Maximize Time Series Data for Data Science with SQL in 3 Steps

Eunice Santos
November 2, 2019 at 2:30PM

Big Mountain Data and Dev Conference 2019
Salt Lake City, Utah

Information about the presenter

Eunice Santos

- Work with time series data in public health for over five years
- Experienced with SQL and R language for data preparation and analysis
- PASS Data Science Virtual Group Chapter Co-Lead (in training)
- Data Quality Committee Co-Chair for Community of Practice
- Coauthored journal article Modeling and Forecasting Influenza-like Illness (ILI) in Houston, Texas Using Three Surveillance Data Capture Mechanisms in Online Journal of Public Health Informatics

https://github.com/gh1121/SQL

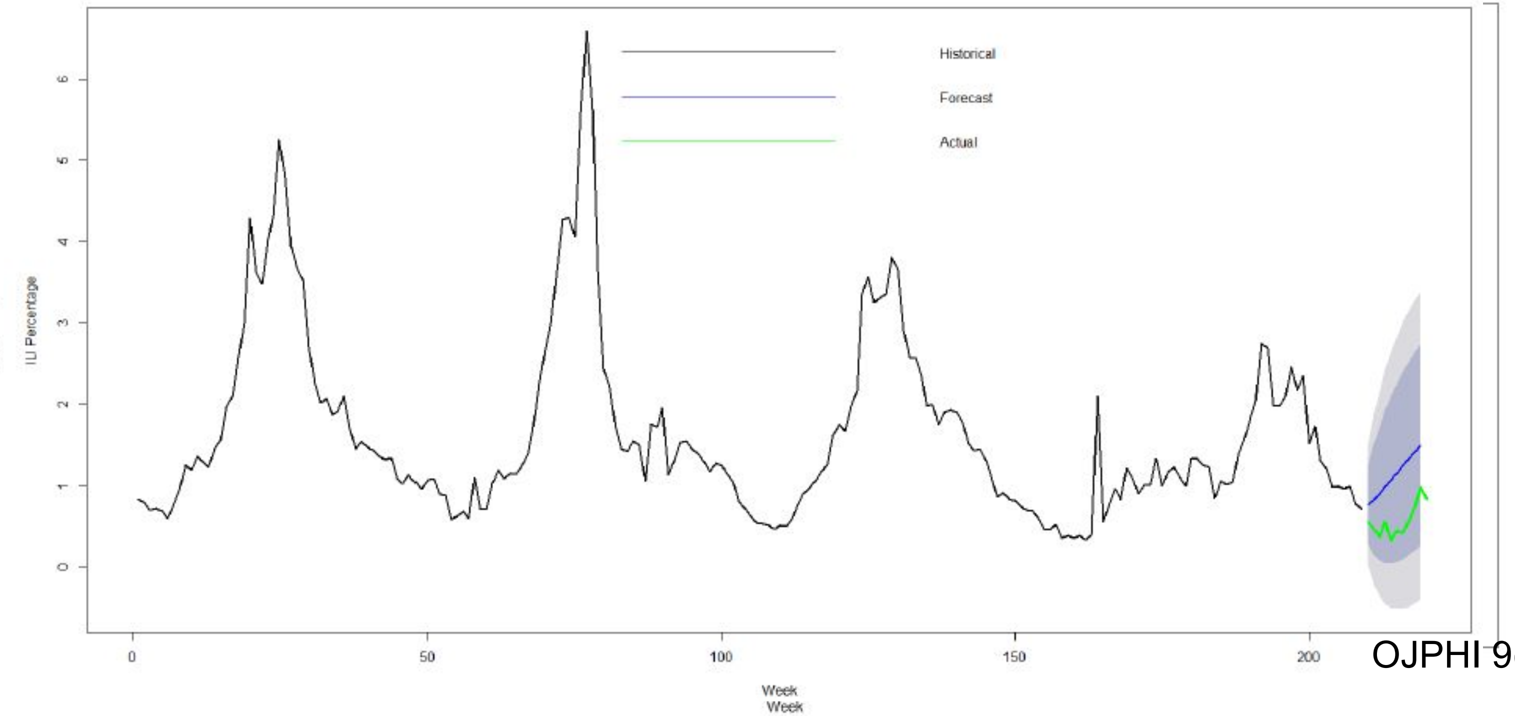
https://www.linkedin.com/in/eunice-rebecca-santos-23293b2

Outline

- **Introduction: Use Cases in Data Science**
- Overview of Time Series Data
- SQL features (Transact-SQL and PostgreSQL)
- Utilizing Time Series Data with PostgreSQL

Use case 1

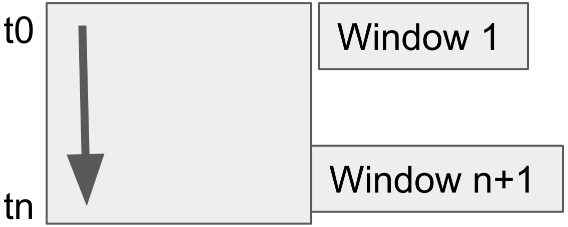
Forecasting Influenza Like Illness with R



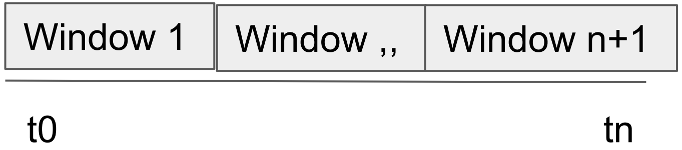
OJPHI 9(2) 2017

Use case 2

Data Table



Time axis

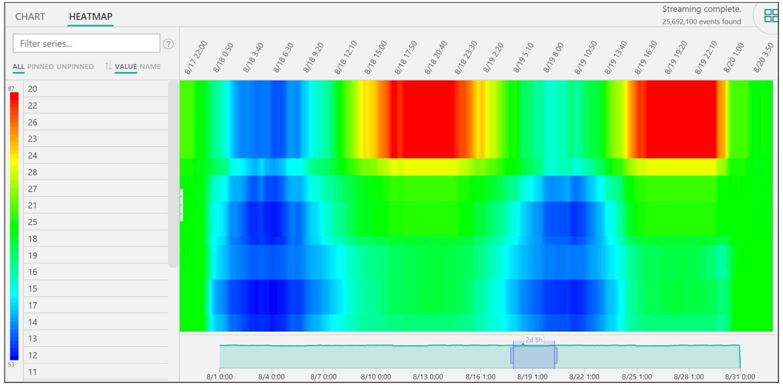


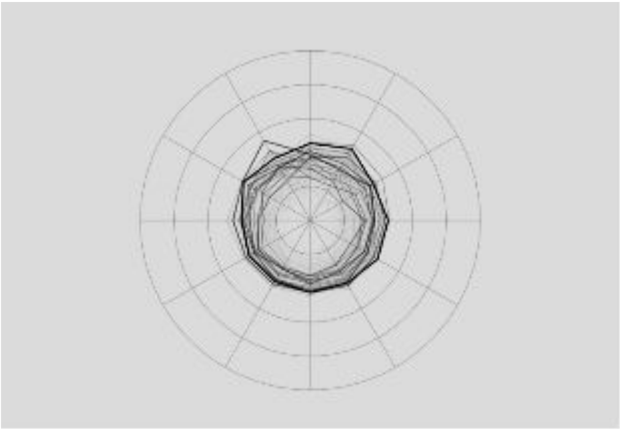
Analytics: Moving Average

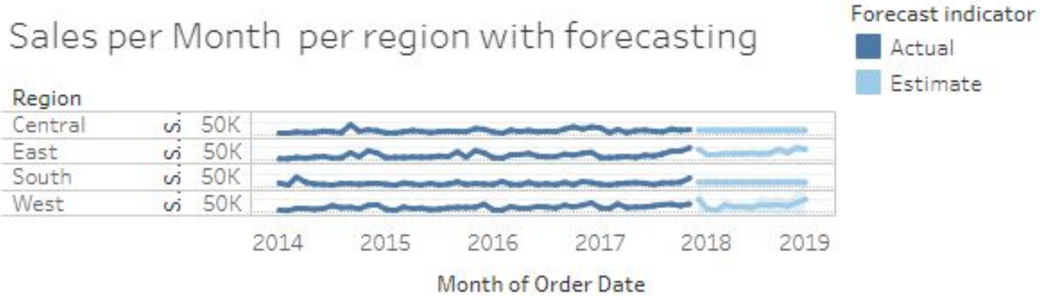
- One SQL solution
  - Windowing operation
- Factors that impact performance
  - Time unit for index (i.e. timestamp when query is run on calendar date)

Use case 3

Time is the main axis







Use case 4

Missing data

- One SQL solution
  - Common Table Expressions
- Factors that impact performance
  - Index (seek or ordered scan)
  - Quantity missing values

Date	ProportionSales
12/23/2019	0.7465212284
12/26/2019	0.7017809484
12/27/2019	0.2838156626
12/28/2019	0.617693722
12/29/2019	0.525151857
12/30/2019	0.1956264643
12/31/2019	0.1962465622
1/2/2020	0.540869351
1/3/2020	0.09519501764
1/4/2020	0.1015091941

SQL Server MVP Deep Dives, CH5

Outline

- Introduction: Use Cases in Data Science
- **Overview of Time Series Data**
- PostgreSQL functions
- Utilizing Time Series Data with PostgreSQL
- Demonstration

Overview of Time Series Data

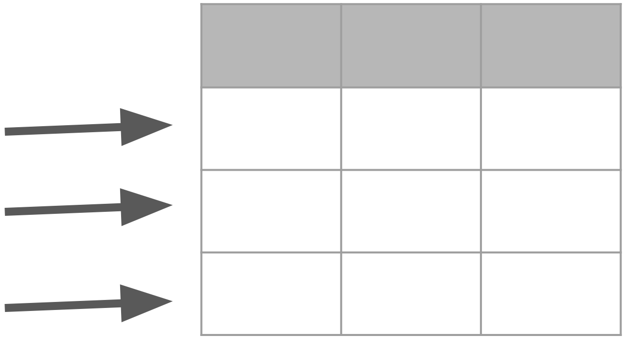
Overview of time series data

- Data values organized by time
- Sensor data in microseconds, daily closing stock price, health conditions in years

MS Timer series solutions

Record ingestion

- Captured in order of time
- Usually insert rather than an update to your database
- Data is added incrementally.
    - Usually the most recent data is queried more often



MS Timer series solutions

Aggregated tables/views

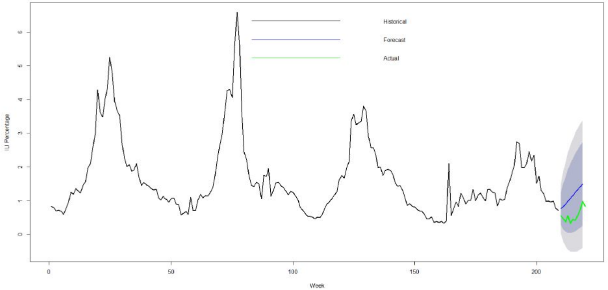
	Wide or Columnar	Long or Interleaved
Pros	Interpretability	New series does not affect the data structure
Cons	New columns affect the data structure	Magnitude must be the same for each series

Time	Series 1	Series 2	Series nth
201901	0.10	0.20	0.30
201902	0.15	0.25	0.35

Time	Series	Magnitude
201901	Value 1	0.10
201901	Value 2	0.20
201903	Value 3	0.30
201902	Value 1	0.15
201902	Value 2	0.25
201902	Value 3	0.35

Analytics and reporting

- Examples of analysis: historical trends, real-time alerts, predictive modeling, or forecasting
- Change is measured over time, enabling you to look backward and to predict future change.
- Visualized for trends, seasonality, cyclic patterns
- Autocorrelation between lagged values of a time series.



Why SQL?

Time Series	SQL feature
Sequential data	Window operations
Lots of data	Index column, partitions
Analyze for changes over time or forecasting	Analytic functions
Autocorrelation	LAG to access data in previous row(s)

Outline

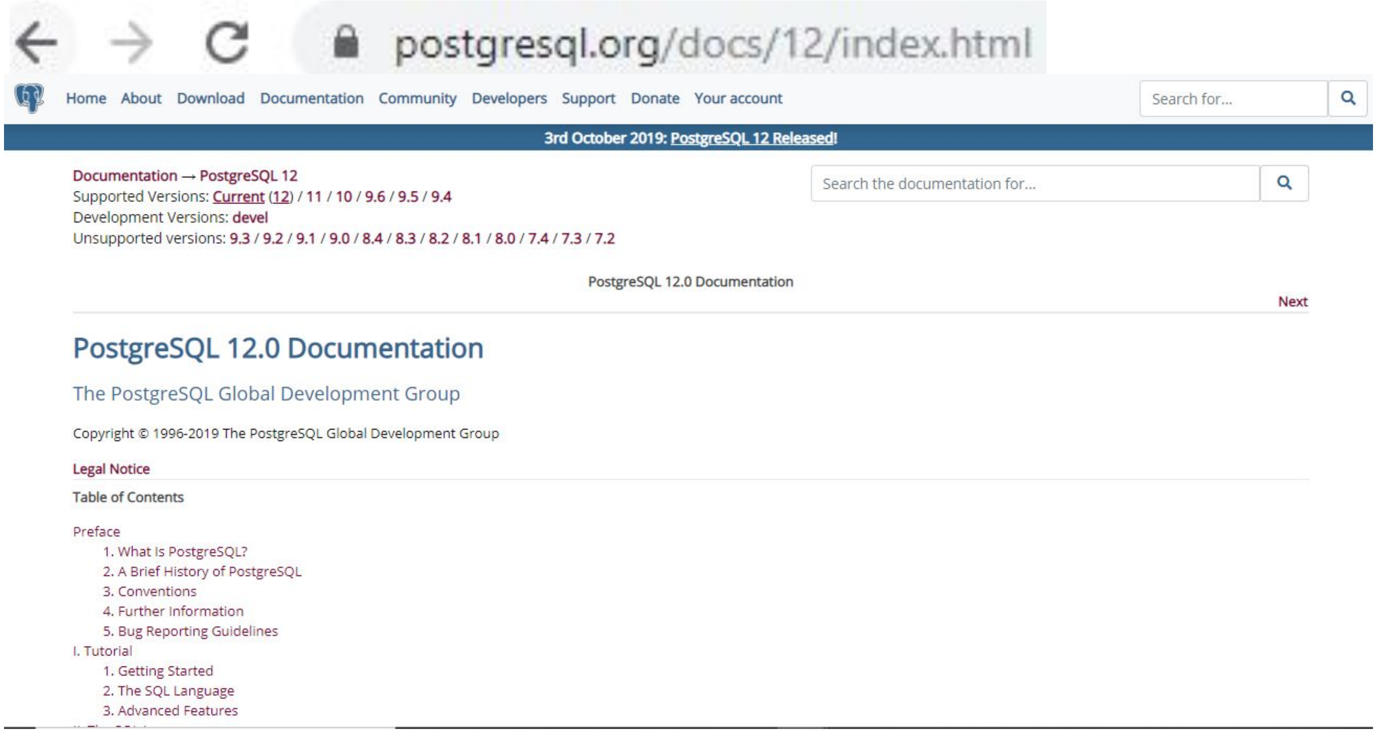
- Introduction: Use Cases in Data Science
- Overview of Time Series Data
- **SQL features**
- Utilizing Time Series Data with PostgreSQL
- Demonstration

SQL features

- Common Table Expressions
- Window Functions
- Aggregate Function

Documentation for PostgreSQL

Table of Contents and Tutorial



Documentation for PostgreSQL

Common Table Expressions (CTE)


postgresql.org/docs/12/queries-with.html

Home About Download Documentation Community Developers Support Donate Your account

Search for...

3rd October 2019: PostgreSQL 12 Released!

Documentation → PostgreSQL 12
Supported Versions: Current (12) / 11 / 10 / 9.6 / 9.5 / 9.4
Development Versions: devel
Unsupported versions: 9.3 / 9.2 / 9.1 / 9.0 / 8.4

Search the documentation for...

7.8. WITH Queries (Common Table Expressions)

Prev Up Chapter 7. Queries Home Next

7.8. WITH Queries (Common Table Expressions)

7.8.1. SELECT in WITH
7.8.2. Data-Modifying Statements in WITH

WITH provides a way to write auxiliary statements for use in a larger query. These statements, which are often referred to as Common Table Expressions or CTEs, can be thought of as defining temporary tables that exist just for one query. Each auxiliary statement in a WITH clause can be a SELECT, INSERT, UPDATE, or DELETE; and the WITH clause itself is attached to a primary statement that can also be a SELECT, INSERT, UPDATE, or DELETE.

7.8.1. SELECT in WITH

The basic value of SELECT in WITH is to break down complicated queries into simpler parts. An example is:

```
WITH regional_sales AS (
    SELECT region, SUM(amount) AS total_sales
```

The syntax of a CTE

```
WITH cte_name1 AS (query_definition1)
[, cte_name2 AS (query_definition2)]
[...]
sql_statement
```

Two CTEs and a query that uses them

```
WITH Summary AS
(
    SELECT VendorState, VendorName, SUM(InvoiceTotal)
        AS SumOfInvoices
    FROM Invoices JOIN Vendors
      ON Invoices.VendorID = Vendors.VendorID
    GROUP BY VendorState, VendorName
),
TopInState AS
(
    SELECT VendorState, MAX(SumOfInvoices) AS SumOfInvoices
    FROM Summary
    GROUP BY VendorState
)
SELECT Summary.VendorState, Summary.VendorName,
    TopInState.SumOfInvoices
FROM Summary JOIN TopInState
    ON Summary.VendorState = TopInState.VendorState AND
       Summary.SumOfInvoices = TopInState.SumOfInvoices
ORDER BY Summary.VendorState;
```

The result set

	VendorState	VendorName	SumOfInvoices
1	AZ	Wells Fargo Bank	662.00
2	CA	Digital Dreamworks	7125.34
3	DC	Reiter's Scientific & Pro Books	600.00
4	MA	Dean Witter Reynolds	1367.50
5	MI	Malloy Lithographing Inc	119892.41
6	NV	United Parcel Service	23177.96
7	OH	Edward Data Services	207.78
8	PA	Cardinal Business Media, Inc.	265.36

(10 rows)

Documentation for PostgreSQL

Window Functions

postgresql.org/docs/12/tutorial-window.html

Home About Download Documentation Community Developers Support Donate Your account

Search for...

3rd October 2019: PostgreSQL 12 Released!

Documentation → PostgreSQL 12
Supported Versions: Current (12) / 11 / 10 / 9.6 / 9.5 / 9.4
Development Versions: devel
Unsupported versions: 9.3 / 9.2 / 9.1 / 9.0 / 8.4

Search the documentation for...

3.5. Window Functions

Prev Up Chapter 3. Advanced Features Home Next

3.5. Window Functions

A *window function* performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. However, window functions do not cause rows to become grouped into a single output row like non-window aggregate calls would. Instead, the rows retain their separate identities. Behind the scenes, the window function is able to access more than just the current row of the query result.

Here is an example that shows how to compare each employee's salary with the average salary in his or her department:

```
SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY depname) FROM empsalary;
```

```
  depname  | empno | salary |          avg
-----------+-------+--------+-----------------------
 develop   |    11 |   5200 | 5020.0000000000000000
 develop   |     7 |   4200 | 5020.0000000000000000
 develop   |     9 |   4500 | 5020.0000000000000000
```

A query that calculates a cumulative total and moving average

```
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal,
    SUM(InvoiceTotal) OVER (ORDER BY InvoiceDate) AS CumTotal,
    COUNT(InvoiceTotal) OVER (ORDER BY InvoiceDate) AS Count,
    AVG(InvoiceTotal) OVER (ORDER BY InvoiceDate) AS MovingAvg
FROM Invoices;
```

The result set

	InvoiceNumber	InvoiceDate	InvoiceTotal	CumTotal	Count	MovingAvg
1	989319-457	2015-12-08 00:00:00	3813.33	3813.33	1	3813.33
2	263253241	2015-12-10 00:00:00	40.20	3853.53	2	1926.765
3	963253234	2015-12-13 00:00:00	138.75	3992.28	3	1330.76
4	2-000-2993	2015-12-16 00:00:00	144.70	4195.23	6	699.205
5	963253251	2015-12-16 00:00:00	15.50	4195.23	6	699.205
6	963253261	2015-12-16 00:00:00	42.75	4195.23	6	699.205

The syntax of the OVER clause

```
aggregate_function OVER ([partition_by_clause]
     [order_by_clause])
```

A query that groups the summary data by date

```
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal,
  SUM(InvoiceTotal) OVER (PARTITION BY InvoiceDate) AS DateTotal,
  COUNT(InvoiceTotal) OVER (PARTITION BY InvoiceDate) AS DateCount,
  AVG(InvoiceTotal) OVER (PARTITION BY InvoiceDate) AS DateAvg
FROM Invoices;
```

The result set

	InvoiceNumber	InvoiceDate	InvoiceTotal	DateTotal	DateCount	DateAvg
1	989319-457	2015-12-08 00:00:00	3813.33	3813.33	1	3813.33
2	263253241	2015-12-10 00:00:00	40.20	40.20	1	40.20
3	963253234	2015-12-13 00:00:00	138.75	138.75	1	138.75
4	2-000-2993	2015-12-16 00:00:00	144.70	202.95	3	67.65
5	963253251	2015-12-16 00:00:00	15.50	202.95	3	67.65
6	963253261	2015-12-16 00:00:00	42.75	202.95	3	67.65

The same query grouped by TermsID

```
SELECT InvoiceNumber, TermsID, InvoiceDate, InvoiceTotal,
    SUM(InvoiceTotal) OVER
        (PARTITION BY TermsID ORDER BY InvoiceDate) AS CumTotal,
    COUNT(InvoiceTotal) OVER
        (PARTITION BY TermsID ORDER BY InvoiceDate) AS Count,
    AVG(InvoiceTotal) OVER
        (PARTITION BY TermsID ORDER BY InvoiceDate) AS MovingAvg
FROM Invoices;
```

The result set

	InvoiceNumber	TermsID	InvoiceDate	InvoiceTotal	CumTotal	Count	MovingAvg
22	97-1024A	2	2016-03-20 00:00:00	356.48	9415.08	16	588.4425
23	31361833	2	2016-03-21 00:00:00	579.42	9994.50	17	587.9117
24	134116	2	2016-03-28 00:00:00	90.36	10084.86	18	560.27
25	989319-457	3	2015-12-08 00:00:00	3813.33	3813.33	1	3813.33
26	263253241	3	2015-12-10 00:00:00	40.20	3853.53	2	1926.765
27	963253234	3	2015-12-13 00:00:00	138.75	3992.28	3	1330.76

A query that uses the LAG function

```
SELECT RepID, SalesYear, SalesTotal AS CurrentSales,
    LAG(SalesTotal, 1, 0)
        OVER (PARTITION BY RepID ORDER BY SalesYear)
            AS LastSales,
    SalesTotal - LAG(SalesTotal, 1, 0)
        OVER (PARTITION BY REPID ORDER BY SalesYear)
            AS Change
FROM SalesTotals;
```

	RepID	SalesYear	CurrentSales	LastSales	Change
1	1	2014	1274856.38	0.00	1274856.38
2	1	2015	923746.85	1274856.38	-351109.53
3	1	2016	998337.46	923746.85	74590.61
4	2	2014	978465.99	0.00	978465.99
5	2	2015	974853.81	978465.99	-3612.18
6	2	2016	887695.75	974853.81	-87158.06

Documentation for PostgreSQL

Aggregate Functions

postgresql.org/docs/12/functions-aggregate.html

Home About Download Documentation Community Developers Support Donate Your account

Search for...

3rd October 2019: PostgreSQL 12 Released!

Documentation → PostgreSQL 12
Supported Versions: Current (12) / 11 / 10 / 9.6 / 9.5 / 9.4
Development Versions: devel
Unsupported versions: 9.3 / 9.2 / 9.1 / 9.0 / 8.4 / 8.3 / 8.2 / 8.1 / 8.0 / 7.4 / 7.3 / 7.2 / 7.1

Search the documentation for...

9.20. Aggregate Functions
Prev Up Chapter 9. Functions and Operators Home Next

9.20. Aggregate Functions

Aggregate functions compute a single result from a set of input values. The built-in general-purpose aggregate functions are listed in Table 9.55 and statistical aggregates in Table 9.56. The built-in within-group ordered-set aggregate functions are listed in Table 9.57 while the built-in within-group hypothetical-set ones are in Table 9.58. Grouping operations, which are closely related to aggregate functions, are listed in Table 9.59. The special syntax considerations for aggregate functions are explained in Section 4.2.7. Consult Section 2.7 for additional introductory information.

Table 9.55. General-Purpose Aggregate Functions

Function	Argument Type(s)	Return Type	Partial Mode	Description
`array_agg(`*`expression`*`)`	any non-array type	array of the argument type	No	input values, including nulls, concatenated into an array
`array_agg(`*`expression`*`)`	any array type	same as argument data type	No	input arrays concatenated into array of one higher dimension (inputs must all have same dimensionality, and cannot be empty or null)
`avg(`*`expression`*`)`	`smallint`, `int`, `bigint`, `real`, `double precision`, `numeric`, or `interval`	`numeric` for any integer-type argument, `double precision` for a floating-point argument, otherwise the same as the argument data type	Yes	the average (arithmetic mean) of all non-null input values

The columns in the SalesReps table

Column name	Data type
RepID	int
RepFirstName	varchar(50)
RepLastName	varchar(50)

The columns in the SalesTotals table

Column name	Data type
RepID	int
SalesYear	char(4)
SalesTotal	money

A query that uses the FIRST_VALUE and LAST_VALUE functions

```
SELECT SalesYear, RepFirstName + ' ' +
       RepLastName AS RepName, SalesTotal,
       FIRST_VALUE(RepFirstName + ' ' + RepLastName)
           OVER (PARTITION BY SalesYear
               ORDER BY SalesTotal DESC)
           AS HighestSales,
       LAST_VALUE(RepFirstName + ' ' + RepLastName)
           OVER (PARTITION BY SalesYear
               ORDER BY SalesTotal DESC
              RANGE BETWEEN UNBOUNDED PRECEDING AND
                            UNBOUNDED FOLLOWING)
           AS LowestSales
FROM SalesTotals JOIN SalesReps
  ON SalesTotals.RepID = SalesReps.RepID;
```

The result set

	SalesYear	RepName	SalesTotal	HighestSales	LowestSales
1	2014	Jonathon Thomas	1274856.38	Jonathon Thomas	Sonja Martinez
2	2014	Andrew Markasian	1032875.48	Jonathon Thomas	Sonja Martinez
3	2014	Sonja Martinez	978465.99	Jonathon Thomas	Sonja Martinez
4	2015	Andrew Markasian	1132744.56	Andrew Markasian	Lydia Kramer
5	2015	Sonja Martinez	974853.81	Andrew Markasian	Lydia Kramer
6	2015	Jonathon Thomas	923746.85	Andrew Markasian	Lydia Kramer
7	2015	Phillip Winters	655786.92	Andrew Markasian	Lydia Kramer
8	2015	Lydia Kramer	422847.86	Andrew Markasian	Lydia Kramer
9	2016	Jonathon Thomas	998337.46	Jonathon Thomas	Lydia Kramer
10	2016	Sonja Martinez	887695.75	Jonathon Thomas	Lydia Kramer
11	2016	Phillip Winters	72443.37	Jonathon Thomas	Lydia Kramer
12	2016	Lydia Kramer	45182.44	Jonathon Thomas	Lydia Kramer

Outline

- Introduction: Use Cases in Data Science
- Overview of Time Series Data
- SQL features
- **Utilizing Time Series Data with PostgreSQL**

Utilizing Time Series Data with PostgreSQL

- Step 1: Indexes, Partitions | Performance testing
- Step 2: Lag | Prep to analyze retrospective data
- Step 3: Moving average, forecasting | Analytics
- NEXT Step: Machine Learning, Visualization

NEXT STEP: Machine learning, Visualization

Neural Network Models

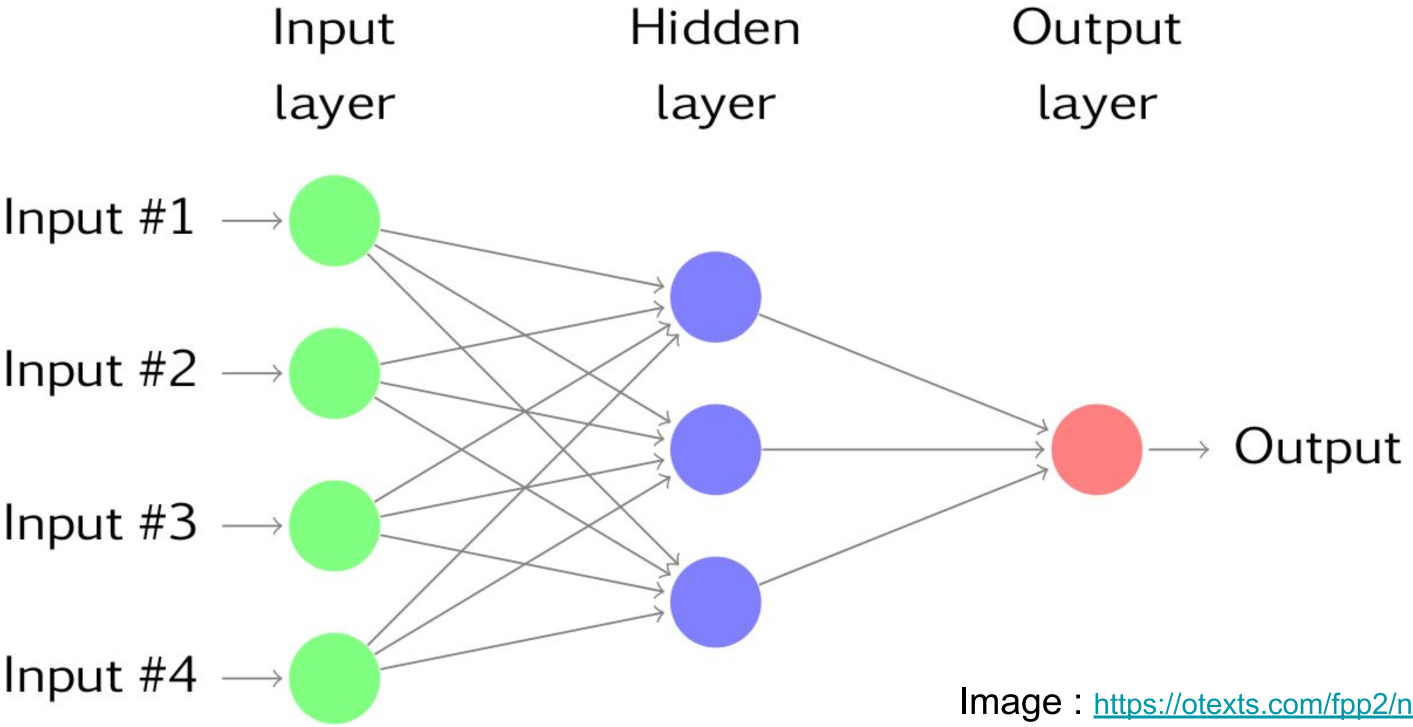
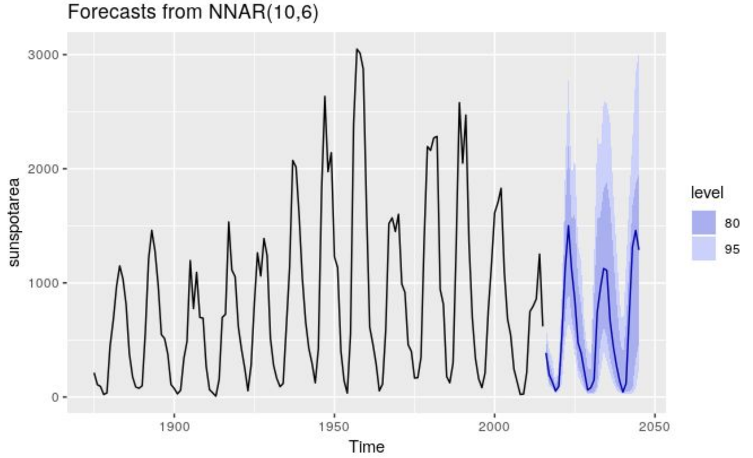


Image : https://otexts.com/fpp2/nnetar.html

Neural Network and Time Series Data



- Autoregression and lagged values
- R function nnetar () fits a Neural Network Autoregression Model
- Non-seasonal time series uses optimal number of lags
- Seasonal time series requires seasonally adjusted data and an optimal linear model with seasonal adjustments.
- Forecast periods are estimated in an iterative manner beginning with historical data plus n-periods.
- OUTPUT: prediction intervals calculated with simulated paths for the future

Why SQL?

Time Series	SQL feature
Sequential data	Window operations
Lots of data	Index column, partitions
Analyze for changes over time or forecasting	Analytic functions
Autocorrelation	LAG to access data in previous row(s)

Resources

- PostgreSQL Documentation | Chapter 3 | 3.5 Window Functions
  https://www.postgresql.org/docs/current/tutorial-window.html
- Lynda.com course Title: Advanced SQL for data science: Time Series
- SQL Server MVP Deep Dives
- Murach's SQL Server 2016 for developers
- Online Journal of Public Health Informatics * ISSN 1947-2579 *
  http://ojphi.org
- Forecasting: Principles and Practice (free online)