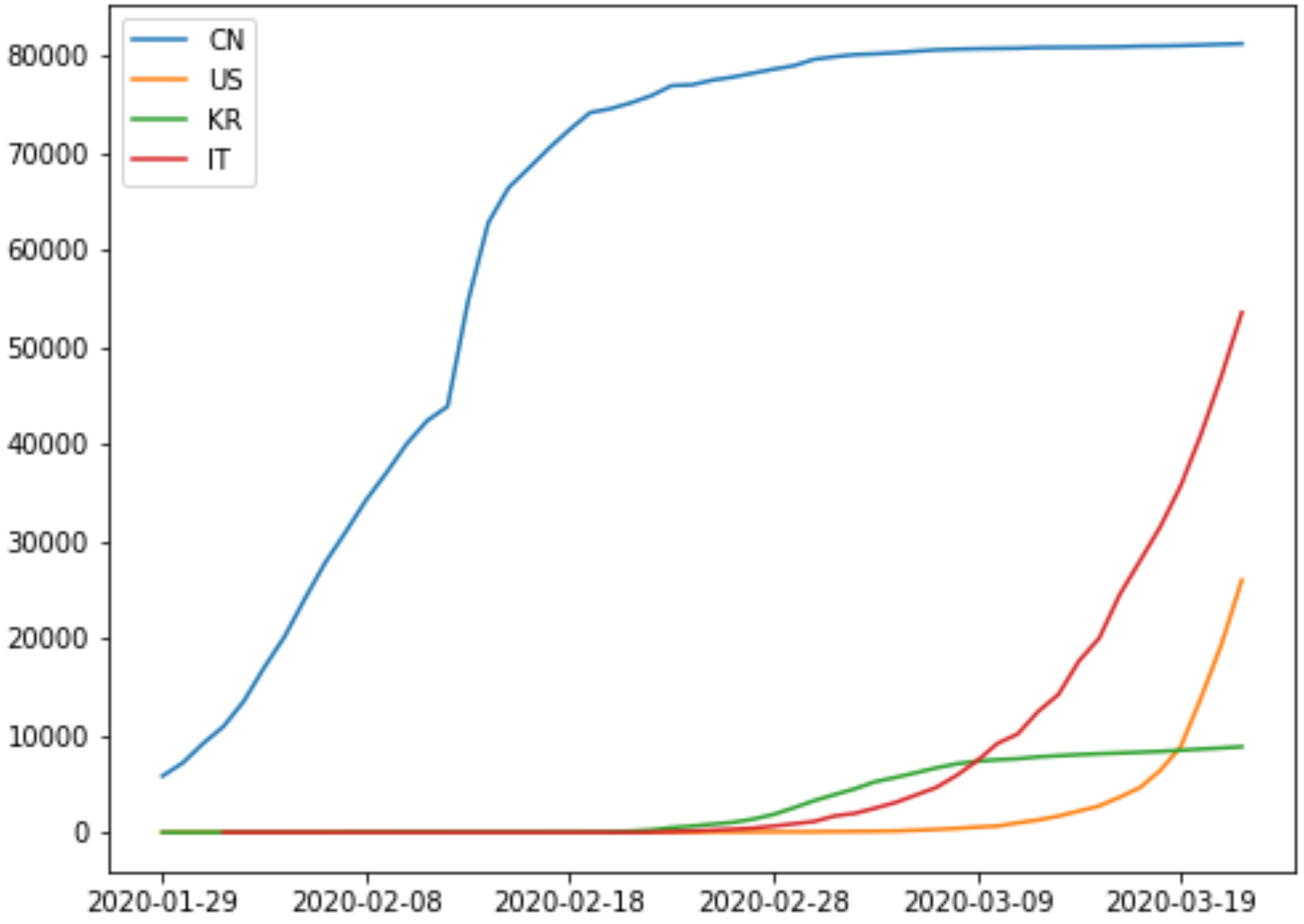
Advanced Data Management (CSCI 490/680)

Time Series Data

Dr. David Koop

Assignment 4

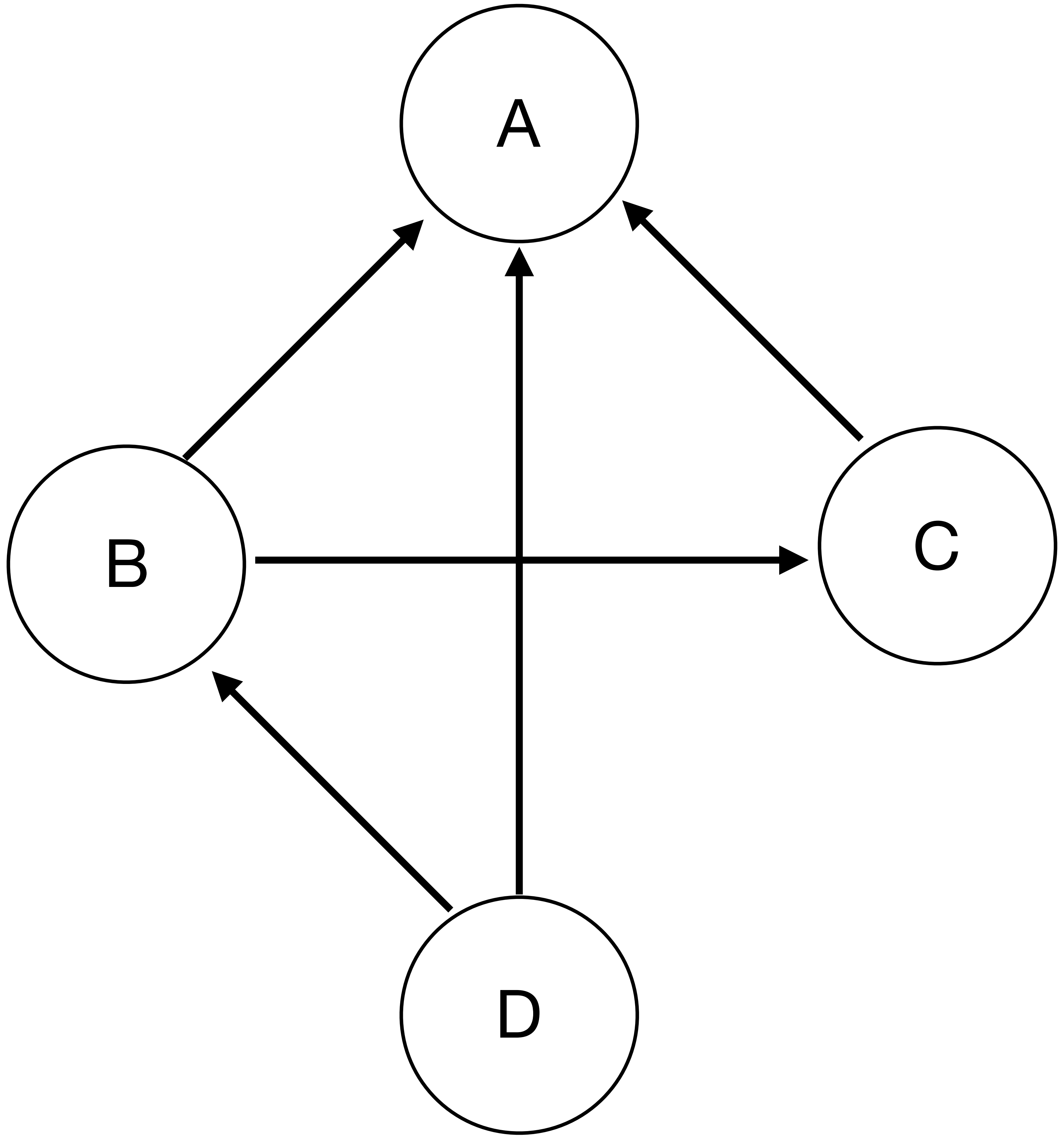


- COVID-19 data
- Data Integration
  - Population
  - Temperature
- Data Fusion:
  - Our World in Data
  - Johns Hopkins
  - Wikipedia
- Questions?

Test 2

- Information
- Online on Blackboard (webcourses.niu.edu)
- Thursday, April 9 from 3:30-4:45pm
- If you have conflicts, **let me know as soon as possible**
- Format:
  - Some multiple choice
  - More short answer/free response
- Focus on topics since the first test

Graphs

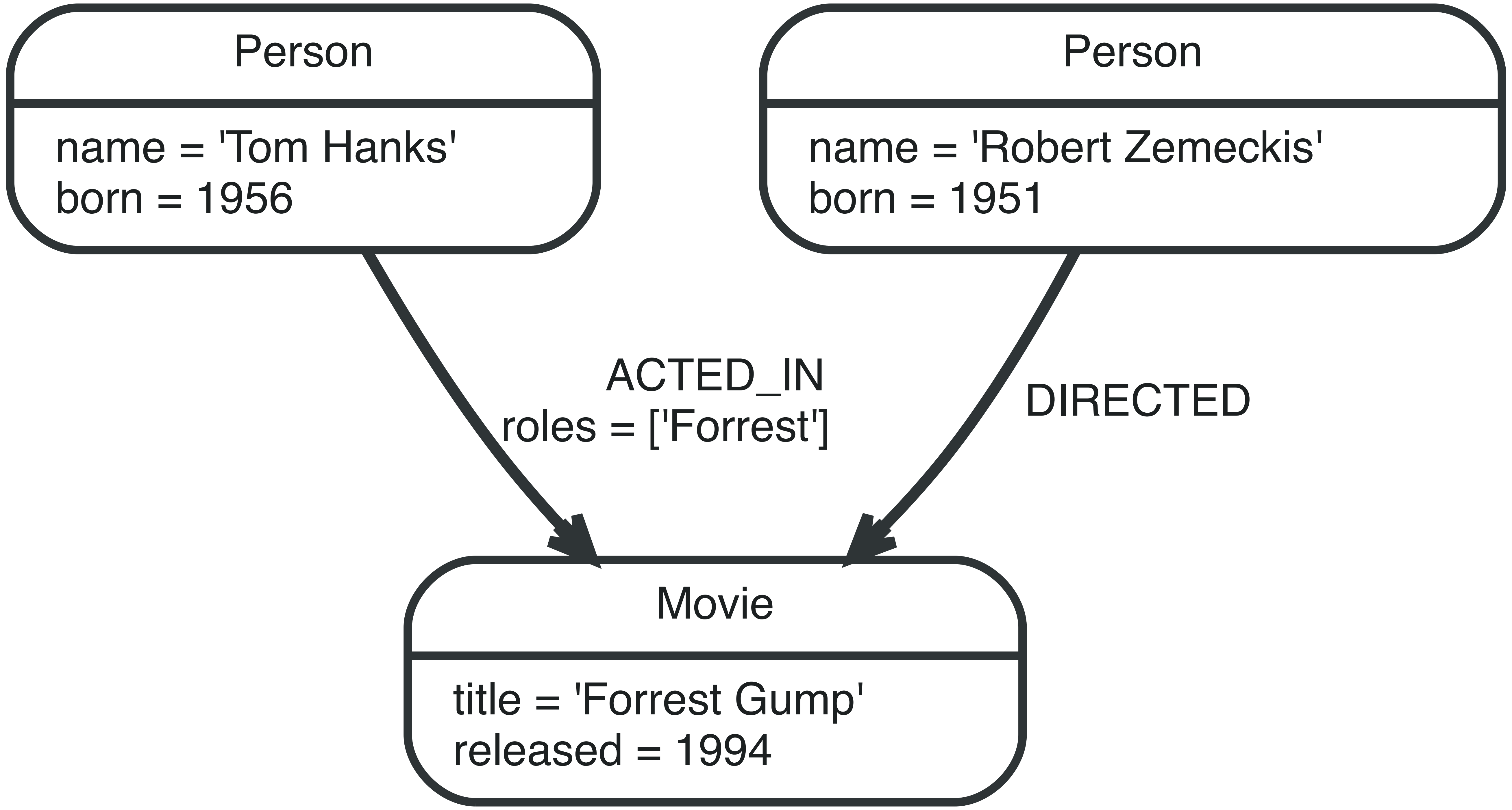


- In computing, a **graph** is an abstract **data structure** that represents set objects and their relationships as **vertices** and **edges/links**, and supports a number of graph-related **operations**
- Objects (nodes): `{A,B,C,D}`
- Relationships (edges): `{(D,B),(D,A),(B,C),(B,A),(C,A)}`
- Operation: shortest path from `D` to `A`

[K. Salama, 2016]

Graphs with Properties

- Each vertex or edge may have properties associated with it
- May include identifiers or classes



[neo4j]

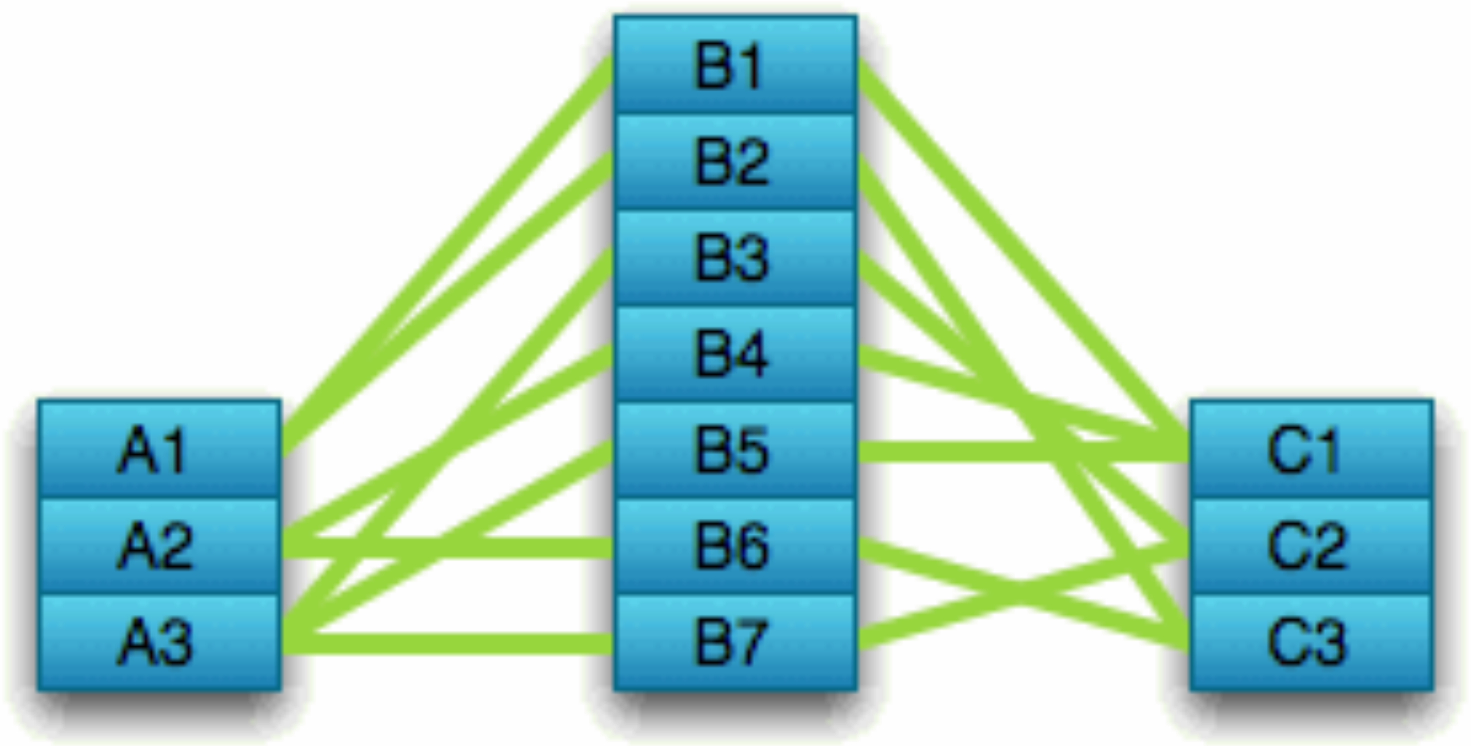
What is a Graph Database?

- A database with an explicit graph structure
- Each node knows its adjacent nodes
- As the number of nodes increases, the cost of a local step (or hop) remains the same
- Plus an Index for lookups

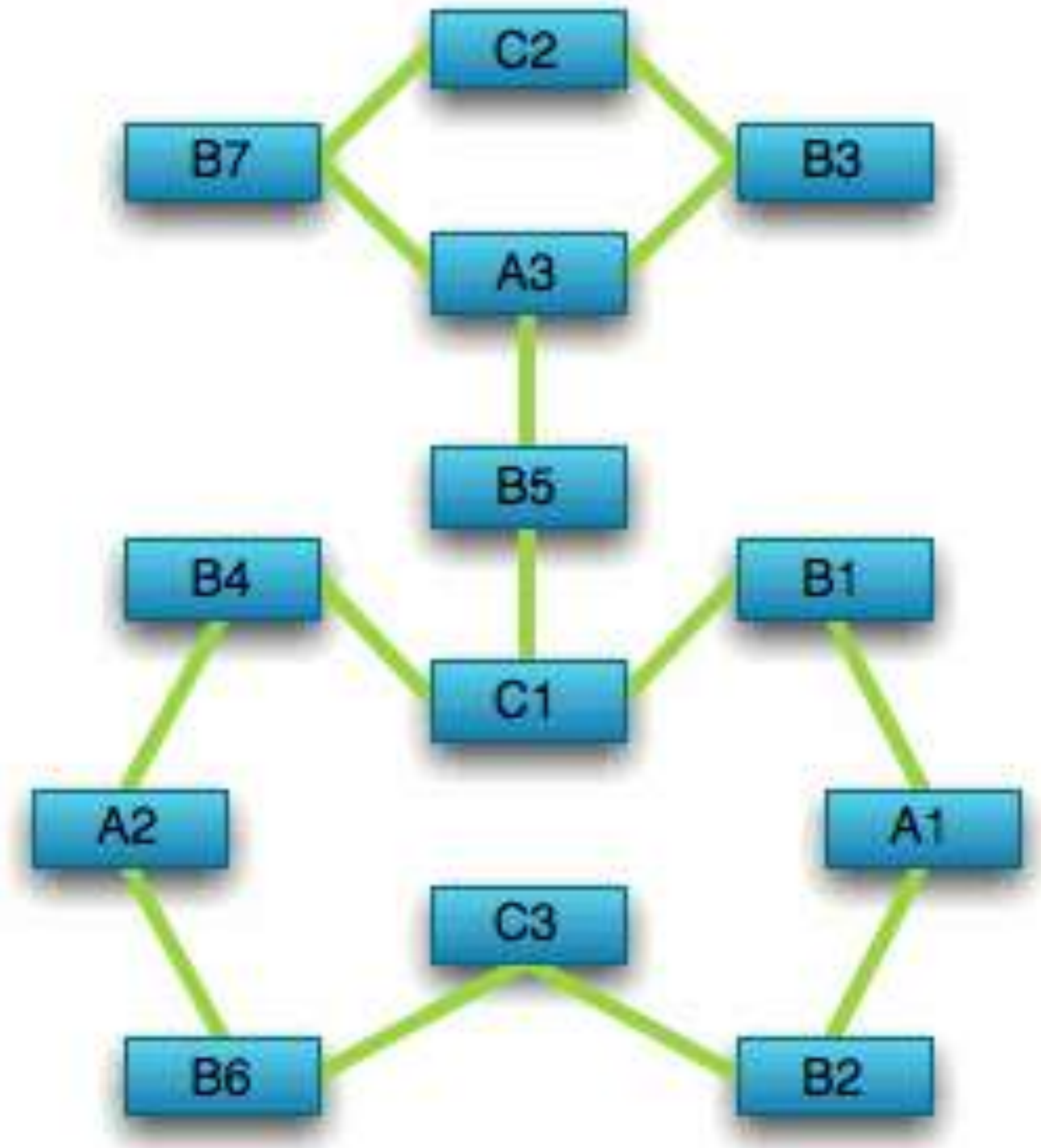
[M. De Marzi, 2012]

Graph Databases Compared to Relational Databases

Optimized for aggregation



Optimized for connections

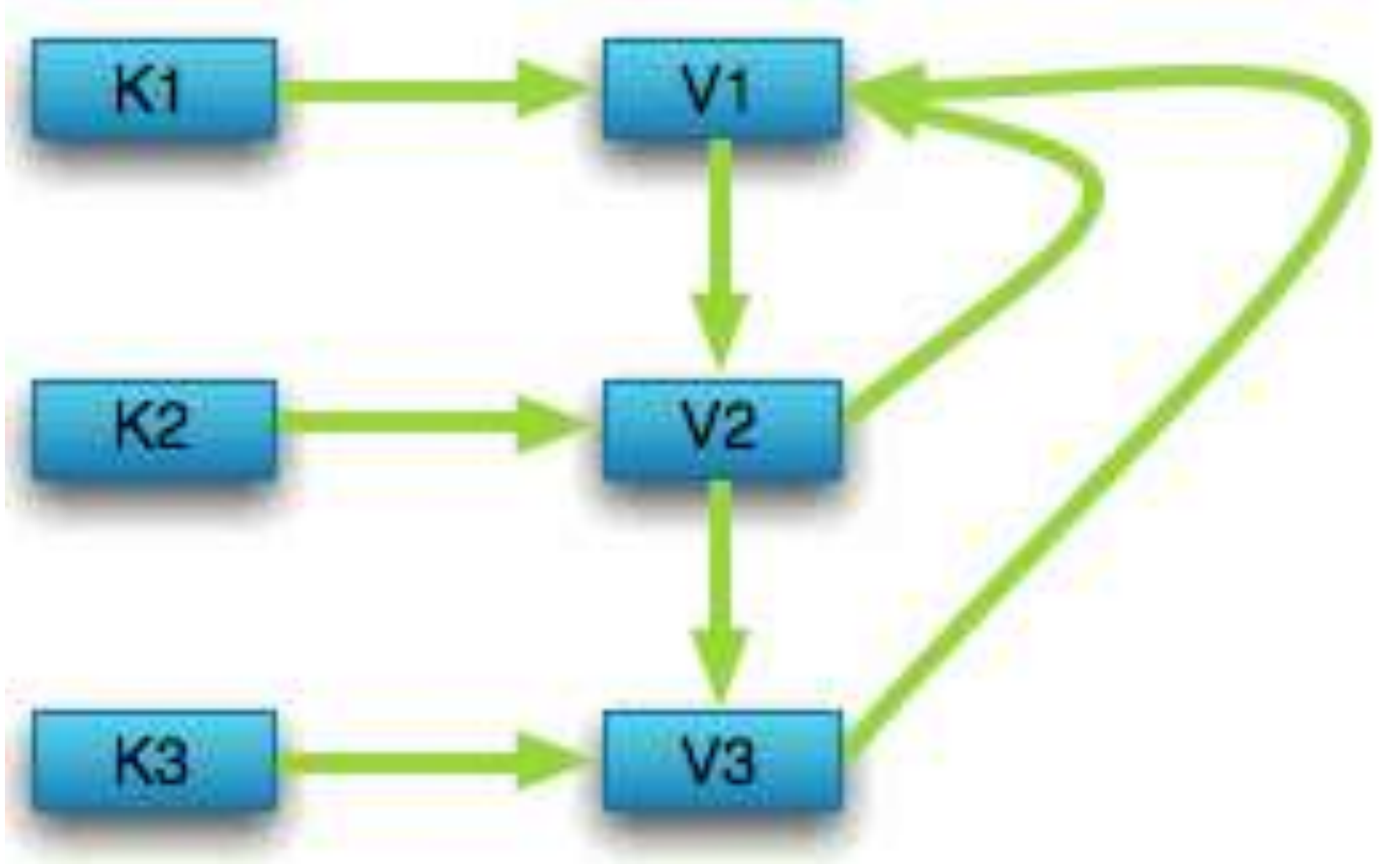


[M. De Marzi, 2012]

Graph Databases Compared to Key-Value Stores

Optimized for simple look-ups

Optimized for traversing connected data



[M. De Marzi, 2012]

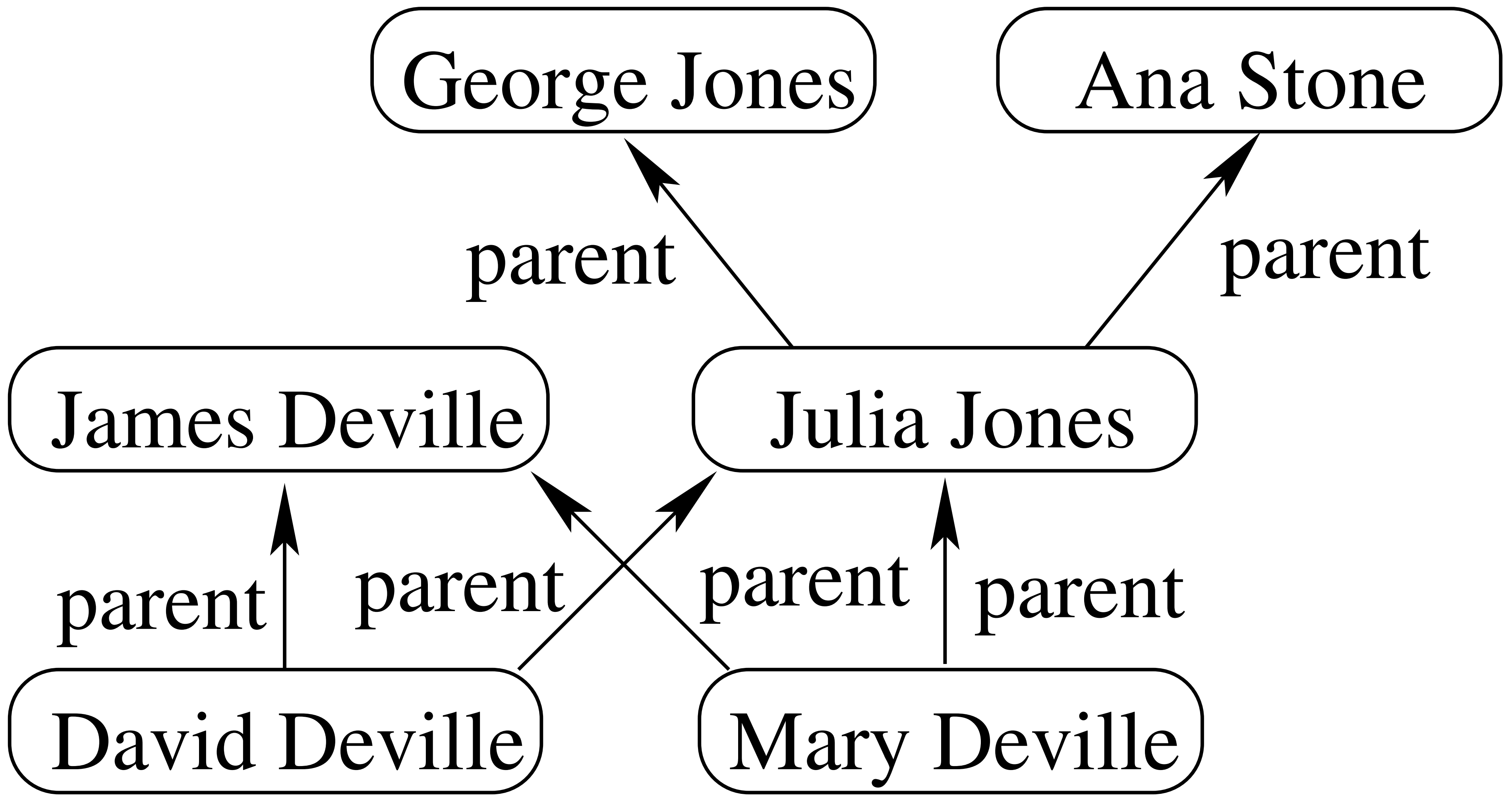
Storing and Traversing Graphs

- Storage:
  - Adjacency List: nodes store their neighbors
  - Incidence List: nodes store edges and edges store incident nodes
  - Adjacency Matrix: adjacency list in matrix form (rows & cols are nodes)
  - Incidence Matrix: rows are vertices, columns are **edges**
- Traversal:
  - Breadth-first Search
  - Depth-first Search

Graph Models: Relational Model

NAME	LASTNAME
George	Jones
Ana	Stone
Julia	Jones
James	Deville
David	Deville
Mary	Deville

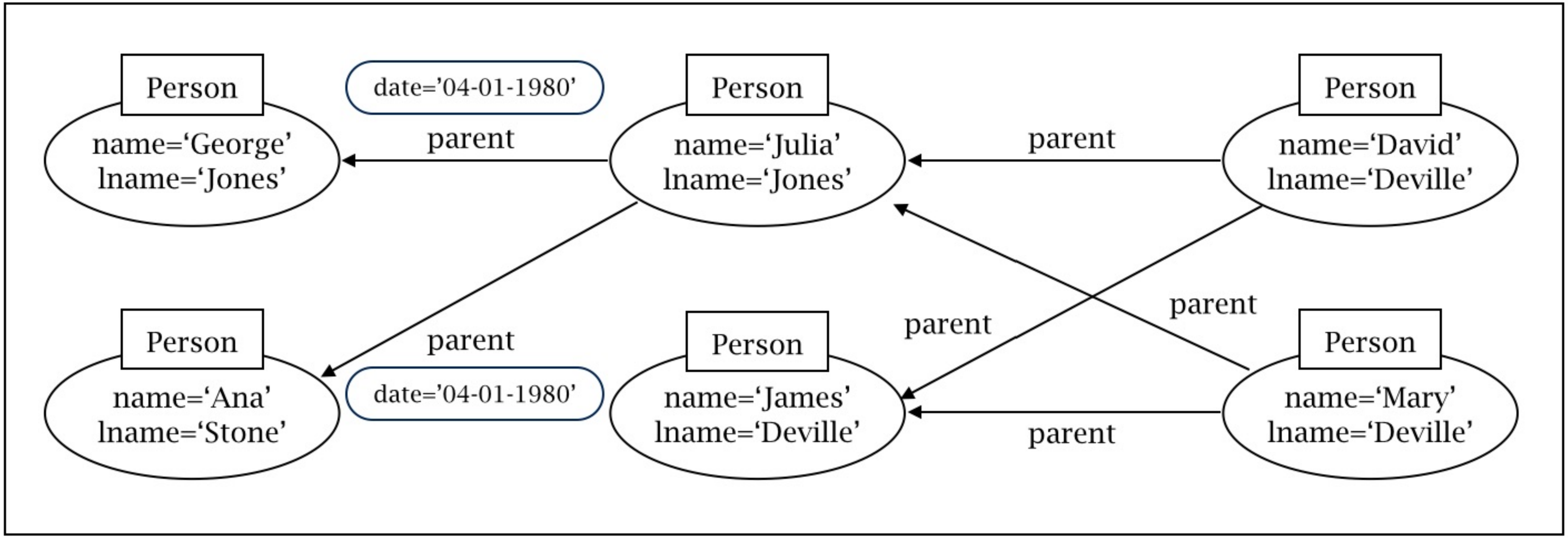
PERSON	PARENT
Julia	George
Julia	Ana
David	James
David	Julia
Mary	James
Mary	Julia



[R. Angles and C. Gutierrez, 2017]

Property Graph Model (Cypher in neo4j)

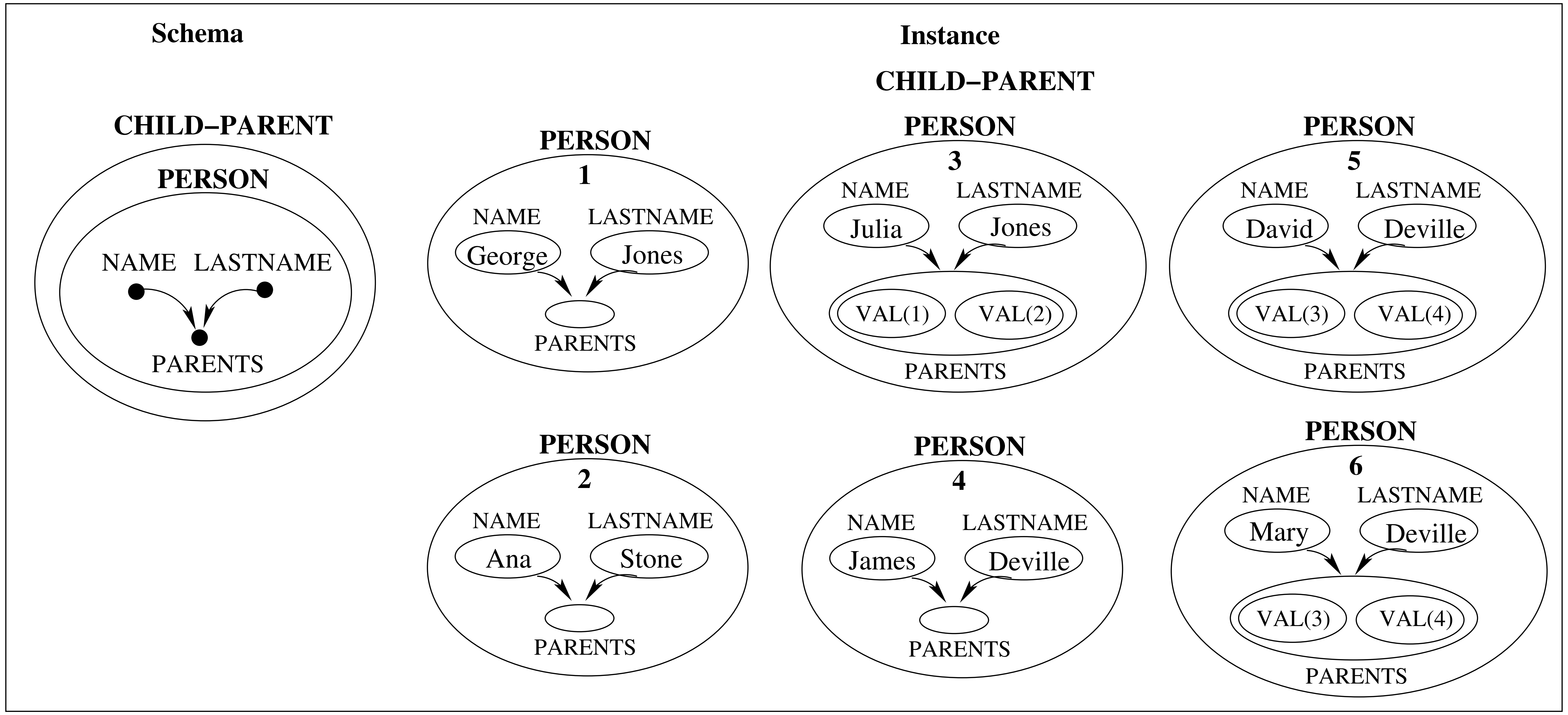
- Directed, labelled, attributed multigraph
- Properties are **key/value pairs** that represent metadata for nodes and edges



[R. Angles and C. Gutierrez, 2017]

Hypergraph Model (Groovy)

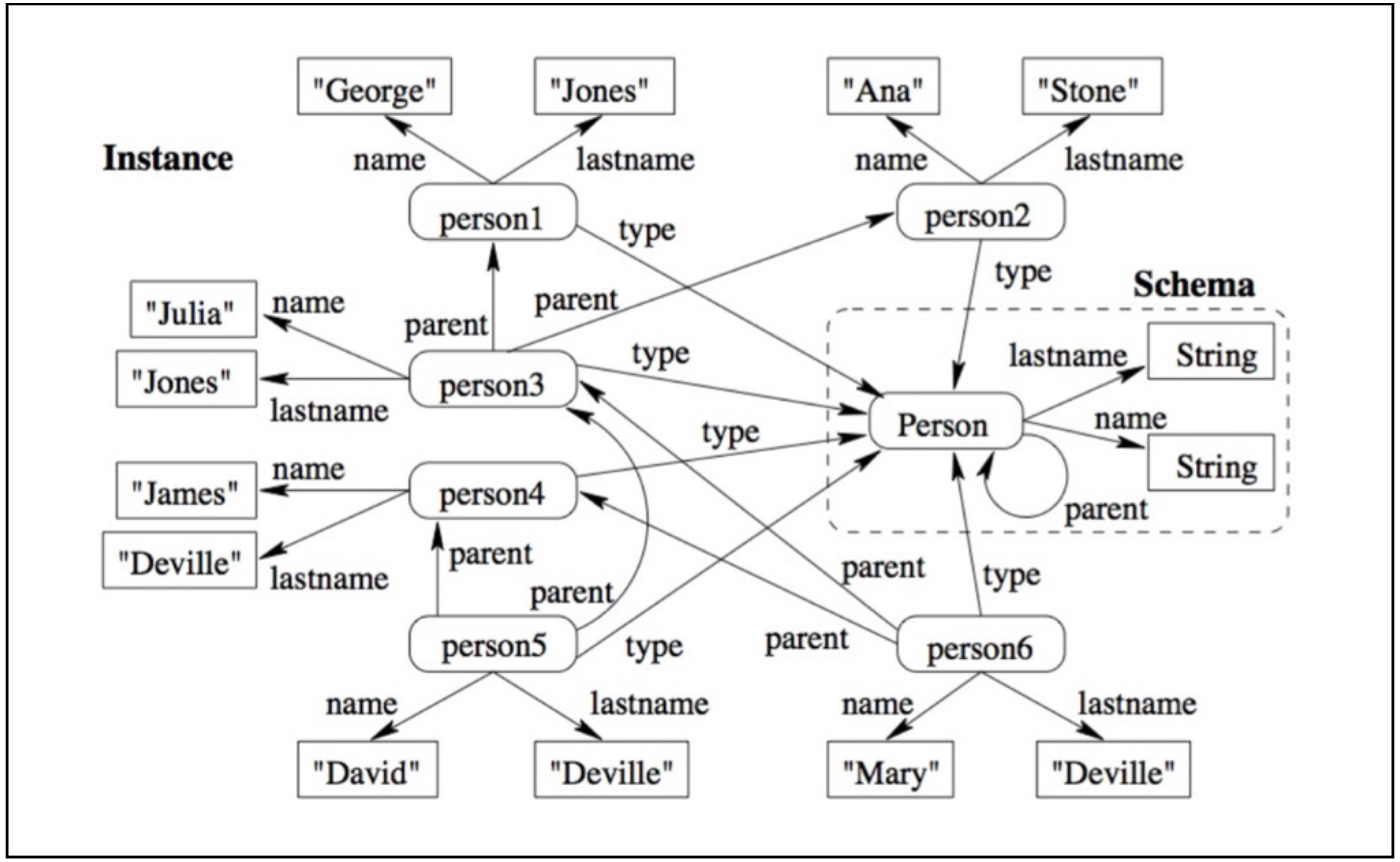
- Notion of edge is extended to **hyperedge**, which relates an arbitrary set of nodes
- Hypergraphs allow the definition of complex objects (undirected), functional dependencies (directed), object-ID and (multiple) structural inheritance



[R. Angles and C. Gutierrez, 2017]

RDF (Triple) Model

- Interconnect resources in an extensible way using graph-like structure for data
- Schema and instance are **mixed** together
- SPAQL to query
- Semantic web



[R. Angles and C. Gutierrez, 2017]

Graph Query Languages: Cypher

- Implemented by neo4j system
- Expresses reachability queries via path expressions
  - `p = (a)-[:knows*]->(b):` nodes from `a` to `b` following `knows` edges
- ```
  START x=node:person(name="John")
  MATCH (x)-[:friend]->(y)
  RETURN y.name
  ```

[R. Angles and C. Gutierrez, 2017]
  ```

Graph Query Languages: SPARQL (RDF)

- Uses SELECT-FROM-WHERE pattern like SQL
-
```
SELECT ?N
FROM <http://example.org/data.rdf>
WHERE { ?X rdf:type voc:Person . ?X voc:name ?N }
```

[R. Angles and C. Gutierrez, 2017]

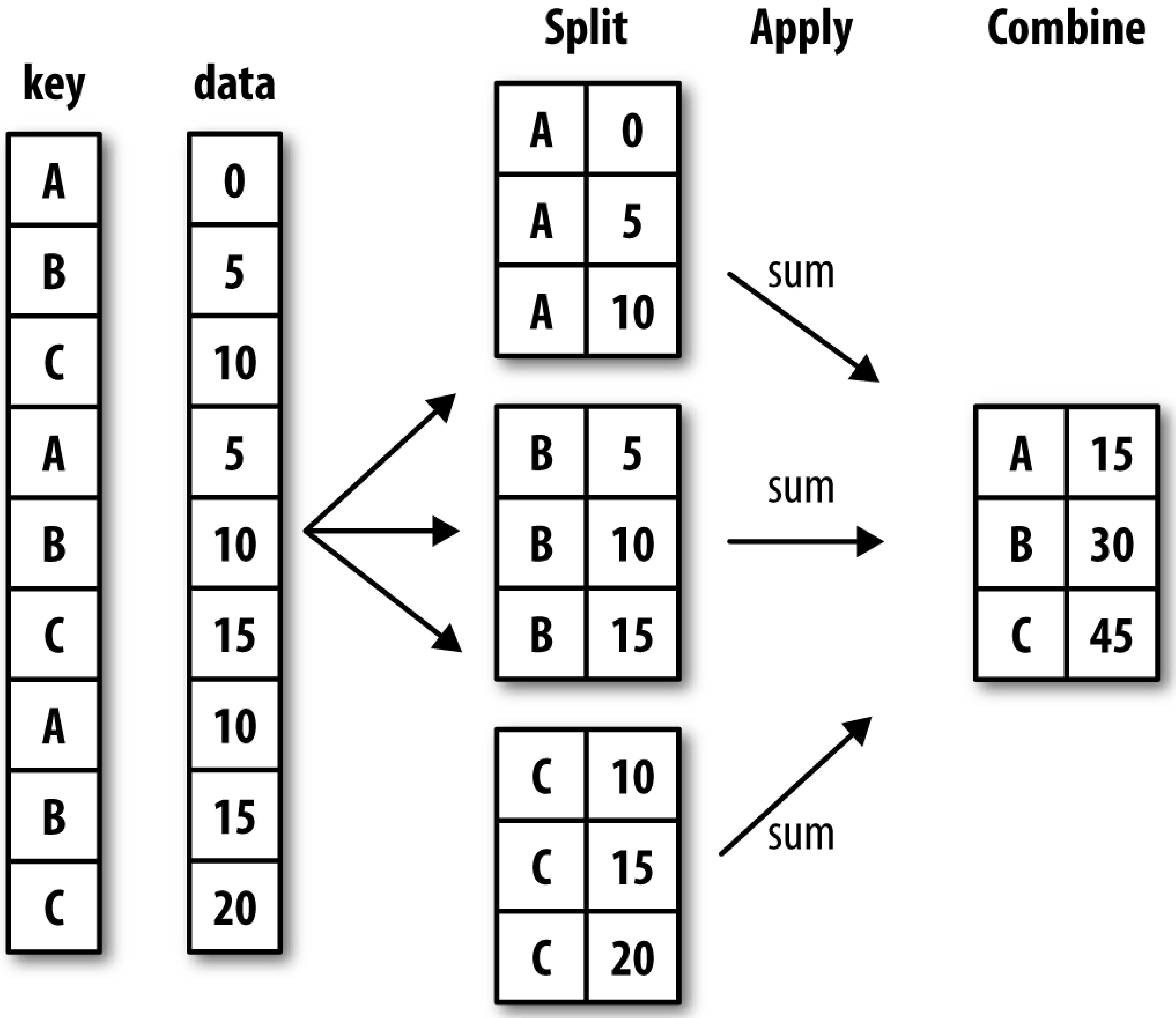
Aggregation

Split-Apply-Combine

- Coined by H. Wickham, 2011
- Similar to Map (split+apply) Reduce (combine) paradigm
- The Pattern:
  1. **Split** the data by some grouping variable
  2. **Apply** some function to each group independently
  3. **Combine** the data into some output dataset
- The apply step is usually one of :
  - Aggregate
  - Transform
  - Filter

[T. Brandt]

Split-Apply-Combine



[W. McKinney, Python for Data Analysis]

Splitting by Variables

name	age	sex
John	13	Male
Mary	15	Female
Alice	14	Female
Peter	13	Male
Roger	14	Male
Phyllis	13	Female

.(sex)

name	age	sex
John	13	*Male*
Peter	13	*Male*
Roger	14	*Male*

name	age	sex
Mary	15	*Female*
Alice	14	*Female*
Phyllis	13	*Female*

.(age)

name	age	sex
John	*13*	Male
Peter	*13*	Male
Phyllis	*13*	Female

name	age	sex
Alice	*14*	Female
Roger	*14*	Male

name	age	sex
Mary	*15*	Female

[H. Wickham, 2011]

Apply+Combine: Counting

.(sex)

sex	value
Male	3
Female	3

.(age)

age	value
13	3
14	2
15	1

.(sex, age)

sex	age	value
Male	13	2
Male	14	1
Female	13	1
Female	14	1
Female	15	1

[H. Wickham, 2011]

In Pandas

- `groupby` method creates a `GroupBy` object
- `groupby` doesn't actually compute anything until there is an apply/aggregate step or we wish to examine the groups
- Choose keys (columns) to group by
- `size()` is the count of each group

Aggregation

- Operations:
  - `count()`
  - `mean()`
  - `sum()`
- May also wish to aggregate only certain subsets
  - Use square brackets with column names
- Can also write your own functions for aggregation and pass then to `agg` function
  - ```
    def peak_to_peak(arr):
        return arr.max() - arr.min()
    grouped.agg(peak_to_peak)
    ```
    ```

Optimized groupby methods

Function name	Description
`count`	Number of non-NA values in the group
`sum`	Sum of non-NA values
`mean`	Mean of non-NA values
`median`	Arithmetic median of non-NA values
`std, var`	Unbiased (n – 1 denominator) standard deviation and variance
`min, max`	Minimum and maximum of non-NA values
`prod`	Product of non-NA values
`first, last`	First and last non-NA values

[W. McKinney, Python for Data Analysis]

Iterating over groups

- ```
  for name, group in df.groupby('key1'):
      print(name)
      print(group)
  ```
- Can also `.describe()` groups
  ```

Apply: Generalized methods

```
In [74]: def top(df, n=5, column='tip_pct'):
   ....:     return df.sort_values(by=column)[-n:]

In [75]: top(tips, n=6)
Out[75]:
     total_bill   tip smoker  day    time  size   tip_pct
109       14.31  4.00    Yes  Sat  Dinner     2  0.279525
183       23.17  6.50    Yes  Sun  Dinner     4  0.280535
232       11.61  3.39     No  Sat  Dinner     2  0.291990
67         3.07  1.00    Yes  Sat  Dinner     1  0.325733
178        9.60  4.00    Yes  Sun  Dinner     2  0.416667
172        7.25  5.15    Yes  Sun  Dinner     2  0.710345

In [76]: tips.groupby('smoker').apply(top)
Out[76]:
            total_bill   tip smoker   day    time  size   tip_pct
smoker
No     88        24.71  5.85     No  Thur   Lunch     2  0.236746
       185       20.69  5.00     No   Sun  Dinner     5  0.241663
       51        10.29  2.60     No   Sun  Dinner     2  0.252672
       149        7.51  2.00     No  Thur   Lunch     2  0.266312
       232       11.61  3.39     No   Sat  Dinner     2  0.291990
Yes    109       14.31  4.00    Yes   Sat  Dinner     2  0.279525
       183       23.17  6.50    Yes   Sun  Dinner     4  0.280535
       67         3.07  1.00    Yes   Sat  Dinner     1  0.325733
       178        9.60  4.00    Yes   Sun  Dinner     2  0.416667
       172        7.25  5.15    Yes   Sun  Dinner     2  0.710345
```

[W. McKinney]

Apply

- `tips.groupby('smoker').apply(top)`
- Function is an **argument**
- Function applied on each **row group**
- All row groups glued together using `concat`

Types of GroupBy

- Aggregation: `agg`
  - `n:1` n group values become one value
  - Examples: mean, min, median
- Apply: `apply`
  - `n:m` n group values become m values
  - Most general (could do aggregation or transform with apply)
  - Example: top 5 in each group, filter
- Transform: `transform`
  - `n:n` n group values become n values
  - Cannot mutate the input

Transform Example

```
In [76]: df
Out[76]:
   key  value
0    a    0.0
1    b    1.0
2    c    2.0
3    a    3.0
4    b    4.0
5    c    5.0
6    a    6.0
7    b    7.0
8    c    8.0
9    a    9.0
10   b   10.0
11   c   11.0
```

```
In [77]: g = df.groupby('key').value

In [78]: g.mean()
Out[78]:
key
a    4.5
b    5.5
c    6.5
Name: value, dtype: float64

In [79]: g.transform(lambda x: x.mean())
Out[79]:
0     4.5
1     5.5
2     6.5
3     4.5
4     5.5
5     6.5
6     4.5
7     5.5
8     6.5
9     4.5
10    5.5
11    6.5
Name: value, dtype: float64
```

[W. McKinney, Python for Data Analysis]

Transform Example

```
In [76]: df
Out[76]:
   key  value
0    a    0.0
1    b    1.0
2    c    2.0
3    a    3.0
4    b    4.0
5    c    5.0
6    a    6.0
7    b    7.0
8    c    8.0
9    a    9.0
10   b   10.0
11   c   11.0
```

```
In [77]: g = df.groupby('key').value

In [78]: g.mean()
Out[78]:
key
a    4.5
b    5.5
c    6.5
Name: value, dtype: float64

In [79]: g.transform(lambda x: x.mean())
Out[79]:
0     4.5
1     5.5
2     6.5
3     4.5
4     5.5
5     6.5
6     4.5
7     5.5
8     6.5
9     4.5
10    5.5
11    6.5
Name: value, dtype: float64
```

or `g.transform('mean')`

[W. McKinney, Python for Data Analysis]

Normalization

```
def normalize(x):
    return (x - x.mean()) / x.std()
```

```
In [84]: g.transform(normalize)
Out[84]:
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
Name: value, dtype: float64
```

==

```
In [85]: g.apply(normalize)
Out[85]:
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
Name: value, dtype: float64
```

[W. McKinney]

Normalization

```
def normalize(x):
    return (x - x.mean()) / x.std()
```

```
In [84]: g.transform(normalize)
Out[84]:
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
Name: value, dtype: float64
```

==

```
In [85]: g.apply(normalize)
Out[85]:
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
Name: value, dtype: float64
```

```
In [87]: normalized = (df['value'] - g.transform('mean')) / g.transform('std')
```

Fastest: "Unwrapped" group operation

[W. McKinney]

Other Operations

- Quantiles: return values at particular splits
  - Median is a 0.5-quantile
  - `df.quantile(0.1)`
  - also works on groups
- Can return data from group-by without having the keys in the index (`as_index=False`) or use `reset_index` after computing
- Grouped weighted average via apply

Pivot Tables

- Data summarization tool in many spreadsheet programs
- Aggregates a table of data by one or more keys with some keys arranged on rows (`index`), others as columns (`columns`)
- Pandas supports via `pivot_table` method
- `margins=True` gives partial totals
- Can use different aggregation functions via `aggfunc` kwarg

Function name	Description
`values`	Column name or names to aggregate. By default aggregates all numeric columns
`rows`	Column names or other group keys to group on the rows of the resulting pivot table
`cols`	Column names or other group keys to group on the columns of the resulting pivot table
`aggfunc`	Aggregation function or list of functions; `'mean'` by default. Can be any function valid in a groupby context
`fill_value`	Replace missing values in result table
`margins`	Add row/column subtotals and grand total, False by default

[W. McKinney, Python for Data Analysis]

Pivot Tables in Pandas

- `tips`

	total_bill	tip	sex	smoker	day	time	size	tip_pct
0	16.99	1.01	Female	No	Sun	Dinner	2	0.059447
1	10.34	1.66	Male	No	Sun	Dinner	3	0.160542
2	21.01	3.50	Male	No	Sun	Dinner	3	0.166587
3	23.68	3.31	Male	No	Sun	Dinner	2	0.139780
4	24.59	3.61	Female	No	Sun	Dinner	4	0.146808
5	25.29	4.71	Male	No	Sun	Dinner	4	0.186240
6	8.77	2.00	Male	No	Sun	Dinner	2	0.228050

- `tips.pivot_table(index=['sex', 'smoker'])`

		size	tip	tip_pct	total_bill
sex	**smoker**				
Female	**No**	2.592593	2.773519	0.156921	18.105185
	Yes	2.242424	2.931515	0.182150	17.977879
Male	**No**	2.711340	3.113402	0.160669	19.791237
	Yes	2.500000	3.051167	0.152771	22.284500

Pivot Tables with Margins and Aggfunc

- `tips.pivot_table(['size'], index=['sex', 'day'], columns='smoker', aggfunc='sum', margins=True)`

		size		
	smoker	**No**	**Yes**	**All**
sex	**day**			
Female	**Fri**	2.0	7.0	9.0
	Sat	13.0	15.0	28.0
	Sun	14.0	4.0	18.0
	Thur	25.0	7.0	32.0
Male	**Fri**	2.0	8.0	10.0
	Sat	32.0	27.0	59.0
	Sun	43.0	15.0	58.0
	Thur	20.0	10.0	30.0
All		151.0	93.0	244.0

Crosstabs

- `crosstab` is a special case for group frequencies (`aggfunc='count'`)

```
In [293]: pd.crosstab(data.Gender, data.Handedness, margins=True)
Out[293]:
Handedness  Left-handed  Right-handed  All
Gender
Female                1             4    5
Male                  2             3    5
All                   3             7   10
```

- Tipping example
- Also see the Federal Election Database example in the book

Crosstabs

- `pd.crosstab([tips.time, tips.day], tips.smoker, margins=True)`

	smoker	No	Yes	All
time	**day**			
Dinner	**Fri**	3	9	12
	Sat	45	42	87
	Sun	57	19	76
	Thur	1	0	1
Lunch	**Fri**	1	6	7
	Thur	44	17	61
All		151	93	244

- or… `tips.pivot_table('total_bill',index=['time', 'day'], columns=['smoker'], aggfunc='count', margins=True, fill_value=0)`

Time Series Data

What is time series data?

- Technically, it's normal tabular data with a timestamp attached
- But… we have observations of the same values over time, usually **in order**
- This allows more analysis
- Example: Web site database that tracks the last time a user logged in
  - 1: Keep an attribute `lastLogin` that is **overwritten** every time user logs in
  - 2: **Add a new row** with login information every time the user logs in
  - Option 2 takes more storage, but we can also do a lot more analysis!

Time Series Databases

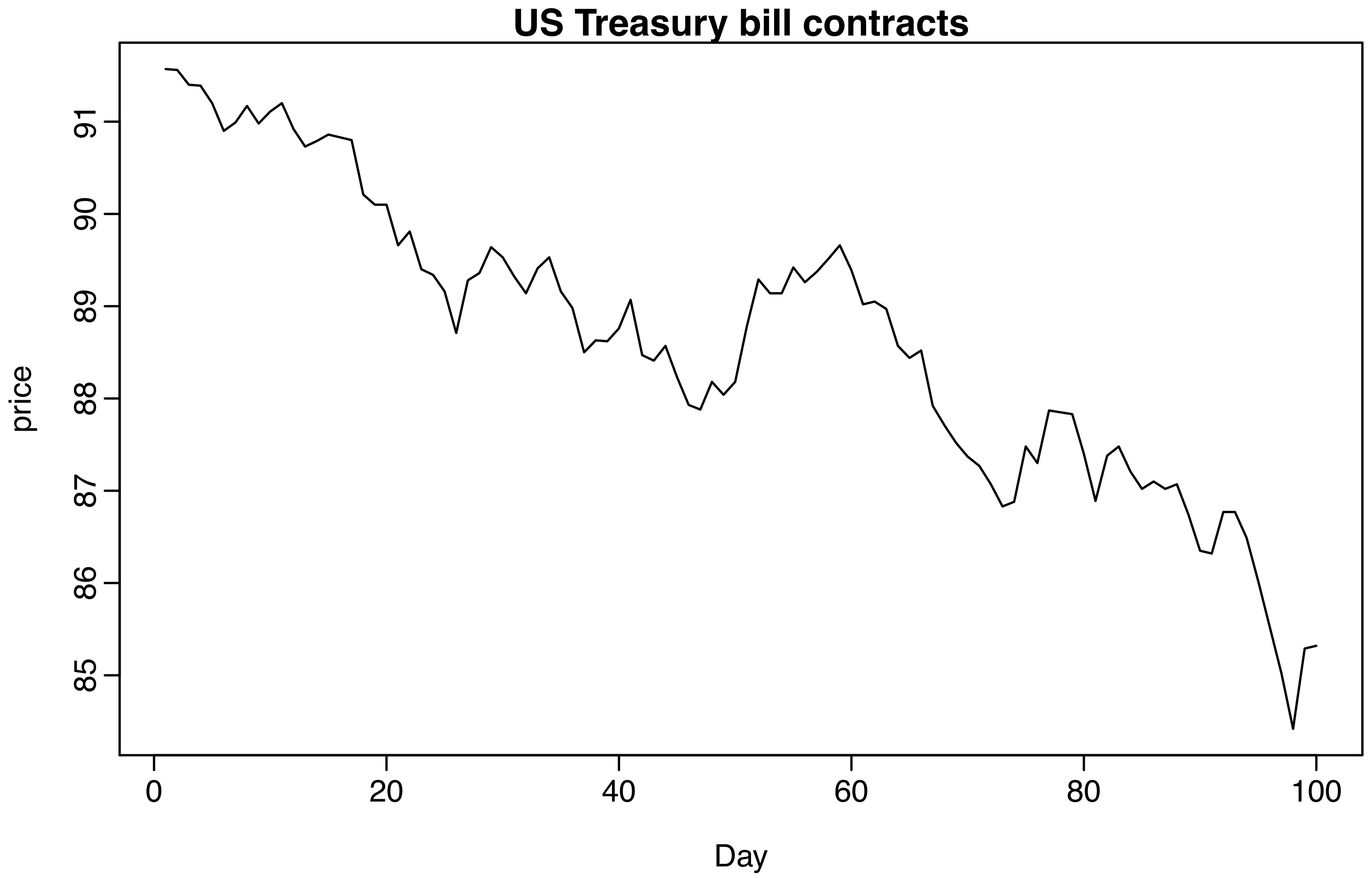
- Most time series data is heavy inserts, few updates
- Also analysis tends to be on ordered data with trends, prediction, etc.
- Can also consider **stream** processing
- Focus on time series allows databases to specialize
- Examples:
  - InfluxDB (noSQL)
  - TimescaleDB (SQL-based)

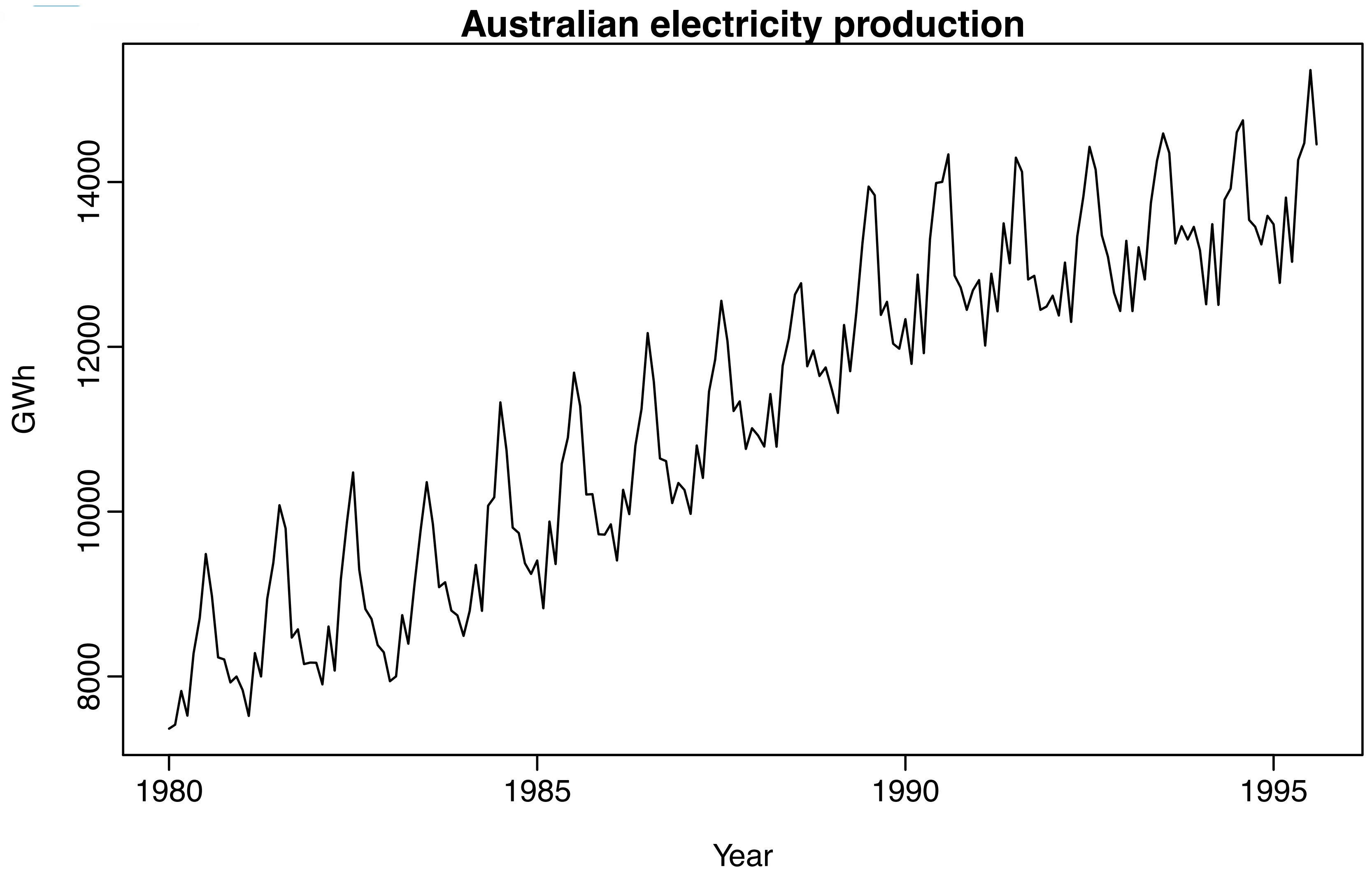
Features of Time Series Data

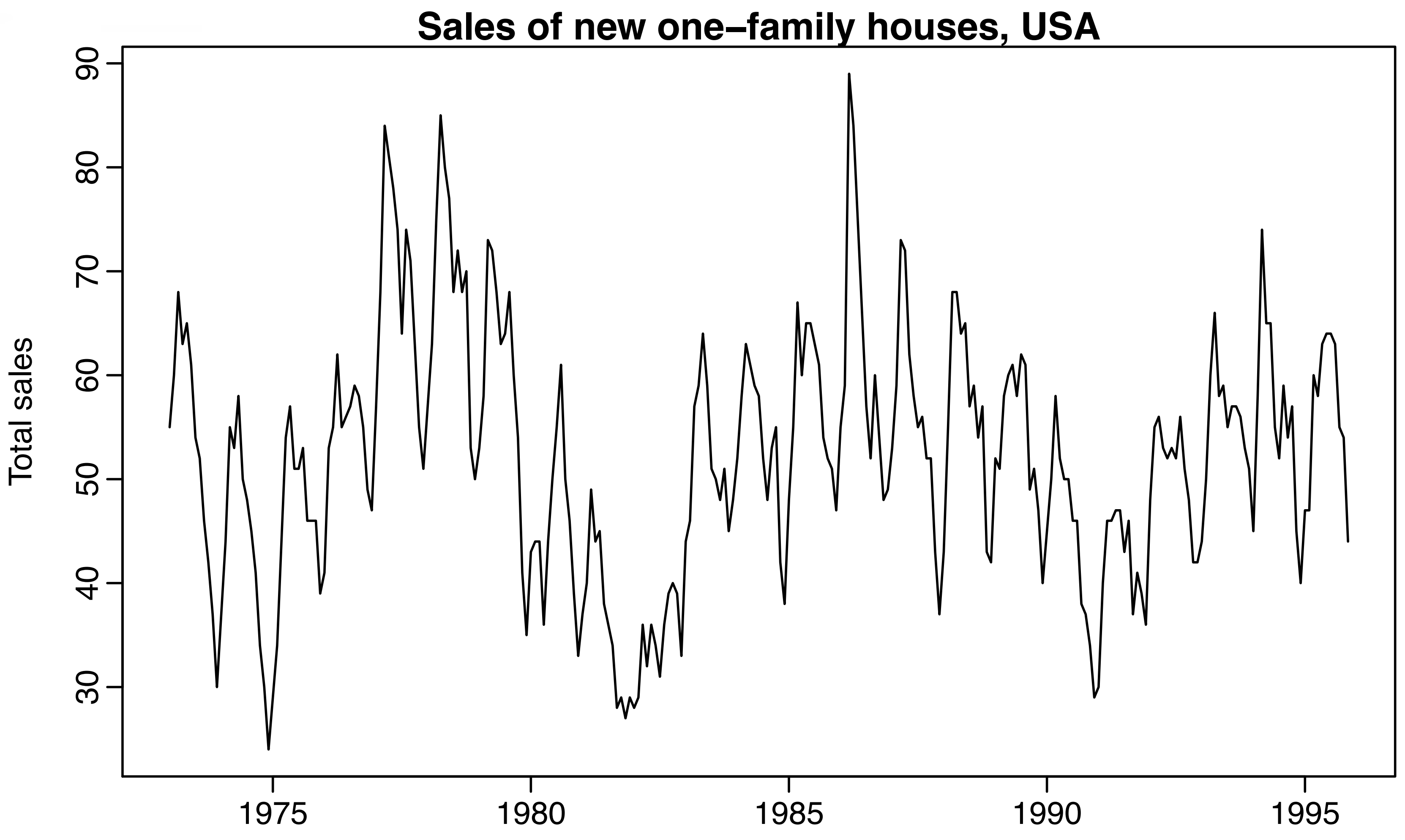
- Trend: long-term increase or decrease in the data
- Seasonal Pattern: time series is affected by seasonal factors such as the time of the year or the day of the week (fixed and of known frequency)
- Cyclic Pattern: rises and falls that are not of a fixed frequency
- Stationary: no predictable patterns (roughly horizontal with constant variance)
  - White noise series is stationary
  - Will look the basically the same whenever you observe it

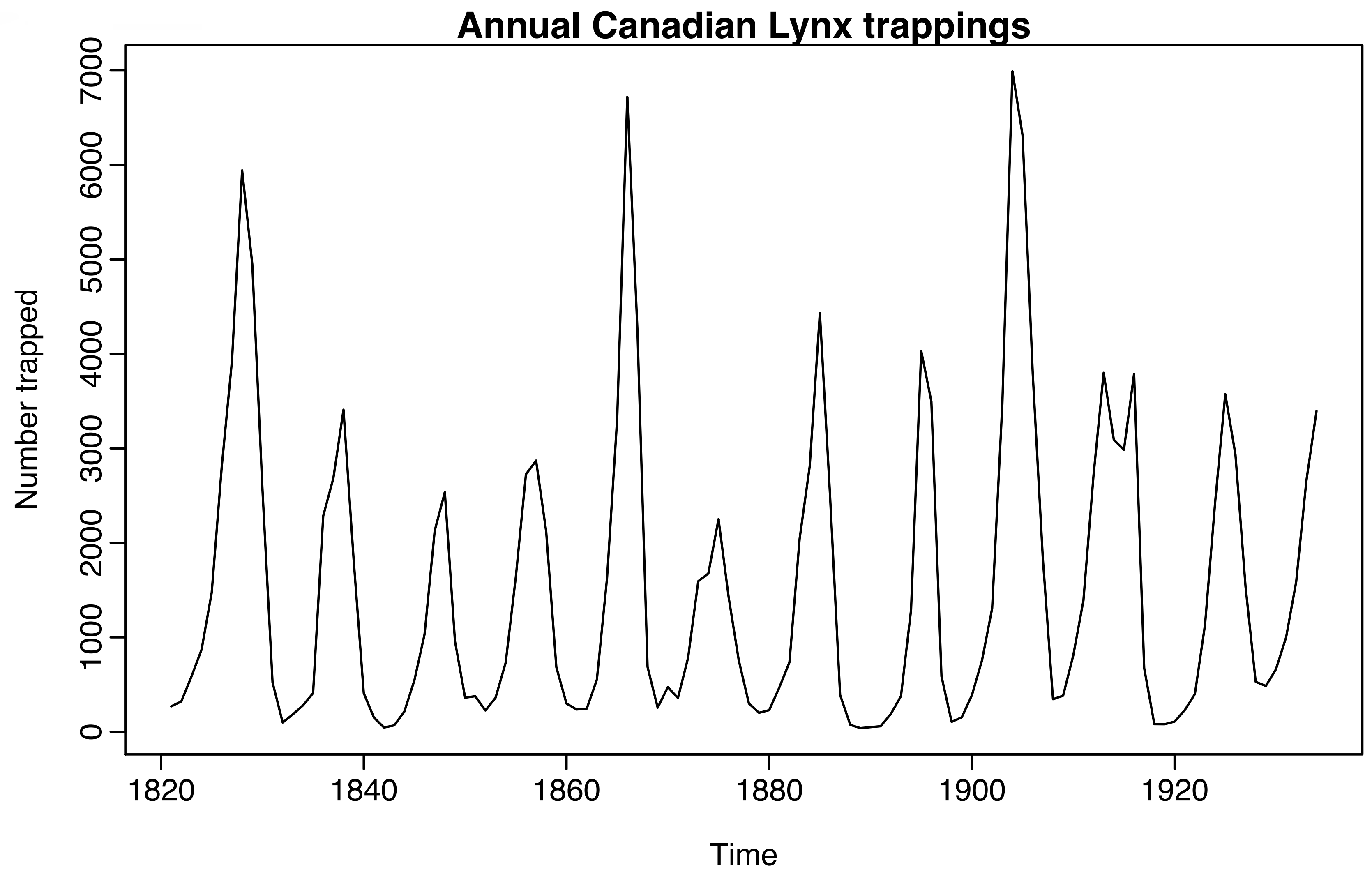
[Hyndman and Athanosopoulos]

Examples





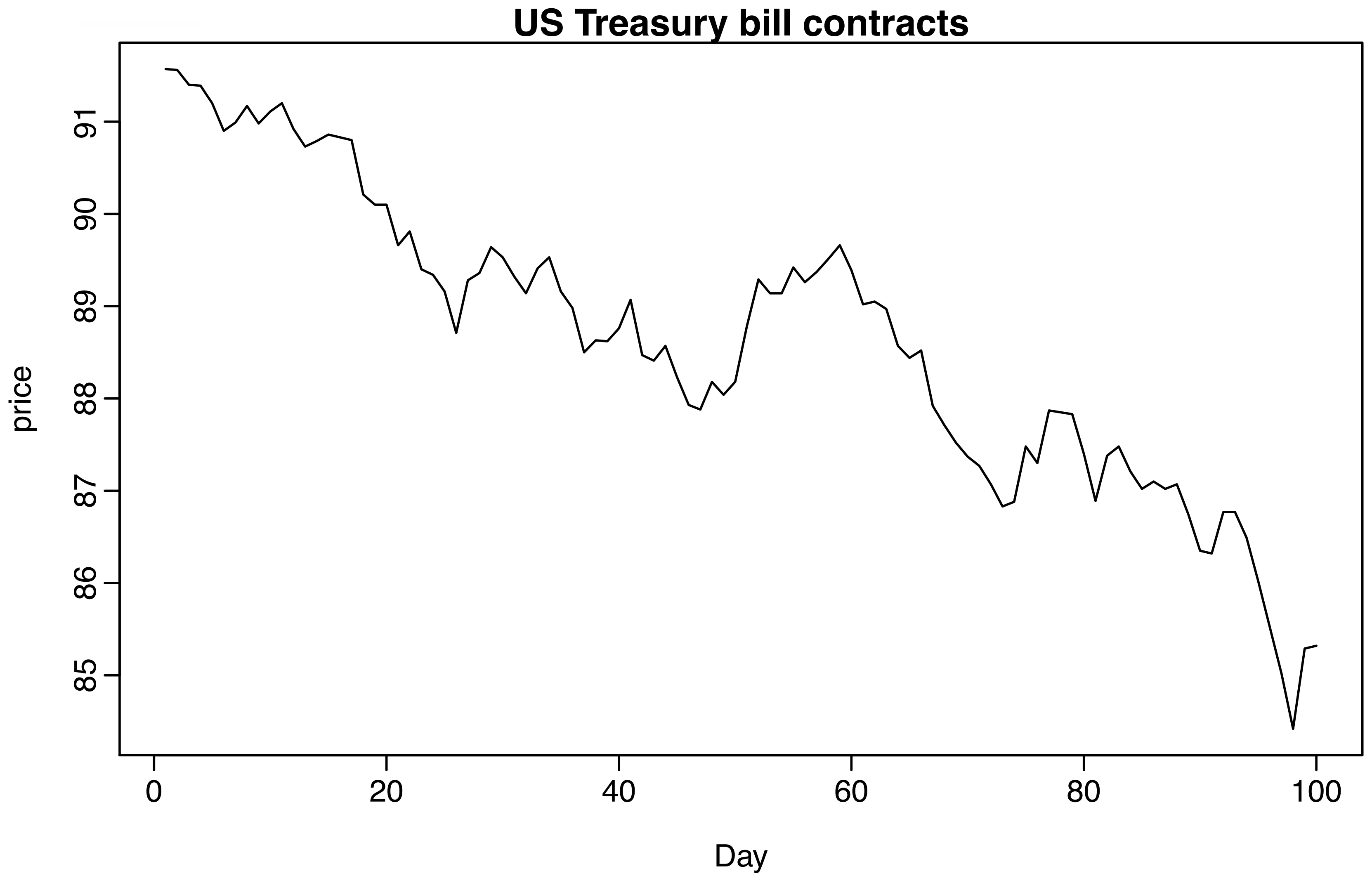


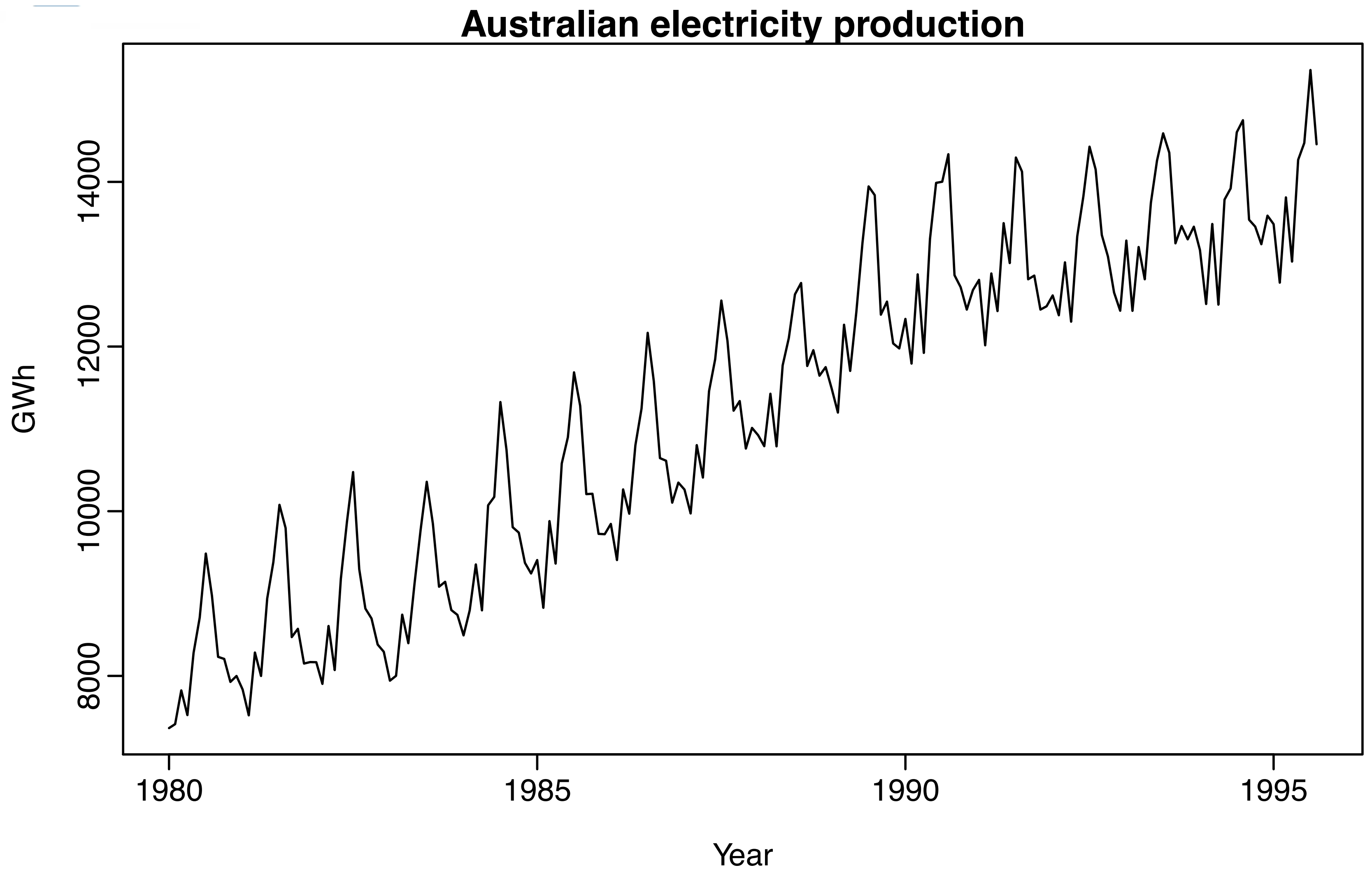


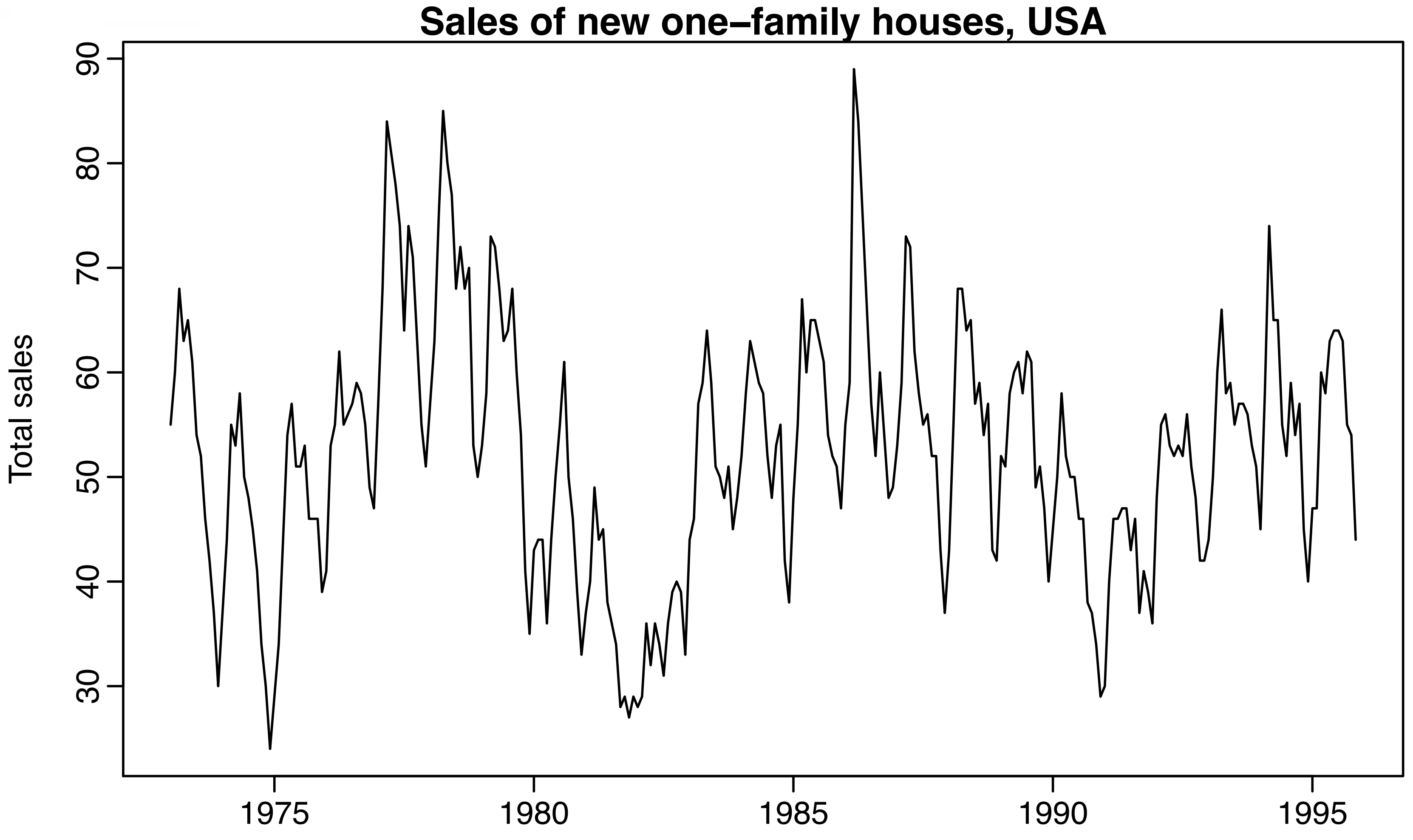
[R. J. Hyndman]

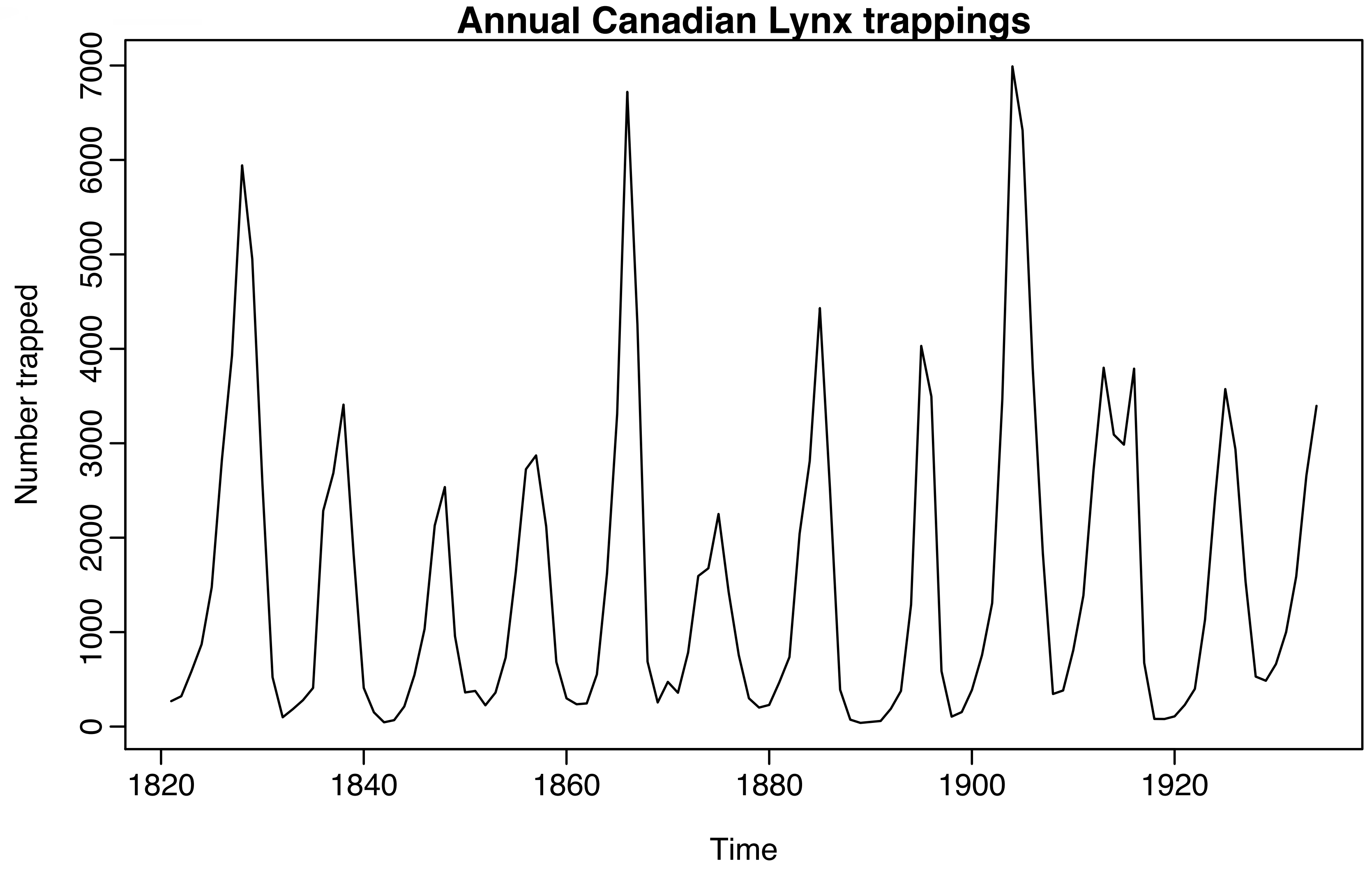
Examples

Trend





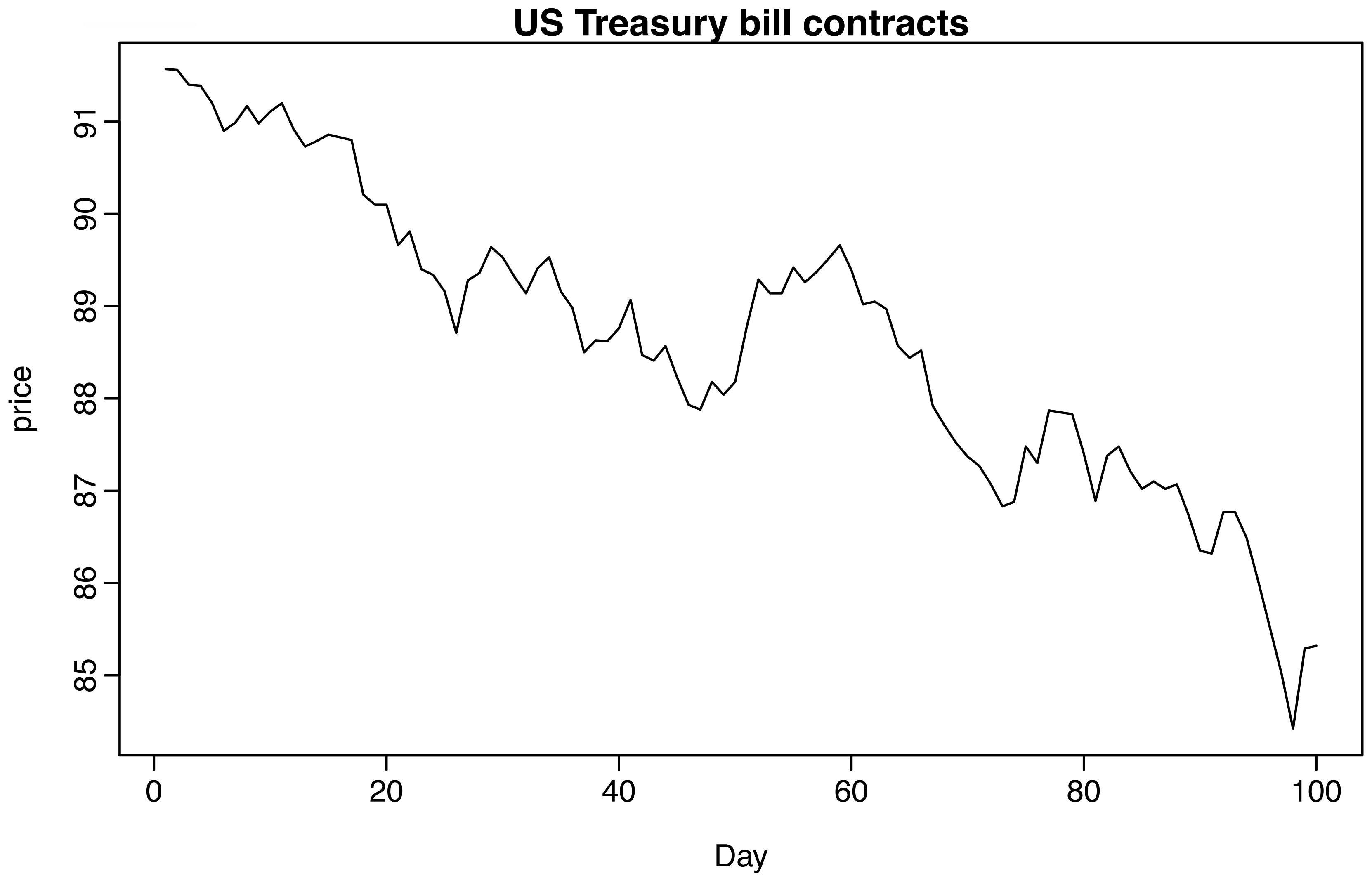


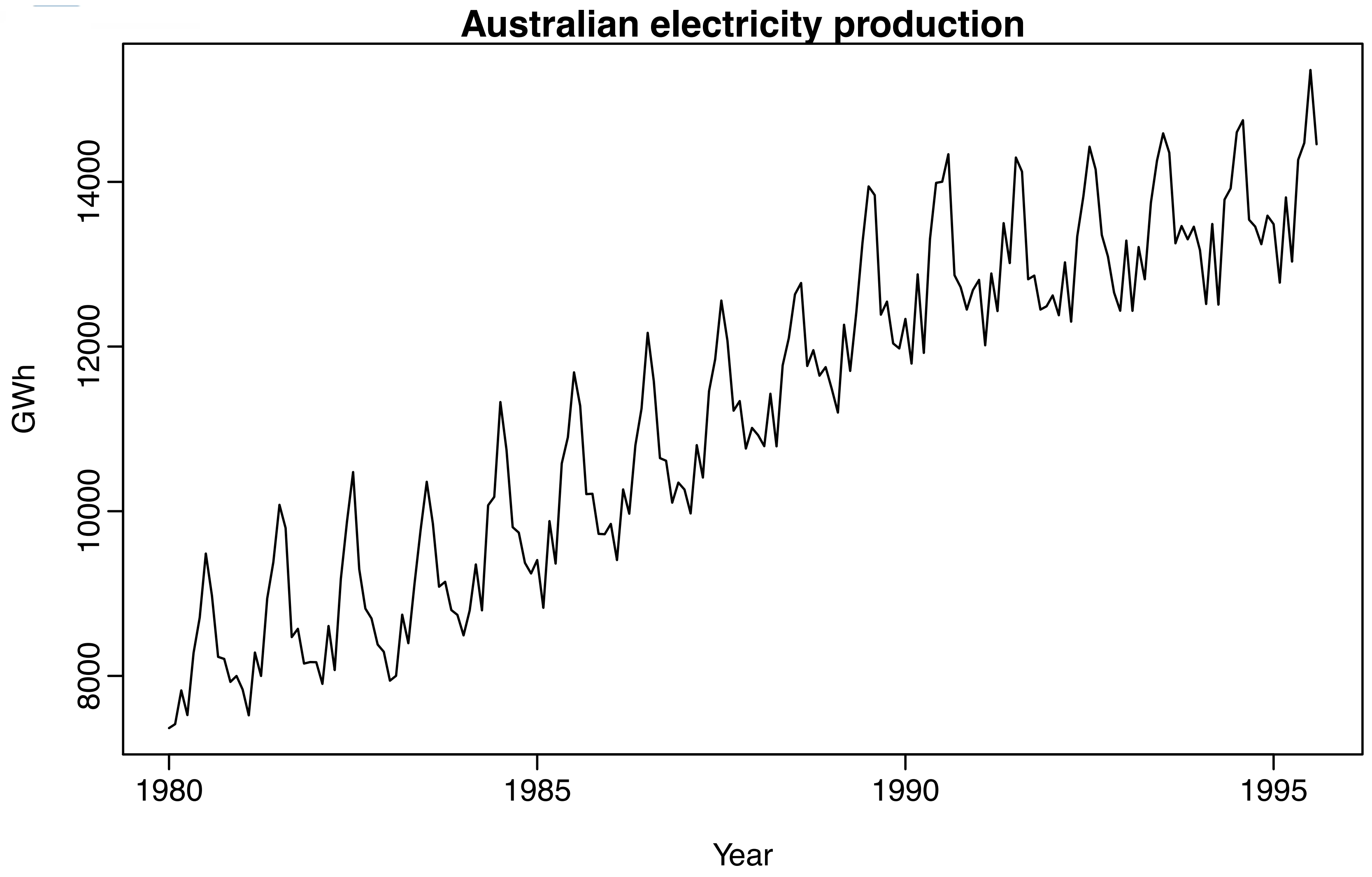


[R. J. Hyndman]

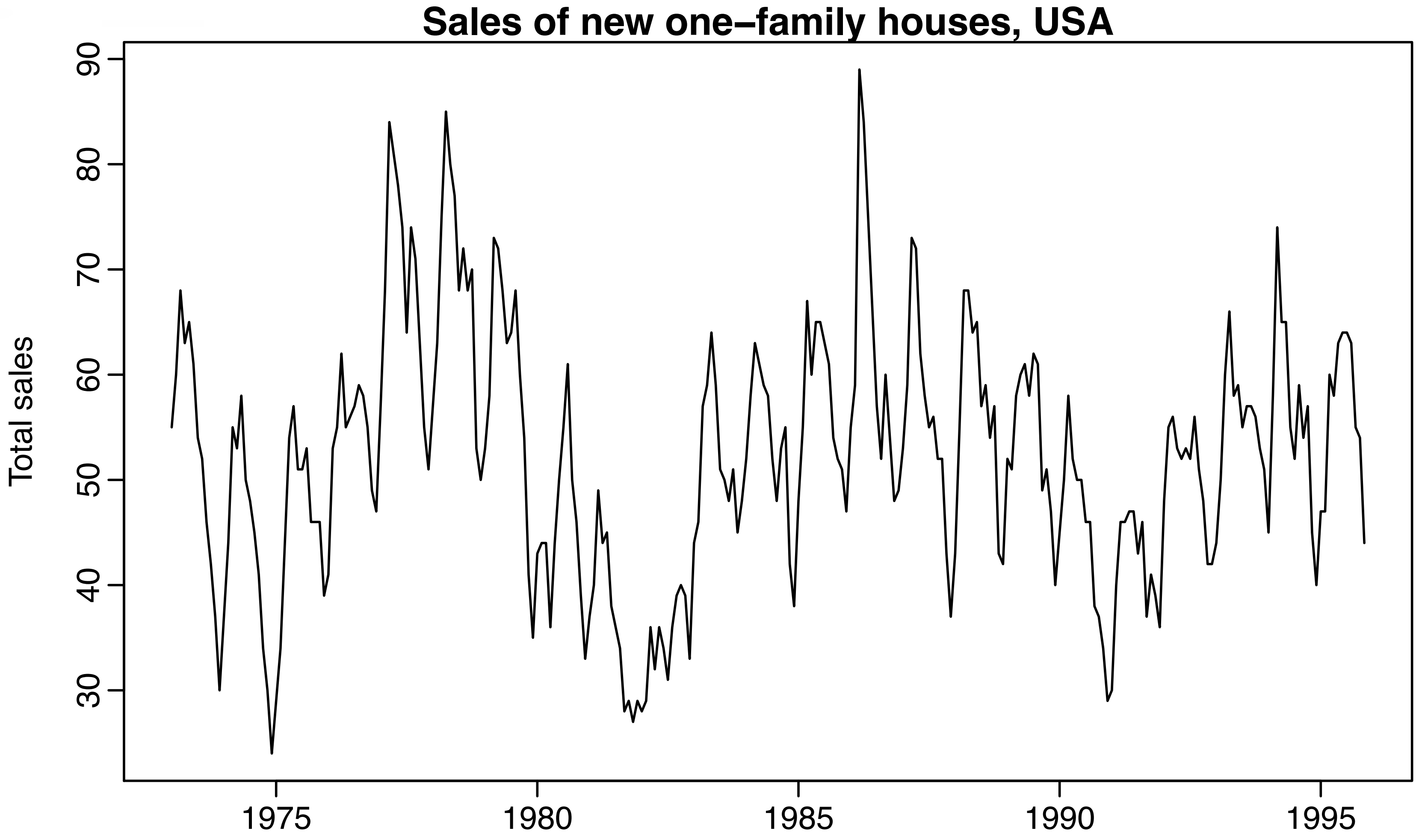
Examples

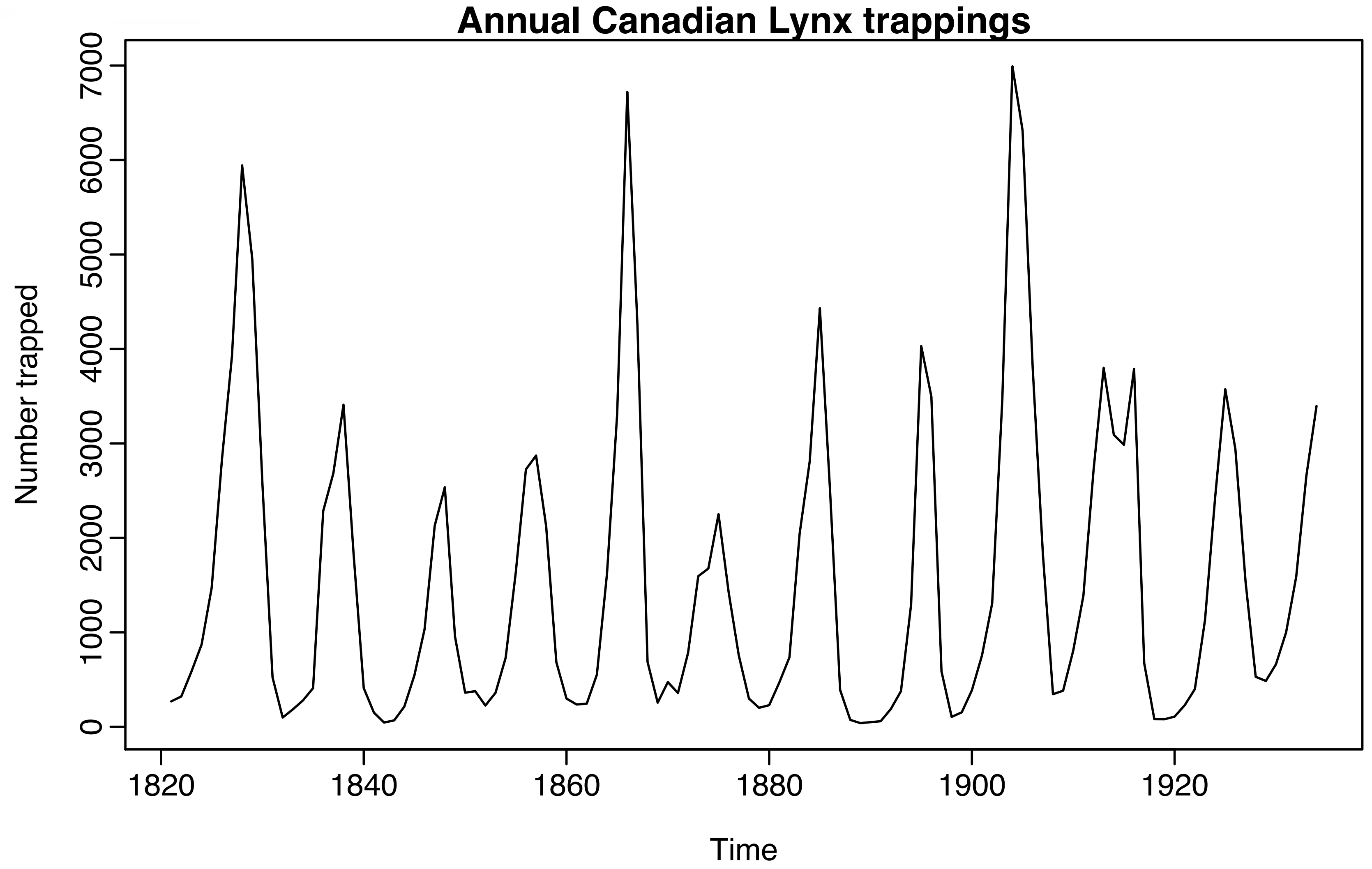
Trend





Trend +
Seasonality

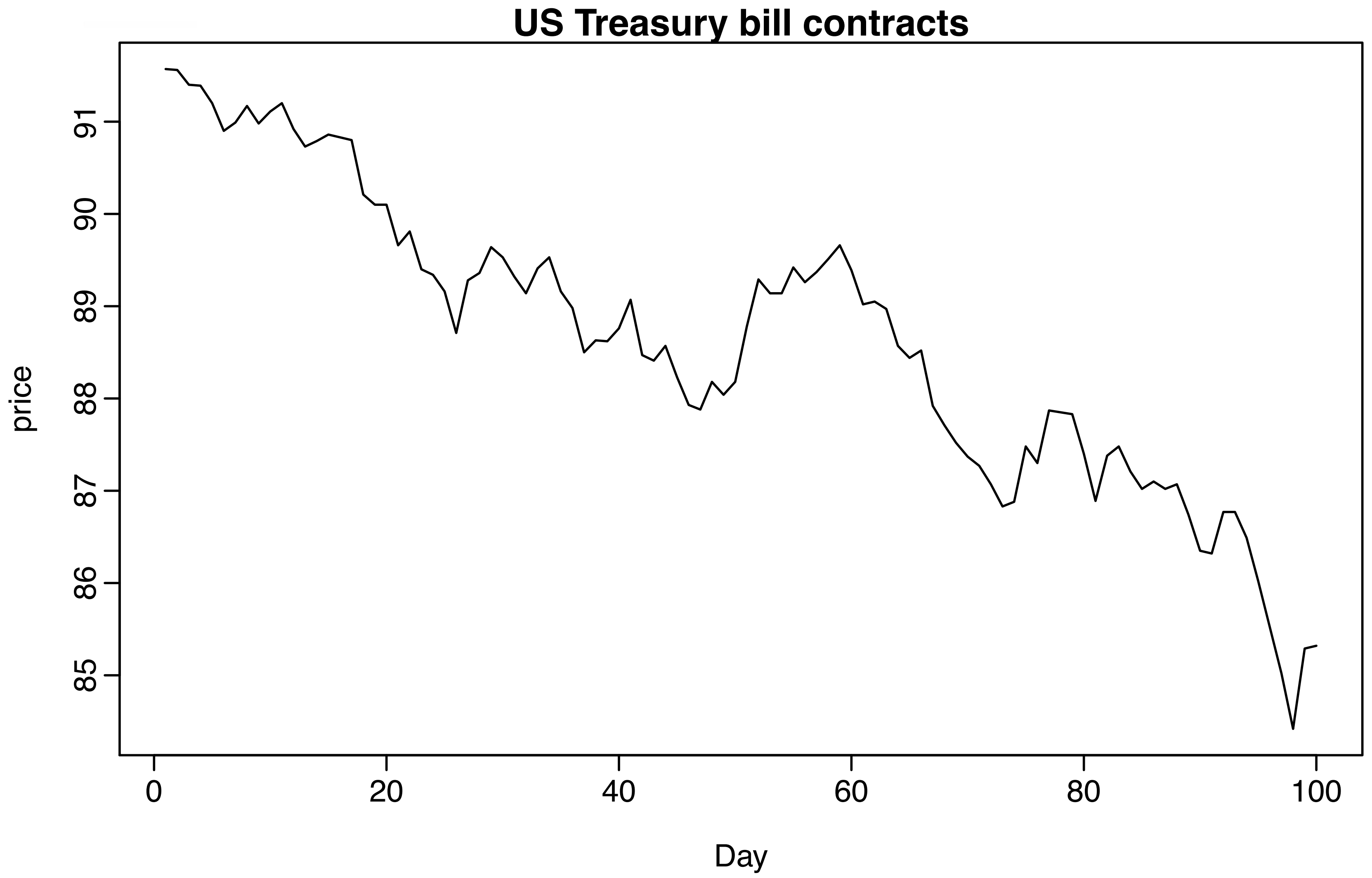




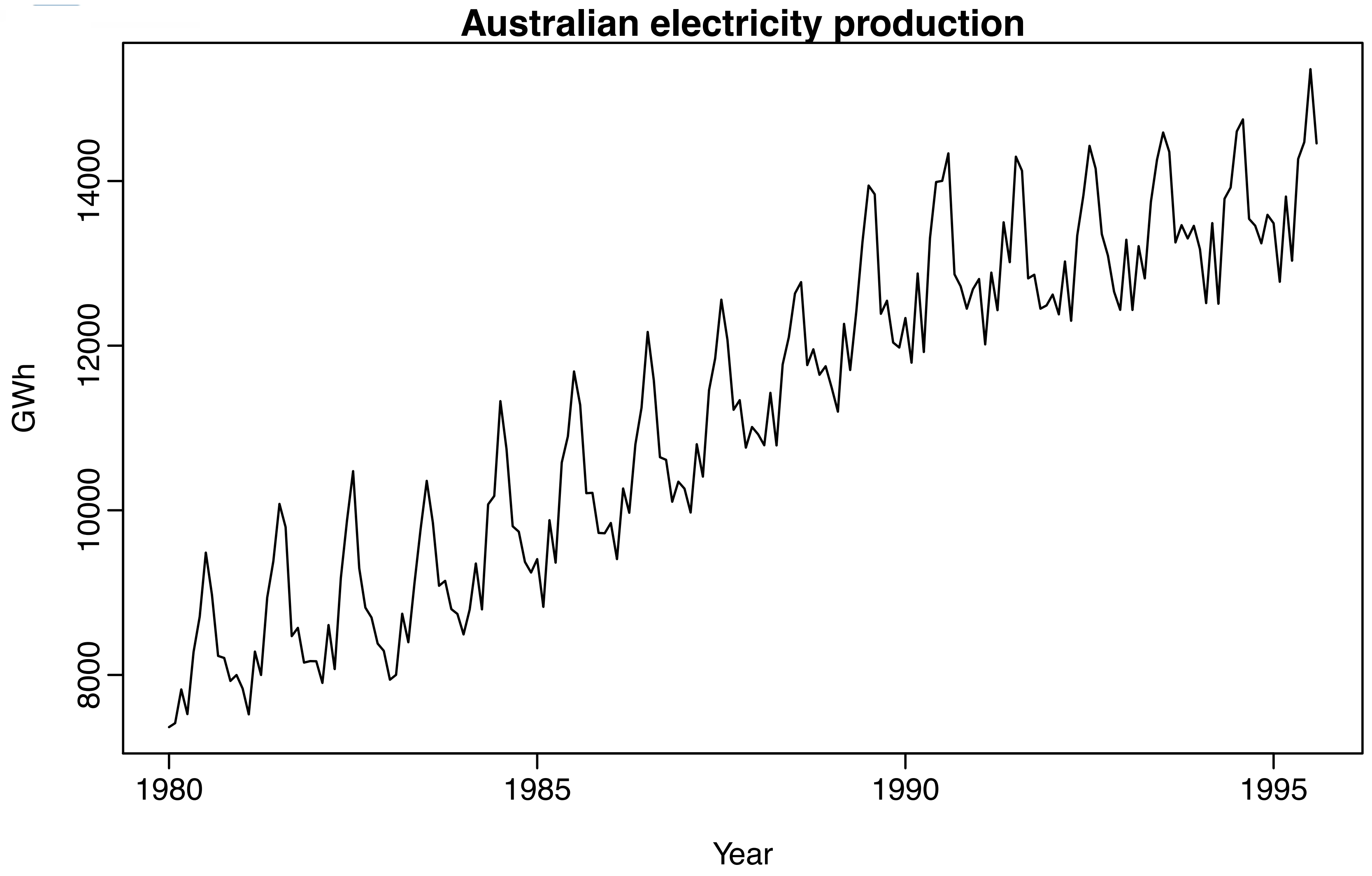
[R. J. Hyndman]

Examples

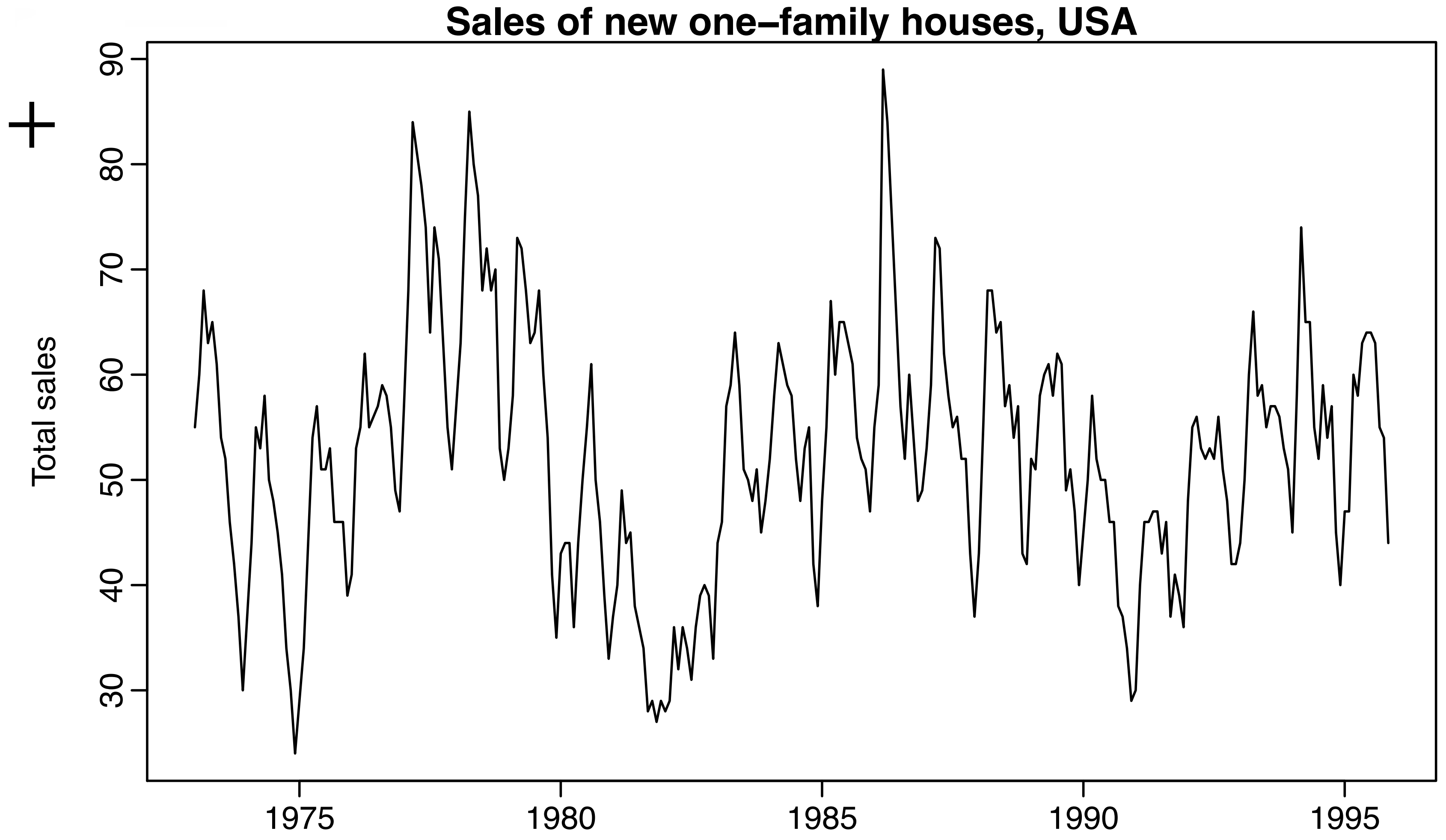
Trend

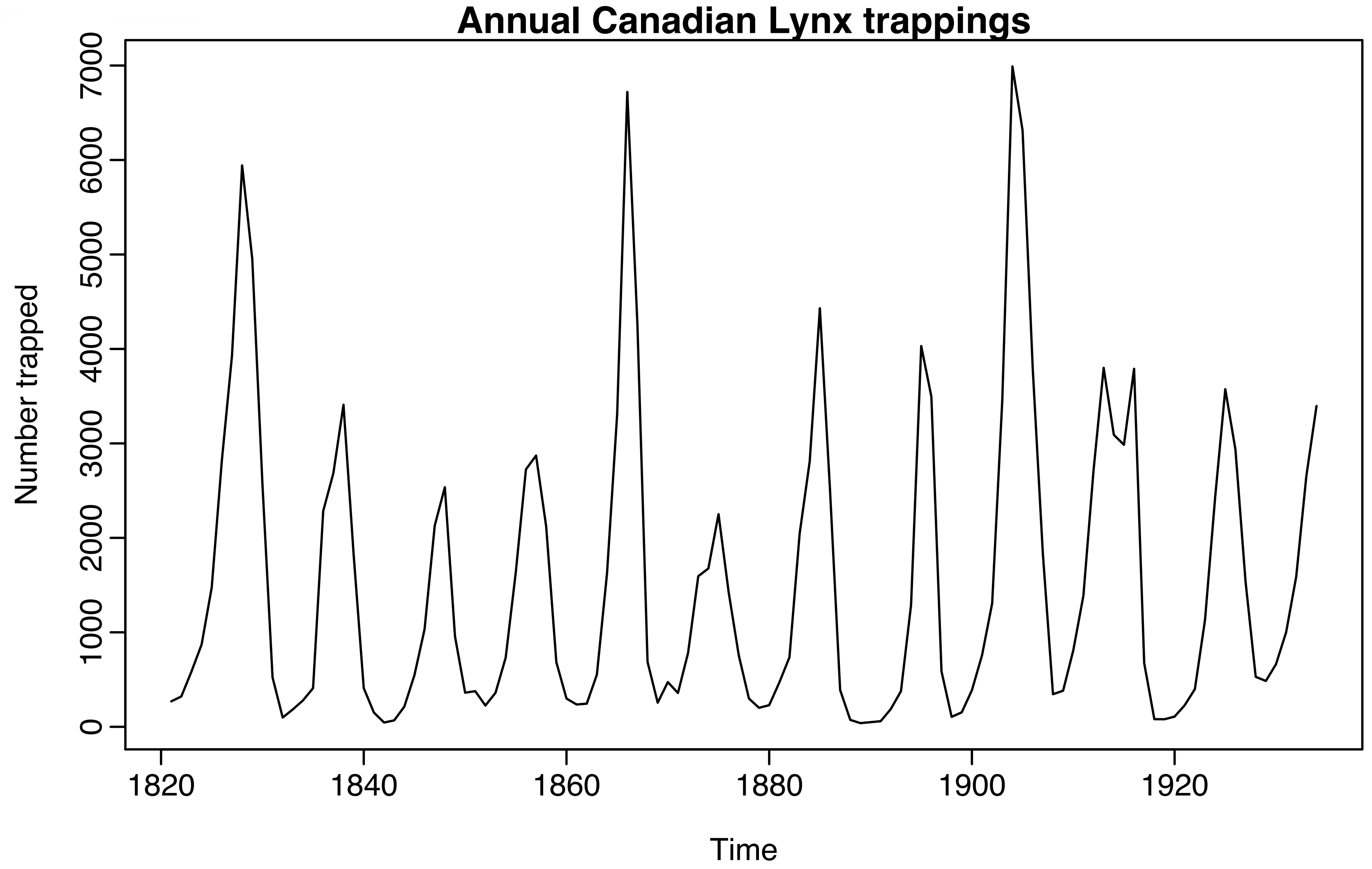


Trend + Seasonality



Seasonality + Cyclic

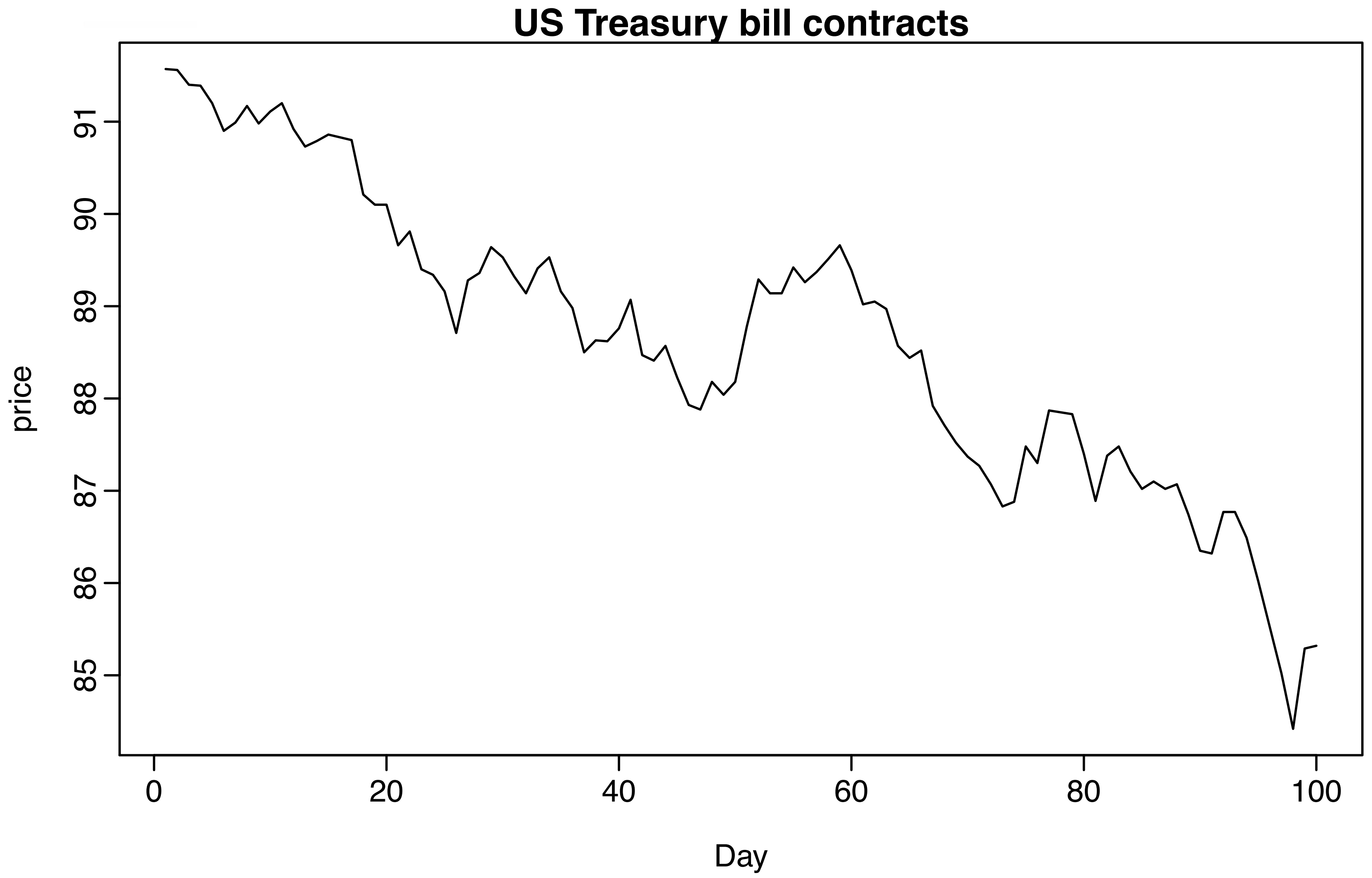




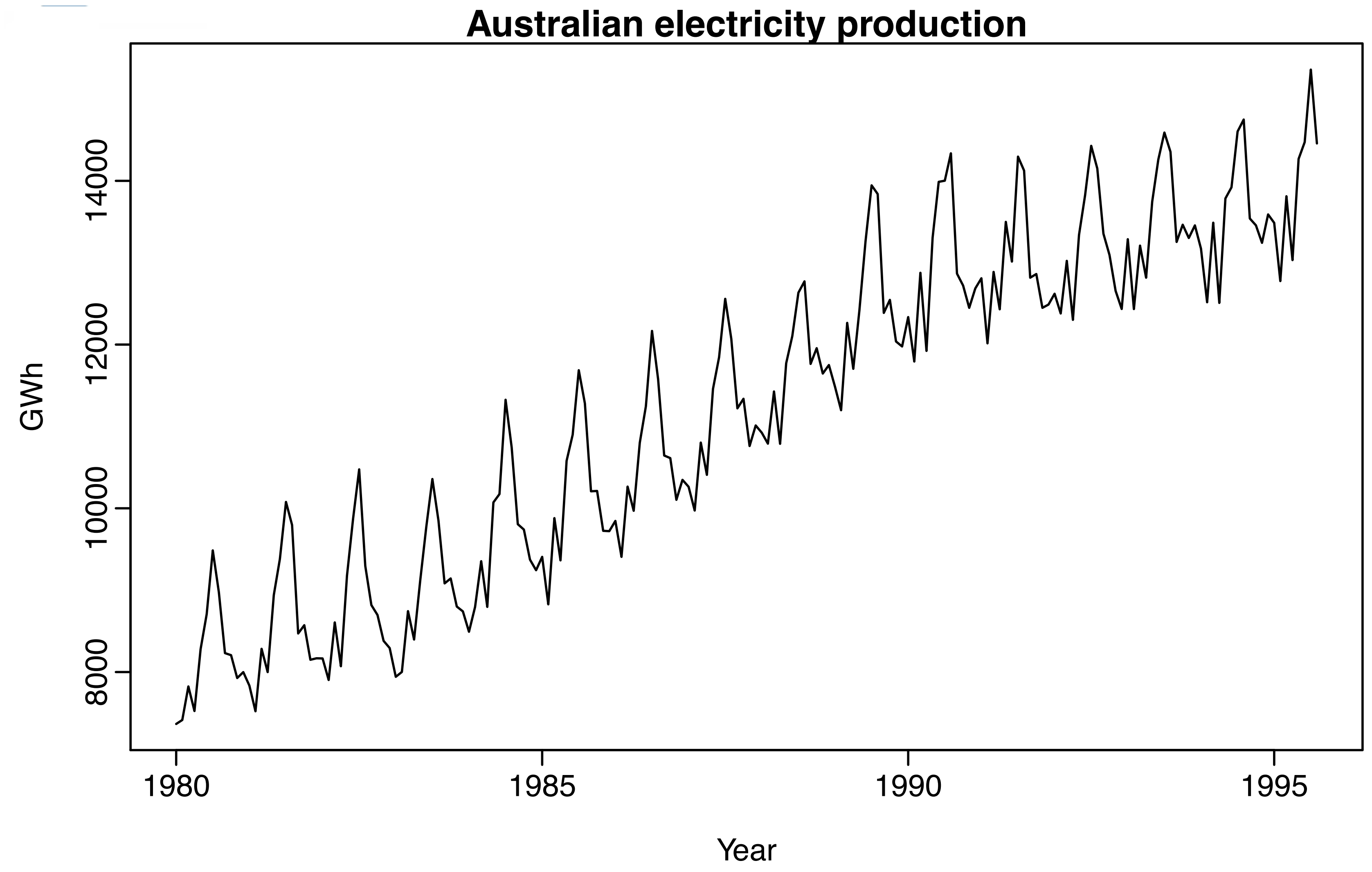
[R. J. Hyndman]

Examples

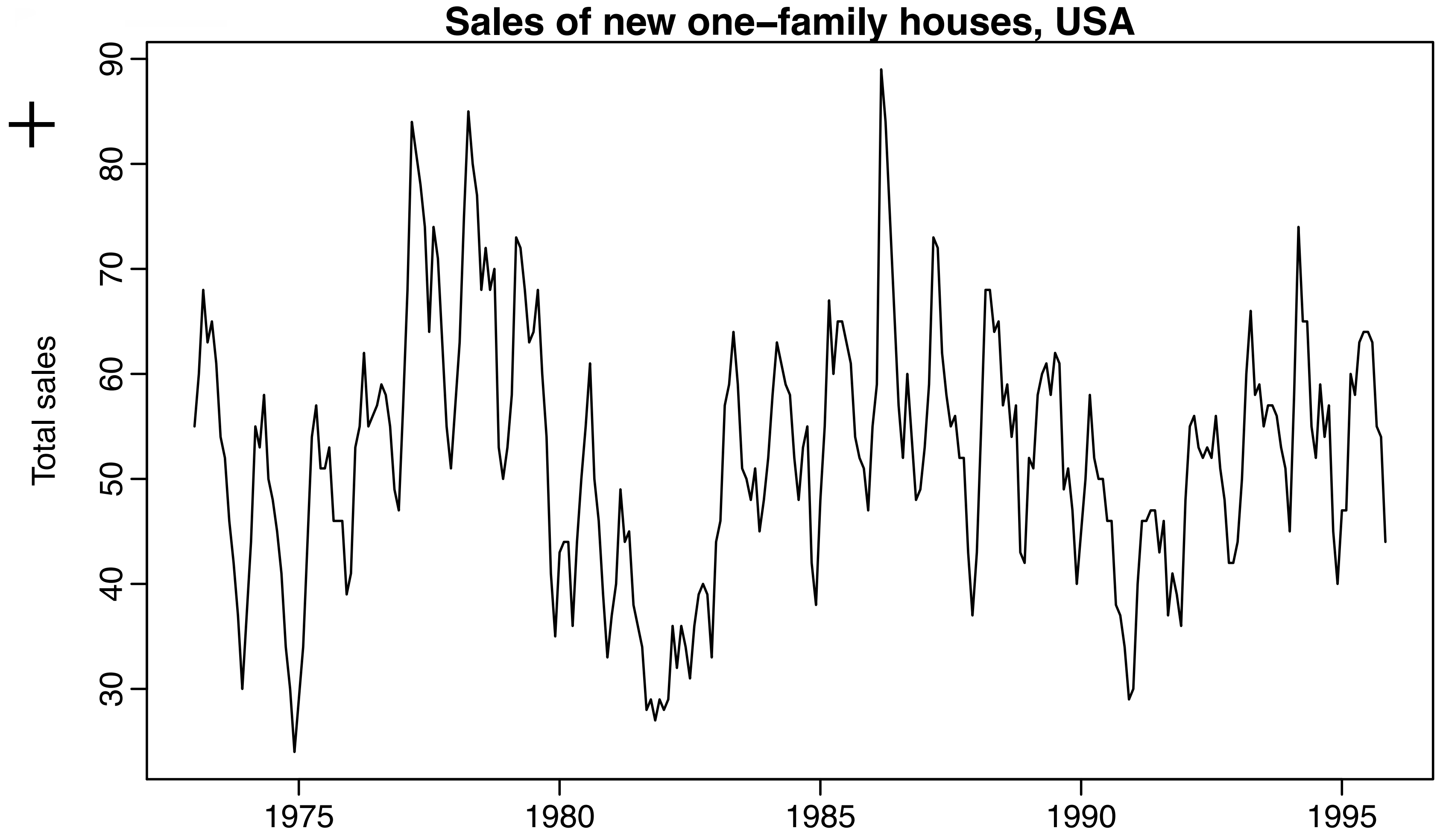
Trend



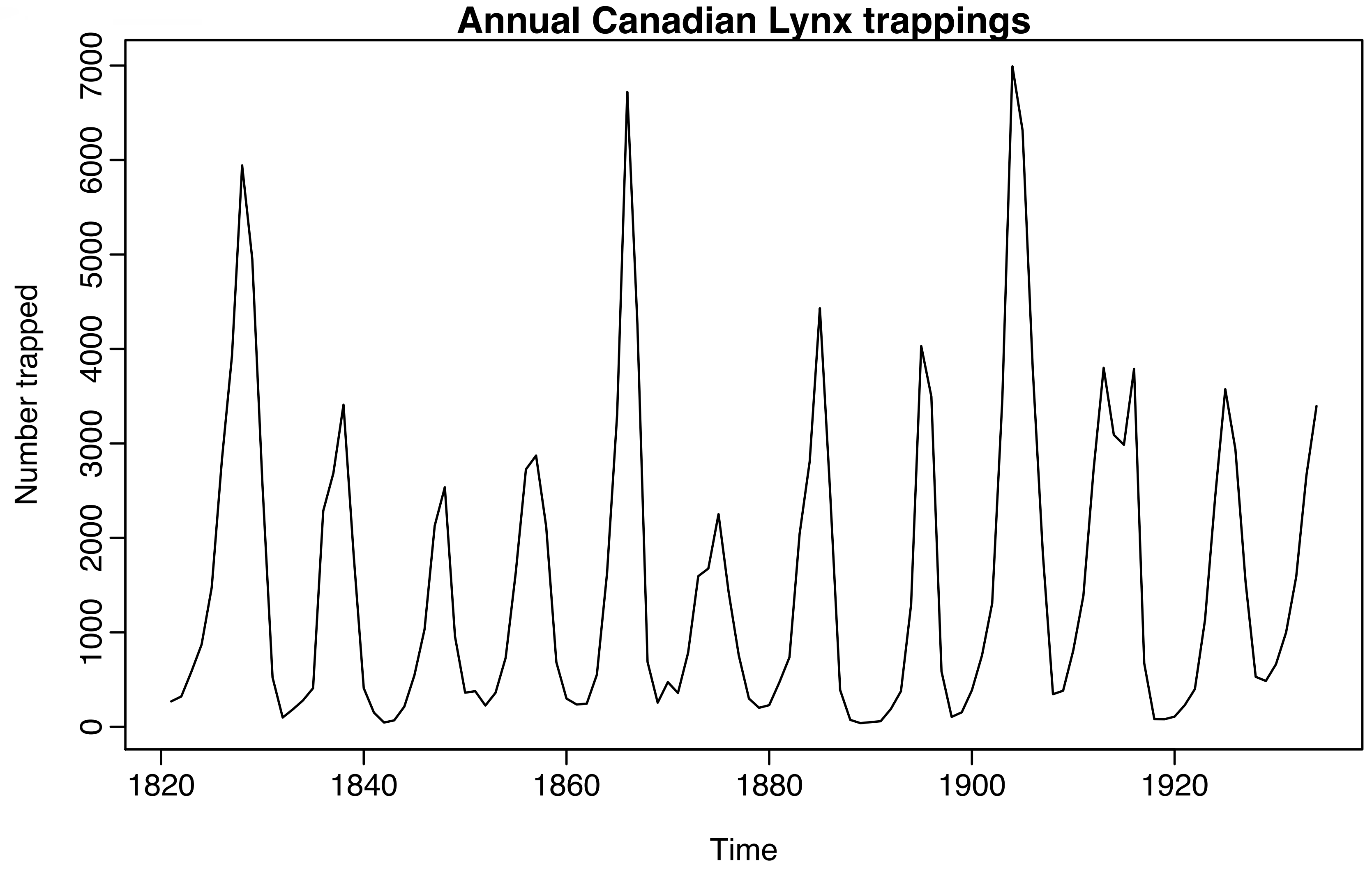
Trend + Seasonality



Seasonality + Cyclic



Stationary



[R. J. Hyndman]

Types of Time Data

- Timestamps: specific instants in time (e.g. `2018-11-27 14:15:00`)
- Periods: have a standard start and length (e.g. the month `November 2018`)
- Intervals: have a start and end timestamp
  - Periods are special case
  - Example: `2018-11-21 14:15:00` — `2018-12-01 05:15:00`
- Elapsed time: measure of time relative to a start time (`15 minutes`)

Dates and Times

- What is time to a computer?
  - Can be stored as seconds since Unix Epoch (January 1st, 1970)
- Often useful to break down into minutes, hours, days, months, years…
- Lots of different ways to write time:
  - How could you write `"November 29, 2016"`?
  - European vs. American ordering…
- What about time zones?

Python Support for Time

- The `datetime` package
  - Has date, time, and datetime classes
  - `.now()` method: the current datetime
  - Can access properties of the time (year, month, seconds, etc.)
- Converting from strings to datetimes:
  - `datetime.strptime`: good for known formats
  - `dateutil.parser.parse`: good for unknown formats
- Converting to strings
  - `str(dt)` or `dt.strftime(<format>)`

Datetime format specification

- Look it up:
  - http://strftime.org
- Generally, can create whatever format you need using these format strings

Code	Meaning	Example
%a	Weekday as locale's abbreviated name.	Mon
%A	Weekday as locale's full name.	Monday
%w	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	1
%d	Day of the month as a zero-padded decimal number.	30
%-d	Day of the month as a decimal number. (Platform specific)	30
%b	Month as locale's abbreviated name.	Sep
%B	Month as locale's full name.	September
%m	Month as a zero-padded decimal number.	09
%-m	Month as a decimal number. (Platform specific)	9
%y	Year without century as a zero-padded decimal number.	13
%Y	Year with century as a decimal number.	2013
%H	Hour (24-hour clock) as a zero-padded decimal number.	07
%-H	Hour (24-hour clock) as a decimal number. (Platform specific)	7
%I	Hour (12-hour clock) as a zero-padded decimal number.	07
%-I	Hour (12-hour clock) as a decimal number. (Platform specific)	7
%p	Locale's equivalent of either AM or PM.	AM
%M	Minute as a zero-padded decimal number.	06
%-M	Minute as a decimal number. (Platform specific)	6
%S	Second as a zero-padded decimal number.	05
%-S	Second as a decimal number. (Platform specific)	5

Pandas Support for Datetime

- `pd.to_datetime`:
  - convenience method
  - can convert an entire column to datetime
- Has a `NaT` to indicate a missing time value
- Stores in a `numpy.datetime64` format
- `pd.Timestamp`: a wrapper for the `datetime64` objects

More Pandas Support

- Accessing a particular time or checking equivalence allows any string that can be interpreted as a date:
  - `ts['1/10/2011']` or `ts['20110110']`
- Date ranges: `pd.date_range('4/1/2012','6/1/2012',freq='4h')`
- Slicing works as expected
- Can do operations (add, subtract) on data indexed by datetime and the indexes will match up
- As with strings, to treat a column as datetime, you can use the `.dt` accessor

Generating Date Ranges

- `index = pd.date_range('4/1/2012', '6/1/2012')`
- Can generate based on a number of periods as well
  - `index = pd.date_range('4/1/2012', periods=20)`
- Frequency (`freq`) controls how the range is divided
  - Codes for specifying this (e.g. `4h`, `D`, `M`)
  -
```
In [90]: pd.date_range('1/1/2000', '1/3/2000 23:59', freq='4h')
Out[90]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-01 00:00:00, ..., 2000-01-03 20:00:00]
Length: 18, Freq: 4H, Timezone: None
```
  - Can also mix them: `'2h30m'`

Time Series Frequencies

Alias	Offset Type	Description
D	Day	Calendar daily
B	BusinessDay	Business daily
H	Hour	Hourly
T or min	Minute	Minutely
S	Second	Secondly
L or ms	Milli	Millisecond (1/1000th of 1 second)
U	Micro	Microsecond (1/1000000th of 1 second)
M	MonthEnd	Last calendar day of month
BM	BusinessMonthEnd	Last business day (weekday) of month
MS	MonthBegin	First calendar day of month
BMS	BusinessMonthBegin	First weekday of month
`W-MON, W-TUE, ...`	Week	Weekly on given day of week: MON, TUE, WED, THU, FRI, SAT, or SUN.
`WOM-1MON, WOM-2MON, ...`	WeekOfMonth	Generate weekly dates in the first, second, third, or fourth week of the month. For example, `WOM-3FRI` for the 3rd Friday of each month.

[W. McKinney, Python for Data Analysis]

DatetimeIndex

- Can use time as an **index**
- ```
  data = [('2017-11-30', 48),
          ('2017-12-02', 45),
          ('2017-12-03', 44),
          ('2017-12-04', 48)]
  dates, temps = zip(*data)
  s = pd.Series(temps, pd.to_datetime(dates))
  ```
- Accessing a particular time or checking equivalence allows any string that can be interpreted as a date:
  - `s['12/04/2017']` or `s['20171204']`
- Using a less specific string will get all matching data:
  - `s['2017-12']` returns the three December entries
  ```

DatetimeIndex

- Time slices do not need to exist:
  - `s['2017-12-01':'2017-12-31']`

Shifting Data

- Leading or Lagging Data

```
In [95]: ts = Series(np.random.randn(4),
   ....:             index=pd.date_range('1/1/2000', periods=4, freq='M'))
```

```
In [96]: ts
Out[96]:
2000-01-31   -0.066748
2000-02-29    0.838639
2000-03-31   -0.117388
2000-04-30   -0.517795
Freq: M, dtype: float64
```

```
In [97]: ts.shift(2)
Out[97]:
2000-01-31         NaN
2000-02-29         NaN
2000-03-31   -0.066748
2000-04-30    0.838639
Freq: M, dtype: float64
```

```
In [98]: ts.shift(-2)
Out[98]:
2000-01-31   -0.117388
2000-02-29   -0.517795
2000-03-31         NaN
2000-04-30         NaN
Freq: M, dtype: float64
```

- Shifting by time:

```
In [99]: ts.shift(2, freq='M')
Out[99]:
2000-03-31   -0.066748
2000-04-30    0.838639
2000-05-31   -0.117388
2000-06-30   -0.517795
Freq: M, dtype: float64
```

Shifting Time Series

- Data:

```
[('2017-11-30', 48), ('2017-12-02', 45),
 ('2017-12-03', 44), ('2017-12-04', 48)]
```

- Compute day-to-day difference in high temperature:
  - `s - s.shift(1)` (same as `s.diff()`)
  - ```
    2017-11-30    NaN
    2017-12-02   -3.0
    2017-12-03   -1.0
    2017-12-04    4.0
    ```
  - `s - s.shift(1, 'd')`
  - ```
    2017-11-30    NaN
    2017-12-01    NaN
    2017-12-02    NaN
    2017-12-03   -1.0
    2017-12-04    4.0
    2017-12-05    NaN
    ```

Timedelta

- Compute differences between dates
- Lives in `datetime` module
- `diff = parse_date("1 Jan 2017") - datetime.now().date()`
  `diff.days`
- Also a `pd.Timedelta` object that take strings:
  - `datetime.now().date() + pd.Timedelta("4 days")`
- Also, Roll dates using anchored offsets

```
from pandas.tseries.offsets import Day, MonthEnd

now = datetime(2011, 11, 17)
In [107]: now + MonthEnd(2)
Out[107]: Timestamp('2011-12-31 00:00:00')
```

Time Zones

- Why?
- Coordinated Universal Time (UTC) is the standard time (basically equivalent to Greenwich Mean Time (GMT)
- Other time zones are UTC +/- a number in [1,12]
- Dartmouth is UTC-5 (aka US/Eastern)

Python, Pandas, and Time Zones

- Time series in pandas are **time zone native**
- The pytz module keeps track of all of the time zone parameters
  - even Daylight Savings Time
- Localize a timestamp using `tz_localize`
  - ```
    ts = pd.Timestamp("1 Dec 2016 12:30 PM")
    ts = ts.tz_localize("US/Eastern")
    ```
- Convert a timestamp using tz_convert
  - `ts.tz_convert("Europe/Budapest")`
- Operations involving timestamps from different time zones become UTC
    ```

Frequency

- Generic time series in pandas are **irregular**
  - there is no fixed frequency
  - we don't necessarily have data for every day/hour/etc.
- Date ranges have frequency

```
In [76]: pd.date_range(start='2012-04-01', periods=20)
Out[76]:
DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04',
               '2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08',
               '2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',
               '2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16',
               '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20'],
              dtype='datetime64[ns]', freq='D')
```

Lots of Frequencies (not comprehensive)

Alias	Offset type	Description
D	Day	Calendar daily
B	BusinessDay	Business daily
H	Hour	Hourly
T or min	Minute	Minutely
S	Second	Secondly
L or ms	Milli	Millisecond (1/1,000 of 1 second)
U	Micro	Microsecond (1/1,000,000 of 1 second)
M	MonthEnd	Last calendar day of month
BM	BusinessMonthEnd	Last business day (weekday) of month
MS	MonthBegin	First calendar day of month
BMS	BusinessMonthBegin	First weekday of month
W-MON, W-TUE, ...	Week	Weekly on given day of week (MON, TUE, WED, THU, FRI, SAT, or SUN)
WOM-1MON, WOM-2MON, ...	WeekOfMonth	Generate weekly dates in the first, second, third, or fourth week of the month (e.g., WOM-3FRI for the third Friday of each month)
Q-JAN, Q-FEB, ...	QuarterEnd	Quarterly dates anchored on last calendar day of each month, for year ending in indicated month (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC)
BQ-JAN, BQ-FEB, ...	BusinessQuarterEnd	Quarterly dates anchored on last weekday day of each month, for year ending in indicated month
QS-JAN, QS-FEB, ...	QuarterBegin	Quarterly dates anchored on first calendar day of each month, for year ending in indicated month
BQS-JAN, BQS-FEB, ...	BusinessQuarterBegin	Quarterly dates anchored on first weekday day of each month, for year ending in indicated month
A-JAN, A-FEB, ...	YearEnd	Annual dates anchored on last calendar day of given month (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC)
BA-JAN, BA-FEB, ...	BusinessYearEnd	Annual dates anchored on last weekday of given month
AS-JAN, AS-FEB, ...	YearBegin	Annual dates anchored on first day of given month
BAS-JAN, BAS-FEB, ...	BusinessYearBegin	Annual dates anchored on first weekday of given month

[W. McKinney, Python for Data Analysis]

Resampling

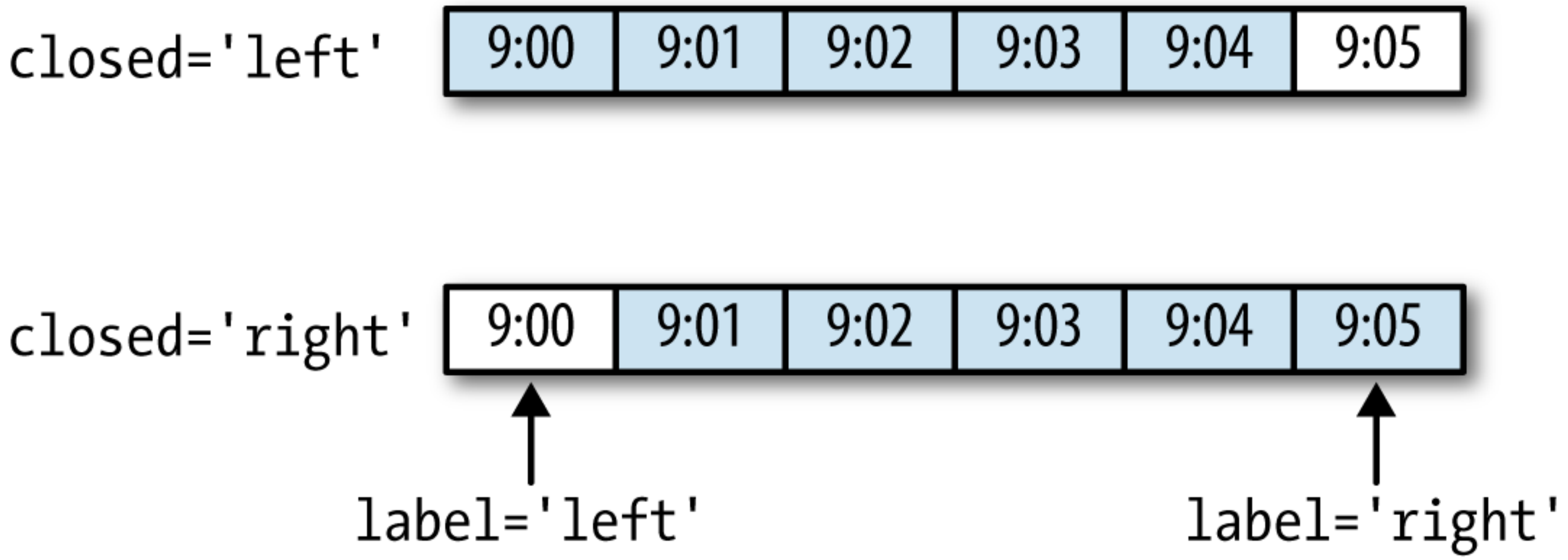
- Could be
  - downsample: higher frequency to lower frequency
  - upsample: lower frequency to higher frequency
  - neither: e.g. Wednesdays to Fridays
- resample method: e.g. `ts.resample('M').mean()`

Argument	Description
`freq`	String or DateOffset indicating desired resampled frequency (e.g., 'M', '5min', or `Second(15)`)
`axis`	Axis to resample on; default axis=0
`fill_method`	How to interpolate when upsampling, as in `'ffill'` or `'bfill'`; by default does no interpolation
`closed`	In downsampling, which end of each interval is closed (inclusive), `'right'` or `'left'`
`label`	In downsampling, how to label the aggregated result, with the `'right'` or `'left'` bin edge (e.g., the 9:30 to 9:35 five-minute interval could be labeled `9:30` or `9:35`)
`loffset`	Time adjustment to the bin labels, such as `'-1s'` / `Second(-1)` to shift the aggregate labels one second earlier
`limit`	When forward or backward filling, the maximum number of periods to fill
`kind`	Aggregate to periods (`'period'`) or timestamps (`'timestamp'`); defaults to the type of index the time series has
`convention`	When resampling periods, the convention (`'start'` or `'end'`) for converting the low-frequency period to high frequency; defaults to `'end'`

[W. McKinney, Python for Data Analysis]

Downsampling

- Need to define **bin edges** which are used to group the time series into **intervals** that can be aggregated
- Remember:
  - Which side of the interval is closed
  - How to label the aggregated bin (start or end of interval)



Upsampling

- No aggregation necessary

```
In [222]: frame
Out[222]:
            Colorado     Texas  New York      Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-12 -0.046662  0.927238  0.482284 -0.867130
```

```
In [223]: df_daily = frame.resample('D').asfreq()

In [224]: df_daily
Out[224]:
            Colorado     Texas  New York      Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-06       NaN       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN       NaN
2000-01-08       NaN       NaN       NaN       NaN
2000-01-09       NaN       NaN       NaN       NaN
2000-01-10       NaN       NaN       NaN       NaN
2000-01-11       NaN       NaN       NaN       NaN
2000-01-12 -0.046662  0.927238  0.482284 -0.867130
```

```
In [225]: frame.resample('D').ffill()
Out[225]:
            Colorado     Texas  New York      Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-06 -0.896431  0.677263  0.036503  0.087102
2000-01-07 -0.896431  0.677263  0.036503  0.087102
2000-01-08 -0.896431  0.677263  0.036503  0.087102
2000-01-09 -0.896431  0.677263  0.036503  0.087102
2000-01-10 -0.896431  0.677263  0.036503  0.087102
2000-01-11 -0.896431  0.677263  0.036503  0.087102
2000-01-12 -0.046662  0.927238  0.482284 -0.867130
```

Rolling Window Calculations

12	8	7	4	9	13	4	11	3	8

Rolling Window Calculations

12	8	7	4	9	13	4	11	3	8

7.8

Rolling Window Calculations

12	8	7	4	9	13	4	11	3	8

7.8

Rolling Window Calculations

12	8	7	4	9	13	4	11	3	8

7.8 7.0

Rolling Window Calculations

12	8	7	4	9	13	4	11	3	8

7.8 7.0

Rolling Window Calculations

12	8	7	4	9	13	4	11	3	8

7.8 7.0 8.3

Window Functions

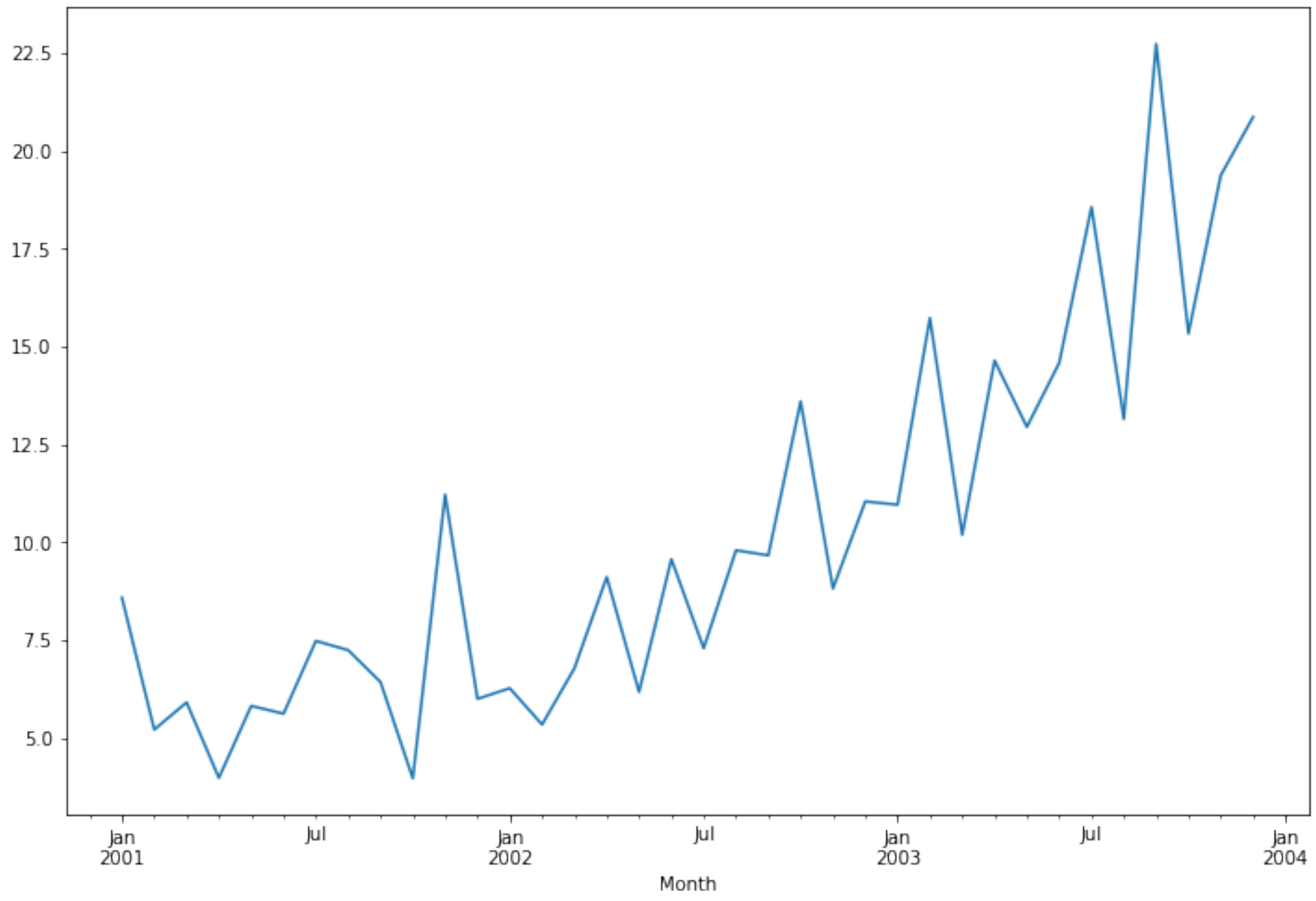
- Idea: want to aggregate over a window of time, calculate the answer, and then slide that window ahead. Repeat.
- `rolling`: smooth out data
- Specify the window size in rolling, then an aggregation method
- Result is set to the right edge of window (change with `center=True`)
- Example:
  - `df.rolling('180D').mean()`
  - `df.rolling('90D').sum()`

Shampoo Sales Example

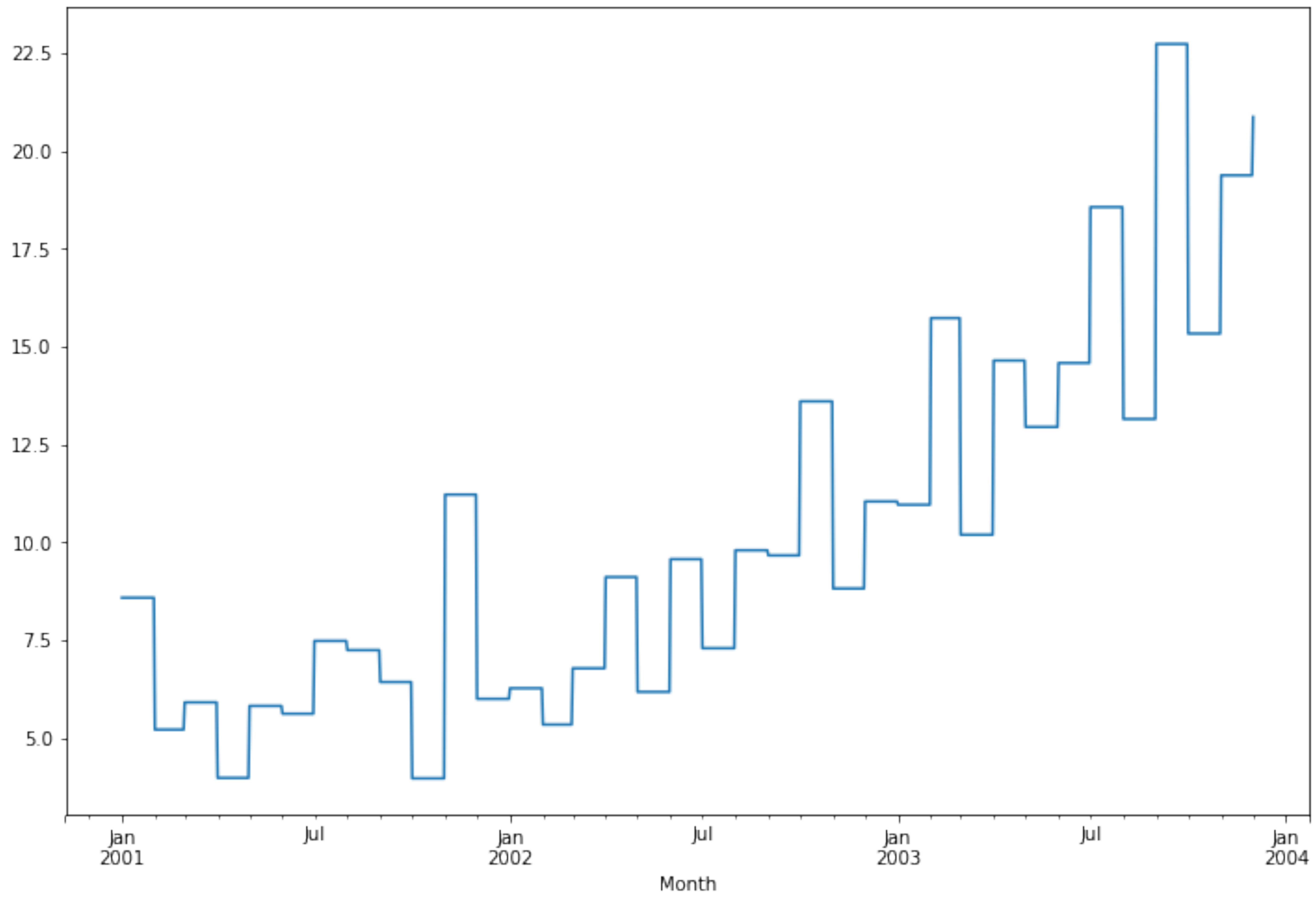
Interpolation

- Fill in the missing values with computed best estimates using various types of algorithms
- Apply after resample

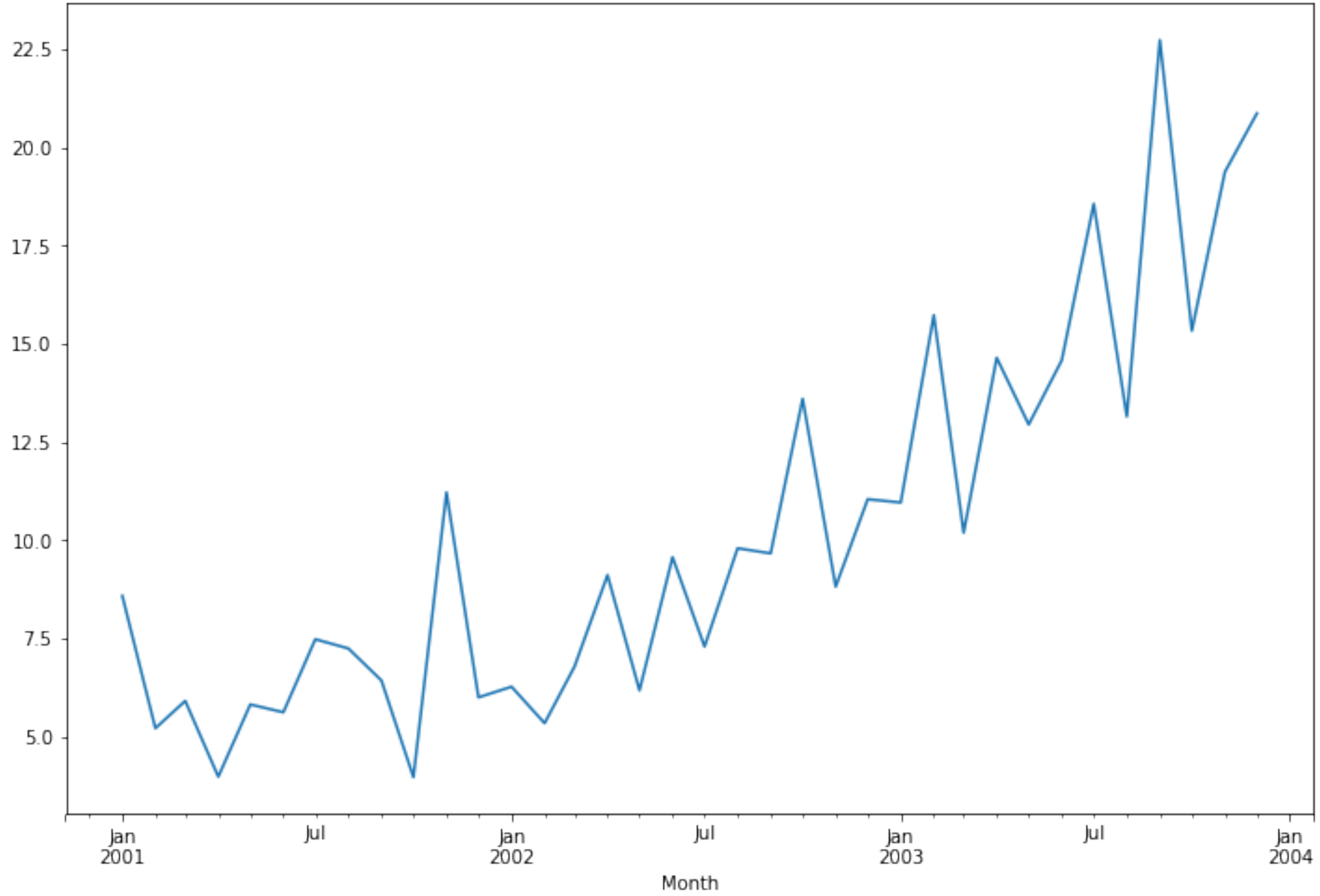
Sales Data by Month



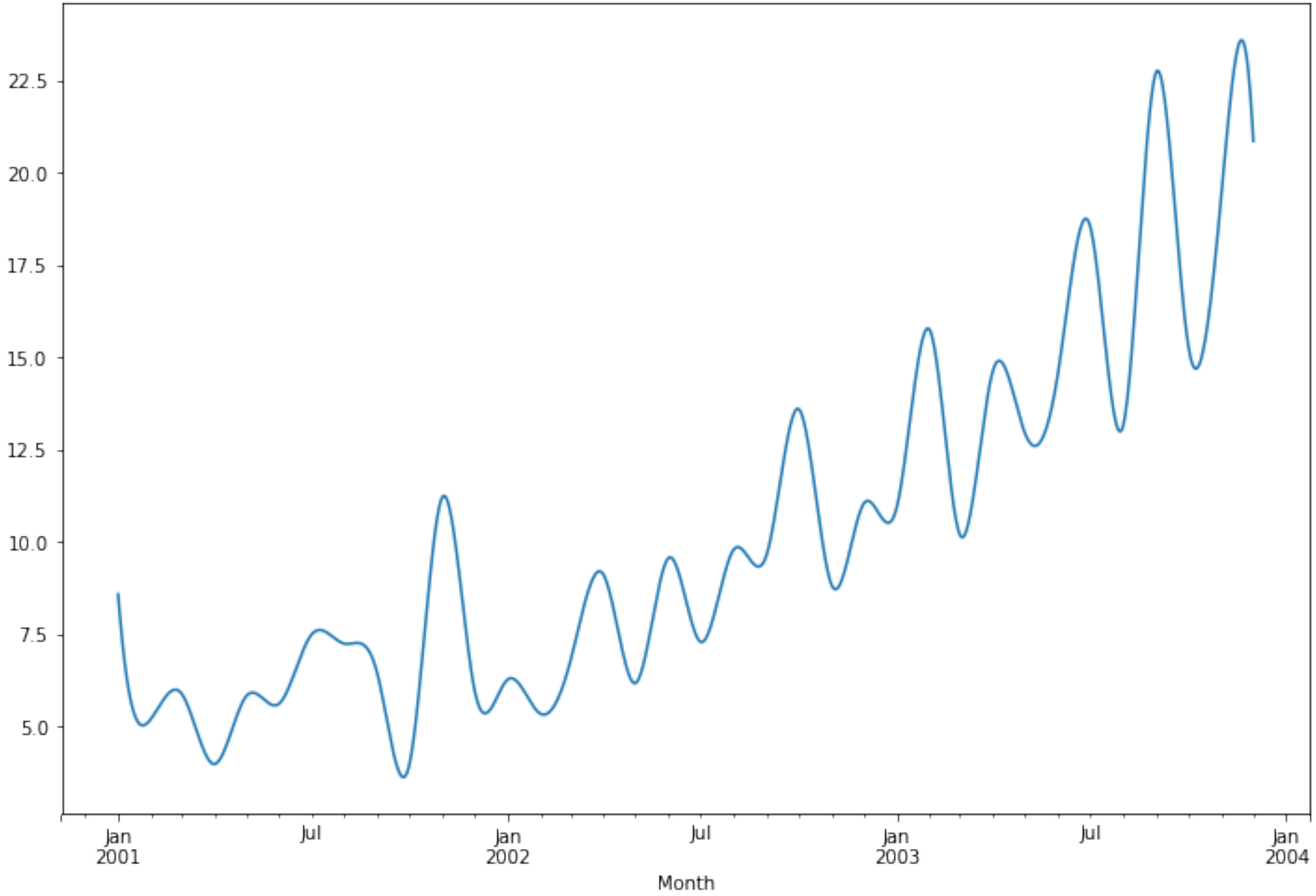
Resampled Sales Data (ffill)



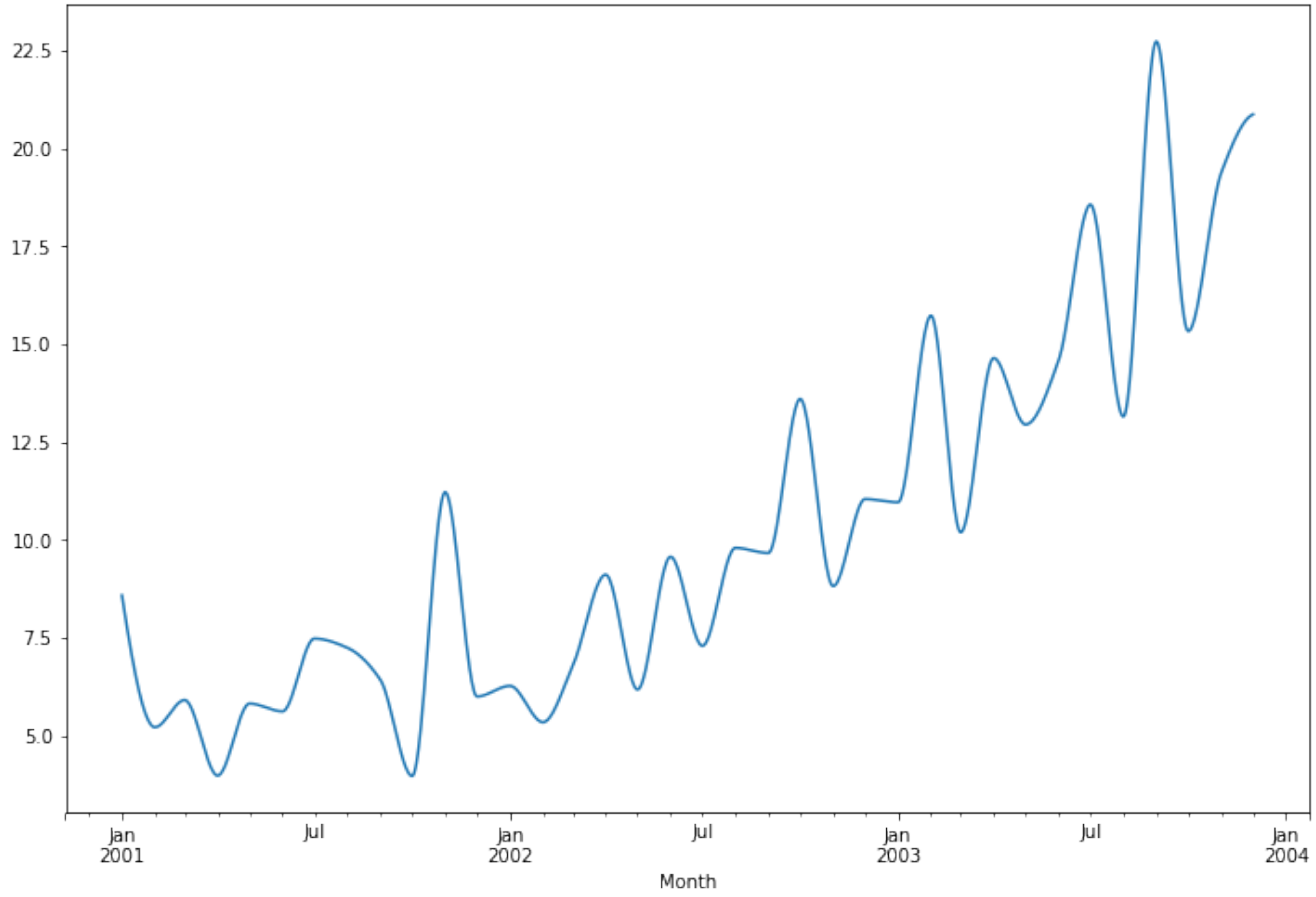
Resampled with Linear Interpolation (Default)



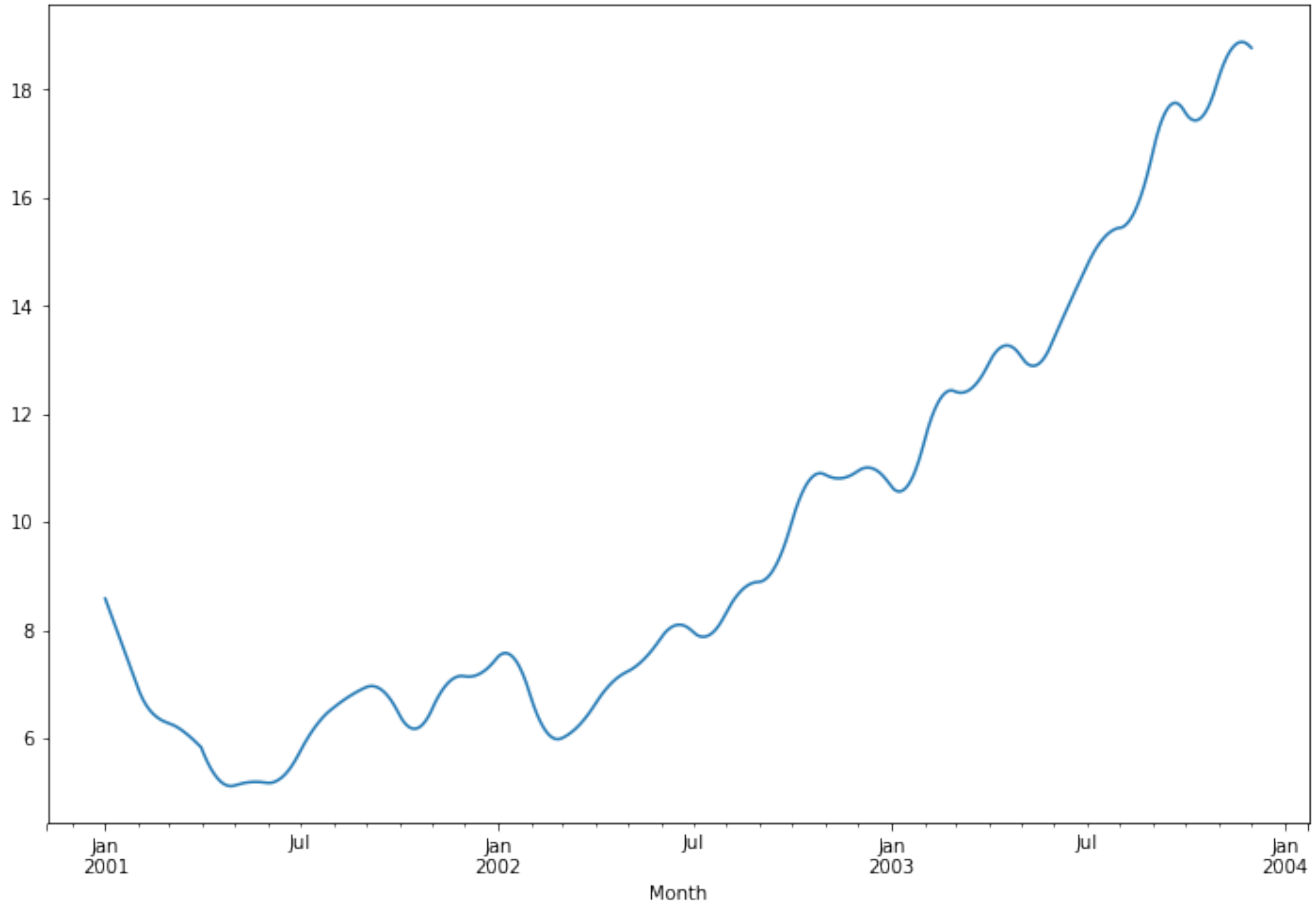
Resampled with Cubic Interpolation



Piecewise Cubic Hermite Interpolating Polynomial



90-Day Rolling Window (Mean)



180-Day Rolling Window (Mean)