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

## Higher-level Abstractions
- MODIN
- Koalas
- Ibis

## APIs
- `pandas`
- **Vaex Dataframe**
- **DASK Dataframe**
- Cuda Dataframe
- Ray Programs
- PySpark Dataframe

## Data Layer
- **NumPy Arrays**
- **ARROW**
- Distributed

## Backends
- Native Python
- Dask
- Nvidia RAPIDS
- Ray
- PySpark

## Extending Python ecosystem
- **DASK**
- RAPIDS
- RAY

## Extending SQL databases

<table>
<thead>
<tr>
<th>SQL Extensions</th>
<th>SQL + Built-in Functions</th>
<th>SQL + User Defined Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft SCOPE</td>
<td>Apache MADlib</td>
<td>Google BigQuery</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Microsoft SQL Server</td>
<td>Azure Synapse Analytics</td>
</tr>
</tbody>
</table>

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

- Pythonic Environment
- Unified Dataframe API
- Magpie Middleware
  - PyFroid Compiler
  - Cross Optimization
  - Common Data Layer
- Polyengines & Mappers
- Database Backends
- Native Python
- Cloud backends

Familiar Python surface
Ongoing standardization
Batching Pandas into large query expressions
Backend selection using past workloads
Cache commonly seen dataframes
Multi-backend environments and libraries

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

- Pandas
  - Ibis API
    - Ibis Expression
      - Cost-based optimization
      - Backend Selection
        - SQL
        - Spark
        - Pandas
        - ... (not all backend options shown)
      - Lazy Translation
  - Dataframe cache

- Interactive experience
  - Cloud backends

- Magpie Architecture

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

[X. Wang, 2022]
Dataframe API?

- SQL, pandas, or something else?

[Image]

D. Koop, CSCI 640/490, Spring 2023
Assignment 4

- Work on Data Integration and Data Fusion
- Integrate travel datasets from different institutions (UN World Tourism Office, World Bank, OECD)
  - Integrate information with population
- Record Matching:
  - Which countries are the same?
- Data Fusion:
  - The receipts/expenditures
  - Country names
Test 2

- Upcoming… April 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

<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>49370000000</td>
</tr>
<tr>
<td>2016-07-12T12:10:00Z</td>
<td>18573000000</td>
</tr>
<tr>
<td>2016-07-12T12:10:00Z</td>
<td>59023000000</td>
</tr>
</tbody>
</table>

[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 2023

Northern Illinois University
Examples

Trend

US Treasury bill contracts

Australian electricity production

Sales of new one-family houses, USA

Annual Canadian Lynx trappings

[R. J. Hyndman]
Examples

Trend

US Treasury bill contracts

Trend + Seasonality

Australian electricity production

Sales of new one-family houses, USA

Annual Canadian Lynx trappings

[R. J. Hyndman]
Examples

Time series patterns
Forecasting: Principles and Practice
Seasonal or cyclic?

US Treasury bill contracts

Trend

Australian electricity production

Trend +
Seasonality

Sales of new one-family houses, USA

Seasonality +
Cyclic

Annual Canadian Lynx trappings

[R. J. Hyndman]
Examples

**US Treasury bill contracts**

- Trend

**Australian electricity production**

- Trend + Seasonality

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

- Seasonality + Cyclic

**Annual Canadian Lynx trappings**

- Stationary
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>%-d</td>
<td>Day of the month as a decimal number. (Platform specific)</td>
<td>30</td>
</tr>
<tr>
<td>%b</td>
<td>Month as locale’s abbreviated name.</td>
<td>Sep</td>
</tr>
<tr>
<td>%B</td>
<td>Month as locale’s full name.</td>
<td>September</td>
</tr>
<tr>
<td>%m</td>
<td>Month as a zero-padded decimal number.</td>
<td>09</td>
</tr>
<tr>
<td>%-m</td>
<td>Month as a decimal number. (Platform specific)</td>
<td>9</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>%-H</td>
<td>Hour (24-hour clock) as a decimal number. (Platform specific)</td>
<td>7</td>
</tr>
<tr>
<td>%I</td>
<td>Hour (12-hour clock) as a zero-padded decimal number.</td>
<td>07</td>
</tr>
<tr>
<td>%-I</td>
<td>Hour (12-hour clock) as a decimal number. (Platform specific)</td>
<td>7</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>%-M</td>
<td>Minute as a decimal number. (Platform specific)</td>
<td>6</td>
</tr>
<tr>
<td>%S</td>
<td>Second as a zero-padded decimal number.</td>
<td>05</td>
</tr>
<tr>
<td>%-S</td>
<td>Second as a decimal number. (Platform specific)</td>
<td>5</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
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'
Some frequencies describe points in time that are not evenly spaced. For example, ‘M’ (calendar month end) and ‘BM’ (last business/weekday of month) depend on the number of days in a month and, in the latter case, whether the month ends on a weekend or not. For lack of a better term, I call these anchored offsets.

See Table 10-4 for a listing of frequency codes and date offset classes available in pandas.

Users can define their own custom frequency classes to provide date logic not available in pandas, though the full details of that are outside the scope of this book.

### Table 10-4. Base 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]
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**

  ```python
  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
  ```

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

  ```python
  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
  ```

  ```python
  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
  dtype: float64
  ```

- **Shifting dates with offsets**

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

  ```python
  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:

  ```python
  In [106]: now + MonthEnd()  
  Out[106]: Timestamp('2011-11-30 00:00:00')
  ```

  ```python
  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:

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

  ```python
  In [110]: offset.rollback(now)  
  Out[110]: Timestamp('2011-11-29 00:00:00')
  ```

- **Shifting by time**:

  ```python
  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:
  
  ```python
  [('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>M</td>
<td>Minute</td>
<td>Minuteley</td>
</tr>
<tr>
<td>S</td>
<td>Second</td>
<td>Secondly</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>NS</td>
<td>MonthBegin</td>
<td>First calendar day of month</td>
</tr>
<tr>
<td>BNS</td>
<td>BusinessMonthBegin</td>
<td>First weekday of month</td>
</tr>
<tr>
<td>W</td>
<td>Week</td>
<td>Weekly on given day of week (MON, TUE, WED, THU, FRI, SAT, or SUN)</td>
</tr>
<tr>
<td>WOM</td>
<td>WeekOnMonth</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</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</td>
<td>BusinessQuarterEnd</td>
<td>Quarterly dates anchored on last calendar day of each month, for year ending in indicated month</td>
</tr>
<tr>
<td>QS</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</td>
<td>BusinessQuarterBegin</td>
<td>Quarterly dates anchored on first calendar day of each month, for year ending in indicated month</td>
</tr>
<tr>
<td>A</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</td>
<td>BusinessYearEnd</td>
<td>Annual dates anchored on last weekday of given month</td>
</tr>
<tr>
<td>AS</td>
<td>YearBegin</td>
<td>Annual dates anchored on first day of given month</td>
</tr>
<tr>
<td>BAS</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><code>freq</code></td>
<td>String or DateOffset indicating desired resampled frequency (e.g., 'M', 'Smin', or Second(15))</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' / Second(-1) 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 ('period') or timestamps ('timestamp'); 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]
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 labels for 'left' and 'right' with timestamps 9:00 to 9:05]
Upsampling

- No aggregation necessary

```python
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

\[
\begin{array}{cccccccc}
12 & 8 & 7 & 4 & 9 & 13 & 4 & 11 & 3 & 8 \\
7.8
\end{array}
\]
Rolling Window Calculations

7.8
Rolling Window Calculations

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

7.8 7.0
Rolling Window Calculations

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

7.8 7.0
Rolling Window Calculations

<table>
<thead>
<tr>
<th>12</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>9</th>
<th>13</th>
<th>4</th>
<th>11</th>
<th>3</th>
<th>8</th>
</tr>
</thead>
</table>

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

```
"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;saarlouis&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;saarlouis&quot;</td>
<td>0.06196699999999994</td>
</tr>
<tr>
<td>2016-07-12T12:10:00Z</td>
<td>&quot;true&quot;</td>
<td>&quot;34&quot;</td>
<td>&quot;7&quot;</td>
<td>&quot;5&quot;</td>
<td>&quot;saarlouis&quot;</td>
<td>49370000000</td>
</tr>
<tr>
<td>2016-07-12T12:10:00Z</td>
<td>&quot;true&quot;</td>
<td>&quot;34&quot;</td>
<td>&quot;6&quot;</td>
<td>&quot;2&quot;</td>
<td>&quot;saarlouis&quot;</td>
<td>18573000000</td>
</tr>
<tr>
<td>2016-07-12T12:10:00Z</td>
<td>&quot;true&quot;</td>
<td>&quot;34&quot;</td>
<td>&quot;5&quot;</td>
<td>&quot;7&quot;</td>
<td>&quot;saarlouis&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.

[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

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.

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.

[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:
  - $D = 0 \rightarrow 0$
  - $D$ in $[-63,64] \rightarrow 10 + value (7$ bits)$
  - $D$ in $[-255,256] \rightarrow 110 + value (9$ bits)$
  - $D$ in $[-2047,2048] \rightarrow 1110 + value (12$ bits)$
  - else $\rightarrow 1111 + value (32$ bits)$
- 1 bit 96% of the time
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

[Pelkonen et al., 2015]
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

![Graph showing percent of memory used over time with a notable decrease around the 25-minute mark.](image)

[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]