



# Leaping Tall Buildings with SQL

**Ric Van Dyke**

**Sr. DBA, Zione Solutions**

**Oracle Ace**

**SQL**

**MAY 10 – 14, 2021**

**[WWW.NEOOUG.ORG/GLOC](http://WWW.NEOOUG.ORG/GLOC)**

- ❑ Core optimization Guidelines
  - Explain plan vs Execution plan
- ❑ Getting the plan
  - Reading the plan
- ❑ Indexes
  - HINTs
- ❑ Common Table Expressions (CTEs)
  - Exadata

# Who is Ric?

- Oracle Ace
- Using Oracle since version 5
- Currently Sr. DBA at Zione Solutions
- Prior experience:
  - Started way back at Ford Motor Credit
    - Developer in Forms 2.3
    - DBA (Versions 5, 6, and 7)
  - Worked for Oracle for 10 years
    - Core DBA Senior Principal instructor
    - Education Manger, central region
    - ATS Technical Manger, north central region
  - Director of Education at Hotsos



# SQL Optimization Guidelines

- Let the numbers lead you to where the problem is.
- The goal for query optimization is to reduce the time it takes for a query to finish.
- What helps one query may not help (and even hurt) another query.
- Never assume anything. Without run time stats you are just guessing as to what a problem might be.
- Make sure the query is doing only what it needs to do for the business question being asked.





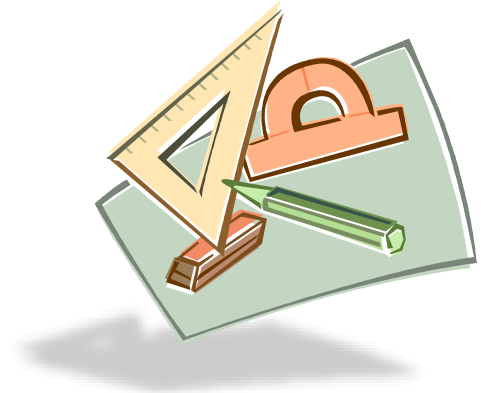
# Basics of SQL Optimization

- Finding why a SQL statement is slow can be easy.
  - Find the section of the plan taking up the most time.
- Fixing the issue may not be so easy.
  - Make that section "go away" (Don't do it).
  - Find a way to do that faster.
- Other times a rewrite is needed.
  - Tune the question, not the query.
  - The plan might look "OK", but is it doing more than it needs to?
  - You always know something the optimizer doesn't.



# The Plan

- It's all about the plan, the execution plan.
- The *explain* plan is a "guess".
- The *execution* plan is what really happened.
- To optimize SQL you need the execution plan with stats.
- With this you know where the problem is.



# Finding what is slow

- Use SQL Monitor
  - Best way to see the execution plan
  - All the details you need
  - Requires the Diagnostics and Tuning packs
- Use `DBMS_XPLAN.DISPLAY_CURSOR`
  - Works quite well
  - Gives you the core details you need
  - No required packs to use

# SQL Monitor

- Use the plan statistic tab to lead you to the problem
- These are the columns most useful to lead you to an issue
- Do not use **COST** as a tuning metric

OEM:

Operation	Name	Line ID	Estimated Rows	Actual Rows	Activity %	Timeline(813s)	Executions
-----------	------	---------	----------------	-------------	------------	----------------	------------

SQL Developer:

Operation	Object Name	Line	Plan Rows	Actual Rows	Timeline	Activity %	Executions
-----------	-------------	------	-----------	-------------	----------	------------	------------



# Columns of interest



- **OPERATION** – What happened
- **NAME** – Who it happened to
- **LINE ID** – Line in the plan, has nothing to do with the statement lines
- **ESTIMATED/PLAN ROWS** – How many rows were guessed to come back
- **ACTUAL ROWS** – How many rows really did come back
- **ACTIVITY %** – How much of the over all active did this step account for
- **TIMELINE** – Time this step was active
- **EXECUTIONS** – How many times did this step fire off

# SQL Developer – SQL Monitor



Operation	Line	Object Name	Executions	Plan Rows	Actual Rows	Timeline	Activity %
▼ SELECT STATEMENT	0		1		1	<input type="text"/>	
▼ SORT (AGGREGATE)	1		1	1	1	<input type="text"/>	
TABLE ACCESS (STORAGE FULL)	2	AFTSB1	1	305	5925	<input type="text"/>	
▼ HASH JOIN (RIGHT SEMI)	3		1	465K	896	<input type="text"/>	
INDEX (STORAGE FAST FULL SCAN)	4	IOT_PK	1	68K	68K	<input type="text"/>	
▼ HASH JOIN	5		1	843K	896	<input type="text"/>	
▼ JOIN FILTER (CREATE)	6	:BF0000	1	48	48	<input type="text"/>	
TABLE ACCESS (STORAGE FULL)	7	ALLUSERS_TAB	1	48	48	<input type="text"/>	
▼ JOIN FILTER (USE)	8	:BF0000	1	1718K	896	<input type="text"/>	
TABLE ACCESS (STORAGE FULL)	9	BIG_TAB	1	1718K	896	<input type="text"/>	

This query was run on a 21c Autonomous Cloud Database, the tables are small.

It took about 4 seconds of run time over all, the database time was 55ms.

Clearly not much to work on for this query on this machine.

# SQL Developer – SQL Monitor



Operation	Line	Object Name	Access Predicates	Filter Predicates
▼ SELECT STATEMENT	0			
▼ SORT (AGGREGATE)	1			
TABLE ACCESS (STORAGE FULL)	2	AFTSB1	"DATA_OBJECT_ID" IS NOT NULL	("DATA_OBJECT_ID" IS NOT NULL AN...
▼ HASH JOIN (RIGHT SEMI)	3		"IT"."USERNAME"="BT"."OWNER"	
INDEX (STORAGE FAST FULL SCAN)	4	IOT_PK		
▼ HASH JOIN	5		"BT"."OWNER"="AT"."USERNAME"	
▼ JOIN FILTER (CREATE)	6	:BF0000		
TABLE ACCESS (STORAGE FULL)	7	ALLUSERS_TAB	"AT"."USER_ID"<1000	"AT"."USER_ID"<1000
▼ JOIN FILTER (USE)	8	:BF0000		
TABLE ACCESS (STORAGE FULL)	9	BIG_TAB	("OBJECT_TYPE"=SYNONYM' AND SY...	("OBJECT_TYPE"=SYNONYM' AND S...

- From this screen, use the predicates to map steps back to the SQL Statement.
- Since was run on an Exadata machine, notice that many of the predicates are both **ACCESS** and **FILTER**. More on this later.

# DBMS\_XPLAN.DISPLAY\_CURSOR

- It is a "manual" thing
- To get accurate timing information either:
  - Use `/*+ gather_plan_statistics*/`
  - Set `statistics_level` to **ALL** (session or system)
- Generally best to let the SQL run to completion then get the plan
- Doesn't show parallel plans very well
- **DBMS\_XPLAN** can be used several ways, this is just one
- *Note:* SQL Developer has this functionality via Autotrace

# DBMS\_XPLAN: A simple example


```
SQL> SELECT PLAN_TABLE_OUTPUT
       2 FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR
       3 ('3431u08q6anmy',0,'ALLSTATS LAST'));
PLAN_TABLE_OUTPUT
```

```
-----
SQL_ID 3431u08q6anmy, child number 0
-----
```

```
SELECT ename FROM emp
```

```
Plan hash value: 3956160932
```

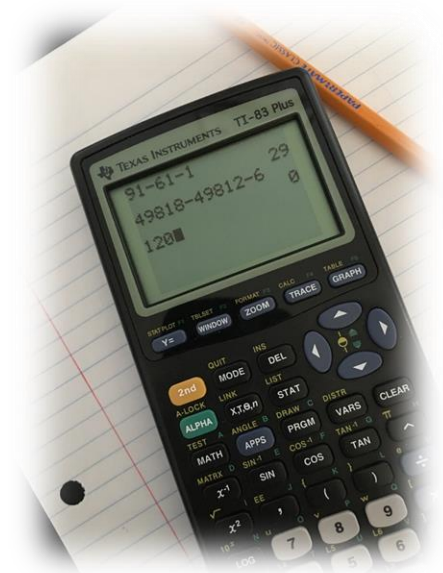
```
-----
| Id | Operation                | Name | Starts | E-Rows | A-Rows | A-Time   | Buffers |
-----|-----|-----|-----|-----|-----|-----|-----|
|  0 | SELECT STATEMENT         |      |       1 |        |      14 | 00:00:00.01 |      8 |
|  1 | TABLE ACCESS FULL      | EMP  |       1 |      14 |      14 | 00:00:00.01 |      8 |
-----
```





# DBMS\_XPLAN: You will need to do some math

- These columns *are not* cumulative:
  - **Starts**, **A-Rows**, **E-Rows**
- These columns *are* cumulative:
  - **A-Time**, **Buffers** (LIOs), **Reads** (PIOs)
- For cumulative columns the value includes the direct children



# Doing the math

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers
0	SELECT STATEMENT		1		16000	00:00:00.91	49818
* 1	<b>HASH JOIN</b>		1	15769	16000	00:00:00.91	49818
2	INDEX FAST FULL SCAN	THIS_IDX	1	500	500	00:00:00.01	6
3	TABLE ACCESS FULL	THAT_TAB	1	2898K	2898K	00:00:00.61	49812

- The A-Time for the **HASH JOIN** step is .29 ( $91 - 61 - 1 = 29$ )
- The Buffers (LIOs) for the **HASH JOIN** step is 0 ( $49,818 - 49,812 - 6 = 0$ )

# Now what?

- This execution plan has over 600 lines
- Look for lines with:
  - High activity
  - Long time consumption
- Attack those lines
  - Map back to the query
  - Use the predicates
- Is the step necessary?
- Is there another way to do that?

Operation	Name	Line ID	Estimated Rows	Actual Rows	Activity %	Trending(144k)	Executions
TABLE ACCESS STORAGE PULL	CLASST	582	14	224			18
PK BLOCK ITERATOR		583	1,522M	1,522M			18
TABLE ACCESS STORAGE PULL	Z_PAT_	584	1,522M	1,522M	76		610
FILTER		585		0			2,4340
SORT JOIN		586	701T	6,263			2,4340
VIEW		587	701T	17K			18
PK BLOCK ITERATOR	SYS_TE	588	701T	17K			18
TABLE ACCESS STORAGE PULL	SYS_TE	589	701T	17K			218
SORT JOIN		590	1,734T	17K			17K
PK RECEIVE		591	1,734T	294K			18
PK SEND BROADCAST	ITQ000	592	1,734T	294K			18
VIEW		593	1,734T	17K			18
WINDOW SORT PUSHED RANK		594	1,734T	17K			17K
HASH JOIN RIGHT OUTER		595	1,734T	32K   1.04			18
TABLE ACCESS STORAGE PULL	EMP_MG	596	663K	119K   1.1			18
HASH JOIN		597	1,139T	32K		18	18
PK RECEIVE		598	751M	750M   79			18
PK SEND HASH	ITQ000	599	751M	750M   2.25			18
HASH JOIN		600	751M	750M   1.36			18
TABLE ACCESS STORAGE PULL	CLASST	601	667K	119K   1.1			18
PK BLOCK ITERATOR		602	752M	750M			18
TABLE ACCESS STORAGE PULL	ROR_PC	603	752M	750M   1.21			259
PK RECEIVE		604	701T	17K			18
PK SEND HASH	ITQ000	605	701T	17K			18
VIEW		606	701T	17K			18
PK BLOCK ITERATOR		607	701T	17K			18
TABLE ACCESS STORAGE PULL	SYS_TE	608	701T	17K   1.12			17K
SORT JOIN		609	18,449M	11K			218
PK RECEIVE		610	18,449M	176K			18
PK SEND BROADCAST	ITQ000	611	18,449M	176K			18
VIEW		612	18,449M	11K			18
WINDOW SORT PUSHED RANK		613	18,449M	11K			18
HASH JOIN RIGHT OUTER		614	18,449M	15K   1.03			18
TABLE ACCESS STORAGE PULL	CLASST	615	667K	119K   2.3			18
HASH JOIN RIGHT OUTER		616	138M	15K   1.02			18
TABLE ACCESS STORAGE PULL	EMP_MG	617	663K	119K   54			18
HASH JOIN		618	91T	15K		11	18
SORT FILTER CREATE	BP0011	619	368M	450M   41			18
PK RECEIVE		620	368M	450M   27			18
PK SEND HASH	ITQ000	621	368M	450M   1.93			18
HASH JOIN BUFFERED		622	368M	450M   17.68			18
PK RECEIVE		623	255M	272M   1.12			18
PK SEND HYBRID HASH	ITQ000	624	255M	272M   56			18
STATISTICS COLLECTOR		625	255M	272M   11			18
HASH JOIN BUFFERED		626	255M	272M   177			18
PK RECEIVE		627	255M	281M   1.1			18
PK SEND HYBRID HASH	ITQ000	628	255M	281M   17			18
STATISTICS COLLECTOR		629	255M	281M   1.02			18
PK BLOCK ITERATOR		630	255M	281M			18
TABLE ACCESS STORAGE PULL	SP_FLW	631	255M	281M   1.11			219
PK RECEIVE		632	1,522M	620M   27			18
PK SEND HYBRID HASH	ITQ000	633	1,522M	620M   27			18
PK BLOCK ITERATOR		634	1,522M	1,522M			18
TABLE ACCESS STORAGE PULL	Z_PAT_	635	1,522M	1,522M	76		610
PK RECEIVE		636	365M	368M   20			18
PK SEND HYBRID HASH	ITQ000	637	365M	368M   56			18
PK BLOCK ITERATOR		638	365M	368M			18
TABLE ACCESS STORAGE PULL	SP_FLW	639	365M	368M   2.41			765
PK RECEIVE		640	701T	11K			18
PK SEND HASH	ITQ000	641	701T	11K			18
JOIN FILTER USE	BP0011	642	701T	11K			18
VIEW		643	701T	17K			18
PK BLOCK ITERATOR		644	701T	17K			18
TABLE ACCESS STORAGE PULL	SYS_TE	645	701T	17K   1.14			218



# Reading the Plan

- The order of execution and of completion are not the same
- The plan executes from top down, but completes "inside out"
- A parent step starts first but can't complete until it's children do
- This plan completes in this ID order: **2, 4, 5, 3, 1, 0**

OPERATION	OBJECT_NAME	OPTIONS	ID
SELECT STATEMENT			0
HASH JOIN			1
Access Predicates			
J.JOB_ID=E.JOB_ID			
TABLE ACCESS	JOBS	FULL	2
HASH JOIN			3
Access Predicates			
E.DEPARTMENT_ID=D.DEPARTMENT_ID			
TABLE ACCESS	DEPARTMENTS	FULL	4
TABLE ACCESS	EMPLOYEES	FULL	5
Other XML			



# What to look for in a plan

- **Look for big time consumers**
  - This step or set of steps will be your main focus
  - It might not be "this" step that is the source of the problem
- **Watch the cardinality (estimated vs actual rows)**
  - Fix where it goes wrong
  - Over or under estimates can cause a plan to go rouge
  - Generally speaking, everything after that step is questionable
- **Observe the execution counts**
  - A set of lines that execute over and over is likely a problem
  - *Note:* the inner part of a **NESTED LOOPS** and a **SORT MERGE** join always executes multiple times.





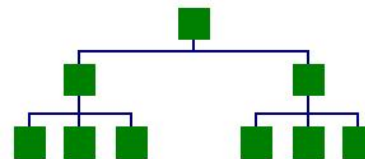
# Index scans

- This is an alternative to a full table scan
- An index returns the ROWID for a given row
- With the ROWID the optimizer can go directly to the block containing the row
- This is quite fast when retrieving a relatively small amount of rows from a table
  - There really isn't a fixed amount or percent when this is "always" best
- An index can be scanned without going to the table
  - Called a "fat" or "covering" indexes



# Indexes

- For the normal B-Tree index best to have a multi-column index
  - When using a B-Tree index you will only get to use one per table
  - Find an index with a set of commonly used columns
  - Ideally create one multi-column index to support many queries
- For Bit Map indexes typically better to have several single column indexes
  - The optimizer can put these together in using either an **AND** or **OR** operation
  - This gives flexibility in data warehouses
  - Bit map indexes and DML don't get along



# Hints: Good vs Bad

- A hint can be considered good when it gives the optimizer information that helps it make good decisions, but doesn't force an access path. For example:
  - **ALL\_ROWS**, **FIRST\_ROWS (n)**
  - **CARDINALITY**
  - **(NO) REWRITE**
  - **DRIVING\_SITE**
  - **DYNAMIC\_SAMPLING**
- A hint can be considered bad when it constrains the optimizer from choosing a possible better access path in the future. For example:
  - **INDEX**
  - **FULL**
  - **USE\_NL**, **USE\_HASH**, **USE\_MERGE**



# Hints

- Generally hints should be avoided
- They aren't hints really, they are directives
- Using them without really knowing what you're doing can be bad
- Hints are an excellent tool for testing
- At times they are needed in production
- The goal is to have production code with no hints
- The following slides are a few hints I have found useful



# MONITOR

```
/*+ MONITOR */
```

- Forces the SQL to show up in SQL Monitor
- Normally only shows up if:
  - Running in Parallel
  - Uses more than 5 seconds of CPU or IO
- Handy for testing, likely not necessary in production

```
SELECT /*+ MONITOR */ ...  
FROM ...  
WHERE ...  
/
```

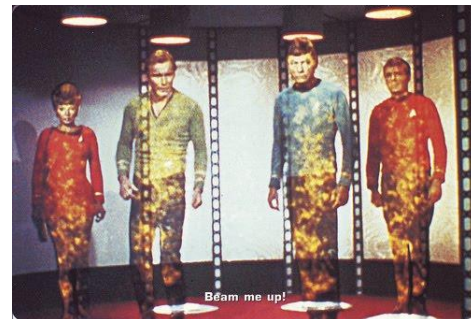


# MATERIALIZER

## /\*+ MATERIALIZER \*/

- Use this for CTEs (Common Table Expressions)
- The **WITH** Clause
- More on this later

```
WITH my_cte AS(  
  SELECT /*+ materialize */  
    region,  
    SUM(order_amt) AS total_orders  
  FROM sales  
  GROUP BY region)  
SELECT *  
  FROM my_cte  
/
```



# NO\_MERGE

`/*+ NO_MERGE */`

- Similar to the materialize hint
- Used with In Line Views (ILVs)
- By default, the optimizer will try to merge ILVs into the query



```
select student_id, last_name
  from (select /*+ NO_MERGE */ * from student order by student_id) stdnt
 where (select count(*) from enrollment enrllmnt
        where stdnt.student_id = enrllmnt.student_id) >
        (select avg(count(*)) from enrollment group by student_id);
```

# NO\_EXPAND

`/*+ NO_EXPAND */`

- Stops the expansion of an "OR" set of predicates ("IN" clauses too)
- If you see a long set of "identical" selects in the execution plan with OR operations, good chance this happened
- Rarely is expanding a good idea

```
SELECT /*+ NO_EXPAND */ ...  
FROM ...  
WHERE ...  
MAILING_STATE in ('MI', 'CT', 'TX', 'AK', 'TN'...)  
...  
/
```

# CARDINALITY

```
/*+ CARDINALITY (<alias,> X ) */
```

- This forces the optimizer to use the cardinality for a given object
- Useful when the optimizer is unable to correctly calculate the cardinality
- Excellent for testing
- Can be useful with CTEs



```
WITH Complex_CTE AS  
(  
  SELECT /*+ materialize cardinality(5000) */  
  ...  
)  
SELECT ...  
  FROM Complex_CTE, ...  
/
```

# OPT\_PARAM

```
/*+ OPT_PARAM('OPTIMIZER_INDEX_COST_ADJ',10000) */
```

- Likely only useful in Exadata (where Full Scans tend to be best)
- Forces the indexes to be costed extremely high
- Best for testing, *not intended* for production
- This is placed just after the main select of a statement
- Many other parameters can be used with this hint

```
SELECT /*+ OPT_PARAM('OPTIMIZER_INDEX_COST_ADJ',10000) */  
...  
/
```



# Some general tips

- **Avoid REGEXP\_LIKE when possible**
  - The **LIKE** function uses significantly less CPU and time
- **Use OUTER joins only when absolutely necessary**
  - These tend to explode the row counts
- **NLS and case insensitive searching**
  - **NLS\_SORT** and **NLS\_COMP**
  - Don't use these in Exadata, predicates don't get offloaded



# A pet peeve of mine

## *DO NOT USE* single character aliases

- When defining an Alias for a table or column don't use single characters
  - `FROM emp_master a` This makes code difficult to read
  - `FROM emp_master em` This is a tiny bit better
  - `FROM emp_master emp_mstr` This is better, remove vowels to shorten words
  - `FROM emp_master emp_master` This is best, use the table/column name
- When using a table multiple times use a different Alias each time
  - `FROM emp_master emp_mstr01`
  - `FROM emp_master emp_mstr02`

# Common Table Expressions (CTEs)



- *Time to get **WITH** it!*
- These alone can solve an incredible amount to performance issues
- The idea is "Run once, use many"
- Oracle calls these Subquery Factors
- The industry standard name is Common Table Expressions (CTEs)

# CTEs: some basic facts

- A CTE is defined using the **WITH** clause
- The **WITH** clause offers the benefit of reusing a query when it occurs more than once within the same SQL.
- The **WITH** clause allows you to give a repeated query block an alias and then reference it by that alias names multiple times.
- This avoids a re-read and re-execution of the query and can result in improvement in overall execution time and resource usage.
- Typically used when querying large volumes of data.
- Can be materialized.
- These have been around for awhile, first available in v9.2.

# Syntax of the WITH Clause

- There may be a limit on how many CTEs you can define in a statement
  - If you hit that limit you likely already have a problem
- A later CTE can refer to an earlier one
  - Not the other way around
- The syntax is like a view definition
  - The CTE disappears once the query is finished
- The **WITH** clause cannot be nested

```
with CTE_1 as (  
    select statement 1  
) ,  
CTE_2 as (  
    select statement 2  
    {possibly referencing to CTE_1}  
)  
select ...  
from  
    CTE_1 CTE1 ,  
    CTE_2 CTE2 ,  
    MY_TABLE MY_TAB  
where ...
```



# Why use CTEs: Correlated Subqueries

- Correlated subqueries are likely the number one performance killer in SQL
- Especially in the SELECT list
- One of the best uses for CTEs is to get rid of correlated subqueries
- Note in the example the *executions* column
- Query went from over 10 minutes to 33 seconds using CTEs

Operation	Obj	Info	Line ID	Est. Rows	Timeline	Execution:	Rows
▲ SELECT STATEMENT			0			1	11K
▲ UNION-ALL			1			1	11K
▲ COUNT STOPKEY			2			9,794	531
▶ NESTED LOOPS			3	1		9,794	531
▲ SORT AGGREGATE			13	1		9,794	9,794
▶ NESTED LOOPS			14	1		9,794	26K
▲ SORT AGGREGATE			31	1		9,794	9,794
▶ NESTED LOOPS			32	1		9,794	34K
▲ SORT AGGREGATE			49	1		9,794	9,794
▶ NESTED LOOPS			50	1		9,794	263
▲ SORT AGGREGATE			67	1		9,794	9,794
▶ NESTED LOOPS			68	1		9,794	1,193
▲ SORT AGGREGATE			85	1		9,794	9,794
▶ NESTED LOOPS			86	1		9,794	62K
▲ SORT AGGREGATE			102	1		9,794	9,794
▲ NESTED LOOPS			103	1		9,794	62K

# Why use CTEs: Correlated Subqueries an Example

```
SELECT
  dname,
  loc,
  (SELECT COUNT(empno) FROM emp
   WHERE emp.deptno = dept.deptno) emp_cnt
FROM
  dept
ORDER BY
  dname;

WITH empcnt_cte AS (
  SELECT /*+ materialize */
    deptno, COUNT(empno) emp_cnt FROM emp
    GROUP BY deptno )
SELECT
  dname, loc, nvl(emp_cnt,0) emp_cnt
FROM
  dept, empcnt_cte
WHERE dept.deptno = empcnt_cte.deptno(+)
ORDER BY dname;
```



As a SUBQUERY



As a CTE

# Why use CTEs: Correlated Subqueries an Example

The result set for both statements



Query Result x

SQL | All Rows Fetched: 4 in 0.053 seconds

	DNAME	LOC	EMP_CNT
1	ACCOUNTING	NEW YORK	5
2	OPERATIONS	BOSTON	0
3	RESEARCH	DALLAS	5
4	SALES	CHICAGO	4

# Why use a CTE: Correlated Subqueries Example

OPERATION	OBJECT_NAME	OPTIONS	LAST_STARTS
SELECT STATEMENT			1
SORT		AGGREGATE	4
INDEX	EMP_DEPT_IDX	RANGE SCAN	4
Access Predicates			
EMP.DEPTNO=:B1			
SORT		ORDER BY	1
TABLE ACCESS	DEPT	STORAGE FULL	1

← As a SUBQUERY

OPERATION	OBJECT_NAME	OPTIONS	LAST_STARTS
SELECT STATEMENT			1
TEMP TABLE TRANSFORMATION			1
LOAD AS SELECT	SYS_TEMP_1FD9DDA06_2AD5F375	(CURSOR DURATION MEMORY)	1
HASH		GROUP BY	1
TABLE ACCESS	EMP	STORAGE FULL	1
SORT		ORDER BY	1
HASH JOIN		OUTER	1
Access Predicates			
DEPT.DEPTNO=EMPCNT_CTE.DEPTNO			
TABLE ACCESS	DEPT	STORAGE FULL	1
VIEW			1
TABLE ACCESS	SYS.SYS_TEMP_1FD9DDA06_2AD5F375	STORAGE FULL	1

← As a CTE

Output above is SQL Developer Autotrace.

Using these tiny tables the time was really the same, @.05 seconds.

The savings will happen as the **DEPT** table (in this case) grows.



# Why use CTEs: Repeated Subqueries

- It is not uncommon that in large SQL statements (hundreds of lines) to have the same subquery repeated
- Often happens when there are multiple **UNION** statements
- It might be identical each time or slight variations
- Replace these with one CTE to get the "super set"
- Then from the CTE, select what you need each time
- Now you will hit the database once for the data and use it over and over



# Why use CTEs: Transforming data

- Another good use is to transform the data thru a set of CTEs
- This allows you to "massage" the data to get it into a form desired
- Likely easier to see and understand when done as a set of steps thru CTEs
- This makes debugging and maintenance easier
- This can run faster then if it was done "all at once" in a single query



# Why use CTEs: Recursive CTEs



- With recursive CTEs you have an alternative for hierarchical queries
- This is an alternative to the classic "**START WITH** .. **CONNECT BY** .."
- There is an **ANCHOR** member and a **RECURSIVE** member in the statement
- These two members are connected with a **UNION ALL**
- You can easily follow by **DEPTH** or **WIDTH** first

# Why use a CTE: Recursive CTEs an Example

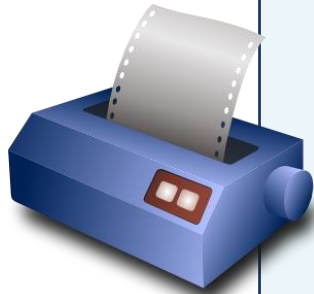
```
WITH
  org_chart (eid, emp_last, mgr_id, reportLevel) AS
  (
    SELECT empno, ename, mgr, 1 reportLevel
    FROM emp
    WHERE job='PRESIDENT'
  UNION ALL
    SELECT e.empno, e.ename, e.mgr,
           reportLevel+1
    FROM org_chart r, emp e
    WHERE r.eid = e.mgr
  )
  SEARCH DEPTH FIRST BY eid SET order1
SELECT lpad(' ', 2*reportLevel)||eid emp_no, emp_last
FROM org_chart
ORDER BY order1
/
```

ANCHOR member

RECURSIVE member

SEARCH BREADTH FIRST BY eid SET order1

# Why use CTEs: Recursive CTEs an Example



```
*****
***** DEPTH FIRST
*****
```

EMP_NO	EMP_LAST
7839	KING
7566	JONES
7788	SCOTT
7876	ADAMS
7902	FORD
7369	SMITH
7698	BLAKE
7499	ALLEN
7521	WARD
7654	MARTIN
7844	TURNER
7900	JAMES
7782	CLARK
7934	MILLER

```
*****
***** BREADTH FIRST
*****
```

EMP_NO	EMP_LAST
7839	KING
7566	JONES
7698	BLAKE
7782	CLARK
7499	ALLEN
7521	WARD
7654	MARTIN
7788	SCOTT
7844	TURNER
7900	JAMES
7902	FORD
7934	MILLER
7369	SMITH
7876	ADAMS

**BREADTH**  
is the  
default

# Why use CTEs: WITH Functions

- Allows you to define a FUNCTION in a CTE structure
  - You can define PROCEDURES too, but that seems highly unusual to do
- The Optimizer will In Line the function if it can automatically
  - This can run much faster then calling to a standalone function
- This is best for something that is a "one-off"
  - There already exists a function that does almost what you need
  - You can replicate the function in the CTE with the change(s) you need
  - So long as this is the only place used it can be a good idea
- Even if you never use this functionality, it's super cool that you could 😊



# Why use CTEs: WITH Functions an Example



```
SQL> get_with_fun
1  with function rtn_date_convert2 (p_unix_gmt in number)
2    return date
3    as
4    v_date  date;
5    begin
6    v_date := to_date('01/01/1970','mm/dd/yyyy') + (p_unix_gmt/86400);
7    RETURN v_date;
8    end ;
9  select count(*) from ord2
10* where rtn_date_convert2(gmt_order_date) > sysdate - 7000
SQL>
```

- Done as a **WITH** function this ran about half the time as a standalone function
- This function takes a **NUMBER** and converts it to a **DATE**
- The number is assumed to be the number for seconds since the Unix epoch
  - 86,400 is the number of seconds in one day (24 hours)

# CTEs: Some Recommendations

Keep the number of them down  
Less than 10 would be ideal  
Keep the row count down per CTE  
A million or less rows returned is great

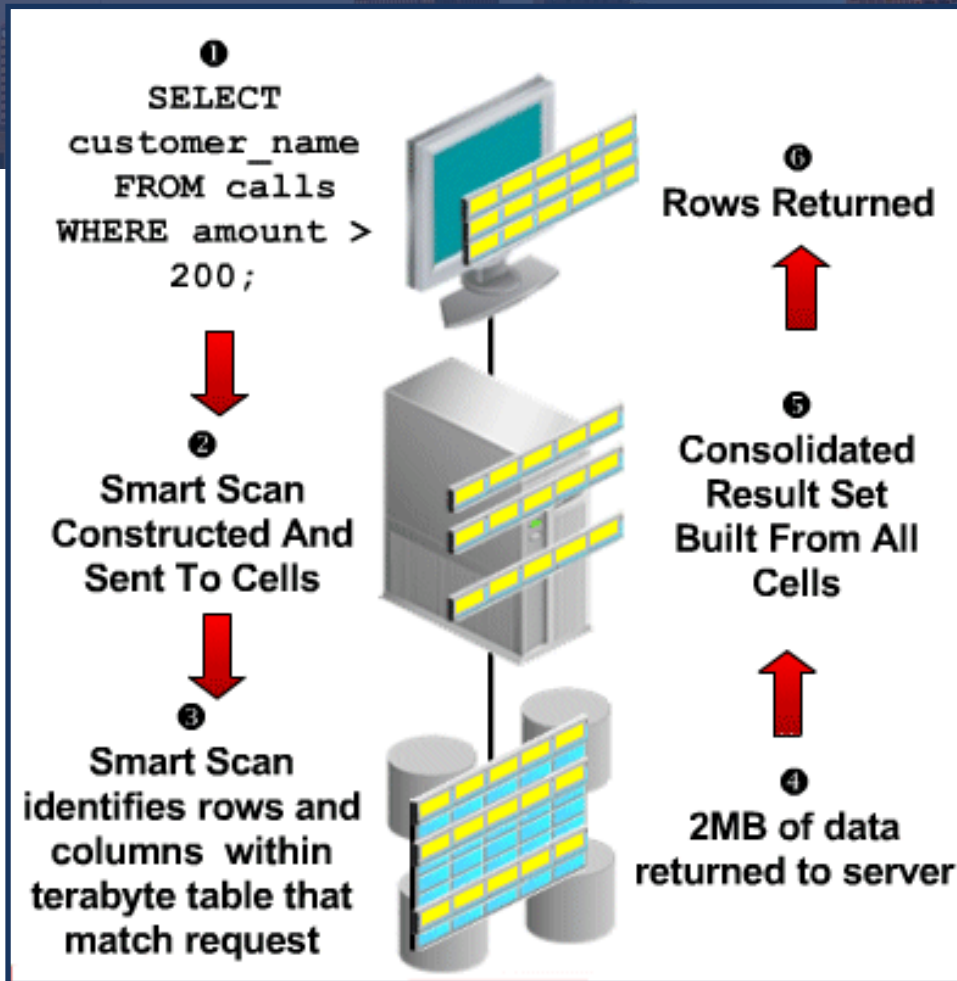
- These aren't about "limits", they are about conserving memory
  - Materialized CTEs take up memory (PGA)
- If a set of 16 CTEs or a 500 million row CTE has significant performance improvement, doing so may well be worth it

# A little about Exadata

- It's all about the Smart scan
- Smart scan is when the storage cell performs a scan
- Data blocks are scanned at the storage level
- Only qualifying blocks are returned to the database server
- Some predicates may not be applied at the storage level
- Only filter predicates can be applied
- Some join filter can be done as well with the use of bloom filters
- Storage indexes are used (if available, created automatically)
- Also referred to as "Offloading"
- Generally only happens for full scan actions (full table or index fast full)



# Smart Scan





A simple example  
of a smart scan





This diagram is from  
*Exadata System Software  
User's Guide* chapter 1  
*Introducing Oracle Exadata  
System Software.*

# Smart Scans

- Smart scans are fast – when they happen
- This operation scanned 377,877,940 rows in 8 seconds.
  - About 47 million rows a second

Operation	Name	Line ID	Estimated R...	Actual Rows	Timeline(...)	Activ...
 PX BLOCK ITERATOR		44	462M	378M		
 TABLE ACCESS STORAGE FULL	2_AP_C	45	462M	378M		.36

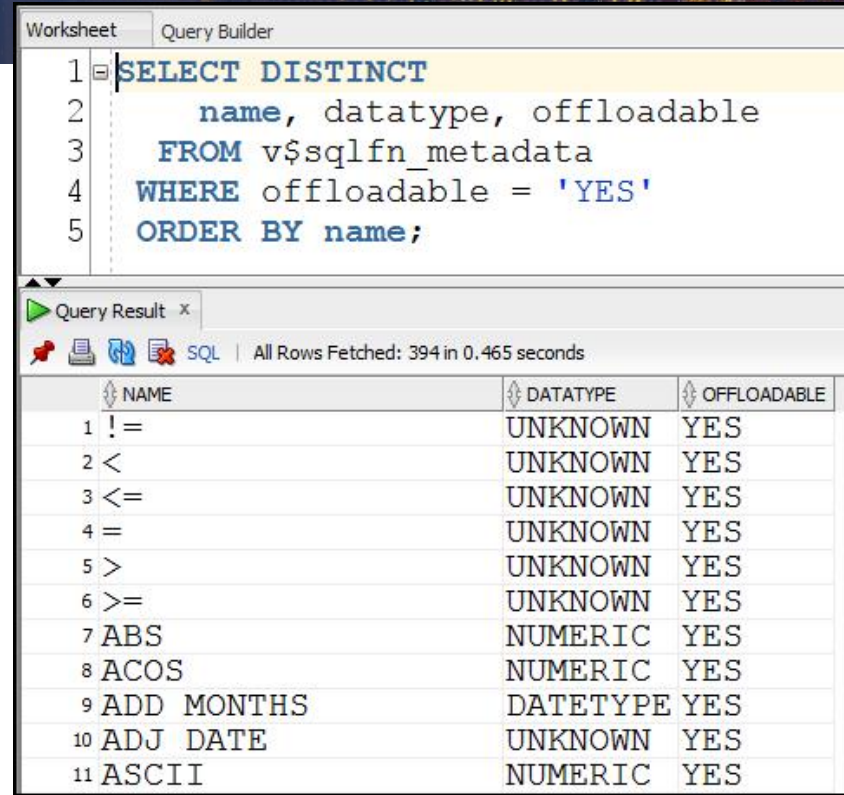
- But when it doesn't, *rats!*
- This operation scanned 4,513,812 rows in 142 seconds.
- About 31 thousand rows a second

Operation	Na...	Line ID	Estimate...	Actu...	Timeline(340s)	Activity %
 PX BLOCK ITERATOR		29	431M	4,514K		
 TABLE ACCESS STORAGE FULL	1_PAT	30	431M	4,514K		71



# Smart Scans: Can we control when it happens?

- Not really but we can influence it's possibility
  - Doing a smart scan is a run time decision
  - If the nodes are too busy they wouldn't even if they could
- Your predicates can influence if it can even be done
  - Generally the simpler the predicate, the more likely it will happen
  - In 21c, 394 distinct operations are offloadable



The screenshot shows a SQL Query Builder interface. The query text is as follows:

```
1 SELECT DISTINCT
2     name, datatype, offloadable
3 FROM v$sqlfn_metadata
4 WHERE offloadable = 'YES'
5 ORDER BY name;
```

Below the query, the 'Query Result' window shows the following data:

	NAME	DATATYPE	OFFLOADABLE
1	!=	UNKNOWN	YES
2	<	UNKNOWN	YES
3	<=	UNKNOWN	YES
4	=	UNKNOWN	YES
5	>	UNKNOWN	YES
6	>=	UNKNOWN	YES
7	ABS	NUMERIC	YES
8	ACOS	NUMERIC	YES
9	ADD MONTHS	DATATYPE	YES
10	ADJ DATE	UNKNOWN	YES
11	ASCII	NUMERIC	YES

# Smart Scans

- Predicates and cell offloading
- It's the FILTER version of the predicate that can be offloaded

Plan Hash Value: 1272887555

Operation	Object	Line ID	Predicate	Pruning
PX SEND HYBRID HASH	:TQ10000	36		
STATISTICS COLLECTOR		37		
PX BLOCK ITERATOR		38		1 .. 3
TABLE ACCESS STORAGE FULL	2_PAT_EM	39		1 .. 3

**TABLE ACCESS STORAGE FULL**

Sub-tree Cost 4,506 (.36% CPU)  
Operation Cost 4,506 (.36% CPU)  
Sub-tree Time <1s  
Rows 840K  
Bytes 48M  
Object 2\_PAT\_ENC\_HSP

Access Predicates :Z>=:Z AND :Z<=:Z AND ("ED\_EPISODE\_ID" IS NOT NULL AND "ADT\_PATIENT\_STAT\_C"=3 AND (INTERNAL\_FUNCTION("ADMIT...))





Filter Predicates ("ED\_EPISODE\_ID" IS NOT NULL AND "ADT\_PATIENT\_STAT\_C"=3 AND (INTERNAL\_FUNCTION("ADMIT...))

Partition Start 1  
Partition Stop 3
















Query Block Name/Object Alias SEL\$8727A776/2\_PAT\_ENC\_HSP@SEL\$17

# Indexes with Exadata

- What about indexes?
- Using indexes to get a small amount of data is still a good idea
- These scans took less then 3 seconds to complete

Operation							Name	Line ID	Estimat...	Actual Rows	Activity %	Timeline(233s)
						TABLE ACCESS BY INDEX ROWID	CLARITY	208	2	115		
						INDEX UNIQUE SCAN	PK_CLAF	209	1	115		
						TABLE ACCESS BY INDEX ROWID	CLARITY	210	1	112		
						INDEX UNIQUE SCAN	PK_CLAF	211	1	112		

# Indexes with Exadata

Operation	Name	Line ID	Estimat...	Actual Rows	Activity %	Timeline(813s)	Executions
 →	⊖ PX RECEIVE	24	61K	488K			16
 →	⊖ PX SEND HYBRID HASH	:TQ1000	61K	502K	.04		16
 →	⊖ NESTED LOOPS OUTER		61K	502K	.01		16
 →	⊖ NESTED LOOPS OUTER		60K	502K	.01		16
 →	⊖ NESTED LOOPS OUTER		60K	471K	.02		16
 →	⊕ HASH JOIN		60K	471K	.21		16
 →	⊖ TABLE ACCESS BY INDEX ROWI...	ED_IEV_	1	15K	 87		471K
 →	└ INDEX RANGE SCAN	EIX_EDI	5	4,773K	 10		471K
 →	⊖ TABLE ACCESS BY INDEX ROWID ...	ED_IEV_	1	114K	 .92		471K
 →	└ INDEX RANGE SCAN	EIX_EDI	5	4,778K	.09		471K
 →	⊖ TABLE ACCESS BY INDEX ROWID B...	ED_IEV_	1	123K	.13		502K
 →	└ INDEX RANGE SCAN	EIX_EDI	5	5,671K	.03		502K

These green arrows means *it was running* when the screen shot was taken!

- This is the kind of thing that can go very wrong with indexes
- Indexes can drive stacks of **NESTED LOOPS** joins
- These can be very inefficient
- *No benefit* from cell offloading

# Indexes and Exadata

- Helping the Optimizer not use an index
- Two techniques I like:
  - **COALESCE** (**NVL** doesn't always work)
  - In Line Views (ILV), need the **NO\_MERGE** hint
- Of course you could use the **FULL** hint, make the index **INVISIBLE** or **DROP** the index as well

```
COALESCE(my_tab.id,-99999) = other_tab.id
```

```
LEFT JOIN (select /*+ no_merge */  
id, name, status from my_tab) my_tab  
ON my_tab.id = other_tab.id
```



# Zione Solutions

Leading System Integrator & IT Service Provider

Strong Big Data & Business Intelligence Expertise

Highly Skilled Pool of Resources

Oracle Service Partner

Cloud & Managed Services

Experience Delivering “Industry Best Practices”

Agile Scrum-Based Application Life Cycle Management

Zione Solutions, LLC.  
37000 Grand River Avenue,  
STE 355  
Farmington Hills, MI 48335



Phone: (248)-442-7404

Email : [contact@zionesolutions.com](mailto:contact@zionesolutions.com)

Site : [www.zionesolutions.com](http://www.zionesolutions.com)



# Questions and Answers



**Ric Van Dyke**

Sr. DBA

Oracle Ace

[rvandyke@zionesolutions.com](mailto:rvandyke@zionesolutions.com)

