Analytical Functions in ORACLE 8i

By Ed Kosciuszko

SeQueL Consulting
sequelconsulting@msn.com
(973) 226-7835

Copyright 2001 - SeQuel Consulting

New SQL Features

Post Processing

- scan results to compute function of selected row set
- for each row in result set, apply function to specified rows
- display aggregate with details for easy comparison or as summary row

Dynamic Table

- use query to define table in FROM clause
- allow multiple levels of filtering the result set

Post Processing

Display sum of salaries per department as portion of the total company salaries.

Without post-processing

```
CREATE VIEW co_tot_sal (total_sal)
AS SELECT SUM(sal) FROM emp
```

→

```
SELECT deptno, (SUM(sal)/total_sal)*100
FROM emp e, co_tot_sal c
GROUP BY deptno, total_sal
```

With post-processing

```
SELECT deptno, (SUM(sal)/SUM(SUM(sal)) OVER ())*100
FROM emp
GROUP BY deptno
```

Simplistic and Efficient

Copyright 2001 - SeQuel Consulting

Details & Summary Data

List sum of salaries per group defined by the same job and deptno.

```
SELECT deptno, job, SUM(sal)
FROM emp
GROUP BY deptno, job
```

DEPTONO	JOB	
10	CLERK	1300
10	MANAGER	2450
10	PRESIDENT	5000
20	ANALYST	6000
20	CLERK	1900
20	MANAGER	2975
30	CLERK	950
30	MANAGER	2850
30	SALESMAN	5600

Details can’t be displayed with summary data.

Comparing detail data with summary data requires views.

Details & Summary Data

```
SELECT empno, deptno, job,
     SUM(sal) OVER
        (PARTITION BY deptno, job) AS sum_sal
FROM emp
```

EMPNO	DEPTNO	JOB	SUM_SAL
7934	10	CLERK	1300
7782	10	MANAGER	2450
7839	10	PRESIDENT	5000
7788	20	ANALYST	6000
7902	20	ANALYST	6000
7369	20	CLERK	1900
7876	20	CLERK	1900
7566	20	MANAGER	2975
7900	30	CLERK	950
7698	30	MANAGER	2850
7499	30	SALESMAN	5600
7654	30	SALESMAN	5600
7844	30	SALESMAN	5600
7521	30	SALESMAN	5600

PARTITION identifies rows to aggregate. Rows must have the same DEPTNO and JOB value as the detail row.

PARTITIONS

SYNTAX:

SUM (column/expression) OVER ([PARTITION BY col/express, [col/express, …]])

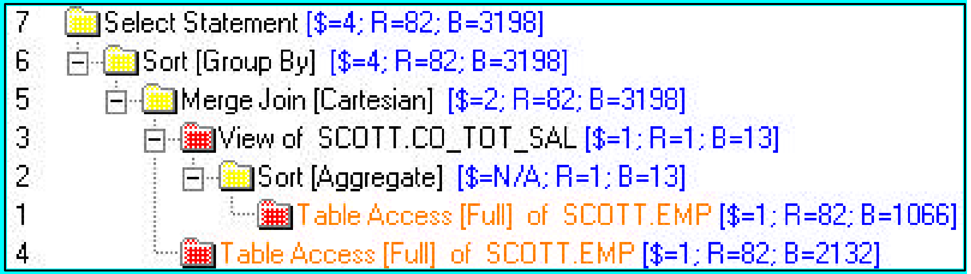
Each detail row can have multiple analytical functions, each with a different partition.

PARTITION clause is optional. If omitted entire result set is the partition.

PARTITION can be defined by multiple columns/expressions. If SQL module has GROUP BY, column/expressions limited to those on the SELECT list.

Internal Operations

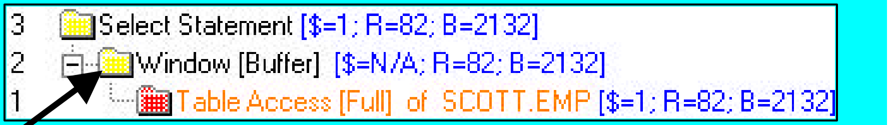
```
SELECT deptno, SUM(sal)/total_sal)
FROM emp e, co_tot_sal c
GROUP BY deptno, total_sal
```



```
7  Select Statement [$=4; R=82; B=3198]
6    Sort [Group By]  [$=4; R=82; B=3198]
5      Merge Join [Cartesian]  [$=2; R=82; B=3198]
3        View of  SCOTT.CO_TOT_SAL [$=1; R=1; B=13]
2          Sort [Aggregate]  [$=N/A; R=1; B=13]
1            Table Access [Full]  of  SCOTT.EMP [$=1; R=82; B=1066]
4        Table Access [Full]  of  SCOTT.EMP [$=1; R=82; B=2132]
```

Both SQL statements produce the same results but at different costs.

```
SELECT empno,
     (sal/SUM(sal) OVER () ) AS percent
FROM emp
```



```
3  Select Statement [$=1; R=82; B=2132]
2    Window [Buffer]  [$=N/A; R=82; B=2132]
1      Table Access [Full]  of  SCOTT.EMP [$=1; R=82; B=2132]
```

Window [Buffer] operation is the post-processing, scanning the result set to compute analytical function.

Ranking Results

Position in sorted list is different from rank in list.

Ties are given different positions but the same rank.

SELECT empno, sal, **RANK**() OVER (ORDER BY sal) Rank_Values,
DENSE_RANK () OVER (ORDER BY sal) Dense_Rank_Values FROM emp

EMPNO	SAL	RANK	DENSE_RANK
7369	800	1	1
7900	950	2	2
7876	1100	3	3
7521	1250	4	4
7654	1250	4	4
7934	1300	6	5
7844	1500	7	6
7499	1600	8	7
7782	2450	9	8
7698	2850	10	9
7566	2975	11	10
7788	3000	12	11
7902	3000	12	11
7839	5000	14	12

DENSE_RANK does not skip rank values due to tie.

Highlighted rows have same SAL value, so same rank. Subsequent ranks differ

Copyright 2001 - SeQuel Consulting

RANK & DENSE_RANK

Syntax:

```
RANK () OVER ([PARTITION BY col/express [,col/express, …] ]
          ORDER BY col/express [,…] [ASC|DESC] [NULLS FIRST|NULLS LAST]
```

- **RANK does not take parameter**
- **ORDER BY is mandatory**
- **ORDER BY clause like that in standard SQL along with option to specify collation order and handling of NULLs**
- **PARTITION is optional. Default is entire result set.**

Criteria Referencing Analytical Functions

Top 2 salary earners

```
SELECT empno, sal, rank_value
FROM (SELECT empno, sal,
                RANK() OVER ( ORDER BY sal DESC) AS rank_value
          FROM emp)
WHERE rank_value <=2
```

FROM query enables us to post-process a result set.

EMPNO	SAL	RANK_VALUE
7839	5000	1
7788	3000	2
7902	3000	2

Query returns 3 rows due to tie for 2nd place.

RANK or DENSE_RANK?

**RANK and DENSE_RANK only differ on skipping rank values due to tie.
Which is appropriate for which application?**

```
SELECT empno, sal, rank_value
FROM (SELECT empno, sal,
           RANK() OVER ( ORDER BY sal DESC) AS rank_value
        FROM emp)
WHERE rank_value <=3
```

EMPNO	SAL	RANK_VALUE
7839	5000	1
7788	3000	2
7902	3000	2

```
SELECT empno, sal, rank_value
FROM (SELECT empno, sal,
           DENSE_RANK() OVER ( ORDER BY sal DESC)
                           AS rank_value
        FROM emp)
WHERE rank_value <=3
```

EMPNO	SAL	RANK_VALUE
7839	5000	1
7788	3000	2
7902	3000	2
7566	**2975**	**3**

RANK doesn't return this row due to tie.

RANK or DENSE_RANK?

EMPNO	SAL	RANK	DENSE_RANK
7839	5000	1	1
7788	3000	2	2
7902	3000	2	2
7566	2975	4	3
7698	2850	5	4
7782	2450	6	5
7499	1600	7	6
7844	1500	8	7
7934	1300	9	8
7521	1250	10	9
7654	1250	10	9
7876	1100	12	10
7900	950	13	11
7369	800	14	12

Use RANK to extract top or bottom rows based on sort values.

Use DENSE_RANK to extract the nth largest or smallest value.

Copyright 2001 - SeQuel Consulting

TOP / BOTTOM

Top 5 employees in terms of hours worked.

```
SELECT *
FROM
   (SELECT emp_seq , SUM (hours ) AS sum_hrs
   FROM time_sheets
   GROUP BY emp_seq )
WHERE 5 >=
   (SELECT COUNT (COUNT (* ) )
   FROM time_sheets
   GROUP BY emp_seq
   HAVING SUM (hours ) > sum_hrs )
```

Query must be embedded in FROM clause in order to have correlated subquery access SUM(hours) per employee.

TIME_SHEETS contains 13,939,925 rows.

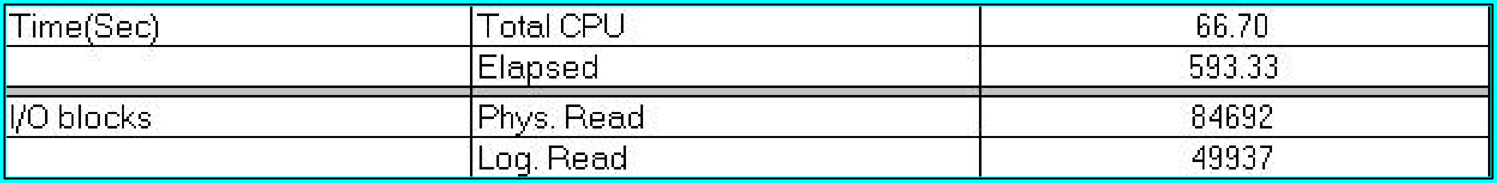
Execution time = Not in your lifetime!

Continued>

TOP / BOTTOM

Using RANK function instead.

```
SELECT *
 FROM (SELECT emp_seq, SUM(hours),
              RANK () OVER (ORDER BY SUM(hours) DESC) AS rnk
              FROM time_sheets
              GROUP BY emp_seq)
 WHERE rnk  <= 5
```



Time(Sec)	Total CPU	66.70
	Elapsed	593.33
I/O blocks	Phys. Read	84692
	Log. Read	49937

TOP / BOTTOM

Last 10 employees hired, and if there is a tie, the youngest employee is ranked lower.

```
SELECT emp_seq, hiredate, birthdate
FROM employees  e1
WHERE 10 > (SELECT count(*) FROM employees e2
                        WHERE e2.hiredate > e1.hiredate
                        OR (e2.hiredate = e1.hiredate AND
                                               e2.birthdate > e1.birthdate) )
```

Intuitive??

```
SELECT *
FROM (SELECT emp_seq, hiredate, birthdate,
              RANK() OVER ( ORDER BY hiredate DESC, birthdate DESC) rnk
            FROM employees)
WHERE rnk <= 10
```

Performance: Standard SQL took over 30 minutes. RANK version took fraction of second.

Copyright 2001 - SeQuel Consulting

Ranking Subtotals

Average salary by department, all departments, job and all jobs .

```
SELECT DECODE(GROUPING(dname), 1, 'All Departments', dname) AS dname,
             DECODE(GROUPING(job), 1, 'All Jobs', job) AS job,
             COUNT(*) "Total Empl",    AVG(sal) * 12 "Average Sal",
      RANK() OVER (PARTITION BY GROUPING(dname), GROUPING(job)
             ORDER BY AVG(sal) DESC) AS rnk
FROM emp, dept
WHERE dept.deptno = emp.deptno
GROUP BY CUBE (dname, job)
HAVING GROUPING(dname) = 1 OR GROUPING(job) = 1
```

DNAME	JOB	Total Empl	Average Sal	RNK
ACCOUNTING	All Jobs	3	35000	1
RESEARCH	All Jobs	5	26100	2
SALES	All Jobs	6	18800	3
All Departments	PRESIDENT	1	60000	1
All Departments	ANALYST	2	36000	2
All Departments	MANAGER	3	33100	3
All Departments	SALESMAN	4	16800	4
All Departments	CLERK	4	12450	5
All Departments	All Jobs	14	24878.5714	1

Copyright 2001 - SeQuel Consulting

Windowing Functions

Partition can be broken into subset via windowing clause.

Windowing Clause

ROWS | RANGE {{UNBOUNDED PRECEDING | <value expression4> PRECEDING}
| BETWEEN {UNBOUNDED PRECEDING | <value expression4> PRECEDING}
AND{CURRENT ROW | <value expression4> FOLLOWING}}

Physical vs Logical Windows

- ROWS - physical window
- RANGE – logical window

Window is relative to current row being processed.

Copyright 2001 - SeQuel Consulting

Logical Window

Sum of salaries for employees with a lower or equal salary.

```
SELECT empno, sal,
            SUM(sal) OVER (ORDER BY sal
              RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
                         AS sum_sal
FROM emp
```

EMPNO	SAL	SUM_SAL
7369	800	800
7900	950	1750
7876	1100	2850
7521	1250	5350
7654	1250	5350
7934	1300	6650
7844	1500	8150
7499	1600	9750
7782	2450	12200
7698	2850	15050
7566	2975	18025

Key to understanding logical windows!

CURRENT ROW = all rows with same ORDER BY values.

Date Intervals

Moving average for 30 days is returned in SQL 15, along with the average for the next 30 days from the current date.

```
SELECT  quote_date, close,
  AVG(close) OVER (ORDER BY quote_date
           RANGE INTERVAL '30' DAY PRECEDING) AS prv_30,
  AVG(close) OVER (ORDER BY quote_date
                      RANGE BETWEEN CURRENT ROW
                                 AND INTERVAL '30' DAY FOLLOWING) AS fol_30
FROM stock_quotes
```

No BETWEEN so this is starting point. Default end point is curent row.

REMEMBER: To compare the output of analytical functions, embed query in FROM clause.

Interval Syntax

‘n’ DAYS|MONTHS|YEARS PRECEDING|FOLLOWING

Date Intervals

Functions supplied to convert numeric values/columns to

NUMTODSINTERVAL (n, ‘DAY|HOUR|MINUTE|SECOND’)

NUMTOYMINTERVAL (n, ‘YEAR|MONTH’)

Using the STOCK_QUOTES table, you can specify a logical window as:

RANGE NUMTODSINTERVAL (open, 'DAY') PRECEDING

Copyright 2001 - SeQuel Consulting

Unwritten Documentation

```
SELECT emp_seq, effective_date, sal,
        MAX(sal) OVER (ORDER BY effective_date DESC
                RANGE BETWEEN 1 PRECEDING AND CURRENT ROW)
                AS Max_Sal
FROM sal_history
```

What does ‘1 PRECEDING’ mean in a logical window?

EMP_SEQ	EFFECTIVE_DATE	SAL	MAX_SAL
1015	11-JAN-01	500	500
1001	06-JAN-01	300	300
1003	06-JAN-01	200	300
1015	06-JAN-01	300	300
1001	01-JAN-01	200	200
1003	01-JAN-01	100	200
1002	01-JAN-01	150	200
1015	01-JAN-01	200	200
1001	22-DEC-00	100	1000
1007	22-DEC-00	400	1000
1009	22-DEC-00	1000	1000

Note that difference in days.

Shouldn’t MAX be 500?

Unwritten Documentation

```
SELECT emp_seq, effective_date, sal,
            MAX(sal) OVER (ORDER BY effective_date DESC
                        RANGE BETWEEN 5 PRECEDING AND CURRENT ROW)
                        AS Max_Sal
FROM sal_history
```

Recall difference in dates was 5 days?

EMP_SEQ	EFFECTIVE_DATE	SAL	MAX_SAL
1015	11-JAN-01	500	500
1001	06-JAN-01	300	500
1003	06-JAN-01	200	500
1015	06-JAN-01	300	500
1001	01-JAN-01	200	300
1003	01-JAN-01	100	300
1002	01-JAN-01	150	300
1015	01-JAN-01	200	300
1001	22-DEC-00	100	1000
1007	22-DEC-00	400	1000
1009	22-DEC-00	1000	1000

When ORDER BY on date column, and logical window used, ‘n’ PRECEDING means ‘n’ DAYS PRECEDING.

Unwritten Documentation

Logical window

ORDER BY numeric column

```
SELECT emp_seq, effective_date, sal,
          MAX(sal) OVER (ORDER BY sal DESC
            RANGE BETWEEN 1 PRECEDING AND CURRENT ROW)
                    AS Max_Sal
FROM sal_history
```

So what does the 1 mean?

The 1 means units of SAL .

So if CURRENT contains SAL of 100, the RANGE includes rows with SAL BETWEEN 99 and 101.

Unwritten Documentation

Example increases range to illustrate proper interpretation.

```
SELECT emp_seq, effective_date, sal,
          MAX(sal) OVER (ORDER BY sal DESC
          RANGE BETWEEN 100 PRECEDING AND CURRENT ROW)
                                AS Max_Sal
FROM sal_history
```

EMP_SEQ	EFFECTIVE_DATE	SAL	MAX_SAL
1009	22-DEC-00	1000	1000
1015	11-JAN-01	500	500
1007	22-DEC-00	400	500
1001	06-JAN-01	300	400
1015	06-JAN-01	300	400
1003	06-JAN-01	200	300
1015	01-JAN-01	200	300
1001	01-JAN-01	200	300
1002	01-JAN-01	150	200
1003	01-JAN-01	100	200
1001	22-DEC-00	100	200

The range now includes other rows producing different MAX values

Physical Windows

Simple

- **Use ROWS instead of RANGE.**
- **Specify exact number of rows preceding and following.**

```
SELECT empno, job,
            MAX(sal) OVER (ORDER BY job ROWS 1 PRECEDING) AS max_job
FROM emp
```

- Rows sorted by JOB
- Window includes current row and 1 row prior in the sort order
- ‘1 PRECEDING’ is **start point**
- **End point** defaults to current row

FIRST_VALUE - LAST_VALUE
vs.
LEAD - LAG

FIRST_VALUE (col/express) – returns first value of "col/express" from **window**

LAST_VALUE (col/express) – returns last value of "col/express" from **window**

LEAD (col/express, [offset, [default]]) – returns value of col/express from row after current row offset by "offset" (default=1) from **partition**

LAG (col/express, [offset, [default]]) – returns value of col/express from row before current row offset by "offset" (default=1) from **partition**

LEAD and LAG do not need window clause. Offset and function name determines which row to access

Copyright 2001 - SeQuel Consulting

LAST_VALUE

Retrieve history of raises

```
SELECT emp_seq, sal, effective_date, sal - LAST_VALUE(sal) OVER
      (PARTITION BY emp_seq ORDER BY effective_date DESC
         ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING) AS raise,
    MIN(effective_date) OVER (PARTITION BY emp_seq ORDER BY effective_date) AS first_sal
FROM sal_history
```

EMP_SEQ	SAL	EFFECTIVE_DATE	RAISE	
1001	300	06-JAN-01	100	22-DEC
1001	200	01-JAN-01	100	22-DEC-00
1001	100	**22-DEC-00**	0	**22-DEC-00**
1002	150	**01-JAN-01**	0	**01-JAN-01**
1003	200	06-JAN-01	100	01-JAN-01
1003	100	**01-JAN-01**	0	**01-JAN-01**
1007	400	**22-DEC-00**	0	**22-DEC-00**
1009	1000	**22-DEC-00**	0	**22-DEC-00**
1015	500	11-JAN-01	200	01-JAN-01
1015	300	06-JAN-01	100	01-JAN-01
1015	200	**01-JAN-01**	0	**01-JAN-01**

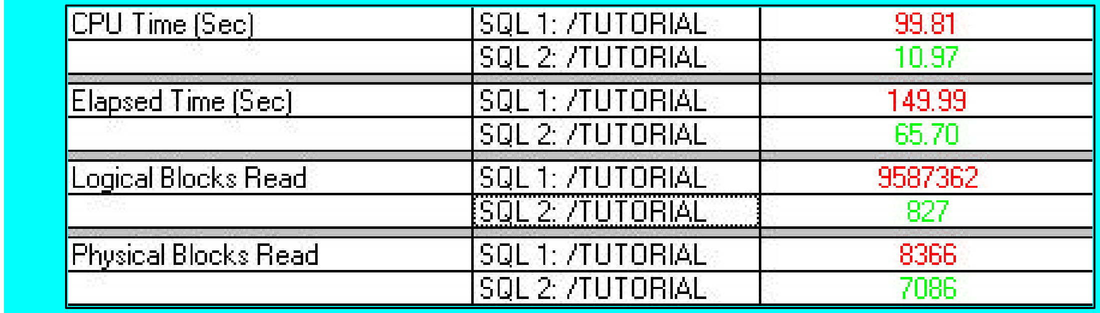
MIN used to list the first SAL_HISTORY row per employee, so that we can filter out misleading zero raises. Embed query in FROM clause and add criterion “**effective_date != first_sal**”

Continued>

Performance Comparison

Listing raise history with standard SQL.

```
SELECT s2.effective_date, s2.sal, s2.sal – s1.sal AS raise
FROM sal_history s1, sal_history s2
WHERE s1.emp_seq = s2.emp_seq
AND s1.effective_date = (SELECT MAX(effective_date) FROM sal_history
                                          WHERE emp_seq = s2.emp_seq
                                          AND effective_date < s2.effective_date)
```



CPU Time (Sec)	SQL 1: /TUTORIAL	99.81
	SQL 2: /TUTORIAL	10.97
Elapsed Time (Sec)	SQL 1: /TUTORIAL	149.99
	SQL 2: /TUTORIAL	65.70
Logical Blocks Read	SQL 1: /TUTORIAL	9587362
	SQL 2: /TUTORIAL	827
Physical Blocks Read	SQL 1: /TUTORIAL	8366
	SQL 2: /TUTORIAL	7086

SQL 1:/TUTORIAL is the standard SQL;

SQL 2:/TUTORIAL uses analytical function.

Default Window

RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

Logical Window

Sum over entire result set

```
SELECT deptno, ename, sal, SUM(sal) OVER () AS tot_sal
FROM emp
```

```
SELECT deptno, ename, sal, SUM(sal) OVER ( ORDER BY sal) AS tot_sal
FROM emp
```

ORDER BY w/o window clause means default window

RATIO_TO_REPORT

- Computes the percentage of the column/expression to the total of column/expression for all rows in the partition.

- ORDER BY is not permitted, which in turns means a window clause is not permitted.

```
SELECT emp_seq, proj_seq,
  SUM(hours) AS sum_hrs,
  RATIO_TO_REPORT(SUM(hours))
                  OVER ( PARTITION BY emp_seq ) AS ratio
FROM time_sheets GROUP BY emp_seq, proj_seq
```

EMP_SEQ	PROJ_SEQ	SUM_HRS	RATIO
2903	10	12	.6
2903	11	8	.4
2907	11	12	1
2921	10	9	.310344828
2921	11	12	.413793103
2921	13	8	.275862069
2934	10	8	1
2941	11	8	1

CASE Function

```
CASE WHEN <criteria> THEN <output> WHEN <criteria> THEN <output>
          ELSE <output> END
```

- If 1st WHEN is FALSE, 2nd WHEN is tested
- Only one ELSE
- Criteria can be any valid SQL criteria, including subquery

```
CASE WHEN sal > 3000 OR JOB = ‘PRESIDENT’ THEN 300 ELSE sal*.2 END

CASE WHEN hiredate < ’01-JAN-97’ THEN ‘Retired’ END

CASE WHEN sal > (SELECT avg(sal) FROM emp) THEN 'above average' END
```

CASE Function

List unpaid invoices by days overdue.

```
SELECT CASE WHEN sysdate-inv_date > 90 THEN '90 days overdue'
                     WHEN sysdate-inv_date > 60 THEN '60 days overdue'
                     WHEN sysdate-inv_date > 30 THEN '30 days overdue‘
                     WHEN sysdate-inv_date >  0 THEN 'less than 30 days overdue' END AS period,
SUM(amount) AS amount
FROM invoices
WHERE paid_date IS NULL
GROUP BY CASE WHEN sysdate-inv_date > 90 THEN '90 days overdue‘
          WHEN sysdate-inv_date > 60 THEN '60 days overdue‘
          WHEN sysdate-inv_date > 30 THEN '30 days overdue‘
          WHEN sysdate-inv_date >  0 THEN 'less than 30 days overdue' END
```

PERIOD	AMOUNT
30 days overdue	4301
60 days overdue	6255
90 days overdue	1012
less than 30 days overdue	10302

CASE Vs. DECODE

Previous query is implemented with DECODE.

```
SELECT DECODE (SIGN(sysdate-inv_date – 90), -1,
                    DECODE(SIGN(sysdate-inv_date-60),-1,
           DECODE(SIGN(sysdate-inv_date-30), -1, ‘less than 30 days overdue’,
          ’30 days overdue’),’60 days overdue’),’90 days overdue’) AS period,
          SUM(amount) AS amount
FROM invoices
GROUP BY DECODE (SIGN(sysdate-inv_date – 90), -1,
          DECODE(SIGN(sysdate-inv_date-60),-1,
          DECODE(SIGN(sysdate-inv_date-30), -1, ‘less than 30 days overdue’,
                    ’30 days overdue’),’60 days overdue’),’90 days overdue’)
```

Complex to specify, and difficult to read

CUME_DIST

CUME_DIST(x) = number of values (different from, or equal to, x) in set coming before x in the specified order/ N

- **Determines the number of values in a sorted list that came before or are equal to the current value.**
- **ORDER BY is mandatory, since a sorted list is required**

```
SELECT student_id, score,
     CUME_DIST() OVER
            (ORDER BY score)
FROM scores
```

STUDENT_ID	SCORE	CUME_DIST
1	45	.083333333
4	50	.166666667
7	58	.25
3	63	.333333333
12	69	.416666667
6	72	.5
9	76	.583333333
2	85	.75
8	85	.75
10	87	.833333333
11	92	.916666667
5	98	1