

# Spark 功能架构与编程

# 提纲

1

Spark 简介

2

Spark 功能与架构

3

Spark 生态圈介绍

4

Spark 编程

# Spark 简介

- 是什么
  - Spark系统是**分布式批处理系统和分析挖掘引擎**；
  - AMP LAB贡献到Apache社区的开源项目，是AMP大数据栈的基础组件；
- 做什么
  - **数据处理**（Data Processing）：可以用来快速处理数据，兼具容错性和可扩展性。
  - **迭代计算**（Iterative Computation）：支持迭代计算，有效应对多步的数据处理逻辑。
  - **数据挖掘**（Data Mining）：在海量数据基础上进行复杂的挖掘分析，可支持各种数据挖掘和机器学习算法。

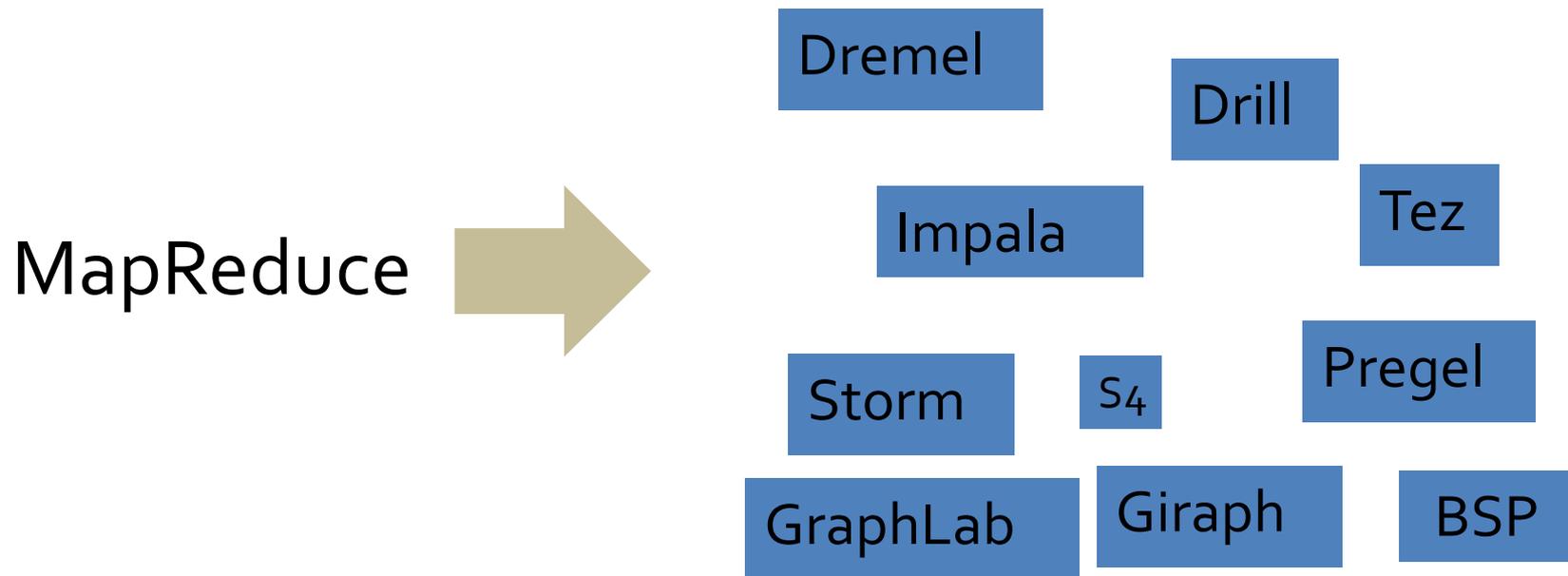
# Spark 特点

- 轻： Spark核心代码有3万行。
  - Scala语言的简洁和丰富表达力
  - 巧妙利用了Hadoop和Mesos的基础设施
- 快： Spark对小数据集可达到亚秒级的延迟，对大数据集的迭代机器学习、即席查询、图计算等应用，Spark版本比基于MR、Hive和Pregel的实现快。
  - 内存计算、数据本地性和传输优化、调度优化
- 灵： Spark提供了不同层面的灵活性。
  - Scala trait动态混入策略（如可更换的集群调度器、序列化库）；
  - 允许扩展新的数据算子、新的数据源、新的language bindings（Java和Python）；
  - Spark支持内存计算、多迭代批量处理、即席查询、流处理和图计算等多种范式。
- 巧： 巧妙借力现有大数据组件。
  - Spark借Hadoop之势，与Hadoop无缝结合；
  - Shark借了Hive的势；

# 为什么需要内存计算？

- Hadoop(MapReduce)极大的简化了大数据分析，广泛地用作大规模的数据分析。但是，随着大数据需求和使用模式的扩大，用户的需求也越来越多：
  - 更复杂的**多重处理**需求（比如迭代计算, ML, Graph）
  - 低延迟的**交互式查询**需求（比如**ad-hoc query**）
- MapReduce作业结果需要到固化到硬盘上，由此产生数据备份、磁盘I/O和序列化等操作产生了大量的开销，也由于这一缺点，MapReduce计算模型的架构不适合于上述场景
- 当年MapReduce框架设计的目的为整合廉价的商用计算机的计算能力而设计的，但随着近10年来硬件的快速发展，硬件成本的降低，内存容量和CPU速度已经不再成为问题，**内存计算的时代已经到来**

# 一种解决思路



批处理系统

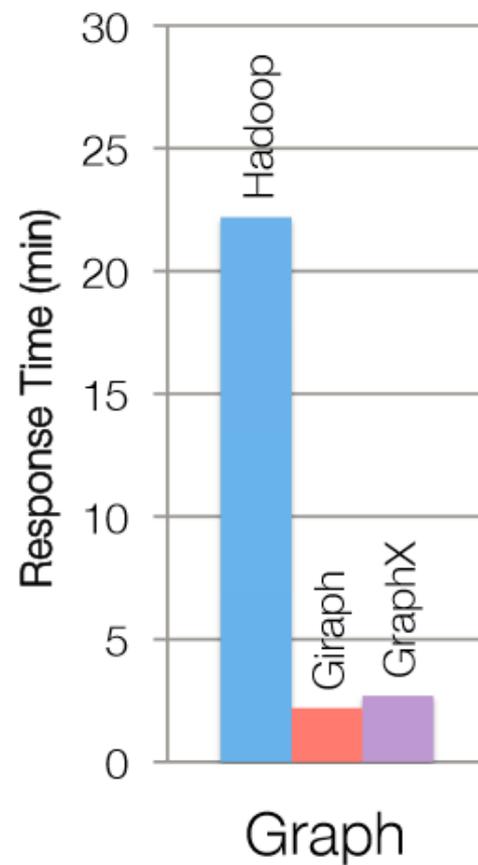
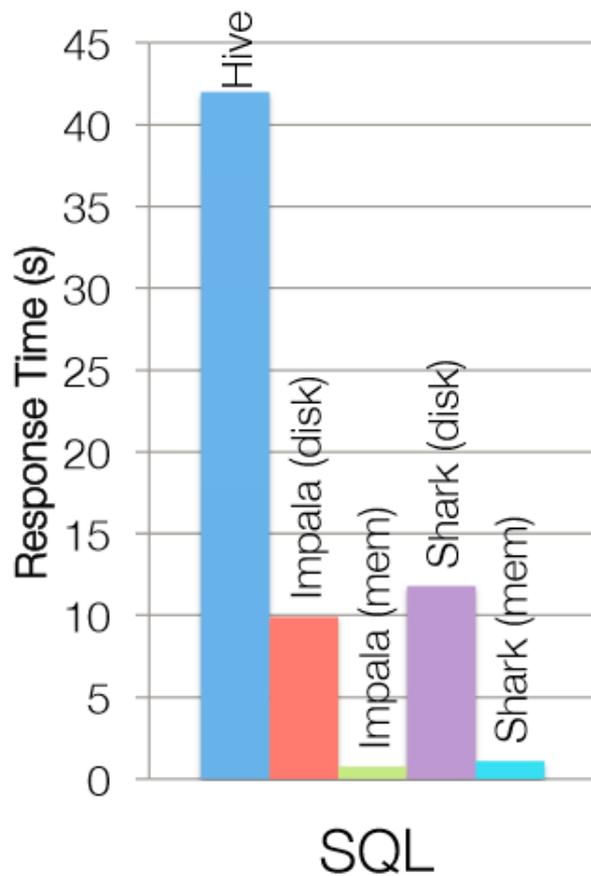
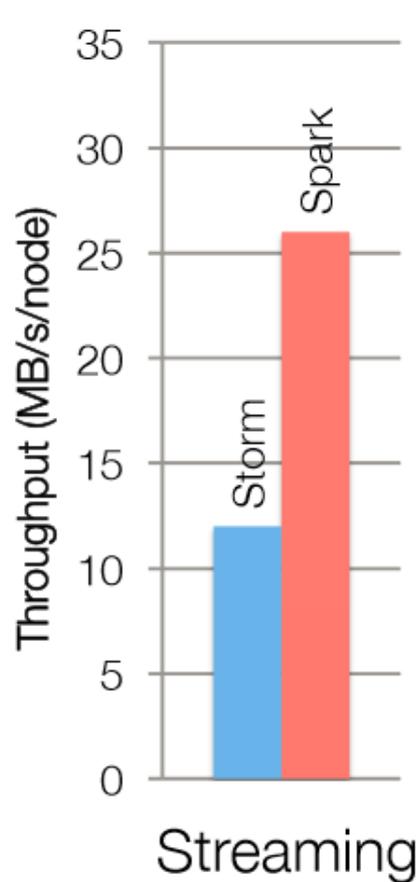
实时计算，流处理，图计算框架

# 另一种解决思路



- **不是** 一款修改过的Hadoop!
- **是** 一款独立的，高速的，开源的分布式计算引擎：
  - >> 内存计算模式
  - >> Resilient Distributed Datasets(弹性分布式数据集)
  - >> 比Hadoop快40倍以上

# Spark 与其他框架对比

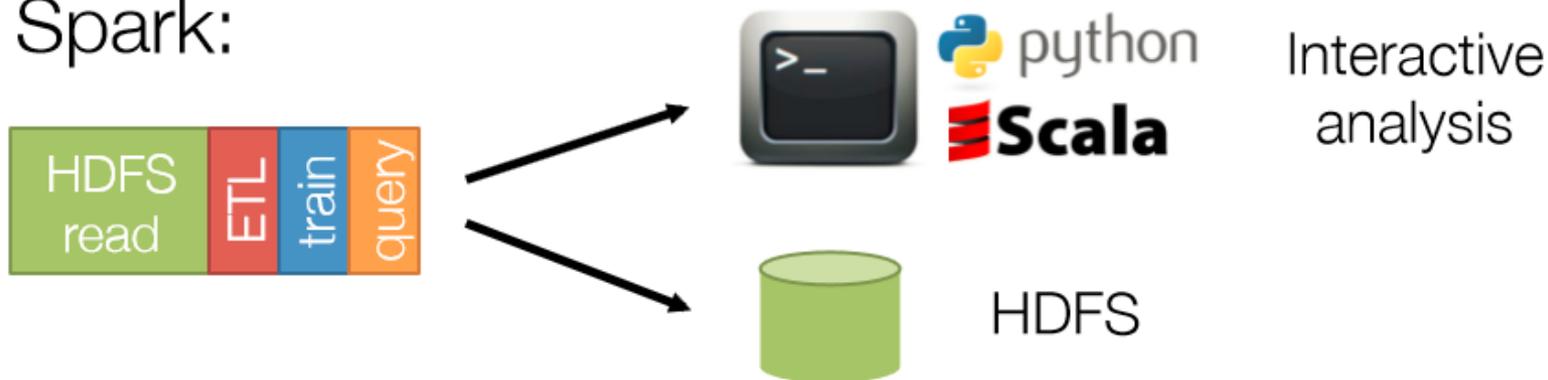


# Spark 目的

Separate frameworks:



Spark:



# 提纲

1

Spark 简介

2

Spark 功能与架构

3

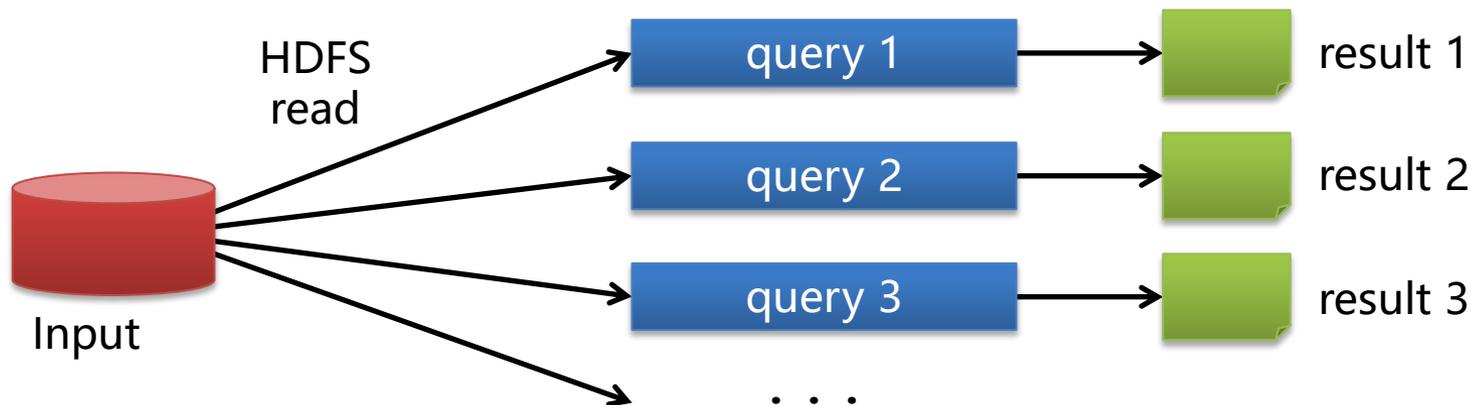
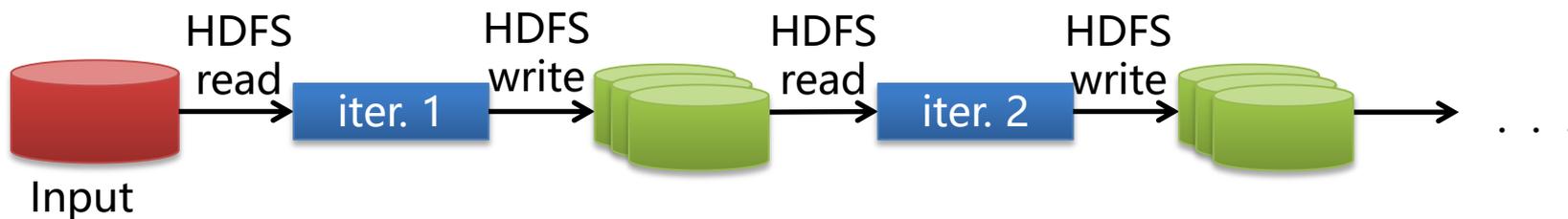
Spark 生态圈介绍

4

Spark 编程

# Spark数据共享机制 (1/2)

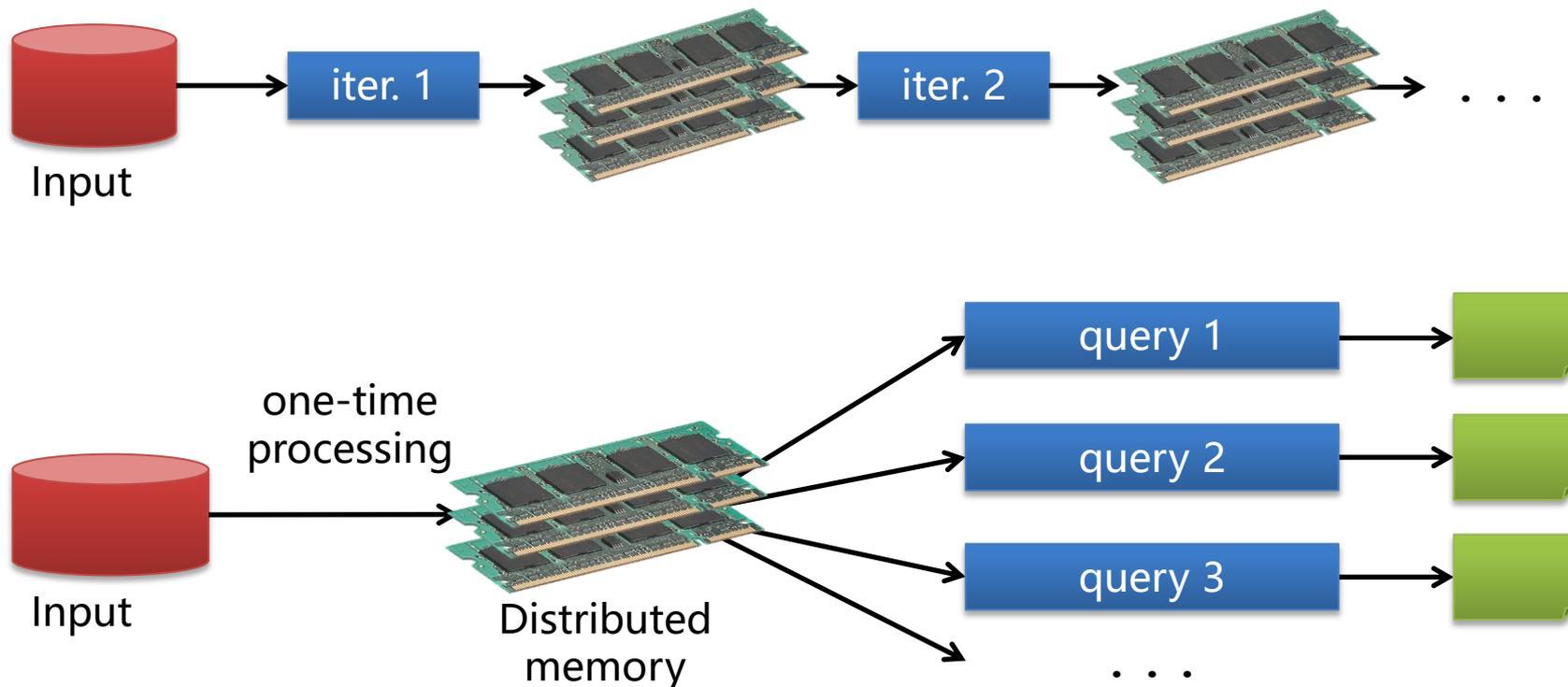
Data Sharing in MapReduce



太慢，冗余读写、序列化、磁盘IO

# Spark数据共享机制 (2/2)

## Data Sharing in Spark



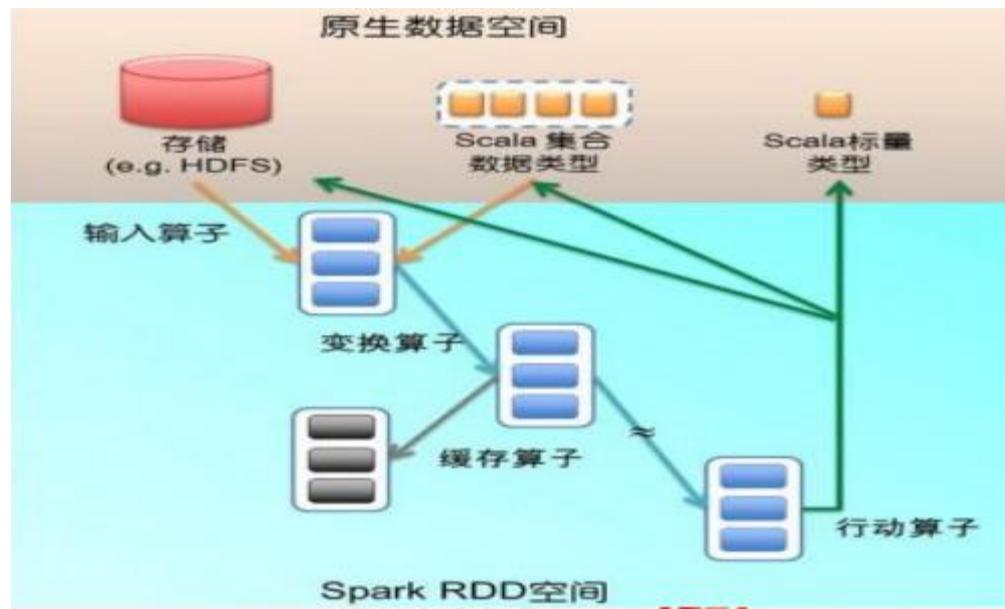
10-100x快于网络和磁盘

# Spark 核心概念-- RDDs

- 弹性分布式数据集 (Resilient Distributed Datasets)
  - A distributed memory abstraction that lets programmers perform **in-memory computations** on large clusters
  - 只读的，可分区的分布式数据集
  - 只能直接通过操作符来创建和处理
  - 支持容错处理

## • RDD 操作:

Transformation & Action



# RDD 操作

- Transformation & Action

<b>Transformations</b>	<pre>           map(f : T =&gt; U)      : RDD[T] =&gt; RDD[U]           filter(f : T =&gt; Bool) : RDD[T] =&gt; RDD[T]           flatMap(f : T =&gt; Seq[U]) : RDD[T] =&gt; RDD[U]           sample(fraction : Float) : RDD[T] =&gt; RDD[T] (Deterministic sampling)           groupByKey()          : RDD[(K, V)] =&gt; RDD[(K, Seq[V])]           reduceByKey(f : (V, V) =&gt; V) : RDD[(K, V)] =&gt; RDD[(K, V)]           union()                : (RDD[T], RDD[T]) =&gt; RDD[T]           join()                 : (RDD[(K, V)], RDD[(K, W)]) =&gt; RDD[(K, (V, W))]           cogroup()              : (RDD[(K, V)], RDD[(K, W)]) =&gt; RDD[(K, (Seq[V], Seq[W]))]           crossProduct()         : (RDD[T], RDD[U]) =&gt; RDD[(T, U)]           mapValues(f : V =&gt; W)   : RDD[(K, V)] =&gt; RDD[(K, W)] (Preserves partitioning)           sort(c : Comparator[K]) : RDD[(K, V)] =&gt; RDD[(K, V)]           partitionBy(p : Partitioner[K]) : RDD[(K, V)] =&gt; RDD[(K, V)]         </pre>
<b>Actions</b>	<pre>           count()      : RDD[T] =&gt; Long           collect()    : RDD[T] =&gt; Seq[T]           reduce(f : (T, T) =&gt; T) : RDD[T] =&gt; T           lookup(k : K) : RDD[(K, V)] =&gt; Seq[V] (On hash/range partitioned RDDs)           save(path : String) : Outputs RDD to a storage system, e.g., HDFS         </pre>

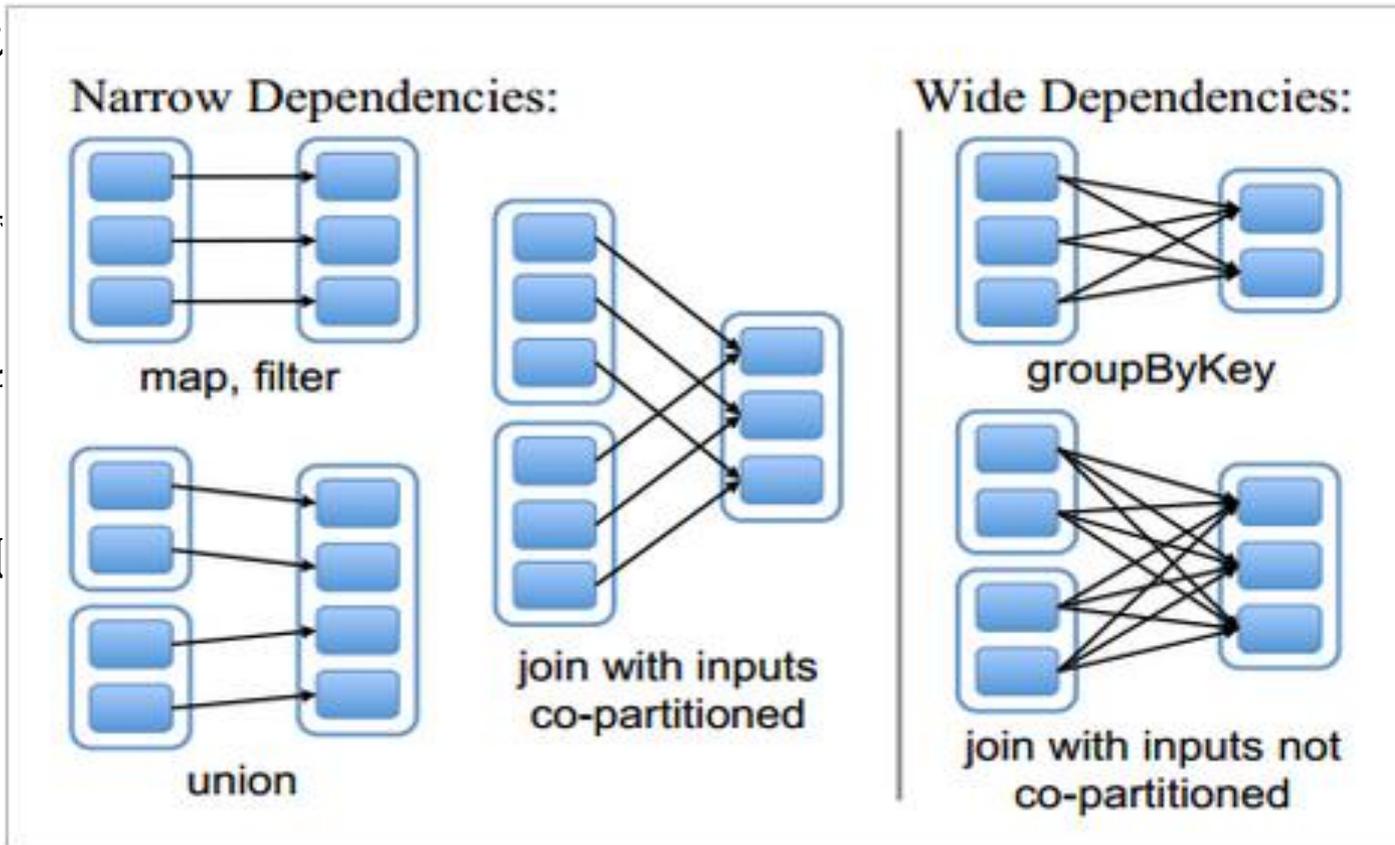
# Spark RDDs

每个RDD都有一些用来描述RDD的元数据信息：

- 一组RDD分区(partition)信息

每个RDD都会分成若干个分区，在为RDD分配task时，都是以

- 依赖于父RDD



分区依赖于父

# 窄依赖与宽依赖的区别

- **流水线操作**

窄依赖允许在一个集群节点上以流水线的方式 (pipeline) 计算所有父分区。例如，逐个元素地执行map、然后filter操作

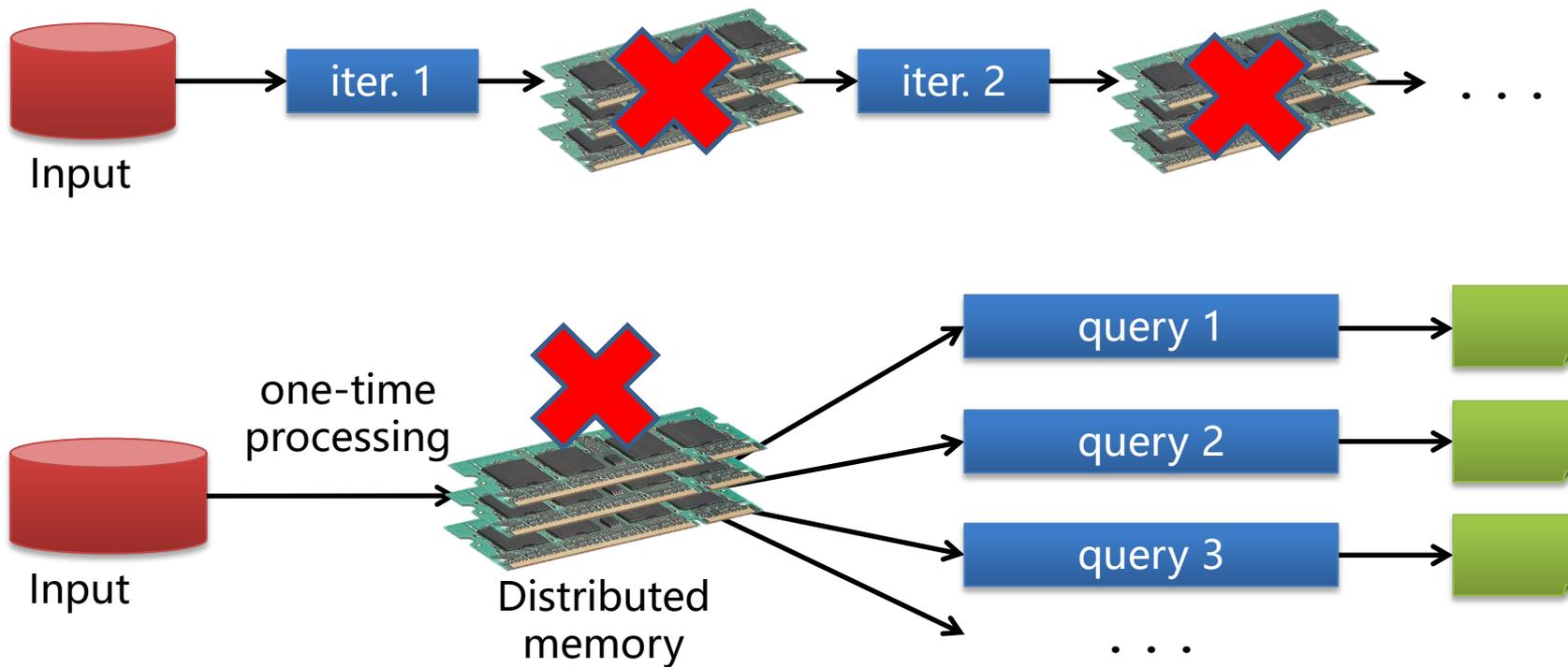
宽依赖则需要首先计算好所有父分区数据，然后在节点之间进行数据混合，这与MapReduce类似。

- **容错机制的处理**

窄依赖能够更有效地进行失效节点的恢复，即只需重新计算丢失RDD分区的父分区，而且不同节点之间可以并行计算

对于一个宽依赖关系的Lineage，单个节点失效可能导致这个RDD的所有祖先丢失部分分区，因而需要整体重新计算。

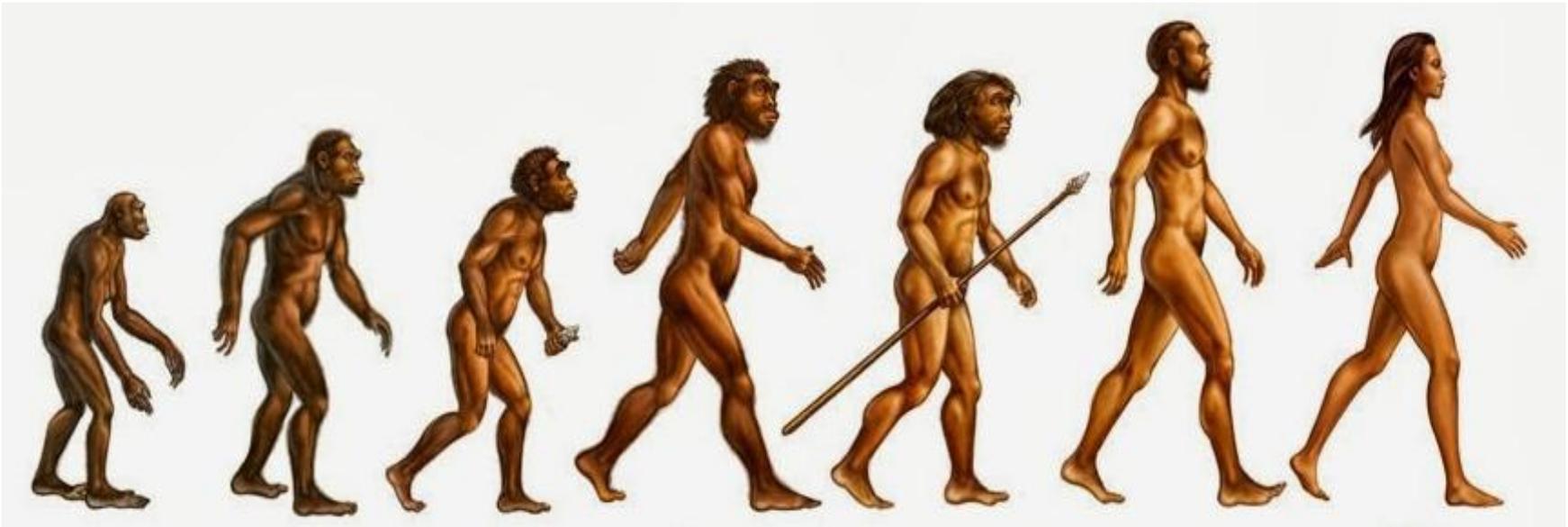
# Spark 容错机制



快的同时，也要保证系统鲁棒性

# Spark 容错机制

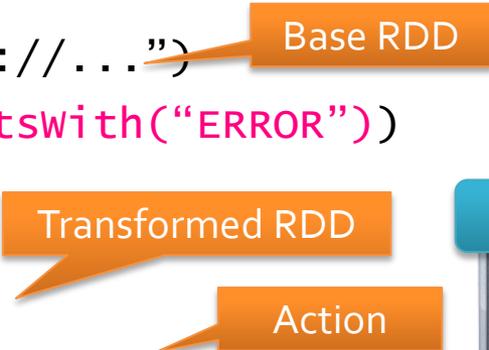
- **血统关系 (Lineage)**: 记录RDD是如何从其它RDD中演变过来的一系列操作
  - 当这个RDD的部分分区数据丢失时，它可以通过Lineage获取足够的信息来重新运算和恢复丢失的数据分区
  - 采用**粗颗粒**的数据模型，性能的提升



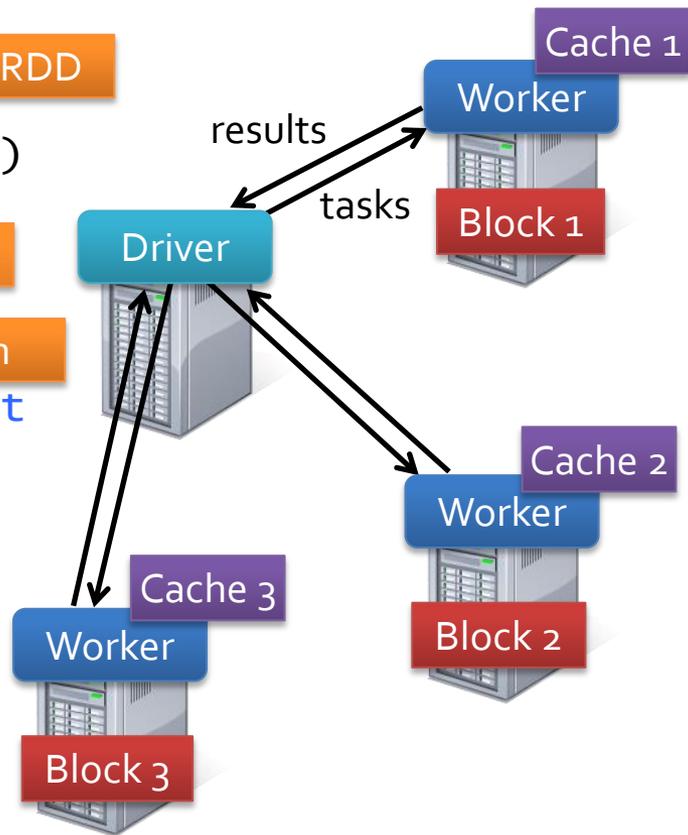
# Spark 例子

例子：假定有一个大型网站出错，操作员想要检查Hadoop文件系统（HDFS）中的日志文件（TB级大小）来找出原因。通过使用Spark，操作员只需将日志中的错误信息装载到一组节点的RAM中，然后执行交互式查询。

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
cachedMsgs = errors.cache()
cachedMsgs.filter(_.contains("MySQL")).count
cachedMsgs.filter(_.contains("HDFS")).count
. . .
```

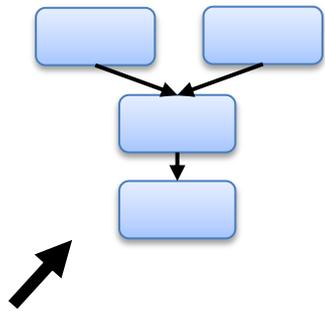


**Result:** scaled to 1 TB data in 5-7 sec  
(vs 170 sec for on-disk data)



# Spark 任务调度 (1/2)

RDD Objects

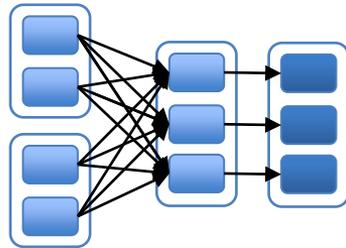


```
rdd1.join(rdd2)  
.groupBy(...)  
.filter(...)
```

build operator DAG

DAG

DAGScheduler



split graph into  
*stages* of tasks

submit each  
stage as ready

TaskSet

TaskScheduler

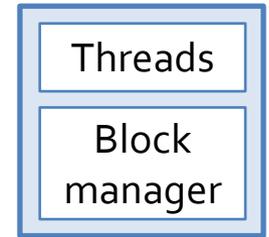


launch tasks via  
cluster manager

retry failed or  
straggling tasks

Task

Worker



execute tasks

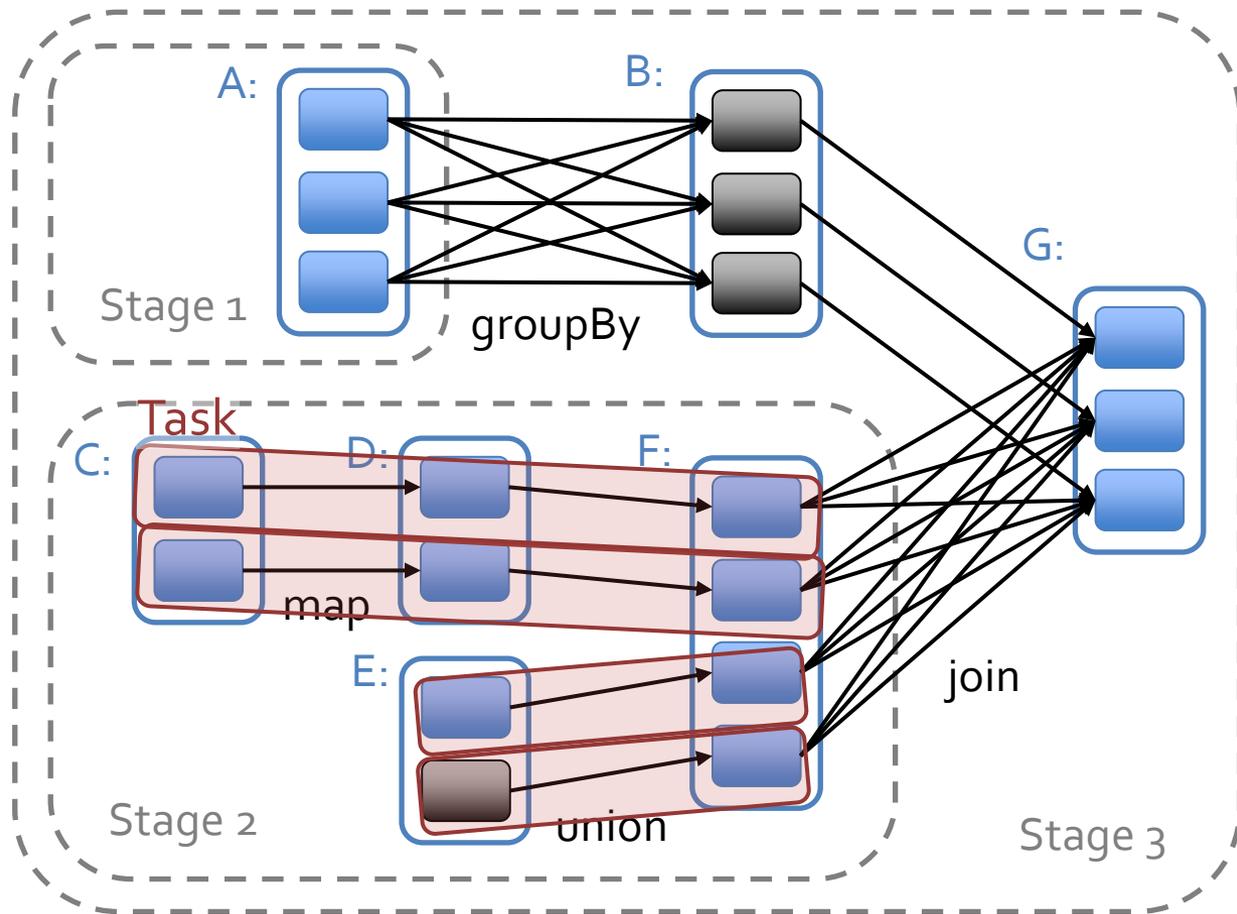
store and serve  
blocks

# Spark 任务调度 (2/2)

按窄依赖划分Stage

依据分区来划分Join操作

重用依据缓存的数据



 = previously computed partition

# 提纲

1

Spark 简介

2

Spark 功能与架构

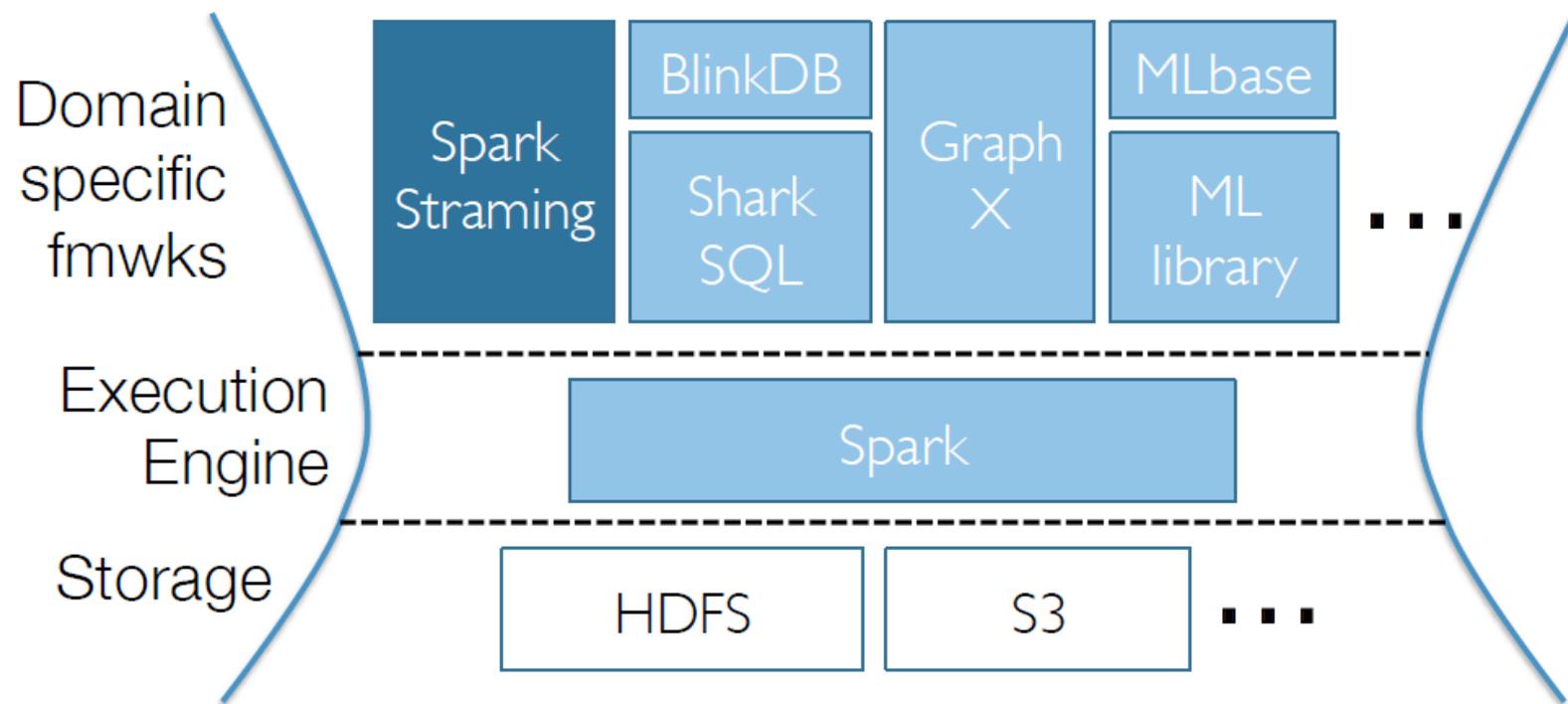
3

Spark 生态圈介绍

4

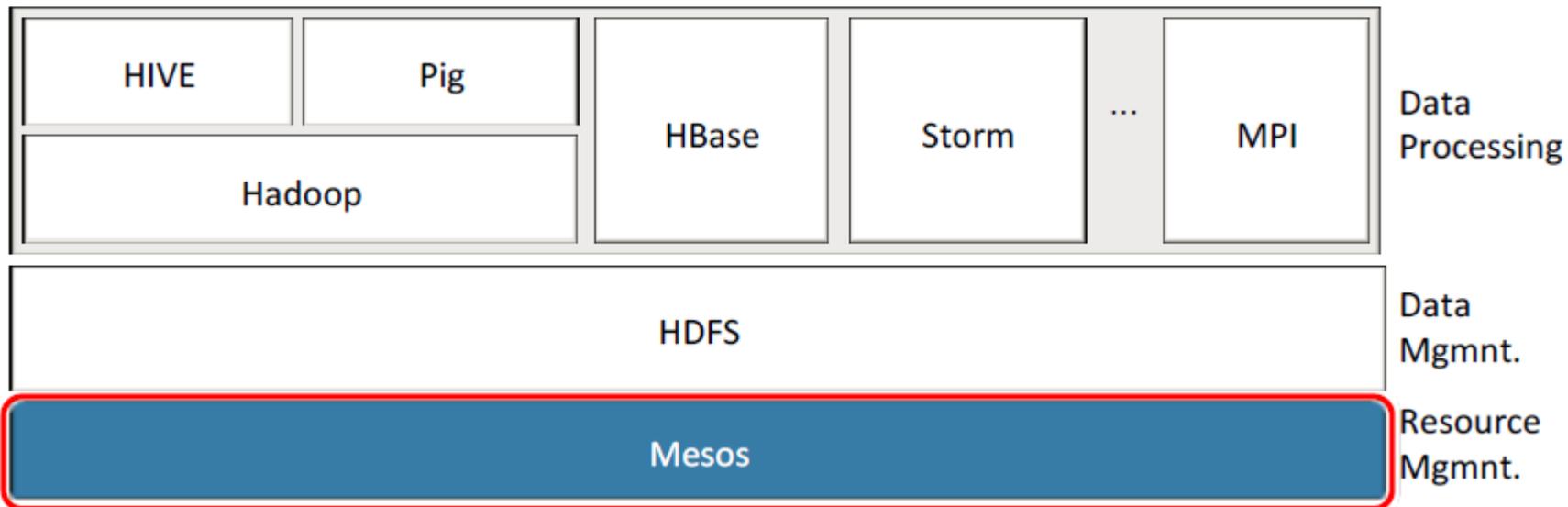
Spark 编程

# Spark 生态圈



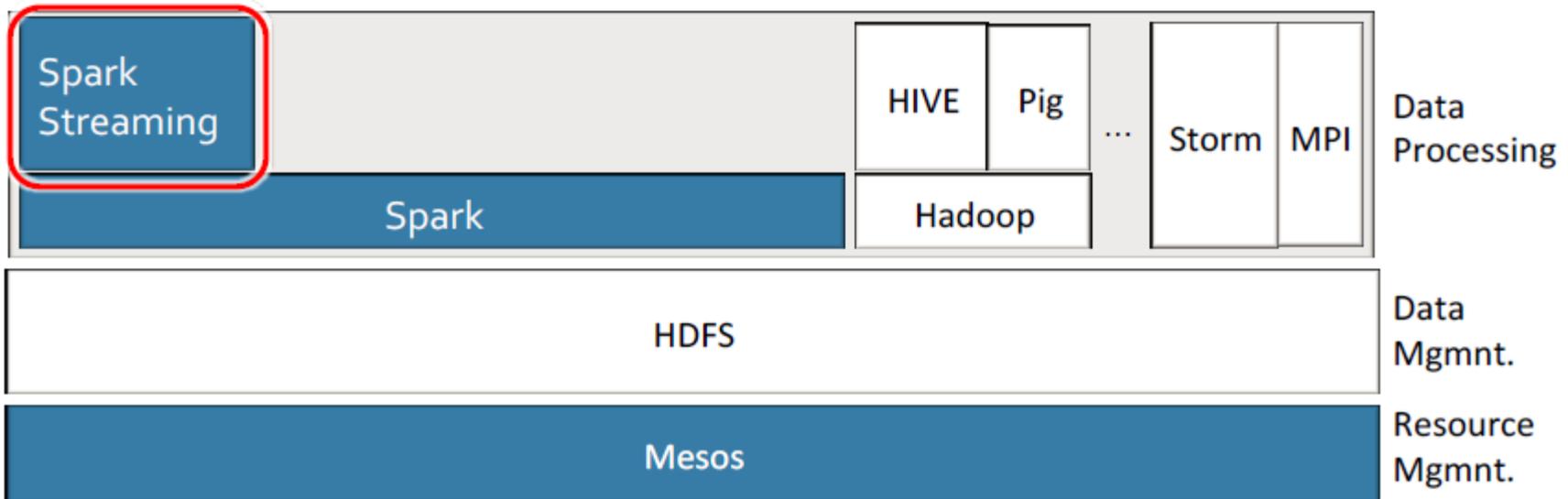
# Spark 生态圈之Mesos

1. 统一集群资源管理系统, 支持多种计算框架共享集群, eg Hadoop, MPI
2. 通过共享集群资源和数据, 提高资源利用率和数据共享率
3. 目前最大部署集群为3500+节点



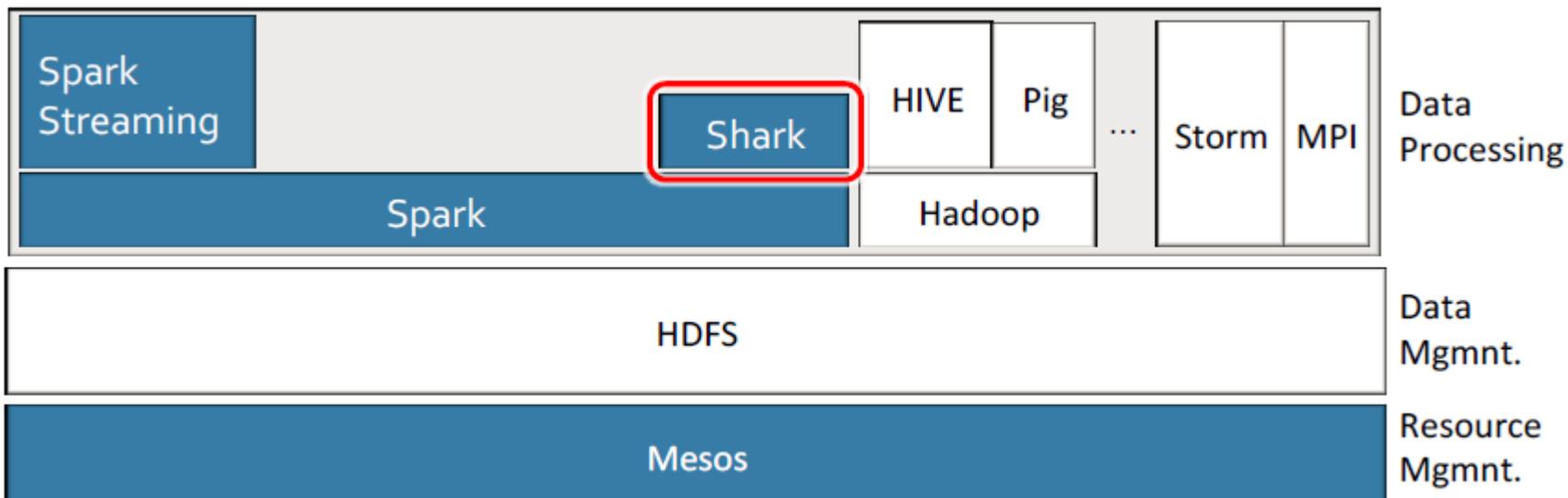
# Spark 生态圈之Spark Streaming

1. 支持大规模流式计算，吞吐量高于Storm
2. 基于Spark单一框架，完善Spark批处理、交互式处理和流式处理模式
3. 将流式计算分解成一系列小而确定的批处理作业



# Spark 生态圈之SparkSQL(Shark)

1. Hive on Spark, 提供SQL访问Spark内的RDDs
2. 比Hive性能高40-100倍
3. SparkSQL抛弃Hive, 直接SQL on Spark
4. Shark项目已经停止, 目前是单独的SparkSQL, Hive现在在做Hive on Spark



# Spark 生态圈之BlinkDB

1. 大规模的模糊查询引擎
2. 允许用户在准确率和响应时间作出权衡
3. 主要是facebook在使用和维护，最新消息  
BlinkDB加入Databricks

```
SELECT avg(sessionTime)
FROM Table
WHERE city='San Francisco'
WITHIN 2 SECONDS
```

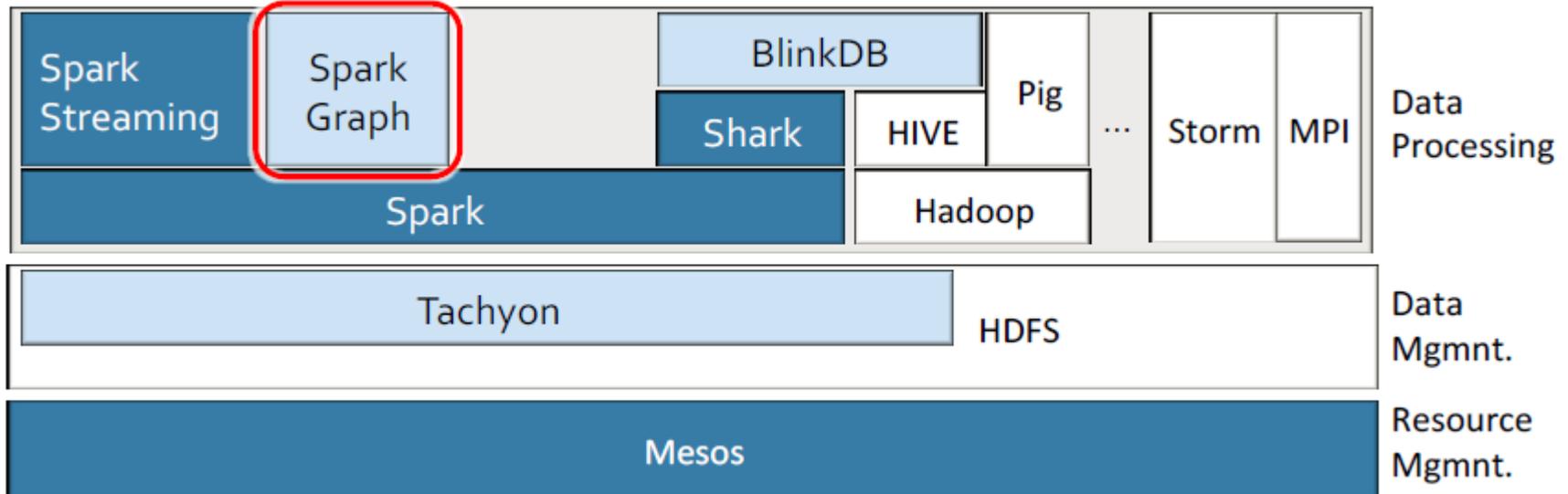
Queries with Time Bounds

```
SELECT avg(sessionTime)
FROM Table
WHERE city='San Francisco'
ERROR 0.1 CONFIDENCE 95.0%
```

Queries with Error Bounds

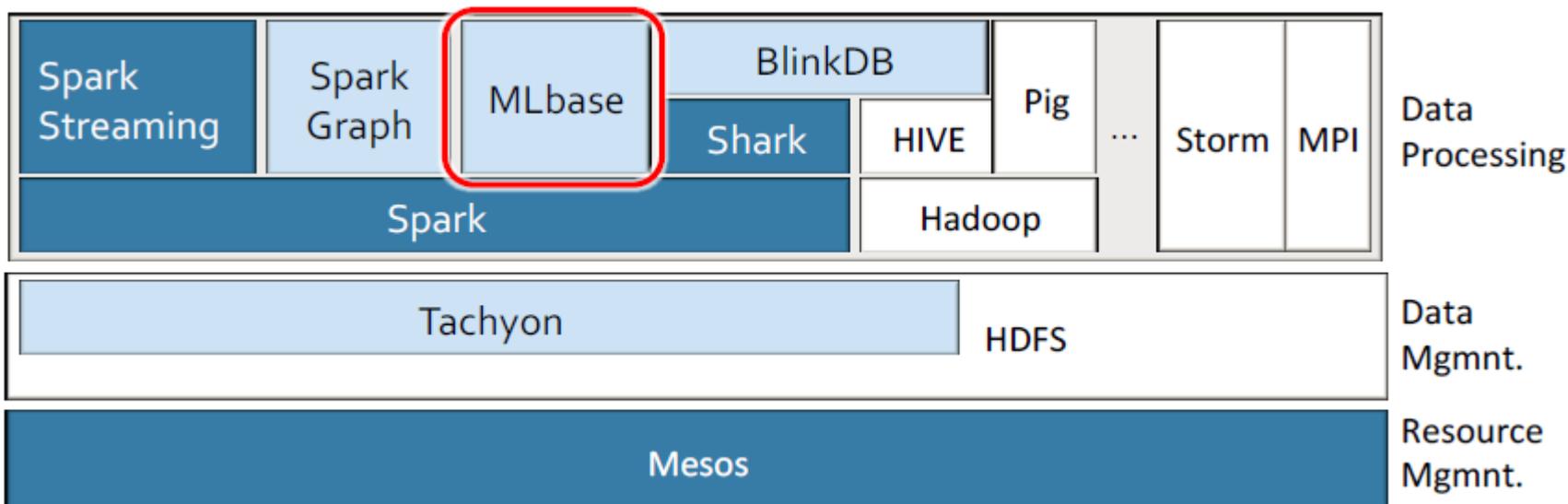
# Spark 生态圈之GraphX

1. 快速的图计算框架，性能优于Giraph和GraphLab
2. 提供GraphLib和API



# Spark 生态圈之MLBase(MLlib)

1. 基于Spark的机器学习算法包
2. 支持可扩展的机器学习算法



# 提纲

1

Spark 简介

2

Spark 功能与架构

3

Spark 生态圈介绍

4

Spark 编程

# Spark 编程

- Spark提供 Java, Python, Scala三种语言的编程接口, 官网上有详细的Api 文档
- Spark提出的最主要抽象概念是弹性分布式数据集 (resilient distributed dataset,RDD), 它是一个元素集合, 划分到集群的各个节点上, 可以被并行操作。每个RDD都封装了不同的操作, 开发者通过合理组合, 应用这些RDD函数来实现需求功能。
- Spark提供的API主要分为两类:
  - 转换 (transformation) : 用来创建新的RDD, 延迟执行, 直到遇到action
  - 行为 (action) : 返回一些值或者把数据输出到存储系统 (HDFS)
- RDDs的创建可以从HDFS上的一个文件开始, 或者通过转换操作获得
- 用户也可以让Spark保留一个RDD在内存中, 使其能在并行操作中被有效的重复使用。

# Spark 编程

- Java编程
- 将spark-assembly-1.0.0-hadoop2.2.0.jar 加到eclipse工程中，之后，export出jar，使用spark\_submit工具调用即可

```
static final String USER = "root";
private static final Pattern SPACE = Pattern.compile(" ");
public static void main(String[] args) throws Exception {
    System.out.println("....."+args[0]);

    System.setProperty("user.name", USER); // 设置访问Spark使用的用户名
    System.setProperty("HADOOP_USER_NAME", USER); // 设置访问Hadoop使用的用户名
    Map<String,String> envs = new HashMap<String,String>();

    envs.put("HADOOP_USER_NAME", USER); // 为Spark环境中服务于本App的各个Executor程序设置访问Hadoop使用的用户名

    System.setProperty("spark.executor.memory", "1024m"); // 为Spark环境中服务于本App的各个Executor程序设置使用内存量的上限

    // 以下构造sc对象的构造方法各参数意义依次为：
    // Spark Master的地址；
    // App的名称；
    // Spark Worker的部署位置；
    // 需要提供给本App的各个Executor程序下载的jar包的路径列表，这些jar包将出现在Executor程序的类路径中；
    // 传递给本App的各个Executor程序的环境信息。

    JavaSparkContext sc = new JavaSparkContext("spark://10.0.50.100:7077", "Spark App ", "/root/cdh5/spark-1.0.0-bin-hadoop2", new String[0], envs);

    String file = "hdfs://10.0.50.100:8020/user/root/lxg/a.txt";
    JavaRDD<String> data = sc.textFile(file, 4).cache();
    System.out.println(data.count());

}
```

# Spark 编程 (wordcount)

相对于Java, Python和Scala 的语法更加简洁  
Python编程

```
import sys
from operator import add

from pyspark import SparkContext

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print >> sys.stderr, "Usage: wordcount <file>"
        exit(-1)
    sc = SparkContext(appName="PythonWordCount")
    lines = sc.textFile(sys.argv[1], 1)
    counts = lines.flatMap(lambda x: x.split(' ')) \
        .map(lambda x: (x, 1)) \
        .reduceByKey(add)
    output = counts.collect()
    for (word, count) in output:
        print "%s: %i" % (word, count)
```

Scala编程(ScalaIDE, 打包方式和java一样)

```
val file=sc.textFile("hdfs://node2:8020/user/root/lxg/a.txt")
val count=file.flatMap(line => line.split(" ")).map(word => (word,1)).reduceByKey(_+_
count.collect()
```

# Spark 编程

<b>Transformations</b>	<p> <math>map(f : T \Rightarrow U) : RDD[T] \Rightarrow RDD[U]</math>  <math>filter(f : T \Rightarrow Bool) : RDD[T] \Rightarrow RDD[T]</math>  <math>flatMap(f : T \Rightarrow Seq[U]) : RDD[T] \Rightarrow RDD[U]</math>  <math>sample(fraction : Float) : RDD[T] \Rightarrow RDD[T]</math> (Deterministic sampling)  <math>groupByKey() : RDD[(K, V)] \Rightarrow RDD[(K, Seq[V])]</math>  <math>reduceByKey(f : (V, V) \Rightarrow V) : RDD[(K, V)] \Rightarrow RDD[(K, V)]</math>  <math>union() : (RDD[T], RDD[T]) \Rightarrow RDD[T]</math>  <math>join() : (RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (V, W))]</math>  <math>cogroup() : (RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (Seq[V], Seq[W]))]</math>  <math>crossProduct() : (RDD[T], RDD[U]) \Rightarrow RDD[(T, U)]</math>  <math>mapValues(f : V \Rightarrow W) : RDD[(K, V)] \Rightarrow RDD[(K, W)]</math> (Preserves partitioning)  <math>sort(c : Comparator[K]) : RDD[(K, V)] \Rightarrow RDD[(K, V)]</math>  <math>partitionBy(p : Partitioner[K]) : RDD[(K, V)] \Rightarrow RDD[(K, V)]</math> </p>
<b>Actions</b>	<p> <math>count() : RDD[T] \Rightarrow Long</math>  <math>collect() : RDD[T] \Rightarrow Seq[T]</math>  <math>reduce(f : (T, T) \Rightarrow T) : RDD[T] \Rightarrow T</math>  <math>lookup(k : K) : RDD[(K, V)] \Rightarrow Seq[V]</math> (On hash/range partitioned RDDs)  <math>save(path : String) : \text{Outputs RDD to a storage system, e.g., HDFS}</math> </p>

