Big Data Summer School

Lecture 3. In Memory Computing

Wenguang CHEN
Tsinghua University

Why MapReduce is successful?

- Programmers only need to write serial code
  - Shared/private code in OpenMP
  - message orders in MPI
- Automatically parallel and distributed computing
- Automatic load-balancing and fault-tolerance

Users require more

- More complex multi-stage applications
  - Iterative graph algorithms and machine learning, such as K-Means
- Interactive query
- Both multi-stage and interactive apps require faster **data sharing** across parallel jobs

Examples of interactive Query

```
>>> import graph
>>> weibo = graph.Graph('weibo')
Time: 1371.97
>>> import degree
>>> degree.count_degree_dist(weibo, graph.UNION)
Time: 5.68
((0, 128856),
 (1, 33929926),
 (2, 13015367),
 (3, 8519516),
 (4, 4755099),
 (5, 3841489),
 (6, 3296567),
 (7, 3007741),
 (8, 3328721),
 (9, 3090643),
 (10, 3306905),
 ...)
>>> import traversal
>>> traversal.count_bfs_length_dist(0, weibo, graph.UNION)
Time: 39.79
((1, 4),
 (2, 3528554),
 (3, 159265485),
 (4, 112063152),
 (5, 7272939),
 (6, 286339),
 (7, 53462),
 (8, 15310),
 (9, 3445),
 (10, 1195),
 (11, 520),
 (12, 204),
 (13, 126))
```

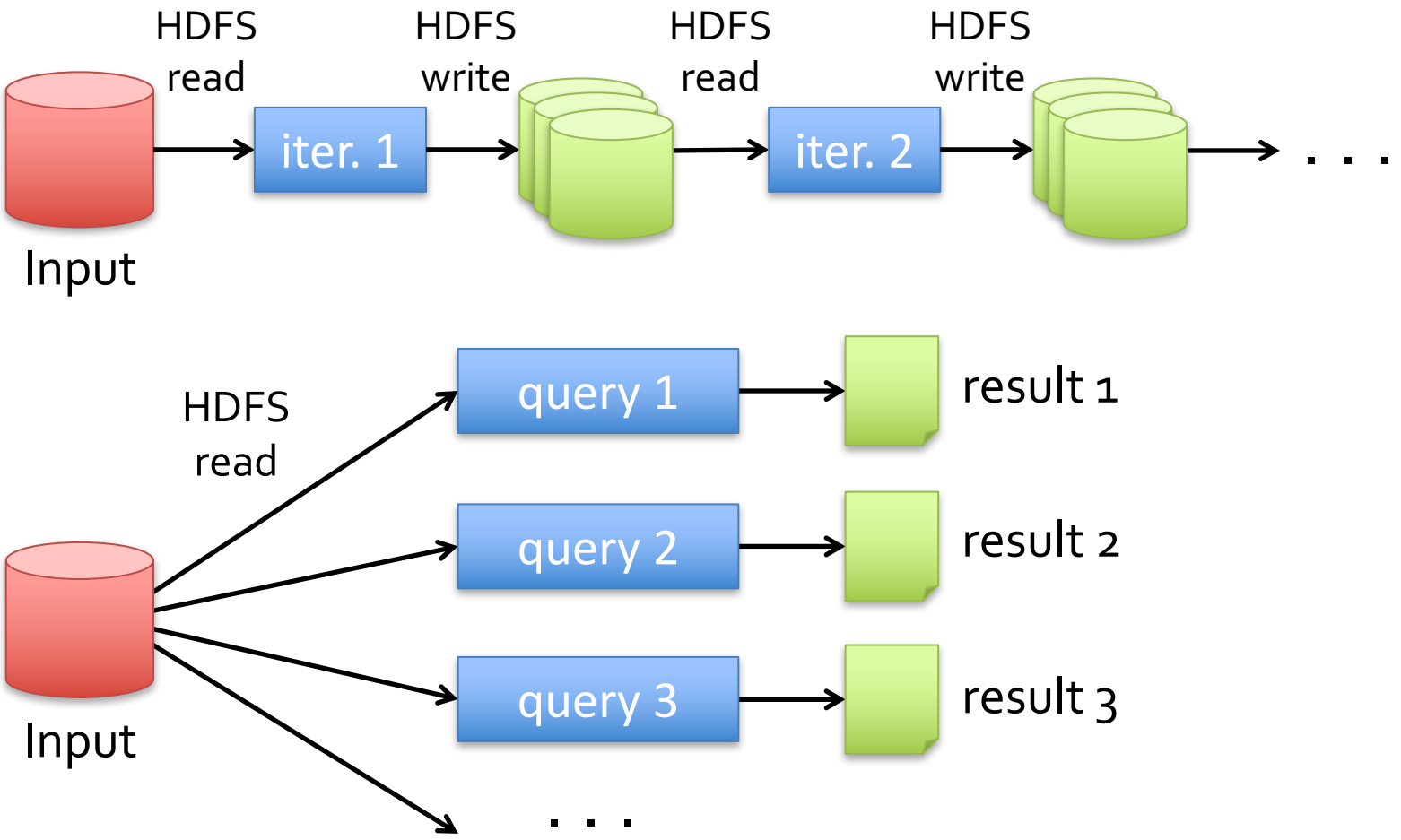
Limitations of MapReduce

- Only Map and Reduce
- Iterative MapReduce
  - Intermediate result on disk
  - Many I/O operations , poor performance
- Unable to support interactive query efficiently
  - No mechanism to share data in memory between MapReduce jobs

Read/Write data from/to disks

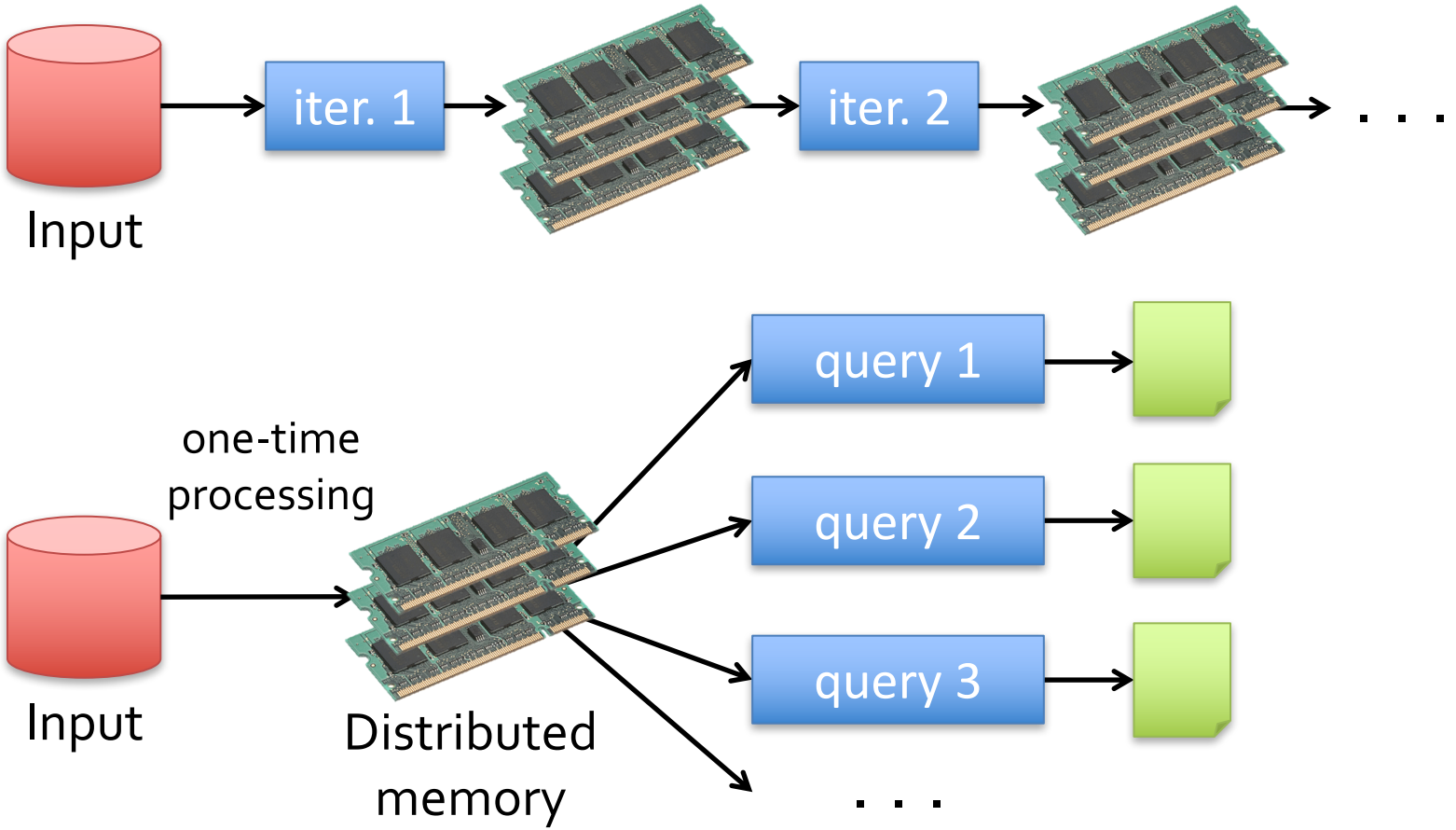
- Not only slowdown the system with poor bandwidth of disks/SSDs
- But also suffers the serialization overhead
  - Transform data into some standard, self-described format
  - Consider if there's a pointer…
  - Parsed by readers which do reserve work of serializer
- Comparable but more heavy than MPI message packaging

MapReduce Use File to Transfer data



The copy, serialization and disk I/O makes it slow

Use memory to store data

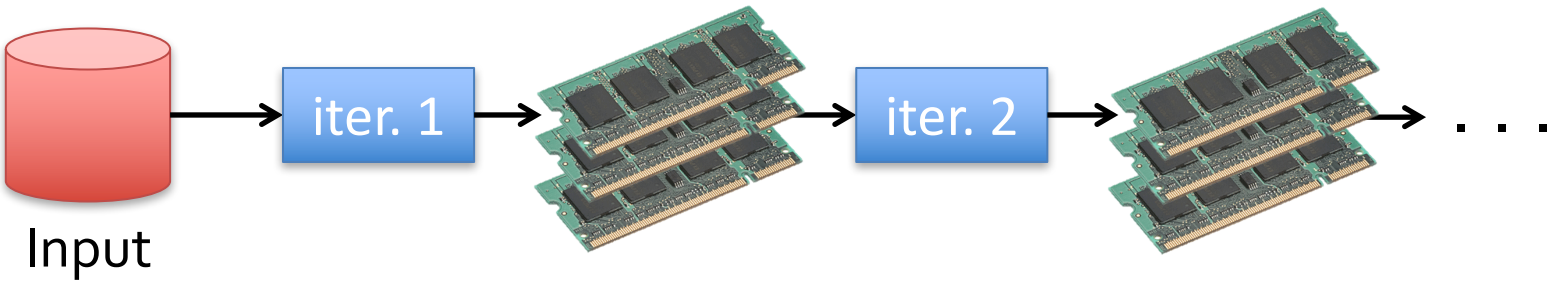


10-100 times faster than the disk approach

IS MEMORY COMPUTING FEASIBLE?

Issues

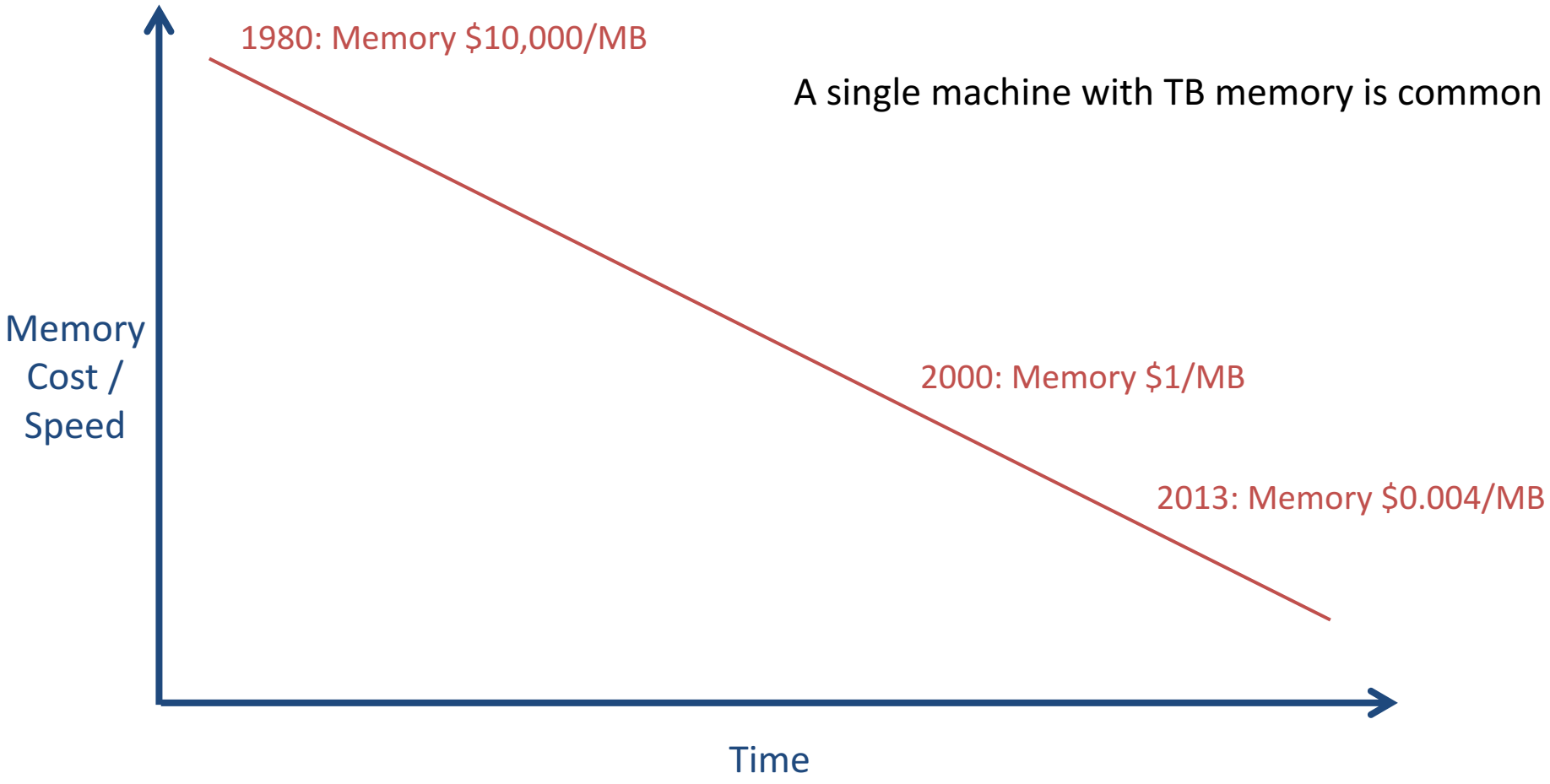
- Is memory large enough to contain all the data?
- How about the cost comparing to disks?
- Fault-tolerance with memory?
- How to represent the data in memory efficiently?



Many big data problems has its upper bound

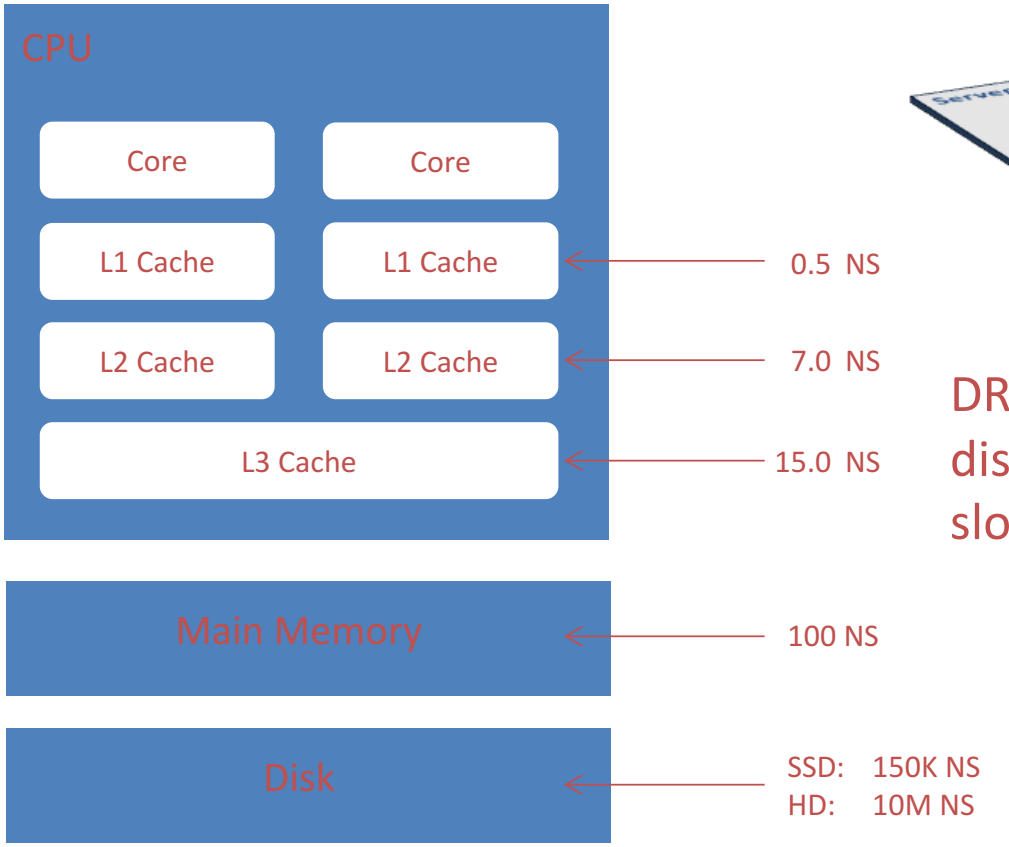
- Number of human beings
  - 10G, the size of the social network is around 10TB
- Features of a user
  - 1M
- Number of products
  - ~1M
- Yesterday's big data problems may be just today's small data problem

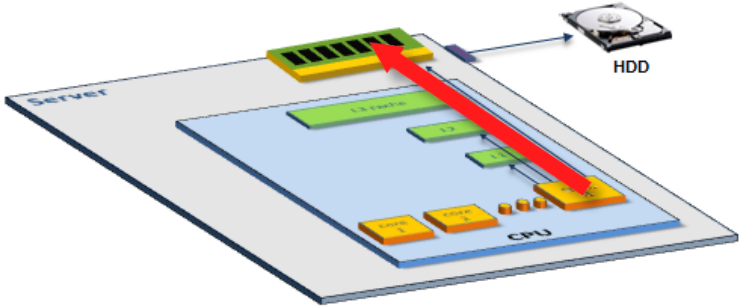
Moore's law and memory cost



Source: Cao et al. In-Memory Database Platform for Big Data

Memory latency

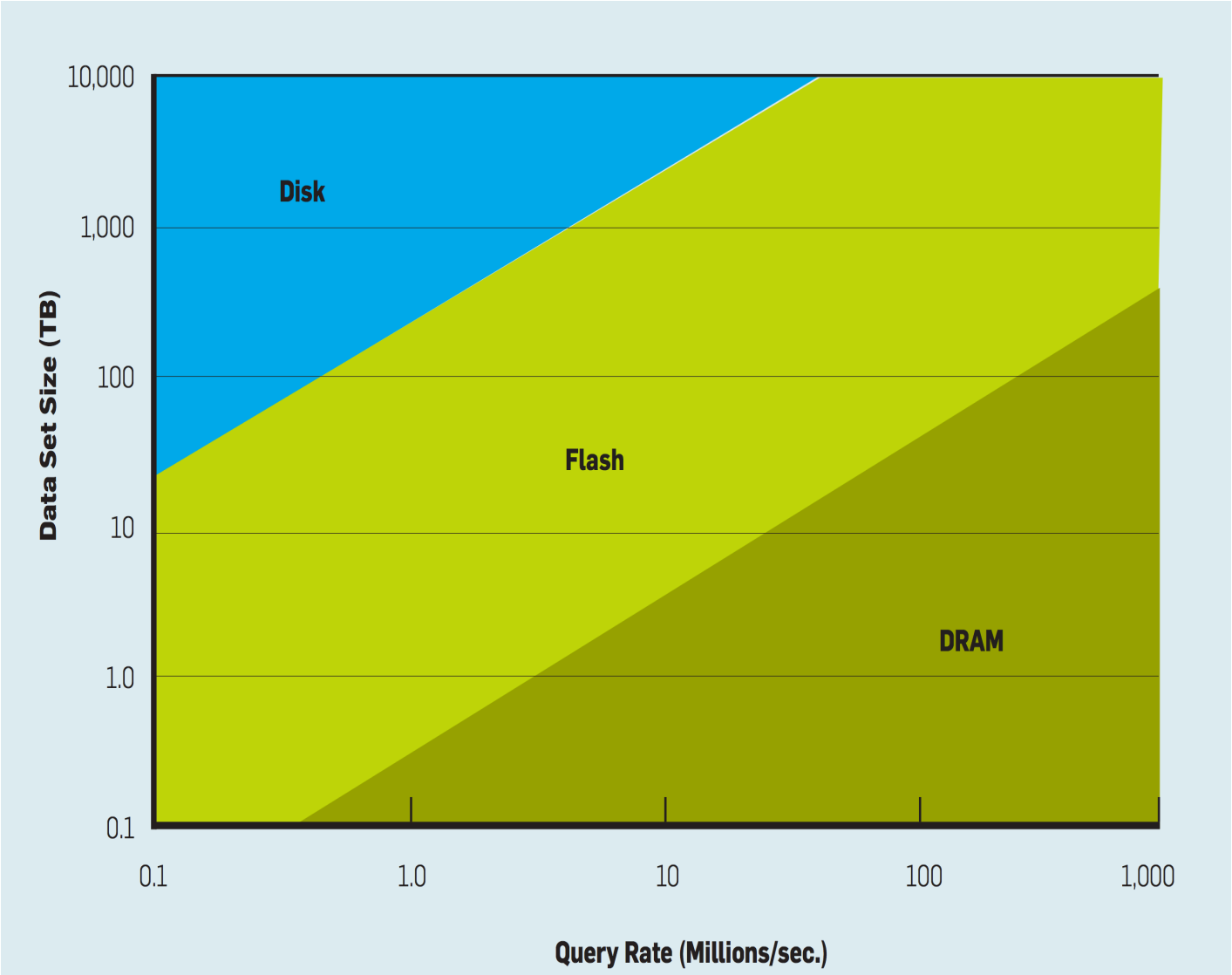




DRAM is 100,000 faster than disks， but still is 6-200 times slower than on-chip caches

Source: SAP

Performance/cost ratio with the data size and query rate



Tape is Dead
Disk is Tape
Flash is Disk
RAM Locality is King

Jim Gray

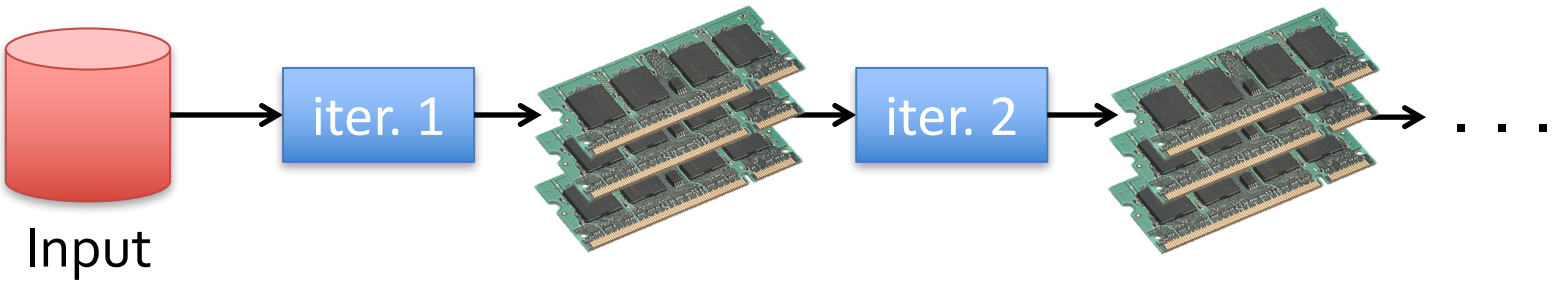
Microsoft

December 2006

In memory of Jim Gray

Issues

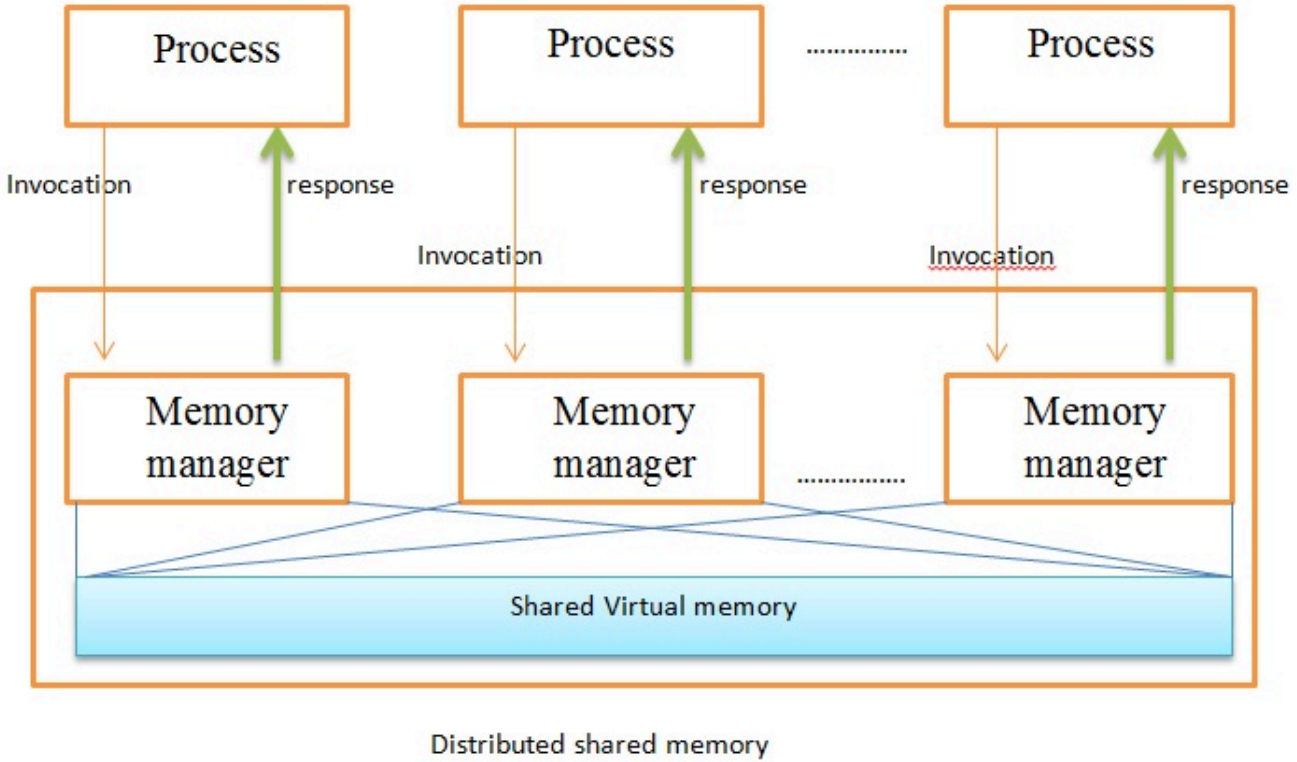
- Is memory large enough to contain all the data?
- How about the cost comparing to disks?
- Fault-tolerance with memory?
- How to represent the data in memory efficiently?



AN EXAMPLE OF IN-MEMORY COMPUTING SPARK

Abstractions on memory of multiple machines

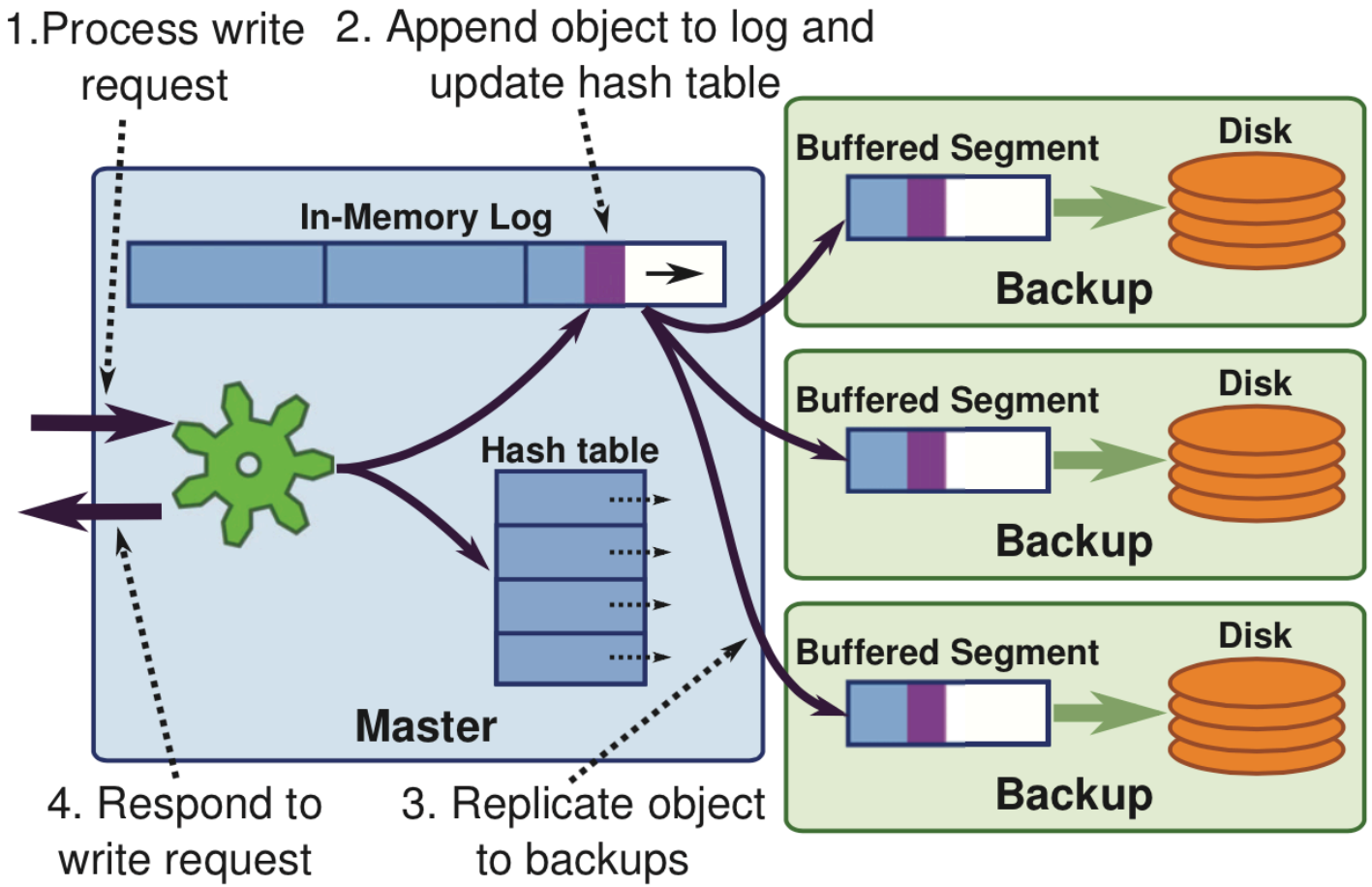
- Distributed Shared Memory
  - Unified memory space
  - Difficult for fault tolerance



Abstractions on memory of multiple machines

- Distributed Key-Value store（Piccolo，RAMCloud, Redis）
  - Allow fine-grain accesses, mutable
  - Big fault-tolerance overhead
    - Replica in memory
    - Erasure code
    - Backup in persistent storage for each write

RAMCloud Approach: Buffered Logging



Optimization: Log truncate and memory sharding

Fault tolerance mechanism of DSM or Key-value stores

- Replica or Log
  - Big overhead for data intensive applications
  - 10-100 times slower than local memory accesses

Solution

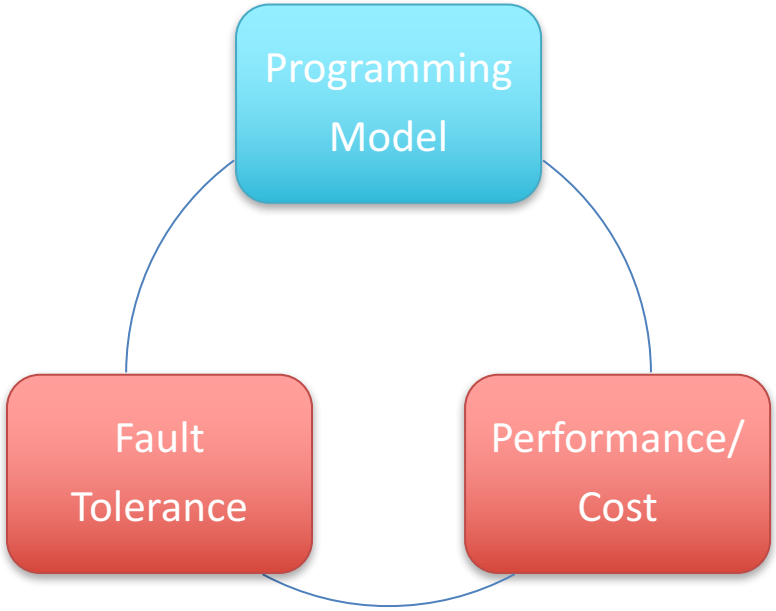
- RDD（Resilient Distributed Datasets）
  - Based on data sets, instead of single data
  - Generated with deterministic coarse grained operations（map，filter，join etc.）
  - Immutable
  - To modify data, create new datasets by transformation

Efficient Fault Tolerance

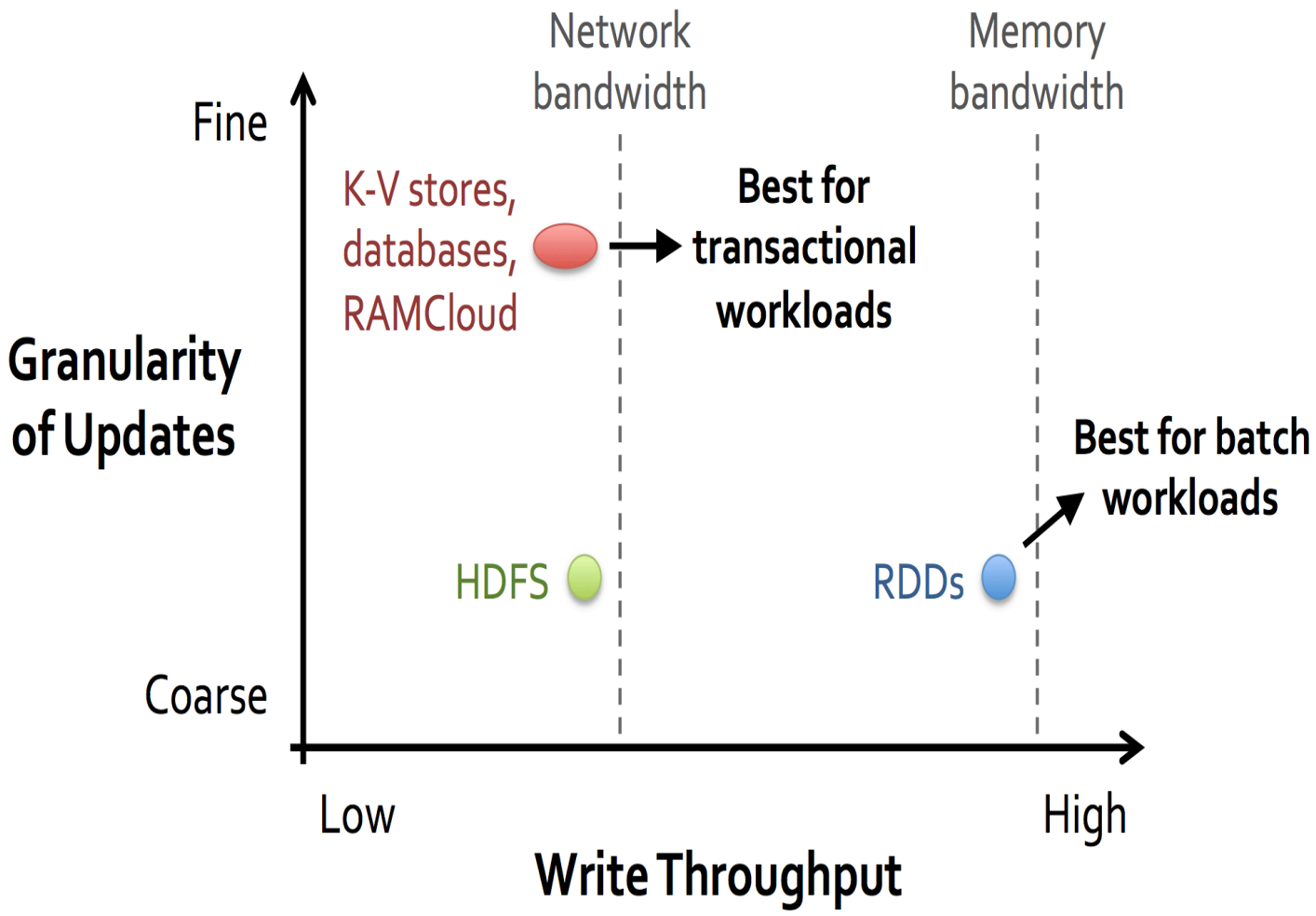
- Once the data is generated deterministically, and immutable
  - Recover the data from “recompute”
  - Only need to log the sequences to generate rdd. small cost if there’s no fault happens

```
messages = textFile(...).filter(_.contains(“error”))
                        .map(_.split(‘\t’)(2))
```

Big data systems



Spark Design Philosophy



SPARK PROGRAMMING INTERFACE

Programming API

- Based on Scala
  - A Java-like functional language
  - Can use spark in the Scala concole
  - Now support Java and Python
- RDD operations
  - Transformation： generate new RDDs from existing RDDs
    - map, filter, groupBy, sort,distinct, sample…
  - Action: return a value from RDD
    - reduce， count, collect, first, foreach…， ,

Scala Expressions

- Expressions and values
  - var a = 1 //Mutable
  - val b = 2 // Immutable

```
scala> var a = 1
a: Int = 1

scala> a = 2
a: Int = 2

scala> val b = 2
b: Int = 2

scala> b = 1
<console>:8: error: reassignment to val
       b = 1
         ^

scala>
```

Scala - Functions

- def helloWord()={ println(“Hello World”)}
- def addOne(x:Int) = {

  x+1

}
- def addOne(x:Int) = x+1
- val addOne = (x:Int) => x + 1

Scala Functions

- Pass a function as a parameter

```
scala> def squareAddOne(f:Int=>Int, arg:Int) = f(arg) + 1
squareAddOne: (f: Int => Int, arg: Int)Int

scala> def square(x:Int) = x*x
square: (x: Int)Int

scala> square(2)
res0: Int = 4

scala> squareAddOne(square, 2)
res1: Int = 5
```

Spark Operators

Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Actions (return a result to driver program)	collect reduce count save lookupKey	

http://spark.apache.org/docs/latest/programming-guide.html#rdd-operations

How to create RDDs

- From Collection

  val a = sc.parallelize(1 to 9, 3)

- from File

  val b = sc.textFile(“/path/to/file”)

- from other RDDs

  val c = b.map(line => line.split(“,”))

Map

- Map each element in source RDD to ONE element in destination RDD

```
scala> val a = sc.parallelize(1 to 9, 3)
scala> val b = a.map(x => x*2)
scala> a.collect
res10: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)
scala> b.collect

res11: Array[Int] = Array(2, 4, 6, 8, 10, 12, 14, 16, 18)
```

flatMap

- One element in source rdd will be transformed to multiple elements(0..n) in destination rdd

```
scala> val a = sc.parallelize(1 to 4, 2)
scala> val b = a.flatMap(x => 1 to x)
scala> b.collect

res0: Array[Int] = Array(1, 1, 2, 1, 2, 3, 1, 2, 3, 4)
```

reduce

- Defines a function to combine two elements in a collection to one elements, and keep on applying it to elements in the collection

```
scala> val c = sc.parallelize(1 to 10)  scala> c.reduce((x, y) => x + y)

res4: Int = 55
```

reduceByKey

- Reduce for key-value list
- Values with the same keys are reduced, and form a new key-value

```
scala> val a = sc.parallelize(List((1,2),(3,4),(3,6)))   scala>
a.reduceByKey((x,y) => x + y).collect

res7: Array[(Int, Int)] = Array((1,2), (3,10))
```

Output

- saveAsTextFile/saveAsSequenceFile/saveAsObjectFile

```
scala> val a = sc.parallelize(List((1,2),(3,4),(3,6)))
scala> a.reduceByKey((x,y) => x + y).saveAsTextFile(“test”)
```

Join

(k,v1) join (k, v2) ➔ (k, (v1, v2))

```
scala> val kv1=sc.parallelize(List((“A”,1),(“B”,2),(“C”,3)))
scala> val kv3=sc.parallelize(List((“A”,4), (“A”,5),(“B”,3),(”D”,30)))
scala> kv1.join(kv3).collect
```

Array[(String, (Int, Int))] = Array((A,(1,4)), (A,(1,5)),(B,(2,3)))

```
scala> kv1.union(kv3).collect
```

Array((A,1), (B,2), (C,3), (A,4), (A,5), (B,3), (D,30))

Use of _

- val a = (1 to 9)
- Val b = a.map(x => x*2)

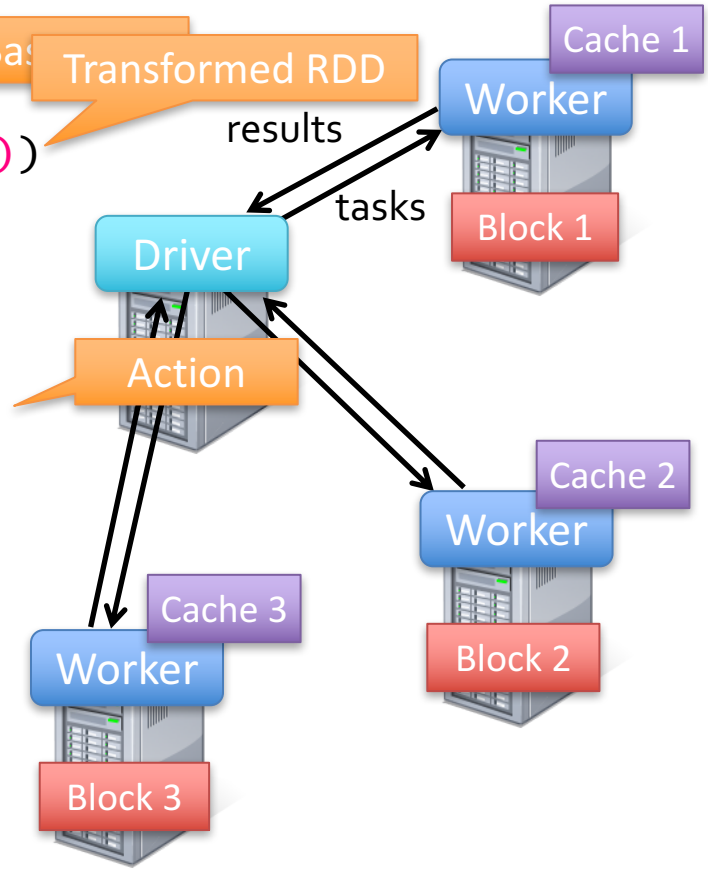
  is the same as val b = a.map(_ * 2)

- val a = sc.parallelize(List((1,2),(3,4),(3,6))) a.reduceByKey((x,y) => x + y).collect is the same as a.reduceByKey(_+_).collect

Log Mining

Loading data from memory, than query the data interactively

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
. . .
```



Performance: 1TB data with in 5-7 seconds
Disks: 170s

Logistic Regression

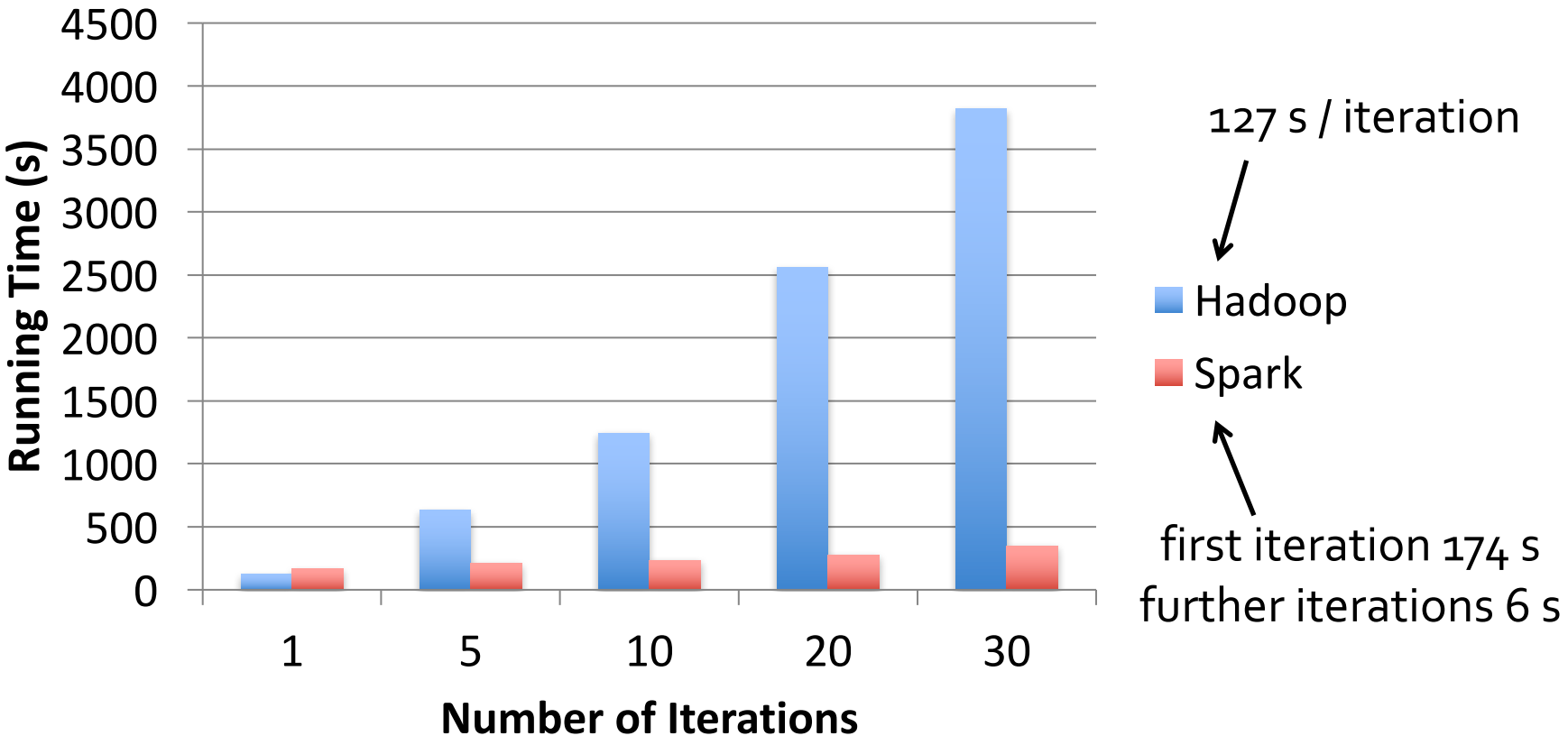
```
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}

println("Final w: " + w)
```

Load data in memory once

Initial parameter vector

Repeated MapReduce steps to do gradient descent

Logistic Regression Performance



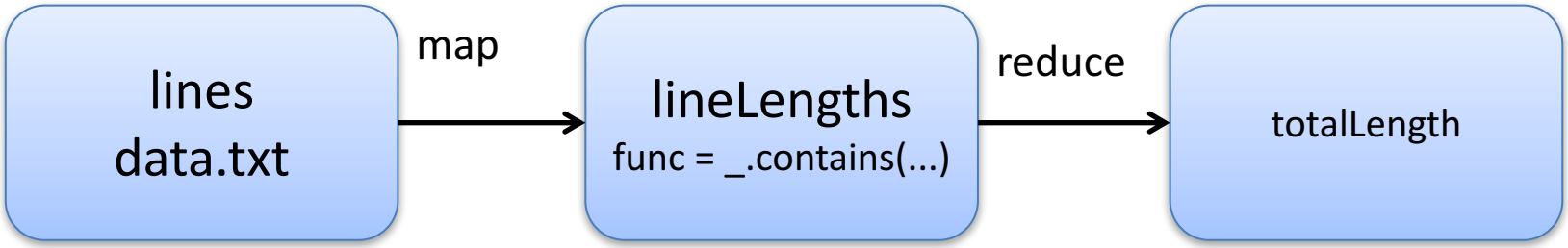
Example： WordCount

```
val spark = new SparkContext(master, appName, [sparkHome], [jars])
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

SPARK IMPLEMENTATION

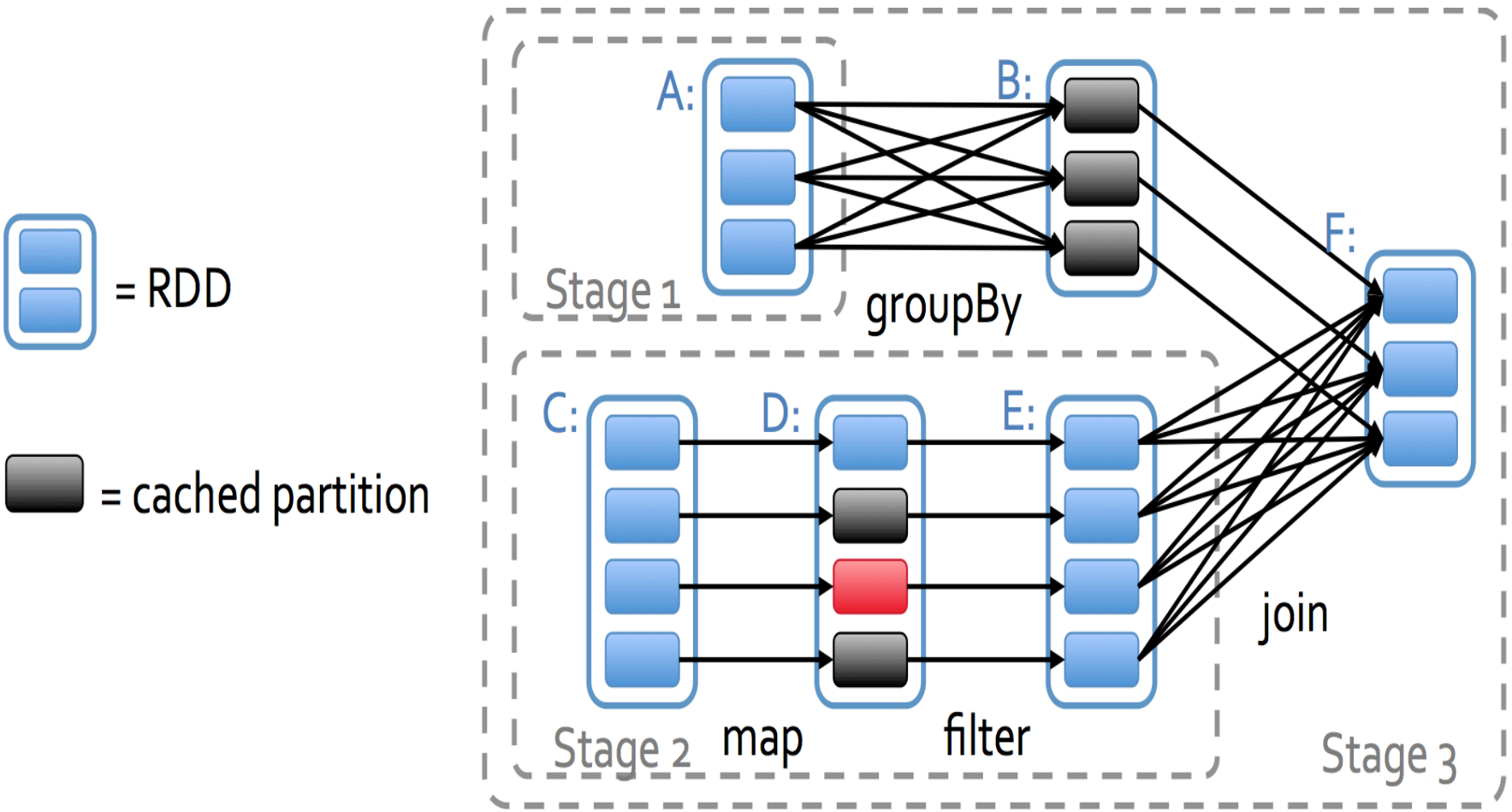
Lazy Evaluation

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```



- Transformations do not trigger computation
- The last reduce will trigger computation and generate the DAG

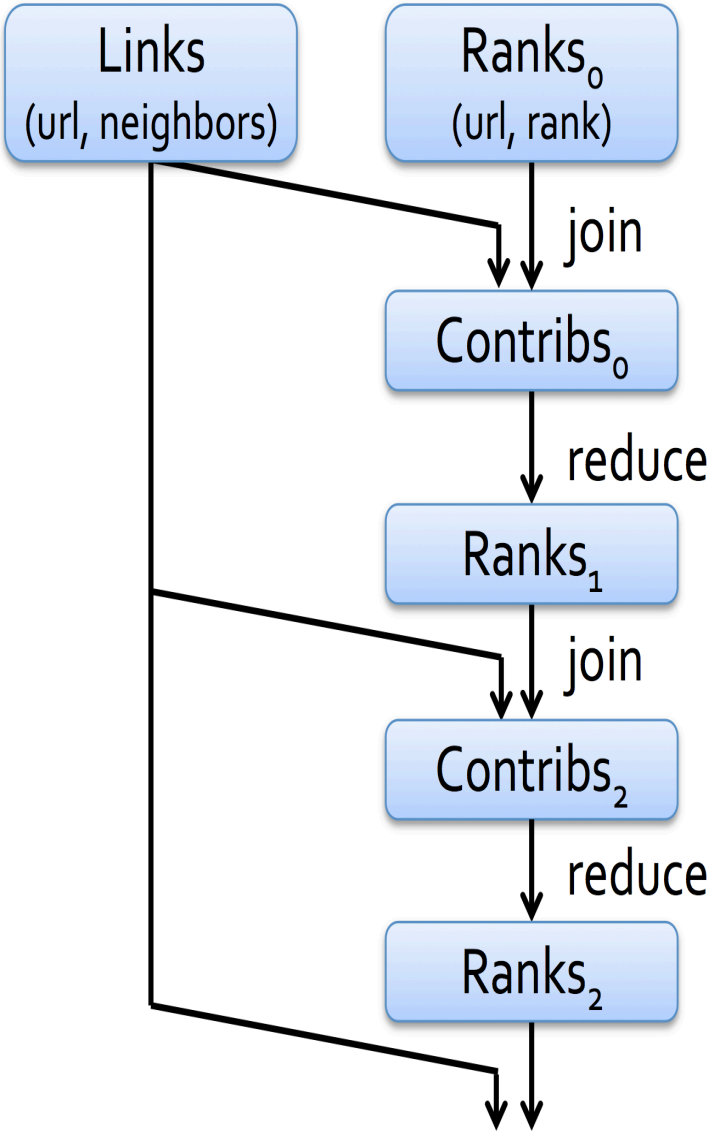
Complex DAGs



Data Partition

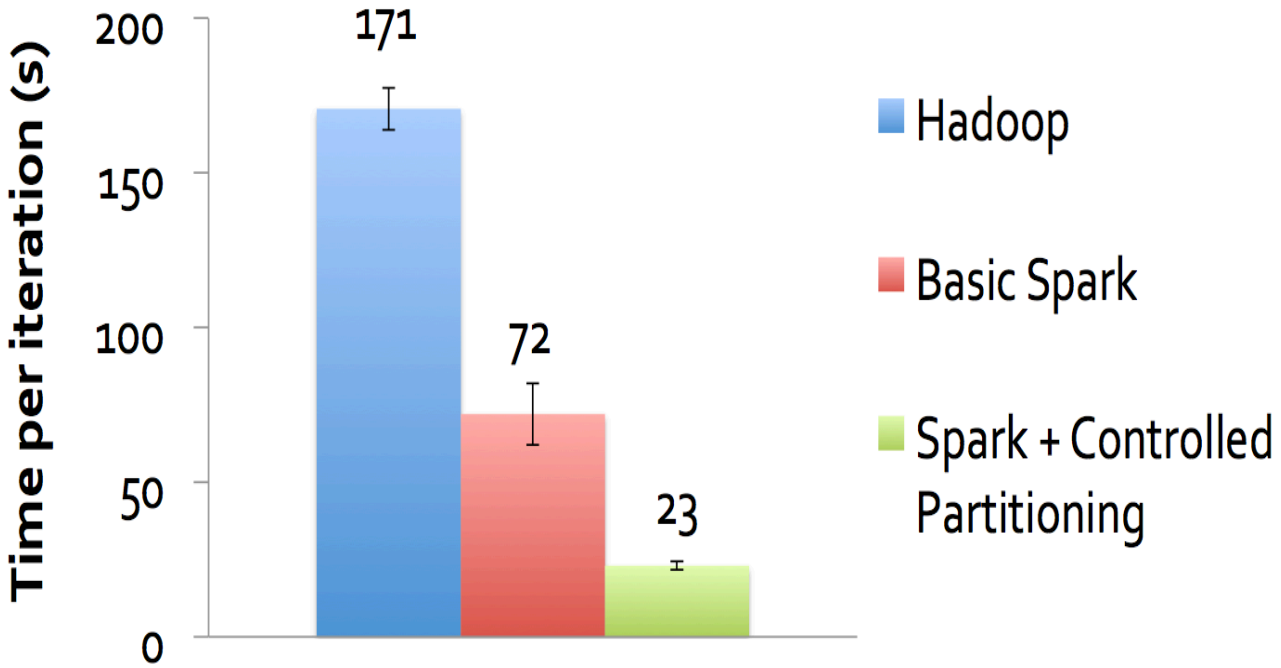
```
links = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
  ranks = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }.reduceByKey(_ + _)
}
```

Optimization on Data Partition



- Links and Ranks joins multiple times
- Partition the links and ranks with sample url together would avoid unnecessary communications
- partitionBy() are used to define how to partition data

PageRank Performance



Persist and Cache

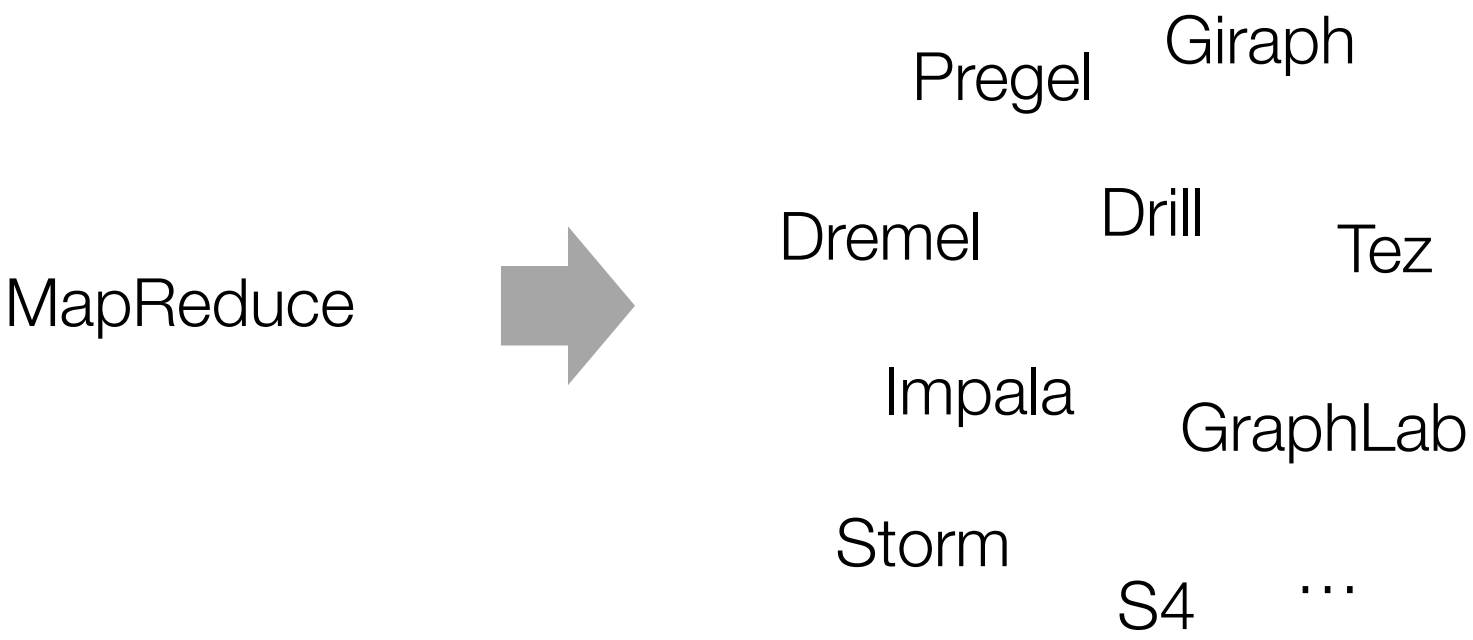
- Cache the message, means the data would be reused in the future and try to maintain it in the memory
  - Avoid recalculation
  - Cache is a special case for Persist

```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(_.startsWith(“ERROR”))
messages = errors.map(_.split(‘\t’)(2))
cachedMsgs = messages.cache()


cachedMsgs.filter(_.contains(“bar”)).count
```

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.

SPARK ECOSYSTEM

Existing Big data system

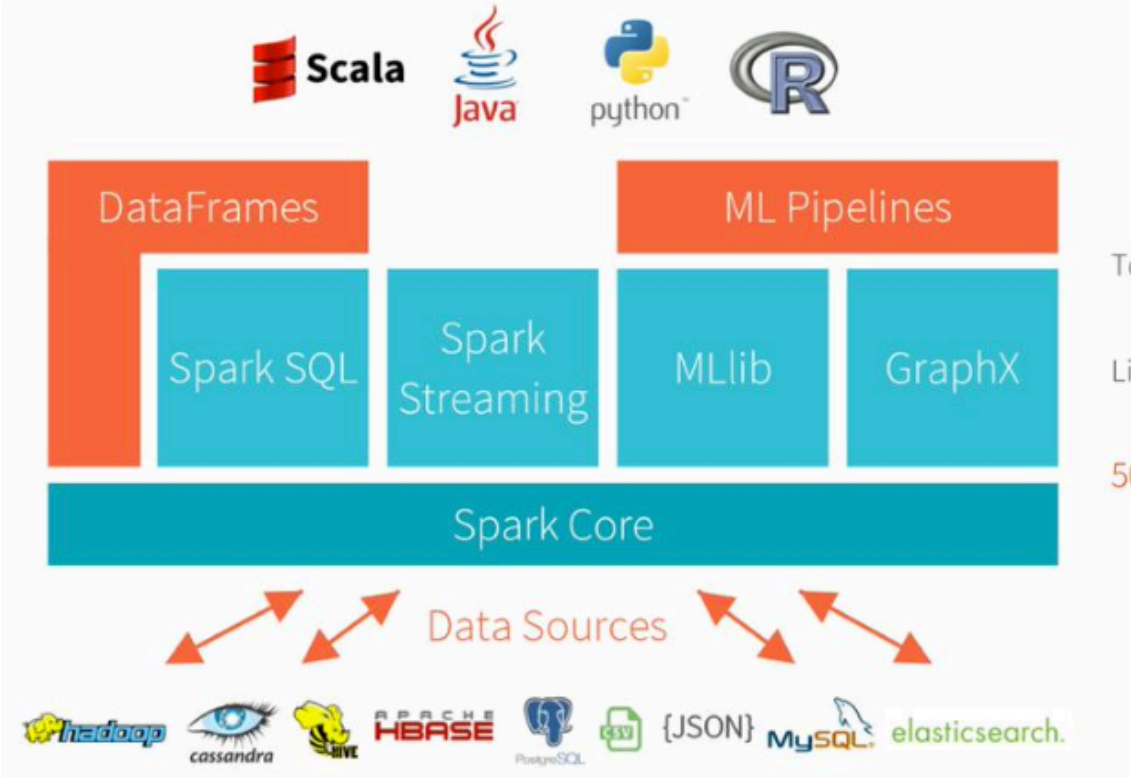


General systems for Batch processing

Special system (Iterative, interactive and streaming)

The methodology of Spark

- Generalized MapReduce
  - Sharing the data and task DAGs
- Unified Programming Framework



Total contributors: 150 → 500

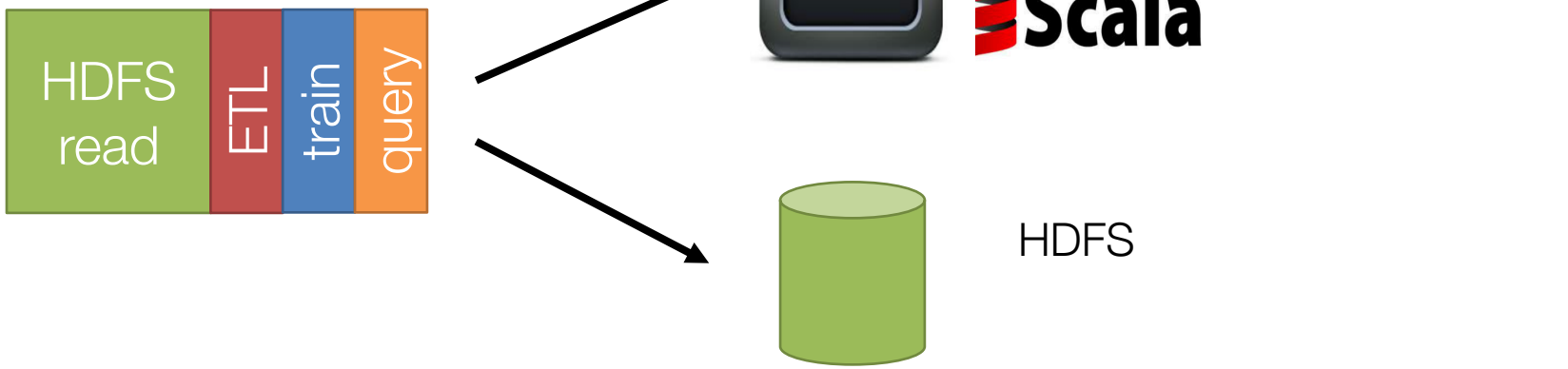
Lines of code: 190K → 370K

500+ active production deployments

Benefit of Using Spark

- With different platforms

Spark:



Limitations of Spark