Subqueries with Joins and Advanced Aggregate Functions

Agenda

- Subqueries and Joins
- Advanced Aggregate Function
  - Rank()
  - Dense_Rank()
  - Percent_rank()
  - LAG() & LEAD() Function
  - First_Value()
  - Last_Value()
  - NTILE()
  - CUME_DIST()
- Recursive Query Expression

Subqueries and Joins

- Subquery and JOIN perform the execution of queries similarly and retrieves the same output but varies with below features

Subqueries and Joins

- Subqueries separate the complex logic from the main query whereas JOINs enclose all the logic in the main query
- Subqueries can be used along with JOINS in many ways and will see with examples
- Calculations can be done in the subquery and return as single value whereas JOIN query performs the calculation in main SELECT query
- Priority of filtering the records in a table is possible via subquery, whereas filtering the records is managed automatically by JOIN query

Subqueries and Joins

- The resultant output of the SubQuery and JOIN is same

Subquery

```
Select Acct_Num, Tran_Amount
FROM  Transaction
WHERE Tran_Date in
(Select Event_dt
 FROM Message
 WHERE Event_dt between
'2020-04-01' and '2020-06-01' );
```

JOIN Query

```
Select Acct_Num, Tran_Amount
FROM Transaction, Message
WHERE Tran_Date = Event_dt AND
Event_dt between '2020-04-01' and
'2020-06-01'
```

Acct_Num	Tran_Amount
4000-1956-2001	-3000.00
4000-1956-5102	-6500.00
5000-1700-9911	2000.00

Row-valued Expression

Row-valued Expression

- Row values are specified together with a series of column values
- The Row values are expressed in several ways
  - A list of scalar/column values separated by comma and parenthesized
  - Retrieving a set of two or more column results from database

Row-valued Expression

- These Row values are used in many contexts like:
  - In DML statements like INSERT
  - In comparison expressions of WHERE clause
  - Refer subquery results in another query using comparative operators like

<, <=, >, >=, =, <>, IS, IS NOT, IN, NOT IN, BETWEEN,.

Row-valued Expression

- Row value expression in an *INSERT* statement:
  - Here, multiple rows can be inserted using one single *INSERT* statement
  - Whereas other RDBMS still expects to *INSERT* statement for each row

```
INSERT INTO CUSTOMER VALUES
(123009,  'Renee', '3305-1, San-Fran',  'SFO',  1677617700 ),
(123010,  'Holly', '3225-2, Concord',   'NJ' ,  1673547700 );
```

Row-valued Expression

- Row value expression in an equality comparison
    - Here multiple columns (cust_id, Address) are assigned with its corresponding values respectively, in order to select rows from the CUSTOMER table

```
SELECT * FROM CUSTOMER WHERE (Cust_Id, Address) = (123010
,'3225-2, Concord')
```

Row-valued Expression

- Row value expression in a subquery referred by Select or update statements:
  - The main query of Account table is using multiple columns to search its corresponding rows in the subquery using IN operator

```
SELECT *
FROM ACCOUNT
WHERE (ACCT_NUM, BALANCE ) IN (SELECT ACCT_NUM,
TRAN_AMOUNT
                                FROM TRANSACTION )
```

Data type like integer, character & date format should match when the columns of main Query are compared with subquery used in an IN operator

- *Rules for row - valued expression using INSERT statement:*
  - *NULLS are accepted in columns of VALUES expression*
  - *Columns in VALUES clause will not support default values which were defined at table level*
  - *When an INSERT statement is need for inserting huge number of rows, it is advised to use import methods from a file to table*

- *Rules for row - valued expression using IN operator and JOIN clauses*
    - *Indexes of single columns of row valued expressions become unused when these are represented in JOINS*

Query Expressions

Query Expression

- Query expression defines search criteria of a pattern from a string of characters.
- These expressions can be used in WHERE clause conditions to filter the rows based on matching patterns.
- Analytics uses these expressions to retrieve the text at granular level. So that these expressions identifies the partial text entries.
- Sometimes the expressions detect the patterns based on vowel sounds. So that users can identify the words with different meaning but same pronunciation.

Query Expression

- LIKE clause identifies the rows based on a partition of pattern matching

```
SELECT * FROM CUSTOMER WHERE Name
LIKE 'Ja%';
```

Cust_Id	Name	Address	State	Phone
123004	Jack	229-5, Concord	CA	1897627999
123005	Jacob	325-7, Mission Dist	SFO	1897637000

- Wild character '%' is used along with LIKE clause to retrieve all records with customer Name having prefix 'Ja……'

Query Expression

- SOUNDEX() - Function used to identify the word with different meaning but same pronunciation

```
SELECT SOUNDEX('JACOB') , SOUNDEX(
'JAKOB');
```

soundex('JACOB')	SOUNDEX('JAKOB')
J100	J100

- It best helps analytics to understand people's native language influence on few words

Query Expression

- INSTR() - Function used to identify the position of a small portion of pattern in a complete string
  - E.g : `SELECT INSTR('Enterprise Data Analytics' , 'Data')` Output: 12

- REPLACE() - Replace Data with Functional in below string
  - E.g : `SELECT REPLACE('Enterprise Data Analytics', 'Data', 'Functional')`

    Output: Enterprise Functional Analytics

Query Expression

- CAST() - Function is used to change one data type to other data type

E.g: Change type of Char to Date

```
SELECT CAST('2003-01-01' AS DATE )
```

Here the date is in character format and CAST() function is used to change the data type to date which is system understandable format

Query Expression

- CONVERT() - This function also works similar to CAST , while CAST is ANSI function, CONVERT is used in commercial RDBMS E.g: Change type of Char to Date

```
SELECT CONVERT ('2003-01-01' , DATE )
```

Advanced Aggregate Functions

Advanced Aggregate Functions

- Many RDBMS provides Analytical functions to perform complex operations and evaluate the results efficiently
- Analytical functions reduces the use of JOINS as these perform many self join operations on same database table
- Analytical functions are otherwise called as *windowing* or Online Analytical Processing (OLAP) functions

Advanced Aggregate Functions

- The Window/ Analytical function uses OVER() clause to calculate aggregate results on group of rows based on candidate key
- The aggregate result thus produced from group of rows is again shared for each row in the group
- This is an advanced feature of GROUP BY clause by sharing aggregate result at row level

Advanced Aggregate Functions - Benefits

- Reporting shows the comparison between current record entry with aggregate result
- Build statistics on the cumulative results rather than aggregate results
- More granular level of cost controlling in Financial organization whereas strategic reports are usually generated by GROUP by clauses as they show overall financial performance

Window Functions - Ranking

- The ranking functions assign a rank for each row in an ordered group of rows
- The rank is assigned to rows in a sequential manner
- The assignment of rank to each of the rows always start with 1 for every new partition
- There are 3 types of ranking functions supported in MySQL-
  - rank()
  - dense_rank()
  - percent_rank()

Rank() Function

- Displays the rank for each record based on highest value of a desired column by calculating on group of rows divided by corresponding candidate key

```
SELECT Cust_Id, Acct_Num, Acct_Type, Balance,
RANK() OVER( PARTITION BY Cust_Id ORDER BY Balance desc ) AS Rank_of_balance_amount
FROM   ACCOUNT
WHERE  Cust_Id in (  123001 , 123002 )
```

Cust_Id	Acct_Num	Acct_Type	Balance	Rank_of_balance_amount
123001	5000-1700-3456	FIXED DEPOSITS	9400000	1
123001	4000-1956-3456	SAVINGS	200000	2
123002	5000-1700-5001	FIXED DEPOSITS	7500000	1
123002	4000-1956-2001	SAVINGS	400000	2

Rank() Function

- In this example, Over() clause groups the rows based on candidate key: CUST_ID
  For each group of rows, "order by" clause sorts the column: Balance in a descending order

- Rank() is a open-closed bracket function that gives the rank for all balances maintained by customer in different accounts

Rank() Function

- Display similar rank for equal values in a column

```
# Insert a record for this example

INSERT INTO ACCOUNT VALUES
(123004, '5000-1899-6092','FIXED DEPOSITS'    , 7500000    , 'ACTIVE' , 'S' ) ;

# Example :

SELECT Cust_Id, Acct_Num, Acct_type, Balance,
RANK() OVER(PARTITION BY Cust_Id  ORDER BY Balance desc ) AS Rank_of_balance_amount
FROM  ACCOUNT
WHERE Cust_Id in (  123004 )
```

Rank() Function

Output:

Cust_Id	Acct_Num	Acct_type	Balance	Rank_of_balance_amount
123004	5000-1700-6091	FIXED DEPOSITS	7500000	1
123004	5000-1899-6092	FIXED DEPOSITS	7500000	1
123004	4000-1956-3401	SAVINGS	655000	3

- For Cust_Id: 123004 , there are equal balance amounts for two of its FIXED DEPOSIT accounts in the descending order
- Hence the *same rank* - (1) is assigned for both the FIXED DEPOSIT accounts, but *skipped* the rank -(2). and assigns rank - (3) for the next balance in descending order

Rank() function will keep skipping the subsequent ranks based on the count of similar column values

No. of Ranks skipped = No. of gaps between similar column values

Dense_Rank() Function

- Dense_rank displays the rank based on highest value of a desired column, but it preserves the rank for next following record without skipping

```
SELECT Cust_Id, Acct_Num, Acct_type, Balance,
DENSE_RANK() OVER (PARTITIoN BY Cust_Id  ORDER BY Balance desc) AS
Dense_rank_of_balance
FROM  ACCOUNT
WHERE Cust_Id =  123004
```

Dense_Rank() Function

Output:

Cust_Id	Acct_Num	Acct_type	Balance	Dense_rank_of_balance
123004	5000-1700-6091	FIXED DEPOSITS	7500000	1
123004	5000-1899-6092	FIXED DEPOSITS	7500000	1
123004	4000-1956-3401	SAVINGS	655000	2

- In this example, Dense_Rank() function assigns the same rank - (1) when the balance : 7500000 is identified same for two different accounts. But it preserved the rank - (2) without skipping it
- So, the Dense_rank() will not skip any rank

Percent_Rank() Function

- While the partitioned rows are in ascending order, the percent_rank () calculates the percentage of rank basis on formula: (*rank* - 1) / (*rows* - 1)
- Eventually, all of the rows in a partition shares a range of fraction from 0 - 1

```
#Insert record
INSERT INTO ACCOUNT VALUES
( 123004 ,'5000-8800-9977', 'FIXED DEPOSITS', 755000, 'ACTIVE',
'S' )
#Example
SELECT Cust_Id, Acct_Num, acct_type, Balance, PERCENT_RANK()OVER (
PARTITION BY Cust_Id  ORDER BY balance) AS Percent_rank_of_balance
FROM  ACCOUNT
WHERE Cust_Id  =  123004
```

Percent_Rank() Function

Output:

Cust_Id	Acct_Num	acct_type	Balance	Percent_rank_of_balance
123004	4000-1956-3401	SAVINGS	655000	0
123004	5000-8800-9977	FIXED DEPOSITS	755000	0.3333333333333333
123004	5000-1700-6091	FIXED DEPOSITS	7500000	0.6666666666666666
123004	5000-1899-6092	FIXED DEPOSITS	7500000	0.6666666666666666

Percent_Rank() Function

- Percent_rank() also skips the rank percentage when identified with duplicate values

```
#Insert record
INSERT INTO  Account VALUES(123004,'6000-3300-9222', 'FIXED
DEPOSITS', 9025300, 'ACTIVE', 'S')

#Example:
SELECT Cust_Id, Acct_Num, acct_type , balance , percent_rank() OVER(
Partition by Cust_Id  order by balance ) AS Percent_rank_of_balance
FROM  Account
WHERE Cust_Id  =  123004
```

Percent_Rank() Function

Output:

Cust_Id	Acct_Num	acct_type	balance	Percent_rank_of_balance
123004	4000-1956-3401	SAVINGS	655000	0
123004	5000-8800-9977	FIXED DEPOSITS	755000	0.25
123004	5000-1700-6091	FIXED DEPOSITS	7500000	0.5
123004	5000-1899-6092	FIXED DEPOSITS	7500000	0.5
123004	6000-3300-9222	FIXED DEPOSITS	9025300	1

LAG() and LEAD() Function

- In a normal select query , all records are interpreted serially; Sometimes there is a need to look back and forth while you are retrieving the current record in SELECT query

```
SELECT Cust_Id, Acct_Num, Acct_Type , Balance ,
LAG (Balance)  OVER(ORDER BY Balance)  previous_balance,
LEAD(Balance)  OVER (ORDER BY Balance)  next_balance
FROM  Account
WHERE Cust_Id  =  123004
```

LAG() and LEAD() Function

Output:

Cust_Id	Acct_Num	Acct_Type	Balance	previous_balance	next_balance
123004	4000-1956-3401	SAVINGS	655000	NULL	755000
123004	5000-8800-9977	FIXED DEPOSITS	755000	655000	7500000
123004	5000-1700-6091	FIXED DEPOSITS	7500000	755000	7500000
123004	5000-1899-6092	FIXED DEPOSITS	7500000	7500000	9025300
123004	6000-3300-9222	FIXED DEPOSITS	9025300	7500000	NULL

- In this example, the current record shows its previous record balance and its next record balance
- 'previous _balance' is Null if no records exist. Similarly, next_balance is Null if there are no further records

FIRST_VALUE() using order by

FIRST_VALUE () function analyzes the results of analytical expression which is defined as OVER(), and then returns the first value from the ordered set of rows.
Here, OVER() expression is defined with Order clause only.
E.g :

```
Select Cust_Id, Acct_Num, Acct_Type, Balance original_balance,
     FIRST_VALUE (Balance)  OVER( order by Balance )   Least_balance
FROM  Account
where acct_type = 'FIXED DEPOSITS'
```

Cust_Id	Acct_Num	Acct_Type	original_balance	Least_balance
123007	4000-1956-9977	FIXED DEPOSITS	7025000	7025000
123002	5000-1700-5001	FIXED DEPOSITS	7500000	7025000
123004	5000-1700-6091	FIXED DEPOSITS	7500000	7025000
123001	5000-1700-3456	FIXED DEPOSITS	9400000	7025000

In this example, the original balance of the ACCOUNT table is placed in an order in OVER() expression. Later , the first_Value () has chosen the least balance out of all FIXED DEPOSITS, and displayed it across all records in total output.

FIRST_VALUE() using Partition and order clauses

FIRST_VALUE () function analyzes the results of analytical expression which is defined as OVER(), and then returns the first value from the ordered set of rows.
Here, OVER() expression is defined with partition by and Order by clauses

E.g :

```
Select Acct_Num, Acct_Type, Balance original_balance,
     FIRST_VALUE (Balance)
     OVER( partition by Acct_type order by balance)   Least_balance
FROM  Account
where balance > 0 ;
```

Acct_Num	Acct_Type	original_balance	Least_balance
4000-1956-9977	FIXED DEPOSITS	7025000	7025000
5000-1700-5001	FIXED DEPOSITS	7500000	7025000
5000-1700-6091	FIXED DEPOSITS	7500000	7025000
5000-1700-3456	FIXED DEPOSITS	9400000	7025000
5000-1700-9911	SAVINGS	2000	2000
4000-1956-3456	SAVINGS	200000	2000
4000-1956-5102	SAVINGS	300000	2000
5000-1700-9800	SAVINGS	355000	2000
4000-1956-2001	SAVINGS	400000	2000

FIRST_VALUE() using Partition and order clauses

In this example,

- Initially, all of the table rows are partitioned into FIXED DEPOSITS and SAVINGS using ACCT_TYPE column.

- Secondly, the two partitioned rows that are FIXED DEPOSITS and SAVINGS are ordered separately based on their respective balance values.

- Finally, the first_Value () is applied on the individual partitioned rows and chooses the least balance from each of partition and displays it across the partitioned rows.

LAST_VALUE() using Range of values in a row order

FIRST_VALUE () function analyzes the results of analytical expression which is defined as OVER(), and then returns the last value from an ordered set of rows.

Here OVER() expression is defined with Order clause and additionally it uses a range of values.

```
E.g:
Select Acct_Num,
       Acct_Type, Balance AS original_balance,
       LAST_VALUE(Balance)
       OVER( order by Balance
                    RANGE BETWEEN
                        UNBOUNDED PRECEDING AND
                        UNBOUNDED FOLLOWING ) AS
last_original_value
FROM  ACCOUNT
WHERE Acct_type = 'FIXED DEPOSITS';
```

Acct_Num	Acct_Type	original_balance	last_original_value
4000-1956-9977	FIXED DEPOSITS	7025000	9400000
5000-1700-5001	FIXED DEPOSITS	7500000	9400000
5000-1700-6091	FIXED DEPOSITS	7500000	9400000
5000-1700-3456	FIXED DEPOSITS	9400000	9400000

LAST_VALUE() using Range of values in a row order

In this example,

- Initially, all of the table rows are ordered by using Balance column values.

- Secondly, RANGE between unbounded PRECEDING and FOLLOWING is used to define the range of values that are returned in an ordered set of rows.

- Finally, the last_Value () choses the last value from the range in an order set of rows and then assigns the last value across each record in the total output.

LAST_VALUE() using partition and Range order

FIRST_VALUE () function analyzes the results of analytical expression which is defined as OVER(), and then returns the last value from an ordered set of rows.

Here OVER() expression is defined with Order clause and additionally it uses a range of values.

```
E.g:
Select Acct_Num,
       Acct_Type, Balance AS original_balance,
       LAST_VALUE(Balance)
       OVER( Partition by ACCT_TYPE order by Balance
                    RANGE BETWEEN
                        UNBOUNDED PRECEDING AND
                        UNBOUNDED FOLLOWING ) AS part_last_value
FROM  ACCOUNT
WHERE balance > 0
```

LAST_VALUE() using partition and Range order

In this example,

- Initially, all of the table rows are initially partitioned by ACCT_TYPE and then ordered by using Balance column values.

- Secondly, RANGE between unbounded PRECEDING and FOLLOWING is used to define the range of values that are returned in an ordered set of rows.

- Finally, the last_Value () choses the last value from the range in an order set of rows from each partitions: FIXED DEPOSITS and SAVINGS, and then displays the last value across each record in the partition.

Acct_Num	Acct_Type	original_balance	part_last_value
4000-1956-9977	FIXED DEPOSITS	7025000	9400000
5000-1700-5001	FIXED DEPOSITS	7500000	9400000
5000-1700-6091	FIXED DEPOSITS	7500000	9400000
5000-1700-3456	FIXED DEPOSITS	9400000	9400000
5000-1700-9911	SAVINGS	2000	750000
4000-1956-3456	SAVINGS	200000	750000
4000-1956-5102	SAVINGS	300000	750000
5000-1700-9800	SAVINGS	355000	750000
4000-1956-2001	SAVINGS	400000	750000
4000-1956-5698	SAVINGS	455000	750000
4000-1956-3401	SAVINGS	655000	750000
4000-1956-2900	SAVINGS	750000	750000

NTILE() categorize the records into buckets

NTILE () function analyzes the results of analytical expression which is defined as OVER().
NTILE () splits the total records into predefined number of buckets.

```
E.g:
Select Acct_Num,
       Acct_Type, Balance AS original_balance,
       NTILE(3)
       OVER( order by Balance
             ) AS sav_bucket
FROM   ACCOUNT
WHERE Acct_type = 'SAVINGS' and balance > 0 ;
```

Acct_Num	Acct_Type	original_balance	sav_bucket
5000-1700-9911	SAVINGS	2000	1
4000-1956-3456	SAVINGS	200000	1
4000-1956-5102	SAVINGS	300000	1
5000-1700-9800	SAVINGS	355000	2
4000-1956-2001	SAVINGS	400000	2
4000-1956-5698	SAVINGS	455000	2
4000-1956-3401	SAVINGS	655000	3
4000-1956-2900	SAVINGS	750000	3

Here,
NTILE is defined with “3” by the developer.
The last bucket - 3 is created with remaining left over records after all the rows are divided equally by Ntile value -3 .

NTILE() with partitioning clause

NTILE () splits the total records into predefined number of buckets within each sorted partition results in OVER() expression.

```
E.g:
Select Acct_Num,
       Acct_Type, Balance AS original_balance,
        NTILE(2)
        OVER( partition by ACCT_TYPE
              order by Balance
              ) AS bucket
FROM   ACCOUNT
WHERE balance > 0 ;
```

Acct_Num	Acct_Type	original_balance	bucket
4000-1956-9977	FIXED DEPOSITS	7025000	1
5000-1700-5001	FIXED DEPOSITS	7500000	1
5000-1700-6091	FIXED DEPOSITS	7500000	2
5000-1700-3456	FIXED DEPOSITS	9400000	2
5000-1700-9911	SAVINGS	2000	1
4000-1956-3456	SAVINGS	200000	1
4000-1956-5102	SAVINGS	300000	1
5000-1700-9800	SAVINGS	355000	1
4000-1956-2001	SAVINGS	400000	2
4000-1956-5698	SAVINGS	455000	2
4000-1956-3401	SAVINGS	655000	2
4000-1956-2900	SAVINGS	750000	2

Here, the buckets are created within the partitioned rows: FIXED DEPOSITS and SAVINGS

CUME_DIST() - Cumulative distribution

Distribution of records means - the percentage of a record occupied in the total record set.
Cumulative distribution means , the cumulative percentage of records from first to current row is calculated out of total result.

E.g:

```
Select Acct_Num,
       Acct_Type,
       Balance AS original_balance,
   CUME_DIST()
   OVER( order by Balance ) AS cum_distribution
FROM  ACCOUNT
WHERE acct_type = 'FIXED DEPOSITS';
```

Here, the first record has occupied 25% , and the 2nd and 3rd record values are same.
Hence it displays 75% for both the records .
Otherwise 2nd record is displayed as 50%.

Acct_Num	Acct_Type	original_balance	cum_distribution
4000-1956-9977	FIXED DEPOSITS	7025000	0.25
5000-1700-5001	FIXED DEPOSITS	7500000	0.75
5000-1700-6091	FIXED DEPOSITS	7500000	0.75
5000-1700-3456	FIXED DEPOSITS	9400000	1

CUME_DIST() - Cumulative distribution for partitions

Cumulative distribution means , the cumulative percentage of records from first to current row is calculated in a partitioned result set.

E.g:

```
Select Acct_Num,
       Acct_Type,
       Balance AS original_balance,
   CUME_DIST()
   OVER( partition by ACCT_TYPE
         order by Balance ) AS Part_Cume_dist
FROM  ACCOUNT
WHERE balance > 0;
```

Acct_Num	Acct_Type	original_balance	Part_Cume_dist
4000-1956-9977	FIXED DEPOSITS	7025000	0.25
5000-1700-5001	FIXED DEPOSITS	7500000	0.75
5000-1700-6091	FIXED DEPOSITS	7500000	0.75
5000-1700-3456	FIXED DEPOSITS	9400000	1
5000-1700-9911	SAVINGS	2000	0.125
4000-1956-3456	SAVINGS	200000	0.25
4000-1956-5102	SAVINGS	300000	0.375
5000-1700-9800	SAVINGS	355000	0.5
4000-1956-2001	SAVINGS	400000	0.625
4000-1956-5698	SAVINGS	455000	0.75
4000-1956-3401	SAVINGS	655000	0.875
4000-1956-2900	SAVINGS	750000	1

Here, cumulative distribution is calculated for each partition : FIXED DEPOSIT and SAVINGS.

Aggregate Functions with Window Functions

- Grouping functions can be used by Window functions to retrieve aggregate results and compare against each of the rows in a group

```
SELECT Cust_Id, Acct_Num, Acct_Type, Balance, SUM(Balance) OVER (
partition by Cust_Id ) AS customer_level_balance
FROM  ACCOUNT
WHERE CUST_ID in ( 123001, 123002 , 123004)
```

Aggregate Functions with Window Functions

Output:

Cust_Id	Acct_Num	Acct_Type	Balance	customer_level_balance
123001	4000-1956-3456	SAVINGS	200000	9600000
123001	5000-1700-3456	FIXED DEPOSITS	9400000	9600000
123002	4000-1956-2001	SAVINGS	400000	7900000
123002	5000-1700-5001	FIXED DEPOSITS	7500000	7900000
123004	5000-1700-6091	FIXED DEPOSITS	7500000	25435300
123004	4000-1956-3401	SAVINGS	655000	25435300
123004	5000-1899-6092	FIXED DEPOSITS	7500000	25435300
123004	5000-8800-9977	FIXED DEPOSITS	755000	25435300
123004	6000-3300-9222	FIXED DEPOSITS	9025300	25435300

- In this example, SUM is grouping function calculates aggregate sum of balance for each Cust_Id. This aggregate SUM result is displayed for each row and is useful for further analytics

Aggregate Functions with Window Functions

- Similarly other grouping functions like below can be used along with analytical functions:
  - AVG()
  - MIN()
  - MAX()

Analytical functions are widely used in organizations ,because it reduces the number of calls to the same table. Especially when there is a need for SELF Join

The performance of the Query is high because it consumes less CPU utilization for mapping of rows between tables. However, it depends on the business logic

Cumulative calculations are also performed in GROUP by clauses, but it is difficult move across the rows among the group

Cross-Tab and Relational Tables

Cross-Tab and Relational Tables

- Cross tabulation of relational database displays the columns as rows and rows as columns
- Hence cross - table is multi-dimensional structure of rows and columns which can pivot the rows vertically and columns horizontally
- This kind of multi - dimensional structured / cross table displays strategic reports with summarized data

Cross-Tab and Relational Tables

- The cross table can display grand totals for columns, rows, or for the whole measure

- It can also display subtotals for columns, or show images on the horizontal or vertical axis

Cross-Tab and Relational Tables

- Cross - tabular calculation is the form presenting the count of transactions in a tabular form
- Especially, it is used for developing trend reports with aggregate reports

```
SELECT monthname(T.Tran_Date) as "Month",
  count(case when T.Channel= 'ATM Withdrawal' THEN 1 END) ATM_transaction,
  count(case when T.Channel = 'UPI transfer' THEN 1 END) UPI_transaction,
  count(case when T.Channel = 'Net banking' THEN 1 END) net_banking,
  count(case when T.Channel = 'Bankers cheque' THEN 1 END) bankers_cheque,
  count(case when T.Channel = 'ECS transfer' THEN 1 END) ECS_transfer,
  count(case when T.Channel = 'Cash Deposit' THEN 1 END) Cash_deposit
FROM Transaction T
GROUP BY month(T.Tran_Date)
```

Cross-Tab and Relational Tables

Output:

Month	ATM_transaction	UPI_transaction	net_banking	bankers_cheque	ECS_transfer	Cash_deposit
January	1	1	0	0	1	1
February	0	0	0	0	1	0
March	1	0	0	1	0	1
April	2	0	2	0	0	1

- In this example, bank is measuring the total number of transactions done through each channel in various months

Recursive Query Expression

Recursive Query Expressions

- To report hierarchical structured data
- The recursive queries follows a chain process of identifying the relationships between the relevant columns
- The query expression is defined with a sub - query using RECURSIVE clause
- This query iterates between main query and the sub-query with predefined number of times
- There should be a terminating condition to recursive expression

Recursive Query Expressions

- Create following table for this exercise

```
CREATE TABLE CUSTOMER_HOUSEHOLD
( Cust_Id INT, NAME VARCHAR(20), PARENT_NAME
VARCHAR(20));

INSERT INTO CUSTOMER_HOUSEHOLD VALUES
(123000,  'GEFF', NULL  ),
(123001, 'MARK', 'GEFF' ),
(123002, 'CHARLIE' , 'MARK' ) ,
(123003, 'CRISTY' , 'MARK' ) ,
(123004, 'SARAH', 'GEFF' ),
(123005, 'ROBERT' , 'SARAH') ,
(123006, 'ANDY' , 'SARAH') ;
```

Generate the report showing hierarchical family relationship between the customers

Recursive Query Expressions

```
WITH RECURSIVE Family ( NAME, PARENT_NAME,  Hierarchy) AS
(SELECT Name, Parent_name, CAST(Name AS CHAR(200))
  FROM   CUSTOMER_HOUSEHOLD
  WHERE Parent_Name IS NULL
  UNION ALL
  SELECT ch.Name, ch.Parent_Name, CONCAT(cf.Hierarchy, ",", ch.name)
  FROM Family cf
  JOIN CUSTOMER_HOUSEHOLD ch
  ON cf.name = ch.parent_name)
SELECT * FROM Family ORDER BY Hierarchy;
```

Recursive Query Expressions

Output:

NAME	PARENT_NAME	Hierarchy
GEFF	NULL	GEFF
MARK	GEFF	GEFF,MARK
CHARLIE	MARK	GEFF,MARK,CHARLIE
CRISTY	MARK	GEFF,MARK,CRISTY
SARAH	GEFF	GEFF,SARAH
ANDY	SARAH	GEFF,SARAH,ANDY
ROBERT	SARAH	GEFF,SARAH,ROBERT

- WITH RECURSIVE is the clause
- The query expression is defined with a sub - query using RECURSIVE clause
- This query iterates between main query and the sub-query with predefined number of times
- There should be a terminating condition to recursive expression

Thank You