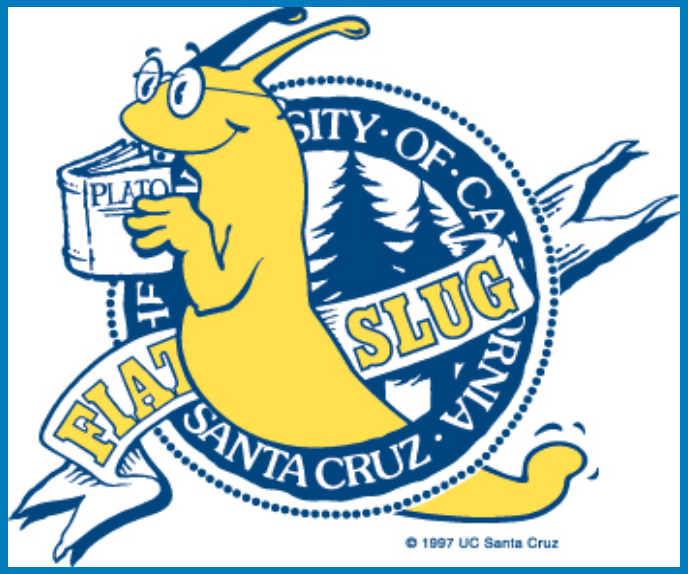
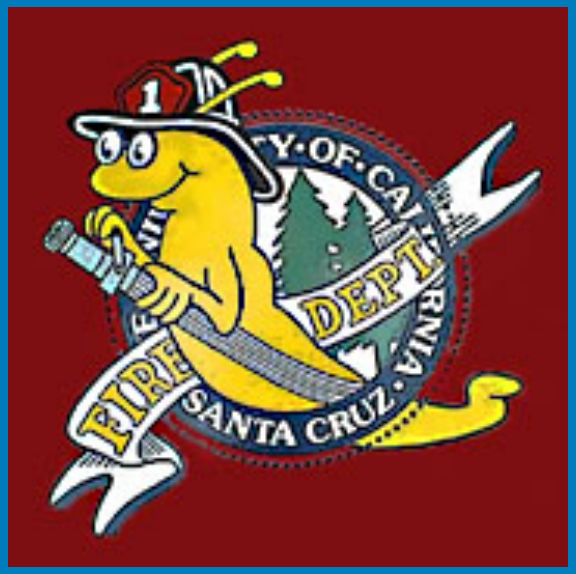
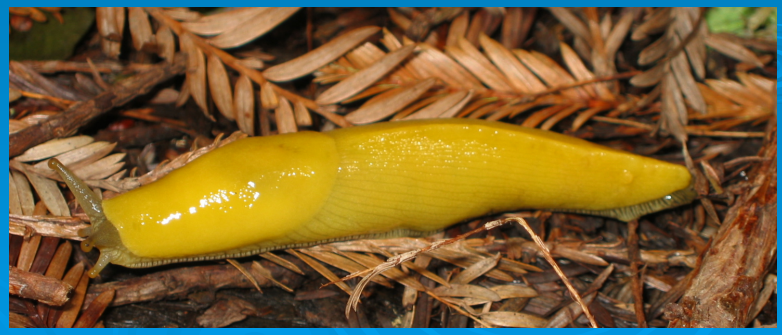
Don't Be In a Funk: Use Analytic Functions





Philip Rice
University of California Santa Cruz



Overview

- My involvement came from a performance problem
- Overlap with some of the “traditional” aggregate functions: e.g. max, avg, count
  -- same keyword, similar syntax
- Goal today: raise awareness of possibilities, know when to consider as an option;
  not a comprehensive view of all functions

Overview

- Functions are used in SELECT statement: likely to be most helpful in reporting situations
- Better functionality: traditional approach can be much more difficult or nearly impossible
- Performance improvement is likely to be more obvious with larger datasets, difference can be hours down to minutes(!)

General Syntax

- Function(arg1, …, argn) OVER ([PARTITION BY <…>] [ORDER BY <…>] [window_clause])
- "OVER" is indicator of analytic function
- PARTITION BY is comparable to GROUP BY
- window_clause is not as commonly used, but can be helpful, e.g. looking at different time periods on same row of output (examples later)
- window_clause (partial) syntax is [ROW or RANGE] BETWEEN <start> AND <end>

Example: "traditional" count

```
select count(*), OBJECT_TYPE
from all_objects
where owner = 'OUTLN'
group by OBJECT_TYPE;

  COUNT(*) OBJECT_TYPE
  -------- -----------
         4 INDEX
         1 PROCEDURE
         3 TABLE
```

- Non-aggregated columns must be in GROUP BY
- What if we want to show detail at same time as the aggregate?

If "OVER" is empty, acts on whole set

```
select object_name,object_type,
  count(*) OVER () tot_count,
  count(*) OVER (PARTITION BY object_type) type_count
from all_objects where owner = 'OUTLN';
```

OBJECT_NAME	OBJECT_TYPE	TOT_COUNT	TYPE_COUNT
OL$NAME	INDEX	8	4
OL$HNT_NUM	INDEX	8	4
OL$SIGNATURE	INDEX	8	4
OL$NODE_OL_NAME	INDEX	8	4
ORA$GRANT_SYS_SELECT	PROCEDURE	8	1
OL$NODES	TABLE	8	3
OL$HINTS	TABLE	8	3
OL$	TABLE	8	3

Total for ALL rows, on each detail line

Total for each object type

Timing of execution in SQL

- Analytic functions are computed after:
  - All joins
  - WHERE clause
  - GROUP BY
  - HAVING
- Main ORDER BY of query is ***after*** analytic function
- So AFs can only appear in select list and in main ORDER BY clause of query

Timing of execution in SQL

- Stages:

Joins, WHERE, GROUP BY, HAVING

Apply analytic functions to result set rows (aka “partition”)

Apply ORDER BY clause (from main query)

Prep for ROW_NUMBER and RANK

- Select object_type "ObjTyp" to_char(last_ddl_time,
  'yyyymmdd hh24miss') last_ddl_time
  from all_objects
  where owner = 'OUTLN'
  and object_type IN ('TABLE','INDEX')
  group by owner, object_type,
  to_char(last_ddl_time,'yyyymmdd hh24miss')
  order by object_type, last_ddl_time;

```
ObjTyp LAST_DDL_TIME
------ ---------------
INDEX  20031001 173156
INDEX  20080906 102159
TABLE  20080906 102610
```

- 2 of 3 have same `LAST_DDL_TIME` at detail level, we'll use as demo for RANK

ROW_NUMBER and RANK

```
select object_type "ObjTyp", substr(object_name,1,10) "ObjName",
     to_char(last_ddl_time,'yyyymmdd hh24miss') last_ddl_time,
     row_number() over
      (partition by object_type order by last_ddl_time) RN,
     rank() over
      (partition by object_type order by last_ddl_time) R,
     dense_rank() over
      (partition by object_type order by last_ddl_time) DR
   from all_objects
   where owner = 'OUTLN' and object_type  IN ('TABLE','INDEX');
```

Note that all three functions have same PARTITION BY and ORDER BY clauses.

Results on next slide …

Row_number, Rank, Dense Rank

- Remember: Last three columns all have:
  `(partition by object_type order by last_ddl_time)`
  `[…]`

ObjTyp	ObjName	LAST_DDL_TIME	RN	R	DR
INDEX	OL$NAME	20031001 173156	1	1	1
INDEX	OL$HNT_NUM	20031001 173156	2	1	1
INDEX	OL$SIGNATU	20031001 173156	3	1	1
INDEX	OL$NODE_OL	20080906 102159	4	**4**	**2**
TABLE	OL$NODES	20080906 102610	1	1	1
TABLE	OL$	20080906 102610	2	1	1
TABLE	OL$HINTS	20080906 102610	3	1	1

Behavior: Row_number, Rank, Dense Rank

- If two records have the same value in the ORDER BY, the two records get different ROW_NUMBER.
  RANK and DENSE_RANK do not work like that.
- If two records have the same value in the ORDER BY, they both get the same RANK or DENSE_RANK.
  The difference between RANK and DENSE_RANK is how they are counted.
  DENSE_RANK uses sequential numbers, RANK does not.
  In the INDEX set, the fourth is different from the first three.
  RANK jumps to display "4", and DENSE RANK is "2".

ROW_NUMBER is similar to ROWNUM

- One key difference:
  ROWNUM gets incremented as rows are returned from the query,
  so we can not say "WHERE ROWNUM = 5".
  But ROW_NUMBER ***can*** be used that way.

Sort both ways in same SQL

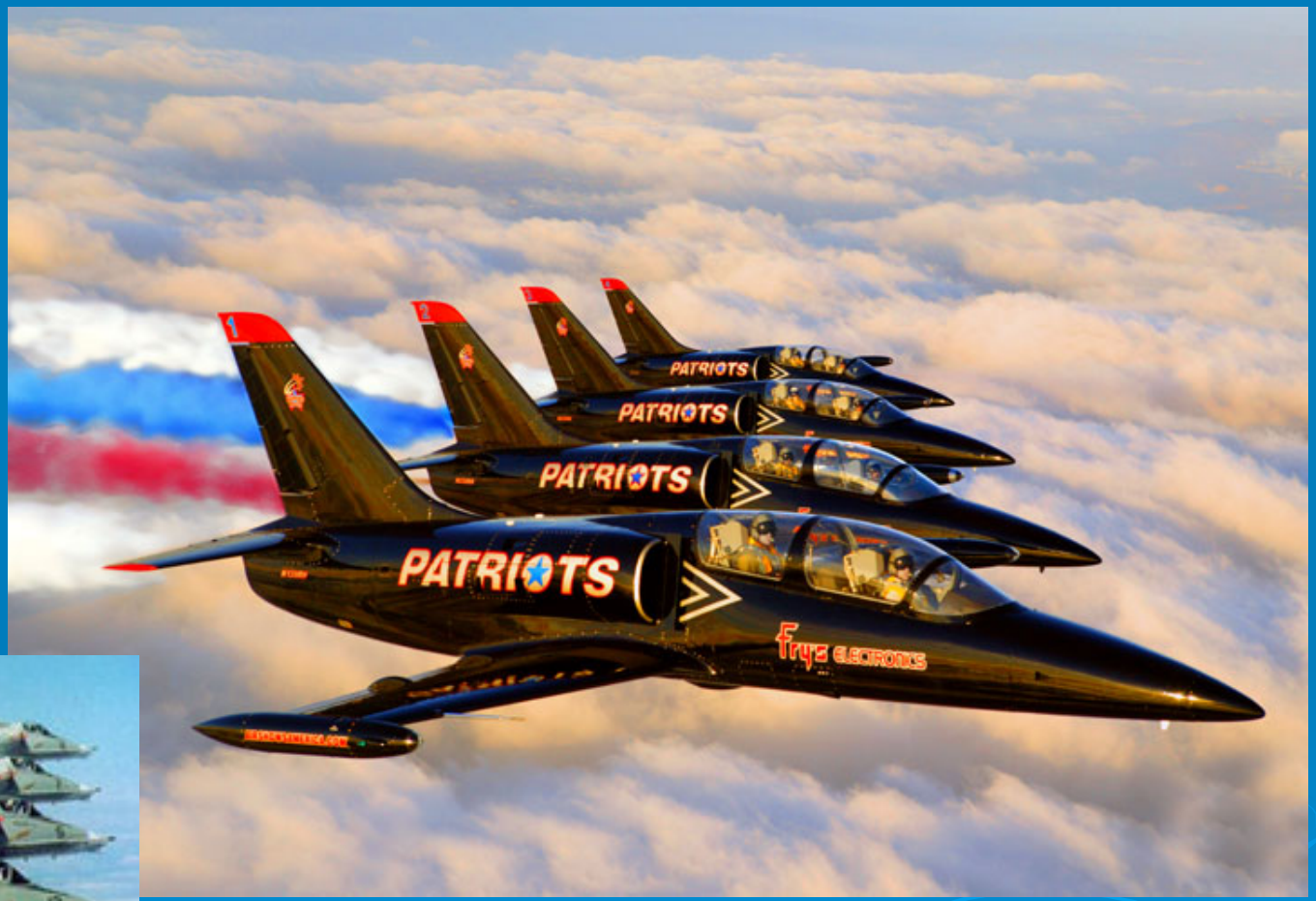
```
Select […],
  row_number() OVER ( partition by object_type
  order by last_ddl_time) SORTUP,
  row_number() OVER ( partition by object_type
  order by last_ddl_time DESC NULLS LAST) SORTDOWN
from all_objects where owner = 'OUTLN'
 and object_type IN ('TABLE','INDEX');
```

ObjTyp	LAST_DDL_TIME	SORTUP	SORTDOWN
INDEX	20031001 173156	1	4
INDEX	20031001 173156	2	3
INDEX	20031001 173156	3	2
INDEX	20080906 102159	4	1
TABLE	20080906 102610	1	3
TABLE	20080906 102610	2	2
TABLE	20080906 102610	3	1

Traditional -- slow way to see mixed detail and summary levels (generated from reporting tool)

```
SELECT <detail columns>, max_effdt, max_effseq
FROM
 ( SELECT <detail columns>,
     MAX (DISTINCT t2.APLAN_EFFDT) max_effdt
   FROM t3, t1 LEFT OUTER JOIN t2 ON […]
   WHERE […] GROUP BY t1.CTERM_EMPLID, t1.CTERM_TERM_CD) d5,
 ( SELECT <detail columns>,
     MAX (t2.APLAN_EFFSEQ) max_effseq
   FROM t3, t1 LEFT OUTER JOIN t2 ON […]
   WHERE […] GROUP BY t1.CTERM_EMPLID, t1.CTERM_TERM_CD,
                      t2.APLAN_EFFDT) d4,
 ( SELECT <detail columns only, no aggregate!!!>
   FROM t3, t1 LEFT OUTER JOIN t2 ON […]
   WHERE […]    < NO group by clause!!!> ) d3
WHERE < predicates for outer select >
ORDER BY < columns for outer select >
```

Analytic Function is FASTER!!

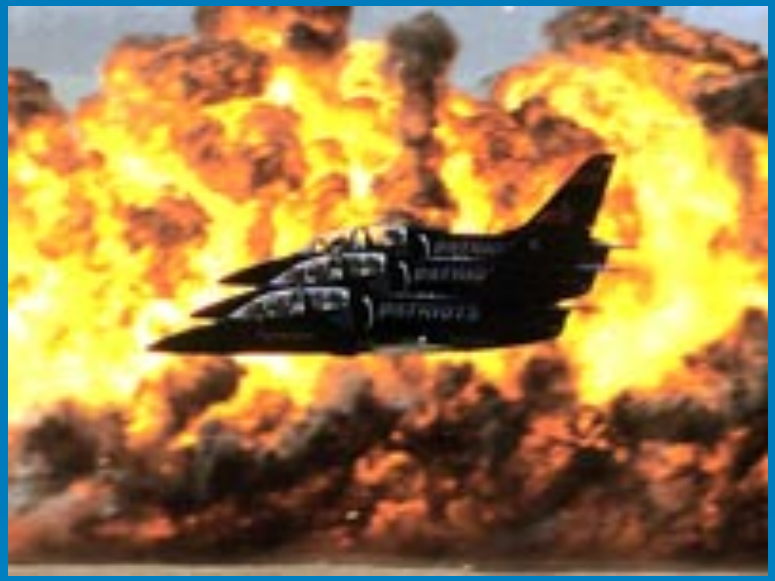


Requires only one inline view, a single pass instead of three…

Improvement is Hours to Minutes

```
SELECT <detail columns>, max_effdt, max_effseq
FROM
 ( SELECT <detail columns>,
   max(t2.APLAN_EFFDT) OVER
     (PARTITION BY t1.cterm_emplid, t1.cterm_term_cd)
      AS  max_effective_date,
   max(t2.APLAN_EFFSEQ) OVER
     (PARTITION BY t1.cterm_emplid, t1.cterm_term_cd,
                   t2.APLAN_EFFDT)
      AS  max_effective_sequence
      FROM t3, t5 (t1 LEFT OUTER JOIN t2 ON […] )
       LEFT OUTER JOIN t4 ON […] WHERE […] )
WHERE < predicates for outer select >
ORDER BY < columns for outer select >
```

Keep adapting, don't be a dinosaur!



Example of UPDATE

```
UPDATE FZBRFCX SET FZBRFCX_ZERO_FLAG = 2
WHERE rowid IN
  (SELECT rowid FROM
    (SELECT rowid, FZBRFCX_ZERO_FLAG Flag,
       sum(FZBRFCX_TRANS_AMT) OVER
         (PARTITION BY FZBRFCX_ACCT_CODE,
                       FZBRFCX_FUND_CODE,
                       FZBRFCX_DOC_REF_NUM) Sum_Amt
     FROM FZBRFCX )
    WHERE Sum_Amt = 0 AND Flag <> 0 ) ;
```

Helpful sidetrack: Query Subfactoring AKA Common Table Expression

- Analytic functions often need an inline view (subquery).
- Sometimes the inline views are nested
- Indentation is helpful, but can be confusing
- Subfactoring used here for clarity with related SQL statements across multiple slides
- Subfactoring easily allows multiple use of alias [11.2 allows recursive too]

Traditional inline: layers with indentation

```
SELECT MID_LVL.po_code, MID_LVL.seq, […]
FROM
 (select INNER_LVL.po_code, INNER_LVL.seq, […]
  from
   (select po_code, seq, […]
    from fprpoda b
    where po_code in
      (select b.po_code
       from fprpoda b
       where activity_date
         between '01-NOV-09' and '09-NOV-09') CODE_LIST
   ) INNER_LVL
 ) MID_LVL
WHERE < [MID_LVL.column] predicates...>
```

Query Subfactoring: Top Down

```
WITH
 CODE_LIST AS
  (select po_code
   from fprpoda
   where activity_date
         between '01-NOV-09' and '09-NOV-09' ),
 INNER_LVL AS
  (select po_code, seq, […]
   from fprpoda
   where po_code in CODE_LIST ),
 MID_LVL AS
  (select po_code, seq, […]
   from INNER_LVL )
SELECT * FROM MID_LVL
WHERE < predicates...>
```

Running Totals and Windowing

- Requirement:
  Show values from current and previous rows, where running total went above $50,000 level, where more stringent approvals are required:
  Is anyone trying to get around audit rules?
- Originally looked like it would need PL/SQL, with cursors starting and stopping
- We'll use query subfactoring to see the pieces build on each other…

Running Totals and Windowing: stmt1

```
WITH
code_list AS ( -- [codes used in next stmt]
SELECT distinct po_code
FROM  fprpoda
WHERE trunc(activity_date)
 BETWEEN '01-NOV-09' AND '09-NOV-09'
  AND seq is not null ),
```

Running Totals and Windowing: stmt2

```
INNER_LVL AS ( -- [sum each code and seq combo]
SELECT po_code,
   LAG(po_code, 1) OVER
        (ORDER BY po_code) "PrevCode",
  seq , amt "CurrAmt", activity_date,
  SUM(amt) OVER (PARTITION BY po_code
      ORDER BY po_code, seq, activity_date)
                running_tot
 FROM fprpoda
 WHERE po_code IN
( select po_code from CODE_LIST )  ),
```

Running Totals and Windowing: stmt3

```
MID_LVL AS ( -- get curr/prev row values
SELECT    po_code, seq ,
 (CASE WHEN "PrevCode" != po_code THEN NULL
  ELSE  LAG(running_tot, 1) OVER
    (ORDER BY po_code, seq) END) "PrevRunTot",
 running_tot "RunningTot",
 activity_date curr_actv,
(CASE WHEN "PrevCode" != po_code THEN NULL
 ELSE  LAG(activity_date) OVER
   (ORDER BY po_code, seq) END)  prev_actv
FROM INNER_LVL)
-- 1 is default for LAG, hard coding would be
--  for clarity
```

Running Totals and Windowing: Final

- Query subfactoring above is done: one isolated stmt shows what we're ultimately trying to do…

```
SELECT po_code, seq, "PrevRunTot",
  "RunningTot"-"PrevRunTot" "DiffChange",
      "RunningTot" , prev_actv, curr_actv
FROM MID_LVL
WHERE "PrevRunTot" <  50000
   AND "RunningTot" >= 50000;
```

PO_CODE	SEQ	PrevRunTot	DiffChange	RunningTot	PREV_ACTV	CURR_ACTV
B0142584	7	46,800.00	5,500.00	52,300.00	05-FEB-09	05-NOV-09
B0181676	1	38,142.00	23,856.34	61,998.34	26-NOV-07	17-NOV-08
S0176940	1	43,371.00	42,156.00	85,527.00	17-JUN-05	23-MAR-06
S0181330	1	1.00	302,069.91	302,070.91	20-JUL-07	28-AUG-07

Detail for one PO, Seq# 0, 2, and 7: exact same date/time, so running total is not gradually increasing

PO_CODE	SEQ	CurrAmt	RunningTot	Activity_Date_Time
B0142584	0	1.00	5,001.00	22-JAN-2003 10:27:00
B0142584	0	5,000.00	5,001.00	22-JAN-2003 10:27:00
B0142584	1	6,500.00	11,501.00	09-OCT-2003 14:36:01
B0142584	2	-1.00	18,500.00	27-OCT-2004 15:51:01
B0142584	2	7,000.00	18,500.00	27-OCT-2004 15:51:01
B0142584	3	9,500.00	28,000.00	05-OCT-2006 13:27:01
B0142584	4	4,000.00	32,000.00	25-OCT-2007 09:45:02
B0142584	5	5,500.00	37,500.00	27-NOV-2007 10:12:03
B0142584	6	9,300.00	46,800.00	05-FEB-2009 11:12:01
B0142584	7	-7,000.00	52,300.00	05-NOV-2009 12:27:01
B0142584	7	7,000.00	52,300.00	05-NOV-2009 12:27:01
B0142584	7	-9,300.00	52,300.00	05-NOV-2009 12:27:01
B0142584	7	9,500.00	52,300.00	05-NOV-2009 12:27:01
B0142584	7	-4,000.00	52,300.00	05-NOV-2009 12:27:01
B0142584	7	-11,500.00	52,300.00	05-NOV-2009 12:27:01
B0142584	7	9,500.00	52,300.00	05-NOV-2009 12:27:01
B0142584	7	-9,500.00	52,300.00	05-NOV-2009 12:27:01
B0142584	7	11,500.00	52,300.00	05-NOV-2009 12:27:01
B0142584	7	9,300.00	52,300.00	05-NOV-2009 12:27:01

Running Totals and Windowing: Notes

- ”LAG” puts curr/prev values on same row, that allows easy WHERE clause to find threshold
- We could not put “LAG” in stmt with running total (needed extra layering), because curr/prev row was not available until running total was done
- We got new running total for each code, because that is in “PARTITION BY” clause

11.2 feature: LISTAGG

- Can be Simple Aggregate OR Analytic
- Concatenates values from rows into a string, i.e. a LIST AGGregation
- Example is continuation of Running Total, which has duplicate dates for some Sequences, and LISTAGG faithfully shows all dups
- Including “distinct” in SQL looks *across* column values, not *within* LISTAGG: can not eliminate dups
- Simple Aggregate example shows dups…

11.2 LISTAGG: Simple Aggregate

```
SELECT seq,
  LISTAGG(to_char(activity_date,'MON-YYYY'), '; ')
  WITHIN GROUP (ORDER BY seq) "Activity_Dates"
FROM fprpoda WHERE po_code = 'B0142584' AND seq < 3
GROUP BY seq;
 SEQ Activity_Dates
----- --------------------
    0 JAN-2003; JAN-2003
    1 OCT-2003
    2 OCT-2004; OCT-2004
```

11.2 LISTAGG: Analytic

- Need "distinct" for analytic, else shows all 19 rows; No GROUP BY, analytic function is at detail level

```
SELECT distinct seq,
  SUM(amt) OVER (ORDER BY seq, activity_date) "RunTot",
  SUM(amt) OVER (PARTITION BY seq ORDER BY
      seq, activity_date) "SeqTot",
  LISTAGG(amt, ';') WITHIN GROUP (ORDER BY seq)
  OVER (PARTITION BY seq) "Amts"
FROM fprpoda WHERE po_code = 'B0142584'
 AND seq IS NOT NULL ORDER BY seq;
```

- Sequence 7 has 10 Amount entries, some cancel each other out…

11.2 LISTAGG: Analytic

- Reminder of syntax, and result from SQL stmt:

```
LISTAGG(amt, ';') WITHIN GROUP (ORDER BY seq)
  OVER (PARTITION BY seq) "Amts"
 Seq RunTot SeqTot Amts
---- ------ ------ ----------------------------
   0   5001   5001 5000;1
   1  11501   6500 6500
   2  18500   6999 7000;-1
   3  28000   9500 9500
   4  32000   4000 4000
   5  37500   5500 5500
   6  46800   9300 9300
   7  52300   5500 9300;-9500;9500;-11500;-4000;
                   9500;-9300;7000;-7000;11500
```

Moving Average

```
WITH SGLCODE AS (
select seq, amt
from fprpoda
where po_code in ('BA177629')
  and seq IS NOT NULL )
SELECT   seq, amt,
  avg(amt) OVER (order by seq rows
   between 1 preceding and 1 following ) ma1,
  avg(amt) OVER (order by seq rows
   between 0 preceding and 1 following ) ma2,
  avg(amt) OVER (order by seq rows
   between 1 preceding and 0 following ) ma3
FROM SGLCODE order by seq;
```

Moving Average: Result

SEQ	AMT	MA1	MA2	MA3
0	.01	1068.86	1068.86	.01
0	2137.70	2379.24	3568.85	1068.86
1	5000.00	2596.23	2825.50	3568.85
2	651.00	18500.00	25250.00	2825.50
3	49849.00	25250.00	49849.00	25250.00

NOTE:

MA1: 1 before, 1 after (3 rows avg)
MA2: 0 before, 1 after (2 rows avg)
MA3: 1 before, 0 after (2 rows avg)

(Finally) NTILE

- Example on next slide is 6 rows of test scores with 4 buckets (Quartile)
- NTILE definition is ordered DESCENDING so that highest test scores are in buckets 1 and 2
- If “DESC” were taken out of SQL, ranking would be reversed, i.e. lowest scores would be in the 1st quartile rather than 4th
- Two extra values (6/4) are allocated to buckets 1 and 2

NTILE: Example with 4 buckets

```
SELECT name, score,
 NTILE(4) OVER (ORDER BY score DESC)
   AS quartile
FROM test_scores ORDER BY name;
NAME                    SCORE   QUARTILE
------------------ ------ ----------
Barry Bottomly          12          4
Felicity Fabulous       99          1
Felix Fair              41          2
Mildred Middlin         55          2
Paul Poor               24          3
Sharon Swell            86          1
```

A & Q

- **A & Q** **A**nswers: Wisdom to share? **Q**uestions?
- Philip Rice price [at] ucsc {dot} edu