RDD Fundamentals

databricks™

Spark RDD

- Resilient Distributed Dataset
  - 弹性分布式数据集（RDD）
  - Spark 的核心数据结构
- 跨服务器分布，映射到磁盘或内存的数据集合
- 提供了受限形式的分布式共享内存

RDD

- 分布在计算机集群上的一个只读的数据集
- 对它的操作是 Lazy 的
- 利用集群中的持久性数据块得以缓存、复制和分发
- 可以使用 join 和各种 Map and Reduce 转换操作来创建新的 RDD
- 基于 RDD 实现了 Spark 中基于内存的 MapReduce 操作
- 在某些数据丢失的情况下，通过跟踪每个 RDD 的 lineage 或重建 RDD 来实现容错
- RDD 有它的 lineage，记录了它如何从其他稳定存储的数据集派生计算过来的。这是一个强大的属性。利用这个 lineage，程序即使失败了，也可以重建 RDD，但需要耗费 CPU。

Spark 并行运算

- Spark 用 Scala 实现
  - Scala 是一种解释性的，静态类型的对象功能语言
- Scala 并行运算符库，类似于 Hadoop 中使用的 Map 和 Reduce 操作
  - 在 RDD 上执行转换
  - 该库有不错的 Python 绑定，可以用 Python 编程

性能（2014 年）

- 在 AWS 帮助下，Databricks 团队参加了 Daytona Gray 测试
  - 对 100 TB 数据（1 万亿条记录）进行分类
  - 前世界纪录是 Yahoo!使用 2100 个节点的 Hadoop MapReduce 集群创造的 72 分钟
  - 他们在 206 个 EC2 节点上使用 Spark，23 分钟
  - 所有排序都在磁盘（HDFS）上进行，没有使用 Spark 的内存缓存
- 不到 4 小时内对 190 台计算机上的 1 PB 数据（10 万亿条记录）排序
  - 以前报告的基于 Hadoop MapReduce 的结果是在 3,800 台计算机上为 16 小时。

Resilient Distributed Datasets (RDDs)

- Write programs in terms of operations on distributed datasets
- Partitioned collections of objects spread across a cluster, stored in memory or on disk
- RDDs built and manipulated through a diverse set of parallel transformations (map, filter, join) and actions (count, collect, save)
- RDDs automatically rebuilt on machine failure

基于 RDD Partition 的并行

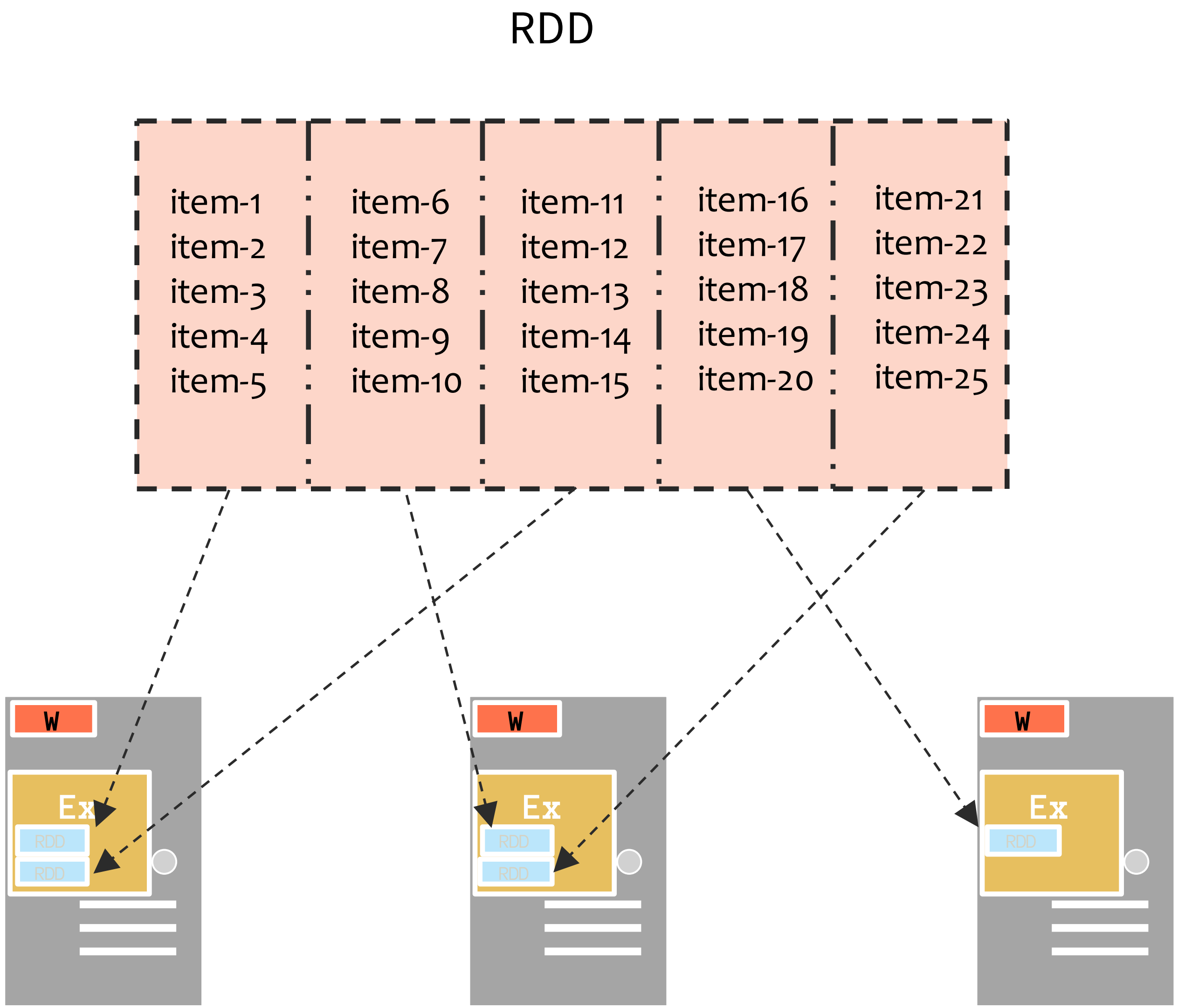
- 并行性通过在每个分区上并行计算获得
  - 每个 Spark 操作都在 RDD 所在的 Worker 运行
  - 每个 Worker 使用多个线程
  - 对于 Reduce 操作，首先在各分区上完成，然后根据需要跨分区进行
- 实际上，Python 程序会编译一个图，然后由 Spark 引擎执行该图
  - Spark Python 库利用 Python lambda 运算符创建匿名函数的能力
  - 首先生成代码，然后由 Spark 工作调度器将它们传送给 Worker，在每个 RDD 分区上执行

Partition 优化

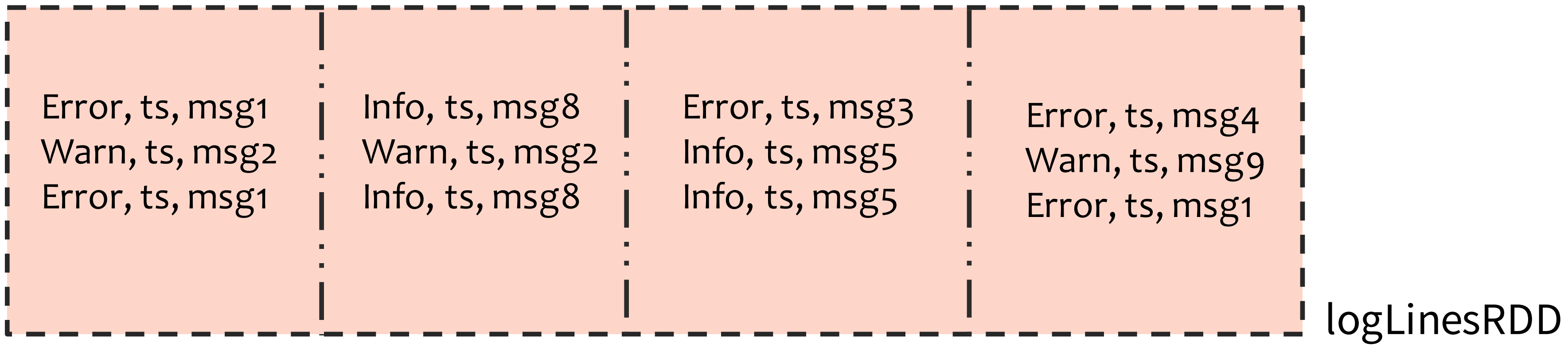
- RDD 中的元素可以基于每个记录的键在计算机之间进行分区
- 分区顺序由 partitioner（分区程序）类确定
- groupByKey，reduceByKey 和 sort 将获得一个分区后的 RDD
- 如果两个数据集将要通过 join 连接到一起，那么可以将它们通过相同的 partition 类进行分区，这对后面的 join 很有帮助
  - 这样的话，联接操作不需要通信
  - 因为要 join 的每行的两个数据都在一个机器上
- 可以编写一个自定义分区程序类来进行分区

```
links = spark.textFile(…).map(…) .partitionBy(myPartFunc)
```

more partitions = more parallelism



RDD w/ 4 partitions



A base RDD can be created 2 ways:

- Parallelize a collection
- Read data from an external source (S3, C*, HDFS, etc)

databricks™

Parallelize

- Take an existing in-memory collection and pass it to SparkContext's parallelize method
- Not generally used outside of prototyping and testing since it requires entire dataset in memory on one machine

```scala
// Parallelize in Scala
val wordsRDD = sc.parallelize(List("fish", "cats", "dogs"))
```

```python
# Parallelize in Python
wordsRDD = sc.parallelize(["fish", "cats", "dogs"])
```

```java
// Parallelize in Java
JavaRDD<String> wordsRDD = sc.parallelize(Arrays.asList("fish", "cats", "dogs"));
```

Read from Text File

There are other methods to read data from HDFS, C*, S3, HBase, etc.

```scala
// Read a local txt file in Scala
val linesRDD = sc.textFile("/path/to/README.md")
```

```python
# Read a local txt file in Python
linesRDD = sc.textFile("/path/to/README.md")
```

```java
// Read a local txt file in Java
JavaRDD<String> lines = sc.textFile("/path/to/README.md");
```

操作

- 两种类型的操作
- Transformations 变换
  - 将 RDD 映射到新 RDD
- Action 动作
  - 返回值给主程序
  - 通常是 read-eval-print 循环，例如 Jupyter

Transformations 和 Action

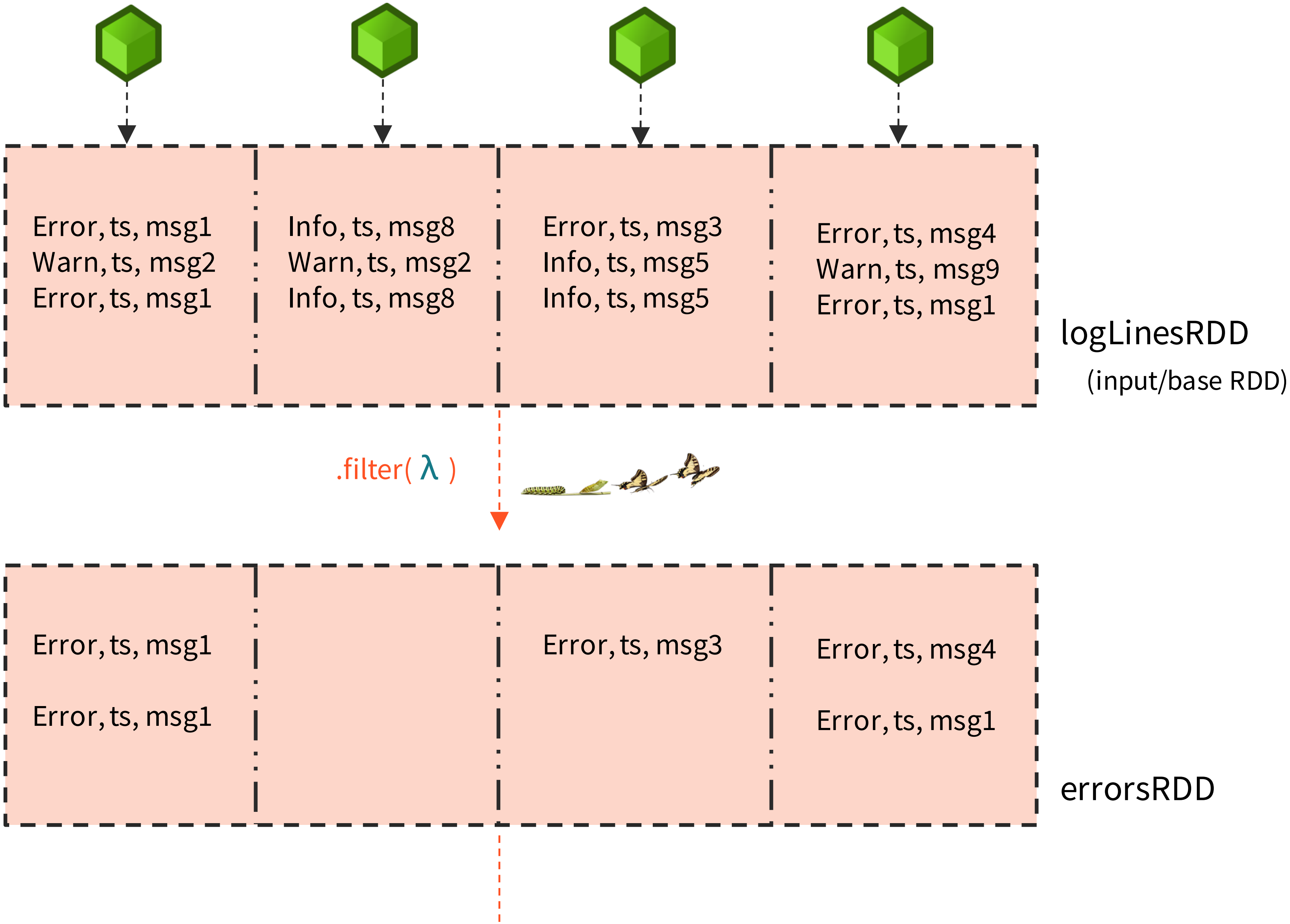
- Transformations（变换）是定义新 RDD 的一种 Lazy 操作
    - 所谓 Lazy，就是它只是记录要执行的操作，并不真正执行
- Action（动作）会启动真正的计算，以将值返回到程序或将数据写入外部存储
- Action 包括
    - count 计数（返回数据集中元素的数量）
    - collect 收集（返回元素本身）
    - save 保存（将数据集输出到存储系统）
- 程序员首先通过对稳定存储中的数据进行 Transformations 来定义一个或多个 RDD。然后执行 Action，将值返回给应用程序或将数据导出到存储系统

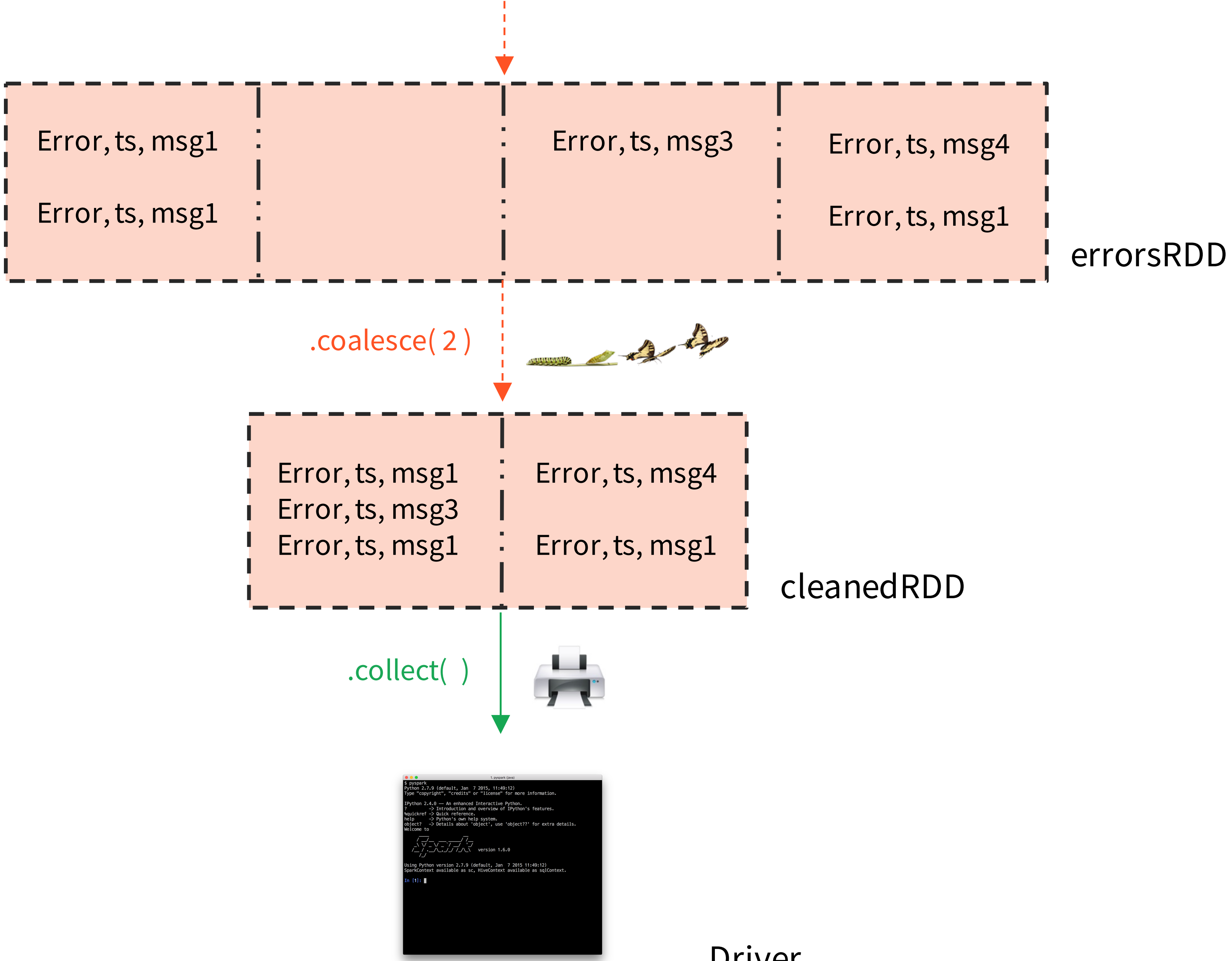
RDD 编程： Persistence 控制

- 在多次迭代的工作中，可能有必要将一些版本的 RDD 存起来，以减少故障恢复时间
- 用户可以调用 persist，带上一个 reliable（可靠）标志，来执行此操作

Operations on Distributed Data

- Two types of operations: *transformations* and *actions*
- Transformations are lazy (*not computed immediately*)
- Transformations are executed when an action is run
- Persist (cache) distributed data in memory or disk


Error, ts, msg1
Warn, ts, msg2
Error, ts, msg1
Info, ts, msg8
Warn, ts, msg2
Info, ts, msg8
Error, ts, msg3
Info, ts, msg5
Info, ts, msg5
Error, ts, msg4
Warn, ts, msg9
Error, ts, msg1
logLinesRDD
(input/base RDD)
.filter(λ)
Error, ts, msg1
Error, ts, msg1
Error, ts, msg3
Error, ts, msg4
Error, ts, msg1
errorsRDD

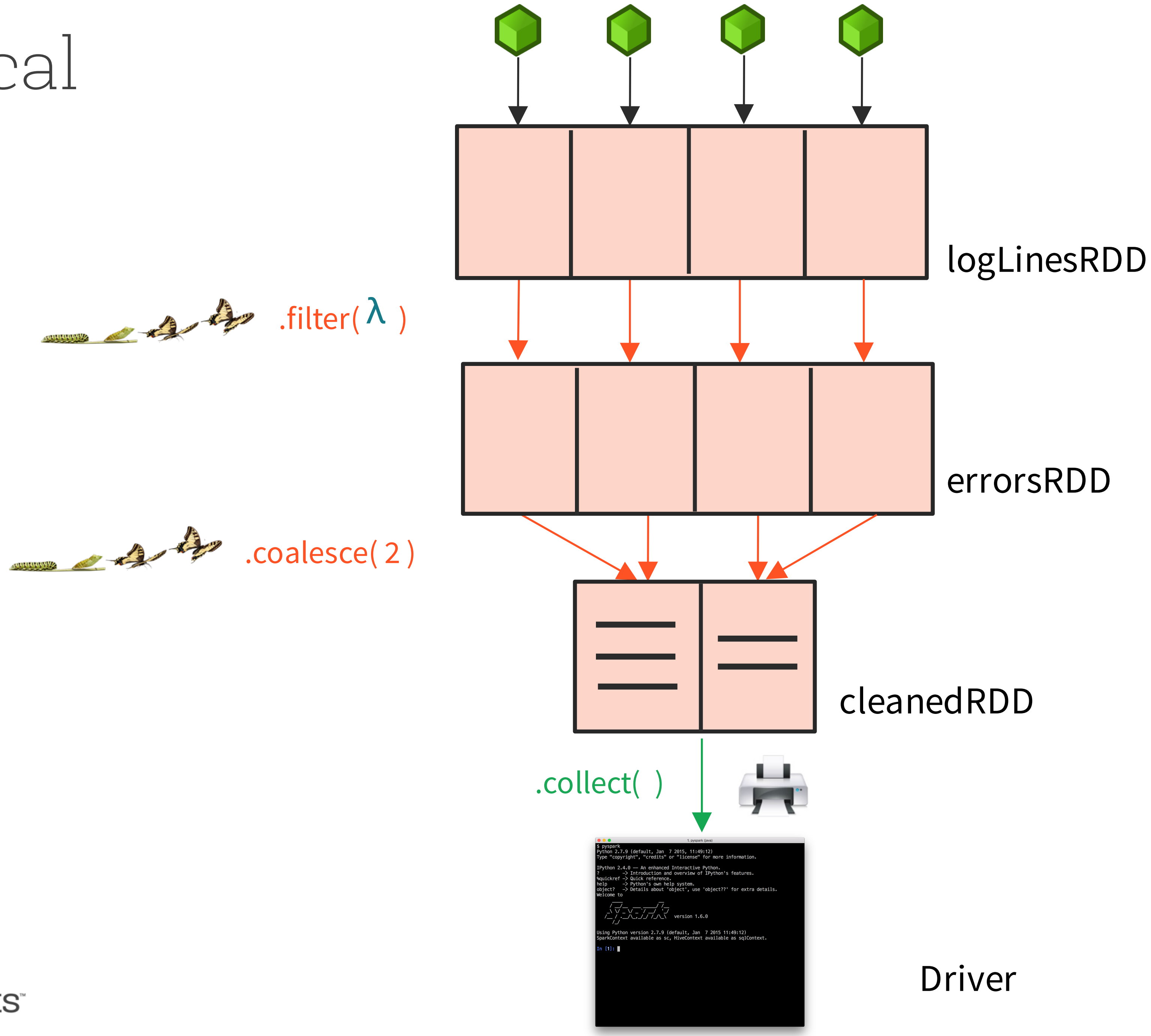

Error, ts, msg1
Error, ts, msg1
Error, ts, msg3
Error, ts, msg4
Error, ts, msg1
errorsRDD
.coalesce(2)
Error, ts, msg1
Error, ts, msg3
Error, ts, msg1
Error, ts, msg4
Error, ts, msg1
cleanedRDD
.collect( )
Driver

databricks

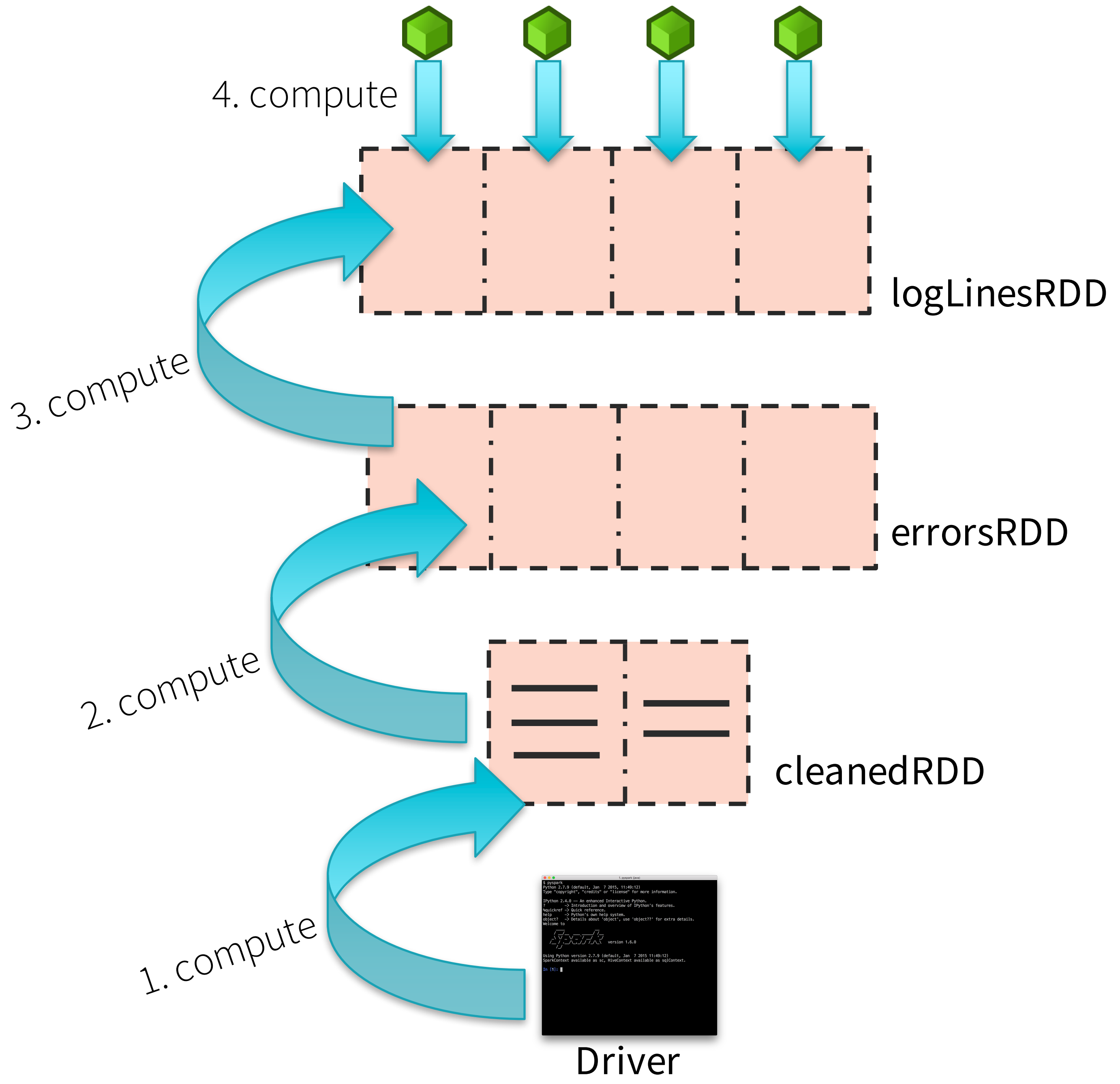

Execute DAG!
.collect( )
1. pyspark (java)
$ pyspark
Python 2.7.9 (default, Jan 7 2015, 11:49:12)
Type "copyright", "credits" or "license" for more information.
IPython 2.4.0 — An enhanced Interactive Python.
? -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.
Welcome to
version 1.6.0
Using Python version 2.7.9 (default, Jan 7 2015 11:49:12)
SparkContext available as sc, HiveContext available as sqlContext.
In [1]:
Driver

databricks™

Logical

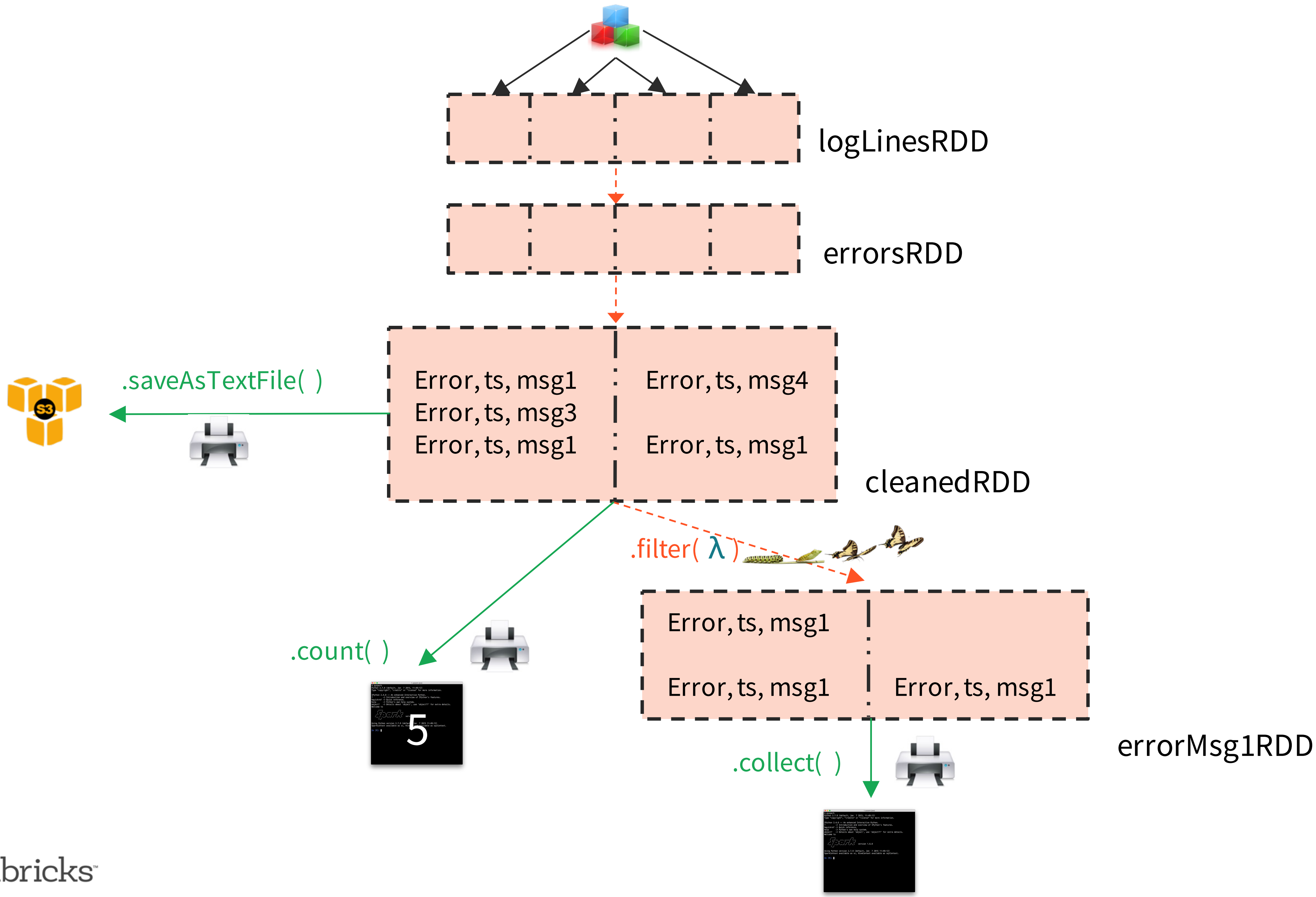

.filter(λ)
logLinesRDD
errorsRDD
.coalesce(2)
cleanedRDD
.collect( )
Driver

Physical

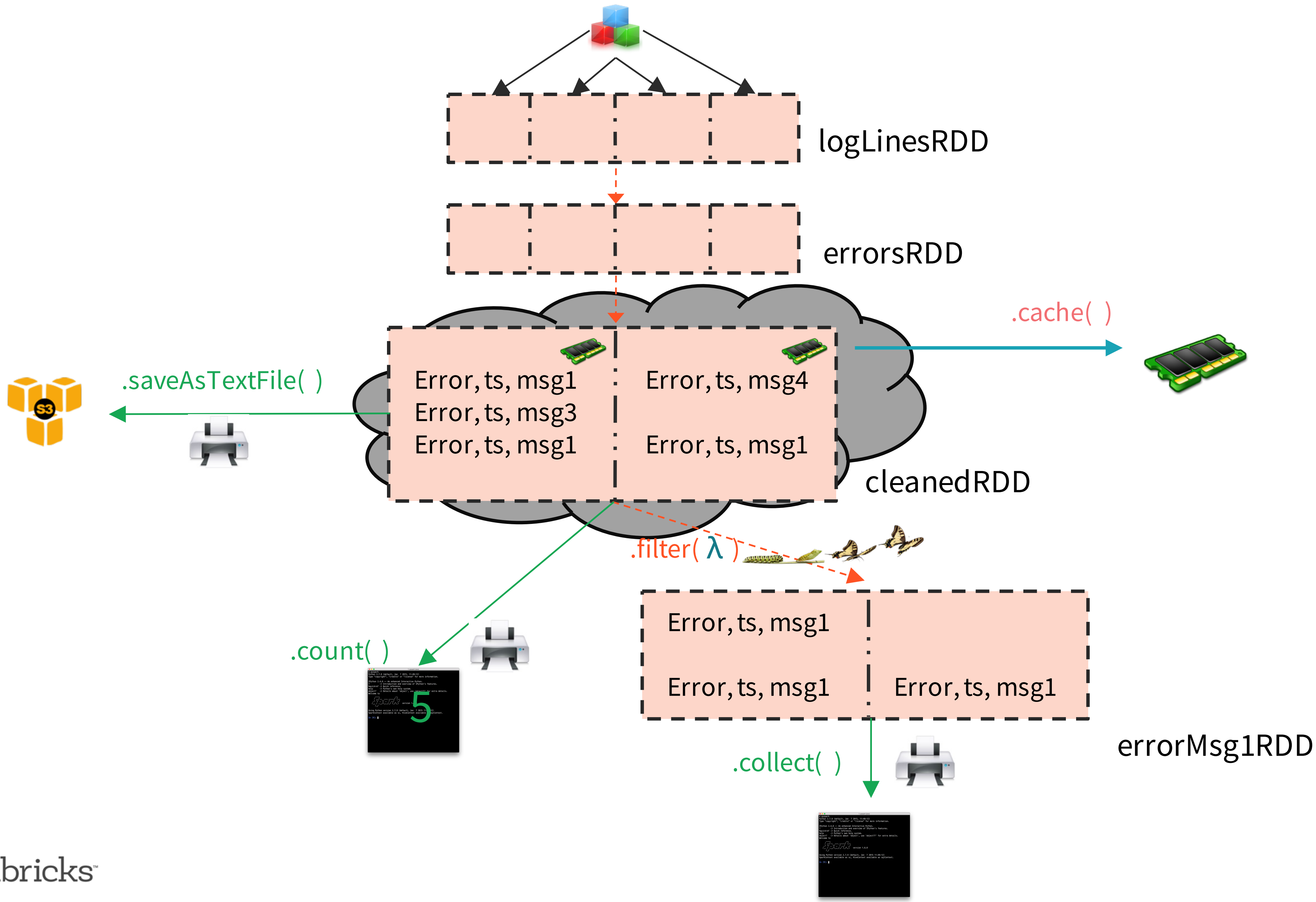


databricks

Driver

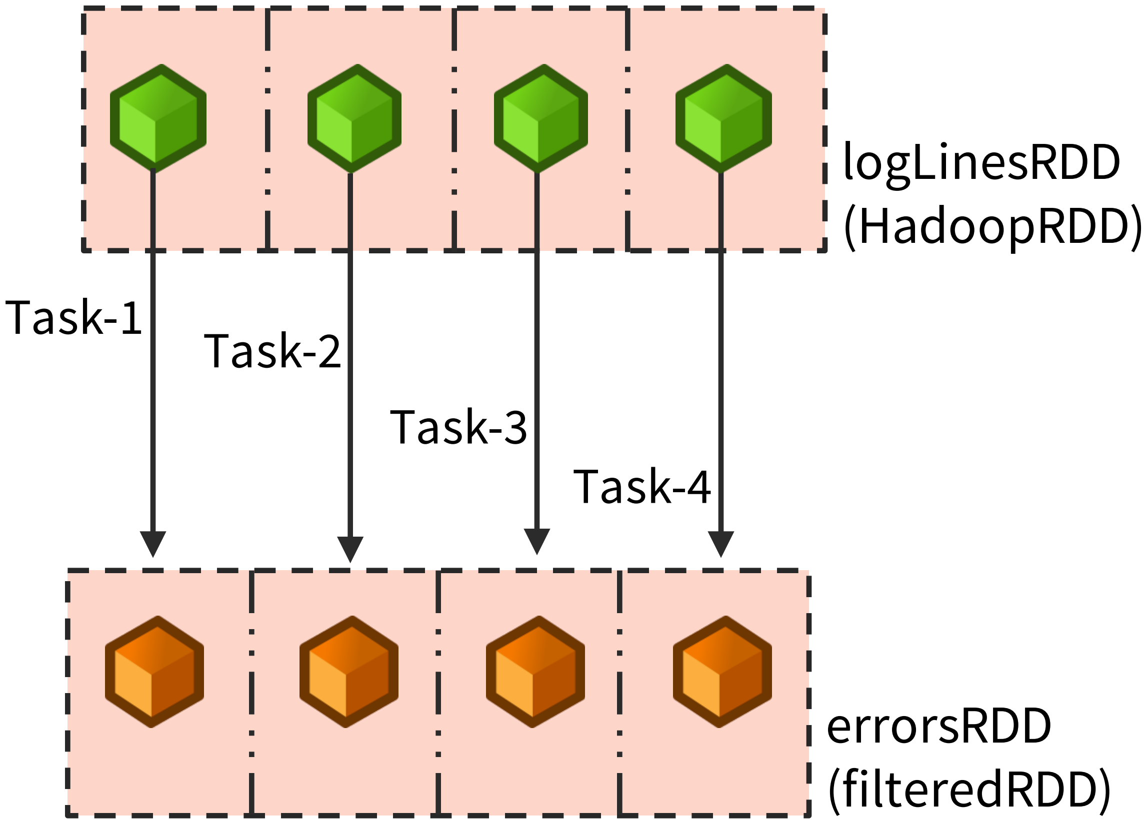

logLinesRDD
errorsRDD
Error, ts, msg1
Error, ts, msg3
Error, ts, msg1
Error, ts, msg4
Error, ts, msg1
cleanedRDD
.saveAsTextFile( )
.filter(λ)
Error, ts, msg1
Error, ts, msg1
Error, ts, msg1
errorMsg1RDD
.count( )
5
.collect( )

databricks


logLinesRDD
errorsRDD
.cache( )
.saveAsTextFile( )
Error, ts, msg1
Error, ts, msg3
Error, ts, msg1
Error, ts, msg4
Error, ts, msg1
cleanedRDD
.filter(λ)
.count( )
5
Error, ts, msg1
Error, ts, msg1
Error, ts, msg1
errorMsg1RDD
.collect( )

Partition >>> Task >>> Partition

.filter(λ)



Lifecycle of a Spark Program

1) Create some input RDDs from external data or parallelize a collection in your driver program.
2) Lazily transform them to define new RDDs using transformations like **`filter()`** or **`map()`**
3) Ask Spark to **`cache()`** any intermediate RDDs that will need to be reused.
4) Launch actions such as **`count()`** and **`collect()`** to kick off a parallel computation, which is then optimized and executed by Spark.

Transformations (lazy)

`map()`	`intersection()`	`cartesian()`
`flatMap()`	`distinct()`	`pipe()`
`filter()`	`groupByKey()`	`coalesce()`
`mapPartitions()`	`reduceByKey()`	`repartition()`
`mapPartitionsWithIndex()`	`sortByKey()`	`partitionBy()`
`sample()`	`join()`	...
`union()`	`cogroup()`	...

Actions

```
reduce()
collect()
count()
first()
take()
takeSample()
saveToCassandra()
takeOrdered()
saveAsTextFile()
saveAsSequenceFile()
saveAsObjectFile()
countByKey()
foreach()
...
```

databricks™

Some Types of RDDs

- HadoopRDD
- FilteredRDD
- MappedRDD
- PairRDD
- ShuffledRDD
- UnionRDD
- PythonRDD

- DoubleRDD
- JdbcRDD
- JsonRDD
- VertexRDD
- EdgeRDD

- CassandraRDD *(DataStax)*
- GeoRDD *(ESRI)*
- EsSpark *(ElasticSearch)*

Resilient Distributed Datasets

- Spark is RDD-centric
- RDDs are immutable
- RDDs are computed lazily
- RDDs can be cached
- RDDs know who their parents are
- RDDs that contain only tuples of two elements are “pair RDDs”

Useful RDD Actions

- take(n) – return the first n elements in the RDD as an array.
- collect() – return all elements of the RDD as an array. Use with caution.
- count() – return the number of elements in the RDD as an int.
- saveAsTextFile(‘path/to/dir’) – save the RDD to files in a directory. Will create the directory if it doesn’t exist and will fail if it does.
- foreach(func) – execute the function against every element in the RDD, but don’t keep any results.

Useful RDD Operations

map()

Apply an operation to every element of an RDD and return a new RDD that contains the results

```
>>> data = sc.textFile('path/to/file')
>>> data.take(3)
[u'Apple,Amy', u'Butter,Bob', u'Cheese,Chucky']
>>> data.map(lambda line: line.split(',')).take(3)
[[u'Apple', u'Amy'], [u'Butter', u'Bob'], [u'Cheese', u'Chucky']]
```

flatMap()

Apply an operation to every element of an RDD and return a new RDD that contains the results after dropping the outermost container

```
>>> data = sc.textFile('path/to/file')
>>> data.take(3)
[u'Apple,Amy', u'Butter,Bob', u'Cheese,Chucky']
>>> data.flatMap(lambda line: line.split(',')).take(6)
[u'Apple', u'Amy', u'Butter', u'Bob', u'Cheese', u'Chucky']
```

mapValues()

Apply an operation to the value of every element of an RDD and return a new RDD that contains the results. Only works with pair RDDs

```
>>> data = sc.textFile('path/to/file')
>>> data = data.map(lambda line: line.split(','))
>>> data = data.map(lambda pair: (pair[0], pair[1]))
>>> data.take(3)
[(u'Apple', u'Amy'), (u'Butter', u'Bob'), (u'Cheese', u'Chucky')]
>>> data.mapValues(lambda name: name.lower()).take(3)
[(u'Apple', u'amy'), (u'Butter', u'bob'), (u'Cheese', u'chucky')]
```

flatMapValues()

Apply an operation to the value of every element of an RDD and return a new RDD that contains the results after removing the outermost container. Only works with pair RDDs

```
>>> data = sc.textFile('path/to/file')
>>> data = data.map(lambda line: line.split(','))
>>> data = data.map(lambda pair: (pair[0], pair[1])).take(3)
>>> data.take(3)
[(u'Apple', u'Amy'), (u'Butter', u'Bob'), (u'Cheese', u'Chucky')]
>>> data.flatMapValues(lambda name: name.lower()).take(3)
[(u'Apple', u'a'), (u'Apple', u'm'), (u'Apple', u'y')]
```

filter()

Return a new RDD that contains only the elements that pass a filter operation

```
>>> import re
>>> data = sc.textFile('path/to/file')
>>> data.take(3)
[u'Apple,Amy', u'Butter,Bob', u'Cheese,Chucky']
>>> data.filter(lambda line: re.match(r'^[AEIOU]', line)).take(3)
[u'Apple,Amy', u'Egg,Edward', u'Oxtail,Oscar']
```

groupByKey()

Apply an operation to the value of every element of an RDD and return a new RDD that contains the results after removing the outermost container. Only works with pair RDDs

```
>>> data = sc.textFile(’path/to/file’)
>>> data = data.map(lambda line: line.split(‘,’))
>>> data = data.map(lambda pair: (pair[0], pair[1]))
>>> data.take(3)
[(u’Apple’, u‘Amy’), (u’Butter’, u’Bob’), (u’Cheese’, u‘Chucky’)]
>>> data.groupByKey().take(1)
[(u’Apple’, <pyspark.resultiterable.ResultIterable object at 0x102ed1290>)]
>>> for pair in data.groupByKey().take(1):
...   print “%s:%s” % (pair[0], “,”.join([n for n in pair[1]))
Apple:Amy,Adam,Alex
```

reduceByKey()

Combine elements of an RDD by key and then apply a reduce operation to pairs of keys until only a single key remains. Return the result in a new RDD.

```
>>> data = sc.textFile('path/to/file')
>>> data = data.map(lambda line: line.split(','))
>>> data = data.map(lambda pair: (pair[0], pair[1]))
>>> data.take(3)
[(u'Apple', u'Amy'), (u'Butter', u'Bob'), (u'Cheese', u'Chucky')]
>>> data.reduceByKey(lambda v1, v2: v1 + ":" + v2).take(1)
[(u'Apple', u'Amy:Alex:Adam')]
```

sortBy()

Sort an RDD according to a sorting function and return the results in a new RDD.

```
>>> data = sc.textFile(’path/to/file’)
>>> data = data.map(lambda line: line.split(‘,’))
>>> data = data.map(lambda pair: (pair[0], pair[1]))
>>> data.take(3)
[(u’Apple’, u‘Amy’), (u’Butter’, u’Bob’), (u’Cheese’, u‘Chucky’)]
>>> data.sortBy(lambda pair: pair[1]).take(3)
[(u’Avocado’, u’Adam’), (u‘Anchovie’, u’Alex’), (u’Apple’, u’Amy’)]
```

sortByKey()

Sort an RDD according to the natural ordering of the keys and return the results in a new RDD.

```
>>> data = sc.textFile('path/to/file')
>>> data = data.map(lambda line: line.split(','))
>>> data = data.map(lambda pair: (pair[0], pair[1]))
>>> data.take(3)
[(u'Apple', u'Amy'), (u'Butter', u'Bob'), (u'Cheese', u'Chucky')]
>>> data.sortByKey().take(3)
[(u'Apple', u'Amy'), (u'Anchovie', u'Alex'), (u'Avocado', u'Adam')]
```

subtract()

Return a new RDD that contains all the elements from the original RDD that do not appear in a target RDD.

```
>>> data1 = sc.textFile('path/to/file1')
>>> data1.take(3)
[u'Apple,Amy', u'Butter,Bob', u'Cheese,Chucky']
>>> data2 = sc.textFile('path/to/file2')
>>> data2.take(3)
[u'Wendy', u'McDonald,Ronald', u'Cheese,Chucky']
>>> data1.subtract(data2).take(3)
[u'Apple,Amy', u'Butter,Bob', u'Dinkel,Dieter']
```

join()

Return a new RDD that contains all the elements from the original RDD joined (inner join) with elements from the target RDD.

```
>>> data1 = sc.textFile('path/to/file1').map(lambda line: line.split(',')).map(lambda
pair: (pair[0], pair[1]))
>>> data1.take(3)
[(u'Apple', u'Amy'), (u'Butter', u'Bob'), (u'Cheese', u'Chucky')]
>>> data2 = sc.textFile('path/to/file2').map(lambda line: line.split(',')).map(lambda
pair: (pair[0], pair[1]))
>>> data2.take(3)
[(u'Doughboy', u'Pilsbury'), (u'McDonald', u'Ronald'), (u'Cheese', u'Chucky')]
>>> data1.join(data2).collect()
[(u'Cheese', (u'Chucky', u'Chucky'))]
>>> data1.fullOuterJoin(data2).take(2)
[(u'Apple',(u'Amy', None)), (u'Cheese', (u'Chucky', u'Chucky'))]
```

Spark Euler 计算 Pi

- Euler 公式 $\lim_{n\to\infty} \Sigma_{i=1}^{n} \frac{1}{i^2} = \frac{\pi^2}{6}$
  - 双核 CPU，分为两个 partition，平行

```
n = 1000000
ar = np.arange(n)
dat = ar.parallelize(ar, 2)
sqrs = dat.map(lambda i: 1.0/(i+1)**2)
t0 = time.time()
x = sqrs.reduce(lambda a,b: a+b)
t1 = time.time()
print("x=%f"%x)
print("time=%f"%(t1-t0))
```

例：K-means 聚类

- k 类
- 函数：寻找最近的类中心点
  - 输入：输入点 p；当前 k 个 类的中心点列表 kPoints
  - 输出：KPoints 中和 p 最近的点的 index

```
def closestPoint(p, kPoints):
    bestIndex = 0
    closest = float("+inf")
    for i in range(len(kPoints)):
        tempDist = np.sum((p - kPoints[i]) ** 2)
        if tempDist < closest:
            closest = tempDist
            bestIndex = i
    return bestIndex
```

例：K-means 聚类

- 归到 k 类
- 将 data 中的每个点 p 都映射为
  - (j, (p,1))
  - j = closestPoint(p, kPoints)
  - (p,1)是一个常见的 MapReduce 习惯用法，请掌握

```
data.map(lambda p:(closestPoint(p,kPoints),(p,1)))
```

例：聚类

- 求 k 类的中心点
  - 找出属于类 j 的所有的点，取它们坐标的均值
- 输入：$(j, (p, 1))$
- 求均值
  - 用 reduceByKey
  - j 是 Key，x 是 p，y 是 1
  - 得到一个大小为 k 的数组 $(j, (\Sigma\, p, \Sigma\, 1))$

```
reduceByKey(lambda x,y:(x[0]+y[0],x[1]+y[1]))
```

例：聚类完整代码

```
tempDist = 1.0
while tempDist > convergeDist:
    newPoints = data \
            .map( lambda p: (closestPoint(p, kPoints), (p, 1))) \
            .reduceByKey(lambda x, y : (x[0] + y[0], x[1] + y[1])) \
            .map(lambda x : (x[0], x[1][0]/ x[1][1])) \
            .collect()

    tempDist = sum(np.sum((kPoints[i] - y) ** 2)  \
                    for (i, y) in newPoints)
    for (i, y) in newPoints:
        kPoints[i] = y
```

- reduceByKey 得到大小为 k 数组 $(j, (\sum p, \sum 1))$
  - 对每一个类 j，计算 $\frac{\sum p}{\sum 1}$，得到属于它的所有点的均值
  - collect 它，作为新的 kPoints
  - 注：仅示例，这不是最好的 k-means 算法实现，Spark 机器学习库有更好的实现

Amazon EMR

- 在集群上部署 Hadoop 需要一组系统的专业人员
- 在公有云上可以轻松创建 YARN 集群
- 需要做
  - 从预配置的列表中选择您喜欢的工具组合，
  - 指定实例类型
  - 指定所需的工作节点数
  - 设置安全规则
  - 单击创建集群
- 大约两分钟，就可以启动并运行

Spark on EMR 示例

- 从 S3 加载一小部分 Wikipedia 访问日志（从 2008 年到 2010 年)

```
rawdata = sc.textFile("s3://support.
     elasticmapreduce/bigdatademo/sample/wiki")
rawdata.count()
rawdata.getNumPartitions()
```

- 将 RDD 重新划分为 10 段，以便后面更好地利用 Spark 的并行性

```
rawdata = rawdata.repartition(10)
```

Spark on EMR 示例

- 通过分割空白字符将每行转换为一个数组。

```
def parseline(line):
    return np.array([x for x in line.split(' ')])

data = rawdata.map(parseline)
```

Spark on EMR 示例

- 过滤，留下有 namelist 中名字的 row

```
def filter_fun(row, titles):
    for title in titles:
        if row[1].find(title) > -1:
            return True
    else:
        return False
```

```
fd=data.filter(lambda p:filter_fun(p,namelist))
```

Spark on EMR 示例

- 检查页面标题中人的名字是否在 names 列表里

```
def mapname(row, names):
    for name in names:
        if row[1].find(name) > -1:
            return name
    else:
        return 'huh?'
```

- Map：用（name，count）对替换每一行
- Reduce by name: 加 count

```
rd=fd.map(lambda row:(
        mapname(row,namelist),int(row[2])))
     .reduceByKey(lambda v1, v2: v1+v2)
```

Plan Optimization & Execution

- Represented internally as a “logical plan”
- Execution is lazy, allowing it to be optimized by Catalyst

图执行模型

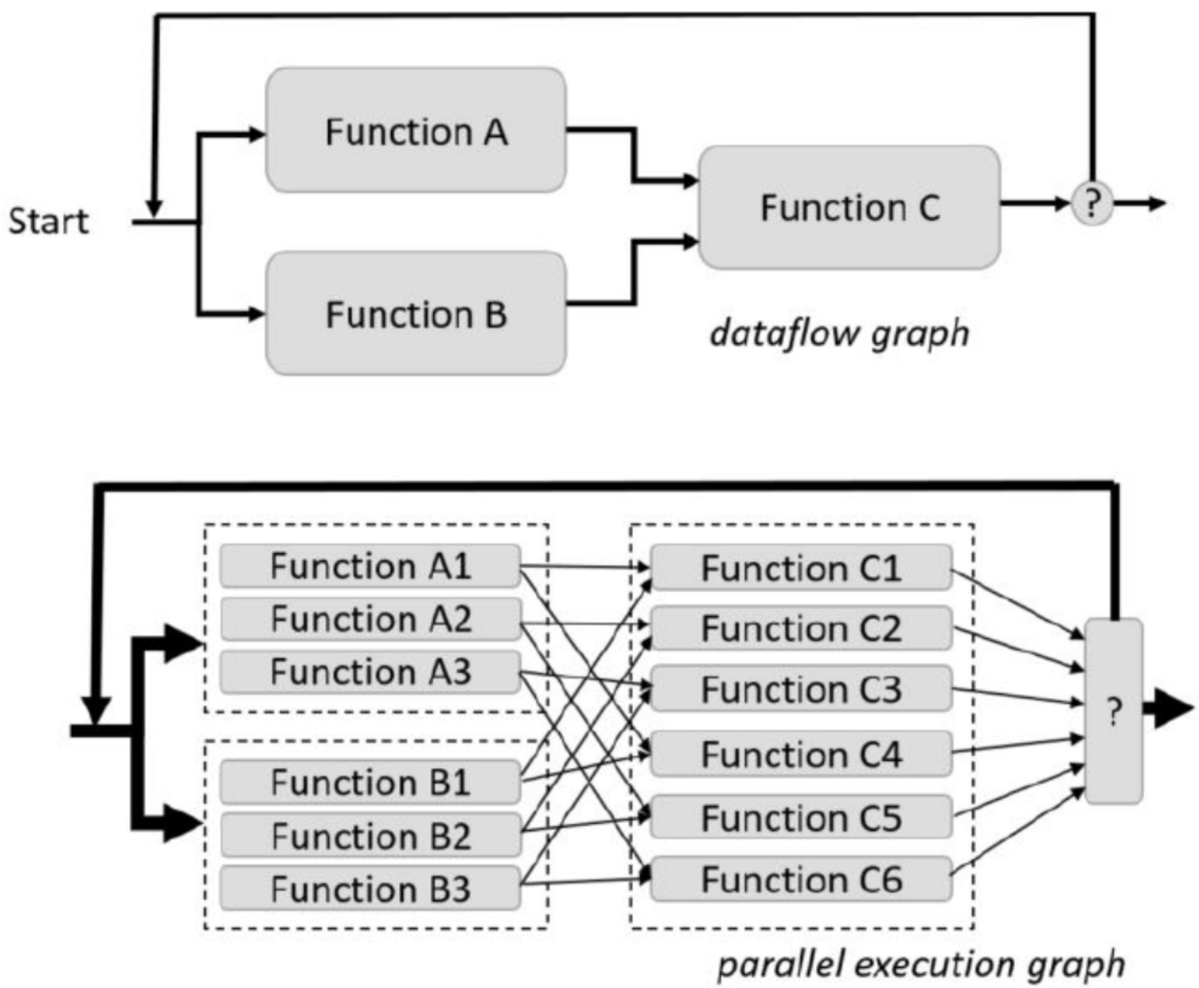
- 计算由有向图（通常为非循环图）的任务图表示
- 执行从图的源开始
- 当节点的所有父节点都已完成时，就安排该节点执行
- 图节点执行涉及一个或多个分布式节点的并行操作

图执行模型

- 可以手动构造图形，也可以由编译器从程序中隐式或显式地构造图形
- 大数据工具
  - Spark，Apache Flink，Storm，Google Dataflow
- 机器学习工具
  - Google TensorFlow，Microsoft Cognitive Toolkit

图 Dataflow 执行

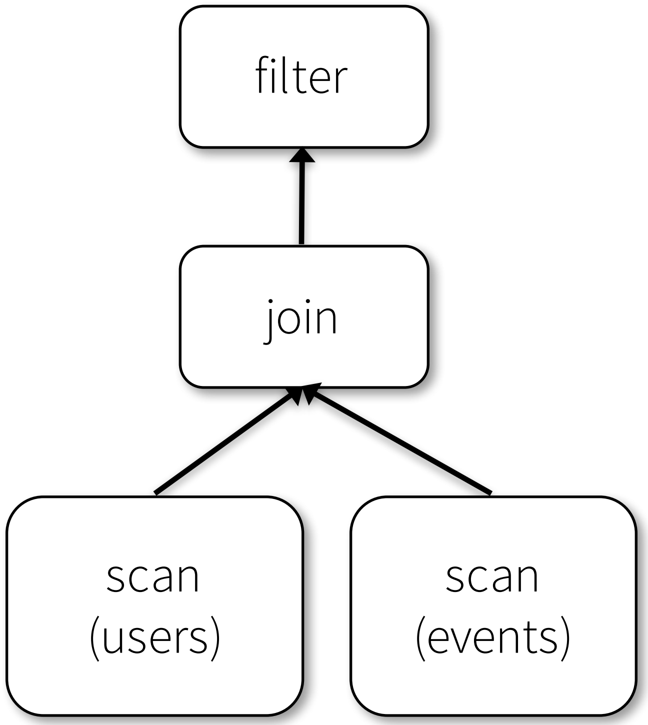
- 数据流图，编译为并行执行图



Plan Optimization & Execution

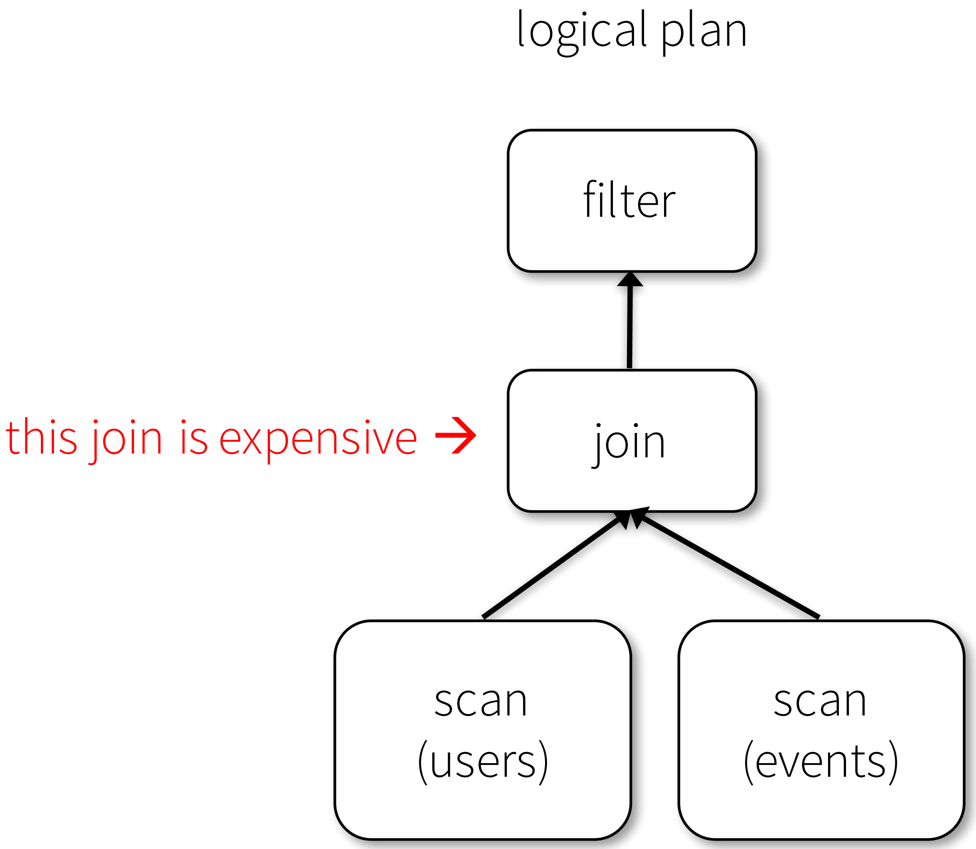
```
joined = users.join(events, users.id == events.uid)
filtered = joined.filter(events.date >= "2015-01-01")
```

logical plan



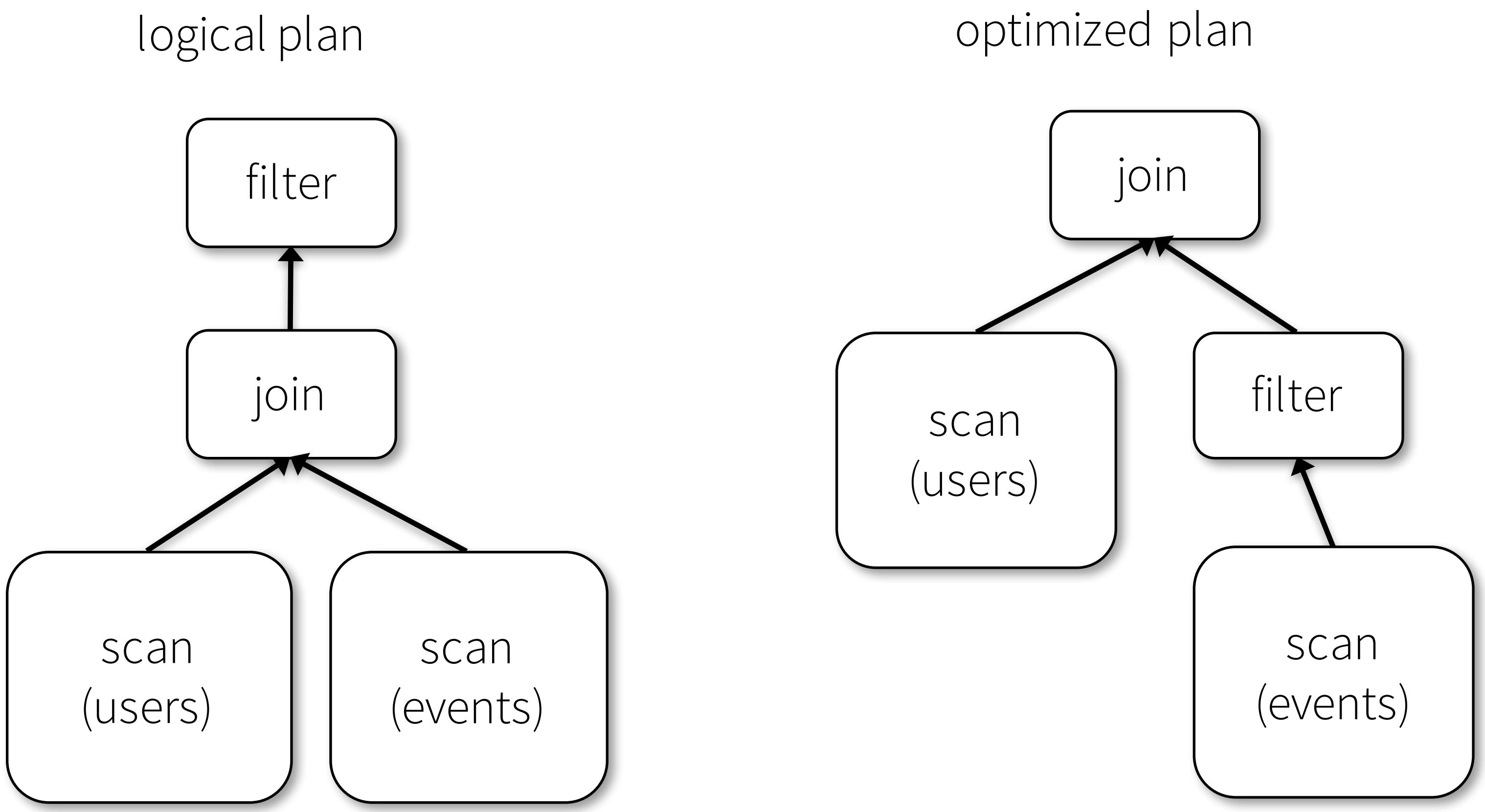
Plan Optimization & Execution

```
joined = users.join(events, users.id == events.uid)
filtered = joined.filter(events.date >= "2015-01-01")
```



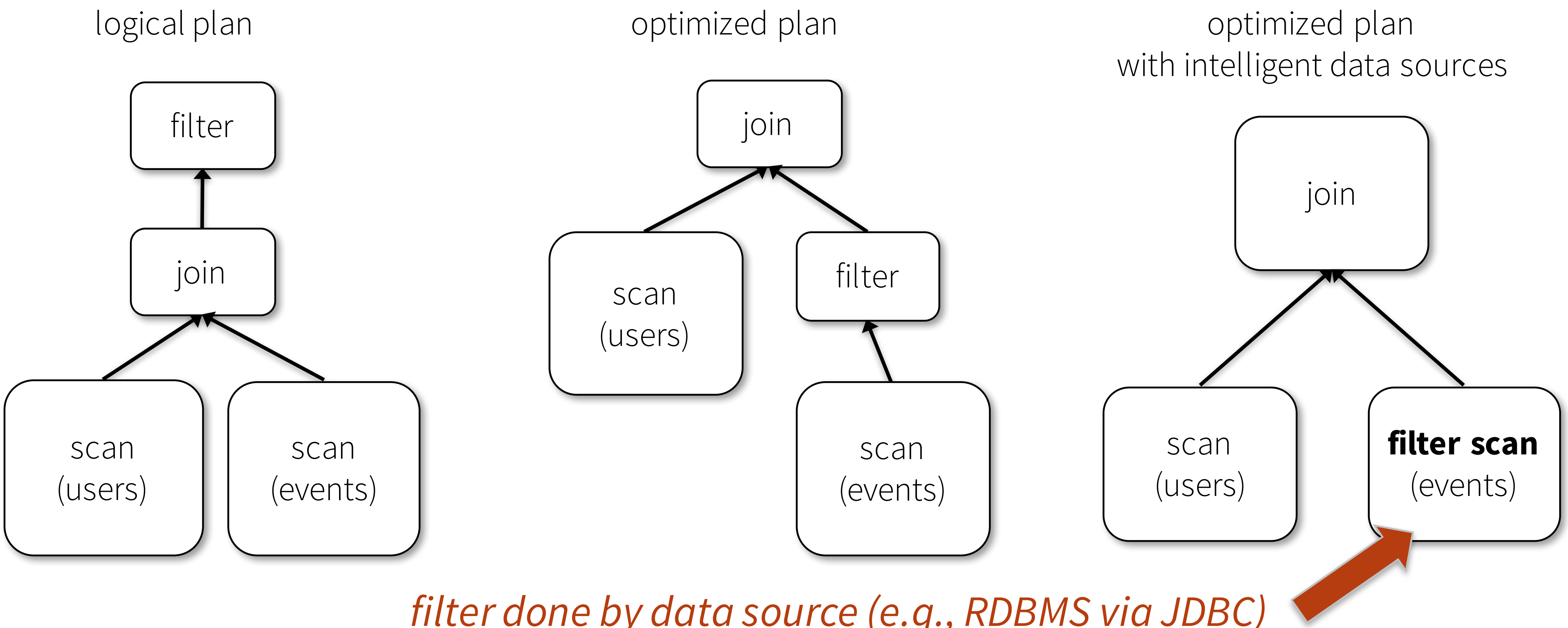
Plan Optimization & Execution

```
joined = users.join(events, users.id == events.uid)
filtered = joined.filter(events.date >= "2015-01-01")
```



Plan Optimization & Execution

```
joined = users.join(events, users.id == events.uid)
filtered = joined.filter(events.date >= "2015-01-01")
```



filter done by data source (e.g., RDBMS via JDBC)