Distributed Data Management
Spark Batch Processing

Thorsten Papenbrock

F-2.04, Campus II

Hasso Plattner Institut

Installation

Installation (Development)

- Checkout the tutorial/homework project
- Install IntelliJ IDEA
- Import the project into IntelliJ
- Load the homework data
- Run the tutorial
- Solve the DDM homework
- Build your final jar file

More details on the next slide!

Installation (Deployment / Interactive Shell)

- Install Java, Scala and sbt.
- Download and install Spark: https://spark.apache.org/downloads.html
- Start the master/slave daemon processes: https://spark.apache.org/docs/latest/spark-standalone.html

Checkout the tutorial/homework project

- git clone git@github.com:HPI-Information-Systems/spark-tutorial.git

Install IntelliJ IDEA

- Use your @student.hpi.uni-potsdam.de email for a free ultimate license
- Select the Scala and sbt plugins to be installed during the installation; otherwise you need to install them later

Import the project into IntelliJ

- Start Intellij, select "Open or Import" and then select the spark-tutorial/build.sbt file and click OK
- Click Open as Project (IntelliJ should automatically recognize the sbt project nature; note that the initialization might take a while for missing downloads and indexing; if IntelliJ does not automatically download the depencencies, you can open the sbt tab on the right edge of the screen and click Reload all sbt projects)

Load the homework data

- Download the tpch.zip from https://hpi.de/fileadmin/user_upload/fachgebiete/naumann/lehre/WS2017/DDA/TPCH.zip
- Extract the zipped content into "spark-tutorial/data", which should create the folder "spark-tutorial/data/TPCH" with 7 csv files

Run the tutorial

- Switch to your IntelliJ window and open "spark-tutorial/src/main/scala/de/hpi/spark_tutorial" in the project tree view
- Right-click "SimpleSpark.scala" and click "Run SimpleSpark"

Solve the DDM homework

- Remove the "Tutorial" and "LongestCommonSubstring" calls
- Implement the Sindy algorithm

Build your final jar file

- Click "Run" -> "Edit Configurations" -> "+" -> "SBT Task"
- Enter a name, such as "SparkTutorial assembly" and the task "clean assembly", then click "OK"
- You can now switch to and run the "SparkTutorial assembly" target in the top right IntelliJ bar
- Find your fat-jar in "spark-tutorial/target/scala-2.12/SparkTutorialSBT-assembly-0.1.jar"

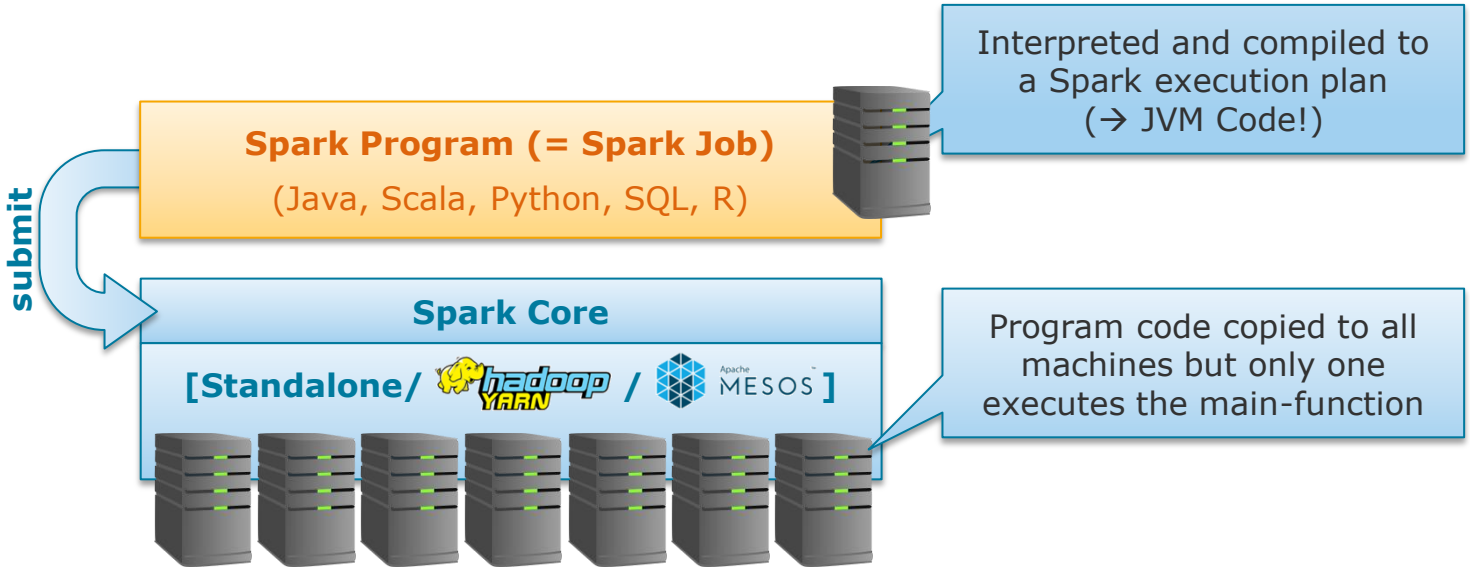
Key-Facts

Spark

- Written in Scala
- Offers Java, Scala, Python, SQL, and R bindings
- Uses (inter alia) Akka under the hood
- Can access data in Cassandra, HBase, Hive, and any Hadoop data source

- Runs standalone (own cluster and resource management) or on managed clusters (Hadoop YARN or Apache Mesos)

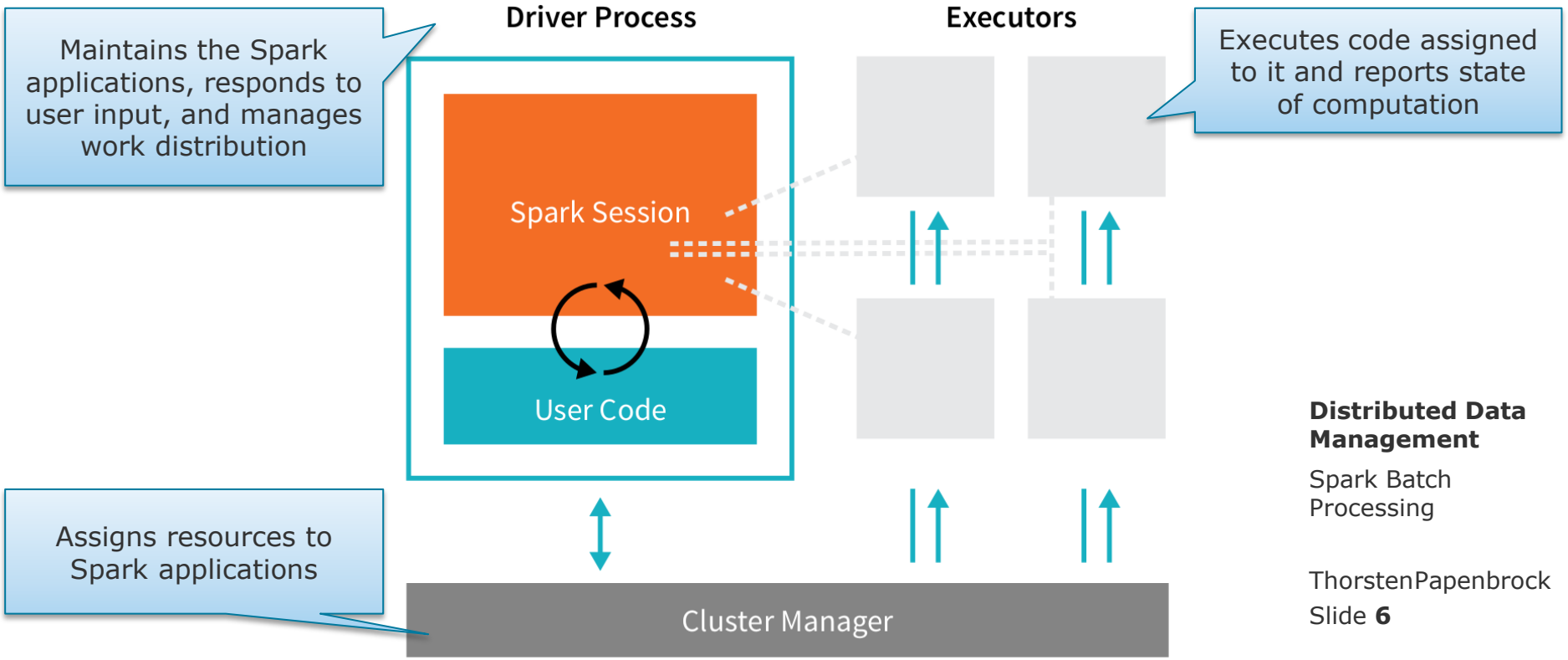
Application



Setup

- On all nodes: Install and configure Java, Scala, SSH, Spark, (Hadoop, Kafka, …)
- See e.g.: https://medium.com/ymedialabs-innovation/apache-spark-on-a-multi-node-cluster-b75967c8cb2b

Application



Application

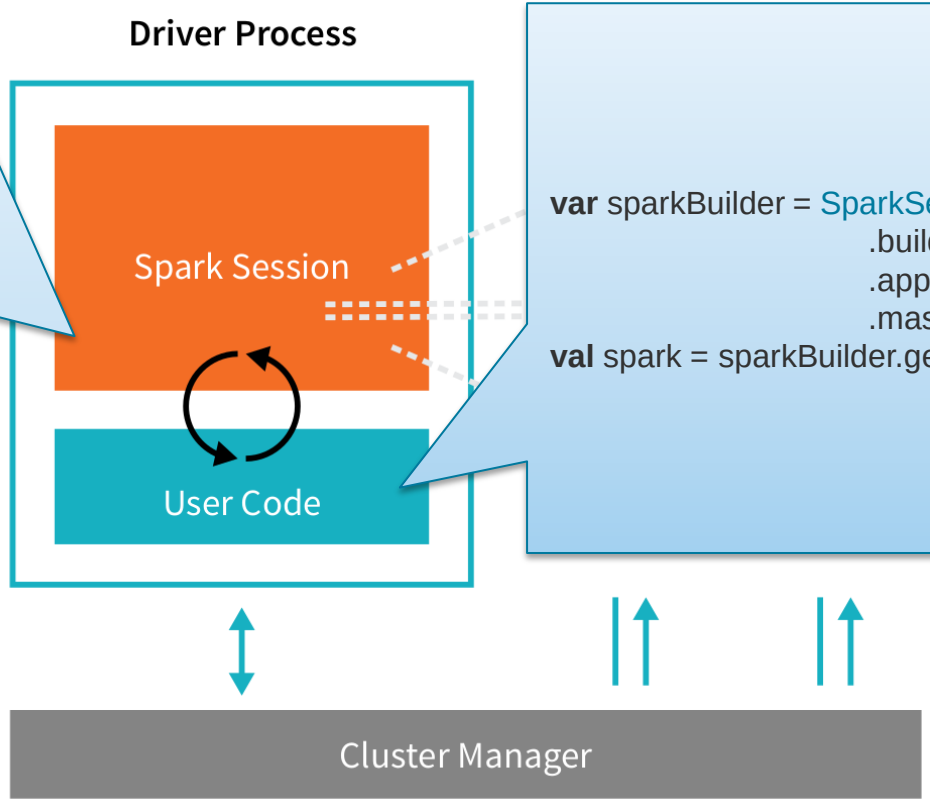
For Spark <2.0 experts:
Where are the contexts?

`SparkContext` object
(connection to a Spark execution environment)

`SQLContext` object
(connection to SparkSQL)

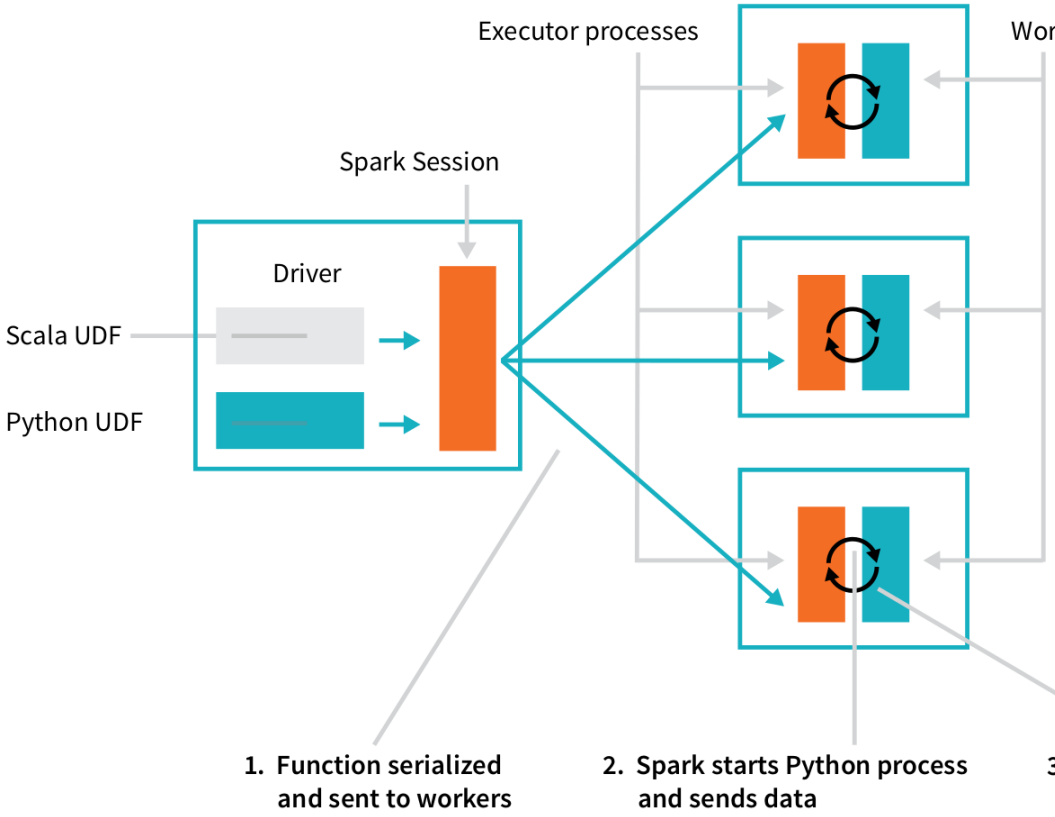
`HiveContext` object
(connection to Hive stores)

Now accessible via `SparkSession` object




```
var sparkBuilder = SparkSession
                   .builder()
                   .appName("ExampleSpark")
                   .master("local[4]")  //local, 4 cores
val spark = sparkBuilder.getOrCreate()
```

UDFs: Python vs. Scala/Java



Writing UDFs in Python is possible but costs performance because UDFs cannot be compiled to JVM code:

- Python processes need to be started
- Python processes compete for memory/CPU with the JVM processes that run anyways
- Data needs to be copied between processes (serialization; inter-process communication)

Application Submit

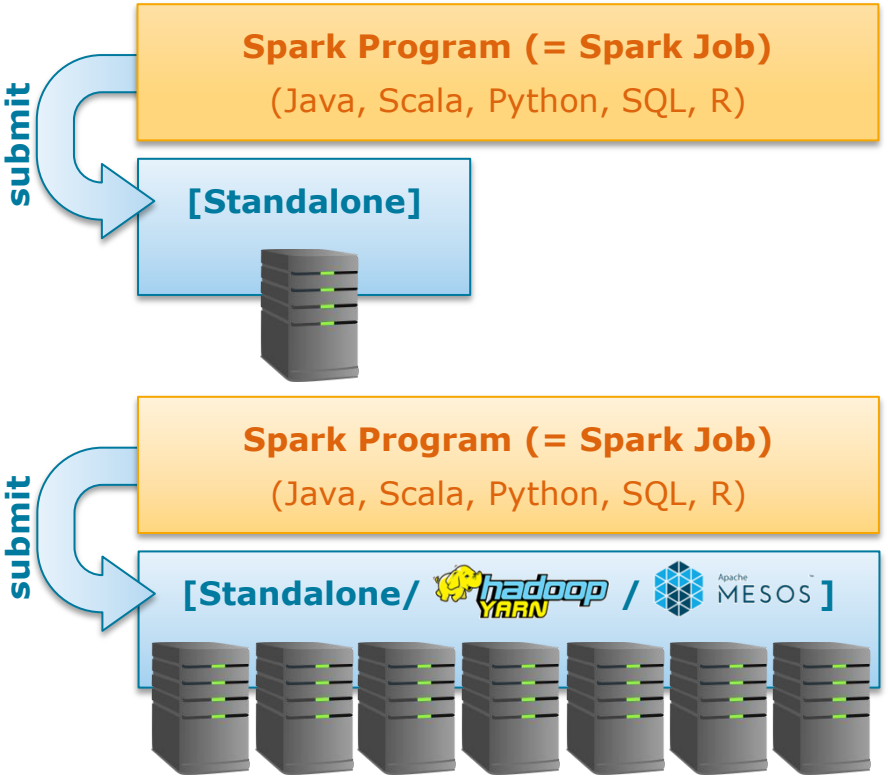
Local Execution

- Execute the `main()`-function

Submit to Cluster

- Build a fat jar including all dependencies
- Commit to cluster via spark-submit script:

```
./bin/spark-submit \
   --class <main-class> \
   --master <master-url> \
   --deploy-mode <deploy-mode> \
   --conf <key>=<value> \
   ... # other options
   <application-jar> \
      [application-arguments]
```



https://spark.apache.org/docs/latest/submitting-applications.html

Application Submit

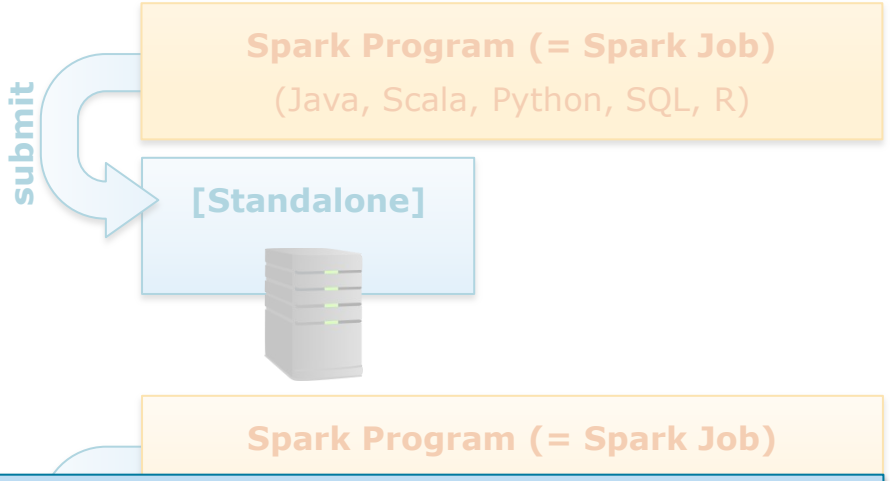
Local Execution

- Execute the `main()`-function

Submit to Cluster

- Build a fat jar including all dependencies
- Commit to cluster via spark-submit script:

```
./bin/spark-submit \
   --class de.hpi.time_series_mining.Main \
   --master spark://192.168.0.18:7077 \
   --deploy-mode client \
   --executor-memory 16G \
   --total-executor-cores 32 \
   /jobs/time_series_mining.jar \
      /data/time_series.csv \
      /results/time_series \
      60
```



`--master local[8]`: local with 8 worker threads
`--master yarn`: on YARN
`--master mesos://192.168.0.18:7077`: on MESOS

`client`: local machine = driver process
`cluster`: one worker = driver process

`file`/absolute path files are served by the driver's HTTP file server.
`hdfs`/`http`/`https`/`ftp` files are pulled from remote locations.
`local` files are expected to exist on every worker.

https://spark.apache.org/docs/latest/submitting-applications.html

Tools

Spark Shell

- Executing `./bin/spark-shell` starts the Scala console with an interactive Spark session.

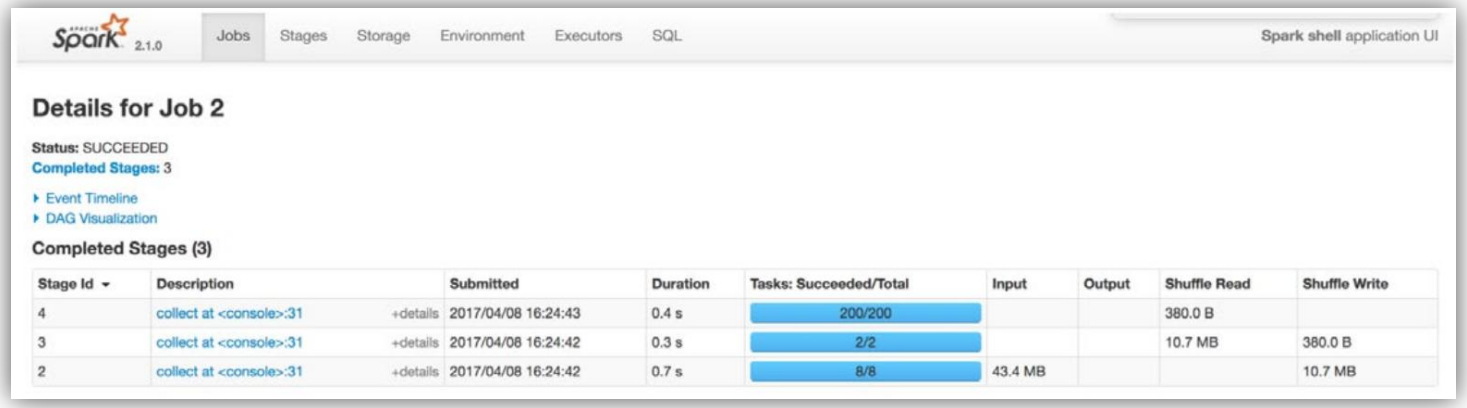
```
19:01:56: >spark-shell
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
17/06/22 19:05:08 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java cl
asses where applicable
17/06/22 19:05:16 WARN ObjectStore: Version information not found in metastore. hive.metastore.schema.verification is no
t enabled so recording the schema version 1.2.0
17/06/22 19:05:16 WARN ObjectStore: Failed to get database default, returning NoSuchObjectException
17/06/22 19:05:17 WARN ObjectStore: Failed to get database global_temp, returning NoSuchObjectException
Spark context Web UI available at http://192.168.1.10:4040
Spark context available as 'sc' (master = local[*], app id = local-1498154709976).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.1.1
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_131)
Type in expressions to have them evaluated.
Type :help for more information.

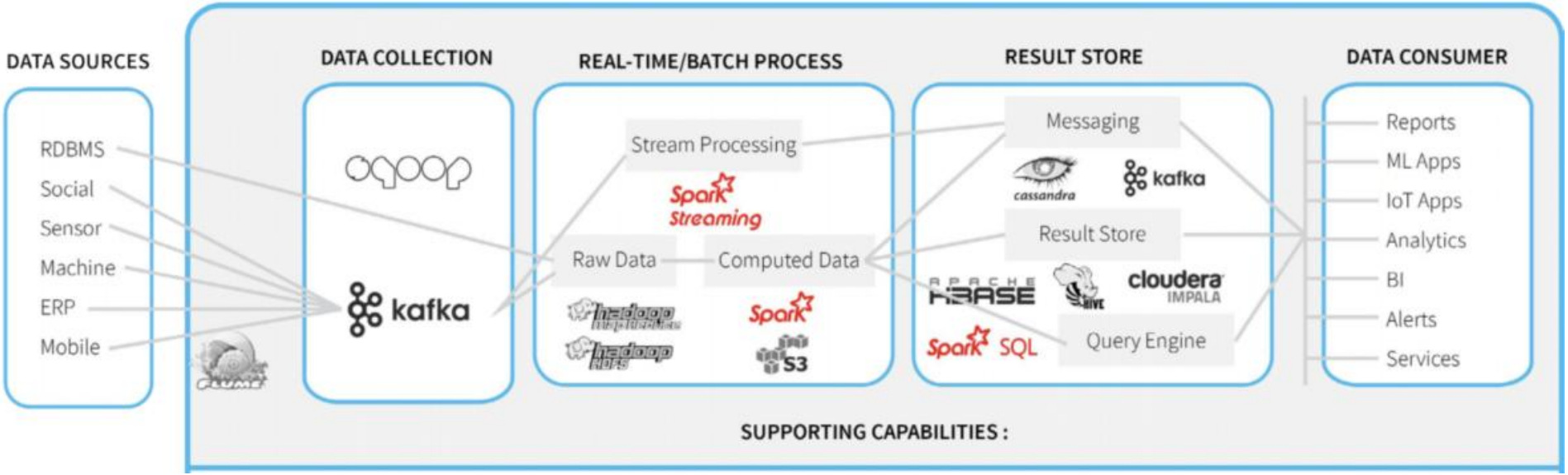
scala>
```

Spark UI

- Accessing `http://localhost:4040` in a web browser opens the Spark UI that monitors the execution of Spark jobs.

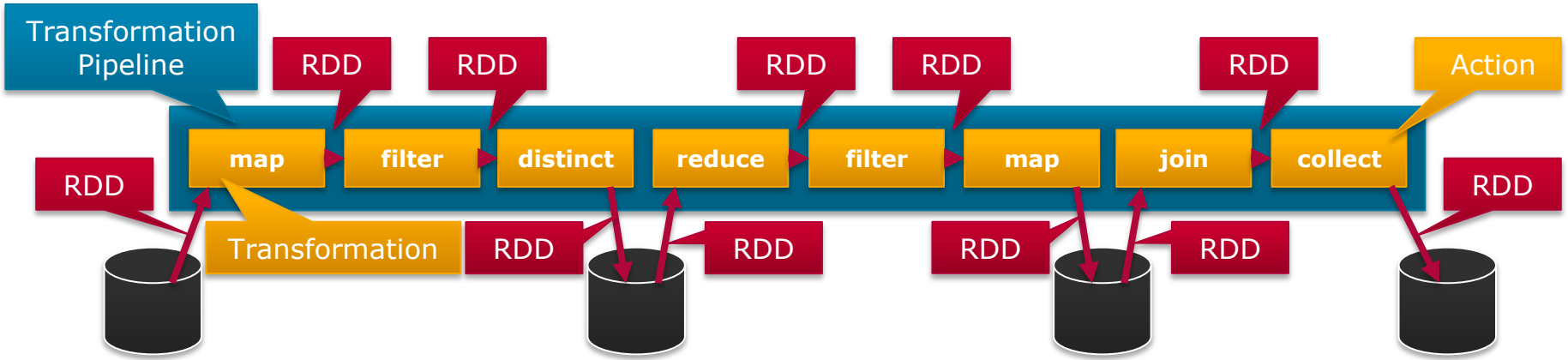


Typical Architectures



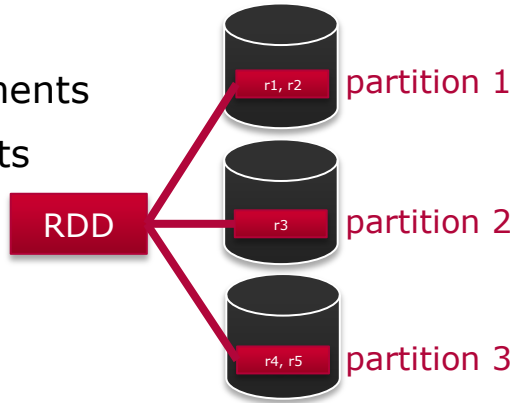
https://unraveldata.com/spark-performance-management-2/

Distributed Data



Resilient Distributed Dataset (RDD)

- An immutable distributed collection of data elements
- Partitioned across nodes; protected against faults
- Can be operated in parallel
- Offers a low-level API with transformation functions and actions



Distributed Data

	RDD	Dataset	DataFrame
Class	`RDD[T]`	`Dataset[T]`	`DataFrame`
Relationship	-	Based on `RDD[T]`	Alias for `Dataset[Row]`
Semantic	A collection of `T` objects	A collection of `T` objects	A collection of `Row` objects
Typing		Example: Tuples with **explicit** filed types dataset.map((a,b,c) => a + 1) // types of a, b, c are known	Example: Tuples with **implicit** field types dataframe.map(r => r.getInt(0) + 1) // types of r's fields are unknown and need casting
Type safety			
Serialization			
Optimization			
Languages			
Special			

```
val rdd = dataset.toJavaRDD()
val dataset = dataframe.as[(String,String)]
val dataframe = dataset.toDF()
val dataframe = rdd.toDF()
```

Distributed Data

	RDD	Dataset	DataFrame
Class	`RDD[T]`	`Dataset[T]`	`DataFrame`
Relationship	-	Based on `RDD[T]`	Alias for `Dataset[Row]`
Semantic	A collection of `T` objects	A collection of `T` objects	A collection of `Row` objects
Typing	Strongly typed	Strongly typed Java beans (no-arg, serializable, get-/setters)	Un-typed `Row` objects (similar to JDBC's `ResultSet`)
Type safety			
Serialization			
Optimization			
Languages			
Special			

Distributed Data

	RDD	Dataset	DataFrame
Class	`RDD[T]`	`Dataset[T]`	`DataFrame`
Relationship	-	Based on `RDD[T]`	Alias for `Dataset[Row]`
Semantic	A collection of `T` objects	A collection of `T` objects	A collection of `Row` objects
Typing	Strongly typed	Strongly typed Java beans (no-arg, serializable, get-/setters)	Un-typed `Row` objects (similar to JDBC's `ResultSet`)
Type safety	Compile-time checking	Compile-time checking	Runtime checking
Serialization	Java `Serializable`	Implicit `Encoder` functions to convert `T` objects to tabular format; Tungsten to generate byte-code	Dynamic byte-code generation via Tungsten execution backend
Optimization	-	Catalyst query optimizer, Tungsten execution engine, and off heap storage mechanism	Catalyst query optimizer, Tungsten execution engine, and off heap storage mechanism
Languages	Java, Scala, Python, R	Java, Scala (as of version 2.1.1)	Java, Scala, Python, R
Special	Low level manipulation options (e.g. for number of partitions)	More transformations and actions	Special support for R and SQL (grouping, sorting, aggregations, …)

complex business logic

querying and transforming

Distributed Data

	RDD	Dataset	DataFrame
Class	`RDD[T]`	`Dataset[T]`	`DataFrame`
Relationship	-	Based on `RDD[T]`	Alias for `Dataset[Row]`
Semantic	A collection of `T` objects	A collection of `T` objects	A collection of `Row` objects
Typing	Strongly ty	beans , get-/setters)	
Type safety	Compile-tin	ng	
Serialization	Java `Seria`	ctions to tabular format; e byte-code	
Optimization	-	nizer, Tungsten nd off heap	
Languages	Java, Scala, Python, R	Java, Scala (as of version 2.1.1)	Java, Scala, Python, R
Special	Low level manipulation options (e.g. for number of partitions)	More transformations and actions	Special support for R and SQL (grouping, sorting, aggregations, …)

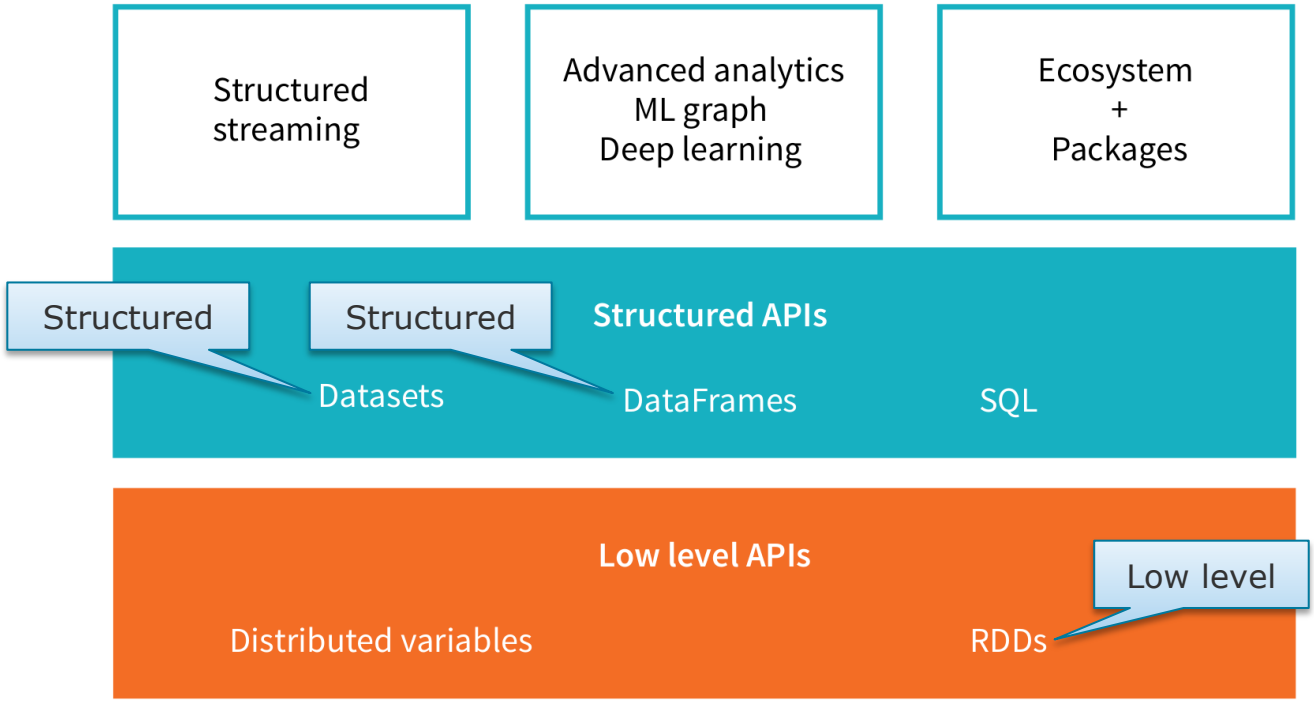
```
val sum = data.as[String]
  .filter(value => value == null)
  .flatMap(value => value.split("\\s+"))
  .map(value => (value,1))
  .reduceByKey(_+_)
  .collect()
```

```
val result = flightData
  .groupBy("DESTINATION")
  .sum("FLIGHTS")
  .sort(desc("sum(FLIGHTS)"))
  .select(
    col("DESTINATION"),
    col("sum(FLIGHTS)").as("sum"))
  .collect()
```

complex business logic

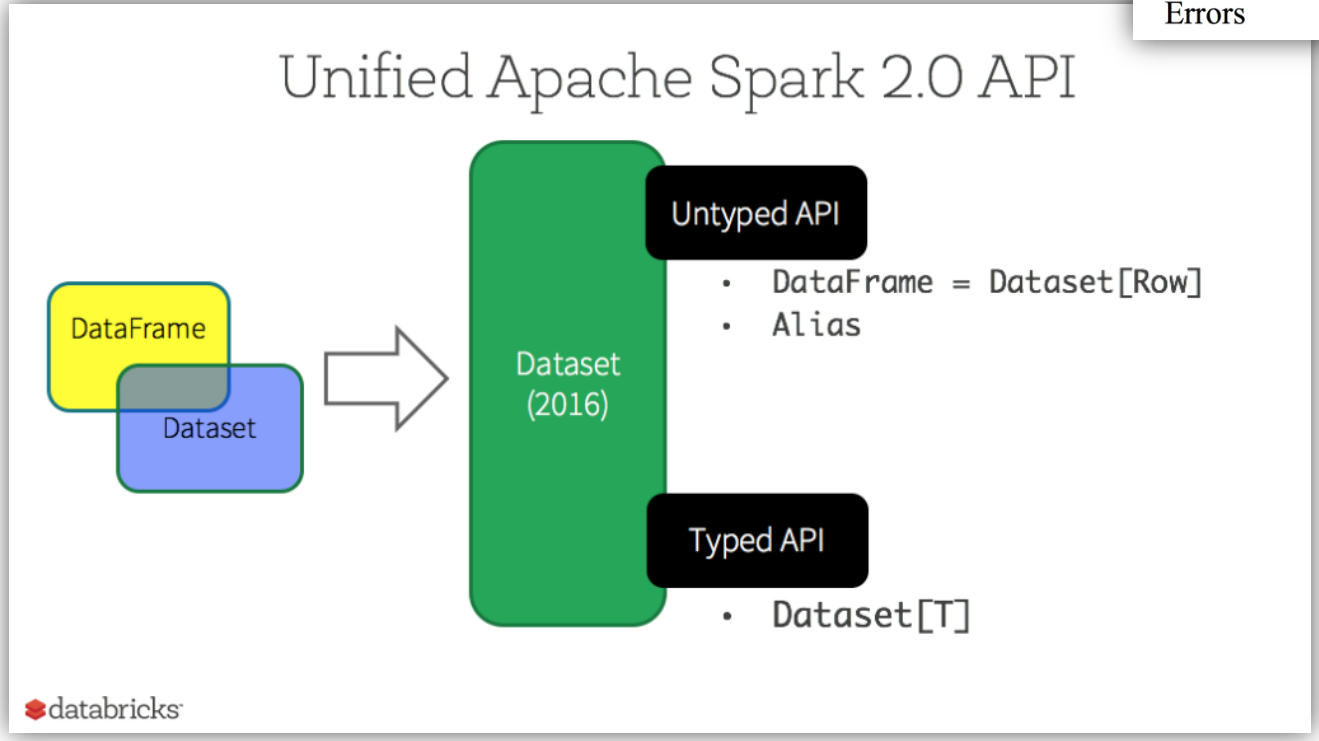
querying and transforming

Distributed Data

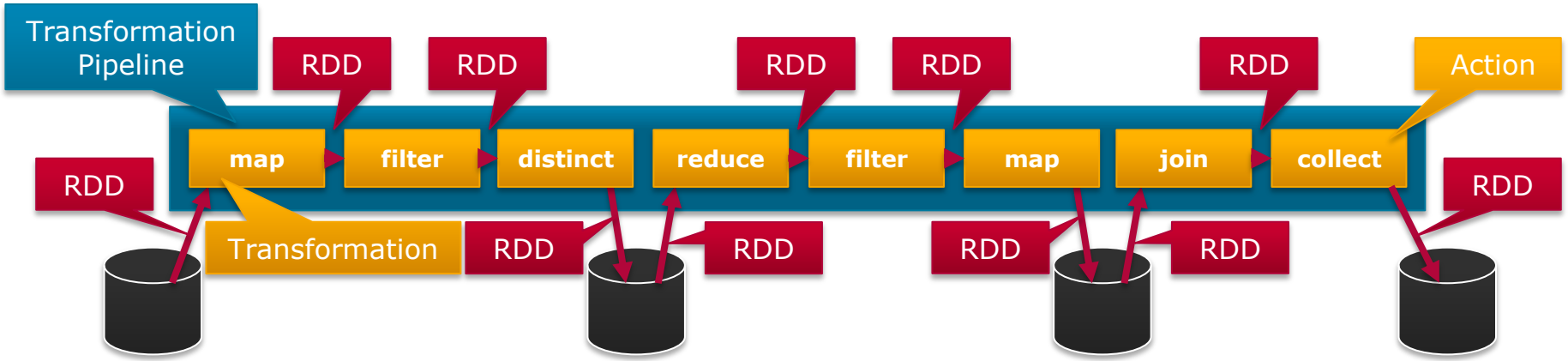


Distributed Data

	SQL	DataFrames	Datasets
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	Compile Time



Distributed Transformations and Actions



Transformation

- A **lazy operation** on an RDD that creates one or many new RDDs
  - RDD => RDD
  - RDD => Seq[RDD]
- Examples: `map()`, `filter()`, `join()`, `reduceByKey()`, `cogroup()`

Action

- An operation on an RDD that **triggers the execution** of a pipeline
- Specifies how to handle the result (view, collect, or write)
- Examples: `collect()`, `reduce()`, `count()`, `take()`

Distributed RDD Transformations

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numTasks*]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numTasks` argument to set a different number of tasks.
reduceByKey(*func*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numTasks*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean `ascending` argument.
join(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin`.
cogroup(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called `groupWith`.
cartesian(*otherDataset*)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
pipe(*command*, *[envVars]*)	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
coalesce(*numPartitions*)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
repartition(*numPartitions*)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
repartitionAndSortWithinPartitions(*partitioner*)	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling `repartition` and then sorting within each partition because it can push the sorting down into the shuffle machinery.

Distributed RDD Actions

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset.
takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(*n*, *[ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(*path*)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(*path*) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(*path*) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using `SparkContext.objectFile()`.
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. **Note**: modifying variables other than Accumulators outside of the `foreach()` may result in undefined behavior. See Understanding closures for more details.

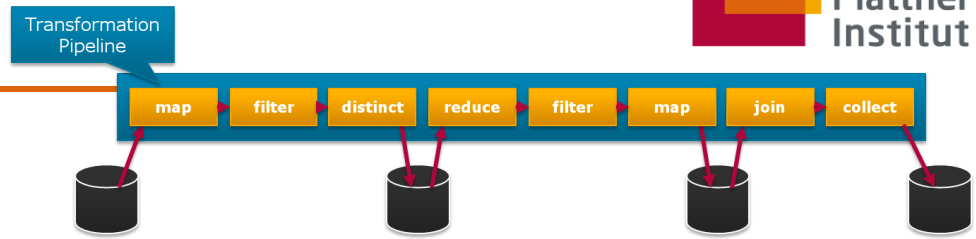
Types of actions

1. actions to view data (in the console)
2. actions to collect data (to native objects)
3. actions to write data (to output sources)

The Dataset and DataFrame APIs add many further transformations and actions:

https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset

Transformation Pipelines



Transformation Pipeline

- A chain of transformations
- Forms an RDD lineage with all the parent RDDs of the final RDD(s)
- Intermediate RDDs can become …
  - smaller (e.g. filter, count, distinct, sample)
  - same size (e.g. map)
  - larger (e.g. flatMap, union, cartesian)
    - … than their parents
- Actions finish a pipeline causing Spark to …
  1. compile the code (from Scala, Java, Python, SQL, R) into an execution plan
  2. optimize the execution plan (e.g. code/partition distribution and filter-push-down)
  3. execute the execution plan (scheduling tasks to executors)
- A Spark Session (i.e. Spark Program) can host multiple pipelines

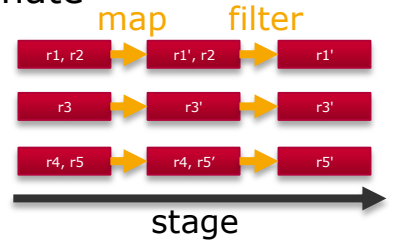
Spark can, in fact, use query optimization techniques similar to SQL optimizers!

Transformation Pipelining

Certain transformations can be pipelined which is an optimization that Spark uses to improve performance of computations

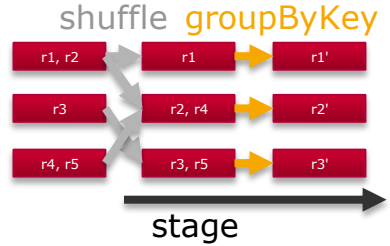
Narrow Transformations

- Transformations where all records in the same output RDD partition originate from the same input RDD partition (ensure self-sustained partitions)
- Examples: `map()` or `filter()`
- Spark groups narrow transformations into one stage (= pipelining)
  → data is kept in memory; transformations do not block later ones

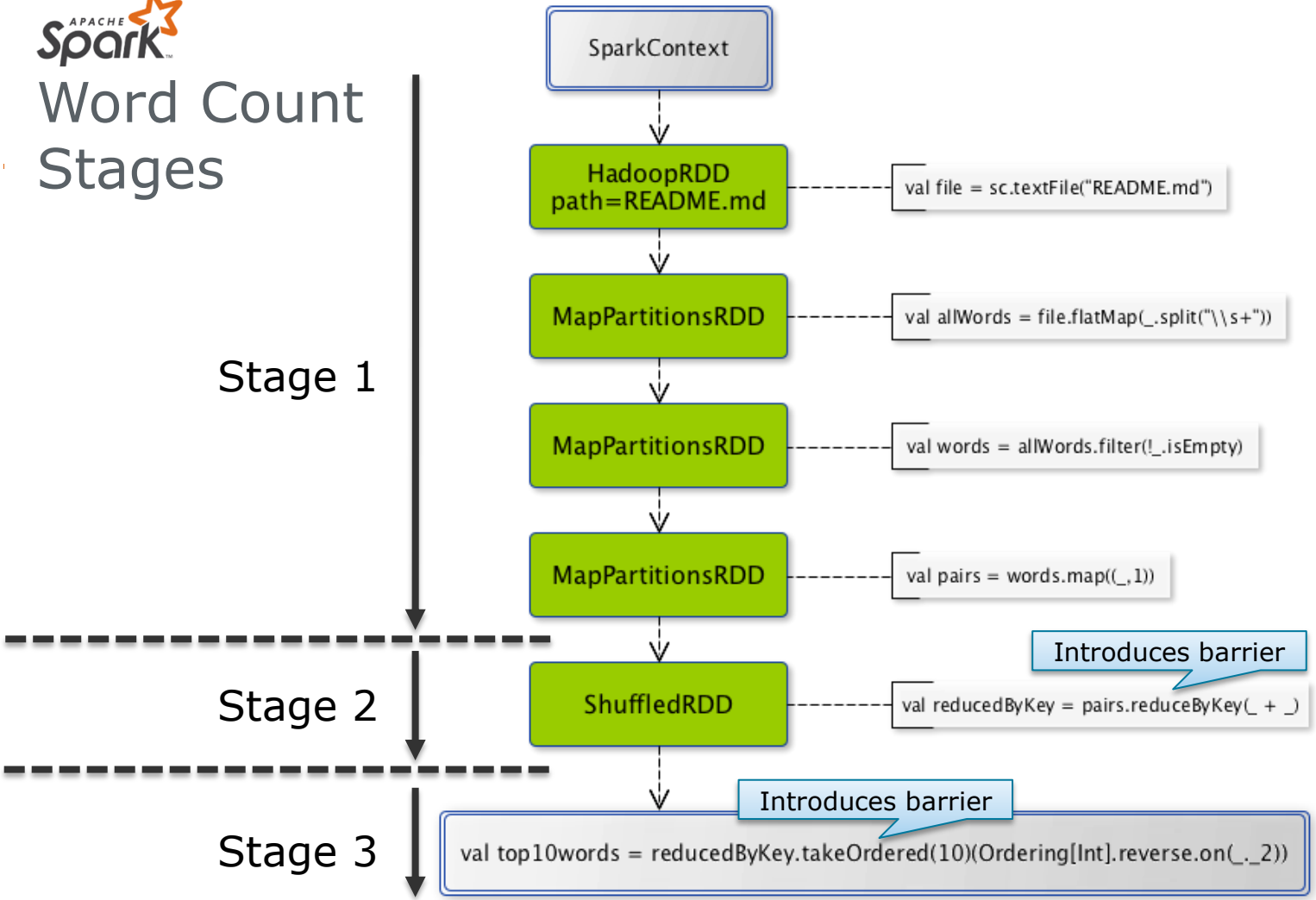


Wide Transformations

- Transformations where records might get shuffled across different RDD partitions; the records required to compute a certain output record may reside in different partitions of the parent RDD
- Examples: `groupByKey()` or `reduceByKey()`
- Spark executes a blocking RDD shuffle to transfer data across the cluster, which creates a new stage with a new set of partitions
  → data is written to disk; transformation blocks later ones



Word Count Stages



Degree of Parallelism

Number of executor threads

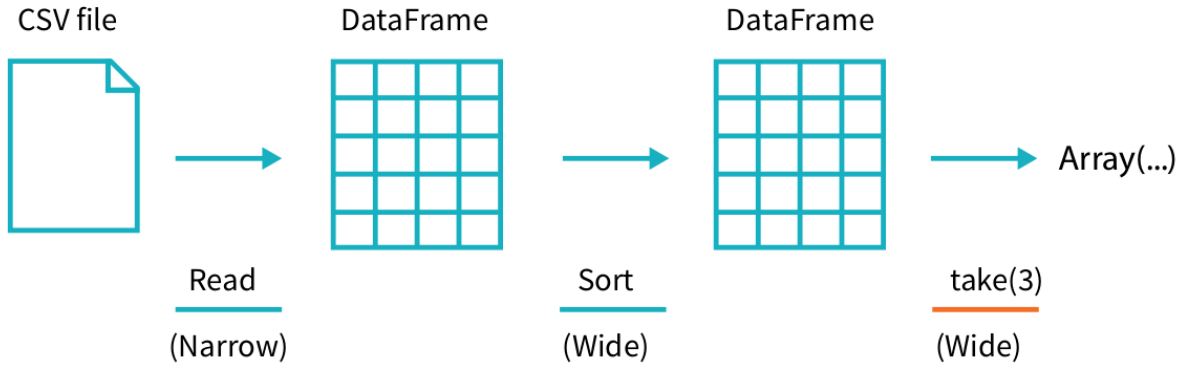
- Configurable, e.g., to the number of cores in the cluster
- Defined by the Cluster Manager (YARN/Mesos) or the SparkConf:
  - ```
    spark.conf.set("spark.executor.instances", "5") // 5 worker nodes
                .set("spark.executor.cores", "8")     // 8 cores on each node
    ```

Number of partitions

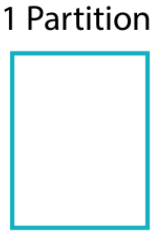
- Configurable, but usually greater than number of executor threads
  to improve load balancing and minimize scheduling overhead (see data replication lecture!)
- Defined by the Spark Session (default 200)
  - Change session-wide: `spark.conf.set("spark.sql.shuffle.partitions", "40")`
  - Change in pipeline: `val rdd2 = rdd.repartition(40)`

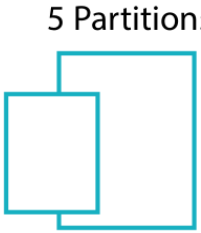
Degree of Parallelism = min(number of executor threads, number of partitions)
    ```

Degree of Parallelism

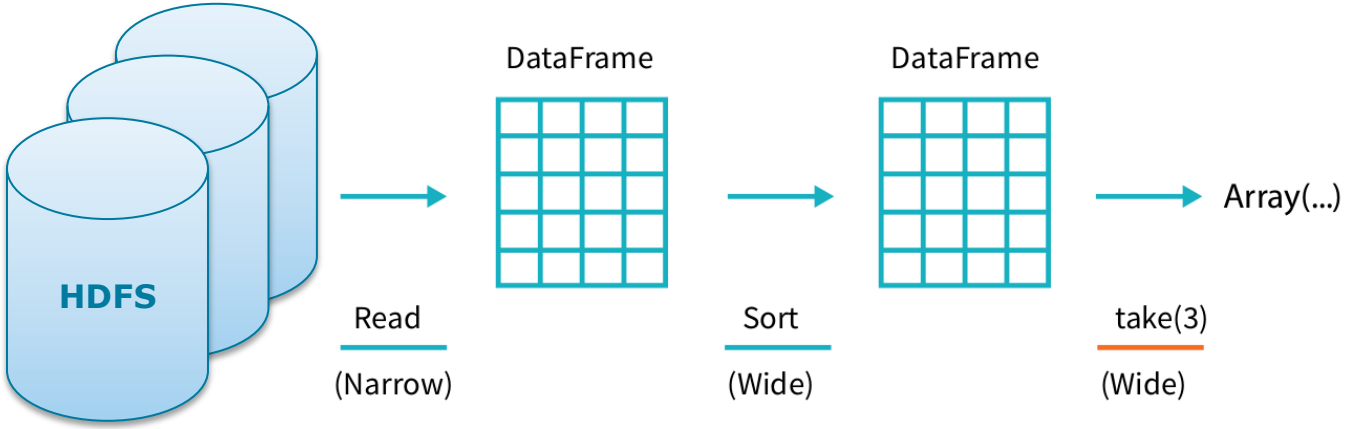


Reads are narrow transformations, i.e., if the input has only one partition, then the pipeline starts with only one partition!



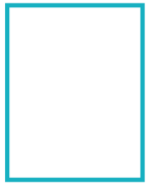


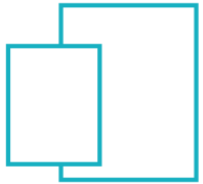
Degree of Parallelism



Spark reads every block in parallel and on the same node!

- #Blocks = #Partitions



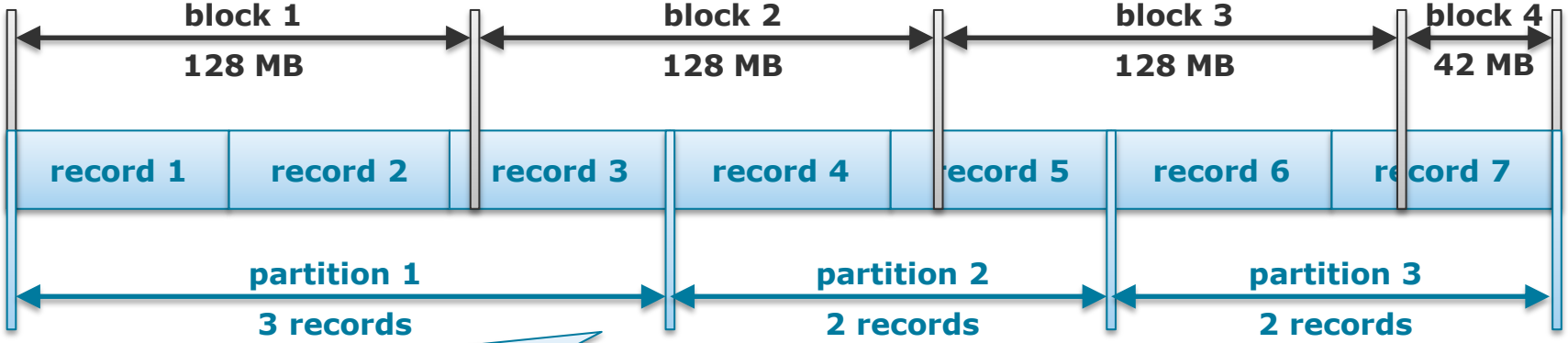


Degree of Parallelism

HDFS blocks vs. Spark partitions

- HDFS splits blocks on byte basis
- Spark splits partitions on line basis ('\n')

What happens to overlapping lines?



- Partition 1 worker fetches rest of record 3 from (remote) block 2
- Partition 2 worker ignores leading record 3 bytes and starts with record 4

https://www.ae.be/blog-en/ingesting-data-spark-using-custom-hadoop-fileinputformat/

Parquet Files

Apache Parquet

- A columnar storage format for files
- Stores data and the data's schema
- Compact due to … (recall lecture "Distributed DBMS")
  - dictionary encoding, bit packing, run-length encoding
- Used by many projects in the Hadoop ecosystem to store (intermediate) data

e.g. RDDs

Spark Parquet support

```
case class Flight(origin: String, destination: String, flights: BigInt)

val flightsDataFrame = spark.read.parquet("/mnt/data/parquet/flight.parquet/")
val flightsDataset = flightsDataFrame.as[Flight]

flightsDataset.printSchema()
flightsDataset.map(f => new Flight(f.origin.toLowerCase(), f.destination.toLowerCase(), f.flights))
flightsDataset.write.parquet("/mnt/data/parquet/flight_mapped.parquet/")
```

CSV and JSON formats work similarly!

Column Functions for DataFrames

```
val flightData = spark
  .read
  .option("inferSchema", "true")
  .option("header", "true")
  .csv("/mnt/data/flight-data-2015.csv")

import org.apache.spark.sql.functions.{col, lit, initcap, pow, month}

val result = flightData
  .select(
    initcap(col("ORIGIN")),
    month(col("DEPARTURE")).as("MONTH"),
    pow(col("FLIGHTS"), 2).as("CALCULATED"),
    lit("15.01.2018").as("UPDATED"))
  .limit(5)
  .collect()
```

Reference column values as Column object

Reference application values as Column object

Uppercase first letter

Take month from date

Calculate power of two

Add constant value

Spark SQL

```scala
val flightData = spark
  .read
  .option("inferSchema", "true")
  .option("header", "true")
  .csv("/mnt/data/flight-data-2015.csv")

import org.apache.spark.sql.functions.desc

val result = flightData
  .groupBy("DESTINATION")
  .sum("FLIGHTS")
  .withColumnRenamed("sum(FLIGHTS)", "total")
  .sort(desc("total"))
  .limit(5)
  .collect()
```

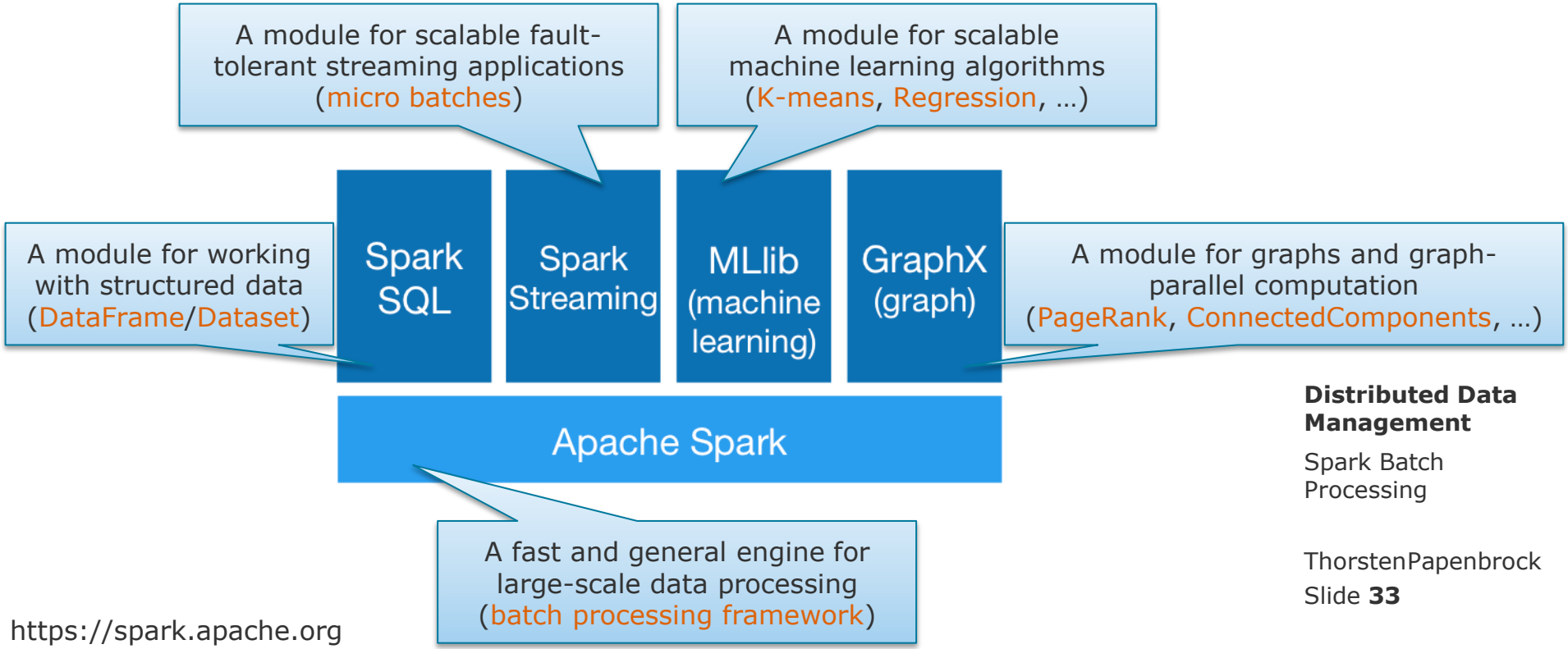
```scala
val flightData = spark
  .read
  .option("inferSchema", "true")
  .option("header", "true")
  .csv("/mnt/data/flight-data-2015.csv")

flightData.createOrReplaceTempView("flight")

val result = spark.sql("""
SELECT DESTINATION, sum(FLIGHTS) as total
FROM flight
GROUP BY DESTINATION
ORDER BY sum(FLIGHTS) DESC
LIMIT 5
""").collect()
```

Both compile to the same execution plan!
(check with `rdd.explain()`)

Modules

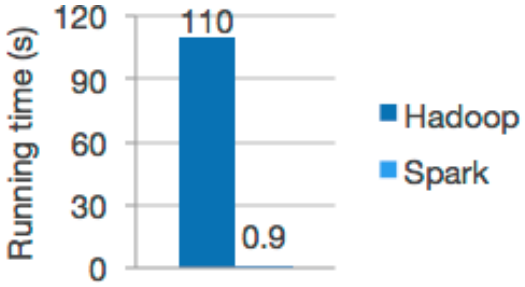


https://spark.apache.org

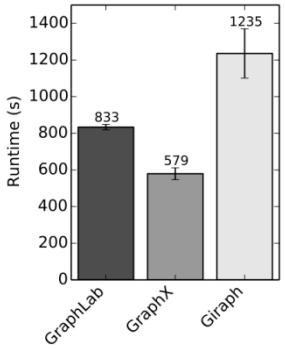
Modules

Performance

Mlib:
Logistic regression

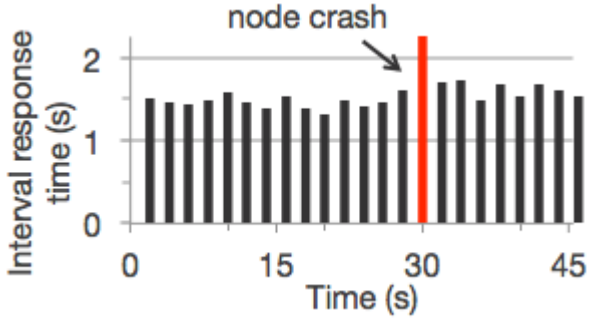


GraphX:
PageRank



20 iterations
3.7B edges

Spark Streaming:
Fault Scenario



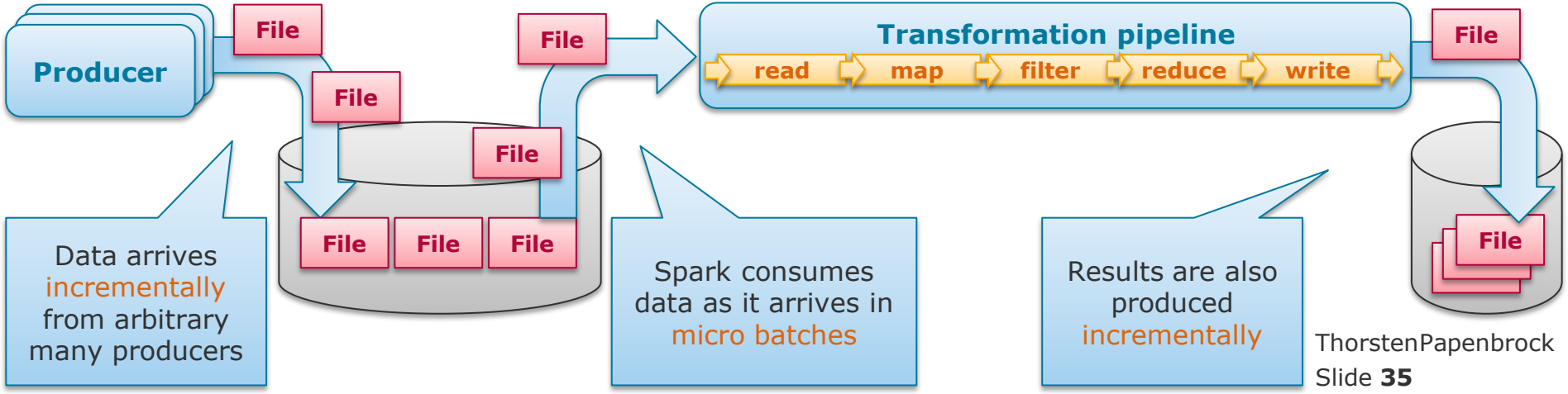
Many performance benchmarks, but consider:
- Technologies still evolve rapidly!
- Experimental setups have a huge influence on measurements!

https://spark.apache.org

Spark Streaming

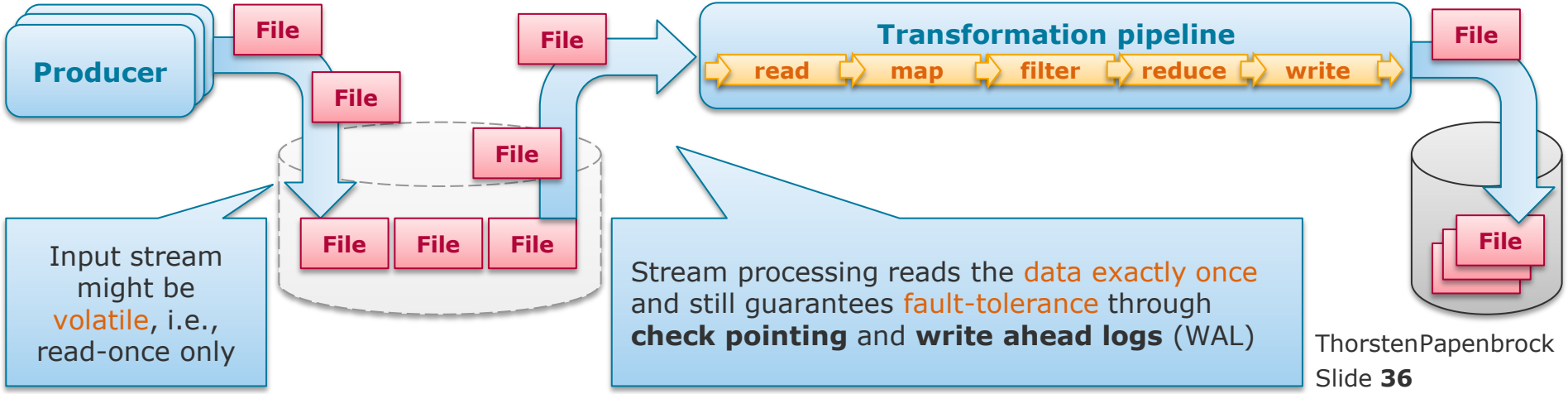
Batched Stream Processing

- Reasons:
  - Incremental processing: start processing data that is still being written to
  - Latency reduction: pipeline data to maximizing resource utilization



Batched Stream Processing

- Reasons:
  - Incremental processing: start processing data that is still being written to
  - Latency reduction: pipeline data to maximizing resource utilization



Spark Streaming

Batch Processing

```
val articles = spark
  .read
  .text("/mnt/data/articles/*.csv")

val words = articles.as[String].flatMap(_.split(" "))
val urls = words.filter(_.startsWith("http"))
val occurrences = urls.groupBy("value").count()

occurrences.show()
```

Batched Streaming

```
val articles = spark
  .readStream
  .text("/mnt/data/articles/*.csv")

val words = articles.as[String].flatMap(_.split(" "))
val urls = words.filter(_.startsWith("http"))
val occurrences = urls.groupBy("value").count()

val query = occurrences.writeStream
  .outputMode("complete")
  .format("console")
  .start()

query.awaitTermination()
```

Streaming input sources:

Files	text, csv, json, parquet
Kafka	Apache Kafka message broker
Socket	UTF8 text data from a socket
Rate	Generated data for testing

"complete"	write the entire result for every result update
"append"	append new results; old results should not change
"update"	output only changed results

Streaming output sinks:

Files	"parquet", "orc", "json", "csv", etc.
Kafka	"kafka" pointing to a Kafka topic
Foreach	.foreach(...)
Console	"console"
Memory	"memory" with .queryName("…")

Broadcast Variables

Enclosing scope variables

```
val names = List("Berget", "Bianka", "Cally")

val filtered1 = employees.filter(e => names.contains(e._1))
val filtered2 = employees.filter(e => !names.contains(e._1))
val filtered3 = employees.filter(e => names(1).equals(e._1))
```

"names" will be serialized and copied to every **executor**
(if a node has multiple executors, it receives the data multiple times, i.e., once per executor)

… and copied again!

… and copied again!

Broadcast variables

```
val bcNames = spark.sparkContext.broadcast(List("Berget", "Bianka", "Cally"))

val filtered1 = employees.filter(e => bcNames.value.contains(e._1))
val filtered2 = employees.filter(e => !bcNames.value.contains(e._1))
val filtered3 = employees.filter(e => bcNames.value(1).equals(e._1))

bcNames.destroy()
```

"names" will be serialized and copied to every executor **node**
(if a node has multiple executors, it still receives the data only once)

… data already present!

… data already present!

Regular Expression Support

Replace by regex

One of many regex functions!

```
import org.apache.spark.sql.functions.{col, regexp_replace}
val regexString = "BERLIN|FRANKFURT|MÜNCHEN|HAMBURG|OSNABRÜCK|POTSDAM"
dataFrame
  .select(regexp_replace(col("ORIGIN"), regexString, "GERMANY").as("O_COUNTRY"))
  .show(10)
```

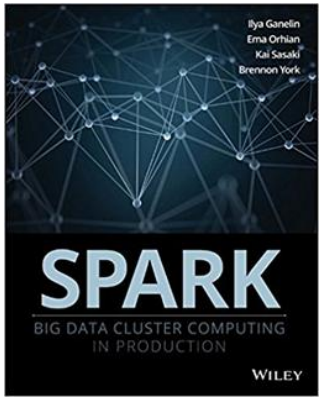
Extract by regex

```
import org.apache.spark.sql.functions.{col, regexp_extract}
val regexString = "(BLACK|WHITE|RED|GREEN|BLUE|YELLOW)"
dataFrame
  .select(regexp_extract (col("DESCRIPTION"), regexString, 1).as("COLOR"))
  .show(10)
```

Further Reading

The Data Engineers Guide to Apache Spark

- PDF on lecture homepage



Spark Documentation

- https://spark.apache.org/docs/latest/
- https://spark.apache.org/docs/latest/rdd-programming-guide.html

Spark API Documentation

- https://spark.apache.org/docs/latest/api/scala/index.html
- https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.<X>.html
  - <X> ϵ {Dataset, DataFrameReader, DataFrameStatFunctions, DataFrameNaFunctions, Encoder, SQLImplicits, SparkSession, Column, functions}

Spark – Big Data Cluster Computing in Production

- https://www.amazon.de/Spark-Data-Cluster-Computing-Production/dp/1119254019/

Homework

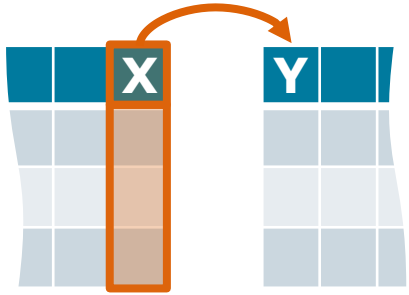
Inclusion Dependency Discovery

Definition: Given two relational instances r_i and r_j for the schemata R_i, and R_j, respectively. The inclusion dependency $R_i[X] \subseteq R_j[Y]$ (short $X \subseteq Y$) with $X \subseteq R_i$, $Y \subseteq R_j$ and $|X| = |Y|$ is valid, iff $\forall\, t_i[X] \in r_i$, $\exists t_j[Y] \in r_j : t_i[X] = t_j[Y]$.

"All values in X are also contained in Y"

We consider only unary INDs for this task, i.e., $|X| = |Y| = 1$

foreign-key candidates



Homework

Inclusion Dependency Discovery

Name	Type	Equatorial diameter	Mass	Orbital radius	Orbital period	Rotation period	Confirmed moons	Rings	Atmosphere
Mercury	Terrestrial	0.382	0.06	0.47	0.24	58.64	0	no	minimal
Venus	Terrestrial	0.949	0.82	0.72	0.62	−243.02	0	no	CO_2, N_2
Earth	Terrestrial	1.000	1.00	1.00	1.00	1.00	1	no	N_2, O_2, Ar
Mars	Terrestrial	0.532	0.11	1.52	1.88	1.03	2	no	CO_2, N_2, Ar
Jupiter	Giant	11.209	317.8	5.20	11.86	0.41	67	yes	H_2, He
Saturn	Giant	9.449	95.2	9.54	29.46	0.43	62	yes	H_2, He
Uranus	Giant	4.007	14.6	19.22	84.01	−0.72	27	yes	H_2, He
Neptune	Giant	3.883	17.2	30.06	164.8	0.67	14	yes	H_2, He

Complexity: $O(n^2-n)$ for n attributes

Example:
10 attr ~ 90 checks
1,000 attr ~ 999,000 checks

- Name ⊆ Type ?
- Name ⊆ Equatorial_diameter ?
- Name ⊆ Mass ?
- Name ⊆ Orbital_radius ?
- Name ⊆ Orbital_period ?
- Name ⊆ Rotation_period ?
- Name ⊆ Confirmed_moons ?
- Name ⊆ Rings ?
- Name ⊆ Atmosphere ?

- Type ⊆ Name ?
- Type ⊆ Equatorial_diameter ?
- Type ⊆ Mass ?
- Type ⊆ Orbital_radius ?
- Type ⊆ Orbital_period ?
- Type ⊆ Rotation_period ?
- Type ⊆ Confirmed_moons ?
- Type ⊆ Rings ?
- Type ⊆ Atmosphere ?

- Mass ⊆ Name ?
- Mass ⊆ Type ?
- Mass ⊆ Equatorial_diameter ?
- …

http://webdatacommons.org/webtables/index.html

Homework

Inclusion Dependency Discovery

Name	Type	Equatorial diameter	Mass	Orbital radius	Orbital period	Rotation period	Confirmed moons	Rings	Atmosphere
Mercury	Terrestrial	0.382	0.06	0.47	0.24	58.64	0	no	minimal
Venus	Terrestrial	0.949	0.82	0.72	0.62	−243.02	0	no	CO_2, N_2
Earth	Terrestrial	1.000	1.00	1.00	1.00	1.00	1	no	N_2, O_2, Ar
Mars	Terrestrial	0.532	0.11	1.52	1.88	1.03	2	no	CO_2, N_2, Ar
Jupiter	Giant	11.209	317.8	5.20	11.86	0.41	67	yes	H_2, He
Saturn	Giant	9.449	95.2	9.54	29.46	0.43	62	yes	H_2, He
Uranus	Giant	4.007	14.6	19.22	84.01	−0.72	27	yes	H_2, He
Neptune	Giant	3.883	17.2	30.06	164.8	0.67	14	yes	H_2, He

Planet	Rotation Period	Revolution Period
Mercury	58.6 days	87.97 days
Venus	243 days	224.7 days
Earth	0.99 days	365.26 days
Mars	1.03 days	1.88 years
Jupiter	0.41 days	11.86 years
Saturn	0.45 days	29.46 years
Uranus	0.72 days	84.01 years
Neptune	0.67 days	164.79 years
Pluto	6.39 days	248.59 years

Symbol	Unicode	Glyph
Sun	U+2609	☉
Moon	U+263D	☽
Moon	U+263E	☾
Mercury	U+263F	☿
Venus	U+2640	♀
Earth	U+1F728	🜨
Mars	U+2642	♂
Jupiter	U+2643	♃
Saturn	U+2644	♄
Uranus	U+2645	♅
Uranus	U+26E2	⛢
Neptune	U+2646	♆
Eris	≈ U+2641	♁
Eris	≈ U+29EC	⧬
Pluto	U+2647	♇
Pluto	not present	--
Aries	U+2648	♈
Taurus	U+2649	♉
Gemini	U+264A	♊
Cancer	U+264B	♋
Leo	U+264C	♌
Virgo	U+264D	♍
Libra	U+264E	♎
Scorpio	U+264F	♏
Sagittarius	U+2650	♐
Capricorn	U+2651	♑
Capricorn	U+2651	♑
Aquarius	U+2652	♒
Pisces	U+2653	♓
Conjunction	U+260C	☌
...	...	...

Planet	Synodic period	Synodic period (mean)	Days in retrograde
Mercury	116	3.8	~21
Venus	584	19.2	41
Mars	780	25.6	72
Jupiter	399	13.1	121
Saturn	378	12.4	138
Uranus	370	12.15	151
Neptune	367	12.07	158

Planet	Mean distance	Relative mean distance
Mercury	57.91	1
Venus	108.21	1.86859
Earth	149.6	1.3825
Mars	227.92	1.52353
Ceres	413.79	1.81552
Jupiter	778.57	1.88154
Saturn	1,433.53	1.84123
Uranus	2,872.46	2.00377
Neptune	4,495.06	1.56488
Pluto	5,869.66	1.3058

Sign	House	Domicile	Detriment	Exaltation	Fall	Planetary Joy
Aries	1st House	Mars	Venus	Sun	Saturn	Mercury
Taurus	2nd House	Venus	Pluto	Moon	Uranus	Jupiter
Gemini	3rd House	Mercury	Jupiter	N/A	N/A	Saturn
Cancer	4th House	Moon	Saturn	Jupiter	Mars	Venus
Leo	5th House	Sun	Uranus	Neptune	Mercury	Mars
Virgo	6th House	Mercury	Neptune	Pluto, Mercury	Venus	Saturn
Libra	7th House	Venus	Mars	Saturn	Sun	Moon
Scorpio	8th House	Pluto	Venus	Uranus	Moon	Saturn
Sagittarius	9th House	Jupiter	Mercury	N/A	N/A	Sun
Capricorn	10th House	Saturn	Moon	Mars	Jupiter	Mercury
Aquarius	11th House	Uranus	Sun	Mercury	Neptune	Venus
Pisces	12th House	Neptune	Mercury	Venus	Pluto, Mercury	Moon

Planet	Calculated (in AU)	Observed (in AU)	Perfect octaves	Actual distance
Mercury	0.4	0.387	0	0
Venus	0.7	0.723	1	1.1
Earth	1	1	2	2
Mars	1.6	1.524	4	3.7
Asteroid belt	2.8	2.767	8	7.8
Jupiter	5.2	5.203	16	15.7
Saturn	10	9.539	32	29.9
Uranus	19.6	19.191	64	61.4
Neptune	38.8	30.061	96	-96.8
Pluto	77.2	39.529	128	127.7

http://webdatacommons.org/webtables/index.html

Homework

Inclusion Dependency Discovery - Rules

Assignment

- Task
  - Write a Spark program that discovers all unary INDs in a given dataset (within and between all tables!).
  - The input dataset may consist of multiple tables.
  - The Spark program should run standalone in local mode.
- Dataset
  - Use the TPCH dataset provided on our website: https://hpi.de/fileadmin/user_upload/fachgebiete/naumann/lehre/WS2017/DDA/TPCH.zip
  - TPCH is a generated dataset often used to benchmark query performance.
- Parameter
  - "java -jar YourAlgorithmName.jar --path TPCH --cores 4"
  - Default path should be "./TPCH".

www.tpc.org/tpch

Inclusion Dependency Discovery - Rules

Assignment

- Expected output
  - Write the discovered INDs lexicographically sorted to the console.
  - Use the following style for your output:
    `<dependent> < <referenced1>, <referenced2>, …`
    - So the correct output should look as follows:

```
C_CUSTKEY < P_PARTKEY
C_NATIONKEY < S_NATIONKEY, N_NATIONKEY
L_COMMIT < L_SHIP, L_RECEIPT
L_LINENUMBER < C_NATIONKEY, S_NATIONKEY, O_ORDERKEY, L_SUPPKEY, N_NATIONKEY, S_SUPPKEY, P_PARTKEY, P_SIZE, C_CUSTKEY, L_PARTKEY
L_LINESTATUS < O_ORDERSTATUS
L_ORDERKEY < O_ORDERKEY
L_PARTKEY < P_PARTKEY
L_SUPPKEY < P_PARTKEY, S_SUPPKEY, C_CUSTKEY
L_TAX < L_DISCOUNT
N_NATIONKEY < C_NATIONKEY, S_NATIONKEY
N_REGIONKEY < C_NATIONKEY, S_NATIONKEY, N_NATIONKEY, R_REGIONKEY
O_CUSTKEY < P_PARTKEY, C_CUSTKEY
O_SHIPPRIORITY < C_NATIONKEY, S_NATIONKEY, N_REGIONKEY, N_NATIONKEY, R_REGIONKEY
P_SIZE < L_SUPPKEY, S_SUPPKEY, P_PARTKEY, C_CUSTKEY, L_PARTKEY
R_REGIONKEY < C_NATIONKEY, S_NATIONKEY, N_REGIONKEY, N_NATIONKEY
S_NATIONKEY < C_NATIONKEY, N_NATIONKEY
S_SUPPKEY < L_SUPPKEY, P_PARTKEY, C_CUSTKEY
```

Homework

Inclusion Dependency Discovery - Rules

Assignment

- Submission deadline
  - 12.07.2021 09:00:00
- Submission channel
  - Your git repositories
- Submission artifacts
  - Source code
  - A slide with your transformation pipeline(s)
- Teams
  - Please solve the homework in teams of two students.

Homework

Inclusion Dependency Discovery - Hint

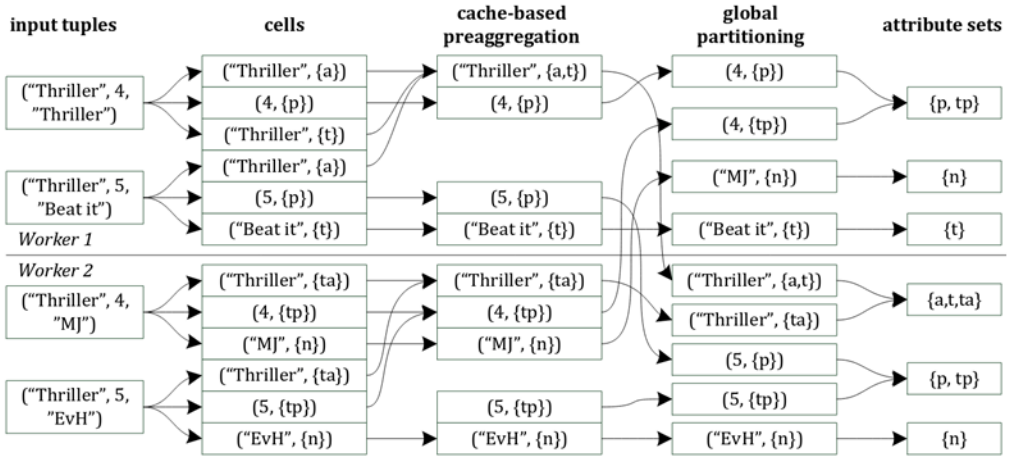


Figure 1: Dataflow for a distributed full outer join on all columns of a dataset. The attribute names and some values are abbreviated for the purpose of lucidity.

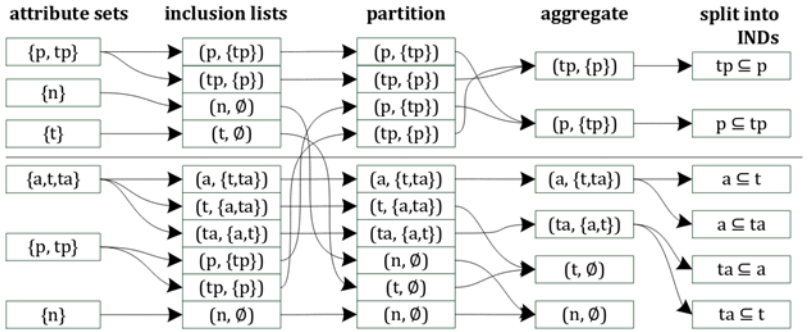


Figure 2: Dataflow for deriving INDs from the outer join product.

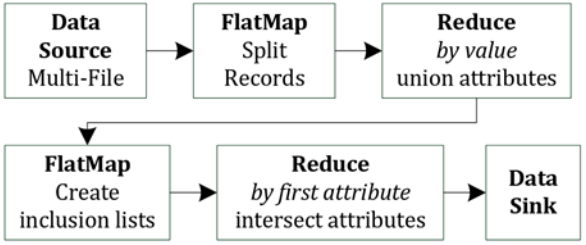


Figure 3: Stratosphere plan for unary IND detection.

Scaling Out the Discovery of Inclusion Dependencies
S. Kruse, T. Papenbrock, F. Naumann,
Proceedings of the conference on Database Systems for Business, Technology, and Web (BTW). pp. 445-454 (2015).

Inclusion Dependency Discovery - Hints

Further hints

- If you choose to implement the pipeline proposed in the paper, useful functions might be …
  - `collect_set()`
  - `size()`
  - `explode()`
- To get the column labels, have a look at `dataframe.columns`.
- Using 4 cores, you should be able to find all INDs on the provided TPCH dataset within ~10 min.
- Start developing your pipeline only with the region and nation tables;
  when everything works, add the other tables to the input.
- Take one step of your pipeline after another and check intermediate results with `dataset.show()`.
- We evaluate all submitted algorithms on a server with 32 cores.

Distributed Data Management
Spark Batch Processing

Thorsten Papenbrock
G-3.1.09, Campus III
Hasso Plattner Institut