Oracle DB-Tuning Essentials

Agenda

1. The DB server and the tuning environment
2. Objective, Tuning versus Troubleshooting, Cost Based Optimizer
3. Object statistics
4. Access paths I
5. Access paths II
6. Monitoring Performance I
7. Monitoring Performance II
8. Application setup

Access Path

- Table Access
- Index Access
- Join Methods
- Sorts and Hash
- Hints
- Parallel Query

Join Methods
What is a join?

Relational database

- Stores entities in separate tables
- Tables are usually stored physically separate
- Queries require columns from several tables
- We need to join tables

Older syntax

- As if we do a Cartesian product, and then filter with predicates

```
select * from DEPT , EMP where DEPT.DEPTNO=EMP.DEPTNO
```

ANSII syntax

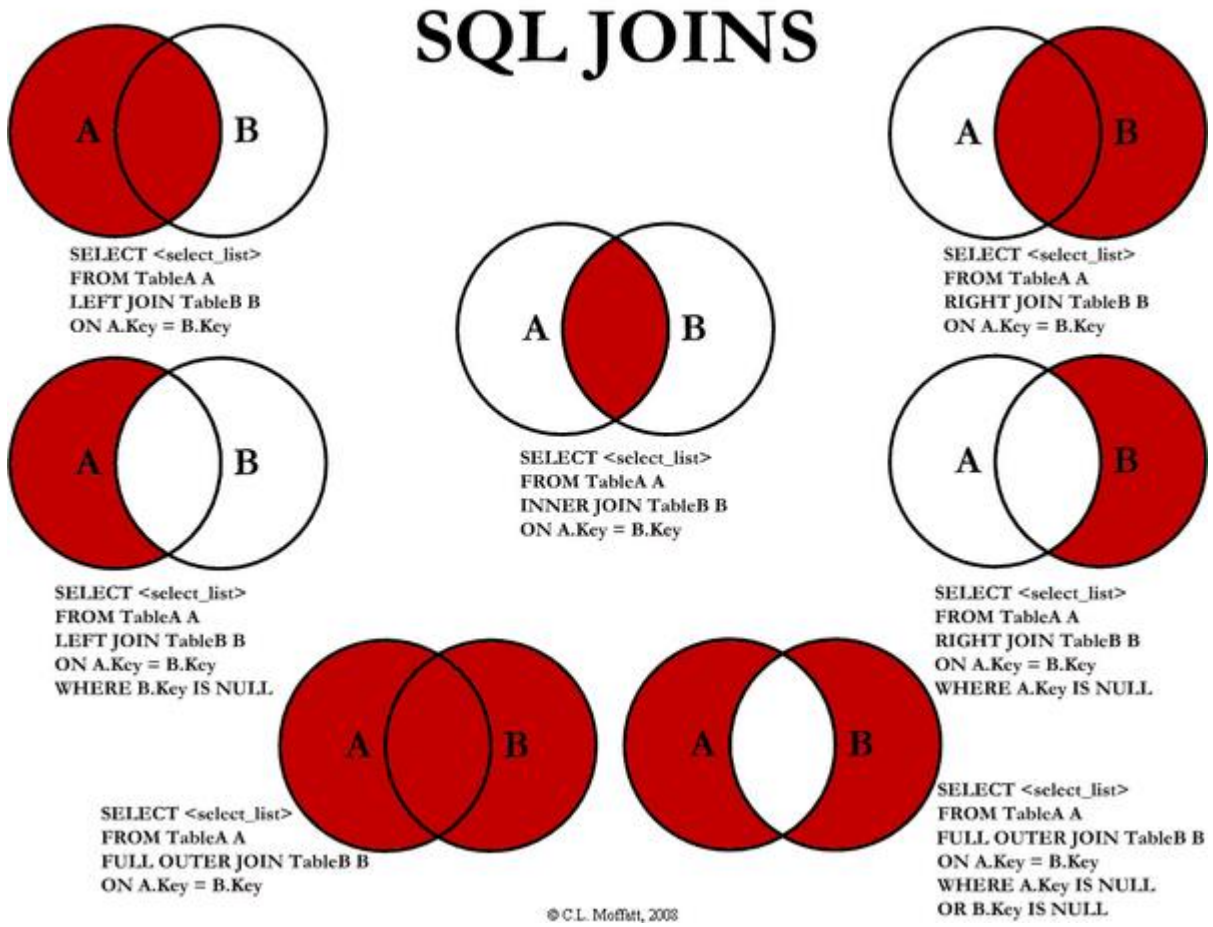
- Join condition is explicit

```
select * from DEPT join EMP on DEPT.DEPTNO=EMP.DEPTNO
```

  - more powerful, more standard ☺
  - unfortunately still some bugs with it ☹

- Don't mix the syntax in the same query !

Join Methods
What is a join?



https://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins

Join Methods
Equi-join and Theta-joins

Equi-join

- The join condition is an equality

```
select * from DEPT join EMP on DEPT.DEPTNO = EMP.DEPTNO
```

Theta-join

- The join condition is usually an inequality (e.g. <, >, >=)

```
select * from EMP e join EMP n on n.HIREDATE >= e.HIREDATE
```

Join cardinality

- (1..1) always have one inner row matching each outer row (not null FK)
- (0..1) may have one inner row matching each outer row (nullable FK)
- (0..n) can increase the number of rows (up to m*n if Cartesian product)
- Inner join (the default)
  returns only rows that matches, and all rows that match

Join Methods
Outer-join, Semi-join and Anti-join

Outer-join

- Returns a row from the outer join side even when there's no match

```
select * from DEPT  left outer join EMP on DEPT.DEPTNO=EMP.DEPTNO
select * from DEPT , EMP on DEPT.DEPTNO(+)=EMP.DEPTNO
```

Semi-join

- Only one row is returned even if several do match

```
select * from DEPT where exists ( select * from EMP where DEPT.DEPTNO=EMP.DEPTNO )
select * from DEPT where deptno in ( select deptno from EMP E )
```

Anti-join

- Return rows that do no match

```
select * from DEPT where not exists (  select * from EMP where DEPT.DEPTNO=EMP.DEPTNO )
select * from DEPT where deptno not in ( select deptno from EMP E where deptno is not null )
```

Join Methods
Nested Loop Join

Table

	EMPNO	ENAME	DEPTNO
#1	7369	SMITH	20
#2	7499	ALLEN	30
#3	7521	WARD	30
#4	7566	JONES	20

outer
outer
outer
outer

Index

DEPTNO	DNAME
10	ACCOUNTING
20	RESEARCH
30	SALES
40	OPERATIONS

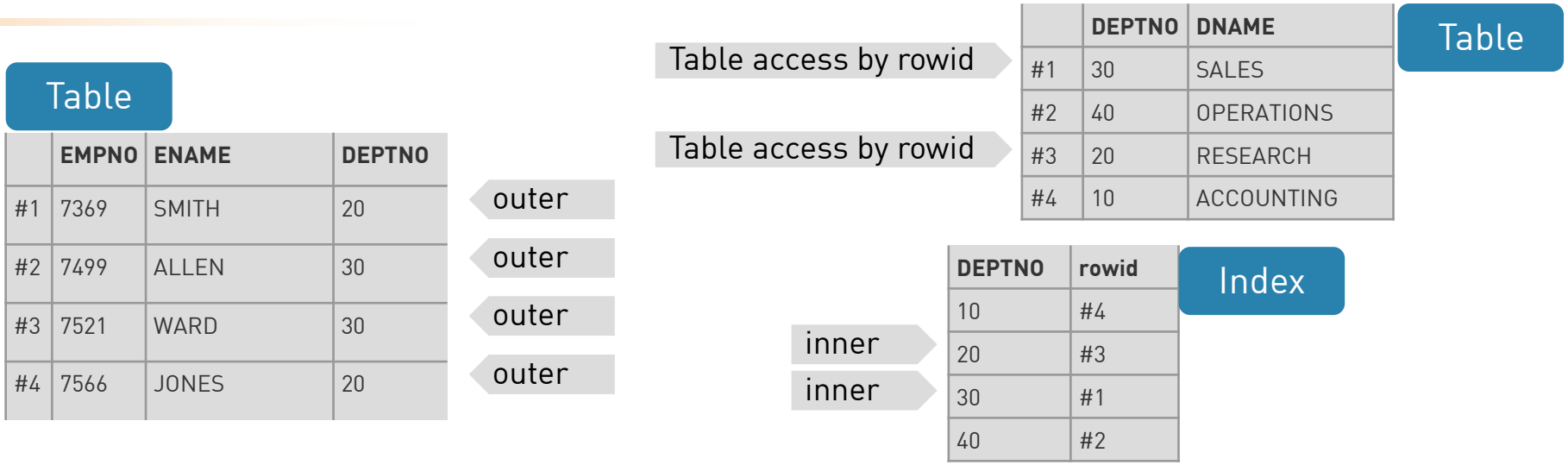
inner
inner

	EMPNO	ENAME	DNAME

Nested Loop Join

- **n** rows from the outer table, access the inner table **n** times
- ☺ Good when having efficient access to the inner table (index, cluster, partition)
- ☹ Bad if too many rows from outer table

Join Methods
Nested Loop Join + table access by rowid



Nested Loop Join

- Often requires to go to the table row by row
- ☹ Very bad if too many rows
- Index should be highly selective
- and have a good clustering factor

	EMPNO	ENAME	DNAME

Join Methods
Nested Loop execution plan (10g)

```
select * from DEPT join EMP using(deptno) where sal>=3000

-------------------------------------------------------------------------------
| Id  | Operation                    | Name    | Starts | A-Rows | Buffers |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |      1 |      3 |      13 |
|   1 |  NESTED LOOPS                |         |      1 |      3 |      13 |
|*  2 |   TABLE ACCESS FULL          | EMP     |      1 |      3 |       8 |
|   3 |   TABLE ACCESS BY INDEX ROWID| DEPT    |      3 |      3 |       5 |
|*  4 |    INDEX UNIQUE SCAN         | PK_DEPT |      3 |      3 |       2 |
-------------------------------------------------------------------------------
```

Nested Loop

- inner table (EMP) has 3 rows after applying the 'sal>=3000' predicate
- for each of them the DEPT is accessed by a rowid

Join Methods
Nested Loop execution plan (11g)

```
select * from DEPT join EMP using(deptno) where sal>=3000

-------------------------------------------------------------------------------
| Id  | Operation                    | Name    | Starts | A-Rows | Buffers |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |      1 |      3 |      13 |
|   1 |  NESTED LOOPS                |         |      1 |      3 |      13 |
|   2 |   NESTED LOOPS               |         |      1 |      3 |      10 |
|*  3 |    TABLE ACCESS FULL         | EMP     |      1 |      3 |       8 |
|*  4 |    INDEX UNIQUE SCAN         | PK_DEPT |      3 |      3 |       2 |
|   5 |   TABLE ACCESS BY INDEX ROWID| DEPT    |      3 |      3 |       3 |
-------------------------------------------------------------------------------
```

Join Batching

- A first loop retrieves a vector of rowid
- A second loop retrieves the rows by batch

Join Methods
Sort Merge Join

Table

SORT

	EMPNO	ENAME	DEPTNO
#1	7369	SMITH	20
#4	7566	JONES	20
#2	7499	ALLEN	30
#3	7521	WARD	30

outer
outer
outer
outer

SORT

Table

inner
inner
inner

DEPTNO	DNAME
10	ACCOUNTING
20	RESEARCH
30	SALES
40	OPERATIONS

Sort Merge Join

- Requires sorted input
- ☺ Good when we need a sorted result anyway
- Replaces hash join for theta joins

	EMPNO	ENAME	DNAME

Join Methods
Sort Merge join

```
select * from DEPT join EMP
on (EMP.deptno between DEPT.deptno and DEPT.deptno+10 ) where sal>=3000

-------------------------------------------------------------------------------
| Id  | Operation            | Name | Starts | A-Rows | Buffers |  OMem |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |      1 |      5 |      14 |       |
|   1 |  MERGE JOIN          |      |      1 |      5 |      14 |       |
|   2 |   SORT JOIN          |      |      1 |      3 |       7 |  2048 |
|*  3 |    TABLE ACCESS FULL | EMP  |      1 |      3 |       7 |       |
|*  4 |   FILTER             |      |      3 |      5 |       7 |       |
|*  5 |    SORT JOIN         |      |      3 |      5 |       7 |  2048 |
|   6 |     TABLE ACCESS FULL| DEPT |      1 |      4 |       7 |       |
-------------------------------------------------------------------------------
```

Sort Merge join

- Both sources are ordered, and then merged

Join Methods
Sort Merge join

```
select * from DEPT join EMP using(deptno) where sal>=3000 order by deptno

-------------------------------------------------------------------------
| Id  | Operation                     | Name        | Starts | A-Rows |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |             |      1 |      3 |
|   1 |  MERGE JOIN                   |             |      1 |      3 |
|*  2 |   TABLE ACCESS BY INDEX ROWID| EMP         |      1 |      3 |
|   3 |    INDEX FULL SCAN            | EMP_DEPTNO  |      1 |     14 |
|*  4 |   SORT JOIN                   |             |      3 |      3 |
|   5 |    TABLE ACCESS FULL          | DEPT        |      1 |      4 |
-------------------------------------------------------------------------
```

Sort Merge join without sorting

- When the first row source is already sorted, no need to sort again
- The second source will always need a buffer

Join Methods
Hash Join

Table

	EMPNO	ENAME	DEPTNO
#1	7369	SMITH	20
#2	7499	ALLEN	30
#3	7521	WARD	30
#4	7566	JONES	20

outer
outer
outer
outer

HASH Table

DEPTNO	DNAME
10	ACCOUNTING
20	RESEARCH
30	SALES
40	OPERATIONS

inner
inner

Hash Join

- Builds a hash table of the smaller input
- Longer to get the first result row
- But faster access later
- ☺ Better when the smaller table fits in memory

	EMPNO	ENAME	DNAME

Join Methods
Hash Join

```
select * from DEPT join EMP using(deptno)

-------------------------------------------------------------------------
| Id  | Operation          | Name | Starts | A-Rows | Buffers |  OMem |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |     14 |      15 |       |
|*  1 |  HASH JOIN         |      |      1 |     14 |      15 |  1321K|
|   2 |   TABLE ACCESS FULL| DEPT |      1 |      4 |       7 |       |
|   3 |   TABLE ACCESS FULL| EMP  |      1 |     14 |       8 |       |
-------------------------------------------------------------------------
```

Hash Join

- Builds a hash table with the inner row source
- Probes it from the outer row source

The order of tables can be swapped

- The smaller one is chosen to be the hash table

Join Methods
Merge Join Cartesian

Table

	EMPNO	ENAME	DEPTNO
#1	7369	SMITH	20
#2	7499	ALLEN	30
#3	7521	WARD	30
#4	7566	JONES	20

outer

outer

outer

outer

inner

BUFFER

Table

DEPTNO	DNAME
10	ACCOUNTING
20	RESEARCH
30	SALES
40	OPERATIONS

Merge Join Cartesian

- ☺ only between very small rowsources
- ☺ When most of the rows matches
- ☹ But disastrous if misestimate rows=1

	EMPNO	ENAME	DNAME

Join Methods
Merge join Cartesian

```
select /*+ leading(DEPT EMP) use_merge_cartesian(EMP) */ * from
DEPT join EMP using(deptno) where sal>3000

-------------------------------------------------------------------------------
| Id  | Operation             | Name | Starts | A-Rows | Buffers |  OMem |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |      |      1 |        |      15 |       |
|   1 |  MERGE JOIN CARTESIAN|      |      1 |      1 |      15 |       |
|   2 |   TABLE ACCESS FULL   | DEPT |      1 |      4 |       8 |       |
|   3 |   BUFFER SORT         |      |      4 |      1 |       7 |  2048 |
|*  4 |    TABLE ACCESS FULL | EMP  |      1 |      1 |       7 |       |
-------------------------------------------------------------------------------
```

Merge Join Cartesian

- When most of rows matches
- Buffers the outer table to avoid several full scan

- The BUFFER SORT is only a buffer, not a sort

Join Methods
Exercise

Check number of lines from SALES and CUSTOMERS

```
SQL> connect demo/demo
SQL> select table_name,num_rows,blocks from user_tables
where table_name in('SALES','CUSTOMERS');
```

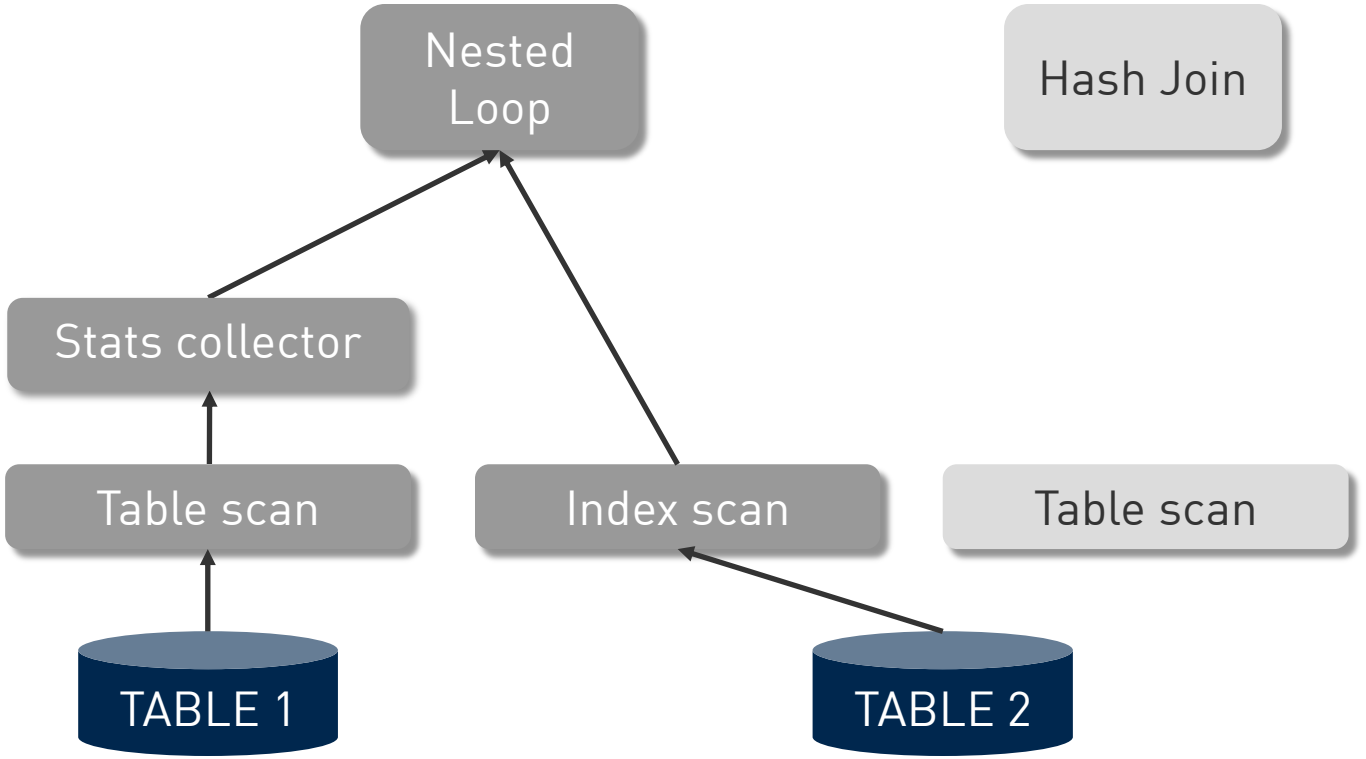
Run a join with different number of rows

```
SQL> set autotrace on

SQL> select
count(*),min(cust_year_of_birth),max(cust_year_of_birth)
from SALES join CUSTOMERS using(cust_id)
where amount>10;

SQL> select
count(*),min(cust_year_of_birth),max(cust_year_of_birth)
from SALES join CUSTOMERS using(cust_id)
where prod_id=20 and quantity>1;
```

Join Methods
Adaptive Plans

The execution plan contains two different paths

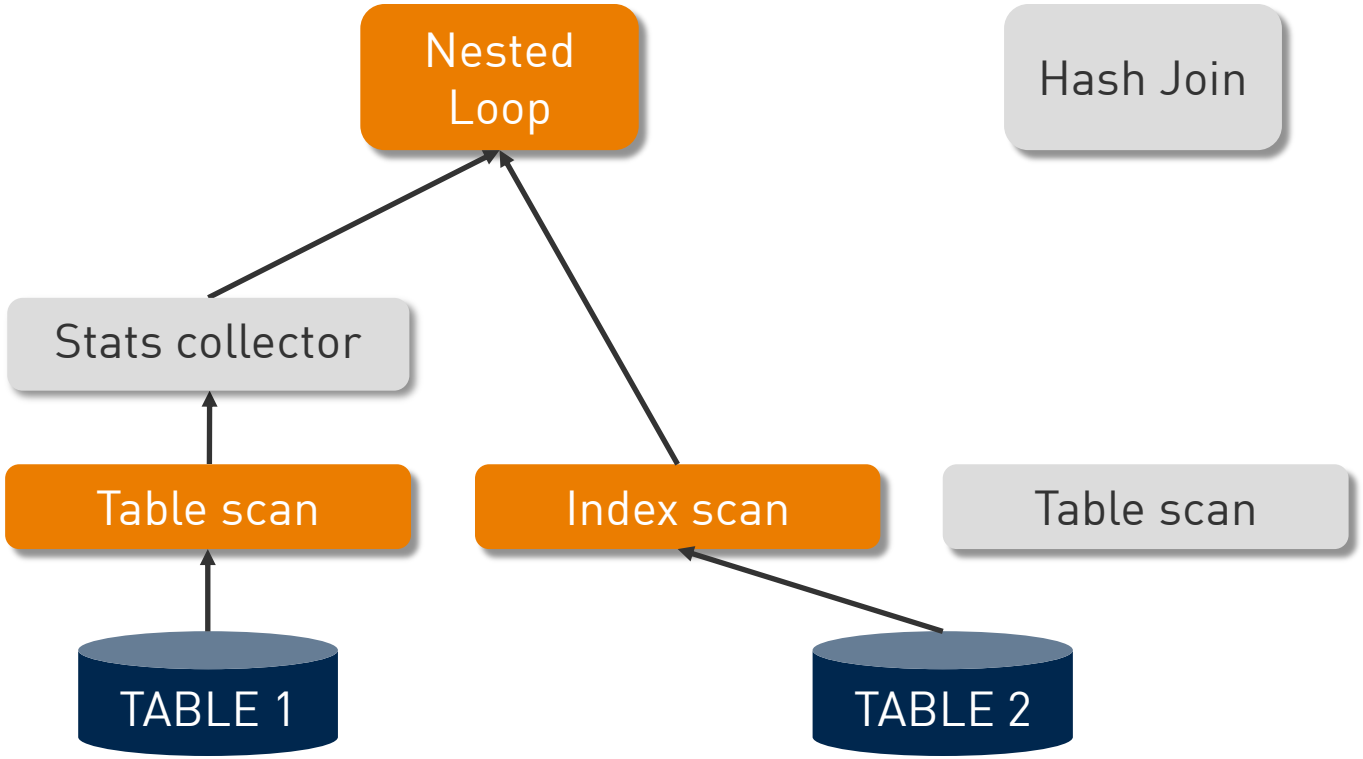
- Oracle evaluates an inflection point and will choose during the execution
- New operation statistics collector will buffer rows during table scan



Join Methods
Adaptive Plans

If the number of rows is less than the inflection point

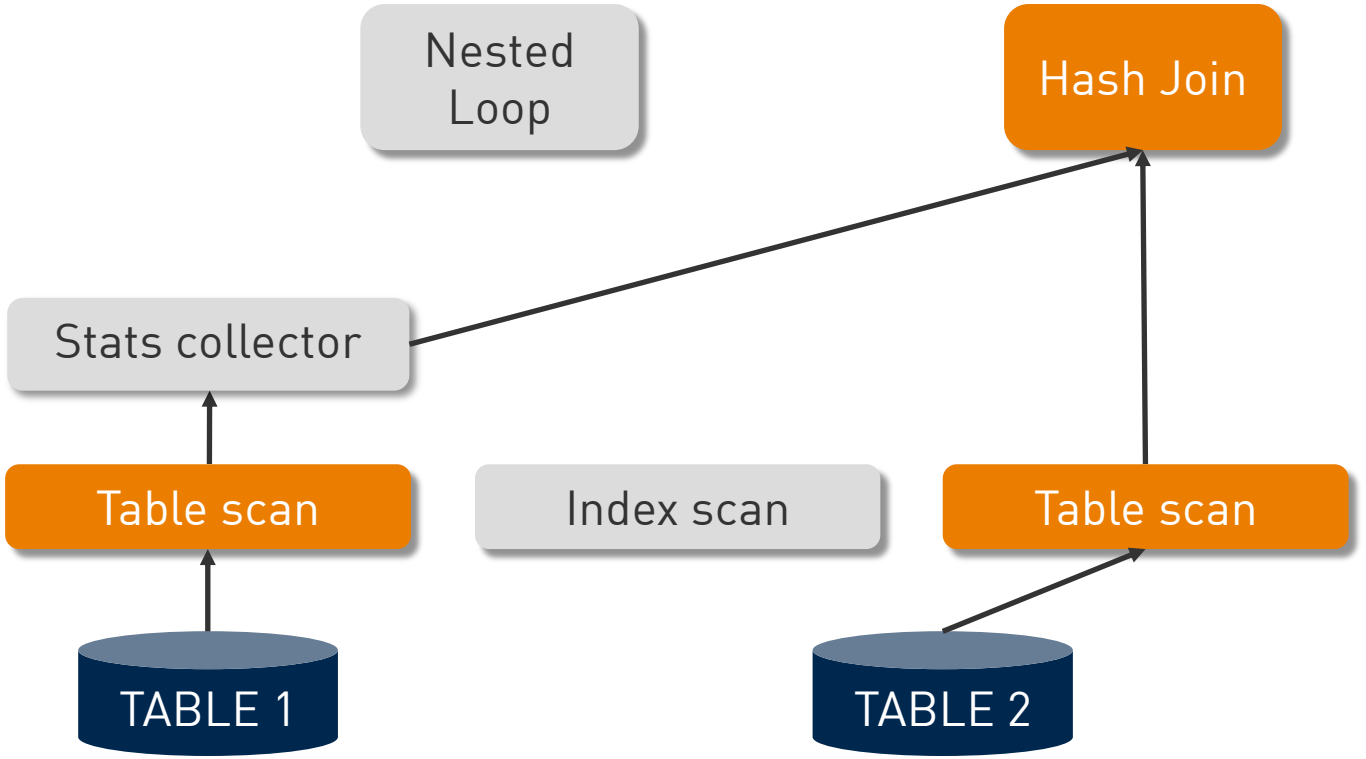
- Oracle executes the default plan
- All needed rows have been read and they are sent to the next operation



Join Methods
Adaptive Plans

If the number of rows is bigger than an inflection point

- Oracle switches to the alternative plan
- Statistics collector is disabled and rows are sent to the next operation



Join Methods
Adaptive Plans

Plan is adaptive only on first execution

Two adaptive methods

- Join method
- Parallel distribution

New information with dbms_xplan package

- Displays a note when execution plan is adaptive

```
Note
-----
   - this is an adaptive plan
```

New option to display adaptive steps in execution plans

```
SQL> select * from table (
            dbms_xplan.display(format=>'LAST ADAPTIVE'));
```

Join Methods
Demo: Adaptive Plans

Objective

- Display the default execution plan
- Display full execution plan including adaptive steps
- Execute the statement
- Display the executed execution plan

Statement used for the exercise

- Schema HR

```
SQL> select distinct DEPARTMENT_NAME from DEPARTMENTS
     join EMPLOYEES using(DEPARTMENT_ID)
     where LOCATION_ID like '%0' and SALARY>2000;
```

- A lot of location ids end with '0' but the optimizer is not aware of it.

Access Path


dbi InSite Workshops
> Table Access
> Index Access
> Join Methods
> Sorts and Hash
> Hints
> Parallel Query

Sorts and Hash
Workarea for ORDER BY, GROUP BY and Hash Joins

Some operations need to buffer data

- Sort rows (ORDER BY, Index creation)
- Deduplicate rows (DISTINCT)
- Build hash tables (HASH JOIN)

- The memory is private to the process: PGA
- When larger than memory swap to tempfiles

workarea_size_policy=auto

- The workarea will be allocated, Oracle try's to keep the sum of all workareas around pga_aggregate_target
- Limited by internal values (pga limited to 200MB per process)

workarea_size_policy=manual

- The workarea is defined by sort_area_size, hash_area_size

Sorts and Hash
Workarea (optimal)

When all the rows (workarea) fit in memory (PGA)

- They are sorted in memory only (optimal)

If they do not fit:

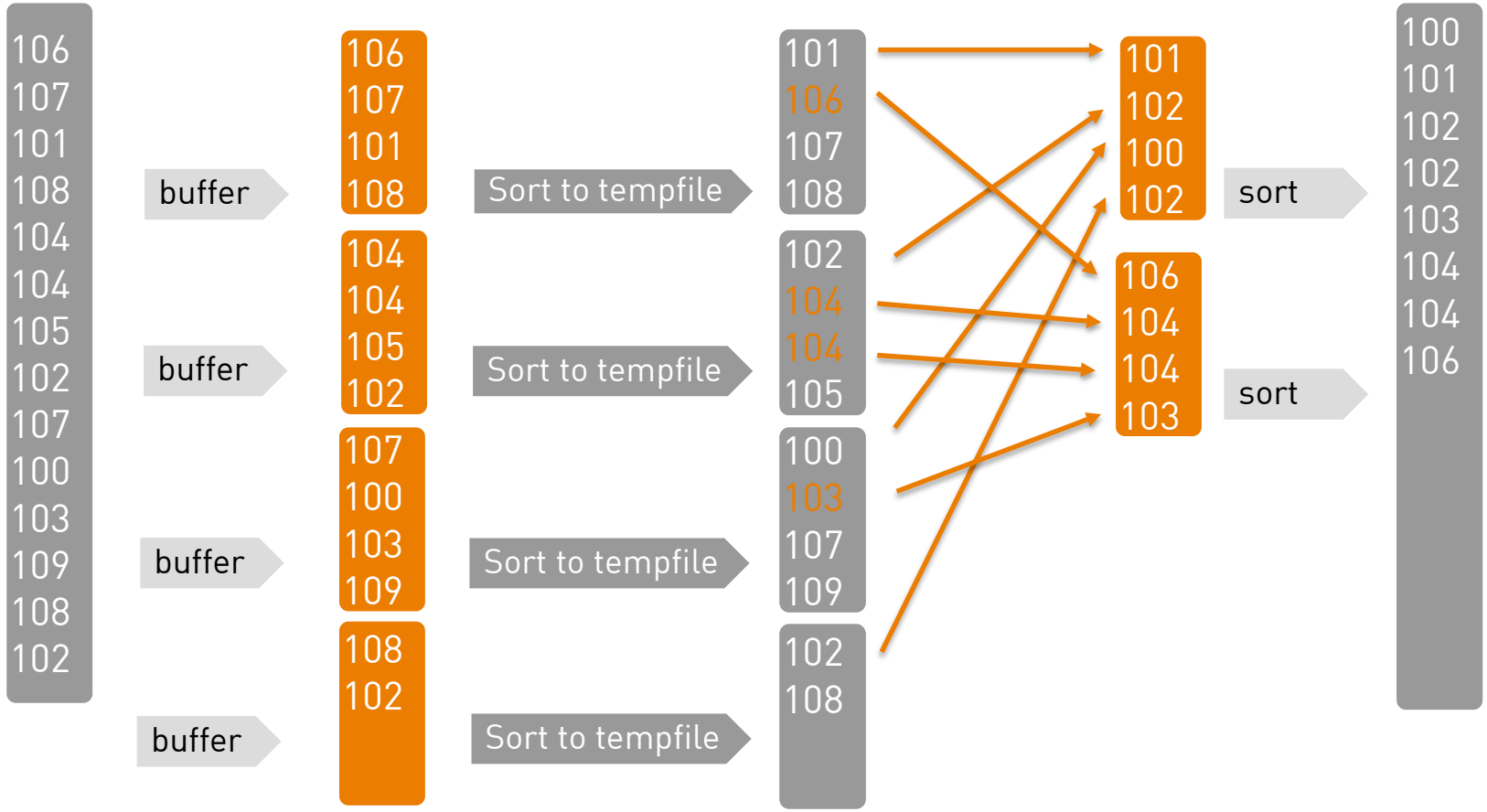
- Buffers will be stored in tempfiles, then chunks are merged (onepass)

Merge itself needs space, if not enough memory:

- The merge put new (larger) chunks in tempfile (multipass)

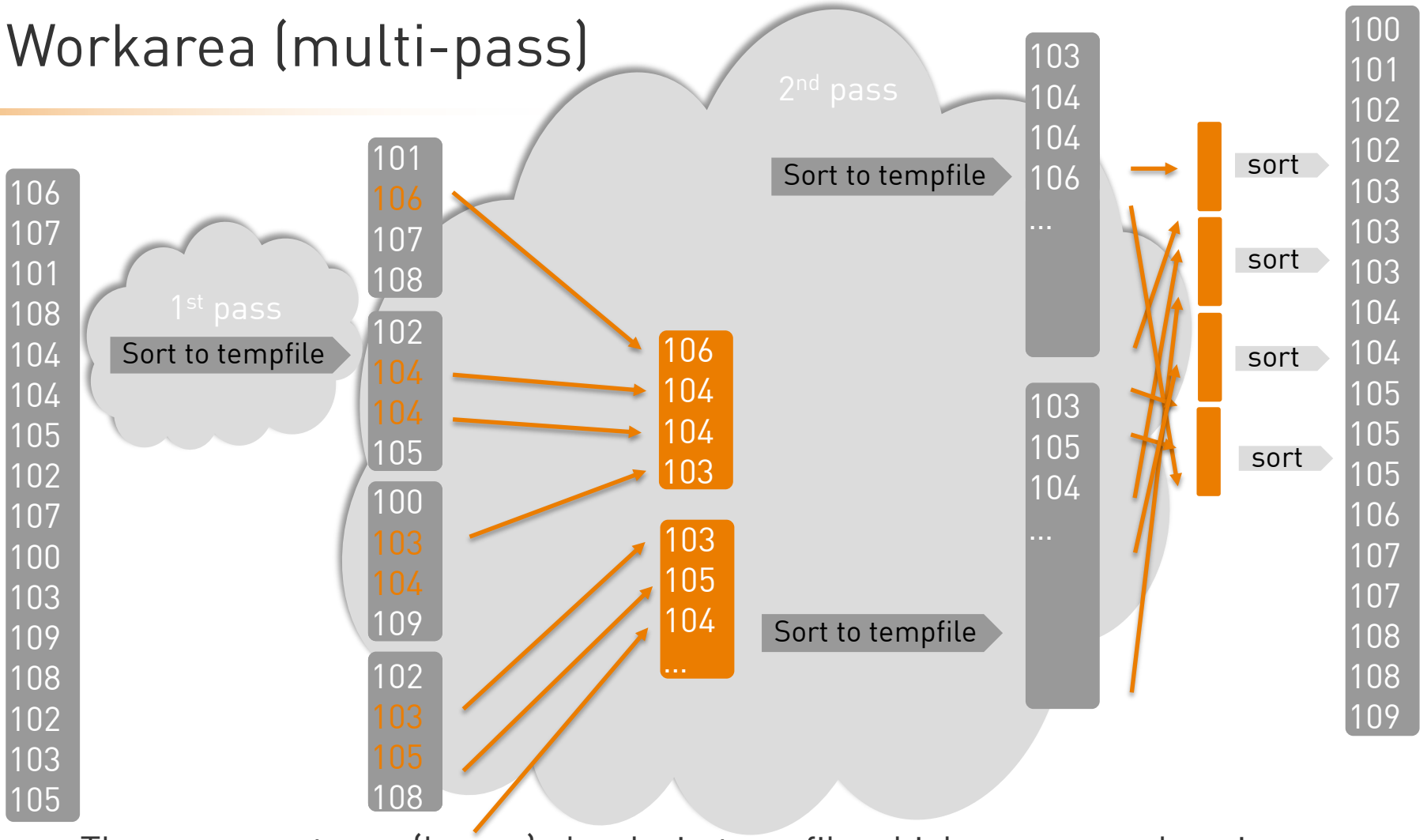
Sorts and Hash

Workarea (one-pass)



Sorts and Hash

Workarea (multi-pass)



The merge put new (larger) chunks in tempfile which are merged again

To be avoided (set larger PGA target)

Sorts and Hash
How to estimate the size?

Explain Plan shows the estimated workarea usage:

```
SQL> explain plan for select * from demo.test order by a;
SQL> select * from table(dbms_xplan.display);

-------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes |TempSpc|
-------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |  1000 |  3907K|       |
|   1 |  SORT ORDER BY     |      |  1000 |  3907K|  4008K|
|   2 |   TABLE ACCESS FULL| TEST |  1000 |  3907K|       |
-------------------------------------------------------------------
```

Frequent small sorts/hash

- Should be optimal

Large sorts/hash

- Can be done in 1 pass (batch, reporting)
- Multipass or high tempfile usage is probably sign of poor execution plan
- Think about partitioning, parallel query, manual sort_area_size

Sorts and Hash
How to view the size?

For current execution:

- V$SQL_WORKAREA_ACTIVE

For recent execution:

- dbms_xplan.display_cursor(format=>'memstat last')

```
----------------------------------------
 |  OMem |  1Mem | Used-Mem | Used-Tmp|
----------------------------------------
 |       |       |          |          |
 |  4500K|   893K|  509K (1)|    4096 |
 |       |       |          |          |
----------------------------------------
```

Note that Used-Tmp is in KB instead of bytes (bug – not fixed yet)

For past execution, or for failed statements:

- SQL Monitoring or ASH (TEMP_SPACE_ALLOCATED)

Sorts and Hash
Workarea size recommendation

Recommended value

- Workarea_policy=auto at system level
- Good for OLTP
- Still good for large volume if Parallel Query is used

Workarea_policy=manual

- When we can allocate 1 our 2 GB
- At session level only, and controlling the total number of sessions
- Set sort_area_size and hash_area_size as needed

Always be sure that enough physical RAM is available

Sorts and Hash
Exercise: manual workarea size

1. Run the following:

```
SQL> set linesize 1000 pagesize 1000
SQL> alter session set current_schema=SH;
SQL> alter session set statistics_level=all;

SQL> select * from ( select SALES.*,row_number()over(order
by amount_sold) rn from SALES ) where rn=1e6;
SQL> select * from table(
dbms_xplan.display_cursor(format=>'allstats last') );
```

- What is the size of the workarea?

2. Run it with different manual workarea sizes:

```
SQL> alter session set workarea_size_policy=manual
sort_area_size=200000000;
```

```
SQL> alter session set workarea_size_policy=manual
sort_area_size=2000000;
```

```
SQL> alter session set workarea_size_policy=manual
sort_area_size=200000;
```

Access Path

- Table Access
- Index Access
- Join Methods
- Sorts and Hash
- Hints
- Parallel Query

Hints
Why do we need hints?

SQL is a declarative language

- We specify only WHAT we want, not the way to do it
- The optimizer chooses HOW to get it

Hints are specified as special comments /*+ */

- Specify a special behavior (APPEND, PARALLEL)
- Specify additional behavior (RESULT_CACHE, MONITOR)
- Change parameters at query level (FIRST_ROWS, DYNAMIC_SAMPLING)
- Test a different execution plan (joins, access, transformations)
- Least resort to workaround a bad CBO choice

Hint behavior change through versions:
Document then, re-evaluate them at each upgrade

Hints
LEADING and joins

The CBO evaluates joins one after the other

- Don't specify a join method without the table order

- USE_HASH(EMP)
  means use Hash Join if joining to EMP
  but you can still have a Nested Loop joining EMP to DEPT

- LEADING(DEPT EMP) USE_HASH(EMP)
  means: start with DEPT and join to EMP using Hash Join method
  but the CBO will still choose which table is the hashed one

- LEADING(DEPT EMP) USE_HASH(EMP) NO_SWAP_JOIN_INPUTS(EMP)
  if you want to be sure to have DEPT as the hash table.
- LEADING(DEPT EMP) USE_HASH(EMP) SWAP_JOIN_INPUTS(EMP)
  if you want to be sure to have EMP as the hash table.

Hints
Explain Plan

Explain Plan shows the hints

```
SQL> select * from
table(dbms_xplan.display_cursor(format=>'+outline'));

…
Outline Data
-------------

  /*+
      BEGIN_OUTLINE_DATA
      IGNORE_OPTIM_EMBEDDED_HINTS
      OPTIMIZER_FEATURES_ENABLE('11.2.0.4')
      DB_VERSION('11.2.0.4')
      ALL_ROWS
      OUTLINE_LEAF(@"SEL$1")
      FULL(@"SEL$1" "EMP"@"SEL$1")
      INDEX(@"SEL$1" "DEPT"@"SEL$1" ("DEPT"."DEPTNO"))
      LEADING(@"SEL$1" "EMP"@"SEL$1" "DEPT"@"SEL$1")
      USE_NL(@"SEL$1" "DEPT"@"SEL$1")
      NLJ_BATCHING(@"SEL$1" "DEPT"@"SEL$1")
      END_OUTLINE_DATA
  */
```

Hints
Explain Plan

Explain Plan shows the hints

```
-------------------------------------------------
| Id  | Operation                     | Name    |
-------------------------------------------------
|   0 | SELECT STATEMENT              |         |
|   1 |  NESTED LOOPS                 |         |
|   2 |   NESTED LOOPS                |         |
|*  3 |    TABLE ACCESS FULL          | EMP     |
|*  4 |    INDEX UNIQUE SCAN          | PK_DEPT |
|   5 |   TABLE ACCESS BY INDEX ROWID | DEPT    |
-------------------------------------------------
…
      FULL(@"SEL$1" "EMP"@"SEL$1")
      INDEX(@"SEL$1" "DEPT"@"SEL$1" ("DEPT"."DEPTNO"))
      LEADING(@"SEL$1" "EMP"@"SEL$1" "DEPT"@"SEL$1")
      USE_NL(@"SEL$1" "DEPT"@"SEL$1")
```

Hints
Few important hints

FIRST_ROWS(n)

- Optimize to get first rows quickly (pagination)
  often favors nested loops and index access

RESULT_CACHE

- Enables result caching at statement level

APPEND

- Inserts after High Water Mark
  direct path, don't reuse free space, locks the table

DRIVING_SITE

- Determines where the joins are done when involving db links
  When DML, the driving site is always where the DML is done

Hints
Exercise: Hinting Joins

1. Run the following

```
SQL> connect / as sysdba
SQL> alter session set current_schema=SCOTT
statistics_level=all;
SQL> select *
from dept join emp using(deptno) join bonus using(ename);
```

- What is the join method chosen?

2. Add hints to the query in order to get

1. Full scan EMP
2. For each employee, get the DEPT by index
3. For each employee get to the BONUS by hash (the hashed table being BONUS)

Solution: the plan hash value should be 135669890

3. Show all the hints with the '+outline' format

Hints
Exercise: First Rows

1. Run the following

```
SQL> connect demo/demo
SQL> set autotrace trace
SQL> select * from customers order by cust_year_of_birth;
```

2. Then run the same with FIRST_ROWS(10) hint

- What is the difference?
- Which is estimated to be faster?
- Which is actually faster?

3. Compare the two following syntax:

```
SQL> select * from (select * from customers order by
cust_year_of_birth) where rownum<=10;

SQL> select * from customers order by cust_year_of_birth
fetch first 10 rows only;
```

- What is the difference?

Access Path

- Table Access
- Index Access
- Join Methods
- Sorts and Hash
- Hints
- Parallel Query

Parallel Query
Introduction

In serial, all the work is done by your session process

- You use at most one CPU
- Today, servers have more CPUs, but not faster ones

In parallel your session process is the coordinator

- Divides the job (e.g. divides full table scan in chunks)
- Use instance parallel processes (P00n) to do part of the job
- Do the serial operations
- Returns the result

Parallel processes communicate with others or with coordinator

- May have 2 sets: producers and consumers
- PX SEND and PX RECEIVE

Parallel Query
Full Table Scan

Coordinator process:

```
-------------------------------------------------------------------------
| Id  | Operation               | Name      | Starts | E-Rows | A-Rows |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |           |      1 |        |      1 |
|   1 |  SORT AGGREGATE         |           |      1 |      1 |      1 |
|   2 |   PX COORDINATOR        |           |      1 |        |      8 |
|   3 |    PX SEND QC (RANDOM)  | :TQ10000  |      0 |      1 |      0 |
|   4 |     SORT AGGREGATE      |           |      0 |      1 |      0 |
|   5 |      PX BLOCK ITERATOR  |           |      0 |  55500 |      0 |
|*  6 |       TABLE ACCESS FULL| CUSTOMERS |      0 |  55500 |      0 |
-------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   6 - access(:Z>=:Z AND :Z<=:Z)
```

- ‘allstats last’ shows only the last execution and this is only the coordinator statistics. Operations done by parallel processes are with starts=0

Parallel Query
Full Table Scan

Parallel processes:

```
-------------------------------------------------------------------------------
| Id  | Operation                | Name      | Starts | E-Rows | A-Rows |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |           |      1 |        |      1 |
|   1 |  SORT AGGREGATE          |           |      1 |      1 |      1 |
|   2 |   PX COORDINATOR         |           |      1 |        |      8 |
|   3 |    PX SEND QC (RANDOM)   | :TQ10000  |      0 |      1 |      0 |
|   4 |     SORT AGGREGATE       |           |      8 |      1 |      8 |
|   5 |      PX BLOCK ITERATOR   |           |      8 |  55500 |  55500 |
|*  6 |       TABLE ACCESS FULL  | CUSTOMERS |     77 |  55500 |  55500 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   6 - access(:Z>=:Z AND :Z<=:Z)
```

- 'allstats' sums all process statistics:
  - 1 coordinator doing steps 0 to 2
  - 8 parallel processes, each doing doing steps 3 to 6
  - 77 chunks (ranges of rowid) processed by 8 processes

Parallel Query
Demo: PX Full Table Scan

1. Run the following query

```
SQL> connect demo/demo
SQL> select max(cust_first_name) from customers;
```

2. Then run in parallel, with DOP = 8

- At query level with /*+ PARALLEL(CUSTOMERS 8) */ hint
- Or alter table CUSTOMERS parallel 8
- Or alter session force parallel query parallel 8

3. Show execution plan

```
SQL> select * from table(
 dbms_xplan.display_cursor(format=>'basic allstats -note')
);
```

Parallel Query
Joins

When joining a table that is read in parallel

- Rows from one set (producer) must be sent to one other set (consumer)
- Each set has a DOP, so with DOP=8 we may have 16 processes

Distribution can be

- Broadcast a small table
- Distribute a large table: more messaging but less reads

New feature in 12c

- The choice is adaptive at execution time

Parallel Query
Hybrid Distribution

Operation	Name	Lin...	Estimated R...	Cost	Execut...	Actual Rows	Memory (...	O...
SELECT STATEMENT		0			9	14		
PX COORDINATOR		1			9	14		
PX SEND QC (RANDOM)	:TQ10002	2	14	5	4	14		
HASH JOIN BUFFERED		3	14	5	4	14	7MB	
BUFFER SORT		4			4	16	8KB	
PX RECEIVE		5	4	3	4	16		
PX SEND HYBRID HASH	:TQ10000	6	4	3	1	16		
STATISTICS COLLECTOR		7			1	4		
TABLE ACCESS FULL	DEPT	8	4	3	1	4		
PX RECEIVE		9	14	2	4	14		
PX SEND HYBRID HASH	:TQ10001	10	14	2	4	14		
PX BLOCK ITERATOR		11	14	2	4	14		
TABLE ACCESS FULL	EMP	12	14	2	5	14		

- We see that the DOP is 4
- There are 4 rows coming from DEPT
- 16 rows has been received by the consumer
- Conclusion: broadcast happened

Access Path
Core Message

The optimizer chooses the access path

- But we have to provide the right structures, and accurate statistics
- Hints are not a long-term solution and must be documented

A bad join can change the response time to 1000x

- Adaptive join helps, but doesn't solve all issues

Indexing must be done at design time

- Too many indexes is overhead for inserts, updates, deletes
- Too few probably miss some efficient access path for queries

Any questions? Please do ask.