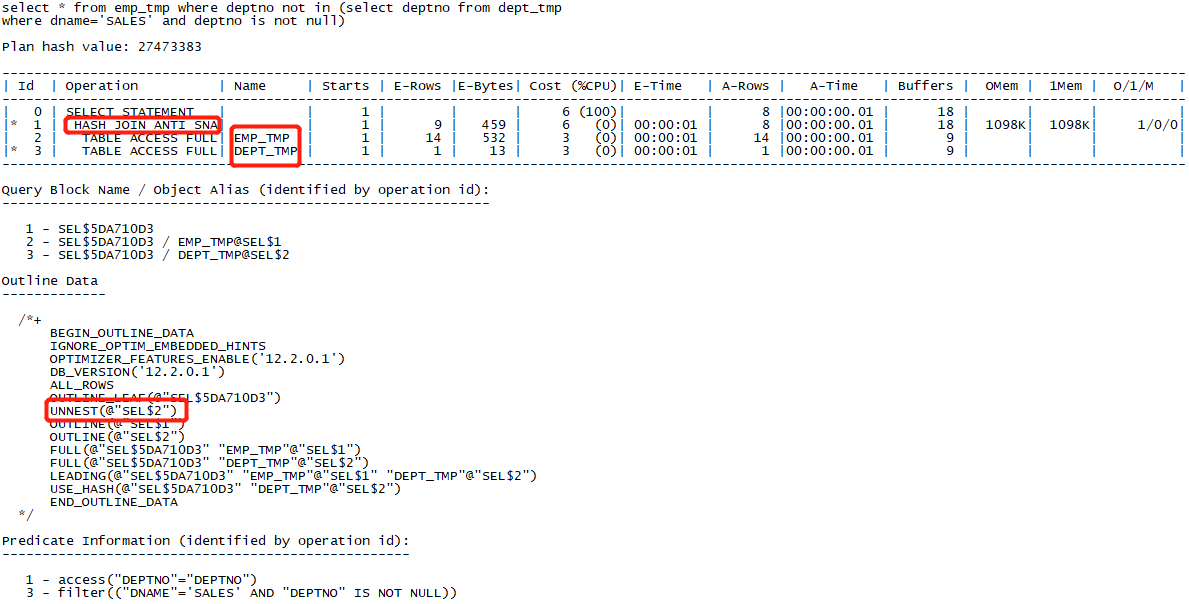
对于从表连接列上有唯一索引，oracle从一开始就没把它当作半连接，而是直接就展开成了我们最终看到的内连接的样子。

#### 反连接

这里我们还是只用not in来做实验:

SQL> select \* from emp\_tmp where deptno not in (select deptno from dept\_tmp where dname='SALES' and deptno is not null);

一定要注意，如果从表上的连接列没有not null的限制，这里一定要加连接列is not null的判断，以防万一。Not exist不需要。这些在之前的报告中有详细介绍。



这条语句实际上就是转化为了：

select \*

from emp\_tmp, dept\_tmp

where emp\_tmp.deptno anti= dept\_tmp.deptno

and dept\_tmp.dname = 'SALES'

and dept\_tmp.deptno is not null;

这里我们看到emp\_tmp是没有走索引的，因为：反连接和<>走不了索引！！！

### 将子查询转换为内嵌视图

select t1.cust\_last\_name, t1.cust\_id

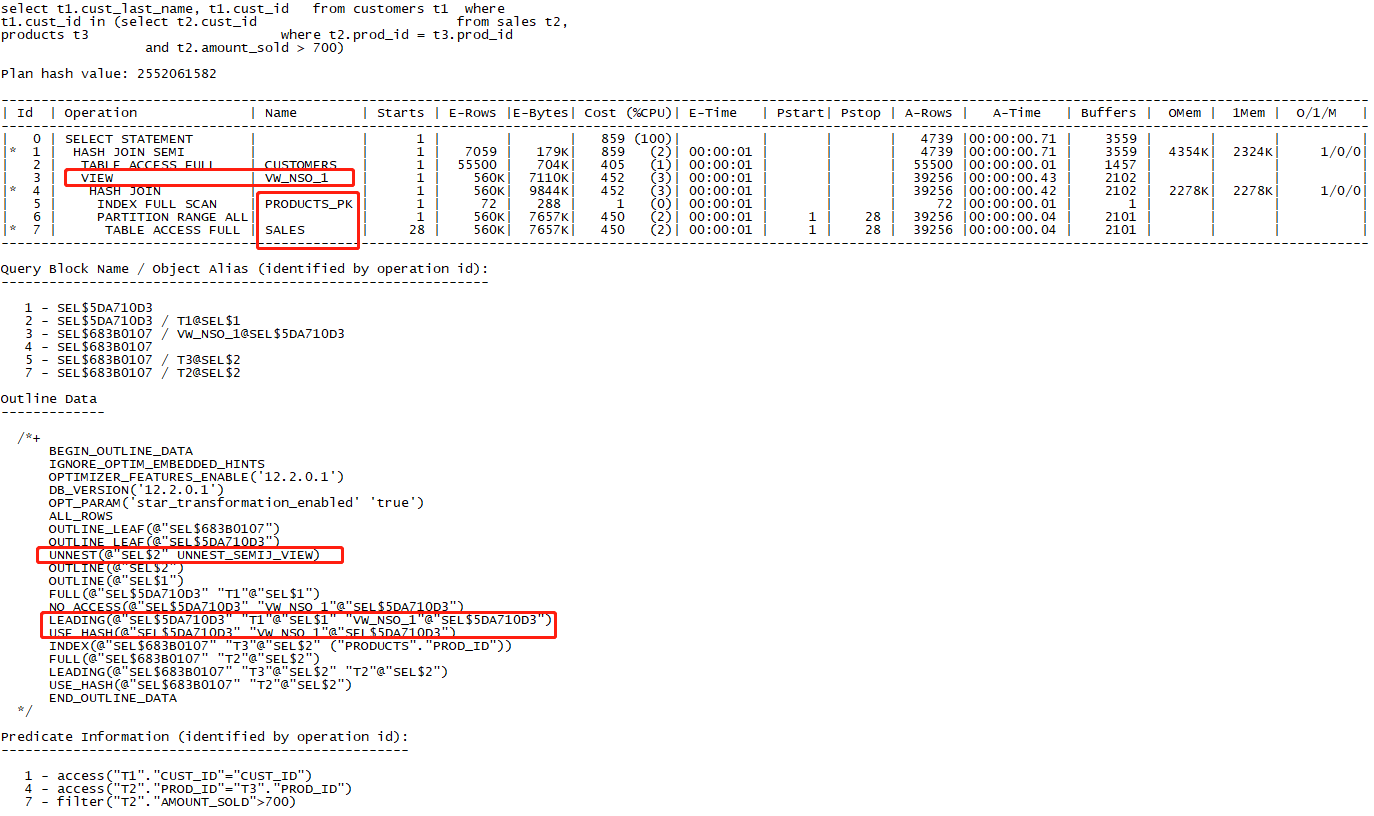
from customers t1

where t1.cust\_id in (select t2.cust\_id

from sales t2, products t3

where t2.prod\_id = t3.prod\_id

and t2.amount\_sold > 700);



通过执行计划可以看出，该sql是先将子查询转化为视图VW\_NSO\_1（nested subquery optimizing），然后customer再与视图VM\_NSO\_1做半连接。即展开如下：

select t1.cust\_last\_name, t1.cust\_id

from customers t1,

(select t2.cust\_id

from sales t2, products t3

where t2.prod\_id = t3.prod\_id

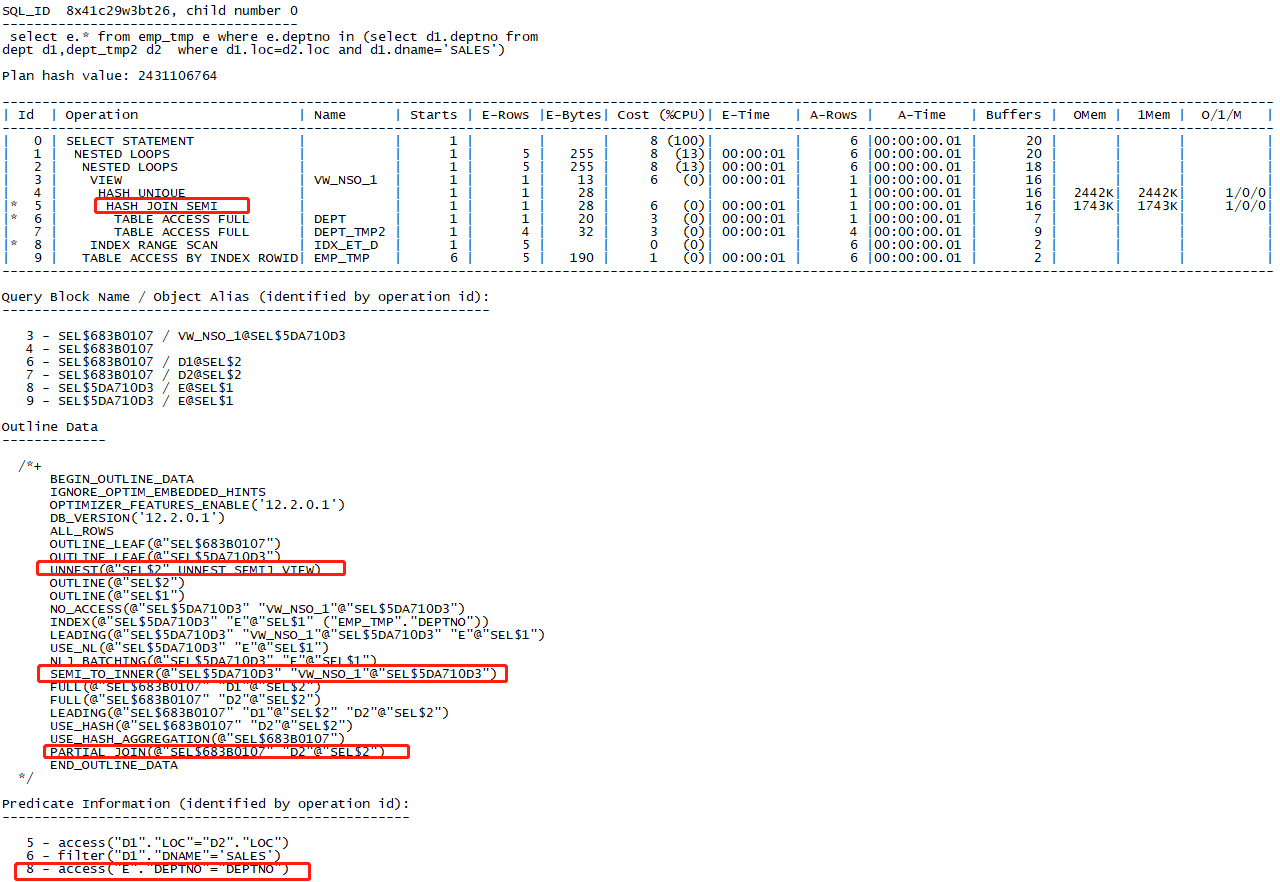
and t2.amount\_sold > 700) VW\_NSO\_1

where t1.cust\_id semi = VW\_NSO\_1;

注意，这里之所以不能从sales开始展开，是因为sales的cust\_id列没有唯一索引，如果展开了，结果就不能保证与原sql一致。结果集的正确性永远是第一位的！

我们来看个例子：

select e.\* from emp\_tmp e where e.deptno in (select d1.deptno from dept d1,dept\_tmp2 d2 where d1.loc=d2.loc and d1.dname='SALES');



这条sql不啰嗦了，搞死我了。看到semi是在子查询里面做的，outline部分有partial\_join。

不过这个unnest的依然是内嵌视图，而且明明说的就是d1.deptno，有主键！但是还是有一步hash unique，看不懂！

使用了各种hint，就是做不到让emp\_tmp和dept先做nl，再和dept\_tmp2做半连接。但是按这个执行计划的意思就是下面的sql：

select e.\*

from emp\_tmp e,

(select distinct d1.deptno

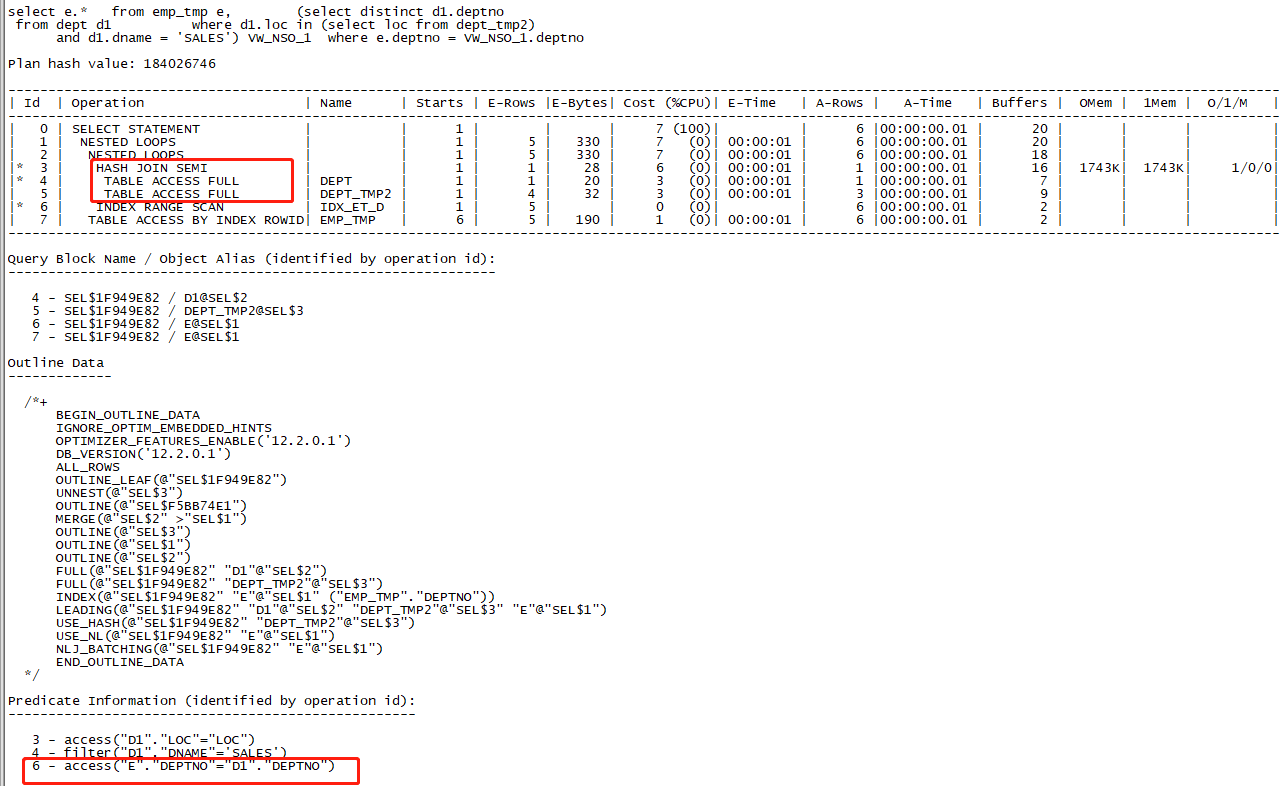
from dept d1

where d1.loc in (select loc from dept\_tmp2)

and d1.dname = 'SALES') VW\_NSO\_1

where e.deptno = VW\_NSO\_1.deptno;

我们看这个执行计划：



这里看到，id=6的谓词条件是e.deptno=d1.deptno了，执行计划里面也没有去重了。我们再来看：

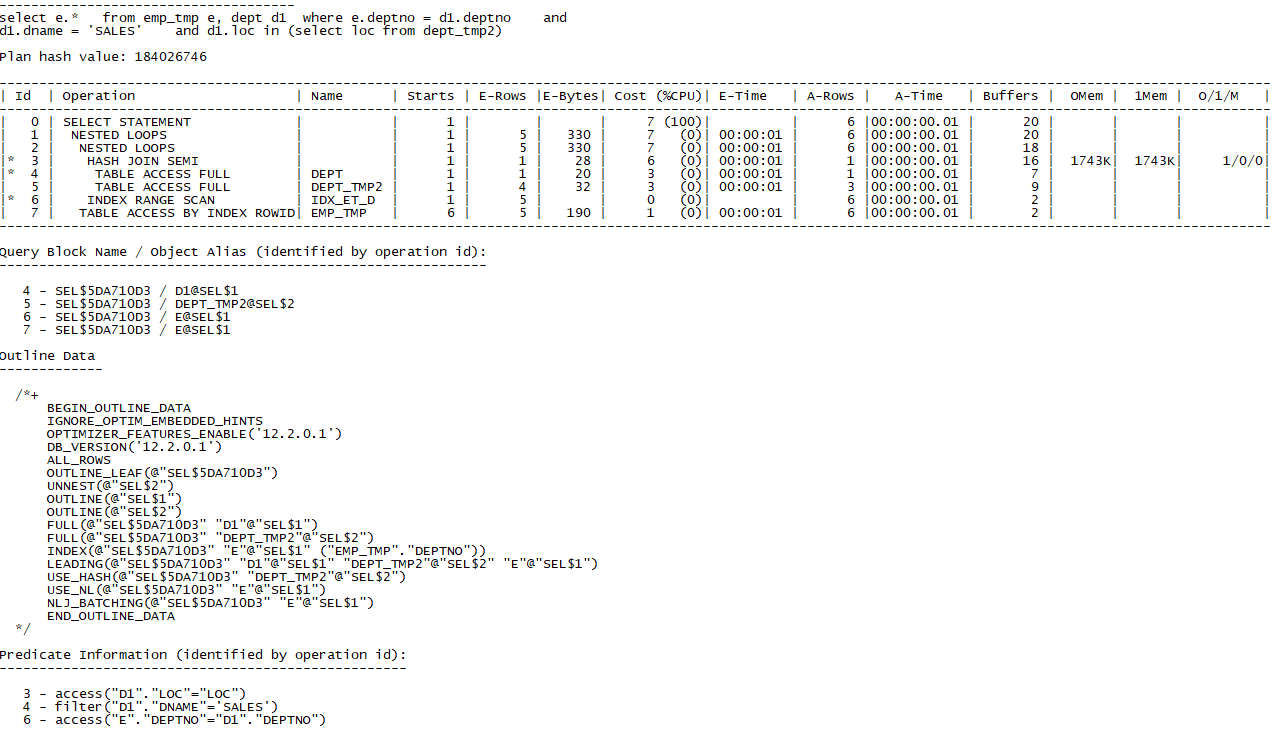
select e.\*

from emp\_tmp e, dept d1

where e.deptno = d1.deptno

and d1.dname = 'SALES'

and d1.loc in (select loc from dept\_tmp2);



看执行计划！是不是一毛一样！！

再看我给这个sql加个hint：

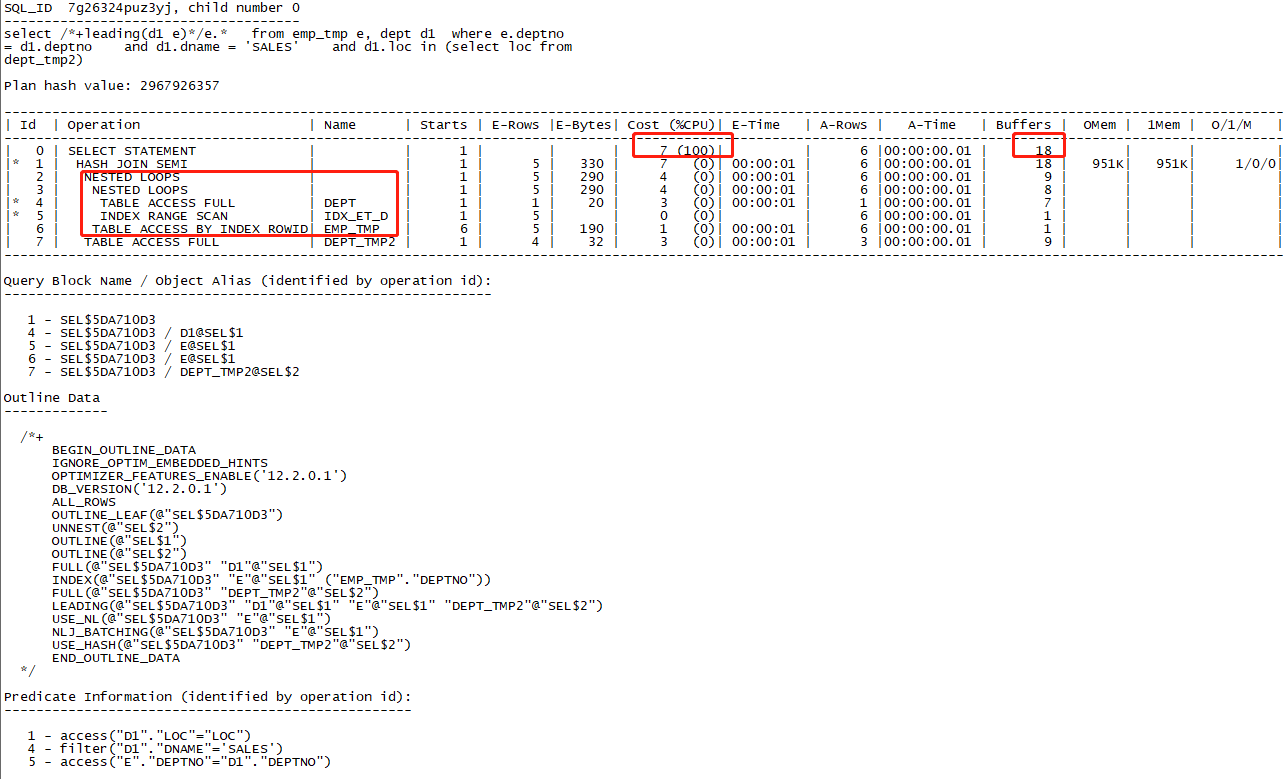
select /\*+leading(d1 e)\*/e.\*

from emp\_tmp e, dept d1

where e.deptno = d1.deptno

and d1.dname = 'SALES'

and d1.loc in (select loc from dept\_tmp2);



看buffers是不是比原来少了2！但是为啥oracle就展不成这样的形势。原因就是那个不知道为什么的hash unique，莫名其妙！

### 对于子查询展开成本的反思

#### 子查询展开不计成本

先说一个事实：10g以后，普通子查询展开不计成本。

这里的普通的意思就是不算把子查询转换为内嵌视图的展开。

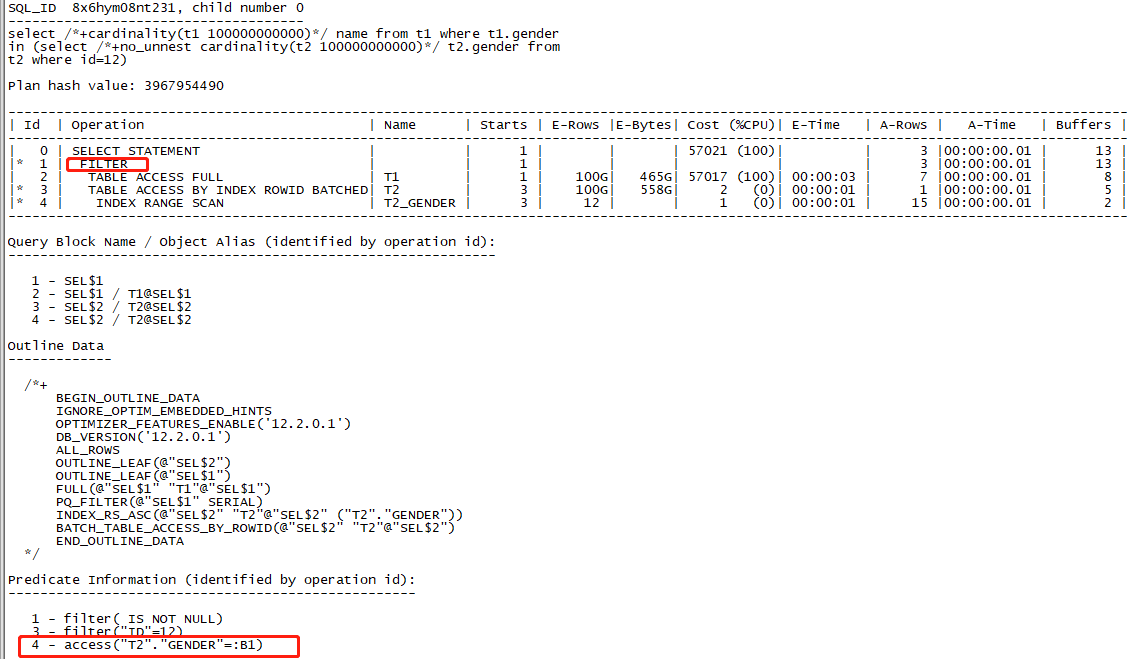
然后我们看个例子：

SQL> select /\*+cardinality(t1 100000000000)\*/ name from t1 where t1.gender in (select /\*+cardinality(t2 100000000000)\*/ t2.gender from t2 where id=12);



很明显，子查询展开了。成本值是413M。然后我们看不展开：

SQL> select /\*+cardinality(t1 100000000000)\*/ name from t1 where t1.gender in (select /\*+no\_unnest cardinality(t2 100000000000)\*/ t2.gender from t2 where id=12);



这个没展开，成本值是57021。所以确实普通子查询展开是不计成本的。

#### oracle为什么这样做

oracle之所以这样设定是因为它认为展开总是比不展开要效率高。

具体的原因就是：

如果展开，就可以改变驱动表，即改变连接顺序，可以有更多的选择，而不展开，就只能走filter，filter的缺点显而易见，一个就是驱动表是固定的，第二个就是驱动表的连接列的基数有多大，那被驱动表就要被循环多少次。

#### 展开or不展开

通常来说，展开确实要比不展开效率高，主要原因就是展开之后，可以选择较小的结果集做驱动表走nl或者hash join，这个就不多说了。

##### 从主表的角度

正是因为不展开走的是filter，所以我们需要来总这个家伙入手，说一种特殊的情况。

filter有一个优点就是，如果驱动表有一亿行，hash join需要做个很大的hash table（PGA甚至有可能放不下），而nl是铁定要把被驱动表循环一亿次。但是filter就不同，假如驱动表的基数只有1，毫无疑问，filter的效率完全碾压hash和nl。也就是说，如果驱动结果集的连接列基数非常低，那filter就有用武之地了。

这里我需要特别强调是驱动结果集的连接列！一亿行的表的某一列唯一值只有1个不多见，但是驱动结果集唯一值只有一个却很容易，比如主表的过滤条件中有唯一索引列的等值查询，再比如主表过滤后有1000行数据，但是这1000行数据的过滤列的值都是一个值。

##### 从子查询的角度

单纯filter这个优点还不足以决定展不展开。

我们注意in exists =any的子查询有一个特质：对子查询连接去重！

如果我们展开子查询，那么去重这一步一般会被推后去做，但是如果子查询本身去重后结果集就是那么几个值，那我们把子查询展开导致去重推后的行为就很蠢了，尤其是子查询的结果集非常大（说明子查询中的表也不会小，展开去做hash或者nl的代价也不会太小），但是连接列的唯一值缺极小。

##### 反思

综合上述俩点，我们在遇到效率不高的子查询展开的时候，就要去真实的去查看子查询的结果集的唯一值有多少，主表过滤后的唯一值又有多少。然后再决定要不要让子查询展开。

之所以有这一小节的总结，是因为在项目中真真实实的遇到过这样一个案例，这个案例在我的博客中有记载: <http://blog.itpub.net/31532509/viewspace-2154615/>

### 子查询展开总结

**子查询展开的目的：**

让cdb有更多的选择。

**对于single-row子查询只需要记住：**

PUSH\_SUBQ

**对于普通子查询展开需要记住：**

1.10g以后展开是不计代价的，能展就展，这也成了优化需要注意的地方

2.反连接连接列不走索引

**对于子查询转化为内嵌视图只需要记住：**

VW\_NSO\_1: VIEW\_NESTED SUBQUERY OPTIMIZING\_1

## 视图合并（View Merging）

### 简单视图合并

这里所说的简单视图是指视图定义中不包含外连接、聚合函数、group by、order by、distinct、rownum、connect by、集合运算符的视图。

我们先来看一个简单视图的例子：

视图view\_1创建如下：

create or replace view view\_1 as

select d.department\_id

from departments d, locations l

where d.location\_id = l.location\_id

and l.city = 'London';

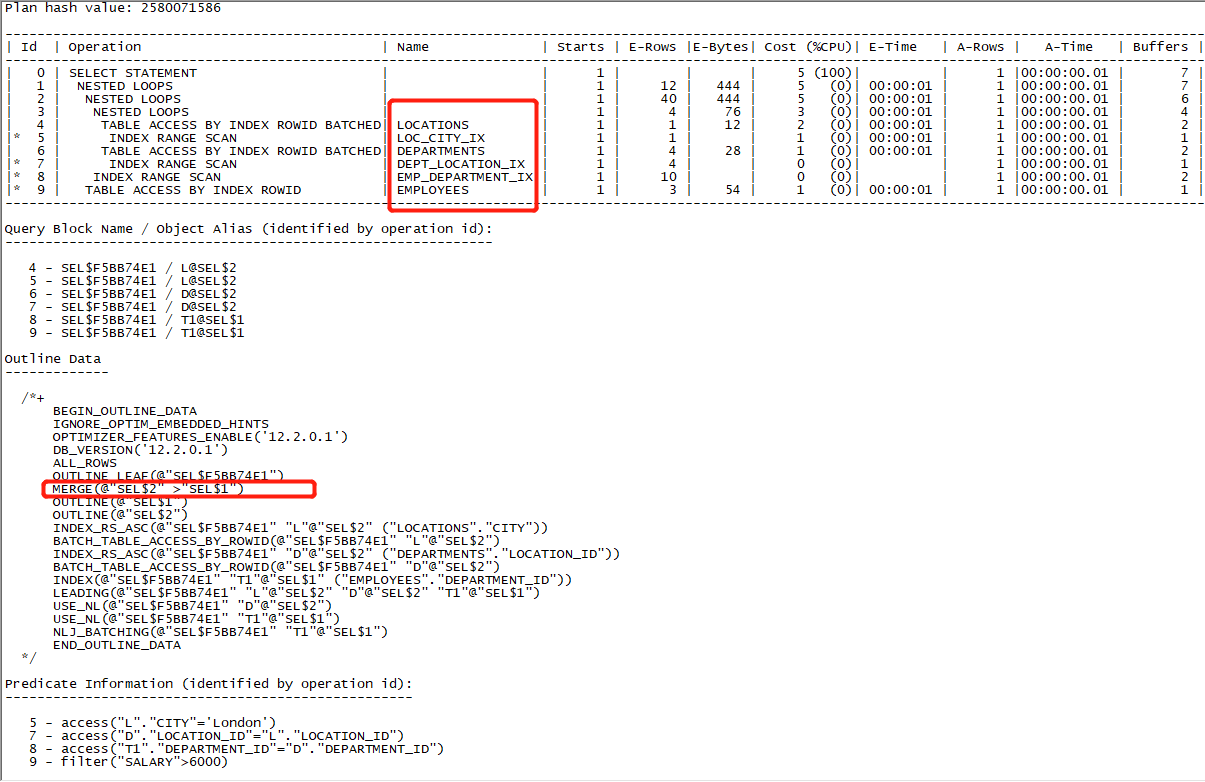
然后进行如下查询：

select employee\_id, first\_name

from employees t1, view\_1

where t1.department\_id = view\_1.department\_id

and salary > 6000;



从outline部分可以看到发生了视图合并，效果很像子查询展开，我们并没有看视图的名字，而是看到的只有基表的关联。

这里我们还需要注意outline是怎么写的：MERGE(@"$SEL2" > "$SEL1")，这里箭头的指向很明确到告诉我们，是视图merge到了外部查询中，所以说视图合并本质上是将视图进行了展开。

如果我们把view\_1的定义带入到sql中，那展开后的效果就是：

select employee\_id, first\_name

from employees t1,departments d, locations l

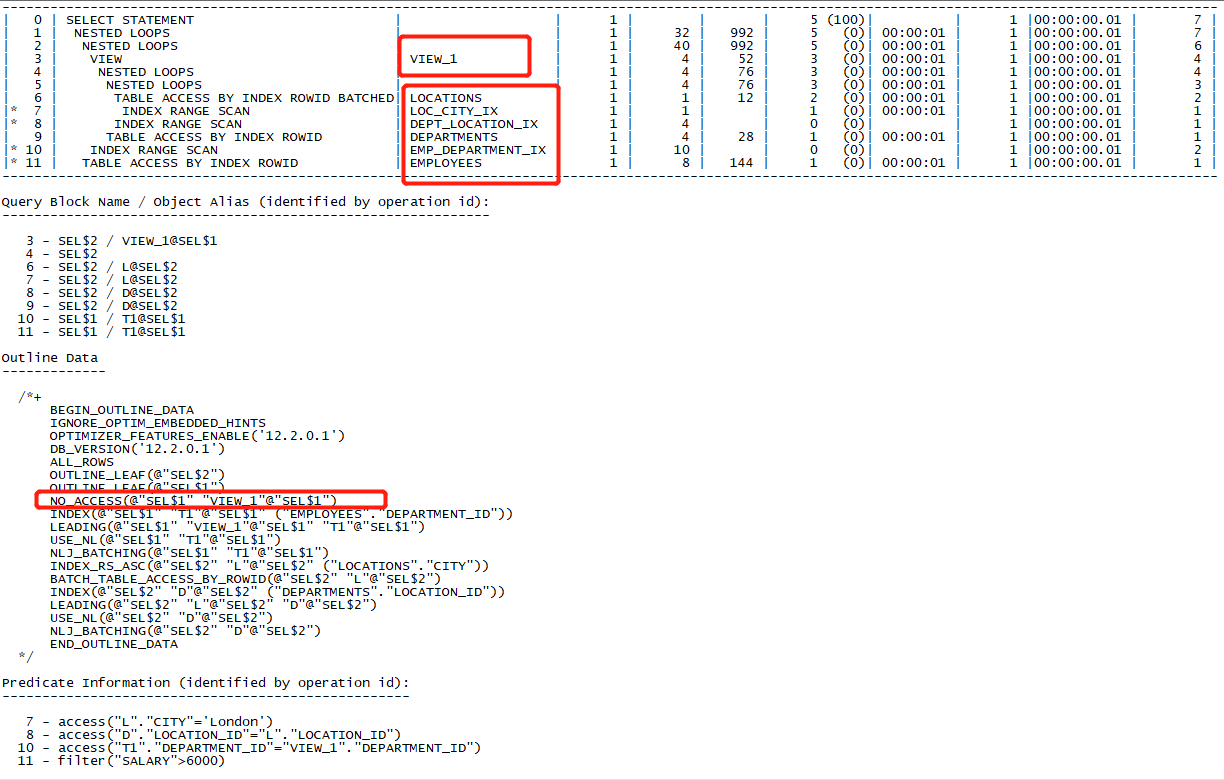
where t1.department\_id = d.department\_id

and salary > 6000

and d.location\_id = l.location\_id

and l.city = 'London';

下面我来看下，如果不进行视图合并是什么效果：



从outline部分可以看到no\_access,而且很明确的说明了是SEL$1的view\_1不让访问，说白了就是不让merge。这里注意，outline中no\_access和merge是对立的，在复杂查询中经常可以遇到，这也有助于帮助我们细分执行计划。

我们发现，这个执行计划跟上个执行计划唯一的不同就是步骤中多了一个VIEW，但我们要注意到VIEW步骤的NAME列正是我们视图的名字。

所以说，如果没有发生视图合并，执行步骤中会出现VIEW字样，且NAME列为视图名。注意不能通过有没有VIEW字样来判断是否发生了视图合并，一定要看对应的NAME列写的是什么。

但是上面俩个查询的执行计划步骤其实是完全一样的，都是locations同departments做完NL然后再同employees做NL。

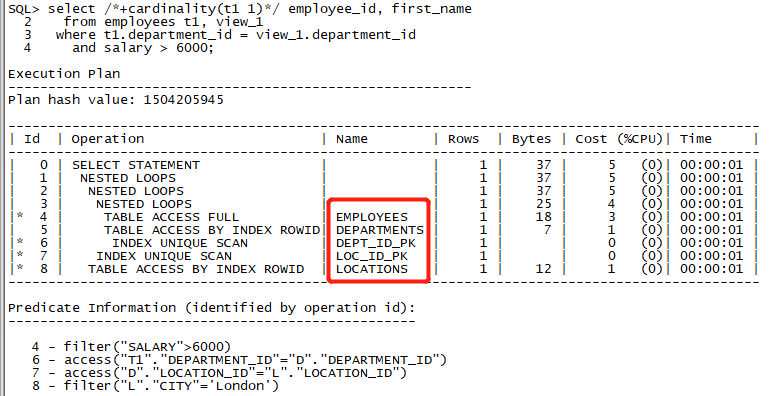
我们下面来欺骗cbo，让它人为employees只有一行数据，且删掉locations的city列上的索引，再来看下效果：

select /\*+cardinality(t1 1)\*/ employee\_id, first\_name

from employees t1, view\_1

where t1.department\_id = view\_1.department\_id

and salary > 6000;



此时我们看到，表连接的顺序已经发生了改变。Employees与deployments先做NL，再与location做NL。

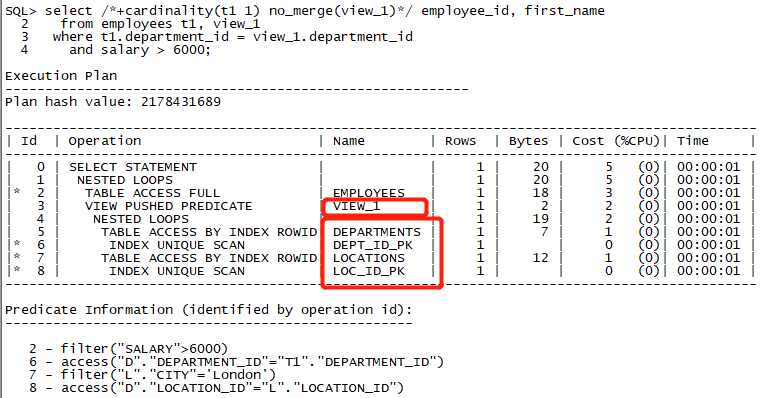
我们来看下不进行视图合并会怎样呢：

select /\*+cardinality(t1 1) no\_merge(view\_1)\*/ employee\_id, first\_name

from employees t1, view\_1

where t1.department\_id = view\_1.department\_id

and salary > 6000;



我们可以看到，虽然同之前的上一次no\_merge的连接顺序不同，但是有一个共同点，就是仍然是departments和locations先做表连接，然后再同employees连接。

综上：我们可以看出来，简单视图的合并只是为CBO提供了更多的执行路径，而这些执行路径中包含着不进行视图合并的执行路径。

所以：对于简单视图合并，oracle是不会考虑成本的。因为合并之后仍可能走到不合并时的执行计划。这一点是同子查询展开不相同的地方（子查询展不展开完全是俩种执行计划）。

### 外连接视图合并

外部查询表与视图是外连接关系，或视图中包含有外连接，视图定义与简单视图相比，除了允许存在外连接，其他都一样不能存在。

这里只说外部查询表与视图是外连接关系的情况，说白了就是与简单试图外连接。

可以进行外连接视图合并的情况有俩种：

1. 视图是驱动表
2. 视图是被驱动表，但视图的定义中只有一个表

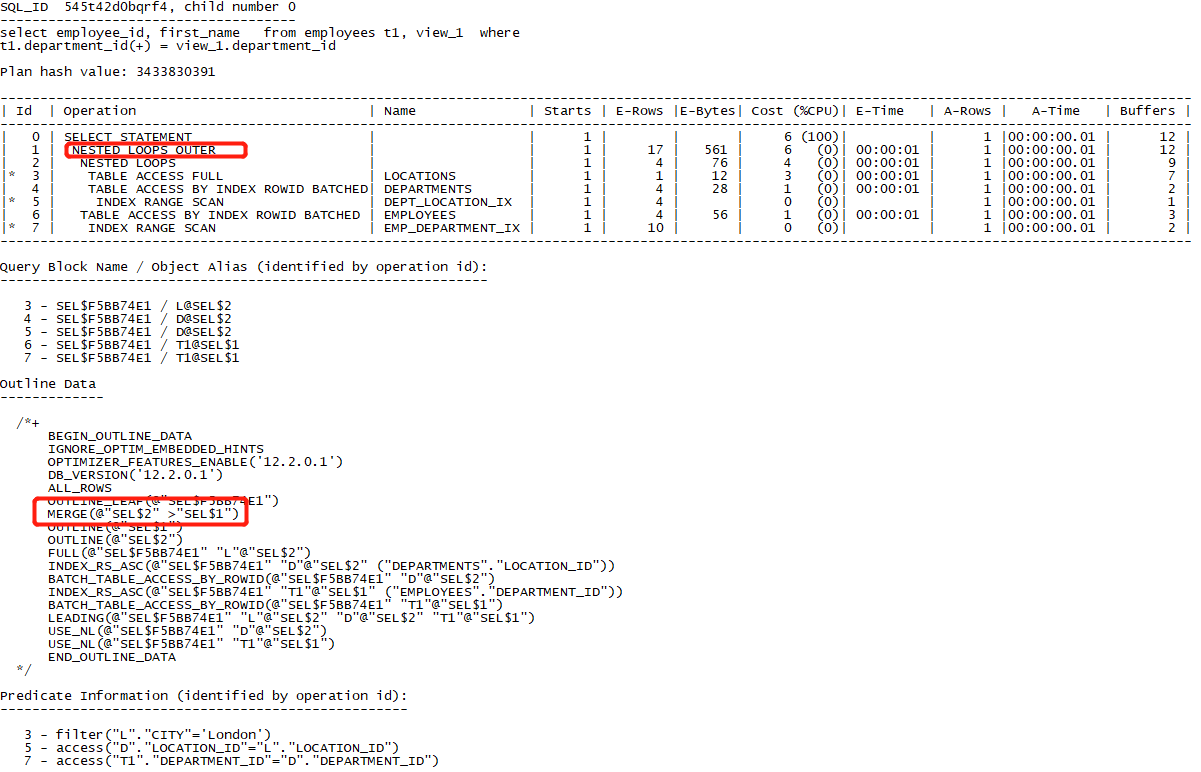
#### 视图是驱动表

我们首先来看下面这条sql：

select employee\_id, first\_name

from employees t1, view\_1

where t1.department\_id(+) = view\_1.department\_id



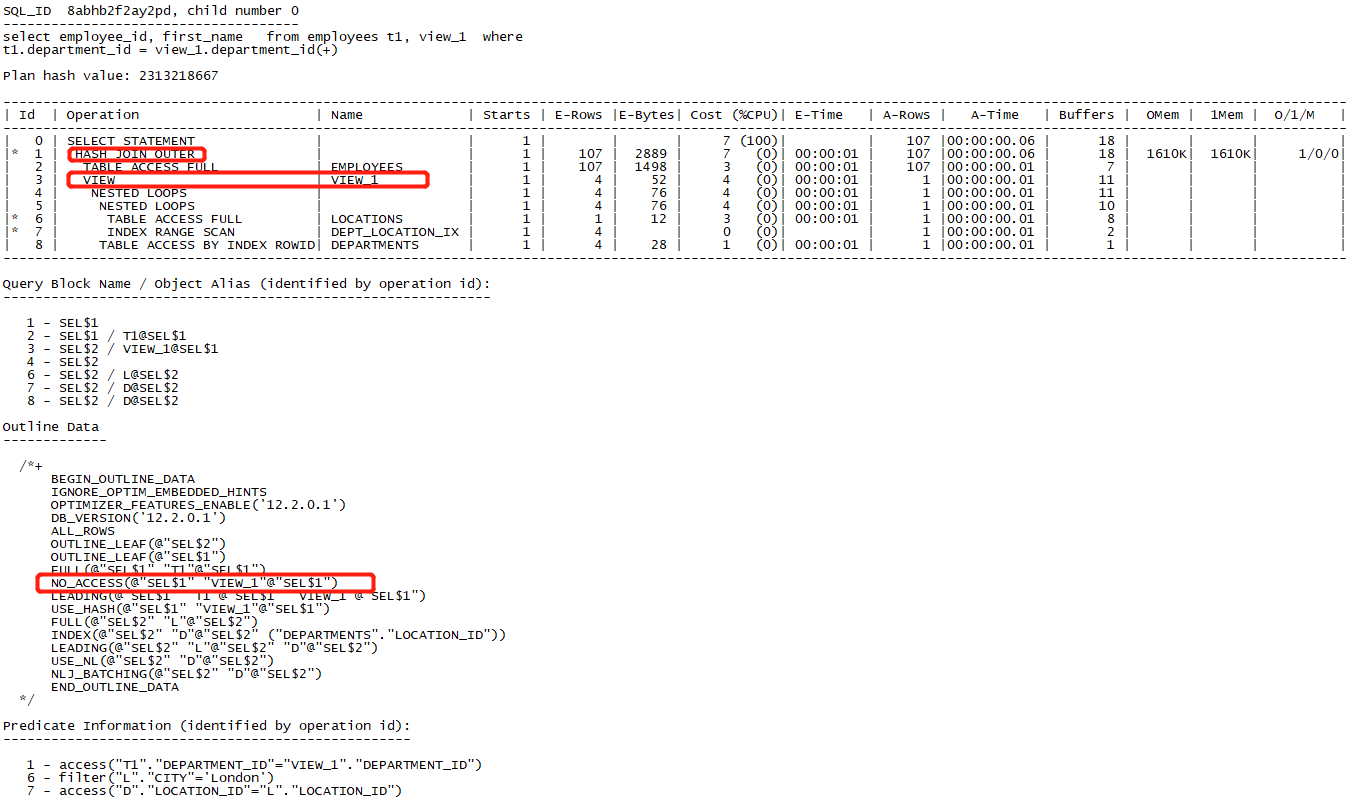
我们看到，当view\_1作为外连接驱动表时，确实发生了视图合并。

如果反过来，让employees表作为外连接驱动表：

select employee\_id, first\_name

from employees t1, view\_1

where t1.department\_id = view\_1.department\_id(+)



可以看到确实如之前所说，因为view\_1的视图定义中有俩张表，所以这里view\_1作为外连接被驱动表并不能被合并。

其实原因很简单，前面的简单视图合并，是因为里外都是内连接，可以交换连接顺序。但是现在是外表做为主表，视图中又有俩张表做连接，employees如果先与departments做外连接，那最终结果就有可能发生改变。

但是如果我们给被驱动表view\_1加一个过滤条件，使外连接可以转化为内连接，那么就又可以进行视图合并了。我们看：

select employee\_id, first\_name

from employees t1, view\_1

where t1.department\_id = view\_1.department\_id(+)

and view\_1.department\_id=40;



确实如我们所说，我们看到发生了视图合并。所以这小节所说的外连接，是不可以转化为内连接的外连接。

#### 视图定义中只有一张表

这个很好理解，关键是视图合并以后sql被等价转换成了什么样子，我们探究一下。

下面是俩张实验表：

SQL> select \* from t1;

COL1 COL2

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

1 A

2 B

3 C

SQL> select \* from t2;

COL2 COL3

----- -----

A A2

B B2

D D2

E2

创建一个视图view\_2：

create or replace view view\_2 as select t2.col2 from t2 where t2.col3='A2';

下面我们来做如下查询：

SQL> select \* from t1,view\_2 where t1.col2=view\_2.col2(+);

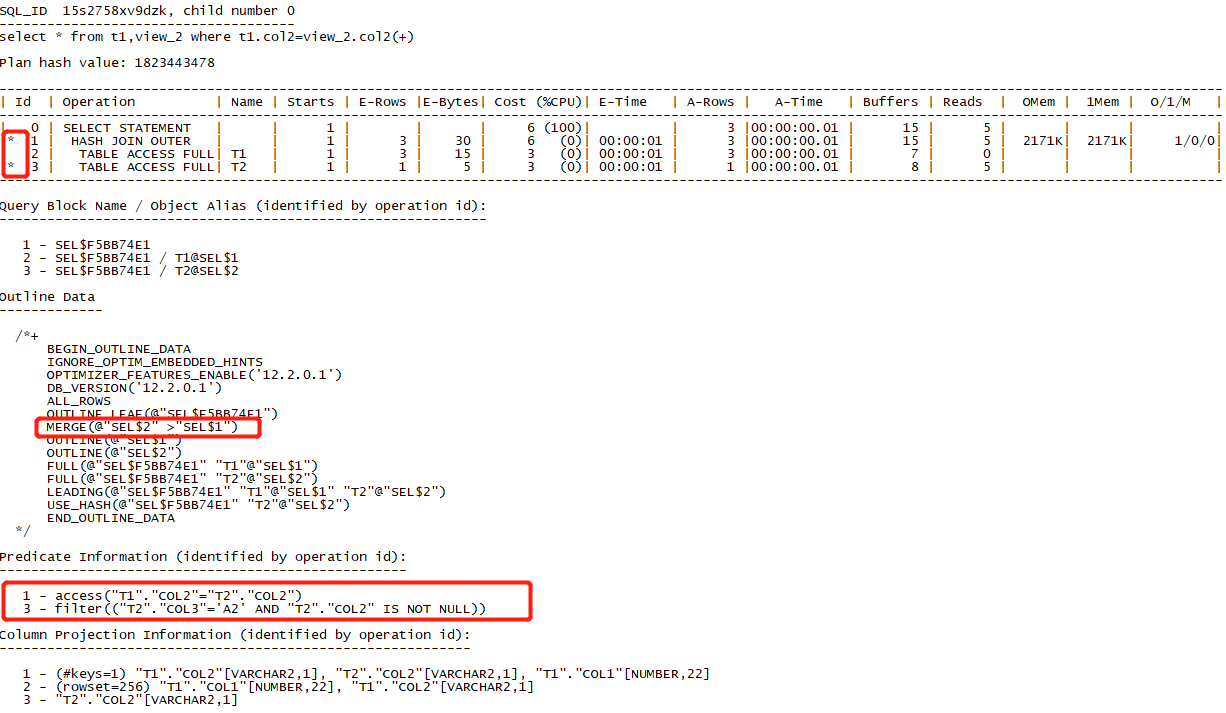
COL1 COL2 COL2

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

1 A A

2 B

3 C



从这个执行计划可以看出来：

1. 确实发生了视图合并
2. 注意id=3的谓词信息，是对t2表先进行了过滤，然后t1才和这个结果集开始外连接

所以，我们对于sql的改写应当如下：

首先把view定义带入到sql中：

select \*

from t1, (select t2.col2 from t2 where t2.col3 = 'A2') view\_2

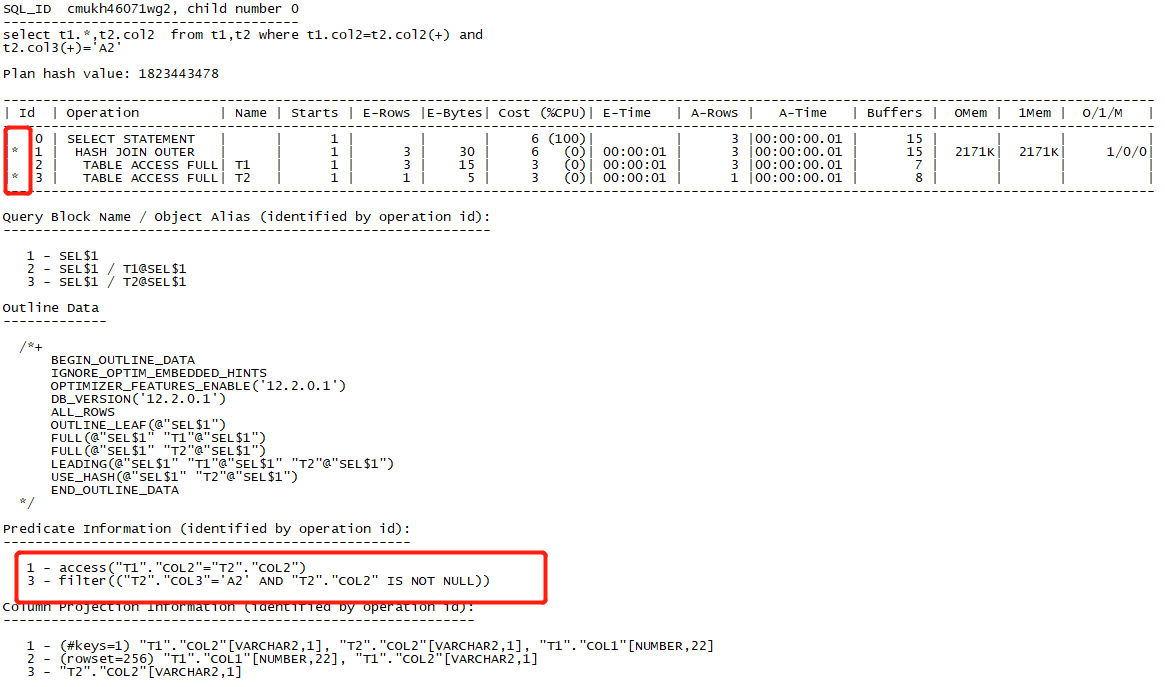
where t1.col2 = view\_2.col2(+);

然后进行视图合并：

select t1.\*,t2.col2 from t1,t2 where t1.col2=t2.col2(+) and t2.col3(+)='A2';

对于t2.col3的这个限制很关键，外连接的限制必须落实到视图中的每一个列上。

我们看下这条sql的执行计划：



可以看到与之前的执行计划完全一样。

执行结果也是一样的：

SQL> select t1.\*,t2.col2 from t1,t2 where t1.col2=t2.col2(+) and t2.col3(+)='A2';

COL1 COL2 COL2

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

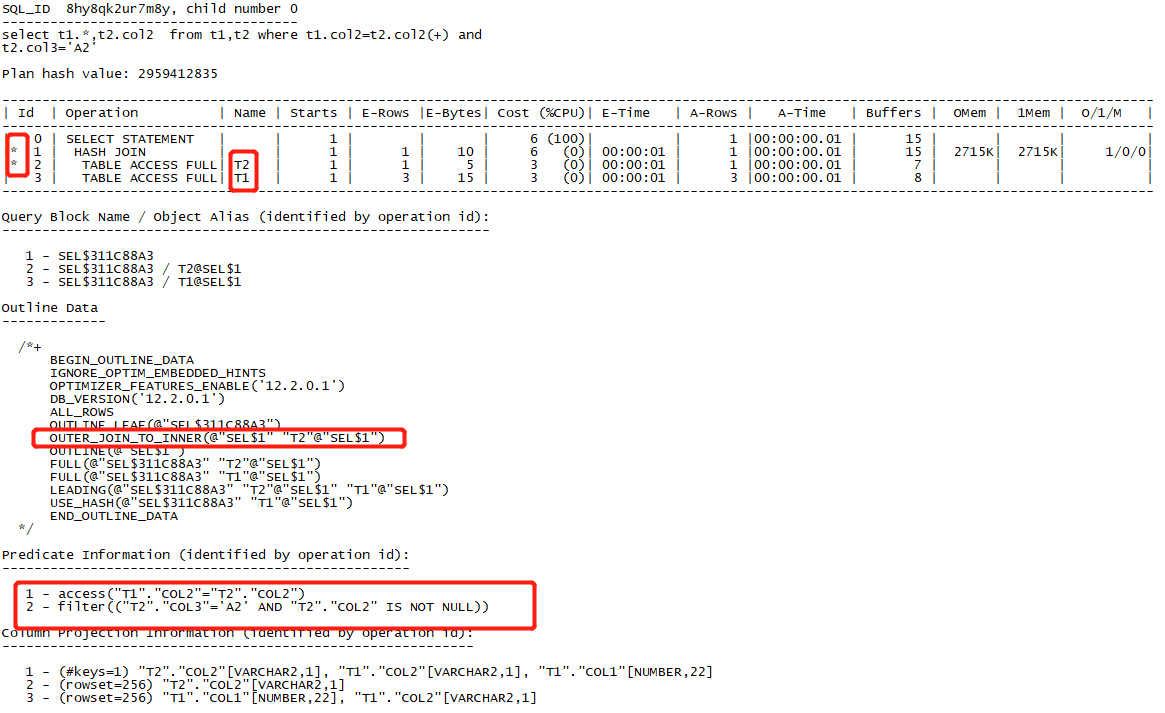
1 A A

2 B

3 C

那如果是换成如下sql：

select t1.\*,t2.col2 from t1,t2 where t1.col2=t2.col2(+) and t2.col3='A2';



我们发现，过滤条件也是对T2直接进行的过滤，好像没什么不一样。

但是现在执行计划的步骤是:

对T2过滤并全表扫描🡪与T1全表扫描后HASH内连接

之前的执行计划的步骤是：

对T1全表扫描后🡪与T2过滤后的结果集做HASH外连接

所以，这里能实现先对T2表进行过滤的原因就是发生了outer\_join\_to\_inner。

我们用hint把它禁了看看:

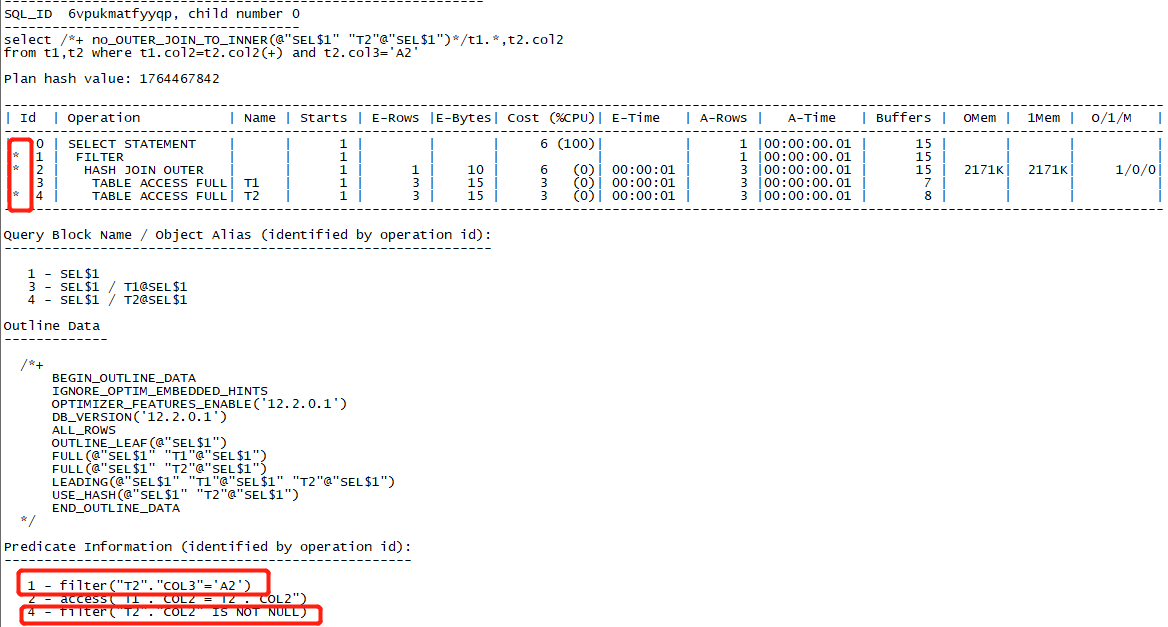
select /\*+ NO\_OUTER\_JOIN\_TO\_INNER(@"SEL$1" "T2"@"SEL$1")\*/

t1.\*, t2.col2

from t1, t2

where t1.col2 = t2.col2(+)

and t2.col3 = 'A2';



这才是这条sql的真面目，id=4的过滤条件只是过滤了连接列的null值，真正对结果集的过滤是在t1与t2做完hash外连接之后才进行的过滤。

其实这一点对比执行结果也是能对比出来的：

SQL> select t1.\*,t2.col2 from t1,t2 where t1.col2=t2.col2(+) and t2.col3='A2';

COL1 COL2 COL2

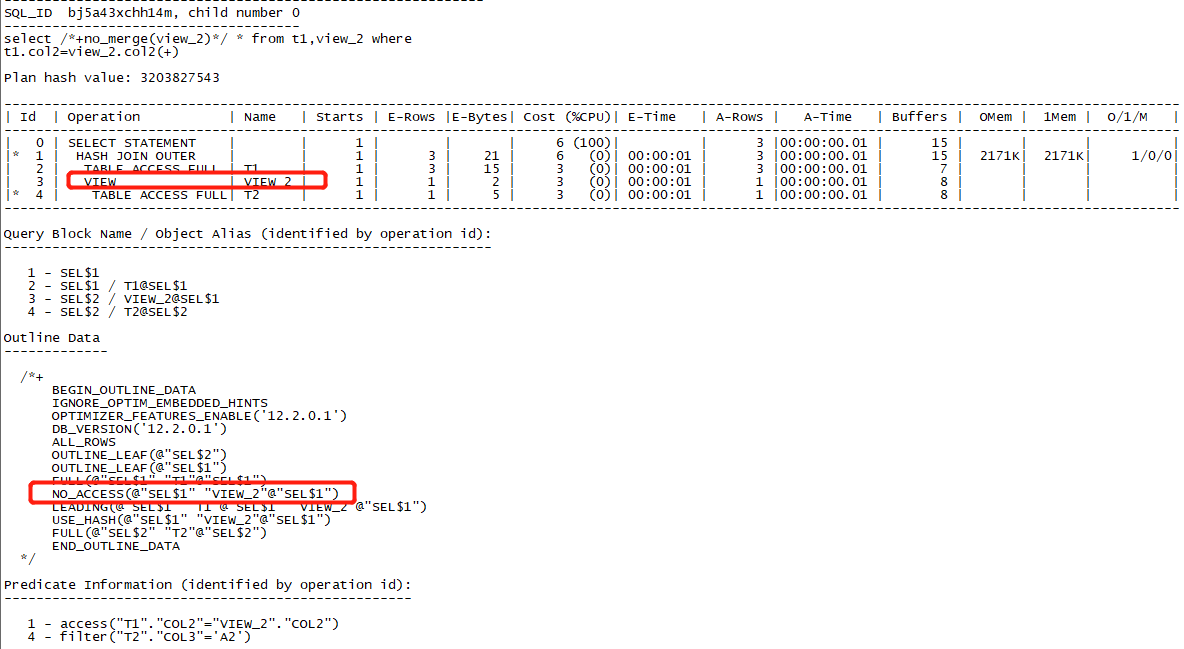
---------- ----- -----

1 A A

但是在这里着重强调这一点，就是以防未来在改写存在外连接的sql时，一不留神改错了。

最后我们看一下不进行视图合并的执行计划：

select /\*+no\_merge(view\_2)\*/ \* from t1,view\_2 where t1.col2=view\_2.col2(+);



### 复杂视图合并

Complex view merging主要说的就是带group by,distinct的，因为这俩个操作在某些情况下可以推迟执行，如果能推迟执行，那就能进行视图合并。

但是复杂视图的视图合并并不总是能带来效率提升的，因为先做group by,distinct可能可以优先过滤掉大量数据，且一旦进行复杂视图合并，也就意味着舍弃掉了不合并的执行路径。

因为group by,distinct这种都是针对一个sql最后的操作，如果视图合并，意味着要先对基表进行关联操作等等，最后再来group by，distinct；如果仍想先做group by，distinct那就必须先把视图中的查询完成，也就不存在视图合并了。

所以，对于复杂视图的视图合并，ORACLE会基于成本进行考虑，如果合并后的成本更高，则CBO会选择不合并。

#### Group by的例子

下面我们来看一个有group by的复杂视图的例子：

create or replace view view\_3 as

select department\_id, job\_id, sum(salary) total

from employees

group by department\_id, job\_id;

select t1.department\_name, t2.job\_title

from departments t1, jobs t2, view\_3 t3

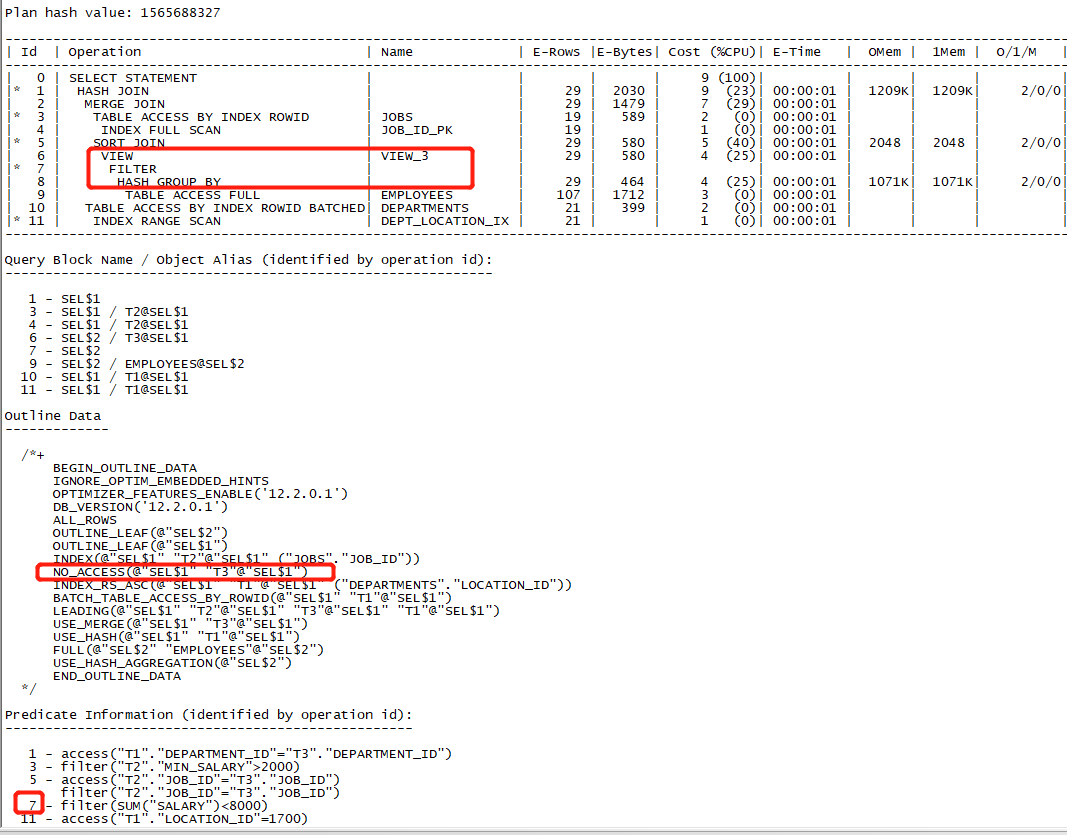
where t1.department\_id = t3.department\_id

and t2.job\_id = t3.job\_id

and t1.location\_id = 1700

and t3.total < 8000

and t2.min\_salary > 2000;



这里我们并没有看到复杂视图合并，我们看到在视图内先把group by做了，顺便把外面的过滤条件也拿到视图里做了。那这个sql到底能不能进行视图合并呢？我们加hint试试：

select /\*+merge(t3)\*/t1.department\_name, t2.job\_title

from departments t1, jobs t2, view\_3 t3

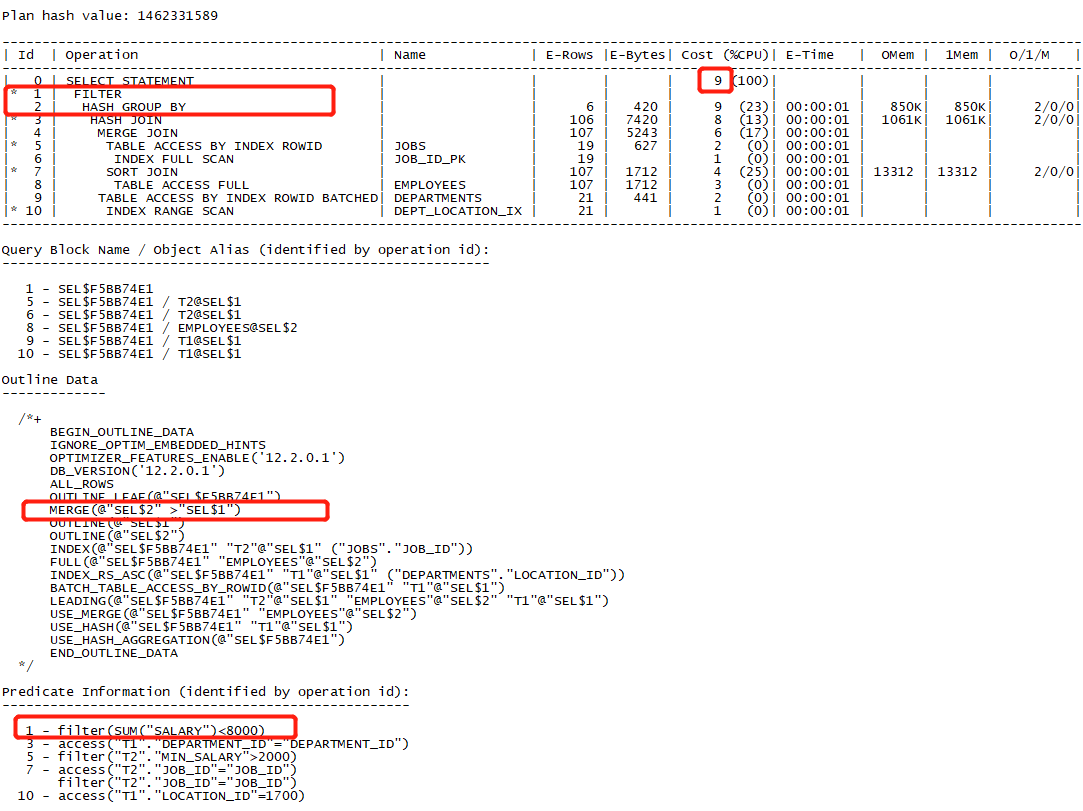
where t1.department\_id = t3.department\_id

and t2.job\_id = t3.job\_id

and t1.location\_id = 1700

and t3.total < 8000

and t2.min\_salary > 2000;



确实是可以进行视图合并的，我们看到这里视图合并后的cost也是9，但是CBO并没有选择视图合并，所以说CBO会基于成本考虑，当视图合并后的cost小于不合并时的cost时，CBO才会选择复杂视图合并。

那进行复杂视图合并到底时怎么个流程呢？

首先把view\_3的定义带入到sql中：

select t1.department\_name, t2.job\_title

from departments t1, jobs t2,

(select department\_id, job\_id, sum(salary) total

from employees

group by department\_id, job\_id) t3

where t1.department\_id = t3.department\_id

and t2.job\_id = t3.job\_id

and t1.location\_id = 1700

and t3.total < 8000

and t2.min\_salary > 2000;

然后，我们思考一个事情。如果在一个查询中对同一列先分组再过滤和先过滤再按该列分组有区别么？显然时没有区别的。所以这就是复杂视图进行合并的原理。那这里我们就可以把t3.job\_id=t2.job\_id这个过滤条件先拿到t3中去，同时，t2.min\_salary>2000仅仅是使t2.jobid的选择范围变得更小，所以这个条件也可以直接拿到t3中去，那我们就可以把jobs这张表整体拿到t3当中去，看一下变成了什么样：

select t1.department\_name,t2.job\_title

from departments t1,

(select department\_id, job\_id, sum(salary) total

from employees e1,jobs t2 where e1.job\_id=t2.job\_id and t2.min\_salary > 2000

group by department\_id, e1.job\_id) t3

where t1.department\_id = t3.department\_id

and t1.location\_id = 1700

and t3.total < 8000;

上面这条sql显然执行不成功，因为job\_title从哪儿来？所以这里必须要把job\_title也要放进t3中，这样jobs这张表才算完全进去了。但问题又来了，job\_title该怎么进去呢？如果想放到选择列里面就必须同时存在于group by子句中，也就是变成group by department\_id,job\_id,job\_title。

那这样做会不会有问题呢？如果在jobs表中，job\_id与job\_title是多对一，那这样写没问题，但是如果是job\_id与job\_title的关系是一对多，那这样写就会导致分组数发生变化（本质上就是分组规则发生了变化）。那怎么才能让job\_title变得唯一呢？我们知道rowid是唯一的，所以把rowid和job\_title“绑”一起，job\_title不就不会影响分组了吗？！所以sql变成如下的样子：

select t1.department\_name

from departments t1,

(select department\_id, job\_id, sum(salary) total,t2.job\_title

from employees e1,jobs t2 where e1.job\_id=t2.job\_id and t2.min\_salary > 2000

group by department\_id, e1.job\_id, t2.job\_title,t2.rowid) t3

where t1.department\_id = t3.department\_id

and t1.location\_id = 1700

and t3.total < 8000;

之后对于departments做同样的处理：

select \*

from (select e1.department\_id,

e1.job\_id,

sum(salary) total,

t2.job\_title,

t1.department\_name

from employees e1, jobs t2, departments t1

where e1.job\_id = t2.job\_id

and t2.min\_salary > 2000

and t1.department\_id = e1.department\_id

and t1.location\_id = 1700

group by e1.department\_id,

e1.job\_id,

t2.job\_title,

t2.rowid,

t1.department\_name,t1.rowid) t3

where t3.total < 8000;

但是我们原本的sql只查询俩列，所以最终版本就是：

select t1.department\_name, t2.job\_title

from employees e1, jobs t2, departments t1

where e1.job\_id = t2.job\_id

and t2.min\_salary > 2000

and t1.department\_id = e1.department\_id

and t1.location\_id = 1700

group by e1.department\_id,

e1.job\_id,

t2.job\_title,

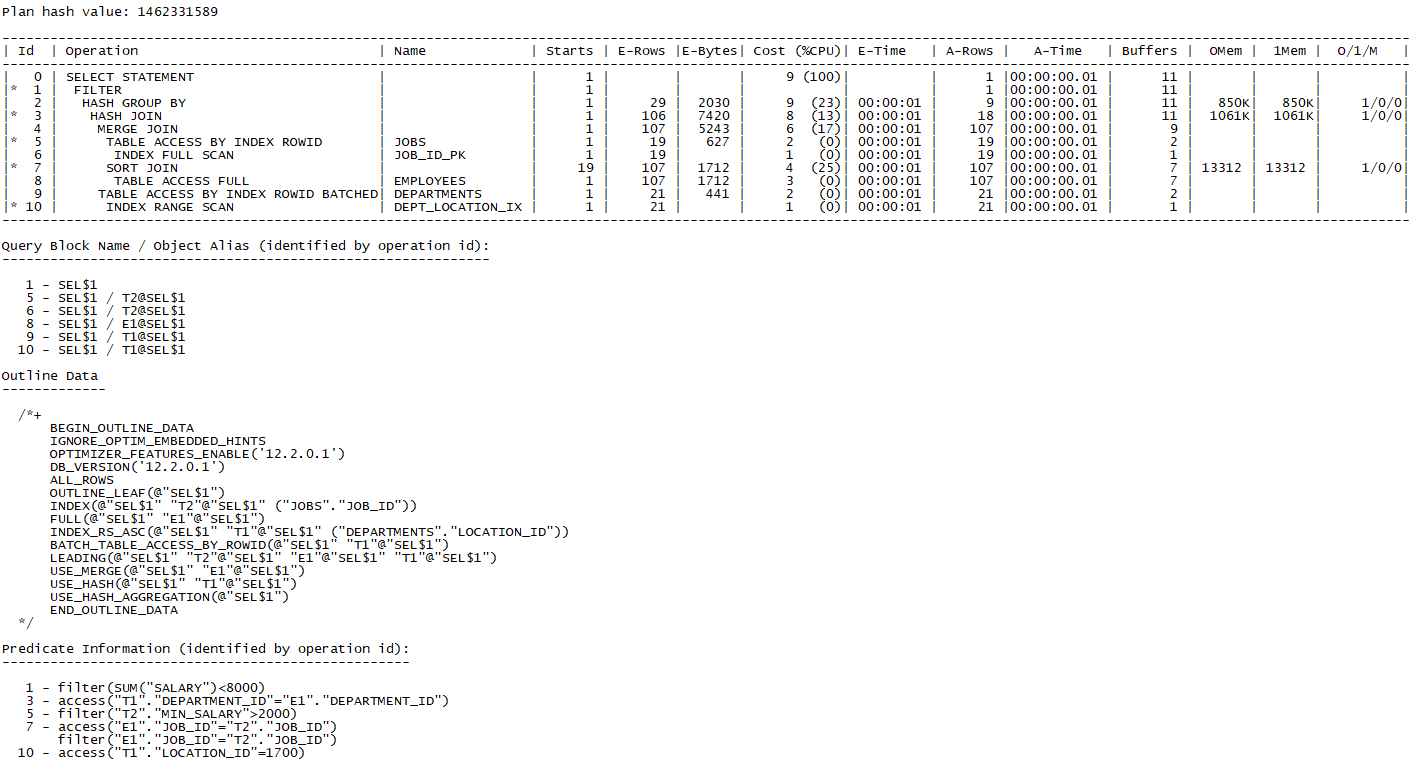
t2.rowid,

t1.department\_name,

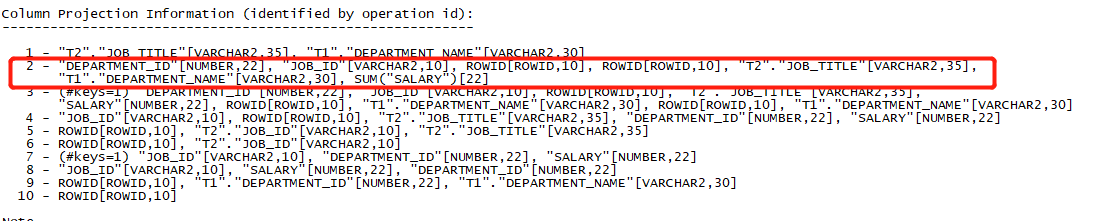
t1.rowid

having sum(salary)<8000;

我们看下这条sql的执行计划：



是不是一模一样！其实我们从原来的那个加了merge hint的sql的执行计划中的Column Projection Information中也能看出oracle在group by这里是怎么处理的：



Id=2的步骤就是hash group by的步骤，可以看到这一步用到了哪些列，确实跟我们在修改后的group by子句几乎一样。

崔华老师这里是把视图由内向外直接展开了，我的解释是从外往里的，我觉得我的解释应该更易懂一些，但是严格意义上是有问题的，我的解释更类似于后面所说的连接谓词推入，而视图合并的本质其实是将视图展开。

#### Distinct的例子

下面我们来看一个有distinct的复杂视图的例子：

create or replace view view\_4 as

select distinct department\_id,job\_id from employees;

select t1.department\_name, t2.job\_title

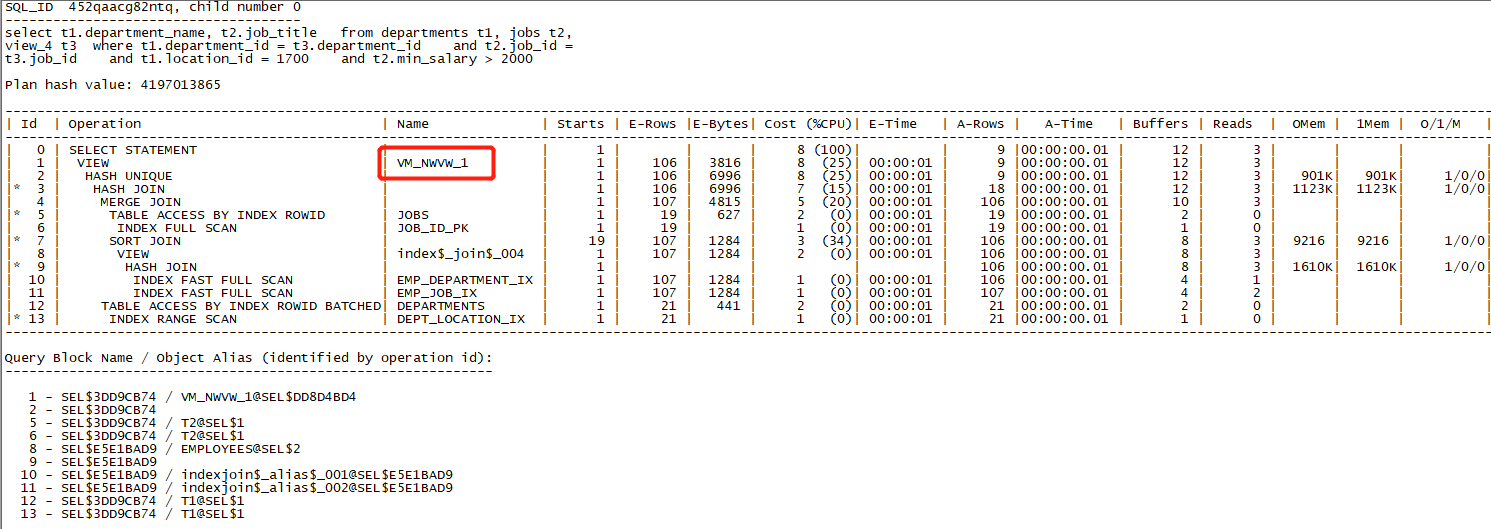
from departments t1, jobs t2, view\_4 t3

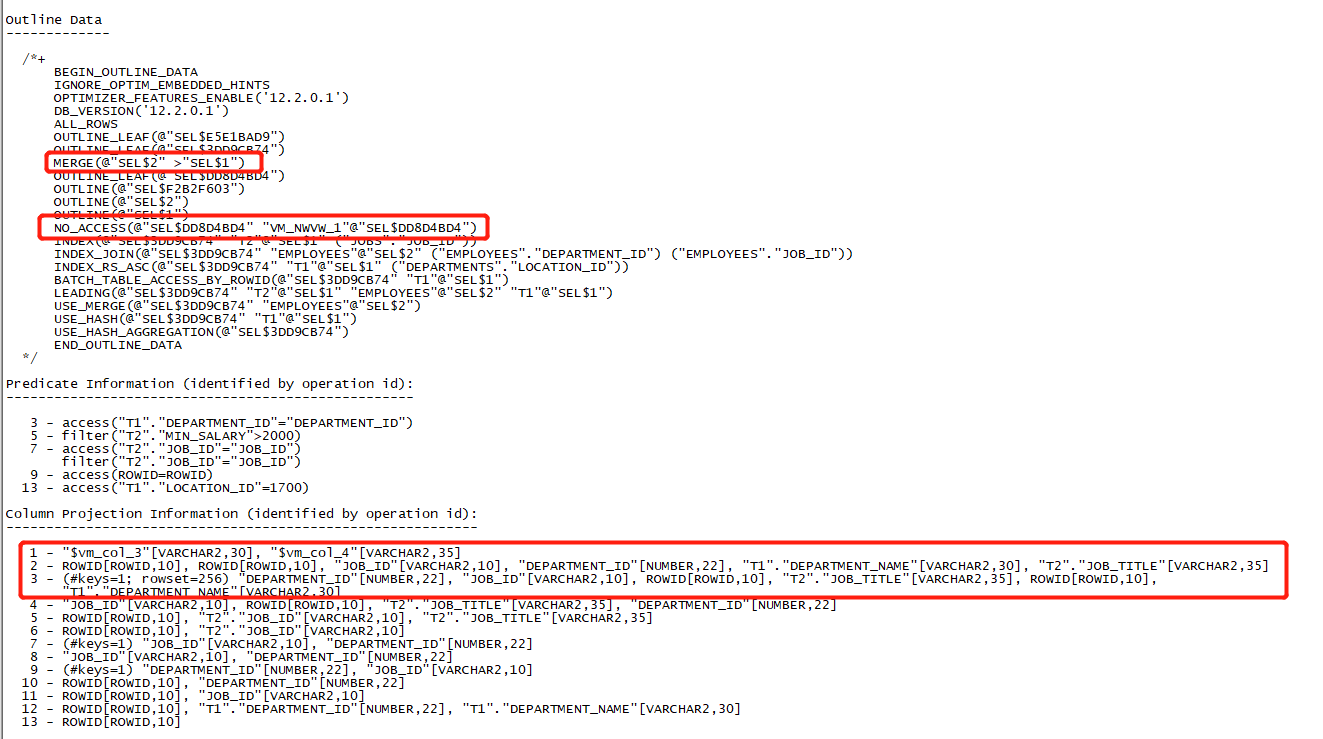
where t1.department\_id = t3.department\_id

and t2.job\_id = t3.job\_id

and t1.location\_id = 1700

and t2.min\_salary > 2000;





从这个执行计划我们看到视图合并确实发生了。而且hash unique在id=2才执行，说明去重行为后延了。

但是有个no\_access，注意no\_access的对象正是执行计划中id=1所对应的view，那这个view是什么呢？这个view是oracle自己产生的虚拟视图，用10053可以跟踪到（后面详细研究）

现在我们来对原sql按照执行计划来进行转换：

首先，还是带入视图定义：

select t1.department\_name, t2.job\_title

from departments t1, jobs t2,

(select distinct department\_id,job\_id from employees) t3

where t1.department\_id = t3.department\_id

and t2.job\_id = t3.job\_id

and t1.location\_id = 1700

and t2.min\_salary > 2000;

distinct其实本质就是分组，所以同之前一样，我们对分组列加加以限制，是不影响分组效果的，并且我们在将选择列带入到t3中的时候需要带着rowid：

select VM\_NWVW\_1.department\_name, VM\_NWVW\_1.job\_title

from (select distinct e1.department\_id,

e1.job\_id,

t1.department\_name,

t1.rowid,

t2.job\_title,

t2.rowid

from employees e1, departments t1, jobs t2

where e1.department\_id = t1.department\_id

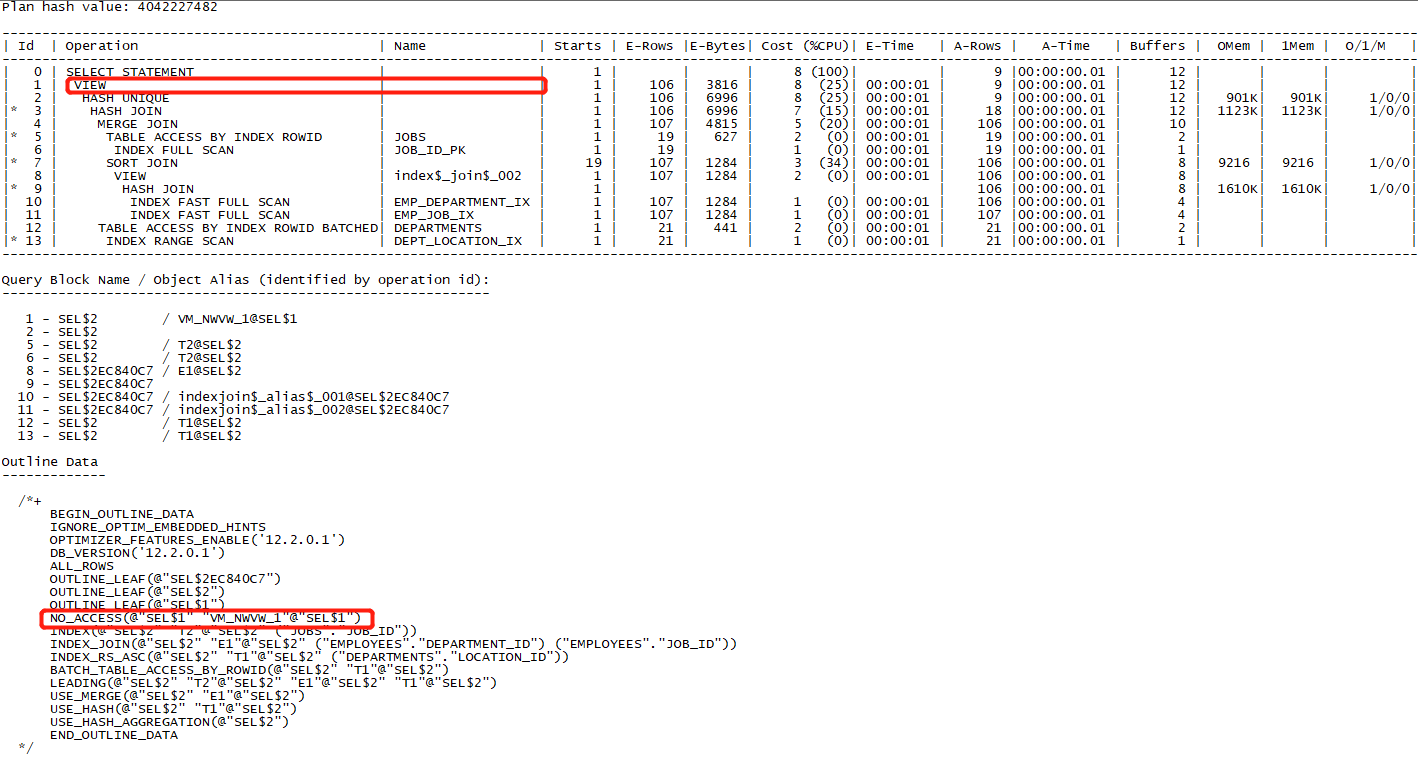
and e1.job\_id = t2.job\_id

and t1.location\_id = 1700

and t2.min\_salary > 2000) VM\_NWVW\_1;

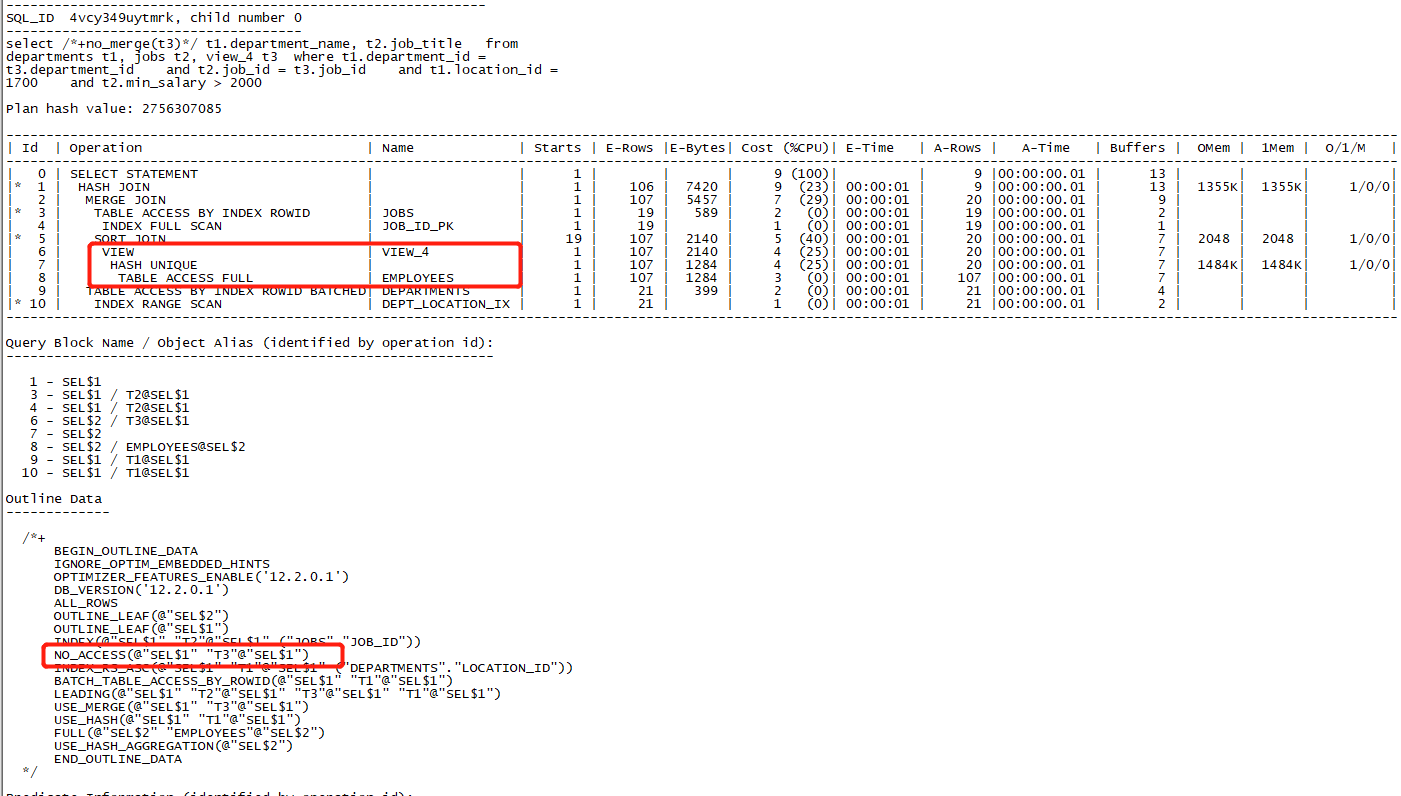
这里直接一步到位了，需要解释一下就是之前group by，可以用having子句把外面这层select去掉，但是这里就没办法了，为了保持与原sql语义一致，这里就必须套一层select，被套住得这部分就是oracle自己生成的一个虚拟视图VM\_NWVM\_1。

我们执行看下执行计划：



跟原计划一模一样，除了少显示了个VM\_NWVM\_1。

这里我们再看一下no\_merge的效果：



可以看到首先再视图中进行了unique操作，然后才同外面的表进行的关联。

为什么这里CBO就选择了复杂视图合并呢？因为合并后的cost是8，未合并的cost是9，所以CBO选择了复杂视图合并。

这里需要mark一下：就是带distinct的复杂视图进行了视图合并后，执行计划中还是有可能出现view的。

#### 相关的隐藏参数

SQL> SELECT ksppinm, ksppstvl, ksppdesc FROM x$ksppi x, x$ksppcv y WHERE x.indx = y.indx AND ksppinm='\_complex\_view\_merging';

KSPPINM KSPPSTVL KSPPDESC

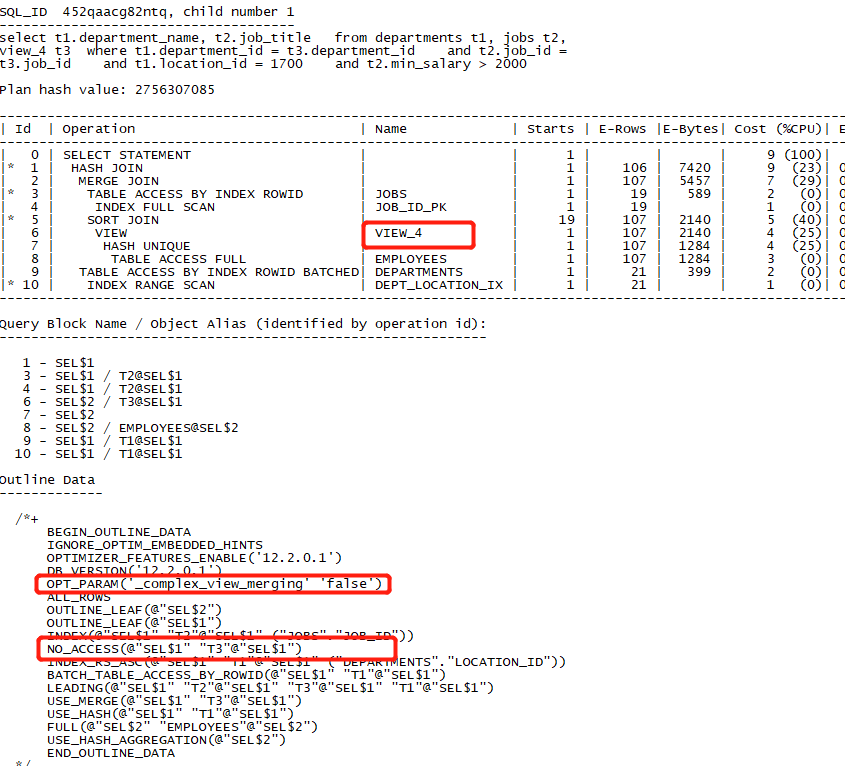
-------------- -------------- --------------------

\_complex\_view\_merging TRUE enable complex view merging

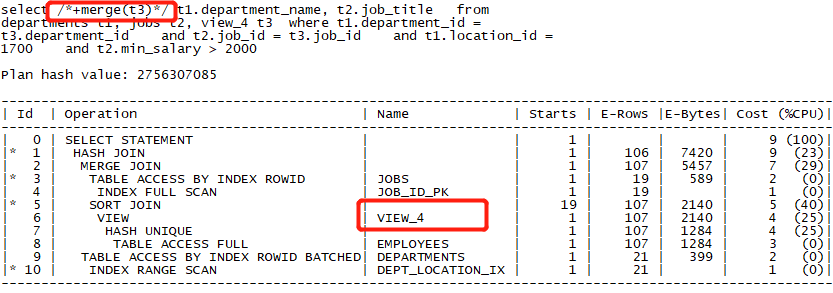
这个参数决定CBO是否可以做复杂视图合并的，默认是true，置为false，则不会进行复杂视图合并。看个例子：

SQL> alter session set "\_complex\_view\_merging"= false;

Session altered.



可以看到，确实CBO自己不会走复杂视图合并了。



可以看到加了hint也不走视图合并了。

### 视图合并总结

1. 简单视图和外连接视图无脑合并；
2. 复杂视图合并需要慎重考虑，合或者不合是个问题；
3. 外连接视图合并：视图是驱动表 or 视图由单一表构成；
4. distinct的视图合并后会产生一个VW\_NWVW\_1的虚拟视图；
5. hint是/\*+merge()\*/，不合并就是/\*+no\_merge()\*/。

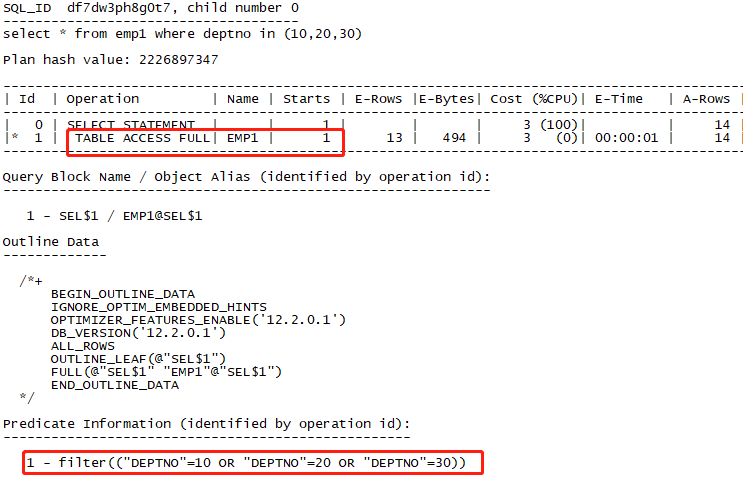
## ORACLE对IN的处理

### IN-List Iterator

IN-List Iterator是in后面常量集合的一种首选处理方法，这种处理就是把后面的常量集合遍历一遍，有就有没有就没有。

看个例子：

select \* from emp1 where deptno in (10,20,30);

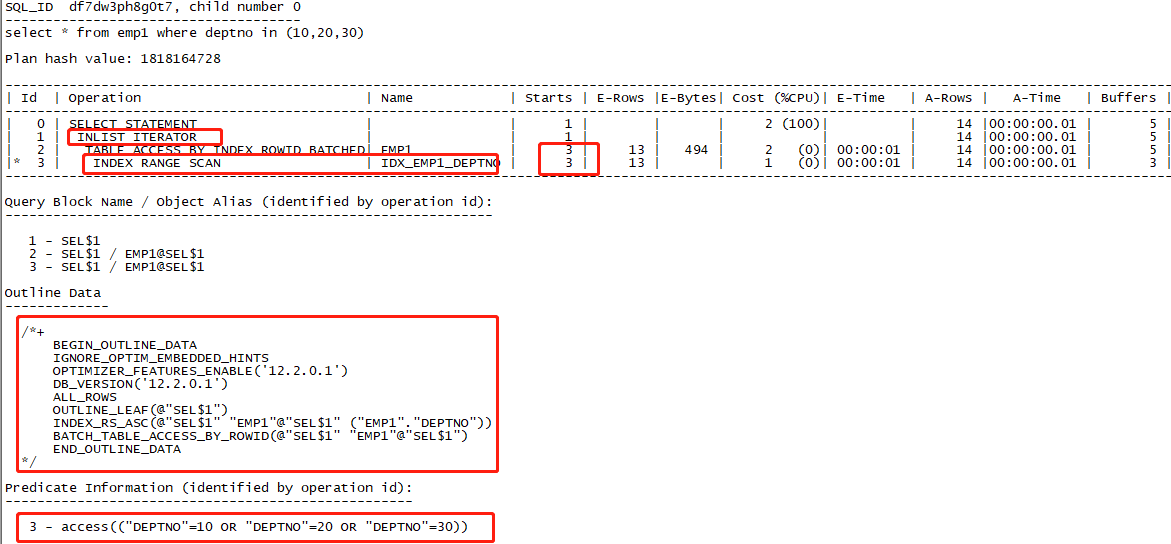


为什么走的是全表扫描，而不是IN-List Iterator呢？我们加个索引看看：

SQL> create index idx\_emp1\_deptno on emp1(deptno);

Index created.

在看执行计划：



现在我们看到，加了索引以后sql就走inlist iterator了，那说明，这种处理方法的一个必要前提条件就是，过滤列上必须有索引。

而且这里有一点需要注意，我们看到id=2和id=3循环次数是3，也就是说一共走了3遍索引扫描，这就是iterator的意思。

同时我们还注意到，在outline部分并没有任何表示该处理方式的hint，也就是说没有hint可以让优化器强行走inlist iterator的。

同时我们看到查询其实是被转化成了"DEPTNO"=10 OR "DEPTNO"=20 OR "DEPTNO"=30，我们按这个条件来执行一下sql：

select \* from emp1 where DEPTNO=10 OR DEPTNO=20 OR DEPTNO=30;



可以看到，确实一模一样。这是因为in(…)和…or…是等价的。

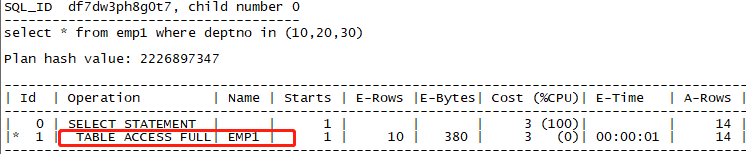
这里还要说明一点，虽然我们不能使用hint强制sql走inlist iterator，但是我们可以通过联合设置10157事件间接禁用掉inlist iterator。

SQL> alter session set events '10157 trace name context forever';

Session altered.

ORA-10157: CBO disable index access path for in-list

我们再来看：



可以看到，现在确实inlist iterator被禁用了。

### IN-List Expansion/OR Expansion

IN-List Expansion也是优化器对in后常量集合的一种处理，因为in和or是等价的，所以也叫做OR Expansion。

这种处理方式是把in后的每一个常量拿出来做单独查询，然后将查询结果union all。所以这种处理方法的本质是将查询转化为union all的写法。

我们可以使用use\_concat来使sql走inlist expansion：

select /\*+use\_concat\*/ \* from emp1 where deptno in (10,20,30);



可以看到concatenation字样，同时我们看到outline中的hint是use\_concat，而且通过谓语条件以及执行计划中的union all，也确实说明了sql被转化成了union all，且每个分支各自走索引。

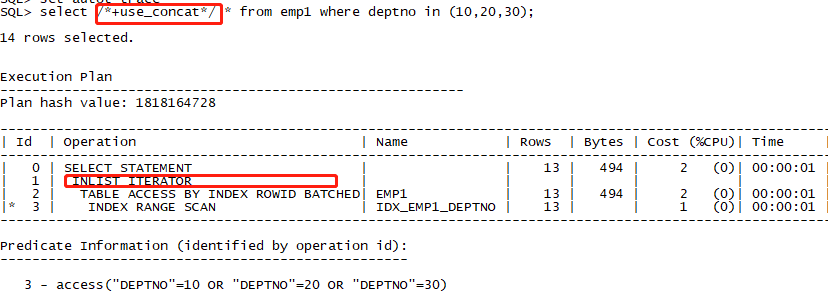
但是为什么我们不加hint，之前sql不会这样走呢？

首先有俩个原因，第一个就是加了hint，inlist iterator也是首选，我们先取消掉10157事件：

SQL> alter session set events '10157 trace name context off';

Session altered.

然后看执行计划：



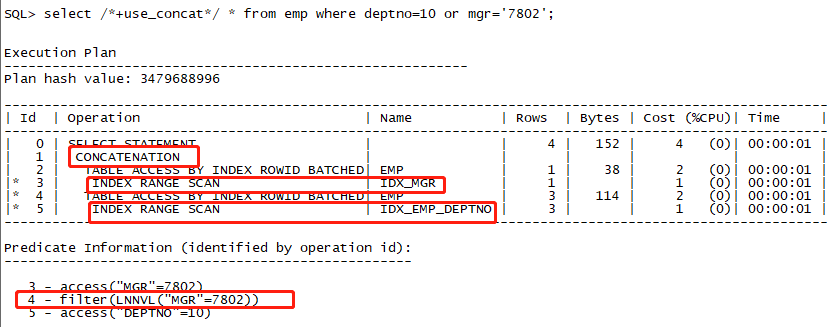
可以看到，在可以使用inlist iterator的时候会首选inlist iterator，用了use\_concat也没用。

第二个原因就是，CBO做inlist expansion是考虑代价的，我们看到全表扫描的cost是比使用inlist expansion的cost要小。

但是inlist expansion的用武之地并不在这里，而是如下：

select /\*+use\_concat\*/ \* from emp where deptno=10 or mgr='7802';

这里我们把hint加上，因为表太小，全表扫描很高效，所以cbo不会考虑OR扩展：



我们可以看到，因为做inlist expansion，所以使用到了mgr列和deptno列各自的索引，对于大表而言，这相对于全表扫描显然是极高效的！

我们还看到谓语信息中存在filter(LNNVL("MGR"=7802))，这表示sql被转换成了：

select \* from emp where mgr='7802'

union all

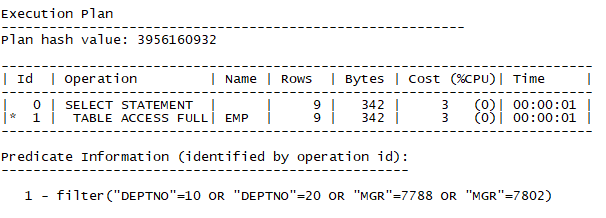
select \* from emp where deptno=10 and LNNVL(mgr='7802');

意思就是把既满足deptno=10同时又满足mgr=7802或者mgr为null的情况除去了（这里一定要记住可能存在deptno=10 and mgr is null的情况）。

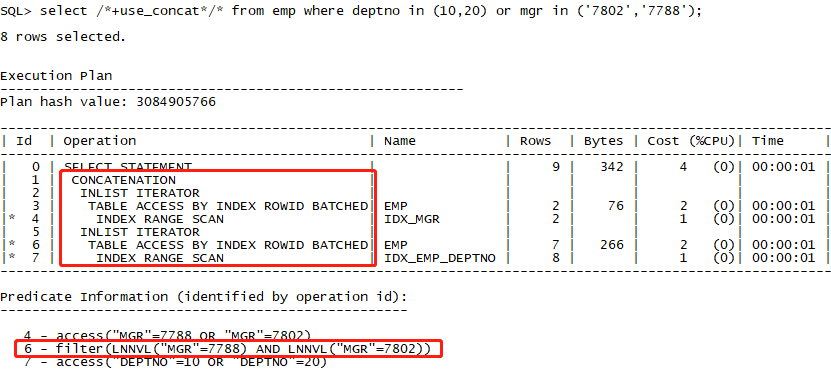
这里我解释下，这个明明就是OR扩展嘛，怎么还是叫inlist expansion，其实in和or总归都可以当作union all的简写，就是个写法而已，不必纠结。

那么inlist扩展和inlist iterator能不能联合使用呢？

select \* from emp where deptno in (10,20) or mgr in ('7802','7788');



可以看到执行计划并不如我们愿，因为这里想走inlist iterator，必须的先走inlist扩展，所以我们用use\_concat应该就能解决问题：

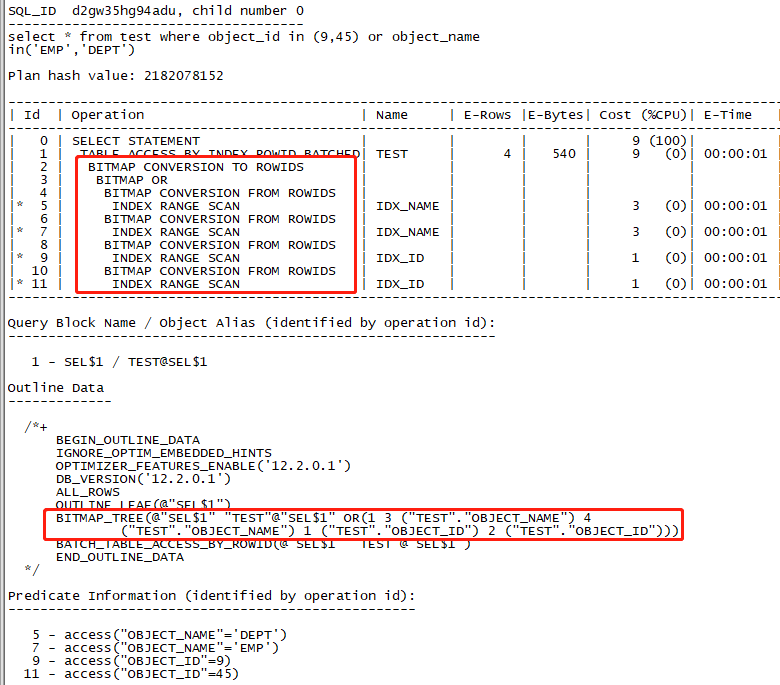


可以看到，确实如此。

但是往往因为统计信息过旧，导致cost计算不准，那么sql可选择错误的执行计划。比如我们的inlist的常量很多，那么就光转换就很耗时，这时候全表扫描就要更高效。可能有人说用/\*+full()\*/就得了，但我们还是介绍一下让sql不进行inlist扩展得hint：/\*+NO\_EXPAND\*/。

这里举个使用该hint的例子：

select \* from test where object\_id in (9,45) or object\_name in('EMP','DEPT');



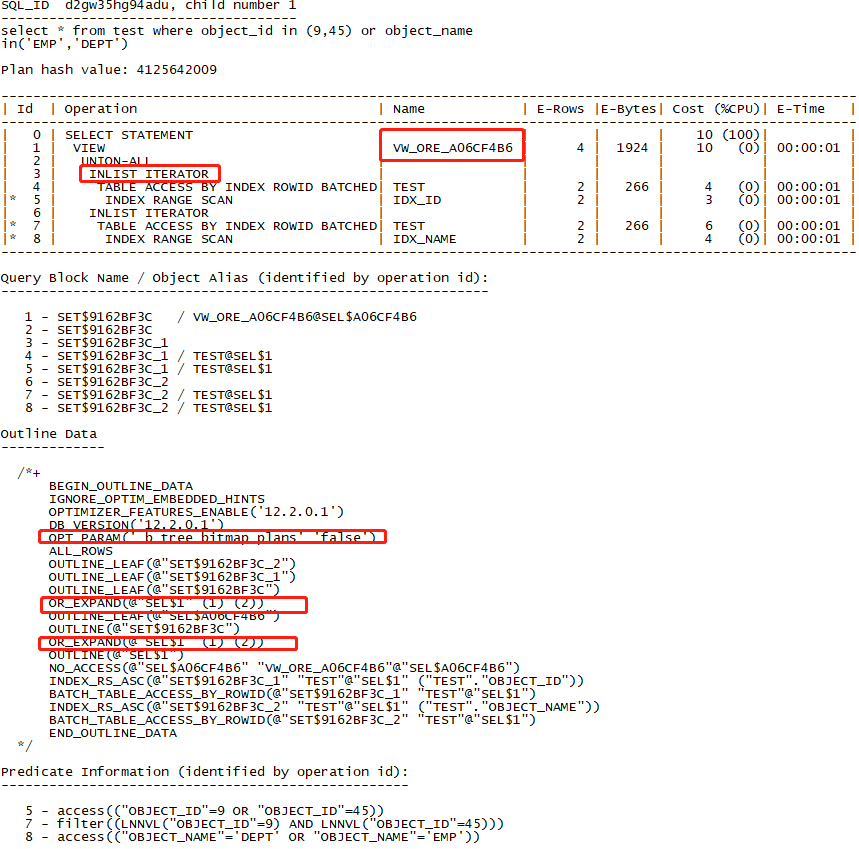
这里出现一个小插曲，也算是涨见识了。这里我们先不深究这个。

我们设置一个隐藏参数来避开这种转换：

SQL> alter session set "\_b\_tree\_bitmap\_plans"=false;

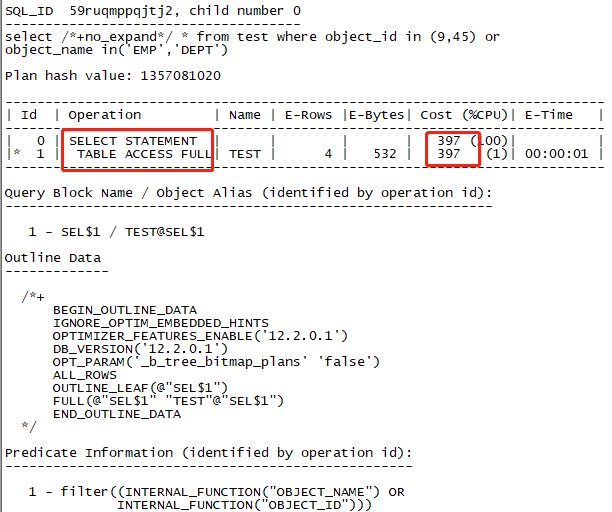
Session altered.

然后我们再执行：



可以看到这里出来个视图，ORE应该就是OR EXPAND的意思。不过我们还是达到了我们的目的，现在我们试下no\_expand：

select /\*+no\_expand\*/ \* from test where object\_id in (9,45) or object\_name in('EMP','DEPT');



我们看到hint生效了，同时也注意到确实cost要比不进行inlist扩展要高的多。

#### OR扩展的补充说明

对于如下sql：

select e.empno, e.sal, d.loc

from emp\_tmp e, dept d

where e.deptno = d.deptno(+)

and d.dname(+) = 'RESEARCH'

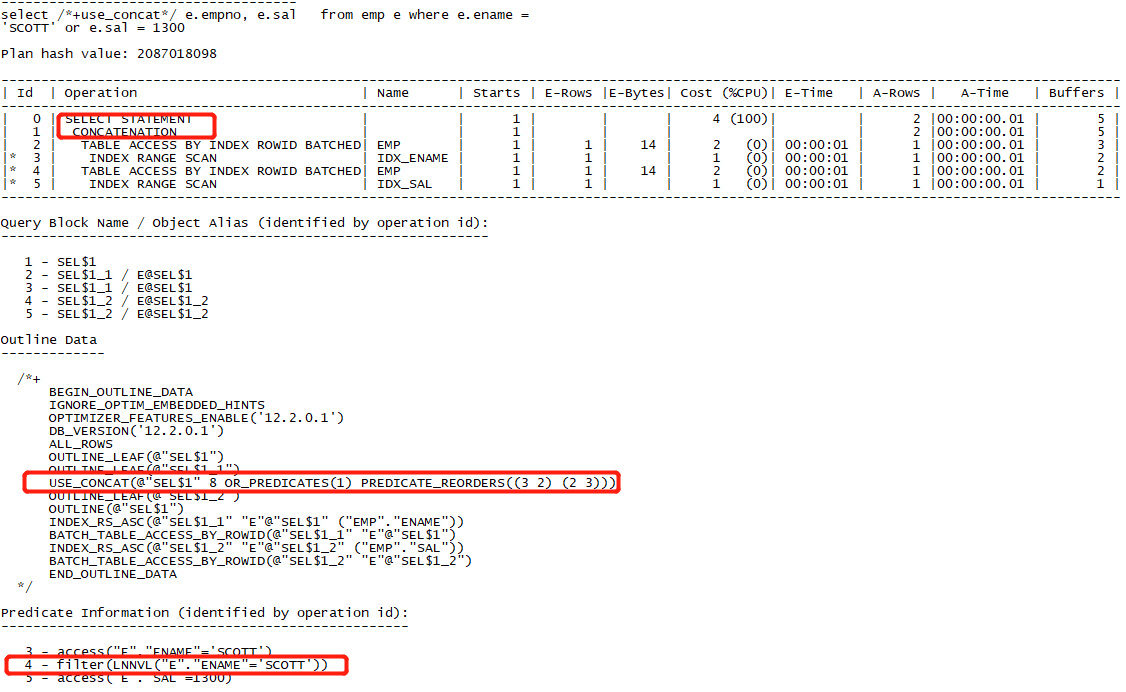
and (e.ename = 'SCOTT' or exists

(select hisal from salgrade s where s.hisal = e.sal));

为什么不能做or扩展呢？！因为or扩展的一个前提条件就是各个分支都要有索引，否则使用use\_concat也没用。这就是为什么我之前说改sql改成union只能是约等于or扩展。我们先来来验证该结论：

SQL> select /\*+use\_concat\*/ e.empno, e.sal from emp e where e.ename = 'SCOTT' or e.sal = 1300;

我们现在给sal列加上索引再执行sql：



现在很清楚的证明了我们的结论，而对于一个or exits，我们看到filter的谓词信息是 or is not null，也就不存在走索引，所以即使使用use\_concat的hint也不会走or扩展。

我们来改写sql看下执行计划：

select e.empno, e.sal, d.loc

from emp\_tmp e, dept d

where e.deptno = d.deptno(+)

and d.dname(+) = 'RESEARCH'

and e.ename = 'SCOTT'

union

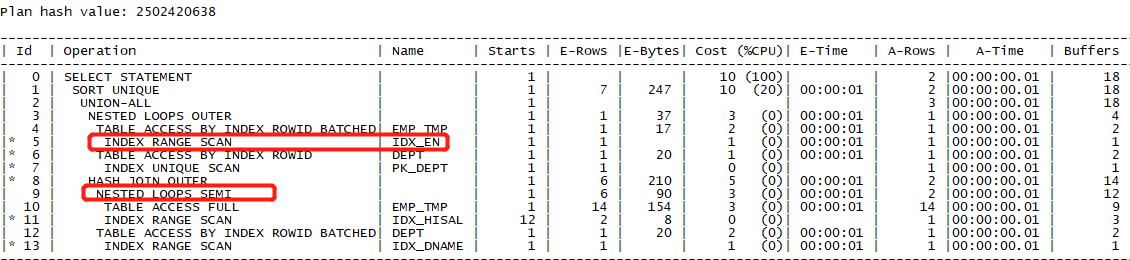
select e.empno, e.sal, d.loc

from emp\_tmp e, dept d

where e.deptno = d.deptno(+)

and d.dname(+) = 'RESEARCH'

and exists (select hisal from salgrade s where s.hisal = e.sal);



综上，目前12.2的版本是想要or扩展，必须要分支有索引。我认为是ORACLE设计不完善的地方。反正目前来看，遇到这种or+子查询的情况，按照or扩展的原理，降or改写为union是最妥善的处理。

### IN-List Filter

这节内容在前面的子查询展开有详细的分析，这里就不做过多赘述了，就是让in后面的子查询不展开，就会发生inlist filter。

### 对in子查询展开

这一小节在前面的子查询展开也做了很详细的说明，这里就额外说明一点：如果子查询中有视图，那么在展开子查询后，能做视图合并就还要做视图合并，不能做就只是子查询展开。

掌握了子查询的展开流程和视图合并的流程和条件，这一点很好理解，这里也不赘述了。

### ORACLE对IN的处理的总结

1. 子查询部分参照子查询展开；
2. 当in后面是常量集合时，才能使用inlist iterator或者inlist expansion；
3. inlist expansion/or expansion的hint是/\*+use\_concat\*/,相对应的不扩展的hint是/\*+no\_expand\*/；
4. OR扩展考虑代价
5. inlist iterator是处理in后面常量集合的首选处理办法，但必须列上有索引！
6. inlist iterator没有hint
7. 10157禁用inlist使用索引，那也就相当于禁用了inlist iterator
8. alter session set "\_b\_tree\_bitmap\_plans"=false;（后面研究）

## 连接谓词推入（Join Predicate Pushdown）

连接谓词推入也是针对视图的查询转换，概括的说，就是在不做视图合并的情况下，将外部表与视图的连接条件推入到视图中，让视图可以走基表在该列上的索引。

这里的这个不做视图合并包括：能做但不做和本来就不能做。

从连接谓词推入流程来看，是不是很像NL，没错，推入以后走的就是nl。我们先准备如下实验环境：

SQL> create table emp1 as select \* from emp;

Table created.

SQL> create table emp2 as select \* from emp;

Table created.

SQL> create index idx\_emp1 on emp1(empno);

Index created.

SQL> create index idx\_emp2 on emp2(empno);

Index created.

SQL> create view emp\_view as select emp1.empno empno1 from emp1;

View created.

SQL> create view emp\_view\_union as select emp1.empno empno1 from emp1 union all select emp2.empno empno1 from emp2;

View created.

### 可merge但是no merge的情况

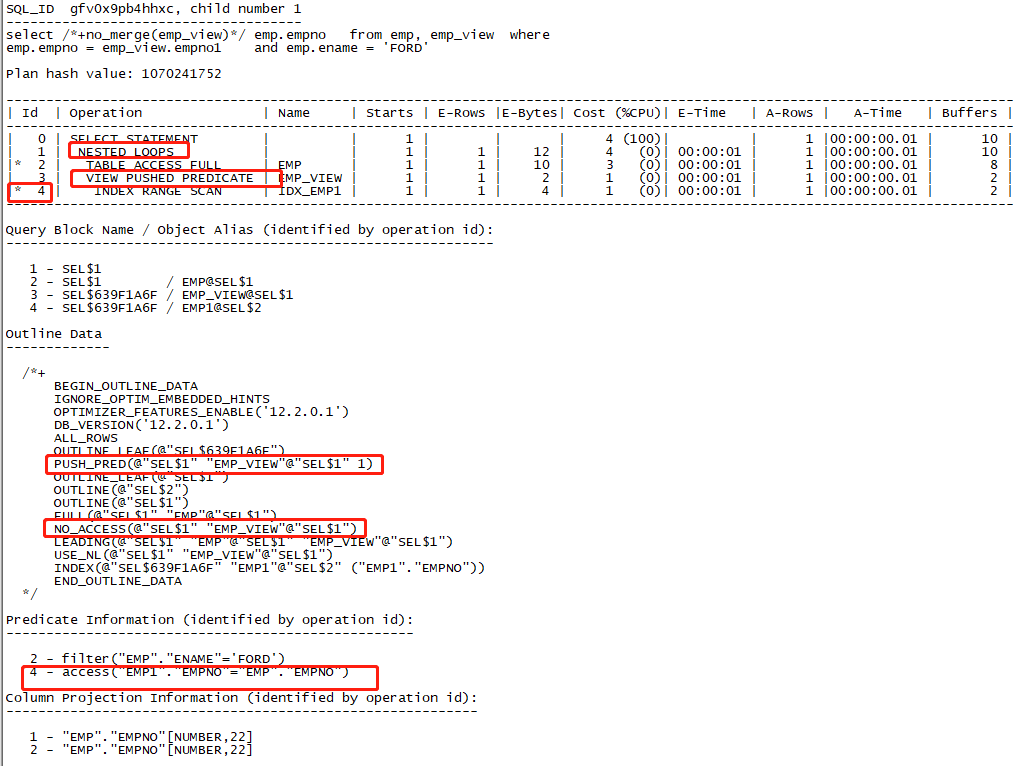
我们先来看个能做但不做视图合并的例子：

select /\*+no\_merge(emp\_view)\*/ emp.empno

from emp, emp\_view

where emp.empno = emp\_view.empno1

and emp.ename = 'FORD';



从上面的执行计划我们可以得到如下三条信息：

1. 我们看到了连接谓词推入关键词--view pushed predicate，后面对应的name列是没展开的视图名；
2. 确实做的是NL，而不是filter。通过谓语信息也可以看出，如果是filter应该是access(emp1.empno=:B1)；
3. 确实走到了emp1上empno列上的索引。

那么我们现在有三个疑问：

1. 连接谓词推入看不看代价，因为我们知道nl性能好需要满足俩个条件，被驱动表列有索引（已满足），驱动表结果集小，那如果外部表过滤后结果集仍很大，显然连接谓词推入是不合理的；
2. 这个NL是不是真的是由于连接谓词推入产生的；
3. 连接谓词不推入是不是就走不到emp1表的索引。

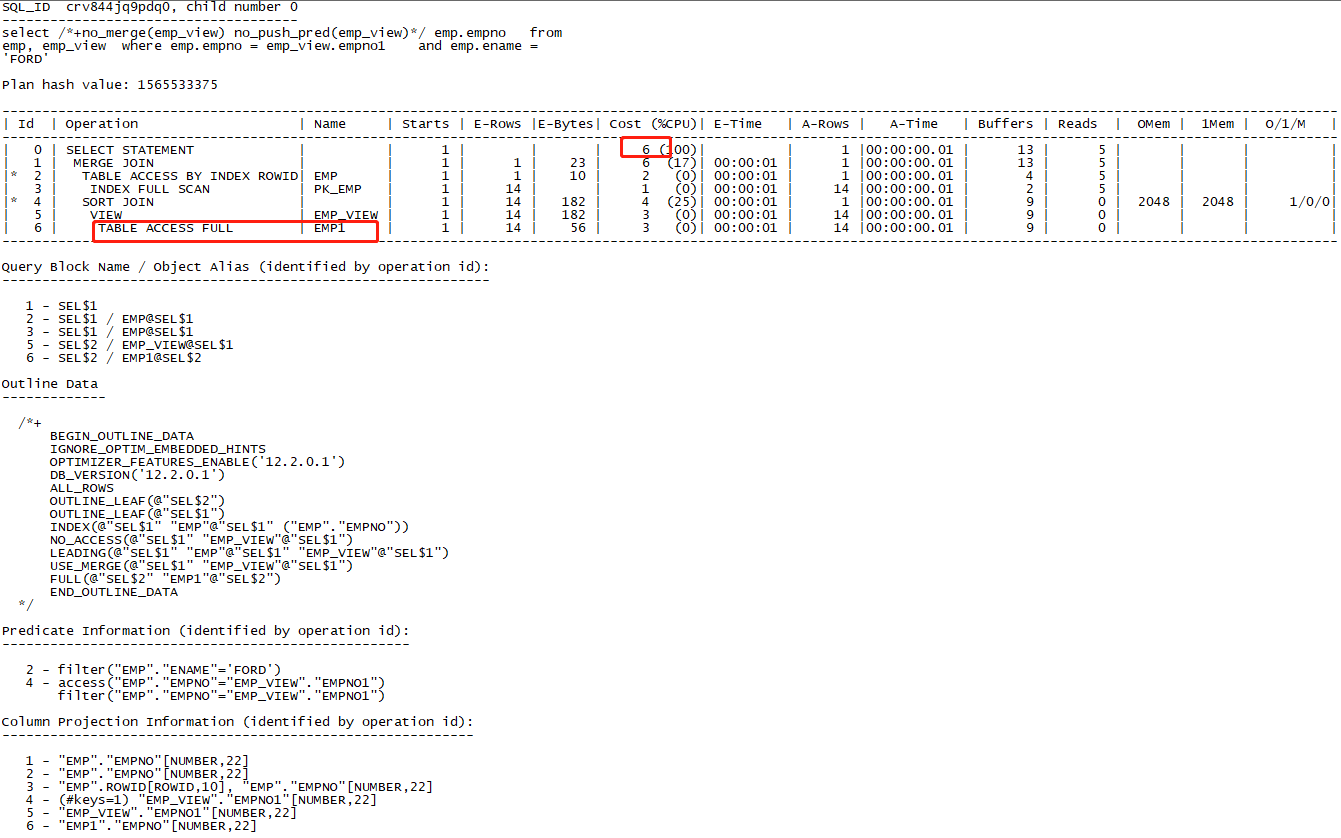
下面我们来做俩个实验：

select /\*+no\_merge(emp\_view) no\_push\_pred(emp\_view)\*/ emp.empno

from emp, emp\_view

where emp.empno = emp\_view.empno1

and emp.ename = 'FORD';



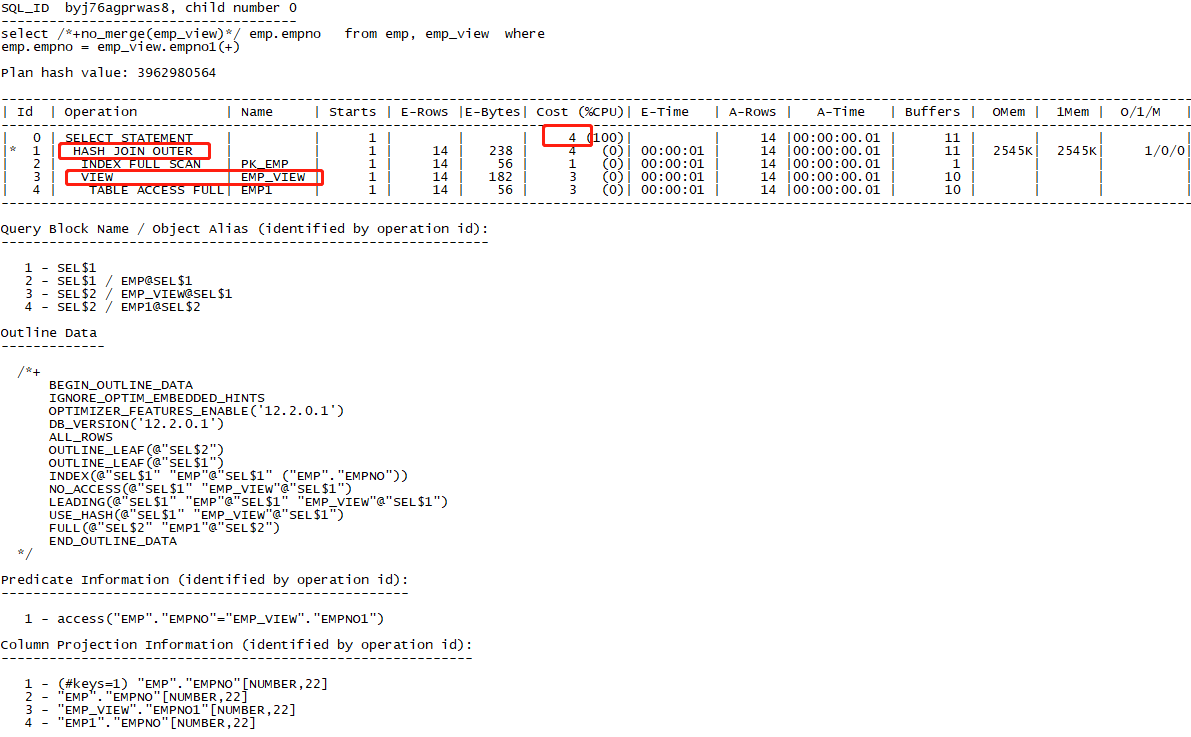
我们可以看到，确实没有走NL了，没有连接谓词推入，也就没办法用到基表emp1的索引列，所以我们看到emp1走的是全表扫描。同时这里我们注意到cost是6，确实别之前做了谓词推入的cost要大。

我们下面再做一个实验，来进一步验证连接谓词推入是否考虑代价。

select /\*+no\_merge(emp\_view)\*/ emp.empno

from emp, emp\_view

where emp.empno = emp\_view.empno1(+);



可以看到，当我们把ename这个过滤条件去掉之后，cbo就没有再选择连接谓词推入。

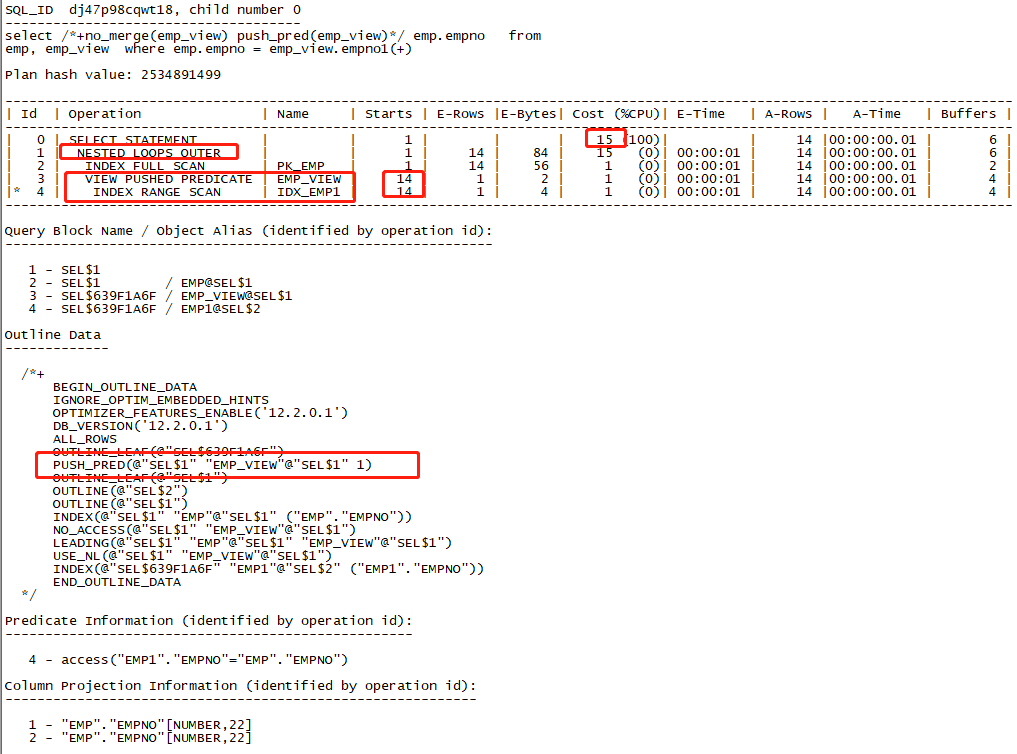
虽然我们这里把内连接变为外连接，但这并不影响结果，这样做的原因也是想把外连接视图合并的情况也展现一下。

为了确认cbo做连接谓词推入确实考虑了成本，我们再来看：

select /\*+no\_merge(emp\_view) push\_pred(emp\_view)\*/ emp.empno

from emp, emp\_view

where emp.empno = emp\_view.empno1(+);



这里我们可以很清楚的看到，视图确实又走了索引扫描，但是被执行了14次，cost也变成了15，确实比原来没有做连接谓词推入大。

所以说，cbo是要基于成本来考虑是否进行连接谓词推入的。

还有就是由连接谓词推入而产生的NL中，被推入的视图一定是被驱动表。

不过，我们这里注意到逻辑读其实还比之前的小，其实我觉得原因还是块儿少，体现不出来全表多块读的优势。

### 没法merge的情况

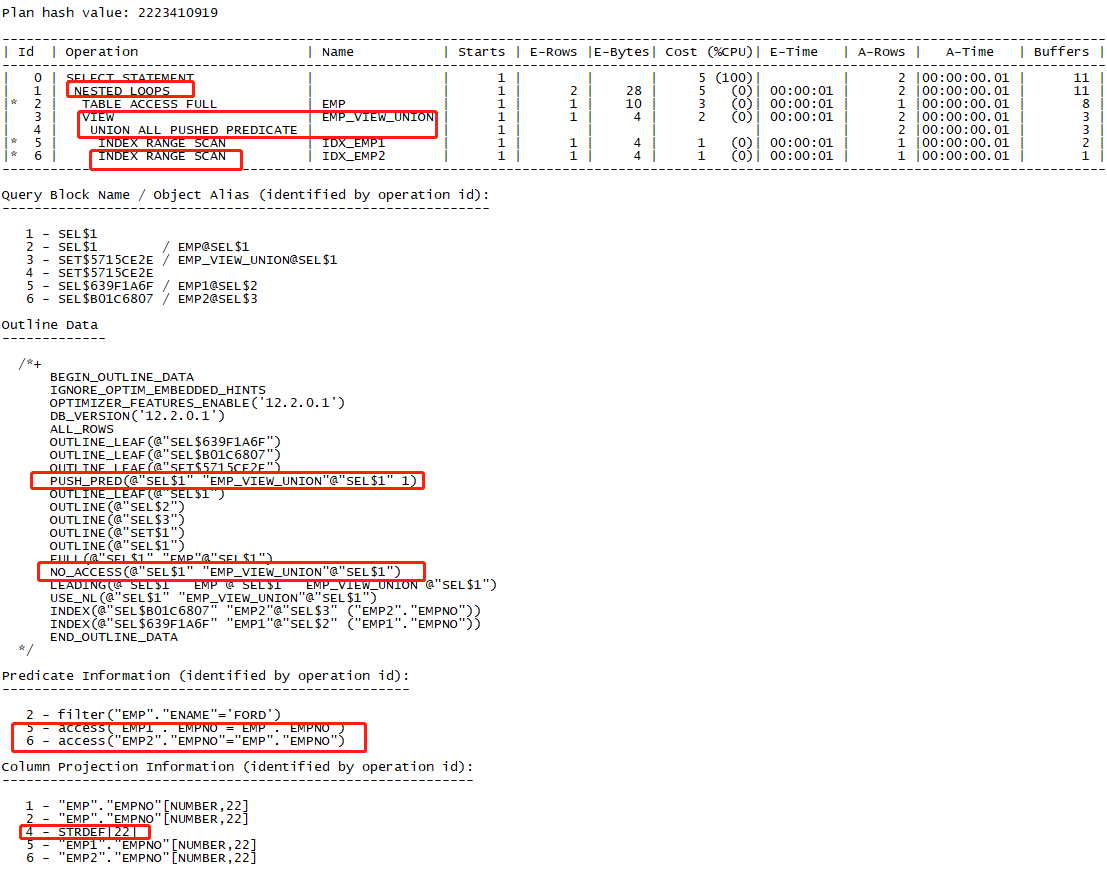
现在我们使用根本做不了视图合并的emp\_view\_union来做实验：

select emp.empno

from emp, emp\_view\_union

where emp.empno = emp\_view\_union.empno1

and emp.ename = 'FORD';



注：STRDEF这个后面找找是个啥

从上面的执行计划我们可以看到：

1. 带union all的视图确实做不了视图合并；
2. 这次关键词成了union all pushed predicate，说明连接谓词先推入到view，然后再分别推入到union all的各个分支；
3. 确实又再次看到了NL，并且俩个分支的表都走了各自的索引。

加上hint不让它连接谓词推入来看看：

select /\*+no\_push\_pred(emp\_view\_union)\*/emp.empno

from emp, emp\_view\_union

where emp.empno = emp\_view\_union.empno1

and emp.ename = 'FORD';



然后就走俩个全表扫描了，这里cost是9，也确实比连接谓词推入的执行计划的cost要大。

我们想下，union all的一个特性就是每个分支可以走自己的索引，那我们删除一个索引是什么效果呢？

我们这里把idx\_emp2删掉。

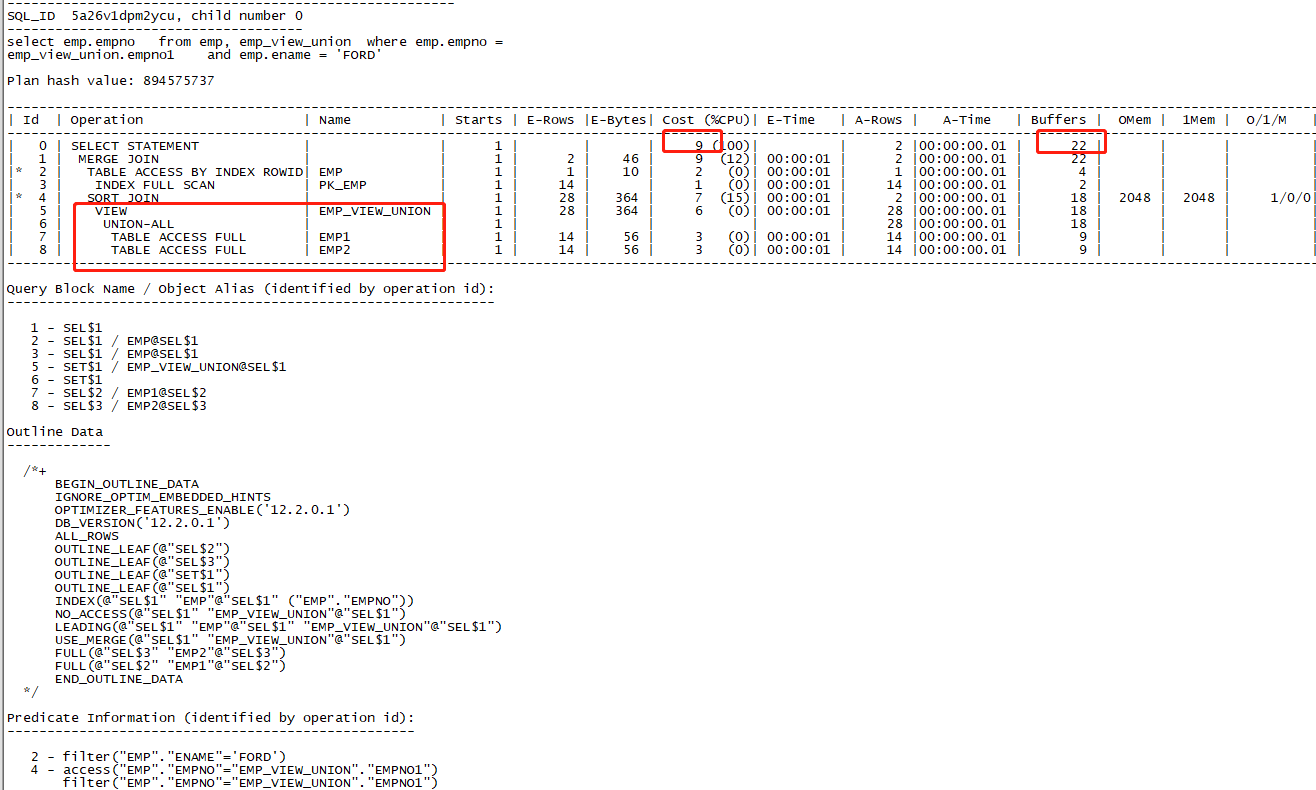
仍然执行原sql：

select emp.empno

from emp, emp\_view\_union

where emp.empno = emp\_view\_union.empno1

and emp.ename = 'FORD';



这里CBO选择了不进行连接谓词推入。那我们这里可以想一下，现在外部驱动表的结果集只有一条数据，那即使做nl，也是只对视图做一次查询，但是emp1却可以走索引，就减少了一次全表扫描，那我们是不是可以通过让连接谓词推入来达到优化的效果呢？

select /\*+push\_pred(emp\_view\_union)\*/emp.empno

from emp, emp\_view\_union

where emp.empno = emp\_view\_union.empno1

and emp.ename = 'FORD';



可以看到连接谓词推入后，buffers变小了，达到的优化的目的。

同时我们也注意到，这里的cost其实是比没有做连接谓词推入的cost小的，但是cbo确实没选择连接谓词推入，可能这是它不完善的地方吧。

### 连接谓词推入总结

一、连接谓词推入会基于成本考虑；

二、可能发生连接谓词推入的情况如下：

1.视图定义中有union all/union、group by、distinct

2.视图是外部表的外连接、半连接、反连接、内连接的被驱动表。

三、连接谓词推入发生之后，一定做的是NESTED LOOP；

四、连接谓词的推入只能发生在视图不合并的情况。

五、连接谓词推入的hint是：/\*+push\_pred(view)\*/ 不推入：/\*+no\_push\_pred(view)\*/

这里备注一下：崔华老师说内连接简单视图CBO无法实现推入，可能是实验环境不同的缘故，可以确定的是，12.2的版本，CBO是可以实现这种情况的，我们第一个实验就证明了。

## 连接因式分解（Join Factorization）

连接因式分解是优化器处理带UNION ALL目标sql的一种方式。11gR2以后才有的。

既然是因式分解，顾名思义就是把目标sql中公有的部分拿出，可以理解为原sql是1\*2+3\*2，查询转换后的sql是(1+3)\*2。

### 单独的UNION ALL sql语句

下面我们直接来看一个例子：

select t2.prod\_id from sales t2,customers t3

where t2.cust\_id = t3.cust\_id

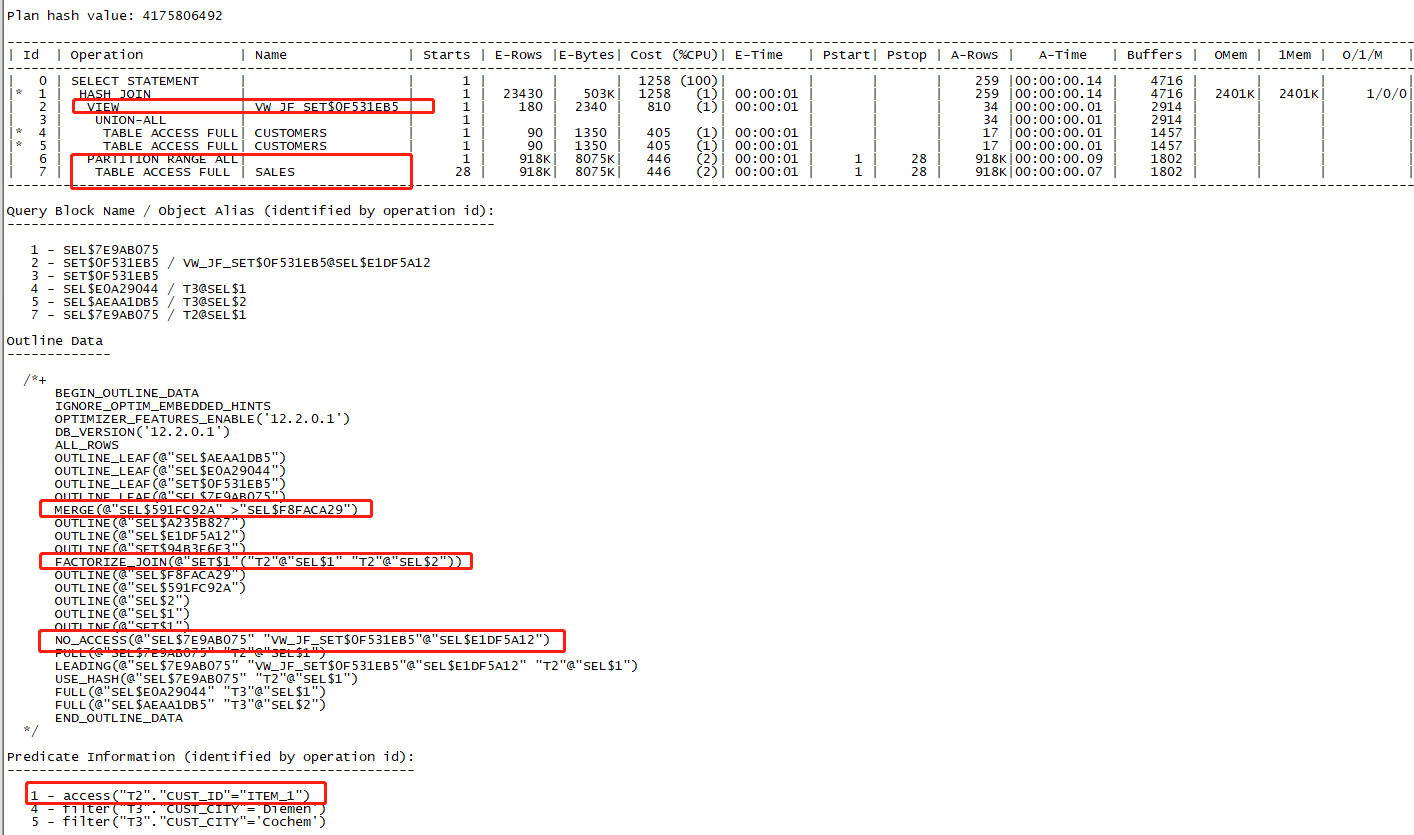
and t3.cust\_city='Diemen'

union all

select t2.prod\_id from sales t2,customers t3

where t2.cust\_id = t3.cust\_id

and t3.cust\_city='Cochem';



从上面的执行计划我们可以看到：

1. 出现一个视图，视图名字叫VW\_JF\_SET$0F531EB5，JF就是join factorization；
2. 确实把sales分离了出来，先对customers进行了俩次全表扫描，然后union all再同sales做连接。这样sales就只用被扫描一次，而不是同customers一同被扫描俩次。
3. outline中出现了视图合并的字样，这应该是一个中间过程，因为括号中的俩个qb\_name在Qurey Block Name部分并没有出现过。
4. 连接因式分解的hint出现了：FACTORIZE\_JOIN(@"SET$1"("T2"@"SEL$1" "T2"@"SEL$2"))
5. no\_access中的视图，正是执行计划中出现的视图名称
6. 谓语信息中，hash join那一步的access有点不一样：access("T2"."CUST\_ID"="ITEM\_1")

由上面我们可以总结出：连接因式分解的好处就是减少对公共因子的扫描次数，同时我个人认为我们看的merge并不是偶然，后面我们做演示的时候，就会发现连接因式分解其实就是视图不完全合并。

对于上面的例子我们进行一下过程推演：

首先把公共因子sales表提出来，然后剩余部分就是：

select t2.cust\_id from customers t3

where t3.cust\_city='Diemen'

union all

select t2.cust\_id from customers t3

where t3.cust\_city='Cochem';

然后原来的查询就变成了：

select t2.prod\_id from sales t2,

(select t3.cust\_id from customers t3

where t3.cust\_city='Diemen'

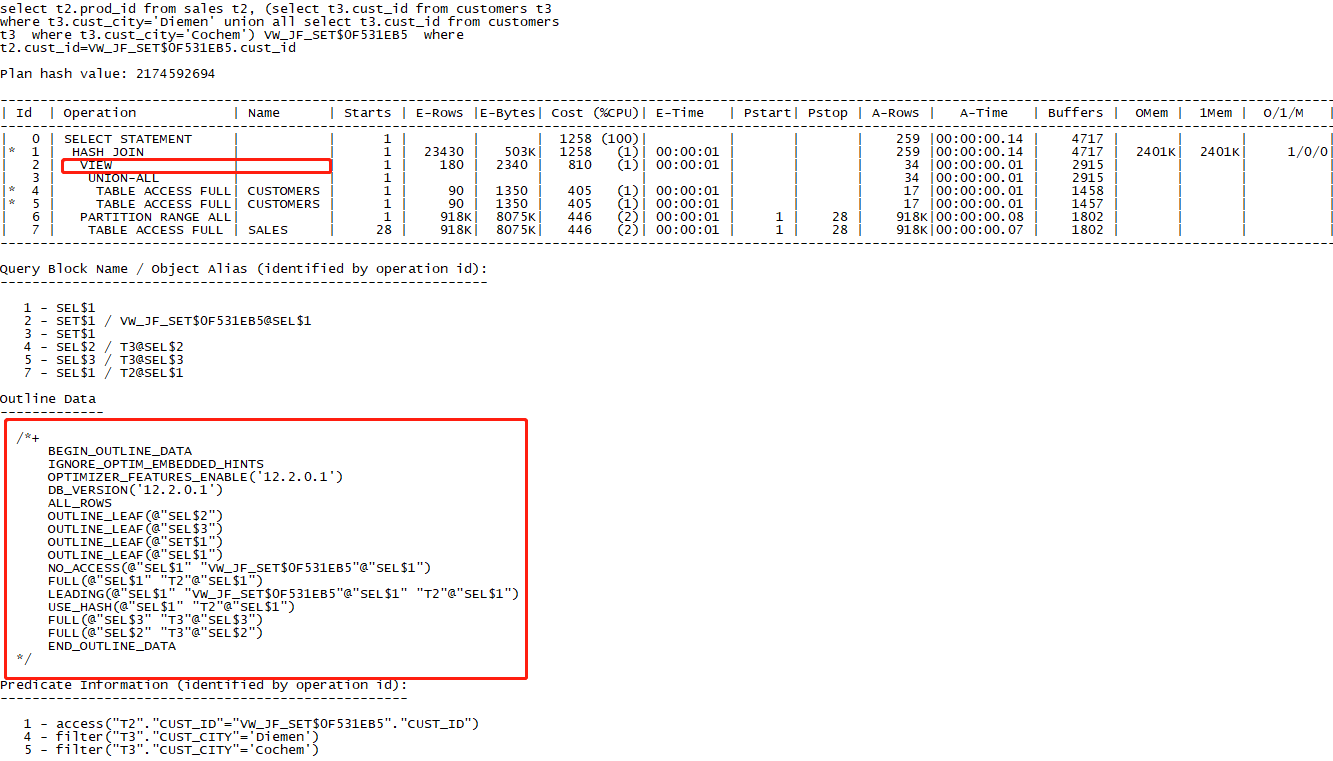
union all

select t3.cust\_id from customers t3

where t3.cust\_city='Cochem') VW\_JF\_SET$0F531EB5

where t2.cust\_id=VW\_JF\_SET$0F531EB5.cust\_id;

执行一下：



执行计划主体可以说是一模一样，不过我们这里也可以注意到，outline中没有merge了，所以我认为，oracle并不是直接把sql转换成了这个样子，后面我们拿10053跟踪一下看看。

如果我们这里把union all变成union，看一下效果呢？

select t2.prod\_id from sales t2,customers t3

where t2.cust\_id = t3.cust\_id

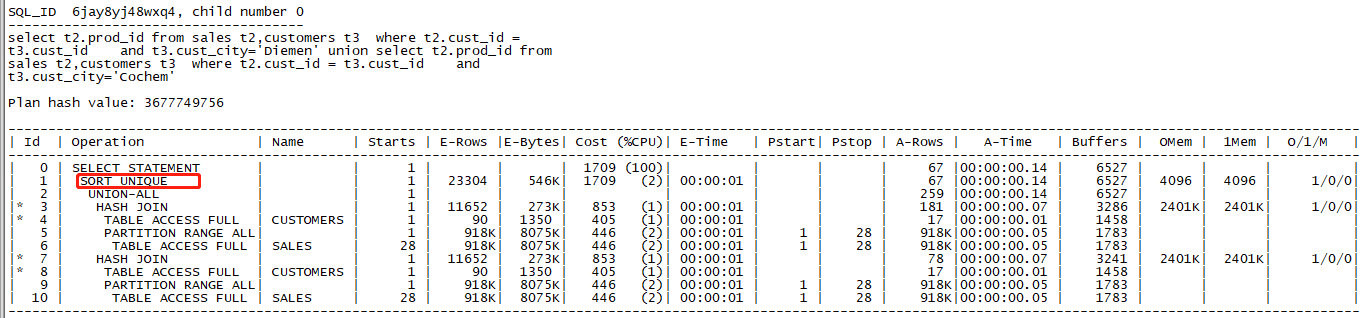
and t3.cust\_city='Diemen'

union

select t2.prod\_id from sales t2,customers t3

where t2.cust\_id = t3.cust\_id

and t3.cust\_city='Cochem';



可以看到，并没有发生连接因式分解，但是我们讲道理，因为sort unique是最后一步做的，所以连接因式分解应该没问题，所以我们这里加入hint：

select /\*+FACTORIZE\_JOIN(@"SET$1"("T2"@"SEL$1" "T2"@"SEL$2"))\*/ t2.prod\_id from sales t2,customers t3

where t2.cust\_id = t3.cust\_id

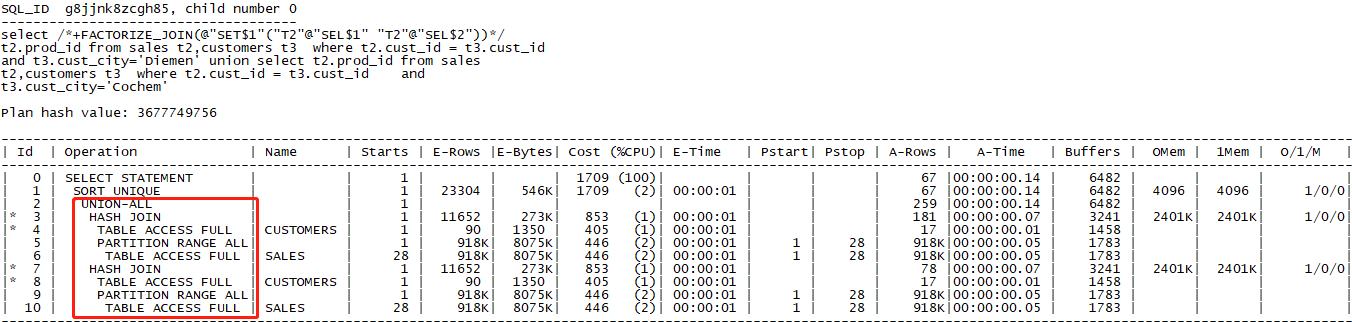
and t3.cust\_city='Diemen'

union

select t2.prod\_id from sales t2,customers t3

where t2.cust\_id = t3.cust\_id

and t3.cust\_city='Cochem';



我们看到，hint并没有起作用。看来对union是不起作用的，连接因式分解确实是只针对union all。这一点，我通过把下面的查询中的union all变为union进行实验得到了肯定，因为empno上有主键，但是执行计划中依然会有sort unique这一步，不知道之后的版本优化器在这一点上会不会做的更智能。

下面我们再来看一个案例：

select t2.empno from emp t2,dept t3

where t2.deptno=t3.deptno

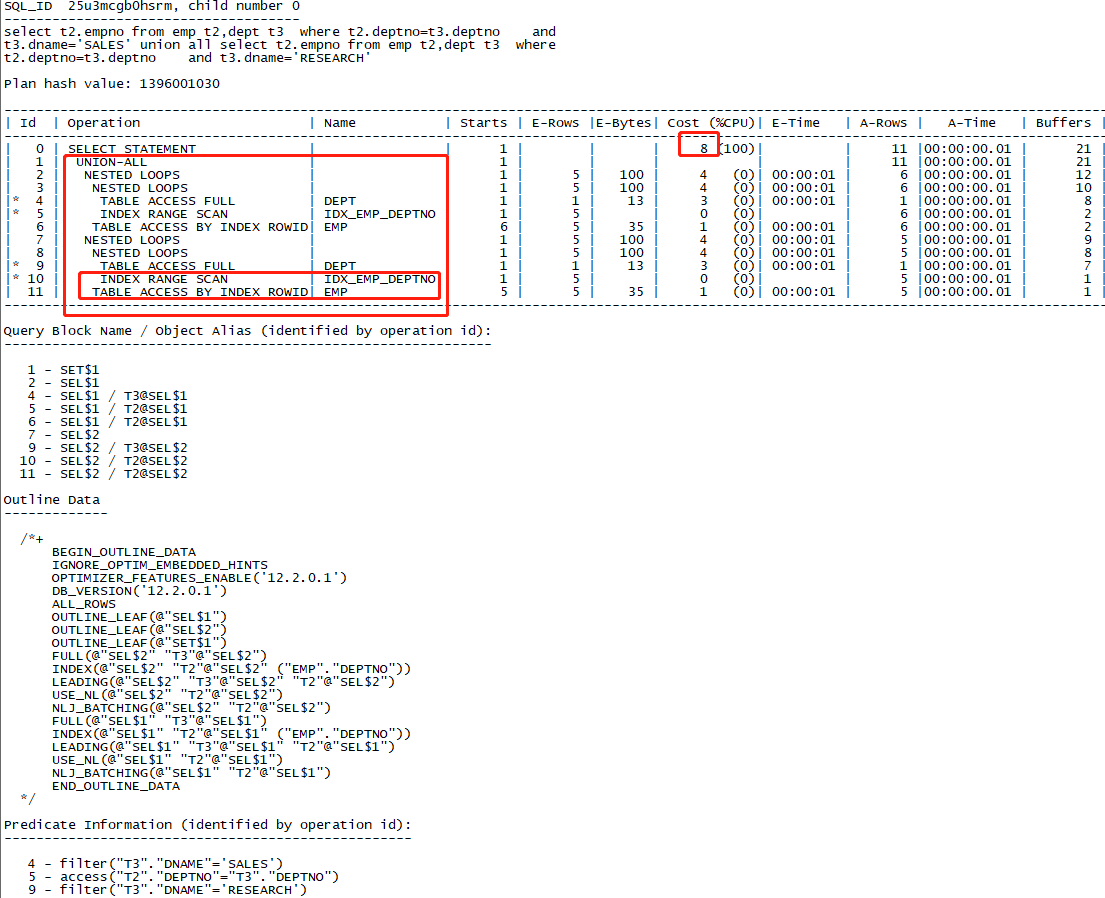
and t3.dname='SALES'

union all

select t2.empno from emp t2,dept t3

where t2.deptno=t3.deptno

and t3.dname='RESEARCH';



这里很明显是可以做连接因式分解的。是不是cbo在做连接因式分解时考虑了代价呢？

select /\*+FACTORIZE\_JOIN(@"SET$1"("T2"@"SEL$1" "T2"@"SEL$2"))\*/ t2.empno from emp t2,dept t3

where t2.deptno=t3.deptno

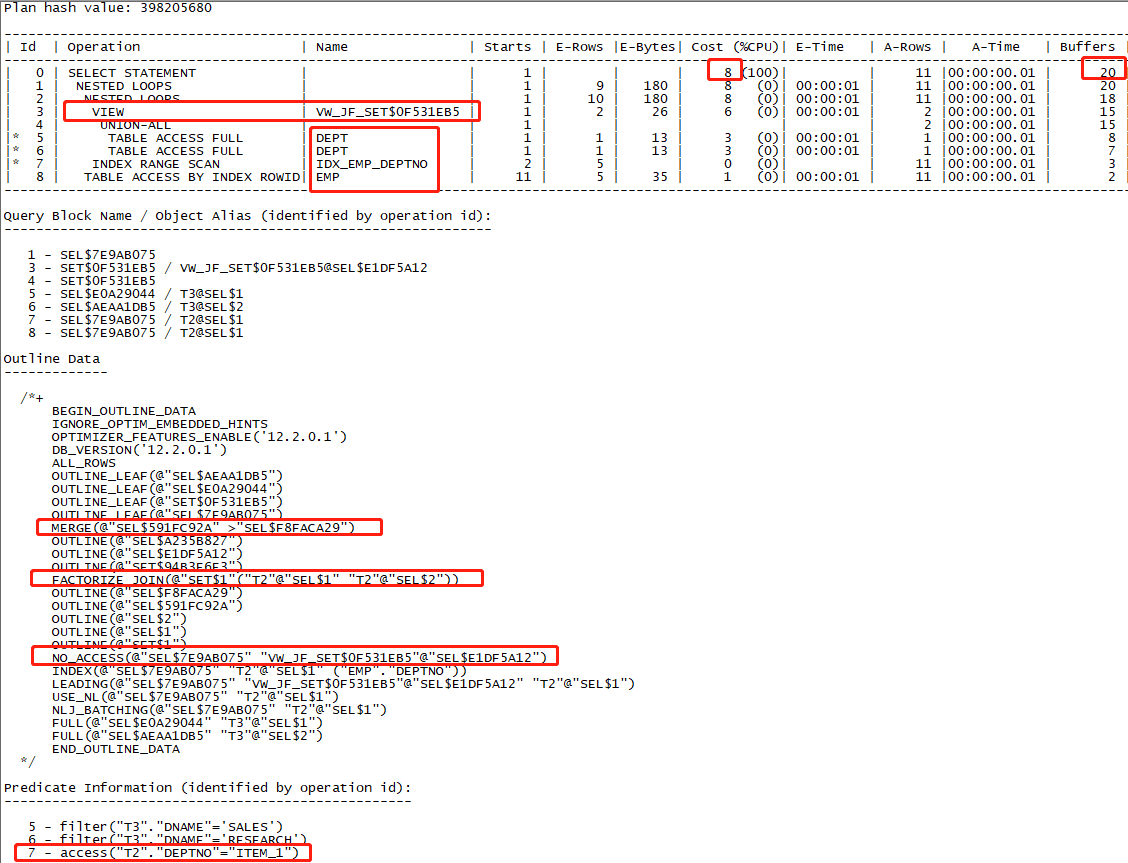
and t3.dname='SALES'

union all

select t2.empno from emp t2,dept t3

where t2.deptno=t3.deptno

and t3.dname='RESEARCH';



可以看到cost确实没有比做转换之前更小，所以，连接因式分解是考虑代价，但是我们看到buffers还是比原来小了，应该效率还是有所提升的。

### 带UNION ALL的视图

我们就把之前的sql做成视图：

create or replace view view\_jf as

select t2.prod\_id from sales t2,customers t3

where t2.cust\_id = t3.cust\_id

and t3.cust\_city='Diemen'

union all

select t2.prod\_id from sales t2,customers t3

where t2.cust\_id = t3.cust\_id

and t3.cust\_city='Cochem';

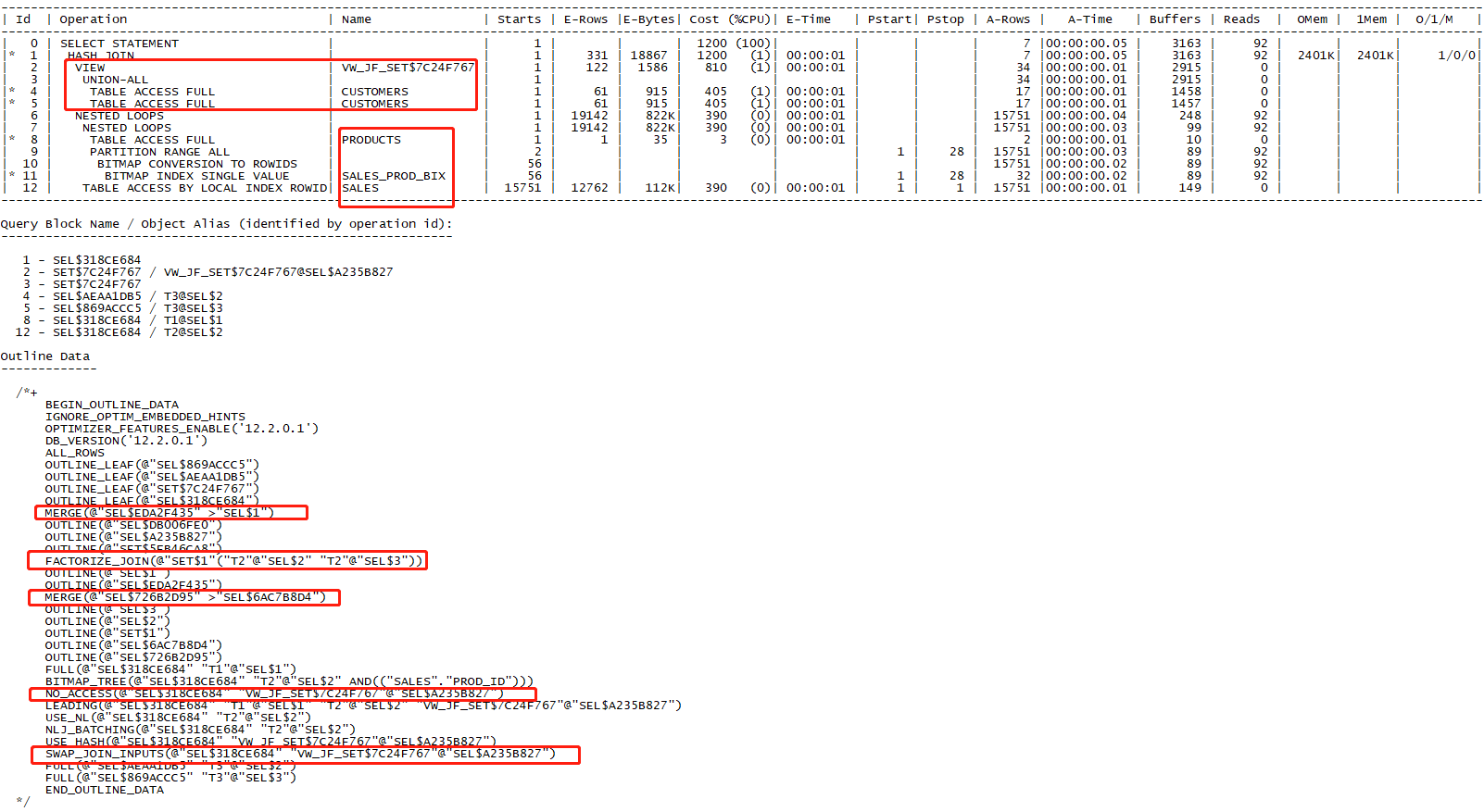
然后我们做如下查询：

select t1.prod\_id,t1.prod\_name

from products t1,view\_jf

where t1.prod\_id=view\_jf.prod\_id

and t1.prod\_list\_price > 1000;



从上面的执行计划中，我们找到俩个merge，我们来看一下整个查询转换的流程：

这里我们直接将之前转换好的结果带入到sql中：

select t1.prod\_id, t1.prod\_name

from products t1,

(select t2.prod\_id

from sales t2,

(select t3.cust\_id

from customers t3

where t3.cust\_city = 'Diemen'

union all

select t3.cust\_id

from customers t3

where t3.cust\_city = 'Cochem') VW\_JF\_SET$0F531EB5

where t2.cust\_id = VW\_JF\_SET$0F531EB5.cust\_id) view\_jf

where t1.prod\_id = view\_jf.prod\_id

and t1.prod\_list\_price > 1000;

我们可以看到通过这一步转换，view\_jf本身已经被转换成了我们在2.2.1中所说的简单视图，所以这里就可以做简单视图合并，这也就是原sql执行计划中outline部分的第一个merge的由来：

select t1.prod\_id, t1.prod\_name

from products t1,sales t2,

(select t3.cust\_id

from customers t3

where t3.cust\_city = 'Diemen'

union all

select t3.cust\_id

from customers t3

where t3.cust\_city = 'Cochem') VW\_JF\_SET$0F531EB5

where t1.prod\_id = t2.prod\_id

and t1.prod\_list\_price > 1000

and t2.cust\_id = VW\_JF\_SET$0F531EB5.cust\_id;



现在我们可以看到，与之前的执行计划一模一样。这里之所以出现swap\_join\_input，是因为leading的顺序本该是t1连t2，然后再连VW\_JF\_SET$0F531EB5，但是t1连完t2，发现VW\_JF\_SET$0F531EB5的结果集小，所以CBO就“临时”决定把VW\_JF\_SET$0F531EB5作为hash join的驱动结果集。

通过之前的实验和分析，我们可以看出，之所以最终cbo可以选择products与sales先做连接，之后再与customers做hash join，是因为cbo对原本不能进行视图合并的带union all的视图做了连接因式分解，而使得view\_jf可以有一部分进行视图合并，从而给了优化器更多的可选择的执行路径。

### 连接因式分解总结

1. 连接因式分解是考虑代价的；
2. 连接因式分解只对UNION ALL可用，对UNION不可用
3. 连接因式分解的hint是：FACTORIZE\_JOIN(@"SET$1"("T2"@"SEL$2" "T2"@"SEL$3"))
4. 连接因式分解让带union all的视图进行部分视图合并成为了可能。
5. 连接因式分解是11gR2之后引入的。

## 表扩展（Table Expansion）

### 表扩展的介绍

表扩展是对分区表的一种优化手段，是当分区表有分区索引失效使，依然能通过将索引没有失效的的分区与索引失效的分区进行union all，来实现索引为失效的分区继续走索引。这项技术使11gR2之后才引入的。

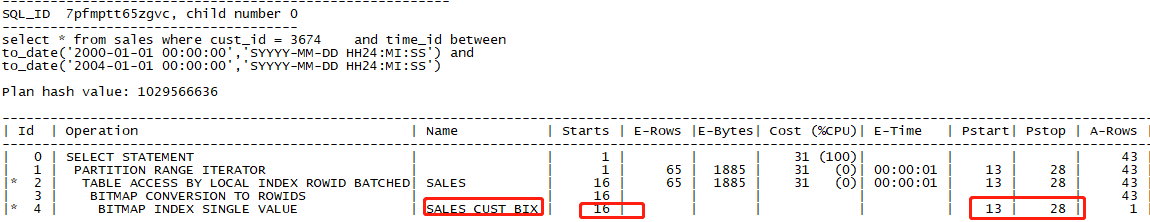
现在我们来看一条sql：

select \* from sales where cust\_id = 3674

and time\_id between

to\_date('2000-01-01 00:00:00','SYYYY-MM-DD HH24:MI:SS') and

to\_date('2004-01-01 00:00:00','SYYYY-MM-DD HH24:MI:SS');



我们看到这条sql走了13分区到28分区。这里是因为我们指定了分区列time\_id，所以oracle采用了分区修建（Partition Pruning）的技术。

假如这时候我们让cust\_id列与time\_id列上的索引在分区SALES\_Q3\_2003上的状态置为unusable，那么sql又会走怎样的执行计划呢：

SQL> alter index sales\_cust\_bix modify partition sales\_q4\_2003 unusable;

Index altered.

SQL> alter index sales\_time\_bix modify partition sales\_q4\_2003 unusable;

Index altered.

然后再执行sql：



现在我们看到，sql并没有因为部分分区索引失效，而没有继续使用分区索引，而是将sql进行了如下转换：

select \* from

(select \* from sales where cust\_id = 3674

and time\_id between

to\_date('2000-01-01 00:00:00','SYYYY-MM-DD HH24:MI:SS') and

to\_date('2003-09-30 00:00:00','SYYYY-MM-DD HH24:MI:SS')

union all

select \* from sales where cust\_id = 3674

and time\_id between

to\_date('2003-10-01 00:00:00','SYYYY-MM-DD HH24:MI:SS') and

to\_date('2004-01-01 00:00:00','SYYYY-MM-DD HH24:MI:SS')) VW\_TE\_2;

这样sql就又可以用到索引没有失效的分区。

我们从执行计划中可以看到VW\_TE\_2的TE就是Table Expansion，同时可以再OUTLINE中看到EXPAND\_TABLE(@"SEL$1" "SALES"@"SEL$1")。

那么这项优化技术有什么意义呢？这里跟崔华老师学到了。

比如说一张按时间分区的分区表，过去的时间分区可能更多的是用于查询，而当前的时间分区会有大量dml操作时，为了加快dml的插入，会把当前时间分区上的索引暂时置为unsable，这样就少了对索引的维护从而加快了dml操作。

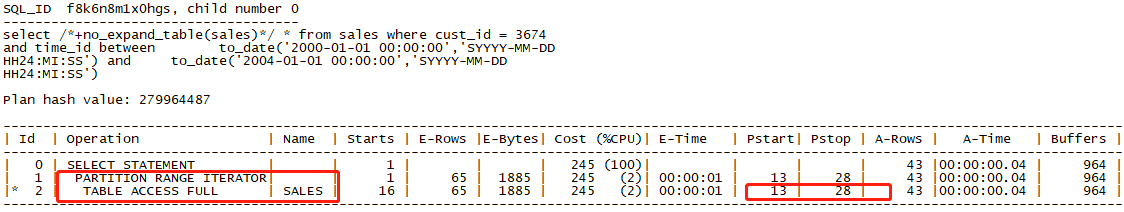
当然我们也可以使用hint让表不扩展：

select /\*+no\_expand\_table(sales)\*/ \* from sales where cust\_id = 3674

and time\_id between

to\_date('2000-01-01 00:00:00','SYYYY-MM-DD HH24:MI:SS') and

to\_date('2004-01-01 00:00:00','SYYYY-MM-DD HH24:MI:SS');



我们看到由于有分区上的索引失效，所以只能走全表扫描了，同时看到partition pruning并不受影响。

### 表扩展+连接因式分解

这里我们在看崔化老师书中的一个例子：

我们先让如下索引失效：

SQL> alter index sales\_channel\_bix modify partition SALES\_Q4\_2003 unusable;

Index altered.

SQL> alter index sales\_prod\_bix modify partition SALES\_Q4\_2003 unusable;

Index altered.

SQL> alter index sales\_cust\_bix rebuild partition SALES\_Q4\_2003;

Index altered.

这里展示一下，rebuild是这么rebuild的。

然后执行如下sql：

select t2.cust\_city, sum(t1.amount\_sold) amount\_sold\_total

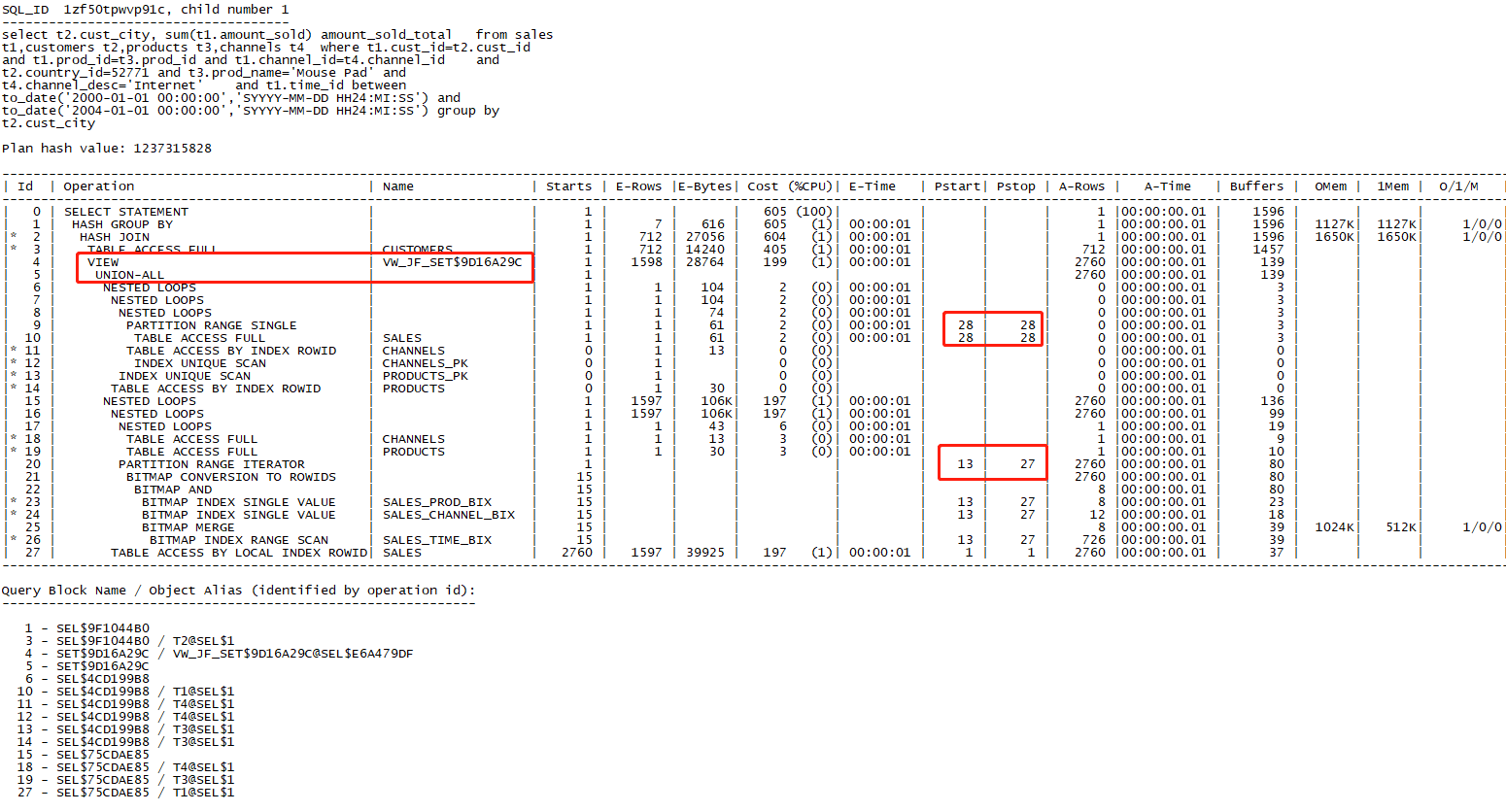
from sales t1,customers t2,products t3,channels t4

where t1.cust\_id=t2.cust\_id and t1.prod\_id=t3.prod\_id and t1.channel\_id=t4.channel\_id

and t2.country\_id=52771 and t3.prod\_name='Mouse Pad' and t4.channel\_desc='Internet'

and t1.time\_id between to\_date('2000-01-01 00:00:00','SYYYY-MM-DD HH24:MI:SS') and

to\_date('2004-01-01 00:00:00','SYYYY-MM-DD HH24:MI:SS') group by t2.cust\_city;





我们看到了因式分解和表扩展。

我们现在看下sql转换的过程：

首先由于分区索引失效，sql先要变成如下：

select view\_te.cust\_city,sum(amount\_sold) amount\_sold\_total from

(select t2.cust\_city,t1.amount\_sold

from sales t1,customers t2,products t3,channels t4

where t1.cust\_id=t2.cust\_id and t1.prod\_id=t3.prod\_id and t1.channel\_id=t4.channel\_id

and t2.country\_id=52771 and t3.prod\_name='Mouse Pad' and t4.channel\_desc='Internet'

and t1.time\_id >=to\_date('2000-01-01 00:00:00','SYYYY-MM-DD HH24:MI:SS') and

t1.time\_id < to\_date('2003-10-01 00:00:00','SYYYY-MM-DD HH24:MI:SS')

union all

select t2.cust\_city,t1.amount\_sold

from sales t1,customers t2,products t3,channels t4

where t1.cust\_id=t2.cust\_id and t1.prod\_id=t3.prod\_id and t1.channel\_id=t4.channel\_id

and t2.country\_id=52771 and t3.prod\_name='Mouse Pad' and t4.channel\_desc='Internet'

and t1.time\_id >= to\_date('2003-10-01 00:00:00','SYYYY-MM-DD HH24:MI:SS')

and t1.time\_id <= to\_date('2004-01-01 00:00:00','SYYYY-MM-DD HH24:MI:SS')) view\_te group by view\_te.cust\_city;

然后我们可以看到，又公因式customers,products,channels都可以提出来：

select t2.cust\_city,sum(VW\_JF\_SET$9D16A29C.amount\_sold) amount\_sold\_total from customers t2,channels t4,products t3,

(select t1.cust\_id,t1.channel\_id,t1.amount\_sold,t1.prod\_id

from sales t1

where t1.time\_id >=to\_date('2000-01-01 00:00:00','SYYYY-MM-DD HH24:MI:SS')

and t1.time\_id < to\_date('2003-10-01 00:00:00','SYYYY-MM-DD HH24:MI:SS')

union all

select t1.cust\_id,t1.channel\_id ,t1.amount\_sold,t1.prod\_id

from sales t1

where t1.time\_id >= to\_date('2003-10-01 00:00:00','SYYYY-MM-DD HH24:MI:SS')

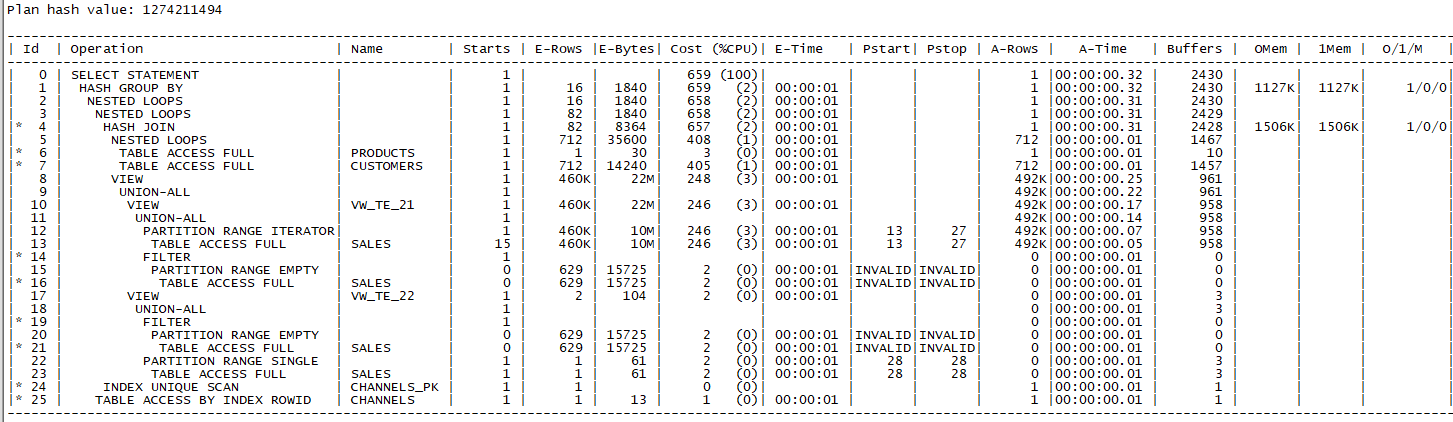
and t1.time\_id <= to\_date('2004-01-01 00:00:00','SYYYY-MM-DD HH24:MI:SS')) VW\_JF\_SET$9D16A29C

where VW\_JF\_SET$9D16A29C.cust\_id=t2.cust\_id and VW\_JF\_SET$9D16A29C.channel\_id=t4.channel\_id and VW\_JF\_SET$9D16A29C.prod\_id=t3.prod\_id

and t2.country\_id=52771 and t4.channel\_desc='Internet' and t3.prod\_name='Mouse Pad'

group by t2.cust\_city;

从结果集上，这样的转换方式也是得到了验证的。但是我们发现执行计划中cbo只提出了customers，这是因为连接因式分解是考虑代价的。如果按我们转化之后的执行计划如下：



我们看到cost是659，而原来的执行计划是605，所以这也同时证明了连接因式分解是考虑代价。

### 表扩展总结

一、11gR2之后才引入的；

二、表扩展是用union all优化分区表分区索引失效的情况；

三、为了减少频繁dml的时间，有时故意让当前分区索引暂时失效；

四、多表连接时发生表扩展，就也可能会同时发生连接因式分解，因为除了分区条件，全是相同部分。

五、表扩展的hint：/\*+expand\_table()\*/，/\*+no\_expand\_table()\*/

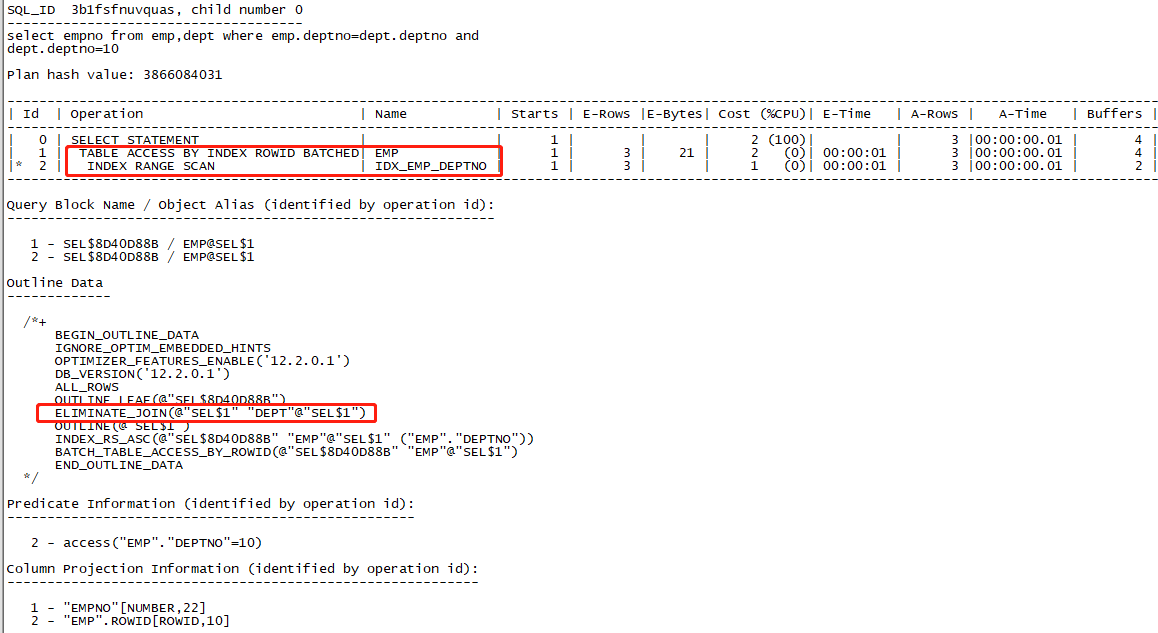
六、表扩展也是考虑成本的

## 表移除（Table Elimination）

表移除是只将目标sql中对执行结果没有影响的表移除掉，减少一次连接，达到优化效果。

我们直接看个sql：

select empno from emp,dept where emp.deptno=dept.deptno and dept.deptno=10;



从执行计划中我们可以看到，并没有出现dept，同时outline中出现ELIMINATE\_JOIN(@"SEL$1" "DEPT"@"SEL$1")，说明dept被移除了。

能做到这一点，必须连接列是主表上的外键，且关联到被移除表的连接列。

# 查询转换总结

这里主要做一个查询转换的归纳分类：

1. 我们要明确，子查询展开是子查询在where子句中，而视图合并是子查询在from子句中；
2. 除了子查询展开、简单视图合并、外连接视图合并不考虑成本，其他都要考虑成本；
3. 视图合并、连接谓词推入、连接因式分解说白了都是对视图的连接的一种处理方式，视图合并是由内到外；连接因式分解是又内到外走一般；连接谓词推入是由外到内；
4. 子查询不展开、连接谓词推入，这俩兄弟的本质都是把执行计划定死是嵌套循环，且视图或者说子查询被定死为被驱动表，所以执行计划中要格外关注filter和push pred；
5. 表扩展是为了弥补有意而为的让索引失效的优化方案，只针对分区表；
6. 发生表移除和连接因式分解的时候，需要认真审查一下sql的写法，确认这俩种情况发生的合理性；
7. 连接因式分解对union无效
8. 反连接、<>不走索引！

然后我们做个hint总结：

single-row subquery: push\_subq

subquery unnesting: unnest no\_unnest

view merging: merge no\_merge/no\_access

OR expansion: or\_expand/use\_concat no\_expand

join predicate pushdown: push\_pred no\_push\_pred

join factorization: factorize join

table expansion: expand\_table no\_expand\_table

table elimination: eliminate\_join

然后对对应的视图名称做个总结：

子查询展开：NSO

OR扩展：ORE

连接因式分解：JF

表扩展：TE

最后补充一点：

SQL> SELECT ksppinm, ksppstvl, ksppdesc FROM x$ksppi x, x$ksppcv y WHERE x.indx = y.indx AND ksppinm='\_optimizer\_cost\_based\_transformation';

KSPPINM KSPPSTVL KSPPDESC

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

\_optimizer\_cost\_based\_transformation LINEAR enables cost-based query transformation

这个是控制查询转换是否基于成本的隐藏参数，如果把这个参数置为off，那就代表CBO会认为查询转换后效率一定比之前高。