Analyzing Weather Data with Apache Spark

Jeremie Juban
Tom Kunicki

Introduction

- Who we are
  - Professional Services Division of The Weather Company
- What we do
  - Aviation
  - Energy
  - Insurance
  - Retail
- Apache Spark at The Weather Company
  - Feature Extraction
  - Predictive Modeling
  - Operational Forecasting

Goals

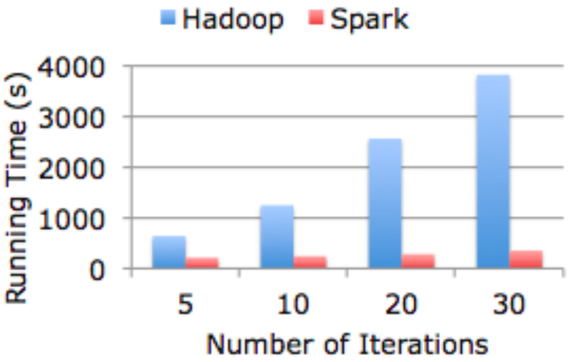
- Present high-level overview of Apache Spark
- Quick overview of gridded weather data formats
- Examples of how we ingest this data into Spark
- Provide insight into simple Spark operations on data

What is Spark?

Spark is a general-purpose cluster computing framework

2009 - Research project at UC Berkeley
2010 - Donated to Apache S.F.
2015 - Current release Spark 1.5

Generalization over MapReduce

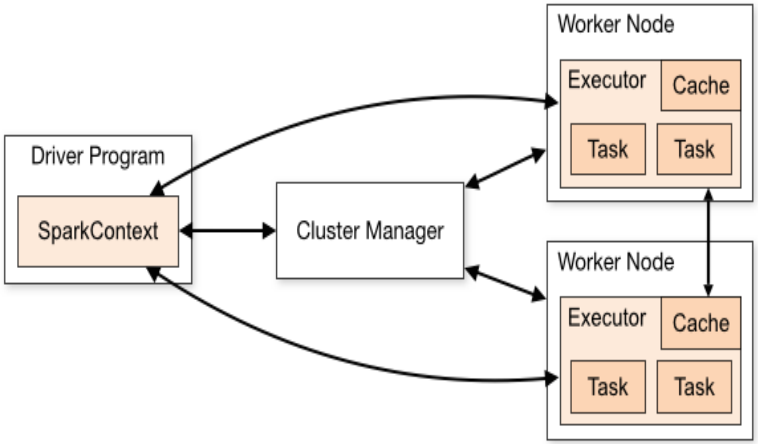


- Fast to run
  - Mode the code not the data
  - Lazy evaluation of big data queries
  - Optimizes arbitrary operator graphs
- Fast to write
  - Provides concise and consistent APIs in Scala, Java and Python.
  - Offers interactive shell for Scala/Python.

Resilient Distributed Dataset

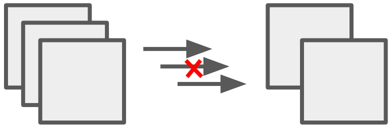
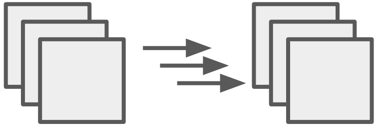
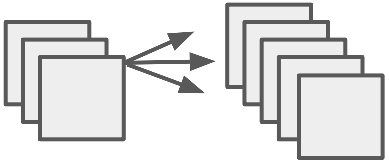
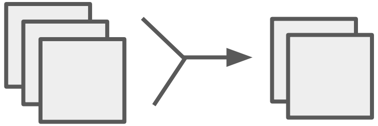
*"A Resilient Distributed Dataset (**RDD**), the basic abstraction in Spark. Represents an immutable, partitioned collection of elements that can be operated on in parallel."*

source: spark documentation



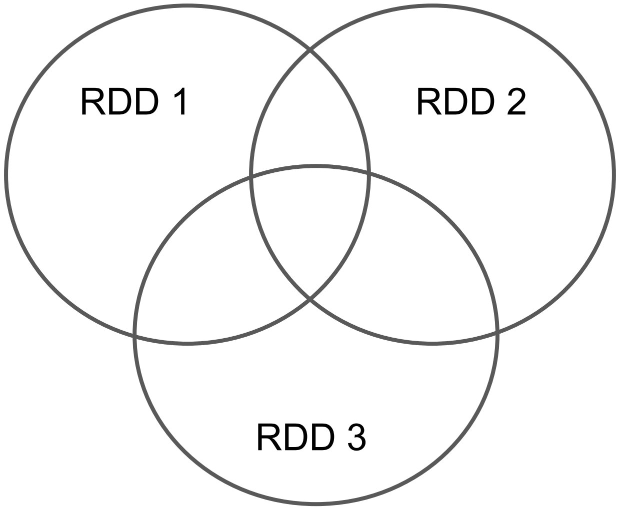
- Data are partitioned at the worker nodes
  - Enable efficient data reuse
- Store data and its transformations
  - Fault tolerant, coarse grain operations
- Two types of operations
  - Transformations (lazy evaluation)
  - Actions (trigger evaluation)
- Allow caching/persisting
  - MEMORY_ONLY, MEMORY_AND_DISK, DISK_ONLY...

RDD operation flow

Flow	Type	Example	Flow Diagram
Filter	Transformation	filter, distinct, substractByKey	
Map	Transformation	map, mapPartitions	
Scatter	Transformation	flatMap, flatMapValues	
Gather	Transformation	aggregate, reduceByKey	
	Action	reduce, collect, count, take	

RDD set operations

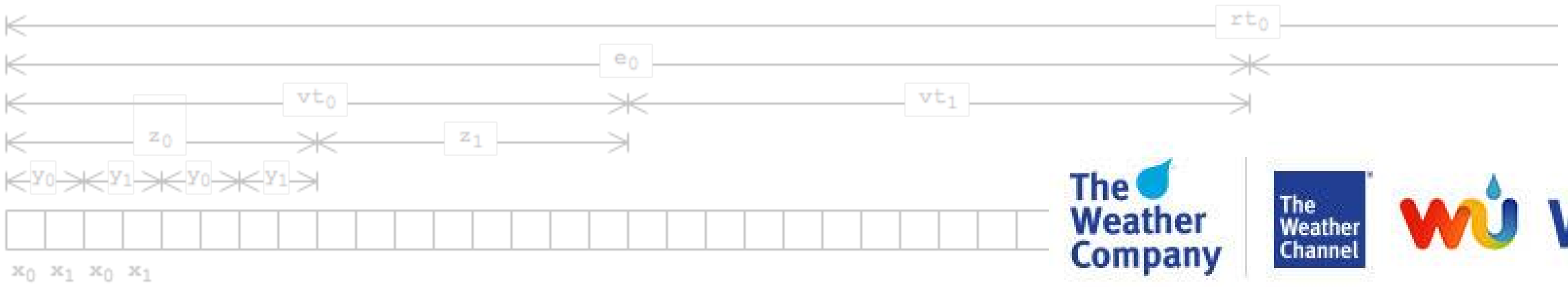
- Union
- Intersection
- Join
- leftOuterJoin
- RightOuterJoin
- Cartesian
- Cogroup



Loading Gridded Data into RDD

- Multi-dimensional gridded data
  - Observational, Forecast
  - Varying dimensionality
- Distributed in various binary formats
  - NetCDF, Grib, HDF, …
- NetCDF-Java/CDM
  - Common Data Model (CDM)
  - Canonical library for reading
- Many. Large. Files.

```
for each rt in ...:
  for each e in ...:
    for each vt in ...:
      for each z in ...:
        for each y in ...:
          for each x in ...:
            // magic!
```



Load Gridded Data into RDD (HDFS?)

- HDFS = **H**adoop **D**istributed **F**ile **S**ystem
- Standard datastore used with Spark
- Text delimited data formats are "standard", *meh...*
- Binary formats available, *conversion? how?*
- What about reading native grid formats from HDFS?
    - Work required to generalize storage assumptions for NetCDF-Java/CDM

Loading Gridded Data into RDD (Options?)

- Want to maintain ability to use NetCDF-Java
- NetCDF-Java assumes file-system and random access
- Distributed filesystems (NFS, GPFS, …)
- Object Store (AWS S3, OpenStack Swift)

Loading Gridded Data into RDD (Object Stores)

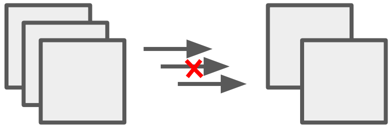
- Partition data and archive to key:value object store
- Map data request to list of keys
- Generate RDD from list of keys and distribute (*partitioning!*)
- **`flatMap`** object store key to RDD w/ data values

```
RDD[key] => RDD[((param, rt, e, vt, z, y, x), value)]
```

Loading Gridded Data into RDD (~~Object Stores~~ S3)

- Influences Spark cluster design
  - Maximize per-executor bandwidth for performance
  - Must colocate AWS EC2 instances in S3 region (no transfer cost)
- Plays well with AWS Spark EMR
- Can store to underlying HDFS in Spark friendly format.
- Now what do we do with our new RDD?

RDD Filtering



data: RDD[(key: (g, rt, e, vt, z, y, x) **, value:** Double**)]**

→ ECMWF Ensemble operational = 150 × 2 × 51 × 85 × 62 × 213988 = 17 trillion data point per day

1. **Filter**

Definition of a filtering function: `f(key) : Boolean`

Example

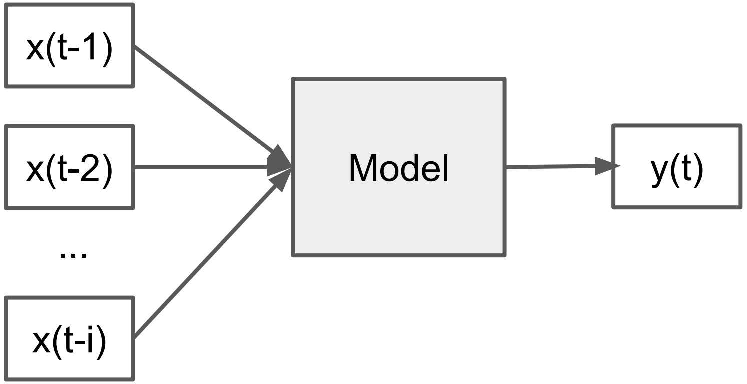
```
// Filter data - option 1: RDD
val dataSlice = data.filter( d => d._1 == "t2m" &&               // 2 meter temperature
                                  d._2 == ”6z”  &&               // 6z run
                                  d._4 <= 24 &&                  // first 24 hours
                                  d._6 > minLa && d._6 < maxLa && // Lo/La bounding box
                                  d._7 > minLo && d._7 < maxLo)


// Filter data - option 2: DataFrame
sqlContext.sql("SELECT * FROM data WHERE g<32 AND rt=’6z’ AND vt<= 24 AND ...")
```

RDD Spatio-temporal Translations

1. flatMap

Definition of a key mapper `f(key) : key`

- Shift time/space key (opposite sign)
- Emit a new variable name



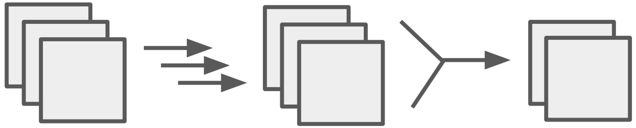
Example

Generate the past 24 hours lagged variables

`data: RDD[(key: (g, rt, vt), value: Double)]`

```
// Lagged variables
val dataset = data.flatMap(x=>(0 until 24).map(i => (
                                    ( x._1._1+"_m"+i+"h", x._1._2+i, x._1._3),// key
                                      x._2 ) )                                 // value
                            )
```

RDD Smoothing/Resampling



1. **Map**

Key truncation function `f(key) : key`

- Spatial - nearest neighbour, rounding/shift
- Temporal - time truncation

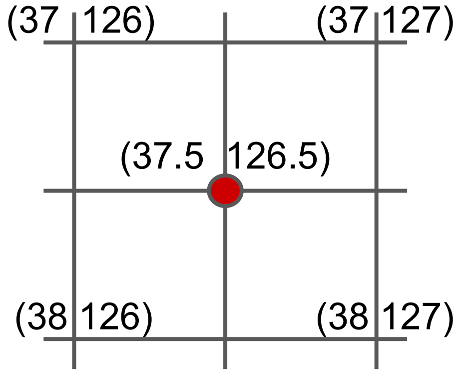
2. **ReduceByKey**

Aggregation function `f(Vector(value)) : value`

- Sum
- Average
- Median
- ...

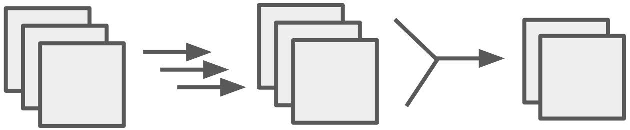
Example

Rounding



(37.386,126.436) → (37.5,126.5)

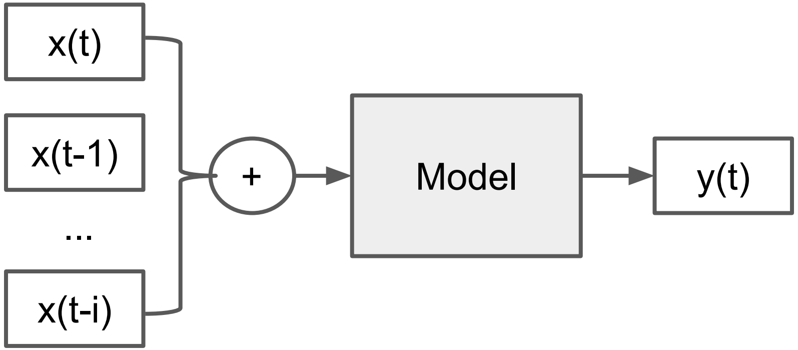
RDD Smoothing/Resampling



Temporal example

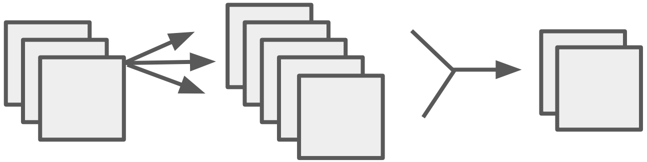
compute daily cumulative value

`dataset: RDD[(key: LocalDateTime, value: Double)]`




```
// Daily sum
val dataset_daily = dataset.map( t => (t._1.truncatedTo(ChronoUnit.DAYS),t._2) )

var dataset_fnl = dataset_daily.reduceByKey( (x,y) => (x+y) )
```

RDD Moving Average



1. Complete missing keys and sort by time
   - **subtract** → list missing key
   - **union** → complete the set
2. Apply a **sliding** mapper
   - key reduction function `f(Vector(Key)) : key`
   - value reduction function `f(Vector(value) : value`

```
// Moving Average
val missKeys = fullKSet.subtract(dataset.keys);
val complete = dataset.union(missKeys.map(x => (x,NaN))).sortByKey()

val slider = complete.sliding(3)

// Key reduction (and NaN cleaning)
val reduced = slider.map(x => ( x.last._1, x.map(_._2).filter(!_.isNaN) ))
// Value reduction
val dataset_fnl = slider.mapValues(x => math.round(x.sum / x.size))
```

Spark 1.5.0 Overview Programming Guides API Docs Deploying More

spark.mllib: data types, algorithms, and utilities

- Data types
- Basic statistics
  - summary statistics
  - correlations
  - stratified sampling
  - hypothesis testing
  - random data generation
- Classification and regression
  - linear models (SVMs, logistic regression, linear regression)
  - naive Bayes
  - decision trees
  - ensembles of trees (Random Forests and Gradient-Boosted Trees)
  - isotonic regression
- Collaborative filtering
  - alternating least squares (ALS)
- Clustering
  - k-means
  - Gaussian mixture
  - power iteration clustering (PIC)
  - latent Dirichlet allocation (LDA)
  - streaming k-means
- Dimensionality reduction
  - singular value decomposition (SVD)
  - principal component analysis (PCA)
- Feature extraction and transformation
- Frequent pattern mining
  - FP-growth
  - association rules
  - PrefixSpan
- Evaluation metrics
- PMML model export
- Optimization (developer)
  - stochastic gradient descent
  - limited-memory BFGS (L-BFGS)

- Feature extraction, transformation, and selection
- Decision trees for classification and regression
- Ensembles
- Linear methods with elastic net regularization
- Multilayer perceptron classifier

Thank you!