PySpark for Time Series Analysis

David Palaitis
Two Sigma Investments

SPARK SUMMIT EAST 2017

About Me

TWO SIGMA

SPARK SUMMIT EAST 2017

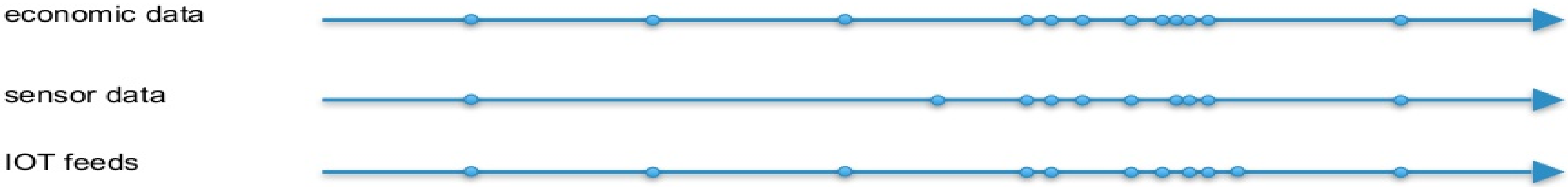
Important Legal Information

The information presented here is offered for recruiting purposes only and should not be used for any other purpose (including, without limitation, the making of investment decisions). Examples provided herein are for illustrative purposes only and are not necessarily based on actual data. Nothing herein constitutes an offer to sell or the solicitation of any offer to buy any security or other interest. We consider this information to be confidential and not for redistribution or dissemination. Some of the images, logos or other material used herein may be protected by copyright and/or trademark. If so, such copyrights and/or trademarks are most likely owned by the entity that created the material and are used purely for identification and comment as fair use under international copyright and/or trademark laws. Use of such image, copyright or trademark does not imply any association with such organization (or endorsement of such organization) by Two Sigma, nor vice versa.

TWO SIGMA

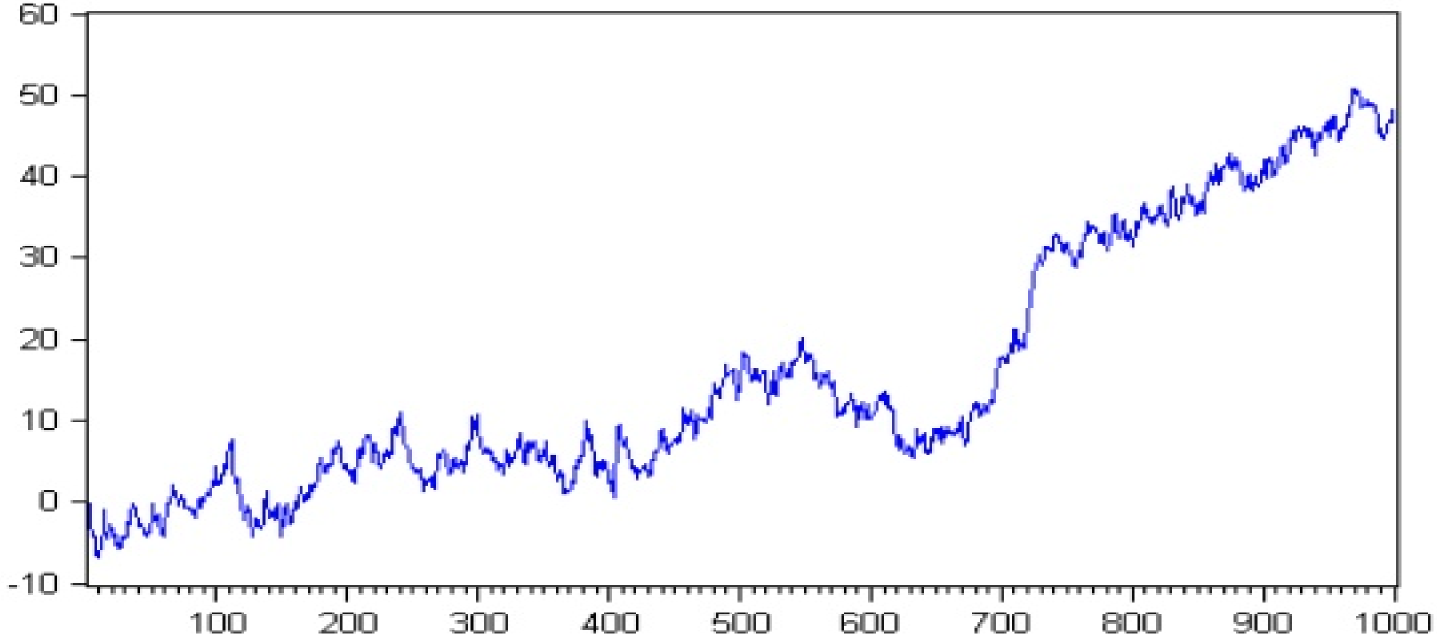
Time Series

An ordered sequence of values of a variable

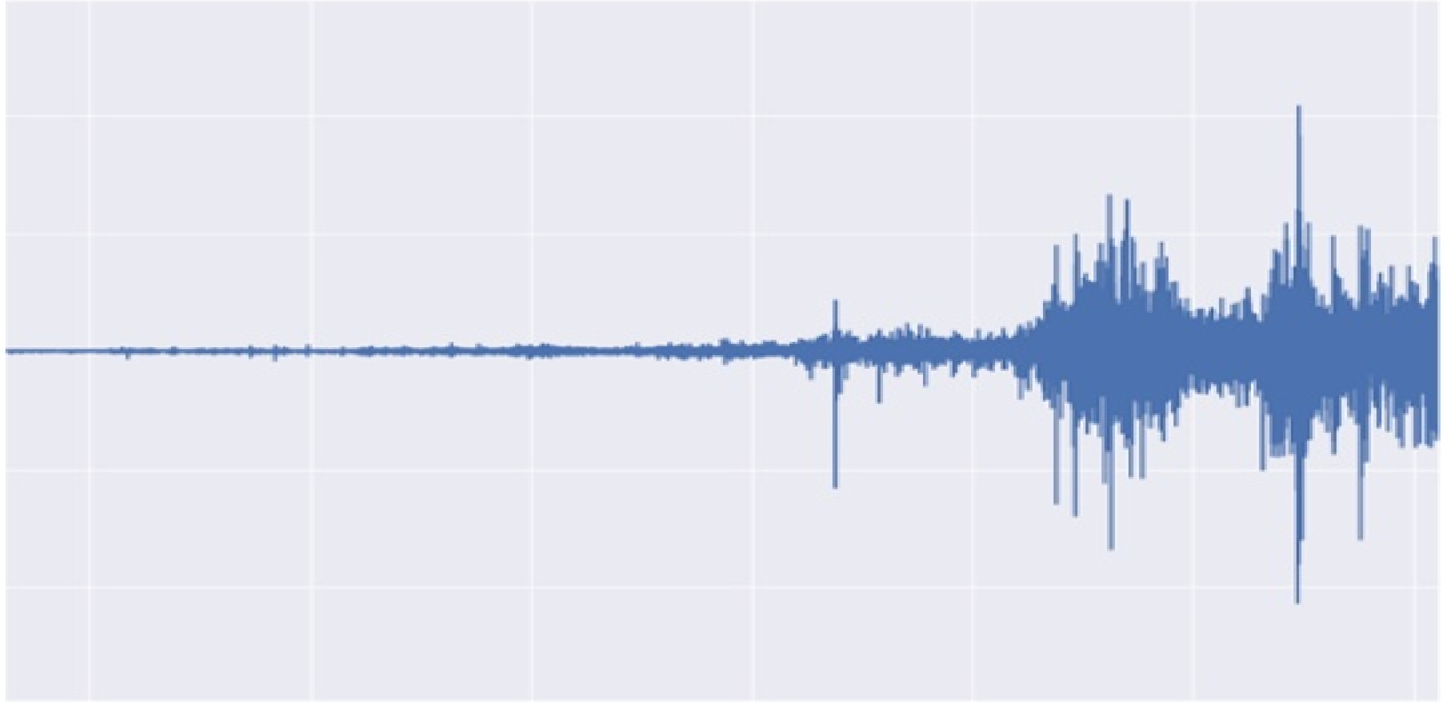


SPARK SUMMIT EAST 2017

Time Series Analysis

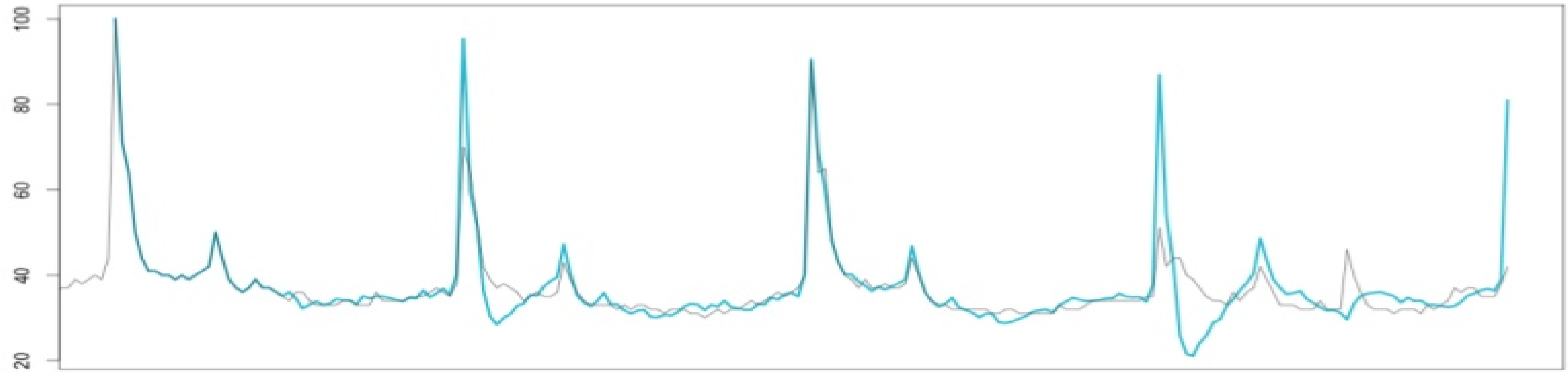


Time Series Analysis



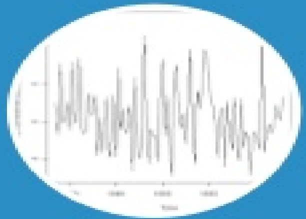
SPARK SUMMIT EAST 2017

Time Series Analysis



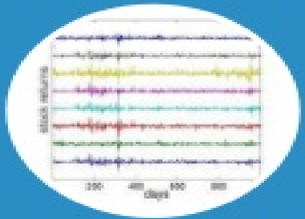
SPARK SUMMIT EAST 2017

Time Series at Two Sigma

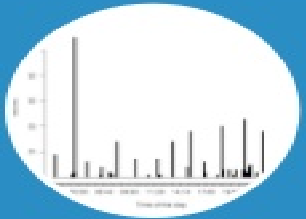


Millions of Time Series

Big and Small (1GB – 1PB)



Narrow (10 columns) and Wide (1MM Columns)



Evenly and Unevenly Spaced Observations

SPARK SUMMIT EAST 2017

Spark

flint

Let’s start from the beginning …

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma,
Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

1 Introduction

Cluster computing frameworks like MapReduce [10] and Dryad [19] have been widely adopted for large-scale data analytics. These systems let users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications: those that *reuse* intermediate results across multiple computations. Data reuse is common in many *iterative* machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression. Another compelling use case is *interactive* data mining, where a user runs multiple ad-hoc queries on the same subset of the data. Unfortunately, in most current frameworks, the only way to reuse data between computations (*e.g.*, between two MapReduce jobs) is to write it to an external stable storage system, *e.g.*, a distributed file system. This incurs substantial overheads due to data replication, disk I/O, and serialization, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (*e.g.*, looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, *e.g.*, to let a user load several datasets into memory and run ad-hoc queries across them.

In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance *efficiently*. Existing abstractions for in-memory storage on clusters, such as distributed shared memory [24], key-value stores [25], databases, and Piccolo [27], offer an interface based on fine-grained updates to mutable state (*e.g.*, cells in a table). With this interface, the only ways to provide fault tolerance are to replicate the data across machines or to log updates across machines. Both approaches are expensive for data-intensive workloads, as they require copying large amounts of data over the cluster network, whose bandwidth is far lower than that of RAM, and they incur substantial storage overhead.

In contrast to these systems, RDDs provide an interface based on *coarse-grained* transformations (*e.g.*, map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its *lineage*) rather than the actual data.[1] If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute

[1] Checkpointing the data in some RDDs may be useful when a lineage chain grows large, however, and we discuss how to do it in §5.4.

SPARK SUMMIT EAST 2017

$RDD[T]$

$$RDD[T]$$

SPARK SUMMIT EAST 2017

$$flatMap(f : T \rightarrow Seq[U]) : RDD[T] \rightarrow RDD[U]$$

$$flatMap(f: T \rightarrow Seq[U]): RDD[T] \rightarrow RDD[U]$$

$$flatMap(f: T \rightarrow Seq[U]): RDD[T] \rightarrow RDD[U]$$

$$reduce(f : (T, T) \rightarrow T) : RDD[T] \rightarrow T$$

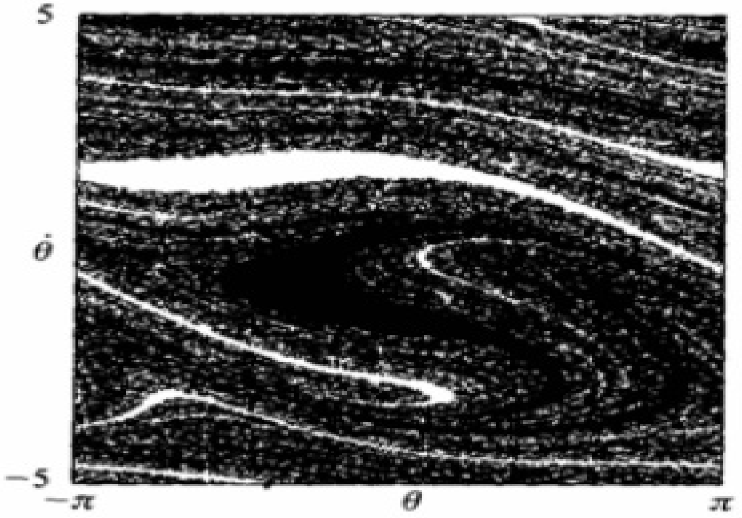
SPARK SUMMIT EAST 2017

$$reduce(f: (T, T) \to T): RDD[T] \to T$$

SPARK SUMMIT EAST 2017

$$reduce(f: (T, T) \to T): RDD[T] \to T$$

Examples!



What's Missing?

You can’t even do “Word Count”

$$PairRDD : RDD[(K,V)]$$

$$PairRDD: RDD[(K,V)]$$

*K has an equivalence relation.

$$PairRDD: RDD[(K, V)]$$

$^{*}K\ has\ an\ equivalence\ relation.$

$$groupByKey(): RDD[(K, V)] \rightarrow RDD[(K, Seq[V])]$$

$$groupByKey(): RDD[(K, V)] \rightarrow RDD[(K, Seq[V])]$$

$$reduceByKey(f:(V,V) \rightarrow V):RDD[(K,V)] \rightarrow RDD[(K,V)]$$

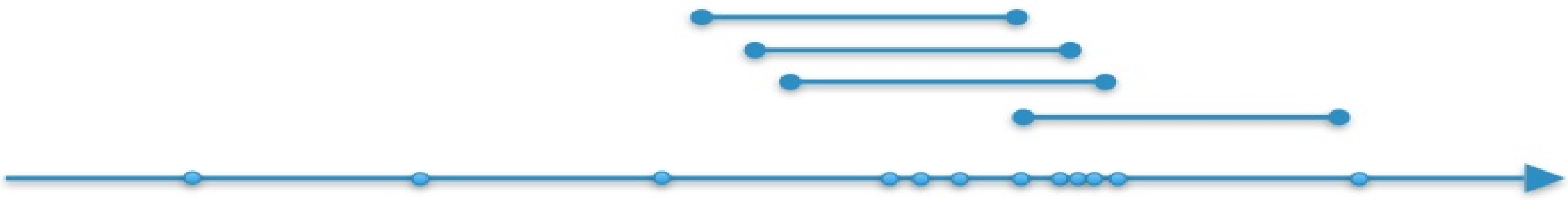
$$reduceByKey(f:(V,V) \to V): RDD[(K,V)] \to RDD[(K,V)]$$

$$reduceByKey(f: (V,V) \rightarrow V): RDD[(K,V)] \rightarrow RDD[(K,V)]$$

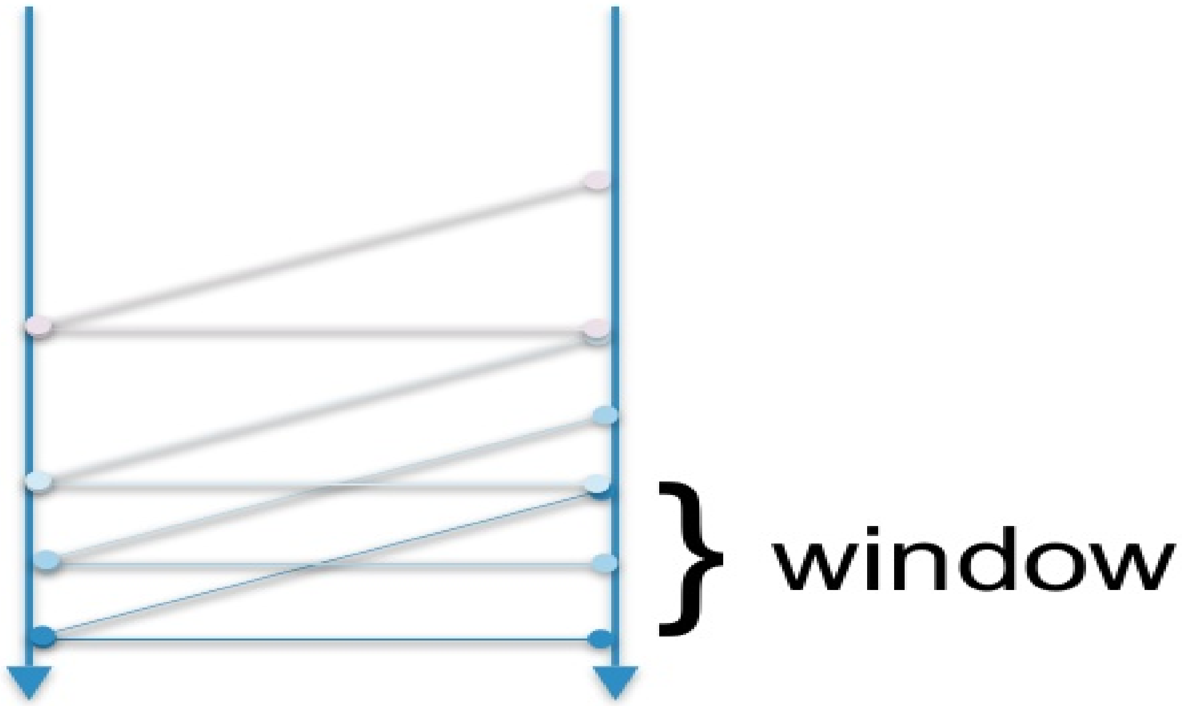
“Word Count” !

What’s missing? Time.

Windowed Aggregations



Temporal Joins



$$TimeSeriesRDD : RDD[(K^*, V)]$$

$$TimeSeriesRDD: RDD[(K^*, V)]$$

*K has a natural ordering and is defined in a metric space.

SPARK SUMMIT EAST 2017

$$TimeSeriesRDD: RDD[(K^*, V)]$$

$^*K$ has a natural ordering and is defined in a metric space.

groupByWindow(w): $RDD[(K,V)] \rightarrow RDD[(K,SEQ(V))]$

groupByWindow(w): $RDD[(K,V)] \text{ -> } RDD[(K,Seq[V])]$

w is a window specification e.g. 500ms, 5s, 3 business days reduceByWindow(f: (V, V) => V, w): RDD[(K, W)] => RDD[(K, V)] reduceByWindow(f: (V, V) => V, w):   RDD[(K, V)] => RDD[(K, V)] flint

https://github.com/twosigma/flint

SPARK SUMMIT EAST 2017

Getting Started …

Secure | https://github.com/twosigma/flint

twosigma / flint

Watch 24 | Star 35 | Fork 6

Code | Issues 2 | Pull requests 0 | Projects 0 | Pulse | Graphs

No description or website provided.

6 commits	1 branch	0 releases	1 contributor	Apache-2.0

Branch: master | New pull request | Find file | Clone or download

icexelloss committed on GitHub Update README.md — Latest commit 6961a78 on Oct 24, 2016

cla	Init commit	3 months ago
project	Init commit	3 months ago
src	Init commit	3 months ago
CONTRIBUTING.md	Init commit	3 months ago
LICENSE	Init commit	3 months ago
README.md	Update README.md	3 months ago
build.sbt	Init commit	3 months ago
scalastyle-config.xml	Init commit	3 months ago

README.md

Flint: A Time Series Library for Apache Spark

The ability to analyze time series data at scale is critical for the success of finance and IoT applications based on Spark. Flint is the Two Sigma's implementation of highly optimized time series operations in Spark. It performs truly parallel and rich analyses on time series data by taking advantage of the natural ordering in time series data to provide locality-based optimizations. Flint is an open source library for Spark based around the `TimeSeriesRDD`, a time series aware data structure, and a collection of time series utility and analysis functions that use `TimeSeriesRDD` s. Unlike `DataFrame` and `Dataset`, Flint's `TimeSeriesRDD` s can leverage the existing ordering properties of datasets at rest and the fact that almost all data manipulations and analysis over these datasets respect their temporal ordering properties. It differs from other time series efforts in Spark in its ability to efficiently compute across panel data or on large scale high frequency data.

How to build

To build this sbt project, one could simply do

```
sbt assembly
```

```
In [ ]: from   ts.flint                import FlintContext
        from   ts.flint                import TimeSeriesDataFrame
        from   ts.flint                import summarizers, windows
```

```
In [ ]: df = spark.read.csv(
             "/path/to/my/dataset", header=True, inferSchema = true)
        df = df.withColumn('time',newDf.time.cast("long"))
```

In [40]:

```
leftTsDf.show()
rightTsDf.show()
```

```
+----+-----+
|time|quote|
+----+-----+
|   0|  0.0|
|   1|  1.0|
|   2|  2.0|
|   3|  3.0|
|   4|  4.0|
+----+-----+

+----+-----+
|time|trade|
+----+-----+
|   0|  0.0|
|   2|  2.0|
|   3|  3.0|
+----+-----+
```

```
In [41]: joined = leftTsDf.leftJoin(rightTsDf, tolerance="1")
         joined.show()

+----+-----+-----+
|time|quote|trade|
+----+-----+-----+
|   0|  0.0|  0.0|
|   1|  1.0|  0.0|
|   2|  2.0|  2.0|
|   3|  3.0|  3.0|
|   4|  4.0|  3.0|
+----+-----+-----+
```

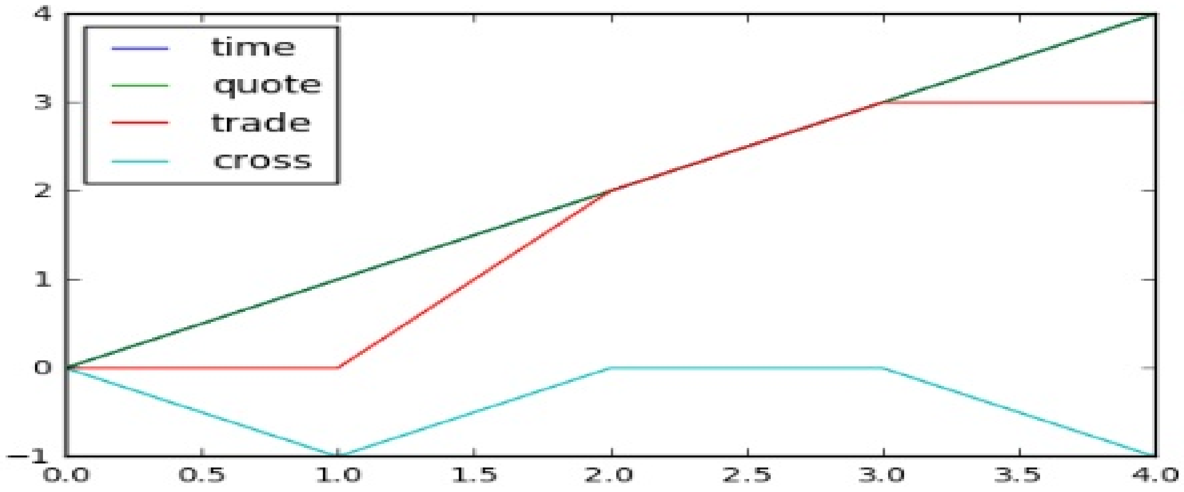
```
In [86]: from   ts.flint                 import FlintContext
         from   ts.flint                 import TimeSeriesDataFrame
         from   ts.flint                 import summarizers, windows
         from pyspark.sql.types import DoubleType
         %matplotlib inline

         @udf(DoubleType())
         def crossing(openPrice, closePrice):
             return closePrice - openPrice

         x = joined.withColumn('cross', crossing(joined.quote, joined.trade))
```

```
In [87]: x.toPandas().plot()

Out[87]: <matplotlib.axes._subplots.AxesSubplot at 0x118e42860>
```



localhost:8889/notebooks/flint/python/docs/Flint%20Python%20Cookbook.ipynb

jupyter Flint Python Cookbook (autosaved)

File Edit View Insert Cell Kernel Widgets Help | Python [default]

Code CellToolbar

```
df.show()
```

Moving average

In [ ]:
```
# Moving average over the last two weeks for IBM

@ts.flint.udf(DoubleType())
def movingAverage(window):
    nrows = len(window)
    if nrows == 0:
        return 0
    return sum(row.closePrice for row in window) / nrows

df = price.addWindows(windows.past_absolute_time("14days"))
df = df.withColumn("movingAverage", movingAverage(df.window_past_14days))
df.show()
```

In [ ]:
```
# Moving average over the last two weeks for all tids in ACTIVE_3000_US

df = price.addWindows(windows.past_absolute_time("14days"), key="tid")
df = df.withColumn("movingAverage", movingAverage(df.window_past_14days))
df.show()
```

Cycles

`TimeSeriesDataFrame.addColumnsForCycle()` can be used to compute a new column based on all rows that share a timestamp.

Adding universe info

In [ ]:
```
# Add a column containing the number of instruments in the universe on each day

def universeSize(rows):
    size = len(rows)
    return {row:size for row in rows}

df = active_price.addColumnsForCycle(
            {"universeSize": (IntegerType(), universeSize)})
df.show()
```

In [ ]:
```
# Add a column containing the number of instruments that share a GICS code
```

Looking ahead.

Thank You.

Find me after the talk to see Flint in action.

SPARK SUMMIT EAST 2017