Advanced Data Management (CSCI 640/490)

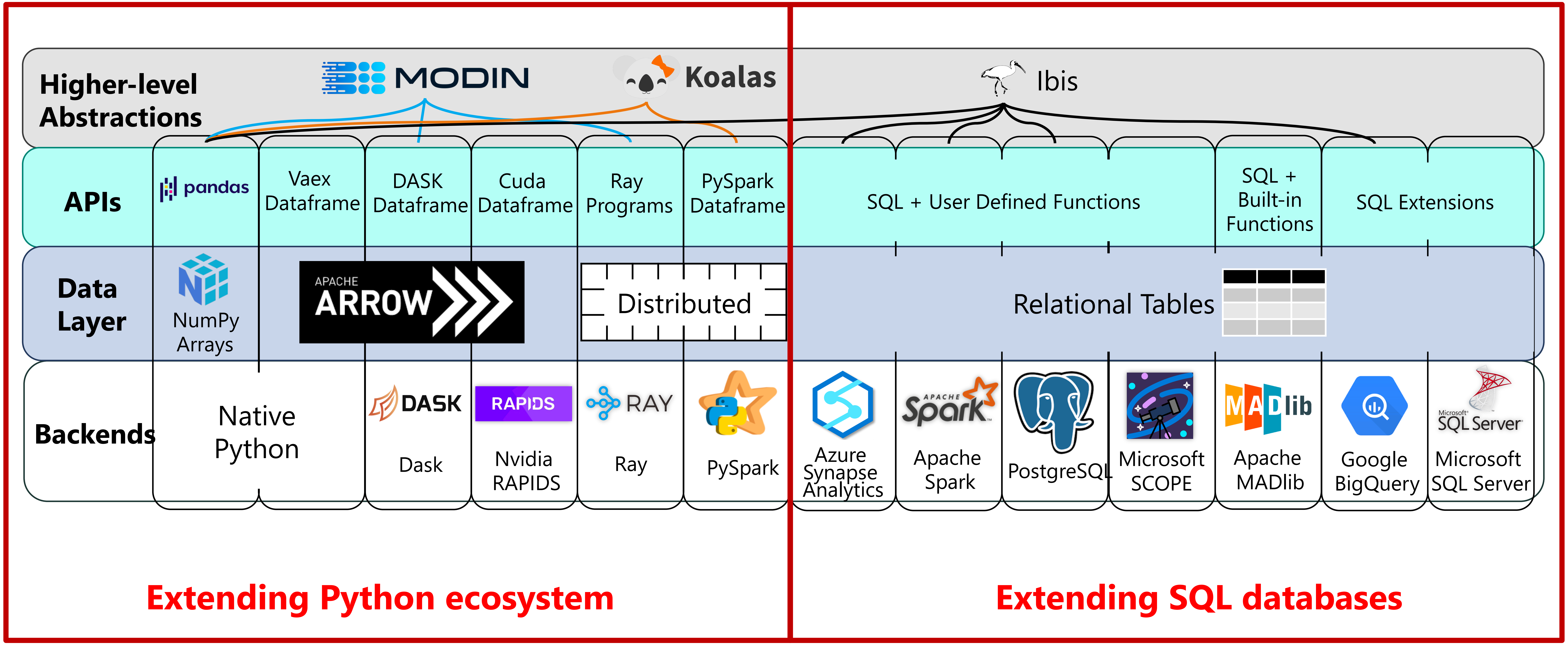
Time Series Data

Dr. David Koop

Dataframes, Databases, and the Cloud

- How do we take advantage of different architectures?
- Lots of work in scaling databases and specialized computational engines
- What is the code that people actually write?

Data Science Jungle



[A. Jindal et al., 2021]

Magpie Goals

Pythonic Environment — python — Familiar Python surface

Unified Dataframe API — pandas — Ongoing standardization

Magpie Middleware

- PyFroid Compiler — Batching Pandas into large query expressions
- Cross Optimization — Backend selection using past workloads
- Common Data Layer — APACHE ARROW — Cache commonly seen dataframes

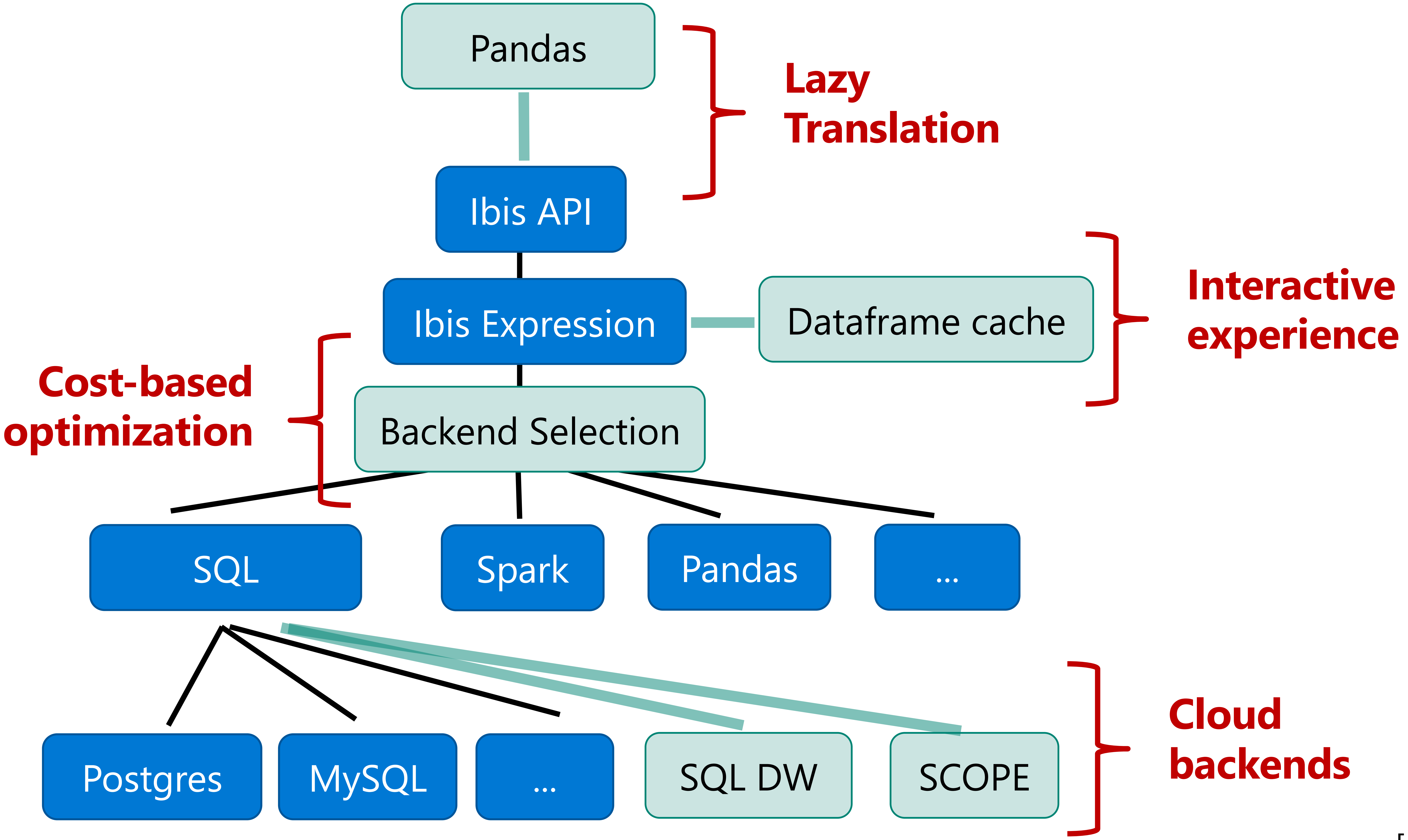
Polyengines & Mappers: Azure Synapse Analytics, Koalas, SQL Alchemy — Multi-backend environments and libraries

Native Python: python, pandas, intel Scalable Dataframe Compiler

Database Backends: Microsoft SCOPE, Apache Spark, PostgreSQL, Apache MADlib, Google BigQuery, SQL Server — Cloud backends

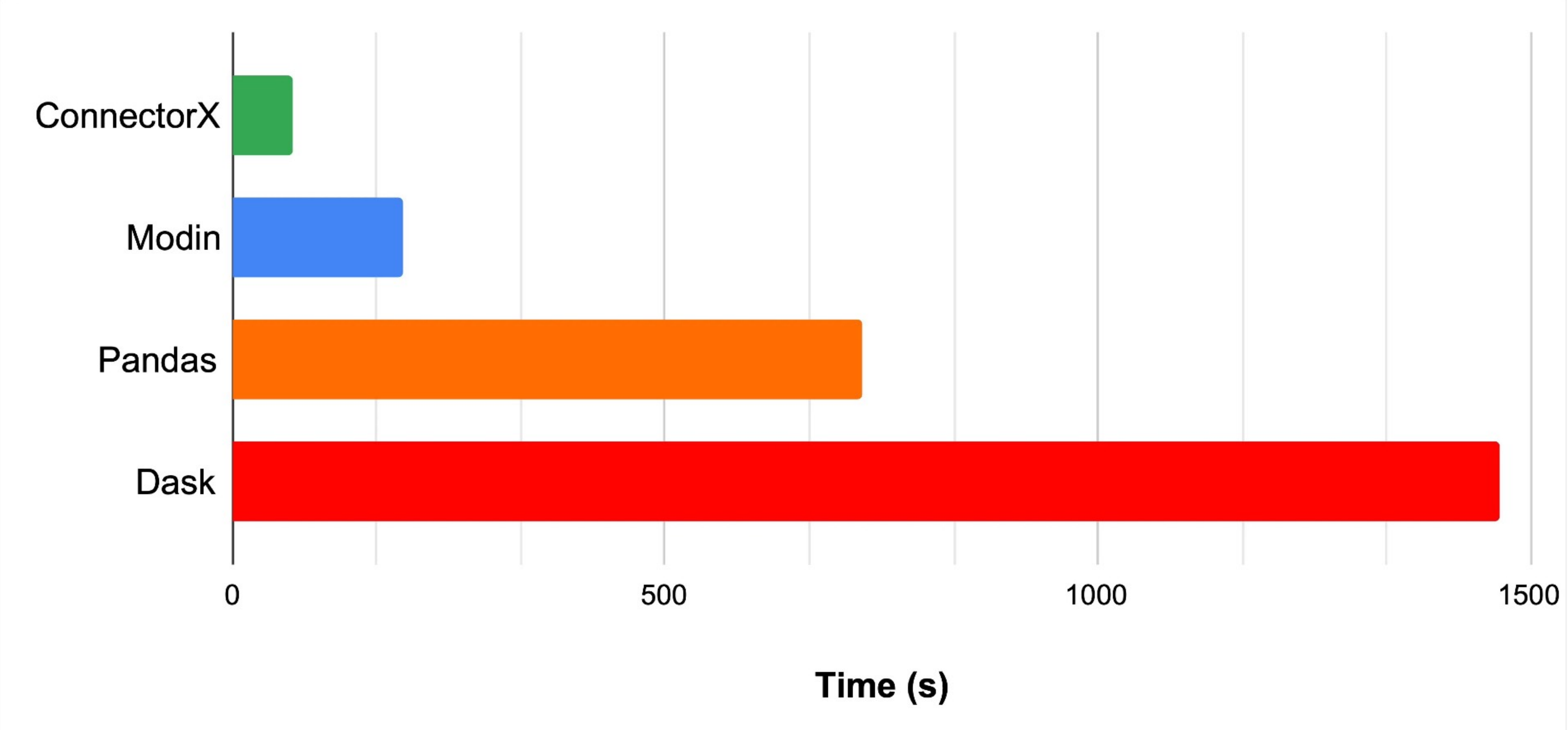
[A. Jindal et al., 2021]

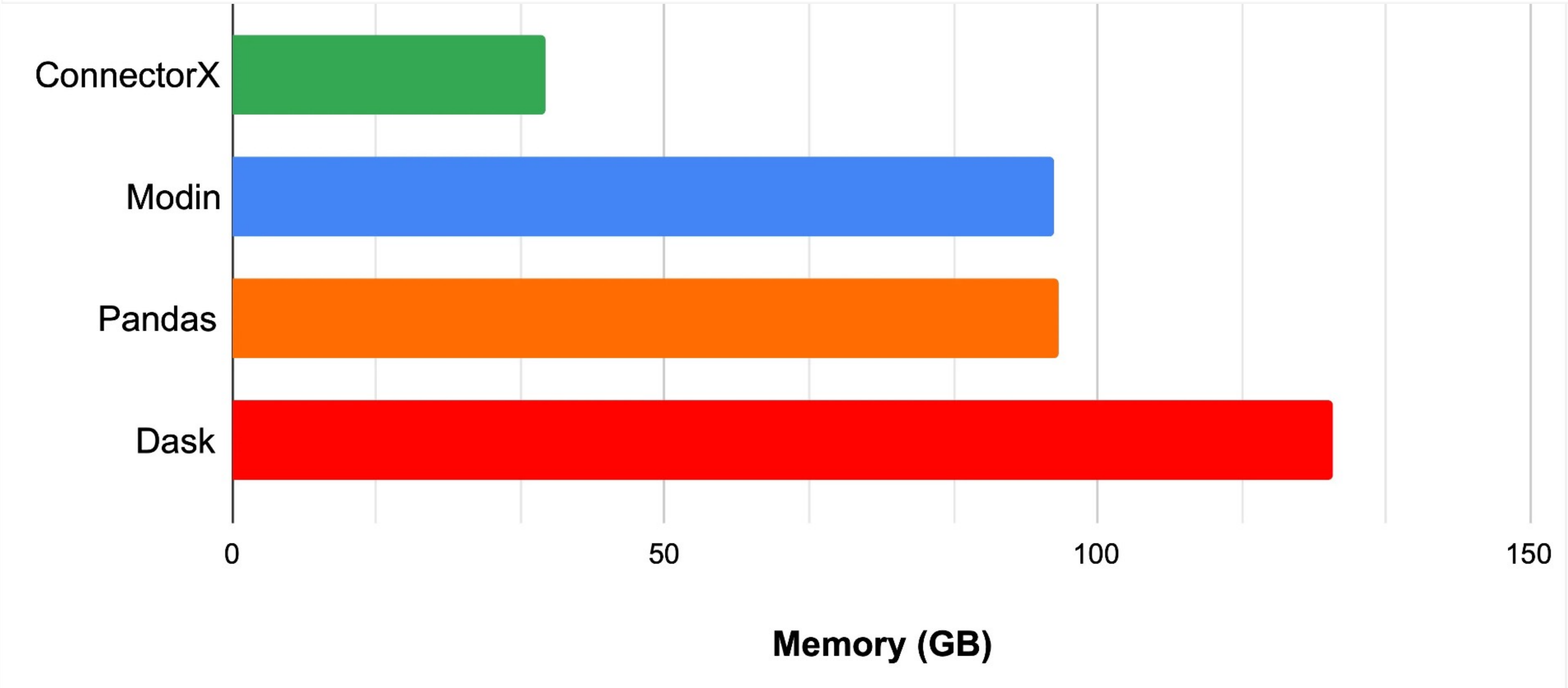
Magpie Architecture



[A. Jindal et al., 2021]

ConnectorX: Databases to Dataframes





[X. Wang, 2022]

Dataframe API?

- SQL, pandas, or something else?

Doris Lee
@dorisjlee

🌶️ Hot Takes on Enterprise Pandas — Day 2 🌶️

🛤️ In many cases, SQL isn't the solution, and pandas is the easier path. 🧵 below

11:15 AM · Mar 27, 2023 · **6,517** Views

Doris Lee
@dorisjlee

🤹 SQL is good for certain things, but there are things that SQL wasn't meant to do, and if you contort SQL to do them, you wind up with nightmarish queries. Many of these can be no more than a few lines in pandas.

11:15 AM · Mar 27, 2023 · **172** Views

Doris Lee
@dorisjlee

🌶️ Hot Takes on Enterprise Pandas — Day 5 🌶️

😱 Beware of "pandas-like" APIs that aren't actually compatible with pandas. Many dataframe libraries may look similar to pandas but lack support for critical pandas functionalities.

11:15 AM · Mar 30, 2023 · **5,225** Views

[D. Lee, Ponder CEO]

Assignment 4

- Work on Data Integration and Data Fusion
- Integrate artist datasets from different institutions (Met, NGA, AIC, CMA)
  - Integrate information based on ids and matching
- Record Matching:
  - Which artists are the same?
- Data Fusion:
  - Names
  - Dates
  - Nationalities

Test 2

- Next Monday… April 8
- Similar format, but more emphasis on topics we have covered including the research papers

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!

What is Time Series Data?

- A row of data that consists of a timestamp, a value, optional tags

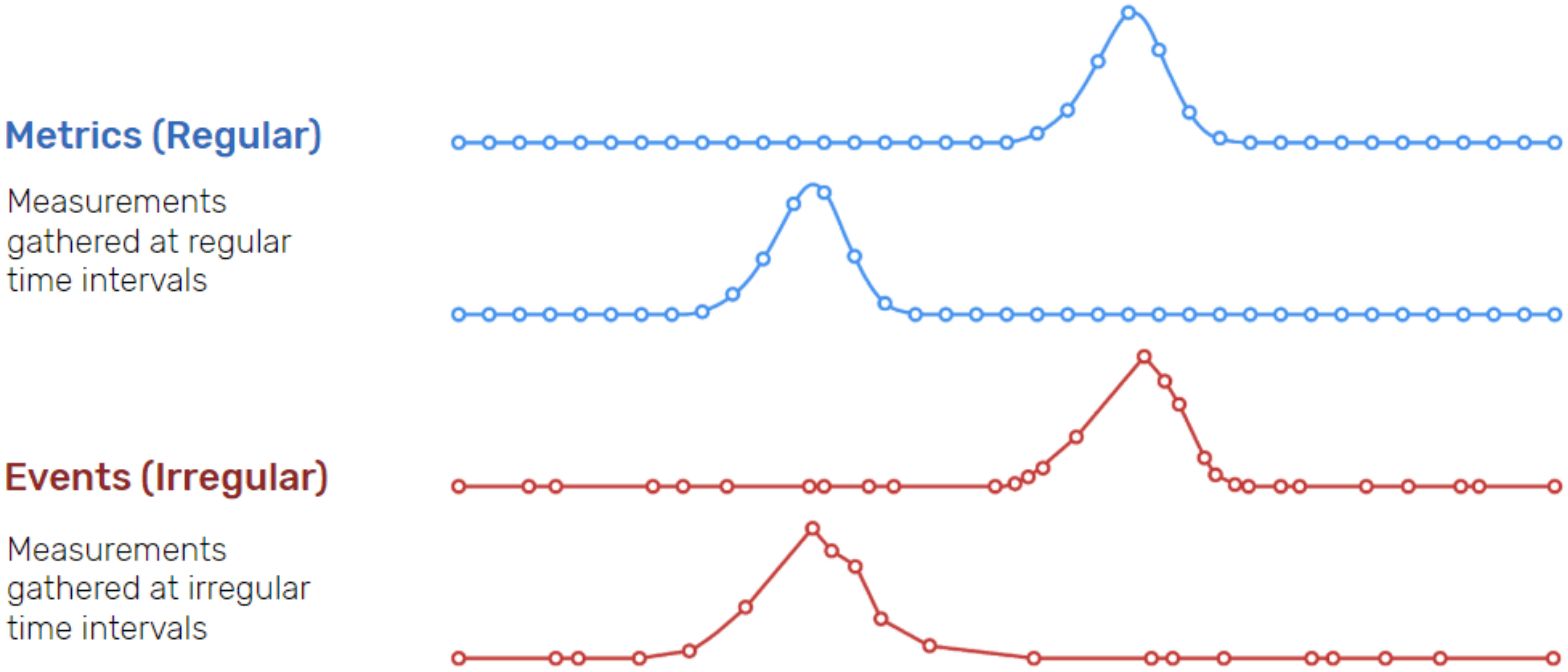
ul1

timestamp	tags					value
time	generated	message_subtype	scaler	short_id	tenant	value
2016-07-12T11:51:45Z	"true"	"34"	"4"	"3"	"saarlouis"	465110000
2016-07-12T11:51:45Z	"true"	"34"	"-6"	"2"	"saarlouis"	0.06196699999999994
2016-07-12T12:10:00Z	"true"	"34"	"7"	"5"	"saarlouis"	49370000000
2016-07-12T12:10:00Z	"true"	"34"	"6"	"2"	"saarlouis"	18573000000
2016-07-12T12:10:00Z	"true"	"34"	"5"	"7"	"saarlouis"	5902300000

[A. Bader, 2017]

Time Series Data

- Metrics: measurements at regular intervals
- Events: measurements that are not gathered at regular intervals



[InfluxDB]

Types of Time Series Data

- time series: observations for a **single** entity at **different** time intervals
  - one patient's heart rate every minute
- cross-section: observations for **multiple** entities at the **same** point in time
  - heart rates of 100 patients at 8:01pm
- panel data: observations for **multiple** entities at **different** time intervals
  - heart rates of 100 patients every minute over the past hour

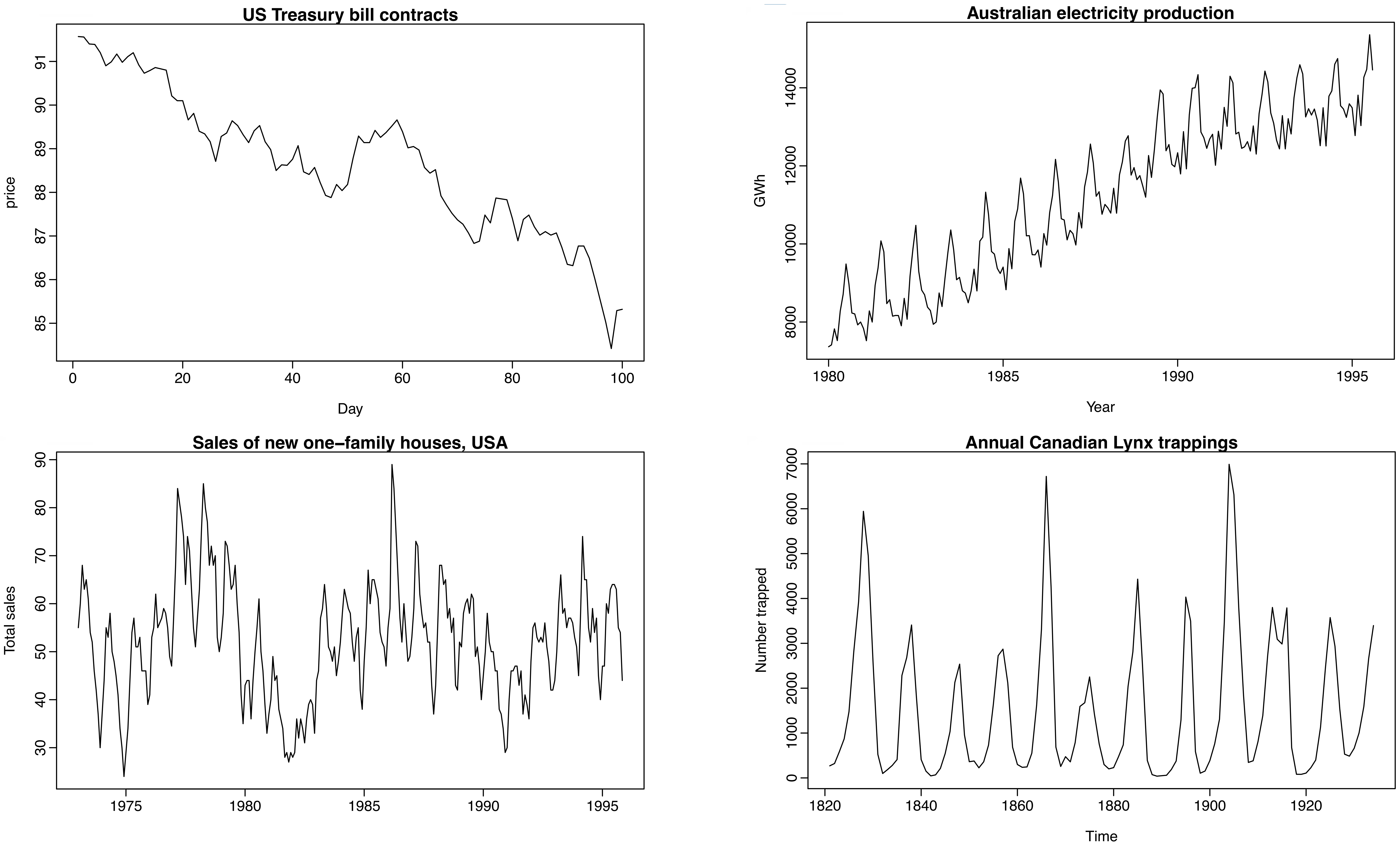
[InfluxDB]

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]

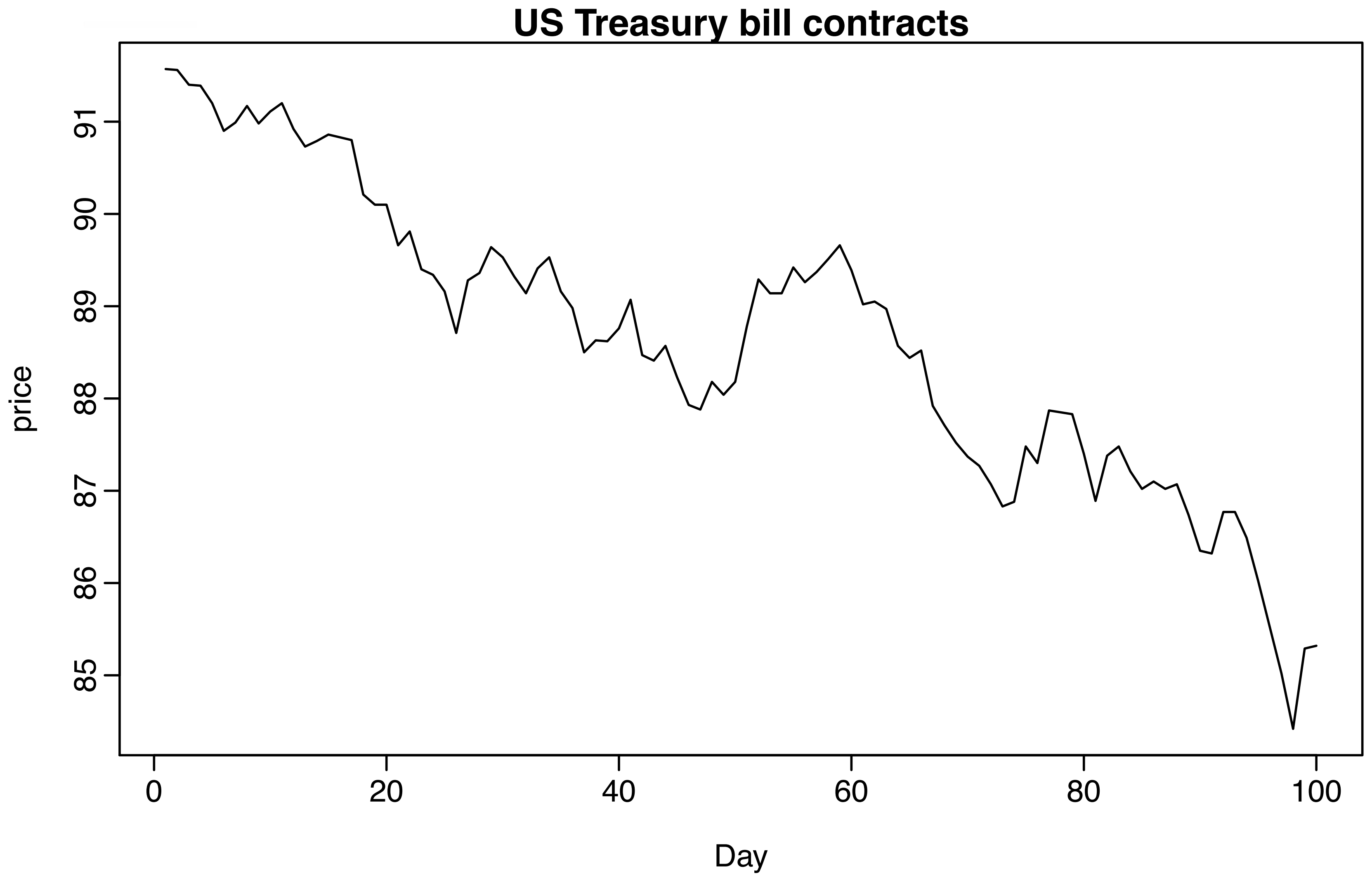
Examples

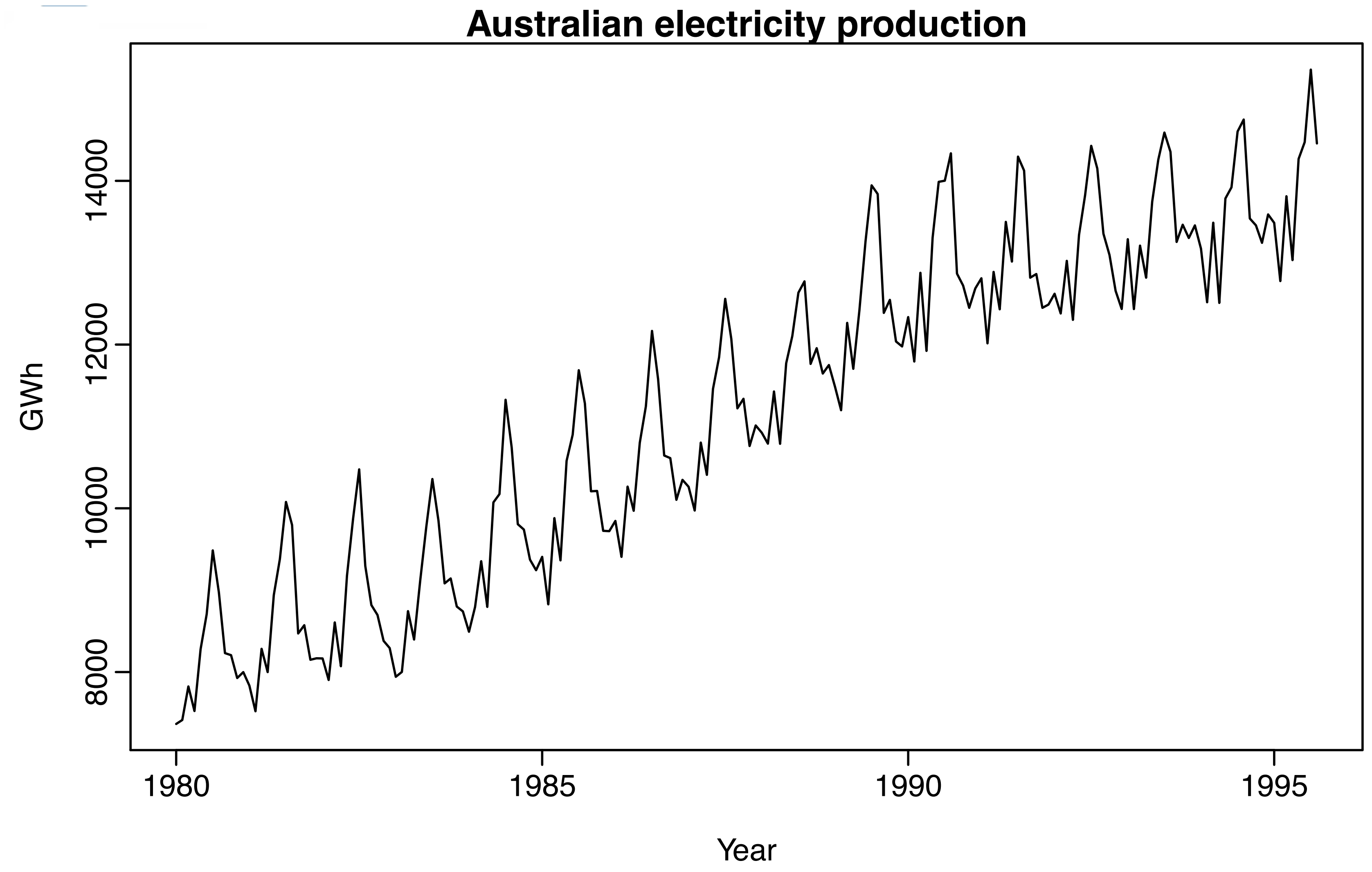


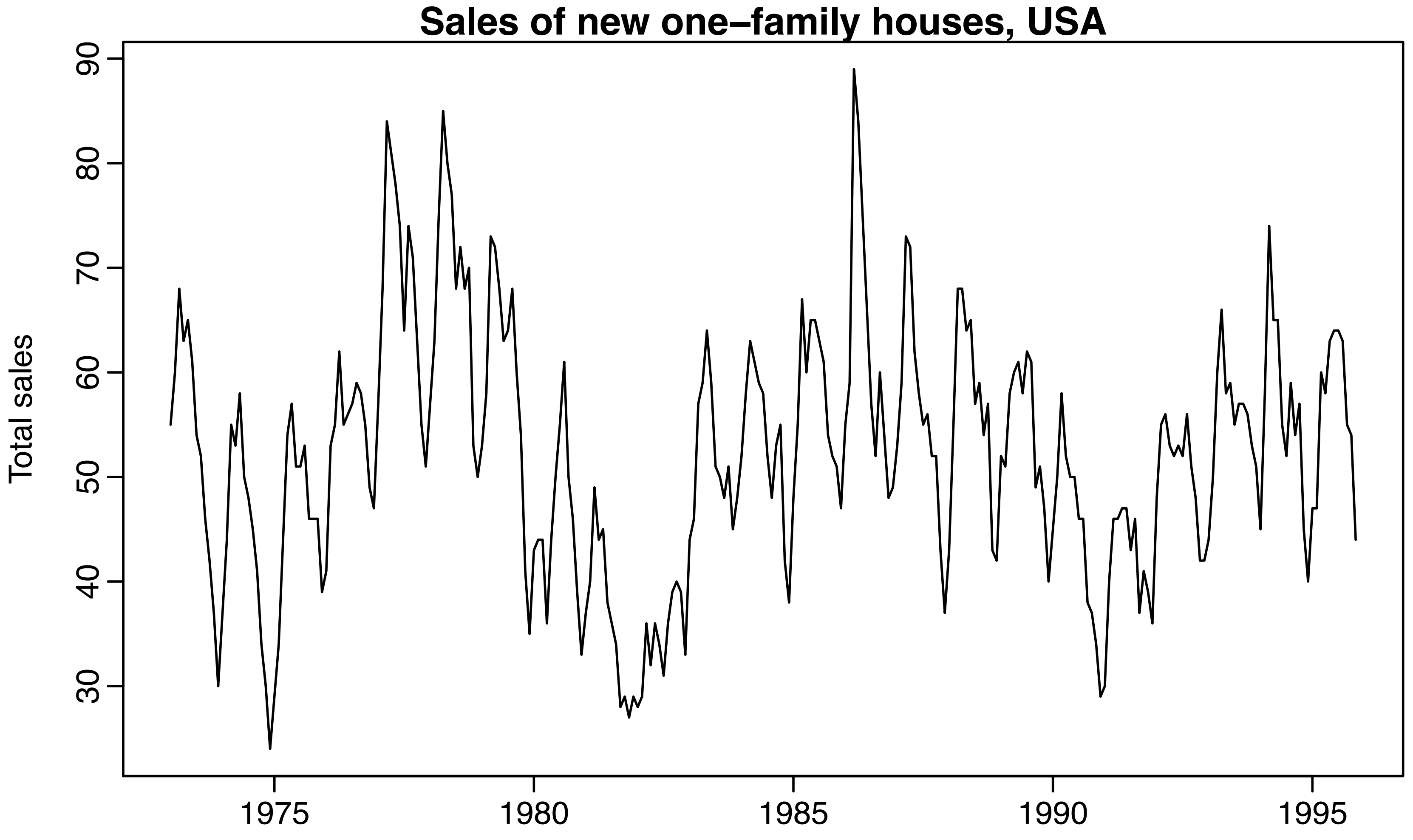
[R. J. Hyndman]

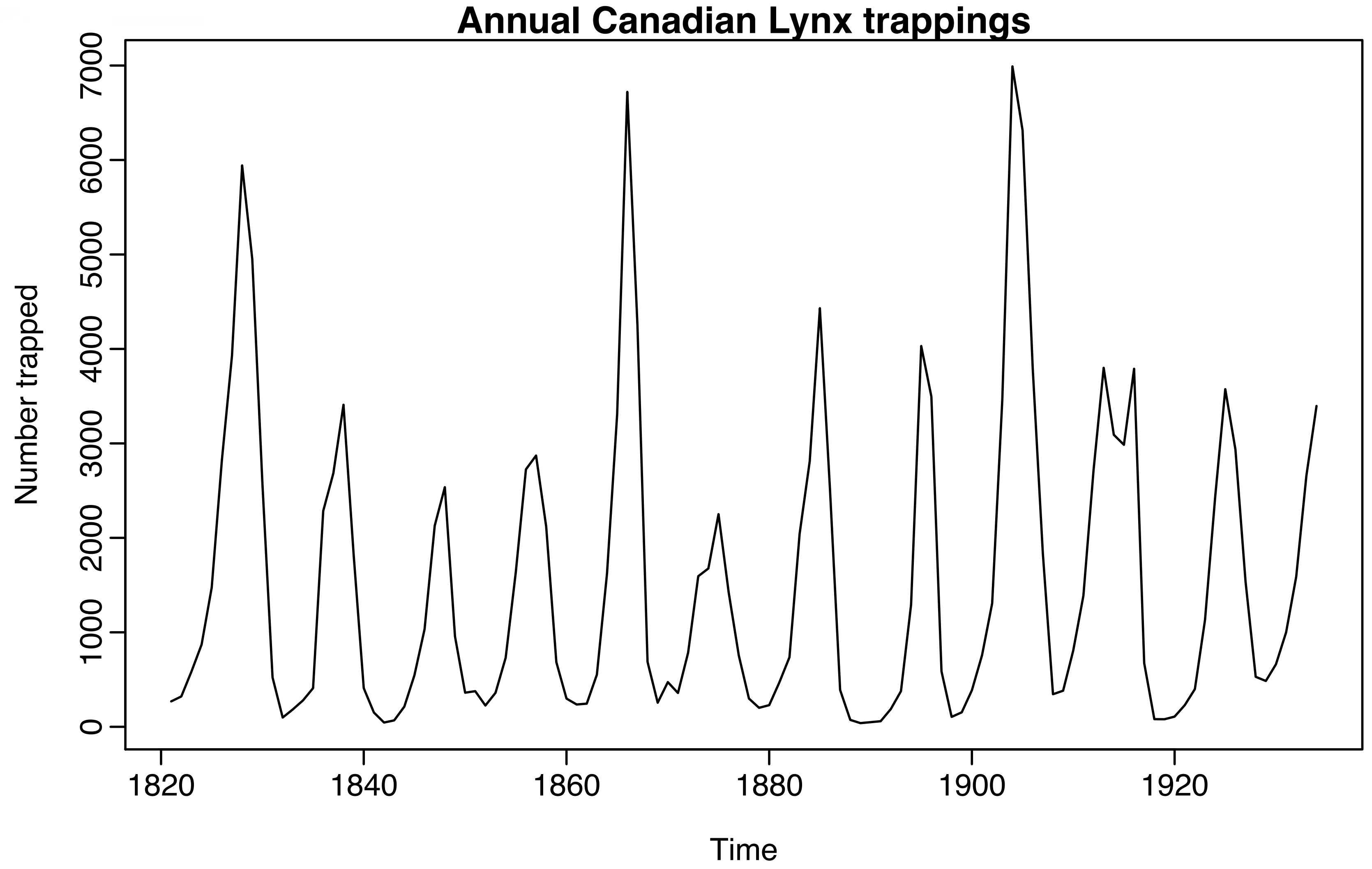
Examples

Trend





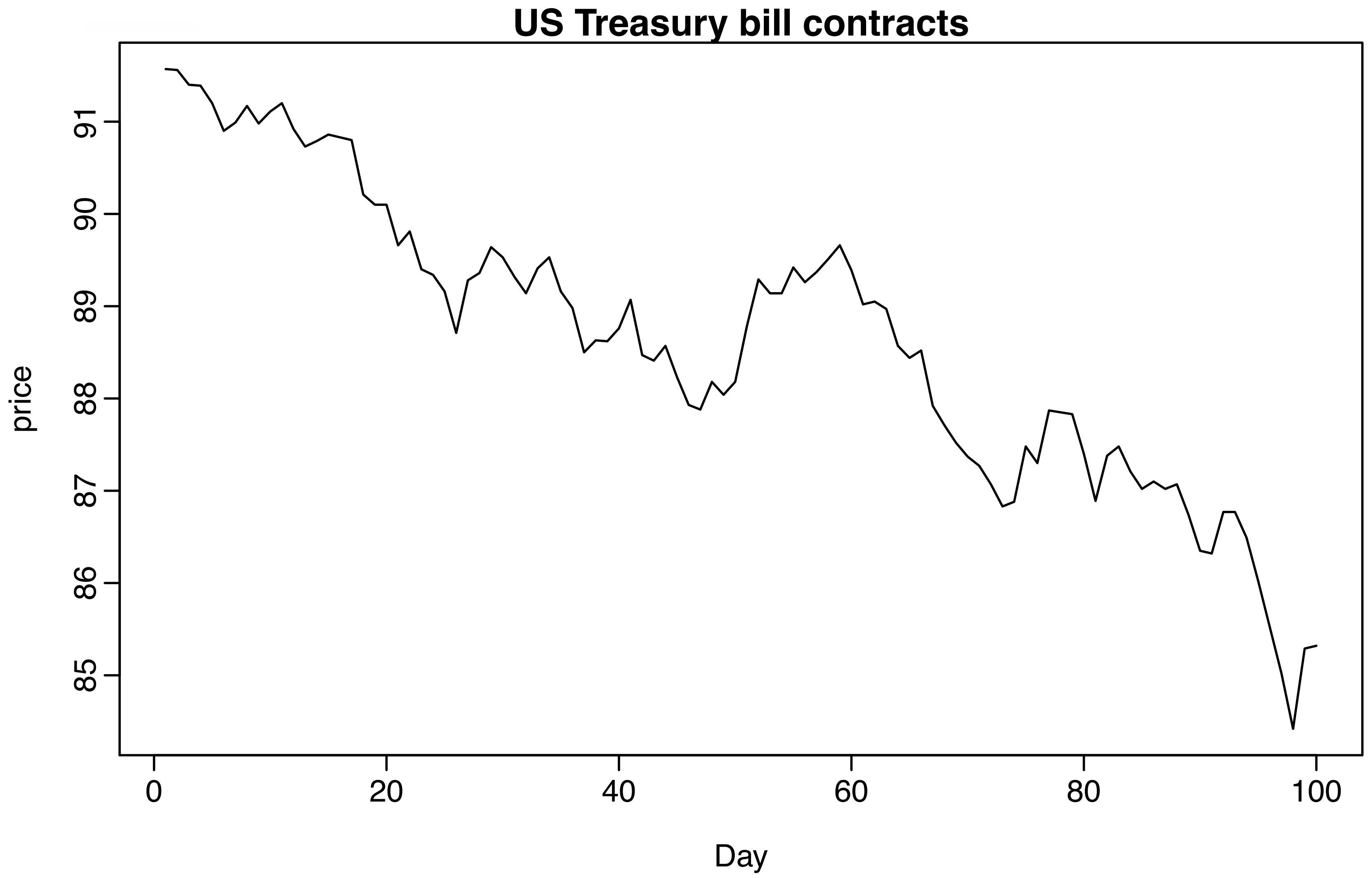


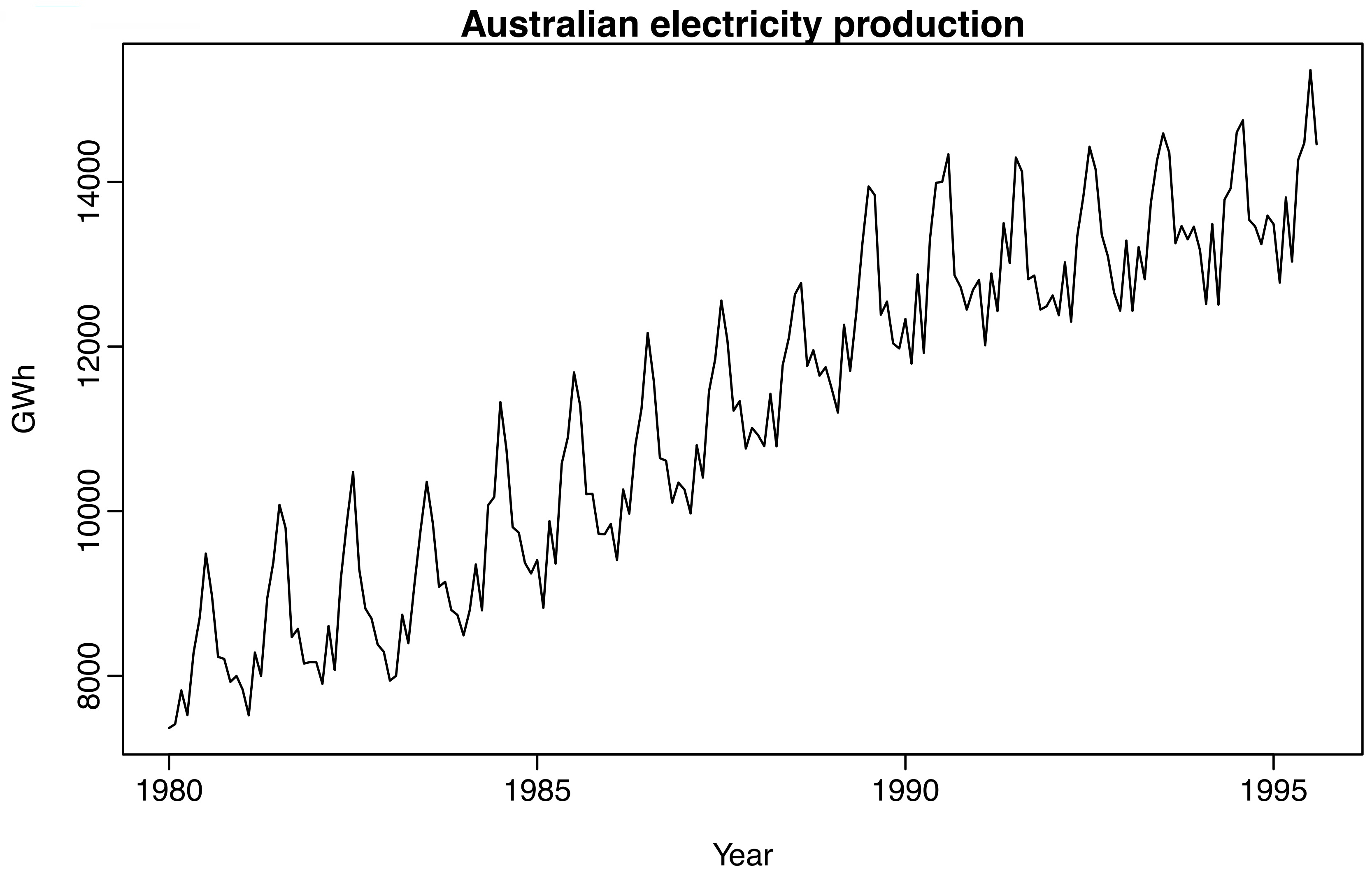


[R. J. Hyndman]

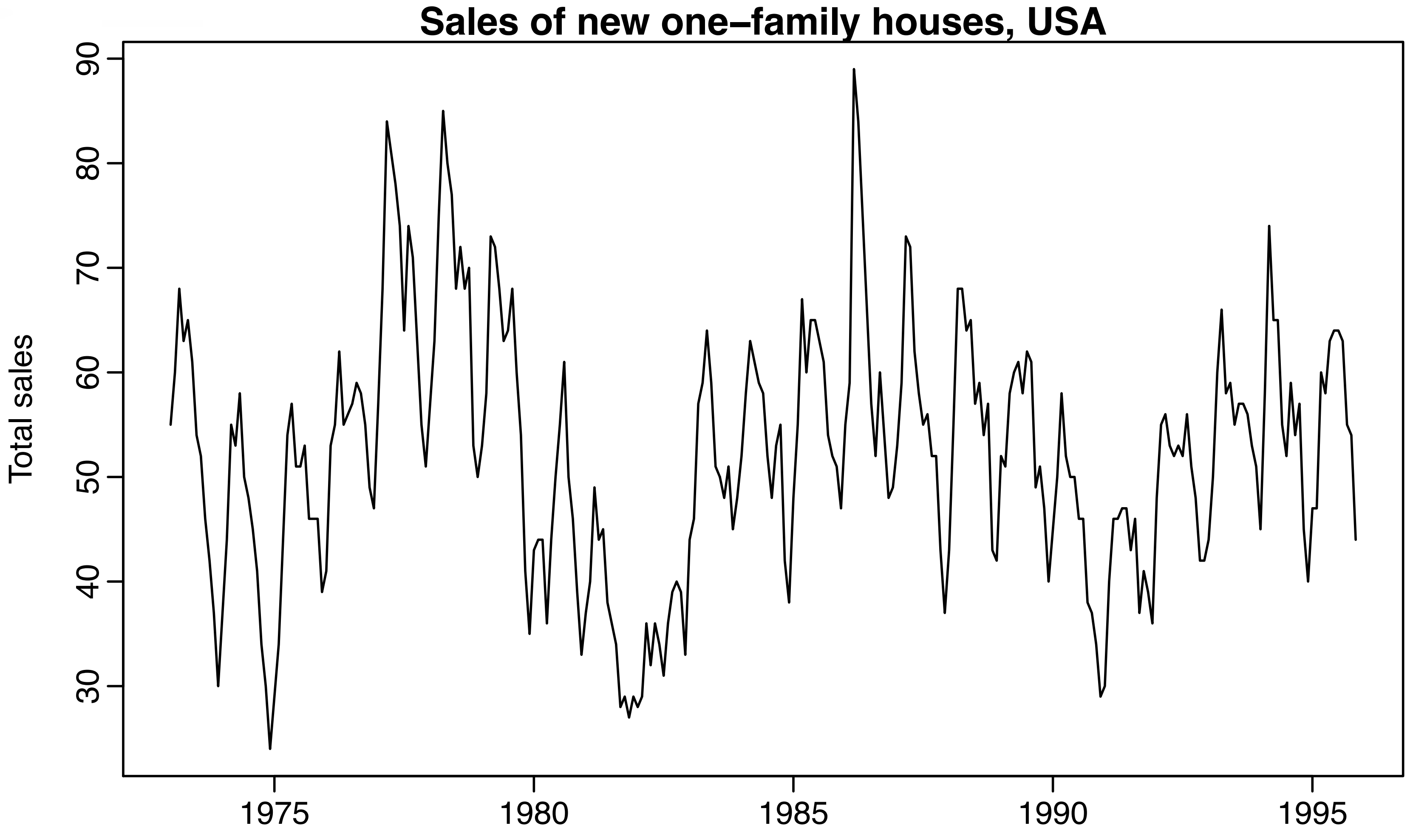
Examples

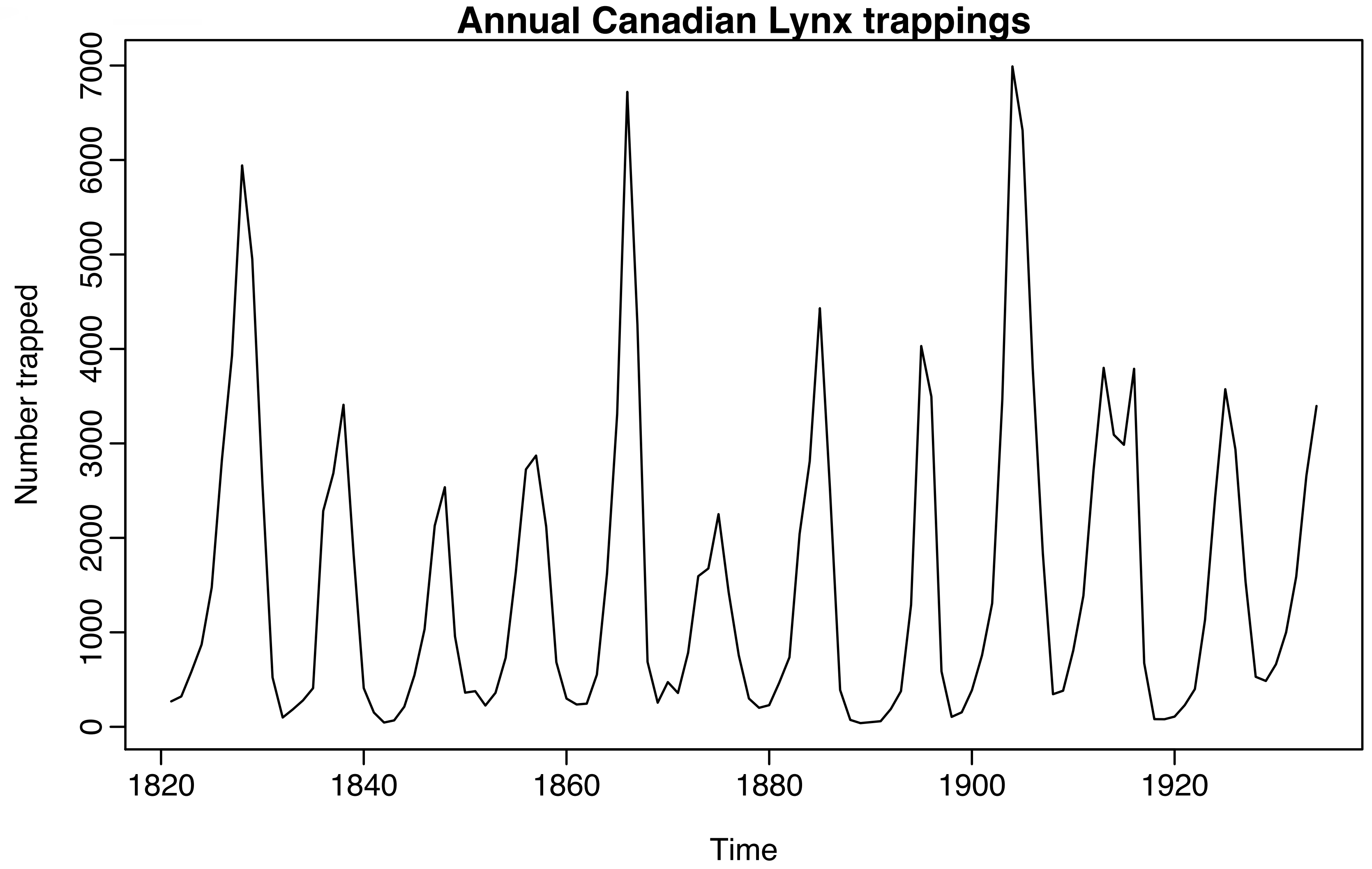
Trend





Trend +
Seasonality

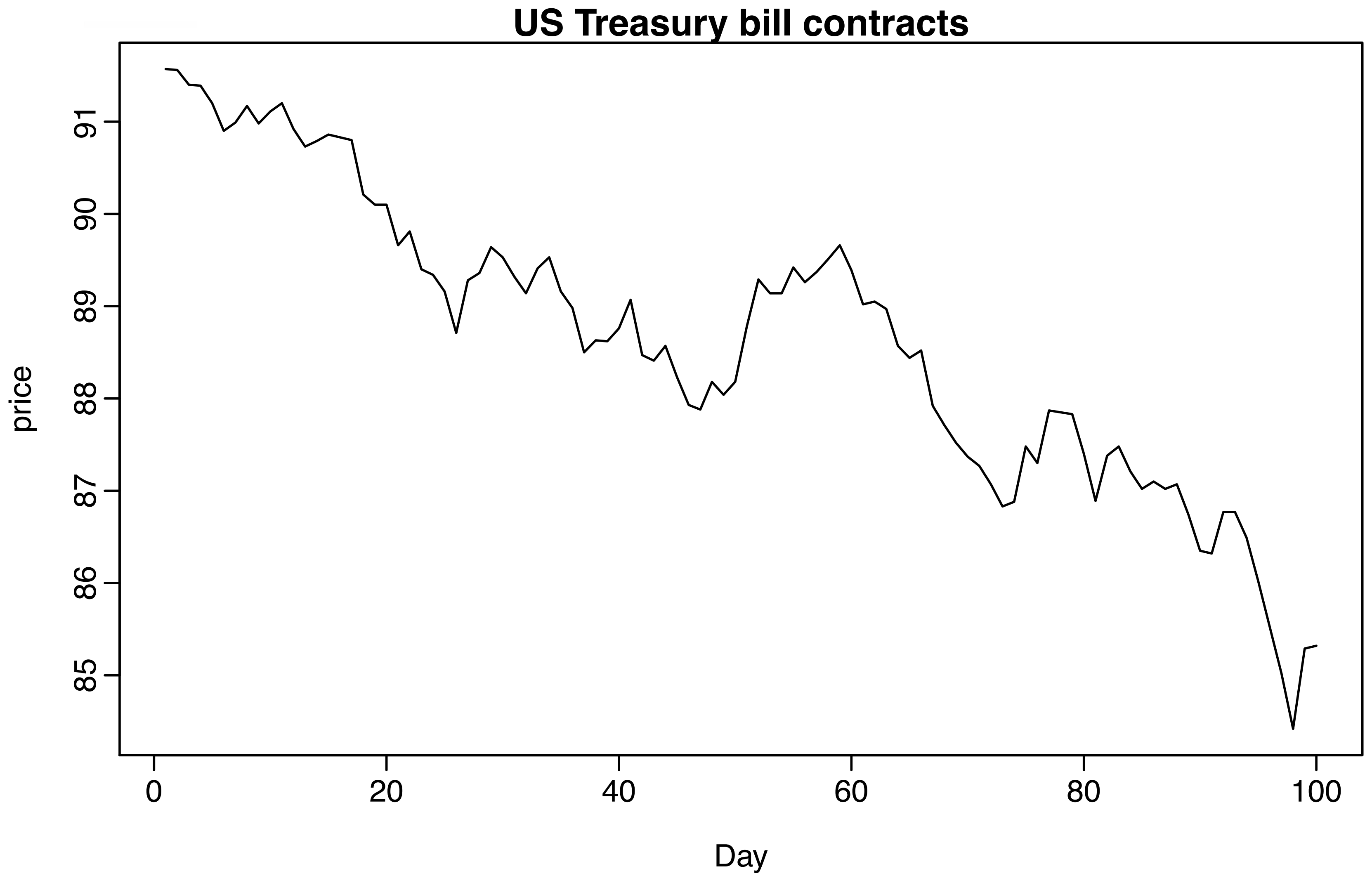


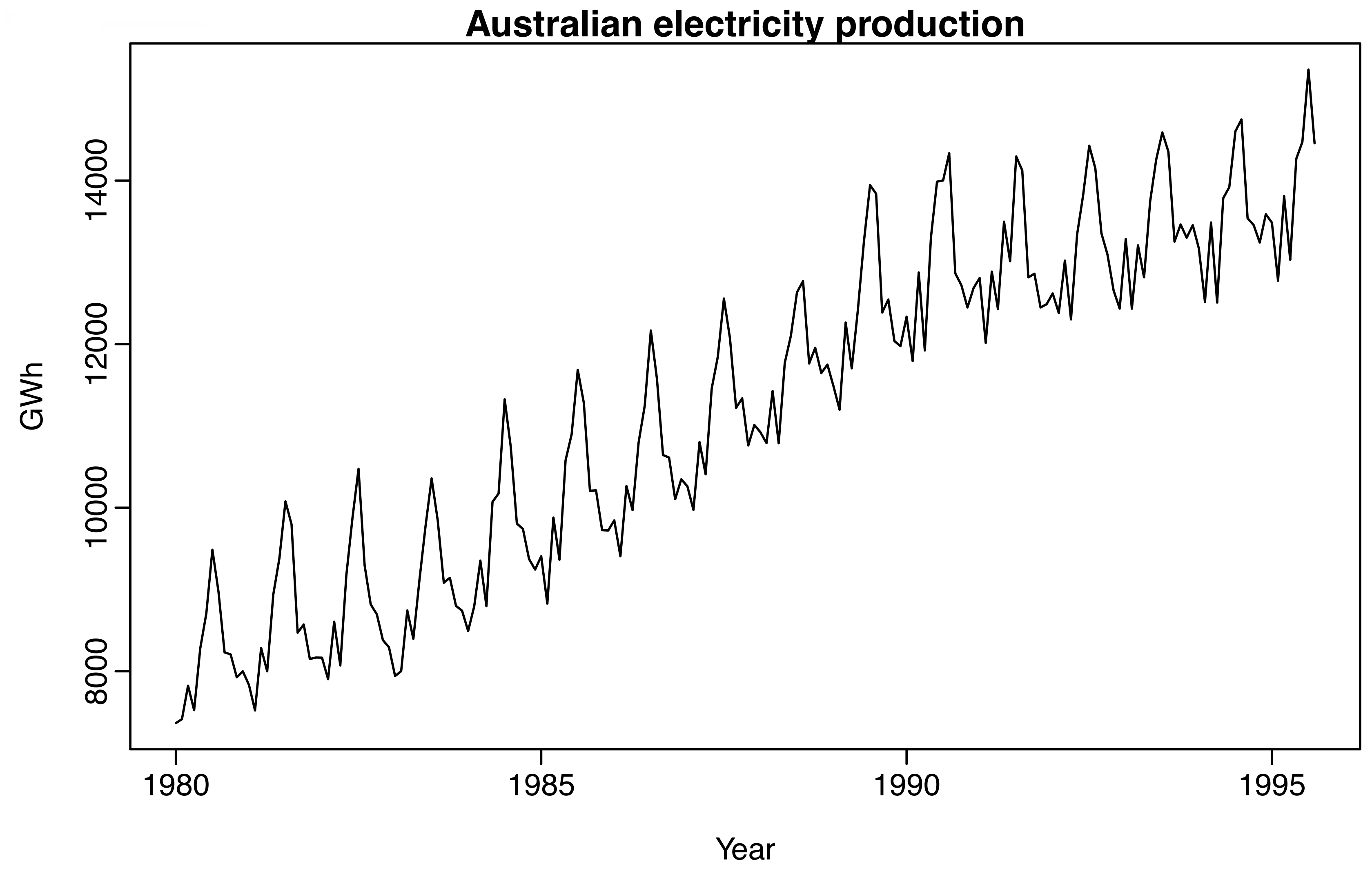


[R. J. Hyndman]

Examples

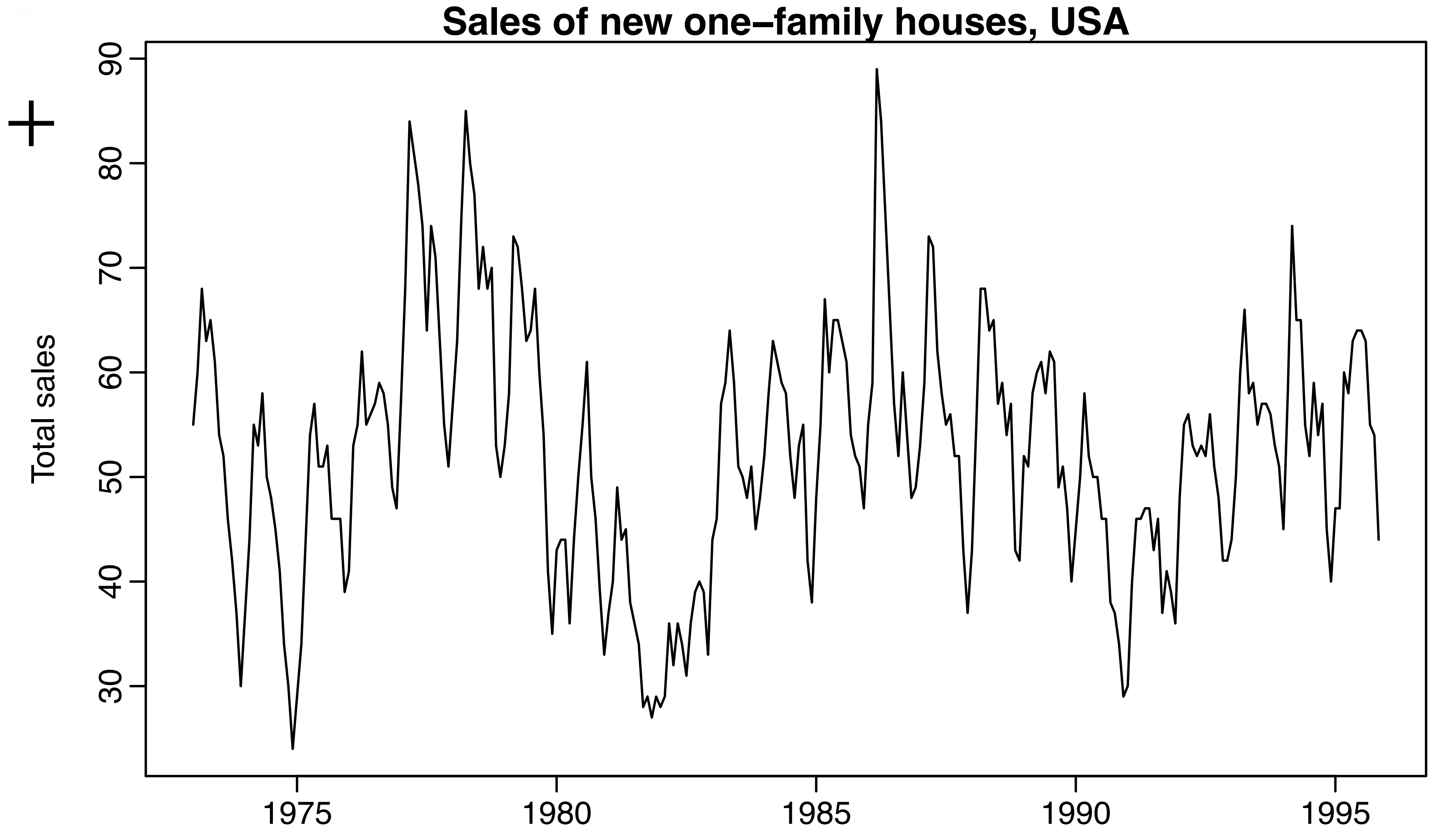
Trend

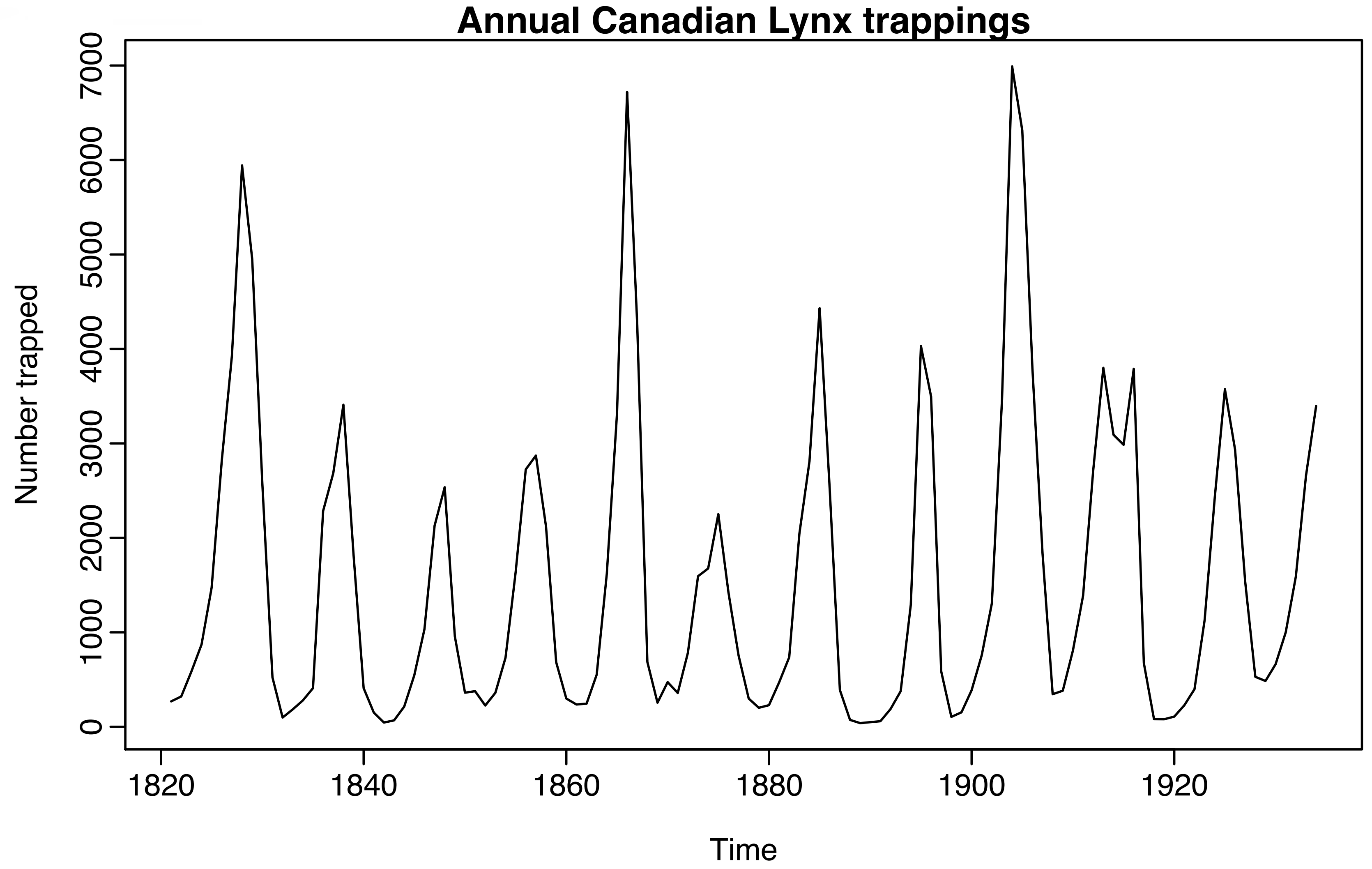




Trend + Seasonality

Seasonality + Cyclic

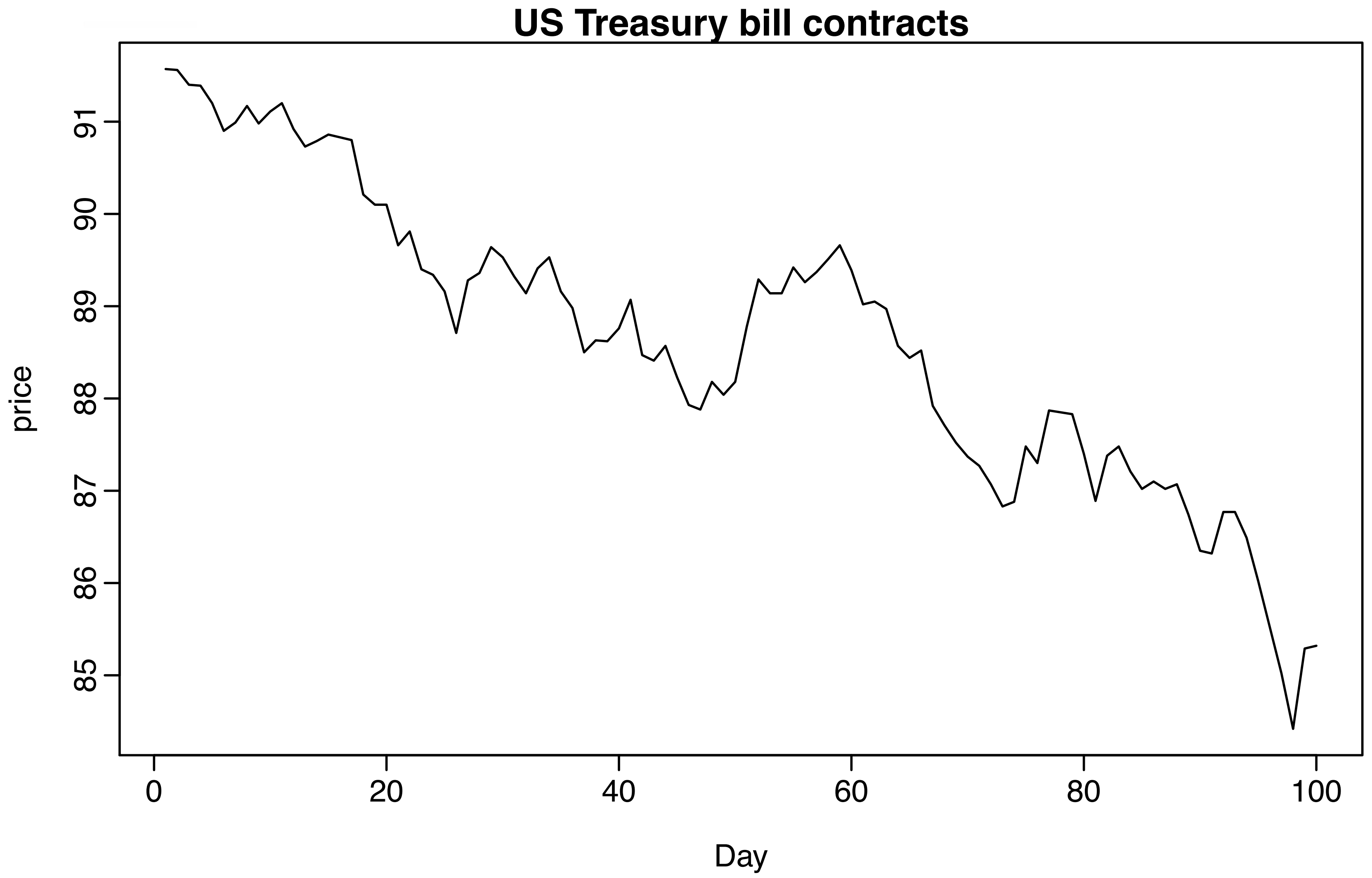




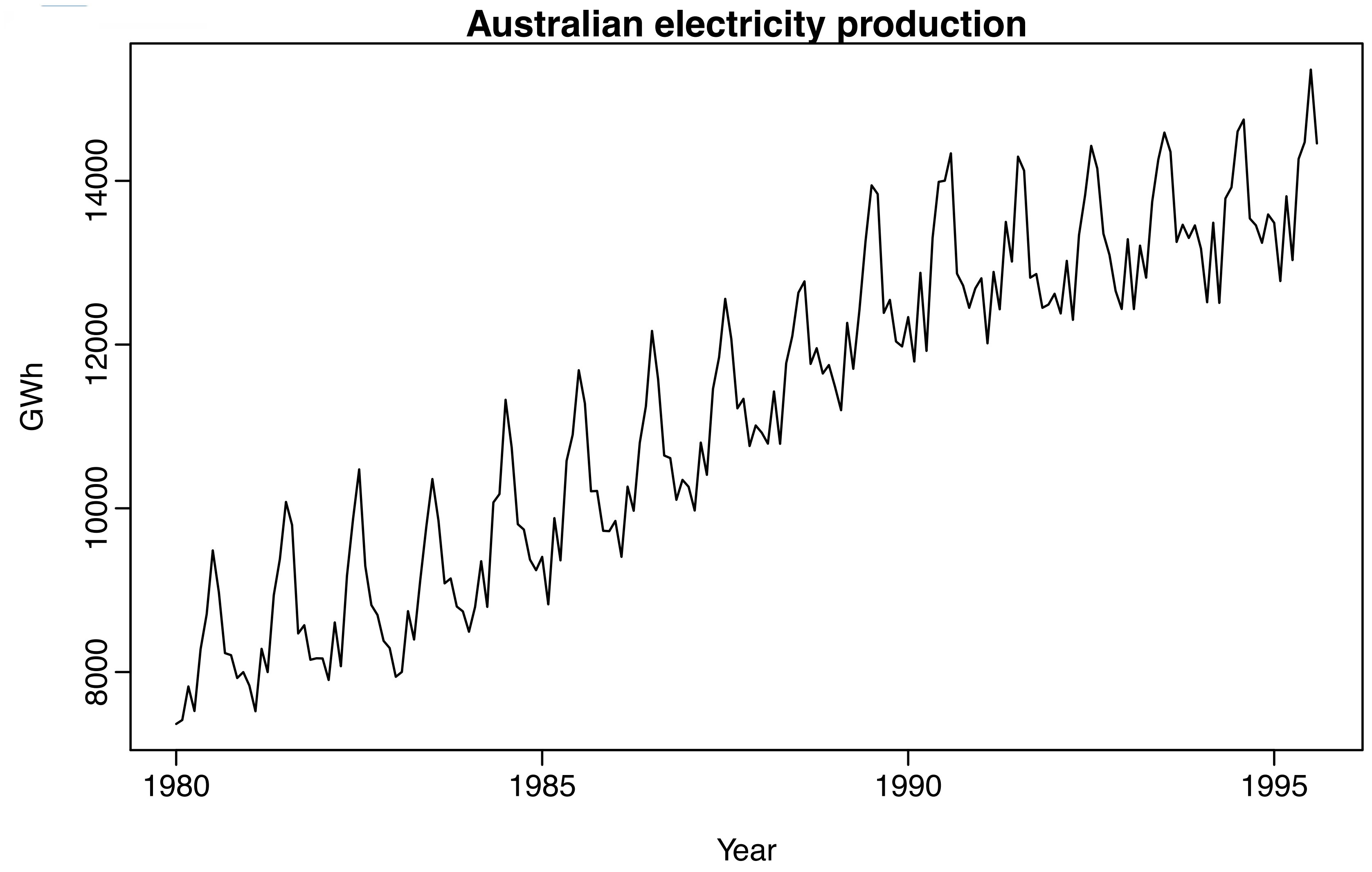
[R. J. Hyndman]

Examples

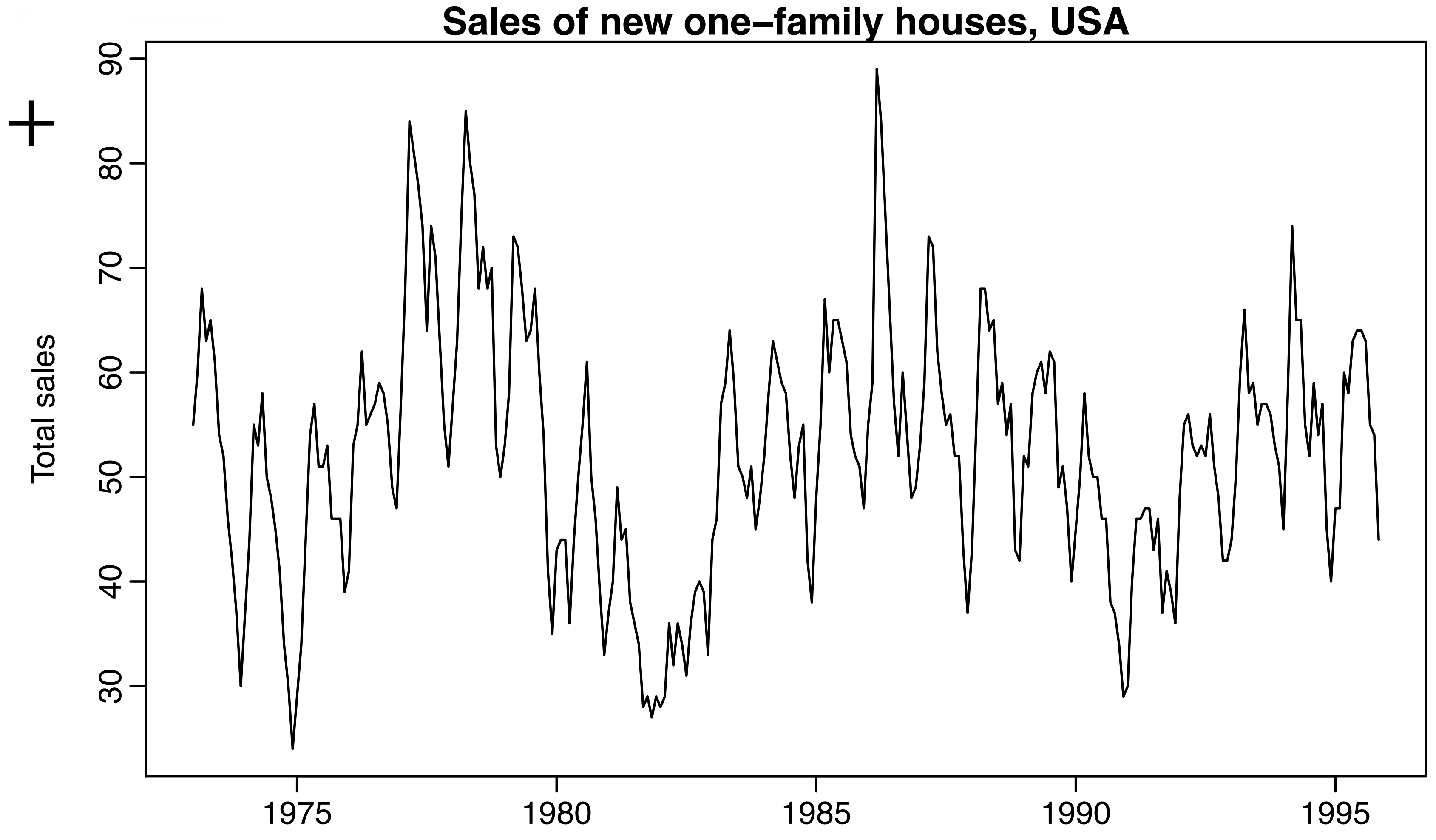
Trend



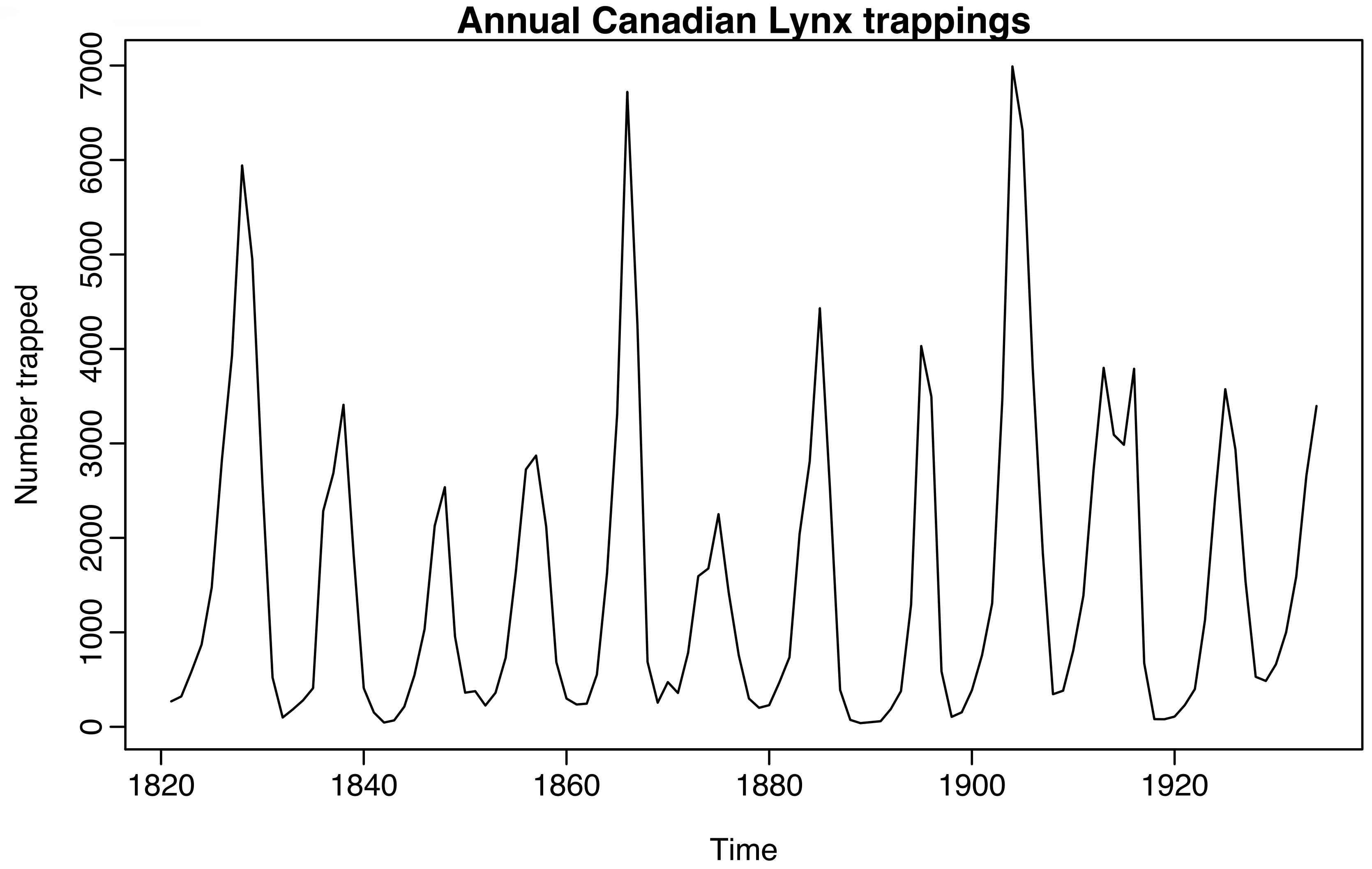
Trend +
Seasonality



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

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]

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]
- DeKalb is UTC-6 (aka US/Central); Daylight Saving Time is UTC-5

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

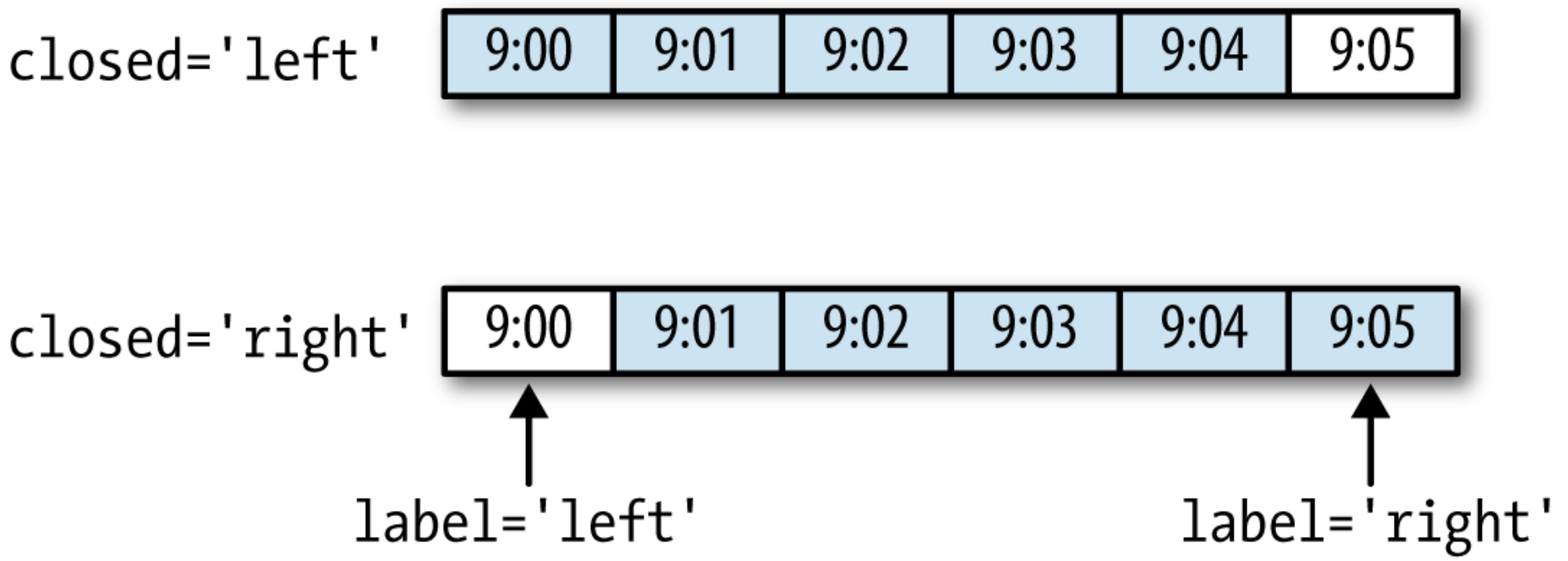
- Could be
  - downsample: higher frequency to lower frequency
  - upsample: lower frequency to higher frequency
  - neither: e.g. Wednesdays to Fridays
- resample method: e.g. `ts.resample('M').mean()`

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

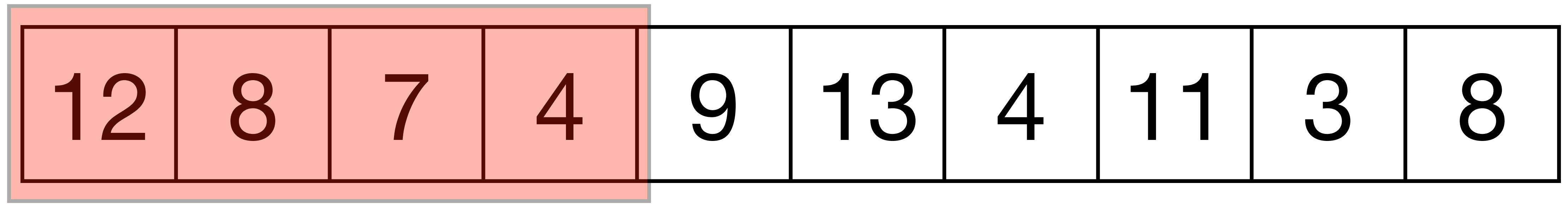
```
In [222]: frame
Out[222]:
            Colorado     Texas  New York      Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-12 -0.046662  0.927238  0.482284 -0.867130
```

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

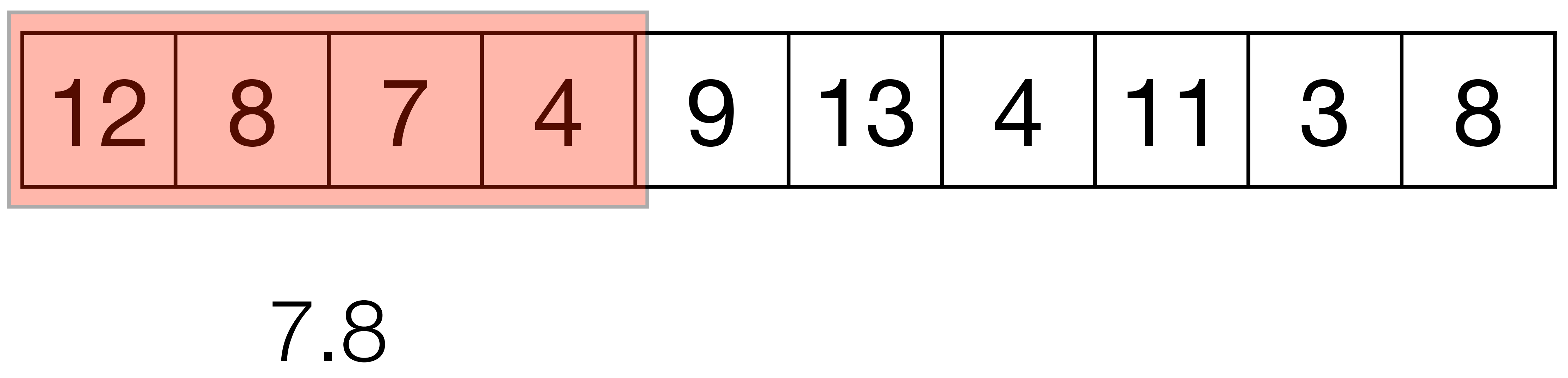
In [224]: df_daily
Out[224]:
            Colorado     Texas  New York      Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-06       NaN       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN       NaN
2000-01-08       NaN       NaN       NaN       NaN
2000-01-09       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
```

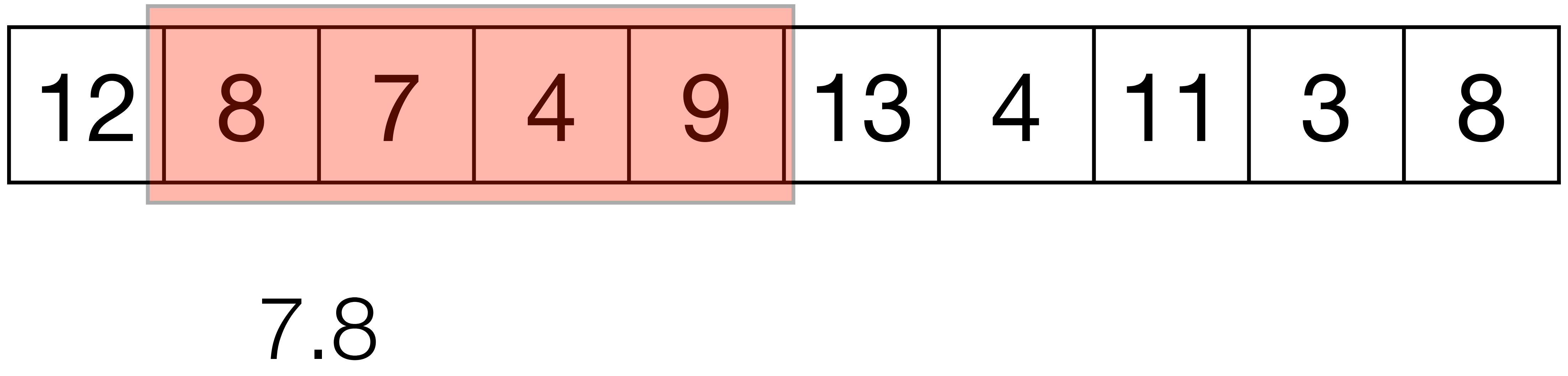
Rolling Window Calculations



Rolling Window Calculations



Rolling Window Calculations



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

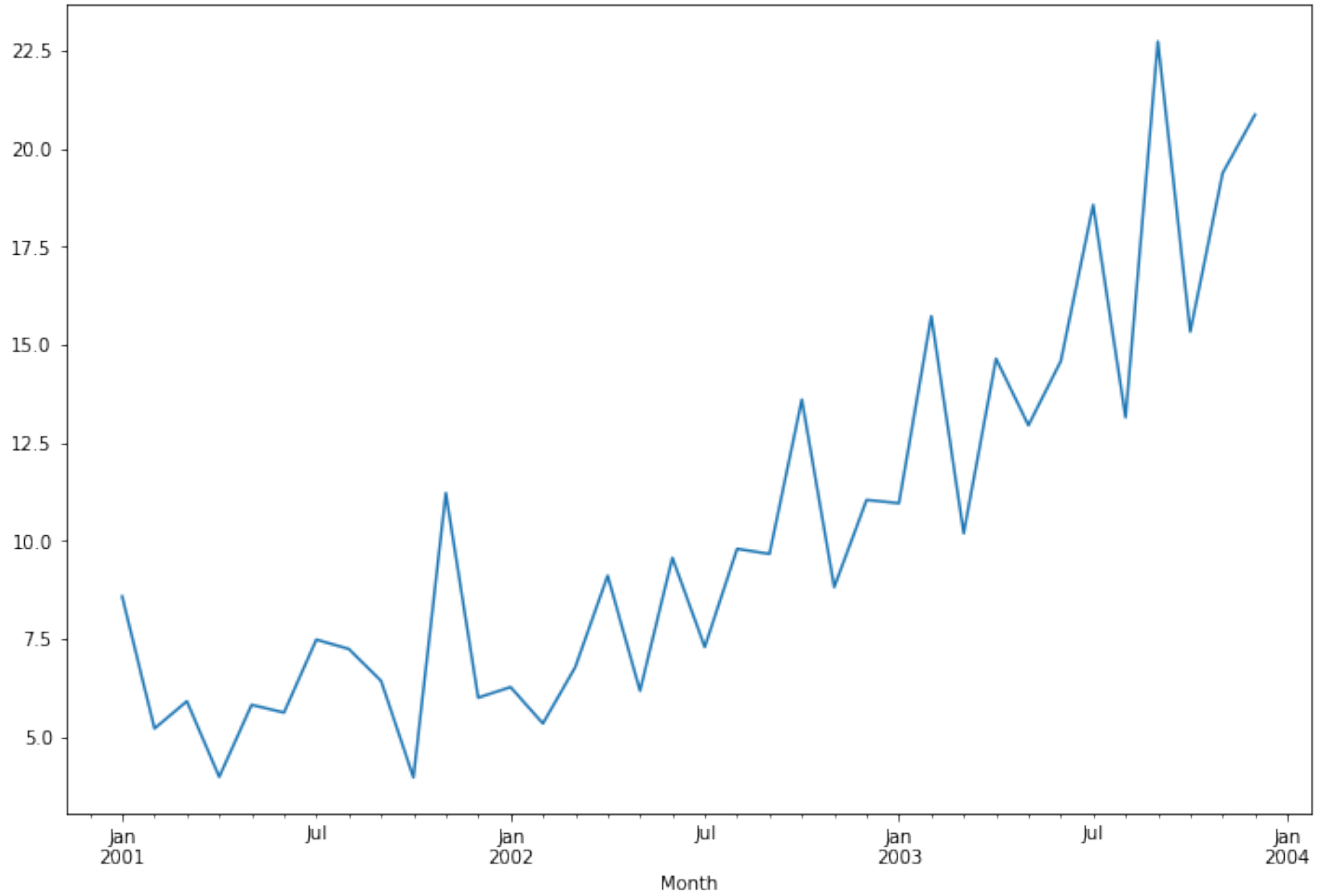
Window Functions

- 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()`

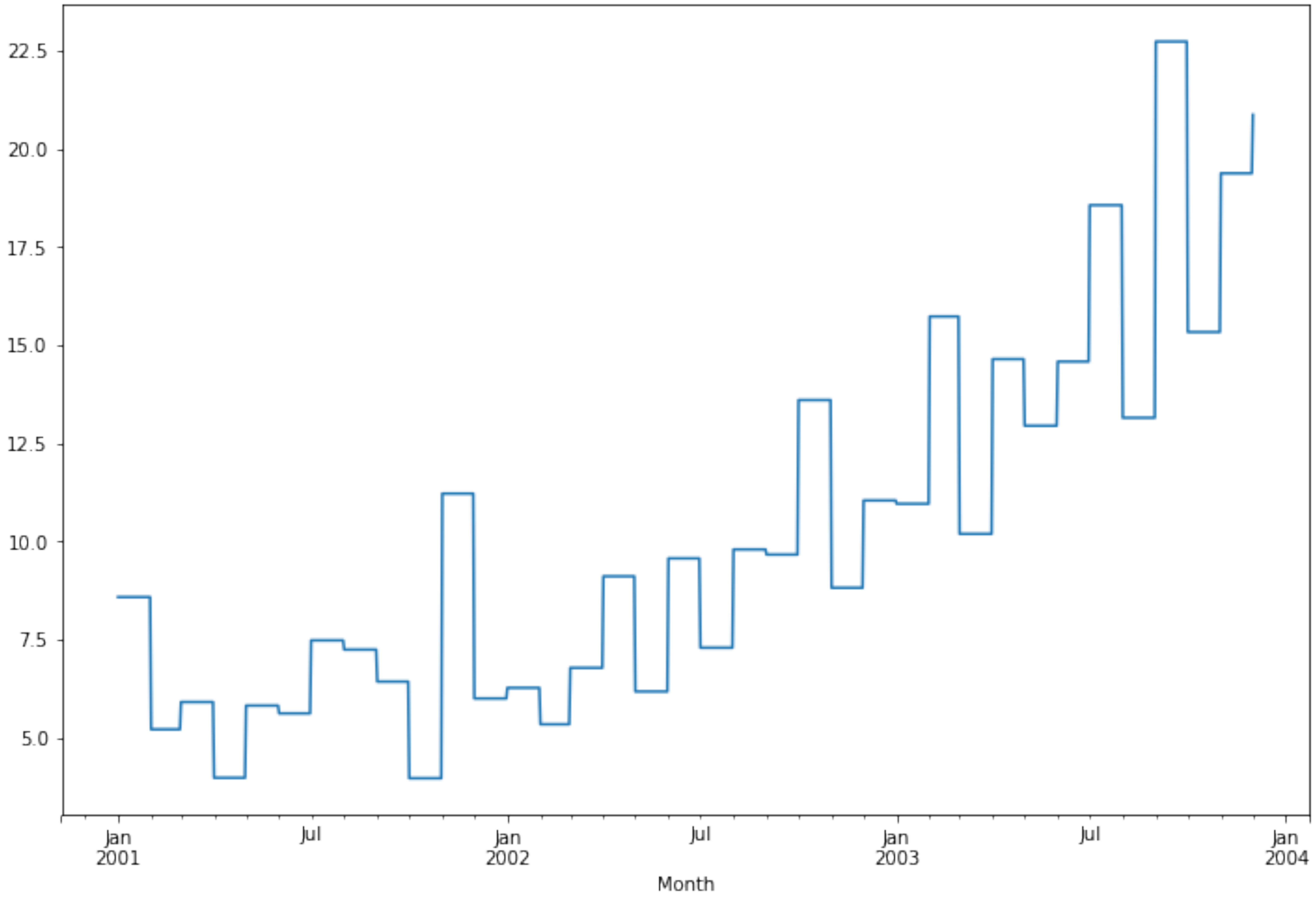
Interpolation

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

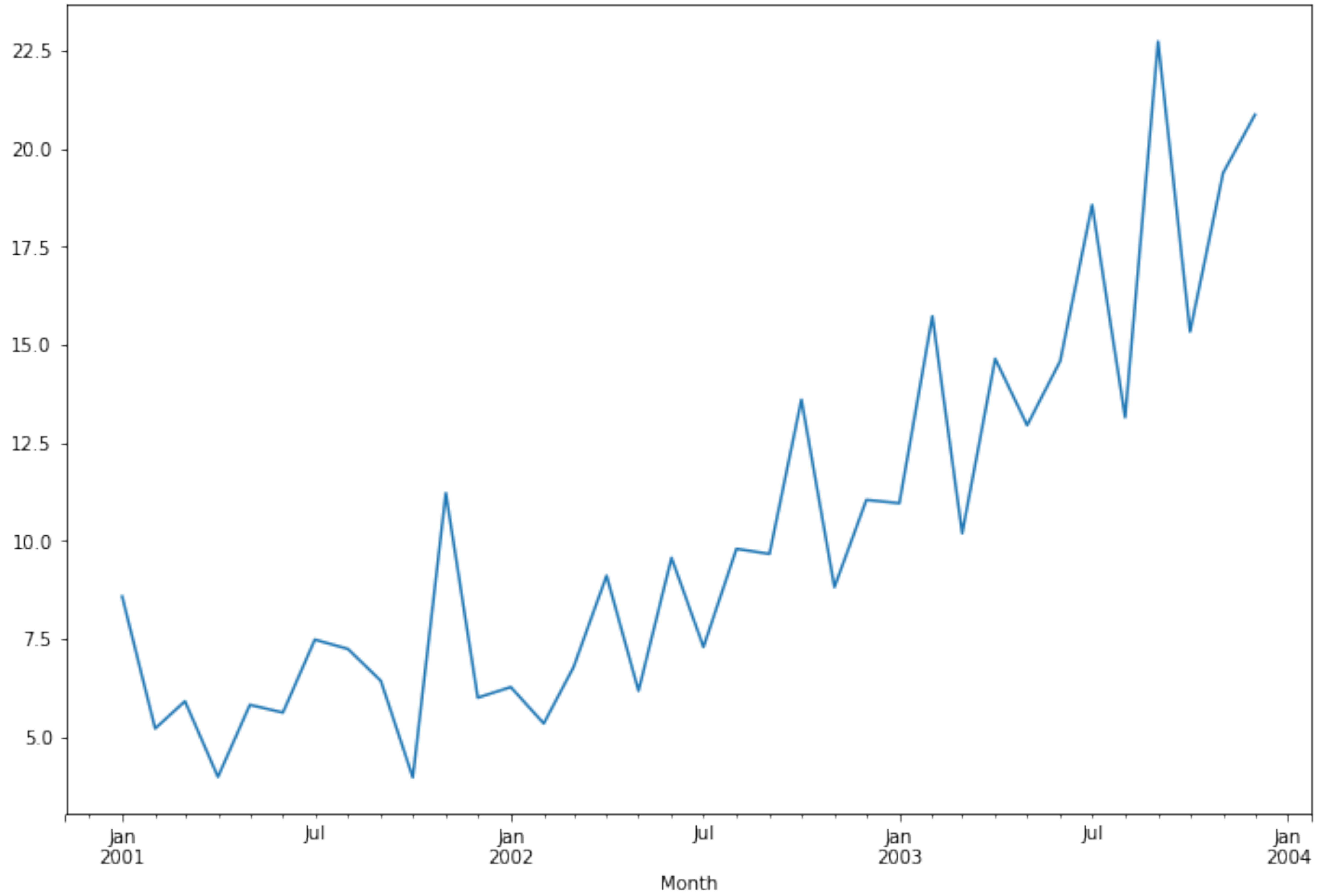
Sales Data by Month



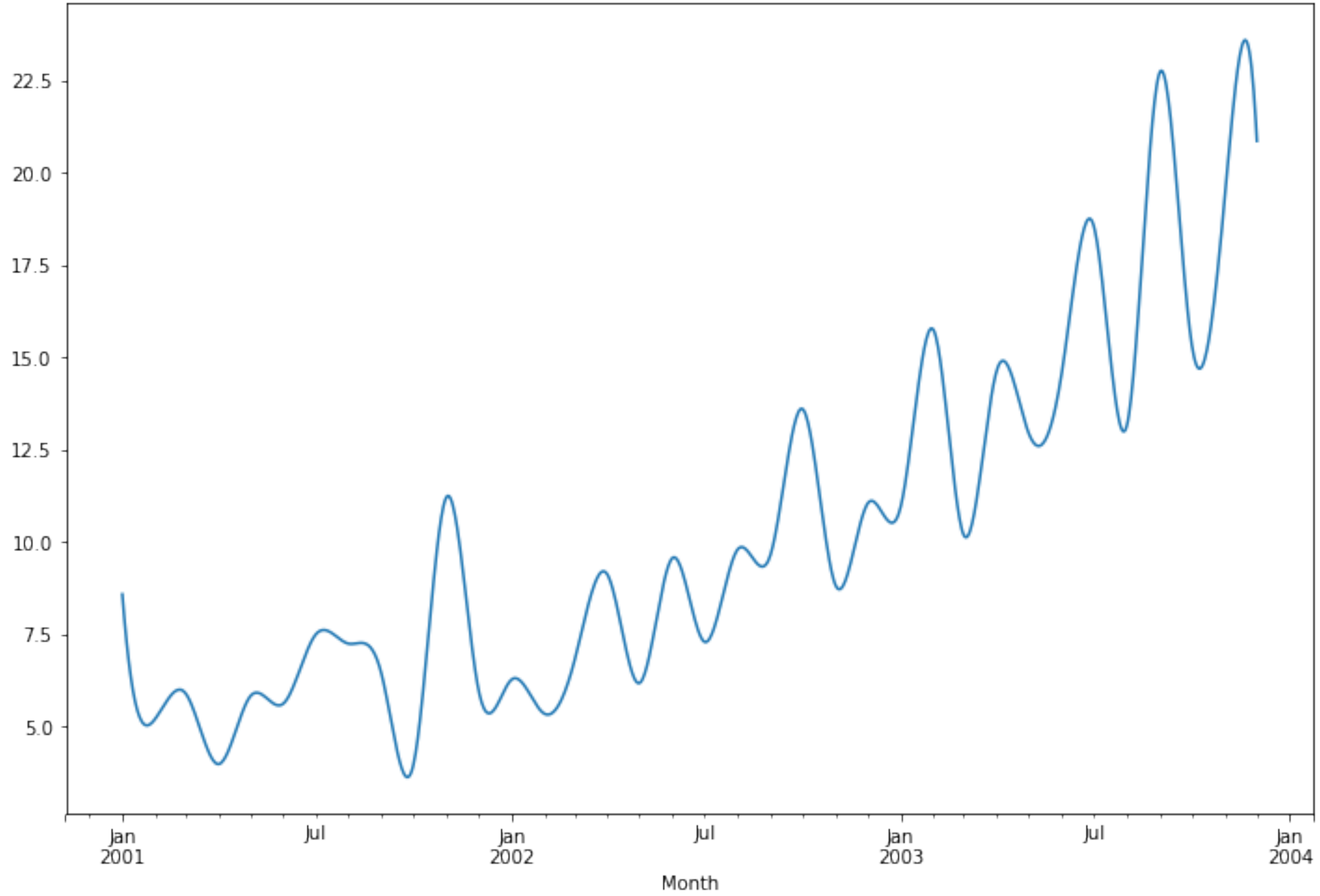
Resampled Sales Data (ffill)



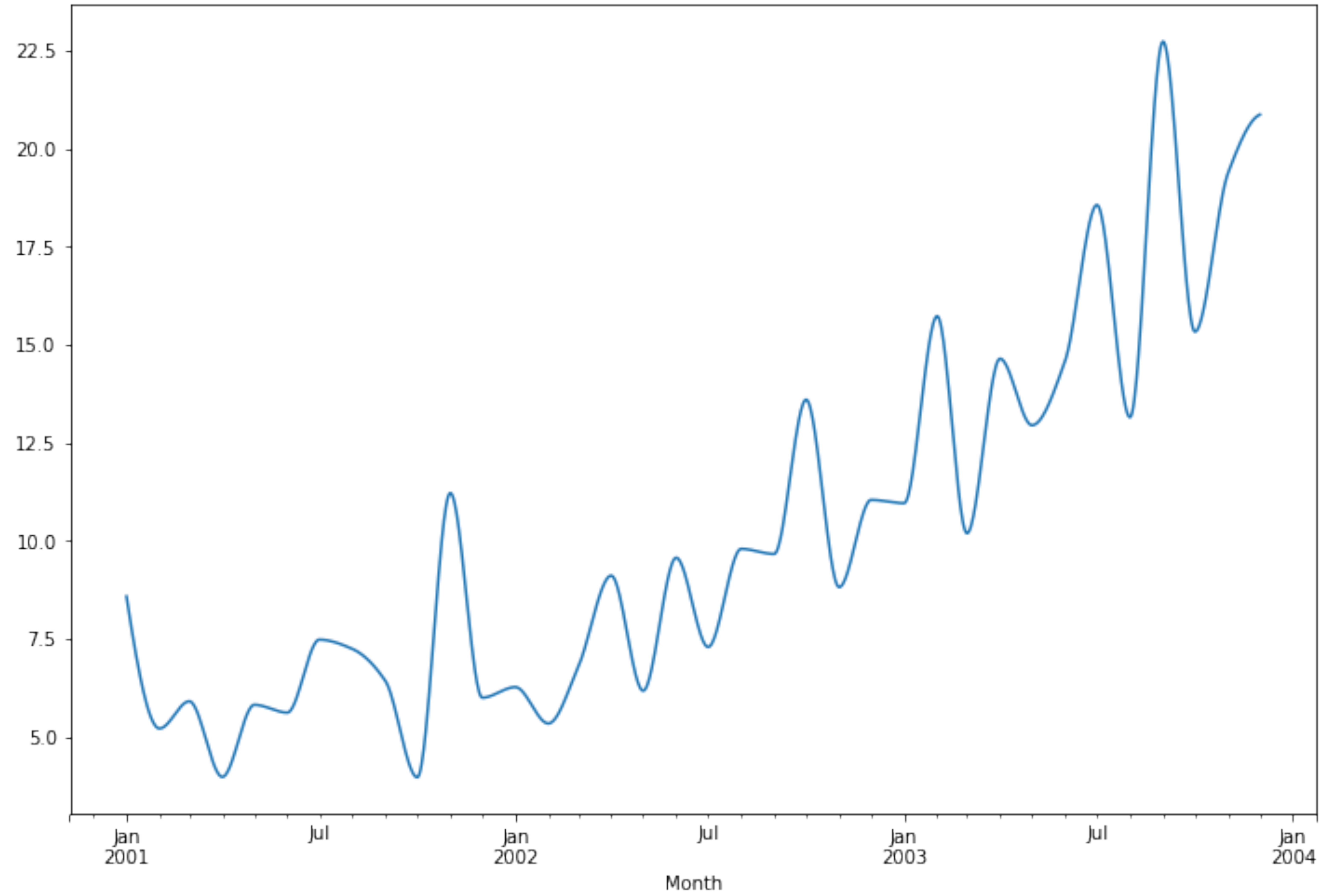
Resampled with Linear Interpolation (Default)



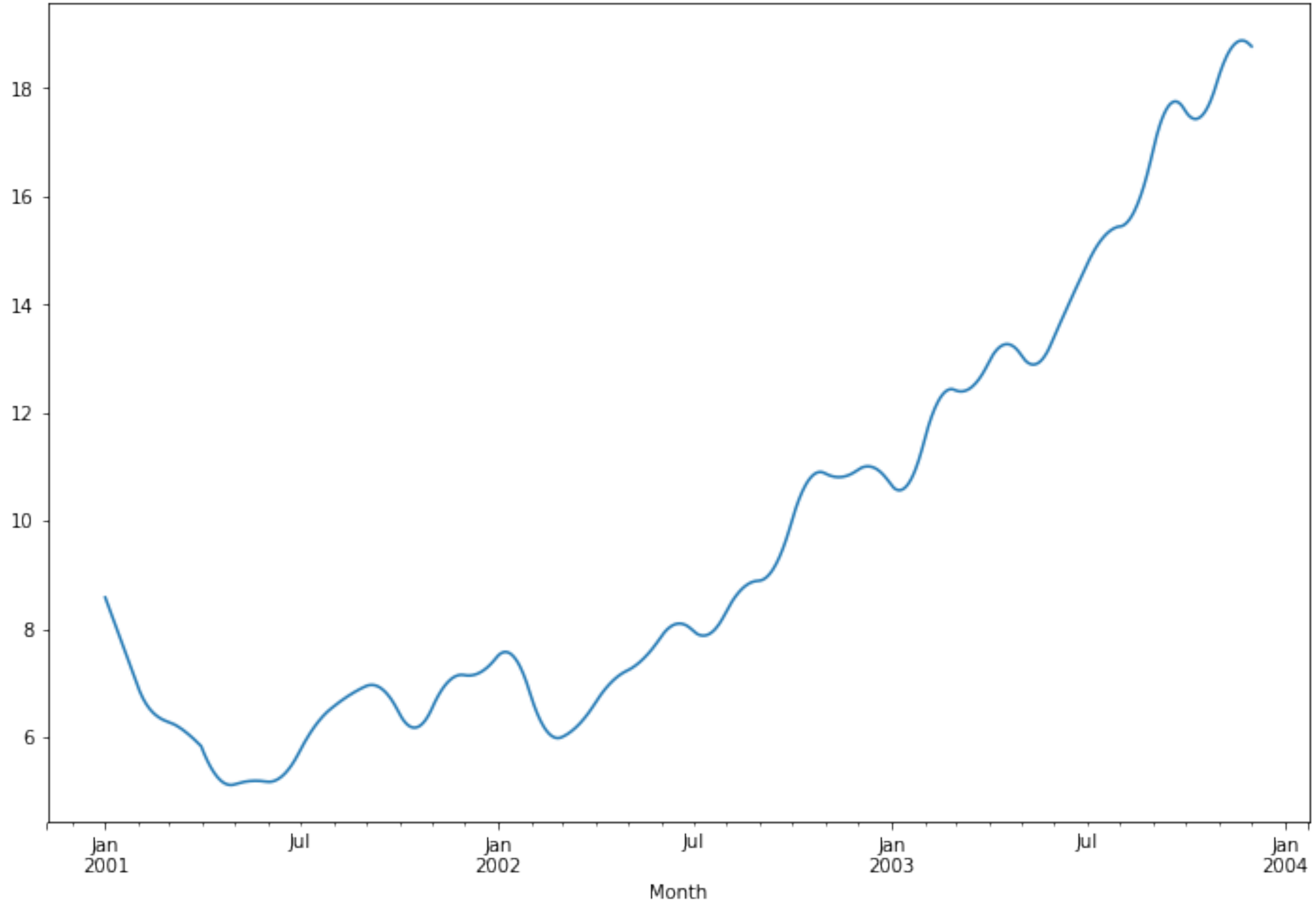
Resampled with Cubic Interpolation



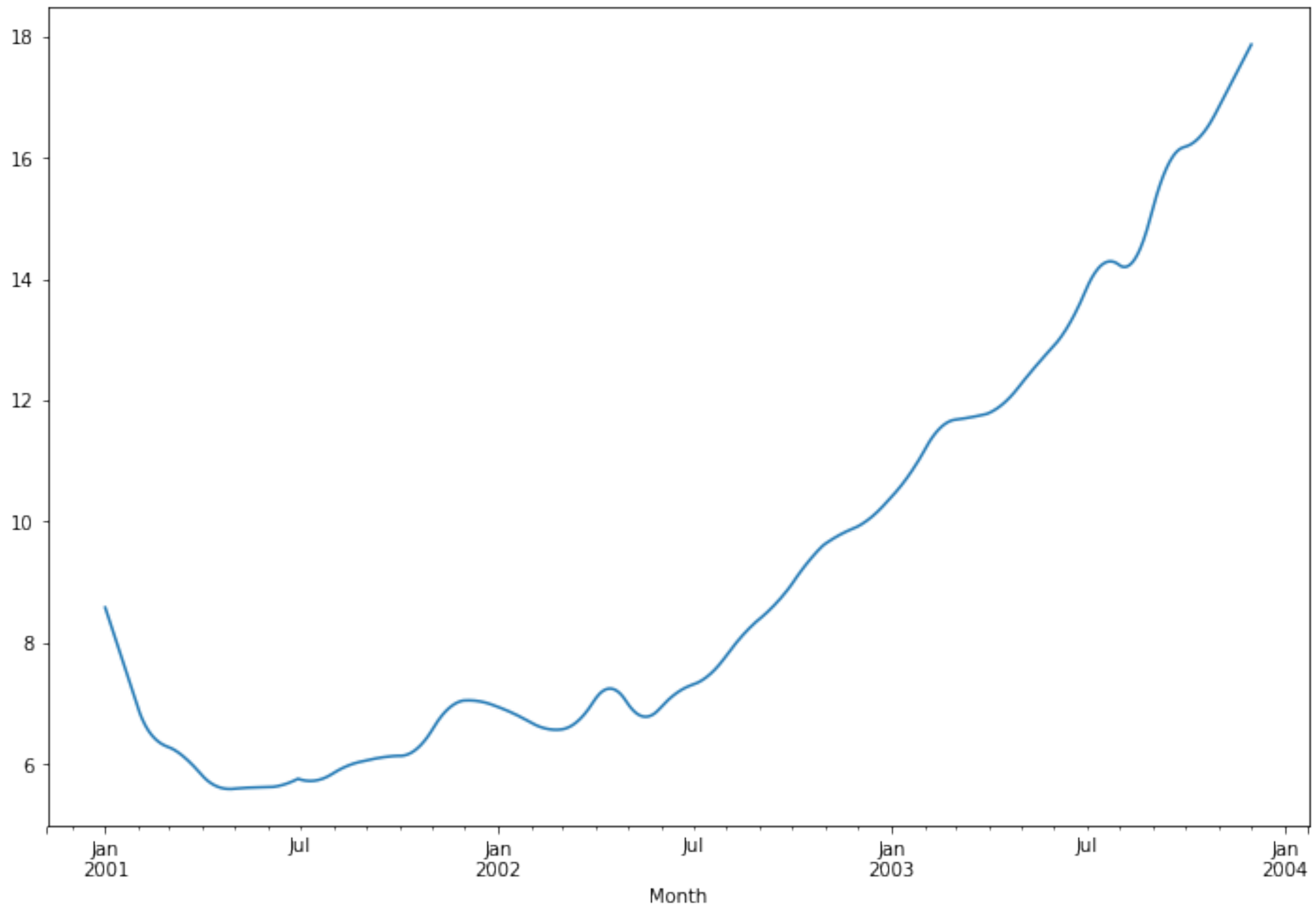
Piecewise Cubic Hermite Interpolating Polynomial



90-Day Rolling Window (Mean)



180-Day Rolling Window (Mean)



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)

Time Series Database Motivation

- Boeing 787 produces 500GB sensor data per flight
- Purposes
  - IoT
  - Monitoring large industrial installations
  - Data analytics
- Metrics (regular) and Events (irregular)
- Events can be obtained from metrics via binning

What is a Time Series Database?

- A DBMS is called TSDB if it can
  - store a row of data that consists of timestamp, value, and optional tags
  - store multiple rows of time series data grouped together
  - can query for rows of data
  - can contain a timestamp or a time range in a query

ul1

"SELECT * FROM ul1 WHERE time >= '2016-07-12T12:10:00Z'"

time	generated	message_subtype	scaler	short_id	tenant	value
2016-07-12T11:51:45Z	"true"	"34"	"4"	"3"	"saarlouis"	465110000
2016-07-12T11:51:45Z	"true"	"34"	"-6"	"2"	"saarlouis"	0.06196699999999994
2016-07-12T12:10:00Z	"true"	"34"	"7"	"5"	"saarlouis"	49370000000
2016-07-12T12:10:00Z	"true"	"34"	"6"	"2"	"saarlouis"	18573000000
2016-07-12T12:10:00Z	"true"	"34"	"5"	"7"	"saarlouis"	5902300000

[A. Bader, 2017]

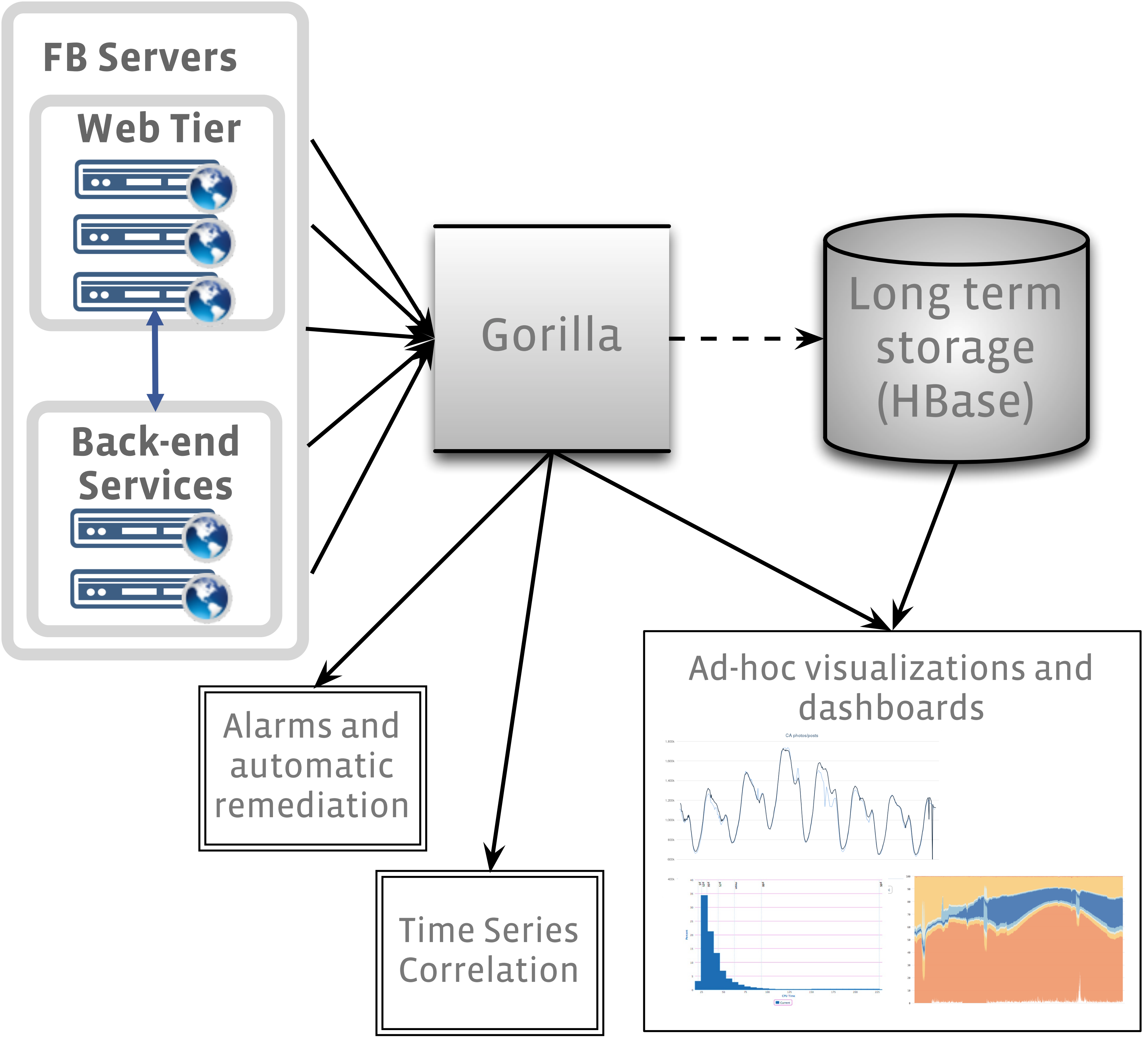
Storing Time Series Data in a RDBMS

- Timestamp as a primary key
- Tags and timestamp as combined primary key
- Use an auto-incrementing primary key (timestamp is a normal attribute)

[A. Bader]

Gorilla Motivation

- Large-scale internet services rely on lots of services and machines
- Want to monitor the health of the systems
- Writes dominate
- Want to detect state transitions
- Must be highly available and fault tolerant



[Pelkonen et al., 2015]

Gorilla Requirements

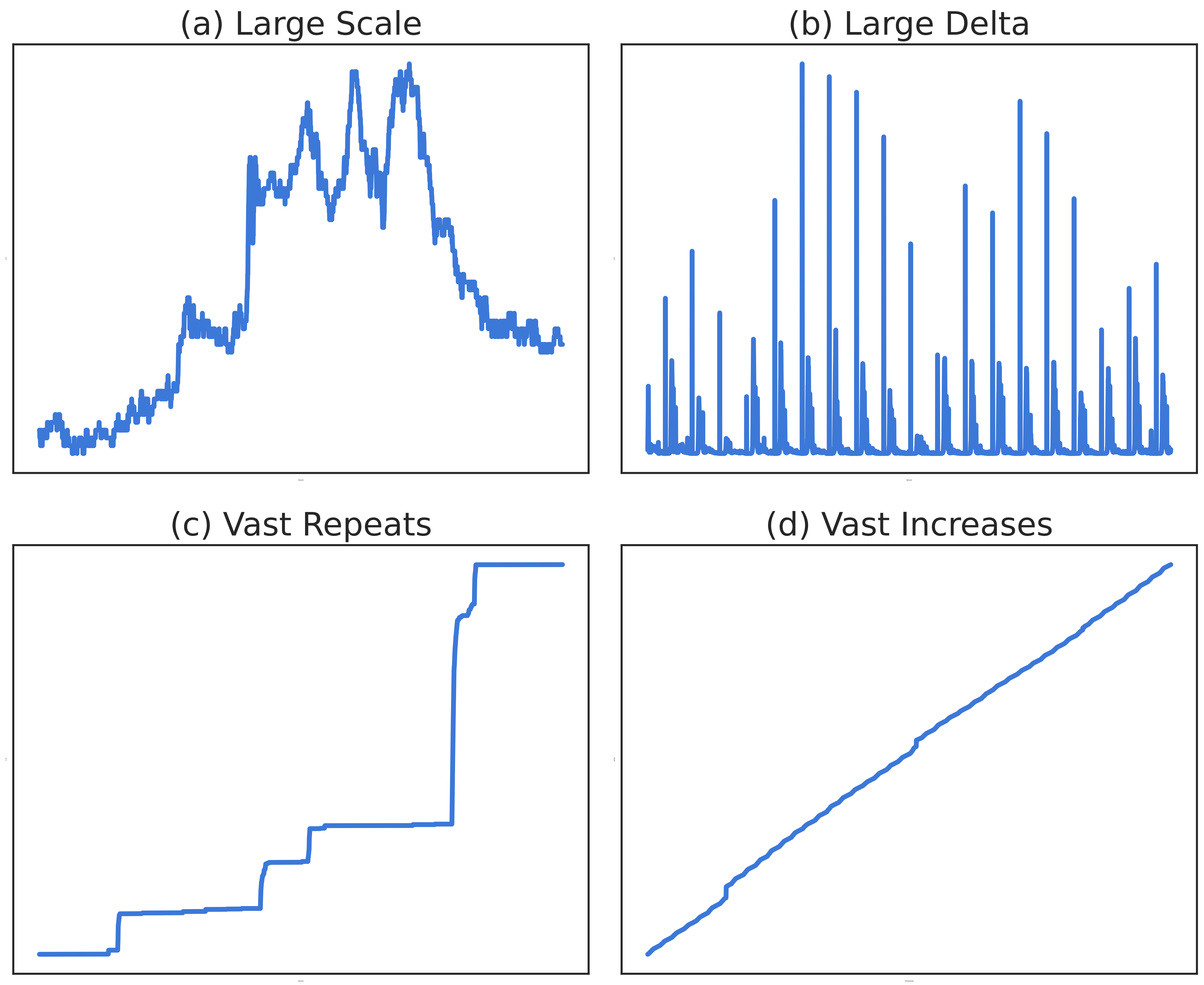
- 2 billion unique time series identified by a string key.
- 700 million data points (time stamp and value) added per minute.
- Store data for 26 hours.
- More than 40,000 queries per second at peak.
- Reads succeed in under one millisecond.
- Support time series with 15 second granularity (4 pts/minute per time series).
- Two in-memory, not co-located replicas (for disaster recovery capacity).
- Always serve reads even when a single server crashes.
- Ability to quickly scan over all in memory data.
- Support at least 2x growth per year.

[Pelkonen et al., 2015]

Gorilla

- In-memory DB
- Data: 3-tuple string key, 64-bit timestamp integer, double-precision float
- Integer compression didn't work

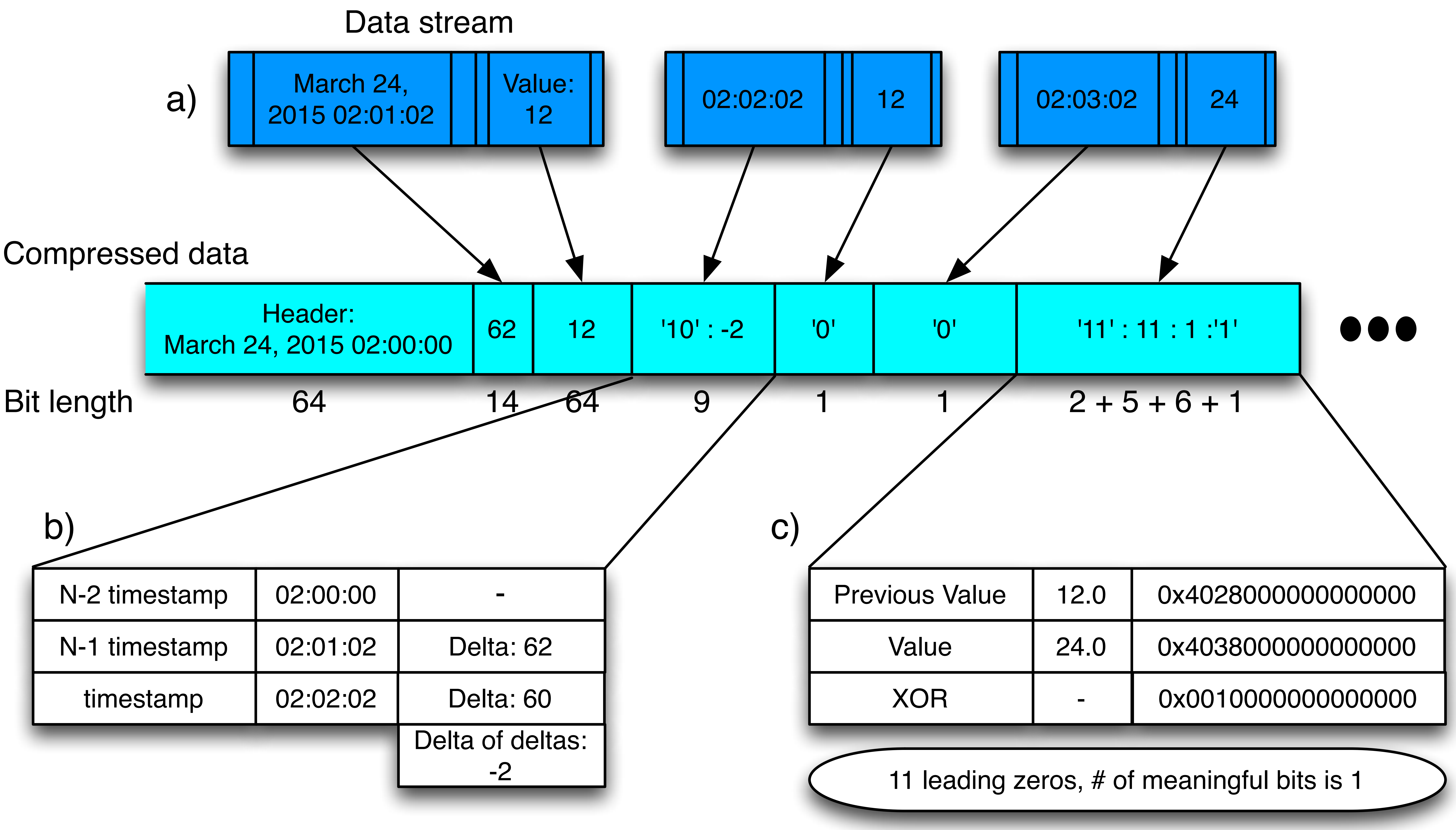
Time Series Data Patterns



- Numerical Data Features:
  - Scale
  - Delta
  - Repeat
  - Increase
- Text Data Features
  - Value
  - Character

[J. Xiao, 2021]

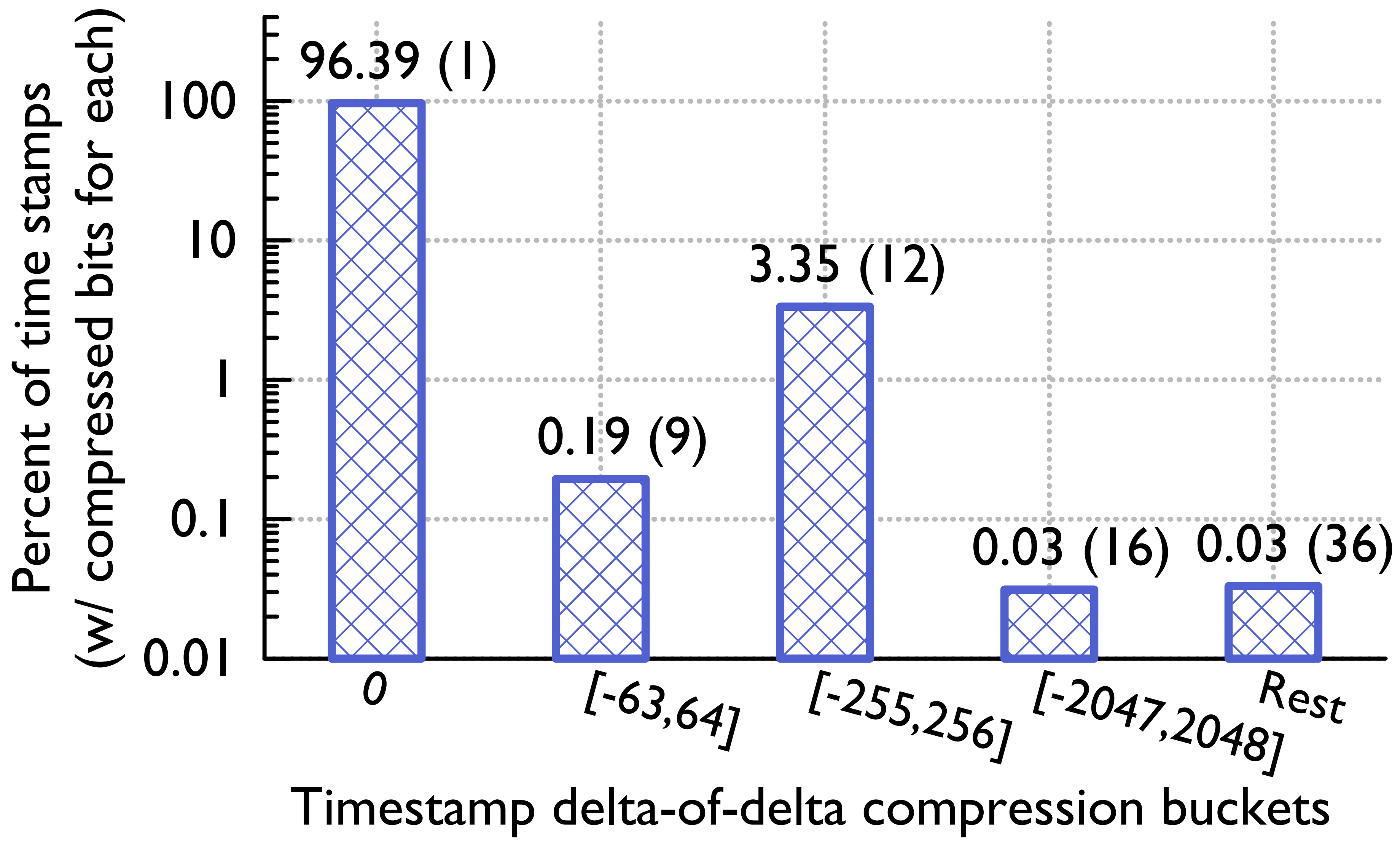
Gorilla Compression



[Pelkonen et al., 2015]

Delta of Delta Compression

- Data usually recorded at regular intervals
- Deltas: 60, 60, 59, 61
- Delta of deltas (D): 0, -1, 2
- Variable-length encoding:
  - D = 0 → 0
  - D in [-63,64] → 10 + value (7 bits)
  - D in [-255,256] → 110 + value (9 bits)
  - D in [-2047,2048] → 1110 + value (12 bits)
  - else → 1111 + value (32 bits)
- 1 bit 96% of the time



[Pelkonen et al., 2015]

XOR Representation

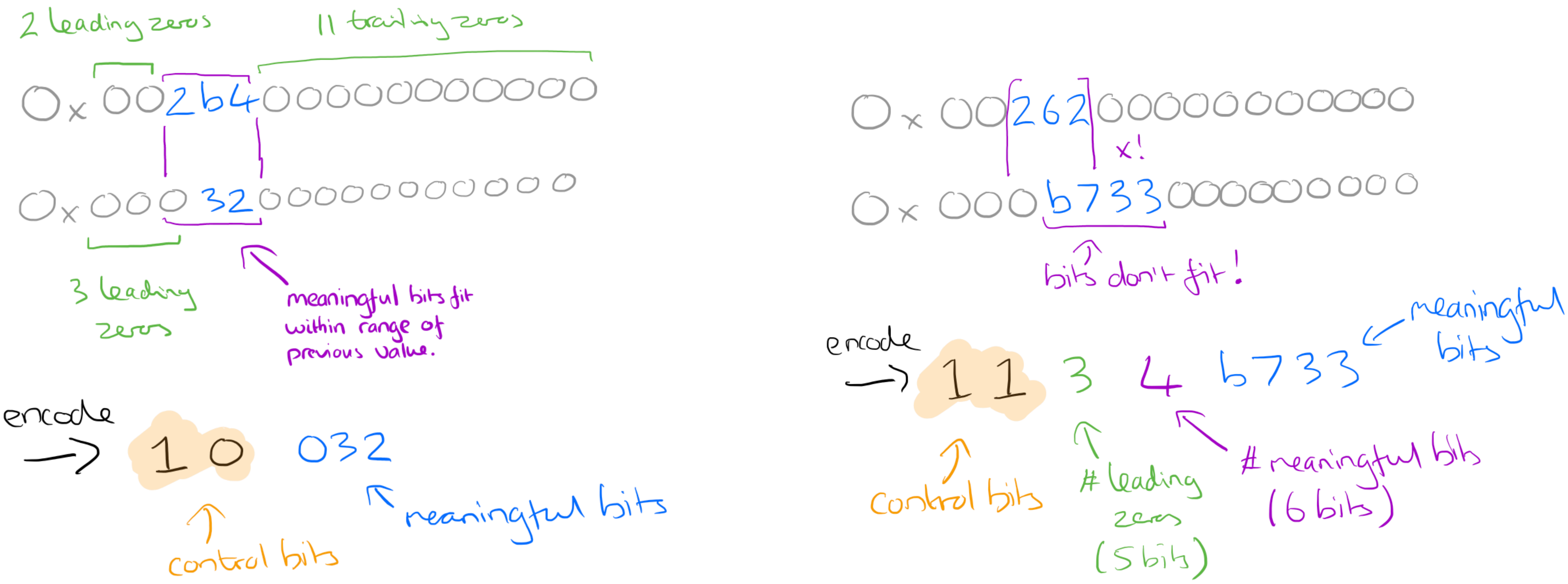
Decimal	Double Representation	XOR with previous
12	0x4028000000000000	
24	0x4038000000000000	0x0010000000000000
15	0x402e000000000000	0x0016000000000000
12	0x4028000000000000	0x0006000000000000
35	0x4041800000000000	0x0069800000000000

Decimal	Double Representation	XOR with previous
15.5	0x402f000000000000	
14.0625	0x402c200000000000	0x0003200000000000
3.25	0x400a000000000000	0x0026200000000000
8.625	0x4021400000000000	0x002b400000000000
13.1	0x402a333333333333	0x000b733333333333

- Values usually do not change significantly
- Look at XOR
  - Same → 0
  - Changes in Meaningful Bits
    - Same as previous value → 10 + changed bits
    - Outside previous value → 11 + leading zeros + length of meaningful bits + bits

[Pelkonen et al., 2015]

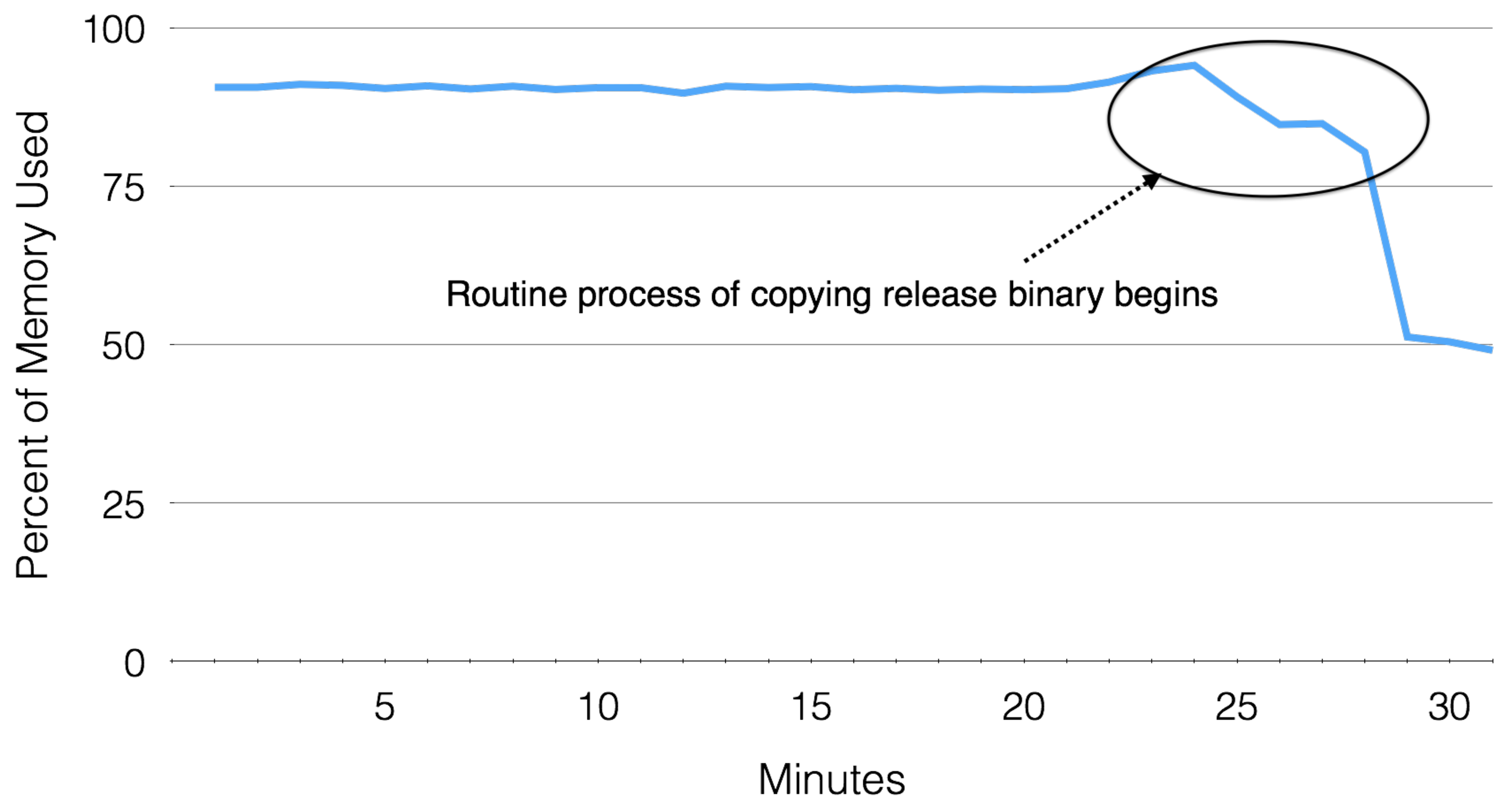
XOR Compression



[A. Colyer]

Enabling Gorilla Features

- Correlation Engine: "What happened around the time my service broke?"
- Charting: Horizon charts to see outliers and anomalies
- Aggregations: Rollups locally in Gorilla every couple of hours



[Pelkonen et al., 2015]

Gorilla Lessons Learned

- Prioritize recent data over historical data
- Read latency matters
- High availability trumps resource efficiency
  - Withstand single-node failures and "disaster events" that affect region
  - "[B]uilding a reliable, fault tolerant system was the most time consuming part of the project"
  - "[K]eep two redundant copies of data in memory"

[Pelkonen et al., 2015]