Window value| Analytics functions

- These functions return values from other rows in the same window.

  - lag()
  - lead()
  - first_value()
  - last_value()
  - nth_value()

lag():

- used to access a value from a previous row in the same window.
- Mandatory requirements:
  - To use lag() ⟶ need must ⇒ 1) over() 2) orderBy()

⟶ {without order; 'previous row' has no meaning}

① Basic lag() ex:

```
df.withColumn(
    'prev_salary',
    lag('salary').over(
        Window.orderBy('salary')
    )
)
```

emp	salary	prev_salary
D	900	NULL
A	1000	900
E	1100	1000
B	1500	1100
C	1500	1500

(first row has no previous row ⇒ NULL)

② lag() with partitionBy:

```
df.withColumn(
    'prevsalary',
    lag('salary').over(
        Window.partitionBy('dept').orderBy('salary')
    )
)
```

emp	dept	salary	prevSalary
A	IT	1000	NULL
B	IT	1500	1000
C	IT	1500	1500
D	HR	900	NULL
E	HR	1200	900

③ lag(column, offset) - jump back more rows

lag('salary', 2)

↓ gives me the salary from 2 rows before

```
df.withColumn(
    'salary_2_back',
    lag('salary', 2).over(
        Window.orderBy('salary')
    )
)
```

o/p

emp	salary	sal_2_back
D	900	NULL
A	1000	NULL
E	1100	900
B	1500	1000
C	1500	1100

---

④ lag(column, offset, default):

(handle NULLs)

lag('salary', 1, 0)

If previous row doesn't exist, return 0 instead of NULL

```
df.withColumn(
    'salary_prev',
    lag('salary', 1, 0).over(
        Window.partitionBy('dept')
              .orderBy('salary')
    )
)
```

o/p

dept	emp	salary	prev_salary
IT	A	1000	0
IT	B	1500	1000
IT	C	1500	1500
HR	D	900	0
HR	E	1100	900

⑤ most common use case :- (difference Calculation)

↓

(salary increase)

```
df.withColumn(
    'salary_diff',
    col('salary') - lag('salary').over(
                        Window.orderBy('salary')
                    )
)
```

↓

salary	salary_diff
900	NULL
1000	100
1100	100
1500	400
1500	0

Real-world production use cases :

- Day-over-day sales diff.
- month-over-month growth
- previous status comparison
- Trend detection

Note :

→ lags, leads ignore frame clauses

→ They only depend on order and partition

How does lag() / lead() behave when a frame clause is present?

↳ lag() / lead() will completely IGNORES the frame clause

↳ Even Default frame also be ignored

Why?

↓

Because lag() / lead() is NOT an aggregate function

Therefore;

Frame clause has no effect on lag() / lead()

lead()

- lead() is exactly opposite to lag()
- used to access a value from a next row in the same window.

→ To use lead(); `over()` `orderBy()` are mandatory.

→ Frame clause has no effect on lead()

① Basic lead():

```
df.withColumn(
    'next_salary',
    lead('salary').over(
        Window.orderBy('salary')
    )
)
```

salary	next_salary
900	1000
1000	1100
1100	1500
1500	1500
1500	NULL

② lead() with partitionBy :

```
df.withColumn(
    'next_sal',
    lead('salary', 2, 0).over(
        Window.partitionBy('dept')
        .orderBy('salary')
    )
)
```

(2 → offset, 0 → default)

emp	dept	salary	next_salary
A	IT	1000	1500
B	IT	1500	0
C	ST	1500	0
D	HR	900	0
E	HR	1100	0

Real world Production use cases :

- Next-day sales comparison
- Event sequence Analysis
- Time-to-next-event calculation.

first_value()

This function looks simple, but it's one of the most confusing in practice Because of the frame clause. Frame clause is implemented here

first_value() : return the FirstValue in the window frame for the current row

→ (mandatory field)

→ orderBy : decides order

frame clause : decides which rows are visible

first_value() : picks the first row from the visible set

emp_id	emp_name	dept	month	salary
101	A	IT	Jan	50000
101	A	IT	feb	52000
101	A	IT	mar	55000
102	B	IT	Jan	60000
102	B	IT	mar	65000
103	C	HR	Jan	40000
103	C	HR	feb	42000
103	C	HR	mar	45000

① first_value() with Default Frame:

```
df.withColumn(
    'starting_salary',
        first_value('salary').over(
            Window.partitionBy('emp_id')
                .orderBy('month')
        )
)
```

emp_id	month	salary	starting_salary
101	Jan	50000	50k
101	feb	52k	50k
101	mar	55k	50k
102	Jan	60k	60k
102	Mar	65k	60k
103	Jan	40k	40k
103	feb	42k	40k
103	mar	45k	40k

Default frame:

(ROWS_Between UnBounded_preceding AND current_row)

Examples

① 'case2': Full partition frame.

```
df.withColumn(
        'salary_first',
        first_value('salary').over(
            Window.partitionBy('emp_id')
                .orderBy('month')
                .rowsBetween(
                    Window.unboundedPreceding,
                    Window.unboundedFollowing
                )
        )
```

Emp_id	month	salary	salary_first
101	Jan	50k	50k
101	feb	52k	50k
101	mar	55k	50k
102	Jab	60k	60k
102	mar	65k	60k
103	Jan	40k	40k
103	feb	42k	40k
103	mar	45k	40k

Case3: Current row only

```
df.withColumn(
        'salary_start',
        first_value('salary').over(
            Window.partitionBy('emp_id')
            .orderBy('month')
            .rowsBetween(0,0)
        )
)
```

emp_id	salary	month	salary_start
101	50k	Jan	50k
101	52k	Feb	52k
101	55k	mar	55k
102	60k	Jan	60k
102	65k	mar	65k
103	40k	Jan	40k
103	42k	Feb	42k
103	45k	mar	45k

Case4: Current row → Future row

```
df.withColumn(
        'start_salary',
        first_value('salary').over(
            Window.partitionBy('emp_id')
            .orderBy('month')
            .rowsBetween(0,
                Window.unboundedFollowing)
        )
)
```

emp_id	salary	month	salary_start
101	50k	Jan	50k
101	52k	feb	52k
101	55k	mar	55k
102	60k	Jan	60k
102	65k	mar	65k
103	40k	Jan	40k
103	42k	feb	42k
103	45k	mar	45k

Case 5: Sliding window (previous + current)

```
df.withColumn(
        'salary_start',
        first_value('salary').over(
                Window.partitionBy('emp_id')
                    .orderBy('month')
                    .rowsBetween(-1, 0)
        )
)
```

emp_id	salary	month	salary_start
101	50k	Jan	50k
101	52k	feb	50k
101	55k	mar	52k
102	60k	Jan	60k
102	65k	mar	60k
103	40k	Jan	40k
103	42k	feb	40k
103	45k	mar	42k

Golden rule: first_value() returns first row in the frame

Real world production use cases:

- Data normalization
  - → compare each row against the starting value
- Baseline comparison
  - → first sale of the day
  - → opening price
  - → initial state

last_value():

Returns the last_value in the window PRIMP for the current row.

Case1: Default frame:

↓

rowsBetween(unboundedPreceding, current_row)

```
last_value('salary').over(
        Window.partitionBy('emp_id').orderBy('month')
)
```

emp_id	month	salary	last_sal
101	Jan	50k	50k
101	feb	52k	52k
101	mar	55k	55k
102	Jan	60k	60k
102	mar	65k	65k
103	Jan	40k	40k
103	feb	42k	42k
103	mar	45k	45k

Case2: Full partition frame:

```
last_value('salary').over(
  Window.partitionBy('emp_id')
        .orderBy('month')
        .rowsBetween(
          Window.unboundedPreceding,
          Window.unboundedFollowing
        )
)
```

emp_id	month	salary	last_sal
101	Jan	50k	55k
101	feb	52k	55k
101	mar	55k	55k
102	Jan	60k	65k
102	mar	65k	65k
103	Jan	40k	45k
103	feb	42k	45k
103	mar	45k	45k

Case3: Current row:

```
last_value('salary').over(
  Window.partitionBy('emp_id')
        .orderBy('month')
        .rowsBetween(0,0)
)
```

emp_id	month	salary	last_sal
101	Jan	50k	50k
101	feb	52k	52k
101	mar	55k	55k
102	Jan	60k	60k
102	mar	65k	65k
103	Jan	40k	40k
103	feb	42k	42k
103	mar	45k	45k

Case4: previous → current { sliding window }

```
last_value('salary').over(
    Window.partitionBy('emp_id')
        .orderBy('month')
        .rowsBetween(-1,0)
)
```

emp_id	month	salary	last_sal
101	Jan	50k	50k
101	Feb	52k	52k
101	mar	55k	55k
102	Jan	60k	60k
102	Mar	65k	65k
103	Jan	40k	40k
103	Feb	42k	42k
103	mar	45k	45k

Note:

`last_value()` → useless with the default frame

Real-world use cases:

- Final salary
- closing balance
- end of period price
- latest status per entity

nth_value():

- nth_value("col", n) returns the Nth value from the window frame.
- 'n' starts from 1 {i.e. from 1st value in frame}.
- It is frame-dependent (like first_value, last_value)
- If the Nth row is not visible in the frame → NULL

case1: Default frame

```
nth_value('salary', 2).over(
    Window.partitionBy('emp_id').orderBy('month')
)
```

o/p:

emp_id	month	salary	nth_value
101	jan	50k	NULL
101	feb	52k	52k
101	mar	55k	52k
102	feb	60k	NULL
102	mar	65k	65k
103	jan	40k	NULL
103	feb	42k	42k
103	mar	45k	42k

why?

> Jan → only 1 row visible → no end is
> Feb → [Jan, Feb] → end = feb
> mar → [jan, feb, mar] → end = fe

case2: Full partition frame

```
nth_value("salary", 2).over(
    Window.partitionBy('emp_id')
        .orderBy('month')
        .rowsBetween(
            Window.unboundedPreceding,
            Window.unboundedFollowing
        )
)
```

emp_id	month	salary	nth_value
101	jan	50k	52k
101	feb	52k	52k
101	mar	55k	52k
102	jan	60k	65k
102	mar	65k	65k
103	jan	40k	42k
103	feb	42k	42k
103	mar	45k	42k

Cases: N larger than available rows

```
nth_value('salary', 3).over(
        Window.partitionBy('emp_id').orderBy('month')
)
```

emp_id	month	salary	nth_value
101	Jan	50K	NULL
101	Feb	52K	NULL
101	Mar	55K	NULL
102	Jan	60K	NULL
102	Mar	65K	NULL
103	Jan	40K	NULL
103	Feb	42K	NULL
103	Mar	45K	NULL

Key rules:

① 'n' starts from 1

② Depends on frame clause

③ Default frame
= partial visibility

④ use full frame for stable results

Real world use cases

- second purchase amt.
- second salary revision
- Nth event in a lifecycle
- Milestone tracking.