Advanced Data Management (CSCI 640/490)

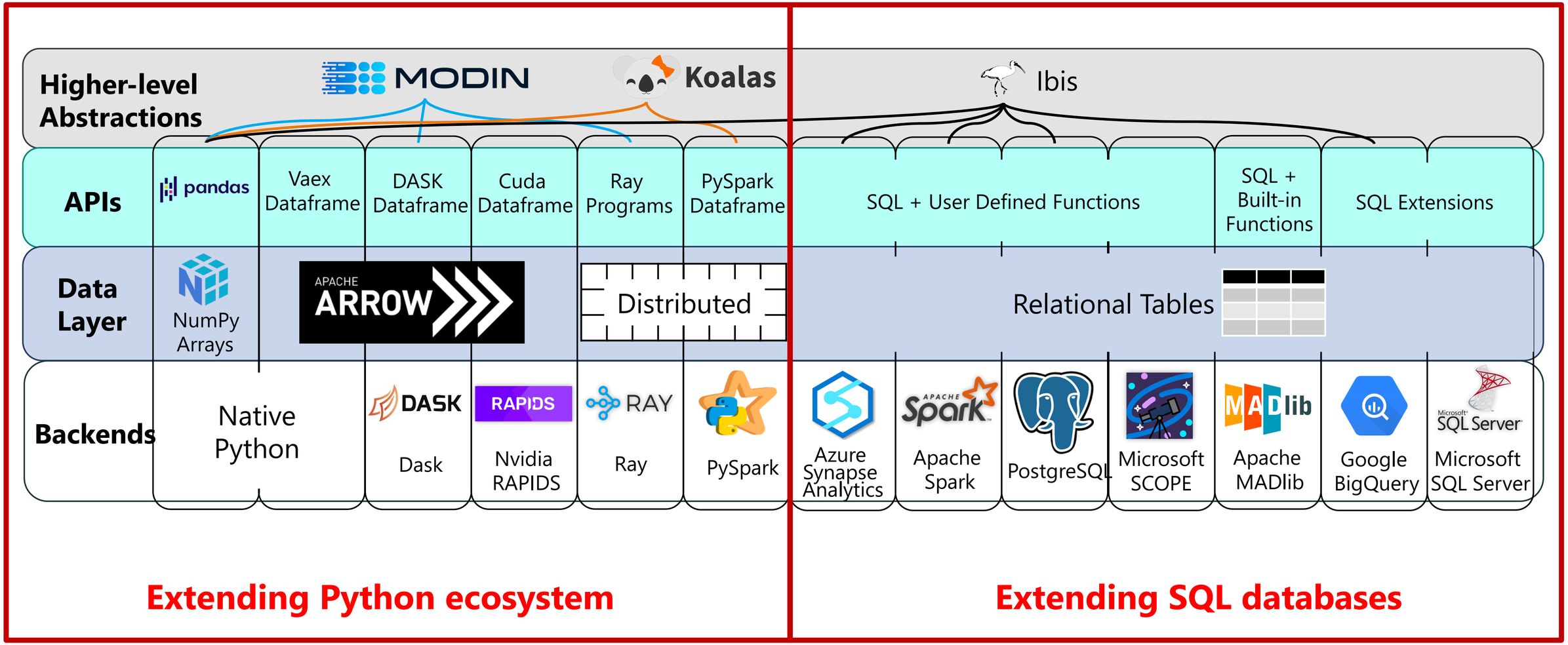
Time Series Data

Dr. David Koop

Dataframes, Databases, and the Cloud

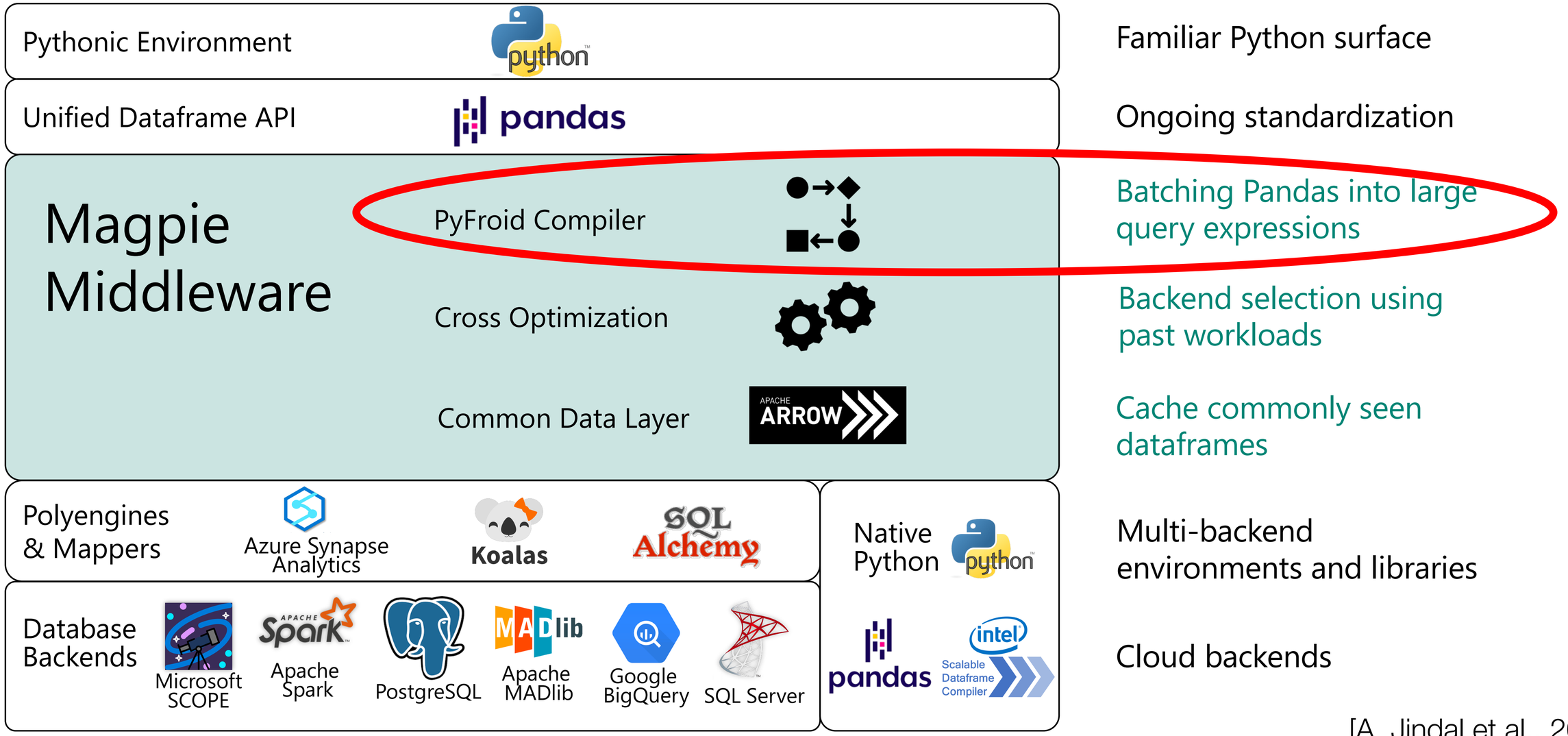
- How do we take advantage of different architectures?
- Lots of work in scaling databases and specialized computational engines
- What is the code that people actually write?

Data Science Jungle



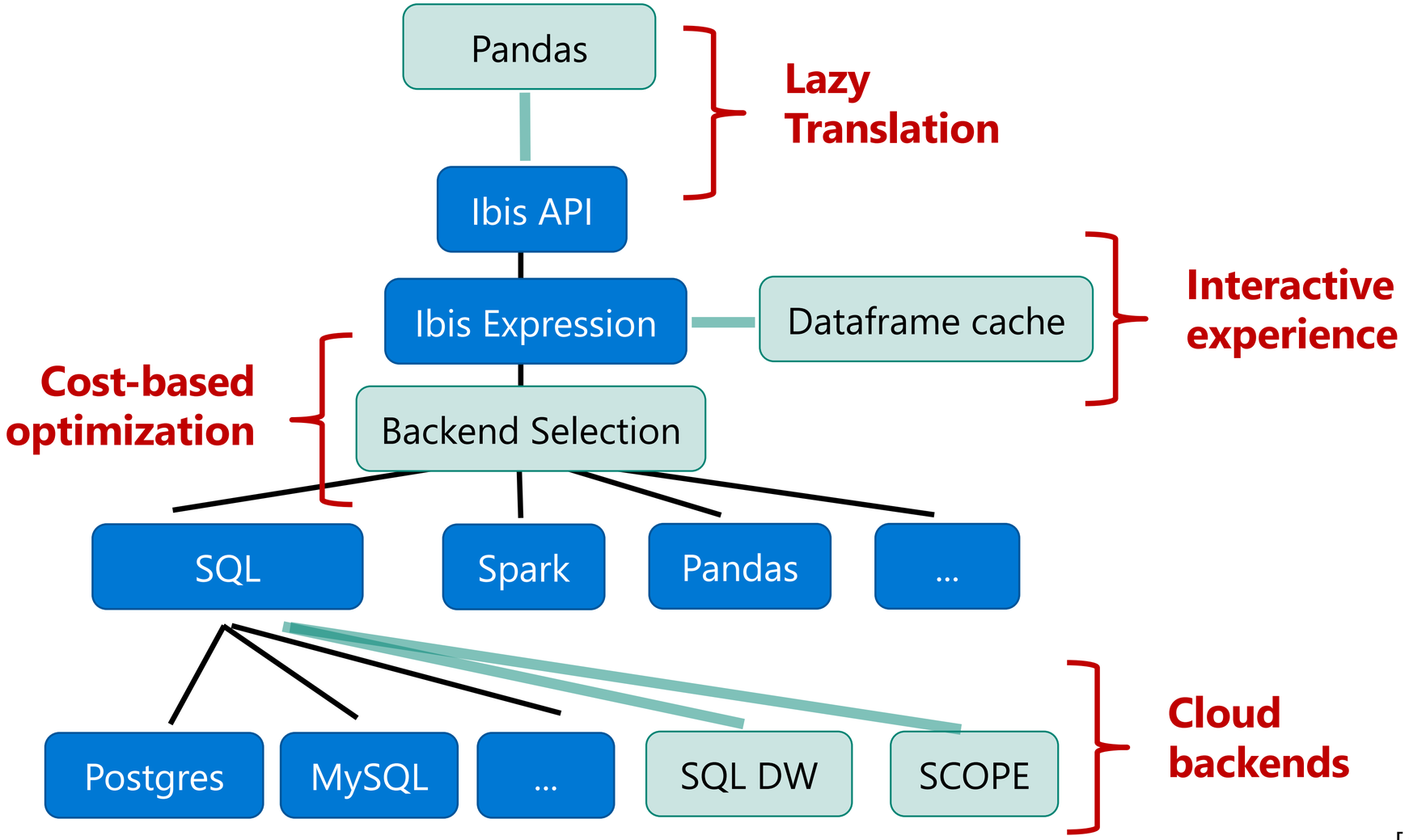
[A. Jindal et al., 2021]

Magpie Goals



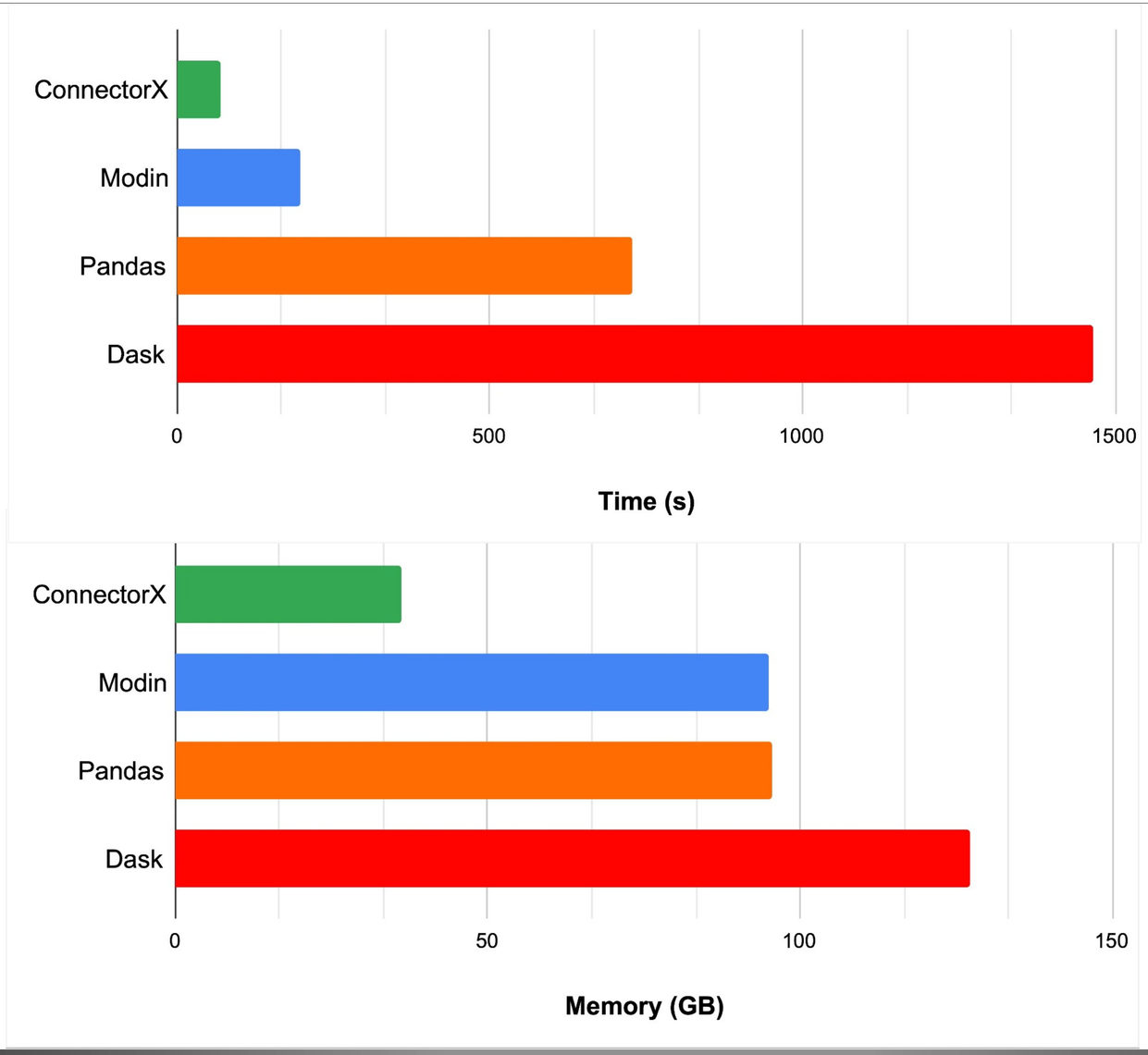
[A. Jindal et al., 2021]

Magpie Architecture



[A. Jindal et al., 2021]

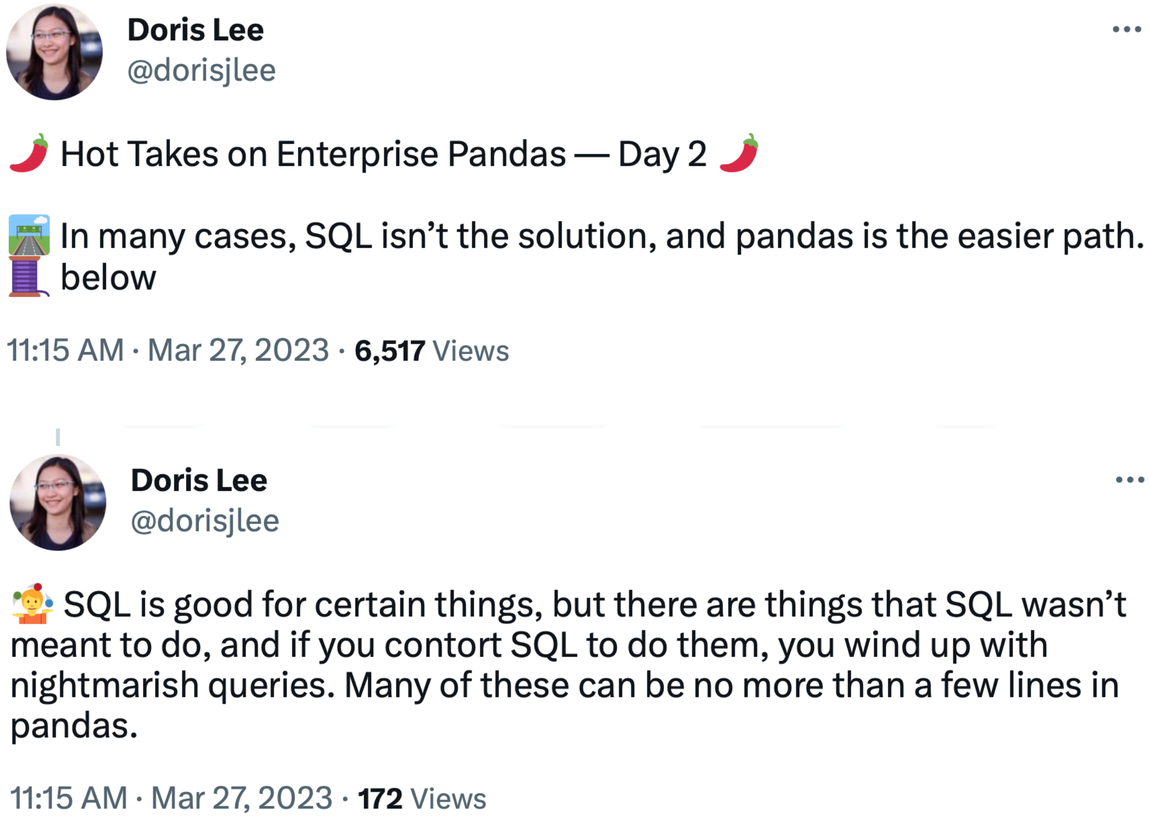
ConnectorX: Databases to Dataframes



[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 university ranking datasets from different institutions
- Integrate information based on names and matching
- Record Matching:
  - Which universities are the same?
- Data Fusion:
  - Names
  - Enrollments
  - Rankings
- Courselet is posted

Courselets

- All should now be available
- You should have received an email with a link to surveys

Test 2

- Next Monday… Nov. 10
- Similar format, but more emphasis on topics we have covered including the research papers

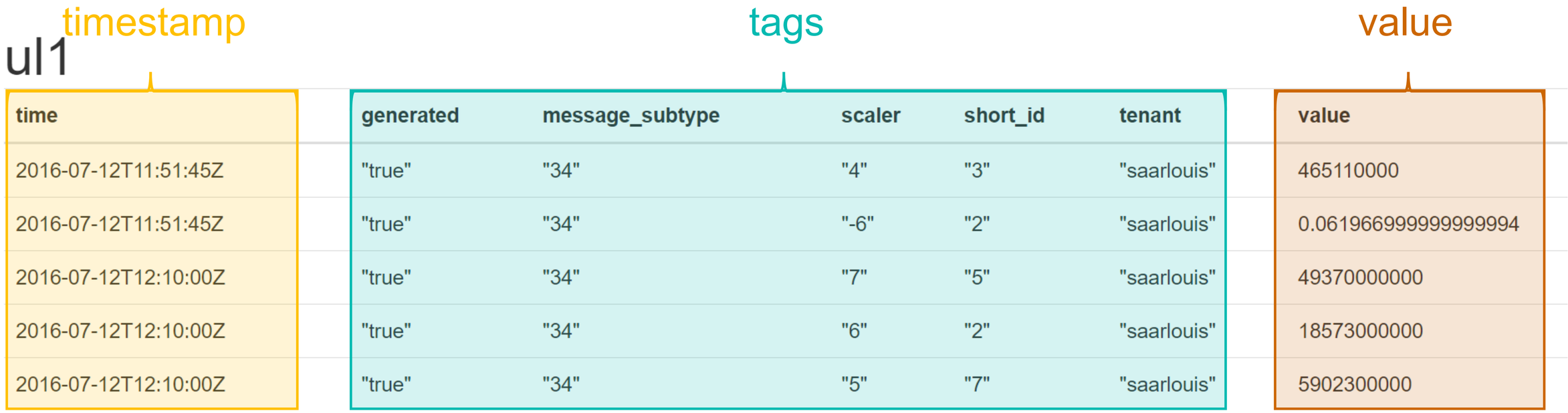
Time Series Data

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

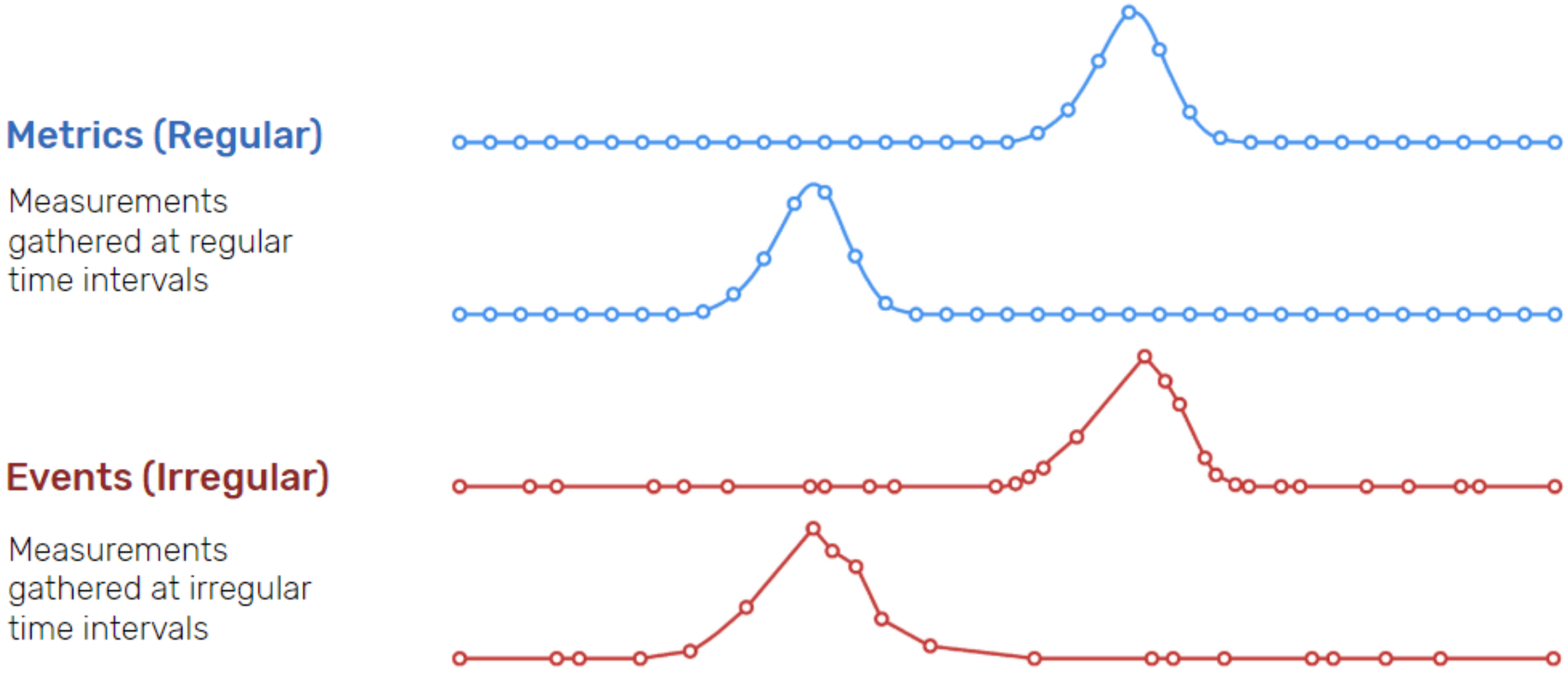


time	generated	message_subtype	scaler	short_id	tenant	value
2016-07-12T11:51:45Z	"true"	"34"	"4"	"3"	"saarlouis"	465110000
2016-07-12T11:51:45Z	"true"	"34"	"-6"	"2"	"saarlouis"	0.061966999999999994
2016-07-12T12:10:00Z	"true"	"34"	"7"	"5"	"saarlouis"	49370000000
2016-07-12T12:10:00Z	"true"	"34"	"6"	"2"	"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

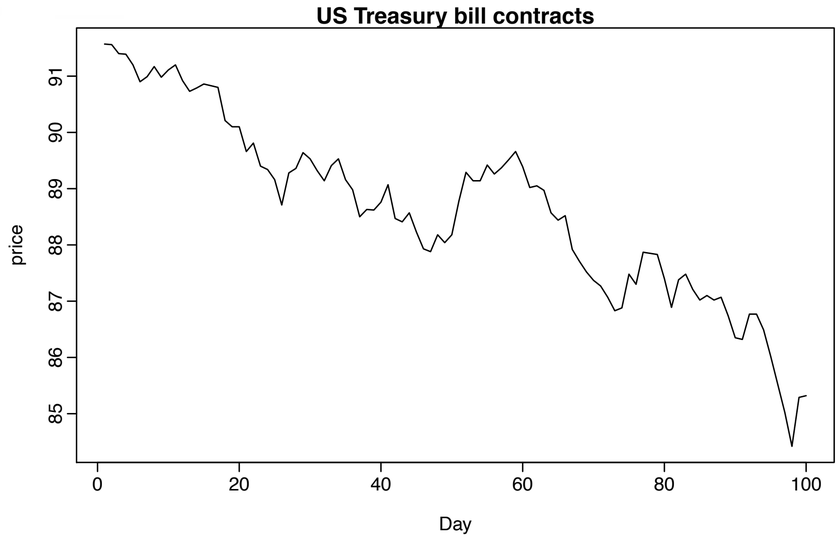
[InfluxDB]

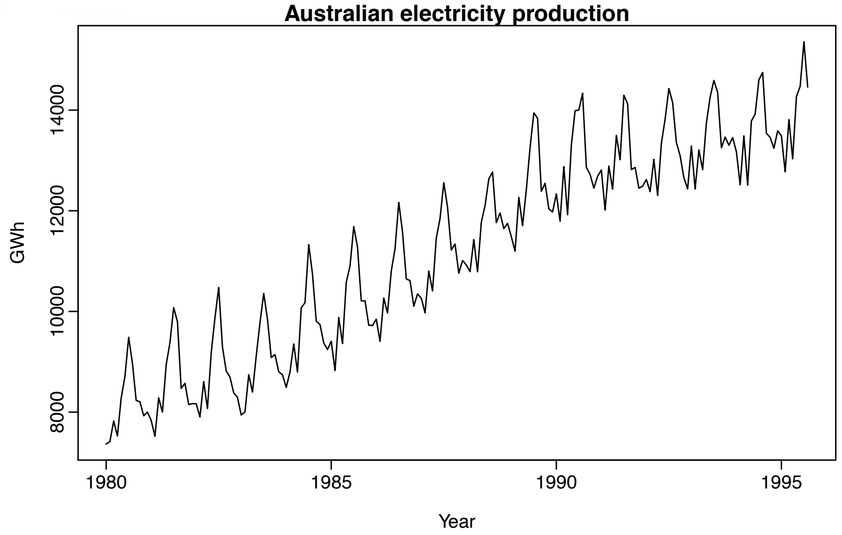
Features of Time Series Data

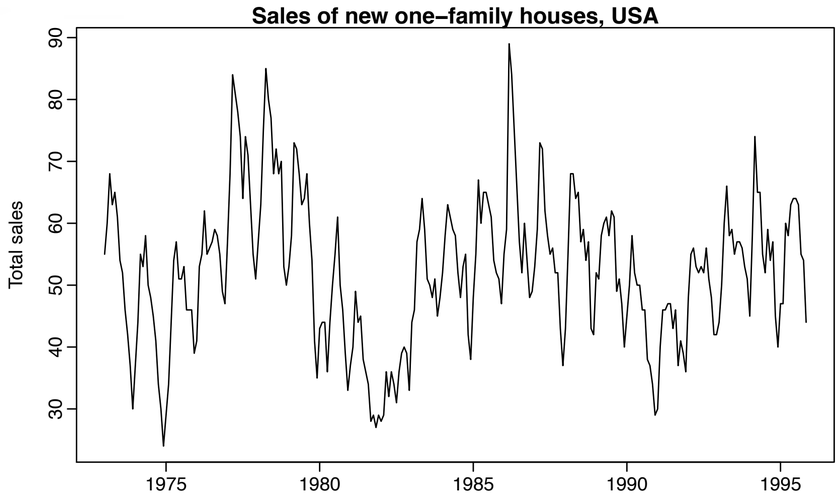
- Trend: long-term increase or decrease in the data
- Seasonal Pattern: time series is affected by seasonal factors such as the time of the year or the day of the week (fixed and of known frequency)
- Cyclic Pattern: rises and falls that are not of a fixed frequency
- Stationary: no predictable patterns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

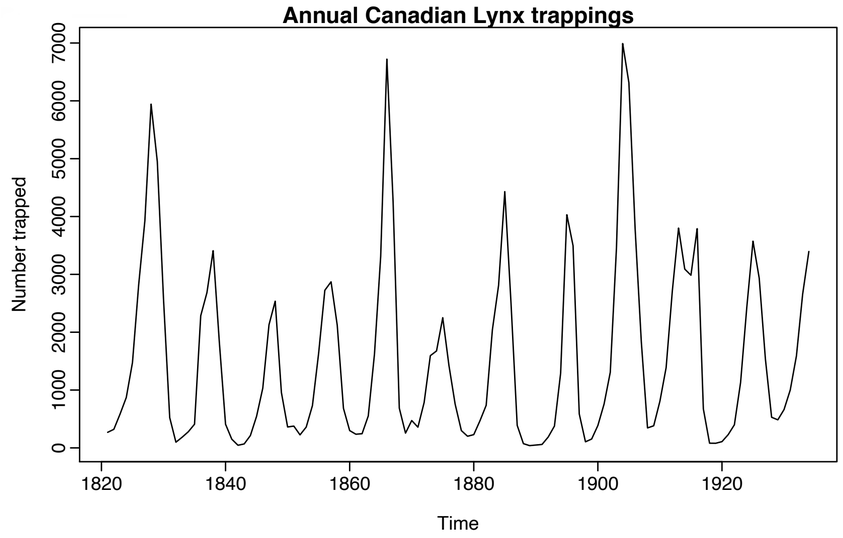
[Hyndman and Athanosopoulos]

Examples





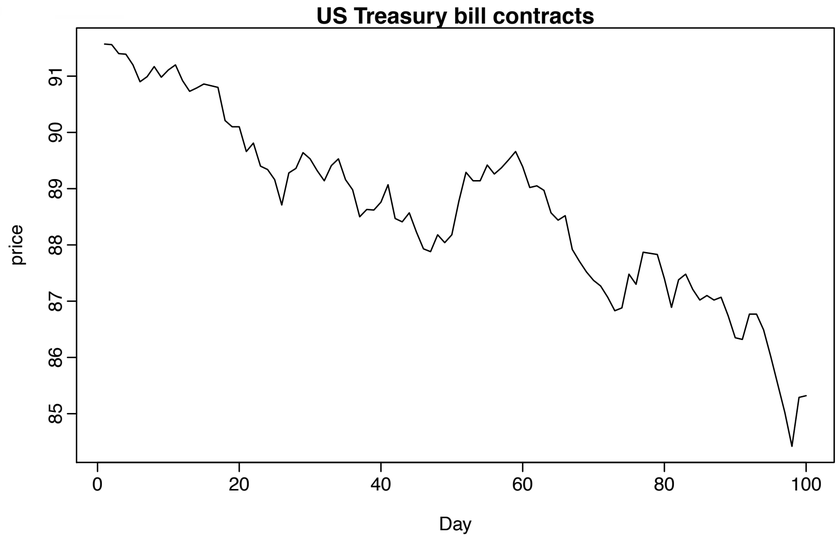


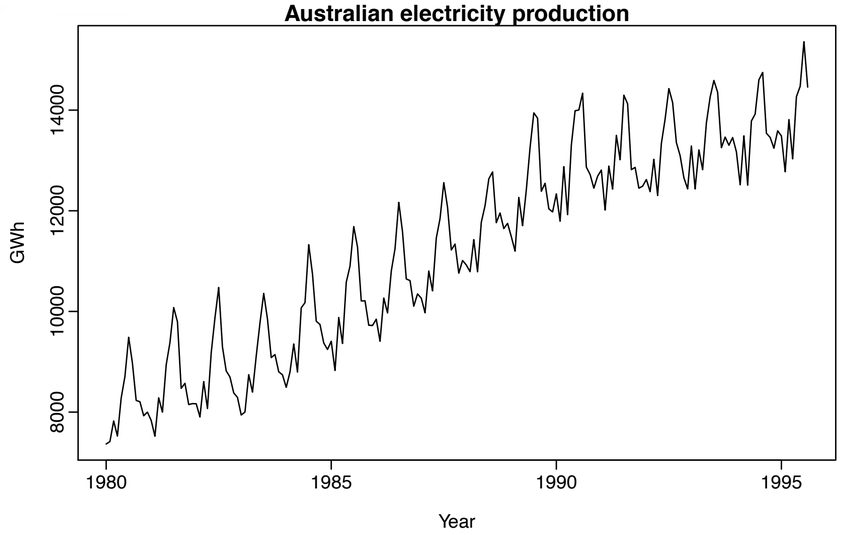


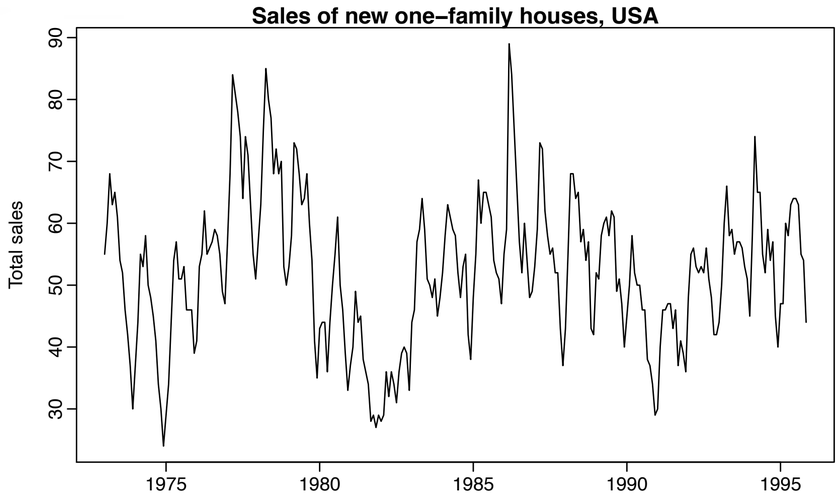
[R. J. Hyndman]

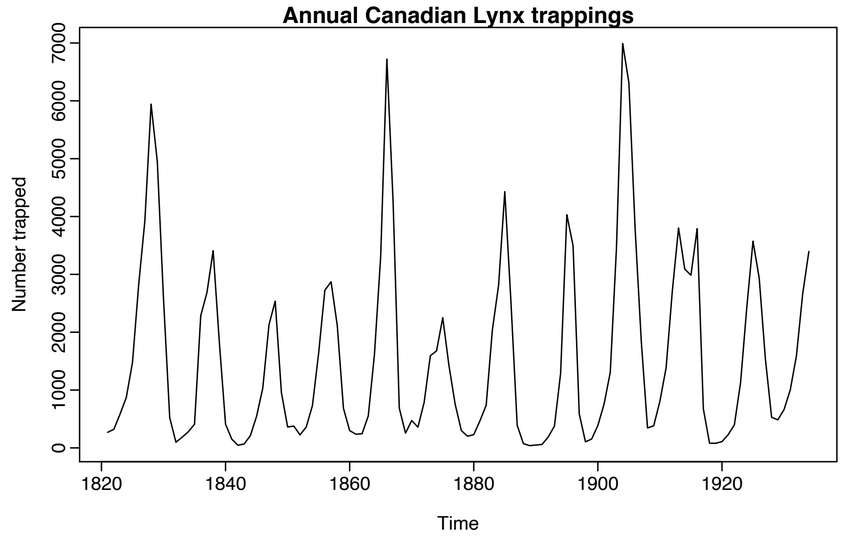
Examples

Trend





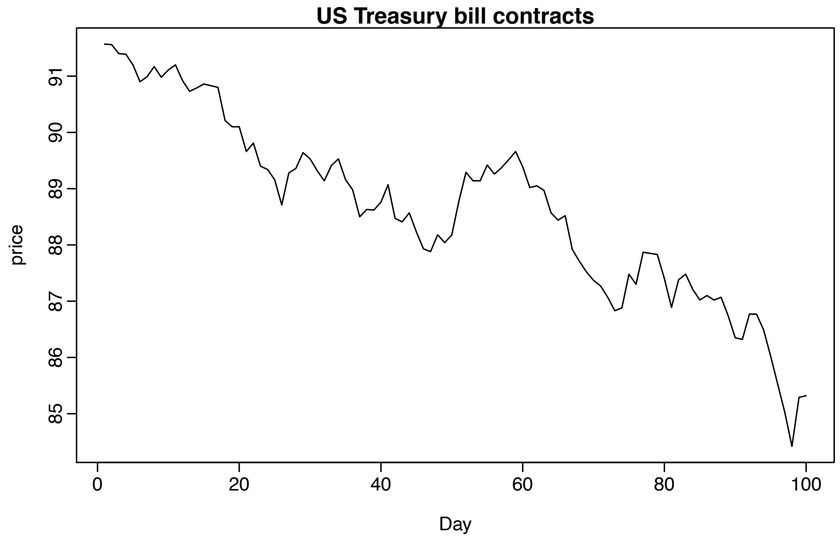


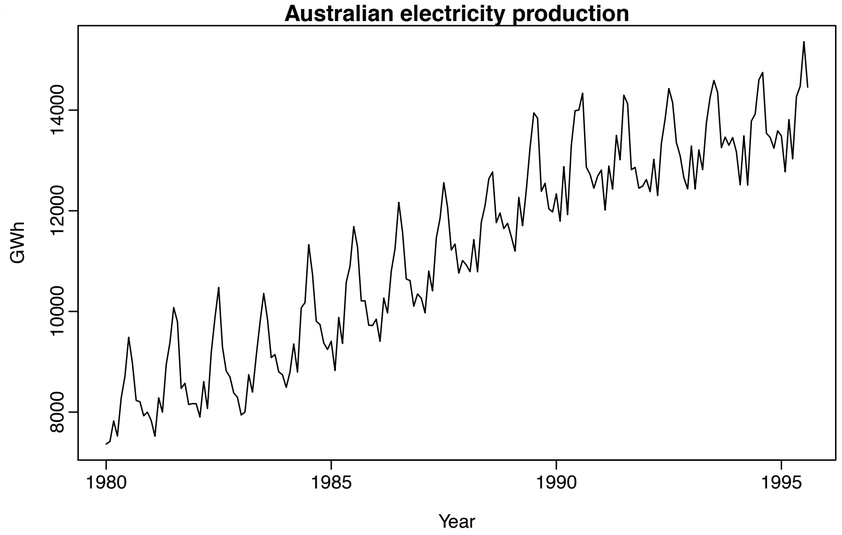


[R. J. Hyndman]

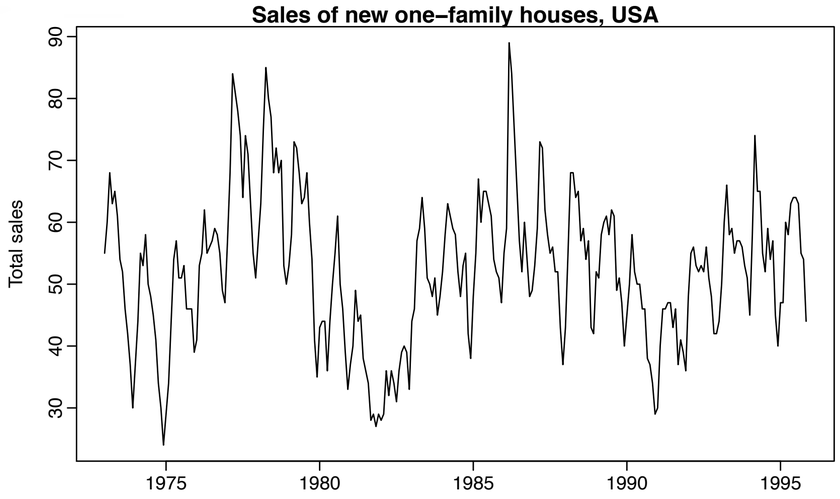
Examples

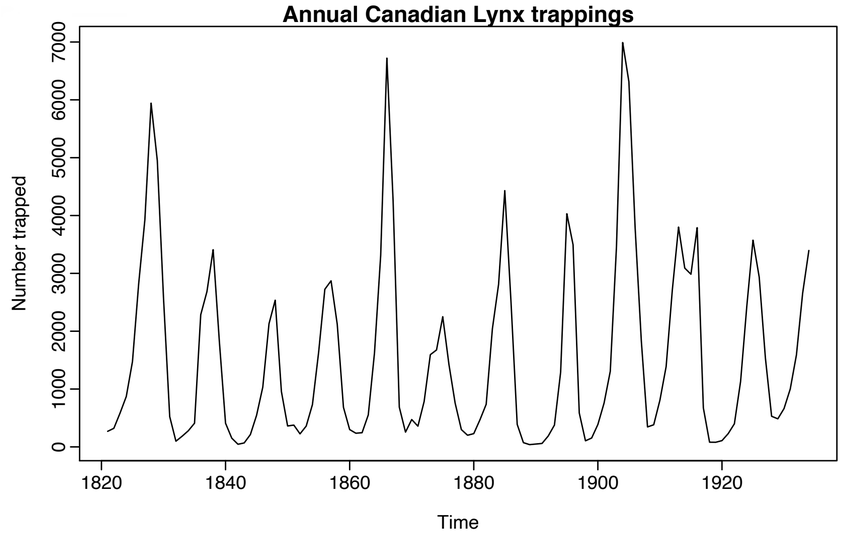
Trend





Trend + Seasonality

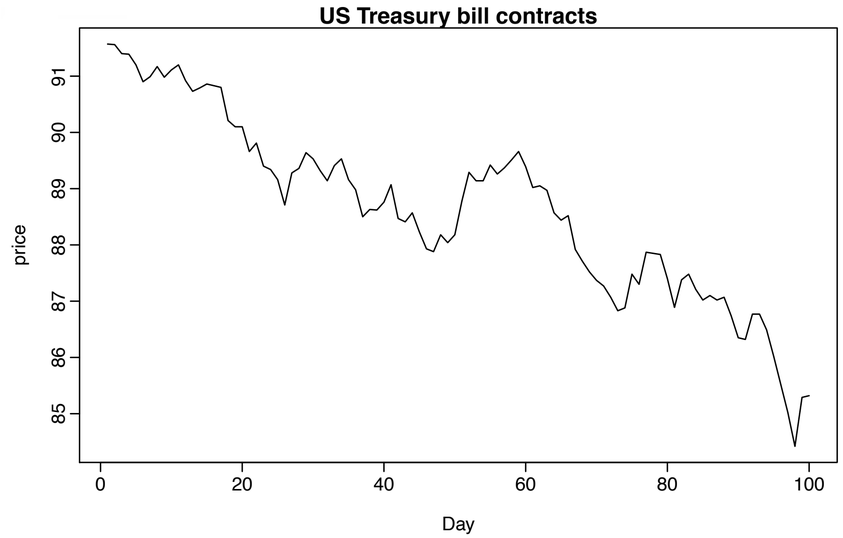


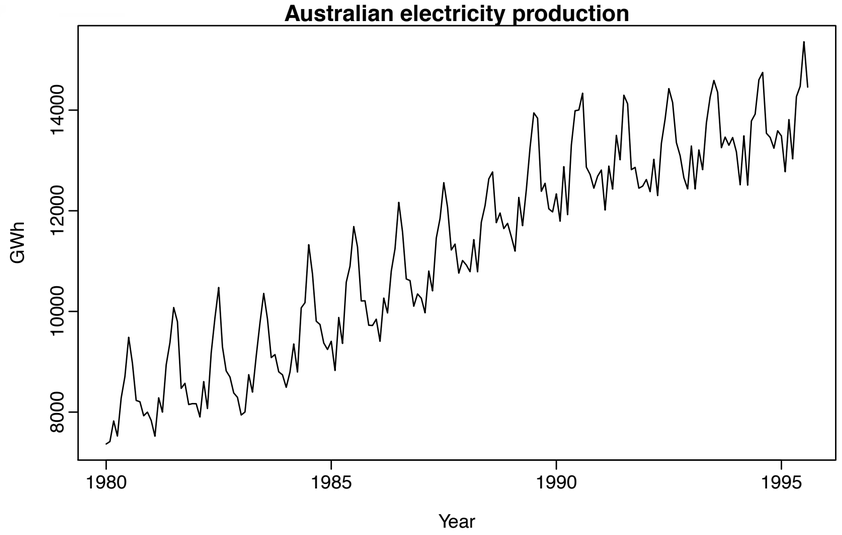


[R. J. Hyndman]

Examples

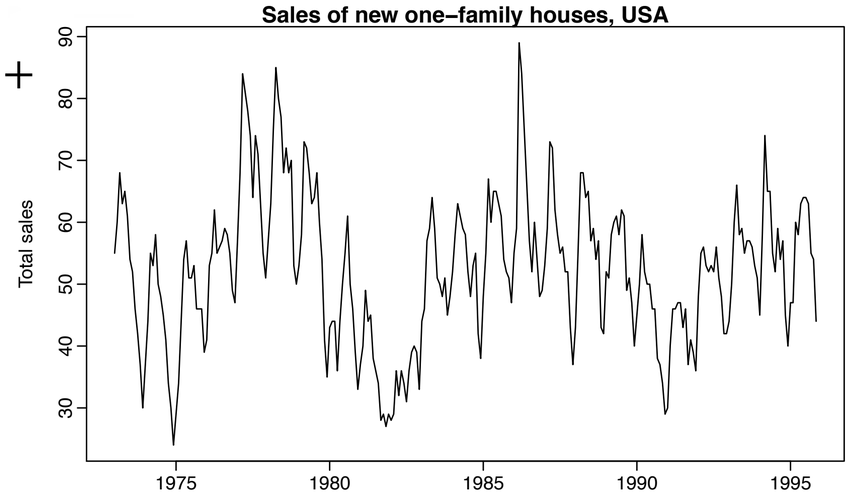
Trend

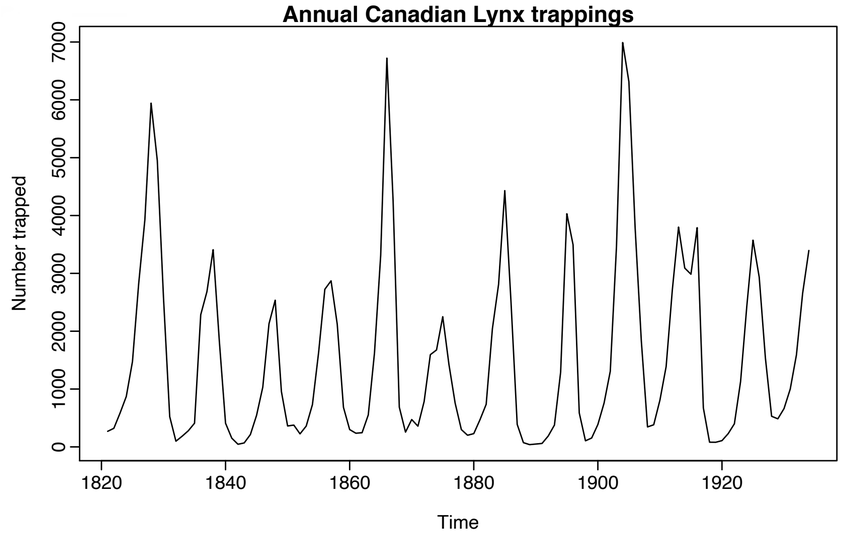




Trend + Seasonality

Seasonality + Cyclic

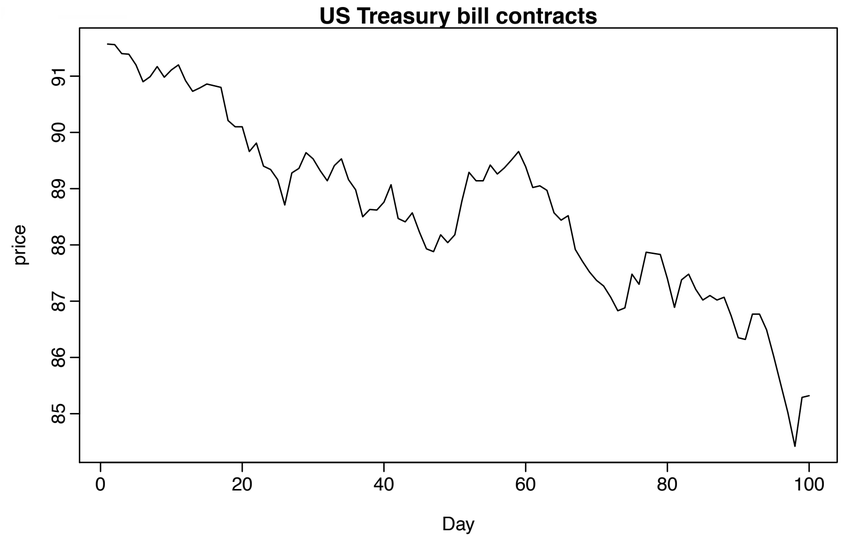


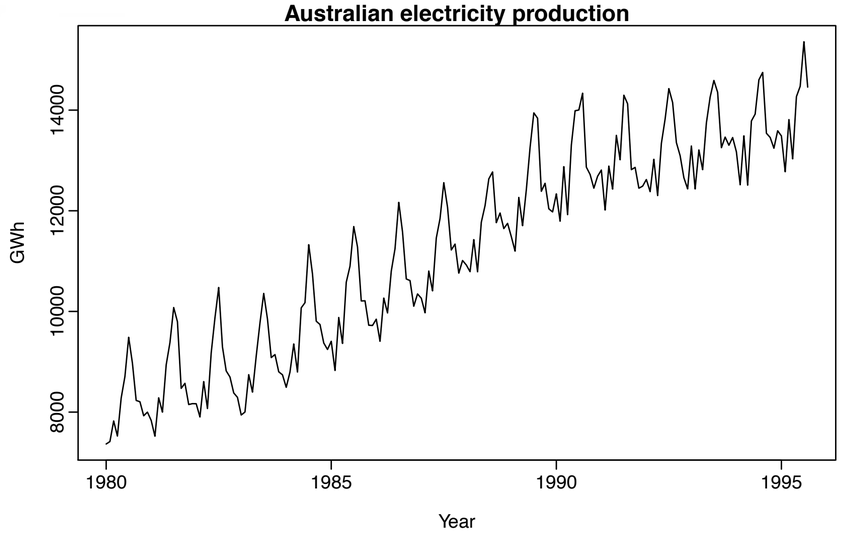


[R. J. Hyndman]

Examples

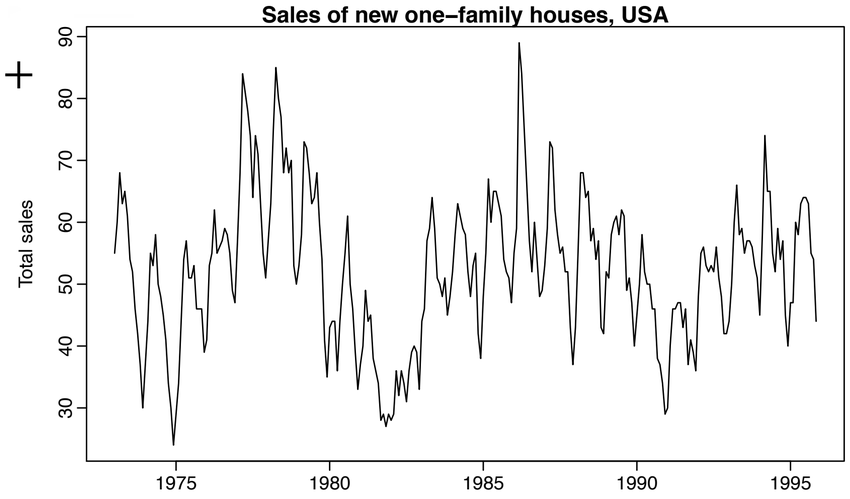
Trend

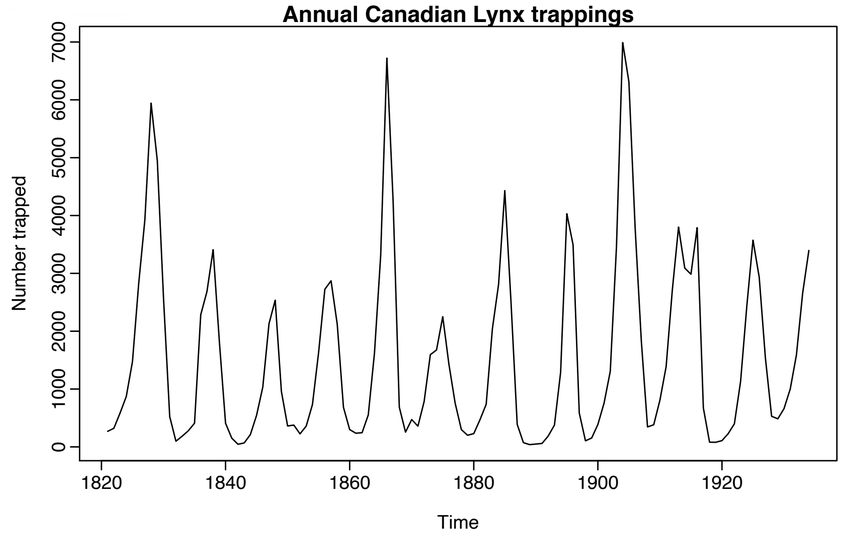




Trend + Seasonality

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

Polars Support for Datetime

- Has separate types for date (`pl.Date`), time (`pl.Time`), and datetime (`pl.Datetime`)
- `pl.date, pl.time, pl.datetime`: convenience method to create objects
- Can convert from a string to a datetime using `.str.to_datetime()`
  - If no format is specified, **infers** the format from the data
  - Can specify the format, but uses **rust** specification (docs)
- Stores as a 64-bit integer representing the number of time units since the UNIX epoch (1970-01-01 00:00:00)
  - Time units can be milliseconds (`ms`), microseconds (`us`), or nanoseconds (`ns`)
  - Defaults to microseconds (`us`)

More Polars Support

- `is_between` as a shortcut for less-than and greater-than clauses
- Can manipulate datetime objects using arithmetic operations:
  - `pl.col('dt') + pl.duration(hours=1) # produces datetime`
  - `pl.col('dt1') - pl.col('dt2') # produces duration`
- As with strings, to treat a column as datetime, you can use the `.dt` accessor
  - `pl.col('dt').dt.year(), pl.col('dt').dt.month()`
- Can also convert back to string
  - `pl.col('dt').dt.to_string(<format>)`
  - `<format>` defaults to `'iso'`

Generating Ranges

- Specify start and end and the interval
- Works for both dates and datetimes
- `pl.date_range(pl.date(2022,1,1), pl.date(2022,3,1), "1mo")`
- `pl.datetime_range(dt1, dt2, dt.timedelta(days=1, hours=12))`
- Also `date_ranges` and `datetime_ranges` to create columns of intervals
  - `pl.datetime_ranges(pl.col('dt1'), pl.col('dt2'), interval="3m")`

Interval String Language

- 1ns (1 nanosecond)
- 1us (1 microsecond)
- 1ms (1 millisecond)
- 1s (1 second)
- 1m (1 minute)
- 1h (1 hour)
- 1d (1 calendar day)
- 1w (1 calendar week)
- 1mo (1 calendar month)
- 1q (1 calendar quarter)
- 1y (1 calendar year)

[Polars documentation]

Shifting vs. Offsetting Data

- Can do this in polars via shift method (shifts column up or down)
  - `pl.col('ts').shift(2),pl.col('ts').shift(-2)`
  - This is done by **index**
- Can also offset data by adding a timedelta, but…
  - `pl.col('ts') + dt.timedelta(days=31)`
- …need calendar-aware offseting
  - `pl.col('ts').offset_by('1mo')`
- Calendar-aware units: `d, w, mo, q, y`
- Also can roll dates to month-end
  - `pl.col('ts').dt.month_end()`

Datetime Periods

- No explicit support in polars, but can use `offset_by` or `month_end`
-
```
pl.datetime_ranges(pl.col('time'),
                   pl.col('time').dt.month_end())
```

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, Polars, and Time Zones

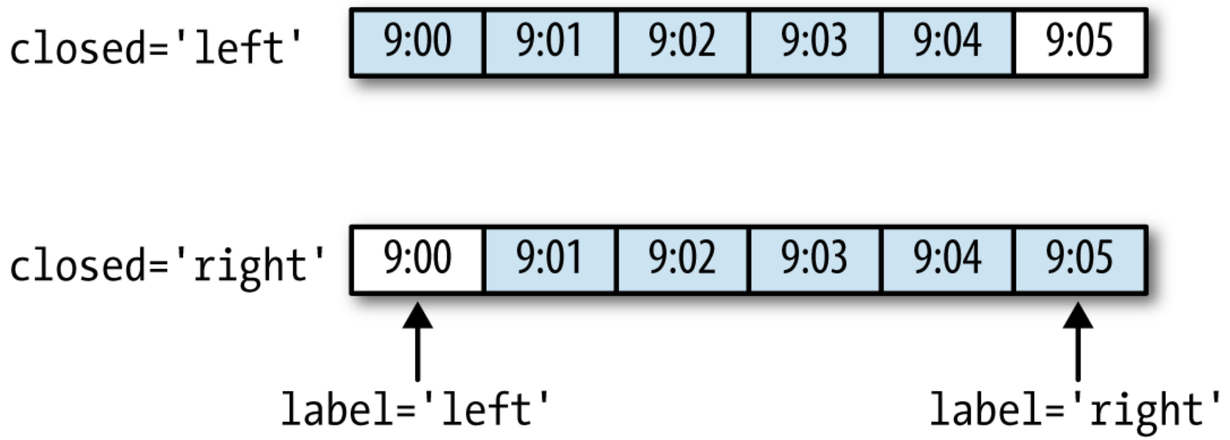
- Time series in pandas are **time zone native**
- The pytz module keeps track of all of the time zone parameters
  - even Daylight Saving Time
- Localize a timestamp using `replace_time_zone`
  - ```
    ts = pl.datetime("1 Dec 2016 12:30 PM")
    ts = ts.dt.replace_time_zone("US/Eastern")
    ```
- Convert a timestamp using `convert_time_zone`
  - ```
    ts.dt.convert_time_zone("Europe/Budapest")
    ```
- Convert time zones to ensure operations are calculated correctly.

Resampling

- Two directions:
  - Downsample: higher frequency to lower frequency
  - Upsample: lower frequency to higher frequency
  - The index or time_column column must be in **sorted** order!
- Downsample is a special case of `group_by` in polars (`group_by_dyanmic`)
  - ```
    (df.group_by_dynamic("date", every="1y")
       .agg(pl.col("close").mean()))
    ```
- Polars has a dedicated `upsample` method:
  - ```
    (df.upsample(time_column="time", every="15m")
       .fill_null(strategy="forward"))
    ```
- String language for the `every` argument

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, or both)



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()

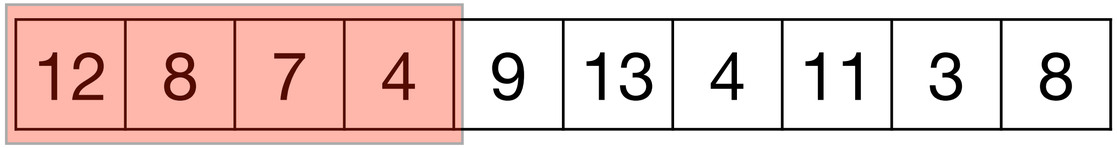
In [224]: df_daily
Out[224]:
            Colorado     Texas  New York      Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-06       NaN       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN       NaN
2000-01-08       NaN       NaN       NaN       NaN
2000-01-09      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

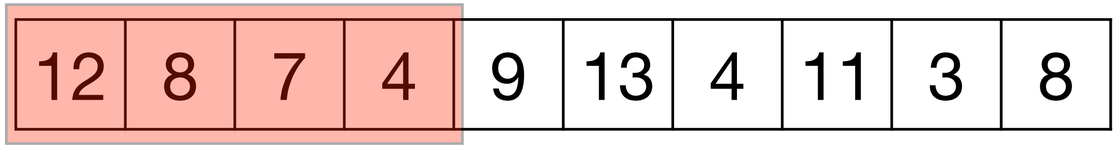
Interpolation

- Fill in the missing values with computed best estimates using various types of algorithms
- Apply after resample
- ```
  df.upsample("time", every="15m").with_columns(
      pl.col("groups").fill_null(strategy="forward"),
      pl.col("values").interpolate()
  )
  ```
  ```

Rolling Window Calculations

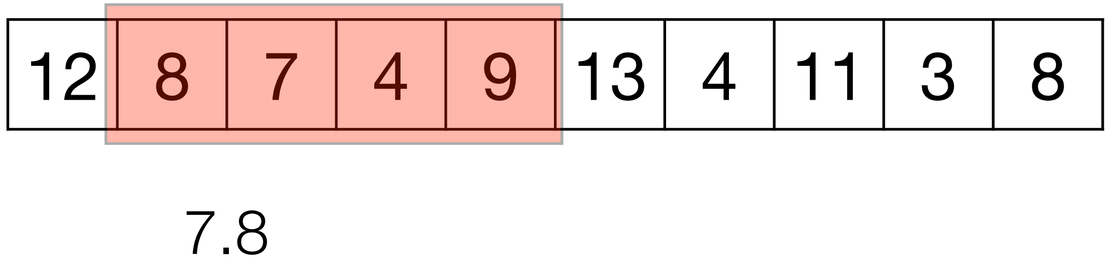


Rolling Window Calculations

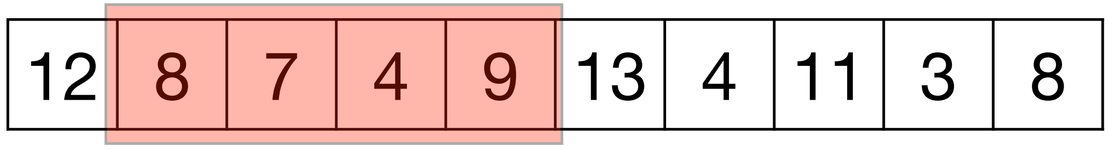


7.8

Rolling Window Calculations

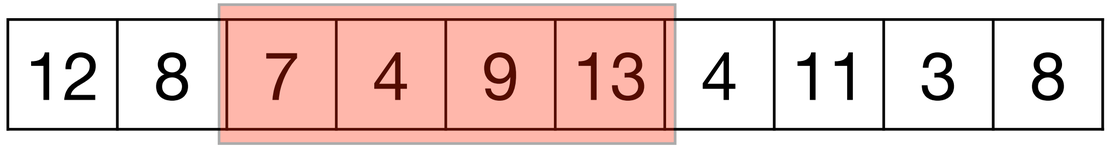


Rolling Window Calculations



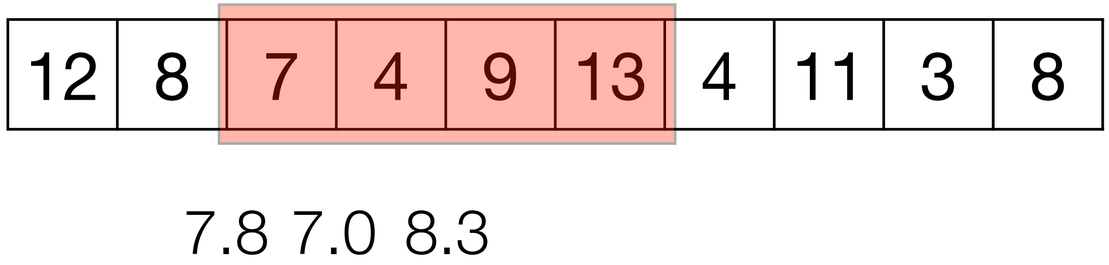
7.8 7.0

Rolling Window Calculations



7.8 7.0

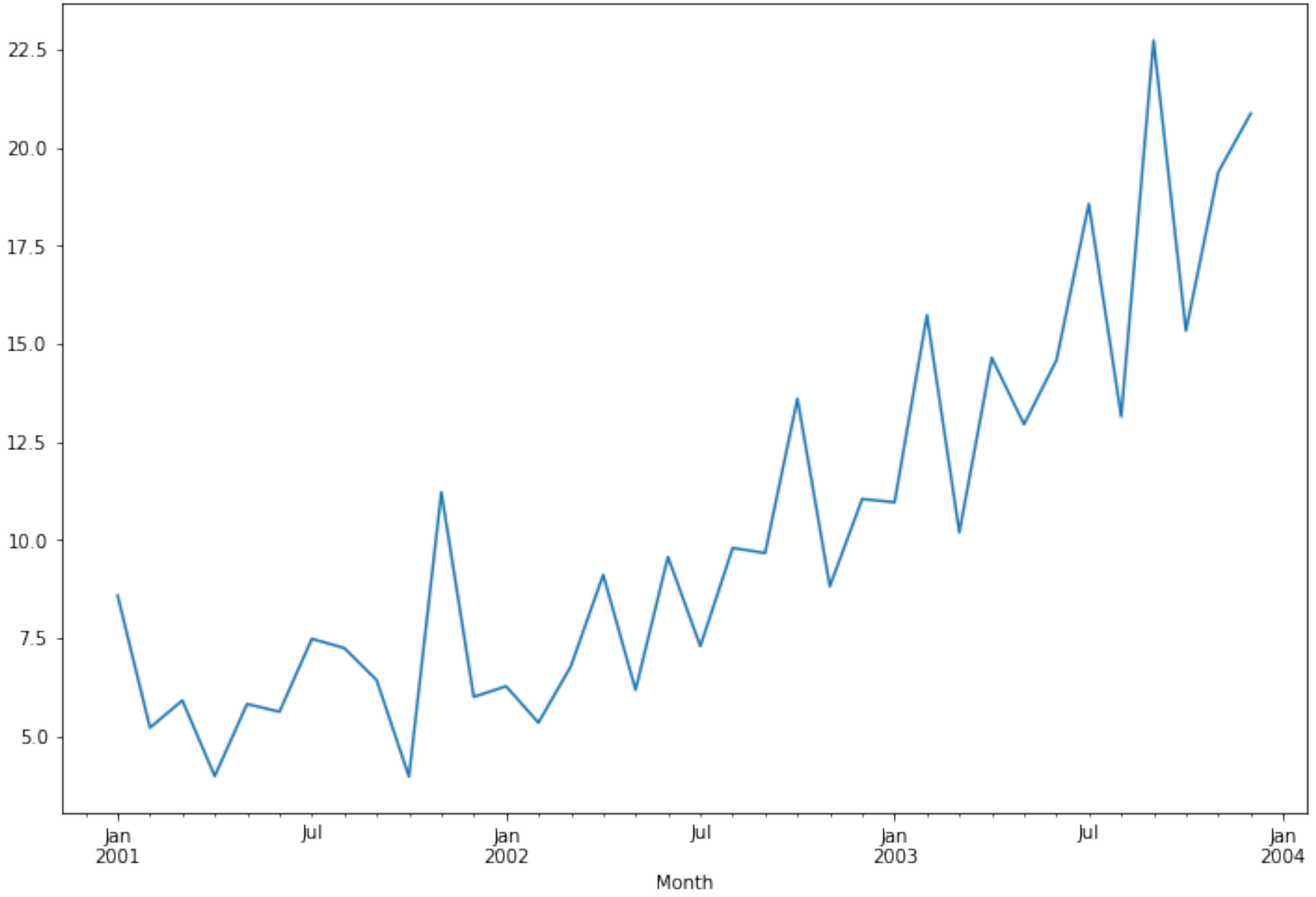
Rolling Window Calculations



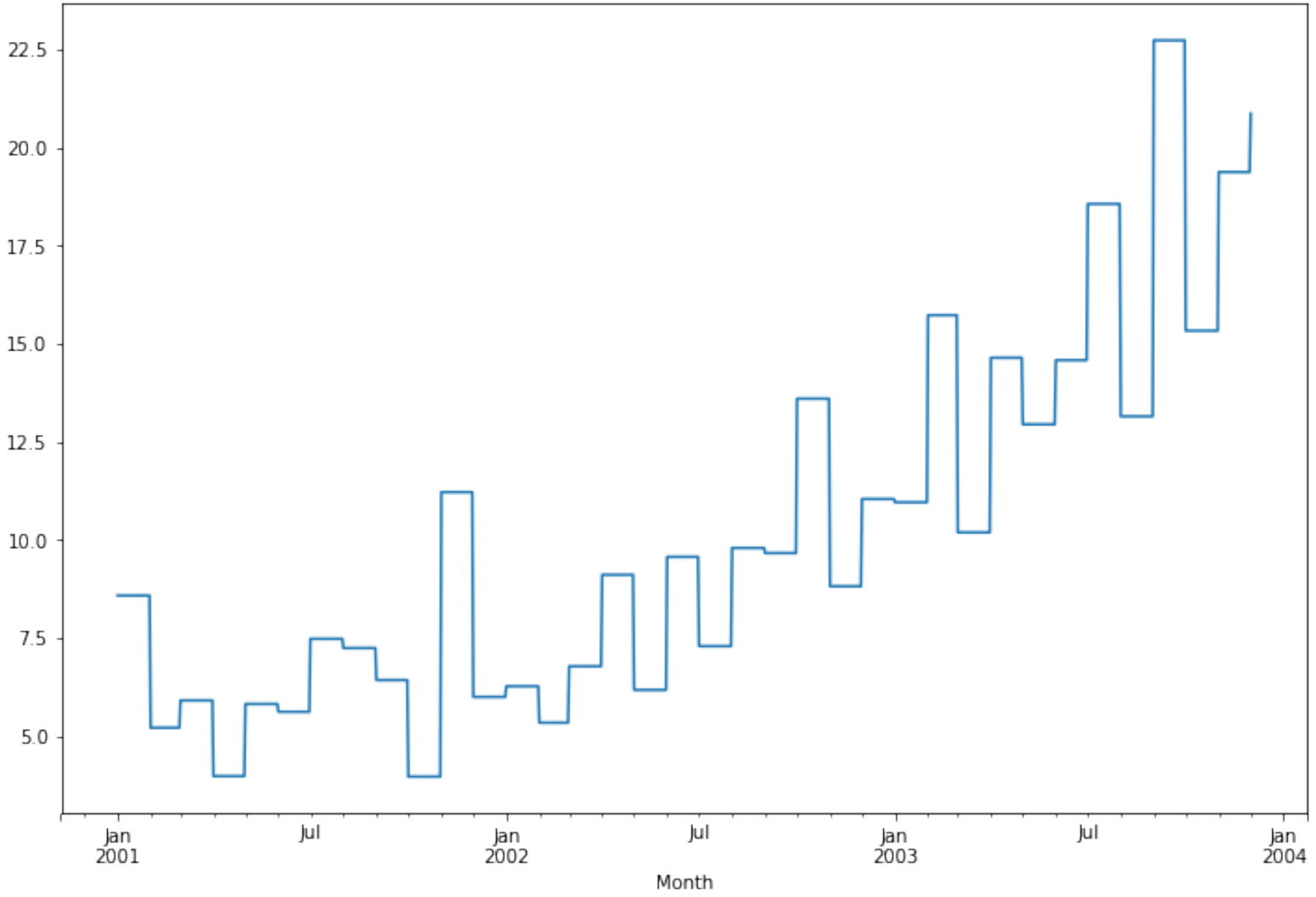
Window Functions

- 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 left edge of window (change with `offset`)
- Example:
  - `df.rolling('time', '1d').agg(pl.min('value'))`
  - `df.rolling('time', '2d').agg(pl.sum('value'))`

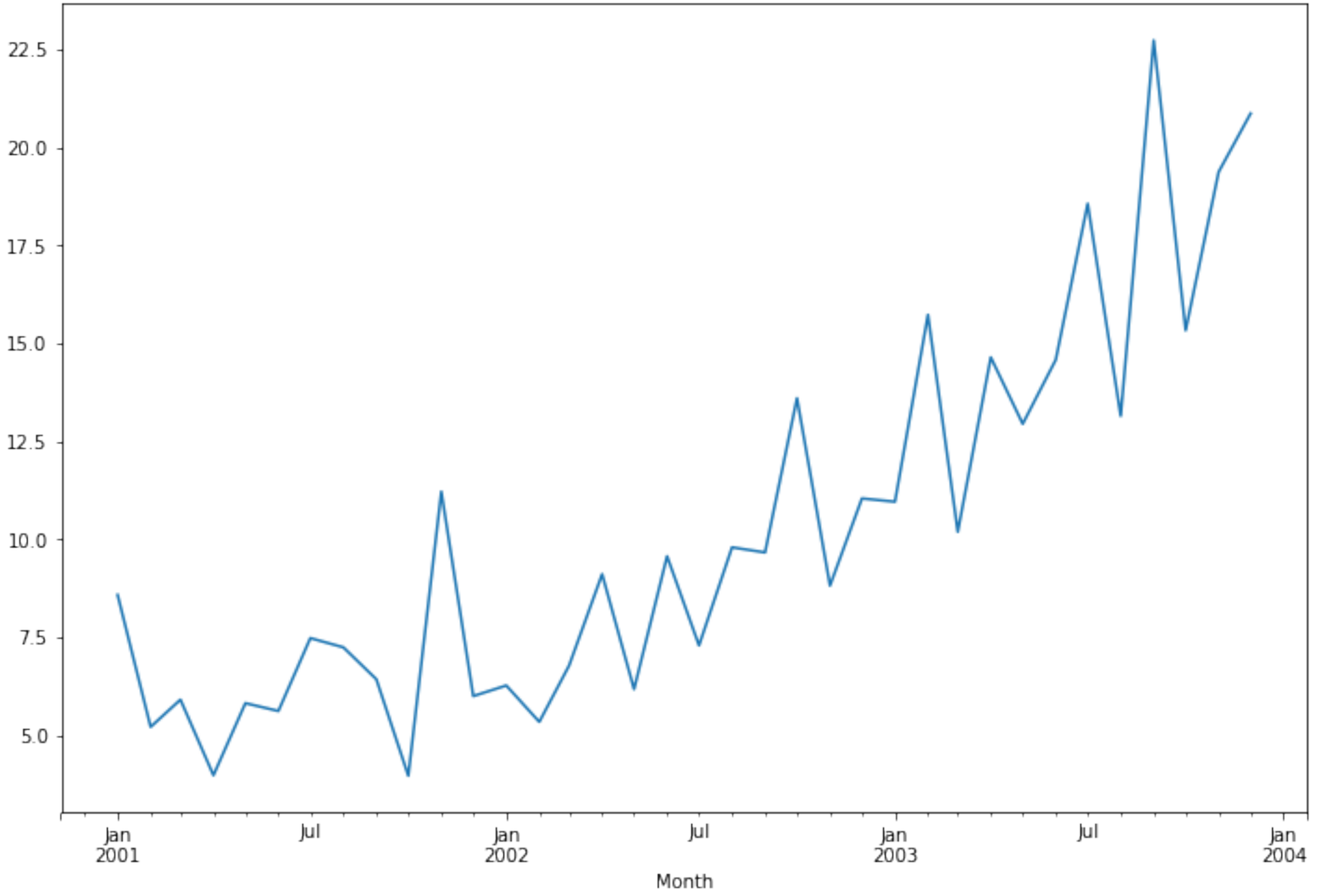
Sales Data by Month



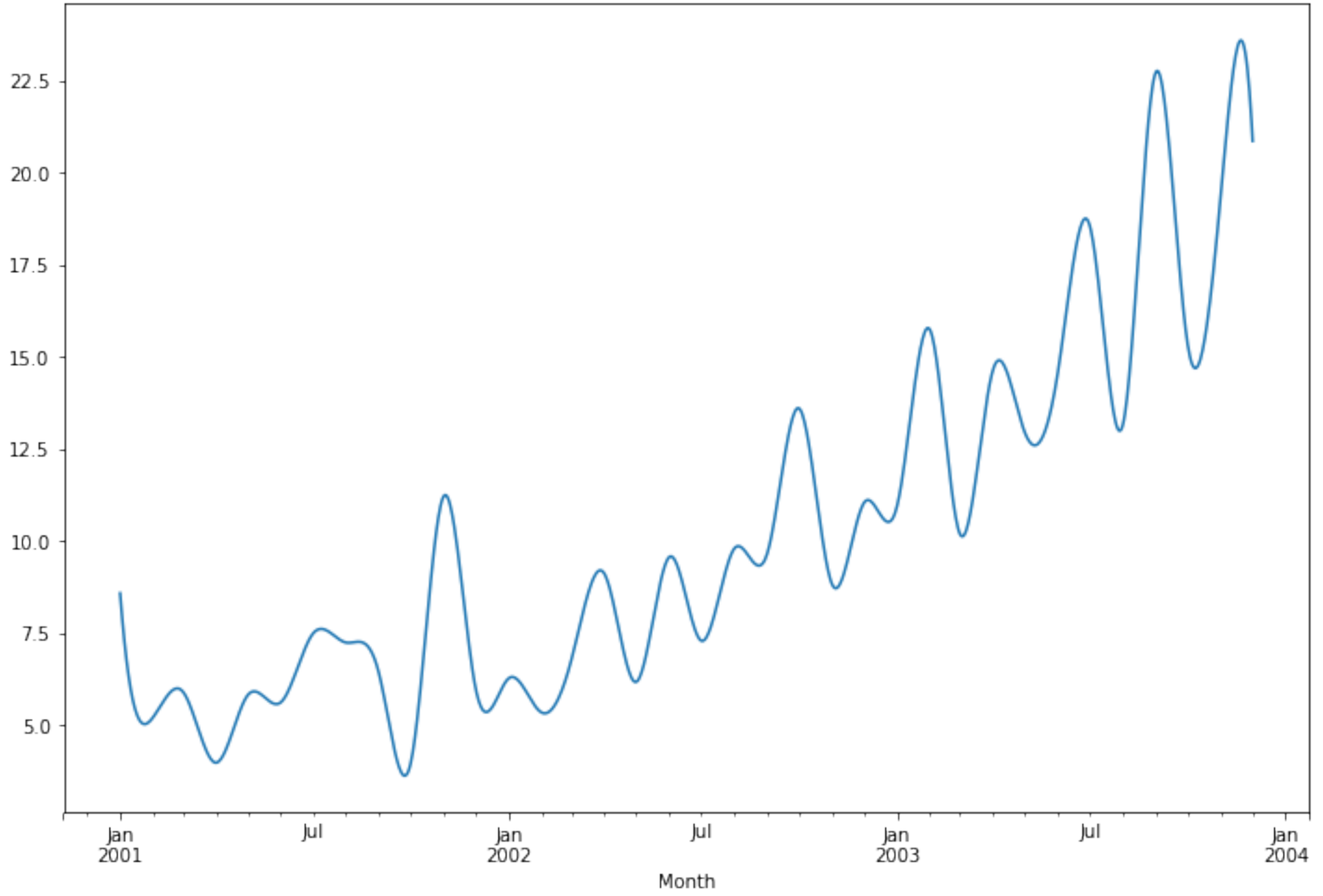
Resampled Sales Data (forward fill)



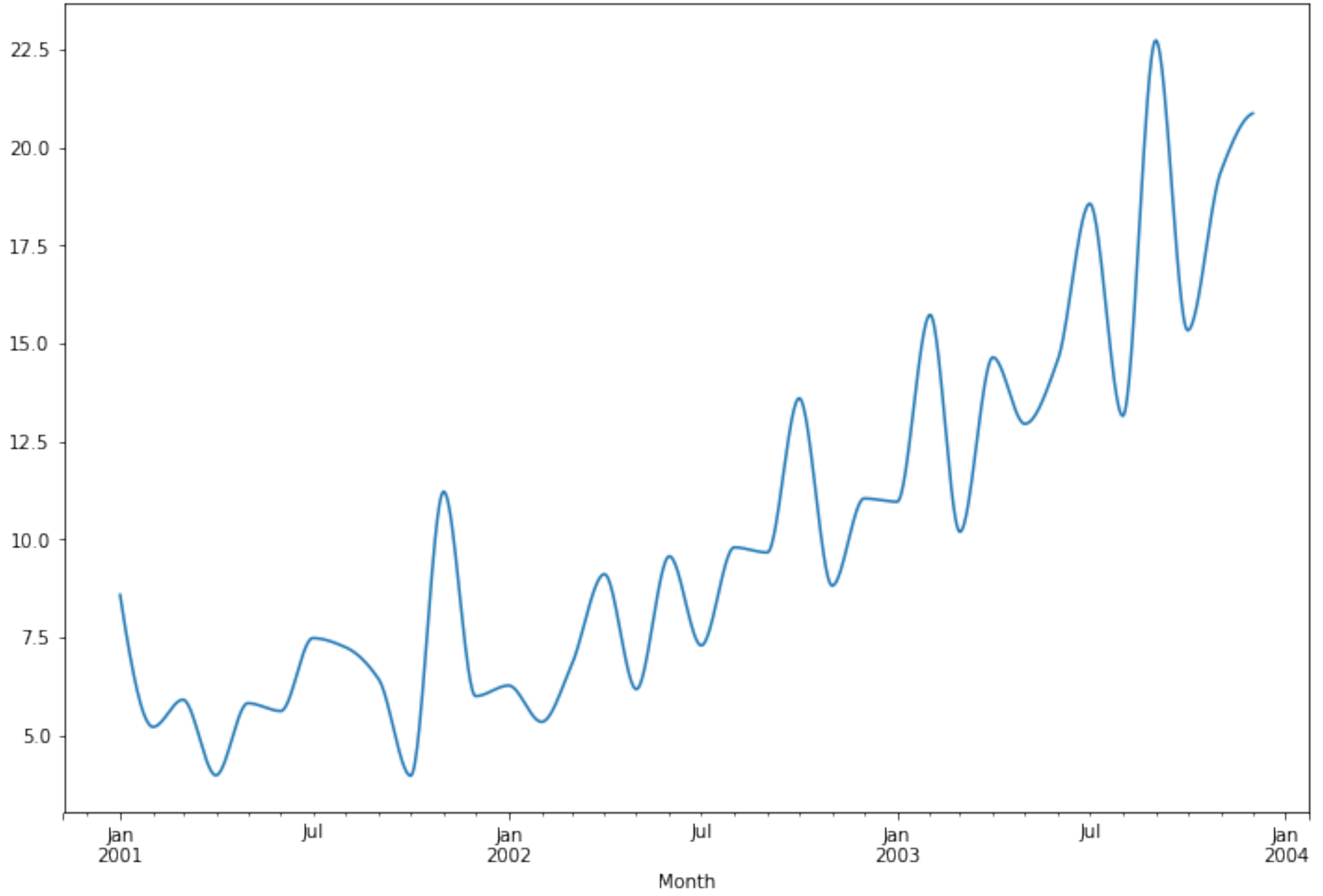
Resampled with Linear Interpolation (default)



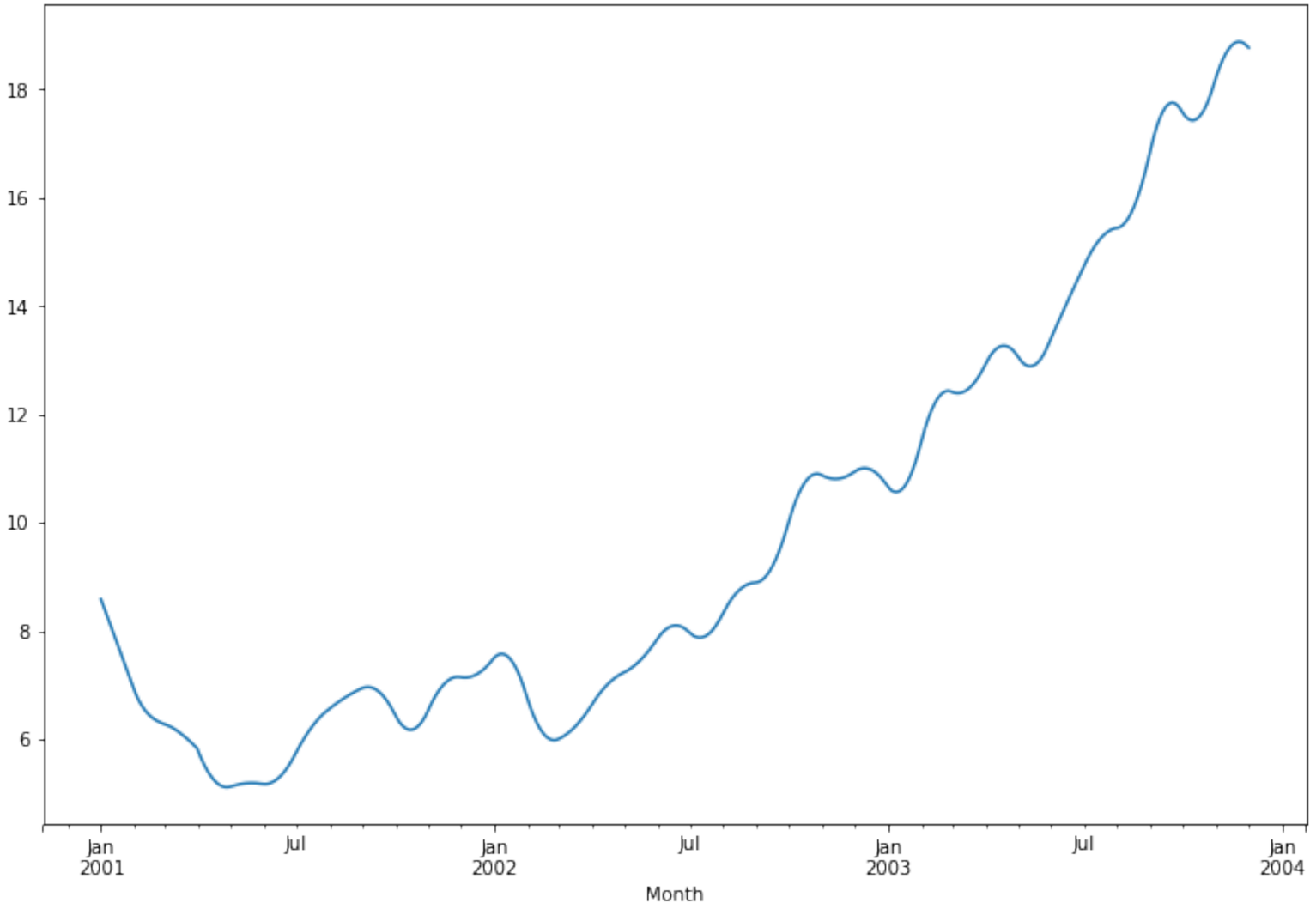
Resampled with Cubic Interpolation (use scipy)



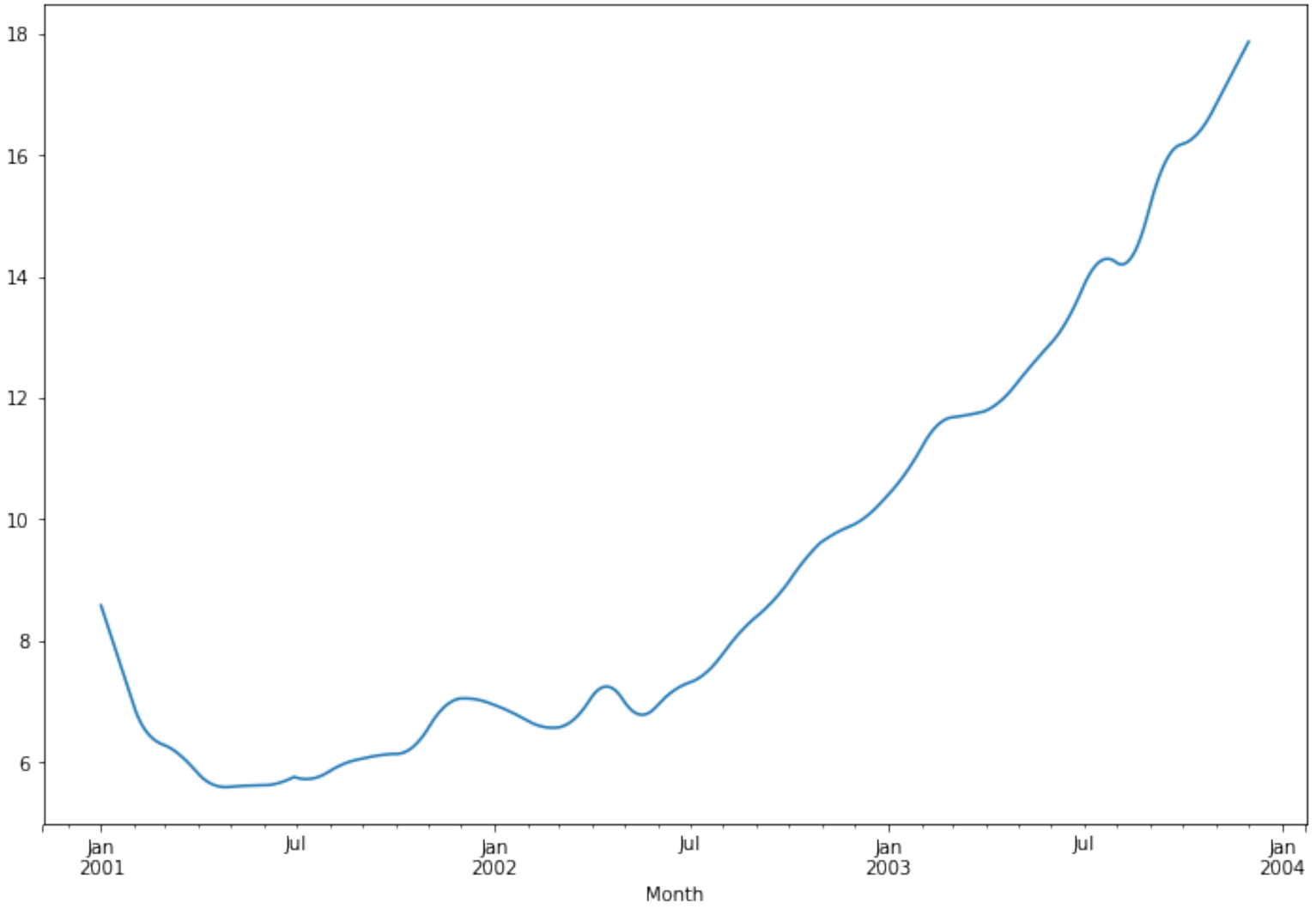
Piecewise Cubic Hermite Interpolating Polynomial (scipy)



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"	0.061966999999999994
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]

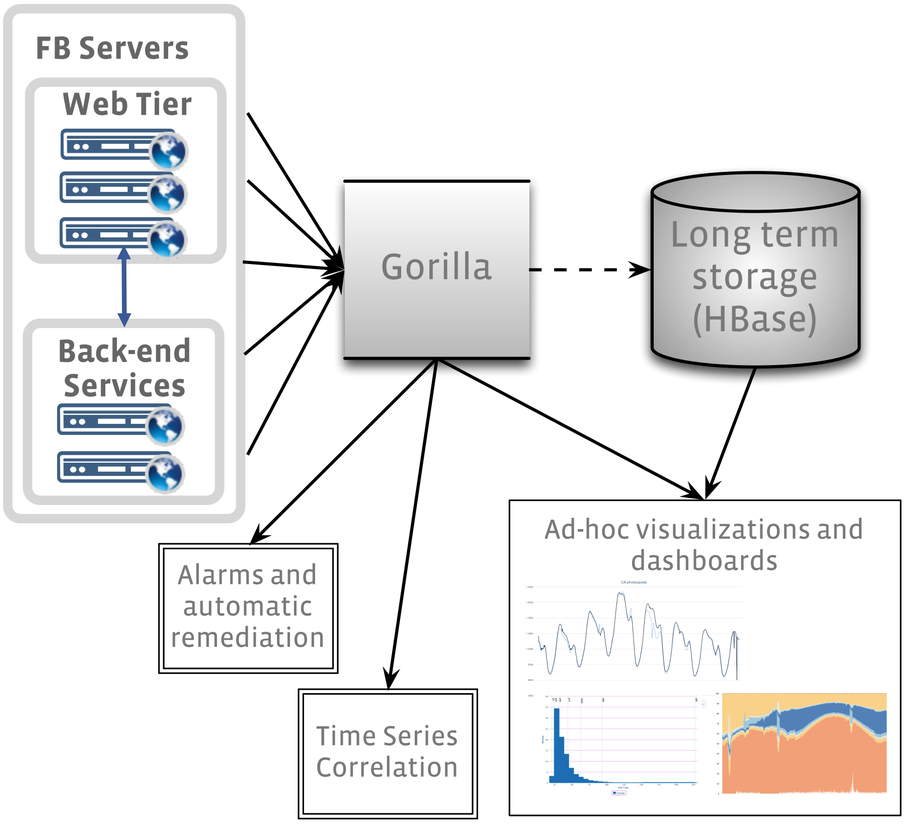
Storing Time Series Data in a RDBMS

- 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

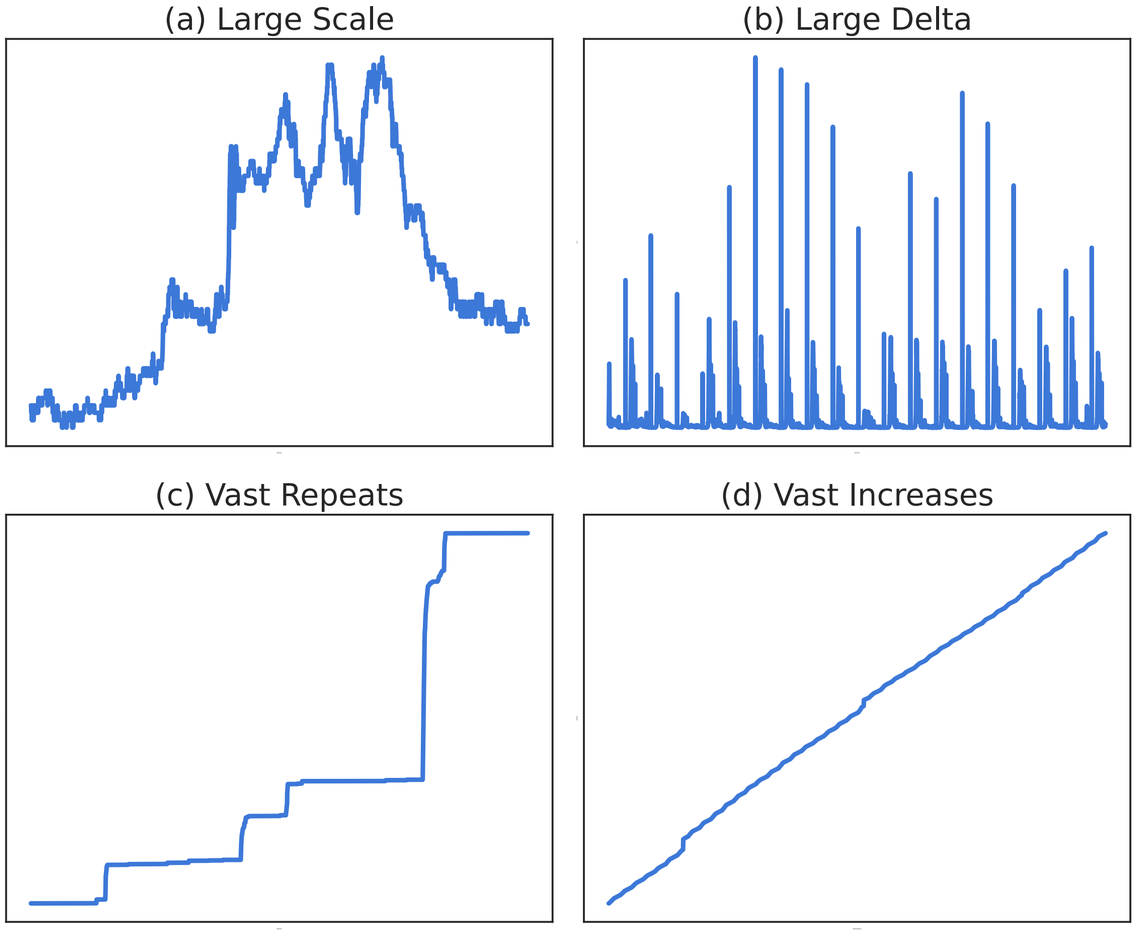
- 2 billion unique time series identified by a string key.
- 700 million data points (time stamp and value) added per minute.
- Store data for 26 hours.
- More than 40,000 queries per second at peak.
- Reads succeed in under one millisecond.
- Support time series with 15 second granularity (4 pts/minute 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

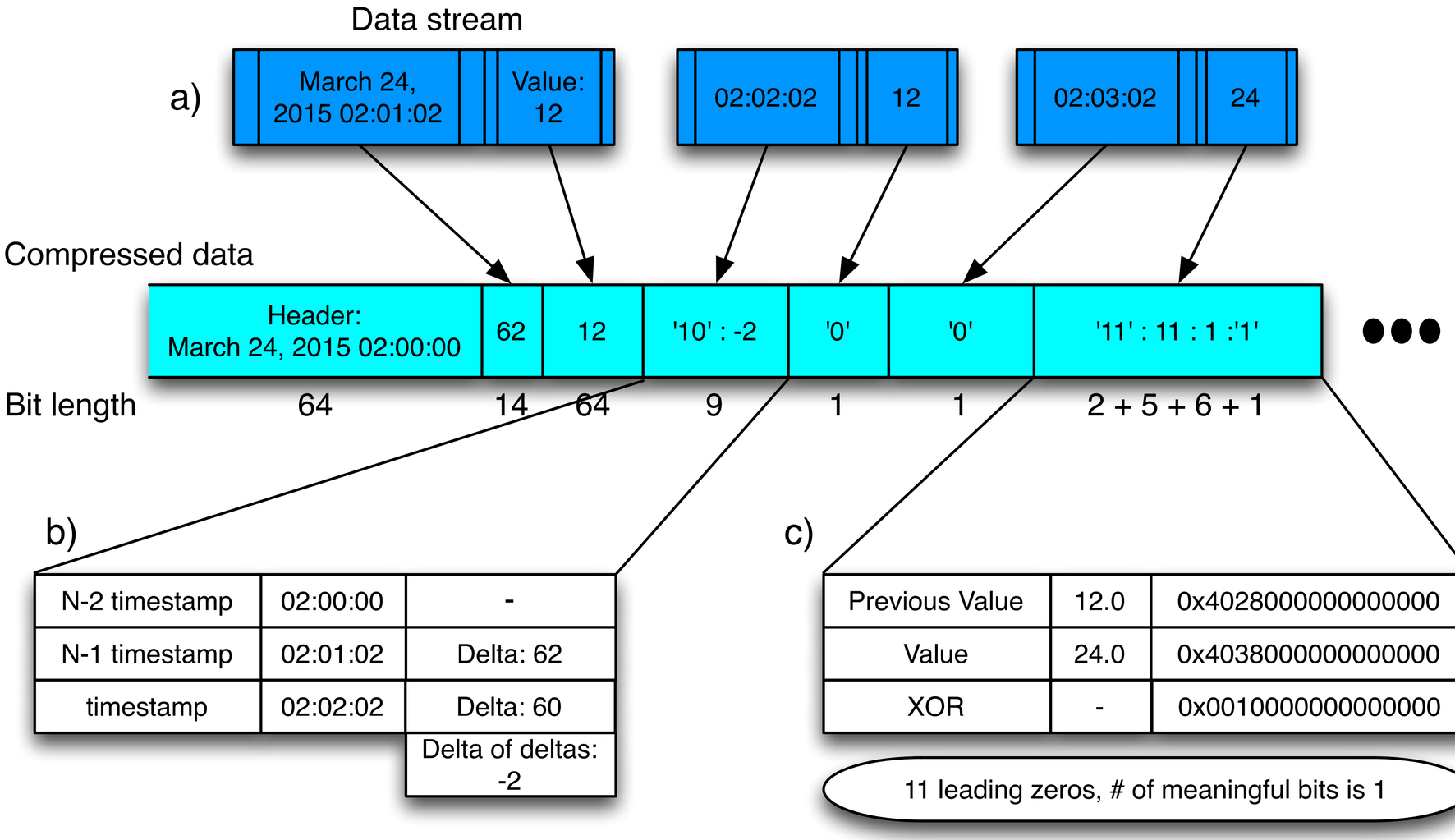
Time Series Data Patterns



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

[J. Xiao, 2021]

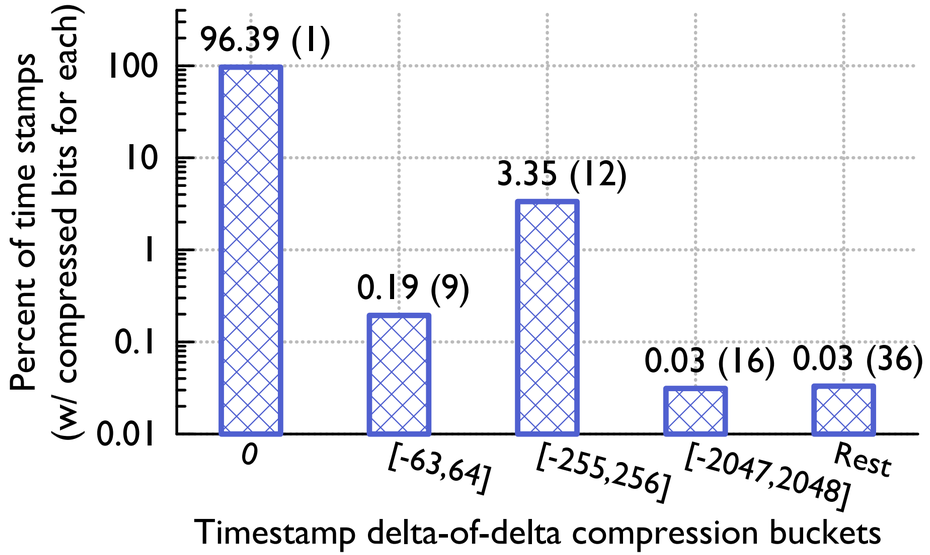
Gorilla Compression



[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

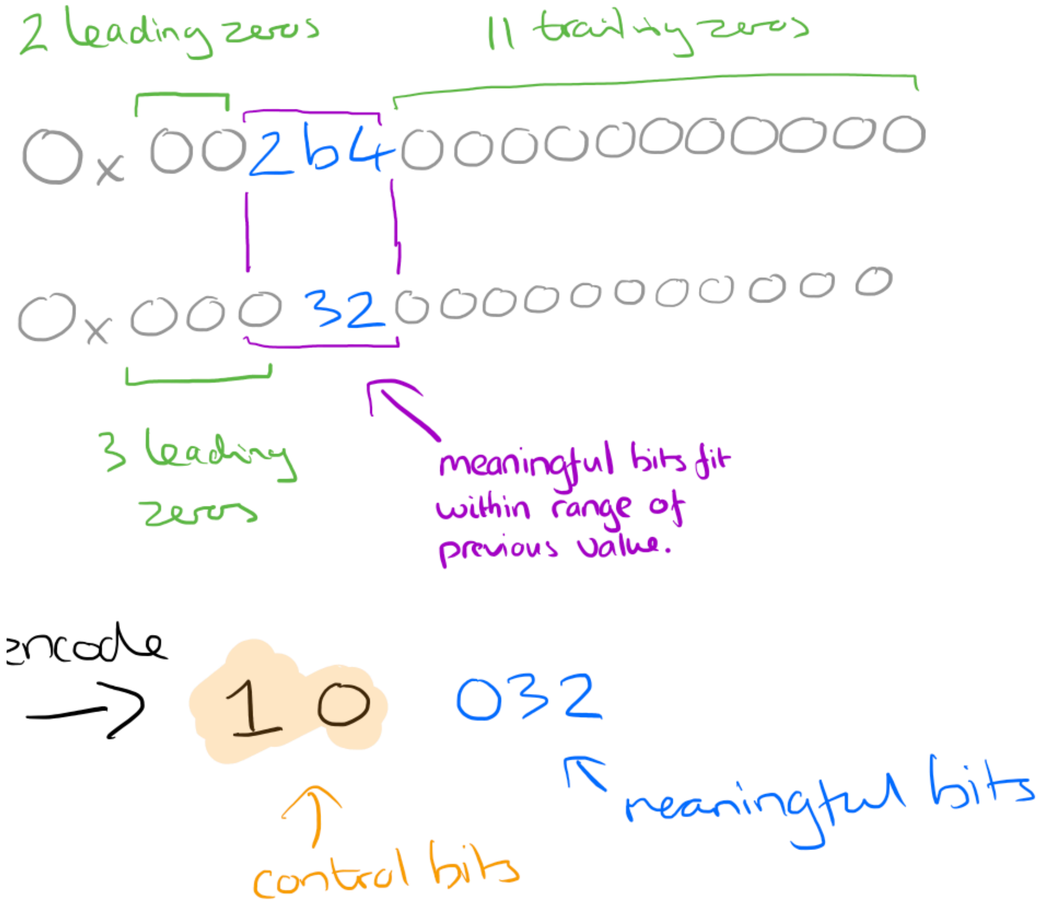
Decimal	Double Representation	XOR with previous
12	0x4028000000000000	
24	0x4038000000000000	0x0010000000000000
15	0x402e000000000000	0x0016000000000000
12	0x4028000000000000	0x0006000000000000
35	0x4041800000000000	0x0069800000000000

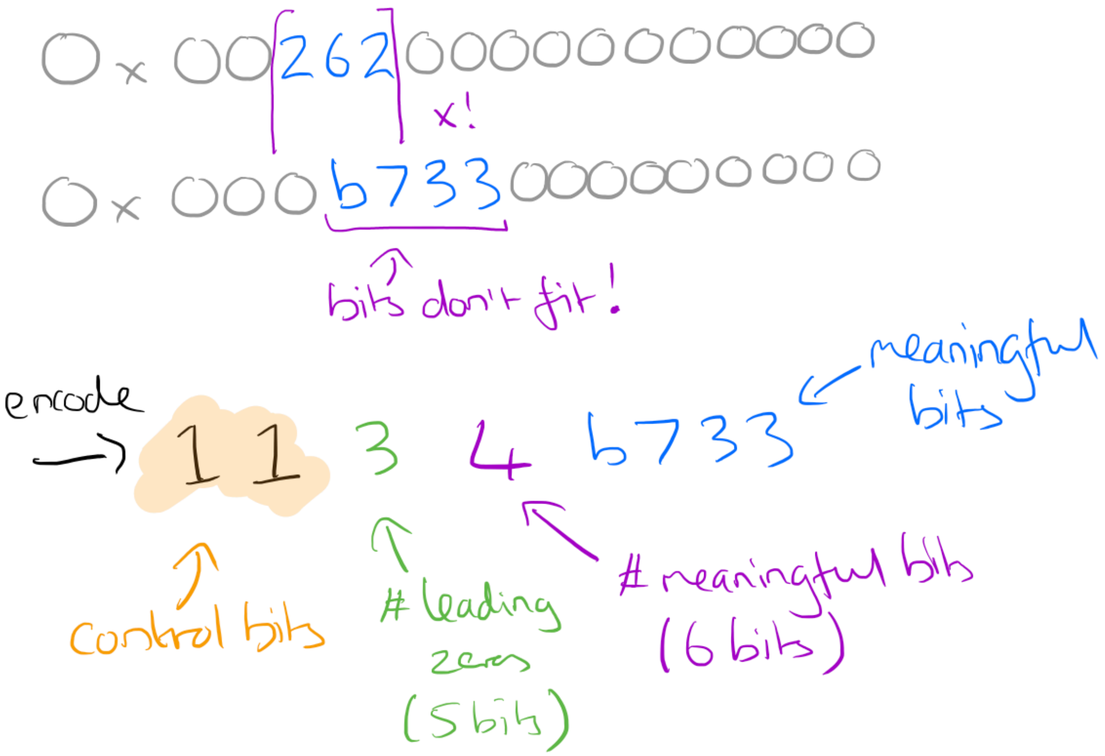
Decimal	Double Representation	XOR with previous
15.5	0x402f000000000000	
14.0625	0x402c200000000000	0x0003200000000000
3.25	0x400a000000000000	0x0026200000000000
8.625	0x4021400000000000	0x002b400000000000
13.1	0x402a333333333333	0x000b733333333333

- 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

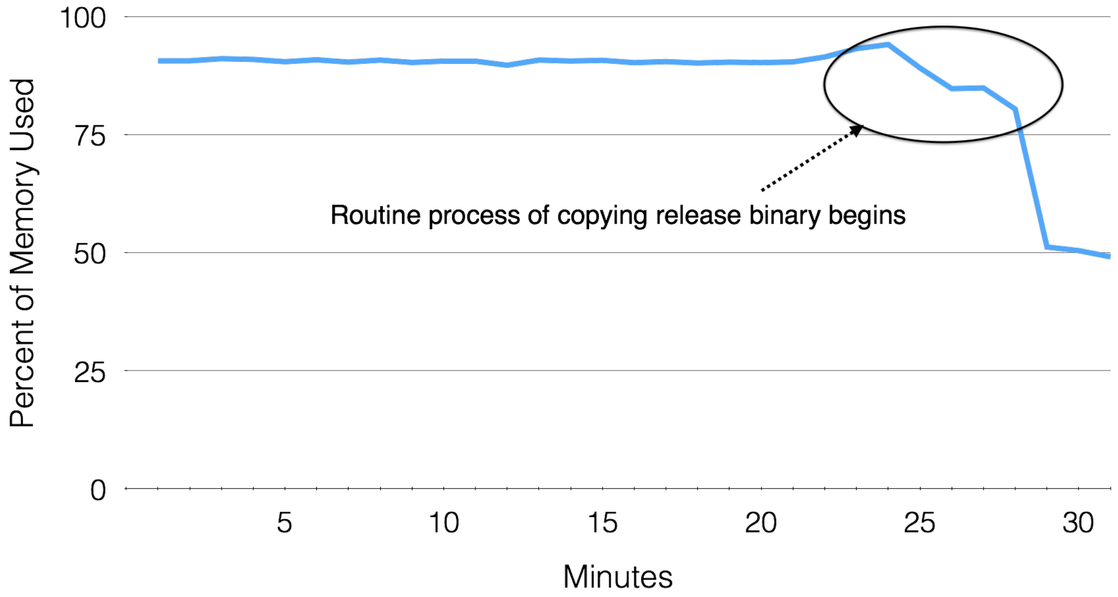




[A. Colyer]

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]