10g Advanced SQL and Performance in 2005

OracleWorld
September, 2005

Tim Quinlan
TLQ Consulting Inc.

Introduction

- We will cover advanced SQL concepts, real-world problems and solutions and the performance impact of advanced SQL coding decisions. Ansi-compliant & Oracle 10g functions and features as well as some creative solutions to SQL problems will be discussed. This will be a fast-paced look at a fun-topic. Some of the things we will look at are:
  - 9i/10g Full outer joins and join indexes
  - Case and Decode
  - 9i and 10g Analytics (e.g. Rankings)
  - Medians
  - First and Last Functions
  - Model and (of course) more…

Finding the First/Last N Rows
Limiting Rows with Rownum

- Want to find the first or last n rows in a table? Use Rownum!
- For each row returned by a query, the ROWNUM pseudo-column returns a number indicating the order in which Oracle selects the row from a table or set of joined rows. This list however is not ordered. An example of this is:
  - select username,rownum from dba_users order by username;

USERNAME	ROWNUM
OUTLN	3
SYS	1
SYSTEM	2

- ORDER BY will usually not solve the problem since rownum is applied to the row before they are sorted.

Finding the First/Last N Rows
Limiting Rows with Rownum

- Ordering can be corrected by retrieving the rows and sorting them in a subquery and then applying the rownum in the outer query.
- greater-than sign used with rownum and a positive integer will never return a row.
- To get the first 3 rows:
  - SELECT username,rownum FROM
    (SELECT username FROM dba_users
    ORDER BY username)
    WHERE ROWNUM < 4;

Finding the First/Last N Rows
Limiting Rows with Rownum

- Greater-than sign used with rownum and a positive integer will not return a row
- To show the last 3 rows we therefore cannot use >, but must instead use < and must order the rows descending:
  - SELECT username,rownum FROM (SELECT username FROM dba_users ORDER BY username desc) WHERE ROWNUM < 4

USERNAME	ROWNUM
TESTUSER	1
SYSTEM	2
SYS	3

Analytic Functions

- Specialized functions that return aggregate values based on a grouping of rows.
- Multiple rows can be returned for each group.
- Each group can be called a “window” although, unfortunately, the term “partition” is used in SQL
- Calculations can be performed on rows in the window.
- Can only appear in “Select” or “Order By” clause.
  - Last operators performed in a query except “order by”
- Let’s look at some …

Finding the First/Last N Rows
Limiting Rows with Row_Number

- Row_Number function not related to Rownum.
- an analytic function that assigns a unique number in the sequence field defined by ORDER BY to each row in a partition. e.g.

SELECT sales_rep, territory, total_sales, row_number() OVER (PARTITION BY territory ORDER BY total_sales DESC) as row_number FROM sales;

Sales_Rep	Territory	Total_Sales	Row_Number
Simpson	1	990	1
Lee	1	800	2
Blake	5	2850	1
Allen	5	1600	2

See Explain on the next slide →

Analytic Example with Row_Number

SELECT sales_rep, terr, total_sales, row_number()
OVER (PARTITION BY terr ORDER BY total_sales DESC) as row_number FROM sales

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.02	0.09	0	1	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	0.00	0.00	0	7	0	4
total	4	0.02	0.09	0	8	0	4

Rows	Execution Plan
0	SELECT STATEMENT MODE: ALL_ROWS
4	WINDOW (SORT)
4	TABLE ACCESS (FULL) OF 'SALES' (TABLE)

next->rank

Ranking: The Rank Function

- An analytic function which allows us to compare a row to a window of rows. Consider a Sales table:

sales_rep	territory	total_sales
Jones	1	345
Smith	1	345
Lee	1	200
Simpson	1	990

- ```
  SELECT sales_rep, territory, total_sales,
  RANK() OVER (PARTITION BY territory ORDER
  BY total_sales DESC) as rank FROM sales;
  ```

sales_rep	territory	total_sales	rank
Simpson	1	990	1
Jones	1	345	2
Smith	1	345	2
Lee	1	200	4
  ```

Ranking and Aggregates

SELECT terr, prod, sum(total_sales), RANK() OVER (PARTITION BY territory ORDER BY sum(total_sales) DESC) as rank FROM sales group by territory, product;

sales_rep	terr	prod	total_sales
Jones	1	8	200
Smith	1	8	400
Lee	1	9	800
Simpson	1	9	990
Blake	5	9	1600
Allen	5	8	1500
Ward	5	9	1250

1) Input data to

2) Results in

territory	product	sum(total_sales)	rank
1	9	1790	1
1	8	600 (typo)	2
5	9	2850	1
5	8	1500	2

Rankings With Different Boundaries

- Rank can be used for different groups. Here are 2 ranks: One for a products total sales in a territory and the second for a product (in a territory) across all territories.

```
SELECT terr, prod, sum(ttl_sales),
RANK() OVER (PARTITION BY terr ORDER BY sum(ttl_sales)
   DESC) as rank_prod_by_terr,
RANK() OVER (ORDER BY sum(ttl_sales) DESC) as rank_prod_ttl
FROM sales GROUP BY territory, product ORDER by territory;
```

territory	product	sum(total_sales)	rank_prod_per_terr	rank_prod_ttl
1	9	1790	1	2
1	8	400	2	4
5	9	2850	1	1
5	7	1500	2	3

Explain for Rankings with Different Boundaries

```
SELECT terr, prod, sum(total_sales),
RANK() OVER (PARTITION BY terr ORDER BY sum(total_sales) DESC) as
    rank_prod_by_terr,
RANK() OVER (ORDER BY sum(total_sales) DESC) as rank_prod_ttl
FROM sales GROUP BY terr, prod ORDER by terr;
```

Rows	Execution Plan
0	SELECT STATEMENT MODE: ALL_ROWS
4	WINDOW (SORT)
4	WINDOW (SORT)
4	SORT (GROUP BY)
7	TABLE ACCESS (FULL) OF 'SALES' (TABLE)

Rankings

- Rankings are extremely flexible and provide the following:
  - Ranking per-cube and rollup-group
  - Dense Rank vs. Rank
    - Handles ties by going to the next value
  - Cume-Dist Ranking (Inverse Percentiles)
    - Dist computes a fraction of a value relative to its position in its partition. It returns the result as a decimal between 0 (not including 0) and 1.
    -

Terr	Prod	Amt	Cume_Dist
1	5	800	.666667
1	7	300	.333333
1	8	1300	1

Rankings

- **Percent Rank Function:** like cume_dist but uses row counts as a numerator and returns values between 0 and 1.
- **Ntile Function:** perform calculations and statistics for tertiles, quartiles, deciles and other summary stats:

```
SELECT sales_rep, total_sales, NTILE(4)
OVER (ORDER BY total_sales DESC NULLS FIRST)
AS quartile FROM sales;
```

Sales_Rep	Total_Sales	Quartile
Jones	2000	1
Smith	1000	2
Blake	700	3
Ward	400	4

Hypothetical Rank

- Analytic functions can be used to determine the rank of a "hypothetical" row inserted into a table.
- For example, consider a set of product categories that have total_sales determines by sub-category.
  - What would the rank of a new product subcategory with sales of $2,000 be?
  - What would the rank of a new product subcategory with sales of $1,480 be?
  - The example on the next slide shows how rank, percent_rank and cume_dist can all be used in a single hypothetical rank query.

Hypothetical Rank

Original List

PROD	TOTAL_SALES
7	700
7	750
7	1600
7	2100
8	150
8	200
8	300
8	400
8	1200
9	1400
9	1475
9	1500
9	1600
9	2400

```
select prod,
RANK(2000) within group (ORDER BY
    total_sales desc) as HRANK,
TO_CHAR(PERCENT_RANK(2000) WITHIN
    Group (ORDER BY total_sales),'9.999')
    as HPERC,
  TO_CHAR(CUME_DIST(2000) WITHIN
    Group (ORDER BY total_sales),'9.999')
    as HCUME
FROM sales Group by prod;
```

PROD	HRANK	HPERC	HCUME
7	2	.750	.800
8	1	1.000	1.000
9	2	.800	.833

Lead Analytic Function

- Get the next value in a list without a self-join or sub-query
- E.g. a table employee with columns ename and hiredate. Develop a query where each row has the employees name, their hiredate and the next employees hire date.

```
SELECT ename, hiredate, LEAD(hiredate, 1)
OVER (ORDER BY hiredate) AS next_hire_date
FROM employee;
```

ENAME	HIREDATE	NEXT_HIRE_DATE
COHEN	1991-APR-01	1991-OCT-31
KING	1991-OCT-31	1992-JAN-10
LEE	1992-JAN-10	

- Offset of 1 (default) tells the function to get the next row.

Explain for Lead Analytic Function

```
SELECT ename, hiredate, LEAD(hiredate, 1)
OVER (ORDER BY hiredate) AS next_hire_date
FROM employee;
```

```
Rows    Execution Plan
   0    SELECT STATEMENT   MODE: ALL_ROWS
   3      WINDOW (SORT)
   3        TABLE ACCESS (FULL) OF 'EMPLOYEE' (TABLE)
```

Lag Analytic Function

- To get the date of the employee hired before the employee on a row, use the LAG analytic function:

```
SELECT ename, hiredate,  LAG(hiredate, 1)
OVER (ORDER BY hiredate) AS  prev_hire_date
FROM employee;
```

ENAME	HIREDATE	PREV_HIRE_DATE
COHEN	1991-APR-01	
KING	1991-OCT-31	1991-APR-01
LEE	1992-JAN-10	1991-OCT-31

- Great for determining effective and expiry dates on a row where only 1 date exists.

First/Last Functions

- Analytic, aggregate functions that operate on a set of values from a set of rows
- When you need the lowest or highest value from a sorted set to compare to another value from a function such as min, max, sum,avg, count. Use the Last and First analytic functions
- For example: to find the max salary of employees with the highest bonus and the lowest salary of employees with the lowest bonus
  - ⋆ See next slide ->

First/Last Functions

Input data

salary	deptid	bonus
1,000,000	100	20,000
100,000	100	20,000
60,000	100	15,000
40,000	100	15,000

```
SELECT deptid,
min(salary) keep (dense_rank FIRST order by bonus) "low",
max(salary) keep (dense_rank LAST order by bonus) "high",
FROM emp_salary group by deptid;
```

deptid	low	high
100	40000	1000000

First/Last Functions

- To achieve the same thing without these functions:

```
SELECT a.deptid deptid, min(a.salary) low, max(c.salary) high
FROM emp_salary a,
  (select min(bonus) bonus from emp_salary) b,
  emp_salary c,
  (select max(bonus) bonus from emp_salary) d
WHERE a.bonus = b.bonus
   and c.bonus=d.bonus
GROUP BY a.deptid;
```

- Compared to:

```
SELECT deptid,
min(salary) keep (dense_rank FIRST order by bonus) "low",
max(salary) keep (dense_rank LAST order by bonus) "high",
FROM emp_salary group by deptid;
```

Explain for First/Last Functions

```
SELECT deptid,
min(salary) keep (dense_rank FIRST order by bonus) "low",
max(salary) keep (dense_rank LAST order by bonus) "high"
FROM emp_salary group by deptid;
```

```
Rows     Row Source Operation
     0  SELECT STATEMENT   MODE: ALL_ROWS
     1   SORT (GROUP BY)
     4    TABLE ACCESS (FULL) OF 'EMP_SALARY' (TABLE)
```

Explain for SQL Not Using First/Last Functions

```
SELECT STATEMENT
  SORT GROUP BY
    MERGE JOIN
      SORT JOIN
        NESTED LOOPS
          MERGE JOIN
            VIEW
              SORT AGGREGATE
                TABLE ACCESS FULL EMP_SALARY
            FILTER
              TABLE ACCESS FULL EMP_SALARY
          VIEW
            SORT AGGREGATE
              TABLE ACCESS FULL EMP_SALARY
      SORT JOIN
        TABLE ACCESS FULL EMP_SALARY
```

First_Value, Last_Value Analytic

- Similar functions, so we will look at first_value.
- An analytic function that gets the first value in an ordered set of rows.
  - e.g. get the name of the employee with the lowest bonus.

Select emp_name, emp_id, bonus, first_value(emp_name)

Over (order by bonus asc rows unbounded preceding) as low_bonus

From (select * from emp_sal order by emp_id);

EMP_NAME	EMP_ID	BONUS	LOW_BONUS
carter	2	20,000	carter
rose	3	20,000	carter
williams	7	25,000	carter
bosh	8	30,000	carter

next->Width-Bucket

Histogram Function
Width-Bucket

- Not Optimizer Histograms: Height-based place the same number of values into each range
- Width-based function: each column value is put into a corresponding bucket
- For each row, returns the number of the histogram bucket for the data
- (expr,min_value,max_value,num_buckets)
- Equiwidth function dividing data into equal interval sizes.
  - Ntile function creates equiheight buckets.

Histogram Function
Width-Bucket

SELECT salesrep_id, total_sales,

WIDTH_BUCKET(total_sales,0,1000.1,10) “sale group"

From Sales Where cityname = 'GOTHAM'

order by total_sales;

salesrep_id	total_sales	sale group
152	150	2
151	200	2
153	400	4
154	400	4
155	785	8
156	800	8
157	1000	10
158	1475	11

ID	**Sales**	**Grp**
149	-20	0
150	0	1

next->Medians

Medians in SQL

- Not supported by standard SQL: e.g. from Celko and Date. Look at an example of 4 Salaries and find the median of $80,000.
- 1st split the table in 2 & get the lowest value of the top half rows
- "Get salaries <= 2 salaries, then get the min value from these"

Input Values

Salary
1,000,000
100,000
60,000
40,000

Get lowest value of top half: 100,000

```
Select MIN(e.salary) FROM
emp_salary e  Where e.salary in
  (Select E2.salary                -- top half rows
 FROM Emp_Salary E1, Emp_Salary E2
WHERE E2.salary <= E1.salary
GROUP BY E2.salary HAVING count(*) <=
    (Select CEIL(count(*) /2) FROM Emp_Salary));
```

Medians in SQL

- Next: Get highest value of bottom half: 60,000
- "Get salaries >= 2 salaries, then get the max value from these"

```
Select MAX(e3.salary) FROM emp_salary e3  Where e3.salary in
-- get the bottom half rows below
  (Select E4.salary
 FROM Emp_Salary E5, Emp_Salary E4
 WHERE E4.salary >= E5.salary
 GROUP BY E4.salary HAVING count(*) <=  (Select
 CEIL(count(*) /2) FROM Emp_Salary));
```

- Next, combine the queries above and take the average to get this median value of $80000 as shown below:

Medians in SQL

```
Select avg(E.salary) AS median From Emp_Salary E  Where E.salary in
(Select MIN(e.salary) FROM emp_salary e
 Where e.salary in
(Select E2.salary FROM Emp_Salary E1, Emp_Salary E2 WHERE
    E2.salary <= E1.salary
 GROUP BY E2.salary HAVING count(*) <=
 (Select CEIL(count(*) /2) FROM Emp_Salary))
 UNION
 Select MAX(e3.salary) FROM emp_salary e3 Where e3.salary in
   (Select E4.salary FROM Emp_Salary E5, Emp_Salary E4
    WHERE E4.salary >= E5.salary GROUP BY E4.salary
    HAVING count(*) <=
   (Select CEIL(count(*) /2) FROM Emp_Salary)));
```

Explain for Median

```
0  SELECT STATEMENT   MODE: ALL_ROWS
1    SORT (AGGREGATE)
2      HASH JOIN
2        VIEW OF 'VW_NSO_3' (VIEW)
2          SORT (UNIQUE)
2            UNION-ALL
1              SORT (AGGREGATE)
2                HASH JOIN
2                  VIEW OF 'VW_NSO_1' (VIEW)
2                    FILTER
4                      SORT (GROUP BY)
10                       NESTED LOOPS
4                          TABLE ACCESS (FULL) OF 'EMP_SALARY' (TABLE)
10                         TABLE ACCESS (FULL) OF 'EMP_SALARY' (TABLE)
1                      SORT (AGGREGATE)
4                        TABLE ACCESS (FULL) OF 'EMP_SALARY' (TABLE)
4                  TABLE ACCESS (FULL) OF 'EMP_SALARY' (TABLE)
1          SORT (AGGREGATE)
2            HASH JOIN
2              VIEW OF 'VW_NSO_2' (VIEW)
2                FILTER
4                  SORT (GROUP BY)
10                   NESTED LOOPS
4                      TABLE ACCESS (FULL) OF 'EMP_SALARY' (TABLE)
10                     TABLE ACCESS (FULL) OF 'EMP_SALARY' (TABLE)
1                  SORT (AGGREGATE)
4                    TABLE ACCESS (FULL) OF 'EMP_SALARY' (TABLE)
4              TABLE ACCESS (FULL) OF 'EMP_SALARY' (TABLE)
4        TABLE ACCESS (FULL) OF 'EMP_SALARY' (TABLE)
```

Medians in 9i

- Can use the new inverse percentile function.
- an inverse distribution function that assumes a discrete distribution model
- An expression evaluates a value to a distribution between 0 and 1 as with cume_dist.
- The inverse percentile can then find the value at the 0.5 level.
  - In the example on the next slide this is 100,000
  - Using 0.51 would have given 60,000
  - Not quite what we want …
    - Requires more sophisticated and complex use of this function … take a look ->

Medians in 9i

```
SELECT salary, deptid, CUME_DIST() OVER
  (PARTITION BY deptid ORDER BY salary DESC)
  cume_dist,
  PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY
  salary DESC) OVER (PARTITION BY deptid)
  percentile_disc
FROM emp_salary ;
```

salary	deptid	cume_dist	percentile_dist
1000000	100	.25	100000
100000	100	.5	100000
60000	100	.75	100000
40000	100	1	100000

Medians in 10g

Finally, a Median function!

- Inverse distribution function assuming continuous distribution.
- Null values are ignored.
- Numeric datatypes and nonnumeric ones can be converted to numeric.
- Median first orders the rows.
- With N as the number of rows, Oracle determines the median row number (MRN) as:

  MRN = 1 + (0.5*(N-1))

Medians in 10g

- Once Oracle has determined the MRN, it gets ceiling row number (CRN) and floor row number (FRN) and uses these to get the Median.
- CRN=ceiling(RN) and FRN = floor(RN)
  - Odd number of rows:
    - If (CRN=FRN=RN) then median = RN_value
  - Even Number of rows:
    - e.g. 4 rows: ("row 3 value" * .5) + ("row 2 value" * .5)

```
Select deptid, median(salary)
From Emp_Salary
Group By deptid ;
```

Explain for the Median Function

```
Select deptid, median(salary)
From emp_salary
Group By deptid;
```

```
Rows   Execution Plan
   0   SELECT STATEMENT   MODE: ALL_ROWS
   1     SORT (GROUP BY)
   4       TABLE ACCESS (FULL) OF 'EMP_SALARY' (TABLE)
```

Full Outer Joins – 9i/10g

- Oracle9i/10g has "many" SQL92 and 99 features
- Old Proprietary Outer join:

select t1.t1col01, t2.t2col01

from t1, t2

where t1col01 = t2col01 (+)

UNION

select t1.t1col01, t2.t2col01

from t1 , t2

where t1.t1col01 (+) = t2.t2col01;

Result:

t1col	t2col
C1	
C5	C5
	C9
	C4

Full Outer Joins – 9i/10g

- THE SAME RESULT AS

```
Select t1.t1col01, t2.t2col01
From t1 FULL OUTER JOIN t2
ON t1col01 = t2col01
order by t1.t1col01;
```

- Ansi SQL99 compliant syntax for full, left, right outer joins.
- Above is slightly more efficient that the outer join on the previous slide.

See Explain on the next slide →

Rows	Execution Plan Full Outer Join with **Union (OLD) Syntax**
0	SELECT STATEMENT MODE: ALL_ROWS
3	SORT (UNIQUE)
4	UNION-ALL
2	HASH JOIN (OUTER)
2	TABLE ACCESS (FULL) OF 'T1' (TABLE)
2	TABLE ACCESS (FULL) OF 'T2' (TABLE)
2	**HASH JOIN (OUTER)**
2	TABLE ACCESS (FULL) OF 'T2' (TABLE)
2	TABLE ACCESS (FULL) OF 'T1' (TABLE)

Rows	Execution Plan Full Outer Join **New Syntax**
0	SELECT STATEMENT MODE: ALL_ROWS
3	SORT (ORDER BY)
3	VIEW
3	UNION-ALL
2	HASH JOIN (OUTER)
2	TABLE ACCESS (FULL) OF 'T1' (TABLE)
2	TABLE ACCESS (FULL) OF 'T2' (TABLE)
1	**HASH JOIN (ANTI)**
2	TABLE ACCESS (FULL) OF 'T2' (TABLE)
2	TABLE ACCESS (FULL) OF 'T1' (TABLE)

next->Partitioned Outer Joins

Partitioned Outer Joins in 10g

- Can convert sparse data into dense data.
- aka: a Group Join
  - Union of outer joins
- Logically partitioned data based on “Partition By” clause.
- Accepted by ISO and ANSI for SQL standards
- E.g. HR application tracks the number of employees in the company every week. If the number does not change, no entry is inserted into the emp_count table. We want an entry for every week regardless of whether this number has changed.

Partitioned Outer Joins in 10g- Example

Select *

From emp_count;

REG	WEEK	COUNT
R1	01-JAN-05	1023
R1	08-JAN-05	1030
R1	22-JAN-05	1033
R1	05-FEB-05	1032

Select start_dt from week

Start_dt
01-JAN-05
08-JAN-05
15-JAN-05
22-JAN-05
29-JAN-05
05-FEB-05

Select region, start_dt, count

From emp_count

Partition By (region)

Right Outer Join week

On (emp_count.week=week.start_dt)

Order by region, start_dt;

REG	START_DT	COUNT	
R1	01-JAN-05	1023	
R1	08-JAN-05	1030	
R1	15-JAN-05		←s.b. 1030
R1	22-JAN-05	1033	
R1	29-JAN-05		←s.b. 1033
R1	05-FEB-05	1032	

Continued on next slide →

Partitioned Outer Join Explain

```
Select region, start_dt, count
From emp_count
Partition By (region)
Right Outer Join week
On (emp_count.week=week.start_dt)
Order by region, start_dt;

Rows    Execution Plan
-------  --------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
      6    VIEW
      6      MERGE JOIN (PARTITION OUTER)
      7        SORT (JOIN)
      6          TABLE ACCESS (FULL) OF 'WEEK' (TABLE)
      4        SORT (PARTITION JOIN)
      4          TABLE ACCESS (FULL) OF 'EMP_COUNT' (TABLE)
```

Partitioned Outer Joins in 10g- Example (cont.) Making Partitioned Data Dense

Select region, start_dt,
LAST_VALUE(count *ignore nulls*)
OVER (Partition By region
Order by region, start_dt) week
FROM (
Select region, start_dt, count
From emp_count
Partition By (region)
Right Outer Join week
On (emp_count.week=week.start_dt))
Order by region, start_dt;

REG	START_DT	COUNT	
R1	01-JAN-05	1023	
R1	08-JAN-05	1030	
R1	15-JAN-05	1030	←dense
R1	22-JAN-05	1033	
R1	29-JAN-05	1033	←dense
R1	05-FEB-05	1032	

Making Partitioned Data Dense Explain

```
Select region, start_dt, LAST_VALUE(count ignore nulls)
OVER (Partition By region Order by region, start_dt) week
FROM (Select region, start_dt, count
From emp_count
Partition By (region)
Right Outer Join week
On (emp_count.week=week.start_dt))
Order by region, start_dt;

Rows    Execution Plan
------- ------------------------------------------------
      0   SELECT STATEMENT   MODE: ALL_ROWS
      6     WINDOW (BUFFER)        ← new from the last explain
      6       VIEW
      6         MERGE JOIN (PARTITION OUTER)
      7           SORT (JOIN)
      6             TABLE ACCESS (FULL) OF 'WEEK' (TABLE)
      4           SORT (PARTITION JOIN)
      4             TABLE ACCESS (FULL) OF 'EMP_COUNT' (TABLE)
```

Group By with Rollup

- Group by can perform a function on a grouping

  Select region, territory, sum(sales_dollars) TOTAL_SALES

  From sales group by region, territory;

- Results in:

REGION	TERRITORY	TOTAL_SALES
EAST	1	1500.00
EAST	2	2000.00
WEST	1	3000.00
WEST	2	500.00

- ROLLUP extends this and can also summarize at the Region level by creating superaggregates.

Group By with Rollup

Select nvl(region,'Total Company') REGION,
nvl(territory, 'Total Region') TERRITORY,
sum(sales_dollars) TOTAL_SALES
FROM sales GROUP BY ROLLUP(region, territory);

- Results in (note substitution of literals from nvl):

REGION	TERRITORY	TOTAL_SALES
EAST	1	1500.00
EAST	2	2000.00
EAST	Total Region	3500.00
WEST	1	3000.00
WEST	2	500.00
WEST	Total Region	3500.00
Total Company		7000.00

Group By with Cube

- Cube generate superaggregates by giving totals for each Territory regardless of Region (as one example).
- Cube gives us totals for all combinations of Columns chosen in the Group By clause for OLAP Services
- Select decode(grouping(region),1,'Total Company', region),

  decode(grouping(territory),1, 'Total Region', territory),

  sum(sales_dollars) Total_Sales

  FROM sales

  GROUP BY CUBE (region, territory);
- Decode is a translation that changes the grouping indicator of '1' to another value of 'Total Company' or 'Total Region'.
- Rollup and Cube return a value of 1 if NULL results from CUBE or ROLLUP and returns 0 if it is a natural result.

Group By with Cube

- Cube result From the previous query

REGION	TERRITORY	TOTAL_SALES
EAST	1	1500.00
EAST	2	2000.00
EAST	Total Region	3500.00
WEST	1	3000.00
WEST	2	500.00
WEST	Total Region	3500.00
Total Company	1	4500.00
Total Company	2	2500.00
Total Company	Total Region	7000.00

Explain for Group By with Cube

```
Select decode(grouping(region),1,'Total Company', region),
 decode(grouping(terr),1, 'Total Region', terr),
 sum(total_sales) Total_Sales
 FROM sales
 GROUP BY CUBE (region, terr);
```

```
Rows     Row Source Operation
   0     SELECT STATEMENT   MODE: ALL_ROWS
   9       SORT (GROUP BY)
  16         GENERATE (CUBE)
   4           SORT (GROUP BY)
  10             TABLE ACCESS (FULL) OF 'SALES' (TABLE)
```

Grouping Sets

- Enhances groupings with Cube and Rollup
- Can specify the exact level of aggregation.
- Aggregations across 3 different groupings.
  - Cube needs many groupings
  - Union All uses 3 queries

```
Select month, terr, prod,
sum(total_sales) sum_sales
From sales
Group By Grouping Sets
((month, terr, prod),
(month, prod), (terr, prod));
```

Input data

month	prod	terr	total_sales
1	1	8	200
2	1	8	150
1	2	8	300
2	2	8	400
1	1	9	1400
2	1	9	1600
1	2	9	1500
2	2	9	1475

Grouping Sets Query Result

MONTH	TERR	PROD	SUM_SALES
1	1	8	200
1	2	8	300
1	1	9	1400
1	2	9	1500
2	1	8	150
2	2	8	400
2	1	9	1600
2	2	9	1475
1		8	500
1		9	2900
2		8	550
2		9	3075
	1	8	350
	1	9	3000
	2	8	700
	2	9	2975

Grouping Sets

- Prunes the aggregates you don’t need.
  - Does not aggregate as much as cube or rollup.
    - BUT the access path is not as efficient!
- Computes all groupings in Grouping Sets and combines results with a Union All.
- Composite columns can be specified by grouping columns in parentheses to be treated as a single unit by the Cube or Rollup.
- Concatenated groupings let you take multiple grouping sets, cube or rollup operations and separate them with commas to form a Group By

Grouping Sets Explain

Select month, terr, prod, sum(total_sales) sum_sales From sales
Group By Grouping Sets ((month, terr, prod), (month, prod), (terr, prod));

```
Rows    Execution Plan
   0  SELECT STATEMENT   MODE: ALL_ROWS
  21   TEMP TABLE TRANSFORMATION
   0     MULTI-TABLE INSERT
   0       DIRECT LOAD INTO OF 'SYS_TEMP_0FD9D6605_88850'
   0       DIRECT LOAD INTO OF 'SYS_TEMP_0FD9D6606_88850'
   0         SORT (GROUP BY ROLLUP)
   0           TABLE ACCESS (FULL) OF 'SALES' (TABLE)
   0     LOAD AS SELECT
  21       SORT (GROUP BY)
  21         TABLE ACCESS (FULL) OF 'SYS_TEMP_0FD9D6605_88850' (TABLE (TEMP))
  21     VIEW
  10       VIEW
  11         UNION-ALL
   2           TABLE ACCESS (FULL) OF 'SYS_TEMP_0FD9D6605_88850' (TABLE (TEMP))
  15           TABLE ACCESS (FULL) OF 'SYS_TEMP_0FD9D6606_88850' (TABLE (TEMP))
```

next->Model

10g Inter-row and Inter-array Calculations: The Model Clause

- Another use of analytic capabilities
- Spread-sheet type functionality
  - But, this is not Excel!
- Map columns into partitions, dimensions and measures
  - Partitions: viewed as an independent array
  - Dimensions: cells in a partition to define characteristics.
  - Measures: data cells (aka facts).
- Model clause is processed after all other clauses except Order By.
- "return updated rows" clause only displays changed rows.
- To insert, update or merge values in a table, you need to use the Model results as input to the insert, update, merge statement.

The Model Clause: Example

Eg: Project Next quarters Sales based on the last 2 quarters.

INPUT:

Select region reg, product prod, quarter qtr, sales from Qtr_Sales;

Reg	Prod	Qtr	Sales
E	1	04Q4	4550
E	1	05Q1	5000
E	2	04Q4	6300
E	2	05Q1	6700
W	1	04Q4	7900
W	1	05Q1	7700
W	2	04Q4	2500
W	2	05Q1	4000

Select * from Qtr_Sales

MODEL *return updated rows*

Partition By (region)

Dimension By (product,quarter)

Measures(sales)

Rules (

sales[1,'05Q2']=sales[1,'05Q1']-sales[1,'04Q4']+sales[1,'05Q1'] ,

sales[2,'05Q2']=sales[2,'05Q1']*0.5)

Order by region, product;

QUERY RESULT

Reg	Prod	Qtr	Sales
E	1	05Q2	5450
E	2	05Q2	3350
W	1	05Q2	7500
W	2	05Q2	2000

Explain of the Model Clause

```
Select * from Qtr_Sales
MODEL return updated rows
Partition By (region)
Dimension By (product,quarter)
Measures(sales)
Rules (
sales[product=1,quarter='05Q1']=sales[1,'05Q1']-
    sales[1,'04Q4']+sales[1,'05Q1'] ,
sales[2,'05Q1']=sales[2,'05Q1']*0.5)
order by region, product;

Rows     Execution Plan
-------  --------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
      4    SORT (ORDER BY)
      4      SQL MODEL (ORDERED FAST)
      8        TABLE ACCESS   MODE: ANALYZED (FULL) OF 'QTR_SALES'
```

The Model Clause

Dimensions

- A cell reference must qualify all dimensions in the “dimension by” clause.
- Positional Reference
  - Dimension By (product,quarter) … sales[2,’05Q2’]=sales[2,’05Q1’]*0.5)
- Symbolic Reference
  - sales[product=2,quarter=’05Q1’]=…
  - Only for updating existing cells. If the second quarter of 05 has no data yet, then no rows will be returned for the following:
    - sales[product=2,quarter=’05Q2’]=…

The Model Clause

Ordering of Rules

- Sequential (the default)
  - The order the rules are listed in the Model clause.

  Select … Model … Rules Sequential Order
- Automatic
  - Dependencies are evaluated and processes depending on this order.

  Select … Model … Rules Automatic Order ….

The Model Clause
Current Value Function cv()

- Apply specs from the left side of a formula to the right.
- Like a short form version of a join condition.

```
Select * from Qtr_Sales
MODEL return updated rows
Partition By (region)
Dimension By (product,quarter)
Measures(sales)
Rules (
sales[2,quarter between '04Q4' and
    '05Q1']=sales[1,CV(quarter)]*1.1)
Order by region, product;
```

R	PRODUCT	QUAR	SALES
E	2	04Q4	5005
E	2	05Q1	5500
W	2	04Q4	8690
W	2	05Q1	8470

Example of the Model Clause with the Current Value Function

```
Select * from Qtr_Sales
MODEL return updated rows
Partition By (region)
Dimension By (product,quarter)
Measures(sales)
Rules (
sales[2,quarter between '04Q4' and '05Q1']=sales[1,CV(quarter)]*1.1)
Order by region, product;

Rows    Execution Plan
------- ------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
      4    SORT (ORDER BY)
      4      SQL MODEL (ORDERED)
      8        TABLE ACCESS  MODE: ANALYZED (FULL) OF 'QTR_SALES'
```

The Model Clause
Reference Models and Main Models

- Reference models
  - Allow you to reference many multi-dimensional arrays from a Main model.
  - Reference models are read-only & used as lookup tables.
  - Reference models have a Name.
  - Cannot have a Partition clause.
- Main Model
  - The multi-dimensional array that has its cells updated.
  - Can have one or more reference models.

The Model Clause
Reference Models and Main Models - Example

- Let’s look at an example that uses a reference model of inflation rates by quarter. This will be used in the main model to convert sales dollars into today’s dollars taking inflation rates into consideration.

Reference Model Select Statement

```
select qtr, rate_pct as rt from
     inflation_rate;
```

Qtr	rate_pct
04Q4	1
05Q1	0.5

Qtr_Sales table used in the Main Model.

```
select * from qtr_sales
where product=1 and
quarter='04Q4';
```

R	PRODUCT	QUAR	SALES
E	1	04Q4	4550
W	1	04Q4	7900

SUM ………… 12450

The Model Clause
Reference Models and Main Models - Example

<u>Query with Main and Reference Model</u>

```
Select product, quarter, sl from Qtr_Sales
Group By product, quarter
MODEL return updated rows
Reference inf_rate on (
select qtr, rate_pct
  as rt from inflation_rate)
Dimension By (qtr)  Measures (rt)
MAIN sale_model
Dimension By (product,quarter)
Measures(sum(sales) sales, 0 sl)
Rules (sl[1,'04Q4']=sales[1,'04Q4'] +
(sales[1,'04Q4']*inf_rate.rt['05Q1']/100) +
(sales[1,'04Q4']*inf_rate.rt['04Q4']/100) );
```

The query on the left uses the inflation rate as input and calculates the current value of ’04Q4’ sales by multiplying it be the ’04Q4’ inflation rate and the ’05Q1’ rate. The result is below. Note that 12636.75 = 12450 + 124.50 + 62.25

<u>RESULT OF THE QUERY</u>

PRODUCT	QUAR	SL
1	04Q4	12636.75

Model Example: Reference Models and Main Model

```
Select product, quarter, sl from Qtr_Sales
Group By product, quarter
MODEL return updated rows
Reference inf_rate on (select qtr, rate_pct as rt from inflation_rate)
Dimension By (qtr)      Measures (rt)      MAIN sale_model
Dimension By (product,quarter)
Measures(sum(sales) sales, 0 sl)
Rules (sl[1,'04Q4']=sales[1,'04Q4'] +
(sales[1,'04Q4']*inf_rate.rt['05Q1']/100) +
(sales[1,'04Q4']*inf_rate.rt['04Q4']/100) ) ;
```

```
Rows    Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
      1    SQL MODEL (ORDERED FAST)
      2      REFERENCE MODEL OF 'INF_RATE'
      2        TABLE ACCESS (FULL) OF 'INFLATION_RATE' (TABLE)
      1      FILTER
      1        SORT (GROUP BY)
      2          TABLE ACCESS   MODE: ANALYZED (FULL) OF 'QTR_SALES'
```

The Model Clause
Other Features

- "For loop" can be used.
  - Rules (sales[FOR product in (2,3),quarter between '04Q4' and '05Q1']=sales[1,CV(quarter)*1.1])
  - 1 formula to calculate many cells.
- Ignoring Nulls
  - Use the IGNORE NAV feature to substitute a value for NULLS.
    - Defaults: number=0; date=01-JAN-2000;char=' '.
    - Select … Model Ignore_NAV return updated rows … ;
- Iterate clause to calculate formulas iteratively for a certain number of times.

Bonus Section
If time allows!

- Joins, sub-queries and anti-joins
- Note: in 9i and 10g access paths are improved
  - The optimizer is better able to distinguish and rewrite queries in the optimal manner. For example, you will now often find correlated and non-correlated queries take the same access path in 10g.

Comparing Joins: Nested Loop

- Along with merge-scan, the most common type. e.g.
  - Select * From Table1 T1, Table2 T2
    Where T1.Table1_Id = T2.Table1_id;
- for each row in the outer table (Table1), the inner table (Table2) will be accessed with an index to retrieve the matching rows. The next row on Table1 is then retrieved and matched to Table2.
- efficient index access is needed on the inner table
- Commonly used in OLTP apps.
- Useful for a small number of rows & first_rows parm.
- Cluster Joins are a special case of Nested-Loop join
  - Have many drawbacks and are rarely used

Comparing Joins: Merge Scan

- aka. sort-merge. Useful for:
  - processing a large number of rows.
  - inefficient index access and sorted data
  - Batch processing and all_rows goal
  - Inequality clause <, <=, > or >=
- Fast because of:
  - database multi-block fetch (helped by init.ora parm db_file_multiblock_read_count) capabilities
  - The fact that each table is accessed once
  - Faster than hash joins if rows are already sorted and sorts do not need to be performed. Otherwise use hash join.
- Steps performed for Merge-Scan are:

Comparing Joins: Merge-Scan

1) Pick an inner and outer table
2) Access the inner table, choose the rows that match the predicates in the Where clause of the SQL statement
3) Sort the rows retrieved from the inner table by join columns, store these as a Temp table. This step is not performed if data is ordered by the keys and efficient index access exists.
4) outer table may also be sorted by the join columns so both tables to be joined are sorted the same way. This step is optional and dependent on whether the outer table is well ordered by the keys and whether efficient index access can be used.
5) Read outer & inner (likely sorted temp) tables, get rows that match the join criteria. This is quick due to sorted data.
6) Optionally sort the data if a Sort was performed (e.g. 'Order By') using different columns than used to perform the join.

Comparing Joins: Hash Join

- very efficient join when used in the right situation: when 1 of the 2 tables is small and fits in memory.
- The larger of the 2 tables is chosen as the Outer table
- Outer and inner are broken into sections and the inner Tables join columns are stored in memory (if hash_area_size is large enough) and 'hashed'.
  - hashing provides an algorithmic pointer that makes data access very efficient.
  - Oracle attempts to keep the inner table in memory since it will be 'scanned' many times.
  - Outer rows that match the query predicates are then selected and for each Outer table row chosen, hashing is performed on the key and the hash value is used to quickly find the matching row in the Inner Table.

Comparing Joins: Hash-Join

- No sorting is performed and index access can be avoided since the hash algorithm is used to locate the block where the inner row is stored.
- Hash-joins are also only used for equi-joins.
- Use init.ora parm pga_aggregate_target to automatically size sql working areas.

Comparing Joins: Star-Joins

- A join common to Data Marts and Data Warehouses.
- a join of a large "Fact" table with 2 or more smaller tables commonly called "Dimensions". Fact tables have transactional properties. The Dimensional tables are used to describe the Fact table (customer, product).
- Star queries get their name because there is a central Fact table surrounded by smaller dimensional tables that are directly related to the Fact table
- Consider the case of the central Fact table that is being joined to 3 smaller Dimensional table. They are transformed from the written query on the left below to the transformed one on the right.

Comparing Joins: Star Joins

ORIGINAL QUERY

```
SELECT *
FROM Fact, Dim1, Dim2,
 Dim3
WHERE
 Fact.dim1_id = Dim1.id  and
 Fact.dim2_id = Dim2.id  and
 Fact.dim3_id = Dim3.id and
 Dim1.name like :in_var1 and
 Dim2.desc between :in_var2
     and :in_var3
and Dim3.Text < :in_var4;
```

TRANSFORMED QUERY

```
SELECT *
FROM Fact, Dim1, Dim2, Dim3
WHERE  Fact.dim1_id in
    (SELECT dim1.id from dim1
     WHERE dim1.name like :in_var1)
 and Fact.dim2_id in
    (SELECT dim2.id from dim2
     WHERE Dim2.desc between
      :in_var2 and :in_var3)
 and Fact.dim3_id in
    (SELECT dim3.id FROM dim3
     WHERE Dim3.Text < :in_var4);
```

Comparing Joins: Star-Joins

- The subselects are performed first. Bitmap indexes on Fact join columns, are merged (in this case ANDed) & Fact rows can be accessed using the resulting index values. The Fact rows retrieved are then joined to the Dimensions to complete the query.
- Using this approach, a Cartesian product is not required.
  - e.g. 3 dim table cartesian product of 10,000 * 10,000 * 10,000 = 1,000,000,000,000 rows
- To implement star_query transformation:
  - Set init.ora parm star_transformation_enabled=true
  - Create bitmap indexes for all of the foreign-key columns on the fact table
  - Implement R.I. Only between the fact table and the dimension tables.

next->Subselects

Dealing With Subqueries
In and Exists

- Correlated Subquery (EXISTS): What is it?
  - A subquery is Correlated when it is joined to the outer query within the Subquery. E.g.
    - Select * From Cust Where cust.city = 'Chicago' and Exists (Select cust_id From Sales s where s.ttl_sales > 10000 and sales.cust_id = Cust.cust_id);
  - the last line in the above query is a join of the outer Cust table and inner Sales tables. The outer query is read and each outer row (Cust = 'Chicago') is joined to the Subquery. i.e., the inner query is executed once for every row read in the outer query.
  - efficient where a small number of rows are processed usually due to efficient index access to the inner table for a small number of rows- not when a large number of rows are read.

Dealing with Subqueries

Non-correlated Subquery (IN): What is it?

- A subquery is said to be uncorrelated when the two tables are not joined together in the inner query. The inner (sub) query is processed 1st and the temporary result set table is joined to the outer table. E.g.
  - ```
    Select last_name, first_name
    From Customer Where customer_id IN
     (Select customer_id From Sales where
      sales.total_sales_amt > 10000);
    ```
- Sales table is processed first and all entries with a total_sales_amt > 10000 will be joined to the Customer table.
- Efficient where a large number of rows is being processed.
    ```

Turn Subqueries into Joins

- When possible, use joins rather than subqueries
- The query on the previous slide becomes:
  - Select cust.last_name, cust.first_name
    From Customer cust, Sales
    Where cust.customer_id = sales.customer_id
    and sales.total_sales_amt > 10000;
- Gives the optimizer more choices when deciding on query plan
  - optimizer can choose between nested loop, merge scan, hash and star joins when a Join is used.
  - The options are limited when the compiler and optimizer are presented with a Subquery.

In vs. Exists

- Use a join where possible
- IN executes subquery once. Exists executes subquery once per outer-table row
- IN is like merge-scan.
- Exists is like nested-loop join.
- EXISTS tries to satisfy the subquery as quickly as possible and returns ‘true’ if the subquery returns 1 or more rows –> it should be indexed. Optimize execution of the subquery.
- You need to understand the number of rows processed and the access paths being used.
  - Large outer-table and small inner-table generally favors “in” over “exists”.
  - Small outer-table *result set* and large *well indexed* inner-table generally favors “exists” over “in”.

Comparing In and Exists

Access Paths

'IN' Access Plan: same as Merge Join

```
Rows     Execution Plan
-------  ----------------------------------------------
      0  select statement  goal: choose
      1    sort (aggregate)
  20000      merge join
  20001        sort (join)
  20000          view of 'VW_NSO_1'
  20000            sort (unique)
  20000              table access goal: analyzed full of 'cust_addr_test'
  20000        sort(join)
  25000          table access  goal: analyzed (full) of  'CUST_TEST'
```

Comparing In and Exists
Access Paths

- Exists access plan: Same as Nested Loop Join

```
Rows     Execution Plan
-------  ---------------------------------------------
    0    select statement  goal: choose
    1      sort (aggregate)
20000        filter
25001          table access goal: analyzed full of 'cust_test'
25000          table access goal: analyzed full of
                   'cust_addr_test'
```

Not In vs. Not Exists

- Subqueries may use NOT IN and NOT EXISTS
- BUT, they are different! Be careful of NOT IN and null values!
  - NOT IN: if the subquery returns NULLS, the results will NOT be returned.
  - NOT EXISTS: a value in the outer query that has a NULL value in the inner will be returned.
- NOT IN can perform well particularly when the access path is a “hash anti-join” and can often outperform a Not Exists
  - NOT IN with /*+ HASH_AJ */ hint is very fast
  - Optimizer uses nested loop algorithm by default unless a hint is used (unlike IN which used merge join).
- NOT EXISTS can <u>sometimes</u> be more efficient since the database only needs to verify non-existence.
  - With NOT IN the entire result set must be materialized.
  - Also uses nested loop algorithm by default

Comparing Not In vs. Not Exists

- Comparison with indexes: Using NOT IN

  SELECT count(*) FROM cust_test ct WHERE ct.cust_no NOT IN (select cat.cust_no from cust_addr_test cat)

call	count	cpu	elapsed	disk	query	current	rows
total	4	291.24	298.83	0	5287981	115946	1

ROWS	EXECUTION PLAN
0	select statement goal: choose
1	sort (aggregate)
5000	filter
25001	table access goal: analyzed full of 'cust_test'
25000	table access goal: analyzed full of 'cust_addr_test'

Comparing Not In vs. Not Exists

- Comparison with indexes: Using NOT EXISTS

  SELECT count(*) FROM cust_test ct WHERE

  NOT EXISTS (select 1 from cust_addr_test cat where cat.cust_no = ct.cust_no)

call	count	cpu	elapsed	disk	query	current	rows
total	4	0.36	0.36	0	50359	5	1

ROWS	EXECUTION PLAN
0	select statement goal: choose
1	sort (aggregate)
5000	filter
25001	table access goal: analyzed full of ‘cust_test’
25000	index goal: analyzed (unique scan) of ‘cust_addr_test_a01’ (unique)

Comparing Not In vs. Not Exists

Be careful of Not In with Null Values

```
select count(*) from t1
   where col01 not in
(select col01 from t2);

    COUNT(*) result = 0
```

```
select count(*) from t1
where not exists
(select 1 from t2
where t1.col01=t2.col01);

      COUNT(*) result = 3
```

Solved By:

```
select count(*) from t1 where col01 not in
(select col01 from t2 where col01 is not null);
```

Comparing Not In vs. Not Exists

Other fast options for performing anti-joins:

- Hash Anti-Join is often be the quickest approach.
- Outer join query is also a very fast way to do this.
  - select count(*) TEST6_with_indexes

    from cust_test ct, cust_addr_test cat

    where ct.cust_no = cat.cust_no (+) and cat.cust_no is null
- You can also use Minus to perform anti-join BUT:
  - Like a Union, number of columns and types must match. It is limiting

Conclusion

- Become familiar with Oracle's SQL functions.
- Try out functions as well as your own solutions to problems. This can help you improve your SQL skills and can build a repertoire that will help your most experienced developers.
- SQL is becoming more complicated.
- We can now do things in native SQL that used to only be possible in advanced query packages.
  - Get familiar with Oracle's supplied PL/SQL packages
- Oracle is becoming more ansi SQL92 and 99 compliant
- Also with 10g: regular expressions
- SQL can be fun! (or at least interesting)