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


Extending Python ecosystem

Extending SQL databases

[A. Jindal et al., 2021]
# Magpie Goals

<table>
<thead>
<tr>
<th>Pythonic Environment</th>
<th>Unified Dataframe API</th>
<th>Magpie Middleware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Familiar Python surface</td>
<td>Ongoing standardization</td>
<td>Batching Pandas into large query expressions</td>
</tr>
<tr>
<td>Polyengines &amp; Mappers</td>
<td>Cross Optimization</td>
<td>Backend selection using past workloads</td>
</tr>
<tr>
<td>Database Backends</td>
<td>Common Data Layer</td>
<td>Cache commonly seen dataframes</td>
</tr>
<tr>
<td>Native Python</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Polyengines & Mappers**: Azure Synapse Analytics, Koalas, Alchemg
- **Database Backends**: Microsoft SCOPE, Apache Spark, PostgreSQL, Apache MADlib, Google BigQuery, SQL Server

**Magpie Goals**

[A. Jindal et al., 2021]
Magpie Architecture

- Pandas
- Ibis API
- Ibis Expression
- Backend Selection
- SQL
- Spark
- Pandas
- ... (rest of the diagram is not fully visible)

Lazy Translation
Cost-based optimization
Interactive experience
Cloud backends

[A. Jindal et al., 2021]
ConnectorX: Databases to Dataframes
Dataframe API?

- SQL, pandas, or something else?

D. Lee, Ponder CEO

📅 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

📅 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. Koop, CSCI 640/490, Spring 2024
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

<table>
<thead>
<tr>
<th>timestamp</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-07-12T11:51:45Z</td>
<td>465110000</td>
</tr>
<tr>
<td>2016-07-12T11:51:45Z</td>
<td>0.06196699999999994</td>
</tr>
<tr>
<td>2016-07-12T12:10:00Z</td>
<td>4937000000</td>
</tr>
<tr>
<td>2016-07-12T12:10:00Z</td>
<td>1857300000</td>
</tr>
<tr>
<td>2016-07-12T12:10:00Z</td>
<td>5902300000</td>
</tr>
</tbody>
</table>

What is a time series data? Comparison of Open Source TSDBs

- timestamp
- value
- generated
- message_subtype
- scaler
- short_id
- tenant

[Survey and Comparison of Open Source Time Series Databases]

[A. Bader, 2017]
Time Series Data

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

US Treasury bill contracts

Australian electricity production

Sales of new one–family houses, USA

Annual Canadian Lynx trappings

D. Koop, CSCI 640/490, Spring 2024
Examples

**US Treasury bill contracts**

**Australian electricity production**

**Sales of new one-family houses, USA**

**Annual Canadian Lynx trappings**

**Trend**

**Forecasting: Principles and Practice**

Seasonal or cyclic?

US Treasury bill contracts

Australian electricity production

Sales of new one-family houses, USA

Annual Canadian Lynx trappings

[R. J. Hyndman]
Examples

**US Treasury bill contracts**

**Australian electricity production**

**Sales of new one-family houses, USA**

**Annual Canadian Lynx trappings**

[Trend]

[Trend + Seasonality]

[R. J. Hyndman]
Examples

US Treasury bill contracts

Australian electricity production

Sales of new one-family houses, USA

Annual Canadian Lynx trappings

Trend

Trend + Seasonality

Seasonality + Cyclic

[R. J. Hyndman]
Examples

Trend

Seasonality + Cyclic

Trend + Seasonality

Stationary

[Examples]

US Treasury bill contracts

Australian electricity production

Sales of new one-family houses, USA

Annual Canadian Lynx trappings

[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](http://strftime.org)

- Generally, can create whatever format you need using these format strings

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>Weekday as locale’s abbreviated name.</td>
<td>Mon</td>
</tr>
<tr>
<td>%A</td>
<td>Weekday as locale’s full name.</td>
<td>Monday</td>
</tr>
<tr>
<td>%w</td>
<td>Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.</td>
<td>1</td>
</tr>
<tr>
<td>%d</td>
<td>Day of the month as a zero-padded decimal number.</td>
<td>30</td>
</tr>
<tr>
<td>%m</td>
<td>Month as a decimal number. (Platform specific)</td>
<td>9</td>
</tr>
<tr>
<td>%m</td>
<td>Month as a zero-padded decimal number.</td>
<td>09</td>
</tr>
<tr>
<td>%y</td>
<td>Year without century as a zero-padded decimal number.</td>
<td>13</td>
</tr>
<tr>
<td>%Y</td>
<td>Year with century as a decimal number.</td>
<td>2013</td>
</tr>
<tr>
<td>%H</td>
<td>Hour (24-hour clock) as a zero-padded decimal number.</td>
<td>07</td>
</tr>
<tr>
<td>%I</td>
<td>Hour (12-hour clock) as a zero-padded decimal number.</td>
<td>07</td>
</tr>
<tr>
<td>%p</td>
<td>Locale’s equivalent of either AM or PM.</td>
<td>AM</td>
</tr>
<tr>
<td>%M</td>
<td>Minute as a zero-padded decimal number.</td>
<td>06</td>
</tr>
<tr>
<td>%S</td>
<td>Second as a zero-padded decimal number.</td>
<td>05</td>
</tr>
<tr>
<td>%f</td>
<td>Microsecond as a decimal number, zero-padded on the left.</td>
<td>000000</td>
</tr>
</tbody>
</table>
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()`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>freq</code></td>
<td>String or DateOffset indicating desired resampled frequency (e.g., 'M', 'Smin', or <code>Second(15)</code>)</td>
</tr>
<tr>
<td><code>axis</code></td>
<td>Axis to resample on; default axis=0</td>
</tr>
<tr>
<td><code>fill_method</code></td>
<td>How to interpolate when upsampling, as in 'ffill' or 'bfill'; by default does no interpolation</td>
</tr>
<tr>
<td><code>closed</code></td>
<td>In downsampling, which end of each interval is closed (inclusive), 'right' or 'left'</td>
</tr>
<tr>
<td><code>label</code></td>
<td>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)</td>
</tr>
<tr>
<td><code>loffset</code></td>
<td>Time adjustment to the bin labels, such as '-1s' / <code>Second(-1)</code> to shift the aggregate labels one second earlier</td>
</tr>
<tr>
<td><code>limit</code></td>
<td>When forward or backward filling, the maximum number of periods to fill</td>
</tr>
<tr>
<td><code>kind</code></td>
<td>Aggregate to periods (<code>period</code>) or timestamps (<code>timestamp</code>); defaults to the type of index the time series has</td>
</tr>
<tr>
<td><code>convention</code></td>
<td>When resampling periods, the convention ('start' or 'end') for converting the low-frequency period to high frequency; defaults to 'end'</td>
</tr>
</tbody>
</table>

[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

<table>
<thead>
<tr>
<th>Alias</th>
<th>Offset Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Day</td>
<td>Calendar daily</td>
</tr>
<tr>
<td>B</td>
<td>BusinessDay</td>
<td>Business daily</td>
</tr>
<tr>
<td>H</td>
<td>Hour</td>
<td>Hourly</td>
</tr>
<tr>
<td>T or min</td>
<td>Minute</td>
<td>Minutely</td>
</tr>
<tr>
<td>S</td>
<td>Second</td>
<td>Secondly</td>
</tr>
<tr>
<td>L or ms</td>
<td>Milli</td>
<td>Millisecond (1/1000th of 1 second)</td>
</tr>
<tr>
<td>U</td>
<td>Micro</td>
<td>Microsecond (1/1000000th of 1 second)</td>
</tr>
<tr>
<td>M</td>
<td>MonthEnd</td>
<td>Last calendar day of month</td>
</tr>
<tr>
<td>BM</td>
<td>BusinessMonthEnd</td>
<td>Last business day (weekday) of month</td>
</tr>
<tr>
<td>MS</td>
<td>MonthBegin</td>
<td>First calendar day of month</td>
</tr>
<tr>
<td>BMS</td>
<td>BusinessMonthBegin</td>
<td>First weekday day of month</td>
</tr>
<tr>
<td>W-MON, W-TUE, ...</td>
<td>Week</td>
<td>Weekly on given day of week: MON, TUE, WED, THU, FRI, SAT, or SUN.</td>
</tr>
<tr>
<td>WOM-1MON, WOM-2MON, ...</td>
<td>WeekOfMonth</td>
<td>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.</td>
</tr>
</tbody>
</table>

[W. McKinney, Python for Data Analysis]
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:
  - \( s[\text{'2017-12-01'}:\text{'2017-12-31'}] \)
Shifting Data

• Leading or Lagging Data

Shifting (Leading and Lagging) Data

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

In [96]: ts                  In [97]: ts.shift(2)         In [98]: ts.shift(-2)
Out[96]:                     Out[97]:                     Out[98]:
2000-01-31   -0.066748       2000-01-31         NaN       2000-01-31   -0.117388
2000-02-29    0.838639       2000-02-29         NaN       2000-02-29   -0.517795
2000-03-31   -0.117388       2000-03-31   -0.066748       2000-03-31         NaN
2000-04-30   -0.517795       2000-04-30    0.838639       2000-04-30         NaN
Freq: M, dtype: float64      Freq: M, dtype: float64      Freq: M, dtype: float64

A common use of `shift` is computing percent changes in a time series or multiple time
series as DataFrame columns. This is expressed as

\[
\frac{ts}{ts\text{.shift}(1)} - 1
\]

Because naive shifts leave the index unmodified, some data is discarded. Thus if the
frequency is known, it can be passed to `shift` to advance the timestamps instead of
simply the data:

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

Other frequencies can be passed, too, giving you a lot of flexibility in how to lead and
lag the data:

In [100]: ts.shift(3, freq='D')        In [101]: ts.shift(1, freq='3D')
Out[100]:                              Out[101]:
2000-02-03   -0.066748                 2000-02-03   -0.066748
2000-03-03    0.838639                 2000-03-03    0.838639
2000-04-03   -0.117388                 2000-04-03   -0.117388
2000-05-03   -0.517795                 2000-05-03   -0.517795
dtype: float64                         dtype: float64

In [102]: ts.shift(1, freq='90T')
Out[102]:
2000-01-31 01:30:00   -0.066748
2000-02-29 01:30:00    0.838639
2000-03-31 01:30:00   -0.117388
2000-04-30 01:30:00   -0.517795
Freq: M, dtype: float64

Shifting dates with offsets

The pandas date offsets can also be used with `datetime` or `Timestamp` objects:

In [103]: from pandas.tseries.offsets import Day, MonthEnd
In [104]: now = datetime(2011, 11, 17)
In [105]: now + 3 * Day()
Out[105]: Timestamp('2011-11-20 00:00:00')

If you add an anchored offset like `MonthEnd`, the first increment will
roll forward a date to the next date according to the frequency rule:

In [106]: now + MonthEnd()
Out[106]: Timestamp('2011-11-30 00:00:00')
In [107]: now + MonthEnd(2)
Out[107]: Timestamp('2011-12-31 00:00:00')

Anchored offsets can explicitly "roll" dates forward or backward using their
`rollforward` and `rollback` methods, respectively:

In [108]: offset = MonthEnd()
In [109]: offset.rollforward(now)
Out[109]: Timestamp('2011-11-30 00:00:00')

• Shifting by time:
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())
  - s - s.shift(1, 'd')
  ```

<table>
<thead>
<tr>
<th>Date</th>
<th>Value</th>
<th>Date</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017-11-30</td>
<td>NaN</td>
<td>2017-11-30</td>
<td>NaN</td>
</tr>
<tr>
<td>2017-12-02</td>
<td>-3.0</td>
<td>2017-12-01</td>
<td>NaN</td>
</tr>
<tr>
<td>2017-12-03</td>
<td>-1.0</td>
<td>2017-12-02</td>
<td>NaN</td>
</tr>
<tr>
<td>2017-12-04</td>
<td>4.0</td>
<td>2017-12-03</td>
<td>-1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2017-12-04</td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2017-12-05</td>
<td>NaN</td>
</tr>
</tbody>
</table>
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

```python
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',
              '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20'],
       dtype='datetime64[ns]', freq='D')
```
Lots of Frequencies (not comprehensive)

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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>freq</td>
<td>String or DateOffset indicating desired resampled frequency (e.g., 'M', 'Smin', or <code>Second(15)</code>)</td>
</tr>
<tr>
<td>axis</td>
<td>Axis to resample on; default axis=0</td>
</tr>
<tr>
<td>fill_method</td>
<td>How to interpolate when upsampling, as in 'ffill' or 'bfill'; by default does no interpolation</td>
</tr>
<tr>
<td>closed</td>
<td>In downsampling, which end of each interval is closed (inclusive), 'right' or 'left'</td>
</tr>
<tr>
<td>label</td>
<td>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)</td>
</tr>
<tr>
<td>loffset</td>
<td>Time adjustment to the bin labels, such as '-is' / <code>Second(-1)</code> to shift the aggregate labels one second earlier</td>
</tr>
<tr>
<td>limit</td>
<td>When forward or backward filling, the maximum number of periods to fill</td>
</tr>
<tr>
<td>kind</td>
<td>Aggregate to periods ('period') or timestamps ('timestamp'); defaults to the type of index the time series has</td>
</tr>
</tbody>
</table>
| 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)

![Diagram showing bin edges and labeling for downsampling](image)

**Example Code:**
```python
In [219]: ts.resample('5min', closed='right', label='right', loffset='-1s').sum()
Out [219]:
```

**You also could have accomplished the effect of** `loffset` **by calling the** `shift` **method on the result without the`loffset`**.
Upsampling

• No aggregation necessary

In [222]: frame
Out[222]:

<table>
<thead>
<tr>
<th></th>
<th>Colorado</th>
<th>Texas</th>
<th>New York</th>
<th>Ohio</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-05</td>
<td>-0.896431</td>
<td>0.677263</td>
<td>0.036503</td>
<td>0.087102</td>
</tr>
<tr>
<td>2000-01-12</td>
<td>-0.046662</td>
<td>0.927238</td>
<td>0.482284</td>
<td>-0.867130</td>
</tr>
</tbody>
</table>

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

In [224]: df_daily
Out[224]:

<table>
<thead>
<tr>
<th></th>
<th>Colorado</th>
<th>Texas</th>
<th>New York</th>
<th>Ohio</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-05</td>
<td>-0.896431</td>
<td>0.677263</td>
<td>0.036503</td>
<td>0.087102</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-09</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-10</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-11</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-12</td>
<td>-0.046662</td>
<td>0.927238</td>
<td>0.482284</td>
<td>-0.867130</td>
</tr>
</tbody>
</table>

In [225]: frame.resample('D').ffill()
Out[225]:

<table>
<thead>
<tr>
<th></th>
<th>Colorado</th>
<th>Texas</th>
<th>New York</th>
<th>Ohio</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-05</td>
<td>-0.896431</td>
<td>0.677263</td>
<td>0.036503</td>
<td>0.087102</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>-0.896431</td>
<td>0.677263</td>
<td>0.036503</td>
<td>0.087102</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>-0.896431</td>
<td>0.677263</td>
<td>0.036503</td>
<td>0.087102</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>-0.896431</td>
<td>0.677263</td>
<td>0.036503</td>
<td>0.087102</td>
</tr>
<tr>
<td>2000-01-09</td>
<td>-0.896431</td>
<td>0.677263</td>
<td>0.036503</td>
<td>0.087102</td>
</tr>
<tr>
<td>2000-01-10</td>
<td>-0.896431</td>
<td>0.677263</td>
<td>0.036503</td>
<td>0.087102</td>
</tr>
<tr>
<td>2000-01-11</td>
<td>-0.896431</td>
<td>0.677263</td>
<td>0.036503</td>
<td>0.087102</td>
</tr>
<tr>
<td>2000-01-12</td>
<td>-0.046662</td>
<td>0.927238</td>
<td>0.482284</td>
<td>-0.867130</td>
</tr>
</tbody>
</table>
Rolling Window Calculations

12 8 7 4 9 13 4 11 3 8
Rolling Window Calculations

\[
\begin{array}{cccccccc}
12 & 8 & 7 & 4 & 9 & 13 & 4 & 11 & 3 & 8 \\
\end{array}
\]

7.8
Rolling Window Calculations

The diagram illustrates a rolling window of data points. The window slides through the data, calculating a statistic (in this case, the mean) over the window's size. The mean of the highlighted window is 7.8.
Rolling Window Calculations

\[
\begin{array}{cccccccc}
12 & 8 & 7 & 4 & 9 & 13 & 4 & 11 & 3 & 8 \\
\end{array}
\]

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=\text{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
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

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

<table>
<thead>
<tr>
<th>time</th>
<th>generated</th>
<th>message_subtype</th>
<th>scaler</th>
<th>short_id</th>
<th>tenant</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-07-12T11:51:45Z</td>
<td>&quot;true&quot;</td>
<td>&quot;34&quot;</td>
<td>&quot;4&quot;</td>
<td>&quot;3&quot;</td>
<td>&quot;saarlois&quot;</td>
<td>465110000</td>
</tr>
<tr>
<td>2016-07-12T11:51:45Z</td>
<td>&quot;true&quot;</td>
<td>&quot;34&quot;</td>
<td>&quot;+6&quot;</td>
<td>&quot;2&quot;</td>
<td>&quot;saarlois&quot;</td>
<td>0.0619669999999994</td>
</tr>
<tr>
<td>2016-07-12T12:00:00Z</td>
<td>&quot;true&quot;</td>
<td>&quot;34&quot;</td>
<td>&quot;7&quot;</td>
<td>&quot;5&quot;</td>
<td>&quot;saarlois&quot;</td>
<td>4937000000</td>
</tr>
<tr>
<td>2016-07-12T12:00:00Z</td>
<td>&quot;true&quot;</td>
<td>&quot;34&quot;</td>
<td>&quot;6&quot;</td>
<td>&quot;2&quot;</td>
<td>&quot;saarlois&quot;</td>
<td>18573000000</td>
</tr>
<tr>
<td>2016-07-12T12:00:00Z</td>
<td>&quot;true&quot;</td>
<td>&quot;34&quot;</td>
<td>&quot;5&quot;</td>
<td>&quot;7&quot;</td>
<td>&quot;saarlois&quot;</td>
<td>59023000000</td>
</tr>
</tbody>
</table>

[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)
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
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.

[Peikonen 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

(a) Large Scale

(b) Large Delta

(c) Vast Repeats

(d) Vast Increases

• Numerical Data Features:
  - Scale
  - Delta
  - Repeat
  - Increase

• Text Data Features
  - Value
  - Character

[J. Xiao, 2021]
Gorilla Compression

Figure 2: Visualizing the entire compression algorithm. For this example, 48 bytes of values and time stamps are compressed to just under 21 bytes/167 bits.

The key specified in the monitoring data is used to uniquely identify a time series. By sharding all monitoring data based on these unique string keys, each time series dataset can be mapped to a single Gorilla host. Thus, we can scale Gorilla by simply adding new hosts and tuning the sharding function to map new time series data to the expanded set of hosts. When Gorilla was launched to production 18 months ago, our dataset of all time series data inserted in the past 26 hours fit into 1.3TB of RAM evenly distributed across 20 machines. Since then, we have had to double the size of the clusters twice due to data growth, and are now running on 80 machines within each Gorilla cluster. This process was simple due to the share-nothing architecture and focus on horizontal scalability.

Gorilla tolerates single node failures, network cuts, and entire datacenter failures by writing each time series value to two hosts in separate geographic regions. On detecting a failure, all read queries are failed over to the alternate region ensuring that users do not experience any disruption.

4.1 Time series compression

In evaluating the feasibility of building an in-memory time series database, we considered several existing compression schemes to reduce the storage overhead. We identified techniques that applied solely to integer data which didn’t meet our requirement of storing double precision floating point values. Other techniques operated on a complete dataset but did not support compression over a stream of data as was stored in Gorilla [7, 13]. We also identified lossy time series approximation techniques used in data mining to make the problem set more easily fit within memory [15, 11], but Gorilla is focused on keeping the full resolution representation of data.

Our work was inspired by a compression scheme for floating point data derived in scientific computation. This scheme leveraged XOR comparison with previous values to generate a delta encoding [25, 17].

Gorilla compresses data points within a time series with no additional compression used across time series. Each data point is a pair of 64 bit values representing the time stamp and value at that time. Timestamps and values are compressed separately using information about previous values. The overall compression scheme is visualized in Figure 2, showing how time stamps and values are interleaved in the compressed block.

Figure 2.a illustrates the time series data as a stream consisting of pairs of measurements (values) and time stamps. Gorilla compresses this data stream into blocks, partitioned by time. After a simple header with an aligned time stamp (starting at 2 am, in this example) and storing the first value in a less compressed format, Figure 2.b shows that time stamps are compressed using delta-of-delta compression, described in more detail in Section 4.1.1. As shown in Figure 2.b the time stamp delta of delta is 2. This is stored with a two bit header (‘10’), and the value is stored in seven bits, for a total size of just 9 bits. Figure 2.c shows floating-point values are compressed using XOR compression, described in more detail in Section 4.1.2. By XORing the floating point value with the previous value, we find that there is only a single meaningful bit in the XOR. This is then encoded with a two bit header (‘11’), encoding that there are eleven leading zeros, a single meaningful bit, and the actual value (‘1’). This is stored in fourteen total bits.
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

From analyzing our ODS data, we discovered that the vast majority of ODS data points can be compressed to a single bit. We leverage this to compute a simple XOR of the current time stamp value from the previous one and previous values rather than employing a delta encoding with the following algorithm:

1. The first value is stored with no compression
2. If XOR with the previous is zero (same value), store a single '0' bit
3. When XOR is non-zero, calculate the number of leading and trailing zeros in the XOR, store bit '1' followed by zero or one more than the number of leading or trailing zeros, respectively
4. If the value (D) is between [-255, 256], store '11' followed by the value (7 bits)
5. If the value is in [-2047,2048], store '110' followed by the value (9 bits)
6. If the value is between [-63,64], store '10' followed by the value (7 bits)
7. If the value is zero, then store a single '0' bit
8. If the value is between [-2047,2048], store '1110' followed by the value (12 bits)
9. If the value is in [-63,64], store '1111' followed by the value (32 bits)
10. Otherwise store '1111' followed by the value (32 bits)

Figure 3 show the results of time stamp compression in different ranges were selected by sampling a set of real time series from the production system, which gives us 0, -1 and 2. An example of how this works is shown in Figure 2.

Rather than storing timestamps in their entirety, we store the first time stamp delta sized at 14 bits, because that is enough to span a bit more than 4 hours (16,384 seconds) allowing the point may have a time stamp that is 1 second early or late, but the window is usually constrained.

We noticed that the vast majority of ODS data points could optimize the compression scheme implemented in Gorilla. We analyzed the time series data stored in ODS so we rather than storing timestamps in their entirety, we store the first time stamp delta sized at 14 bits, because that is enough to span a bit more than 4 hours (16,384 seconds) allowing the point may have a time stamp that is 1 second early or late, but the window is usually constrained.

We then encode these XOR'd values with the following algorithm:

Figure 3: Distribution of time stamp compression

[Pelkonen et al., 2015]
XOR Representation

- 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

---

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Double Representation</th>
<th>XOR with previous</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x4028000000000000</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>0x4038000000000000</td>
<td>0x0010000000000000</td>
</tr>
<tr>
<td>15</td>
<td>0x402e000000000000</td>
<td>0x0016000000000000</td>
</tr>
<tr>
<td>12</td>
<td>0x4028000000000000</td>
<td>0x0006000000000000</td>
</tr>
<tr>
<td>35</td>
<td>0x4041800000000000</td>
<td>0x0059800000000000</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Double Representation</th>
<th>XOR with previous</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.5</td>
<td>0x402f000000000000</td>
<td></td>
</tr>
<tr>
<td>14.0625</td>
<td>0x402c200000000000</td>
<td>0x0003200000000000</td>
</tr>
<tr>
<td>3.25</td>
<td>0x400c000000000000</td>
<td>0x0026200000000000</td>
</tr>
<tr>
<td>8.625</td>
<td>0x4021400000000000</td>
<td>0x002b400000000000</td>
</tr>
<tr>
<td>13.1</td>
<td>0x402a333333333333</td>
<td>0x0009733333333333</td>
</tr>
</tbody>
</table>

[Pelkonen et al., 2015]
XOR Compression

2 leading zeros 11 leading zeros
0 \times 002640000000000000
0 \times 000320000000000000

3 leading zeros meaningful bits fit within range of previous value.

encode

10 032 meaningful bits

encode

13 4 6733 meaningful bits

# leading zeros (5 bits)

# meaningful bits (6 bits)

bits don't fit!
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
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]