ts-flint Documentation

Release 0+unknown

Two Sigma Open Source, LLC.

Sep 20, 2018

Contents

ts-flint is a collection of modules related to time series analysis for PySpark.

CHAPTER 1

Reading Data with FlintContext

Reading and Writing Data shows how to read data into a `ts.flint.TimeSeriesDataFrame`, which provides additional time-series aware functionality.

```
>>> prices = (flintContext.read
...           .range(begin, end)
...           .uri(uri))
```

CHAPTER 2

Manipulating and Analyzing Data

Manipulating and Analyzing Data describes the structure of *ts.flint.TimeSeriesDataFrame*, which is a time-series aware version of a `pyspark.sql.DataFrame`. Being time-series aware, it has optimized versions of some operations like joins, and also some new features like temporal joins. *ts.flint.summarizers* contains aggregation functions like EMAs.

```
>>> events.leftJoin(returns, tolerance='5d', key='id')
```

Contents:

2.1 Reading and Writing Data

Contents

A *ts.flint.FlintContext* is similar to a `pyspark.sql.SQLContext` in that it is the main entry point to reading Two Sigma data sources into a *ts.flint.TimeSeriesDataFrame*.

2.1.1 Converting other data sources to TimeSeriesDataFrame

You can also use a *ts.flint.FlintContext* to convert an existing `pandas.DataFrame` or `pyspark.sql.DataFrame` to a *ts.flint.TimeSeriesDataFrame* in order to take advantage of its time-aware functionality:

```
>>> df1 = flintContext.read.pandas(pd.read_csv(path))
>>> df2 = (flintContext.read
```

(continues on next page)

(continued from previous page)

```
...         .option('isSorted', False)
...         .dataframe(sqlContext.read.parquet(hdfs_path)))
```

2.1.2 Writing temporary data to HDFS

You can materialize a `pyspark.sql.DataFrame` to HDFS and read it back later on, to save data between sessions, or to cache the result of some preprocessing.

```
>>> import getpass
>>> filename = 'hdfs:///user/{}/filename.parquet'.format(getpass.getuser())
>>> df.write.parquet(filename)
```

The Apache Parquet format is a good fit for most tabular data sets that we work with in Flint.

To read a sequence of Parquet files, use the *`flintContext.read.parquet`* method. This method assumes the Parquet data is sorted by time. You can pass the `.option('isSorted', False)` option to the reader if the underlying data is not sorted on time:

```
>>> ts_df1 = flintContext.read.parquet(hdfs_path)  # assumes sorted by time
>>> ts_df2 = (flintContext.read
...             .option('isSorted', False)
...             .parquet(hdfs_path))  # this will sort by time before load
```

2.2 Manipulating and Analyzing Data

Contents

2.2.1 TimeSeriesDataFrame

A *`ts.flint.TimeSeriesDataFrame`* is a time-series aware version of a `pyspark.sql.DataFrame`. Being time-series aware, it has optimized versions of some operations like joins, and also some new features like temporal joins.

Like a normal `pyspark.sql.DataFrame`, a *`ts.flint.TimeSeriesDataFrame`* is a collection of `pyspark.sql.Row` objects, but which always must have a `time` column. The rows are always sorted by `time`, and the API affords special join/aggregation operations that take advantage of that temporal locality.

Note on Dataframes and Immutability

Note: All the methods on *ts.flint.TimeSeriesDataFrame* that appear to "add" something do not modify the target of the method. Instead, they return a new *ts.flint.TimeSeriesDataFrame* which shares most of its data with the original one. In fact, this is true of all Spark transformations.

Therefore, don't discard the results of one of these calls, assign it to a different variable. That way, you can always go back and refer to something before you transformed it.

Bad Example:

This completely discards the results of these operations. You'll simply get the wrong data.

```
>>> df.select('time', 'id', 'openPrice', 'closePrice')
>>> df.addWindows(windows.past_absolute_time('5d'), key='id')
```

Okay Example:

This is going to compute the right thing, but if you decide you want to try something without that window, you have to clear everything and start over. In addition, if you run a notebook cell like this multiple times, you'll add multiple layers of the same transformations to your dataframes.

```
>>> df = df.select('time', 'id', 'openPrice', 'closePrice')
>>> df = df.addWindows(windows.past_absolute_time('5d'), key='id')
```

Good Example:

This is the best way to work with *ts.flint.TimeSeriesDataFrame* and `pyspark.sql.DataFrame`. You can run this cell any number of times and you'll always get the same thing. Furthermore, you can now chain multiple things off price_df later, without re-reading raw_df.

```
>>> price_df = raw_df.select('time', 'id', 'openPrice', 'closePrice')
>>> window_df_7d = price_df.addWindows(windows.past_absolute_time('7d'), key='id')
>>> window_df_14d = price_df.addWindows(windows.past_absolute_time('14d'), key='id')
```

2.2.2 Summarizers

In Flint, we specify the summarizations we want to do in terms of answering two orthogonal questions:

- What aggregation/summarization function do you want to apply to a given set of rows?
- Which rows do you want to aggregate/summarize together?

The functions in *ts.flint.summarizers* are the way to specify what functions you want to apply. These are suitable for passing to functions like *ts.flint.TimeSeriesDataFrame.summarize()*, and *ts.flint.TimeSeriesDataFrame.summarizeCycles()*, which answer the second question, which rows should be aggregated together.

The Flint summarizer library augments the analysis capabilities of the normal `pyspark.sql.DataFrame` such as those available in `pyspark.sql.functions`.

2.3 Flint Cookbook

This page is a grab-bag of useful code snippets for Flint Python. We assume you can run Flint code in a Python process or Jupyter Notebook.

Please see python/docs/cookbook.ipynb

2.4 Reference

2.4.1 ts.flint

FlintContext

class ts.flint.**FlintContext**(*sqlContext*)

Main entry point for time-series Spark functionality.

A *FlintContext* can be used to create a *ts.flint.TimeSeriesDataFrame* from TS data sources. Those can then be manipulated with methods from that class, and by using summarizers from *ts.flint.summarizers*.

Parameters **sqlContext** – The pyspark.sql.SQLContext backing this *FlintContext*.

read

Entry point to access TS data. Returns a *readwriter.TSDataFrameReader* which can be used to read data sources.

class ts.flint.readwriter.**TSDataFrameReader**(*flintContext*)

Interface used to load a *TimeSeriesDataFrame*

This reader has builder methods that enable setting parameter values before calling a reader method. Multiple calls to the same builder method will take the last values set in the call chain.

Example usage:

```
>>> df = (flintContext.read
...       .range('20170101', '20170201', timezone='America/New_York')
...       .option('partitionGranularity', '1m')
...       .option('columns', ['x', 'y', 'z'])
...       .uri('...'))
```

clock(*name*, *frequency*, *offset=None*, *end_inclusive=True*)

Return TimeSeriesDataFrame using the specified clock.

The returned *TimeSeriesDataFrame* will only have a time column.

Example:

```
>>> (flintContext.read
...  .range('20170101', '20180101')
...  .clock('uniform', '30s'))
```

Supported options:

range (required) Set the inclusive-begin and **inclusive-end** time (by default). See documentation on the end_inclusive parameter for why this source is end-inclusive. Specified using *range()*.

Parameters

- **name** (*str*) – The name of the clock. Currently supported: uniform.
- **frequency** (*str*) – the time interval between rows, e.g., “1s”, “2m”, “3d” etc.
- **offset** (*str*) – the time to offset this clock from the begin time. Default: “0s”. Note that specifying an offset greater than the frequency is the same as specifying (offset % frequency).

- **end_inclusive** (*bool*) – If true, a clock tick will be created at the end time if the last tick falls at the end of the time range. This defaults to true because it is typically used with `summarizeInterval()` to handle values that are rounded up to the next clock tick. Set this parameter to False to be end-exclusive. Default: True.

dataframe(*df*, *begin=None*, *end=None*, ***, *timezone='UTC'*, *is_sorted=None*, *time_column=None*, *unit=None*)

Creates a `TimeSeriesDataFrame` from an existing `pyspark.sql.DataFrame`.

Note: The `pyspark.sql.DataFrame` must be sorted on the time column, otherwise specify is_sorted=False, or call `.option("isSorted", False)`.

Supported options:

range (optional) Set the inclusive-begin and exclusive-end time range. Begin and end are optional and either begin, end, or both begin and end can be omitted. If omitted, no boundary on time range will be set. Specified using *TSDataFrameReader.range()*.

isSorted (optional) Whether the input dataframe is sorted on *timeColumn*. Default: true.

timeUnit (optional) Time unit of the time column. Default: "ns"

timeColumn (optional) Column in parquet table that specifies time. Default: "time"

Parameters

- **df** (*pyspark.sql.DataFrame*) – the `pyspark.sql.DataFrame` to convert
- **begin** (*str*) – **Deprecated**. Inclusive. Supports most common date formats. Use `range(begin, end)` instead.
- **end** (*str*) – **Deprecated**. Exclusive. Supports most common date formats. Use `range(begin, end)` instead.
- **timezone** (*str*) – **Deprecated**. Timezone of the input time range. Only used if `begin` and `end` parameter are set. Default: 'UTC'. Use `range(begin, end, timezone="...")` instead.
- **is_sorted** (*bool*) – Default True. Whether the input data is already sorted (if already sorted, the conversion will be faster)
- **time_column** (*str*) – **Deprecated**. Column name used to sort rows. Default: "time". Use `option("timeColumn", column)` instead.
- **unit** (*str*) – **Deprecated**. Unit of time_column, can be (s,ms,us,ns). Default: "ns". Use `option("timeUnit", unit)` instead.

Returns a new `TimeSeriesDataFrame` containing the data in `df`

expand(*begin=None*, *end=None*)

Builder method to set time distance to expand the begin and end date range for the reader. This is useful to read a DataFrame that is on the right side of leftJoin or futureLeftJoin:

```
>>> begin, end = ("2017-01-01", "2018-01-01")
>>> tolerance = "21days"
>>> left = flintContext.read.range(begin, end).uri(...)
>>> right = flintContext.read.range(begin, end).expand(begin=tolerance).uri(..
↪.)
>>> joined = left.leftJoin(right, tolerance=tolerance)
```

If called multiple times, only the last call is effective.

Parameters

- **begin** (*str*) – The time distance to expand the begin time, e.g., "1hour", "7days"
- **end** (*str*) – The time distance to expand the end time, e.g., "1hour", "7days"

Return type *TSDataFrameReader*

option(*key*, *value*)

Set a key-value option for the data reader.

Refer to the documentation for data reader, e.g., `TSDataFrameReader.uri()`, *`TSDataFrameReader.parquet()`*, for a list of its supported options.

Example usage:

```
>>> (flintContext.read
...   .range('2017-01-01', '2017-02-01')
...   .option('numPartitions', 4)
...   .option('columns', ['x', 'y', 'z'])
...   .uri('...'))
```

Parameters

- **key** (*str*) – The string key, e.g., "numPartitions", "partitionGranularity"
- **value** – The value for the option. Any value that is not a string will be converted using `str(..)`. For keys that support multiple values, separate values with commas ',', List are converted into a string where the values are comma-separated.

Returns The *TSDataFrameReader*

Return type *TSDataFrameReader*

options(***options*)

Set one or more options using kwarg syntax. Keys with a value of None are ignored.

Example usage:

```
>>> (flintContext.read
...   .range('2017-01-01', '2017-02-01')
...   .options(numPartitions=4, columns=['x', 'y', 'z'])
...   .uri('...'))
```

Returns The *TSDataFrameReader*

Return type *TSDataFrameReader*

pandas(*df*, *schema=None*, ***, *is_sorted=None*, *time_column=None*, *unit=None*)

Creates a *TimeSeriesDataFrame* from an existing `pandas.DataFrame`.

Note: The `pandas.DataFrame` must be sorted on the time column, otherwise specify is_sorted=False, or call `.option("isSorted", False)`.

Supported options:

timeUnit (optional) Time unit of the time column. Default: "ns"

timeColumn (optional) Name of the time column. Default: "time"

Parameters

- **df** (*pandas.DataFrame*) – the pandas.DataFrame to convert
- **is_sorted** (*bool*) – Default True. Whether the input data is already sorted (if already sorted, the conversion will be faster)
- **time_column** (*str*) – **Deprecated**. Column name used to sort rows Default: "time". Use option("timeColumn", column) instead.
- **unit** (*str*) – **Deprecated**. Unit of time_column, can be (s,ms,us,ns) Default: "ns". Use option("timeUnit", unit) instead.

Returns a new TimeSeriesDataFrame containing the data in df

parquet(**paths*)

Create a TimeSeriesDataFrame from one or more paths containing parquet files.

Note: The Parquet files must be sorted on the time column, otherwise specify is_sorted=False, or call .option("isSorted", False).

Supported options:

range (optional) Set the inclusive-begin and exclusive-end time range. Begin and end are optional and either begin, end, or both begin and end can be omitted. If omitted, no boundary on time range will be set. Specified using *TSDataFrameReader.range()*.

isSorted (optional) Whether the input dataframe is sorted on *timeColumn*. Default: true.

timeUnit (optional) Time unit of the time column. Default: "ns"

timeColumn (optional) Column in parquet table that specifies time. Default: "time"

columns* (optional) A subset of columns to retain from the parquet table. Specifying a subset of columns can greatly improve performance by 10x compared to reading all columns in a set of parquet files. Default: all columns are retained.

Parameters **paths** (*str*) – one or more paths / URIs containing parquet files

Returns a new TimeSeriesDataFrame

Return type *TimeSeriesDataFrame*

range(*begin=None*, *end=None*, *timezone='UTC'*)

Builder method to set the begin and end date range for the reader. Dates specified without a time have their time set to midnight in the timezone specified in the tz parameter. Default: UTC.

Supported date specifications for begin and end:

- A string or object supported by pandas.to_datetime() e.g., "2017-01-01", "20170101", "20170101 10:00" "2017-07-14T10:00:00-05:00"
- A YYYYMMDD integer, e.g., 20170714
- A datetime.datetime
- A pandas.Timestamp

Note: The time range is begin-inclusive and *end*-exclusive.

`end` is exclusive, taking the last nanoseconds before the specified datetime. For example, if `end` is "2017-02-01" then the reader will read data up to and including "2017-01-31 23:59:59.999999999" but excluding "2017-02-01 00:00".

Examples for specifying a `begin` time of "2017-01-01 00:00 UTC" inclusive and `end` time of "2017-02-01 00:00 UTC" exclusive:

```
>>> flintContext.read.range('2017-01-01', '2017-02-01').uri('...')
...
>>> flintContext.read.range('20161231 19:00',
...                         '20170131 19:00',
...                         'America/New_York').uri('...')
...
>>> flintContext.read.range(20170101, 20170201).uri('...')
...
>>> from datetime import datetime
... flintContext.read.range(datetime(2017, 1, 1, 0, 0),
...                         datetime(2017, 2, 1, 0, 0)).uri('...')
```

Parameters

- **begin** (str, int, `pandas.Timestamp`) – The inclusive begin date of the date range.
- **end** (str, int, `pandas.Timestamp`) – The exclusive end date of date range.
- **tz** (*str*) – the timezone to localize the begin and end dates if the provided dates are timezone-naive. Default: UTC.

Returns The *TSDataFrameReader*

See `pandas.to_datetime()` for examples of supported formats for strings

Return type *TSDataFrameReader*

TimeSeriesDataFrame

class `ts.flint.`**TimeSeriesDataFrame**(*df*, *sql_ctx*, ***, *time_column='time'*, *is_sorted=True*, *unit='ns'*, *tsrdd_part_info=None*)

A `pyspark.sql.DataFrame` backed by time-ordered rows, with additional time-series functionality.

A *TimeSeriesDataFrame* supports a subset of `pyspark.sql.DataFrame` operations: *cache()*, *count()*, *drop()*, *dropna()*, *filter()*, *persist()*, *select()*, *unpersist()*, *withColumn()*, *withColumnRenamed()*

as well as time series operations:

leftJoin(), ***futureLeftJoin()*** time-aware ("asof") joins

addWindows() time-aware windowing operations, in concert with *windows*

addColumnsForCycle(), ***groupByCycle()*** processing rows with the same timestamp

groupByInterval() processing rows within the same interval

summarize(), ***addSummaryColumns()***, ***summarizeCycles()***, ***summarizeIntervals()***, ***summarizeWindows***
data summarization, in concert with *summarizers*.

A *TimeSeriesDataFrame* can be created by reading a Two Sigma URI with `TSDataFrameReader.uri()`, or from a pandas or Spark DataFrame.

Warning: Pay special attention to *Note on Dataframes and Immutability*.

See also:

Class ***ts.flint.FlintContext*** Entry point for reading data in to a *TimeSeriesDataFrame*.

Class **pyspark.sql.DataFrame** A *TimeSeriesDataFrame* also has most of the functionality of a normal PySpark DataFrame.

Inherited members:

cache()
Same as pyspark.sql.DataFrame.cache()

collect()
Same as pyspark.sql.DataFrame.collect()

drop(*col*)
Same as pyspark.sql.DataFrame.drop()

dropna(*col*)
Same as pyspark.sql.DataFrame.dropna()

filter(*col*)
Same as pyspark.sql.DataFrame.filter()

persist(*storageLevel*)
Same as pyspark.sql.DataFrame.persist()

select(**cols*)
Select columns from an existing *TimeSeriesDataFrame*

Note: Column names to be selected must contain "time"

Example:

```
openPrice = price.select("time", "id", "openPrice")
```

Parameters **cols** (*list(str)*) – list of column names

Returns a new *TimeSeriesDataFrame* with the selected columns

Return type *TimeSeriesDataFrame*

unpersist(*blocking*)
Same as pyspark.sql.DataFrame.unpersist()

withColumn(*colName*, *col*)
Adds a column or replaces the existing column that has the same name. This method invokes pyspark.sql.DataFrame.withColumn(), but only allows pyspark.sql.Column expressions that preserve order. Currently, only a subset of column expressions under pyspark.sql.functions are supported.

Supported expressions:

Arithmetic expression: +,-,*,/,log,log2,log10,log1p,pow,exp,expm1,sqrt,abs,rand,randn,rint,round,ceil,signum,factorial

String expression: lower,upper,ltrim,rtrim,trim,lpad,rpad,reverse,split,substring,substring_index,concat,concat_ws conv,base64,format_number,format_string,hex,translate

Condition expression: when, nanvl

Boolean expression: isnan, isnull, >, <, ==, >=, <=

Example:

```
>>> priceWithNewCol = price.withColumn("newCol",
...             (price.closePrice + price.openPrice) / 2)
```

```
>>> priceWithNewCol = price.withColumn("newCol",
...             when(price.closePrice > price.openPrice, 0).otherwise(1))
```

If these column expressions don't do the thing you want, you can use `pyspark.sql.functions.udf()` (user defined function, or UDF).

Note: UDFs are much slower than column expressions and you should ONLY use UDFs when the computation cannot expressed using column expressions.

Example:

```
>>> @ts.flint.udf(DoubleType())
... def movingAverage(window):
...     nrows = len(window)
...     if nrows == 0:
...         return 0.0
...     return sum(row.closePrice for row in window) / nrows
...
>>> priceWithMA = (price
...                .addWindows(windows.past_absolute_time("14days"), "id")
...                .withColumn("movingAverage", movingAverage(col("window"))))
```

Parameters

- **colName** (*str*) – name of the new column
- **col** (`pyspark.sql.Column`) – column expression to compute values in the new column

Returns a new *TimeSeriesDataFrame* with the new column

Return type *TimeSeriesDataFrame*

withColumnRenamed(*existing*, *new*)
Same as `pyspark.sql.DataFrame.withColumnRenamed()`

Time-series specific members:

DEFAULT_TIME_COLUMN = 'time'
The name of the column assumed to contain timestamps, and used for ordering rows.

DEFAULT_UNIT = 'ns'
The units of the timestamps present in *DEFAULT_TIME_COLUMN*.

Acceptable values are: `'s'`, `'ms'`, `'us'`, `'ns'`.

addColumnsForCycle(*columns*, *, *key=None*)

Adds columns to each cycle by computing over data in the cycle.

The columns are specified as a column spec, which is a `dict`. Each entry can be either:

1. column name to UDF column. A UDF column is defined by *ts.flint.functions.udf()* with a python function, a return type and a list of input columns. The map entry can be one of the following:

   (a) str -> udf

   This will add a single column. The input args to the python function are `pandas.Series` or `pandas.DataFrame`. The return value of the function must be `pandas.Series`. The returned `pandas.Series` must have the same length as inputs. The `returnType` argument of the udf object must be a single `DataType` describing the type of the added column.

   (b) tuple(str) -> udf

   This will add multiple columns. The input args to the python function are `pandas.Series` or `pandas.DataFrame`. The return value of the function must be `pandas.Series`. The returned *pandas.Series* must have the same length as inputs. The `returnType` argument of the udf object must be a single `DataType` describing the types of the added columns.

   The cardinality of the column names, return data types and returned `pandas.Series` must match, i.e, if you are adding two columns, then the column names must be a tuple of two strings, the return type must be two data types, and the python must return a tuple of two `pandas.Series`.

User-defined function examples:

Use user-defined functions

```
>>> from ts.flint.functions import udf
>>>
>>> # v is a pandas.Series of double
>>> @udf(DoubleType())
... def pct_rank(v):
...     return v.rank(pct=True)
>>>
>>> df.addColumnsForCycle({
...      'rank': pct_rank(df.v)
... })
```

Add multiple-columns

```
>>> from ts.flint.functions import udf
>>>
>>> # v is a pandas.Series of double
>>> @udf((DoubleType(), DoubleType()))
... def ranks(v):
...     return v.rank(), v.rank(pct=True)
>>>
>>> df.addColumnsForCycle({
...      ('rank', 'rank_pct'): ranks(df.v)
... })
```

Parameters

- **columns** (*collections.Mapping*) – a column spec
- **key** (*str, list of str*) – Optional. One or multiple column names to use as the grouping key

Returns a new dataframe with the columns added

Return type *TimeSeriesDataFrame*

addSummaryColumns(*summarizer*, *key=None*)

Computes the running aggregate statistics of a table. For a given row R, the new columns will be the summarization of all rows before R (including R).

Example:

```
>>> # Add row number to each row
>>> dfWithRowNum = df.addSummaryColumns(summarizers.count())
```

Parameters

- **summarizer** – A summarizer or a list of summarizers that will calculate results for the new columns. Available summarizers can be found in *summarizers*.
- **key** (*str, list of str*) – One or multiple column names to use as the grouping key

Returns a new dataframe with the summarization columns added

Return type *TimeSeriesDataFrame*

addWindows(*window*, *key=None*)

Add a window column that contains a list of rows which can later be accessed in computations, such as *withColumn()*.

Example:

```
>>> dfWithWindow = df.addWindows(windows.past_absolute_time("365days"))
```

Parameters

- **window** – A window that specifies which rows to add to the new column. Lists of windows can be found in *windows*.
- **key** (*str, list of str*) – Optional. One or multiple column names to use as the grouping key

Returns a new dataframe with the window columns

Return type *TimeSeriesDataFrame*

count()

Counts the number of rows in the dataframe

Returns the number of rows in the dataframe

Return type int

futureLeftJoin(*right*, ***, *tolerance=None*, *key=None*, *left_alias=None*, *right_alias=None*, *strict_lookahead=False*)

Left join this dataframe with a right dataframe using inexact timestamp matches. For each row in the left dataframe, append the closest row from the right table at or after the same time. Similar to *leftJoin()* except it joins with future rows when no matching timestamps are found.

Example:

```
>>> leftdf.futureLeftJoin(rightdf, tolerance='100ns', key='id')
>>> leftdf.futureLeftJoin(rightdf, tolerance=pandas.
↪Timedelta(nanoseconds=100), key='id')
>>> leftdf.futureLeftJoin(rightdf, tolerance=pandas.
↪Timedelta(nanoseconds=100), key=['id', 'industryGroup'])
```

Parameters

- **right** (*TimeSeriesDataFrame*) – A dataframe to join
- **tolerance** (`pandas.Timedelta` or str) – The closest row in the future from the right dataframe will only be appended if it was within the specified time of the row from left dataframe. If a str is specified, it must be parsable by `pandas.Timedelta`. A tolerance of 0 means only rows with exact timestamp match will be joined.
- **key** (*str, list of str*) – Optional. One or multiple column names to use as the grouping key
- **left_alias** (*str*) – Optional. The prefix for columns from left in the output dataframe.
- **right_alias** (*str*) – Optional. The prefix for columns from right in the output dataframe.
- **strict_lookahead** (*bool*) – Optional. Default False. If True, rows in the left dataframe will only be joined with rows in the right dataframe that have strictly larger timestamps.

Returns a new dataframe that results from the join

Return type *TimeSeriesDataFrame*

groupByCycle(*key=None*)

Groups rows that have the same timestamp. The output dataframe contains a "rows" column which contains a list of rows of same timestamps. The column can later be accessed in computations, such as *withColumn()*.

Example:

```
>>> @ts.spark.udf(DoubleType())
... def averagePrice(cycle):
...     nrows = len(cycle)
...     if nrows == 0:
...         return 0.0
...     return sum(row.closePrice for row in window) / nrows
...
>>> averagePriceDF = (price
...                   .groupByCycle()
...                   .withColumn("averagePrice", averagePrice(col("rows"))))
```

Parameters **key** (*str, list of str*) – Optional. One or multiple column names to use as the grouping key

Returns a new dataframe with list of rows of the same timestamp

Return type *TimeSeriesDataFrame*

groupByInterval(*clock*, *key=None*, *inclusion='begin'*, *rounding='end'*)

Groups rows within the intervals specified by a clock dataframe. For each adjacent pair of rows in the

clock dataframe, rows from the dataframe that have time stamps between the pair are grouped. The output dataframe will have the first timestamp of each pair as the time column. The output dataframe contains a "rows" column which can be later accessed in computations, such as *withColumn()*.

Example:

```
>>> clock = clocks.uniform(sqlContext, frequency="1day", offset="0ns", begin_
↪date_time="2016-01-01", end_date_time="2017-01-01")
>>> intervalized = price.groupByInterval(clock)
```

Parameters

- **clock** (*TimeSeriesDataFrame*) – A dataframe used to determine the intervals
- **key** (*str, list of str*) – Optional. One or multiple column names to use as the grouping key
- **inclusion** (*str*) – Defines the shape of the intervals, i.e, whether intervals are [begin, end) or (begin, end]. "begin" causes rows that are at the exact beginning of an interval to be included and rows that fall on the exact end to be excluded, as represented by the interval [begin, end). "end" causes rows that are at the exact beginning of an interval to be excluded and rows that fall on the exact end to be included, as represented by the interval (begin, end]. Defaults to "begin".
- **rounding** (*str*) – Determines how timestamps of input rows are rounded to timestamps of intervals. "begin" causes the input rows to be rounded to the beginning timestamp of an interval. "end" causes the input rows to be rounded to the ending timestamp of an interval. Defaults to "end".

Returns a new dataframe with list of rows of the same interval

Return type *TimeSeriesDataFrame*

leftJoin(*right*, *, *tolerance=None*, *key=None*, *left_alias=None*, *right_alias=None*)

Left join this dataframe with a right dataframe using inexact timestamp matches. For each row in the left dataframe, append the most recent row from the right table at or before the same time.

Example:

```
>>> leftdf.leftJoin(rightdf, tolerance='100ns', key='id')
>>> leftdf.leftJoin(rightdf, tolerance=pandas.Timedelta(nanoseconds=100), key=
↪'id')
>>> leftdf.leftJoin(rightdf, tolerance=pandas.Timedelta(nanoseconds=100),␣
↪key=['id', 'industryGroup'])
```

Parameters

- **right** (*TimeSeriesDataFrame*) – A dataframe to join
- **tolerance** (`pandas.Timedelta` or str) – The most recent row from the right dataframe will only be appended if it was within the specified time of the row from left dataframe. If a str is specified, it must be parsable by `pandas.Timedelta`. A tolerance of 0 means only rows with exact timestamp match will be joined.
- **key** (*str, list of str*) – Optional. One or multiple column names to use as the grouping key
- **left_alias** (*str*) – Optional. The prefix for columns from left in the output dataframe.

- **right_alias** (*str*) – Optional. The prefix for columns from right in the output dataframe.

Returns a new dataframe that results from the join

Return type *TimeSeriesDataFrame*

merge(*other*)

Merge this dataframe and the other dataframe with the same schema. The merged dataframe includes all rows from each in temporal order. If there is a timestamp ties, the rows in this dataframe will be returned earlier than those from the other dataframe.

Example:

```
>>> thisdf.merge(otherdf)
```

Parameters **other** (*TimeSeriesDataFrame*) – The other dataframe to merge. It must have the same schema as this dataframe.

Returns a new dataframe that results from the merge

Return type *TimeSeriesDataFrame*

preview(*n=10*)

Return the first n rows of the *TimeSeriesDataFrame* as `pandas.DataFrame`

This is only available if Pandas is installed and available.

The time column will be converted to timestamp type.

Parameters **n** – number of rows to return. Default is 10.

shiftTime(*shift*, *, *backwards=False*)

Returns a :class: *TimeSeriesDataFrame* by shifting all timestamps by given amount.

When time type is timestamp: If shift forward amount is less than 1 microsecond, then this is a no op. If shift backward amount if less than 1 microsecond, then this will shift back 1 microsecond.

Example:

```
>>> tsdf.shiftTime('100s')
>>> tsdf.shiftTime(pandas.Timedelta(seconds=100))
>>> tsdf.shiftTime(windows.futureTradingTime('1day', 'US'))
```

Parameters

- **shift** – Amount to shift the dataframe time column, shift can be a `pandas.Timedelta` or a string that can be formatted by `pandas.Timedelta` or a `window`.
- **backwards** – Shift time backwards (defaults to False). Ignored when shift is a `window`.

Returns a new *TimeSeriesDataFrame*

summarize(*summarizer*, *key=None*)

Computes aggregate statistics of a table.

Example:

```
>>> # calcuates the weighted mean of return and t-statistic
>>> result = df.summarize(summarizers.weighted_mean("return", "volume"), key=
↪"id")
>>> result = df.summarize(summarizers.weighted_mean("return", "volume"), key=[
↪"id", "industryGroup"])
```

Parameters

- **summarizer** – A summarizer or a list of summarizers that will calculate results for the new columns. Available summarizers can be found in *summarizers*.
- **key** (*str, list of str*) – Optional. One or multiple column names to use as the grouping key

Returns a new dataframe with summarization columns

Return type *TimeSeriesDataFrame*

summarizeCycles(*summarizer*, *key=None*)

Computes aggregate statistics of rows that share a timestamp using a summarizer spec.

A summarizer spec can be either:

1. A summarizer or a list of summarizers. Available summarizers can be found in *summarizers*.
2. A map from column names to columnar udf objects. A columnar udf object is defined by *ts.flint.functions.udf()* with a python function, a return type and a list of input columns. Each map entry can be one of the following:

   (a) str -> udf

   This will add a single column. The python function must return a single scalar value, which will be the value for the new column. The `returnType` argument of the udf object must be a single `DataType`.

   (b) tuple(str) -> udf

   This will add multiple columns. The python function must return a tuple of scalar values. The `returnType` argument of the udf object must be a tuple of `DataType`. The cardinality of the column names, return data types and return values must match, i.e, if you are adding two columns, then the column names must be a tuple of two strings, the return type must be two data types, and the python must return a tuple of two scalar values.

Examples:

Use built-in summarizers

```
>>> df.summarizeCycles(summarizers.mean('v'))
```

```
>>> df.summarizeCycles([summarizers.mean('v'), summarizers.stddev('v')])
```

Use user-defined functions (UDFs):

```
>>> from ts.flint.functions import udf
>>> @udf(DoubleType())
... def mean(v):
...     return v.mean()
>>>
>>> @udf(DoubleType())
... def std(v):
```

(continues on next page)

(continued from previous page)

```
...     return v.std()
>>>
>>> df.summarizeCycles({
...     'mean': mean(df['v']),
...     'std': std(df['v'])
... })
```

Use a OrderedDict to specify output column order

```
>>> from collections import OrderedDict
>>> df.summarizeCycles(OrderedDict([
...     ('mean', mean(df['v'])),
...     ('std', std(df['v'])),
... ]))
```

Return multiple columns from a single udf as a tuple

```
>>> @udf((DoubleType(), DoubleType()))
>>> def mean_and_std(v):
...     return (v.mean(), v.std())
>>> df.summarizeCycles({
...     ('mean', 'std'): mean_and_std(df['v']),
... })
```

Use other python libraries in udf

```
>>> from statsmodels.stats.weightstats import DescrStatsW
>>> @udf(DoubleType())
... def weighted_mean(v, w):
...     return DescrStatsW(v, w).mean
>>>
>>> df.summarizeCycles({
...     'wm': weighted_mean(df['v'], df['w'])
... })
```

Use `pandas.DataFrame` as input to udf

```
>>> @udf(DoubleType())
... def weighted_mean(cycle_df):
...     return DescrStatsW(cycle_df.v, cycle_df.w).mean
>>>
>>> df.summarizeCycles({
...     'wm': weighted_mean(df[['v', 'w']])
... })
```

Parameters

- **summarizer** – A summarizer spec. See above for the allowed types of objects.
- **key** (*str, list of str*) – Optional. One or multiple column names to use as the grouping key

Returns a new dataframe with summarization columns

Return type *TimeSeriesDataFrame*

See also:

ts.flint.functions.udf()

summarizeIntervals(*clock*, *summarizer*, *key=None*, *inclusion='begin'*, *rounding='end'*)

Computes aggregate statistics of rows within the same interval using a summarizer spec.

A summarizer spec can be either:

1. A summarizer or a list of summarizers. Available summarizers can be found in *summarizers*.
2. A map from column names to columnar udf objects. A columnar udf object is defined by *ts.flint.functions.udf()* with a python function, a return type and a list of input columns. Each map entry can be one of the following:

   (a) str -> udf

   This will add a single column. The python function must return a single scalar value, which will be the value for the new column. The `returnType` argument of the udf object must be a single `DataType`.

   (b) tuple(str) -> udf

   This will add multiple columns. The python function must return a tuple of scalar values. The `returnType` argument of the udf object must be a tuple of `DataType`. The cardinality of the column names, return data types and return values must match, i.e, if you are adding two columns, then the column names must be a tuple of two strings, the return type must be two data types, and the python must return a tuple of two scalar values.

Examples:

Create a uniform clock

```
>>> from ts.flint import clocks
>>> clock = clocks.uniform(sqlContext, '1day')
```

Use built-in summarizers

```
>>> df.summarizeIntervals(clock, summarizers.mean('v'))
```

```
>>> df.summarizeIntervals(clock, [summarizers.mean('v'), summarizers.stddev('v
↪')])
```

Use user-defined functions (UDFs):

```
>>> from ts.flint.functions import udf
>>> @udf(DoubleType())
... def mean(v):
...     return v.mean()
>>>
>>> @udf(DoubleType())
... def std(v):
...     return v.std()
>>>
>>> df.summarizeIntervals(
...     clock,
...     {
...         'mean': mean(df['v']),
...         'std': std(df['v'])
...     }
... )
```

Use a OrderedDict to specify output column order

```
>>> from collections import OrderedDict
>>> df.summarizeIntervals(
...     clock,
...     OrderedDict([
...         ('mean', mean(df['v'])),
...         ('std', std(df['v'])),
...     ])
... )
```

Return multiple columns from a single udf as a tuple

```
>>> @udf((DoubleType(), DoubleType()))
>>> def mean_and_std(v):
...     return (v.mean(), v.std())
>>>
>>> df.summarizeIntervals(
...     clock,
...     {
...         ('mean', 'std'): mean_and_std(df['v']),
...     }
... )
```

Use `pandas.DataFrame` as input to udf

```
>>> @udf(DoubleType())
... def weighted_mean(cycle_df):
...     return numpy.average(cycle_df.v, weights=cycle_df.w)
>>>
>>> df.summarizeIntervals(
...    clock,
...    {
...       'wm': weighted_mean(df[['v', 'w']])
...    }
... )
```

Parameters

- **clock** (*TimeSeriesDataFrame*) – A *TimeSeriesDataFrame* used to determine the intervals
- **summarizer** – A summarizer spec. See above for the allowed types of objects.
- **key** (*str, list of str*) – Optional. One or multiple column names to use as the grouping key
- **inclusion** (*str*) – Defines the shape of the intervals, i.e, whether intervals are [begin, end) or (begin, end]. "begin" causes rows that are at the exact beginning of an interval to be included and rows that fall on the exact end to be excluded, as represented by the interval [begin, end). "end" causes rows that are at the exact beginning of an interval to be excluded and rows that fall on the exact end to be included, as represented by the interval (begin, end]. Defaults to "begin".
- **rounding** (*str*) – Determines how timestamps of input rows are rounded to timestamps of intervals. "begin" causes the input rows to be rounded to the beginning timestamp of an interval. "end" causes the input rows to be rounded to the ending timestamp of an interval. Defaults to "end".

Returns a new dataframe with summarization columns

Return type *TimeSeriesDataFrame*

See also:

ts.flint.functions.udf()

summarizeState(*summarizer*, *key=None*)

Undocumented function for the bravest.

Returns a Java map from key to summarize state (also Java object). This function can be changed/removed/broken without notice.

Use at your own risk.

summarizeWindows(*window*, *summarizer*, *key=None*)

Computes aggregate statistics of rows in windows using a window spec and a summarizer spec.

A window spec can be created using one the functions in *windows*.

A summarizer spec can be either:

1. A summarizer or a list of summarizers. Available summarizers can be found in *summarizers*.
2. A map from column names to UDF columns. A UDF column is defined by *ts.flint.functions.udf()* with a python function, a return type and a list of input columns. Each map entry can be one of the following:
   (a) str -> udf

   This will add a single column. The python function must return a single scalar value, which will be the value for the new column. The `returnType` argument of the udf object must be a single `DataType`.

   (b) tuple(str) -> udf

   This will add multiple columns. The python function must return a tuple of scalar values. The `returnType` argument of the udf object must be a tuple of `DataType`. The cardinality of the column names, return data types and return values must match, i.e, if you are adding two columns, then the column names must be a tuple of two strings, the return type must be two data types, and the python must return a tuple of two *pandas.Series*.

Built-in summarizer examples:

Use built-in summarizers

```
>>> # calculates rolling weighted mean of return for each id
>>> result = df.summarizeWindows(
...     windows.past_absolute_time("7days"),
...     summarizers.weighted_mean("return", "volume"),
...     key="id"
... )
```

User-defined function examples:

Use user-defined functions

```
>>> from ts.flint.functions import udf
>>>
>>> # v is a pandas.Series of double
>>> @udf(DoubleType())
... def mean(v):
...     return v.mean()
>>>
```

(continues on next page)

(continued from previous page)

```
>>> # v is a pandas.Series of double
>>> @udf(DoubleType())
... def std(v):
...     return v.std()
>>>
>>> df.summarizeWindows(
...     windows.past_absolute_time('7days'),
...     {
...       'mean': mean(df['v']),
...       'std': std(df['v'])
...     },
...     key='id'
... )
```

Use an OrderedDict to specify output column order

```
>>> # v is a pandas.Series of double
>>> from ts.flint.functions import udf
>>> @udf(DoubleType())
... def mean(v):
...     return v.mean()
>>>
>>> # v is a pandas.Series of double
>>> @udf(DoubleType())
... def std(v):
...     return v.std()
>>>
>>> udfs = OrderedDict([
...     ('mean', mean(df['v'])),
...     ('std', std(df['v']))
... ])
>>>
>>> df.summarizeWindows(
...     windows.past_absolute_time('7days'),
...     udfs,
...     key='id'
... )
```

Return multiple columns from a single UDF

```
>>> # v is a pandas.Series of double
>>> @udf((DoubleType(), DoubleType()))
>>> def mean_and_std(v):
...     return v.mean(), v.std()
>>>
>>> df.summarizeWindows(
...     windows.past_absolute_time('7days'),
...     {
...       ('mean', 'std'): mean_and_std(df['v'])
...     },
...     key='id'
... )
```

Use multiple input columns

```
>>> from ts.flint.functions import udf
>>> # window is a pandas.DataFrame that has two columns - v and w
```

(continues on next page)

(continued from previous page)

```
>>> @udf(DoubleType())
... def weighted_mean(window):
...     return np.average(window.v, weights=window.w)
>>>
>>> df.summarizeWindows(
...    windows.past_absolute_time('7days'),
...    {
...      'weighted_mean': weighted_mean(df[['v', 'w']])
...    },
...    key='id'
... )
```

Use numpy user-defined function to compute rank:

```
>>> from scipy import stats
>>> @udf(DoubleType(), arg_type='numpy')
>>> def rank_np(v):
>>>     # v is a numpy.ndarray
>>>     return stats.percentileofscore(v, v[-1], kind='rank') / 100.0
>>>
>>> df.summarizeWindows(
...     windows.past_absolute_time('7days'),
...     {
...       'rank': rank_np(df['v']),
...     },
...     key='id'
... )
```

Use numpy user-defined function to compute weighted mean:

```
>>> @udf(DoubleType(), arg_type='numpy')
>>> def weighted_mean_np(window):
>>>     # window is a list of numpy.ndarray
>>>     return np.average(window[0], weights=window[1])
>>> df.summarizeWindows(
...    windows.past_absolute_time('7days'),
...    {
...       'weighted_mean': weighted_mean_np(df[['v', 'w']])
...    },
...    key='id'
... )
```

Parameters

- **window** – A window that specifies which rows to add to the new column. Lists of windows can be found in *windows*.
- **summarizer** – A summarizer spec
- **key** (*str, list of str*) – Optional. One or multiple column names to use as the grouping key.

Returns a new dataframe with summarization columns

Return type *TimeSeriesDataFrame*

timeSeriesRDD

Returns a Scala TimeSeriesRDD object

Returns `py4j.java_gateway.JavaObject` (com.twosigma.flint.timeseries.TimeSeriesRDD)

timestamp_df()
Returns a :class: *pyspark.sql.DataFrame* by casting the time column (Long) to a timestamp

Returns a new *TimeSeriesDataFrame*

Summarizers

This module contains summarizer routines suitable as arguments to:

- *TimeSeriesDataFrame.summarizeCycles()*
- *TimeSeriesDataFrame.summarizeIntervals()*
- *TimeSeriesDataFrame.summarizeWindows()*
- *TimeSeriesDataFrame.summarize()*
- *TimeSeriesDataFrame.addSummaryColumns()*

Example:

```
>>> from ts.flint import summarizers
>>> prices.summarize(summarizers.correlation('openPrice', 'closePrice'), key='id')
```

class ts.flint.summarizers.**SummarizerFactory**(*func*, **args*)
SummarizerFactory represents an intended summarization that will be instantiated later when we have access to a SparkContext.

Typical usage is that a user constructs one of these (using the summarization functions in this module), then passes it to one of the summarize methods of `TimeSeriesDataFrame`, where we have a SparkContext. In those methods, we have this factory construct the actual summarizer the user wanted.

prefix(*prefix*)
Adds prefix to the column names of output schema. All columns names will be prepended as format '<prefix>_<column>'.

ts.flint.summarizers.**correlation**(*cols*, *other=None*)
Computes pairwise correlation of columns.

Example:

```
>>> from ts.flint import summarizers
>>> # Compute correlation between columns 'col1' and 'col2'
... prices.summarize(summarizers.correlation('col1', 'col2'))
```

```
>>> # Compute pairwise correlations for columns 'col1', 'col2' and 'col3',
... prices.summarize(summarizers.correlation(['co1', 'col2', 'col3']))
```

```
>>> # Compute only correlations for pairs of columns
... # ('col1', 'col3'), ('col1', 'col4'), ('col1', 'col5') and
... # ('col2', 'col3'), ('col2', 'col4'), ('col1', 'col5')
... prices.summarize(summarizers.correlation(['col1', 'col2'], ['col3', 'col4
↪', 'col5']))
```

Adds columns:

<col1>_<col2>_correlation (*float*) The correlation between columns `<col1>` and `<col2>`.

<col1>_<col2>_correlationTStat (*float*) The t-stats of the correlation coefficient between the columns.

Parameters

- **cols** (*str, list of str*) – names of the columns to be summarized. If the `other` is None, this summarizer will compute pairwise correlation and t-stats for only columns in `cols`.
- **other** (*str, list of str, None*) – other names of columns to be summarized. If it is None, this summarizer will compute pairwise correlation and t-stats for columns only in `cols`; otherwise, it will compute all possible pairs of columns where the left is one of columns in `cols` and the right is one of columns in `other`. By default, it is None.

ts.flint.summarizers.**count**()

Counts the number of rows.

Adds columns:

count (*int*) The number of rows.

ts.flint.summarizers.**covariance**(*x_column*, *y_column*)

Computes covariance of two columns.

Adds columns:

<x_column>_<y_column>_covariance (*float*) covariance of x_column and y_column

Parameters

- **x_column** (*str*) – name of column X
- **y_column** (*str*) – name of column y

ts.flint.summarizers.**dot_product**(*x_column*, *y_column*)

Computes the dot product of two columns.

Adds columns:

<x_column>_<y_column>_dotProduct (*float*) the dot product of x_column and y_column

Parameters

- **x_column** (*str*) – name of column X
- **y_column** (*str*) – name of column y

ts.flint.summarizers.**ema_halflife**(*column*, *halflife_duration*, *time_column='time'*, *interpolation='previous'*, *convention='legacy'*)

Calculates the exponential moving average given a specified half life. Supports the same default behaviors as the previous in-house implementation.

See doc/ema.md for details on different EMA implementations.

Adds columns:

<column>_ema (*float*) The exponential moving average of the column

Parameters

- **column** (*str*) – name of the column to be summarized
- **halflife_duration** (*str*) – string representing duration of the half life
- **time_column** (*str*) – Name of the time column.

- **interpolation** – the interpolation type of the ema - options are 'previous', 'current', and 'linear'
- **convention** – the convention used to compute the final output - options are 'convolution', 'core', and 'legacy'

Type interpolation: str

ts.flint.summarizers.**ewma**(*column*, *alpha=0.05*, *time_column='time'*, *duration_per_period='1d'*, *convention='legacy'*)

Calculate the exponential weighted moving average over a column.

It maintains a primary EMA for the series `x_1, x_2, ...` as well as an auxiliary EMA for the series `1.0, 1.0, ...`. The primary EMA `EMA_p(X)` keeps track of the sum of the weighted series, whereas the auxiliary EMA `EMA_a(X)` keeps track of the sum of the weights.

The weight of i-th value `decay(t_i, t_n)` is: `decay(t_i, t_n) = exp(ln(1 - alpha) * (t_n - t_i) / duration_per_period)`

If duration_per_period is "constant", the decay will be defined as `decay(t_i, t_n) = exp(ln(1 - alpha) * (n - i))`

Finally, if the convention is "core", we will the following EMA(X) as output where `(EMA(X))_i = (EMA_p(X))_i / (EMA_a(X))_i`

However, if the convention is "legacy", we will simply return EMA(X) such that `(EMA(X))_i = (EMA_p(X))_i`

See doc/ema.md for details on different EMA implementations.

Adds columns:

<column>_ewma (*float*) The exponential weighted moving average of the column

Parameters

- **column** (*str*) – name of the column to be summarized
- **alpha** (*float*) – parameter setting the decay rate of the average
- **time_column** (*str*) – Name of the time column.
- **duration_per_period** (*str*) – duration per period. The option could be "constant" or any string that specifies duration like "1d", "1h", "15m" etc. If it is "constant", it will assume that the number of periods between rows is constant (c = 1); otherwise, it will use the duration to calculate how many periods should be considered to have passed between any two given timestamps.
- **convention** – the convention used to compute the final output - options are 'core' and 'legacy'.

ts.flint.summarizers.**geometric_mean**(*column*)

Computes the geometric mean of a column.

Adds columns:

<column>_geometricMean (*float*) The geometric mean of the column

Parameters **column** (*str*) – name of the column to be summarized

ts.flint.summarizers.**kurtosis**(*column*)

Computes the excess kurtosis of a column.

Adds columns:

<column>_kurtosis (*float*) The excess kurtosis of the column

Parameters **column** (*str*) – name of the column to be summarized

ts.flint.summarizers.**linear_regression**(*y_column*, *x_columns*, *weight_column=None*, ***, *use_intercept=True*, *ignore_constants=False*, *constant_error_bound=1e-12*)

Computes a weighted multiple linear regression of the values in several columns against values in another column, using values from yet another column as the weights.

Note: Constant columns When there is at least one constant variable in x_columns with intercept = True or there are multiple constant variables in x_columns, a regression will fail unless ignore_constants is True.

When ignore_constants is True, the scalar fields of regression result are the same as if the constant variables are not included in x_columns. The output beta, tStat, stdErr still have the same dimension as x_columns. Entries corresponding to constant variables will have 0.0 for beta and stdErr; and NaN for tStat.

When ignore_constants is False and x_columns includes constant variables, the regression will output NaN for all regression result.

If there are multiple constant variables in x_columns and the user wants to include a constant variable, it is recommended to set both of ignore_constant and use_intercept to be True.

Adds columns:

samples (*int*) The number of samples.

r (*float*) Pearson's correlation coefficient.

rSquared (*float*) Coefficient of determination.

beta (*list* of *float*) The betas of each of x_columns, without the intercept component.

stdErr_beta (*list* of *float*) The standard errors of the betas.

tStat_beta (*list* of *float*) The t-stats of the betas.

hasIntercept (*bool*) True if using an intercept.

intercept (*float*) The intercept fit by the regression.

stdErr_intercept (*float*) The standard error of the intercept.

tStat_intercept (*float*) The t-stat of the intercept.

logLikelihood (*float*) The log-likelihood of the data given the fitted model

akaikeIC (*float*) The Akaike information criterion of the data given the fitted model

bayesIC (*float*) The Bayes information criterion of the data given the fitted model

cond (*float*) The condition number of Gramian matrix, i.e. X^TX.

const_columns (*list* of *string*) The list of variables in x_columns that are constants.

Parameters

- **y_column** (*str*) – Name of the column containing the dependent variable.

- **x_columns** (*list of str*) – Names of the columns containing the independent variables.
- **weight_column** (*str*) – Name of the column containing weights for the observations.
- **use_intercept** (*bool*) – Whether the regression should consider an intercept term. (default: True)
- **ignore_constants** (*bool*) – Whether the regression should ignore independent variables, defined by x_columns, that are constants. See constant columns above. (default: False)
- **constant_error_bound** (*float*) – Used when ignore_constants = True, otherwise ignored. The error bound on (|observations| * variance) to determine if a variable is constant. A variable will be considered as a constant c if and only if the sum of squared differences to c is less than the error bound. Default is 1.0E-12.

ts.flint.summarizers.**max**(*column*)

Get the max of a column.

Adds columns:

<column>_max The max of the column

Parameters **column** (*str*) – name of the column to be summarized

ts.flint.summarizers.**mean**(*column*)

Computes the arithmetic mean of a column.

Adds columns:

<column>_mean (*float*) The mean of the column

Parameters **column** (*str*) – name of the column to be summarized

ts.flint.summarizers.**min**(*column*)

Get the min of a column.

Adds columns:

<column>_min The min of the column

Parameters **column** (*str*) – name of the column to be summarized

ts.flint.summarizers.**nth_central_moment**(*column*, *n*)

Computes the nth central moment of the values in a column.

Adds columns:

<column>_<n>thCentralMoment (*float*) The n th central moment of values in <column>.

Parameters

- **column** (*str*) – name of the column to be summarized
- **n** (*int*) – which moment to compute

ts.flint.summarizers.**nth_moment**(*column*, *n*)

Computes the nth raw moment of the values in a column.

Adds columns:

<column>_<n>thMoment (*float*) The n th raw moment of values in <column>.

Parameters

- **column** (*str*) – name of the column to be summarized
- **n** (*int*) – which moment to compute

ts.flint.summarizers.**product**(*column*)

Computes the product of a column.

Adds columns:

<column>_product (*float*) The product of the column

Parameters **column** (*str*) – name of the column to be summarized

ts.flint.summarizers.**quantile**(*column*, *phis*)

Computes the quantiles of the values in a column.

Adds columns:

<column>_<phi>quantile (*float*) The quantiles of values in <column>.

Parameters

- **sc** (*SparkContext*) – Spark context
- **column** (*str*) – name of the column to be summarized
- **phis** (*list*) – the quantiles to compute, ranging in value from (0.0,1.0]

ts.flint.summarizers.**skewness**(*column*)

Computes the skewness of a column.

Adds columns:

<column>_skewness (*float*) The skewness of the column

Parameters **column** (*str*) – name of the column to be summarized

ts.flint.summarizers.**stddev**(*column*)

Computes the stddev of a column

Adds columns:

<column>_stddev (*float*) The standard deviation of <column>

Parameters **column** (*str*) – name of the column to be summarized

ts.flint.summarizers.**sum**(*column*)

Computes the sum of the values in a column.

Adds columns:

<column>_sum (*float*) The sum of values in <column>.

Parameters **column** (*str*) – name of the column to be summarized

ts.flint.summarizers.**variance**(*column*)
Computes the variance of a column.

Adds columns:

<column>_variance (*float*) The variance of the column

Parameters **column** (*str*) – name of the column to be summarized

ts.flint.summarizers.**weighted_correlation**(*x_column*, *y_column*, *weight_column*)
Computes weighted correlation of two columns.

Adds columns:

<x_column>_<y_column>_<weight_column>_weightedCorrelation (*float*) correlation of x_column and y_column

Parameters

- **x_column** (*str*) – name of column X
- **y_column** (*str*) – name of column y
- **weight_column** (*str*) – name of weight column

ts.flint.summarizers.**weighted_covariance**(*x_column*, *y_column*, *weight_column*)
Computes unbiased weighted covariance of two columns.

Adds columns:

<x_column>_<y_column>_<weight_column>_weightedCovariance (*float*) covariance of x_column and y_column

Parameters

- **x_column** (*str*) – name of column X
- **y_column** (*str*) – name of column y
- **weight_column** (*str*) – name of weight column

ts.flint.summarizers.**weighted_mean**(*value_column*, *weight_column*)
Computes the mean, standard deviation, and t-stat of values in one column, with observations weighted by the values in another column. Also computes the number of observations.

Adds columns:

<value_column>_<weight_column>_weightedMean (*float*) The weighted mean of the values in <value_column> weighted by values in <weight_column>.

<value_column>_<weight_column>_weightedStandardDeviation (*float*) The weighted standard deviation of the values in <value_column> weighted by values in <weight_column>.

<value_column>_<weight_column>_weightedTStat (*float*) The t-stats of the values in <value_column> weighted by values in <weight_column>.

<value_column>_<weight_column>_observationCount (*int*) The number of observations.

Parameters

- **value_column** (*str*) – name of the column of values to be summarized
- **weight_column** (*str*) – name of the column of weights to be used

ts.flint.summarizers.**zscore**(*column*, *in_sample*)

Computes the z-score of values in a column with respect to the sample mean and standard deviation observed so far.

Optionally includes the current observation in the calculation of the sample mean and standard deviation, if `in_sample` is true.

Adds columns:

<column>_zScore (*float*) The z-scores of values in `<column>`.

Parameters

- **column** (*str*) – name of the column to be summarized
- **in_sample** (*bool*) – whether to include or exclude a data point from the sample mean and standard deviation when computing the z-score for that data point

Windows

This module contains window constructors suitable as arguments to:

- `TimeSeriesDataFrame.addWindows()`
- `TimeSeriesDataFrame.summarizeWindows()`

Example:

```
>>> from ts.flint import windows
>>> prices.addWindows(windows.past_absolute_time('1d'), key='id')
```

ts.flint.windows.**past_absolute_time**(*duration*)

Creates a window over a fixed amount of time into the past.

The `duration` should be specified as a time string with units, such as `'5 days'` or `'100s'`. These strings are interpreted by `scala.concurrent.duration.Duration`, so the rules there apply. Importantly:

> Format is `"<length><unit>"`, where whitespace is allowed before, between and after the parts. Infinities are designated by `"Inf"`, `"PlusInf"`, `"+Inf"` and `"-Inf"` or `"MinusInf"`.
>
> Valid choices [for unit] are:
>
> `d`, `day`, `h`, `hour`, `min`, `minute`, `s`, `sec`, `second`, `ms`, `milli`, `millisecond`, `µs`, `micro`, `microsecond`, `ns`, `nano`, `nanosecond` and their pluralized forms (for every but the first mentioned form of each unit, i.e. no "ds", but "days").

Example:

```
>>> prices.addWindows(windows.past_absolute_time('1d'), key='id')
```

Parameters **duration** (*str*) – the size of the window, as a string with units

ts.flint.windows.**future_absolute_time**(*duration*)

Creates a window over a fixed amount of time into the future.

The `duration` should be specified as a time string with units, such as `'5 days'` or `'100s'`. These strings are interpreted by `scala.concurrent.duration.Duration`, so the rules there apply. Importantly:

> Format is `"<length><unit>"`, where whitespace is allowed before, between and after the parts. Infinities are designated by `"Inf"`, `"PlusInf"`, `"+Inf"` and `"-Inf"` or `"MinusInf"`.

Valid choices [for unit] are:

`d`, `day`, `h`, `hour`, `min`, `minute`, `s`, `sec`, `second`, `ms`, `milli`, `millisecond`, `µs`, `micro`, `microsecond`, `ns`, `nano`, `nanosecond` and their pluralized forms (for every but the first mentioned form of each unit, i.e. no "ds", but "days").

Example:

```
>>> prices.addWindows(windows.future_absolute_time('1d'), key='id')
```

Parameters **duration** (*str*) – the size of the window, as a string with units

Clocks

`ts.flint.clocks.`**`uniform`**(*sql_ctx*, *frequency*, *offset='0s'*, *begin_date_time='1970-01-01'*, *end_date_time='2030-01-01'*, *time_zone='UTC'*)

Return an evenly sampled TimeSeriesDataFrame with only a time column.

This function is end-inclusive and will create a tick if the last tick falls on the `end_date_time`.

Parameters

- **`sql_ctx`** – pyspark.sql.SQLContext
- **`frequency`** – the time interval between rows, e.g "1s", "2m", "3d" etc.
- **`offset`** – the time to offset this clock from the begin time. Defaults to "0s". Note that specifying an offset greater than the frequency is the same as specifying (offset % frequency).
- **`begin_date_time`** – the begin date time of this clock. Defaults to "1970-01-01".
- **`end_date_time`** – the end date time of this clock. Defaults to "2030-01-01".
- **`time_zone`** – the time zone which will be used to parse *begin_date_time* and *end_date_time* when time zone information is not included in the date time string. Defaults to "UTC".

Returns a new `TimeSeriesDataFrame`

Functions

`ts.flint.functions.`**`udf`**(*f=None*, *returnType=<Mock name='mock()' id='139654164846464'>*, *arg_type='pandas'*)

Creates a column expression representing a user defined function (UDF).

This behaves the same as `udf()` when used with a PySpark function, such as `withColumn()`.

This can also be used with Flint functions, such as *`ts.flint.TimeSeriesDataFrame.summarizeCycles()`*.

This can be used to define a row user define function or a columnar user define function:

1. Row udf

   A row udf takes one or more scalar values for each row, and returns a scalar value for that row.

   A `Column` object is needed to specifiy the input, for instance, `df['v']`.

   Example:

```
>>> @udf(DoubleType())
>>> def plus_one(v):
...     return v+1
>>> col = plus_one(df['v'])
```

2. Pandas Columnar udf

   A pandas columnar udf takes one or more `pandas.Series` or `pandas.DataFrame` as input, and returns either a scalar value or a `pandas.Series` as output.

   If the user function takes `pandas.Series`, a `Column` is needed to specify the input, for instance, `df['v']`.

   If the user function takes a `pandas.DataFrame`, a `DataFrame` is needed to specify the input, for instance, `df[['v', 'w']]`.

   Default return type is DoubleType.

   Example:

   Takes `pandas.Series`, returns a scalar

```
>>> @udf(DoubleType())
>>> def weighted_mean(v, w):
...     return numpy.average(v, weights=w)
>>> col = weighted_mean(df['v'], df['w'])
```

   Takes a `pandas.DataFrame`, returns a scalar

```
>>> @udf(DoubleType())
>>> def weighted_mean(df):
...     return numpy.average(df.v, weighted=df.w)
>>> col = weighted_mean(df[['v', 'w']])
```

   Takes a `pandas.Series`, returns a `pandas.Series`

```
>>> @udf(DoubleType())
>>> def percent_rank(v):
...     return v.rank(pct=True)
>>> col = percent_rank(df['v'])
```

   Different functions take different types of udf. For instance,

   - `pyspark.sql.DataFrame.withColumn()` takes a row udf
   - *`ts.flint.TimeSeriesDataFrame.summarizeCycles()`* takes a columnar udf that returns a scalar value.

3. Numpy Columnar udf

   Numpy columnar udf is similar to pandas columnar udf. The main difference is numpy udf expects the function input to be numpy data structure and types, i.e., numpy.ndarray or numpy.flaat64. When a named input is expected, the input to the udf would be a python ordered dict from str to numpy.ndarray or numpy primitive type.

   Numpy columnar udf is faster than pandas columnar udf, particularly in summarizeWindows, where the overhead of creating pandas.Series and pandas.DataFrame for each window can be large. Therefore, user should try to use numpy columnar udf with summarizeWindows.

   Examples:

```
>>> @udf(DoubleType(), arg_type='numpy')
>>> def mean_udf(v):
...     # v is numpy.ndarray
...     return v.mean()
>>> col = mean_udf(df['v'])
```

See also:

ts.flint.TimeSeriesDataFrame.summarizeCycles() ts.flint.TimeSeriesDataFrame.addColumnsForCycles() *ts.flint.TimeSeriesDataFrame.summarizeIntervals()* *ts.flint.TimeSeriesDataFrame.summarizeWindows()*

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

Q

R

S

T

U

V

W

Z