
OLAP Queries

6 June 2010

Lecture 12

Topics for Today

- Multidimensional Aggregation Queries
 - ROLLUP
 - CUBE
- Window Queries
 - PARTITION BY Framing a Window
- New Aggregate Functions
- Finding Answers Quickly
- Implementation Techniques for OLAP
 - Bitmap Indexes

Source: RG 25.3-25.6.1

Pivot Tables

Another common tool is **pivoting**: Creating summary tables based on one or two dimensions

In a pivot table, each value in the dimension is a row/column and its values in the fact table are detailed and then summed

- The result is a [cross-tabulation](#)
-

Example: A pivot of the sales fact table by the location and time dimensions

- The location dimension is rolled up to the state attribute
- The time dimension is rolled up to the month attribute

	WI	CA	Total
Jan	63	81	144
April	38	107	145
July	75	35	110
Total	176	223	399

Pivot Table Example

In order to get the table shown, we need to run a bunch of queries:

```
SELECT T.month, L.state, SUM (S.sales)
FROM Sales S, Times T, Locations L
WHERE S.timeid = T.timeid AND S.locid = L.locid
GROUP BY T.year, L.state
```

```
SELECT T.month, SUM(S.sales)
FROM Sales S, Times T
WHERE T.timeid = S.timeid
GROUP BY T.month
```

```
SELECT L.state, SUM(S.sales)
FROM Locations L, Sales S
WHERE L.locid = S.locid
GROUP BY L.state
```

```
SELECT SUM(S.sales)
FROM Sales S, Locations L
WHERE S.locid = L.locid
```

	WI	CA	Total
Jan	63	81	144
April	38	107	145
July	75	35	110
Total	176	223	399

How many pivots?

To compute a pivot table on **two dimensions** and we need 4 queries

- We could choose any two dimensions for the pivot table, so with k dimensions there are 2^k possible SQL queries to write
- We can't pre-compute all of them, but ones that are **commonly used we can and should**
 - This goes back to what we mentioned before about design

For ones we **don't pre-compute**, we can speed them up:

- Since the queries are **related**, the DBMS can run them all together to save time in the calculations
- This also saves the user the time of typing all of them

To do pivots easier, SQL:1999 introduced **two new operators** which modify the GROUP BY commands:

- **CUBE**
- **ROLLUP**

CUBE Command Example

The CUBE operation is identical to running a GROUP BY separately on all subsets of the dimensions listed

Example query:

```
SELECT T.month, L.state,
SUM(S.sales)
FROM Sales S, Times T,
Locations L
WHERE S.timeid = T.timeid
AND S.locid = L.locid
GROUP BY CUBE (T.month,
L.state)
```

The query generates the table:

The *nulls* indicate where the summations for the dimension are

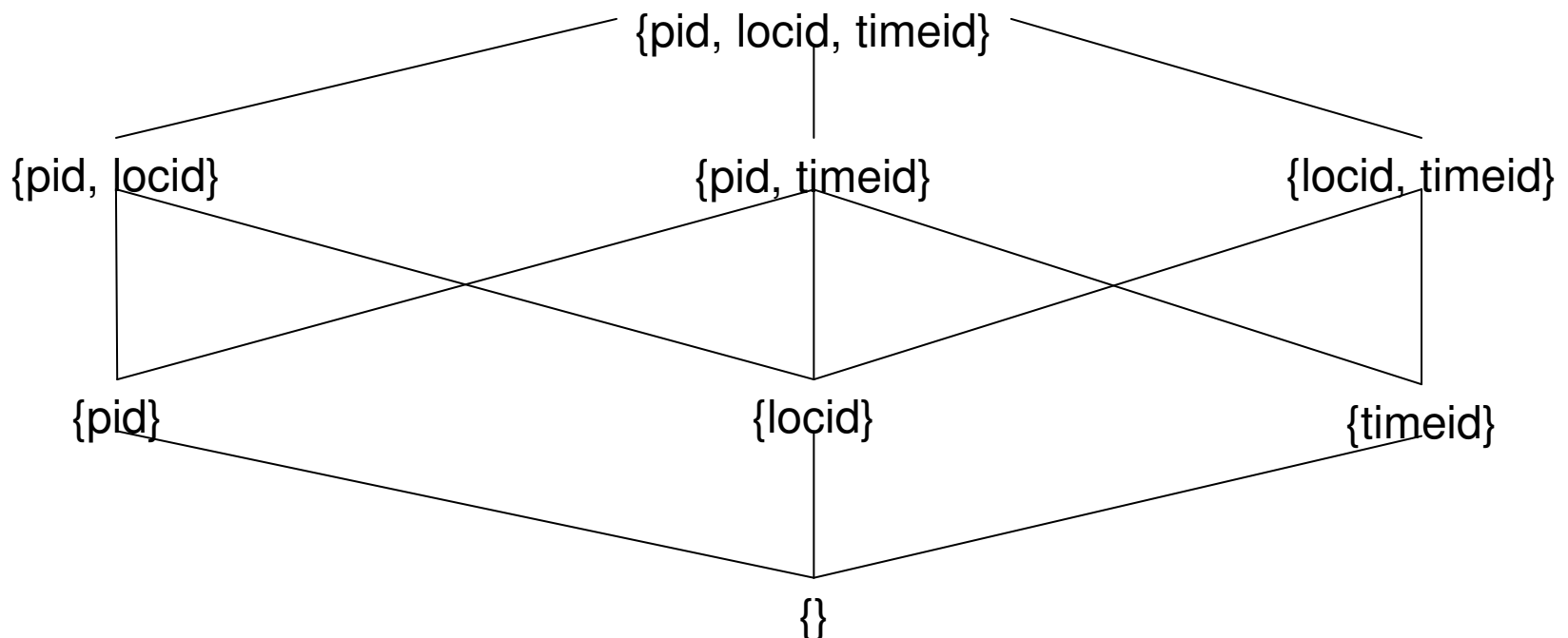
T.month	L.state	SUM(S.sales)
Jan	WI	63
Jan	CA	81
Jan	<i>null</i>	144
April	WI	38
April	CA	107
April	<i>null</i>	145
July	WI	75
July	CA	35
July	<i>null</i>	110
<i>null</i>	WI	176
<i>null</i>	CA	223
<i>null</i>	<i>null</i>	399

CUBE Command

The CUBE calculates the powerset of all dimensions

- Running CUBE on more than two dimensions gives us a lot more summations
- For k dimensions, we have about 2^k subsets

Example: `SELECT SUM(S.Sales) FROM Sales S GROUP BY CUBE (pid, locid, timeid)`



ROLLUP

ROLLUP is similar (also modifies GROUP BY), but it doesn't compute all subsets

Example:

```
SELECT T.month, L.state,
SUM(S.sales)
FROM Sales S, Times T,
Locations L
WHERE S.timeid = T.timeid
AND S.locid = L.locid
GROUP BY ROLLUP (T.month,
L.state)
```

Computes all (year, state) values and per month sums

- **Not** for each state by itself (*null*, WI, 176), (*null*, CA, 22)

T.month	L.state	SUM(S.sales)
Jan	WI	63
Jan	CA	81
Jan	<i>null</i>	144
April	WI	38
April	CA	107
April	<i>null</i>	145
July	WI	75
July	CA	35
July	<i>null</i>	110
<i>null</i>	<i>null</i>	399

So Far

- Multidimensional Aggregation Queries
 - ROLLUP
 - CUBE
- Window Queries
 - PARTITION BY Framing a Window
- New Aggregate Functions
- Finding Answers Quickly
- Implementation Techniques for OLAP
 - Bitmap Indexes

Window Queries

- Time needs better support
 - Complex groupings based on it
- The result: A new and more powerful way to deal with groups – **query window**

The Goal of Windows

How could we write the following queries in SQL-92?

1. Find total sales by month.
 - GROUP BY
2. Find total sales by month for each city.
 - GROUP BY
3. Find the percentage change in the total monthly sales for each product.
 - GROUP BY (but complex)
4. Find the top five products ranked by total sales.
 - GROUP BY (but complex)
5. Find the trailing n day moving average of sales. (For each day, we must compute the average daily sales over the preceding n days.)
 - Complex, but impossible if n is not fixed
6. Find the top five products ranked by cumulative sales, for every month over the past year.
 - GROUP BY, ORDER BY, but only if 5 is fixed
7. Rank all products by total sales over the past year, and, for each product, print the difference in total sales relative to the product ranked behind it.
 - Impossible

Goal of Windows

Common feature: A better way to make groups

- Dynamic
- Ranking
- Position aware
- Sums and averages over a group

Main idea: For each result line, create a *window* of rows around the result

- We know how many there are
- We can set a range or interval
- We can check position, rank, and do aggregation functions

In contrast to GROUP BY, we may have more than one value result per group

Window Example

```
SELECT L.state, T.month, AVG(S.sales) OVER W
       as movavg
FROM Sales S, Times T, Locations L
WHERE S.timeid = T.timeid AND S.locid =
       L.locid
WINDOW W AS (PARTITION BY L.state
              ORDER BY T.month
              RANGE BETWEEN INTERVAL '1' month PRECEDING
              AND INTERVAL '1' month FOLLOWING)
```

In English: For each state, calculate the moving 3 month average of sales for each month

- Moving Average is the Average of the sales for months $x-1$, x , $x+1$

Evaluating the Window

```
SELECT L.state, T.month, AVG(S.sales) OVER W as
    movavg
FROM Sales S, Times T, Locations L
WHERE S.timeid = T.timeid AND S.locid = L.locid
WINDOW W AS (PARTITION BY L.state
    ORDER BY T.month
    RANGE BETWEEN INTERVAL '1' month PRECEDING
    AND INTERVAL '1' month FOLLOWING)
```

Step 1: Do the usual WHERE

Step 2: Divide up by the PARTITION BY clause

- Here, by L.state

Step 3: Order the partition by the ORDER BY clause

- Here, T.month

Step 4: Define a frame for the partition, how large, how to choose

- Here it's 1 before and 1 after on the T.month attribute

Step 5: Aggregate by the function

- Here it's AVG over the sales

The Tables

Time Dimension

TimeId	Date	Week	Month	Quarter	Year	Holiday?
1	'1 Jan 09'	1	1	1	2009	True
2	'8 Jan 09'	2	1	1	2009	False
3	'15 Jan 09'	3	1	1	2009	False
4	'22 Jan 09'	4	1	1	2009	False
5	'29 Jan 09'	5	1	1	2009	False
6	'5 Feb 09'	6	2	1	2009	False
7	'12 Feb 09'	7	2	1	2009	False
8	'19 Feb 09'	8	2	1	2009	False
9	'26 Feb 09'	9	2	1	2009	False
10	'5 Mar 09'	10	3	1	2009	False
11	'12 Mar 09'	11	3	1	2009	False
12	'19 Mar 09'	12	3	1	2009	False
13	'26 Mar 09'	13	3	1	2009	True
14	'2 Apr 09'	1	4	2	2009	False
15	'9 Apr 09'	2	4	2	2009	True
16	'16 Apr 09'	3	4	2	2009	True
17	'23 Apr 09'	4	4	2	2009	False
18	'30 Apr 09'	5	4	2	2009	False
19	'7 May 09'	6	5	2	2009	False
20	'14 May 09'	7	5	2	2009	False
21	'21 May 09'	8	5	2	2009	False
22	'28 May 09'	9	5	2	2009	True
23	'4 Jun 09'	10	6	2	2009	False
24	'11 Jun 09'	11	6	2	2009	False
25	'18 Jun 09'	12	6	2	2009	False
26	'25 Jun 09'	13	6	2	2009	False

Products Dimension

pid	pname	category	price
11	Lee James	Apparel	25
12	Zork	Games	18
13	Biro Pen	Stationery	2

Locations Dimension

locid	city	state	country
1	Madison	WI	USA
2	Fresno	CA	USA
3	Chennai	TN	India

pid	timeid	locid	sales
11	3	1	87
11	5	1	64
11	7	1	51
11	2	1	86
11	9	1	70
11	6	1	86
11	8	1	70
12	7	1	49
12	8	2	57
12	3	2	104
12	9	2	72
13	1	2	193
11	3	2	86
11	10	1	70
12	8	1	49
12	6	2	57
12	4	2	104
12	5	2	72
13	12	2	193
13	14	2	153
13	16	2	143
13	15	2	133
13	17	2	13
13	18	2	293
13	18	1	93

Step 1: Connect Tables

```
SELECT L.state, T.month,  
       AVG(S.sales) OVER W as  
       movavg  
FROM Sales S, Times T,  
       Locations L  
WHERE S.timeid = T.timeid  
       AND S.locid = L.locid  
WINDOW W AS (PARTITION BY  
              L.state  
              ORDER BY T.month  
              RANGE BETWEEN INTERVAL  
              '1' month PRECEDING  
              AND INTERVAL '1' month  
              FOLLOWING)
```

L.State	T.Month	sales
CA	1	72
CA	1	86
CA	1	104
CA	1	104
CA	1	193
CA	2	57
CA	2	57
CA	2	72
CA	3	193
CA	4	13
CA	4	133
CA	4	143
CA	4	153
CA	4	293
WI	1	64
WI	1	86
WI	1	87
WI	2	49
WI	2	49
WI	2	51
WI	2	70
WI	2	70
WI	2	86
WI	3	70
WI	4	93

Step 2 & 3: Partition (by state) and Order

```
SELECT L.state, T.month,
       AVG(S.sales) OVER W as
       movavg
FROM Sales S, Times T,
       Locations L
WHERE S.timeid = T.timeid
      AND S.locid = L.locid
WINDOW W AS (PARTITION BY
             L.state
             ORDER BY T.month
             RANGE BETWEEN INTERVAL
             '1' month PRECEDING
             AND INTERVAL '1' month
             FOLLOWING)
```

L.State	T.Month	sales
CA	1	72
CA	1	86
CA	1	104
CA	1	104
CA	1	193
CA	2	57
CA	2	57
CA	2	72
CA	3	193
CA	4	13
CA	4	133
CA	4	143
CA	4	153
CA	4	293

L.State	T.Month	sales
WI	1	64
WI	1	86
WI	1	87
WI	2	49
WI	2	49
WI	2	51
WI	2	70
WI	2	70
WI	2	86
WI	3	70
WI	4	93

Step 4: Break into Partitions (CA)

```
SELECT L.state, T.month,
       AVG(S.sales) OVER W as
       movavg
FROM Sales S, Times T,
     Locations L
WHERE S.timeid = T.timeid
     AND S.locid = L.locid
WINDOW W AS (PARTITION BY
             L.state
             ORDER BY T.month
             RANGE BETWEEN INTERVAL
             '1' month PRECEDING
             AND INTERVAL '1' month
             FOLLOWING)
```

L.State	T.Month	sales	W	W	W	W
CA	1	72	1	2		
CA	1	86				
CA	1	104				
CA	1	104				
CA	1	193				
CA	2	57			3	
CA	2	57				
CA	2	72				
CA	3	193				4
CA	4	13				
CA	4	133				
CA	4	143				
CA	4	153				
CA	4	293				

Step 4: Break into Partitions (WI)

```
SELECT L.state, T.month,
       AVG(S.sales) OVER W as
       movavg
FROM Sales S, Times T,
       Locations L
WHERE S.timeid = T.timeid
      AND S.locid = L.locid
WINDOW W AS (PARTITION BY
              L.state
              ORDER BY T.month
              RANGE BETWEEN INTERVAL
              `1` month PRECEDING
              AND INTERVAL `1` month
              FOLLOWING)
```

L.State	T.Month	sales	W	W	W	W
WI	1	64	1	2	3	4
WI	1	86				
WI	1	87				
WI	2	49				
WI	2	49				
WI	2	51				
WI	2	70				
WI	2	70				
WI	2	86				
WI	3	70				
WI	4	93				

Step 5: Calculate Average (CA)

```

SELECT L.state, T.month,
       AVG(S.sales) OVER W as
       movavg
FROM Sales S, Times T,
       Locations L
WHERE S.timeid = T.timeid
      AND S.locid = L.locid
WINDOW W AS (PARTITION BY
              L.state
              ORDER BY T.month
              RANGE BETWEEN INTERVAL
              '1' month PRECEDING
              AND INTERVAL '1' month
              FOLLOWING)
    
```

L.State	T.Month	sales	W	W	W	W
CA	1	72	1 93	2 104		
CA	1	86				
CA	1	104				
CA	1	104				
CA	1	193				
CA	2	57		3 123		
CA	2	57				
CA	2	72				
CA	3	193			4 154	
CA	4	13				
CA	4	133				
CA	4	143				
CA	4	153				
CA	4	293				

Step 5: Calculate Average (WI)

```
SELECT L.state, T.month,
       AVG(S.sales) OVER W as
       movavg
FROM Sales S, Times T,
       Locations L
WHERE S.timeid = T.timeid
      AND S.locid = L.locid
WINDOW W AS (PARTITION BY
              L.state
              ORDER BY T.month
              RANGE BETWEEN INTERVAL
              `1` month PRECEDING
              AND INTERVAL `1` month
              FOLLOWING)
```

L.State	T.Month	sales	W	W	W	W
WI	1	64	1 68	2 68	3 67	4 81
WI	1	86				
WI	1	87				
WI	2	49	1 68	2 68	3 67	4 81
WI	2	49				
WI	2	51				
WI	2	70				
WI	2	70				
WI	2	86				
WI	3	70	1 68	2 68	3 67	4 81
WI	4	93				

Result

```
SELECT L.state, T.month,
       AVG(S.sales) OVER W as
       movavg
FROM Sales S, Times T,
       Locations L
WHERE S.timeid = T.timeid
      AND S.locid = L.locid
WINDOW W AS (PARTITION BY
              L.state
              ORDER BY T.month
              RANGE BETWEEN INTERVAL
              '1' month PRECEDING
              AND INTERVAL '1' month
              FOLLOWING)
```

L.State	T.Month	movavg
CA	1	93
CA	2	104
CA	3	123
CA	4	154
WI	1	68
WI	2	68
WI	3	67
WI	4	81

Calculating Without Windows

```
SELECT L.state, T.month,  
       AVG(S.sales) OVER W as  
       movavg  
FROM Sales S, Times T,  
       Locations L  
WHERE S.timeid = T.timeid  
       AND S.locid = L.locid  
WINDOW W AS (PARTITION BY  
              L.state  
              ORDER BY T.month  
              RANGE BETWEEN INTERVAL  
              '1' month PRECEDING  
              AND INTERVAL '1' month  
              FOLLOWING)
```

```
SELECT L.state as state, 1, AVG(S.sales) as avg  
FROM Sales2 S, Times2 T, Locations L  
WHERE S.timeid = T.timeid AND S.locid = L.locid  
and (month = 0 OR month = 1 or month = 2)  
group by L.state
```

```
SELECT L.state as state, 2, AVG(S.sales) as avg  
FROM Sales2 S, Times2 T, Locations L  
WHERE S.timeid = T.timeid AND S.locid = L.locid  
and (month = 1 OR month = 2 or month = 3)  
group by L.state
```

```
SELECT L.state as state, 3, AVG(S.sales) as avg  
FROM Sales2 S, Times2 T, Locations L  
WHERE S.timeid = T.timeid AND S.locid = L.locid  
and (month = 2 OR month = 3 or month = 4)  
group by L.state
```

```
SELECT L.state as state, 4, AVG(S.sales) as avg  
FROM Sales2 S, Times2 T, Locations L  
WHERE S.timeid = T.timeid AND S.locid = L.locid  
and (month = 3 OR month = 4 or month = 5)  
group by L.state
```

Framing Options

We framed a window using `RANGE`

- Range can only be defined for numbers and dates
- Work over an `ordering`
- Could be made unbounded: `UNBOUNDED PRECEDING` or `UNBOUNDED FOLLOWING`

Another option: frame based on a `fixed` number of rows before and after the result

```
SELECT L.state, T.month, AVG(S.sales) OVER W as movavg
FROM Sales S, Times T, Locations L
WHERE S.timeid = T.timeid AND S.locid = L.locid
WINDOW W AS (PARTITION BY L.state
              ORDER BY T.month
              RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING)
```

- The one-before, one-after window makes sense if there is exactly **one row for each entry** based on the columns used for `ORDER BY`
 - For example, one row per month

So Far

- Multidimensional Aggregation Queries
 - ROLLUP
 - CUBE
- Window Queries
 - PARTITION BY Framing a Window
- New Aggregate Functions
- Finding Answers Quickly
- Implementation Techniques for OLAP
 - Bitmap Indexes

Aggregation: Rank

OLAP extensions include operations which show the **position of a row in the output**

- **RANK()** inserts the position of each row in the output based on the sorting column
 - If there are 15 rows, the results are 1-15
 - If rows 2 and 3 have the same value in the RANK column, the result is: 1, 2, 2, 4, . . .
- **DENSE_RANK()** skips the gaps, so in the above case, you'd have: 1, 2, 2, 3, . . .
- In MS SQL Server you need to define rank over a given ORDER BY clause
 - And an optional PARTITION BY as well

Rank Examples

```
SELECT S.sid, S.sname, S.age,  
       S.rating, RANK() OVER (ORDER  
       BY S.rating) FROM Sailors S
```

sid	sname	age	rating	(rank)
29	Brutus	33	1	1
85	Art	25	3	2
95	Bob	63	3	2
45	Tina	23	6	4
64	Horatio	35	7	5
22	Dustin	45	7	5
31	Lubber	55	8	7
32	Andy	25	8	7
74	Horatio	35	9	9
71	Zorba	16	10	10
58	Rusty	35	10	10

```
SELECT S.sid, S.sname, S.age,  
       S.rating, RANK() OVER  
       (PARTITION BY S.age ORDER BY  
       S.rating) FROM Sailors S
```

sid	sname	age	rating	(rank)
71	Zorba	16	10	1
45	Tina	23	6	1
85	Art	25	3	1
32	Andy	25	8	2
29	Brutus	33	1	1
64	Horatio	35	7	1
74	Horatio	35	9	2
58	Rusty	35	10	3
22	Dustin	45	7	1
31	Lubber	55	8	1
95	Bob	63	3	1

Dense_Rank Examples

```
SELECT S.sid, S.sname, S.age,  
       S.rating, DENSE_RANK() OVER  
       (ORDER BY S.rating) FROM  
       Sailors S
```

sid	sname	age	rating	(rank)
29	Brutus	33	1	1
85	Art	25	3	2
95	Bob	63	3	2
45	Tina	23	6	3
64	Horatio	35	7	4
22	Dustin	45	7	4
31	Lubber	55	8	5
32	Andy	25	8	5
74	Horatio	35	9	6
71	Zorba	16	10	7
58	Rusty	35	10	7

```
SELECT S.sid, S.sname, S.age,  
       S.rating, DENSE_RANK() OVER  
       (PARTITION BY S.age ORDER BY  
       S.rating) FROM Sailors S
```

sid	sname	age	rating	(rank)
71	Zorba	16	10	1
45	Tina	23	6	1
85	Art	25	3	1
32	Andy	25	8	2
29	Brutus	33	1	1
64	Horatio	35	7	1
74	Horatio	35	9	2
58	Rusty	35	10	3
22	Dustin	45	7	1
31	Lubber	55	8	1
95	Bob	63	3	1

Aggregation: Percent_Rank

PERCENT_RANK() shows the relative position of a row in the partition:

– $\frac{RANK()-1}{n-1}$ if there are n rows

Meaning: Where we would insert the row into the set if it weren't already there.

Practically, for a n row group:

- Row 1 is 0
- Row n is 1
- Rank increase in equal steps to 1

Not supported in MS SQL Server, only in Oracle

- Could be done using user code and stored procedures

Example: SELECT DeptID, Surname, Salary, Sex, PERCENT_RANK() OVER (PARTITION BY Sex ORDER BY Salary DESC) AS PctRank FROM Employees WHERE State = 'NY';

	DeptID	Surname	Salary	Sex	PctRank
1	200	Martel	55700	M	0.0
2	100	Guevara	42998	M	0.33333333 3
3	100	Soo	39075	M	0.66666666 7
4	400	Ahmed	34992	M	1
5	300	Davidson	57090	F	0
6	400	Blaikie	54900	F	0.33333333 3
7	100	Whitney	45700	F	0.66666666 7
8	400	Wetherby	35745	F	1

Aggregation: Cume_Dist

CUME_DIST computes the relative position of a result row in the partition

- The result is always between (0, 1]

Example: `SELECT job_id, last_name, salary, CUME_DIST() OVER (PARTITION BY job_id ORDER BY salary) AS cumedist FROM Emp WHERE job_id LIKE 'PU%'`

Job_ID	Lastname	Salary	cumedist
PU-CLERK	Chico	2500	0.2
PU-CLERK	Groucho	2600	0.4
PU-CLERK	Harpo	2800	0.6
PU-CLERK	Gummo	2900	0.8
PU-CLERK	Zeppo	3100	1
PU-MAN	Frenchy	11000	1

40% of clerks have salaries equal to or less than Groucho

What about NULL?

- Oracle lets you define whether NULLs are considered first or last
 - The command NULLS FIRST or NULLS LAST does the obvious

So Far

- Multidimensional Aggregation Queries
 - ROLLUP
 - CUBE
- Window Queries
 - PARTITION BY Framing a Window
- New Aggregate Functions
- Finding Answers Quickly
- Implementation Techniques for OLAP
 - Bitmap Indexes

Finding Answers Quickly

- Sometimes you want the top 10 or top 100 results for the query, not every one
- Sometimes you want an approximate answer at first and then let the details later

Top N Rows

For the top N rows for a query, a few options

- In MS SQL, we can write the following query:
`SELECT TOP 10 eid, ename, salary, age
FROM Emp ORDER BY age;`

Shows the top 10 rows for the result based on an ordering of the age column

To get the bottom 10 by running the ordering in reverse:

```
SELECT TOP 10 eid, ename, salary, age  
FROM Emp ORDER BY age DESC;
```

Top N Rows

Another option: Set the ranking by percent rather than absolute position (PERCENT) or allow ties to count as a single row (WITH TIES)

Examples:

1. `SELECT TOP 10 sid, sname, rating FROM Sailors ORDER BY rating DESC`
2. `SELECT TOP 10 WITH TIES sid, sname, rating FROM Sailors ORDER BY rating DESC`
3. `SELECT TOP 10 PERCENT sid, sname, rating FROM Sailors ORDER BY rating DESC`
4. `SELECT TOP 10 PERCENT WITH TIES sid, sname, rating FROM Sailors ORDER BY rating DESC`

Efficiency?

To get the top 10 did you need to calculate the entire set and then just take the top ten results?

- In IBM DB2 (and something similar in Oracle) you can tell the DBMS to optimize for a given result subset:

```
SELECT P.pid, P.pname, S.sales  
FROM Sales S, Products P  
WHERE S.pid = P.pid AND S.locid = 1 AND S.timeid = 3  
ORDER BY S.sales DESC  
OPTIMIZE FOR 10 ROWS
```

The above query has the same semantics as the TOP query in MS SQL

Oracle Options

- In Oracle you can write: `WHERE ROWNUM < 11` to get the top 10
- You **can't** write `WHERE ROWNUM > 11` to skip the top 11
Why?
 - The first row selected is given rownum 1, so it's discarded
 - Then the second row is brought in and it also gets rownum 1, so it's discarded, etc.

How does Optimize work?

Optimize runs more efficiently using heuristics about the result relation

- The DBMS estimates the distribution of the values in the sort column, and guesses where the top 10 sales values are. Call it c .
- It then runs the query below, assuming that the value c is guaranteed to be below the top 10 sales:

```
SELECT P.pid, P.name, S.sales  
FROM Sales S, Products P WHERE S.pid = P.pid AND S.locid = 1  
AND S.timeid = 4 AND S.sales > c  
ORDER BY S.sale DESC
```

If c is too high, we'll get less than 10, if it's too low, we'll get more than 10, so there is more work to be done.

The DBMS can maintain a histogram of the values in the ORDER BY column to help (if there aren't too many filters on the WHERE clause)

Online Aggregation

Another task: DBMS may optimize long aggregation queries

The DBMS may be able to give preliminary results from long running queries, an approximation

- Still only in theory now

This is **online aggregation** and depends aggregation functions which produce intermediate results with a given confidence interval

- Results can be computed from non-blocking algorithms which produce partial results
- Another option: perform a fast randomized algorithm to estimate the answer at first and then update it when the results are actually computed

Example: Online Aggregation

- Consider the mockup in Figure 12.1 for some idea about how it could work on the query:

```
SELECT L.state, AVG(S.sales)
FROM Sales S, Locations L
WHERE S.locid = L.locid GROUP BY L.state
```

Status	Prioritize?	State	AVG(sales)	Confidence	Interval
85%	✓	Alabama	5,232.5	97%	103.4
30%		Alaska	2,832.5	93%	132.2
90%	✓	Arizona	6,432.5	98%	52.3
⋮	⋮	⋮	⋮	⋮	⋮
20%		Wyoming	4,243.5	92%	152.3

So Far

- Multidimensional Aggregation Queries
 - ROLLUP
 - CUBE
- Window Queries
 - PARTITION BY Framing a Window
- New Aggregate Functions
- Finding Answers Quickly
- Implementation Techniques for OLAP
 - Bitmap Indexes

Implementation Techniques for OLAP

OLAP applications make interesting variations on the demands for database implementation

We'll talk about some indexing techniques which are appropriate for OLAP

Why are OLAP indexes different?

- OLAP is **mostly read-only**, so indexes won't be updated often
- Indexes must be very fast and able to deal with large quantities of data (10,000 rows, 100,000 rows, etc.)

Bitmap Indexes

A **bitmap index** is a very efficient method for storing **sparse data columns**

- The idea is applicable to many places where sparse data is stored

Sparse data columns are ones which contain data values from a very small set of possibilities

General idea: Prepare an index table with a column for each possible value in the original column c (only a few)

- *IndexTable* (*val1:bit, val2:bit, val3:bit, val4:bit, val5:bit, ...*)

- For a table with n rows, make an n row index table
 - If column c in row m has value X , put a 1 in column X of row m of the index table

Bitmap Index Example

Original Table:

Customers (*custid* integer, *name* varchar(25), *sex* varchar(1), *rating* integer)

- Rating only holds values between 1 and 5 and sex can be either M (male) or F (female)

Original Table

custid	name	sex	rating
112	Joe	M	3
115	Ram	M	5
119	Sue	F	5
113	Woo	M	4

Sex Index

M	F
1	0
1	0
0	1
1	0

Rating Index

1	2	3	4	5
0	0	1	0	0
0	0	0	0	1
0	0	0	0	1
0	0	0	1	0

Using Bitmap Indexes

Bitmap indexes gives us two advantages over standard B+ indexes

1. You can check AND and OR conditions on columns with just a simple logical AND or OR over the bit values
 - **Example:** Find how many male customers have rating 5.
 - Extract the first column from the sex index and the fifth column from the rating index
 - Perform a bit-wise AND between them
 - **Example:** Find how many customer have rating 2 or 5
 - Extract the second and fifth columns from the rating index
 - Perform a logical OR
2. They are very easily compressed on disk or in memory using standard compression algorithms

When to use them?

Bitmaps are great when there are a few possible values and...

1. We **don't add** too many rows (and care about order)
 2. We **don't delete** too many rows
 - Either we reshuffle the index or mark the row as deleted in the index
 3. There are **many entries** for each possible value
 - If there are only a **few entries** for each value, a normal (value, rid-list) approach is better
-

So for a column/column set with:

1. **Many possible values and many entries per value** → normal tree/hash index
 2. **Few possible values and many read-mostly entries per value** → bitmap index
 3. **Few possible values and few read-mostly entries per value** → (value, rid-list) index
-

Conclusion

- Multidimensional Aggregation Queries
 - ROLLUP
 - CUBE
- Window Queries
 - PARTITION BY Framing a Window
- New Aggregate Functions
- Finding Answers Quickly
- Implementation Techniques for OLAP
 - Bitmap Indexes