MLDS-413 Introduction to Databases and Information Retrieval

Lecture 16
Window Functions
Datetime Functions

Instructor: Nikos Hardavellas

Last Lecture

- Showed how indexes are added to tables
- Explained how multiple indexes can co-exist
- Described composite indexes
- Gave guidelines for when to index columns
- Showed which columns should be indexed to speed-up some example queries

SQL difficulties with aggregates

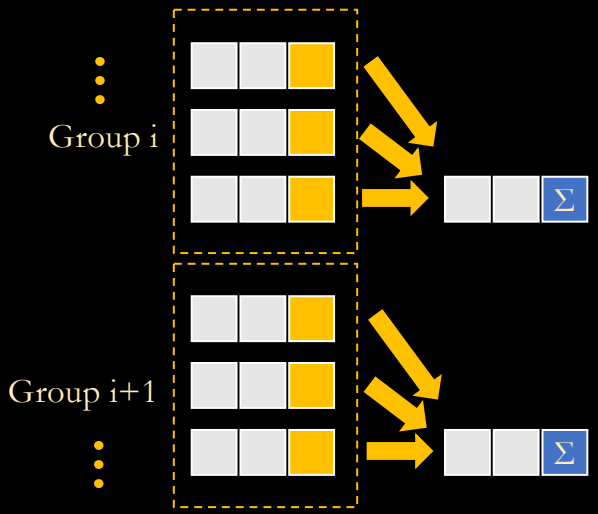
- Regular aggregation causes rows to become grouped into a single result row
- What if this is undesirable?
- Can you perform calculations for each row, based on nearby rows?
  - e.g., calculating moving average

- Window functions allow the use of a "window" of rows on calculations
  - This window is typically called a ***frame***
  - The frame can be programed as
    - a range
    - a partition
    - rows relative to the current row
    - ...

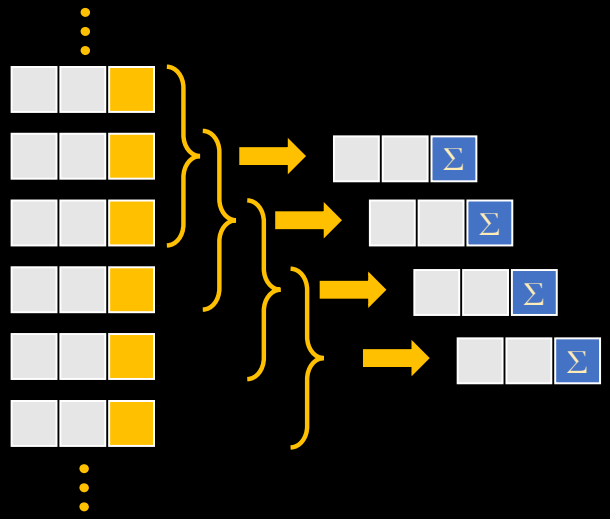
- Support for window functions was added to SQLite in 2018 (v3.25)

Window Functions

- Aggregate functions
  - Group rows into one
    - i.e., one result per group
  - Aggregate over entire group

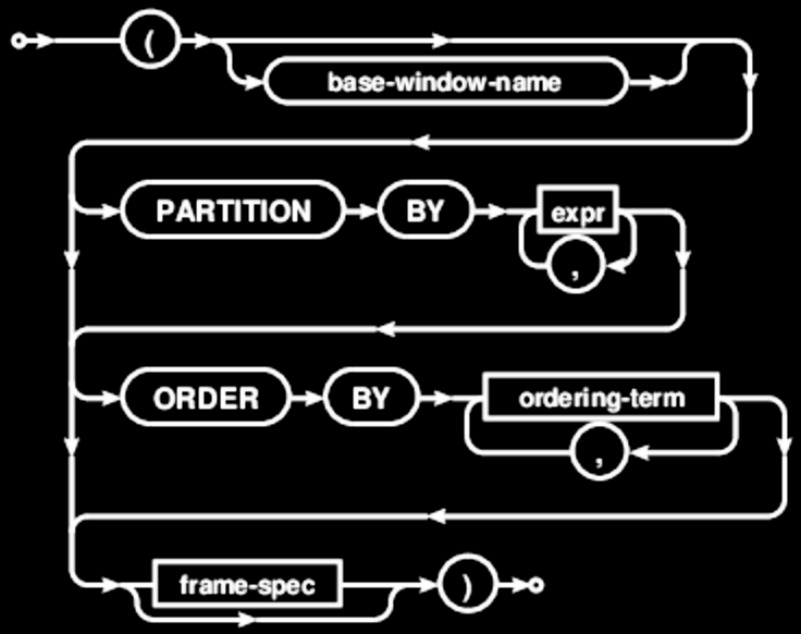


- Window functions
  - Retain row identity
    - i.e., one result per row
  - Aggregate over the frame
    - i.e., the "window" of rows



Window definitions

- Window partition (`PARTITION BY`)
  - Groups rows into partitions
  - Window-function processing is performed separately for each partition
  - Like a "*windowed group*"
- Window ordering (`ORDER BY`)
  - Defines the order or sequence of rows within each window
- Window frame (`ROWS`, `RANGE`, `GROUPS`)
  - Defines the window by use of an offset from the specified row, value or group



Window example: prepare `GROUP BY` #1

- Calculate the running total orders in `SalesOrder.sqlite` per month
  - i.e., if sales were $10, $10, $30, the running total would be $10, $20, $50

- First, let's calculate the total orders per individual month

```
SELECT OrderDate, SUM (OrderTotal)
FROM Orders
GROUP BY OrderDate;
```

	OrderDate	SUM (OrderTotal)
1	2012-09-01	52083.52
2	2012-09-02	56111.08
3	2012-09-03	34505.04
4	2012-09-04	23538.34

- ...this gives the total order per day, not per month...
- Correctness: need to convert date to month for `GROUP BY`
- Precision: it would be nice to print the last day of the month on the output

- We can do all these with **datetime functions**

Detour 1: Datetime functions: `timestring`

- A `timestring` is a character string in any of the following formats:
  - **YYYY-MM-DD HH:MM:SS.SSS[+-]HH:MM**
    - time is optional: HH:MM or HH:MM:SS or HH:MM:SS.SSS
    - time zone (following time) is also optional, or "**Z**" to indicate "Zulu" time (UTC)
    - ISO-8601 requires literal character "**T**" separating the date and the time
  - **HH:MM:SS.SSS**
    - Assume a date of 2000-01-01
    - seconds are optional: SS or SS.SSS
    - time zone is also optional
  - **now**
  - **DDDDDDDDDD**
    - Julian day, expressed as a floating-point value

Datetime functions

- `date(timestring, modifier, modifier, ...)` → YYYY-MM-DD
- **`time(timestring, modifier, modifier, ...)`** → HH:MM:SS
- `datetime(timestring, modifier, modifier, ...)` → YYYY-MM-DD HH:MM:SS
- **`julianday(timestring, modifier, modifier, ...)`** → Julian Day
  - The number of days since noon in Greenwich on November 24, 4714 B.C.
- `strftime(format, timestring, modifier, modifier, ...)` → date string
  - Date formatted according to the ***format*** string specified as the first argument:

`%d`	day of month: 00	`%s`	seconds since 1970-01-01
`%f`	fractional seconds: SS.SSS	`%S`	seconds: 00-59
`%H`	hour: 00-24	`%w`	day of week 0-6 with Sunday==0
`%j`	day of year: 001-366	`%W`	week of year: 00-53
`%J`	Julian day number	`%Y`	year: 0000-9999
`%m`	month: 01-12	`%%`	%
`%M`	minute: 00-59		

Datetime function modifiers

- The `timestring` can be followed by zero or more modifiers
- Modifiers precedence rules: transformations applied from left to write, in order
  - **[+-]NN days**
  - **[+-]NNN hours**
  - **[+-]NNN minutes**
  - **[+-]NNN.NNNN seconds**
  - **[+-]NNN months**
    - Normalizes for months with <31 days
    - e.g., 2001-03-31 '+1 month'
      → 2001-04-31; April has only 30 days
      → normalize to 2001-05-01
  - **[+-]NNN years**
    - Normalizes like above for leap years
  - **start of month**
  - **start of year**
  - **start of day**
  - **weekday N**
    - Advances date forward to desired weekday
    - Sunday = 0
  - **unixepoch**
    - Can only follow a DDDDDDDDDD timestring
    - Causes timestring to be interpreted as Unix Epoch
  - **localtime**
    - Can only follow a timestring that is UTC
    - Causes timestring to be converted to local time
  - **utc**
    - Can only follow a timestring that is local time
    - Causes timestring to be converted to UTC

Datetime function examples

- Compute the current date
```
SELECT date('now');
```
- Compute the last day of the current month
```
SELECT date('now','start of month','+1 month','-1 day');
```
- Compute the date and time given a unix timestamp 1092941466, and translate to local time
```
SELECT datetime(1092941466, 'unixepoch', 'localtime');
```
- Compute the current unix timestamp (i.e., unix epoch time)
```
SELECT strftime('%s','now');
```
- Compute the number of days since the signing of the US Declaration of Independence
```
SELECT julianday('now') - julianday('1776-07-04');
```
- Compute the date of the first Tuesday in October for the current year
```
SELECT date('now','start of year','+9 months','weekday 2');
```

- https://www.sqlitetutorial.net/sqlite-date-functions/sqlite-date-function/
- https://sqlite.org/lang_datefunc.html

Datetime function examples in PostgreSQL

- Compute the current date

```
SELECT now(); --- YYYY-MM-DD HH:MM:SS in UTC
SELECT current_date; --- YYYY-MM-DD
```

- Compute the first day of the current month

```
SELECT date_trunc('month', current_date); --- YYYY-MM-01 00:00:00-04
SELECT date_trunc('month', current_date)::DATE; --- YYYY-MM-01
```

- Compute the last day of the current month

```
SELECT (date_trunc('month', current_date)
        + interval '1 month' - interval '1 day')::DATE;
```

- Compute the date and time given a unix timestamp 1092941466, and translate to local time

```
SELECT to_timestamp(1092941466);
SELECT to_timestamp(1092941466) AT TIME ZONE 'America/Chicago';
```

- Compute the current unix timestamp (i.e., unix epoch time)

```
SELECT extract(epoch FROM now());
```

- Compute the number of days since the signing of the US Declaration of Independence

```
SELECT current_date - '1776-07-04'::DATE;
```

- https://www.postgresql.org/docs/10/functions-datetime.html

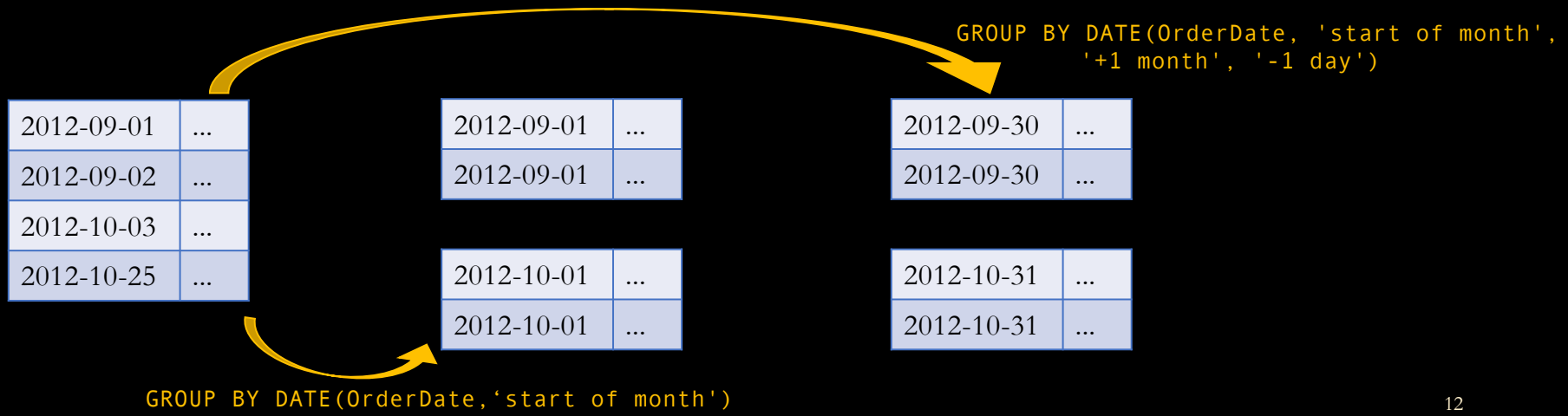
Window example: prepare `GROUP BY` #2

- Calculate the running total orders in `SalesOrder.sqlite` per month
- datetime → first day of the month

```
DATE(OrderDate, 'start of month')
```

- datetime → last day of the month

```
DATE(OrderDate, 'start of month', '+1 month', '-1 day')
```



Window example: prepare `GROUP BY` #3

- Calculate the running total orders in `SalesOrder.sqlite` per month
- datetime → first day of the month

```
DATE(OrderDate, 'start of month')
```

- datetime → last day of the month

```
DATE(OrderDate, 'start of month', '+1 month', '-1 day')
```

```
SELECT DATE(OrderDate, 'start of month', '+1 month', '-1 day') AS EndOfMonth,
       SUM (OrderTotal) AS MonthlyTotal
FROM Orders
GROUP BY DATE(OrderDate, 'start of month');
```

- This query calculates the sum of orders per month

	EndOfMonth	MonthlyTotal
1	2012-09-30	820475.89
2	2012-10-31	726899.770000001
3	2012-11-30	757151.65
4	2012-12-31	618910.4
5	2013-01-31	936516.5
6	2013-02-28	770941.16

Window example: prepare `GROUP BY` #4

- Calculate the running total orders in `SalesOrder.sqlite` per month
- First, let's turn the `GROUP BY` query into a CTE to use later:

```
WITH monthly_sales(EndOfMonth, MonthlyTotal) AS (
      SELECT DATE(OrderDate, 'start of month', '+1 month', '-1 day'),
            SUM (OrderTotal)
      FROM Orders
      GROUP BY DATE(OrderDate, 'start of month'))
SELECT *
FROM monthly_sales;
```

	EndOfMonth	MonthlyTotal
1	2012-09-30	820475.89
2	2012-10-31	726899.770000001
3	2012-11-30	757151.65
4	2012-12-31	618910.4
5	2013-01-31	936516.5
6	2013-02-28	770941.16

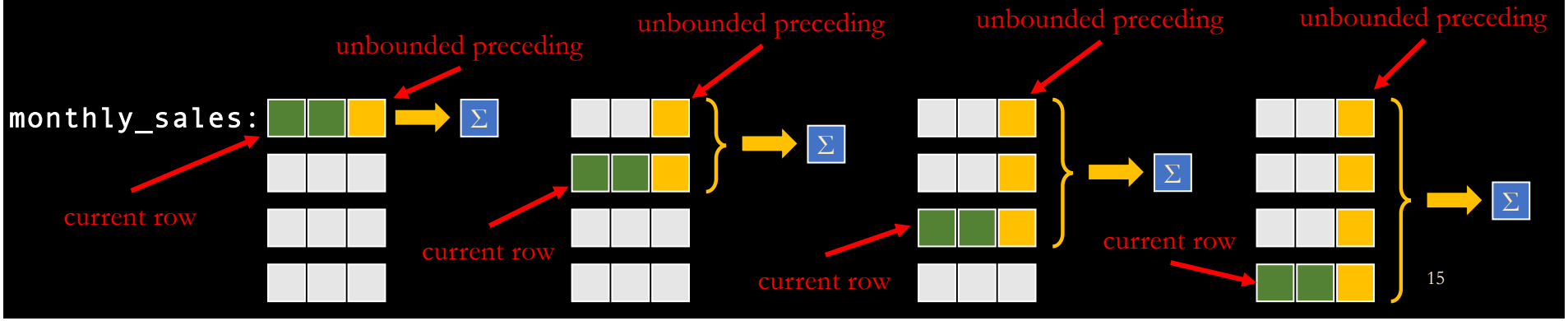
- Now we can use this to calculate the running total

Window example: running total orders per month

SalesOrder.sqlite

```
WITH monthly_sales(EndOfMonth, MonthlyTotal) AS (
       SELECT DATE(OrderDate, 'start of month', '+1 month', '-1 day'),
              SUM (OrderTotal)
       FROM Orders
       GROUP BY DATE(OrderDate, 'start of month'))
SELECT EndOfMonth,
       SUM(MonthlyTotal) OVER (ORDER BY EndOfMonth
              ROWS UNBOUNDED PRECEDING) AS RunningTotal
FROM monthly_sales;
```

	EndOfMonth	RunningTotal
1	2012-09-30	820475.89
2	2012-10-31	1547375.66
3	2012-11-30	2304527.31
4	2012-12-31	2923437.71
5	2013-01-31	3859954.21
6	2013-02-28	4630895.37



ROWS vs. RANGE vs. GROUPS

- Consider a database with rows representing item quantities
- The windowed sums below will produce different results each

```
SELECT SequenceNum as RowNumber,
       Quantity,
       SUM(Quantity) OVER (ORDER BY Quantity ROWS 3 PRECEDING) AS SumOverRows,
       SUM(Quantity) OVER (ORDER BY Quantity RANGE 3 PRECEDING) AS SumOverRange,
       SUM(Quantity) OVER (ORDER BY Quantity GROUPS 3 PRECEDING) AS SumOverGroups
FROM Items
ORDER BY SequenceNum;
```

ROWS vs. RANGE vs. GROUPS

RowNumber	Quantity	SumOverRows	SumOverRange	SumOverGroups
1	1	1	2	2
2	1	2	2	2
[illegible]	[illegible]	5	5	5
[illegible]	[illegible]	10	13	15
[illegible]	5	14	13	15
6	7	20	17	22
7	8	25	49	52
8	8	28	49	52
9	8	31	49	52
10	8	32	49	52
11	9	33	48	58
[illegible]	[illegible]	35	58	58
13	11	38	84	84
14	11	41	84	84
15	11	43	84	84
16	22	55	22	74

current row – 3 = row 6

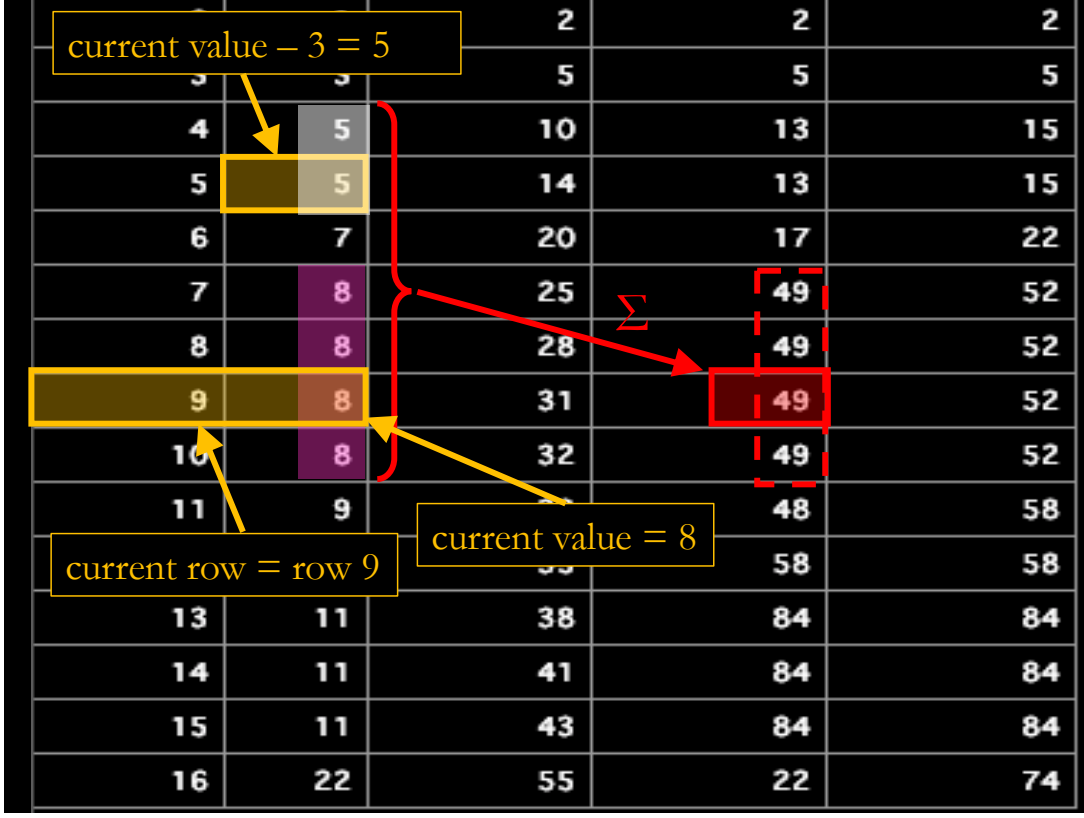
Σ

current row = row 9

```
SELECT
  SequenceNum as RowNumber,
  Quantity,

  SUM(Quantity) OVER (
    ORDER BY Quantity
    ROWS 3 PRECEDING)
      AS SumOverRows,

  SUM(Quantity) OVER (
    ORDER BY Quantity
    RANGE 3 PRECEDING)
      AS SumOverRange,

  SUM(Quantity) OVER (
    ORDER BY Quantity
    GROUPS 3 PRECEDING)
      AS SumOverGroups

FROM Items
ORDER BY SequenceNum;
```

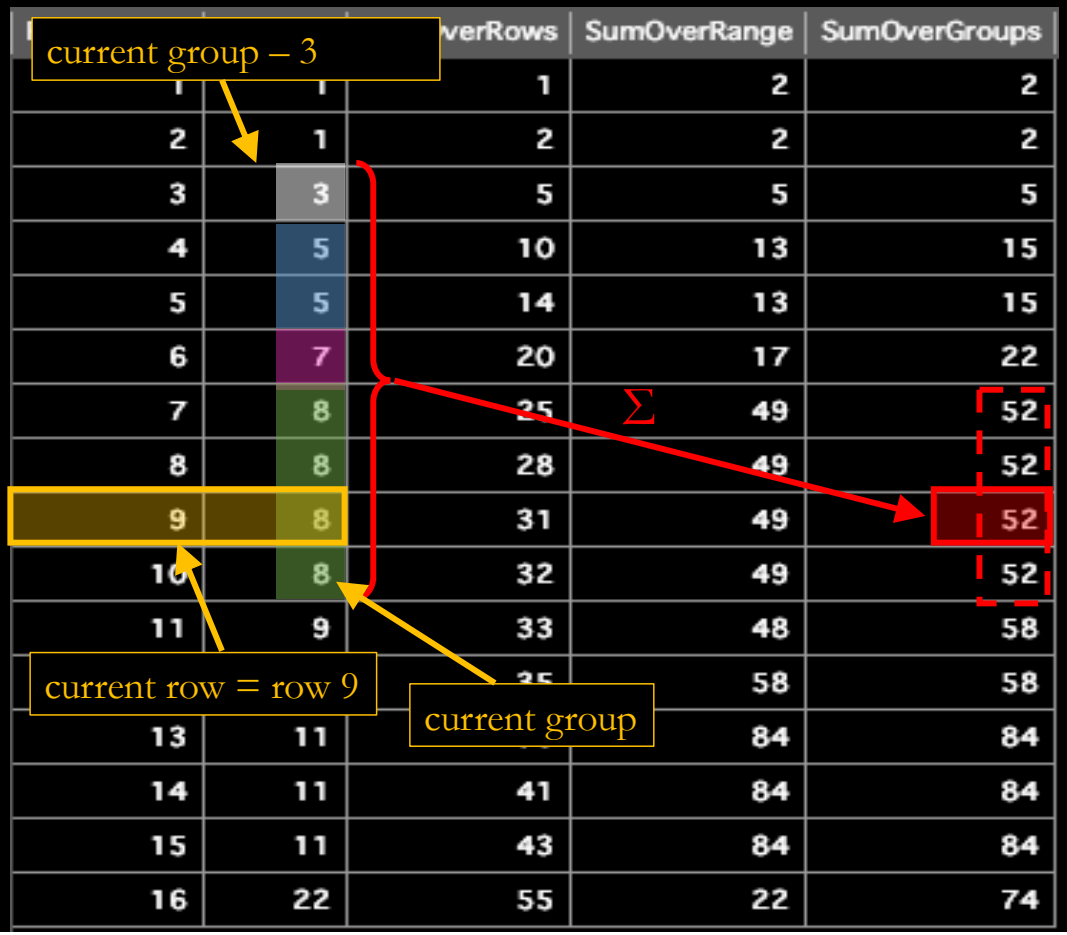
ROWS vs. RANGE vs. GROUPS

RowNumber	Quantity	SumOverRows	SumOverRange	SumOverGroups
1	1	1	2	2
		2	2	2
		5	5	5
4	5	10	13	15
5	5	14	13	15
6	7	20	17	22
7	8	25	49	52
8	8	28	49	52
9	8	31	49	52
10	8	32	49	52
11	9		48	58
			58	58
13	11	38	84	84
14	11	41	84	84
15	11	43	84	84
16	22	55	22	74




```
SELECT
  SequenceNum as RowNumber,
  Quantity,

  SUM(Quantity) OVER (
    ORDER BY Quantity
    ROWS 3 PRECEDING)
      AS SumOverRows,

  SUM(Quantity) OVER (
    ORDER BY Quantity
    RANGE 3 PRECEDING)
      AS SumOverRange,

  SUM(Quantity) OVER (
    ORDER BY Quantity
    GROUPS 3 PRECEDING)
      AS SumOverGroups

FROM Items
ORDER BY SequenceNum;
```

ROWS vs. RANGE vs. GROUPS



		SumOverRows	SumOverRange	SumOverGroups
1	1	1	2	2
2	1	2	2	2
3	3	5	5	5
4	5	10	13	15
5	5	14	13	15
6	7	20	17	22
7	8	25	49	52
8	8	28	49	52
9	8	31	49	52
10	8	32	49	52
11	9	33	48	58
12	11	35	58	58
13	11	39	84	84
14	11	41	84	84
15	11	43	84	84
16	22	55	22	74

```
SELECT
  SequenceNum as RowNumber,
  Quantity,

  SUM(Quantity) OVER (
    ORDER BY Quantity
    ROWS 3 PRECEDING)
      AS SumOverRows,

  SUM(Quantity) OVER (
    ORDER BY Quantity
    RANGE 3 PRECEDING)
      AS SumOverRange,

  SUM(Quantity) OVER (
    ORDER BY Quantity
    GROUPS 3 PRECEDING)
      AS SumOverGroups

FROM Items
ORDER BY SequenceNum;
```

ROWS vs. RANGE vs. GROUPS full results

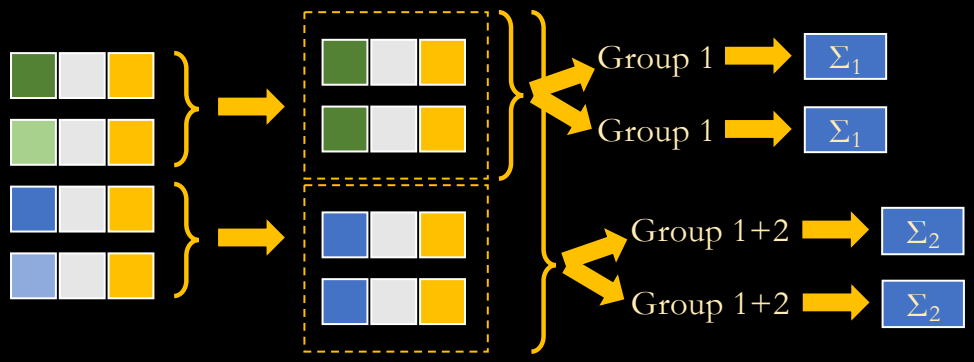
RowNumber	Quantity	SumOverRows	SumOverRange	SumOverGroups
1	1	1	2	2
2	1	2	2	2
3	3	5	5	5
4	5	10	13	15
5	5	14	13	15
6	7	20	17	22
7	8	25	49	52
8	8	28	49	52
9	8	31	49	52
10	8	32	49	52
11	9	33	48	58
12	10	35	58	58
13	11	38	84	84
14	11	41	84	84
15	11	43	84	84
16	22	55	22	74

```
SELECT
  SequenceNum as RowNumber,
  Quantity,

  SUM(Quantity) OVER (
    ORDER BY Quantity
    ROWS 3 PRECEDING)
      AS SumOverRows,

  SUM(Quantity) OVER (
    ORDER BY Quantity
    RANGE 3 PRECEDING)
      AS SumOverRange,

  SUM(Quantity) OVER (
    ORDER BY Quantity
    GROUPS 3 PRECEDING)
      AS SumOverGroups

FROM Items
ORDER BY SequenceNum;
```

Window example: running total orders per month

SalesOrder.sqlite

✗
```
SELECT
        DATE(OrderDate, 'start of month', '+1 month', '-1 day') AS EndOfMonth,
        SUM(OrderTotal) OVER (
                ORDER BY DATE(OrderDate, 'start of month')
                GROUPS UNBOUNDED PRECEDING) AS RunningTotal
FROM Orders;
```



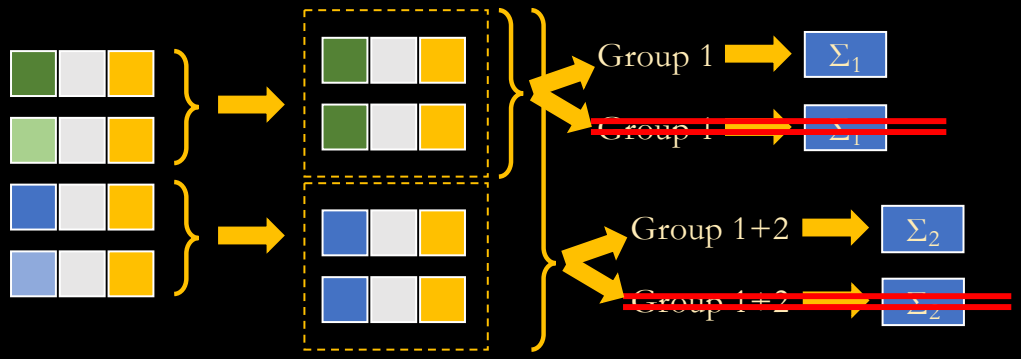
	EndOfMonth	RunningTotal
1	2012-09-30	820475.89
2	2012-09-30	820475.89
3	2012-09-30	820475.89
⋮		
163	2012-09-30	820475.89
164	2012-10-31	1547375.66
165	2012-10-31	1547375.66

- ...but, there are many duplicates (one output per row)

Window example: running total orders per month

SalesOrder.sqlite

```
SELECT DISTINCT
       DATE(OrderDate, 'start of month', '+1 month', '-1 day') AS EndOfMonth,
       SUM(OrderTotal) OVER (
              ORDER BY DATE(OrderDate, 'start of month')
              GROUPS UNBOUNDED PRECEDING) AS RunningTotal
FROM Orders;
```



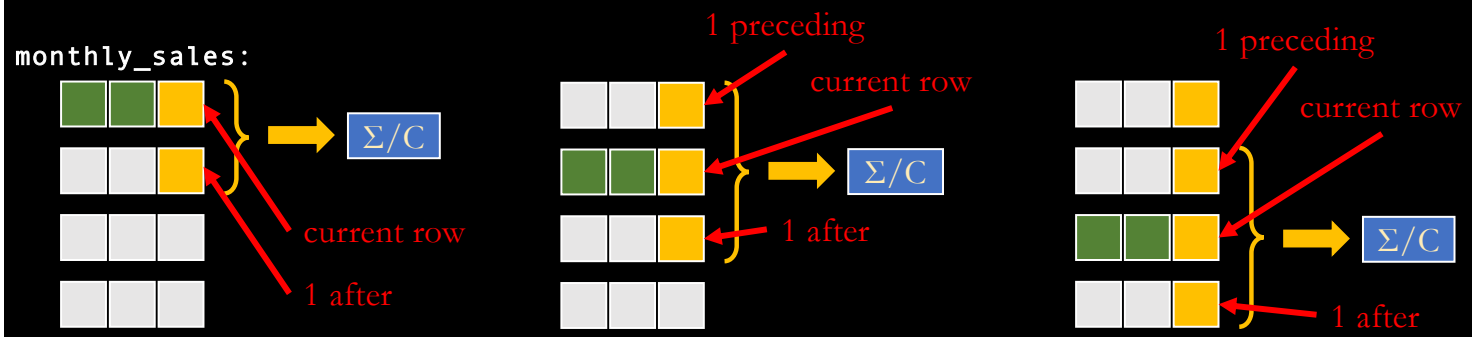
	EndOfMonth	RunningTotal
1	2012-09-30	820475.89
2	2012-10-31	1547375.66
3	2012-11-30	2304527.31
4	2012-12-31	2923437.71
5	2013-01-31	3859954.21
6	2013-02-28	4630895.37

Window example: running average of monthly orders

SalesOrder.sqlite

```
WITH monthly_sales(EndOfMonth, MonthlyTotal) AS (
       SELECT DATE(OrderDate, 'start of month', '+1 month', '-1 day'),
              SUM (OrderTotal)
       FROM Orders
       GROUP BY DATE(OrderDate, 'start of month'))
SELECT EndOfMonth,
       AVG(MonthlyTotal) OVER (ORDER BY EndOfMonth
                     ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
              AS RunningAvg
FROM monthly_sales;
```

	EndOfMonth	RunningAvg
1	2012-09-30	773687.83
2	2012-10-31	768175.77
3	2012-11-30	700987.273333333
4	2012-12-31	770859.516666667
5	2013-01-31	775456.02
6	2013-02-28	853728.83

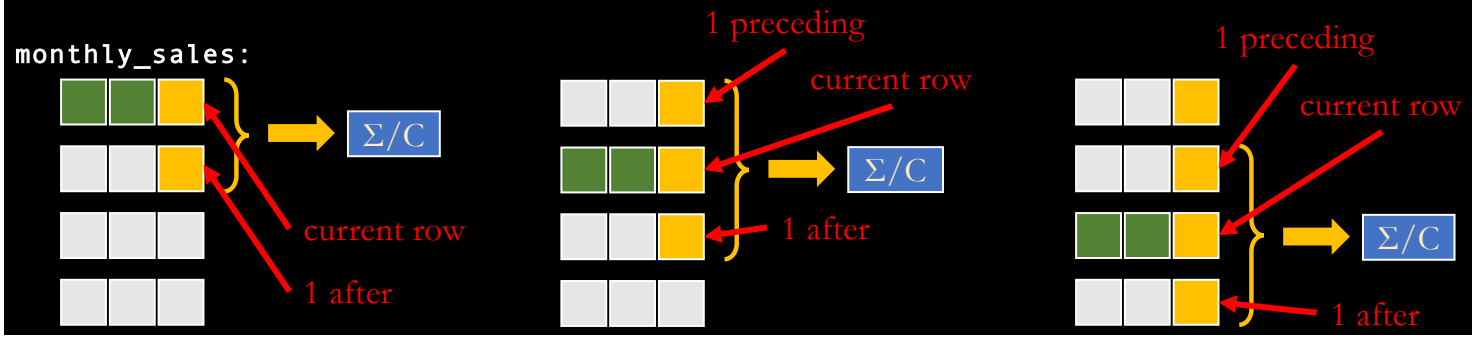


Window example: running average of monthly orders

SalesOrder.sqlite

```
WITH monthly_sales(EndOfMonth, MonthlyTotal) AS (
       SELECT DATE(OrderDate, 'start of month', '+1 month', '-1 day'),
              SUM (OrderTotal)
       FROM Orders
       GROUP BY DATE(OrderDate, 'start of month'))
SELECT EndOfMonth,
       AVG(MonthlyTotal) OVER win
              AS RunningAvg
FROM monthly_sales
WINDOW win AS (ORDER BY EndOfMonth ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING);
```

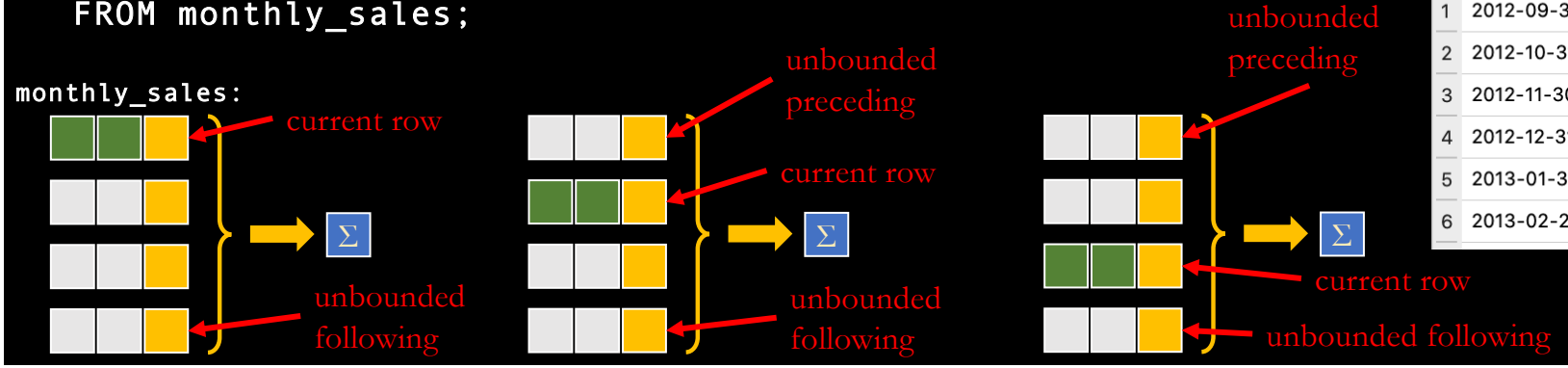
	EndOfMonth	RunningAvg
1	2012-09-30	773687.83
2	2012-10-31	768175.77
3	2012-11-30	700987.273333333
4	2012-12-31	770859.516666667
5	2013-01-31	775456.02
6	2013-02-28	853728.83



Window example: monthly orders as % of total

SalesOrder.sqlite

```
WITH monthly_sales(EndOfMonth, MonthlyTotal) AS (
       SELECT DATE(OrderDate, 'start of month', '+1 month', '-1 day'),
              SUM (OrderTotal)
       FROM Orders
       GROUP BY DATE(OrderDate, 'start of month'))
SELECT EndOfMonth,
       MonthlyTotal * 100.0 / SUM(MonthlyTotal) OVER (ORDER BY EndOfMonth
              ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
              AS MonthlyPercent
FROM monthly_sales;
```



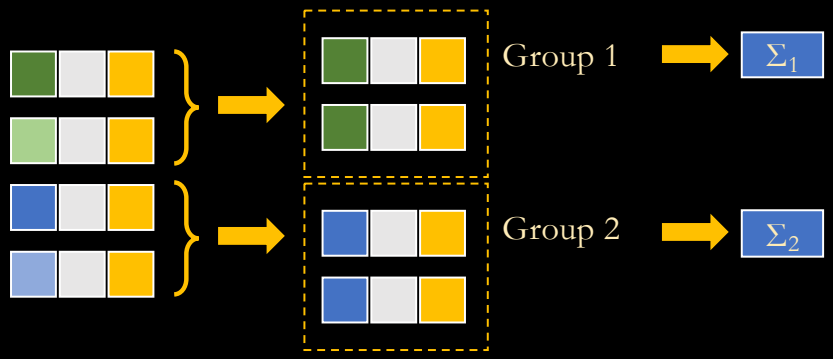
	EndOfMonth	MonthlyPercent
1	2012-09-30	17.7174352786122
2	2012-10-31	15.6967435435709
3	2012-11-30	16.3500055497907
4	2012-12-31	13.3648107018233
5	2013-01-31	20.2232273712546
6	2013-02-28	16.6477775549483

Window example: average monthly shipping delay

SalesOrder.sqlite

- Solution using `GROUP BY`:

```
SELECT DATE(OrderDate, 'start of month', '+1 month', '-1 day') AS EndOfMonth,
       AVG(julianday(ShipDate) - julianday(OrderDate))
FROM Orders
GROUP BY DATE(OrderDate, 'start of month');
```



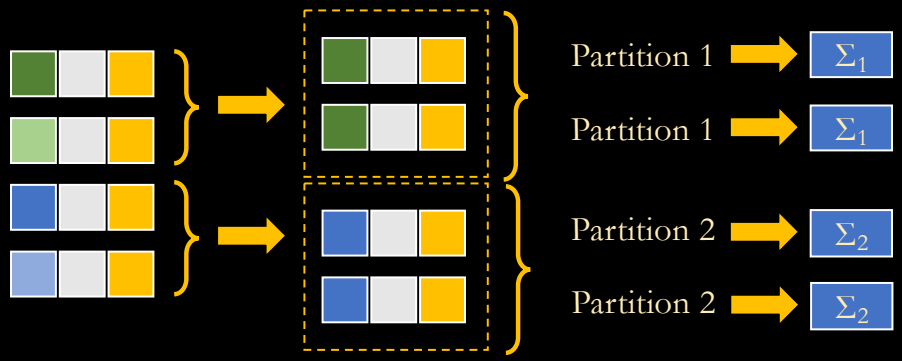
	EndOfMonth	AvgDelay
1	2012-09-30	2.09815950920245
2	2012-10-31	2.08843537414966
3	2012-11-30	2.46206896551724
4	2012-12-31	2.2027972027972
5	2013-01-31	2.1027027027027
6	2013-02-28	2.59627329192547

Window example: average monthly shipping delay

SalesOrder.sqlite

- Solution using windows (partitions) attempt 1

✗
```
SELECT
    DATE(OrderDate, 'start of month', '+1 month', '-1 day') AS EndOfMonth,
    AVG(julianday(ShipDate) - julianday(OrderDate)) OVER win AS AvgDelay
FROM Orders
WINDOW win AS (PARTITION BY DATE(OrderDate, 'start of month'));
```



	EndOfMonth	AvgDelay
1	2012-09-30	2.09815950920245
2	2012-09-30	2.09815950920245
3	2012-09-30	2.09815950920245
⋮		
163	2012-09-30	2.09815950920245
164	2012-10-31	2.08843537414966
165	2012-10-31	2.08843537414966

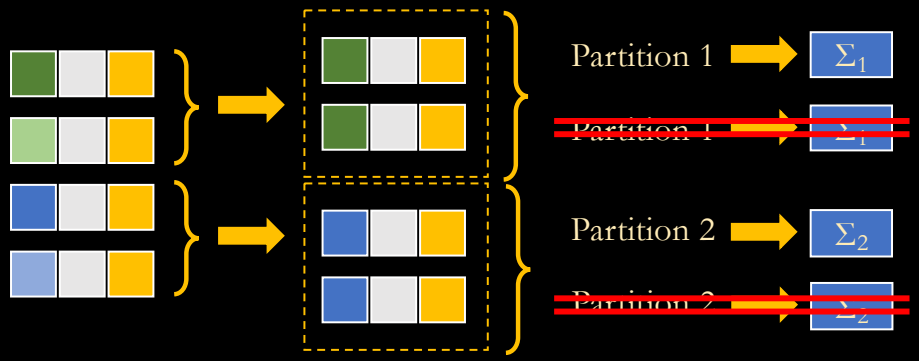
...but, there are many duplicates (one output per row)

Window example: average monthly shipping delay

SalesOrder.sqlite

- Solution using windows (partitions) attempt 2

✓
```
SELECT DISTINCT
    DATE(OrderDate, 'start of month', '+1 month', '-1 day') AS EndOfMonth,
    AVG(julianday(ShipDate) - julianday(OrderDate)) OVER win AS AvgDelay
FROM Orders
WINDOW win AS (PARTITION BY DATE(OrderDate, 'start of month'));
```



	EndOfMonth	AvgDelay
1	2012-09-30	2.09815950920245
2	2012-10-31	2.08843537414966
3	2012-11-30	2.46206896551724
4	2012-12-31	2.2027972027972
5	2013-01-31	2.1027027027027
6	2013-02-28	2.59627329192547

Window Functions

Value	Ranking	Aggregate
FIRST_VALUE()	CUME_DIST()	AVG()
LAST_VALUE()	DENSE_RANK()	COUNT()
LAG()	NTILE()	MAX()
LEAD()	RANK()	MIN()
NTH_VALUE()	ROW_NUMBER()	SUM()
	PERCENT_RANK()	

Built-in Window Function Definitions

Value	Operation
`FIRST_VALUE()`	Get the value of the first row in a specified window frame
`LAST_VALUE()`	Get the value of the last row in a specified window frame
`LAG()`	Provide access to a row at a given physical offset that comes before the current row
`LEAD()`	Provide access to a row at a given physical offset that follows the current row
`NTH_VALUE()`	Return the value of an expression evaluated against the row N of the window frame in the result set

Ranking	Operation
`CUME_DIST()`	Compute the cumulative distribution of a value in an ordered set of values
`DENSE_RANK()`	Compute the rank for a row in an ordered set of rows with no gaps in rank values
`NTILE()`	Divide a result set into a number of buckets as evenly as possible and assign a bucket number to each row
`PERCENT_RANK()`	Calculate the percent rank of each row in an ordered set of rows
`RANK()`	Assign a rank to each row within the partition of the result set
`ROW_NUMBER()`	Assign a sequential integer starting from one to each row within the current partition

Window ranking function example: ROW_NUMBER()

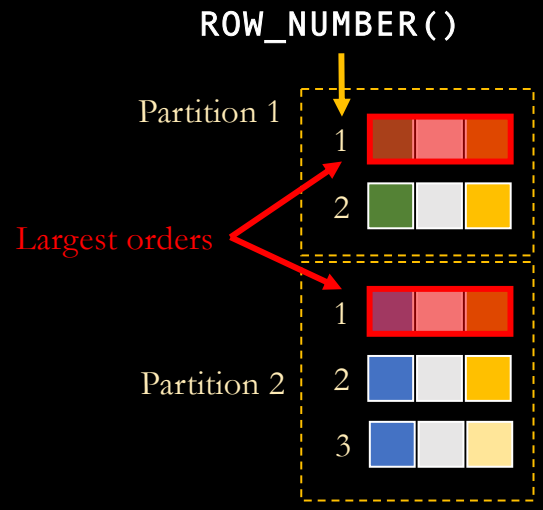
SalesOrder.sqlite

- Which customer issued the largest (in $) order at each month?

```
WITH BigOrders(Month, CustomerID, OrderRank) AS (
  SELECT strftime('%Y-%m', OrderDate), CustomerID,
         ROW_NUMBER() OVER (PARTITION BY DATE(OrderDate, 'start of month')
             ORDER BY OrderTotal DESC) AS OrderRank
  FROM Orders)
SELECT Month, CustomerID,
       CustFirstName || " " || CustLastName AS Name
FROM BigOrders NATURAL JOIN Customers
WHERE OrderRank = 1;
```

strftime('%Y-%m', OrderDate)

	Month	CustomerID	Name
1	2012-09	1004	Robert Brown
2	2012-10	1017	Manuela Seidel
3	2012-11	1006	John Viescas
4	2012-12	1006	John Viescas
5	2013-01	1013	Rachel Patterson
6	2013-02	1005	Dean McCrae



Window Chaining

- A window can be defined by using another window as its base
- Implicitly copy the `PARTITION BY` and optionally `ORDER BY` clauses of the base window

```
SELECT b OVER (
            win ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
            )
FROM t1
WINDOW win AS (PARTITION BY a ORDER BY c);
```

- Rules of engagement:
  - No `PARTITION BY` clause in new window
  - No `ORDER BY` clause in new window if the base window has one
  - No frame specification in base window. The frame spec can only be given in the new window

Window Chaining Example

```
-- The following SELECT statement returns:
--
--   c     | a | b | group_concat
-------------------------------
--   one   | 1 | A | A.D.G
--   one   | 4 | D | D.G
--   one   | 7 | G | G
--   three | 3 | C | C.F
--   three | 6 | F | F
--   two   | 2 | B | B.E
--   two   | 5 | E | E
--
SELECT c, a, b, group_concat(b, '.') OVER (
  PARTITION BY c ORDER BY a RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
) AS group_concat
FROM t1 ORDER BY c, a;
```

Compare to No Window Chaining

```
-- The following SELECT statement returns:
--
--   c     | a | b | group_concat
-------------------------------
--   one   | 1 | A | A.D.G.C.F.B.E
--   one   | 4 | D | D.G.C.F.B.E
--   one   | 7 | G | G.C.F.B.E
--   three | 3 | C | C.F.B.E
--   three | 6 | F | F.B.E
--   two   | 2 | B | B.E
--   two   | 5 | E | E
--
SELECT c, a, b, group_concat(b, '.') OVER (
  ORDER BY c, a ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
) AS group_concat
FROM t1 ORDER BY c, a;
```

Window Chaining – SELECT's ORDER BY is immaterial

```
-- The following SELECT statement returns:
--
--   c     | a | b | group_concat
-------------------------------
--   one   | 1 | A | A.D.G
--   two   | 2 | B | B.E
--   three | 3 | C | C.F
--   one   | 4 | D | D.G
--   two   | 5 | E | E
--   three | 6 | F | F
--   one   | 7 | G | G
--
SELECT c, a, b, group_concat(b, '.') OVER (
  PARTITION BY c ORDER BY a RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
) AS group_concat
FROM t1 ORDER BY a;
```

Window Filters

`FILTER` clause:

- If a filter clause is specified
  - Input rows are evaluated against the filter clause
    - If true → the row is fed to the window function
    - If false → the row is discarded
  - Can only be used with aggregate window functions (no value, ranking)

```
SELECT COUNT(*) FILTER (WHERE Quantity > 10)
            AS FilteredCount
FROM Items;
```

	FilteredCount
1	4

```
SELECT COUNT(*) FILTER (WHERE Quantity > 10)
            AS FilteredCount
FROM Items WHERE Quantity < 12;
```

	FilteredCount
1	3

RowNumber	Quantity
1	1
2	1
3	3
4	5
5	5
6	7
7	8
8	8
9	8
10	8
11	9
12	10
13	11
14	11
15	11
16	22

FILTER ≠ WHERE

example_table:

	value	category
1	10	A
2	NULL	A
3	20	B
4	30	B

```
SELECT category,
   COUNT(value)
     AS count_values
FROM example_table
WHERE value > 15
GROUP BY category;
```

	category	count_values
1	B	2

example_table:

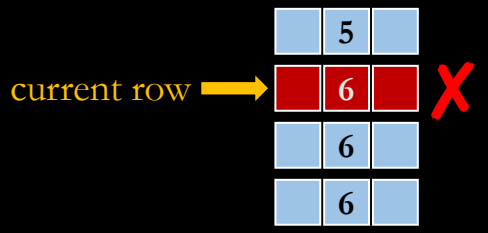
	value	category
1	10	A
2	NULL	A
3	20	B
4	30	B

```
SELECT category,
   COUNT(*)
   FILTER(WHERE value > 15)
     AS count_values
FROM example_table
GROUP BY category;
```

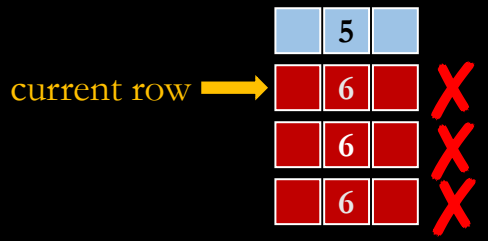
	category	count_values
1	A	0
2	B	2

Frame Exclusion

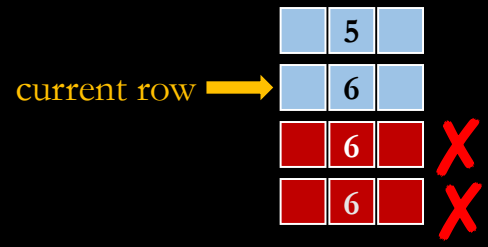
- `EXCLUDE CURRENT ROW`
  - Remove the current row itself from the frame
  - Independent of the frame unit—remove even if the `RANGE` or `GROUPS` unit is used, and the current row has peers



- `EXCLUDE GROUP`
  - Remove the current row + all its peers from the frame
  - Independent of the frame unit



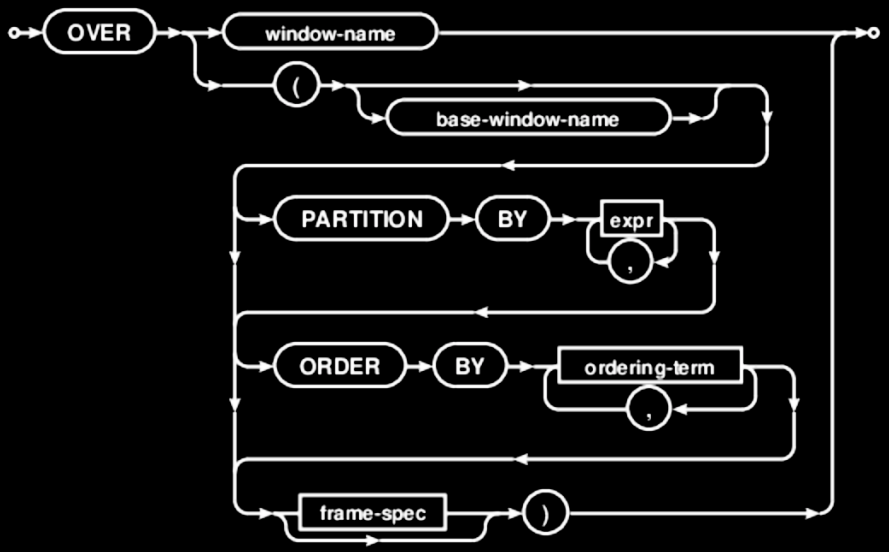
- `EXCLUDE TIES`
  - Remove the peers of the current row, but not the current row itself



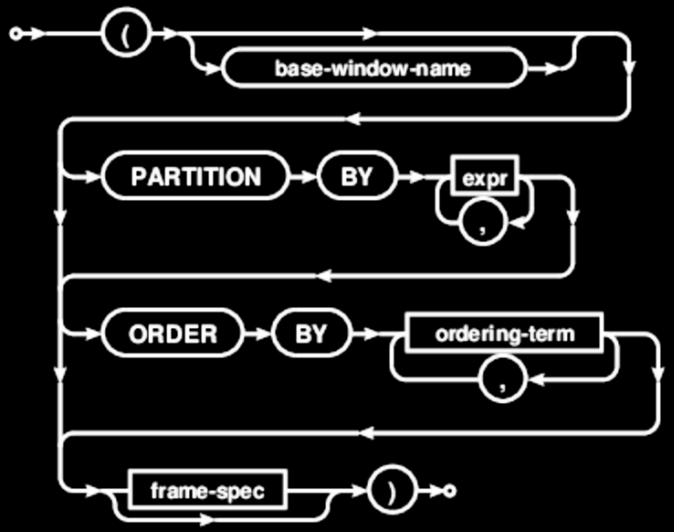
- `EXCLUDE NO OTHERS`
  - does not remove any rows

Window Syntax Diagrams

`OVER` clause:

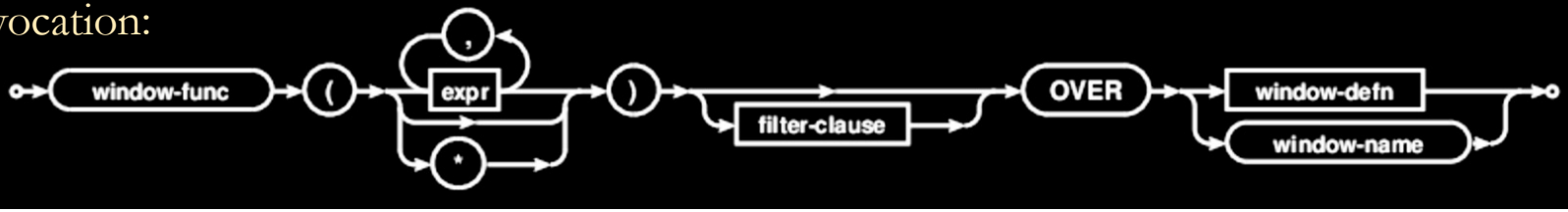


Window definition:



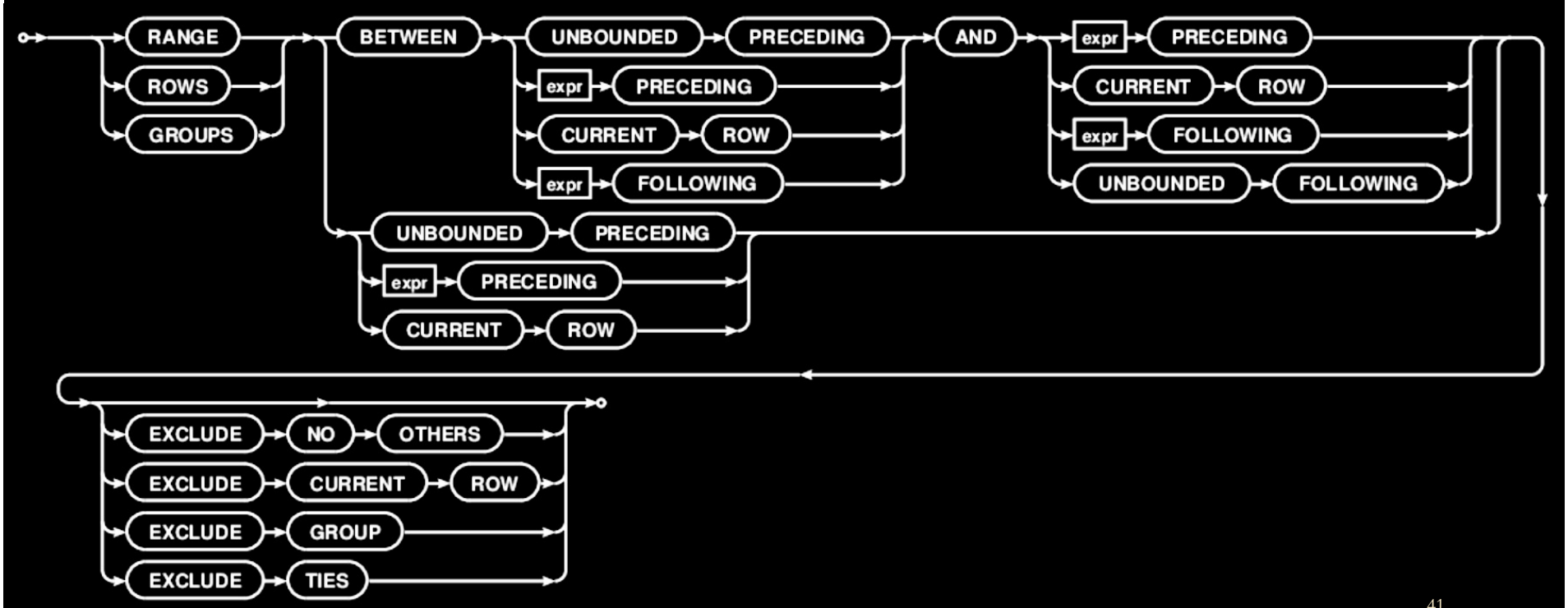
`FILTER` clause:

Window function invocation:



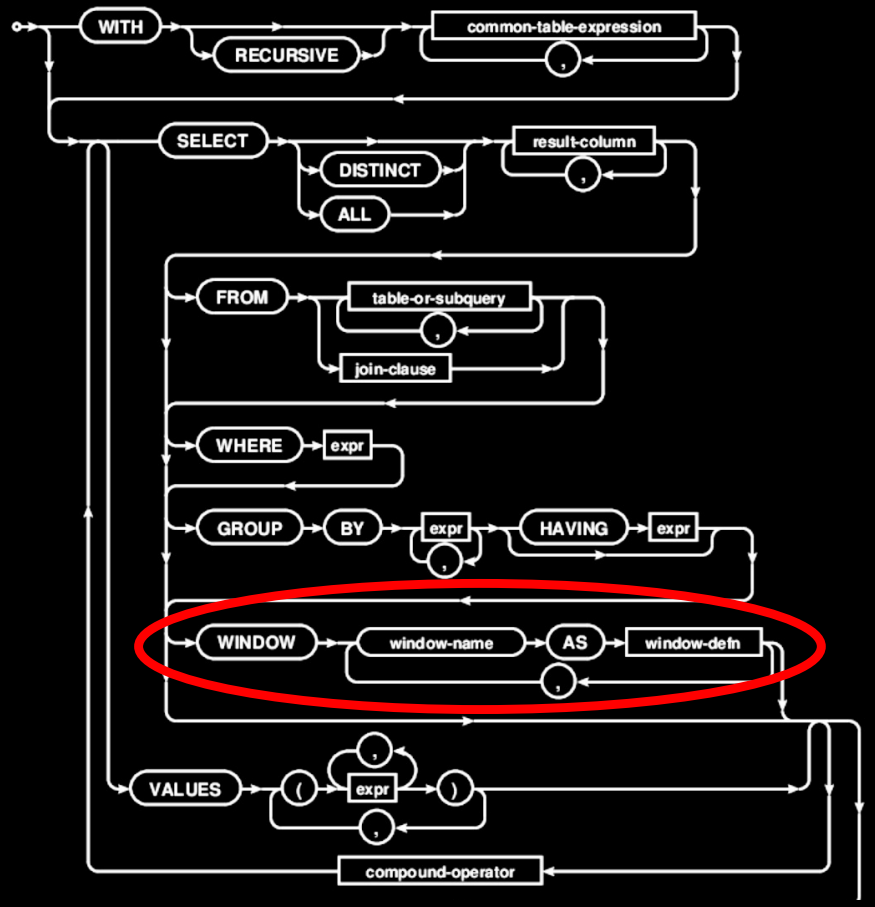
Window Syntax Diagrams

Frame specification:



Window Syntax Diagrams

Windowing in `SELECT`:



Windows in PostgreSQL

- Window frames with `ROWS`: supported since 2009
- Window frames with `RANGE`: full support added in 2018 with PostgreSQL 11
  - Previously only `UNBOUNDED PRECEDING` or `CURRENT ROW`
- Window frames with `GROUPS`: full support added in 2018 with PostgreSQL 11
  - Previously not supported *at all*

- `EXCLUDE`: support added in 2018 with PostgreSQL 11
  - Previously not supported *at all*

- More details at **https://www.postgresql.org/docs/10/functions-window.html**