**第二周学习计划总结**

**表的连接方式及hint**

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 | 2018-04-27 | 初始版本 |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **编制** | 燕鑫 | （签字） | 日期 | 2018-04-27 |
| **校对** |  | （签字） | 日期 |  |
| **审核** |  | （签字） | 日期 |  |
| **批准** |  | （签字） | 日期 |  |

目录

[1. 与表访问相关的执行计划 - 5 -](#_Toc512949348)

[1.1 TABLE ACCESS FULL（多块读） - 5 -](#_Toc512949349)

[1.2 TABLE ACCESS BY USER ROWID（单块读） - 5 -](#_Toc512949350)

[1.3 TABLE ACCESS BY ROWID RANGE(多块读) - 7 -](#_Toc512949351)

[1.4 TABLE ACCESS BY INDEX ROWID/INDEX ROWID BATCHED（单块读） - 8 -](#_Toc512949352)

[2. 与B树索引相关的执行计划 - 10 -](#_Toc512949353)

[2.1 Clustering factor - 10 -](#_Toc512949354)

[2.2 INDEX UNIQUE SCAN（单块读） - 13 -](#_Toc512949355)

[2.3 INDEX RANGE SCAN/INDEX RANGE SCAN DESCENDING（单块读） - 14 -](#_Toc512949356)

[2.4 INDEX SKIP SCAN(单块读) - 17 -](#_Toc512949357)

[2.5 INDEX FULL SCAN(单块读) - 19 -](#_Toc512949358)

[2.6 INDEX FULL SCAN(MIN/MAX)(单块读) - 20 -](#_Toc512949359)

[2.7 INDEX FAST FULL SCAN（多块读） - 22 -](#_Toc512949360)

[2.8 本章HINT及等待事件的总结： - 23 -](#_Toc512949361)

[2.8.1 Hint - 23 -](#_Toc512949362)

[2.8.2 等待事件 - 24 -](#_Toc512949363)

[3. 其他常见的执行计划 - 24 -](#_Toc512949364)

[3.1 AND-EQUAL - 24 -](#_Toc512949365)

[3.2 INDEX JOIN - 25 -](#_Toc512949366)

[3.3 VIEW - 26 -](#_Toc512949367)

[3.4 FILTER - 27 -](#_Toc512949368)

[3.4.1 FILTER的概述 - 27 -](#_Toc512949369)

[3.4.2 FILTER的产生原因 - 29 -](#_Toc512949370)

[3.4.3 FILTER的消除 - 29 -](#_Toc512949371)

[3.5 UNION ALL/UNION - 30 -](#_Toc512949372)

[3.6 CONCAT - 30 -](#_Toc512949373)

[3.7 本章HINT总结 - 31 -](#_Toc512949374)

[4. 表连接方式 - 32 -](#_Toc512949375)

[4.1 NESTED LOOPS - 32 -](#_Toc512949376)

[4.1.1 显式的嵌套循环 - 32 -](#_Toc512949377)

[4.1.2 天然的嵌套循环 - 34 -](#_Toc512949378)

[4.2 HASH JOIN（消耗PGA） - 35 -](#_Toc512949379)

[4.3 SORT MERGE JOIN（消耗PGA） - 37 -](#_Toc512949380)

[4.4 CARTESIAN JOIN - 39 -](#_Toc512949381)

[4.5 本章HINT总结 - 39 -](#_Toc512949382)

[5. 特殊的连接 - 40 -](#_Toc512949383)

[5.1 反连接（Anti Join） - 40 -](#_Toc512949384)

[5.2 半连接（Semi join） - 44 -](#_Toc512949385)

[5.3 星型连接（Star join） - 49 -](#_Toc512949386)

# 与表访问相关的执行计划

## TABLE ACCESS FULL（多块读）

全表扫描，首先最主要的一点就是，全表扫描是多块读！

当然，多块读是读的物理上连续的块，也就是说全表扫描不能跨区读，因为区与区之间的块不一定连续。

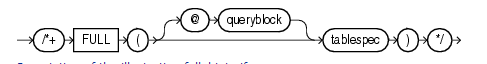
一般的操作系统一次io最多只支持读取或写入1M，也就是说数据块为8k的话，一次最多读取128个块。

如果表中的块有部分缓存在buffer cache中，扫描到有已经被缓存的块的区，就会发生IO中断，如果有大量的IO中断，全表扫描的性能就会严重下降，导致每次IO不能扫描1M的块。

一般来说所求结果集超过全表的百分之5就可以走全表扫描了。也就是说这时候多块读的效率就已经超过单块读了。

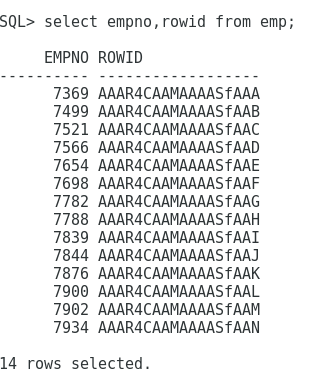
还有一点就是，全表扫描会扫描到该表的高水位线。也就是说连碎片都扫。

全表扫描的hint是:



## TABLE ACCESS BY USER ROWID（单块读）

先直观的看下什么是个rowid：

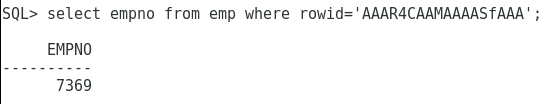


其实就是每一行的唯一标识，这个唯一是表内唯一哦！从[oracle](https://www.baidu.com/s?wd=oracle&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3nAfduHubmHubPWfzmH630ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPWf1nHD3njc) 8开始rowid变成了extend rowid，由data\_object\_id# rfile# block# row#组成，占用10个bytes的空间， 32bit的 data\_object\_id#,10 bit 的 rfile#,22bit 的 block#,16 bit 的 row#.由于rowid的组成从file#变成了rfile#，所以数据文件数的限制也从整个库不能超过1023个变成了每个表空间不能超过1023个 数据文件。官文里面是如下这样说的的：

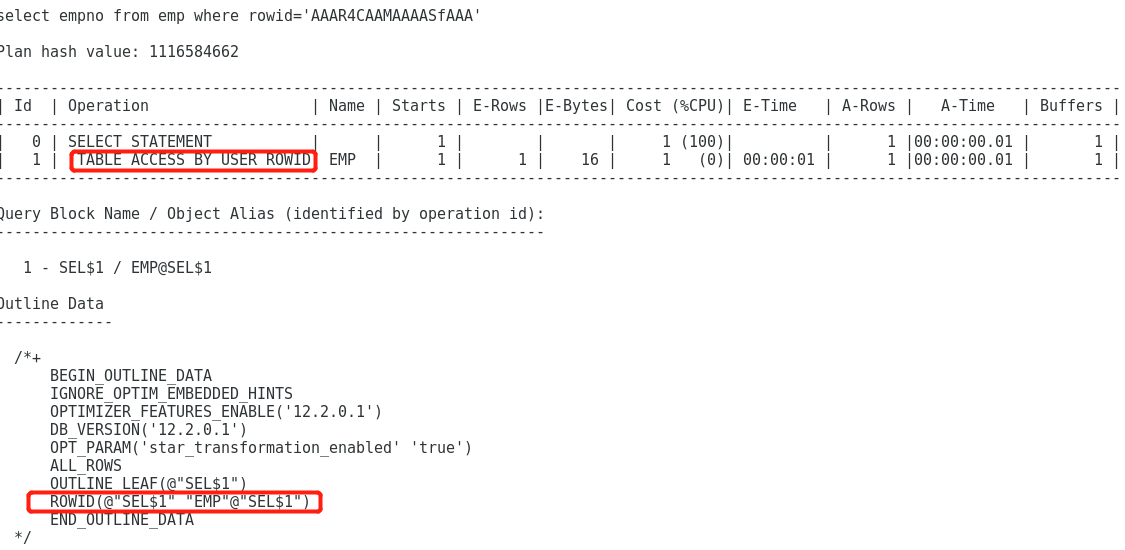
Rowid values have several important uses:

* They are the fastest way to access a single row.
* They can show you how the rows in a table are stored.
* They are unique identifiers for rows in a table.

上面的查询之所以能直接查出rowid，是因为每个表都有rowid这个伪列，查出这个列是为了做下面这个查询：



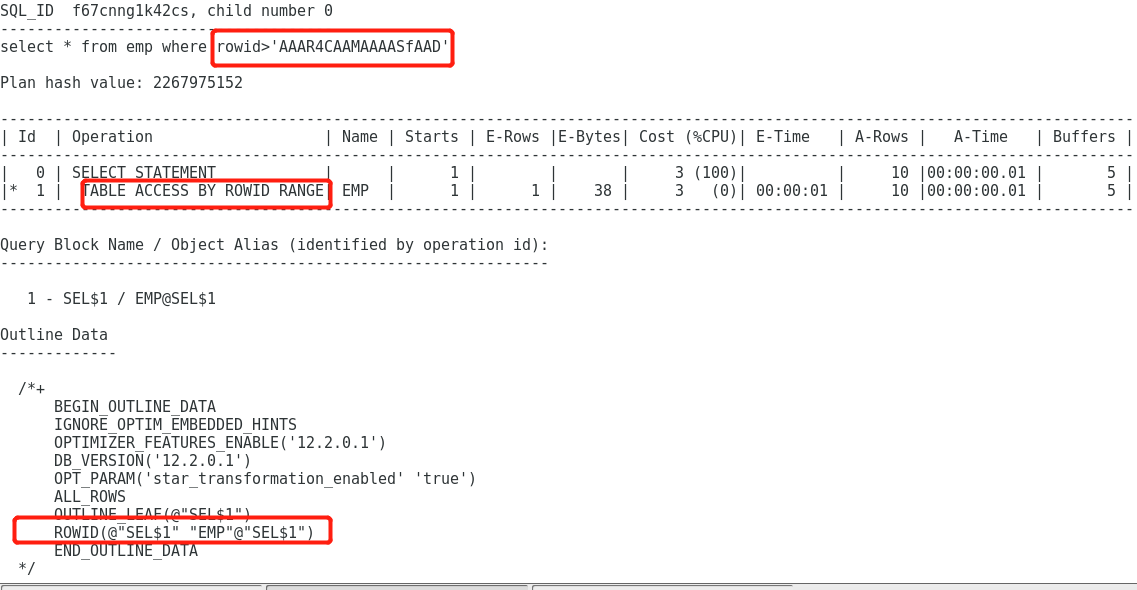
执行计划如下：



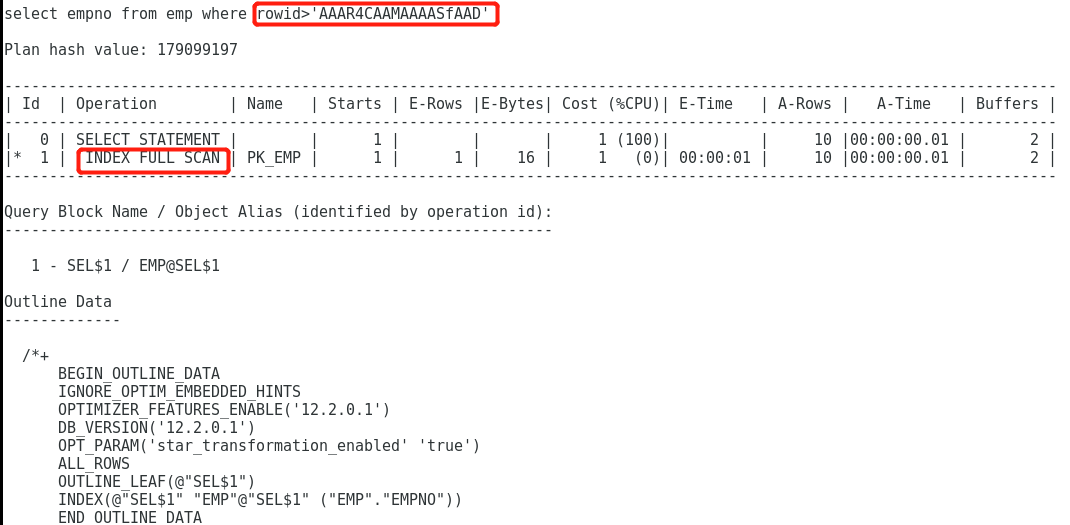
你想看到这个table access by user rowid就只有通过rowid直接（向示例查询那样）查询，没别的办法了。Hint就是上面红圈标着的，这个你要不这样查，用这个hint也没用。

这么简单的一个东西，而且又不常用，为什么要再这里做了一个比较详细的解释呢？因为这个不常用是你写sql的时候不常用，对于Oracle来说，可就非常常用了，这个就是回表的基本原理。我们接着往下看就知道了。

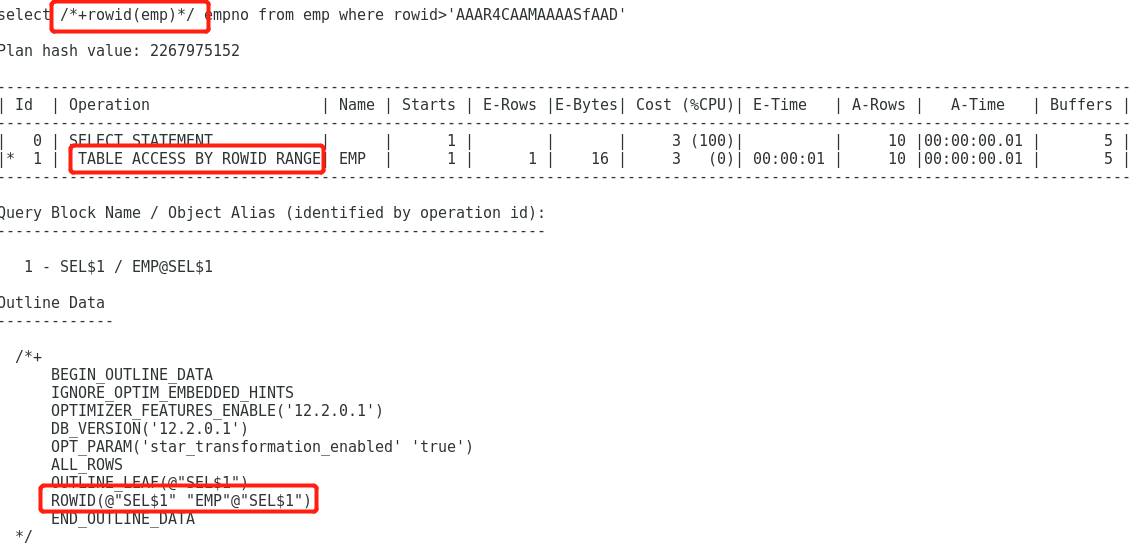
## TABLE ACCESS BY ROWID RANGE(多块读)



这里做个小实验啊，就是咋能用上rowid这个hint，虽然很蠢，没啥意义：



我们看到，虽然我们用rowid范围查询，但是并没有走rowid，而是走了主键，这个index full scan后面还会详细说，这里之所以这样走道理很简单，因为你就哪怕只找一行数据，对于一张14行数据的表，你直接在主键里面全扫描单块读去找这一个值，也比你用rowid去表的块里范围找快，所以oracle选择这种方式没问题。但这时候我们就可以试一下rowid的这个hint了：



没啥好说的，就是随便玩儿一玩儿。

## TABLE ACCESS BY INDEX ROWID/INDEX ROWID BATCHED（单块读）

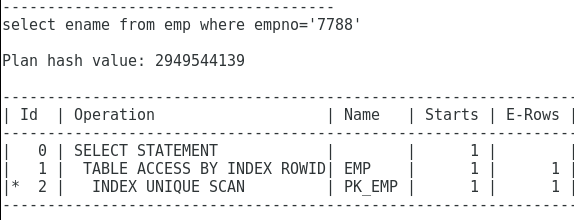
这个千万注意，单块读！所以回表多了，性能消耗会非常大！这个table access by index rowed batched是后面为了优化table access by index rowid的一个新特性。他们俩的区别就是：batched会将一个块里的rowid一次访问完；没batched，就是如果俩个rowid在一个块里，那么就会访问这个块俩次。

这个table access by index rowid就只有一个名字:回表！

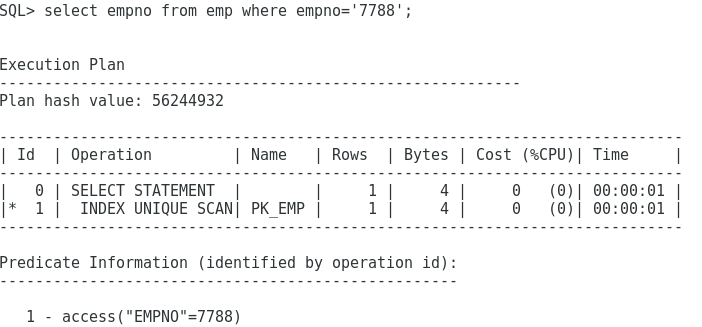
说下，回表的情况有哪些，或者说为什么要回表：简单的说就是我想要的数据索引块提供不全！

具体的情况有：走的索引中没有包含所有的查询列，或者没有包含所有的过滤条件，或者没有包含排序列，总之就是只要出现索引包含的列以外的列就要回表。

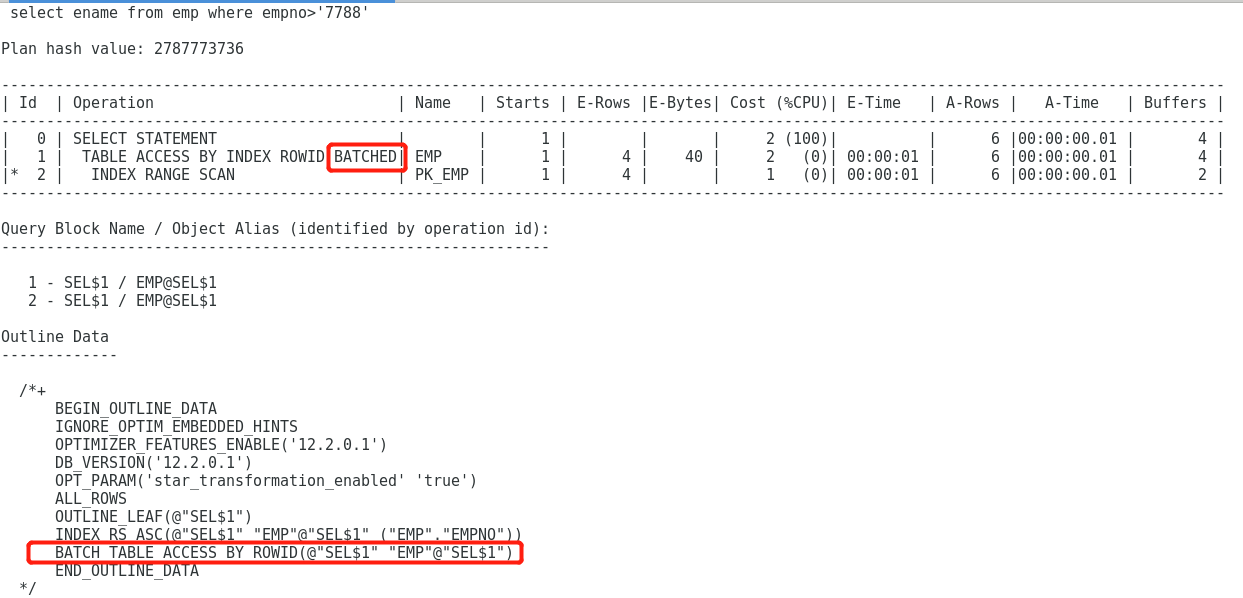
先上个例子看一眼：



看到是table accesss by index rowid，首先产生这一步的原因是因为要查的列ename没有包含在主键PK\_EMP中，我们看看如果把ename换成empno：



发现没有回表了，也佐证了我们之前关于回表产生的说法。第二，之所以刚才没有batched是因为走了唯一索引扫描，本来就只能找到一条，还批量返回什么。现在我们再来看个例子：



看到了那个outline data了么？这个就可以算是这个执行计划的hint吧。这是为什么，其实就是因为oracle这次回表的时候拿了一把rowid，为了加快回表，所以就把在一个块里的rowid的行的数据批量取回来，就不用同一个块访问多次了。这就是table access by index rowed batched。

这个是12c的新特性，由参数\_optimizer\_batch\_table\_access\_by\_rowid控制，默认是打开着的。

# 与B树索引相关的执行计划

## Clustering factor

这里为了后面更好的理解索引相关的执行计划，先介绍一个概念，就是聚簇因子。

注意聚簇因子是判断索引回表需要消耗的物理IO次数的，也就是说聚簇因子对已经缓存在内存中的数据块没影响。

select owner,index\_name,clustering\_factor

from dba\_indexes where owner = ' ' and index\_name=' ';

可以查出索引的聚簇因子。

select count(distinct dbms\_rowid.rowid\_block\_number(rowid)) blocks

from TNAME;

可以查出表的块数。

索引叶子块中是有序的存着索引的键值和对应的rowid，然后我们看看聚簇因子的算法：

首先，从第一个键值开始，跟紧接这的键值(这里就是第二个)的rowid核对，这俩rowid是不是在一个块里，如果是，那就clustering factor保持原值不变(这里就是还是1),如果不在一个块，就自加1，然后在用第二个键值对应的rowid去和第三个键值的rowid核对，一直核对到最后。

所以，很显然，如果聚簇因子最终是1，那说明所有的所有行的数据都在一个块里，那要是table access by index rowid batched一把就读完了。如果说表的行数14，结果聚簇因子也是14，倒不是说明数据分布在14个块中(有可能就在俩个块中，算术题)，只能说明该表的行数据存放的很离散，也就是说这时候回表就要产生14次IO，咋回都一样，batched还是不batched都一样。

所以我们得出一个结论：

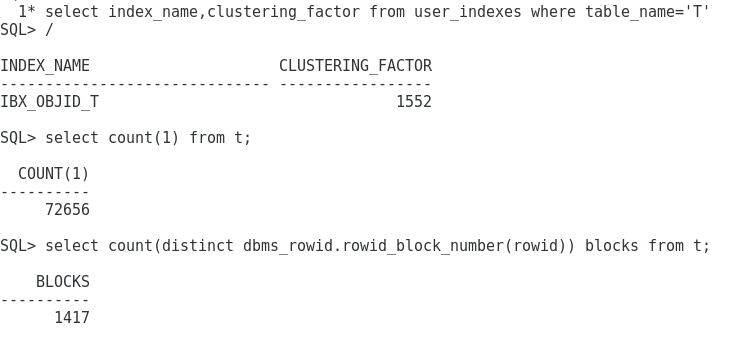
如果聚簇因子接近表的块数，那说明表的数据比较集中，而且索引列的数据存放比较有序，这样在进行index range scan或者index full scan的时候，回表只需读少量的块；

如果聚簇因子接近表的行数，那回表就要多更多的块。

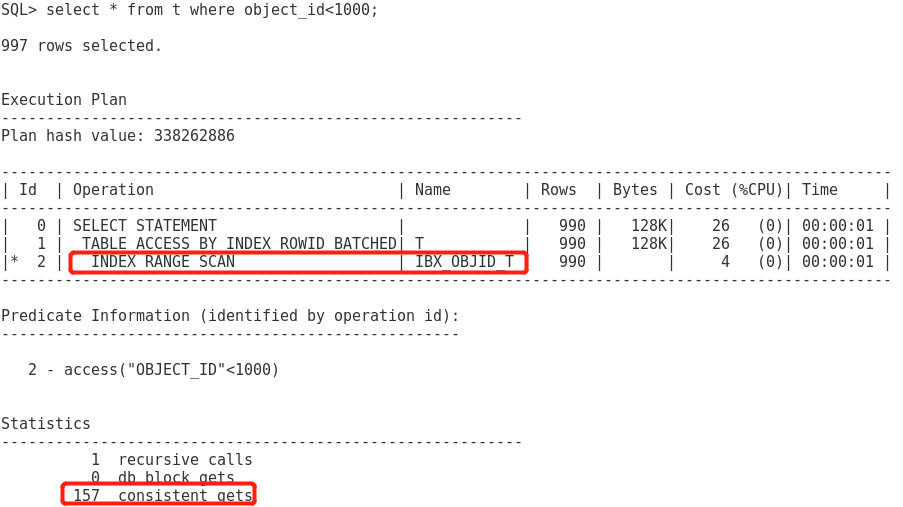
聚簇因子只会影响index range scan和index full scan，因为这俩种索引扫描方式会有大量回表。

而index unique scan和index fast full scan就没影响，因为前一个是只取一行数据，后一个是根本不回表。

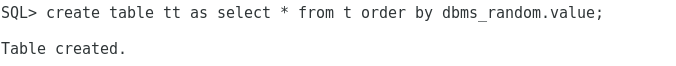
看个例子：

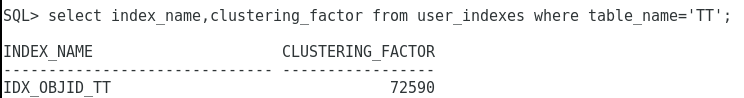


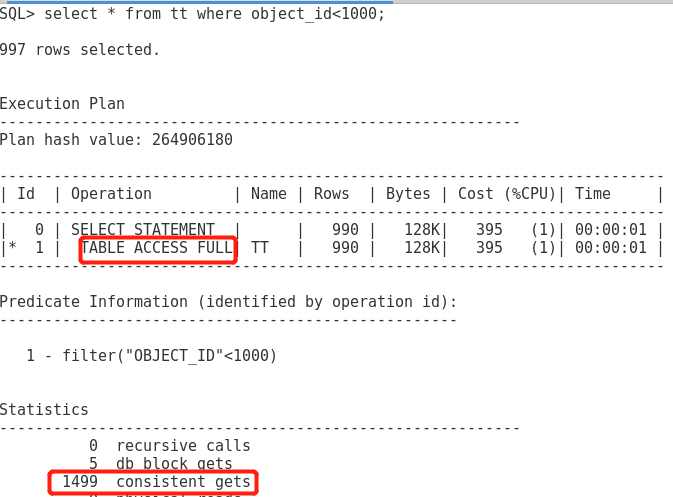
可以看到，t表的object\_id列上的索引聚簇因子比较接近表块数，我们来看一个查询：



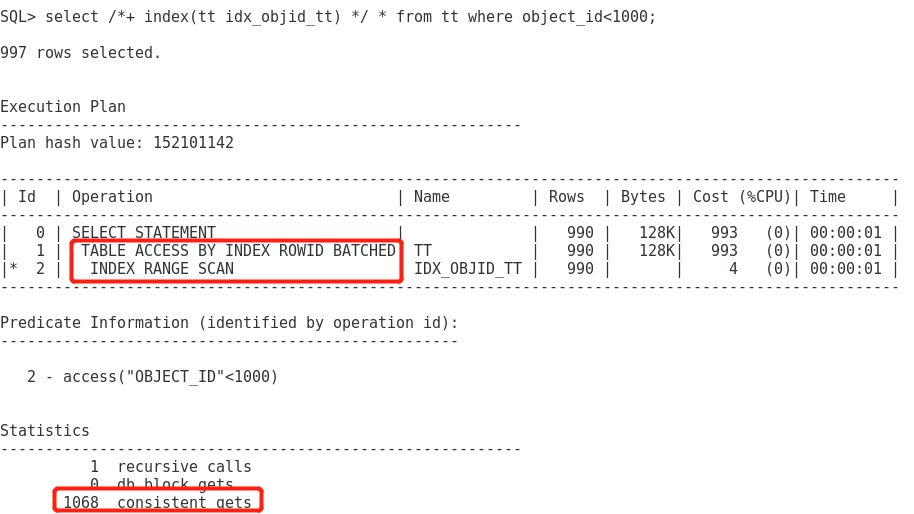
我们来对比一下聚簇因子比较大的情况：







看这里我们发现，居然在有索引的情况下，走了全表扫描，为什么？因为数据太离散，oracle认为此时单块读效率这时候不如全表扫描多块读效率高，我们来验证一下：



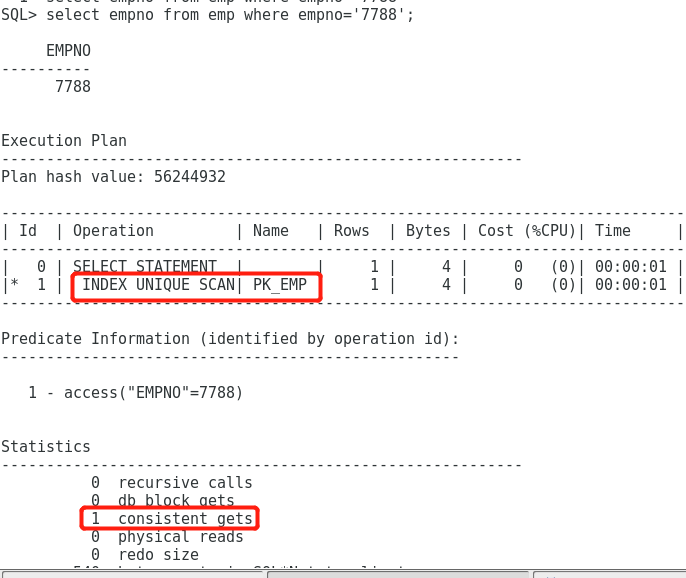
看到，这里有1068个逻辑读，但是oracle计算出来的cost要比全表扫描计算出来的cost大，所以选择走了全表扫描。我们现在主要比对这个跟之前t表查询的效率。发现之前只有157个逻辑读，现在有1068个。可以看到聚簇因子对于回表的影响非常大。

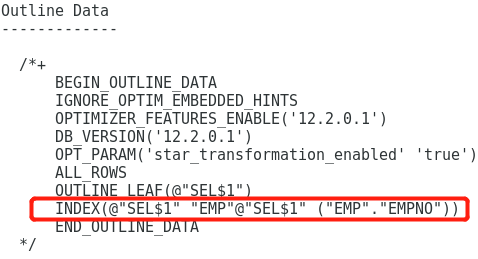
但是，调整聚簇因子是徒劳的，因为聚簇因子是与索引一一对应的，就算我们建表的时候对某一索引列排序，也很难做到同时满足其他的索引列。所以我们唯一解决问题的途径就是：减小甚至消除回表！当你数据量返回非常少的时候，聚簇因子对回表的影响就很微弱了。

## INDEX UNIQUE SCAN（单块读）

这个是最喜欢看到的，因为发生这个的前提是：走唯一索引或主键，并且等值查询，也就是只返回一个值。

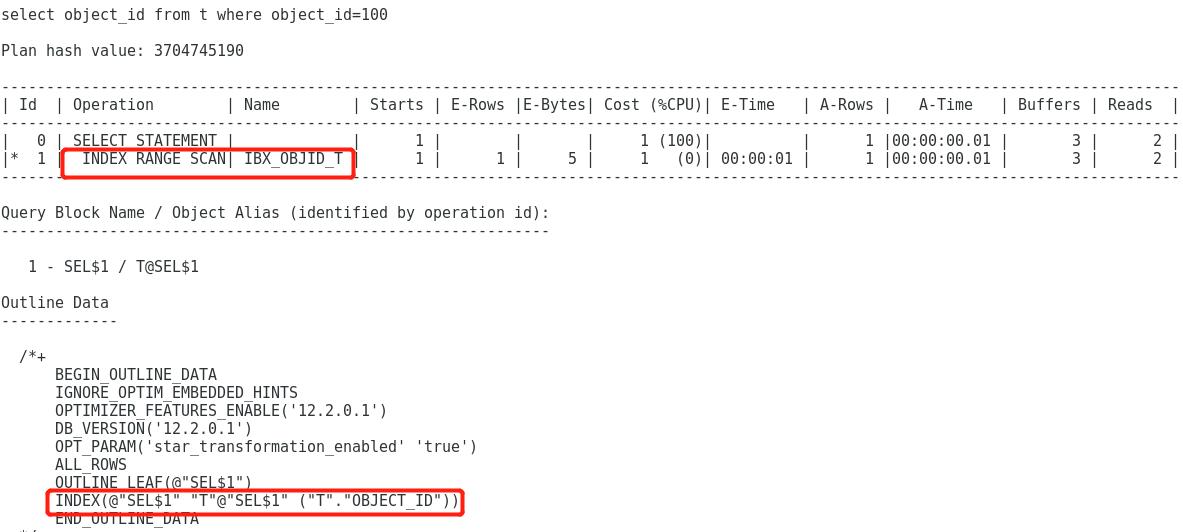
Index unique scan只会扫描索引高度的索引块，其性能仅次于table access by user rowed。

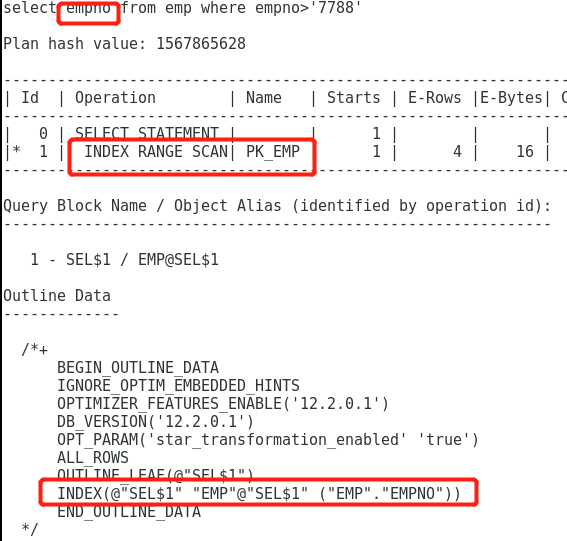


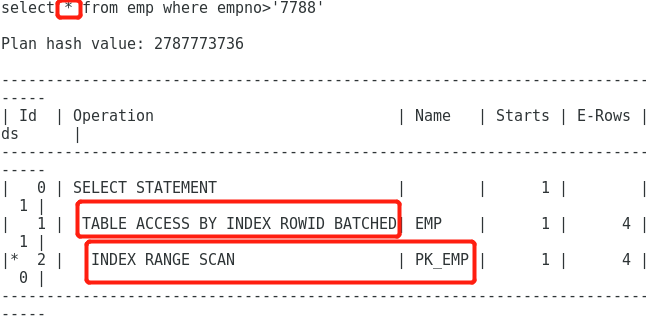


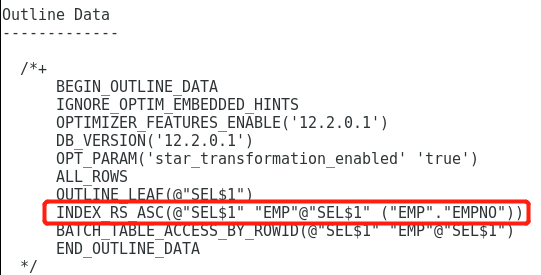
## INDEX RANGE SCAN/INDEX RANGE SCAN DESCENDING（单块读）

索引范围扫描，注意返回的数据是有序的，而且是升序的。如果是按索引列降序排列，就会是index range scan descending。也是可以从outline中看到的，我们现在分别来看一下这俩种情况的例子：



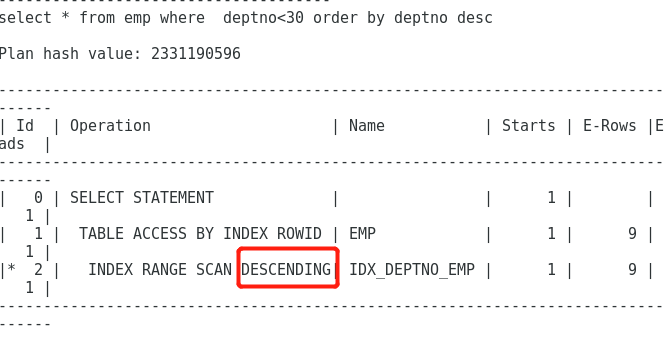


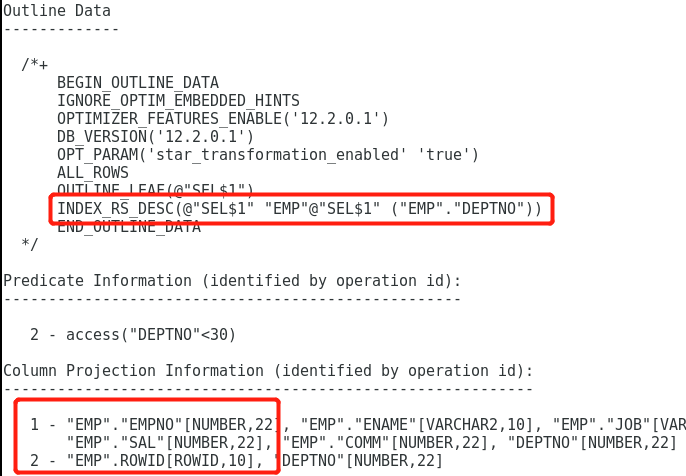




通过上面3个案例可以总结出以下几点：

1. 通过图一图二，说明index range scan发生的条件是要么就是对非唯一索引，即使是等值查询也会范围扫描（因为oracle并不能确定这个值的记录是唯一的，所以至少要扫描俩个键值）；要么就是范围查询，即使是对唯一索引列也不例外。
2. 通过图三（跟图四是一起的）的outline与前俩图的outline比较，发现回表的时候，hint变为了INDEX\_RS\_ASC也就是索引升序范围扫描，说明其默认按照索引列对表的数据做了个排序，因为索引默认就是以升序存储键值的（注意：这里并没有发生真正的排序）。





通过这个案例，我们可以看到index range scan descending，并且outline中的部分变为index\_rs\_desc，同时注意到在column projection中并没有keys字样，说明没有发生真正的排序。

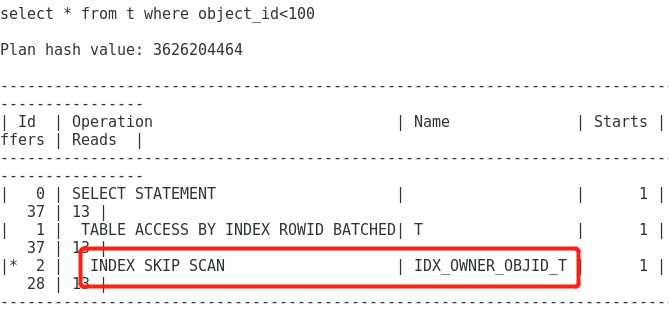
其实，索引范围扫描默认是从最左边的叶子快开始，然后往右边的叶子块扫描（这里又一个小的知识点：就是叶子块之间是双向链表，也就是说从一个叶子块到隔壁叶子块直接就能过去，不需要再从根扫描一回），检查到不匹配的数据的时候就停止扫描。而index range scan descending的意思就是，既然是要求按降序排列，那就从最右边的叶子块往左扫描好了，这样本来就是降序的，省的把数据取出来再做一次降序。

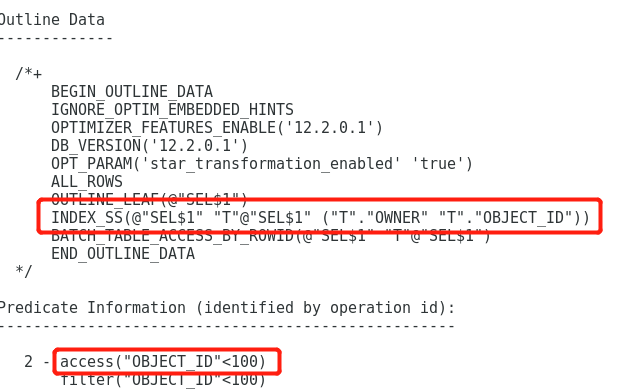
索引扫描扫描的也是数据块，它比全表扫描的优势就在于它的定向性。那如果，要得到的结果集比较大，那它的优势就显的不明显，而单块读的劣势就被放大了，再加上如果要回表，那单块读的劣势就又被放大了一倍。

所以，我们要注意返回结果集的大小，以及在建索引的时候要考虑这一列的选择性好不好，还有就是如果有回表，看能不能通过建联合索引消除回表。

## INDEX SKIP SCAN(单块读)

索引跳跃扫描，就是过滤条件中的列在一个索引中，且不是引导列，且引导列基数很低，就会发生索引跳跃扫描。来看例子：



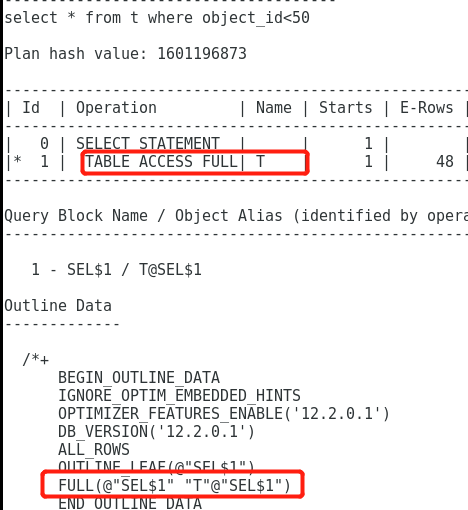


这个索引是建在t表的(owner,object\_id)上的，但是查询条件中并没有owner列，而且owner列的选择率是26/72656，可以说很低，于是就发生了索引跳跃扫描。

一般，如果条件允许的话，可以直接在发生索引跳跃扫描的列上建索引。

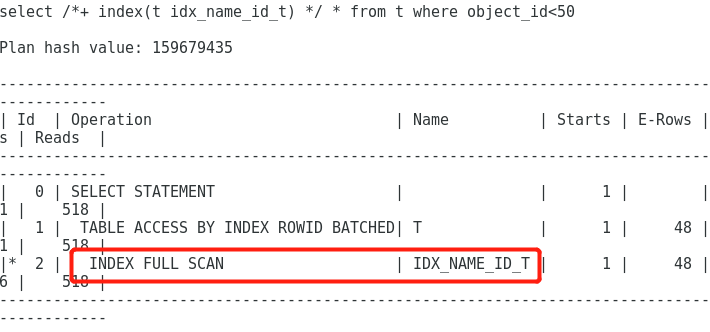
所以在建组合索引的时候一定要注意，列的顺序，尽量不要把基数低的列排在基数大的列前面。

我们换下索引，把索引建在(object\_name,object\_id)上，把之前的索引删掉，再来看下效果：

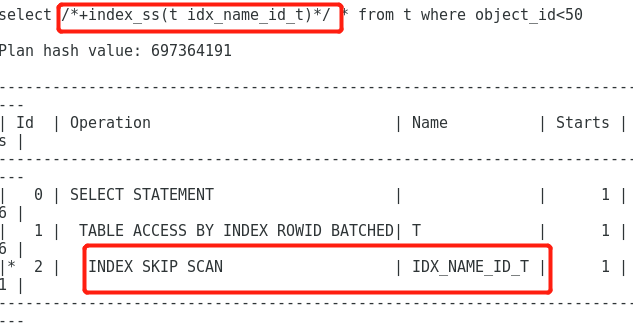


现在就不走跳跃扫描了，就会走全表扫描，因为object\_name的基数是60333，比较大，所以oracle就不会再选则跳跃了。

那如果我直接指定这个索引呢？



并不会走跳跃扫描，而是走的索引全扫描，看来只能用skip scan的hint了：



所以不同与前面，想强行走skip scan就必须是用index\_ss()的hint。

## INDEX FULL SCAN(单块读)

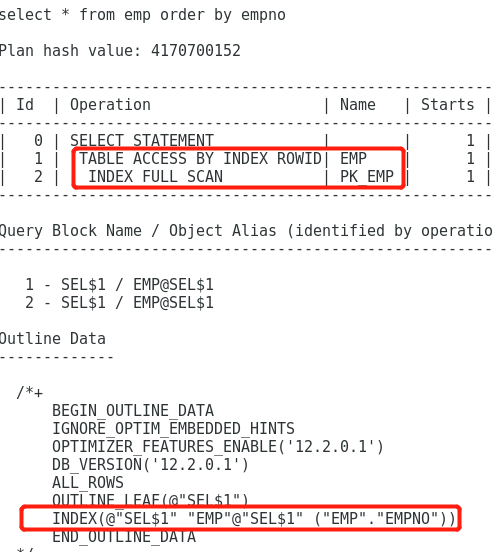
这个就像table access full一样，一般是不太想见到的。索引全扫描，是扫描所有的索引块，而且是一块一块的扫，所以索引很大的时候，性能就极差。

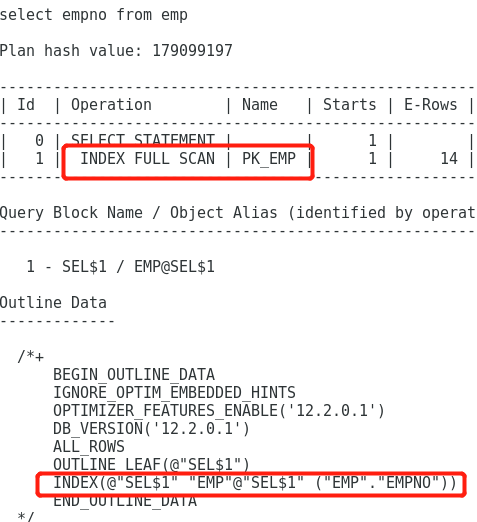
首先，发生这种情况的条件有很多，但是有一条是必须的，该列必须非空（无论是过滤成非空还是本身列定义就是非空）。

Index full scan唯一的一个好处就是返回的如果是按索引列排序的，那可以省去排序这道工序。分页框架主要也就是利用这个特性，所以在分页框架中出现index full scan还回表就不用奇怪。

除了上面这种情况，index full scan基本上都该优化，尤其是还发生了回表。一般index full scan是不该发生回表的，因为index full scan是单块读，回表也是单块读，其实就是做了个单块读的全表扫描，这是显然不合理。所以除了分页框架，发生了索引全扫描加回表都该排查优化，比如上面的3.4中的强行指定索引。

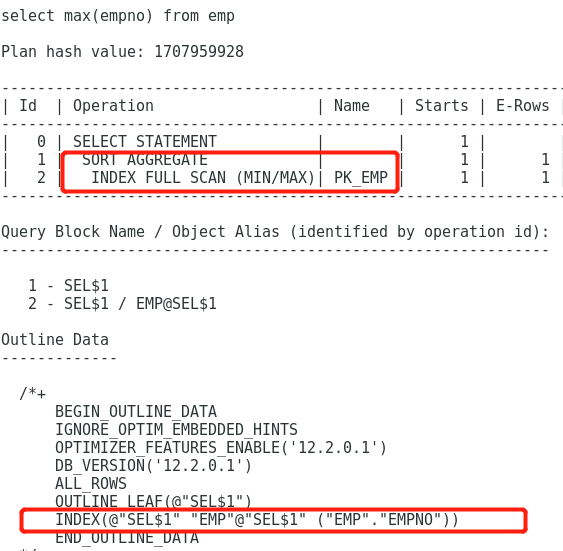
这里举个例子看下俩种合理的情况就是了：

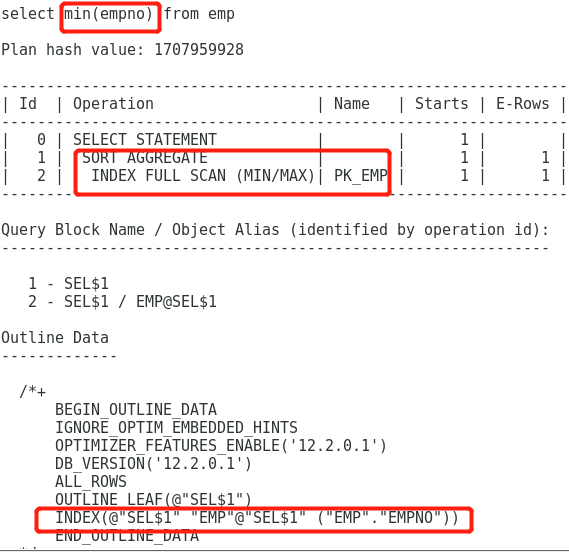




## INDEX FULL SCAN(MIN/MAX)(单块读)

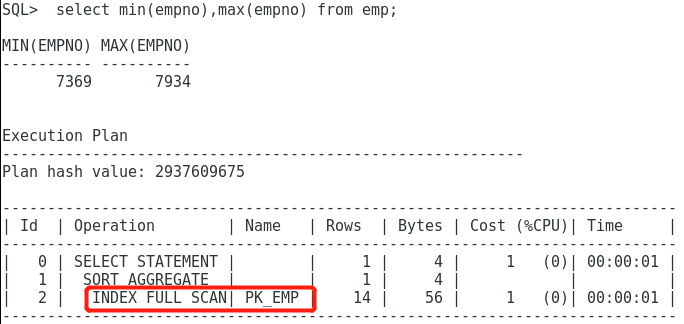
这个别看也是full scan，这个就是这么写一下，其实只扫描索引高度个索引块，也就是扫描最左边或者最右边的叶子块，效率跟index unique scan是一样的。举俩个例子，就把发生这个的所有情况都涵盖了：



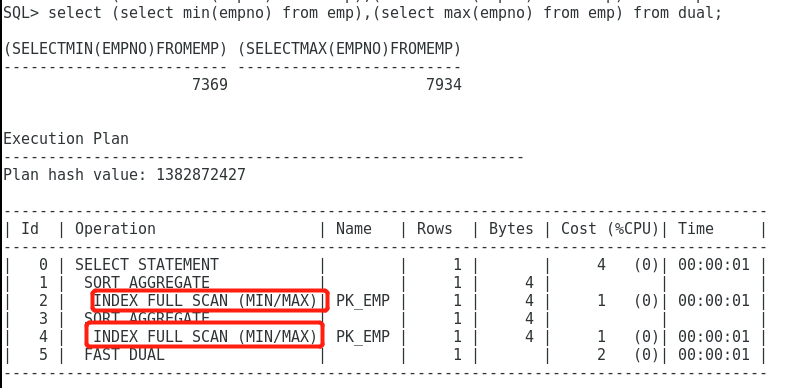


这里说一个书上看到的优化案例：

Select max(empno),min(empno) from emp;



发现走了索引全扫描，这要是索引块很大，那性能就非常差了，怎么优化呢?很简单，拆开查，看下效果：



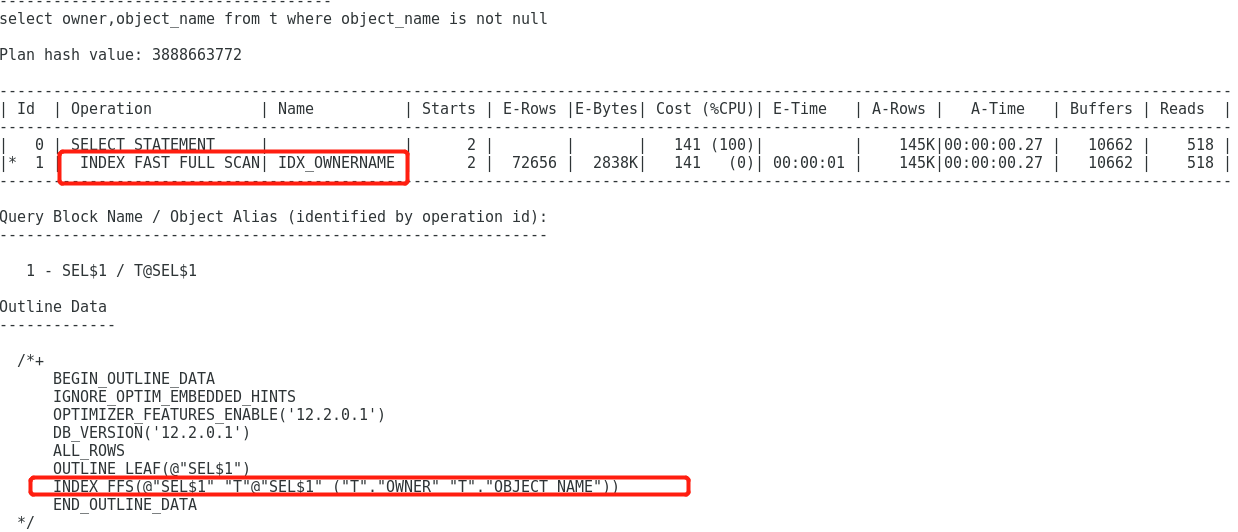
这样就相当于去查了俩次索引块，但每次都只查一个值，这个就要比把所有叶子块都扫一遍效率高的多了。

## INDEX FAST FULL SCAN（多块读）

索引快速全扫描，用途有俩个：

1. 优化查询部分列的全表扫描
2. 优化不需要排序的索引全扫描

看个index fast full scan的例子：



这个索引是建在t表的(owner,object\_name)列上的。注意到hint是INDEX\_FFS。

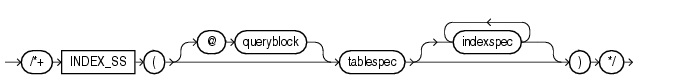
对于INDEX FAST FULL SCAN主要注意以下几点：

1. IFFS是扫描所有的索引块，而index full scan只扫描所有的叶子块，但即使这样，一般来说索引快速全扫描的多块读还是比索引全扫描的单块读效率高。因为IFFS耗费的IO更少。
2. 能用IFFS优化全表扫描，是因为不管你要查询多少列，全表扫描都要扫描所有列，而IFFS就只扫描要查询的列。
3. 如果索引块已经都缓存在内存中的话，index fast full scan和index full scan的效率就差不多了。
4. 因为oracle是行存储数据库，所以IFFS可以用来优化全表扫描，如果是列存储模式的的数据库，IFFS就没有意义了。

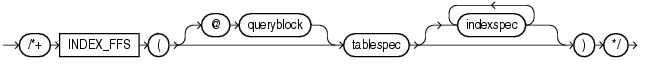
## 本章HINT及等待事件的总结：

### Hint

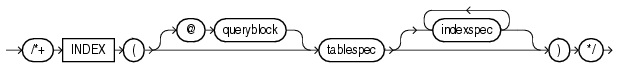
INDEX SKIP SCAN:

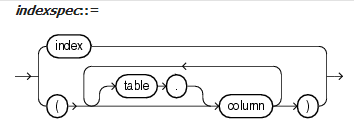


INDEX\_FAST\_FULL\_SCAN:



其他：





### 等待事件

INDEX FAST FULL SCAN:

如果开启了并行，就是direct path read，一般情况是：db file scattered(离散) read

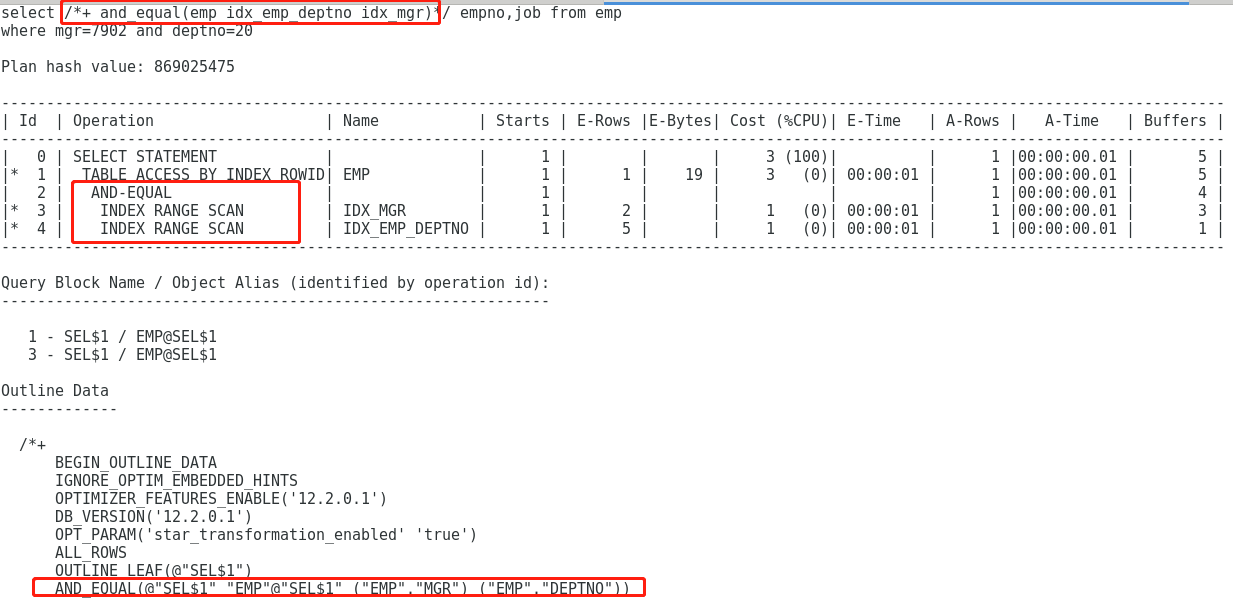
INDEX RANGE SCAN, INDEX SKIP SCAN, INDEX FULL SCAN:

这三个都是：db file sequential read

# 其他常见的执行计划

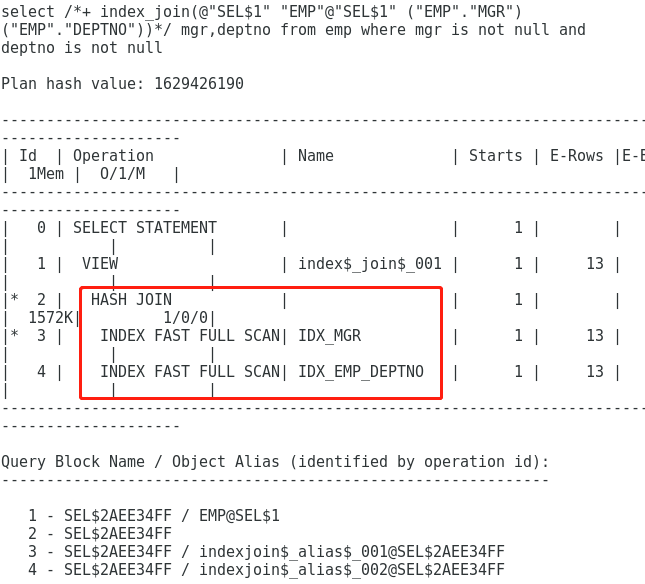
## AND-EQUAL

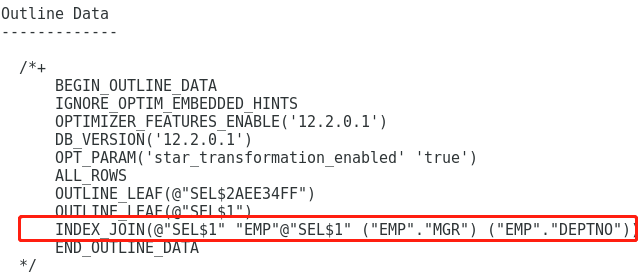
就是index merge，就是where条件中出现不同单列的等值过滤条件，而且过滤列上都有索引，这时候oracle就可以在扫描完索引不着急回表，而是拿着各个索引扫描来的rowid集合进行比较，匹配到一样的rowid，再回表得出最后想要的结果集。



## INDEX JOIN

index join就是俩个索引扫描后的得到的俩个结果集然后通过rowid关联(join)。这当然没有直接扫描这俩个列的组合索引好，这就是Oracle为CBO提供的一种可选的执行路径罢了。



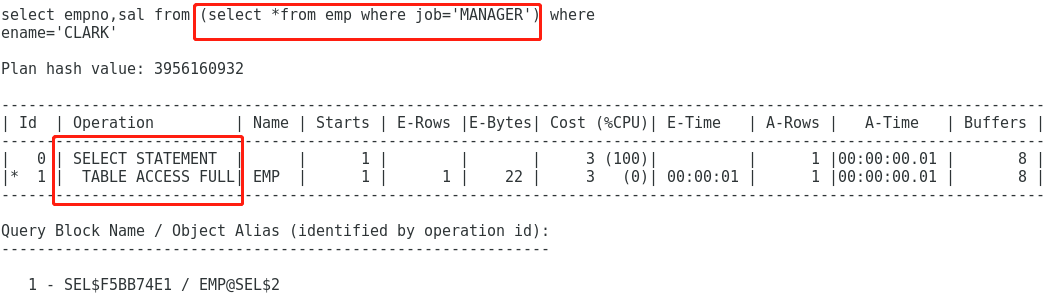


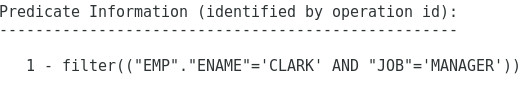
## VIEW

view出现的情况我结合实际工作和书本，总结如下：

1. 没有进行视图合并的视图，当然有可能做了视图合并，但也会出现view；
2. 没有展开的子查询；
3. 比如3.4中的例子，oracle临时性的将一个结果集规定为一个视图。

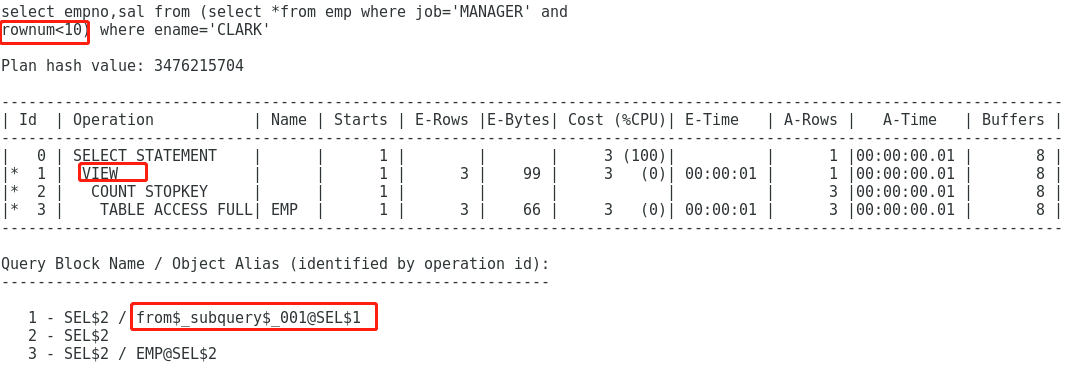
其实，我认为所有的归结一下，本质都是子查询不展开。这里举个简单的例子：

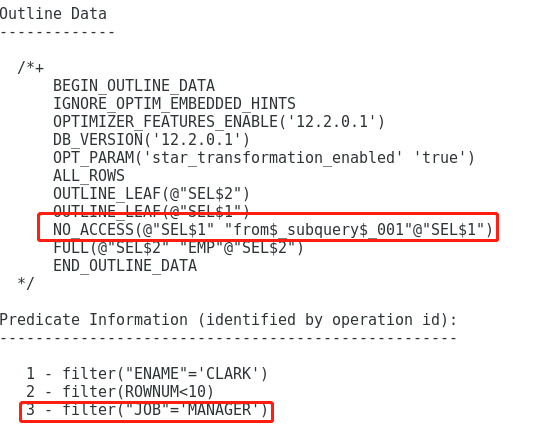




其实视图就是个子查询，所以我在这里直接使用了子查询，我们发现这里子查询或者说是视图被展开了。谓语信息对这一行为反映的很明确。

那么现在我们不让子查询展开，或者说如果这是个视图，我们不让它合并，对于子查询我们可以使用no\_unnest，对于视图，我们可以用类似rownum的方式把它的意义固定为一张不能展开得视图（固化子查询）。





这次我们看到，首先走的是SEL$2中得emp按照job的过滤条件走的全表扫描，也就是说子查询并没有被展开，或者说没有发生视图合并，于是我们看到了VIEW的关键字。

## FILTER

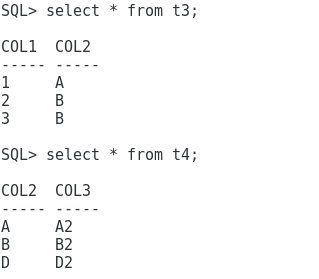
### FILTER的概述

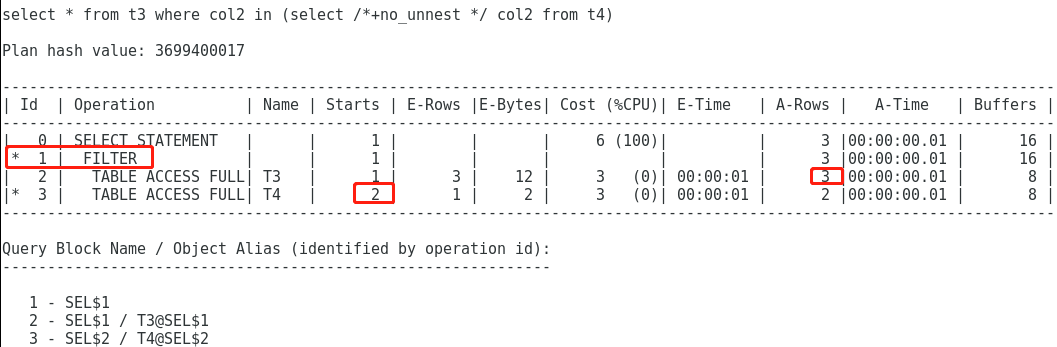
FILTER的执行过程：

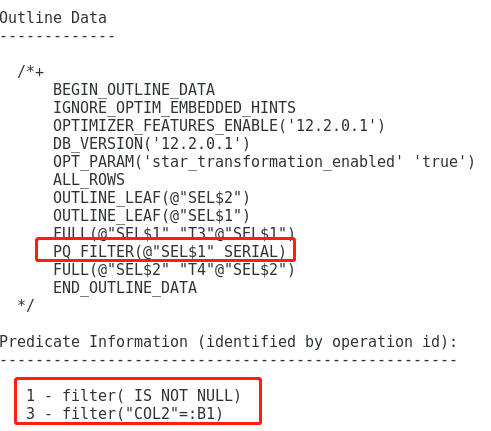
1. 得到一个驱动结果集；
2. 根据一定的过滤条件从上述驱动结果集中滤除不满足条件的记录；
3. 结果集剩下的记录返回到下一个执行步骤。

一般FILTER在整个sql快要执行完执行，一般id小于等于3。

下面我们来看一个例子：





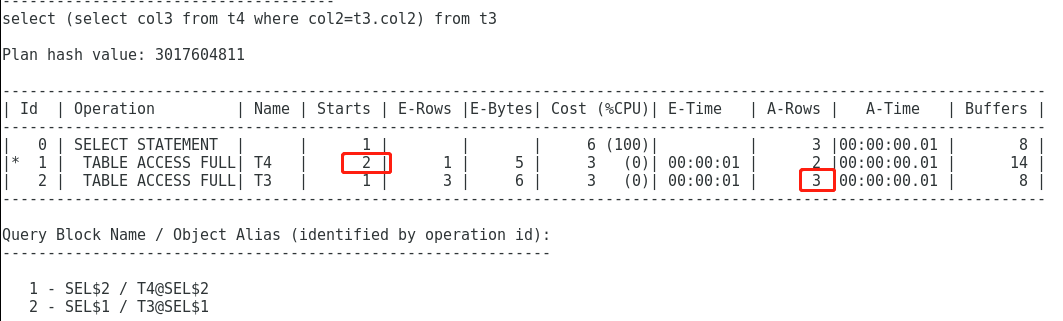


可以看到id=3的谓语信息中，是发生了一个传值的过程（发生传值的步骤就是循环还是的步骤），这个值很显然来自于t3表，也就是说想到于t3和t4做了一个嵌套循环，连接列是col2，并且驱动表是t3。

而我们又发现，t3得到的结果集是3行，而t4只被扫描了俩次。因为t3的col2列的唯一值是俩个。

所以filter其实是一种改良的嵌套循环，它不像嵌套循环那样驱动结果集，有多少行就循环多少次。

我们来看下标量子查询的情况：



注意T3的结果集和T4的循环次数，结果一模一样！

还有一种filter，下面只有一个儿子，这种filter只有过滤作用，一般只需要关注有俩个或者俩个以上的儿子的filter。

### FILTER的产生原因

如果子查询没能展开，在执行计划中就会出现FILTER，它的算法跟标量子查询一模一样。具体的说就是，如果在子查询语句中含有exists或not exists，并且子查询别固化（union/union all /start with connect by/rownum/cube/rollup），那么就容易产生filter。

### FILTER的消除

消除filter的道理和后面4.1.2.2中的消除标量子查询的道理一样，就是为了使执行计划更灵活更容易让我们掌控，因为filter一旦产生，也就意味着驱动表被固定，就几乎意味着执行计划被固定了。

所以消除filter并不是因为filter性能差。

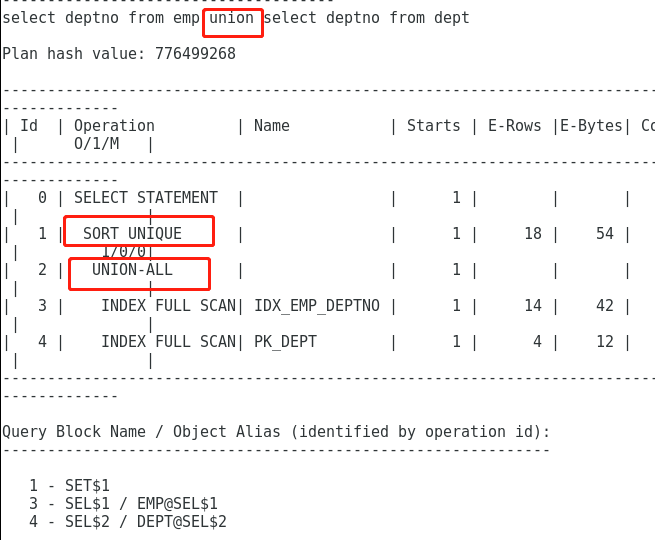
消除FILTER的思路就是：

1. 把exists改成in，因为虽然子查询都是被固化了的，但是exists的子查询中有主表的连接列，而in没有。意思就是虽然被固化了，但是没有主表连接列，CBO就不一定选择传值进入子查询，也就不会有filter。
2. sql改写，使优化器可以对子查询进行展开，一般消除用filter是需要改写sql的，意图用unnest来强行展开子查询往往是徒劳的，因为一般来说，能展开的话CBO自己就展开了。

## UNION ALL/UNION

这个真没啥可说的，执行计划里面都是union all关键字，union就是先union all,然后排序去重（sort unique）。

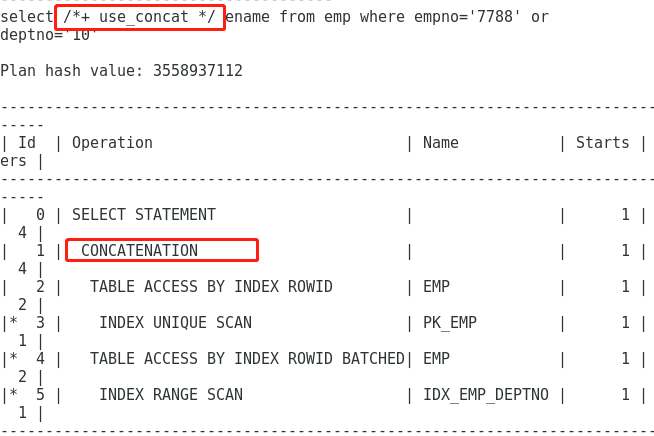
所以union对应的关键字要多一个sort unique:

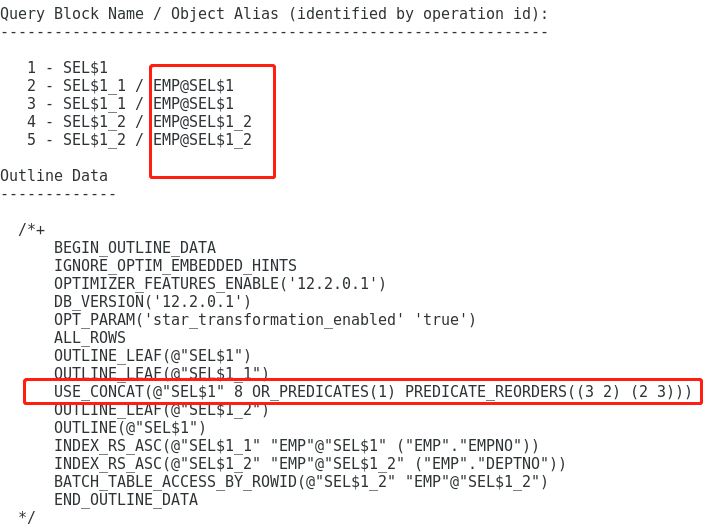


## CONCAT

IN-List扩展或者OR扩展，执行计划中对应的关键字是concatenation。

我们先看个例子：





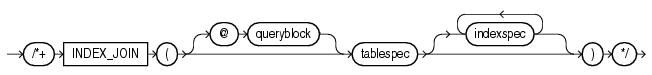
从query block 可以看出，其实Oracle是把这个语句拆成了俩块，相当于做了俩个查询然后union all。

## 本章HINT总结

AND-EQUAL:

/\*+ and\_equal(@queryblock tablespec indexspec) \*/

INDEX JOIN：

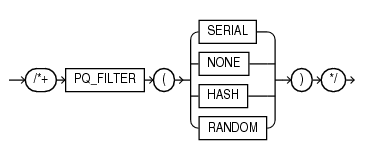


VIEW：

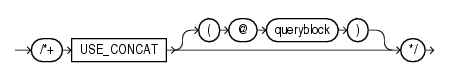
这里就只展示下不允许子查询展开的hint：



FILTER：



CONCAT:



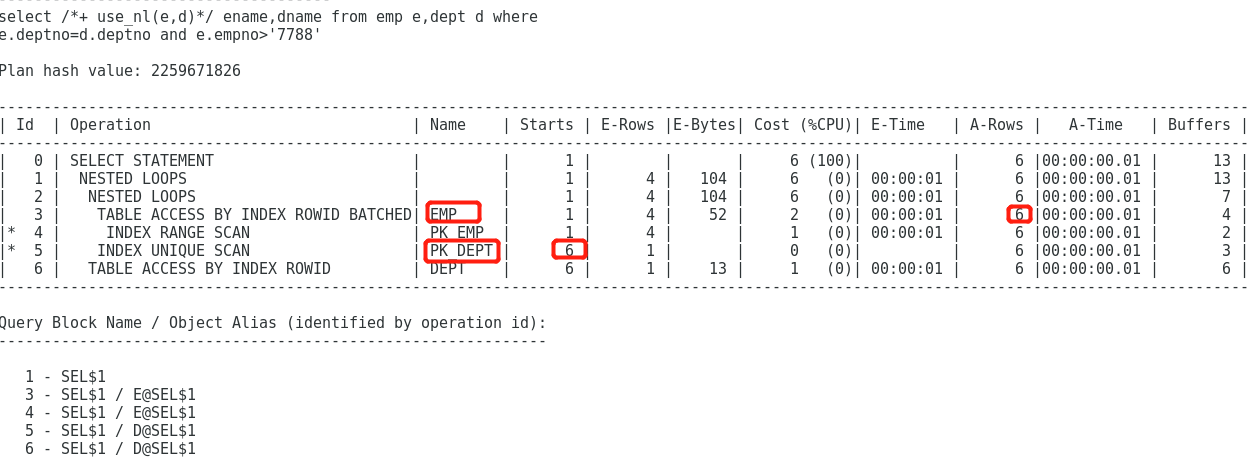
# 表连接方式

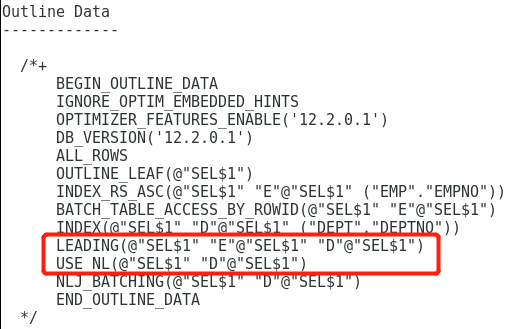
## NESTED LOOPS

### 显式的嵌套循环

嵌套循环就是驱动表返回一行数据，然后通过连接列传值给被驱动表，驱动表返回多少行结果集，被驱动表就被扫描多少次。

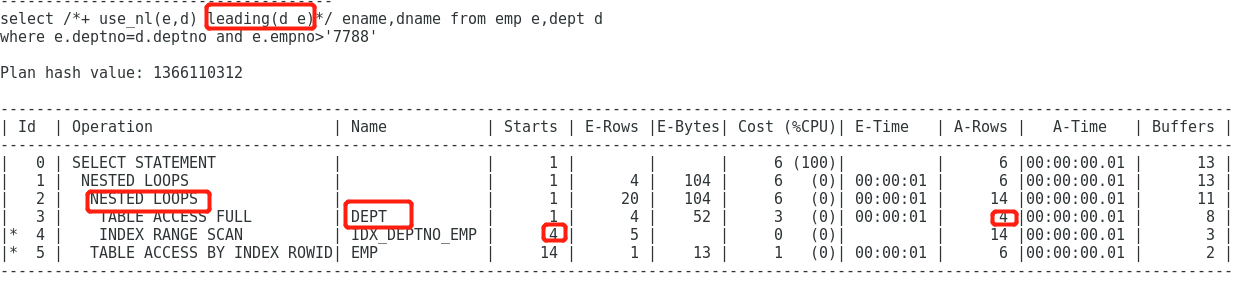
我们来看个例子，看下nl在执行计划中的表现：

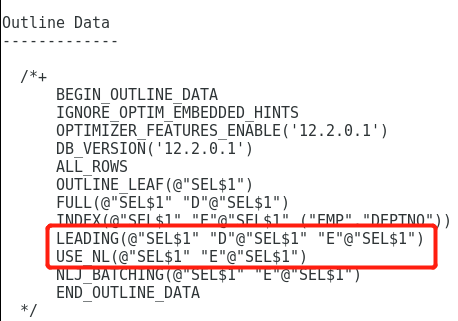




先看执行计划部分，靠上的表（结果集）是驱动表，那下面的就是被驱动表，这里驱动表是emp，被驱动表是dept。然后看starts列，驱动表只有一次循环了一次，而被驱动表循环了6次，为什么呢，看到驱动表哪一行的a-rows是6行，也就是说被驱动表循环了6次，是因为驱动表的返回的结果集有6行。这就很明确的表达了嵌套循环的意思。驱动表返回的结果集行数就是嵌套循环进行循环的次数。

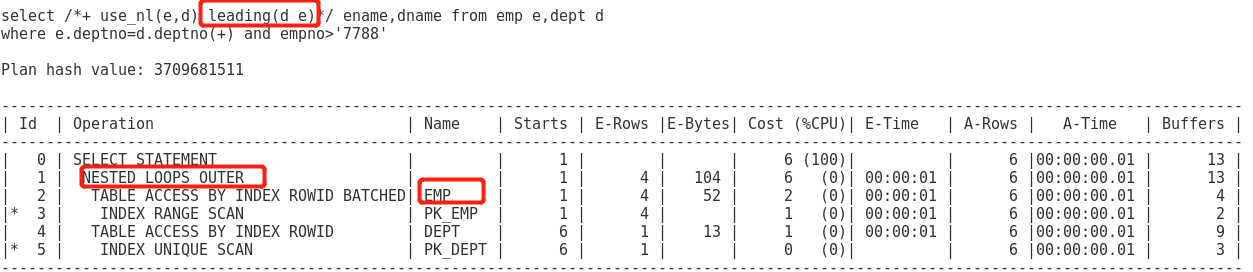
然后看outline部分，USE\_NL就是嵌套循环的hint，leading括号中靠前的表就是驱动表。也就是说，嵌套循环的驱动表有的时候是可以改变的：



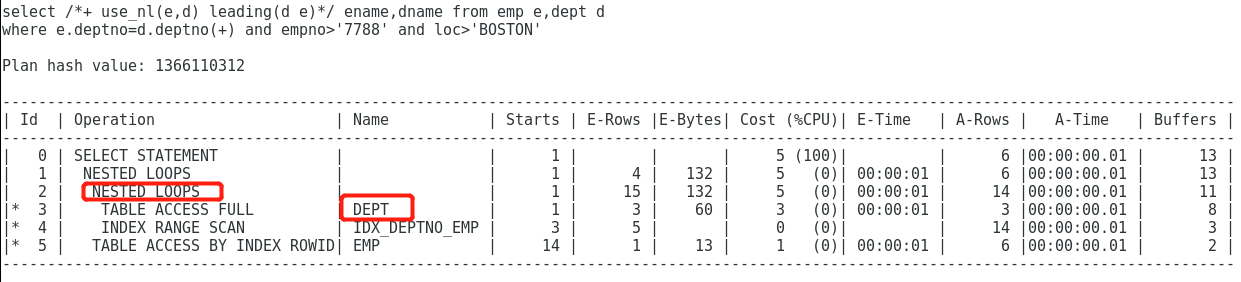


其实对比上下的outline可以发现，use\_nl的括号里只放被驱动表就好了，leading括号里其实写的就是扫描表的顺序，因为驱动表先被扫描，所以驱动表写在前面。

但是，hint是只有当采用hint的执行计划合理时才会生效，比如说左连接的时候，驱动表就是固定为左表，因为右表遇到与左表不匹配的要返回null值。但是如果右表有条件，驱动表又可以变为右表，因为去除了右表与左表匹配不上而置为null的可能性，所以本质就变成了内连接，所以驱动表又可以变换了。



看到，leading并没有生效，执行计划中也明确标识这是个外连接(outer)。



看到，加上了限制从表的条件loc>’BOSTON’的条件以后，oracle把查询转换成了内连接，leading就生效了。

用NL必须返回结果集的大小，如果查询返回结果集较小，就可以考虑用NL，又因为驱动表的结果集行数决定循环次数，所以要选择通过过滤条件能过滤出较小的结果集的表作为驱动表，而又因为被驱动表时通过连接列接受到驱动表传来的值进行扫描的，所以被驱动表的连接列上应当有索引，而且被驱动表通过过滤后的结果集也应当较小，就时不能太大，否则索引扫描消耗也很大，这就是为什么要看最总查询返回的结果集的大小。

如果满足上述条件，就可以考虑用NL。

还有一点比较重要，就是嵌套循环是可以实现快速响应的，也就是说不用等到全部关联完，就会开始返回已经等到的结果集，如果hint中加入FIRST\_ROWS，优化器会更倾向于选则嵌套循环。

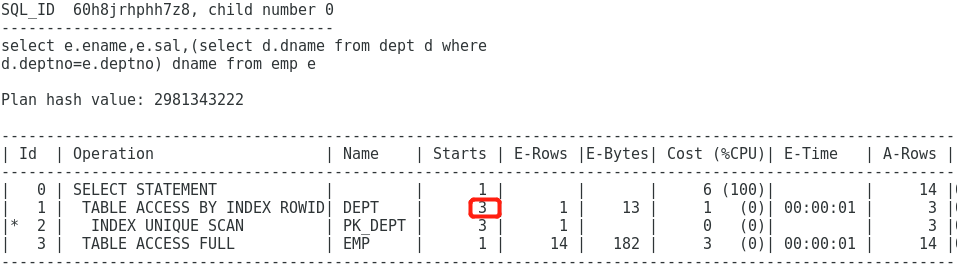
### 天然的嵌套循环

#### PLSQL代码中的游标循环嵌套sql

这种嵌套循环是纯天然的，游标返回的结果集就是驱动表，循环中的sql就是被驱动表，性能消耗道理同上，所以注意点也同上。

#### SCALAR SUBQUERY

标量子查询也是一种天然的嵌套循环，看个例子：



可以看到，虽然没有nested loop字样，但是，dept表被扫描了3次，这里可以看出来，其实相当于emp表做了驱动表，标量子查询中的dept做了被驱动表。只不过被驱动表循环的次数取决于与主表连接的连接列在主表中的基数，因为emp表中只有3种deptno值，所以dept表循环了3次。

标量子查询的主表是固定的，如果主表的连接列基数很大，那性能就会很差。

优化标量子查询的思路有：

1. 因为标量子查询中的表是驱动表，所以同理，驱动表的连接列上应建索引。
2. 因为标量子查询这种嵌套循环主表是固定的，循环方式也是固定的，那么不便于后期数据量上涨后调优，所以可以把标量子查询改为外连接，这样可以使用hash连接，并且有了调整驱动表的可能性。只所以应当改为外连接，是因为，标量子查询如果匹配不到连接条件就会返回null，这于外连接是同原理的。
3. 如果主表上的连接列是外键，标量子查询中表的连接列是主键，那就可以直接改为内连接，因为不存在匹配不上。
4. 12c中，简单的标量子查询会被优化器改为外连接。

### 连接列的数据类型不同

我们知道，用NL的前提之一就是被驱动表的连接列上有索引。那么如果连接列在俩张表上的数据类型，就存在隐式转换，此时索引还可以用的到么？

我们先看如果被驱动表的连接列是number，驱动表的连接列是varchar2，会是怎样的情况呢？

SQL> desc dept

Name Null? Type

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

DEPTNO NOT NULL NUMBER(2)

DNAME VARCHAR2(14)

LOC VARCHAR2(13)

SQL> desc dept1

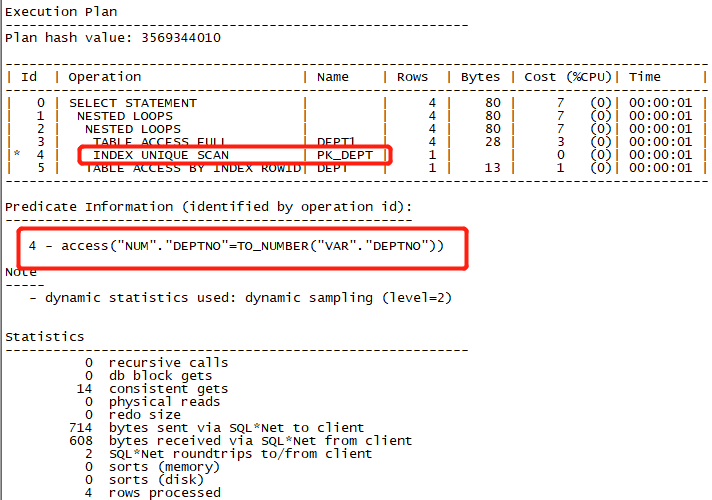
Name Null? Type

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

DEPTNO VARCHAR2(10)

我们现在来看一条sql：

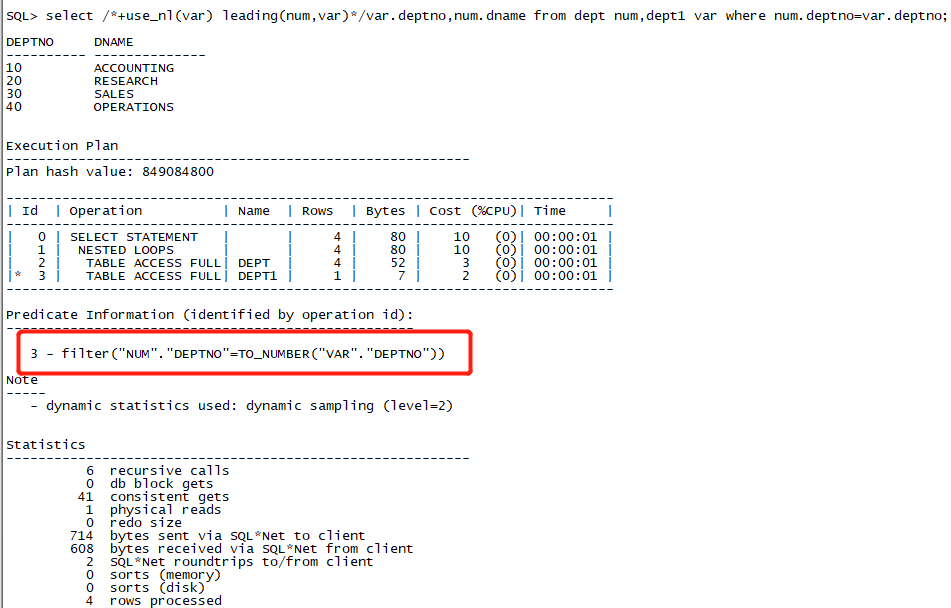
SQL> select /\*+use\_nl(num) leading(var,num)\*/var.deptno,num.dname from dept num,dept1 var where num.deptno=var.deptno;



可以看到，确实发生隐式转换，但是由于是驱动表的连接列转换为number，所以并不影响走被驱动表连接列上的索引。

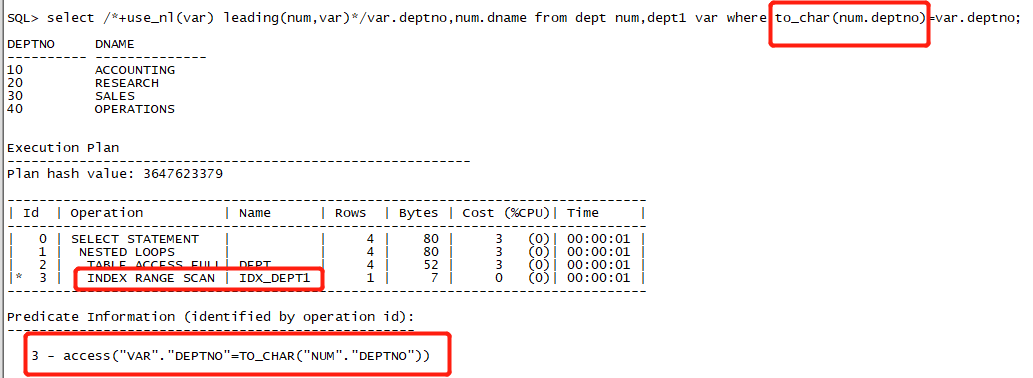
现在我们换一下，如果是被驱动表连接列是varchar2，驱动表的连接列是number，又会怎样？

SQL> select /\*+use\_nl(var) leading(num,var)\*/var.deptno,num.dname from dept num,dept1 var where num.deptno=var.deptno;



这里dept1的deptno上是有索引的，但是我们发现由于隐式转换扔是选择将字符串类型转化为数字类型，导致被驱动表的连接列上的索引无法被使用。

我们现在把sql进行改写，显示的将驱动表的连接列使用to\_char进行转换，再看下效果：



现在可以看到，执行计划可以走到被驱动表连接列上的索引了。

所以，当发现使用nl调整执行计划，并强制sql走连接列的索引时，发现走的是index full scan或者扔没有走连接列索引时，就要去确定连接列在俩表的数据类型是否一致，当然，如果有谓语信息的话，这一点可以直接从谓语信息看出来。

## HASH JOIN（消耗PGA）

哈希连接的内部实现这里做个简单的说明：驱动表过滤后的结果集做成hash table，放在pga中，如果放不下，就放在磁盘中，被驱动表的按照连接列做哈希运算，然后用一个个哈希值到驱动表的hash table进行probe，找到数据就是关联上了，没找到就没关联上。

hash join只支持等值连接。

一般来说驱动表是经过过滤条件后得到的结果集小的表。

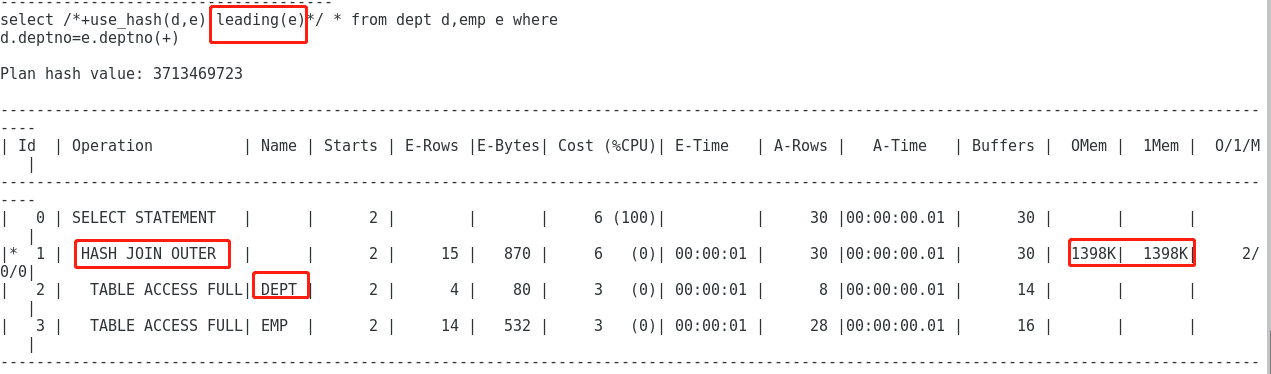
可以通过10104事件来观察sql做hash join时的具体过程。

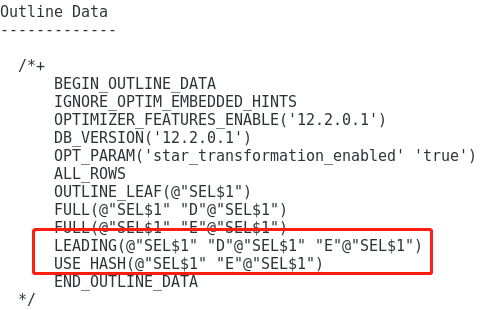
如果返回的结果集大的话，就应该考虑使用hash join。因为hash join驱动表与被驱动表只扫描一次。

hash join的瓶颈在于PGA的大小，当PGA放不下用驱动表结果集生成的hash table时就会把多余出来的放到临时表空间，进而产生磁盘的哈希连接。

哈希连接的驱动表与被驱动表的连接列都不用建索引，因为这表关联的时候并不是一个传值的过程。

如果是外连接使用hash join，我们是可以改变驱动表的，但是不是用leading，而是用swap\_join\_inputs。如果是内连接，用leading就可以了。

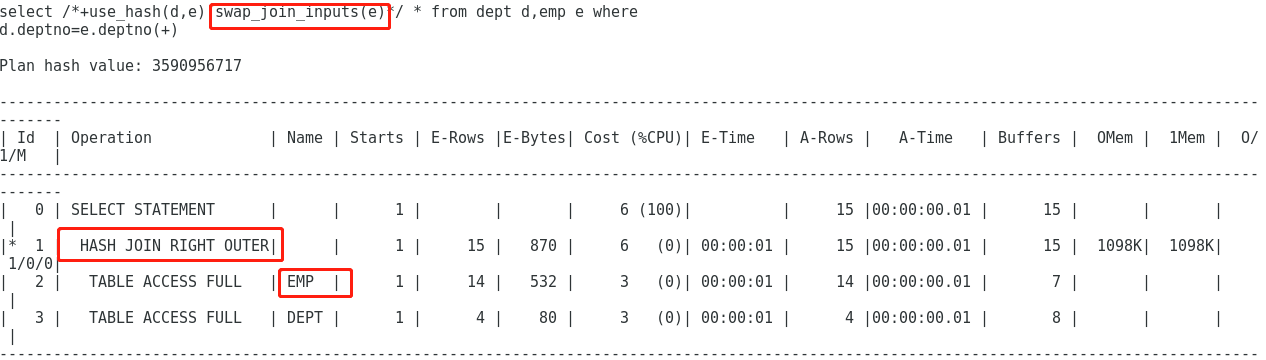


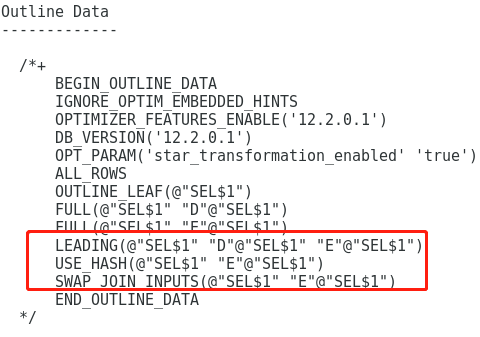


看到leading并没有起作用。

这里顺便介绍一下hash join的执行计划，依然是靠上的表是驱动表，执行计划后面的最后那三列，说的是PGA的使用情况。hint就是use\_hash。

接着我们看下用swap\_join\_inputs的效果：





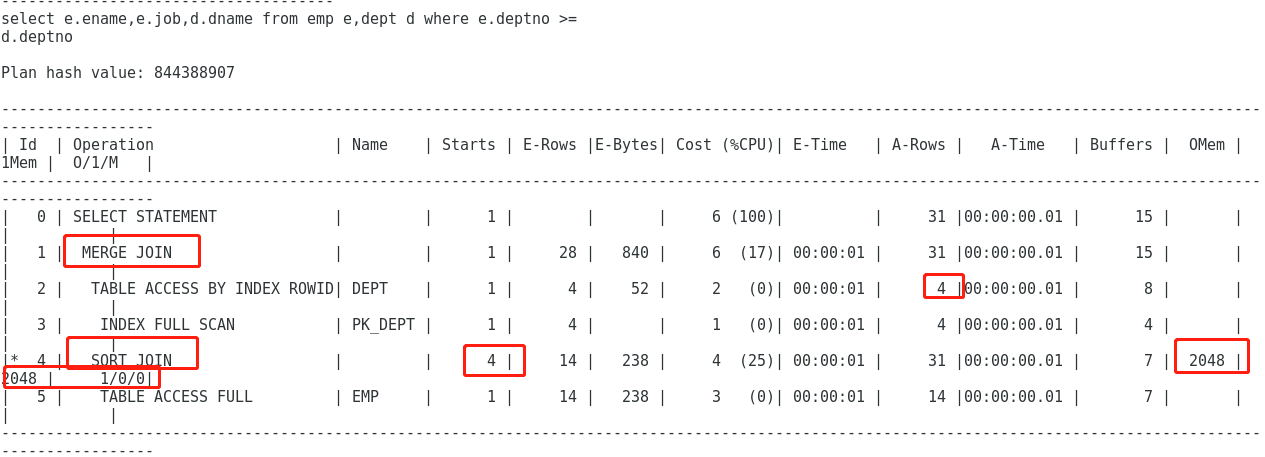
看到hash join outer变为了hash join right outer，驱动表变为了emp表，outline部分，leading括号里的内容并没有改变，而是多了一条swap\_join\_inputs。

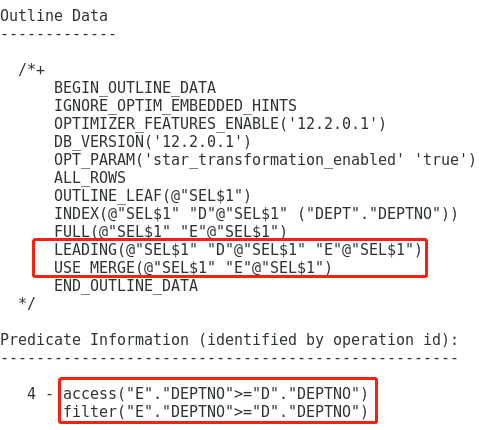
然后我们来看下hash join的优化思路：

1. 因为是用要查询的列和连接列做的hash table，所以尽量不要使用select \*，这样可以避免hash table过大溢出到临时表空间。如果无法避免，我们可以将临时表空间建在SSD或者RAID0上，加快临时数据的交换速度。
2. 如果PGA采用自动管理，单个进程的work area被限制在1G以内，手动管理就是2G。如果驱动表比较大，比如4GB，就可以开4个并行，从而避免溢出到临时表空间。
3. 总体来说，优化hash join就是从驱动表入手，减小由结果集生成的hash table的大小，再简单些说，就是避免这个哈希表溢出到临时表空间，产生磁盘连接。

## SORT MERGE JOIN（消耗PGA）

排序合并主要用于俩表非等值关联，比如>,>=,<,<=,<>，但是不能用于instr,substr,like呀这些的，这些只能走NESTED lOOPS。





从id=4的循环次数以及outline可以看到，罗老师和崔老师认为sort merge join是有驱动表的我认为是对的。同时我们发现，在sort join步骤有PGA的使用，是因为sort merge join是在PGA中匹配数据，这一点与嵌套循环不同。

我们发现dept表没有sort join这一步，是因为dept表走了index full scan，返回的数据本来就是有序的。

merge sort join的sort部分会把俩张表的都放到PGA中，且就算有一张表是通过索引扫描不需要放入PGA中排序，那剩下的那一张表也是返回结果集较大表（被驱动表），而hash join只把驱动表放在PGA中，且较小的结果集作为驱动结果集。

所以等值连接，一般不用SMJ。

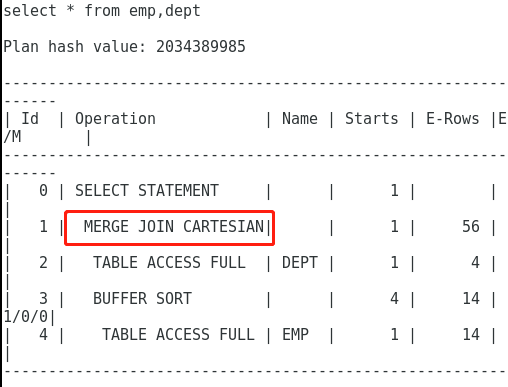
对于非等值连接，尽量从业务逻辑上去优化，尽量改写成等值连接，或者加更多的过滤条件使俩个将要进行排序的结果集尽可能的小。

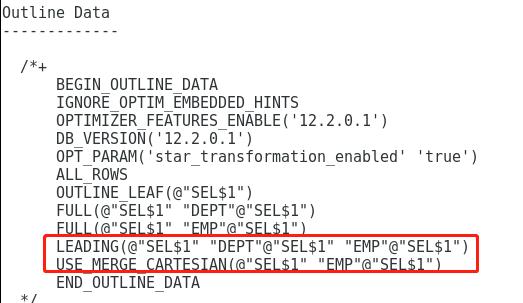
## CARTESIAN JOIN

发生笛卡尔连接的情况：

1. 基本就是多表关联，忘写关联条件了
2. 多表关联，当其中某俩个表没有关联条件的时候，优化器错误的认为某一个表返回的结果集只有一行时，也可能会发生笛卡尔连接。

虽然说当结果集真的只返回一行的时候，发生笛卡尔集没什么，但是就算是这种情况，NL也可以，所以笛卡尔连接基本没啥用，出现了直接找问题，一般就是sql写错了，少写关联条件了。





## 本章HINT总结

LEADING:



ORDERD:



NETSTED LOOPS:



HASH JOIN:



SORT MERGE JOIN:



CARTESIAN JOIN:

通过这个hint /\*+ opt\_param('\_optimizer\_mjc\_enabled','false')\*/ 禁止笛卡尔连接。

# 特殊的连接

## 反连接（Anti Join）

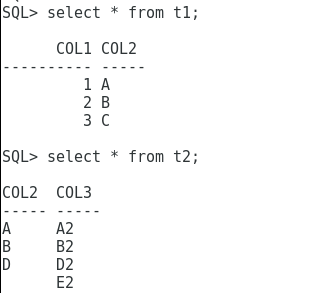
以为oracle中没有专门为反连接指定关键字，所有这里为了表述清楚，用“t1.x anti= t2.y”来表示。

注意：这里t1是驱动表，t2是被驱动表，也就是说t1取一个值到t2，然后判断，如果t2.y=t1.x，

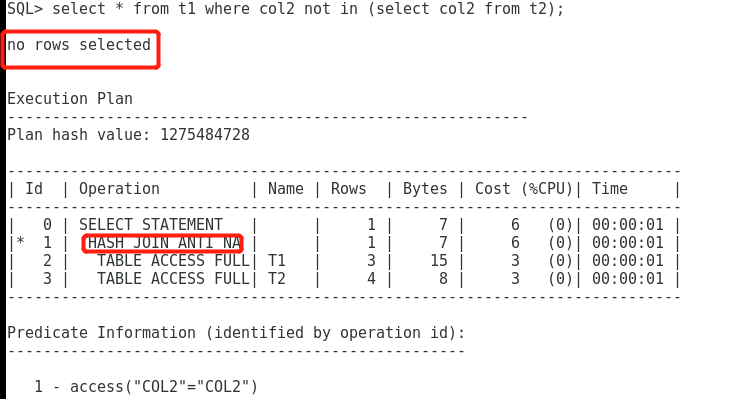
那么就从t1中把满足t1.x=t2.y的记录删掉。

Oracle通常会把not exists, not in, <>all的子查询转换为反连接，也就是说，这种连接方式属于子查询展开。

实验用表如下:

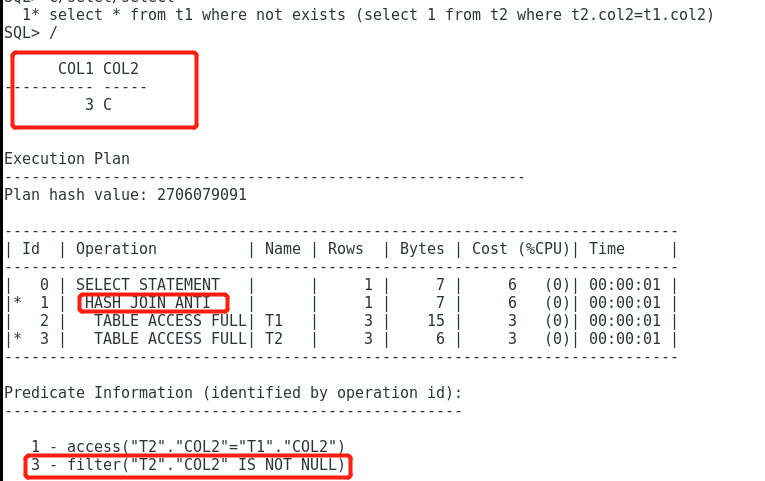


接下来看一下not in,not exists的实验效果（not ino和<>all一样）：



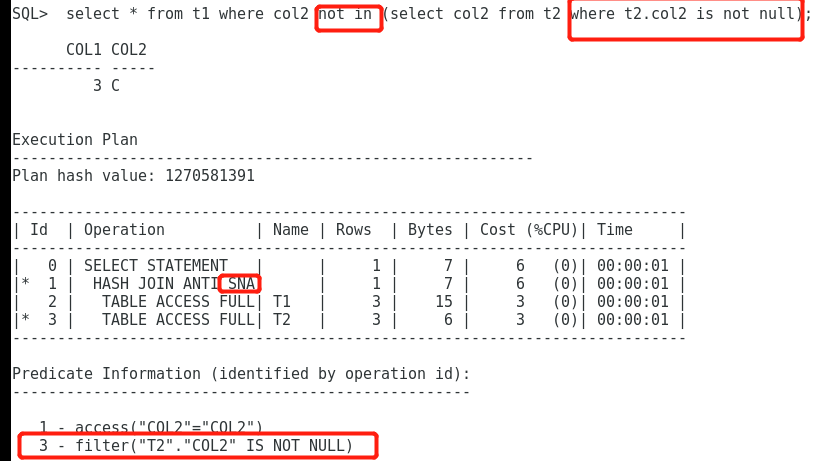
很奇怪，为啥没找到值，难道结果不应该是3,C么？

我们看下not exists的效果：



我们注意到，这次得出了我们想要的结果。那我们来找找不同。

首先我们注意到not in的执行计划中的hash半连接多了一个NA；其次，not exists的谓语信息中多了一个t2.col2 is not null。那我们再来做个实验：



现在我们发现，得到了我们想要的结果集。

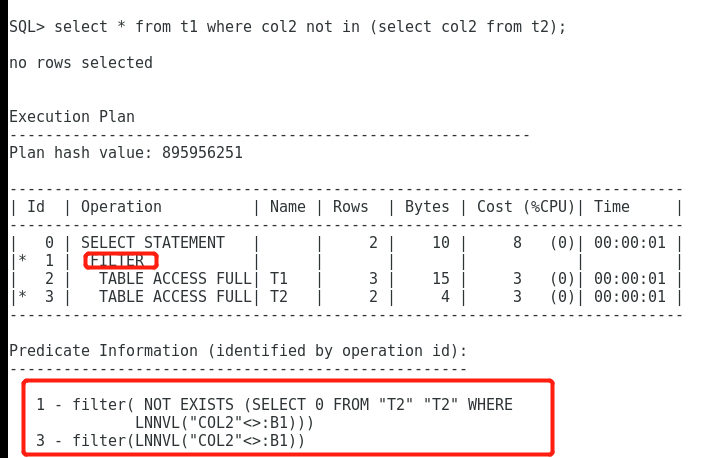
所以问题很明显，not in(同样对于<>all)是对null敏感的,也就是说如果子查询的结果集中存在null，则直接返回空结果集，所以：

Not exists 恒等于 not in (…where …is not null)

我们再来看下，这个NA是null-aware，是11gR2的新特性，这个意思就是oracles虽然使用了反连接，但是已经意识到子查询的结果集中有null，这是一种改良的反连接，如果把隐藏参数设为false：

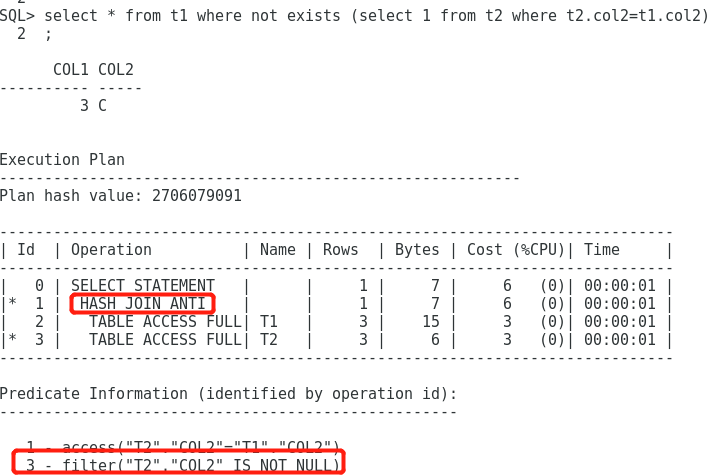
Alter session set  "\_optimizer\_null\_aware\_antijoin" =false;

再来看下实验效果：



注意到，现在oracle不会再选择走反连接了，因为再走反连接，结果集就不对了，结果正确性才是第一位的。

但对not exists不影响，因为从上面的实验可以看到，not exists会自己把null去掉。

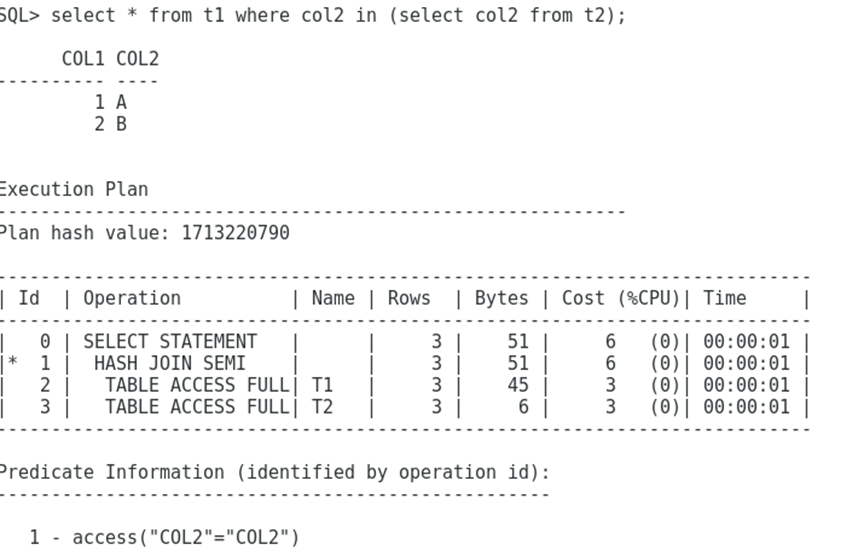


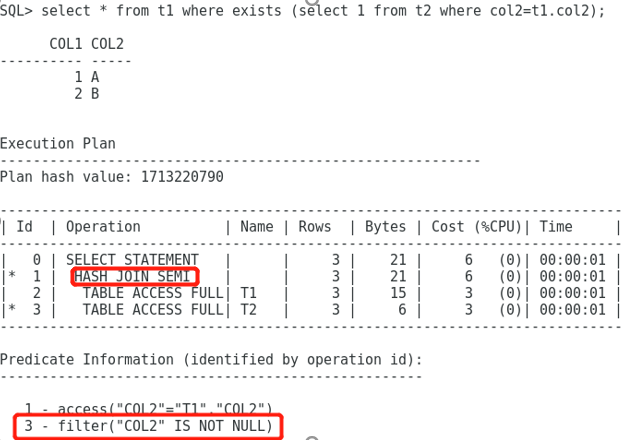
## 半连接（Semi join）

半连接的逻辑表述就是：t1.x semi= t2.y。这里通常来说t1是驱动表，也就是说t1中取x一个值，到t2中找，一旦符合t2.y=t1.x，则不管后面还有没有符合条件的，直接返回到t1表将t1表中该x值的所有行返回。也就是说半连接与普通内连接相比，实际上会去重。

Oracle通常会把exists, in, =any的子查询转换为对应的半连接。

还是上面的实验表，现在我们来看下实验效果：





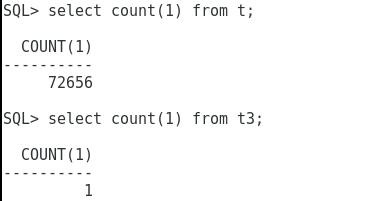
可以看到，执行计划跟结果集都是完全一样的，也就是说这仨兄弟等价。但我们还是看到，在exists的谓语信息里面有去null的行为，但是因为in对null不敏感，所以就无所谓了。

前面说过，通常来说，子查询的外表会作为驱动表，但是也有特殊情况。我们来考虑一种比较极端的情况：

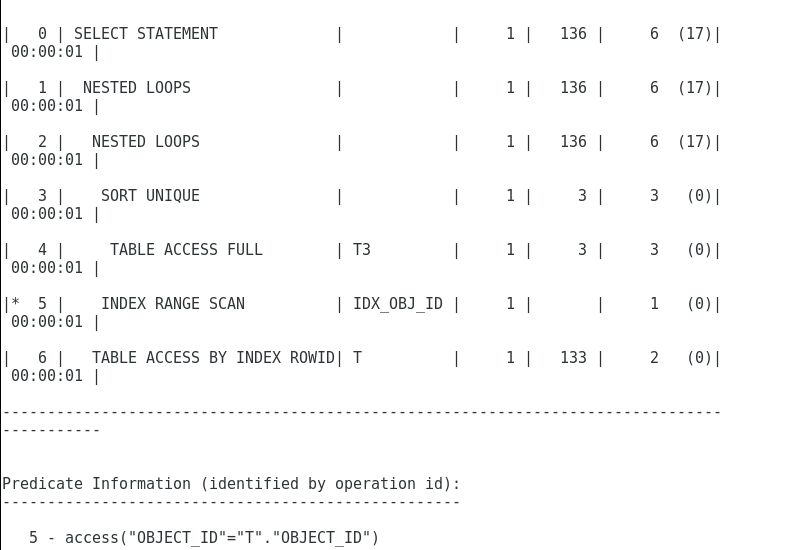
假如t2表中就只有一行数据，t1表中有7万多条数据，这时候如果t1表还作为驱动表去做到t2表中找符合条件的结果集做半连接就很蠢了，那么我又想用t2表做为驱动表做半连接，又想保持现在的语句，可以不可以呢？

可以是可以，但如果我们不用hint的话，需要做一些小小的工作：那就是给t2表的连接列建上索引。

下面就是所做的实验：



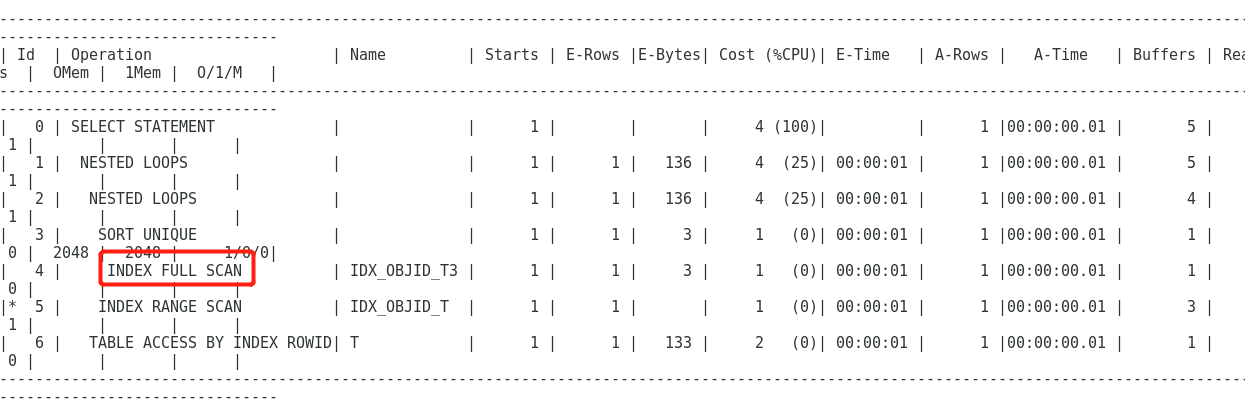
现在看看，如果t3的连接列上有索引，但是t表上没有，会怎样：



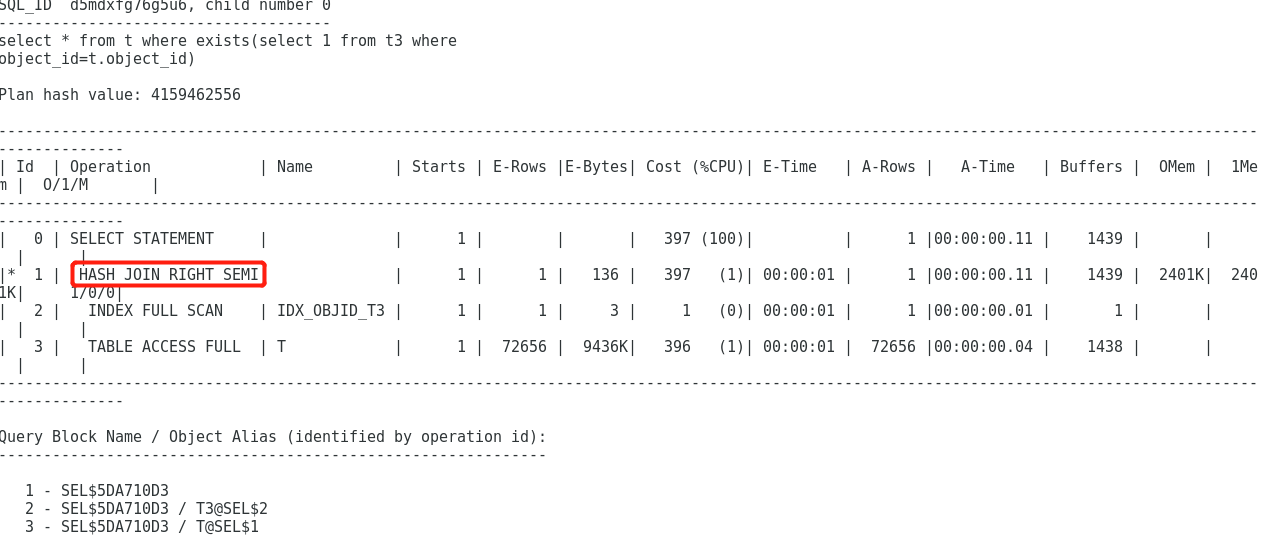
看到没有，Oracle发现如果再那大表，也就是这里的t表做驱动表，这个行为非常蠢，但是半连接的驱动表又是固定的，怎么办呢？Oracle选择不用半连接，在保证结果集的正确性下做了内连接。

我们现在给t3的连接列object\_id加上索引看看，这里把sql语句补充一下：

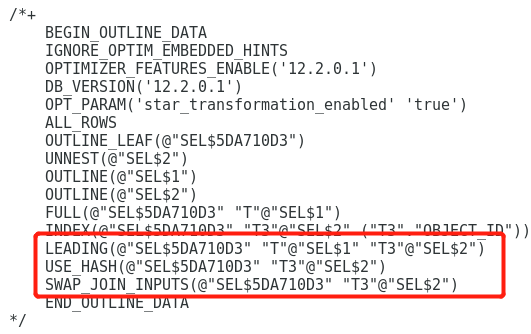




Oracle居然还是不走半连接，只是用index full scan代替了table full scan，那没办法，要把Oracle逼上绝路，我们把大表上的索引删了，听说你喜欢走索引。我们再来看：

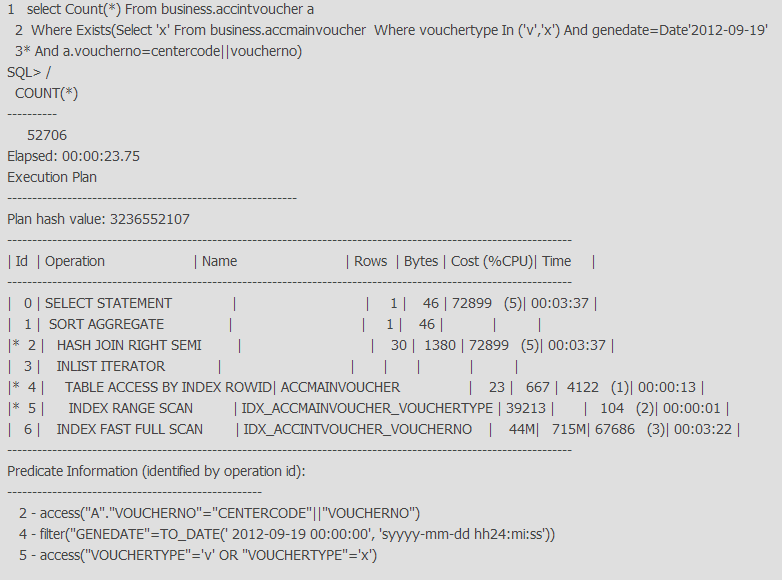


这次总算老老实实的走了半连接，其实我们从逻辑读看出来，这样做并不高效，因为t表是被驱动表，被驱动表上有索引才高效，而驱动表的结果集又只有一行，这里走NL就非常合适，所以Oracle的判断是对的，但是为了实验出这个，没错！就是这个RIGHT SEMI，我们只能这样折腾。现在我们来看看，它到底走了个什么hint：



我们看到，oracle通过swap\_join\_inputs把驱动表换了一下，然后来走这个hash join right，这样又正好可以是半连接所以，后面加了个semi，其实这个跟外连接驱动表互换的道理是一样的，后面在外连接的时候再说。

这里再贴一个论坛看到的例子：



## 星型连接（Star join）

这个真没啥好说的，OLTP用不到，这里也就不说了，道理也不难，关键一点，一定要配合位图索引。