Overview of Analytical SQL in Data Warehousing

Supriya Ananth
Curriculum Developer

supriya.ananth@oracle.com
What everybody knows about Databases

CREATE TABLE emp;

INSERT INTO emp VALUES (...);

SELECT * FROM emp;

DROP TABLE emp;
What everybody should know
Comprehensive in-database Analytics
What everybody should know
SQL Analytics

- Pattern matching
- Analysis and reporting
- Aggregation
Analytical SQL
Analytical SQL
Address Common Business Questions

- What is January’s sales as a percentage of the entire year’s?
  - Compare aggregates on different levels
- Who are the top ten sales-reps in each region?
  - Rank
- Find large transactions occurring within a specified time interval
  - Pattern matching
Analytical SQL in the Database

Benefits

- Dramatically enhances the analysis capabilities with SQL
- Simplifies SQL development
- Increased performance
  - Minimized analysis time
  - Internal optimizations
Agenda

- Grouping and Aggregating Data Using SQL
- Hierarchical Retrieval
- Working with Regular Expressions
- Analyzing and Reporting Data Using SQL
- Pattern Matching
Agenda

- Grouping and Aggregating Data Using SQL
  - Hierarchical Retrieval
  - Working with Regular Expressions
  - Analyzing and Reporting Data Using SQL
  - Pattern Matching
Grouping and Aggregating Data Using SQL

Overview

To improve aggregation performance in a data warehouse, Oracle Database provides the following functionality:

- CUBE and ROLLUP extensions to the GROUP BY clause
- GROUPING SETS expression
- GROUPING function
Using the ROLLUP and CUBE Operators

Using the ROLLUP operator

```sql
SELECT department_id, job_id, SUM(salary)
FROM hr.employees
WHERE department_id < 60
GROUP BY ROLLUP(department_id, job_id);
```

Using the CUBE operator

```sql
SELECT department_id, job_id, SUM(salary)
FROM hr.employees
WHERE department_id < 60
GROUP BY CUBE(department_id, job_id);
```
Grouping Sets

Overview

- The `GROUPING SETS` syntax is used to define multiple groupings in the same query.
- Grouping set efficiency:
  - Only one pass over the base table is required.
  - There is no need to write complex `UNION` statements.
  - The more elements `GROUPING SETS` has, the greater the performance benefit.
Example of Grouping Sets

Comparison

Without GROUPING SETS expression

SELECT department_id, job_id, NULL as manager_id, AVG(salary) as AVGSAL
FROM hr.employees
GROUP BY department_id, job_id
UNION ALL
SELECT NULL, job_id, manager_id, avg(salary) as AVGSAL
FROM hr.employees
GROUP BY job_id, manager_id;

With GROUPING SETS expression

SELECT department_id, job_id, manager_id, AVG(salary)
FROM hr.employees
GROUP BY GROUPING SETS
((department_id, job_id),
 (job_id, manager_id)), ()

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>JOB_ID</th>
<th>MANAGER_ID</th>
<th>AVG(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SH_CLERK</td>
<td>121</td>
<td>3200</td>
</tr>
<tr>
<td>2</td>
<td>AC_MGR</td>
<td>101</td>
<td>12000</td>
</tr>
<tr>
<td>3</td>
<td>ST_MAN</td>
<td>100</td>
<td>7200</td>
</tr>
<tr>
<td>4</td>
<td>ST_CLERK</td>
<td>121</td>
<td>2675</td>
</tr>
<tr>
<td>5</td>
<td>SA_REP</td>
<td>148</td>
<td>8500</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>JOB_ID</th>
<th>MANAGER_ID</th>
<th>AVG(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>110 AC_ACCOUNT</td>
<td>8300</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>90 AD_MGR</td>
<td>17000</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>50 ST_CLERK</td>
<td>2705</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>6461.83</td>
<td>130841</td>
<td></td>
</tr>
</tbody>
</table>
The Grouping Function

Explanation with Example

The `GROUPING` function:

- Is used with the `CUBE` or `ROLLUP` operator
- Is used to find the groups forming the subtotal in a row
- Returns 0 or 1

```sql
SELECT department_id DEPTID, job_id JOB, SUM(salary),
GROUPING(department_id) GRP_DEPT,
GROUPING(job_id) GRP_JOB
FROM hr.employees
WHERE department_id < 50
GROUP BY ROLLUP(department_id, job_id);
```
You will perform business intelligence queries for a company named XYZ Sales Consultants.

One of their requirements involves creating business statistics reports to aid in decision support.

To help support the ITDB folks process the requirement, you will perform business intelligence queries using analytical SQL functions on the data available in the data warehouse.
Case Study

Scenario 1- Using Aggregate Functions

Create a cross-tabular report showing the total sales by both, `country_id` and `channel_desc` for the US and France through the Internet and direct sales in September 2000.

```sql
SELECT channels.channel_desc, countries.country_iso_code,
       TO_CHAR(SUM(amount_sold), '9,999,999,999') AS SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id = times.time_id AND
  sales.cust_id = customers.cust_id AND
    sales.channel_id = channels.channel_id AND
  channels.channel_desc IN ('Direct Sales', 'Internet') AND
  times.calendar_month_desc = '2000-09'
  AND customers.country_id = countries.country_id
  AND countries.country_iso_code IN ('US', 'FR')
GROUP BY CUBE(channels.channel_desc, countries.country_iso_code);
```

<table>
<thead>
<tr>
<th>CHANNEL_DESC</th>
<th>COUNTRY_ISO_CODE</th>
<th>SALES$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Sales</td>
<td>FR</td>
<td>835,224</td>
</tr>
<tr>
<td>Internet</td>
<td>US</td>
<td>714,730</td>
</tr>
<tr>
<td>Direct Sales</td>
<td>US</td>
<td>132,821</td>
</tr>
<tr>
<td>Internet</td>
<td>FR</td>
<td>9,597</td>
</tr>
<tr>
<td>Direct Sales</td>
<td>US</td>
<td>124,224</td>
</tr>
<tr>
<td>Direct Sales</td>
<td>FR</td>
<td>61,202</td>
</tr>
<tr>
<td>Direct Sales</td>
<td>US</td>
<td>638,201</td>
</tr>
</tbody>
</table>

Output
Agenda

- Grouping and Aggregating Data Using SQL
- Hierarchical Retrieval
- Working with Regular Expressions
- Analyzing and Reporting Data Using SQL
- Pattern Matching
Hierarchical Retrieval

Using Hierarchical Queries

- You can use hierarchical queries to retrieve data based on a natural hierarchical relationship between rows in a table.
- A hierarchical query is possible only when a relationship exists between rows in a table.
- A process called “tree walking” enables you to construct hierarchy between the rows in a single table.
Hierarchical Retrieval

Using Hierarchical Queries - Example

MANAGER_ID = 101 (Child)

EMPLOYEE_ID = 100 (Parent)
Hierarchical Retrieval
Walking the Tree

- Specify the starting point
  - Example:

    ```
    . . .
    START WITH last_name = 'Kochhar'
    ```

- Specify the direction of the query
  - Example:

    ```
    . . .
    CONNECT BY PRIOR employee_id = manager_id
    ```

Syntax:

```sql
SELECT [LEVEL], column, expr...
FROM   table
[WHERE condition(s)]
[START WITH condition(s)]
[CONNECT BY PRIOR condition(s)] ;
```
Hierarchical Retrieval

Example

- The example, walking from the bottom up, displays a list of managers starting with the employee whose employee ID is 101.

```sql
SELECT employee_id, last_name, job_id, manager_id
FROM hr.employees
START WITH employee_id = 101
CONNECT BY PRIOR manager_id = employee_id;
```
Hierarchical Retrieval

The `LEVEL` Pseudocolumn

- Oracle SQL uses the `LEVEL` pseudocolumn to show parent and child rows in a table.

```
SELECT employee_id, last_name, manager_id, LEVEL
FROM hr.employees
START WITH employee_id = 100
CONNECT BY PRIOR employee_id = manager_id
ORDER siblings BY last_name;
```
Hierarchical Retrieval
Pruning Nodes and Branches

- Use the `WHERE` clause to eliminate a node.
  ```sql
  ... WHERE last_name != 'Higgins'
  ```

- Use the `CONNECT BY` clause to eliminate a branch.
  ```sql
  ... CONNECT BY PRIOR employee_id = manager_id AND last_name != 'Higgins'
  ```
Case Study
Scenario 2 - Using Hierarchical Retrieval

Help the ITDB folks generate a company organization chart excluding:

- All employees with a job ID of IT_PROG
- Manager De Haan and those employees who report to De Haan.

```sql
SELECT last_name, employee_id, manager_id
FROM hr.employees
WHERE job_id != 'IT_PROG'
START WITH manager_id IS NULL
CONNECT BY PRIOR employee_id = manager_id
AND last_name != 'De Haan';
```

Output
Agenda

- Grouping and Aggregating Data Using SQL
- Hierarchical Retrieval
- Working with Regular Expressions
- Analyzing and Reporting Data Using SQL
- Performing Pivot Operations to Analyze and Report Data
- Using Regular Expressions for Pattern Matching
Regular Expressions

Overview

- You can use regular expressions to search for (and manipulate) simple and complex patterns in string data.
- You specify a regular expression by using:
  - Metacharacters
  - Literals
- Regular expressions make string manipulations more powerful and less cumbersome
## Regular Expressions

### Regular Expressions Functions in SQL

<table>
<thead>
<tr>
<th>Function or Condition Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGEXP_LIKE condition</td>
<td>Similar to the <strong>LIKE</strong> operator, but performs regular expression matching instead of simple pattern matching</td>
</tr>
<tr>
<td>REGEXP_REPLACE function</td>
<td>Searches for a regular expression pattern and replaces it with a replacement string</td>
</tr>
<tr>
<td>REGEXP_INSTR function</td>
<td>Searches a string for a regular expression pattern and returns the position where the match is found</td>
</tr>
<tr>
<td>REGEXP_SUBSTR function</td>
<td>Searches for a regular expression pattern within a given string and extracts the matched substring</td>
</tr>
<tr>
<td>REGEXP_COUNT function</td>
<td>Returns an integer indicating the number of occurrences of pattern</td>
</tr>
</tbody>
</table>
## Regular Expressions

### List of Metacharacters

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Matches any character in the supported character set, except NULL</td>
</tr>
<tr>
<td>+</td>
<td>Matches one or more occurrence</td>
</tr>
<tr>
<td>?</td>
<td>Matches zero or one occurrence</td>
</tr>
<tr>
<td>*</td>
<td>Matches zero or more occurrences of preceding subexpression</td>
</tr>
<tr>
<td>{m}</td>
<td>Matches exactly ( m ) occurrences of the preceding expression</td>
</tr>
<tr>
<td>{m,}</td>
<td>Matches at least ( m ) occurrences of the preceding subexpression</td>
</tr>
<tr>
<td>{m,n}</td>
<td>Matches at least ( m ), but not more than ( n ) occurrences of the preceding subexpression</td>
</tr>
<tr>
<td>[...]</td>
<td>Matches any single character in the list within the brackets</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>{...}</td>
<td>Treats the enclosed expression within the parentheses as a unit. The subexpression can be a string of literals or a complex expression containing operators.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Matches the beginning of a string</td>
</tr>
<tr>
<td>$</td>
<td>Matches the end of a string</td>
</tr>
<tr>
<td>\</td>
<td>Treats the subsequent metacharacter in the expression as a literal</td>
</tr>
<tr>
<td>\n</td>
<td>Matches the ( n )th (1–9) preceding subexpression of whatever is grouped within parentheses. The parentheses cause an expression to be remembered; a backreference refers to it.</td>
</tr>
<tr>
<td>\d</td>
<td>A digit character</td>
</tr>
<tr>
<td>[:class:]</td>
<td>Matches any character belonging to the specified POSIX character class</td>
</tr>
<tr>
<td>[^:class:]</td>
<td>Matches any single character not in the list within the brackets</td>
</tr>
</tbody>
</table>
Regular Expressions

Activity

Create a regular expression to search for a word which:
- Starts with literals `f` or `ht`
- Followed by literals `tp` with an optional literal `s` after literal `p`
- Ends with a colon (`:`)
Regular Expressions

Metacharacters

- Metacharacters are characters that have a special meaning such as a wildcard, a repeating character, a non-matching character, or a range of characters.
- For example, the ‘^ (f | ht)tps?:$’ regular expression searches for the following from the beginning of the string:
  - The literals f or ht at the beginning of the string
  - The t literal
  - The p literal, optionally followed by the s literal
  - The colon “:” literal at the end of the string
Regular Expressions
Using the `REGEXP_LIKE` Condition

Syntax:

```
REGEXP_LIKE(source_char, pattern [, match_parameter ])
```

Example:

```
SELECT first_name, last_name
FROM employees
WHERE REGEXP_LIKE
  (first_name, '^St(e|v|ph)en$');
```

<table>
<thead>
<tr>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Steven</td>
<td>King</td>
</tr>
<tr>
<td>Steven</td>
<td>Markle</td>
</tr>
<tr>
<td>Stephen</td>
<td>Stiles</td>
</tr>
</tbody>
</table>

3 rows selected
Regular Expressions
Using the `REGEXP_INSTR` Function

Syntax:

```
REGEXP_LIKE(source_char, pattern [, match_parameter ])
```

Example:

```
SELECT street_address,
REGEXP_INSTR
  (street_address, '[:alpha:]')
AS First_Alpha_Position
FROM locations;
```

<table>
<thead>
<tr>
<th>STREET_ADDRESS</th>
<th>FIRST_ALPHA_POSITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1297 Via Cola di Rie</td>
<td>6</td>
</tr>
<tr>
<td>93091 Calle della Testa</td>
<td>7</td>
</tr>
<tr>
<td>2017 Shinjuku-ku</td>
<td>6</td>
</tr>
<tr>
<td>9459 Kameya-cho</td>
<td>6</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Rue Feti Canova 1350</td>
<td>1</td>
</tr>
<tr>
<td>20 Rue des Corps-Saints</td>
<td>4</td>
</tr>
<tr>
<td>Kantonstrasse 921</td>
<td>1</td>
</tr>
<tr>
<td>Pieter Breugelstrasse 837</td>
<td>1</td>
</tr>
<tr>
<td>Marcos Escobedo 9991</td>
<td>1</td>
</tr>
</tbody>
</table>

23 rows selected
Regular Expressions
Using the `REGEXP_SUBSTR` Function

Syntax:

```
REGEXP_SUBSTR (source_char, pattern [, position
 [, occurrence [, match_option]]])
```

Example:

```
SELECT REGEXP_SUBSTR(street_address , ' \[^ ]+ $') "Road"
FROM locations;
```

```
ROAD
------------------------
Via
Calle
Jabberwocky
Interiors
Sapata
Charade
Spindola
Boxwood
...
```
Regular Expressions

Using the `REGEXP_REPLACE` Function

Syntax:

```
REGEXP_REPLACE(source_char, pattern [,replacestr [, position [, occurrence [, match_option]]]]]
```

Example:

```
SELECT last_name,
    REGEXP_REPLACE(phone_number, '\.', '-') AS phone
FROM employees;
```

```
<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>PHONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>515-123-4567</td>
</tr>
<tr>
<td>Soothar</td>
<td>515-123-4568</td>
</tr>
<tr>
<td>De Hoan</td>
<td>515-123-4569</td>
</tr>
<tr>
<td>Humbold</td>
<td>590-423-4567</td>
</tr>
<tr>
<td>Ernst</td>
<td>590-423-4566</td>
</tr>
<tr>
<td>Austin</td>
<td>590-423-4569</td>
</tr>
<tr>
<td>Patakala</td>
<td>590-423-4560</td>
</tr>
<tr>
<td>Lozentz</td>
<td>590-613-5567</td>
</tr>
<tr>
<td>Greenberg</td>
<td>515-126-4560</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Higgins</td>
<td>515-123-8889</td>
</tr>
<tr>
<td>Giese</td>
<td>515-123-8181</td>
</tr>
</tbody>
</table>

107 rows selected
```
Regular Expressions

Using the `REGEXP_COUNT` Function

Syntax:

```
REGEXP_COUNT (source_char, pattern [, position
[, occurrence [, match_option]]])
```

Example:

```
SELECT REGEXP_COUNT(
    'ccacctttcccctccactcttcacgttctcaacctgttaagcggtcctcctctcatc
    cccatgcccccccttacctctcagggtagagttaggctaggaacccagagagctccaag
cctccatctggtgagagggtgcccacctcttgggctgcagagaggagaatttgccc
    caaggtgcgtcagagctccacccctta
gtctgcagagcttcaccaccctta
    ggtc
    tcacaaagccttgagttcatag
catttctttgagtttttcaccctggccacgagacactgcagcacccaaaggcttc
    ccagggagtagggttgccctcaagaggtcctttgggtcgtatggccacactcctgggaatgtttttcaggttagtgtcagagctccctgagcagcatgtaggtggtggtgggtgtggtggtgtgcgc
tctgtctgtgctctctctctctctgagaacccctctgagctaccacccagacact
tagagccag','gtc' ) AS Count
FROM dual;
```
Regular Expressions

CHECK Constraints

Example:

```
ALTER TABLE emp8
ADD CONSTRAINT email_addr
CHECK(REGEXP_LIKE(email,'@')) NOVALIDATE;

INSERT INTO emp8 VALUES
(500,'Christian','Patel','ChrisP2creme.com',
1234567890,'12-Jan-2004','HR_REP',2000,null,102,40);
```

Error starting at line 2 in command:
INSERT INTO emp8 VALUES
(500,'Christian','Patel',
'ChrisP2creme.com',
1234567890,'12-Jan-2004','HR_REP',2000,null,102,40)

Error report:
```
SQL Error: 014-02290: check constraint (HR.EMAIL_ADDR) violated
02290.00000 - "Check constraint (%s) violated"
```

*Cause: The values being inserted do not satisfy the named check

*Action: do not insert values that violate the constraint.
You may need to find a specific subpattern that identifies a protein needed for immunity in the mouse DNA.

```sql
SELECT REGEXP_INSTR('ccacctttcctccactcctacgttctcacctgtaaagcgtccctcctcctctcctgccatccccatgcccttcctaccccttcaggttagagttagctagagaaccagagagatacctccacatccccatgcccccttaccctgcagggtagagtaggctagaaaccagagagctccaaagctccatctgtggagaggtgccatccttgggctgcagagagaggagaatttgccccaaagctgcctgcagagcttcaccaccctta
gtctcacaaag
ccttgagttcatagcatttcttgagttttcaccctgccagcaggacactgcagcacccaaagggcttcccaggagtagggttgccctcaagaggctttggtgtcgtatggcaccatctctggaatggttttcaaggttgggtgtgcacaca
gccctgagcatgtaggggcttggggtctgtcgctctctctctctctctctctctctctcagacaaccct
taaacccctgtgctaccccaagagcacttagagccag', 'gtc(tcac)(aaag)', 1, 1, 0, 'i', 1) "Position" FROM dual;
```

```
<table>
<thead>
<tr>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>195</td>
</tr>
</tbody>
</table>
```

1 rows selected
Case Study
Scenario 3 - Using Regular Expressions

- Reposition names that are in the format "first middle last" to the format "last, first middle ".

```sql
SELECT names "names",
REGEXP_REPLACE(names, '^([^/S]+)/s([^/S]+)/s([^/S]+)$', '\3, \1 \2')
AS "names after regexp"
FROM people;
```

<table>
<thead>
<tr>
<th>names</th>
<th>names after regexp</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Quincy Adams</td>
<td>Adams, John Quincy</td>
</tr>
<tr>
<td>Harry S. Truman</td>
<td>Truman, Harry S.</td>
</tr>
<tr>
<td>John Adams</td>
<td>John Adams</td>
</tr>
<tr>
<td>John Quincy Adams</td>
<td>John Quincy Adams</td>
</tr>
<tr>
<td>John_Quincy_Adams</td>
<td>John_Quincy_Adams</td>
</tr>
</tbody>
</table>
Agenda

- Grouping and Aggregating Data Using SQL
- Hierarchical Retrieval
- Working with Regular Expressions
- Analyzing and Reporting Data Using SQL
- Pattern Matching
SQL for Analysis and Reporting

The **RANK** Function

- **Syntax:**

  ```sql
  RANK ( ) OVER ( [query_partition_clause] order_by_clause )
  ```

- **Example:**

  ```sql
  SELECT department_id, last_name, salary,
         RANK() OVER (PARTITION BY department_id
                      ORDER BY salary DESC) "Rank"
  FROM employees
WHERE department_id = 60
ORDER BY department_id, "Rank", salary;
  ```
**SQL for Analysis and Reporting**

**Difference in the RANK and DENSE_RANK Functions**

```
SELECT last_name, salary,
       RANK() OVER (PARTITION BY department_id
                    ORDER BY salary DESC) "Rank"
FROM employees
WHERE department_id = 60
ORDER BY salary DESC, "Rank" DESC;
```

DENSE_RANK function leaves no gaps in ranking sequence when there are ties

```
SELECT last_name, salary,
       DENSE_RANK() over (PARTITION BY department_id
                           ORDER BY salary DESC) "Drank"
FROM employees
WHERE department_id = 60
ORDER BY salary DESC, "Drank" DESC;
```

RANK function leaves gaps in ranking sequence when there are ties
SQL for Analysis and Reporting

The **NTILE** Function

```sql
SELECT last_name, salary, NTILE(4)
    OVER (ORDER BY salary DESC) AS quartile
FROM employees
WHERE department_id = 100
ORDER BY last_name, salary, quartile;
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>QUARTILE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chen</td>
<td>8200</td>
<td>2</td>
</tr>
<tr>
<td>Faviet</td>
<td>9000</td>
<td>1</td>
</tr>
<tr>
<td>Greenberg</td>
<td>12008</td>
<td>1</td>
</tr>
<tr>
<td>Popp</td>
<td>8900</td>
<td>4</td>
</tr>
<tr>
<td>Sciarra</td>
<td>7700</td>
<td>3</td>
</tr>
<tr>
<td>Urman</td>
<td>7800</td>
<td>2</td>
</tr>
</tbody>
</table>

6 rows selected
The `RATIO_TO_REPORT` function computes the ratio of a value to the sum of a set of values.

```
SELECT last_name, salary, 
    RATIO_TO_REPORT(salary) 
OVER () AS rr 
FROM employees 
WHERE job_id = 'PU_CLERK' 
ORDER BY last_name, salary, rr;
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>RR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baida</td>
<td>2900</td>
<td>0.2086330935</td>
</tr>
<tr>
<td>Colmenares</td>
<td>2500</td>
<td>0.1798561151</td>
</tr>
<tr>
<td>Himuro</td>
<td>2600</td>
<td>0.1870503597</td>
</tr>
<tr>
<td>Khoo</td>
<td>3100</td>
<td>0.2230215827</td>
</tr>
<tr>
<td>Tobias</td>
<td>2800</td>
<td>0.2014388489</td>
</tr>
</tbody>
</table>
The **LAG** and **LEAD** Functions

**The LAG function** computes the value of a specified expression for the current row and the row preceding it. The syntax is as follows:

```
LAG(expression, offset, default)
```

**The LEAD function** computes the value of a specified expression for the current row and the row following it. The syntax is similar to LAG:

```
LEAD(expression, offset, default)
```

### Example 1: Using LAG and LEAD Functions

**Query 1:**

```
SELECT employee_id, last_name, salary,
       LAG(salary, 1, 0) OVER (ORDER BY salary) AS sal_prev,
       salary - LAG(salary, 1, 0) OVER (ORDER BY salary) AS sal_diff
FROM   employees
WHERE job_id = 'IT_PROG';
```

**Output:**

<table>
<thead>
<tr>
<th>employee_id</th>
<th>last_name</th>
<th>salary</th>
<th>sal_prev</th>
<th>sal_diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>107 Lorentz</td>
<td></td>
<td>4200</td>
<td>0</td>
<td>4200</td>
</tr>
<tr>
<td>105 Austin</td>
<td></td>
<td>4800</td>
<td>4200</td>
<td>600</td>
</tr>
<tr>
<td>106 Pataballa</td>
<td></td>
<td>4800</td>
<td>4800</td>
<td>0</td>
</tr>
<tr>
<td>104 Ernst</td>
<td></td>
<td>6000</td>
<td>4800</td>
<td>1200</td>
</tr>
<tr>
<td>103 Humaid</td>
<td></td>
<td>9000</td>
<td>6000</td>
<td>3000</td>
</tr>
</tbody>
</table>

**Example 2:**

**Query 2:**

```
SELECT employee_id, last_name, salary,
       LEAD(salary, 1, 0) OVER (ORDER BY salary) AS sal_next,
       salary - LEAD(salary, 1, 0) OVER (ORDER BY salary) AS sal_diff
FROM   employees
WHERE job_id = 'IT_PROG';
```

**Output:**

<table>
<thead>
<tr>
<th>employee_id</th>
<th>last_name</th>
<th>salary</th>
<th>sal_next</th>
<th>sal_diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>107 Lorentz</td>
<td></td>
<td>4200</td>
<td>600</td>
<td>-600</td>
</tr>
<tr>
<td>105 Austin</td>
<td></td>
<td>4800</td>
<td>0</td>
<td>-4800</td>
</tr>
<tr>
<td>106 Pataballa</td>
<td></td>
<td>4800</td>
<td>1200</td>
<td>-4600</td>
</tr>
<tr>
<td>104 Ernst</td>
<td></td>
<td>6000</td>
<td>3000</td>
<td>-3000</td>
</tr>
<tr>
<td>103 Humaid</td>
<td></td>
<td>9000</td>
<td>0</td>
<td>-9000</td>
</tr>
</tbody>
</table>
Case Study
Scenario 4 - Creating Statistical Report

- Help the ITDB folks find the top-selling channels in US, in the last quarter of the year 2000.

<table>
<thead>
<tr>
<th>CHANNEL_DESC</th>
<th>SALES$</th>
<th>RANKING</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Sales</td>
<td>1,851,511</td>
<td>1</td>
</tr>
<tr>
<td>Partners</td>
<td>1,147,564</td>
<td>2</td>
</tr>
<tr>
<td>Internet</td>
<td>511,124</td>
<td>3</td>
</tr>
</tbody>
</table>

Output →
Code:

```sql
SELECT channel_desc, SUM(amount_sold) SALES$, 
    RANK() OVER (ORDER BY SUM(amount_sold) desc) AS ranking
FROM sales, products, customers, times, channels, countries
WHERE sales.prod_id=products.prod_id
    AND sales.cust_id=customers.cust_id
    AND customers.country_id = countries.country_id
    AND sales.time_id=times.time_id
    AND sales.channel_id=channels.channel_id
    AND times.calendar_month_desc IN ('2000-10', '2000-11', '2000-12')
    AND country_iso_code='US'
GROUP BY channel_desc;
```
Agenda

- Grouping and Aggregating Data Using SQL
- Hierarchical Retrieval
- Working with Regular Expressions
- Analyzing and Reporting Data Using SQL
- Pattern Matching
Pattern Matching using SQL: 12c Feature

Why Pattern Matching??

- Identifies price patterns, such as V-shapes and W-shapes in stock charts, along with performing many types of calculations.
- Has the ability to recognize patterns found across multiple rows.
- Is essential for many kinds of work:
  - In security applications to detect unusual behavior.
  - In financial applications to seek patterns of pricing, trading volume, and other behavior.
Pattern Matching using SQL

The **MATCH_RECOGNIZE** Clause

- Pattern matching in SQL is performed using the **MATCH_RECOGNIZE** clause.
- The **MATCH_RECOGNIZE** clause performs the following tasks:
  - Logically partition and order the data
  - Define patterns of rows to seek
  - Specify the logical conditions to map rows to a row patterns
  - Define expressions usable in other parts of the SQL query
Pattern Matching

Keywords in the \texttt{MATCH\_RECOGNIZE} clause

- \texttt{PARTITION BY}: Logically divides rows into groups
- \texttt{MEASURES}: Defines calculations for export from the pattern matching
- \texttt{[ONE ROW | ALL ROWS] PER MATCH}: For each row in the match, displays one output row or all output rows
- \texttt{AFTER MATCH SKIP}: Restarts the matching process after a match is found
- \texttt{PATTERN}: Defines the row pattern that will be matched
- \texttt{DEFINE}: Defines primary pattern variables
SQL Pattern Matching

Example: Find Double Bottom (W) Pattern

Find double bottom (W) patterns and report:

- The beginning and end date of the pattern
- Patterns that lasted less than a week
SQL Pattern Matching

Example: Find Double Bottom (W)

Find double bottom (W) patterns and report:

• The beginning and end date of the pattern
• Patterns that lasted less than a week
SQL Pattern Matching

Example: Find Double Bottom (W)

Find double bottom (W) patterns and report:

- Beginning and ending date of the pattern
- Modify the search to find only patterns that lasted less than a week

PATTERN (X+ Y+ W+ Z+)
DEFINE X AS (price < PREV(price))
SQL Pattern Matching

Example: Find Double Bottom (W)

Find double bottom (W) patterns and report:

- Beginning and ending date of the pattern
- Modify the search to find only patterns that lasted less than a week

```sql
PATTERN (X+ Y+ W+ Z+)
DEFINE X AS (price < PREV(price))
     Y AS (price > PREV(price))
```
SQL Pattern Matching

Example: Find Double Bottom (W)

Find double bottom (W) patterns and report:

- Beginning and ending date of the pattern
- Modify the search to find only patterns that lasted less than a week

```
SELECT first_x, last_z
FROM ticker MATCH_RECOGNIZE (  
    PARTITION BY name ORDER BY time  
    MEASURES FIRST(x.time) AS first_x  
    LAST(z.time) AS last_z  
    ONE ROW PER MATCH  
    PATTERN (X+ Y+ W+ Z+)  
    DEFINE X AS (price < PREV(price))  
    Y AS (price > PREV(price))  
    W AS (price < PREV(price))  
    Z AS (price > PREV(price))
```
SQL Pattern Matching

Example: Find Double Bottom (W)

Find double bottom (W) patterns and report:

- Beginning and ending date of the pattern
- Modify the search to find only patterns that lasted less than a week

Example: Find Double Bottom (W)

```
SELECT first_x, last_z
FROM ticker MATCH_RECOGNIZE (PARTITION BY name ORDER BY time
    MEASURES FIRST(x.time) AS first_x,
        LAST(z.time) AS last_z
    ONE ROW PER MATCH
    PATTERN (X+ Y+ W+ Z+)
    DEFINE X AS (price < PREV(price)),
        Y AS (price > PREV(price)),
        W AS (price < PREV(price)),
        Z AS (price > PREV(price)))
```
SQL Pattern Matching

Example: Find Double Bottom (W)

Find double bottom (W) patterns and report:

- Beginning and ending date of the pattern
- Modify the search to find only patterns that lasted less than a week

SQL Code:

```sql
SELECT first_x, last_z
FROM ticker
MATCH_RECOGNIZE (
    PARTITION BY name ORDER BY time
    MEASURES FIRST(x.time) AS first_x,
                       LAST(z.time) AS last_z
    ONE ROW PER MATCH
    PATTERN (X+ Y+ W+ Z+)
    DEFINE X AS (price < PREV(price)),
                Y AS (price > PREV(price)),
                W AS (price < PREV(price)),
                Z AS (price > PREV(price)) AND
                LAST(z.time) - FIRST(x.time) <= 7)
```

Can refer to previous variables
Case Study
Scenario 5 - Suspicious Money Transfers

- Help ITDB folks detect suspicious money transfer pattern for an account
  - Three or more small amount (<2K) money transfers within 30 days
  - Subsequent large transfer (>=1M) within 10 days of last small transfer
- Report account, date of first small transfer, date of large transfer

<table>
<thead>
<tr>
<th>TIME</th>
<th>USER ID</th>
<th>EVENT</th>
<th>AMOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/1/2012</td>
<td>John</td>
<td>Deposit</td>
<td>1,000,000</td>
</tr>
<tr>
<td>1/2/2012</td>
<td>John</td>
<td>Transfer</td>
<td>1,000</td>
</tr>
<tr>
<td>1/5/2012</td>
<td>John</td>
<td>Withdrawal</td>
<td>2,000</td>
</tr>
<tr>
<td>1/10/2012</td>
<td>John</td>
<td>Transfer</td>
<td>1,500</td>
</tr>
<tr>
<td>1/20/2012</td>
<td>John</td>
<td>Transfer</td>
<td>1,200</td>
</tr>
<tr>
<td>1/25/2012</td>
<td>John</td>
<td>Deposit</td>
<td>1,200,000</td>
</tr>
<tr>
<td>1/27/2012</td>
<td>John</td>
<td>Transfer</td>
<td>1,000,000</td>
</tr>
<tr>
<td>2/2/2012</td>
<td>John</td>
<td>Deposit</td>
<td>500,000</td>
</tr>
</tbody>
</table>

Three small transfers within 30 days

Large transfer within 10 days of last small transfer
SELECT userid, first_t, last_t, amount
FROM (SELECT * FROM event_log WHERE event = 'transfer')
MATCH_RECOGNIZE
(  PARTITION BY userid ORDER BY time
  MEASURES FIRST(x.time) first_t, y.time last_t, y.amount amount
  PATTERN ( x{3}, Y )
  DEFINE X as (event='transfer' AND amount < 2000),
  Y as (event='transfer' AND amount >= 1000000 AND
  last(X.time) - first(X.time) < 30 AND
  Y.time - last(X.time) < 10 )
)

Three or more transfers of small amount
Within 30 days of each other
Followed by a large transfer
Within 10 days of last small
Thank You