



# Maclean Liu的Oracle性能优化讲座 第一回-真正读懂Oracle SQL执行计划Execution Plan

刘相兵(Maclean Liu)  
liu.maclean@gmail.com



ORA-ALLSTARS  
红桌议会 QQ群:23549328

# About Me

- Email: [liu.maclean@gmail.com](mailto:liu.maclean@gmail.com)
- Blog: [www.askmaclean.com](http://www.askmaclean.com)
- Founder of Shanghai Oracle Users Group - SHOUG
- Over 7 years experience with Oracle RDBMS technology
- Over 8 years experience with Linux technology
- Presents for advanced Oracle topics: RAC, DataGuard, Performance Tuning and Oracle Internal.



# How to Find Maclean Liu?

+你 搜索 图片 地图 Play YouTube 新闻 Gmail 更多 ▾

Google askmaclean  

网页 图片 地图 更多 ▾ 搜索工具

找到约 35,400 条结果 (用时 0.19 秒)

[ORACLE数据库数据恢复、性能优化、故障诊断来问问...](#)  
[www.askmaclean.com/](http://www.askmaclean.com/) ▾  
14 小时前 - 国内Oracle技术的领跑者，独特视角解析Oracle数据库。擅长于Oracle数据库性能优化、数据恢复。  
[有含金量的Oracle技术讨论 - 关于我|了解Maclean Liu - About Me - 网站地图](#)

[ORACLE数据库数据恢复、性能优化、故障诊断来问问...](#)  
[t.askmaclean.com/](http://t.askmaclean.com/) ▾  
ORACLE数据库数据恢复、性能优化、故障诊断来问问  
MACLEAN-小 ...

[askmaclean 视频\\_播客\\_个人多媒体土豆网](#)  
[www.tudou.com/home/askmaclean](http://www.tudou.com/home/askmaclean) ▾  
[askmaclean](#)的个人主页.  
<http://www.tudou.com/home/askmaclean>. 复制链接 | 加入收藏.  
主页 · 推过的 · 上传的视频 · 编辑的豆单 · 日志 · 小组 ...

# 议程

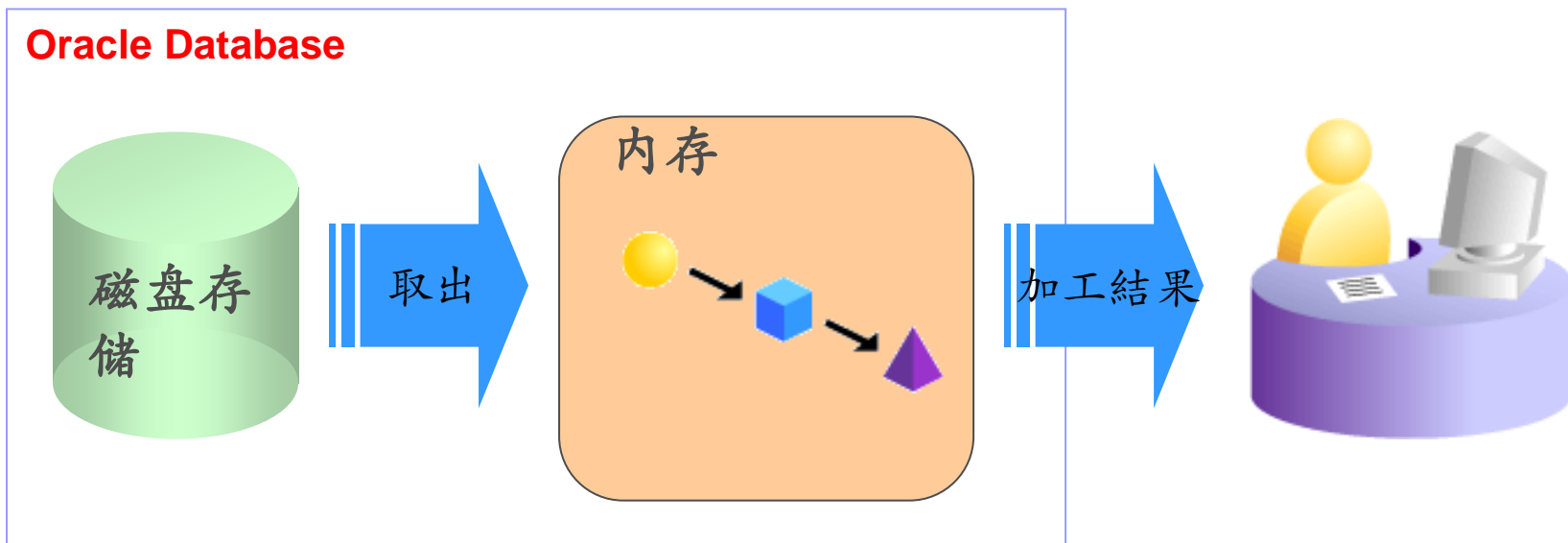
- 如何查看SQL执行计划
- 正确的执行计划执行顺序
- 通过示例来实践正确阅读执行计划的方法
- 介绍执行步骤的特性

# 读懂执行计划有什么用呢？

- 执行计划贯穿Oracle调优始终
- 了解执行计划的真实执行过程，将有助于优化
- 对于Oracle的原理理解有一定帮助
- 解决部分同学心中多年的疑惑
- 读懂执行计划，SQL调优的第一步

# 什么是SQL Execution Plan执行计划？

- SQL是声明型语言，她只说我要去哪里，但很少告诉你到底如何去？
- RDBMS所要做的是基于算法和现有统计信息计算最佳路径：
  - Access Path访问路径分析：访问数据是用TableScan还是index (FFS)
  - 对返回的行结果集做例如Join的进一步处理，以便返回行给客户端
- SQL语句的执行最终会落实为Oracle执行步骤的组合 =》【SQL执行计划】



# 执行计划示例

查询各部门员工信息的SQL，涉及到EMP和DEPT 2张示例表

```
SQL> set autotrace on
```

```
SQL> select d.dname,e.empno,e.ename,e.job  
       from emp e,dept d  
       where e.deptno=d.deptno;
```

连接方法

## Execution Plan

```
-----  
0  SELECT STATEMENT Optimizer=CHOOSE (Cost=5 Card=14 Bytes=392)  
1  0  HASH JOIN (Cost=5 Card=14 Bytes=392)  
2  1  TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=4 Bytes=44)  
3  1  TABLE ACCESS (FULL) OF 'EMP' (Cost=2 Card=14 Bytes=238)
```

EMP员工表的访问路径

DEPT 部门表的访问路径

# 查看执行计划的方法

1. Explain Plan For SQL
  - 不实际执行SQL语句，生成的计划未必是真实执行的计划
  - 必须要有plan\_table
2. SQLPLUS AUTOTRACE
  - 除set autotrace traceonly explain外均实际执行SQL，但仍未必是真实计划
  - 必须要有plan\_table
3. SQL TRACE
  - 需要启用10046或者SQL\_TRACE
  - 一般用tkprof看的更清楚些，当然10046里本身也有执行计划信息
4. V\$SQL和V\$SQL\_PLAN
  - 可以查询到多个子游标的计划信息了，但是看起来比较费劲
5. Enterprise Manager
  - 可以图形化显示执行计划，但并非所有环境有EM可用
6. 其他第三方工具
  - 注意 PL/SQL developer之类工具F5看到的执行计划未必是真实的



# 查看执行计划的方法:更靠谱的方法

## 最详细的执行计划信息收集

```
alter session set STATISTICS_LEVEL = ALL;    --不设置无法获得A-ROWS等信息
```

```
select * From DATA_SKEW_HB where source='Maclean Search';
```

```
select * from table(dbms_xplan.DISPLAY_CURSOR(null, null, 'ALLSTATS'));
```

```
PLAN_TABLE_OUTPUT
```

```
-----  
SQL_ID  dyysbpz0y6aw2, child number 0  
-----
```

```
select * From DATA_SKEW_HB where source='Maclean Search'
```

```
Plan hash value: 2604078056
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers
0	SELECT STATEMENT		1		2000	00:00:00.02	4378
* 1	TABLE ACCESS FULL	DATA_SKEW_HB	1	370	2000	00:00:00.02	4378

```
Predicate Information (identified by operation id):  
-----
```

```
1 - filter("SOURCE"='Maclean Search')
```

```
alter session set STATISTICS_LEVEL = TYPICAL;
```

E-Rows 是优化器评估返回的行数  
A-Rows 是实际执行时返回的行数

# 使用DBMS\_XPLAN包

```
select * from table(dbms_xplan....);
```

方法	使用	数据源
DISPLAY	Explain plan	Plan Table
DISPLAY_CURSOR	Real Plan	Shared pool中的游标缓存
DISPLAY_AWR	History	AWR仓库基表WRH\$_SQL_PLAN
DISPLAY_SQLSET	SQL Tuning Set	SQL Set视图

# DBMS\_XPLAN.DISPLAY\_CURSOR

三个输入值：

- SQL\_ID
- Child Number
- Format

如果child number置为NULL，则返回所有子游标的执行计划

```
SQL> select * from TABLE(dbms_xplan.display_cursor('fk641nh8gjzv',NULL,'ADVANCED +PEEKED_BINDS'));
```

PLAN\_TABLE\_OUTPUT

-----  
**SQL\_ID fk641nh8gjzv, child number 0**

Plan hash value: 2604078056

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT				1157 (100)	
* 1	TABLE ACCESS FULL	DATA_SKEW_HB	20042	352K	1157 (1)	00:00:14

-----  
**SQL\_ID fk641nh8gjzv, child number 1**

select /\*+ MACLEAN \*/ source from DATA\_SKEW\_HB where access\_no between  
20 and 200000

Plan hash value: 1444339438

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT				113 (100)	
1	TABLE ACCESS BY INDEX ROWID	DATA_SKEW_HB	20042	352K	113 (0)	00:00:02
* 2	INDEX RANGE SCAN	INDEX_NO	20042		1 (0)	00:00:01

# 获取SQL\_ID

- SQL\_ID是比HASH\_VALUE更好用的SQL语句标示符
- 如果输入NULL 则默认为之前运行的一条SQL，但注意要保持set serveroutput off，否则最后一句SQL将不是你运行的SQL
- 使用脚本查询SQL\_ID

```
select sql_id,sql_text from v$sql Where  
sql_text not like '%like%'  
and sql_text like '%$SQL%';
```

--\$SQL处填入你的SQL的文本

- 为了避免你的SQL和其他SQL混在一起，考虑增加一个注释 例如

```
Select /* MACLEAN_TEST_PLAN_1 */ * from MAC;
```

- 如果你之前执行过该语句，那么为了引发该语句的再次硬解析，对注释略作修改，例如上面的 PLAN\_1 改为PLAN\_2

# Child Number

- 父游标所在

```
Select * from v$SQLAREA where SQL_ID='YOUR_SQL_ID';
```

- 子游标：执行计划和优化环境

```
Select * from v$SQL where SQL_id='YOUR_SQL_ID';
```

计划：

```
Select * from v$SQL_PLAN where SQL_id='YOUR_SQL_ID';
```

优化环境：

```
Select * from v$SQL_OPTIMIZER_ENV where SQL_id='YOUR_SQL_ID';
```

- NULL意味着打印所有子游标

# Format 格式

- ALLSTATS  
IOSTATS + MEMSTATS
- IOSTATS 显示该游标累计执行的IO统计信息(Buffers, Reads)
- MEMSTATS 累计执行的PGA使用信息(Omem 1Mem Used-Mem)
- LAST  
仅显示最后一次执行的统计信息
- Advanced  
显示outline、Query Block Name、Column Projection等信息
- PEEKED\_BINDS  
打印解析时使用的绑定变量
- Typical 不打印PROJECTION, ALIAS

组合使用的方式如下，注意每个关键词后面要加空格

例如 'typical +peeked\_binds' => work  
'typical+peeked\_binds' => Error: format 'TYPICAL+peeked\_binds' not  
valid for DBMS\_XPLAN

# 推荐Format 格式

```
select * from table(dbms_xplan.display_cursor(null,null,'ADVANCED ALLSTATS LAST PEEKED_BINDS'));
```

```
alter session set statistics_level=ALL;
```

```
select * from table(dbms_xplan.display_cursor(null,null,'ADVANCED ALLSTATS LAST PEEKED_BINDS'));
```

Plan hash value: 196517798

Id	Operation	Name	Starts	E-Rows	E-Bytes	Cost	(%CPU)	E-Time	A-Rows	A-Time	Buffers	Reads	OMem	lMem	Used-Mem
0	SELECT STATEMENT		1			1231	(100)		2	00:00:00.40	4484	4482			
1	HASH GROUP BY		1	3	87	1231	(2)	00:00:15	2	00:00:00.40	4484	4482	9750K	3061K	1007K (0)
* 2	HASH JOIN		1	142K	4045K	1227	(1)	00:00:15	117K	00:00:00.38	4484	4482	1451K	1451K	1519K (0)
* 3	TABLE ACCESS FULL	TIMES	1	227	3632	16	(0)	00:00:01	182	00:00:00.01	54	53			
4	TABLE ACCESS FULL	SALES	1	918K	11M	1209	(1)	00:00:15	918K	00:00:00.13	4430	4429			

Query Block Name / Object Alias (identified by operation id):

```
1 - SEL$1
3 - SEL$1 / T@SEL$1
4 - SEL$1 / S@SEL$1
```

Outline Data

```
/*+
  BEGIN_OUTLINE_DATA
  IGNORE_OPTIM_EMBEDDED_HINTS
  OPTIMIZER_FEATURES_ENABLE('11.2.0.4')
*/
```

Predicate Information (identified by operation id):

```
2 - access("S"."TIME_ID"="T"."TIME_ID")
3 - filter(("T"."CALENDAR_QUARTER_DESC"='1999-12' OR "T"."CALENDAR_QUARTER_DESC"='2000-01' OR "T"."CALENDAR_QUARTER_DESC"='2000-02'))
```

Column Projection Information (identified by operation id):

```
1 - "T"."CALENDAR_QUARTER_DESC"[CHARACTER,7], SUM("S"."AMOUNT_SOLD") [22]
2 - (#keys=1) "T"."CALENDAR_QUARTER_DESC"[CHARACTER,7], "S"."AMOUNT_SOLD"[NUMBER,22]
3 - "T"."TIME_ID"[DATE,7], "T"."CALENDAR_QUARTER_DESC"[CHARACTER,7]
4 - "S"."TIME_ID"[DATE,7], "S"."AMOUNT_SOLD"[NUMBER,22]
```

# 收集all stats 的方法

- 默认系统只收集 SQL的typical statistics
- 为了了解A-Row , Cardinality Feedback等信息需要收集SQL执行的所有 stats
- Session级别 :

```
ALTER SESSION SET STATISTICS_LEVEL=ALL;
```

- 语句级别 使用HINT

```
select /*+ gather_plan_statistics*/ ...
```

收集all stats 有额外的负载 回设成默认值:

```
ALTER SESSION SET STATISTICS_LEVEL=TYPICAL;
```



# 重要的AWR视图

- V\$ACTIVE\_SESSION\_HISTORY
- V\$ 度量视图
- DBA\_HIST 视图:
  - DBA\_HIST\_ACTIVE\_SESS\_HISTORY
  - DBA\_HIST\_BASELINE
  - DBA\_HIST\_DATABASE\_INSTANCE
  - DBA\_HIST\_SNAPSHOT
  - DBA\_HIST\_SQL\_PLAN
  - DBA\_HIST\_WR\_CONTROL

# 查询AWR

- 检索在特定 SQL\_ID 下存储的所有执行计划。

```
select plan_table_output from table (dbms_xplan.display_awr('&sql_id',null,null,'ADVANCED +PEEKED_BINDS'));
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT				1626 (100)	
1	HASH GROUP BY		24	1824	1626 (1)	00:00:20
2	HASH JOIN		11808	876K	1625 (1)	00:00:20
3	TABLE ACCESS FULL	CUSTOMERS	2334	60684	396 (1)	00:00:05
4	HASH JOIN		35715	1743K	1229 (1)	00:00:15
5	MERGE JOIN CARTESIAN		227	6583	18 (0)	00:00:01
6	TABLE ACCESS FULL	CHANNELS	1	13	2 (0)	00:00:01
7	BUFFER SORT		227	3632	16 (0)	00:00:01
8	TABLE ACCESS FULL	TIMES	227	3632	16 (0)	00:00:01
9	TABLE ACCESS FULL	SALES	918K	18M	1209 (1)	00:00:15

Query Block Name / Object Alias (identified by operation id):

1 - SEL\$1

Note

- cardinality feedback used for this statement

- 显示包含 MAC 的所有语句的所有执行计划。

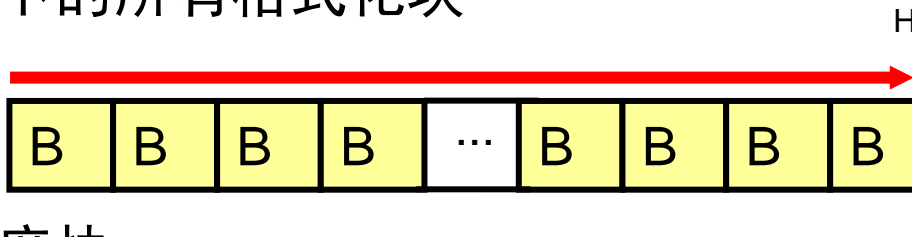
```
SELECT tf.* FROM DBA_HIST_SQLTEXT ht, table  
  (DBMS_XPLAN.DISPLAY_AWR(ht.sql_id,null, null, 'ALL' )) tf  
WHERE ht.sql_text like '%MAC%';
```

注意使用 SQL\_ID 确认已在 DBA\_HIST\_SQLTEXT 字典视图中捕获了该语句。如果查询没有返回行，则表明该语句尚未加载到 AWR 中。可以手动捕获 AWR 快照，而不必一直等待系统捕获下一张快照（每小时一次）。然后在 DBA\_HIST\_SQLTEXT 中检查是否已捕获快照：  
SQL> exec dbms\_workload\_repository.create\_snapshot;

# 优化程序步骤

结构	访问路径
表	1. Full Table Scan 2. Rowid Scan 3. Sample Table Scan
索引	4. Index Scan (Unique) 5. Index Scan (Range) 6. Index Scan (Full) 7. Index Scan (Fast Full) 8. Index Scan (Skip) 9. Index Scan (Index Join) 10. Using Bitmap Indexes 11. Combining Bitmap Indexes

# 全表扫描

- 如允许则执行多块读取  
(此处 `DB_FILE_MULTIBLOCK_READ_COUNT = 4`)
- 读取高水位标记以下的格式化块
- 可以过滤行
- 数据量很大时，
- 比索引范围扫描速度快

```
select * from emp where ename='King';
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)
0	SELECT STATEMENT		1	37	3 (0)
* 1	TABLE ACCESS FULL	EMP	1	37	3 (0)

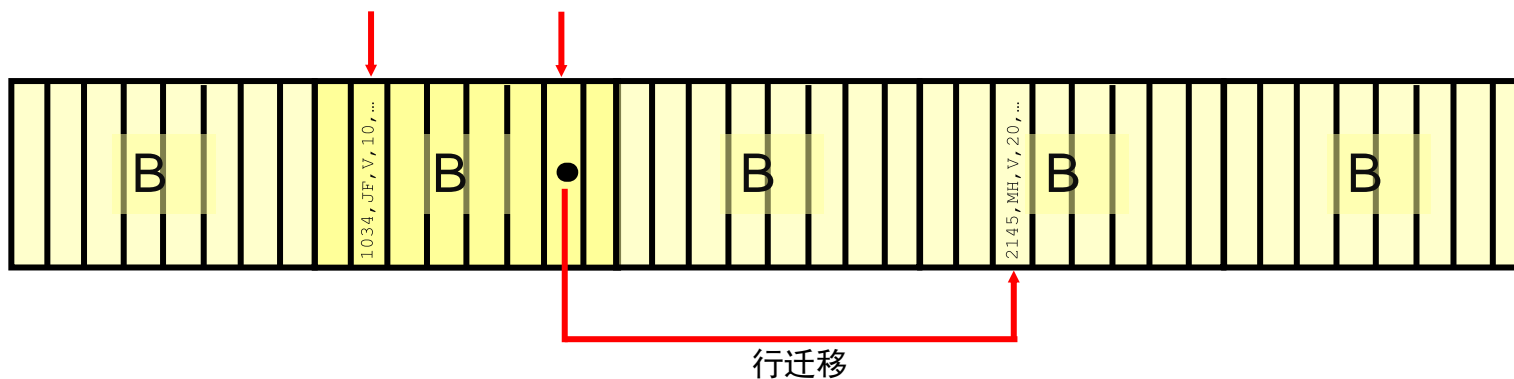
Predicate Information (identified by operation id):

- 1 - `filter("ENAME"='King')`

# 基于ROWID访问表

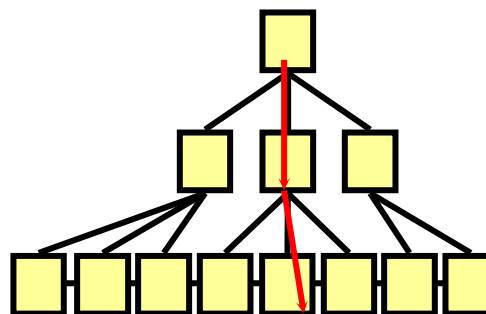
```
select * from scott.emp where rowid='AAQ+LAAEAAAAfAAJ';
```

Id	Operation	Name	Rows	Bytes	Cost
0	SELECT STATEMENT		1	37	1
1	TABLE ACCESS BY USER ROWID	EMP	1	37	1



# 索引唯一扫描

index UNIQUE Scan PK\_EMP



```
create unique index PK_EMP on EMP(empno)
```

```
select * from emp where empno = 9999;
```

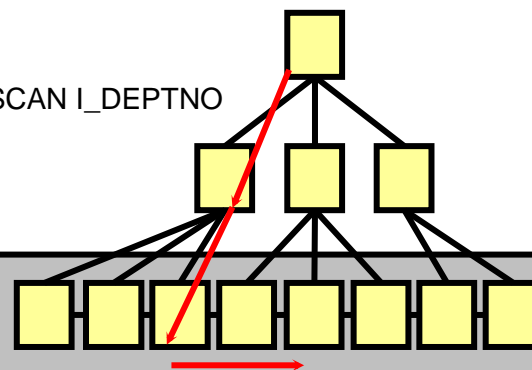
Id	Operation	Name	Rows	Bytes	Cost
0	SELECT STATEMENT		1	37	1
1	TABLE ACCESS BY INDEX ROWID	EMP	1	37	1
2	INDEX UNIQUE SCAN	PK_EMP	1		0

Predicate Information (identified by operation id):

2 - access("EMPNO"=9999)

# 索引范围扫描

Index Range SCAN I\_DEPTNO



```
create index I_DEPTNO on EMP(deptno);
```

```
select /*+ INDEX(EMP I_DEPTNO) */ *  
from emp where deptno = 10 and sal > 1000;
```

Id	Operation	Name	Rows	Bytes	Cost
0	SELECT STATEMENT		3	261	2
1	TABLE ACCESS BY INDEX ROWID	EMP	3	261	2
2	INDEX RANGE SCAN	I_DEPTNO	3		1

Predicate Information (identified by operation id):

- 1 - filter("SAL">1000)
- 2 - access("DEPTNO "=10)

# 一段执行计划及相关指标

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop
0	SELECT STATEMENT		607	46132	66 (10)	00:00:01		
1	HASH GROUP BY		607	46132	66 (10)	00:00:01		
* 2	HASH JOIN		2337	173K	65 (8)	00:00:01		
* 3	TABLE ACCESS FULL	TIMES	274	4384	18 (0)	00:00:01		
* 4	HASH JOIN		12456	729K	47 (11)	00:00:01		
* 5	TABLE ACCESS BY INDEX ROWID	CUSTOMERS	383	9958	26 (4)	00:00:01		
6	BITMAP CONVERSION TO ROWIDS							
7	BITMAP INDEX FULL SCAN	CUSTOMERS_GENDER_BIX						
8	NESTED LOOPS		229K	7627K	20 (15)	00:00:01		
9	NESTED LOOPS		229K	7627K	20 (15)	00:00:01		
* 10	TABLE ACCESS FULL	CHANNELS	1	13	3 (0)	00:00:01		
11	PARTITION RANGE ALL						1	28
12	BITMAP CONVERSION TO ROWIDS							
* 13	BITMAP INDEX SINGLE VALUE	SALES_CHANNEL_BIX					1	28
14	TABLE ACCESS BY LOCAL INDEX ROWID	SALES	229K	4710K	20 (15)	00:00:01	1	1

Predicate Information (identified by operation id):

```
2 - access("S"."TIME_ID"="T"."TIME_ID")
3 - filter("T"."CALENDAR_QUARTER_DESC"='1999-12' OR "T"."CALENDAR_QUARTER_DESC"='2000-01' OR
          "T"."CALENDAR_QUARTER_DESC"='2000-02')
4 - access("S"."CUST_ID"="C"."CUST_ID")
5 - filter("C"."CUST_STATE_PROVINCE"='FL')
10 - filter("CH"."CHANNEL_DESC"='Direct Sales')
13 - access("S"."CHANNEL_ID"="CH"."CHANNEL_ID")
```

- Id 分配给执行计划中每一个步骤的一个数字 每个步骤（执行计划中的行，或树中的节点）代表行源 (row source)。
- Operation 该步骤实施的内部操作名 id=0的operation一般是 SELECT/INSERT/UPDATE/DELETE Statement
- Name 该步骤操作的表或者索引名
- Rows CBO基于统计信息估计该操作将返回的行数
- Bytes CBO基于统计信息估计该操作将返回的字节数



# 一段执行计划及相关指标

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop
0	SELECT STATEMENT		607	46132	66 (10)	00:00:01		
1	HASH GROUP BY		607	46132	66 (10)	00:00:01		
* 2	HASH JOIN		2337	173K	65 (8)	00:00:01		
* 3	TABLE ACCESS FULL	TIMES	274	4384	18 (0)	00:00:01		
* 4	HASH JOIN		12456	729K	47 (11)	00:00:01		
* 5	TABLE ACCESS BY INDEX ROWID	CUSTOMERS	383	9958	26 (4)	00:00:01		
6	BITMAP CONVERSION TO ROWIDS							
7	BITMAP INDEX FULL SCAN	CUSTOMERS_GENDER_BIX						
8	NESTED LOOPS		229K	7627K	20 (15)	00:00:01		
9	NESTED LOOPS		229K	7627K	20 (15)	00:00:01		
* 10	TABLE ACCESS FULL	CHANNELS	1	13	3 (0)	00:00:01		
11	PARTITION RANGE ALL						1	28
12	BITMAP CONVERSION TO ROWIDS							
* 13	BITMAP INDEX SINGLE VALUE	SALES_CHANNEL_BIX					1	28
14	TABLE ACCESS BY LOCAL INDEX ROWID	SALES	229K	4710K	20 (15)	00:00:01	1	1

Predicate Information (identified by operation id):

```
2 - access("S"."TIME_ID"="T"."TIME_ID")
3 - filter("T"."CALENDAR_QUARTER_DESC"='1999-12' OR "T"."CALENDAR_QUARTER_DESC"='2000-01' OR
          "T"."CALENDAR_QUARTER_DESC"='2000-02')
4 - access("S"."CUST_ID"="C"."CUST_ID")
5 - filter("C"."CUST_STATE_PROVINCE"='FL')
10 - filter("CH"."CHANNEL_DESC"='Direct Sales')
13 - access("S"."CHANNEL_ID"="CH"."CHANNEL_ID")
```

- Cost 在默认启用CPU Costing的环境中  $Cost = IO\ Cost + CPU\ Cost$
- %CPU 代表CPU Cost占总的Cost的比例， $(Cost - Io\ Cost) / Cost$ ，对于Table Access FULL而言一般%CPU 很低
- Time CBO评估该操作将要消耗的时间，单位为秒
- 与CBO相关的参数Cost、Rows、Bytes、Time等当使用RBO优化器时全部为NULL
- Pstart 访问多个分区时的 起始分区
- Pstop 访问多个分区时的 停止分区

# 谓词信息 Predicate Information

Predicate Information (identified by operation id):

```
-----  
2 - access("S"."TIME_ID"="T"."TIME_ID")  
3 - filter("T"."CALENDAR_QUARTER_DESC"='1999-12' OR "T"."CALENDAR_QUARTER_DESC"='2000-01' OR  
    "T"."CALENDAR_QUARTER_DESC"='2000-02')  
4 - access("S"."CUST_ID"="C"."CUST_ID")  
5 - filter("C"."CUST_STATE_PROVINCE"='FL')  
10 - filter("CH"."CHANNEL_DESC"='Direct Sales')  
13 - access("S"."CHANNEL_ID"="CH"."CHANNEL_ID")
```

- Access 谓词多用于使用索引访问的场景。所谓Access即是不遍历全量的数据，而利用对应的查询条件或者约束来驱动访问索引。
- 过滤 谓词多用于无法使用索引访问的场景，例如步骤3 是在对整张表全表扫描的过程中，对于每一条记录做识别 看是否符合过滤相关的条件；但是主要filter过滤并不会真的是 物理读一个块然后就对里面的记录做过滤，仍会一次物理读取多个块，之后逻辑读这些块并做逻辑过滤。Filter 一般没有驱动作用。
- Column Projection Information 指的是该节点所感兴趣的字段/column的集合

# 谓词信息 Predicate Information: Projection

```
select mactab1.*, (select sum(id2) from mactab2 where  
mactab2.id=mactab1.id) s from mactab1 where mactab1.id=100
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT				2 (100)	
1	SORT AGGREGATE		1	8		
2	TABLE ACCESS BY INDEX ROWID	MACTAB2	1	8	2 (0)	00:00:01
* 3	INDEX RANGE SCAN	IDX_MACTAB2	1		1 (0)	00:00:01
4	TABLE ACCESS BY INDEX ROWID	MACTAB1	1	10	2 (0)	00:00:01
* 5	INDEX RANGE SCAN	IDX_MACTAB1	1		1 (0)	00:00:01

Query Block Name / Object Alias (identified by operation id):

```
1 - SEL$2  
2 - SEL$2 / MACTAB2@SEL$2  
3 - SEL$2 / MACTAB2@SEL$2  
4 - SEL$1 / MACTAB1@SEL$1  
5 - SEL$1 / MACTAB1@SEL$1
```

Predicate Information (identified by operation id):

```
3 - access("MACTAB2"."ID"=:B1)  
5 - access("MACTAB1"."ID"=100)
```

Column Projection Information (identified by operation id):

```
1 - (#keys=0) SUM("ID2") [22]  
2 - "ID2" [NUMBER,22]  
3 - "MACTAB2".ROWID[ROWID,10]  
4 - "MACTAB1"."ID" [NUMBER,22], "MACTAB1"."C1" [VARCHAR2,50]  
5 - "MACTAB1".ROWID[ROWID,10], "MACTAB1"."ID" [NUMBER,22]
```

- Column Projection Information 指的是该节点所感兴趣的字段/column的集合
- 如INDEX RANGE SCAN IDX\_MACTAB2 这一节点感兴趣的是MACTAB2表的ROWID，这样其父节点才能通过INDEX ROWID 访问表数据块
- 而INDEX RANGE SCAN IDX\_MACTAB1 则同时对 “MACTAB1”.ROWID[ROWID,10] 和 “MACTAB1”. “ID” [NUMBER,22]感兴趣，因为其谓词中含有5 - access("MACTAB1"."ID"=100)

# Note 信息

Note

-----

- cardinality feedback used for this statement

说明该执行计划使用了11g 基数反馈新特性，该新特性的介绍见<http://www.askmaclean.com/archives/11g-new-feature-cardinality-feedback.html>

Note

-----

- SQL plan baseline SQL\_PLAN\_6w2j9bx7xvu6h5268a54a used for this statement

说明该执行计划使用了11g SPM新特性，该新特性介绍见 <http://www.askmaclean.com/archives/plan-baseline-selection.html>

Note

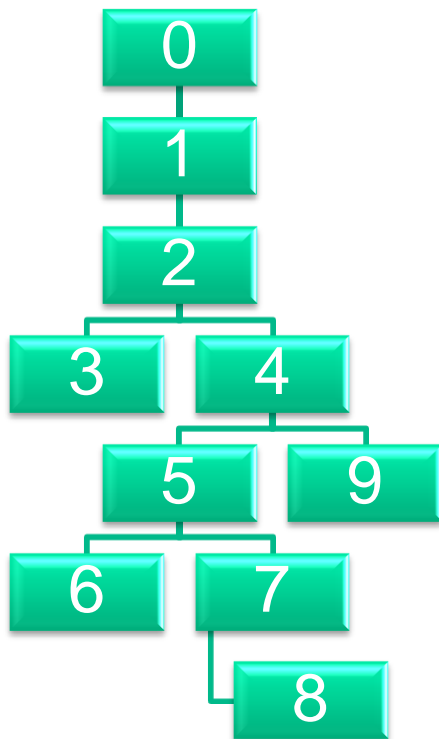
-----

- SQL profile "SYS\_SQLPROF\_012ad8267d9c0000" used FOR this statement

说明该执行计划使用了SQL profile特性，该新特性介绍见 <http://www.askmaclean.com/archives/how-to-validate-sql-profile-performance.html>

# 父节点与子节点树形图展示

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1288	97888	1626 (1)	00:00:20
1	HASH GROUP BY		1288	97888	1626 (1)	00:00:20
* 2	HASH JOIN		11808	876K	1625 (1)	00:00:20
* 3	TABLE ACCESS FULL	CUSTOMERS	2334	60684	396 (1)	00:00:05
* 4	HASH JOIN		35715	1743K	1229 (1)	00:00:15
5	MERGE JOIN CARTESIAN		227	6583	18 (0)	00:00:01
* 6	TABLE ACCESS FULL	CHANNELS	1	13	2 (0)	00:00:01
7	BUFFER SORT		227	3632	16 (0)	00:00:01
* 8	TABLE ACCESS FULL	TIMES	227	3632	16 (0)	00:00:01
9	TABLE ACCESS FULL	SALES	918K	18M	1209 (1)	00:00:15



在 PLAN\_TABLE 和 V\$SQL\_PLAN 中，用于检索树结构的重要元素是 ID、PARENT\_ID 和 POSITION 列。在跟踪文件中，这些列分别对应于 id、pid 和 pos 字段。

读取执行计划的一种方法是将其转换为以树结构表示的图。您可以从顶部的 id=1 开始，这是树的根节点。接下来，必须查找为根节点提供数据的操作。这由 parent\_id 或 pid 的值为 1 的操作完成。

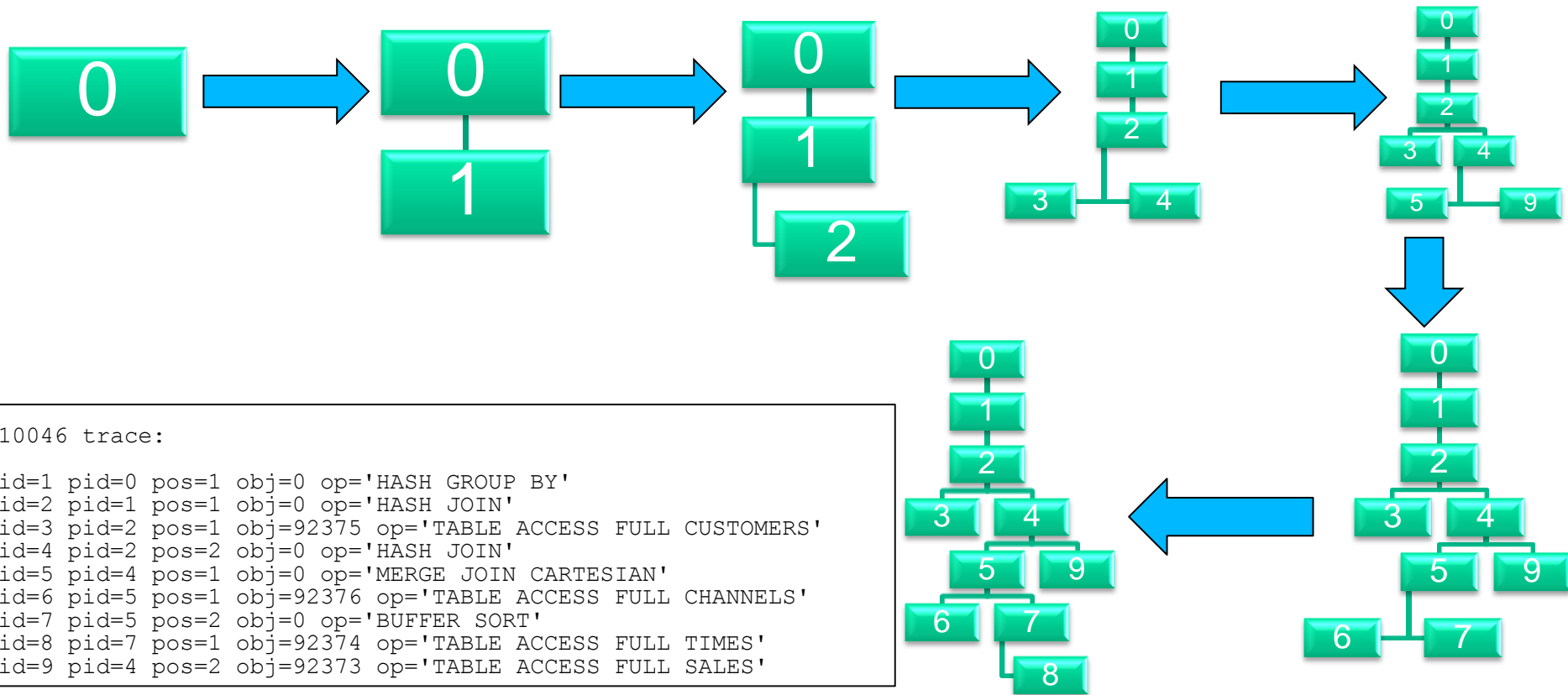
10046 trace:

```
id=1 pid=0 pos=1 obj=0 op='HASH GROUP BY'
id=2 pid=1 pos=1 obj=0 op='HASH JOIN'
id=3 pid=2 pos=1 obj=92375 op='TABLE ACCESS FULL CUSTOMERS'
id=4 pid=2 pos=2 obj=0 op='HASH JOIN'
id=5 pid=4 pos=1 obj=0 op='MERGE JOIN CARTESIAN'
id=6 pid=5 pos=1 obj=92376 op='TABLE ACCESS FULL CHANNELS'
id=7 pid=5 pos=2 obj=0 op='BUFFER SORT'
id=8 pid=7 pos=1 obj=92374 op='TABLE ACCESS FULL TIMES'
id=9 pid=4 pos=2 obj=92373 op='TABLE ACCESS FULL SALES'
```

# 如何画执行计划树形图？

绘制计划树形图的步骤：

1. 找出ID最小的节点，置于顶部
2. 查找父ID等于此值的节点
3. 按照Position从小到大按照从左到右的顺序，将节点置于父节点之下
4. 对于新放置的这一层节点重复 2-3的步骤，直到所有节点都被放置



# 隐藏着的执行计划指标:Parent\_ID

- 执行计划的步骤并不按编号顺序执行。步骤之间存在父子关系。
- PARENT\_ID 本节点的父节点，父节点将会处理本节点的输出
- 显示解释计划时会缩进子节点，表明它们是其上父节点的子节点。
- 一般来**在本节点最接近的上方比本节点缩进左移**的是本节点的父节点
- 如下面的例子中 INDEX Range Scan节点的父节点是Sort AGGREGATE

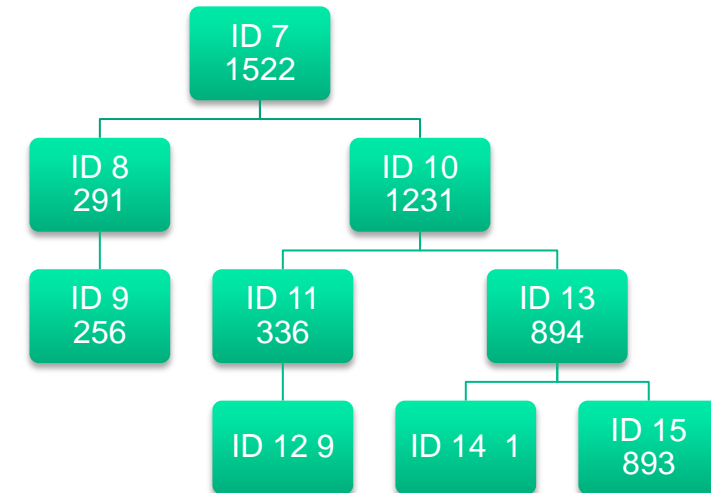
```
SQL> select id,parent_id,operation||options from v$sql_plan where sql_id='5yxwh66bpvjyb';
```

ID	PARENT_ID	OPERATION  OPTIONS
0		SELECT STATEMENT
1	0	SORT AGGREGATE
2	1	INDEX RANGE SCAN

Operation	Object	Order	Rows	B
▼ SELECT STATEMENT		3		
<缩进-----> ▼ SORT AGGREGATE		2	1	
<缩进-----> INDEX RANGE SCAN	<u>INDEX_NO</u>	1	20	

# 父节点的成本一般等于其子节点成本相加

Id	Operation	Name	Cost	(%CPU)
0	SELECT STATEMENT		1566	(100)
1	SORT AGGREGATE			
* 2	FILTER			
* 3	HASH JOIN RIGHT OUTER		1560	(1)
4	INDEX FAST FULL SCAN	I_HH_OBJ#_INTCOL#	36	(0)
* 5	HASH JOIN RIGHT OUTER		1523	(1)
6	INDEX FULL SCAN	I_USER2	1	(0)
* 7	HASH JOIN RIGHT OUTER		1522	(1)
8	TABLE ACCESS BY INDEX ROWID	OBJ\$	291	(0)
* 9	INDEX SKIP SCAN	I_OBJ1	256	(0)
* 10	HASH JOIN RIGHT OUTER		1231	(1)
11	TABLE ACCESS BY INDEX ROWID	COLTYPE\$	336	(0)
12	INDEX FULL SCAN	I_COLTYPE2	9	(0)
* 13	HASH JOIN		894	(1)
14	INDEX FULL SCAN	I_USER2	1	(0)
* 15	HASH JOIN		893	(1)
* 16	TABLE ACCESS FULL	USER\$	3	(0)
* 17	HASH JOIN		890	(1)
18	INDEX FAST FULL SCAN	I_OBJ2	239	(0)
19	TABLE ACCESS FULL	COL\$	393	(1)
* 20	TABLE ACCESS CLUSTER	TAB\$	2	(0)
* 21	INDEX UNIQUE SCAN	I_OBJ#	1	(0)



ID 7 Cost = ID8 + ID 10

ID 8 Cost= 291

ID 10 Cost= ID 11 + ID 13

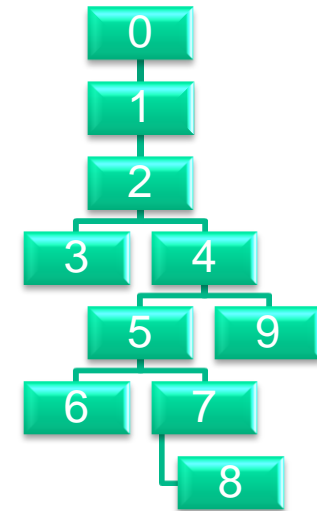
ID 11 Cost=336

ID 13=1+893= 894



# 隐藏着的执行计划指标: Depth

- Depth 步骤深度，如果 depth是3 则说明该步骤有3层父步骤
- 在看执行计划的时候 缩进层数代表了Depth
- 例如在右图中ID=6的深度为5, ID=9的深度为4

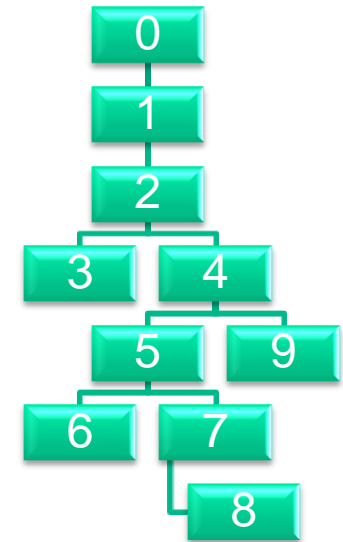


```
SQL> select id,parent_id,depth,operation||' '|| options,object_name from v$sql_plan where  
sql_id='d6jhhrsc63b22';
```

ID	PARENT_ID	DEPTH	OPERATION  ' '  OPTIONS	OBJECT_NAME
0		0	SELECT STATEMENT	
1	0	1	HASH GROUP BY	
2	1	2	HASH JOIN	
3	2	3	TABLE ACCESS FULL	CUSTOMERS
4	2	3	HASH JOIN	
5	4	4	MERGE JOIN CARTESIAN	
6	5	5	TABLE ACCESS FULL	CHANNELS
7	5	5	BUFFER SORT	
8	7	6	TABLE ACCESS FULL	TIMES
9	4	4	TABLE ACCESS FULL	SALES

# 隐藏着的执行计划指标: Position

- Position 代表拥有相同父节点Parent\_id的节点的处理顺序
  - 父节点之下节点的顺序指明相应级别中节点的执行顺序。
  - 如果两个步骤的缩进级别相同，则首先执行第一个步骤。
  - 在树形图中，树中每个级别最左端的叶节点最先执行。
- 
- ID=3 和 ID=4 拥有相同的父步骤 ID=2
  - ID=3 的Position为1，而ID=4的Position为2
  - 则ID=3的分支比 ID=4的分支先运行

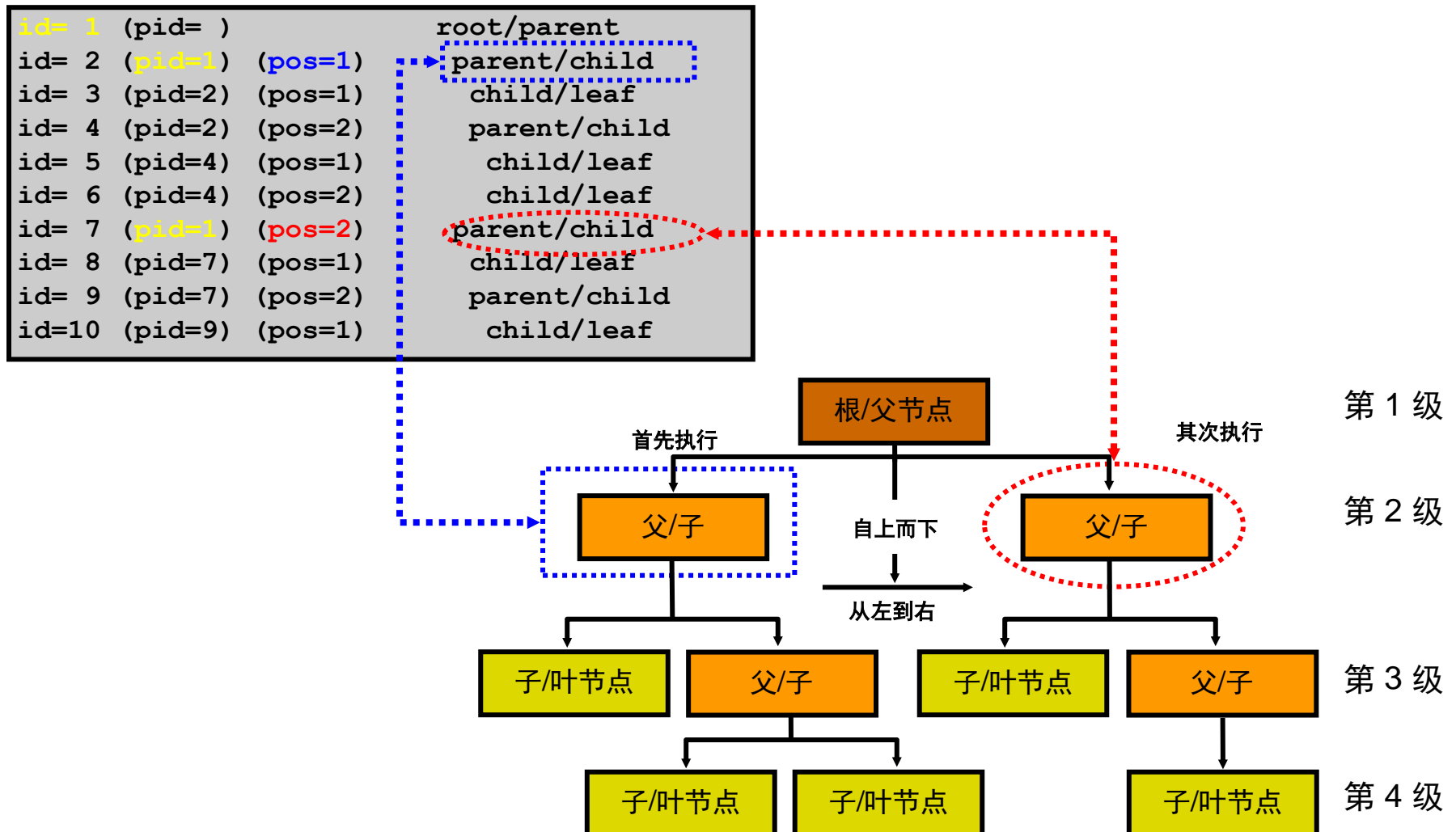


```
SQL> select id,parent_id,position,operation||' '|| options,object_name from v$sql_plan where  
sql_id='d6jhhrsc63b22';
```

ID	PARENT_ID	POSITION	OPERATION  ' '  OPTIONS	OBJECT_NAME
0		1626	SELECT STATEMENT	
1	0	1	HASH GROUP BY	
2	1	1	HASH JOIN	
3	2	1	TABLE ACCESS FULL	CUSTOMERS
4	2	2	HASH JOIN	
5	4	1	MERGE JOIN CARTESIAN	
6	5	1	TABLE ACCESS FULL	CHANNELS
7	5	2	BUFFER SORT	
8	7	1	TABLE ACCESS FULL	TIMES
9	4	2	TABLE ACCESS FULL	SALES

# 解释执行计划

将其转换为树状结构。



# 错误的执行顺序看法！

国内对于执行计划执行顺序有一种错误的说法：

最右最上最先执行的原则

## 三、执行计划层次关系

When looking at a plan, the rightmost (ie most indented) uppermost operation is the first thing that is executed. --采用最右最上最先执行的原则看层次关系，在同一级如果某个动作没有子ID就先执行

...

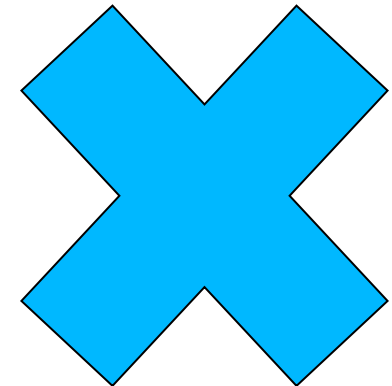
### (3) 执行计划层次关系

采用最右最上最先执行的原则

例子：

```
0      SELECT STATEMENT Optimizer=CHOOSE
1  0    MERGE JOIN
2    1  SORT (JOIN)
3      2  NESTED LOOPS
4        3    TABLE ACCESS (FULL) OF 'B'
5          3    TABLE ACCESS (BY INDEX ROWID) OF 'A'
6            5      INDEX (RANGE SCAN) OF 'INX_COL12A' (NON-UNIQUE)
7          1  SORT (JOIN)
8        7    TABLE ACCESS (FULL) OF 'C'
```

看执行计划的第3列，即字母部分，每列值的左面有空格作为缩进字符。在该列值左边的空格越多，说明该列值的缩进越多，该列值也越靠右。如上面的执行计划所示：第一列值为6的行的缩进最多，即该行最靠右；第一列值为4、5的行的缩进一样，其靠右的程度也一样，但是第一列值为4的行比第一列值为5的行靠上；谈论上下关系时，只对连续的、缩进一致的行有效。



# 错误的执行顺序看法！

我们来验证这种说法，按照最右最上原则 第一步该执行TABLE ACCESS FULL TIMES,因为它是唯一缩进最大Depth=6的节点

```
SELECT c.cust_city,
       t.calendar_quarter_desc,
       SUM(s.amount_sold) sales_amount
FROM   sales s, times t, customers c, channels ch
WHERE  s.time_id = t.time_id
      AND s.cust_id = c.cust_id
      AND s.channel_id = ch.channel_id
      AND c.cust_state_province = 'FL'
      AND ch.channel_desc = 'Direct Sales'
      AND t.calendar_quarter_desc IN ('2000-01', '2000-02', '1999-12')
GROUP BY c.cust_city, t.calendar_quarter_desc;
```

Id	Operation	Name	Rows	Bytes	Cost	(%CPU)	Time
0	SELECT STATEMENT		1288	97888	1626	(1)	00:00:20
1	HASH GROUP BY		1288	97888	1626	(1)	00:00:20
* 2	HASH JOIN		11808	876K	1625	(1)	00:00:20
* 3	TABLE ACCESS FULL	CUSTOMERS	2334	60684	396	(1)	00:00:05
* 4	HASH JOIN		35715	1743K	1229	(1)	00:00:15
5	MERGE JOIN CARTESIAN		227	6583	18	(0)	00:00:01
* 6	TABLE ACCESS FULL	CHANNELS	1	13	2	(0)	00:00:01
7	BUFFER SORT		227	3632	16	(0)	00:00:01
* 8	TABLE ACCESS FULL	TIMES	227	3632	16	(0)	00:00:01
9	TABLE ACCESS FULL	SALES	918K	18M	1209	(1)	00:00:15

# 验证执行顺序

我们可以通过10046 trace来跟踪Oracle，如果首先读取的是Times表则最右最上原则是正确的，否则显然不正确

```
alter system flush buffer cache;
alter session set events '10046 trace name context forever,level 8';
RUN TEST SQL
```

10046 trace:

nam='Disk file operations I/O'	obj#=92375
nam='db file sequential read'	obj#=92375
nam='db file sequential read'	obj#=92376
nam='db file sequential read'	obj#=92374
nam='direct path read'	obj#=92373

```
SQL> select object_id,object_name from dba_objects where object_id in
( 92375,92376 ,92374,92373 );
```

OBJECT_ID	OBJECT_NAME	
92375	CUSTOMERS	➔ ID=3
92376	CHANNELS	➔ ID=6
92374	TIMES	➔ ID=8
92373	SALES	➔ ID=9

通过10046 的obj#信息可以看到 最先读取的是Customers表 而非TIMES表

**通过上述验证 我们可以知道 最右最上最先执行的说法是错误的！**

**Top RightMost不是指 最右最上，这个错误的观念可能来源于对文档的误读**

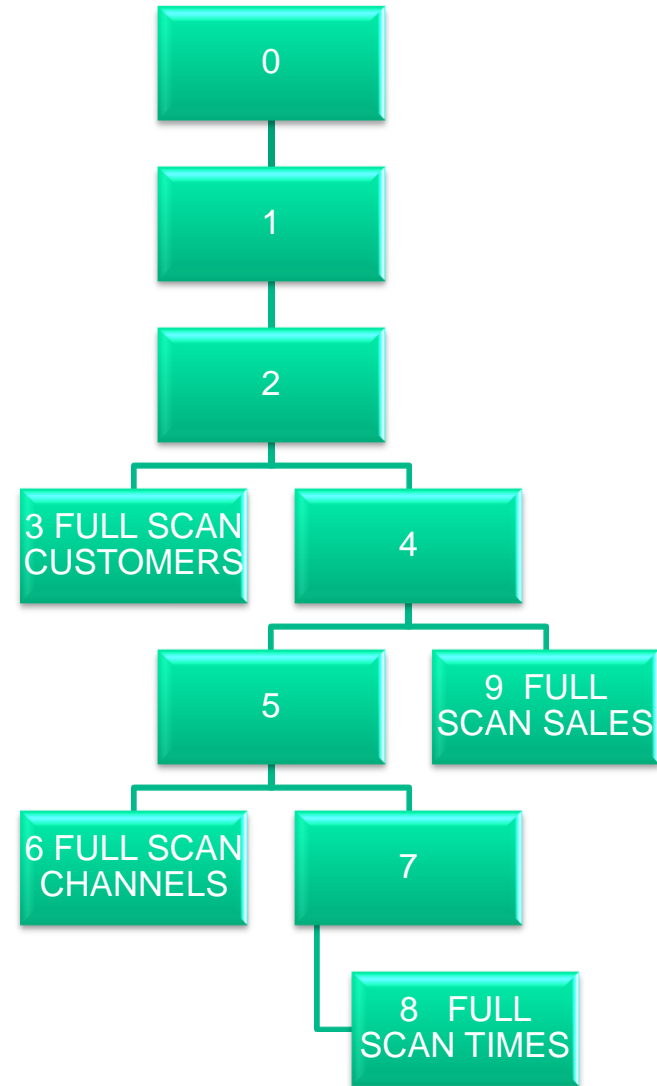
# 正确的计划树形图解析顺序

树形图的解析过程：

1. 从顶部开始。
2. 在树中向左下移，直至到达左节点(没有子节点的节点)。首先执行此节点。
3. 查看此行源的同级行源。接下来执行这些行源。
4. 执行子行源后，接着执行父行源。
5. 完成此父行源及其子行源后，在树中向上退一级，查看相应父行源的同级行源和父行源。按前述方式执行。
6. 在树中不断上移，直至用完所有行源为止。

如右图例： 执行顺序 3 6 8 7 5 9 4 2 1

1. 左下移动 3节点没有子节点，优先执行3
2. 遍历3的同级别行源4，左下移动 执行6
3. 之后遍历6的同级行源7，执行8
4. 执行7
5. 执行5
6. 执行9
7. 执行4
8. 执行2
9. 执行1



# 正确的执行计划的解析顺序：

执行计划的解析过程：

1. 从顶部开始
2. 在行源中向下移动，直至找到一个生成数据且不依赖于其他数据源的行源，这是起始节点
3. 查看此行源的同级行源。按照从上到下顺序来执行这些行源。
4. 执行子行源后，接着执行父行源。
5. 完成此父行源及其子行源后，在树中向上退一级，查看相应父行源的同级行源和父行源。按前述方式执行。
6. 在计划中不断上移，直至用完所有行源为止。

Id	Operation	Name	Rows	Bytes	Cost	(%CPU)	Time
0	SELECT STATEMENT		1288	97888	1626	(1)	00:00:20
1	HASH GROUP BY		1288	97888	1626	(1)	00:00:20
* 2	HASH JOIN		11808	876K	1625	(1)	00:00:20
* 3	TABLE ACCESS FULL	CUSTOMERS	2334	60684	396	(1)	00:00:05
* 4	HASH JOIN		35715	1743K	1229	(1)	00:00:15
5	MERGE JOIN CARTESIAN		227	6583	18	(0)	00:00:01
* 6	TABLE ACCESS FULL	CHANNELS	1	13	2	(0)	00:00:01
7	BUFFER SORT		227	3632	16	(0)	00:00:01
* 8	TABLE ACCESS FULL	TIMES	227	3632	16	(0)	00:00:01
9	TABLE ACCESS FULL	SALES	918K	18M	1209	(1)	00:00:15



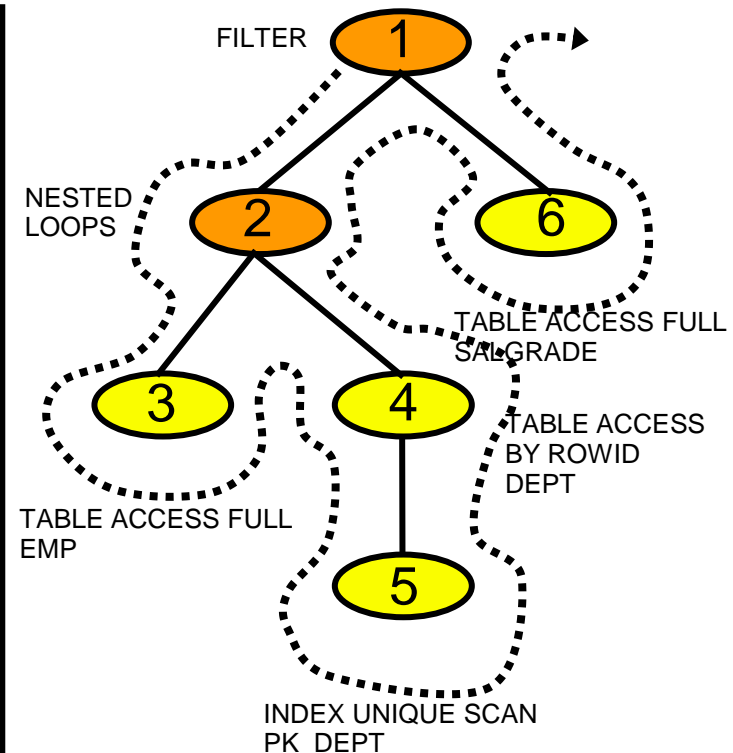
# 读懂Oracle执行计划，示例1

```
SELECT /*+ RULE */ ename,job,sal,dname
FROM emp,dept
WHERE dept.deptno=emp.deptno and not exists(SELECT *
      FROM salgrade
      WHERE emp.sal between losal and hisal);
```

Id	Operation	Name
0	SELECT STATEMENT	
* 1	FILTER	
2	NESTED LOOPS	
3	TABLE ACCESS FULL	EMP
4	TABLE ACCESS BY INDEX ROWID	DEPT
* 5	INDEX UNIQUE SCAN	PK_DEPT
* 6	TABLE ACCESS FULL	SALGRADE

Predicate Information (identified by operation id):

```
1 - filter( NOT EXISTS
  (SELECT 0 FROM "SALGRADE" "SALGRADE" WHERE
    "HISAL">=:B1 AND "LOSAL"<=:B2))
5 - access("DEPT"."DEPTNO"="EMP"."DEPTNO")
6 - filter("HISAL">=:B1 AND "LOSAL"<=:B2)
```



# 读懂Oracle执行计划，示例1

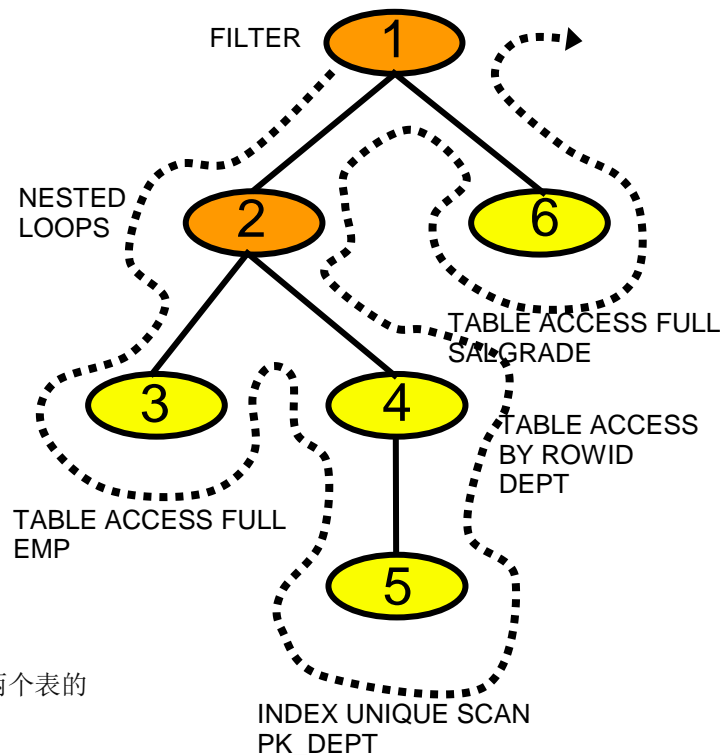
Id	Operation	Name
0	SELECT STATEMENT	
* 1	FILTER	
2	NESTED LOOPS	
3	TABLE ACCESS FULL	EMP
4	TABLE ACCESS BY INDEX ROWID	DEPT
* 5	INDEX UNIQUE SCAN	PK_DEPT
* 6	TABLE ACCESS FULL	SALGRADE

Predicate Information (identified by operation id):

1 - filter( NOT EXISTS  
(SELECT 0 FROM "SALGRADE" "SALGRADE" WHERE  
"HISAL">=:B1 AND "LOSAL"<=:B2))

5 - access("DEPT"."DEPTNO"="EMP"."DEPTNO")

6 - filter("HISAL">=:B1 AND "LOSAL"<=:B2)



此查询尝试查找薪金不在薪金等级表中的薪金范围内的雇员。此查询包含一条检索两个表的 SELECT 语句，同时包含一个子查询，子查询根据另一个表查找薪金等级。

3 - 5 - 4 - 2 - 6 - 1:

- 3: 此计划首先对 EMP 执行全表扫描 (ID=3)。
- 5: 这些行传递回控制嵌套循环联接步骤 (ID=2)，此步骤在 ID=5 的操作中使用这些行在索引 PK\_DEPT 中查找行。
- 4: 在 ID=4 的操作中使用来自索引的 ROWID 在 DEPT 表中查找其它信息。
- 2: ID=2，即嵌套循环联接步骤，将执行到完成为止。
- 6: 在 ID=2 的操作用尽其行源后，ID=6 的操作（与 ID=2 的操作位于树的同一级别，二者是同级关系）对 SALGRADE 执行全表扫描。
- 1: 用于对来自 ID2 和 ID6 的行进行过滤。
- 注意：子项先于父项执行，因此，虽然联接结构必须在子项执行前设置，但子项被标记为首先执行。最简单的方式是按执行的完成顺序进行考虑，因此，对于 ID=2 处的 NESTED LOOPS 联接，只有其两个子项 {ID=3 和 ID=4（及其子项）} 完成执行后，才可以完成 ID=2 的操作。

# 读懂Oracle执行计划，示例2

```
alter session set statistics_level=ALL;

select /*+ RULE to make sure it reproduces 100% */ ename,job,sal,dname
from emp,dept where dept.deptno = emp.deptno and not exists
(select * from salgrade where emp.sal between losal and hisal);

set linesize 200 pagesize 1400
select * from table(dbms_xplan.display_cursor(null,null,'TYPICAL IOSTATS LAST'));

SQL_ID 4xb2say20cqg8, child number 0
-----
select /*+ RULE to make sure it reproduces 100% */ ename,job,sal,dname from
emp,dept where dept.deptno = emp.deptno and not exists (select * from salgrade
where emp.sal between losal and hisal)

Plan hash value: 1175760222

-----
| Id | Operation | Name | Starts | A-Rows | A-Time | Buffers |
-----
| 0 | SELECT STATEMENT | | 1 | 0 | 00:00:00.01 | 59 |
|* 1 | FILTER | | 1 | 0 | 00:00:00.01 | 59 |
| 2 | NESTED LOOPS | | 1 | 14 | 00:00:00.01 | 23 |
| 3 | TABLE ACCESS FULL | EMP | 1 | 14 | 00:00:00.01 | 7 |
| 4 | TABLE ACCESS BY INDEX ROWID | DEPT | 14 | 14 | 00:00:00.01 | 16 |
|* 5 | INDEX UNIQUE SCAN | PK_DEPT | 14 | 14 | 00:00:00.01 | 2 |
|* 6 | TABLE ACCESS FULL | SALGRADE | 12 | 12 | 00:00:00.01 | 36 |
-----

Predicate Information (identified by operation id):
-----

1 - filter( IS NULL)
5 - access("DEPT"."DEPTNO"="EMP"."DEPTNO")
6 - filter(("HISAL">=:B1 AND "LOSAL"<=:B2))
```

"A-Rows"对应于相应行源生成的行数。  
"Buffers"对应于行源执行的一致读取数。  
"Starts"指示处理相应操作的次数。

系统获取 EMP 表中每一行的 ENAME、SAL、JOB 和 DEPTNO。

此后，系统通过 DEPT 表的唯一索引 (PK\_DEPT) 访问该表，以便使用来自上一结果集的 DEPTNO 获得 DNAME。  
仔细观察统计信息将发现，EMP 表上的 TABLE ACCESS FULL 操作 (ID=3) 启动了一次。但是，ID5 和ID4 操作启动了 14 次；对每个 EMP 行执行一次。在 ID=2 的步骤中，系统获得了所有的 ENAME、SAL、JOB 和 DNAME。

此时，系统必须过滤出其薪金不在薪金等级表中的薪金范围内的雇员。为此，系统针对来自 ID2 的每一行，使用 FULL TABLE SCAN 操作访问 SALGRADE 表，以便检查雇员的薪金是否不在薪金范围内。在本例中，由于在运行时系统检查每个不同的薪金，而 EMP 表中有 12 个不同的薪金，因此此操作只需执行 12 次。

# 读懂Oracle执行计划，示例3

```
select /*+ USE_NL(d) use_nl(m) */ m.last_name as  
dept_manager  
  ,      d.department_name  
  ,      l.street_address  
from    hr.employees m  join  
        hr.departments d on (d.manager_id =  
m.employee_id)  
      natural join  
        hr.locations l  
where   l.city = 'Seattle';
```

ID	SELECT STATEMENT
0	SELECT STATEMENT
1 0	NESTED LOOPS
2 1	NESTED LOOPS
3 2	TABLE ACCESS BY INDEX ROWID LOCATIONS
4 3	INDEX RANGE SCAN LOC_CITY_IX
5 2	TABLE ACCESS BY INDEX ROWID DEPARTMENTS
6 5	INDEX RANGE SCAN DEPT_LOCATION_IX
7 1	TABLE ACCESS BY INDEX ROWID EMPLOYEES
8 7	INDEX UNIQUE SCAN EMP_EMP_ID_PK

此查询检索所在部门位于西雅图并有经理的雇员的姓名、地址和所在部门的名称。

出于格式方面的原因，此解释计划将 ID 置于第一列，将 PID 置于第二列。位置通过缩进反映出来。此执行计划显示存在两个嵌套循环联接操作。

按照前一示例中的步骤执行：

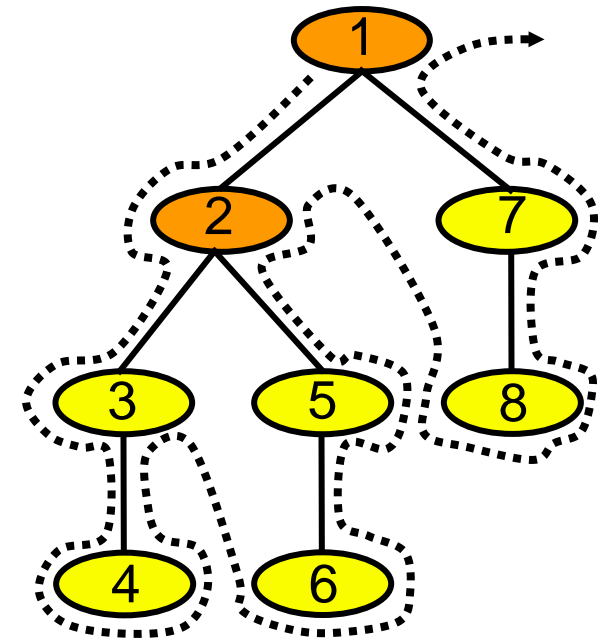
1. 从顶部开始。ID=0
2. 在行源中一直下移，直至找到一个生成数据且不使用任何数据的行源。在本例中，ID0、ID1、ID2 和 ID3 使用数据。ID=4 的操作是第一个不使用任何数据的行源。这便是起始行源。首先执行 ID=4 的操作。索引范围扫描生成 ROWID，ROWID 用于在 ID=3 的操作中查找 LOCATIONS 表。
3. 查看此行源的同级行源。接下来执行这些行源。与 ID=3 的操作处于同一级别的操作是 ID=5 的操作。ID=5 的节点有一个 ID=6 的子节点，后者先于前者执行。这将再执行一个生成 ROWID 的索引范围扫描，以便在 ID=5 的操作中查找 DEPARTMENTS 表。
4. 执行子操作后，接着执行父操作。接下来执行 ID=2 处的 NESTED LOOPS 联接，以便将底层数据联接起来。
5. 完成此父行源及其子行源后，在树中向上退一级，查看父行源的同级行源和父行源。按前述方式执行。在此计划中，与 ID=2 的操作处于同一级别的操作是 ID=7 的操作。此同级行源有一个 ID=8 的子行源，后者先于前者执行。索引范围扫描生成 ROWID，ROWID 用于在 ID=7 的操作中查找 EMPLOYEES 表。
6. 在计划中不断上移，直至用完所有行源为止。最后使用 ID1 处的 NESTED LOOPS 将这些数据联接起来，ID1 将结果传回 ID0。
7. 执行顺序是：4-3-6-5-2-8-7-1-0

此计划的完整说明如下：

首先，将 LOCATIONS 用作驱动表，使用 CITY 列上的索引，执行内部嵌套循环。这样操作是因为您仅搜索设在西雅图的部门。使用 LOCATION\_ID 联接列上的索引，将此结果与 DEPARTMENTS 表联接；第一个联接操作的结果是第二个嵌套循环联接的驱动行源。

第二个联接探测 EMPLOYEES 表的 EMPLOYEE\_ID 列上的索引。系统可以做到这一点，因为它从第一个联接了解到设在西雅图的部门的所有经理的下属雇员 ID。注意，由于基于主键，所以这是一个唯一扫描。

最后访问 EMPLOYEES 表，来检索姓氏。

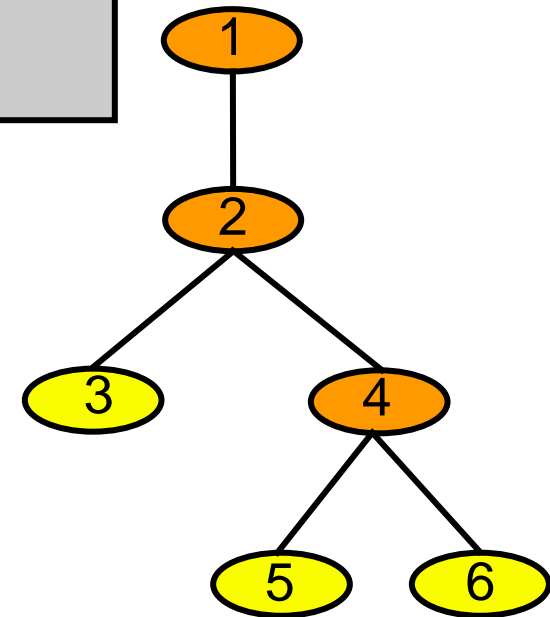


# 读懂Oracle执行计划，示例4

```
select /*+ ORDERED USE_HASH(b) SWAP_JOIN_INPUTS(c) */ max(a.i)
from t1 a, t2 b, t3 c
where a.i = b.i and a.i = c.i;
```

```
0  SELECT STATEMENT
1  SORT AGGREGATE
2 1  HASH JOIN
3 2  TABLE ACCESS FULL T3
4 2  HASH JOIN
5 4  TABLE ACCESS FULL T1
6 4  TABLE ACCESS FULL T2
```

<a href="#">Expand All</a>   <a href="#">Collapse All</a>		
Operation	Object	Order
SELECT STATEMENT		7
SORT AGGREGATE		6
HASH JOIN		5
TABLE ACCESS FULL	T3	1
HASH JOIN		4
TABLE ACCESS FULL	T1	2
TABLE ACCESS FULL	T2	3



此计划的解释过程如下：

1. 系统首先将 T3 表（操作 ID=3）以散列形式加载到内存中。
2. 然后将 T1 表（操作 ID=5）以散列形式加载到内存中。
3. 接下来开始扫描 T2 表（操作 ID=6）。
4. 系统从 T2 选取一行，并探测 T1 (T1.i=T2.i)。
5. 如果此行保留下来，系统将探测 T3 (T1.i=T3.i)。
6. 如果此行保留下来，系统将其发送到下一操作。
7. 系统输出上一结果集中的最大值。

即执行顺序为：3 - 5 - 6 - 4 - 2 - 1

联接顺序为：T1 - T2 - T3

# 利用Enterprise Manager高效理解执行计划

Enterprise Manager -> Search SQL -> Plan  
Order 字段显示了执行的先后顺序

使用Enterprise Manager可以最快速度理解执行计划，DBA值得拥有

Operation	Object	Order	Rows	Bytes	Cost	CPU (%)	Time	Query Block Name/Object Alias	Predicate	Filter
▼ SELECT STATEMENT		10			1,626	100				
▼ HASH GROUP BY		9	1,288	95.594K	1,626	1	0:0:20	SEL\$1		
▼ HASH JOIN		8	11,808	876.375K	1,625	1	0:0:20		"S"."CUST_ID"="C"."CUST_ID"	
TABLE ACCESS FULL	CUSTOMERS	1	2,334	59.262K	396	0	0:0:5	SEL\$1 / C@SEL\$1		"C"."CUST_ID"="S"."CUST_ID"
▼ HASH JOIN		7	35,715	1.703M	1,229	1	0:0:15		"S"."TIME_ID"="T"."TIME_ID"	
▼ MERGE JOIN CARTESIAN		5	227	6.429K	18	0	0:0:1			
TABLE ACCESS FULL	CHANNELS	2	1	13	2	0	0:0:1	SEL\$1 / CH@SEL\$1		"CH"."CHANNEL_ID"="S"."CHANNEL_ID"
▼ BUFFER SORT		4	227	3.547K	16	0	0:0:1			
TABLE ACCESS FULL	TIMES	3	227	3.547K	16	0	0:0:1	SEL\$1 / T@SEL\$1		("T"."TIME_ID"="S"."TIME_ID")
TABLE ACCESS FULL	SALES	6	918,843	18.402M	1,209	1	0:0:15	SEL\$1 / S@SEL\$1		

# 使用脚本获取执行计划顺序

由Maclean 开发的脚本SQL Execution Order

下载地址: <http://t.askmaclean.com/thread-3229-1-1.html>

无需Enterprise manager, 无需你了解算法, 只需要输入SQL\_ID和Child\_number即可获得执行计划的正确顺序

```
SQL> @execution_order
Enter value for macsqlid: d6jhhrsc63b22
old 1: select 'Input SQL_ID          : ',lower('&&macsqlid') macsqlid from dual
new 1: select 'Input SQL_ID          : ',lower('d6jhhrsc63b22') macsqlid from dual

Input SQL_ID          : d6jhhrsc63b22

Enter value for child_number: 0
old 1: select 'Input Child_number    : ',lower('&&child_number') child_number from dual
new 1: select 'Input Child_number    : ',lower('0') child_number from dual

OO                                OBJECT_NAME                                Execution Order
-----
SELECT STATEMENT                                10
HASH GROUP BY                                    9
HASH JOIN                                        8
TABLE ACCESS FULL                                TIMES                                1
HASH JOIN                                        7
MERGE JOIN CARTESIAN                            5
TABLE ACCESS FULL                                CHANNELS                            2
BUFFER SORT                                      4
TABLE ACCESS FULL                                CUSTOMERS                            3
TABLE ACCESS FULL                                SALES                                6
```

# 特殊的例外

```
select mactab1.*, (select sum(id2) from mactab2 where mactab2.id=mactab1.id) s
from mactab1
where mactab1.id=100;
```

Id	Operation	Name	Starts	E-Rows	E-Bytes	Cost (%CPU)	E-Time	A-Rows	A-Time	Buffers	Reads
0	SELECT STATEMENT		1			2 (100)		1	00:00:00.01	4	3
1	SORT AGGREGATE		1	1	8			1	00:00:00.01	3	3
2	TABLE ACCESS BY INDEX ROWID	mactab2	1	1	8	2 (0)	00:00:01	1	00:00:00.01	3	3
* 3	INDEX RANGE SCAN	IDX_mactab2	1	1		1 (0)	00:00:01	1	00:00:00.01	2	2
4	TABLE ACCESS BY INDEX ROWID	mactab1	1	1	10	2 (0)	00:00:01	1	00:00:00.01	4	3
* 5	INDEX RANGE SCAN	IDX_mactab1	1	1		1 (0)	00:00:01	1	00:00:00.01	3	2

Predicate Information (identified by operation id):

3 - access("MACTAB2"."ID"=:B1)

5 - access("MACTAB1"."ID"=100)

OBJECT\_ID OBJECT\_NAME

92618 mactab1

92619 mactab2

92620 IDX\_mactab1

92621 IDX\_mactab2

WAIT #140552674633184: nam='db file sequential read' ela= 45 file#=1 block#=109393 blocks=1 obj#=92620 tim=1381803018648264

WAIT #140552674633184: nam='db file sequential read' ela= 29 file#=1 block#=109361 blocks=1 obj#=92618 tim=1381803018648573

WAIT #140552674633184: nam='db file sequential read' ela= 19 file#=1 block#=109417 blocks=1 obj#=92621 tim=1381803018648804

WAIT #140552674633184: nam='db file sequential read' ela= 17 file#=1 block#=109418 blocks=1 obj#=92621 tim=1381803018648903

WAIT #140552674633184: nam='db file sequential read' ela= 18 file#=1 block#=109369 blocks=1 obj#=92619 tim=1381803018648986

FETCH #140552674633184:c=0,e=1216,p=6,cr=6,cu=0,mis=0,r=1,dep=0,og=1,plh=4037143566,tim=1381803018649065

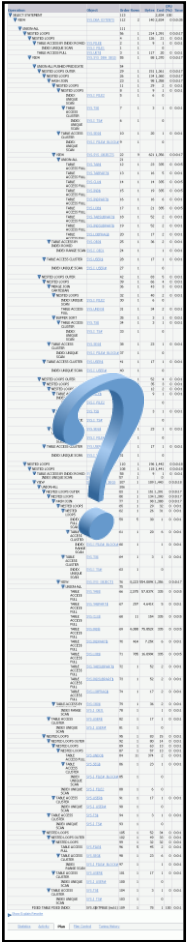
- 当表达式中存在子查询时可能出现例外
- 此例中优先执行的是 ID 5 而不是我们期望的ID 3
- 不保证没有其他意外现象

Operation	Object	Order	Rows	E
SELECT STATEMENT		6		
SORT AGGREGATE		3	1	
TABLE ACCESS BY INDEX ROWID	MACTAB2	2	1	
INDEX RANGE SCAN	IDX_MACTAB2	1	1	
TABLE ACCESS BY INDEX ROWID	MACTAB1	5	1	
INDEX RANGE SCAN	IDX_MACTAB1	4	1	



# 理解更为复杂的执行计划

```
SELECT owner , segment_name , segment_type
FROM dba_extents
WHERE file_id = 1
AND 123213 BETWEEN block_id AND block_id + blocks -1;
```



<a href="#">Expand All</a>   <a href="#">Collapse All</a>							
Operation	Object	Order	Rows	Bytes	Cost	CPU (%)	Time
SELECT STATEMENT		113			2,834	100	
VIEW	<a href="#">SYS.DBA_EXTENTS</a>	112	2	140	2,834	0	0:0:35
UNION-ALL		111					
NESTED LOOPS		56	1	214	1,391	0	0:0:17
NESTED LOOPS		110	1	196	1,442	0	0:0:18

使用缩进来折叠并且重点关注占用了大部分资源的操作。始终可以将计划折叠起来，以便于理解。右侧说明了这一点，从中可以看到同一计划折叠后的效果。如图中所示，在使用 Oracle Enterprise Manager 图形界面时，这易如反掌。可以清楚地看出，此计划是对两个分支执行 UNION ALL。

利用掌握的数据字典知识，认识到这两个分支对应于字典管理的表空间和本地管理的表空间。基于您对数据库的了解，您知道不存在字典管理的表空间。因此，如果存在问题，它必然在第二个分支上。为了确认，您必须查看每个行源的计划信息和执行统计信息，找出计划中占用大部分资源的部分。然后，您只需要展开要调查的分支（在这上面花费时间）。要使用此方法，您必须查看执行统计信息，这些信息通常可在 V\$SQL\_PLAN\_STATISTICS 或根据跟踪文件生成的 tkprof 报表中找到。

例如，tkprof 会累积每个父操作自身执行时间与其所有子操作执行总时间之和。

# 复查执行计划

在联机事务处理 (OLTP) 环境中优化 SQL 语句时，目标是将过滤性最强的表作为驱动表。这意味着，传递给下一步骤的行数较少。如果下一步骤执行联接，这意味着联接的行数较少。检查访问路径是否最佳。在检查优化程序执行计划时，请检查以下事项：

- 在计划中，驱动表具有最强的过滤性。
- 每个步骤的联接顺序都可保证返回给下一步的行数最少（即，联接顺序应使系统转到尚未使用的最强过滤器）。
- 就返回的行数而言，相应的联接方法是适合的例如，返回的行很多时，使用索引的嵌套循环联接可能不是最佳方法。
- 高效地使用视图。查看 SELECT 列表，确定访问的视图是否必需。
- 是否存在预料之外的笛卡尔积（即使对于小表，也是如此）。
- 高效地访问每个表：考虑 SQL 语句中的谓词和表的行数。查找可疑活动，例如对行数很多的表执行全表扫描（在 WHERE 子句中有谓词）。而对于小表，或根据返回的行数利用更好的联接方法（例如 hash\_join）时，全表扫描也许更有效。

如果这些条件中的任何一个都不是最佳的，请考虑调整 SQL 语句或表上的索引。

# SQL操作可以分为三类

```
1* select distinct KQLFXPL_OOPT from X$KQLFXPL  
KQLFXPL_OOPT
```

-----

AGGREGATE  
FULL  
BY INDEX ROWID  
OUTER  
FAST FULL SCAN  
RIGHT SEMI  
GROUP BY ROLLUP  
MULTI-COLUMN  
CREATE  
RIGHT OUTER  
ANTI SNA  
UNIQUE NOSORT  
SINGLE  
RIGHT ANTI  
FULL SCAN  
UNIQUE SCAN  
SAMPLE  
SKIP SCAN  
TO ROWIDS  
GROUP BY NOSORT  
NO FILTERING WITH SW (UNIQUE)  
RANGE SCAN DESCENDING  
SUBQUERY

大约有200多种执行操作，可以分配三类：

- 独立执行
- 无关组合
- 相关组合

# 阻塞操作与 非阻塞操作

操作按照其处理数据返回的方式可以分为：

- 阻塞操作  
处理集合数据，只有完成处理完符合要求才会返回数据  
例如：排序 SORT 必须要对数据集合完成整个排序才能返回第一行
- 非阻塞操作  
一次处理一行数据 或者一次处理几个块的数据，并立即返回  
例如：过滤 Filter 可以独立处理每一行数据

对于非阻塞操作 一次处理一行数据或者一次处理几个块的数据 马上可以和 其同级的节点结合或立即返回给父节点，父节点可以返回给上层节点，这样这一行数据马上能出现在用户客户端上 类似First\_Rows

对于阻塞操作 必须在本数据源处理完一定集合数据后才能返回

# 独立操作

- 这些操作如果有子节点，那么一般只有一个子节点
- 大多数类型的操作都是独立操作
- 独立操作的特点：

子节点一般都只运行一次

子节点反馈数据行给父节点

子节点在父节点之前运行，但存在2种例外

# 独立操作的例子

```
set linesize 200 pagesize 1400
select * from table(dbms_xplan.display_cursor(null,null,'TYPICAL IOSTATS LAST'));

select /*+ index(data_skew_hb ind_an ) */ ACCESS_DATE,count(*) from data_skew_hb where access_no between 10000 and 999999 and
source='Maclean Search' group by ACCESS_DATE;
```

Id	Operation	Name	Starts	E-Rows	E-Bytes	Cost (%CPU)	E-Time	A-Rows	A-Time	Buffers	Reads
0	SELECT STATEMENT		1			561K(100)		1	00:00:26.70	568K	11237
1	HASH GROUP BY		1	1	27	561K (1)	01:52:24	1	00:00:26.70	568K	11237
* 2	TABLE ACCESS BY INDEX ROWID	DATA_SKEW_HB	1	982K	25M	561K (1)	01:52:23	990K	00:00:23.76	568K	11237
* 3	INDEX RANGE SCAN	IND_AN	1	978K		6897 (1)	00:01:23	990K	00:00:08.04	6975	6975

Predicate Information (identified by operation id):

```
2 - filter("SOURCE"='Maclean Search')
3 - access("ACCESS_NO">=10000 AND "ACCESS_NO"<=999999)
```

- 3个步骤均是独立操作
- ID 1 和 ID 2均有子节点，所以不能先执行
- 执行将从ID 3开始



# 独立操作的例子

```
set linesize 200 pagesize 1400
select * from table(dbms_xplan.display_cursor(null,null,'TYPICAL IOSTATS LAST'));

select /*+ index(data_skew_hb ind_an ) */ ACCESS_DATE,count(*) from data_skew_hb where access_no between 10000 and 999999 and
source='Maclean Search' group by ACCESS_DATE;
```

Id	Operation	Name	Starts	E-Rows	E-Bytes	Cost (%CPU)	E-Time	A-Rows	A-Time	Buffers	Reads
0	SELECT STATEMENT		1			561K(100)		1	00:00:26.70	568K	11237
1	HASH GROUP BY		1	1	27	561K (1)	01:52:24	1	00:00:26.70	568K	11237
* 2	TABLE ACCESS BY INDEX ROWID	DATA_SKEW_HB	1	982K	25M	561K (1)	01:52:23	990K	00:00:23.76	568K	11237
* 3	INDEX RANGE SCAN	IND_AN	1	978K		6897 (1)	00:01:23	990K	00:00:08.04	6975	6975

Predicate Information (identified by operation id):

```
2 - filter("SOURCE"='Maclean Search')
3 - access("ACCESS_NO">=10000 AND "ACCESS_NO"<=999999)
```

- ID 3 范围扫描以"ACCESS\_NO">=10000 AND "ACCESS\_NO"<=999999驱动范围扫描DATA\_SKEW\_HB表的索引，并返回990K个rowid给Parent ID 2
- ID 2 使用rowid来访问表上的数据块，找到990K行记录(filter("SOURCE"='Maclean Search'))并返回给ID 1
- ID 1将990k行数据分组后返回1条记录



# 探究阻塞与非阻塞操作

```
alter system flush buffer_cache;
alter session set events '10046 trace name context forever,level 8';
select /*+ index(data_skew_hb ind_an ) */ ACCESS_DATE,count(*) from data_skew_hb where access_no between 10000 and 999999 and
source='Maclean Search' group by ACCESS_DATE;
```

```
SQL> select object_name,object_id from dba_objects where object_id in (100730,100729);
```

OBJECT_NAME	OBJECT_ID
DATA_SKEW_HB	100729
IND_AN	100730

```
WAIT #1: nam='db file sequential read' ela= 10 file#=4 block#=927604 blocks=1 obj#=100730 tim=1349329163290513
WAIT #1: nam='db file sequential read' ela= 10 file#=4 block#=930497 blocks=1 obj#=100730 tim=1349329163290566
WAIT #1: nam='db file sequential read' ela= 9 file#=4 block#=929864 blocks=1 obj#=100730 tim=1349329163290601
WAIT #1: nam='db file sequential read' ela= 10 file#=4 block#=929865 blocks=1 obj#=100730 tim=1349329163290655
WAIT #1: nam='db file parallel read' ela= 28 files=1 blocks=3 requests=3 obj#=100729 tim=1349329163299009
WAIT #1: nam='db file sequential read' ela= 11 file#=1 block#=447587 blocks=1 obj#=100729 tim=1349329163299071
WAIT #1: nam='db file sequential read' ela= 11 file#=1 block#=448467 blocks=1 obj#=100729 tim=1349329163299332
WAIT #1: nam='db file sequential read' ela= 11 file#=4 block#=929866 blocks=1 obj#=100730 tim=1349329163299411
WAIT #1: nam='db file sequential read' ela= 10 file#=1 block#=448548 blocks=1 obj#=100729 tim=1349329163299488
WAIT #1: nam='db file parallel read' ela= 16 files=1 blocks=2 requests=2 obj#=100729 tim=1349329163299636
WAIT #1: nam='db file sequential read' ela= 11 file#=4 block#=929867 blocks=1 obj#=100730 tim=1349329163299753
WAIT #1: nam='db file sequential read' ela= 10 file#=1 block#=448607 blocks=1 obj#=100729 tim=1349329163299801
WAIT #1: nam='db file sequential read' ela= 13 file#=1 block#=448669 blocks=1 obj#=100729 tim=1349329163299950
WAIT #1: nam='db file sequential read' ela= 11 file#=4 block#=929868 blocks=1 obj#=100730 tim=1349329163300079
WAIT #1: nam='db file sequential read' ela= 12 file#=1 block#=448703 blocks=1 obj#=100729 tim=1349329163300123
WAIT #1: nam='db file sequential read' ela= 11 file#=4 block#=929869 blocks=1 obj#=100730 tim=1349329163300279
WAIT #1: nam='db file sequential read' ela= 12 file#=1 block#=447588 blocks=1 obj#=100729 tim=1349329163300419
WAIT #1: nam='db file sequential read' ela= 8 file#=1 block#=444711 blocks=1 obj#=100729 tim=134932916330046
```

```
FETCH #1:c=1559763,e=1533141,p=11237,cr=568972,cu=0,mis=0,r=1,dep=0,og=1,tim=1349329164823517
```

通过OBJ#可以看到先对索引做了多次 'db file sequential read'，之后利用这几次获得的rowid去访问表 obj#=100729，访问表时是 'db file parallel read'，并过滤数据返回给ID1的 HASH GROUP BY

- ID 3 的操作不是基于行 而是基于数据块，获得一定量rowid后返回给上级行源，ID 3是非阻塞的
- ID 2 的操作也是基于数据块的，使用rowid访问一定表块后返回给上级行源，ID 2是非阻塞的
- ID 1 HASH GROUP BY是阻塞的，仅在完成HASH 分组后才能返回结果



# 独立操作的例外情况

对于独立操作而言，一般子节点要比父节点先执行  
但是也存在例外：

- 父节点可以决定是否完成子节点的操作
- 对于阻塞操作 父节点也没办法停止它
- 父节点可以决定是否执行子节点的操作

也就是说父节点可以在一定程度上控制子节点的运行

# 父节点停止子节点操作的例子: STOPKEY

```
select * from emp where rownum <= 10;
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		10	370	3 (0)	00:00:01
* 1	COUNT STOPKEY					
2	TABLE ACCESS FULL	EMP	10	370	3 (0)	00:00:01

Predicate Information (identified by operation id):

1 - filter(ROWNUM<=10)

- ID 1 父节点当ID 2的子节点返回10条记录后将停止 ID 2的操作

# 对于阻塞操作STOPKEY没用

```
SQL> select count(*) from emp;
```

```
COUNT(*)
```

```
-----  
14
```

```
select * from (select * from emp order by sal desc) where rownum < 10;
```

```
-----  
| Id | Operation | Name | Rows | Bytes | Cost (%CPU) | Time |  
-----  
| 0 | SELECT STATEMENT | | 9 | 783 | 4 (25) | 00:00:01 |  
|* 1 | COUNT STOPKEY | | | | | |  
| 2 | VIEW | | 14 | 1218 | 4 (25) | 00:00:01 |  
|* 3 | SORT ORDER BY STOPKEY | | 14 | 518 | 4 (25) | 00:00:01 |  
| 4 | TABLE ACCESS FULL | EMP | 14 | 518 | 3 (0) | 00:00:01 |  
-----
```

```
Predicate Information (identified by operation id):  
-----
```

```
1 - filter(ROWNUM<10)
```

```
3 - filter(ROWNUM<10)
```

- 阻塞操作无法被停止，因为他们需要完全处理后才能将第一行数据返回给上层
- 子节点ID 4 (FULL SCAN EMP)由于有order by的需求，所以停不下来

# 父节点可以控制子节点是否执行

```
select * from emp where job = 'MACLEAN' and 0=1;
```

Id	Operation	Name	Starts	E-Rows	E-Bytes	Cost (%CPU)	E-Time	A-Rows	A-Time
0	SELECT STATEMENT		1			1 (100)		0	00:00:00.01
* 1	FILTER		1					0	00:00:00.01
* 2	TABLE ACCESS FULL	EMP	0	3	111	2 (0)	00:00:01	0	00:00:00.01

Predicate Information (identified by operation id):

1 - filter(NULL IS NOT NULL)  
2 - filter("JOB"='MACLEAN')

- ID 2 的 FULL SCAN EMP 的Starts为0，说明该步骤没被执行过
- 0=1 为恒假，所以FILTER操作控制子节点一次都不运行

# 无关组合

- 特点为：有多个子节点，子节点一般均是独立执行

AND-EQUAL, BITMAP AND, BITMAP OR,  
BITMAP MINUS, CONCATENATION,  
CONNECT BY WITHOUT FILTERING,  
HASH JOIN, INTERSECTION,  
MERGE JOIN, MINUS,  
MULTI-TABLE INSERT, SOL MODEL,  
TEMP TABLE TRANSFORMATION,

UNION-ALL

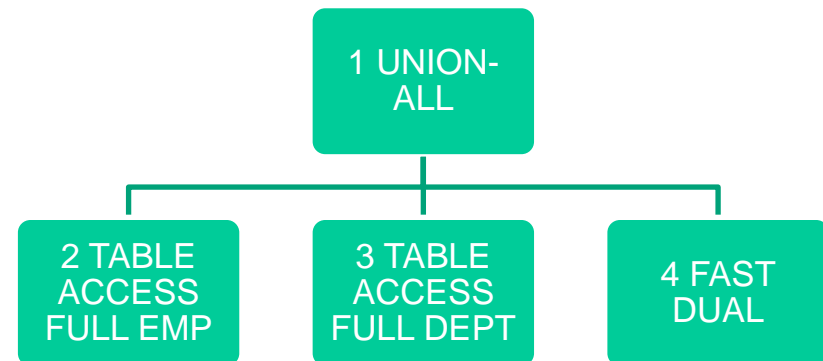
# 无关组合的特点

- 子节点于父节点之前运行
- 子节点按照ID 顺序执行
- 每个子节点必须完成后才能执行下一个子节点
- 子节点将返回数据给父节点

# 无关组合的例子1

```
select ename from emp
union all
select dname from dept
union all
select '%' from dual;
```

Id	Operation	Name	Starts	E-Rows	E-Bytes	Cost (%CPU)	E-Time	A-Rows	A-Time	Buffers
0	SELECT STATEMENT		1			6 (100)		19	00:00:00.01	8
1	UNION-ALL		1					19	00:00:00.01	8
2	TABLE ACCESS FULL	EMP	1	14	84	2 (0)	00:00:01	14	00:00:00.01	4
3	TABLE ACCESS FULL	DEPT	1	4	36	2 (0)	00:00:01	4	00:00:00.01	4
4	FAST DUAL		1	1		2 (0)	00:00:01	1	00:00:00.01	0



- ID 1 有三个子节点，其中ID 2的position最小，所以先执行ID 2
- ID 2 将14行数据返回给ID 1, ID 3开始执行
- ID 3 将4行数据返回给ID 1, ID 4开始执行
- ID 4 将1行数据返回给ID 1, ID 1构造一个19行的结果集返回给客户端

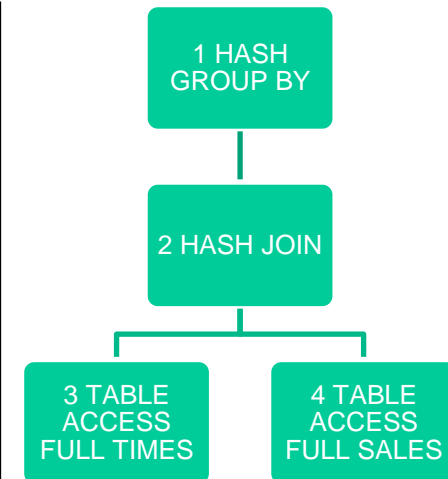
# 无关组合的例子2: HASH JOIN

```
SELECT t.calendar_quarter_desc,
       SUM(s.amount_sold) sales_amount
FROM sales s, times t
WHERE s.time_id = t.time_id
AND t.calendar_quarter_desc IN ('2000-01', '2000-02', '1999-12')
group by t.calendar_quarter_desc;
```

Id	Operation	Name	Starts	E-Rows	E-Bytes	Cost (%CPU)	E-Time	A-Rows	A-Time	Buffers	Reads
0	SELECT STATEMENT		1			1016 (100)		2	00:00:12.34	4490	4482
1	HASH GROUP BY		1	3	87	1016 (4)	00:00:13	2	00:00:12.34	4490	4482
* 2	HASH JOIN		1	194K	5517K	1006 (3)	00:00:13	117K	00:00:11.98	4490	4482
* 3	TABLE ACCESS FULL	TIMES	1	274	4384	13 (0)	00:00:01	182	00:00:00.01	55	53
4	TABLE ACCESS FULL	SALES	1	938K	11M	986 (2)	00:00:12	918K	00:00:03.68	4435	4429

Predicate Information (identified by operation id):

```
2 - access("S"."TIME_ID"="T"."TIME_ID")
3 - filter(("T"."CALENDAR_QUARTER_DESC"='1999-12' OR "T"."CALENDAR_QUARTER_DESC"='2000-01' OR
          "T"."CALENDAR_QUARTER_DESC"='2000-02'))
```



```
WAIT #1: nam='db file sequential read' ela= 29 file#=1 block#=437241 blocks=1 obj#=100607
WAIT #1: nam='db file scattered read' ela= 96 file#=1 block#=437242 blocks=7 obj#=100607
WAIT #1: nam='db file scattered read' ela= 67 file#=1 block#=441617 blocks=8 obj#=100607
WAIT #1: nam='db file scattered read' ela= 69 file#=1 block#=441625 blocks=8 obj#=100607
WAIT #1: nam='db file scattered read' ela= 36 file#=1 block#=441633 blocks=8 obj#=100607
WAIT #1: nam='db file scattered read' ela= 107 file#=1 block#=441641 blocks=5 obj#=100607
WAIT #1: nam='db file sequential read' ela= 27 file#=1 block#=80305 blocks=1 obj#=100605
WAIT #1: nam='db file scattered read' ela= 55 file#=1 block#=80306 blocks=7 obj#=100605
WAIT #1: nam='db file scattered read' ela= 48 file#=1 block#=80313 blocks=8 obj#=100605
WAIT #1: nam='db file scattered read' ela= 39 file#=1 block#=80321 blocks=8 obj#=100605
WAIT #1: nam='db file scattered read' ela= 34 file#=1 block#=437129 blocks=8 obj#=100605
WAIT #1: nam='db file scattered read' ela= 31 file#=1 block#=437137 blocks=8 obj#=100605
WAIT #1: nam='db file scattered read' ela= 33 file#=1 block#=437145 blocks=8 obj#=100605
WAIT #1: nam='db file scattered read' ela= 37 file#=1 block#=437153 blocks=8 obj#=100605
.....
```

- 通过10046 可以发现 先完成了ID 3 对TIMES的全表扫描后才开始对ID 4SALES表全表扫描
- ID 3返回182行给 ID 2, 并构造HASH TABLE
- 之后 ID 4返回918k行给ID 2
- ID 2 完成HASH JOIN后返回117k 数据到上层



## 无关组合的例子2: Hash Join Build Hash Table

```
select object_id,object_name from dba_objects where object_id in (100605,100607);

OBJECT_ID OBJECT_NAME
-----
100605 SALES
100607 TIMES

alter session set events '10046 trace name context forever,level 8: 10104 trace name
context forever, level 10';

10046 + 10104 HASH TRACE该SQL

kxhfSetPhase: phase=BUILD
WAIT #1: nam='db file sequential read' ela= 45 file#=1 block#=437241 blocks=1 obj#=100607
WAIT #1: nam='db file scattered read' ela= 107 file#=1 block#=437242 blocks=7 obj#=100607
WAIT #1: nam='db file scattered read' ela= 67 file#=1 block#=441609 blocks=8 obj#=100607
kxhfAddChunk: add chunk 0 (sz=32) to slot table
kxhfAddChunk: chunk 0 (lbs=0x7fd0574d7b08, slotTab=0x7fd0574d7cd0) successfully added
WAIT #1: nam='db file scattered read' ela= 68 file#=1 block#=441617 blocks=8 obj#=100607
WAIT #1: nam='db file scattered read' ela= 67 file#=1 block#=441625 blocks=8 obj#=100607
kxhfSetPhase: phase=PROBE_1
qerhjFetch: max build row length (mbl=24)
*** RowSrcId: 2 END OF BUILD (PHASE 1) ***
*** (continued) HASH JOIN BUILD HASH TABLE (PHASE 1) ***
### Hash table ###
# NOTE: The calculated number of rows in non-empty buckets may be smaller
#       than the true number.
Number of buckets with 0 rows:      133
Number of buckets with 1 rows:      82
Number of buckets with 2 rows:      28
Number of buckets with 3 rows:       9
Number of buckets with 4 rows:       4

Number of buckets with 100 or more rows:      0
### Hash table overall statistics ###
Total buckets: 256 Empty buckets: 133 Non-empty buckets: 123
Total number of rows: 182
Maximum number of rows in a bucket: 4
Average number of rows in non-empty buckets: 1.479675
```

- 通过10046 可以发现 先完成了ID 3 对TIMES的全表扫描后才开始对ID 4SALES表全表扫描
- 注意不是把整张表都物理读到内存中才开始构造，而是读一部分构造一部分
- 这不代表ID 3是阻塞操作，因为其行源一直返回给ID 2 HASH JOIN 以便构造哈希表
- 扫描TIMES表的同时在构造HASH TABLE
- HASH TABLE中共256个bucket

## 无关组合的例子2: Hash Join Probe Table

```
kxhfSetPhase: phase=PROBE_1

WAIT #1: nam='db file scattered read' ela= 40 file#=1 block#=80313 blocks=8 obj#=100605
WAIT #1: nam='db file scattered read' ela= 36 file#=1 block#=80321 blocks=8 obj#=100605
WAIT #1: nam='db file scattered read' ela= 36 file#=1 block#=437129 blocks=8 obj#=100605
WAIT #1: nam='db file scattered read' ela= 32 file#=1 block#=437137 blocks=8 obj#=100605
WAIT #1: nam='db file scattered read' ela= 41 file#=1 block#=437145 blocks=8 obj#=100605
WAIT #1: nam='db file scattered read' ela= 34 file#=1 block#=437153 blocks=8 obj#=100605
WAIT #1: nam='db file scattered read' ela= 78 file#=1 block#=437161 blocks=8 obj#=100605
WAIT #1: nam='db file scattered read' ela= 31 file#=1 block#=437169 blocks=8 obj#=100605

qerhjFetch: max probe row length (mpl=0)
*** RowSrcId: 2, qerhjFreeSpace(): free hash-join memory
kxhfRemoveChunk: remove chunk 0 from slot table
FETCH
#1:c=12675073,e=12397779,p=4482,cr=4490,cu=0,mis=0,r=1,dep=0,og=1,tim=1349337616628138

qerhjFetch → Hash Join Row source fetch

kxhfSetPhase - KXHF Set Phase

kxhfSetPhase 负责给HASH JOIN 设置阶段, 有三种阶段

Build Probe_1 Probe_2
```

- 构造完HASH TABLE后 进入 PROBE 阶段, 读取探测表SALES
- 不是把整张表都读到buffer cache里才开始probe, 如果这样做 当buffer cache很小 而表很大时根本没法处理, 而是物理多块读一部分, 然后在内存里探测, 然后再物理读一部分 再探测, 直到全表扫描结束

## 相关组合

- **特点为：有多个子节点，一个子节点控制其他子节点的执行**

NESTED LOOPS,  
UPDATE\*,  
FILTER\*,  
CONNECT BY WITH FILTERING,  
and BITMAP KEY ITERATION

# 相关组合的特点

- 子节点在父节点之前执行
- ID最小的子节点控制其他节点的执行
- 子节点按照ID顺序执行，但并不是 ID N执行完成后才执行ID N+1，而是交叉执行
- 负责控制的子节点一般只运行一次，其他节点可能运行0次或者很多次
- 不是每一个节点都返回数据给父节点

## 最普遍的相关组合: **Nested Loop**嵌套循环

- 典型的Join连接, 一般有2个子节点
- ID 最小的子节点作为驱动数据源, 在Nested loop中常叫做outer loop
- 其他的子节点为inner loop
- Inner Loop将为outer loop中返回的每一行而执行一次

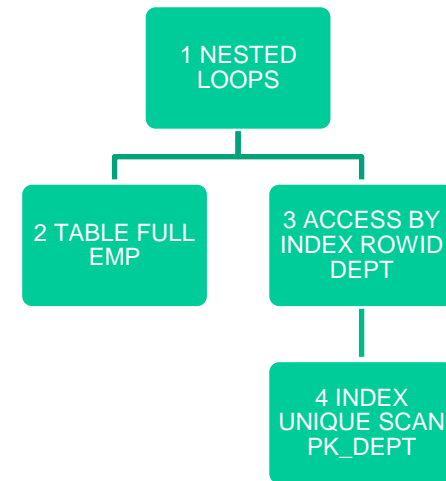
# 相关组合的例子: Nested Loop

```
select /*+ RULE */ * from emp, dept where emp.deptno= dept.deptno and emp.comm is null  
and dept.dname!= 'SALES'
```

Id	Operation	Name	Starts	A-Rows	A-Time	Buffers	Reads
0	SELECT STATEMENT		1	8	00:00:00.01	20	8
1	NESTED LOOPS		1	8	00:00:00.01	20	8
* 2	TABLE ACCESS FULL	EMP	1	10	00:00:00.01	8	6
* 3	TABLE ACCESS BY INDEX ROWID	DEPT	10	8	00:00:00.01	12	2
* 4	INDEX UNIQUE SCAN	PK_DEPT	10	10	00:00:00.01	2	1

Predicate Information (identified by operation id):

```
2 - filter("EMP"."COMM" IS NULL)  
3 - filter("DEPT"."DNAME"<>'SALES')  
4 - access("EMP"."DEPTNO"="DEPT"."DEPTNO")
```



- 按照基本原则，先执行2，同时2是控制子节点
- 对EMP的全表扫描将告知ID 3你需要做10次循环
- 基于独立操作的原则，ID 4先执行，由于是UNIQUE SCAN所以一次只返回一个rowid 给ID 3
- ID 3 再查看DEPT表上的数据并过滤出8行，返回给ID 1

## 另一种相关组合: Filter

- 对于Filter而言如果只有一个子节点，则是独立操作
- 如果有2个或更多子节点，则与Nested Loop类似

```
select /*+ RULE */ *  
from emp  
where not exists ( select 0  
from dept  
where dept.dname= 'SALES'  
and dept.deptno= emp.deptno)  
and not exists ( select 0  
from bonus  
where bonus.ename= emp.ename);
```

```
select dname, count(*)  
from emp, dept  
where emp.deptno= dept.deptno  
group by dname;
```

DNAME	COUNT (*)
ACCOUNTING	3
RESEARCH	5
SALES	6

## 另一种相关组合: Filter 示例

Id	Operation	Name	Starts	A-Rows	A-Time	Buffers
0	SELECT STATEMENT		1	8	00:00:00.01	14
* 1	FILTER		1	8	00:00:00.01	14
2	TABLE ACCESS FULL	EMP	1	14	00:00:00.01	8
* 3	TABLE ACCESS BY INDEX ROWID	DEPT	3	1	00:00:00.01	6
* 4	INDEX UNIQUE SCAN	PK_DEPT	3	3	00:00:00.01	3
* 5	TABLE ACCESS FULL	BONUS	8	0	00:00:00.01	0

Predicate Information (identified by operation id):

```
1 - filter(( IS NULL AND  IS NULL))
3 - filter("DEPT"."DNAME"='SALES')
4 - access("DEPT"."DEPTNO"=:B1)
5 - filter("BONUS"."ENAME"=:B1)
```

- ID 1有三个子节点 ID 2 3 5 , ID 2最小 , 先执行ID 2
- ID 2 对EMP 全表扫描, 将返回14行给 ID 1
- 在相关组合中ID 2应当控制 ID 3 ID 5的执行, 由于Oracle此处对distinct value做了优化, 所以ID 3只执行了3次。
- 基于独立操作规则, ID 4 执行3次, 并返回3个rvoid 到ID 3
- ID 3使用ID 4个给的3个ROWID来访问数据表块, 过滤filter("DEPT"."DNAME"='SALES')的数据, 由于是NOT exist所以这导致 ID 1 原来获得的14行 排除6 行的".DNAME"='SALES', 只剩下8行
- 这8行数据 影响了 ID 5 的执行次数, 将执行8次, 其中filter("BONUS"."ENAME"=:B1) 过滤条件的 :B1由 ID 1的8行数据提供, ID 5 没有返回数据, 所以那8行没有减少
- ID 1 将8行彻底过滤的数据返回给客户端



才开了个头哦。。。。

本讲座网上讨论答疑地址：

<http://t.askmaclean.com/thread-3237-1-1.html>

才开了个头哦。。。。

**To Be  
Continued.....**

つづく.....

敬请期待开Maclean Liu的Oracle  
性能优化讲座 第二回 索引

更多信息

www.askmaclean.com



or

<http://www.askmaclean.com/archives/tag/tuning>

# Question & Answer



**If you have more questions later, feel free to ask**