

---

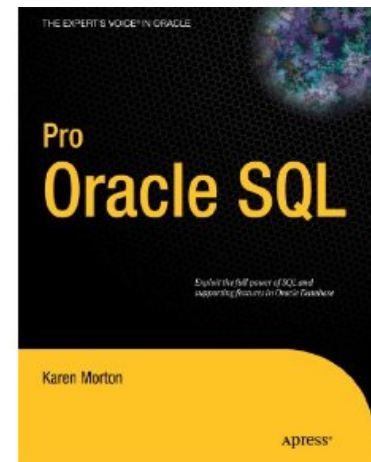
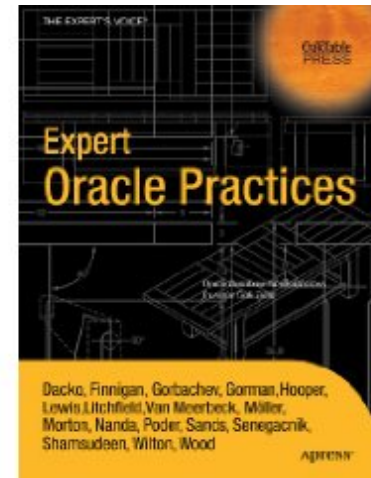
# Performance tuning A Brief Introduction

**By**  
***Riyaj Shamsudeen***



# Who am I?

- 18 years using Oracle products/DBA
- OakTable member
- Oracle ACE
- Certified DBA versions 7.0,7.3,8,8i,9i &10g
- Specializes in RAC, performance tuning, Internals and E-business suite
- Chief DBA with OraInternals
- Co-author of “Expert Oracle Practices” ‘2009
- Co-author of “Pro Oracle SQL” ‘2010
- Email: [rshamsud@orainternals.com](mailto:rshamsud@orainternals.com)
- Blog : [orainternals.wordpress.com](http://orainternals.wordpress.com)
- URL: [www.orainternals.com](http://www.orainternals.com)



---

# Disclaimer

**These slides and materials represent the work and opinions of the author and do not constitute official positions of my current or past employer or any other organization. This material has been peer reviewed, but author assume no responsibility whatsoever for the test cases.**

**If you corrupt your databases by running my scripts, you are solely responsible for that.**

**This material should not should not be reproduced or used without the authors' written permission.**

---

# Agenda

- **Tracing SQL execution**
- **Analyzing Trace files**
- **Understanding Explain plans**
- **Joins**
- **Writing optimal SQL**
- **Good hints and no-so good hints**
- **Effective indexing**
- **Partitioning for performance**

---

# SQL Tracing

- ❑ Easiest way to trace SQL execution is with the following SQL statements:

```
alter session set sql_trace=true;
```

```
exec dbms_session.set_sql_trace(sql_trace=>true);
```

- ❑ But, this level of tracing does not provide all the relevant needed details.
- ❑ For example, this level does not tell us whether the SQL spent time waiting for I/O or for a lock.

---

# Waits

- ❑ SQL statement execution is in one of two states:
  - ✓ On CPU executing the statement (or)
  - ✓ Waiting for an event such as I/O, lock etc.
  
- ❑ For example, SQL statement waiting for I/O will wait for one of these events:
  - ✓ db file sequential read
  - ✓ db file scattered read etc.
  
- ❑ It is important to understand time spent in the events.
  
- ❑ CPU queue wait state is not discussed to improve clarity.

# Tracing with waits

- ❑ SQL statement executions can be traced with waits printed to a trace file using:

```
alter session set events '10046 trace name context forever, level 8';
```

```
exec dbms_session.session_trace_enable(waits => true);
```

Trace level is a 4 bit integer:

level 1 : Lowest level, same as sql\_trace=true

level 4 : Level 1 + captures bind variables

level 8 : Level 1 + captures wait events at SQL level

level 12: level 1 + captures bind and wait events at SQL level

# Trace file explained

...1

```
PARSING IN CURSOR #3 len=83 dep=0 uid=173 oct=3 lid=173 tim=12972441295985
hv=855947554 ad='fa2e5530'
select transaction_id from mtl_transaction_accounts where transaction_id=1682944981
END OF STMT
```

```
PARSE #3:c=10000,e=5080,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=1,tim=12972441295971
```

```
EXEC #3:c=0,e=198,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,tim=12972441296397
```

```
WAIT #3: nam='SQL*Net message to client' ela= 8 driver id=1650815232 #bytes=1 p3=0 obj#=38149 tim=12972441296513
```

```
WAIT #3: nam='db file sequential read' ela= 8337 file#=803 block#=213006 blocks=1 obj#=38172 tim=12972441305307
```

```
WAIT #3: nam='db file sequential read' ela= 12840 file#=234 block#=65037 blocks=1 obj#=38172 tim=12972441318487
```

```
WAIT #3: nam='gc cr grant 2-way' ela= 807 p1=803 p2=213474 p3=1 obj#=38172 tim=12972441320096
```

```
WAIT #3: nam='db file sequential read' ela= 4095 file#=803 block#=213474 blocks=1 obj#=38172 tim=12972441324278
```

- ❑ Highlighted lines indicates that SQL statement is being parsed.

# Trace file explained

...2

```
PARSING IN CURSOR #3 len=83 dep=0 uid=173 oct=3 lid=173 tim=12972441295985
hv=855947554 ad='fa2e5530'
select transaction_id from mtl_transaction_accounts where transaction_id=1682944981
END OF STMT
PARSE #3:c=10000,e=5080,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=1,tim=12972441295971
EXEC #3:c=0,e=198,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,tim=12972441296397
WAIT #3: nam='SQL*Net message to client' ela= 8 driver id=1650815232 #bytes=1 p3=0 obj#=38149 tim=12972441296513
```

```
WAIT #3: nam='db file sequential read' ela= 8337 file#=803 block#=213006 blocks=1 obj#=38172
tim=12972441305307
WAIT #3: nam='db file sequential read' ela= 12840 file#=234 block#=65037 blocks=1 obj#=38172
tim=12972441318487
WAIT #3: nam='db file sequential read' ela= 4095 file#=803 block#=213474 blocks=1 obj#=38172
tim=12972441324278
```

Event waited for

Elapsed time (micro seconds)

□ Later, we will show how tkprof utility can aggregate these events and print formatted output.

---

# Tracing with binds

- ❑ SQL statement executions can be traced with bind variable values also:

```
alter session set events '10046 trace name context forever, level 12';
```

```
exec dbms_session.session_trace_enable(waits => true, binds => true);
```

- ❑ Generally, you wouldn't need to trace with binds unless you want to find the bind variables also.
- ❑ If there are many executions of SQL statements (millions) in a process then turning on trace with binds can reduce performance.

---

# Agenda

- Tracing SQL execution
- **Analyzing Trace files**
- Understanding Explain plans
- Joins
- Writing optimal SQL
- Good hints and no-so good hints
- Effective indexing
- Partitioning for performance

---

# tkprof

- ❑ Trace files generated from SQLTrace is not exactly readable.
- ❑ tkprof utility provides formatted output from the trace files.

```
tkprof trace_file outfile sort=option explain=user/pwd sys=no
```

Example:

```
tkprof devl_ora_123.trc /tmp/devl_ora_tkp.out sort=exeela,fchela
```

- ❑ tkprof comes with help options. Just type tkprof and enter to see help.

---

# Sort option

- ❑ Sort option in tkprof is useful. For example, sort=exeela will sort the SQL statements by Elapsed time at execution step.
- ❑ Top SQL statements by elapsed time will show up at the top of the tkprof output file.
- ❑ Generally, I use sort=exeela, fchela to sort the SQL statements.
- ❑ Other common options are: execpu, fchcpu, prsela etc.

---

# explain option

- ❑ Explain=userid/password@dev can be supplied to print execution plan of that SQL statement.
- ❑ Explain option logs on to the database and executes explain plan for every SQL in the trace file.
- ❑ But, SQL Trace also prints execution plan in the trace file.
- ❑ It is a good practice not to use explain option and use the execution plan in the trace file itself.
- ❑ If you are using SQLPlus or TOAD to execute SQL, make sure to close the connection or exit so that trace files will be complete.

---

# Agenda

- Tracing SQL execution
- Analyzing Trace files
- **Understanding Explain plans**
- Joins
- Writing optimal SQL
- Good hints and no-so good hints
- Effective indexing
- Partitioning for performance

---

# Execution plan

- ❑ Understanding execution plan is an important step in resolving performance issues.
- ❑ There are many ways to generate execution plans and easiest way is:

explain plan for





```
select t1.id, t1.vc10 , t2.id
```

```
From t1,t2 where t1.id=t2.id and t2.id <1000;
```




```
Select * from table(dbms_xplan.display);
```

# Simple SELECT





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

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		975	22425	512 (1)	00:00:02
* 1	 HASH JOIN		975	22425	512 (1)	00:00:02
* 2	 INDEX RANGE SCAN	T2_N1	976	5856	5 (0)	00:00:01
3	 TABLE ACCESS BY INDEX ROWID	T1	1000	17000	506 (1)	00:00:02
* 4	 INDEX RANGE SCAN	T1_N1	1000		5 (0)	00:00:01

Predicate Information (identified by operation id):

-  1 - access("T1"."ID"="T2"."ID")
-  2 - access("T2"."ID"<1000)
-  4 - access("T1"."ID"<1000)

## Execution sequence

-  (i) Index T2\_N1 was scanned with access predicate t2.id <1000 [ Step 2]
-  (ii) Index T1\_N1 was scanned with access predicate t1.id <1000 [ Step 4]
-  (iii) Table T1 is accessed to retrieve non-indexed column using rowids returned from step 2. [ Step 3].
-  (iv) Rows from step 2 and 3 are joined to create the final result set.[Step 1]

# Predicates

- ❑ Predicates are very useful to debug performance issues.
- ❑ Cardinality estimate is a good guess. 20 rows are returned in step 5, but just one row returned from the table (step 4).

Set lines 120 pages 0

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

Id	Operation	Name	Rows	Bytes	Cost (%CPU)
0	SELECT STATEMENT		1	214	9 (23)
1	SORT ORDER BY		1	214	9 (23)
* 2	TABLE ACCESS BY INDEX ROWID	RCV_STAGING_LINES	1	155	4 (25)
3	NESTED LOOPS		1	214	8 (13)
* 4	TABLE ACCESS BY INDEX ROWID	RCV_STAGING_HEADERS	1	59	5 (20)
* 5	INDEX RANGE SCAN	RCV_STAGING_HEADERS_N5	20		4 (25)
* 6	INDEX RANGE SCAN	RCV_STAGING_LINES_N3	1		3 (34)

Predicate Information (identified by operation id):

2 - filter("RSL"."RMA\_NUMBER"=:Z AND "RSL"."RMA\_LINE\_NUMBER"=TO\_NUMBER(:Z))

4 - filter("RSH"."RMA\_NUMBER"=:Z AND "RSH"."REQUEST\_ID"=TO\_NUMBER(:Z))  
5 - access("RSH"."STATUS"='PROCESSED')

6 - access("RSH"."SOURCE\_HEADER\_ID"="RSL"."SOURCE\_HEADER\_ID")

# Index selection.

- Notice that all predicates are applied in step 5 using index.
- Cardinality estimates improved from 20 to 1.

```
create index apps.rcv_staging_headers_n5 on apps.RCV_STAGING_HEADERS  
  (status, rma_number, request_id ) compress 1 compute statistics;
```

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

Id	Operation	Name	Rows	Bytes	Cost (%CPU)
0	SELECT STATEMENT		1	206	9 (23)
1	SORT ORDER BY		1	206	9 (23)
* 2	TABLE ACCESS BY INDEX ROWID	RCV_STAGING_LINES	1	155	4 (25)
3	NESTED LOOPS		1	206	8 (13)
4	TABLE ACCESS BY INDEX ROWID	RCV_STAGING_HEADERS	1	51	5 (20)
* 5	INDEX RANGE SCAN	RCV_STAGING_HEADERS_N5	1		4 (25)
* 6	INDEX RANGE SCAN	RCV_STAGING_LINES_N3	1		3 (34)

Predicate Information (identified by operation id):

- 2 - filter("RSL"."RMA\_NUMBER"=:Z AND "RSL"."RMA\_LINE\_NUMBER"=TO\_NUMBER(:Z))
- 5 - access("RSH"."STATUS"='PROCESSED' AND "RSH"."RMA\_NUMBER"=:Z AND "RSH"."REQUEST\_ID"=TO\_NUMBER(:Z))
- 6 - access("RSH"."SOURCE\_HEADER\_ID"="RSL"."SOURCE\_HEADER\_ID")

---

# Package: dbms\_xplan

- ❑ Dbms\_xplan is a package available from 9i. It has many rich features.
- ❑ dbms\_xplan package has following features:
  - ❑ Print execution plan for recently explained SQL statement:  
`dbms_xplan.display;`
  - ❑ Print execution plan for a SQL statement executed by passing `sql_id` or `hash_value`:  
`dbms_xplan.display_cursor ('&sqlid', ", ");`
  - ❑ Print execution plan for a SQL statement captured by AWR report:  
`dbms_xplan.display_awr ('&sqlid', ", ");`

# Autotrace deprecated

□ Many of us are used to autotrace in SQL\*Plus. Use dbms\_xplan instead ( from 10g onwards):

```
select t1.id, t1.vc10 , t2.id
From t1,t2 where t1.id=t2.id and t2.id <1000;
```

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

```
SQL_ID cd6h52abfgfg3, child number 0
```

```
-----
select t1.id, t1.vc10 , t2.id From t1,t2 where t1.id=t2.id and t2.id <1000
```

```
Plan hash value: 3286489634
```

```
-----
| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |
-----
| 0 | SELECT STATEMENT | | | | 512 (100)| |
|* 1 | HASH JOIN | | 975 | 22425 | 512 (1)| 00:00:02 |
...

```

---

## But..

- ❑ `dbms_xplan` accesses `v$sql`, `v$sql_plan_statistics_all` and `v$sql_plan` internally. So, you need to have select access on these fixed views to execute `dbms_xplan`.
- ❑ Executing `dbms_xplan.display_cursor` without specifying any `sql_id` retrieves the execution plan for the query executed last.
- ❑ From 10g onwards, `serveroutput` is on by default. You need to disable `serveroutput` to see the execution plan correctly:

```
Select * from table (dbms_xplan.display_cursor);
```

```
SQL_ID 9babjv8yq8ru3, child number 1  
BEGIN DBMS_OUTPUT.GET_LINES(:LINES, :NUMLINES); END;
```

# Display\_cursor

- ❑ Suppose, you want to find the execution plan of a SQL statement executed recently, then `display_cursor` is handy.
- ❑ Find `sql_id` or `hash_value` of the SQL statement from `v$sql` and pass that to `display_cursor`:

```
select * from table(dbms_xplan.display_cursor ('&sql_id',''));
```

```
SQL_ID fqmkvmdyb043r, child number 0
```

```
-----
```

```
select t1.id, t1.vc10 , t2.id From t1,t2 where t1.id=t2.id and t2.id <1000
```

```
Plan hash value: 3286489634
```

```
-----
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT				512 (100)	
* 1	HASH JOIN		975	22425	512 (1)	00:00:02
* 2	INDEX RANGE SCAN	T2_N1	976	5856	5 (0)	00:00:01
3	TABLE ACCESS BY INDEX ROWID	T1	1000	17000	506 (1)	00:00:02
...						

```
-----
```

# Time spent?

Total of 330 seconds spent on this SQL.  
Can you identify the step that incurred most time?

Rows	Row Source Operation
49996	WINDOW SORT
644538	NESTED LOOPS
704891	NESTED LOOPS
704891	NESTED LOOPS
704891	NESTED LOOPS OUTER
704891	NESTED LOOPS
704891	NESTED LOOPS
1	TABLE ACCESS BY INDEX ROWID HR_ALL_ORGANIZATION_UNITS
1	INDEX UNIQUE SCAN HR_ORGANIZATION_UNITS_PK
704891	TABLE ACCESS BY INDEX ROWID WSH_DELIVERY_DETAILS
704891	INDEX RANGE SCAN WSH_DELIVERY_DETAILS_N8
704891	TABLE ACCESS BY INDEX ROWID MTL_SYSTEM_ITEMS_B
704891	INDEX RANGE SCAN MTL_SYSTEM_ITEMS_B_U1
644783	VIEW PUSHED PREDICATE
644783	NESTED LOOPS
704891	TABLE ACCESS BY INDEX ROWID FND_FLEX_VALUE_SETS
704891	INDEX UNIQUE SCAN FND_FLEX_VALUE_SETS_U2
644783	TABLE ACCESS BY INDEX ROWID FND_FLEX_VALUES
644783	INDEX RANGE SCAN FND_FLEX_VALUES_N1
704891	TABLE ACCESS BY INDEX ROWID OE_ORDER_LINES_ALL
704891	INDEX UNIQUE SCAN OE_ORDER_LINES_U1
704891	TABLE ACCESS BY INDEX ROWID WSH_DELIVERY_ASSIGNMENTS
704891	INDEX RANGE SCAN WSH_DELIVERY_ASSIGNMENTS_N3
644538	TABLE ACCESS BY INDEX ROWID WSH_NEW_DELIVERIES
704891	INDEX UNIQUE SCAN WSH_NEW_DELIVERIES_U1

---

# Statistics\_level

- ❑ Statistics\_level parameter provides valuable row source execution statistics.
- ❑ This parameter can be changed at session or instance level:  
`alter session set statistics_level=all;`
- ❑ Default value of this parameter is TYPICAL which do not collect row source execution statistics.
- ❑ This parameter incurs additional overhead. But, still, tracing at session level is acceptable.

# Statistics\_level=ALL

Setting this parameter to ALL provides valuable Row source statistics.

```
Rows      Row Source Operation
-----
49996 WINDOW SORT (cr=11333940 r=66442 w=46044 time=327835520 us)
644538 NESTED LOOPS (cr=11333940 r=41631 w=0 time=302305095 us)
704891 NESTED LOOPS (cr=9219265 r=34793 w=0 time=263965257 us)
704891 NESTED LOOPS (cr=7093885 r=27898 w=0 time=217008667 us)
704891 NESTED LOOPS OUTER (cr=4979210 r=460 w=0 time=80832751 us)
704891 NESTED LOOPS (cr=2219750 r=460 w=0 time=46780270 us)
704891 NESTED LOOPS (cr=51405 r=0 w=0 time=15862881 us)
1      TABLE ACCESS BY INDEX ROWID HR_ALL_ORGANIZATION_UNITS (cr=3 r=0 w=0 time=79 us)
1      INDEX UNIQUE SCAN HR_ORGANIZATION_UNITS_PK (cr=2 r=0 w=0 time=42 us)(object id 43498)
704891 TABLE ACCESS BY INDEX ROWID WSH_DELIVERY_DETAILS (cr=51402 r=0 w=0 time=15456581 us)
704891 INDEX RANGE SCAN WSH_DELIVERY_DETAILS_N8 (cr=2692 r=0 w=0 time=1677304 us)(object id 5124003)
704891 TABLE ACCESS BY INDEX ROWID MTL_SYSTEM_ITEMS_B (cr=2168345 r=460 w=0 time=26259264 us)
704891 INDEX RANGE SCAN MTL_SYSTEM_ITEMS_B_U1 (cr=1409795 r=213 w=0 time=15069327 us)(object id 38017)
644783 VIEW PUSHED PREDICATE (cr=2759460 r=0 w=0 time=30859890 us)
644783 NESTED LOOPS (cr=2759460 r=0 w=0 time=29181767 us)
704891 TABLE ACCESS BY INDEX ROWID FND_FLEX_VALUE_SETS (cr=1409784 r=0 w=0 time=11031089 us)
704891 INDEX UNIQUE SCAN FND_FLEX_VALUE_SETS_U2 (cr=704893 r=0 w=0 time=6257393 us)(object id 33768)
644783 TABLE ACCESS BY INDEX ROWID FND_FLEX_VALUES (cr=1349676 r=0 w=0 time=13839752 us)
644783 INDEX RANGE SCAN CCW_FND_FLEX_VALUES_N1 (cr=704893 r=0 w=0 time=8683861 us)(object id 5153800)
704891 TABLE ACCESS BY INDEX ROWID OE_ORDER_LINES_ALL (cr=2114675 r=27438 w=0 time=133292520 us)
704891 INDEX UNIQUE SCAN OE_ORDER_LINES_U1 (cr=1409784 r=2025 w=0 time=14863664 us)(object id 42102)
704891 TABLE ACCESS BY INDEX ROWID WSH_DELIVERY_ASSIGNMENTS (cr=2125380 r=6895 w=0 time=42893004 us)
704891 INDEX RANGE SCAN WSH_DELIVERY_ASSIGNMENTS_N3 (cr=1413108 r=2580 w=0 time=24181814 us)(object id
46295)
644538 TABLE ACCESS BY INDEX ROWID WSH_NEW_DELIVERIES (cr=2114675 r=6838 w=0 time=35307346 us)
704891 INDEX UNIQUE SCAN WSH_NEW_DELIVERIES_U1 (cr=1409784 r=362 w=0 time=7381948 us)(object id 46306)
```

# Statistics\_level=ALL



Rows	Row Source Operation
49996	WINDOW SORT (cr=11333940 r=66442 w=46044 time=327835520 us)
644538	NESTED LOOPS (cr=11333940 r=41631 w=0 time=302305095 us)
704891	NESTED LOOPS (cr=9219265 r=34793 w=0 time=263965257 us)
704891	NESTED LOOPS (cr=7093885 r=27898 w=0 time=217008667 us)
704891	NESTED LOOPS OUTER (cr=4979210 r=460 w=0 time=80832751 us)
704891	NESTED LOOPS (cr=2219750 r=460 w=0 time=46780270 us)
704891	NESTED LOOPS (cr=51405 r=0 w=0 time=15862881 us)
1	TABLE ACCESS BY INDEX ROWID HR_ALL_ORGANIZATION_UNITS (cr=3 r=0 w=0 time=79 us)
1	INDEX UNIQUE SCAN HR_ORGANIZATION_UNITS_PK (cr=2 r=0 w=0 time=42 us)(object id 43498)
704891	TABLE ACCESS BY INDEX ROWID WSH_DELIVERY_DETAILS (cr=51402 r=0 w=0 time=15456581 us)
704891	INDEX RANGE SCAN WSH_DELIVERY_DETAILS_N8 (cr=2692 r=0 w=0 time=1677304 us)(object id 5124003)
704891	TABLE ACCESS BY INDEX ROWID MTL_SYSTEM_ITEMS_B (cr=2168345 r=460 w=0 time=26259264 us)
704891	INDEX RANGE SCAN MTL_SYSTEM_ITEMS_B_U1 (cr=1409795 r=213 w=0 time=15069327 us)(object id 38017)
644783	VIEW PUSHED PREDICATE (cr=2759460 r=0 w=0 time=30859890 us)
644783	NESTED LOOPS (cr=2759460 r=0 w=0 time=29181767 us)
704891	TABLE ACCESS BY INDEX ROWID FND_FLEX_VALUE_SETS (cr=1409784 r=0 w=0 time=11031089 us)
704891	INDEX UNIQUE SCAN FND_FLEX_VALUE_SETS_U2 (cr=704893 r=0 w=0 time=6257393 us)(object id 33768)
644783	TABLE ACCESS BY INDEX ROWID FND_FLEX_VALUES (cr=1349676 r=0 w=0 time=13839752 us)
644783	INDEX RANGE SCAN CCW_FND_FLEX_VALUES_N1 (cr=704893 r=0 w=0 time=8683861 us)(object id 5153800)
704891	TABLE ACCESS BY INDEX ROWID OE_ORDER_LINES_ALL (cr=2114675 r=27438 w=0 time=133292520 us)
704891	INDEX UNIQUE SCAN OE_ORDER_LINES_U1 (cr=1409784 r=2025 w=0 time=14863664 us)(object id 42102)
704891	TABLE ACCESS BY INDEX ROWID WSH_DELIVERY_ASSIGNMENTS (cr=2125380 r=6895 w=0 time=42893004 us)
704891	INDEX RANGE SCAN WSH_DELIVERY_ASSIGNMENTS_N3 (cr=1413108 r=2580 w=0 time=24181814 us)(object id 46295)
644538	TABLE ACCESS BY INDEX ROWID WSH_NEW_DELIVERIES (cr=2114675 r=6838 w=0 time=35307346 us)
704891	INDEX UNIQUE SCAN WSH_NEW_DELIVERIES_U1 (cr=1409784 r=362 w=0 time=7381948 us)(object id 46306)

---

# Statistics\_level

- ❑ So, statistics\_level parameter can be used to understand the step consuming much time.
- ❑ To improve performance we try to reduce the step consuming time.
- ❑ But, of course, functional knowledge of the SQL statement will come handy.

---

# All trace commands

❑ If you are tracing from your session, you might want to set these parameters too:

❑ Sets maximum file size to unlimited.

```
Alter session set max_dump_file_size=unlimited;
```

❑ This sets trace files to have file names easily distinguishable!

```
alter session set tracefile_identifier='riyaj';
```

❑ Print row source timing information.

```
alter session set Statistics_level=all;
```

```
exec dbms_session.session_trace_enable(waits => true);
```

---

---

# Agenda

- Tracing SQL execution
- Analyzing Trace files
- **Understanding Explain plans** —————→ **SQL Monitor**  
DBMS\_XPLAN
- Access, Joins, filters and unnest
- Writing optimal SQL
- Good hints and no-so good hints
- Effective indexing
- Partitioning for performance

---

## Static tracing

- ❑ Problem with `dbms_xplan` is that you need to wait for at least one execution of the cursor before you can get meaningful information.
- ❑ In Production environments, this is a problem for long running queries.
- ❑ Oracle 11g introduces Real time SQL monitoring feature.
- ❑ Execution step statistics are printed and can be queried even before the execution of the cursor. Truly a very useful feature indeed.

---

## Dbms\_sqltune

- ❑ SQL Monitoring is on if the `statistics_level` is set to `ALL` or `TYPICAL`. Of course, you would need diagnostic pack license.
- ❑ Long running statements are monitored. Statements running for more than 5 seconds or parallel executions qualify.
- ❑ `dbms_sqltune.report_sql_monitor` provides rich functionality to retrieve the statistics.
- ❑ There are few new views `v$sql_monitor`, `v$sql_plan_monitor` introduced and `dbms_sqltune` use these views to print readable output.

## Example

```
set lines 180
col rpt format a180
set long 1000000
select dbms_sqltune.report_sql_monitor('&sqlid') rpt from dual;
SQL Monitoring Report
```

SQL Text

```
-----
select count(a.n1), count(distinct(a.v1)) from rs.hugetable a , rs.hugetable b , rs.hugetable c
```

Error: ORA-1013

```
-----
ORA-01013: user requested cancel of current operation
```

Global Information

```
-----
Status           : DONE (ERROR)
Instance ID      : 1
Session          : SYS (48:21)
SQL ID           : 8jwwx25pq37j1
SQL Execution ID : 16777217
Execution Started : 03/17/2011 20:56:05
First Refresh Time : 03/17/2011 20:56:09
Last Refresh Time : 03/17/2011 20:56:59
Duration         : 54s
Module/Action    : sqlplus@solrac1 (TNS V1-V3)/-
Service         : SYS$USERS
Program         : sqlplus@solrac1 (TNS V1-V3)
Fetch Calls     : 1
```

# Example

...2

## Global Stats

```
=====
| Elapsed | Cpu  | IO   | Application | Other | Fetch | Buffer | Read | Read |
| Time(s) | Time(s) | waits(s) | waits(s)  | waits(s) | Calls | Gets  | Reqs | Bytes |
=====
|      54 |    33 |  4.51 |           0.00 |    16 |     1 | 26913 | 1718 | 210MB |
=====
```

## SQL Plan Monitoring Details (Plan Hash Value=1443272992)

```
=====
| Id |      Operation      | Name | Rows | Cost | Time | Start | Execs | Rows | Read | Read | Mem |
Activity | Activity Detail |      | (Estim) |      | Active(s) | Active |      | (Actual) | Reqs | Bytes | (Max) |
(%) | (# samples) |
=====
| 0 | SELECT STATEMENT |      |      |      |      |      | 1 |      |      |      |      |
| 1 |   SORT AGGREGATE |      | 1 |      |      |      | 1 |      |      |      |      |
| 2 |    VIEW          | VW_DAG_0 | 40293 | 6T |      |      | 1 |      |      |      |      |
|
...

```

---

# Agenda

- Tracing SQL execution
- Analyzing Trace files
- Understanding Explain plans
- Access, Joins, filters and unnest
- Writing optimal SQL
- Good hints and no-so good hints
- Effective indexing
- Partitioning for performance

# Index Unique Scan

- ❑ One row returned from the index key.

```
explain plan for select ename from demo1.emp where empno=:b1
```

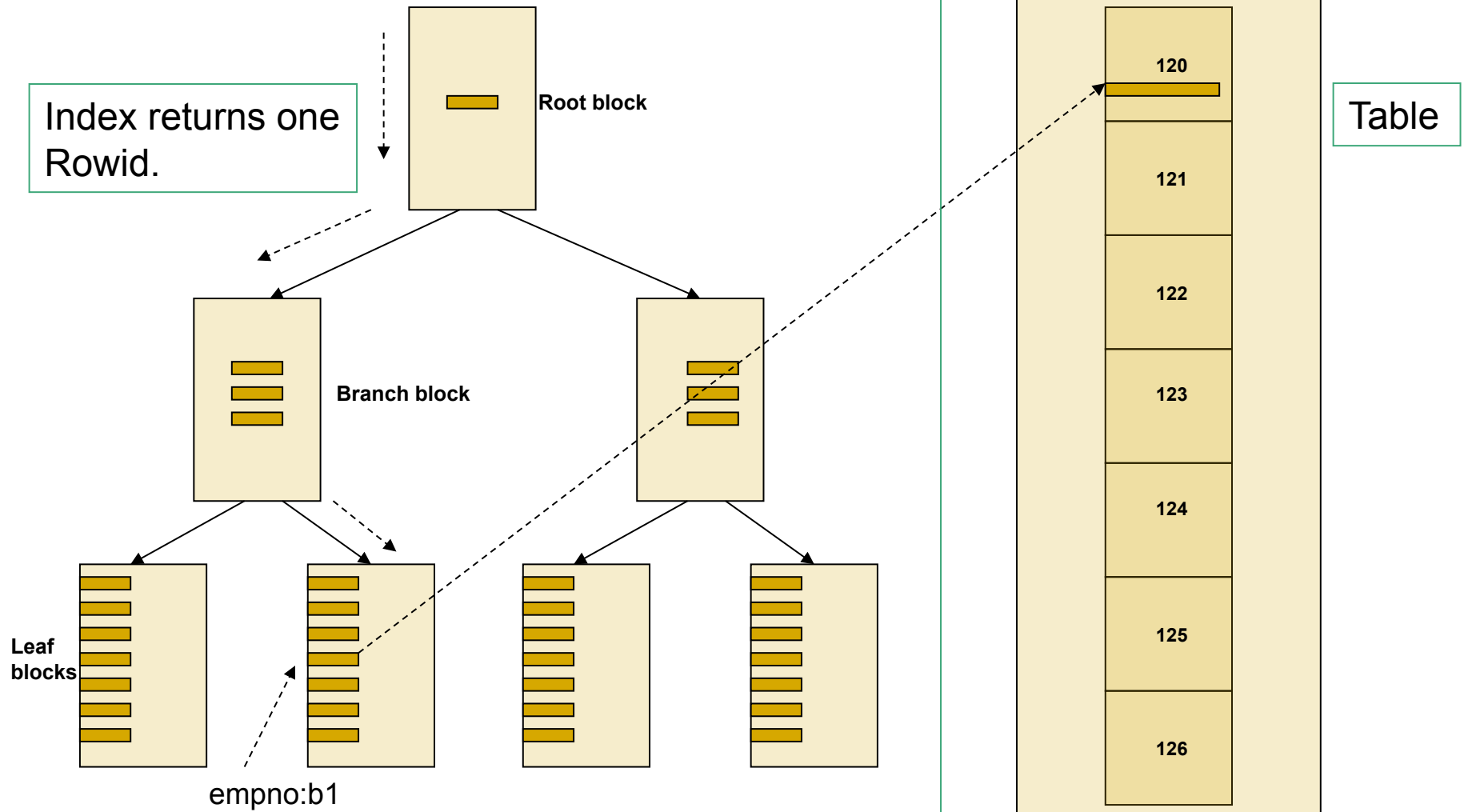
Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	10	1 (0)	00:00:01
1	TABLE ACCESS BY INDEX ROWID	EMP	1	10	1 (0)	00:00:01
* 2	INDEX UNIQUE SCAN	EMP_PK	1		0 (0)	00:00:01

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

```
2 - access("EMPNO"=TO_NUMBER(:B1))
```

- ❑ Usually, quite efficient since just one row returned from the index. Index must be unique index for this access path.

# Index unique scan



# Index Range Scan

□ Multiple rowids retrieved from the index. For every rowid returned, table block is accessed to retrieve other selected columns.

explain plan for select ename from demo1.emp where empno between :b1 and :b2;

```
-----  
| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |  
-----  
| 0 | SELECT STATEMENT | | 1 | 10 | 2 (0)| 00:00:01 |  
|* 1 | FILTER | | | | | |  
| 2 | TABLE ACCESS BY INDEX ROWID | EMP | 1 | 10 | 2 (0)| 00:00:01 |  
|* 3 | INDEX RANGE SCAN | EMP_PK | 2 | | 1 (0)| 00:00:01 |  
-----
```

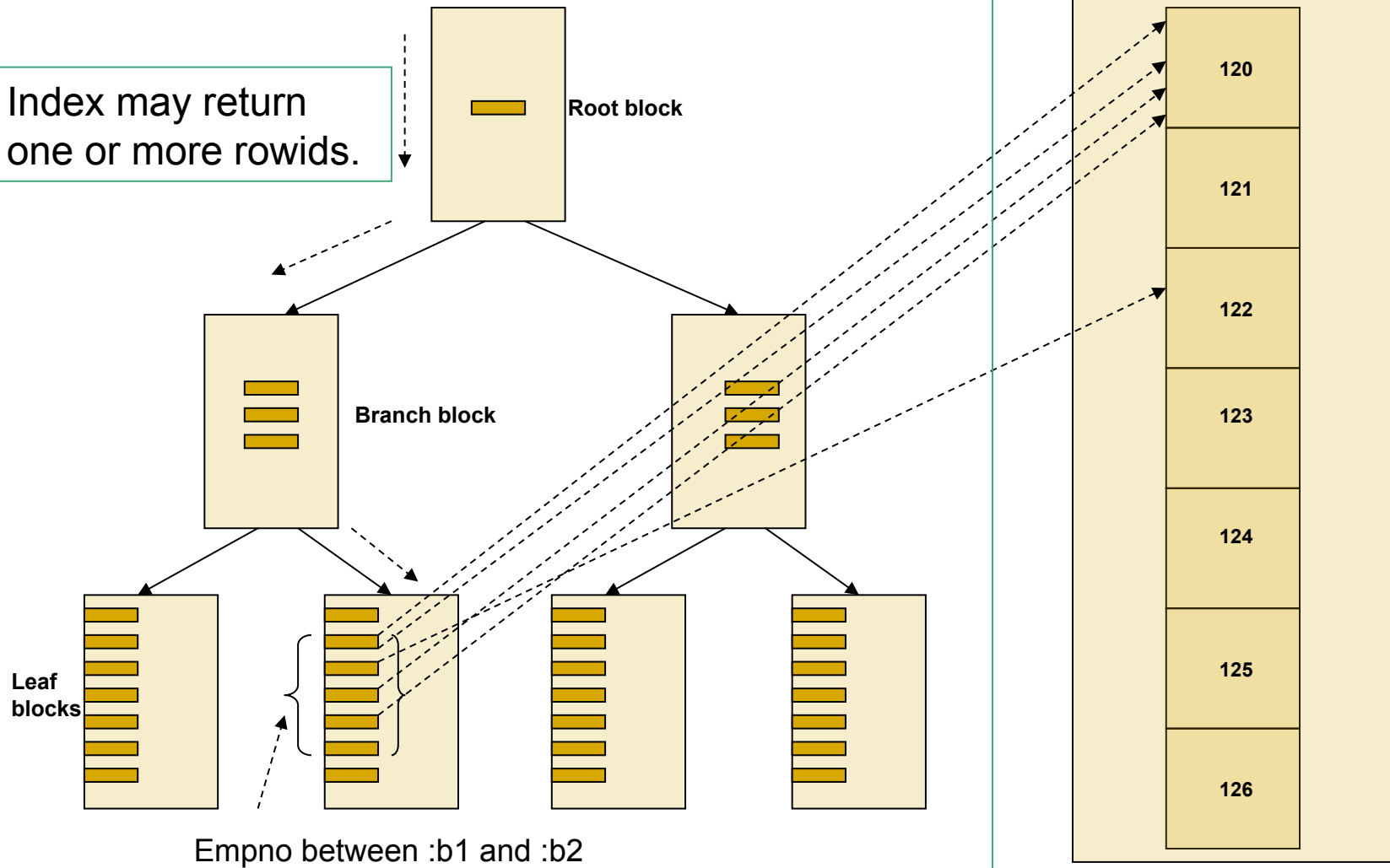
Predicate Information (identified by operation id):

- ```
-----  
1 - filter(TO_NUMBER(:B1)<=TO_NUMBER(:B2))  
3 - access("EMPNO">=TO_NUMBER(:B1) AND "EMPNO"<=TO_NUMBER(:B2))
```

# Index range scan

Data file

Index may return one or more rowids.



# Index Skip Scan

□ Index is searched with non-leading columns of the index.  
Generally, performs better if the number of distinct values for the leading columns are very few.

| Index  | Column      | Type           |
|--------|-------------|----------------|
| EMP_C2 | LOCATION_ID | 1 NUMBER(22)   |
|        | ENAME       | 2 VARCHAR2(10) |

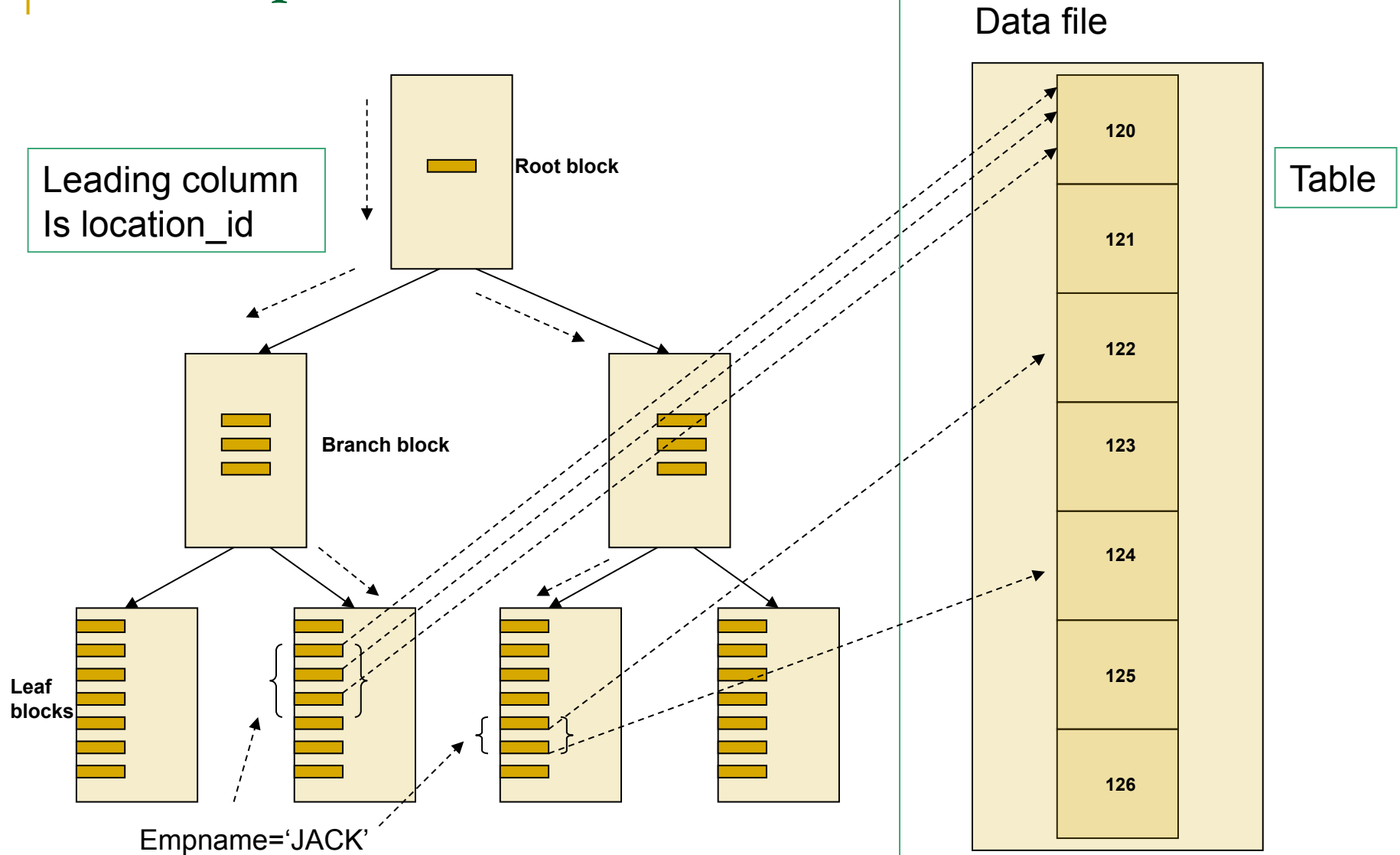
```
select * from emp a where ename='JACK';
```

| Id  | Operation                   | Name   | Rows | Bytes | Cost (%CPU) | Time     |
|-----|-----------------------------|--------|------|-------|-------------|----------|
| 0   | SELECT STATEMENT            |        | 1    | 43    | 3 (0)       | 00:00:01 |
| 1   | TABLE ACCESS BY INDEX ROWID | EMP    | 1    | 43    | 3 (0)       | 00:00:01 |
| * 2 | INDEX SKIP SCAN             | EMP_C2 | 1    |       | 2 (0)       | 00:00:01 |

Predicate Information (identified by operation id):

```
2 - access("ENAME"='JACK')
   filter("ENAME"='JACK')
```

# Index skip scan



---

# Joins

- ❑ Type of join operators
  - ❑ Equi-join
  - ❑ Outer join
  - ❑ Merge Join
  - ❑ Anti-join
  
- ❑ Type of join techniques in Oracle
  - ❑ Nested Loops join
  - ❑ Hash Join
  - ❑ Merge Join

---

# Nested loops join

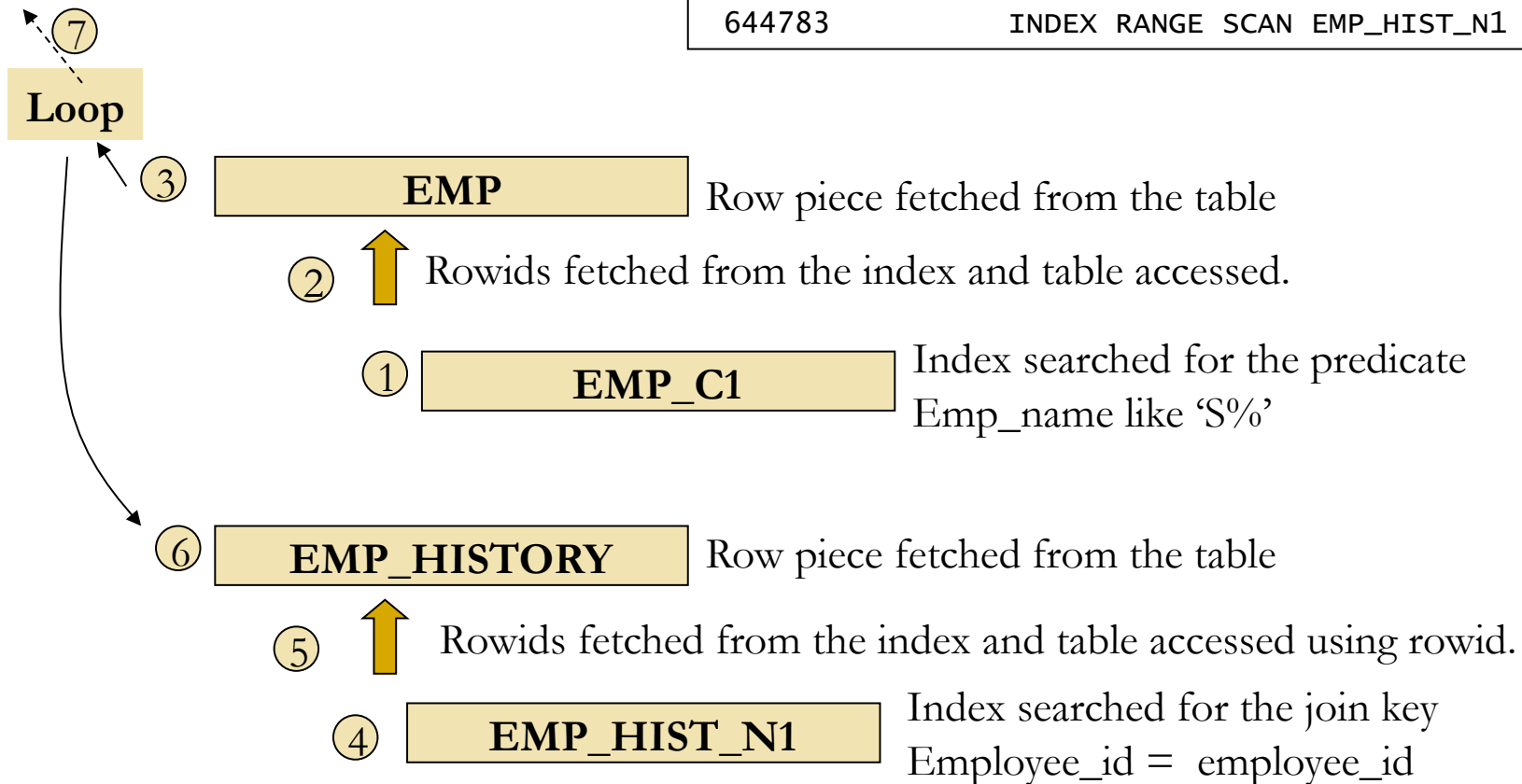
- ❑ For every row from outer row source, inner row source is probed for a matching row satisfying join key condition.

```
644783          NESTED LOOPS
704891          TABLE ACCESS BY INDEX ROWID EMP
704891          INDEX UNIQUE SCAN EMP_C1
644783          TABLE ACCESS BY INDEX ROWID EMP_HISTORY
644783          INDEX RANGE SCAN EMP_HIST_N1
```

- ❑ Generally, performs better if number of rows from the inner row source is much lower.
- ❑ Typically, OLTP uses this type of join.

# Nested loops join

|        |                                         |
|--------|-----------------------------------------|
| 644783 | NESTED LOOPS                            |
| 704891 | TABLE ACCESS BY INDEX ROWID EMP         |
| 704891 | INDEX UNIQUE SCAN EMP_C1                |
| 644783 | TABLE ACCESS BY INDEX ROWID EMP_HISTORY |
| 644783 | INDEX RANGE SCAN EMP_HIST_N1            |



# Hash join

- Rows from two row sources fetched and then joined.

```
explain plan for select t1.color, t2.color, t1.shape from
t1, t2 where t1.color_id = t2.color_id and t1.color=:b1 and t2.color=:b1
/
Explained.
```

```
SQL> select * from table (dbms_xplan.display);
```

```
Plan hash value: 2339531555
```

| Id  | Operation                   | Name     | Rows | Bytes | Cost (%CPU) | Time     |
|-----|-----------------------------|----------|------|-------|-------------|----------|
| 0   | SELECT STATEMENT            |          | 333  | 9990  | 8 (13)      | 00:00:01 |
| * 1 | HASH JOIN                   |          | 333  | 9990  | 8 (13)      | 00:00:01 |
| 2   | TABLE ACCESS BY INDEX ROWID | T1       | 333  | 6660  | 3 (0)       | 00:00:01 |
| * 3 | INDEX RANGE SCAN            | T1_COLOR | 333  |       | 1 (0)       | 00:00:01 |
| 4   | TABLE ACCESS BY INDEX ROWID | T2       | 500  | 5000  | 4 (0)       | 00:00:01 |
| * 5 | INDEX RANGE SCAN            | T2_COLOR | 500  |       | 2 (0)       | 00:00:01 |

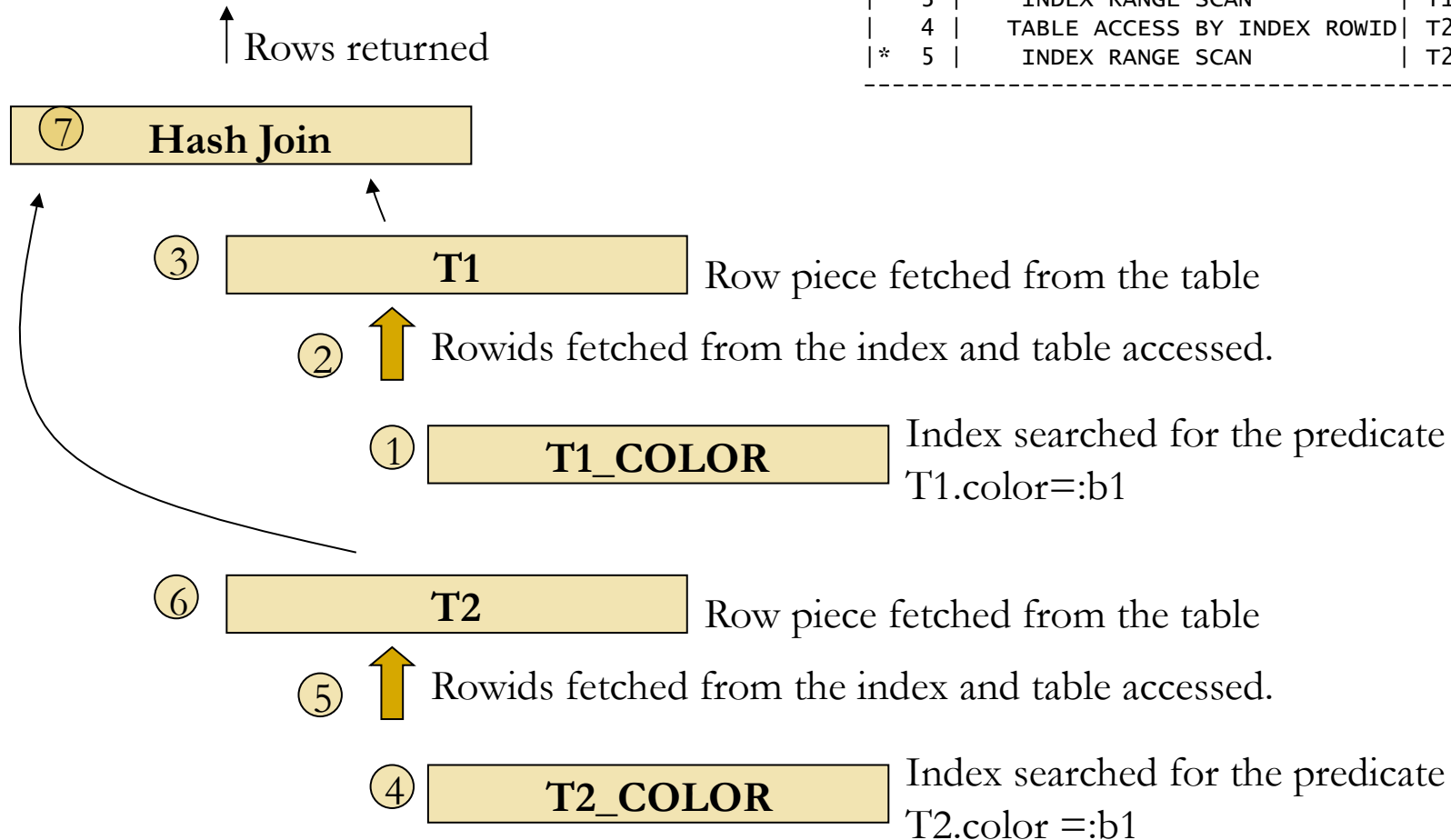
```
Predicate Information (identified by operation id):
```

- 1 - access("T1"."COLOR\_ID"="T2"."COLOR\_ID")
- 3 - access("T1"."COLOR"=:B1)
- 5 - access("T2"."COLOR"=:B1)

```
19 rows selected.
```

# Hash join

| Id  | Operation                   | Name     |
|-----|-----------------------------|----------|
| 0   | SELECT STATEMENT            |          |
| * 1 | HASH JOIN                   |          |
| 2   | TABLE ACCESS BY INDEX ROWID | T1       |
| * 3 | INDEX RANGE SCAN            | T1_COLOR |
| 4   | TABLE ACCESS BY INDEX ROWID | T2       |
| * 5 | INDEX RANGE SCAN            | T2_COLOR |



# Merge join

- Rows from two row sources are sorted and joined using merge join technique.

```
select /*+ use_merge (t1, t2) */ t1.color, t2.color, t1.shape from
t1, t2 where t1.color_id = t2.color_id and t1.color='black' and
t2.color='black'
```

| Id  | Operation                   | Name     | Rows | Bytes | Cost (%CPU) | Time     |
|-----|-----------------------------|----------|------|-------|-------------|----------|
| 0   | SELECT STATEMENT            |          |      |       | 10 (100)    |          |
| 1   | MERGE JOIN                  |          | 500  | 15000 | 10 (20)     | 00:00:01 |
| 2   | SORT JOIN                   |          | 500  | 5000  | 5 (20)      | 00:00:01 |
| 3   | TABLE ACCESS BY INDEX ROWID | T2       | 500  | 5000  | 4 (0)       | 00:00:01 |
| * 4 | INDEX RANGE SCAN            | T2_COLOR | 500  |       | 2 (0)       | 00:00:01 |
| * 5 | SORT JOIN                   |          | 900  | 18000 | 5 (20)      | 00:00:01 |
| * 6 | TABLE ACCESS FULL           | T1       | 900  | 18000 | 4 (0)       | 00:00:01 |

Predicate Information (identified by operation id):

```
4 - access("T2"."COLOR"='black')
5 - access("T1"."COLOR_ID"="T2"."COLOR_ID")
   filter("T1"."COLOR_ID"="T2"."COLOR_ID")
6 - filter("T1"."COLOR"='black')
```

# Merge join

Rows returned

| Id  | Operation                   | Name     |
|-----|-----------------------------|----------|
| 0   | SELECT STATEMENT            |          |
| * 1 | HASH JOIN                   |          |
| 2   | TABLE ACCESS BY INDEX ROWID | T1       |
| * 3 | INDEX RANGE SCAN            | T1_COLOR |
| 4   | TABLE ACCESS BY INDEX ROWID | T2       |
| * 5 | INDEX RANGE SCAN            | T2_COLOR |

⑦ Merge Join

⑥ SORT

⑤ ↑ Rows filtered

③ T2 Row piece fetched from the table

② ↑ Rowids fetched from the index and table accessed.

① T2\_COLOR Index searched for the predicate T1.color=:b1

⑥ SORT Rows are sorted

⑤ ↑ Rows filtered

④ T1 Index searched for the predicate T2.color =:b1

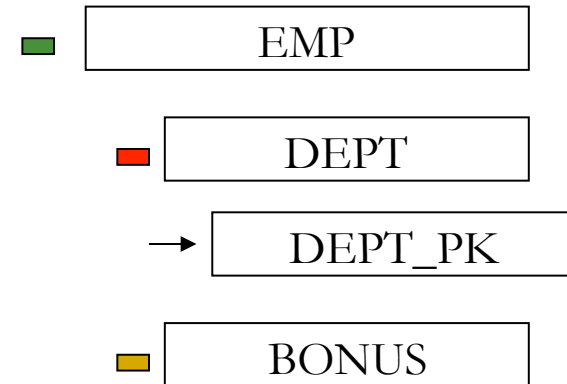
# FILTER step

```
select * from emp e
where
  e.deptno in
    (select deptno from dept where loc like 'A%') and
  e.ename in
    (select ename from bonus where sal >100000);
```

| Id  | Operation                   | Name    | Rows |
|-----|-----------------------------|---------|------|
| 0   | SELECT STATEMENT            |         |      |
| * 1 | FILTER                      |         |      |
| 2   | TABLE ACCESS FULL           | EMP     | 14   |
| * 3 | TABLE ACCESS BY INDEX ROWID | DEPT    | 1    |
| * 4 | INDEX RANGE SCAN            | DEPT_PK | 1    |
| * 5 | TABLE ACCESS FULL           | BONUS   | 1    |

Predicate Information (identified by operation id):

- 1 - filter(( IS NOT NULL AND IS NOT NULL))
- 3 - filter("LOC" LIKE 'A%')
- 4 - access("DEPTNO"=:B1)
- 5 - filter(("ENAME"=:B1 AND "SAL">100000))



For every row from EMP  
Filter checking with Dept\_pk and  
then DEPT table  
Filter checking with BONUS table

## Subquery unnesting

Subquery unnesting is an optimization step in which subqueries are unnested to a join.

```
select * from emp e
where
  e.deptno in
    (select deptno from dept where loc like 'A%') and
  e.ename in
    (select ename from bonus where sal >100000);
```

| Id  | Operation         | Name  | Rows | Bytes | Cost (%CPU) | Time     |
|-----|-------------------|-------|------|-------|-------------|----------|
| 0   | SELECT STATEMENT  |       |      |       | 7 (100)     |          |
| * 1 | HASH JOIN SEMI    |       | 5    | 675   | 7 (15)      | 00:00:01 |
| * 2 | HASH JOIN SEMI    |       | 5    | 575   | 5 (20)      | 00:00:01 |
| 3   | TABLE ACCESS FULL | EMP   | 14   | 1316  | 2 (0)       | 00:00:01 |
| * 4 | TABLE ACCESS FULL | DEPT  | 1    | 21    | 2 (0)       | 00:00:01 |
| * 5 | TABLE ACCESS FULL | BONUS | 1    | 20    | 2 (0)       | 00:00:01 |

---

# Summary

- ❑ There are different types of join techniques.
- ❑ There are no one join techniques superior to other join technique. If that is the case, Oracle product would not choose inferior techniques.
- ❑ Choosing a join method suitable for the SQL statement is also a necessary step of tuning that statement.

---

## **Common problem areas in an execution plan.**

---

# Identify plan issues

- ❑ There are couple of clues in the execution plan that can give you a hint about the root cause.
- ❑ Following few slides details these clues. But, this is not a complete list.
- ❑ It is probably a better idea to test your theory
  - ❑ by rewriting the query
  - ❑ by adjusting the execution plan with hints
  - ❑ by testing with a different database or different set of data.

# Cardinality feedback

- ❑ Cardinality is the measure of optimizer estimate for the number of rows returned in a step.
- ❑ In this case, optimizer estimates that step 6, accessing through index returned 20 rows.
- ❑ Step 4 reduced that just one row. So, you need to investigate to see why there is many rows estimated to be returned in that step.

```
Select * from table (dbms_xplan.display);
```

| Id  | Operation                   | Name                   | Rows | Bytes | Cost (%CPU) |
|-----|-----------------------------|------------------------|------|-------|-------------|
| 0   | SELECT STATEMENT            |                        | 1    | 214   | 9 (23)      |
| 1   | SORT ORDER BY               |                        | 1    | 214   | 9 (23)      |
| * 2 | TABLE ACCESS BY INDEX ROWID | RCV_STAGING_LINES      | 1    | 155   | 4 (25)      |
| 3   | NESTED LOOPS                |                        | 1    | 214   | 8 (13)      |
| * 4 | TABLE ACCESS BY INDEX ROWID | RCV_STAGING_HEADERS    | 1    | 59    | 5 (20)      |
| * 5 | INDEX RANGE SCAN            | RCV_STAGING_HEADERS_N5 | 20   |       | 4 (25)      |
| * 6 | INDEX RANGE SCAN            | RCV_STAGING_LINES_N3   | 1    |       | 3 (34)      |

# Cardinality feedback (2)

(2)

- ❑ Match the run time statistics and the execution plan. Identify the difference in the estimate and actual statistics.
- ❑ In this example below, optimizer estimates are off, especially for step 5..

```
Rows      Row Source Operation
-----
0         TABLE ACCESS BY INDEX ROWID RCV_STAGING_LINES
0         NESTED LOOPS
0         NESTED LOOPS
14        TABLE ACCESS BY INDEX ROWID RCV_STAGING_HEADERS
2608158   INDEX RANGE SCAN RCV_STAGING_HEADERS_N5
14        INDEX RANGE SCAN RCV_STAGING_LINES_N3
```

```
-----
| Id | Operation                               | Name                | Rows | Bytes | Cost (%CPU)|
-----
0	SELECT STATEMENT		1	214	9 (23)
1	SORT ORDER BY		1	214	9 (23)
* 2	TABLE ACCESS BY INDEX ROWID	RCV_STAGING_LINES	1	155	4 (25)
3	NESTED LOOPS		1	214	8 (13)
* 4	TABLE ACCESS BY INDEX ROWID	RCV_STAGING_HEADERS	1	59	5 (20)
* 5	INDEX RANGE SCAN	RCV_STAGING_HEADERS_N5	20		4 (25)
* 6	INDEX RANGE SCAN	RCV_STAGING_LINES_N3	1		3 (34)
-----
```

# Leading table

- ❑ In most cases, if the optimizer chooses correct starting table, then the performance of that SQL statement will be tolerable.
- ❑ Try to understand why the optimizer did not chose correct starting table.
- ❑ In this example, leading table is chosen unwisely.

| Rows     | Row Source Operation                                                                |
|----------|-------------------------------------------------------------------------------------|
| 1000     | SORT GROUP BY (cr=33914438 r=0 w=0 time=386540124 us)                               |
| 1000     | NESTED LOOPS (cr=33914438 r=0 w=0 time=386162642 us)                                |
| 16540500 | TABLE ACCESS BY INDEX ROWID OBJ#(38049)(cr=831438 r=0 w=0 time=146900634us )        |
| 16540500 | INDEX RANGE SCAN OBJ#(149297) (cr=154584 r=0 w=0 time=25026467 us)(obj 149297)      |
| 1000     | INDEX RANGE SCAN OBJ#(5743997) (cr=33083000 r=0 w=0 time=202120686 us)(obj 5743997) |

# Leading table

(2)

- ❑ With a leading hint, SQL was tuned to start from object id 5743997, which was a temporary table.
- ❑ Performance of that SQL is much better with temporary table chosen as a leading table.

| ROWS | Row Source Operation                                                              |
|------|-----------------------------------------------------------------------------------|
| 0    | SORT GROUP BY (cr=3067 r=0 w=0 time=38663 us)                                     |
| 0    | TABLE ACCESS BY INDEX ROWID OBJ#(38049) (cr=3067 r=0 w=0 time=26516 us)           |
| 1000 | NESTED LOOPS (cr=3067 r=0 w=0 time=24509 us)                                      |
| 0    | INDEX RANGE SCAN OBJ#(5743997) (cr=3067 r=0 w=0 time=23379 us)(object id 5743997) |
| 0    | INDEX RANGE SCAN OBJ#(1606420) (object id 1606420)                                |

# Cartesian merge join

□ Cartesian merge join (CMJ) is a special case of merge join. It is a merge join with no join predicates between two row sources.

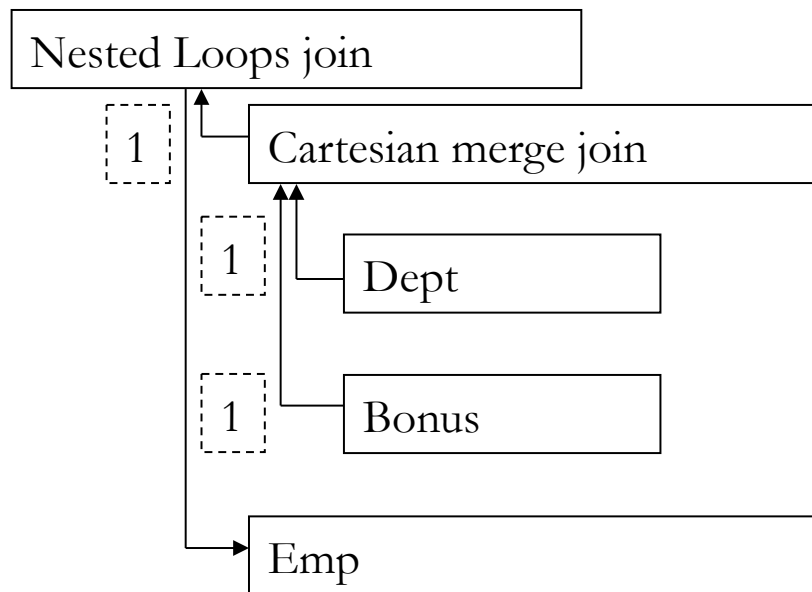
```
select * from emp e where
e.deptno in (select deptno from dept where loc like 'A%') and
e.ename in (select ename from bonus where sal >100000);
```

| Id  | Operation                   | Name   | E-Rows |
|-----|-----------------------------|--------|--------|
| * 1 | TABLE ACCESS BY INDEX ROWID | EMP    | 5      |
| 2   | NESTED LOOPS                |        | 1      |
| 3   | MERGE JOIN CARTESIAN        |        | 1      |
| 4   | SORT UNIQUE                 |        | 1      |
| * 5 | TABLE ACCESS FULL           | BONUS  | 1      |
| 6   | BUFFER SORT                 |        | 1      |
| 7   | SORT UNIQUE                 |        | 1      |
| * 8 | TABLE ACCESS FULL           | DEPT   | 1      |
| * 9 | INDEX RANGE SCAN            | EMP_FK | 5      |

```
1 - filter("E"."ENAME"="ENAME")
5 - filter("SAL">100000)
8 - filter("LOC" LIKE 'A%')
9 - access("E"."DEPTNO"="DEPTNO")
```

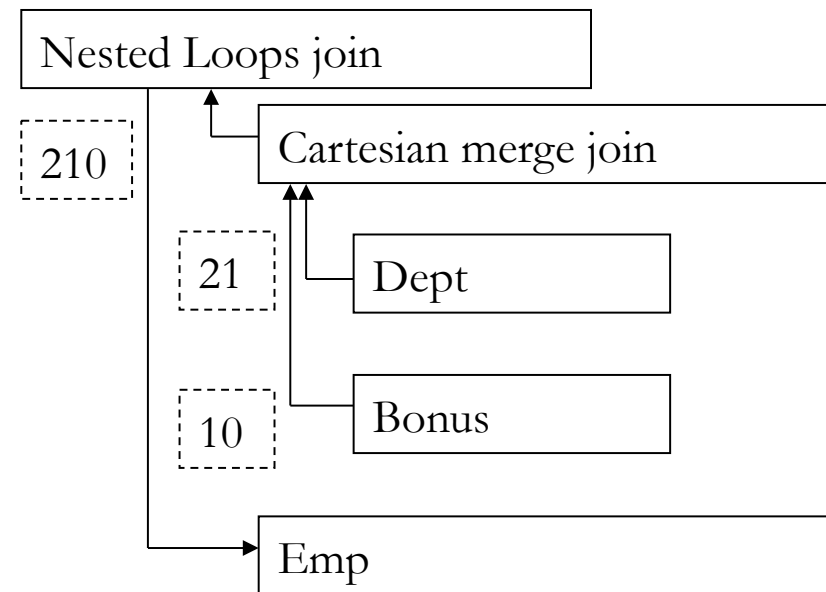
# CMJ - Analysis

Estimate



Emp estimated to be scanned once.

Actual



Emp scanned 210 times

# CMJ & cardinality

❑ If the row source cardinality is incorrectly estimated as 1, then SQL statements with cartesian merge join can be a performance bottleneck.

❑ In some cases, it may be necessary to avoid this unnesting with an hint.

```
select * from emp e where  
e.deptno in (select /*+ no_unnest */ deptno from dept where loc like 'A%') and  
e.ename in (select ename from bonus where sal >100000);
```

| Id  | Operation                   | Name    | E-ROWS |
|-----|-----------------------------|---------|--------|
| * 1 | FILTER                      |         |        |
| * 2 | HASH JOIN SEMI              |         | 1      |
| 3   | TABLE ACCESS FULL           | EMP     | 14     |
| * 4 | TABLE ACCESS FULL           | BONUS   | 1      |
| * 5 | TABLE ACCESS BY INDEX ROWID | DEPT    | 1      |
| * 6 | INDEX RANGE SCAN            | DEPT_PK | 1      |

# View merging

- ❑ Normal views do not store data, just the definition of the view.
- ❑ View definitions are merged with calling query or DML statement and merged to create final execution plan.

```
create or replace view bonus_simple_vw as select job, sal from bonus;
```

```
select e.ename, e.job, e.sal, b.sal from emp e, bonus_simple_vw b where e.job=b.job;
```

| Id  | Operation         | Name  | E-ROWS |
|-----|-------------------|-------|--------|
| 1   | NESTED LOOPS      |       | 1      |
| 2   | TABLE ACCESS FULL | EMP   | 14     |
| * 3 | TABLE ACCESS FULL | BONUS | 1      |

Predicate Information (identified by operation id):

```
3 - filter("E"."JOB"="JOB")
```

# View merging

- ❑ But, some views are not merge-able.
- ❑ For example, following complex view is not merged. Note the keyword *VIEW* in the execution plan.

```
create or replace view bonus_vw as
  select job, avg(sal) avg_sal from bonus group by job;

select e.ename, e.job,e.sal, b.avg_sal from emp e, bonus_vw b where e.job=b.job
```

| Id  | Operation         | Name     | E-Rows | OMem | 1Mem | Used-Mem |
|-----|-------------------|----------|--------|------|------|----------|
| 1   | NESTED LOOPS      |          | 3      |      |      |          |
| 2   | TABLE ACCESS FULL | EMP      | 14     |      |      |          |
| * 3 | VIEW              | BONUS_VW | 1      |      |      |          |
| 4   | SORT GROUP BY     |          | 1      | 1024 | 1024 |          |
| 5   | TABLE ACCESS FULL | BONUS    | 1      |      |      |          |

Predicate Information (identified by operation id):

```
3 - filter("E"."JOB"="B"."JOB")
```

---

# Non-mergeable

- ❑ Usually, complex views with group by, union all, union, function calls are not merge-able.
- ❑ In some cases, it is better to merge the views and in some cases, it is not.
- ❑ This depends upon how optimized each execution of the view is.
- ❑ Use tkprof output file to understand how much time is spent in the step joining with non-merged view. Decide whether to merge or not based upon that.

---

## Indexing, partitioning and more

---

## Choose index wisely

- ❑ Leading column of indices should contain most common predicates.

```
explain plan for select * from emp where ename ='JACOB' and hiredate > sysdate-365;
```

```
-----  
| Id | Operation          | Name | Rows | Bytes | Cost (%CPU)| Time     |  
-----  
|  0 | SELECT STATEMENT  |      |    1 |    40 |    2   (0)| 00:00:01 |  
|*  1 | TABLE ACCESS FULL| EMP  |    1 |    40 |    2   (0)| 00:00:01 |  
-----
```

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

```
1 - filter("ENAME"='JACOB' AND "HIREDATE">SYSDATE@!-365)
```

- ❑ In the example above, creating an index on ename would be helpful.

## Join keys for indices

□ For frequently executed queries, try to see if indices can be created on join key columns.

```
explain plan for select * from emp e where
  e.deptno in
  (select /*+ qb_name (sel_dept) */ deptno from dept where loc like 'A%') and
  e.ename in
  (select /*+ qb_name (sel_bonus) */ ename from bonus where sal >100000)
/
```

| Id  | Operation         | Name  | Rows | Bytes | Cost (%CPU) | Time     |
|-----|-------------------|-------|------|-------|-------------|----------|
| 0   | SELECT STATEMENT  |       | 1    | 71    | 7 (15)      | 00:00:01 |
| * 1 | HASH JOIN SEMI    |       | 1    | 71    | 7 (15)      | 00:00:01 |
| * 2 | HASH JOIN SEMI    |       | 5    | 255   | 5 (20)      | 00:00:01 |
| 3   | TABLE ACCESS FULL | EMP   | 14   | 560   | 2 (0)       | 00:00:01 |
| * 4 | TABLE ACCESS FULL | DEPT  | 1    | 11    | 2 (0)       | 00:00:01 |
| * 5 | TABLE ACCESS FULL | BONUS | 1    | 20    | 2 (0)       | 00:00:01 |

Predicate Information (identified by operation id):

```
1 - access("E"."ENAME"="ENAME")
2 - access("E"."DEPTNO"="DEPTNO")
4 - filter("LOC" LIKE 'A%')
5 - filter("SAL">100000)
```

Creating indices on these join key columns would be beneficial.

## Column selectivity

❑ Selectivity of the column indices how many rows will be returned from the table.

| Table Name                       | Number of Rows | Blocks     | Empty Blocks | Average Space |
|----------------------------------|----------------|------------|--------------|---------------|
| MTL_MATERIAL_TRANSACTIONSACTIONS | 170,392,833    | 13,551,182 | 0            | 0             |

| Column Name       | Column Details      | Distinct Values |
|-------------------|---------------------|-----------------|
| ...               |                     |                 |
| ORGANIZATION_ID   | NUMBER(22) NOT NULL | 2,589           |
| INVENTORY_ITEM_ID | NUMBER(22) NOT NULL | 7,703           |
| ...               |                     |                 |

Where organization\_id = :b1  
and inventory\_item\_id=:b2

❑ For organization\_id alone in the index, approximately, each key entry will return 65,814 rows:  $170392833/2589 = 65814$

❑ For organization\_id , inventory\_item\_id; Each combination of key entry will return just 8 rows:  $170392833/(2589*7703) = 8$

## Avoid updated columns

- ❑ Do not index columns that are updated heavily.
- ❑ Updates to indexed columns are equivalent of a delete and insert at index level.
- ❑ In this example, quantity is a good candidate column, but updated heavily. So, that column should not be indexed.

```
explain plan for
select attribute1, quantity, quantity_cancelled, quantity_delivered from
  PO_REQUISITION_LINES_ALL where item_id=:b1 and quantity >100
/
```

| Id  | Operation                   | Name                     | Rows |
|-----|-----------------------------|--------------------------|------|
| 0   | SELECT STATEMENT            |                          | 1545 |
| * 1 | TABLE ACCESS BY INDEX ROWID | PO_REQUISITION_LINES_ALL | 1545 |
| * 2 | INDEX RANGE SCAN            | PO_REQUISITION_LINES_N7  | 1545 |

```
1 - filter("QUANTITY">100)
2 - access("ITEM_ID"=TO_NUMBER(:B1))
```

---

## Other index types

- ❑ There are more index types than just b-tree indices
  - ❑ bitmap indices
  - ❑ Index organized tables
  - ❑ Partitioned indices etc.
  
- ❑ Consider activity while choosing index. For example, bitmap indices are useful only for read only tables (like data warehouse tables).
  
- ❑ Index Organized tables are suitable for tables which are accessed just using just leading columns of the table.

# Index efficiency?

```
DESC T1
...
N2      NUMBER
N3      NUMBER
...

select count(distinct(n2)) n2_count,
       count(distinct(n3)) n3_count
from t1;

   N2_COUNT  N3_COUNT
-----
          62         63
```

1 row selected.

```
create index t1_n2 on t1(n2);
create index t1_n3 on t1(n3);

Begin
  dbms_Stats.gather_table_stats
  ( user, 't1' ,
    estimate_percent =>100,
    cascade =>true);
End;
/
```

## Why would the optimizer not choose t1\_n2 index?

```
explain plan for
select n1 from t1 where n2=:b1;
```

| Id  | Operation         | Name | Rows | Cost (%CPU) |
|-----|-------------------|------|------|-------------|
| 0   | SELECT STATEMENT  |      | 62   | 19 (0)      |
| * 1 | TABLE ACCESS FULL | T1   | 62   | 19 (0)      |

Predicate Information (identified by operation id):

1 - filter("N2"=TO\_NUMBER(:B1))

```
explain plan for
select n1 from t1 where n3=:b1;
```

| Id  | Operation                   | Name  | Rows | Cost (%CPU) |
|-----|-----------------------------|-------|------|-------------|
| 0   | SELECT STATEMENT            |       | 61   | 3 (0)       |
| 1   | TABLE ACCESS BY INDEX ROWID | T1    | 61   | 3 (0)       |
| * 2 | INDEX RANGE SCAN            | T1_N3 | 61   | 1 (0)       |

Predicate Information (identified by operation id):

2 - access("N3"=TO\_NUMBER(:B1))

## Index properties..

```
drop table t1;
create table t1
(n1 number,
 n2 number,
 n3 number,
 v1 varchar2(100) );

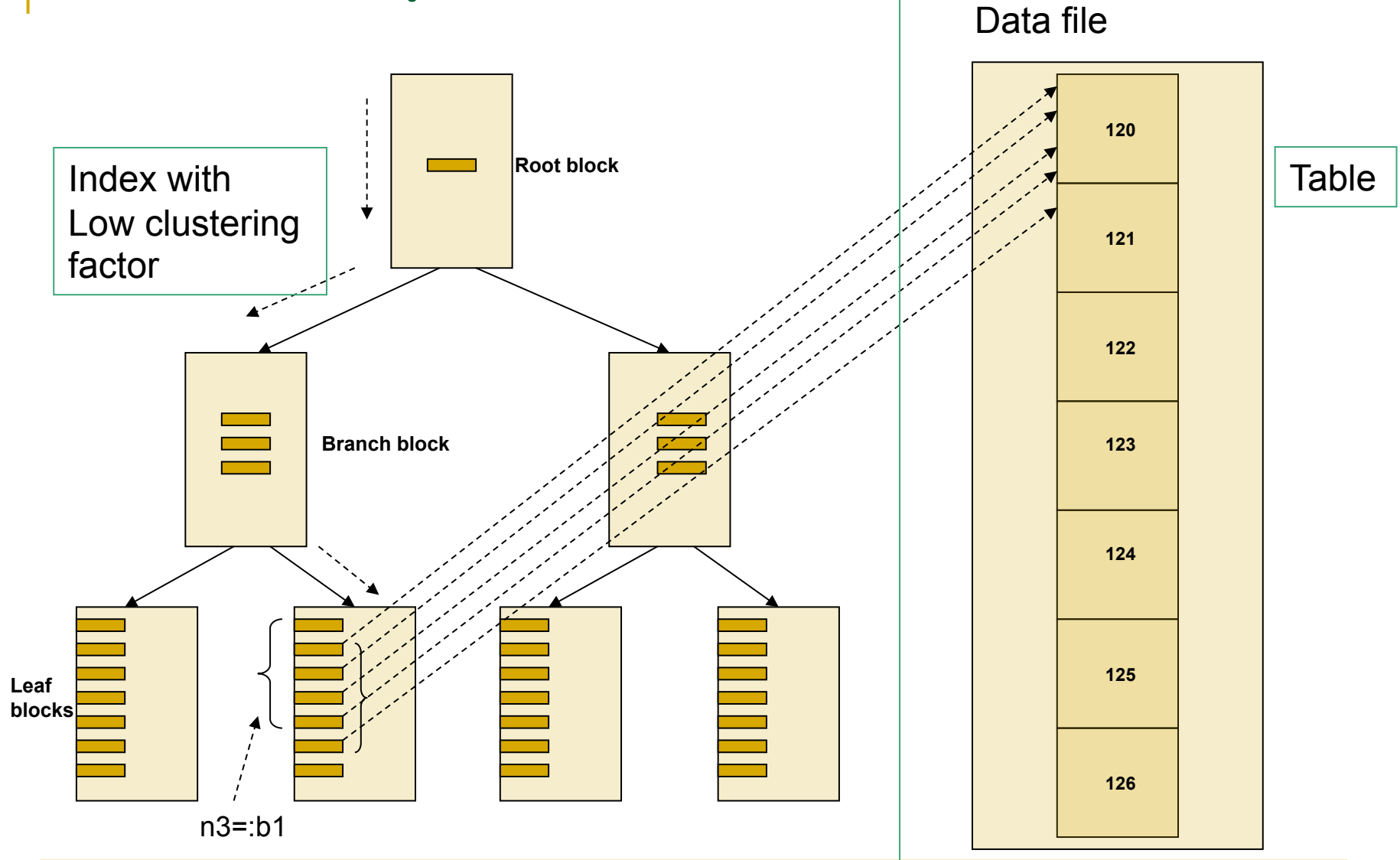
insert into t1
select l1,
       mod(l1, 62) n2,
       round (l1/62) n3,
       lpad(l1, 100,'x') v1
from
(select level l1 from dual
 connect by level
 <=3844);

commit;
```

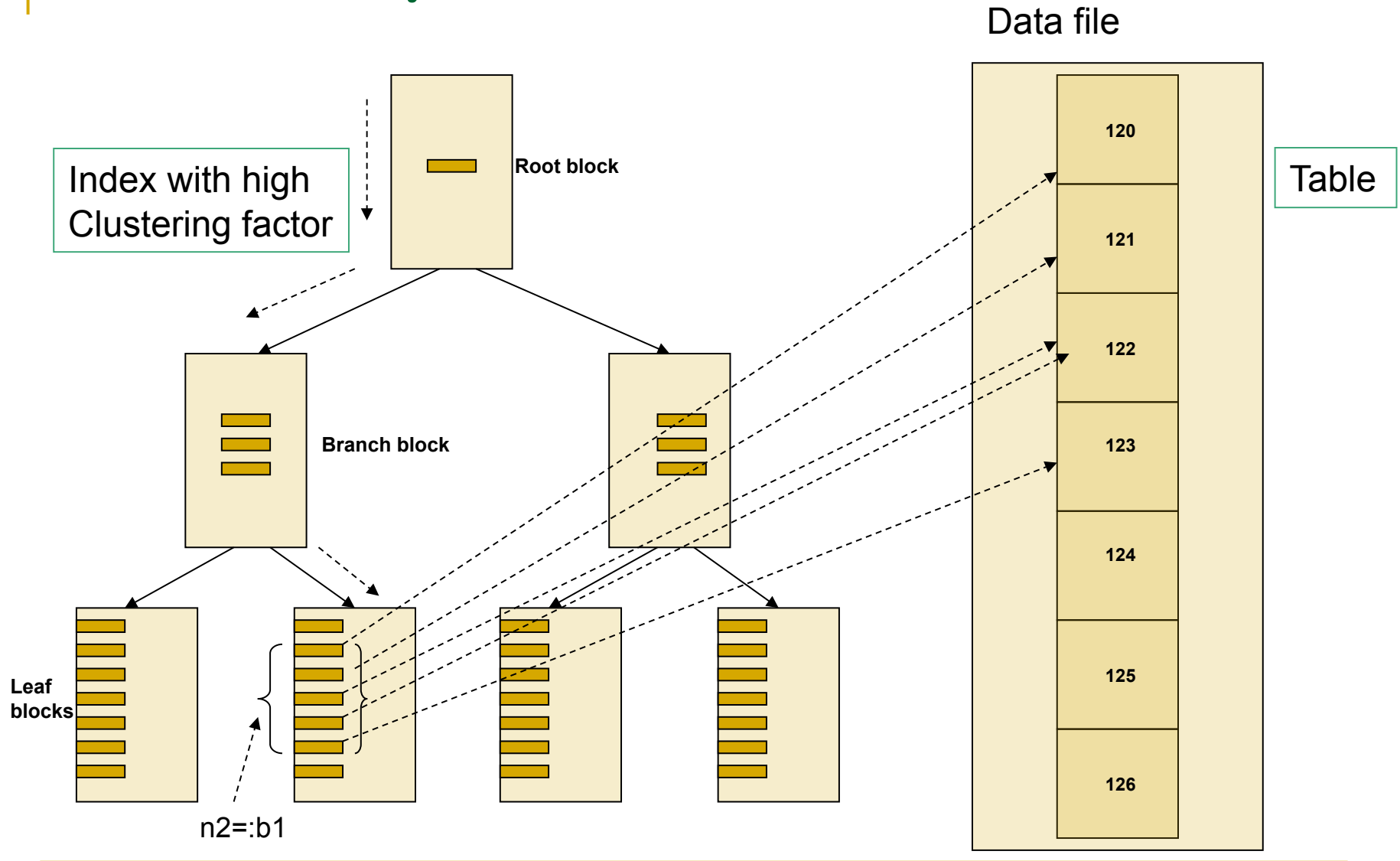
| Property            | T1_N2 | T1_N3 |
|---------------------|-------|-------|
| Unique              | No    | No    |
| Level               | 1     | 1     |
| Leaf blocks         | 8     | 8     |
| Distinct keys       | 62    | 63    |
| # of rows           | 3844  | 3844  |
| Avg. leaf block/key | 1     | 1     |
| Avg. datablock/key  | 61    | 1     |
| Clustering factor   | 3843  | 78    |



# Index efficiency



# Index efficiency



---

## Index efficiency

- Optimizer did not choose index on n2 since clustering factor of that index is very high.
- Accessing table through the index will be costlier due to visits to table blocks can increase I/O.
- Avoiding table blocks by adding more columns to that index might be of help.

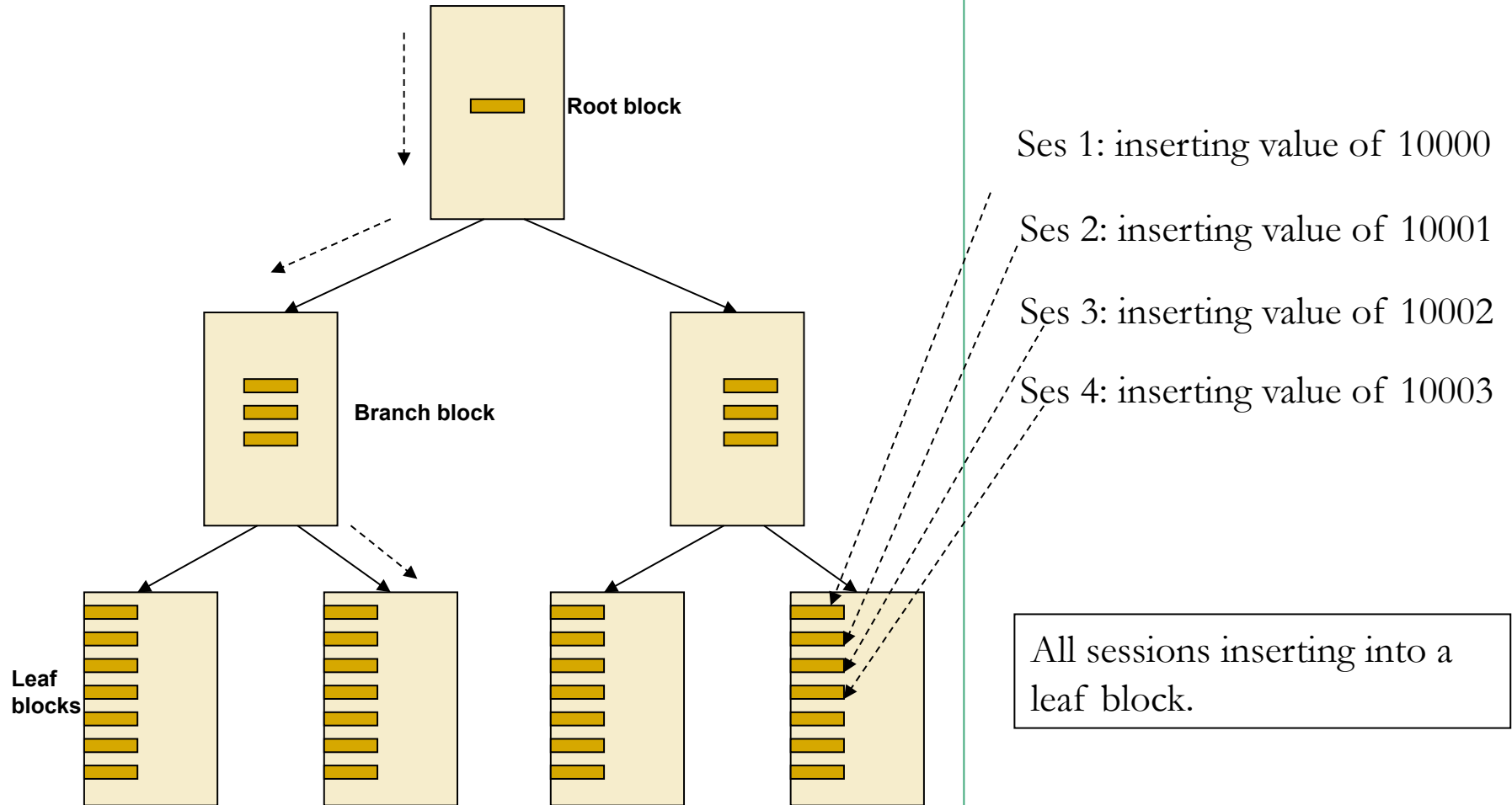
---

## Partitioning

- ❑ Consider table partitioning for many reasons:
  - ❑ Easy archiving of data. Example: gl\_balances on period\_name
  
- ❑ Performance of table scan. For example, report SQLs accessing just last period on gl\_balances table.
  
- ❑ Partition to improve scalability of the table. Hash partition the table inserted aggressively with primary key/unique keys.

# Unscalable index

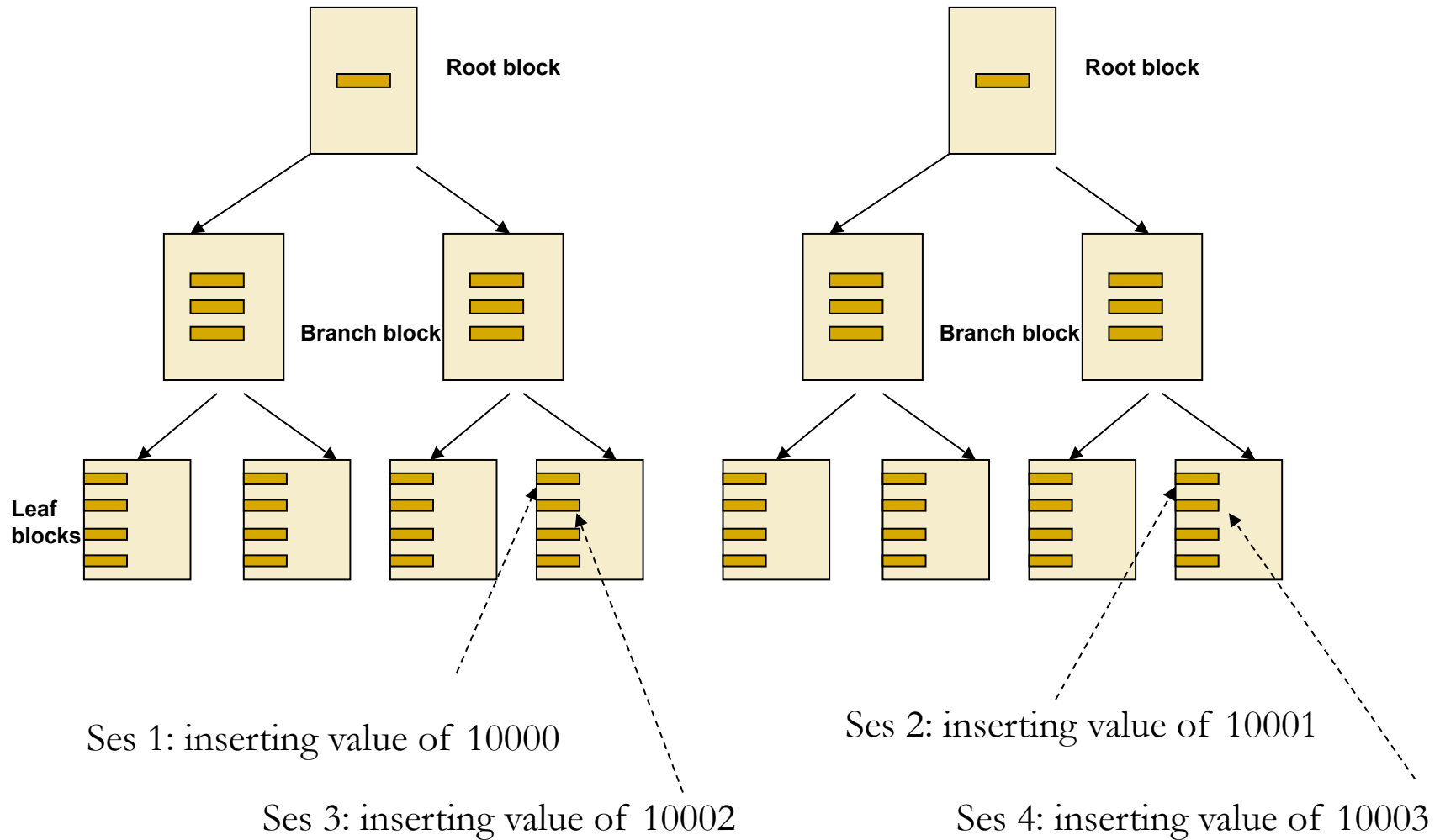
Primary key index on line\_id



# Hash partitioning

Primary key index on line\_id

All sessions inserting into a leaf block.



Thank you for attending!

If you like this presentation, you will love my  
upcoming intensive RAC webinar in June.

Watch for updates in:

[www.tanelpoder.com](http://www.tanelpoder.com)

[Orainternals.wordpress.com](http://Orainternals.wordpress.com)

