codecentric

SMACK Workshop Cassandra & Spark_

Matthias Niehoff

Agenda_

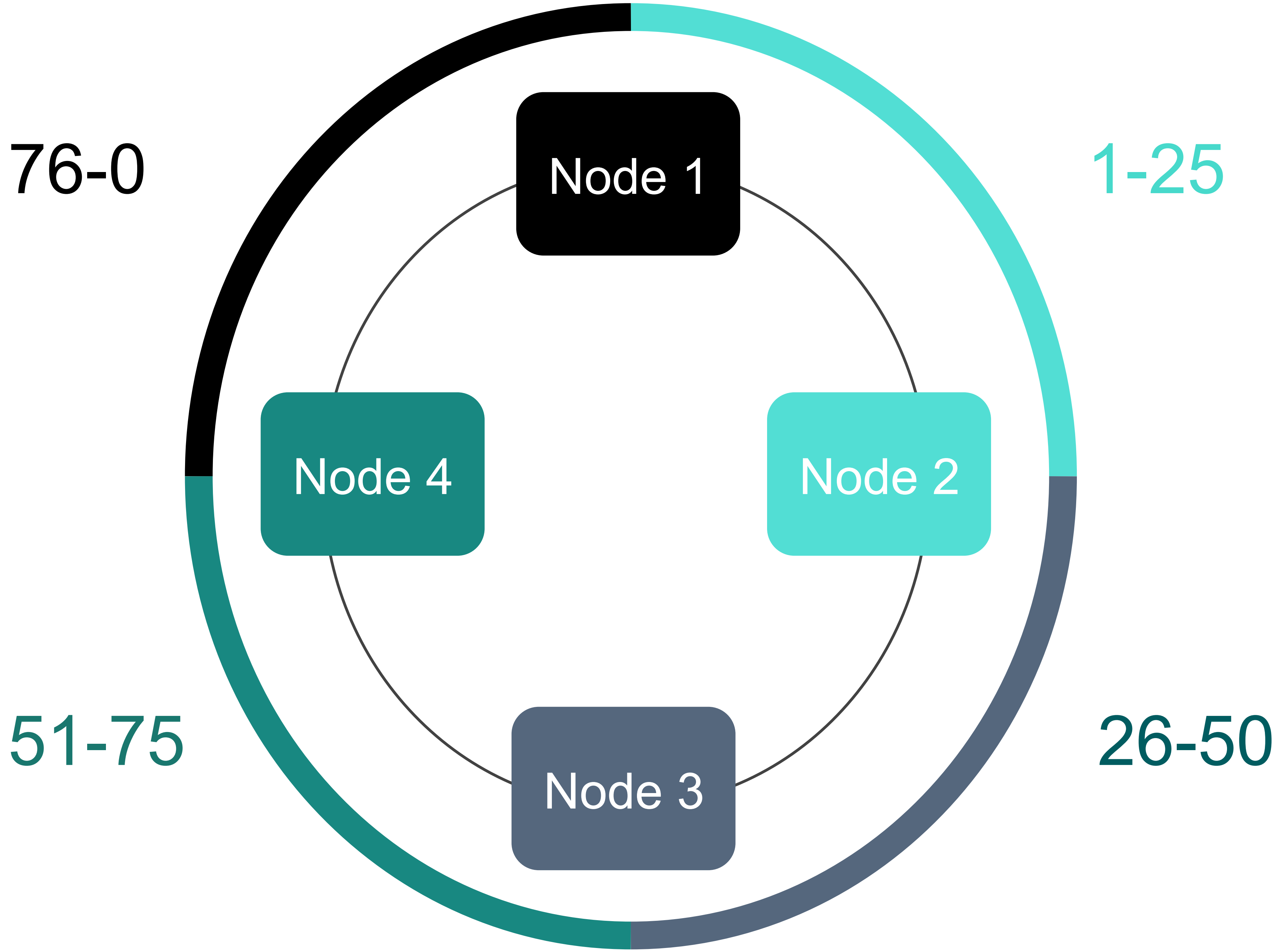
- Cassandra
- Spark
- Spark & Cassandra
- Spark Applications
- Spark Streaming
- Spark SQL
- Spark MLLib

Cassandra

Cassandra_

- Distributed database
- Highly Available
- Linear Scalable
- Multi Datacenter Support
- No Single Point Of Failure
- CQL Query Language
  - *Similiar to SQL*
  - *No Joins and aggregates*
- Eventual Consistency „Tunable Consistency“

Distributed Data Storage_



CQL - Querying Language With Limitations_

performer
name (PK)
genre
country

```
SELECT * FROM performer WHERE name = 'ACDC'
—> ok

SELECT * FROM performer WHERE name = 'ACDC' and country =
'Australia'
—> not ok

SELECT country, COUNT(*) as quantity FROM artists GROUP BY
country ORDER BY quantity DESC
—> not supported
```

Cassandra Exercise_

- Create tunnel to Mesos Cluster

```
ssh -i <<pfad/zur/pem>> core@<<masterIp>> -L 9042:node-0.cassandra.mesos:9042
```

- Start CQLSH in Virtual box

```
/home/smack/tools/apache-cassandra-3.0.8/bin/cqlsh
```

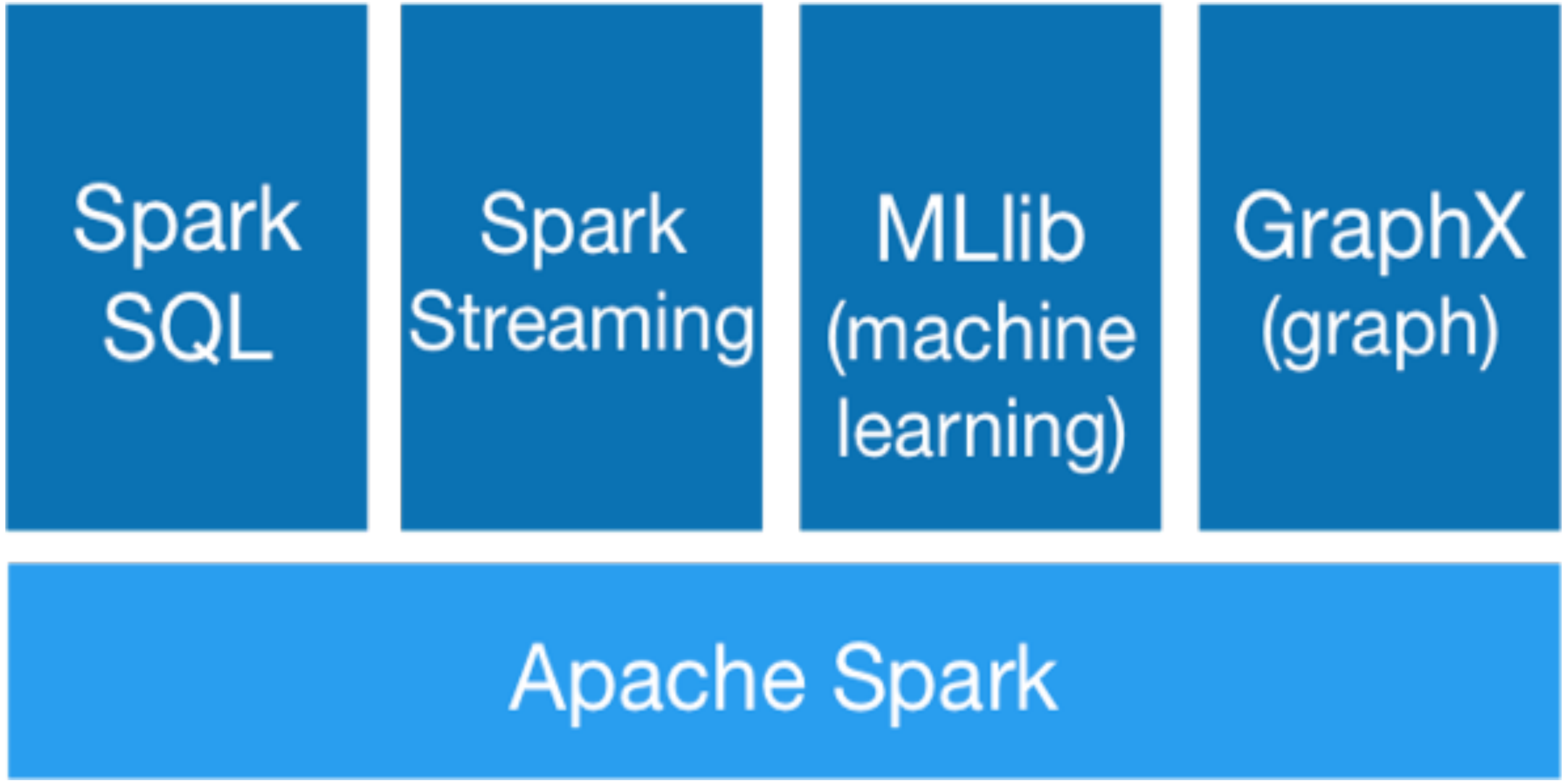
- Create Schema in Cassandra

```
src/main/resources/datamodel.cql
```

Spark

What Is Apache Spark_

- **Open Source & Apache project since 2010**
- **Data processing Framework**
  - *Batch processing*
  - *Stream processing*



Why Use Spark_

- **Fast**
  - *up to 100 times faster than Hadoop*
  - *a lot of in-memory processing*
  - *linear scalable using more nodes*
- **Easy**
  - *Scala, Java and Python API*
  - *Clean Code (e.g. with lambdas in Java 8)*
  - *expanded API: map, reduce, filter, groupBy, sort, union, join, reduceByKey, groupByKey, sample, take, first, count*
- **Fault-Tolerant**
  - *easily reproducible*

Easily Reproducable?_

- RDD's – Resilient Distributed Dataset
  - *Read–Only description of a collection of objects*
  - *Distributed among the cluster (on memory or disk)*
  - *Determined through transformations*
  - *Allows automatically rebuild on failure*
- Operations
  - *Transformations (map,filter,reduce...) —> new RDD*
  - *Actions (count, collect, save)*
- Only Actions start processing!

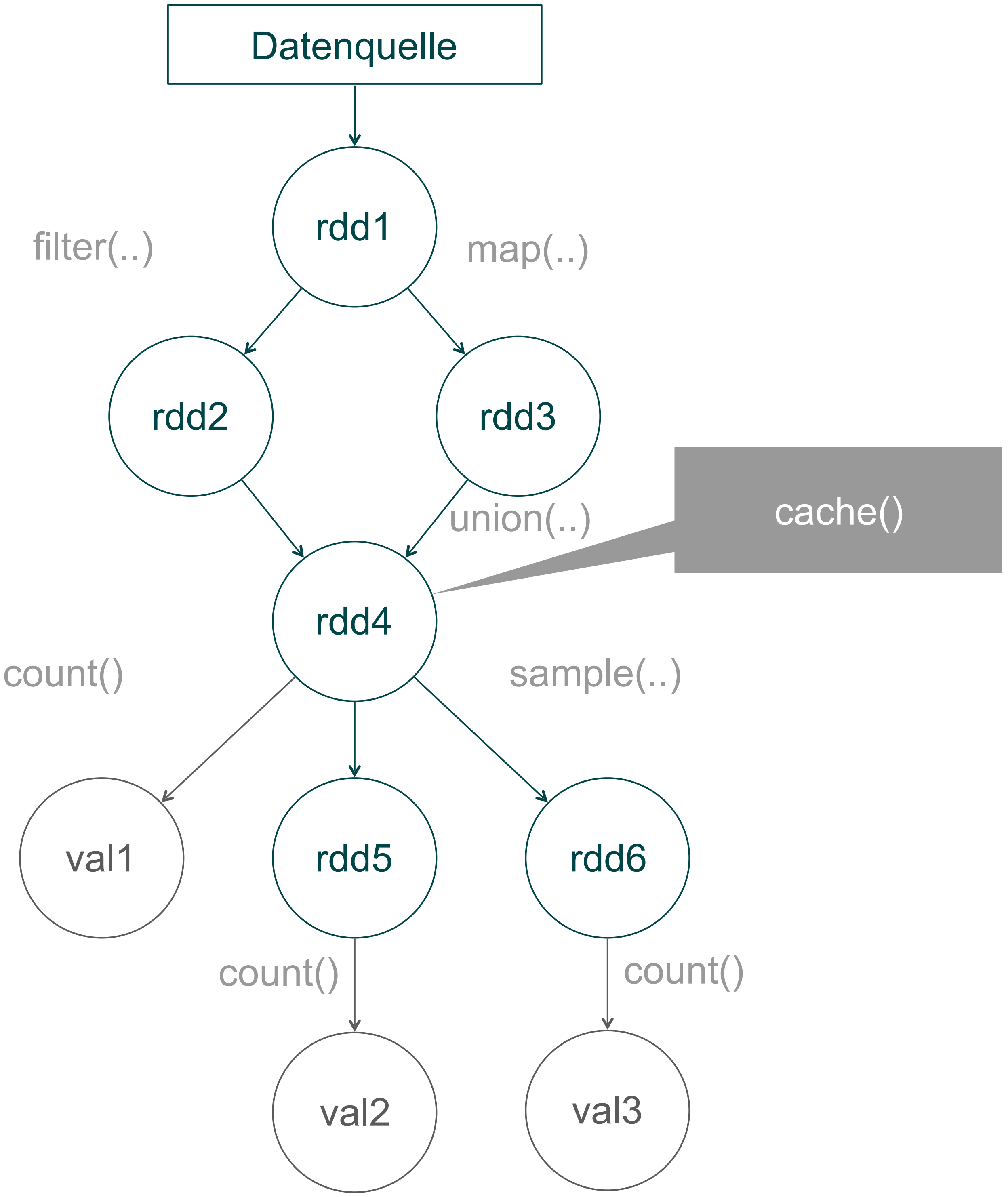
Properties Of A RDD_

- **Partitions**
  - *Describes the Partitions (i.e. one per Cassandra Partition)*
- **Dependencies**
  - *dependencies on parent RDD's*
- **Compute**
  - *The function to compute the RDD's partitions*
- **(Optional) Partitioner**
  - *How is the data partitioned? (Hash, Range..)*
- **(Optional) Preferred Location**
  - *Where to get the data (i.e. List of Cassandra Node IP's)*

RDD Example_

```
scala> val textFile = sc.textFile("README.md")
textFile: spark.RDD[String] = spark.MappedRDD@2ee9b6e3

scala> val linesWithSpark = textFile.filter(line =>
line.contains("Spark"))
linesWithSpark: spark.RDD[String] = spark.FilteredRDD@7dd4af09

scala> linesWithSpark.count()
res0: Long = 126
```

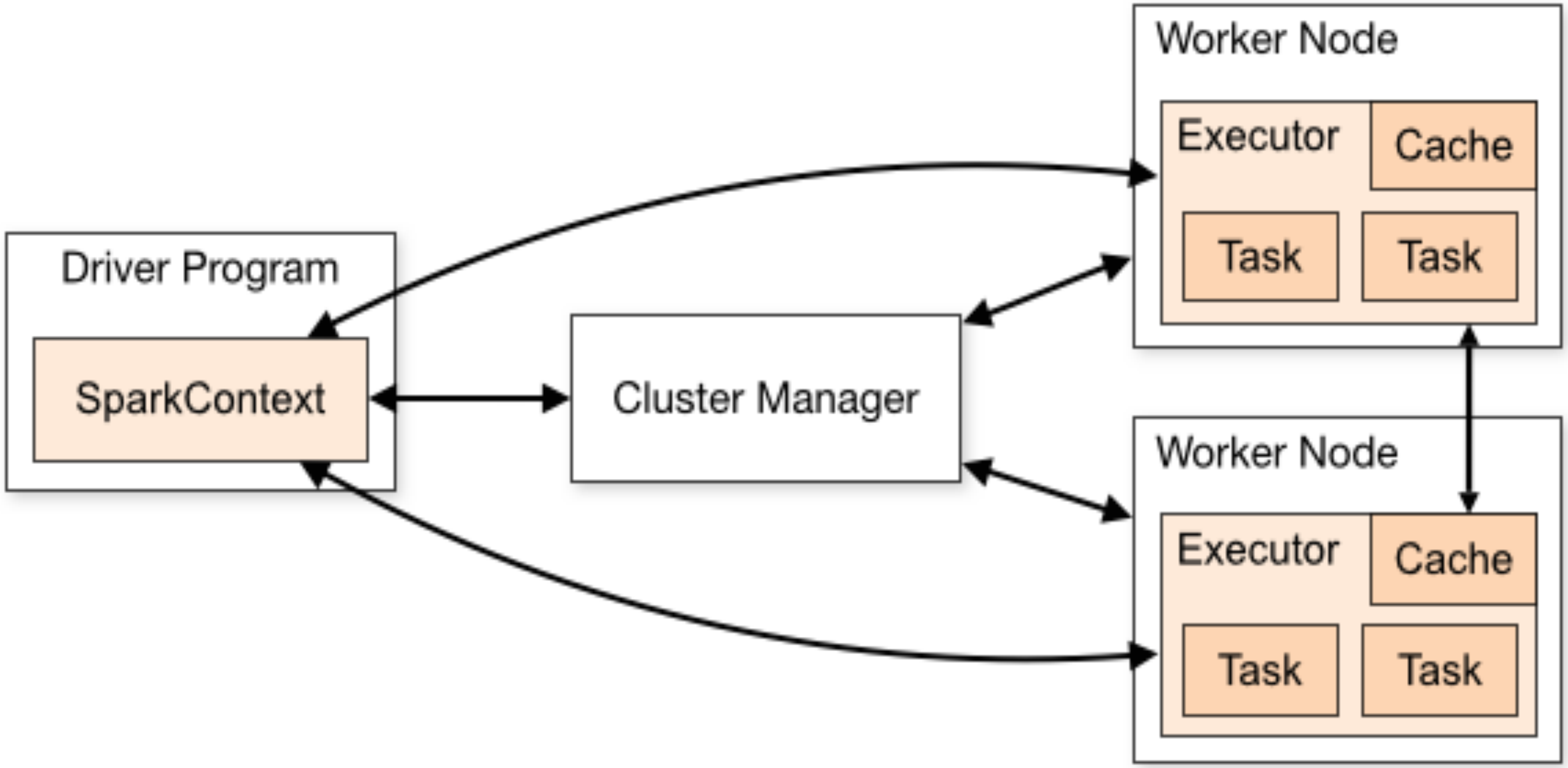
Reproduce RDD’s Using A Tree_



Spark Transformations & Actions_

- Transformations
  - *map, flatMap*
  - *sample, filter, distinct*
  - *union, intersection, cartesian*
- Actions
  - *reduce*
  - *count*
  - *collect,first, take*
  - *saveAsTextFile*
  - *foreach*

Run Spark In A Cluster_



Spark MESOS hadoop

Run Spark In A Cluster_

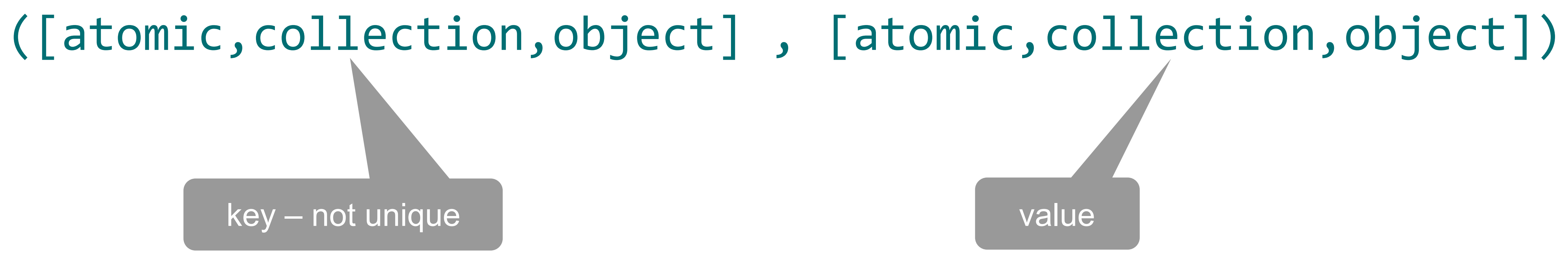
- **DAG is built from RDD**
- **DAG Scheduler**
  - *splits graph into stages of tasks*
  - *submits each stage*
  - *resubmits failed stages*
- **TaskScheduler**
  - *launches tasks via cluster manager*
  - *resubmits failed tasks*
- **Executors**
  - *execute tasks*
  - *store and serve blocks*

Driver Node

Master Node

Worker Nodes

Pair RDDs_



([atomic,collection,object] , [atomic,collection,object])

key – not unique

value

```
val fluege =
 List( ("Thomas", "Berlin"),("Mark", "Paris"),("Thomas", "Madrid"))
val pairRDD = sc.parallelize(fluege)

pairRDD.filter(_._1 == "Thomas")
 .collect
 .foreach(t => println(t._1 + " flog nach " + t._2))
```

Why Use Pair RDD's_

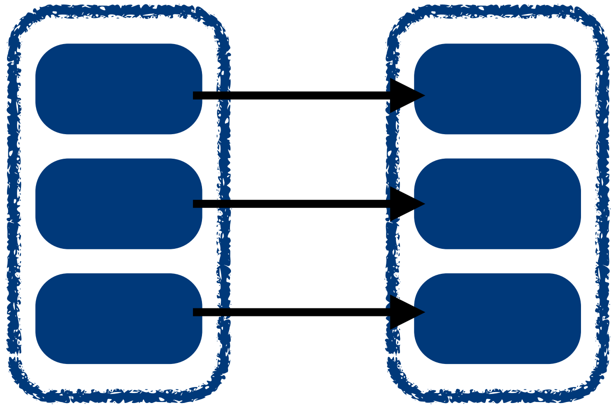
- **Parallelization!**
  - *keys are use for partitioning*
  - *pairs with different keys are distributed across the cluster*
- **Efficient processing of**
  - *aggregate by key*
  - *group by key*
  - *sort by key*
  - *joins, union based on keys*

Special OPs for PairRDDs_

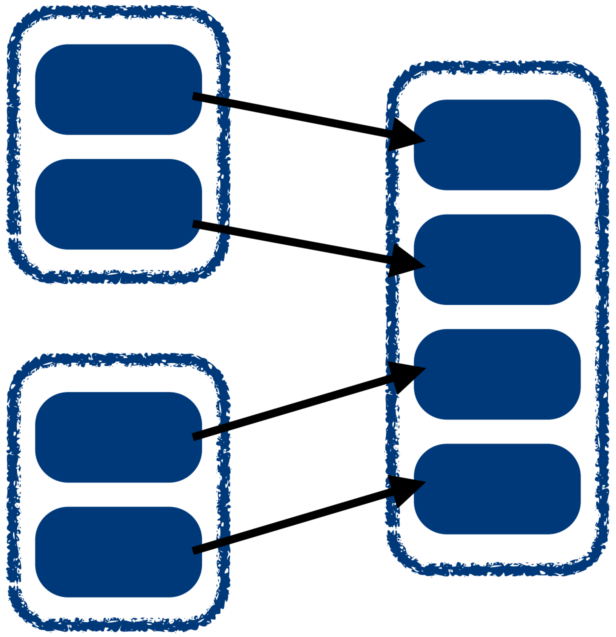
```
keys(), values()
mapValues(func), flatMapValues(func)
lookup(key), collectAsMap(), countByKey()

reduceByKey(func), foldByKey(zeroValue)(func)
groupByKey(), cogroup(otherDataset)
sortByKey([ascending])
join(otherDataset), leftOuterJoin(otherDataset),
rightOuterJoin(otherDataset)
union(otherDataset), substractByKey(otherDataset)
```

RDD Dependencies_

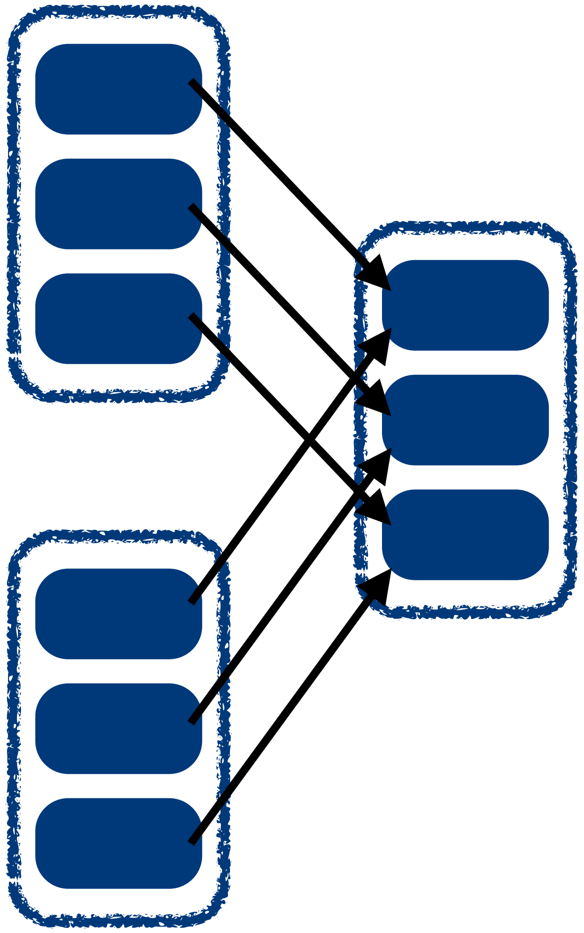
„Narrow“ (pipeline-able)



map, filter

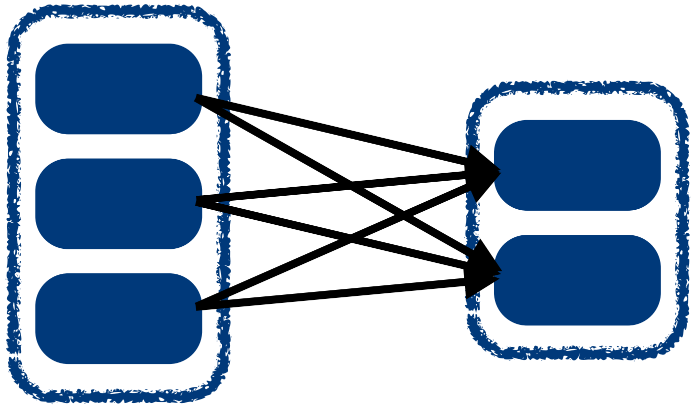


union

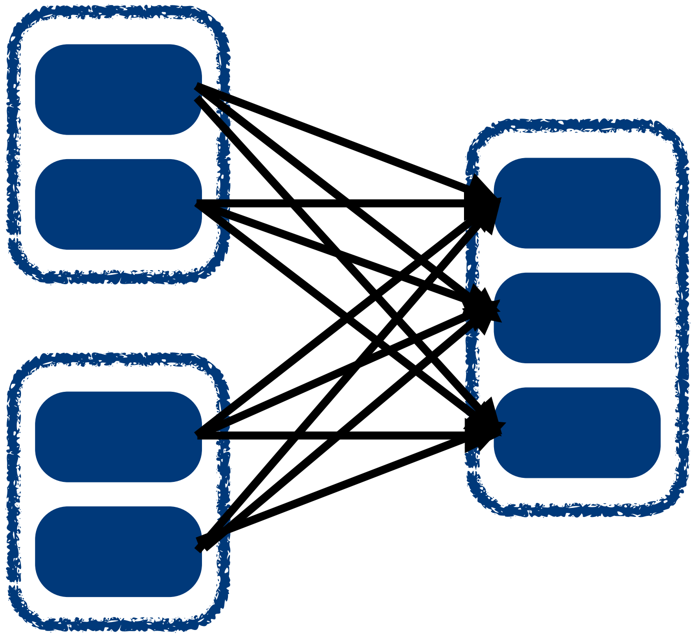


join on co partitioned data

„Wide“ (shuffle)



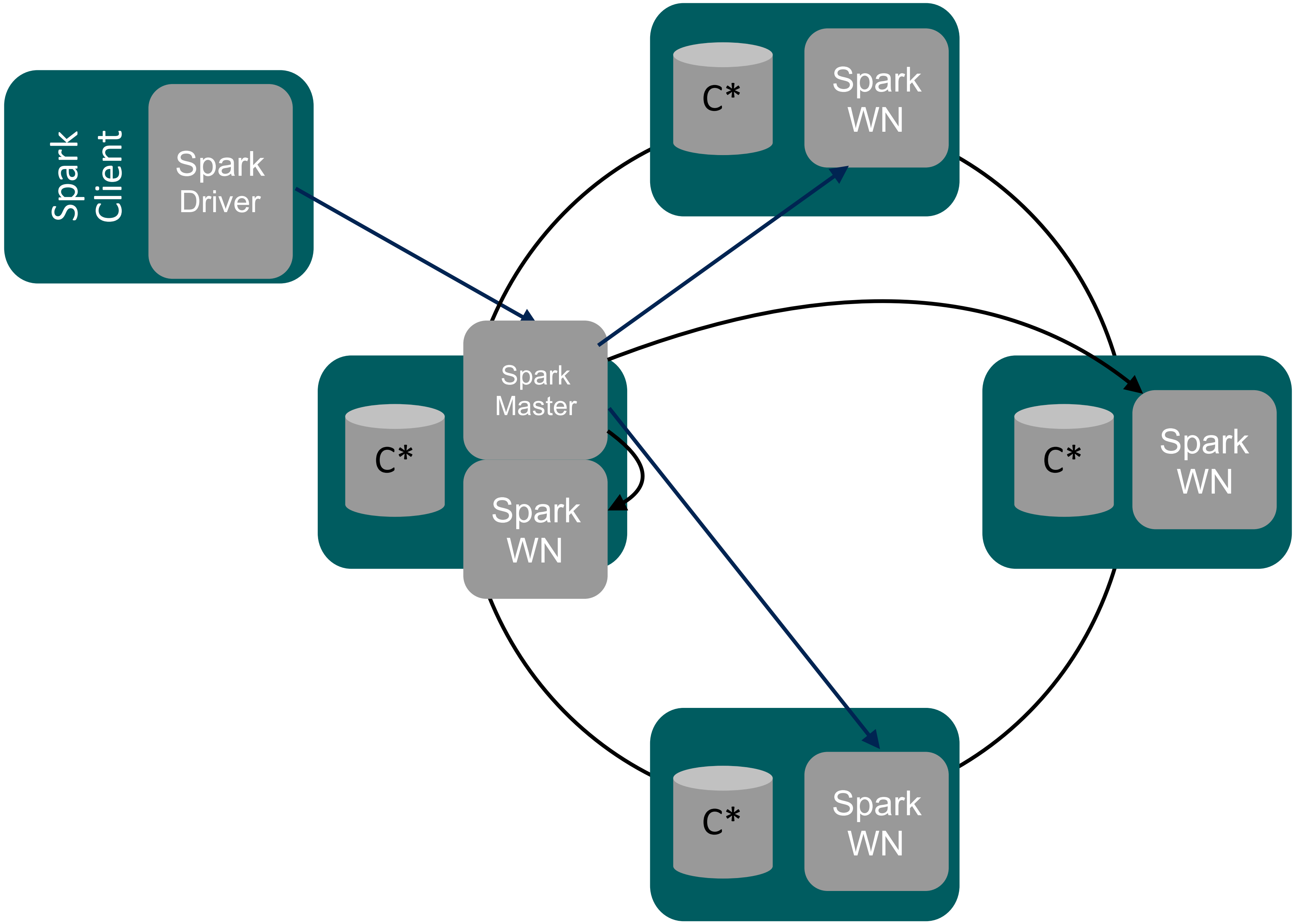
groupBy
on non partitioned data



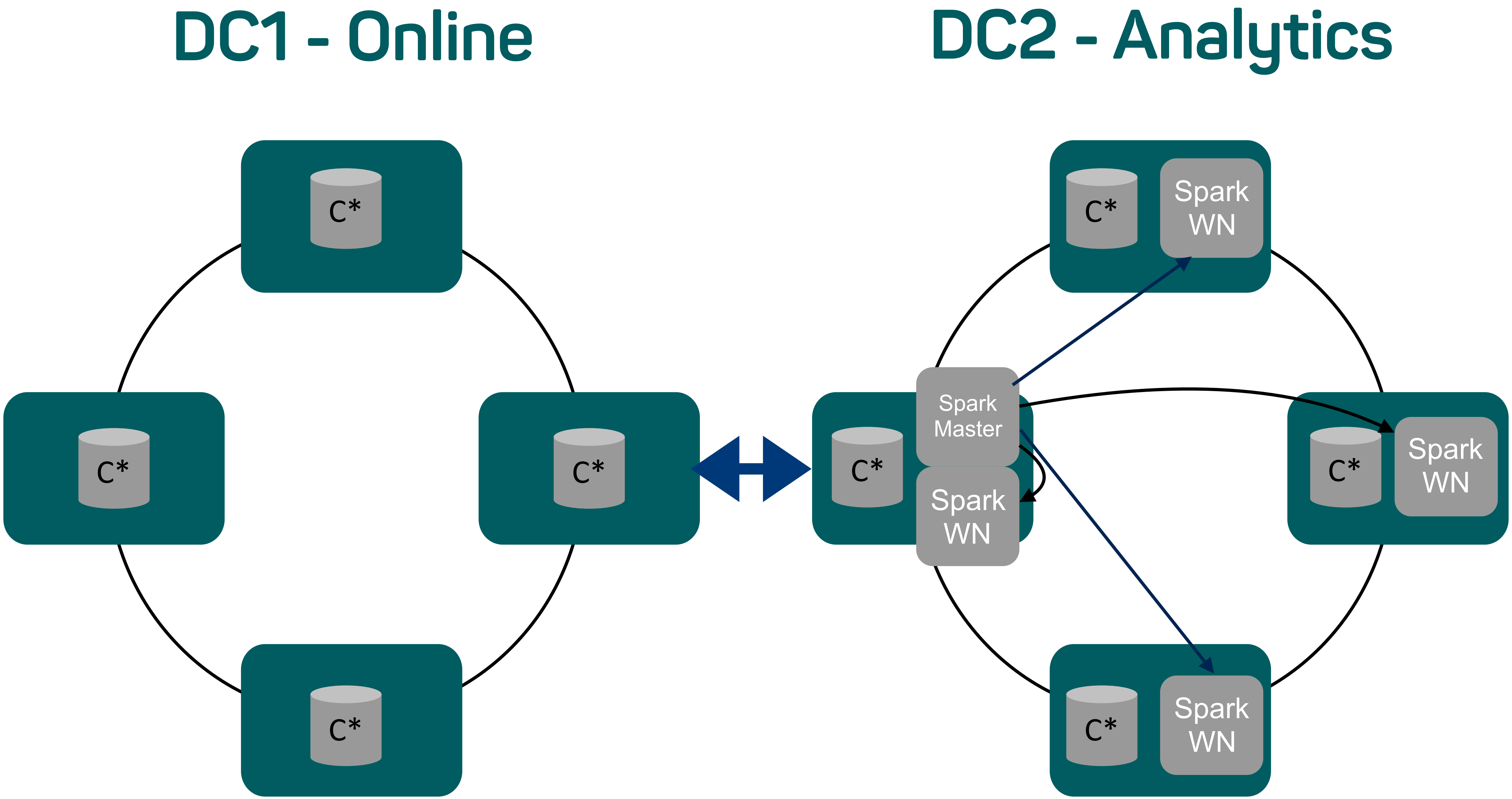
join on non co partitioned data

Spark & Cassandra

Use Spark And Cassandra In A Cluster_



Two Datacenter - Two Purposes_



Connecting Spark With Cassandra_

- Spark Cassandra Connector by Datastax
  - *https://github.com/datastax/spark-cassandra-connector*
- Cassandra tables as Spark RDD (read & write)
- Mapping of C* tables and rows onto Java/Scala objects
- Server-Side filtering („where“)
- Compatible with
  - *Spark ≥ 0.9*
  - *Cassandra ≥ 2.0*
- Clone & Compile with SBT or download at Maven Central

Read A Cassandra Table_

Read complete table

```
val movies = sc.cassandraTable("movie","movies")
// returns CassandraRDD[CassandraRow]
```

Read selected columns

```
val movies = sc.cassandraTable("movie","movies").select("title","year")
```

Filter rows

```
val movies = sc.cassandraTable("movie","movies").where("title = 'Die
Hard'")
```

Access Columns in Result Set

```
movies.collect.foreach(r => println(r.get[String]("title")))
```

Read A Cassandra Table_

Read As Tuple

```
val movies =
sc.cassandraTable[(String,Int)]("movie","movies")
.select("title","year")

val movies =
sc.cassandraTable("movie","movies")
.select("title","year")
.as((_: String, _:Int))

// both result in a CassandraRDD[(String,Int)]
```

Read As Case Class

```
case class Movie(title: String, year: Int)

sc.cassandraTable[Movie]("movie","movies").select("title","year")

sc.cassandraTable("movie","movies").select("title","year").as(Movie)
```

Write Table_

- **Every RDD can be saved**
  - *Using Tuples*

```
val tuples = sc.parallelize(Seq(("Hobbit",2012),("96 Hours",2008)))
tuples.saveToCassandra("movie","movies", SomeColumns("title","year")
```

  - *Using Case Classes*

```
case class Movie (title:String, year: int)
val objects =
   sc.parallelize(Seq(Movie("Hobbit",2012),Movie("96 Hours",2008)))
objects.saveToCassandra("movie","movies")
```

Pair RDDs With Cassandra_

```
// Load and format as Pair RDD
val pairRDD = sc.cassandraTable("movie","director")
  .map(r => (r.getString("country"),r))

// Directors / Country, sorted
pairRDD.mapValues(v => 1).reduceByKey(_+_)
  .sortBy(-_._2).collect.foreach(println)

// or, unsorted
pairRDD.countByKey().foreach(println)

// All Countries
pairRDD.keys()
```

director		
name	*text*	*K*
country	*text*	

Using Data Locality With Cassandra_

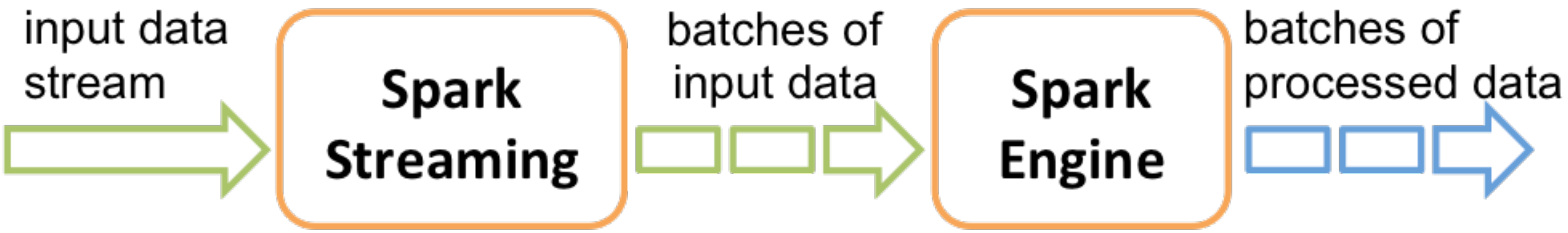
- **Automatically on read**
- **Not automatically on write**
  - *No Shuffling Spark Operations -> Writes are local*
  - *Shuffeling Spark Operartions*
    - *Fan Out writes to Cassandra*
    - *repartitionByCassandraReplica("keyspace", "table") before write*
- **Joins with data locality**

```
sc.cassandraTable[CassandraRow](KEYSPACE, A)
.repartitionByCassandraReplica(KEYSPACE, B)
.joinWithCassandraTable[CassandraRow](KEYSPACE, B)
.on(SomeColumns("id"))
```

Spark Streaming

Stream Processing With Spark Streaming_

- Real Time Processing using micro batches
- Supported sources: TCP, S3, Kafka, Twitter,..
- Data as Discretized Stream (DStream)
- Same programming model as for batches
- All Operations of the GenericRDD & SQL & MLLib
- Stateful Operations & Sliding Windows



Spark Streaming - Example_

```
import org.apache.spark.streaming._

val ssc = new StreamingContext(sc,Seconds(1))

val stream = ssc.socketTextStream("127.0.0.1",9999)

stream.map(x => 1).reduce(_ + _).print()

ssc.start()

// await manual termination or error
ssc.awaitTermination()

// manual termination
ssc.stop()
```

Create an Application

Scala Application_

- Normal Scala Application
- SBT as build tool
- source in *src/main/scala-2.10*
- *assembly.sbt* in root and project directory
- *build.sbt* in root directory

```
libraryDependencies += "com.datastax.spark" % "spark-cassandra-connector" % "1.3.0"
libraryDependencies += "org.apache.spark" % "spark-core" % "1.3.1" % "provided"
libraryDependencies += "org.apache.spark" % "spark-mllib_2.10" % "1.3.1" % "provided"
libraryDependencies += "org.apache.spark" % "spark-streaming_2.10" % "1.3.1" %
"provided"
```

- *sbt assembly* to build

Submit Spark Application on DCOS_

```
dcos spark run --submit-args="--class org.apache.spark.examples.SparkPi https://downloads.mesosphere.com.s3.amazonaws.com/assets/spark/spark-examples_2.10-1.5.0.jar"
```

Spark Exercises

Spark Exercices_

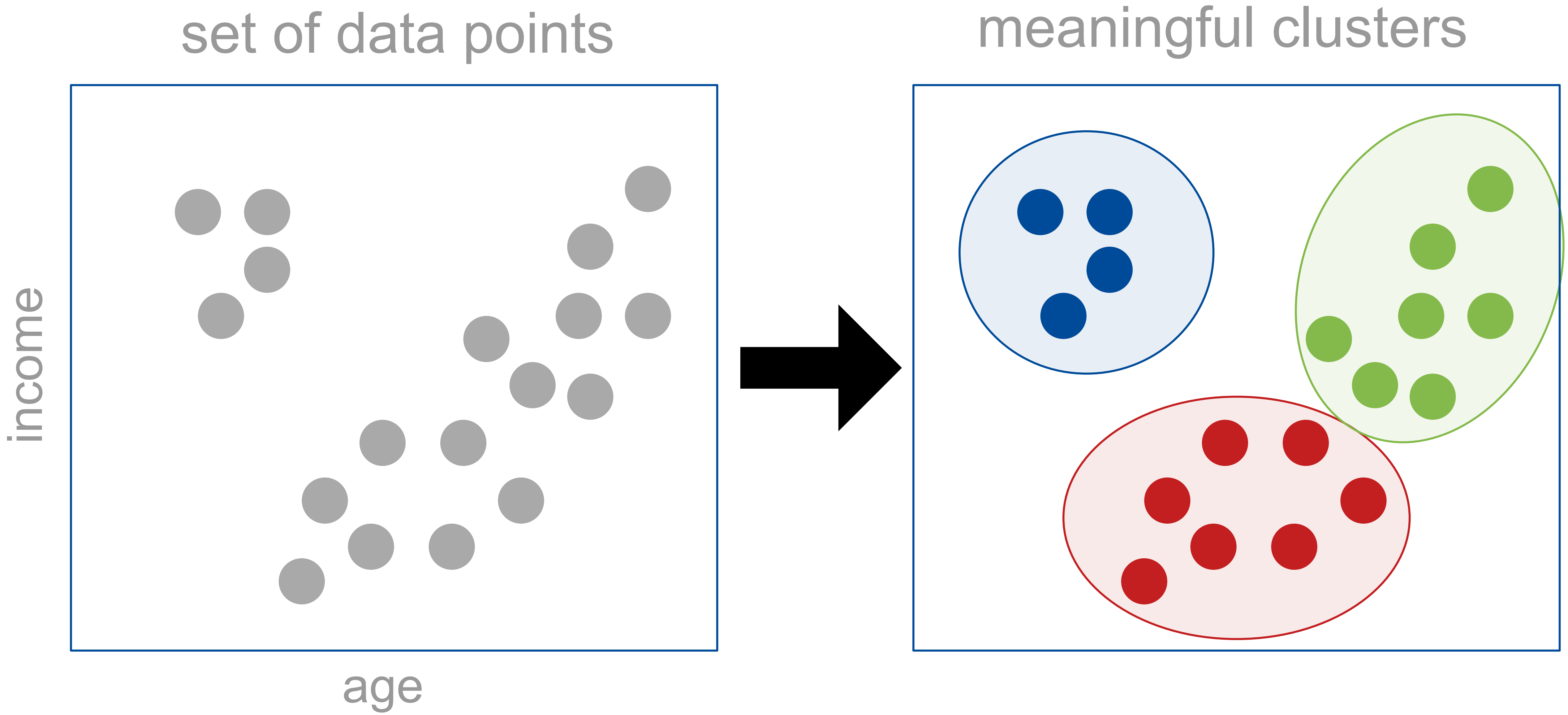
- **Streaming Application 1**
  - *Simple Count & Aggregate*
  - *Save to Cassandra*
- **Streaming Application 2**
  - *State & Window Processing*
  - *Advanced*
- **Batch Application**
  - *Read from Cassandra*
  - *Filter Data*

Spark MLLib

Spark MLLib_

- Fully integrated in Spark
  - *Scalable*
  - *Scala, Java & Python APIs*
  - *Use with Spark Streaming & Spark SQL*
- Packages various algorithms for machine learning
- Includes
  - *Clustering*
  - *Classification*
  - *Prediction*
  - *Collaborative Filtering*
- Still under development
  - *performance, algorithms*

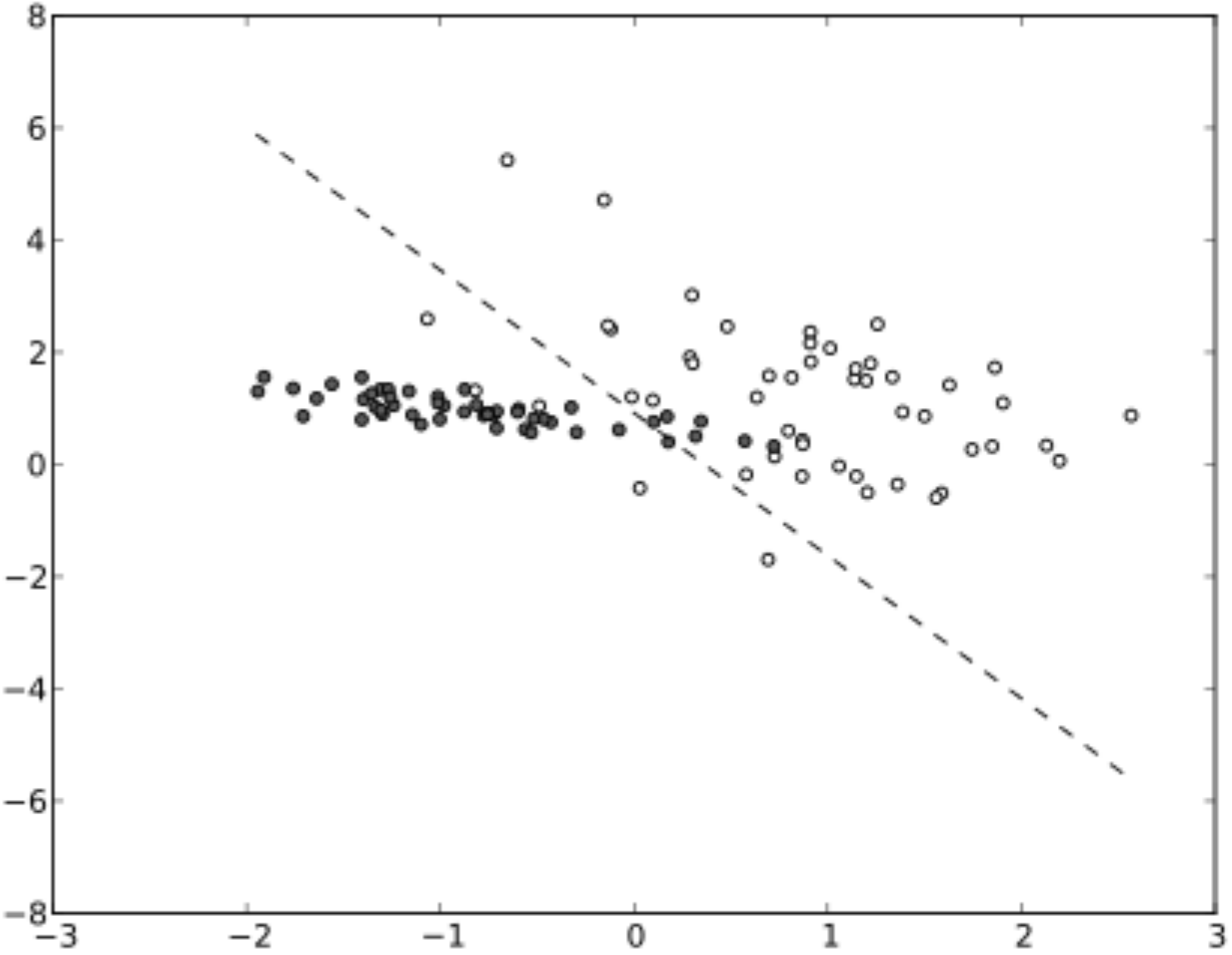
MLLib Example - Clustering_



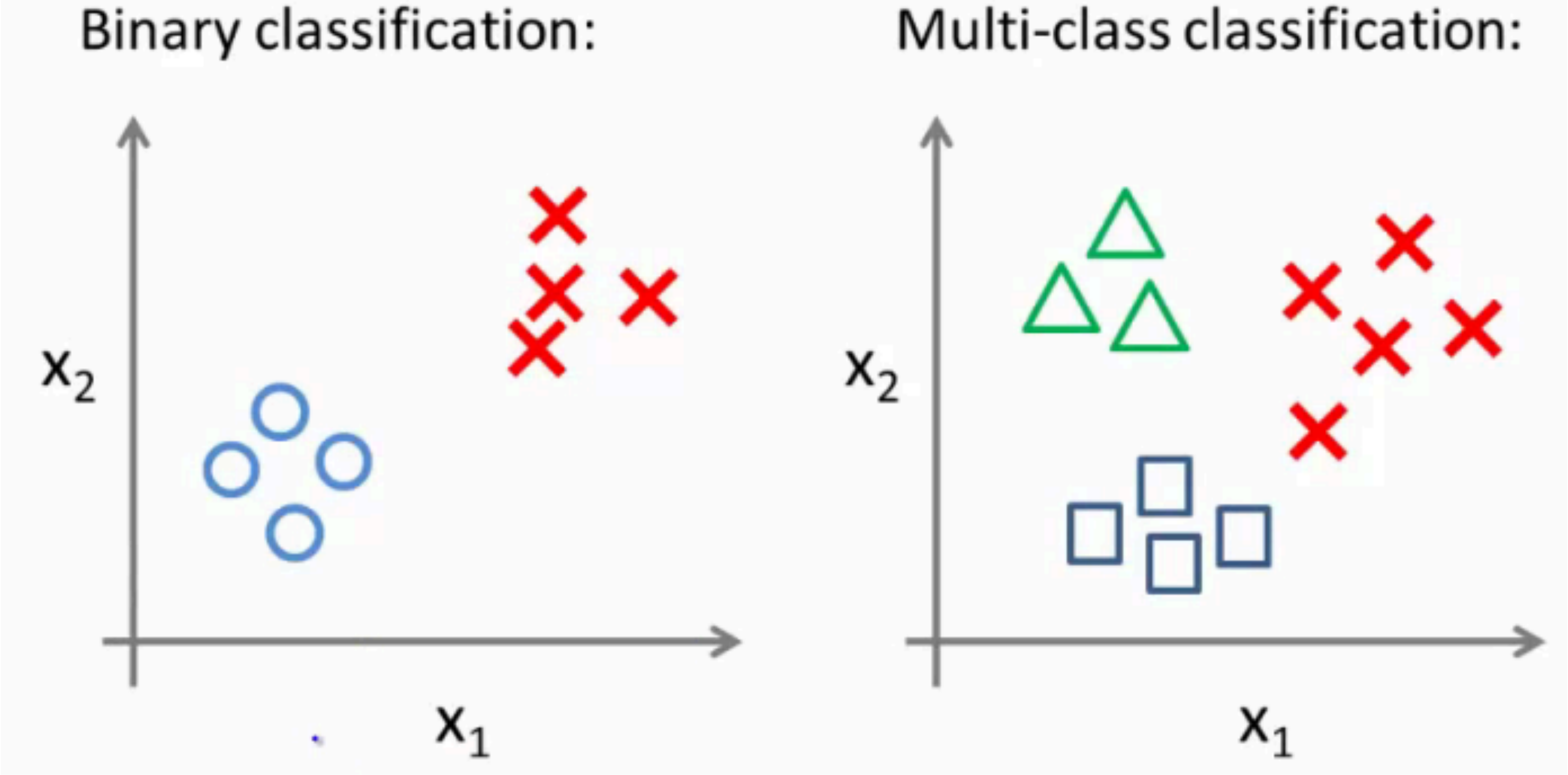
MLLib Example - Clustering (using KMeans)_

```
// Load and parse data
val data = sc.textFile("data/mllib/kmeans_data.txt")
val parsedData = data
.map(s => Vectors.dense(s.split(' ')
.map(_.toDouble))).cache()

// Cluster the data into 3 classes using KMeans with 20
iterations
val clusters = KMeans.train(parsedData, 2, 20)

// Evaluate clustering by computing Sum of Squared Errors
val SSE = clusters.computeCost(parsedData)
println("Sum of Squared Errors = " + WSSSE)
```

MLLib Example - Classification_



MLLib Example - Classification_



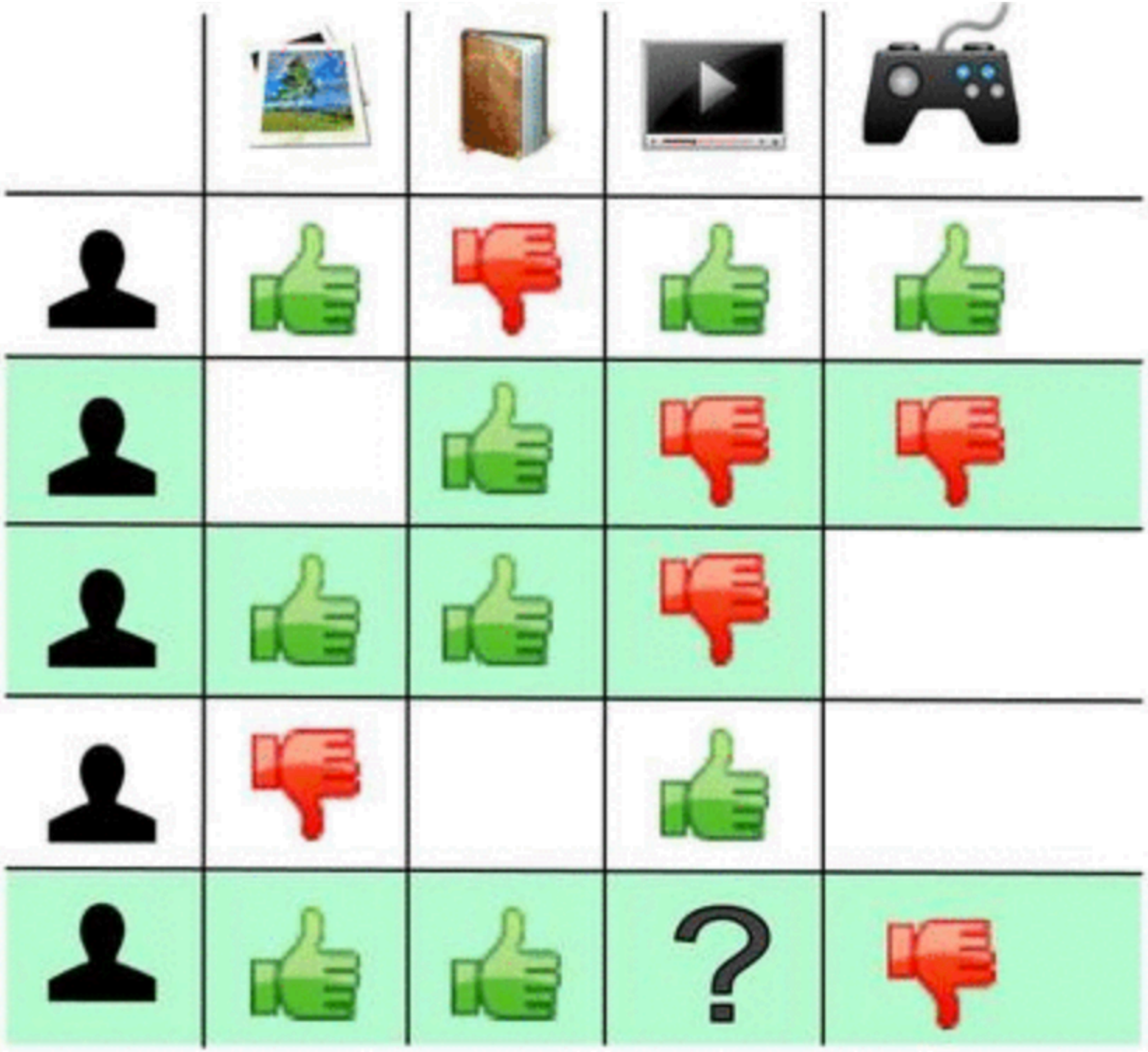
MLLib Example - Classification (Linear SVM)_

```
// Load training data in LIBSVM format.
val data =
 MLUtils.loadLibSVMFile(sc, "sample_libsvm_data.txt")

// Split data into training (60%) and test (40%).
val splits = data.randomSplit(Array(0.6, 0.4), seed = 11L)
val training = splits(0).cache()
val test = splits(1)

// Run training algorithm to build the model
val numIterations = 100
val model = SVMWithSGD.train(training, numIterations)
```

MLLib Example - Classification (Linear SVM)_

```
// Compute raw scores on the test set.
val scoreAndLabels = test.map { point =>
  val score = model.predict(point.features)
  (score, point.label)
}

// Get evaluation metrics.
val metrics = new
BinaryClassificationMetrics(scoreAndLabels)
val auROC = metrics.areaUnderROC()
println("Area under ROC = " + auROC)
```

MLLib Example - Collaborative Filtering_



	🖼	📖	▶	🎮
👤	👍	👎	👍	👍
👤		👍	👎	👎
👤	👍	👍	👎	
👤	👎		👍	
👤	👍	👍	?	👎

MLLib Example - Collaborative Filtering using ALS_

```
// Load and parse the data (userid,itemid,rating)
val data = sc.textFile("data/mllib/als/test.data")
val ratings = data.map(_.split(',') match
 {
  case Array(user, item, rate) => Rating(user.toInt,
  item.toInt, rate.toDouble)
 })

// Build the recommendation model using ALS
val rank = 10
val numIterations = 20
val model = ALS.train(ratings, rank, numIterations, 0.01)
```

MLLib Example - Collaborative Filtering using ALS_

```
// Evaluate the model on rating data
val usersProducts = ratings.map {
 case Rating(user, product, rate) => (user, product) }

val predictions = model.predict(usersProducts).map {
 case Rating(user, product, rate) => ((user, product), rate)
}

val ratesAndPredictions = ratings.map {
 case Rating(user, product, rate) =>((user, product), rate)}
 .join(predictions)
val MSE = ratesAndPredictions.map {
 case ((user, product), (r1, r2)) => val err = (r1 - r2);
 err * err }.mean()
println("Mean Squared Error = " + MSE)
```

Use Cases

Data Loading

- In particular for huge amounts of external data
- Support for CSV, TSV, XML, JSON und other

```
case class User (id: java.util.UUID, name: String)

val users = sc.textFile("users.csv")
.repartition(2*sc.defaultParallelism)
.map(line => line.split(",") match { case Array(id,name) =>
User(java.util.UUID.fromString(id), name)})

users.saveToCassandra("keyspace","users")
```

Validation & Normalization

Validate consistency in a Cassandra database

- syntactic
  - *Uniqueness (only relevant for columns not in the PK)*
  - *Referential integrity*
  - *Integrity of the duplicates*
- semantic
  - *Business- or Application constraints*
  - *e.g.: At least one genre per movies, a maximum of 10 tags per blog post*

Analyses (Joins, Transformations,..)

- Modelling, Mining, Transforming, ...
- Use Cases
  - *Recommendation*
  - *Fraud Detection*
  - *Link Analysis (Social Networks, Web)*
  - *Advertising*
  - *Data Stream Analytics (Spark Streaming)*
  - *Machine Learning (Spark ML)*

Schema Migration

- Changes on existing tables
  - *New table required when changing primary key*
  - *Otherwise changes could be performed in-place*
- Creating new tables
  - *data derived from existing tables*
  - *Support new queries*
- Use the CassandraConnectors in Spark

Questions?

Matthias Niehoff,
IT-Consultant

codecentric AG
Zeppelinstraße 2
76185 Karlsruhe, Germany

mobil: +49 (0) 172.1702676
matthias.niehoff@codecentric.de

www.codecentric.de
blog.codecentric.de

matthiasniehoff