Gestión de datos

Una perspectiva sobre Big Data

Lorena Etcheverry (lorenae@fing.edu.uy)
Instituto de Computación, FING, UdelaR

Un poco de historia

Y en el comienzo, fueron los archivos …

Cada programa resolvía la gestión de datos.

La concurrencia y consistencia se resolvían en cada caso.

Optimización, *caching*, *pre-fetching* también.

No había separación entre
la representación (estructura de datos)
y los datos.

Almacenamiento: en cintas de acceso secuencial

El disco magnético aparece en los 50s!

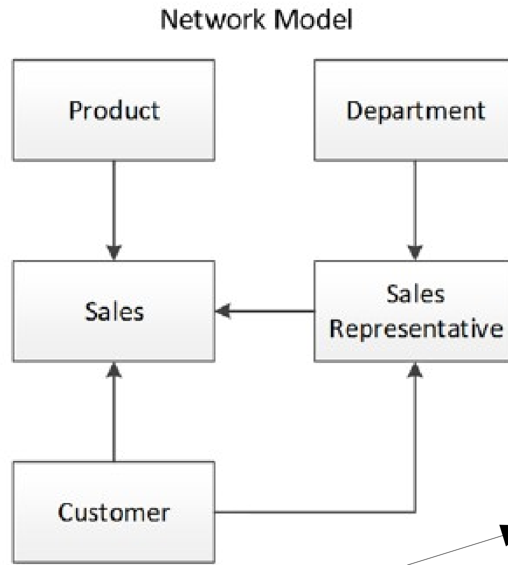
Los primeros DBMS

Programas dedicados a la gestión de datos:
desacoplar de la lógica de la aplicación para permitir el **reuso** de esta lógica!

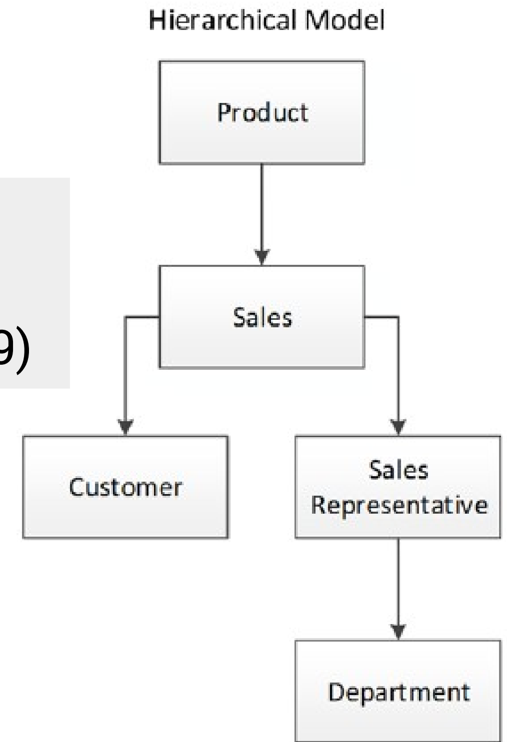
Sistemas ***mainframe***

Naturaleza **navegacional**: modelos jerárquico y de red

Modelo de Red
(CODASYL, 1969)



Modelo Jerárquico
(IBM IMS, 1969)



1960

1970

IBM System/360
(IBM 1964) [1]

[1]Image By Sandstein - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=16305441

Algunas desventajas de los primeros DBMS

Rigidez tanto en las estructuras de datos como en las consultas que podían resolver.

Enfocadas en el registro: operaciones **CRUD** (Create, Read, Update, Delete)

Hacer **consultas de análisis** demandaba hacer programas complejos.

No había **separación** entre la **representación lógica** y el almacenamiento físico

Information Retrieval

A Relational Model of Data for Large Shared Data Banks

E. F. Codd
IBM Research Laboratory, San Jose, California

(Codd, 1970-1972)

The Entity-Relationship Model—Toward a Unified View of Data

PETER PIN-SHAN CHEN
Massachusetts Institute of Technology

(Chen, 1976)

Arquitecturas cliente-servidor (90s)

OODBMS (90s)

OLAP (90s)

1960

1970

1980

1990

2000

2010

INGRES (1974, U. Berckley)

System R SEQUEL (1977, IBM)

IBM PC (1981) [1]

SQL-86 (ANSI, 1986)

Comienzo de la Internet comercial (1995)

[1] CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=51833

Algunos aciertos de los RDBMS

Basados en un **modelo formal**: el modelo relacional

Tuplas, relaciones, restricciones

Un álgebra de operaciones

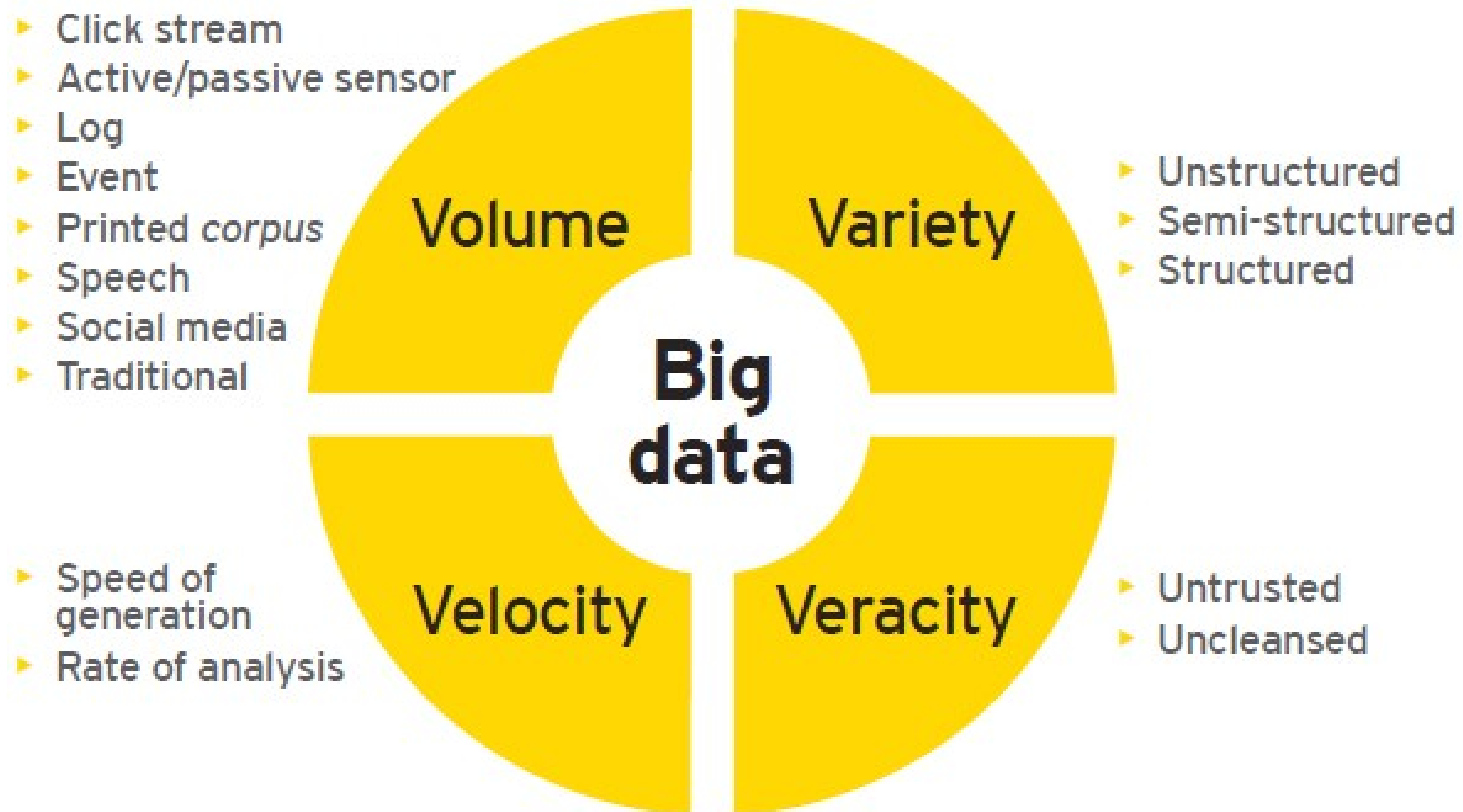
Soporte para **transacciones** (ACID)

El **lenguaje de consultas** SQL

Los RDBMS fueron LA herramienta de gestión de datos durante casi 30 años

Big Data

¿de qué hablamos cuando hablamos de Big Data?



Monga, Yuvraj & Mittal, Mohit. (2017). Study and analytical perspective on big data. International Journal of Computational Systems Engineering. 3. 193. 10.1504/IJCSYSE.2017.089190.

Batch processing vs Stream processing

Procesamiento *batch*:

- Transformation, Join and Aggregation
- (historical) Analytics, Prediction and Modeling

Procesamiento de *streams*:

- Transformation, Join and (temporal) Aggregation
- (real-time) Analytics, Inferencing Prediction models (ej: *sentiment analysis*)

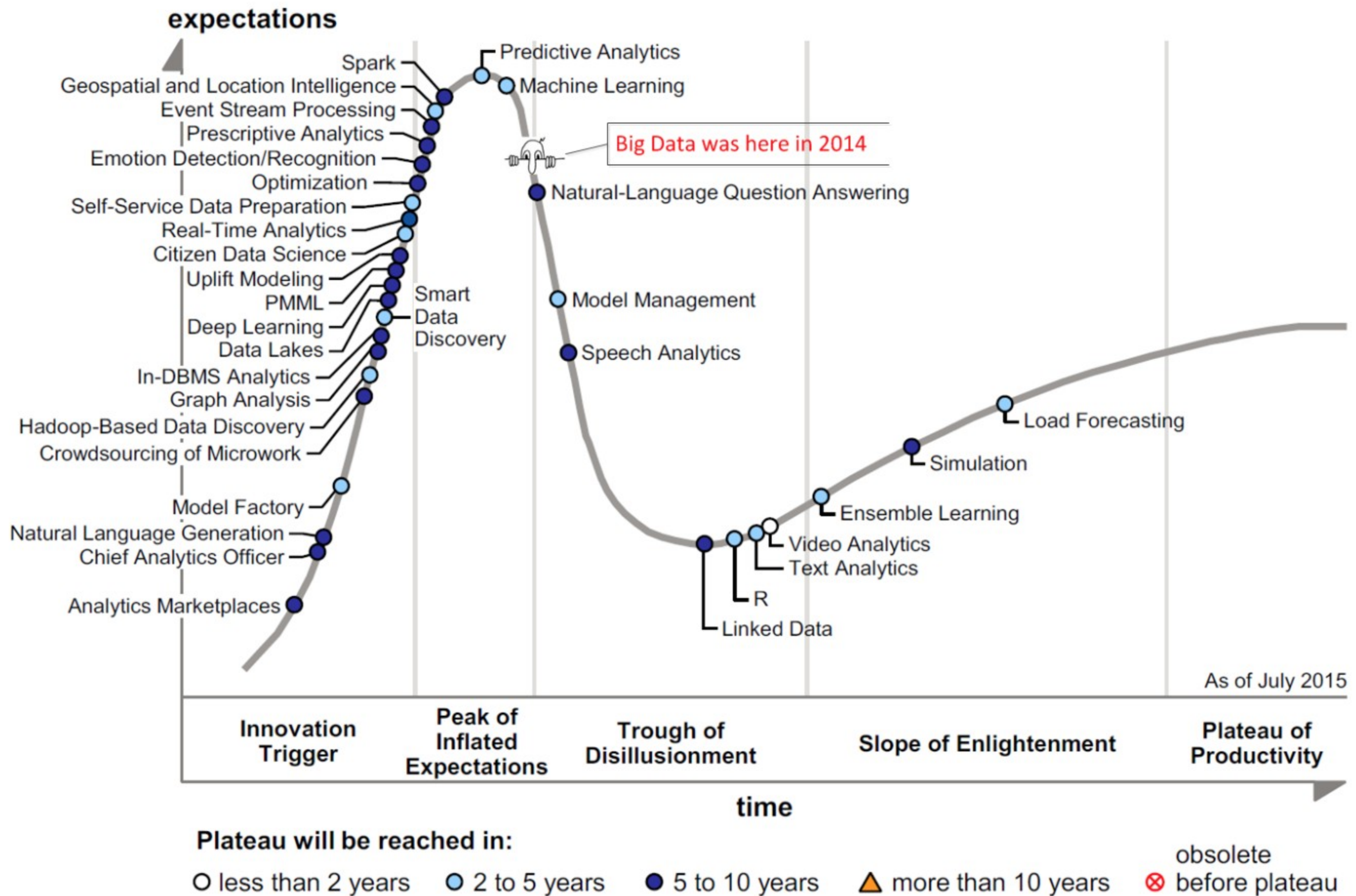
Data Management

- Aquisition & recording
- Extraction, cleaning & annotation
- Integration, aggregation & representation

Analytics

- Modelling & analysis
- Interpretation

Beyond the hype: Big data concepts, methods, and analytics. A. Gandomi | M.Haider , IJIM 2015



https://www.datasciencecentral.com/profiles/blogs/big-data-falls-off-the-hype-cycle

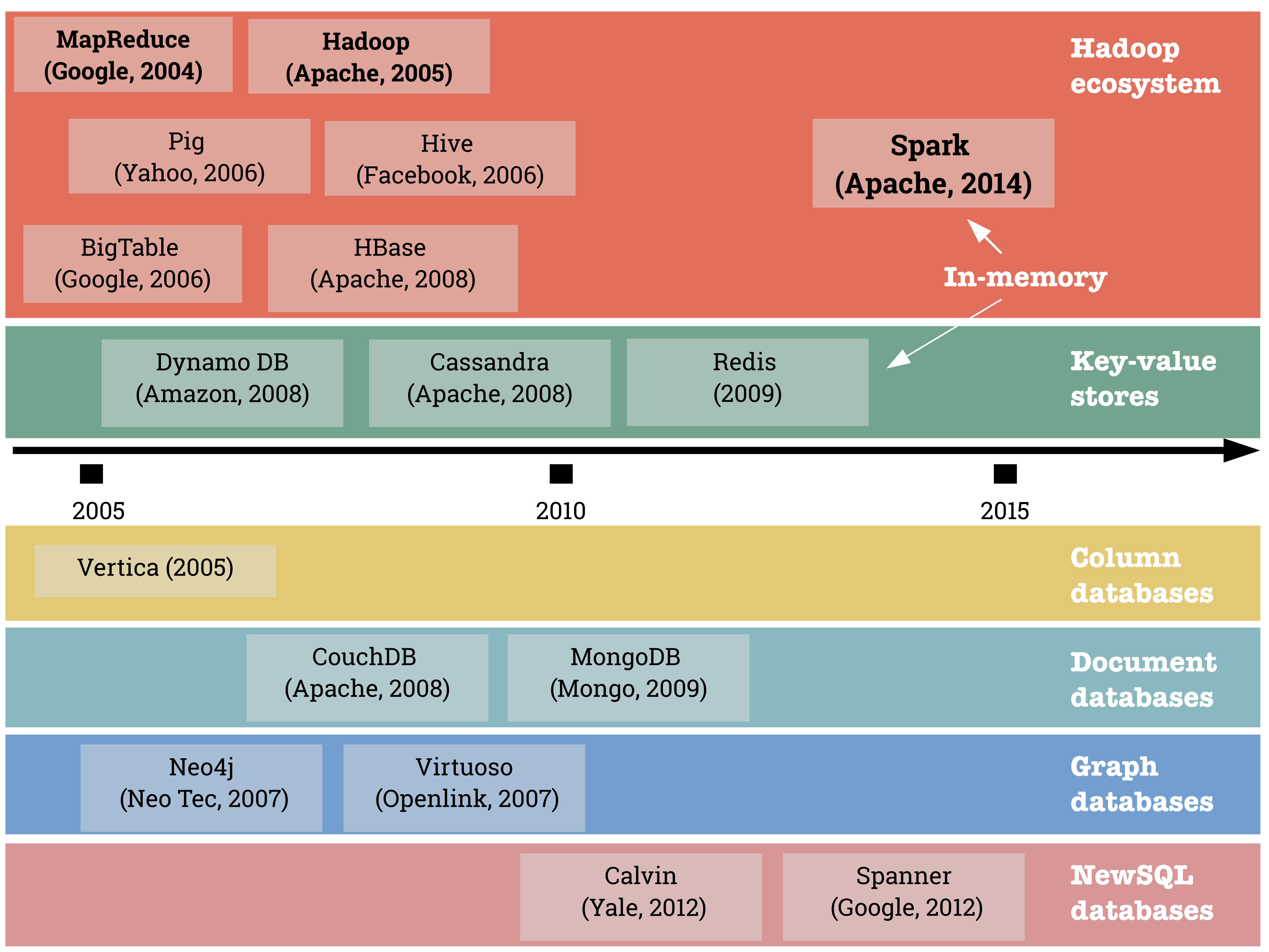
Big Data, Disruption and the 800 Pound Gorilla in the Corner

Michael Stonebraker

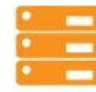
0:00 / 41:17

Keynote - Big Data, Technological Disruption, And The 800-Pound Gorilla In The Corner

Herramientas para Big Data


MapReduce
(Google, 2004)
Hadoop
(Apache, 2005)
Hadoop ecosystem
Pig
(Yahoo, 2006)
Hive
(Facebook, 2006)
Spark
(Apache, 2014)
BigTable
(Google, 2006)
HBase
(Apache, 2008)
In-memory
Dynamo DB
(Amazon, 2008)
Cassandra
(Apache, 2008)
Redis
(2009)
Key-value stores
2005
2010
2015
Vertica (2005)
Column databases
CouchDB
(Apache, 2008)
MongoDB
(Mongo, 2009)
Document databases
Neo4j
(Neo Tec, 2007)
Virtuoso
(Openlink, 2007)
Graph databases
Calvin
(Yale, 2012)
Spanner
(Google, 2012)
NewSQL databases

CLOUD vs. ON-PREMISE



OVERVIEW

Cloud		On-premise
• Low-cost up front • Predictable cost over time • No hardware/server investments	👍	• Reduced initial price
• May end up spending more over the course of the system's life cycle	👎	• Upfront investment can be seen as riskier • Have to pay for hardware and servers • Responsible for IT maintenance and setup

SETUP

Cloud		On-premise
• Quick and easy (done by your vendor) • Adding new users and instances is easy • Remote access requires no work on your part	👍	• Setup is done by you, giving you greater control over the process
	👎	• Implementation may take much longer • Responsible for setting up remote access • Adding users and instances may be costly

CUSTOMIZATION

Cloud		On-premise
• Greater consistency and stability • More vendor support for customizations	👍	• Direct database access is possible, enabling complex customizations
• Direct database access is not allowed for security reasons, which may limit complex customization	👎	• Bespoke integrations may break when the vendor updates the software

MAINTENANCE

Cloud		On-premise
• Server and hardware taken care of by vendor • Updates, patches and fixes are installed automatically and regularly	👍	
	👎	• Perform updates, patches and fixes yourself • Maintain supporting servers, hardware, resources

SECURITY & DISASTER RECOVERY

Cloud		On-premise
• Security and backups taken care of by vendor	👍	• Security is in your hands; greater personal control over your data
• Security and backups taken care of by vendor – quality of data center will vary	👎	• Security is in your hands; you are responsible for data breaches and server failures • You carry cost of backups and server redundancy

Seguridad

Flexibilidad

Costos $

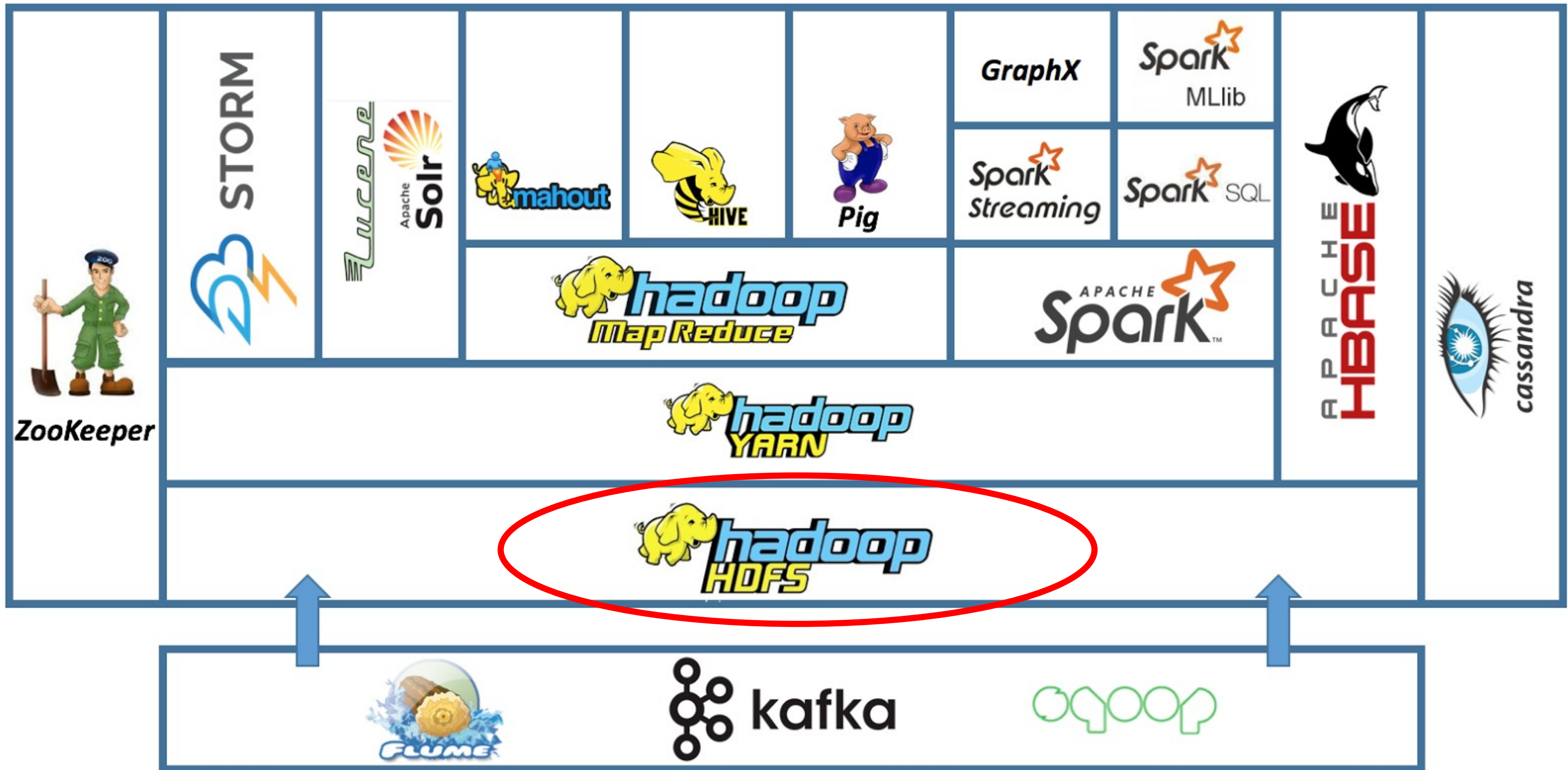
Control

Escalabilidad y agilidad

Apache Hadoop

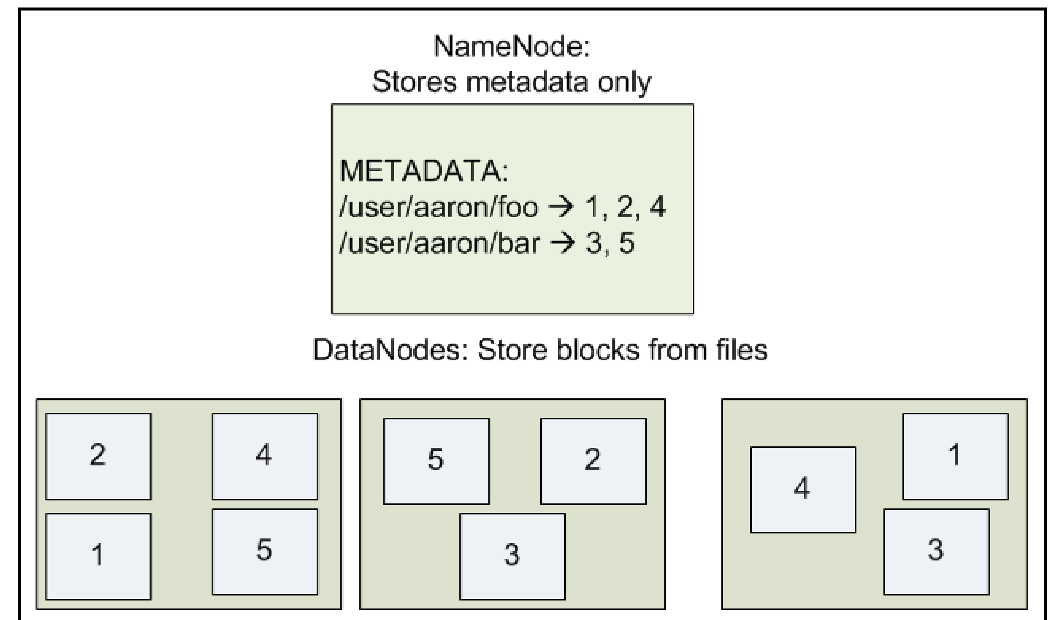
- Es un framework para almacenar y procesar grandes volúmenes de datos.
- Provee de un entorno de ejecución distribuído.
- Pensado para ejecutar en *commodity hardware.*
- Altamente escalable.
- Redundancia de datos.
- *Schema on read* en lugar de *Schema on write.*

El ecosistema Hadoop

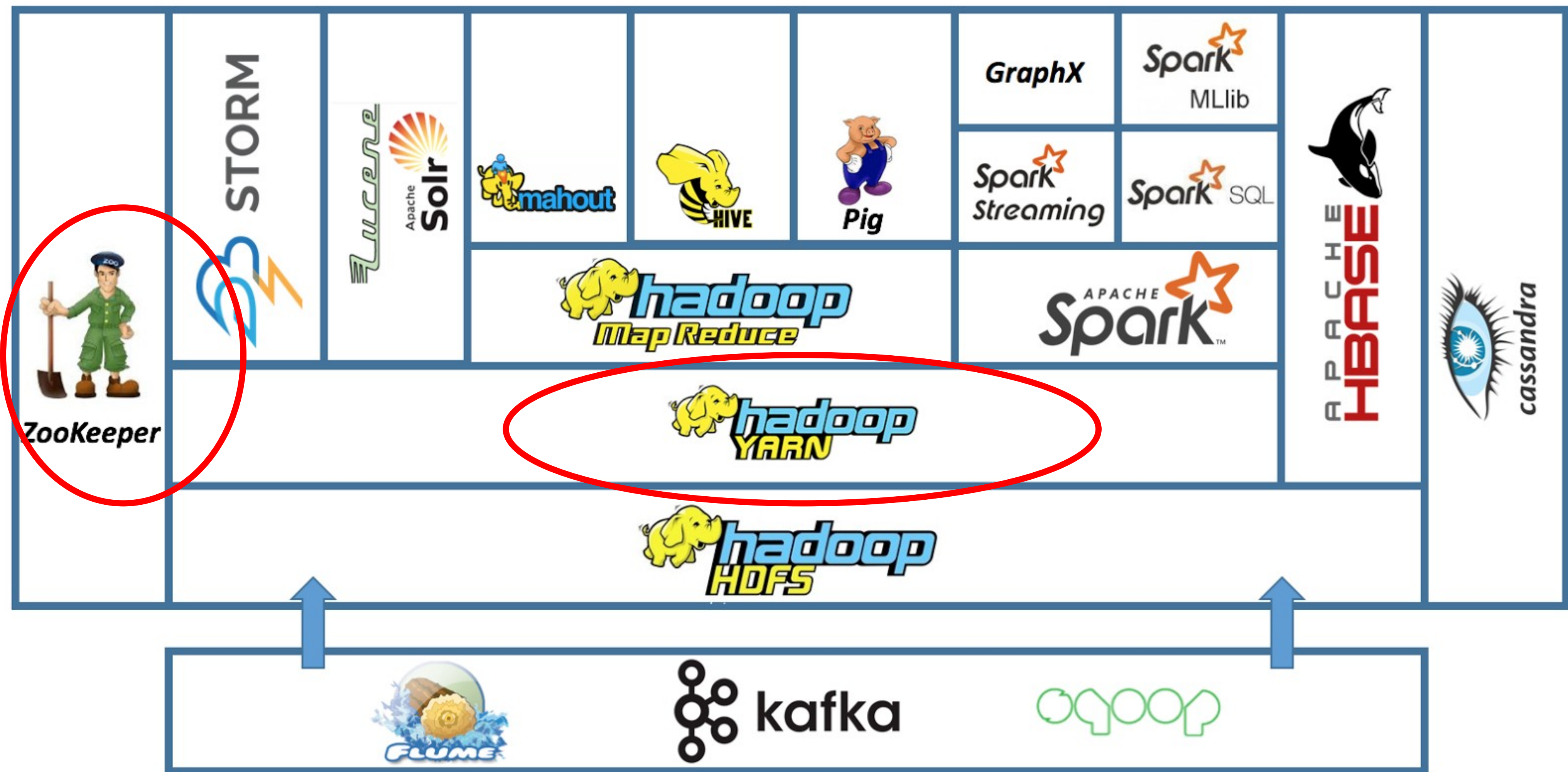


Hadoop Distributed Filesystem (HDFS)

- Sistema de archivos distribuído que almacena y replica datos en un cluster.
- HDFS es la implementación open-source del Google File System (GFS) (Jeff Dean and Sanjay Ghemawat, 2003)
- Dos tipos de nodos:
  - *namenode*
  - *datanodes*

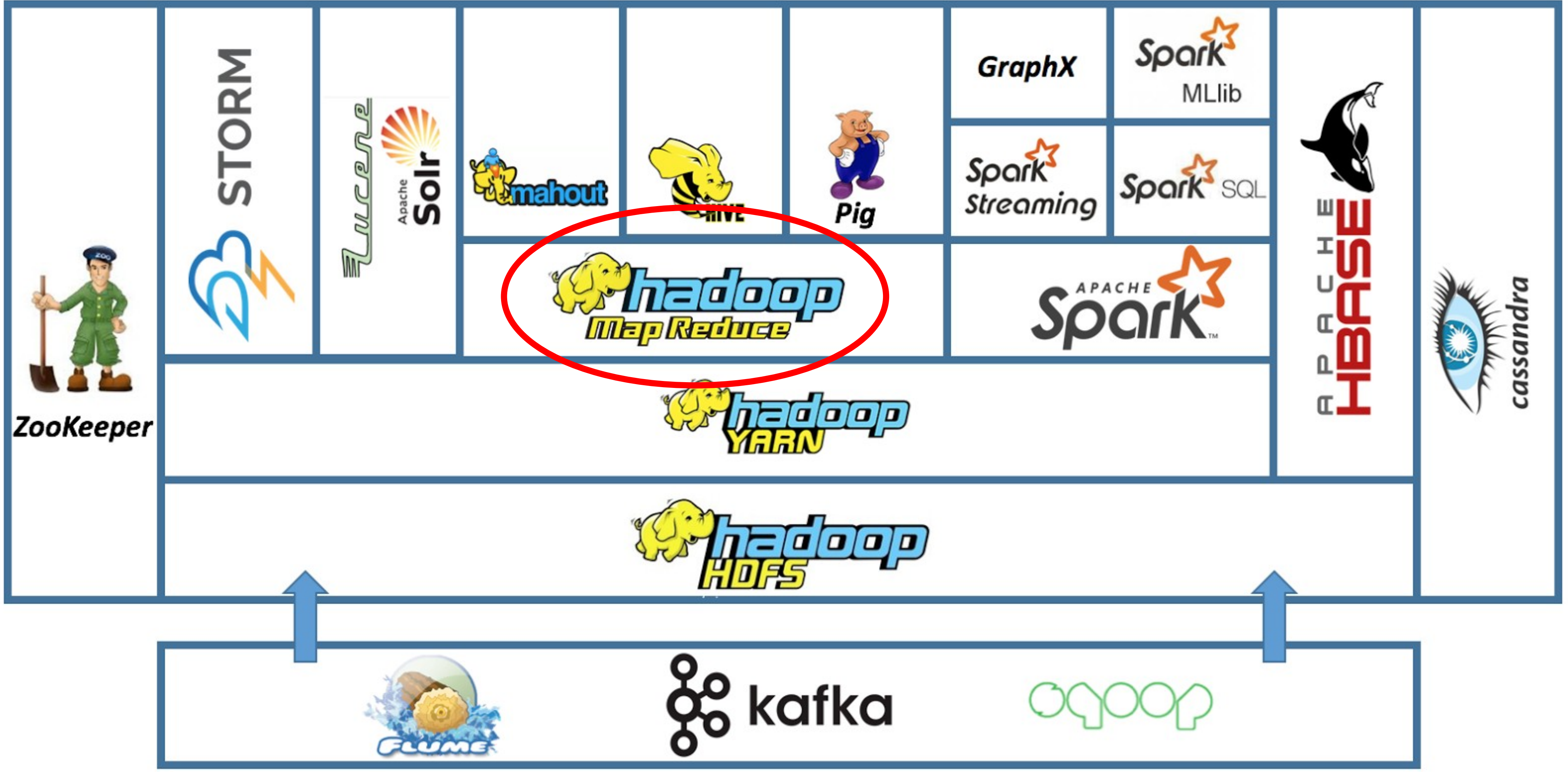


"Hadoop Tutorial from Yahoo!" by Yahoo! Inc.


ZooKeeper
STORM
Lucene
Apache Solr
mahout
HIVE
Pig
GraphX
Spark MLlib
Spark Streaming
Spark SQL
hadoop Map Reduce
APACHE Spark
hadoop YARN
APACHE HBASE
cassandra
hadoop HDFS
FLUME
kafka
sqoop

Apache YARN (*Yet Another Resource Negotiator*)

- Gestiona los recursos (CPUs, RAM, GPUs, etc.) y las tareas dentro del cluster
- Aparece en Hadoop 2 para desacoplar el motor MapReduce de la gestión de los recursos.
- Se encarga de la distribución de tareas sobre nodos, orquestación de la ejecución, recolección de logs, etc.


STORM
Lucene
Apache Solr
mahout
HIVE
Pig
GraphX
Spark MLlib
Spark Streaming
Spark SQL
hadoop Map Reduce
APACHE Spark
APACHE HBASE
cassandra
ZooKeeper
hadoop YARN
hadoop HDFS
FLUME
kafka
sqoop

MapReduce

- Paradigma de programación paralela
- Entorno de ejecución distribuido.
- El modelo básico tiene dos fases:
  - Fase **map**: generar parejas (clave,valor) a partir de la entrada
  - Fase **reduce**: agrupar las parejas a partir del valor de la clave y producir una salida

Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." OSDI (2004)

MapReduce (ii)

- En el modelo básico se deben programar dos funciones:
  - map: $(k_1,v_1) \rightarrow [(k_2,v_2)]$
  - reduce: $(k_2,[v_2]) \rightarrow [(k_3,v_3)]$

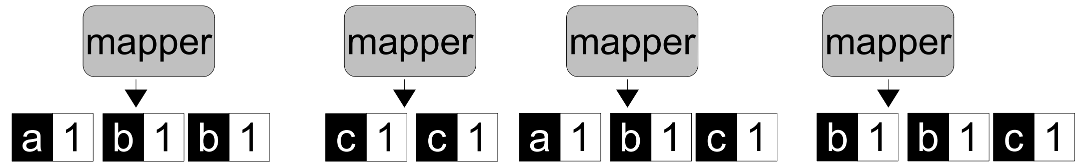
- Vamos a aplicarlo a un problema simple: conteo de palabras

Ejemplo: conteo de palabras

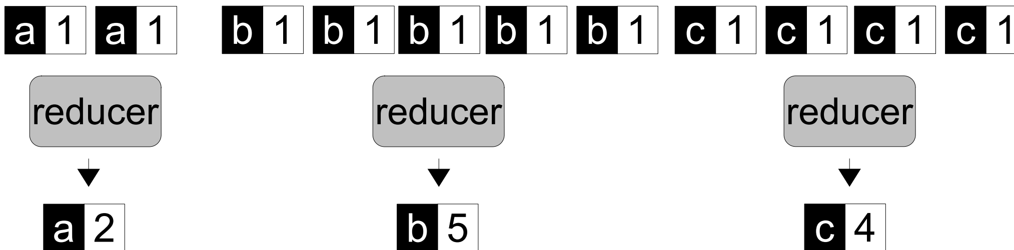
- Dado un texto contar la cantidad de ocurrencias de cada palabra

Split

A α B β C γ D δ E ε F ζ

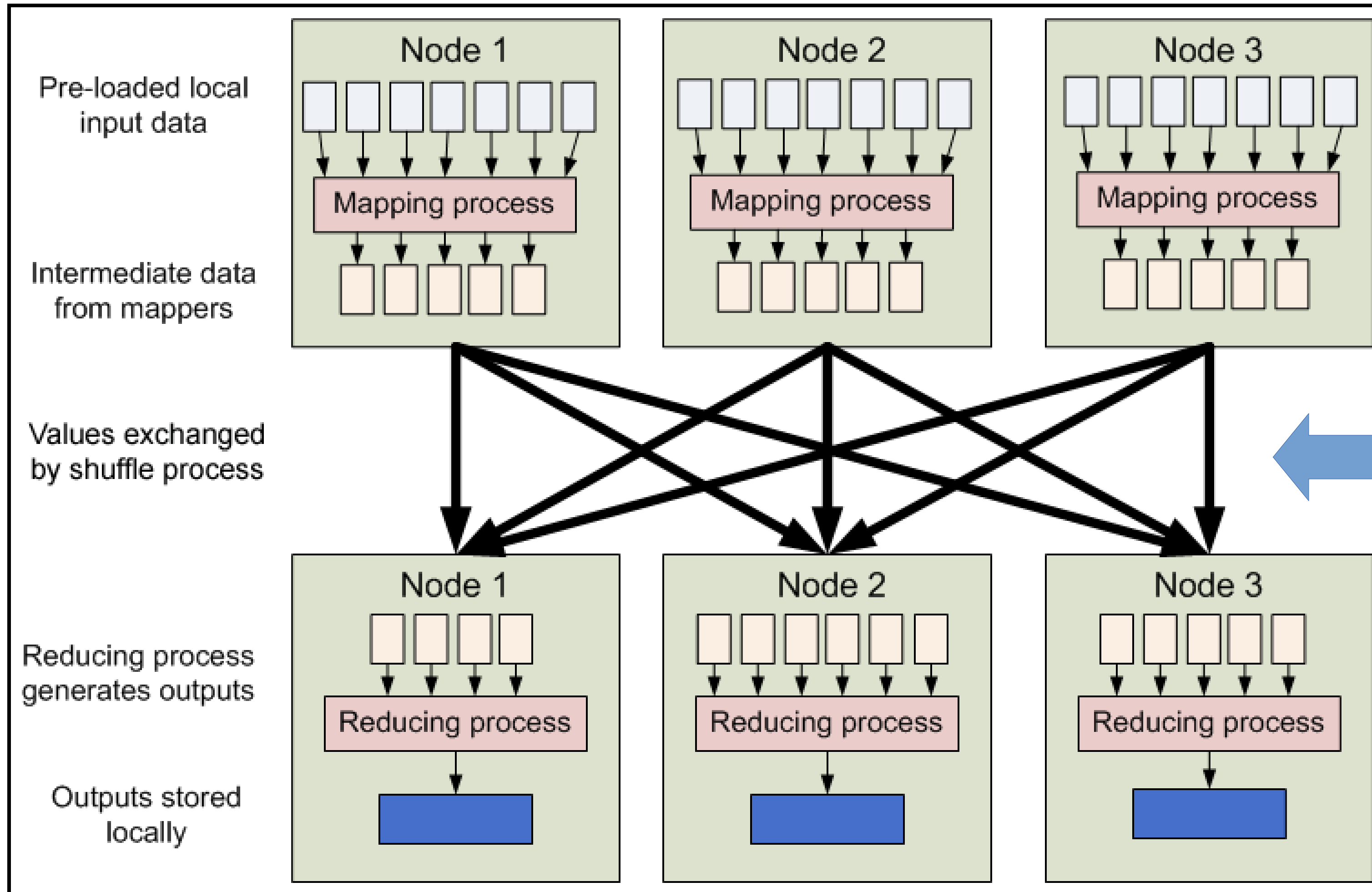


Shuffle & sort




```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)
```

```
1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count sum)
```



Reducir todo lo posible la cantidad de datos que pasan a la fase de *reduce*

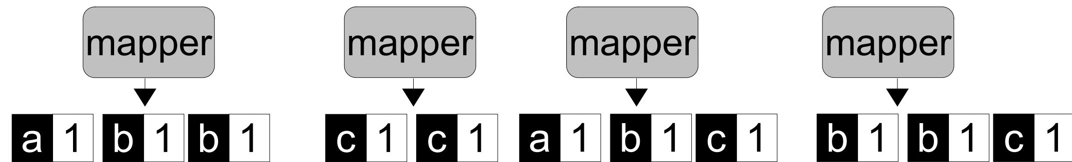
"Hadoop Tutorial from Yahoo!" by Yahoo! Inc.

Conteo de palabras : *combiners*

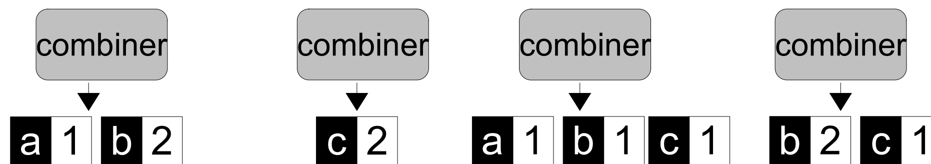
- Permiten computar resultados parciales a la salida de cada *mapper* (mini-reducers)

Split

A α B β C γ D δ E ε F ζ

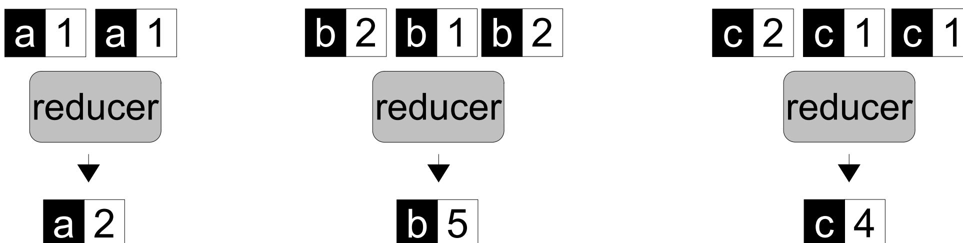



```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)
```




```
1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count sum)
```

Shuffle & sort




```
1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count sum)
```

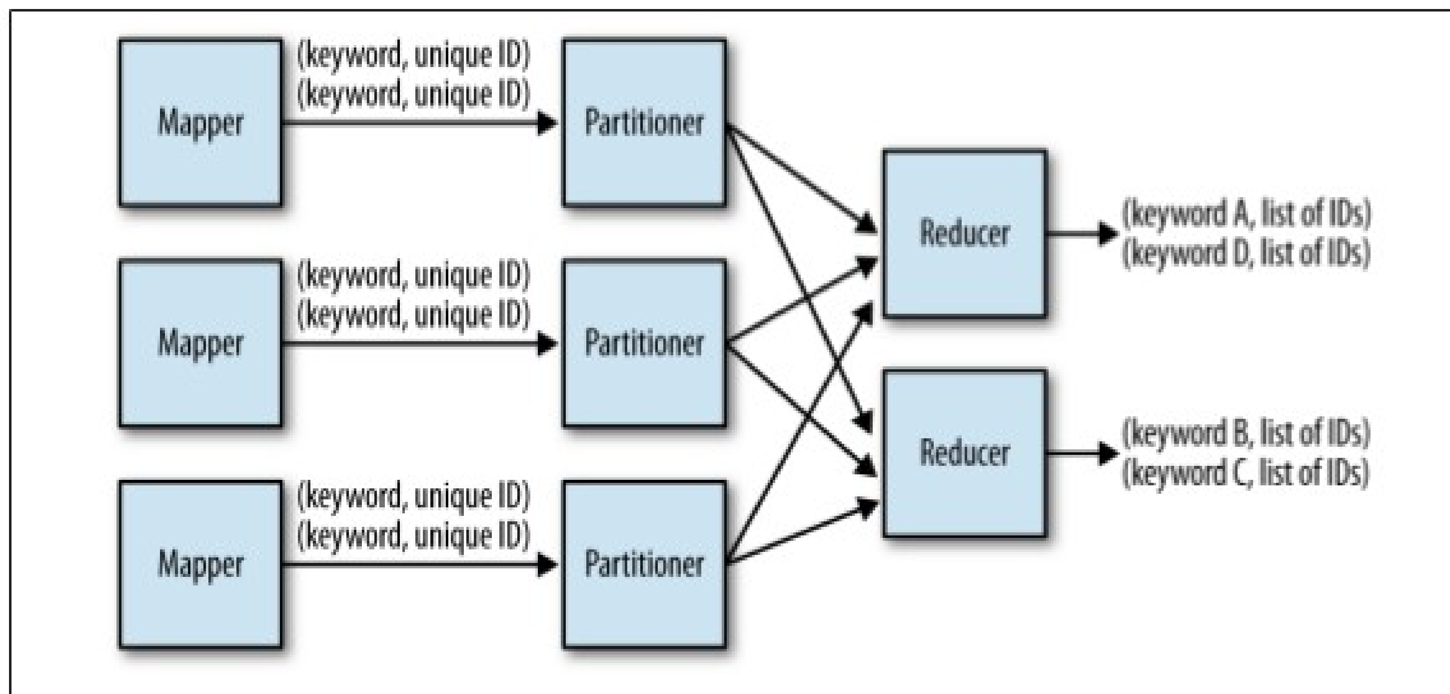
Patrones en MapReduce

- No puedo resolver cualquier problema con esta técnica
- Para algunos problemas hay patrones de diseño definidos:
  - Sumarización e indexado
  - Filtrado
  - Organización de datos (particionado, transformaciones)
  - Join

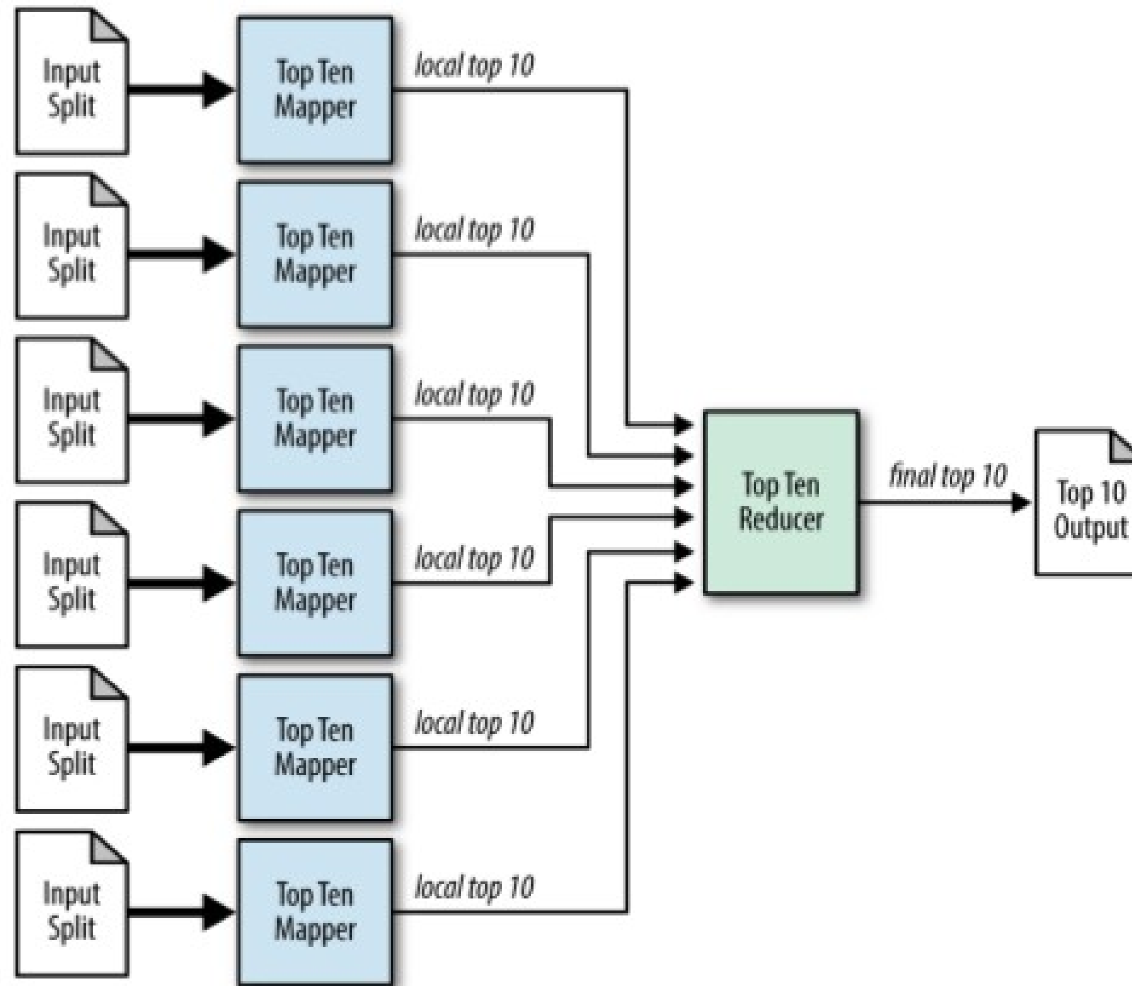
Donald Miner and Adam Shook. MapReduce Design Patterns Building Effective Algorithms and Analytics for Hadoop and Other Systems. O'Reilly Media,Inc., 2012.

Indices inversos

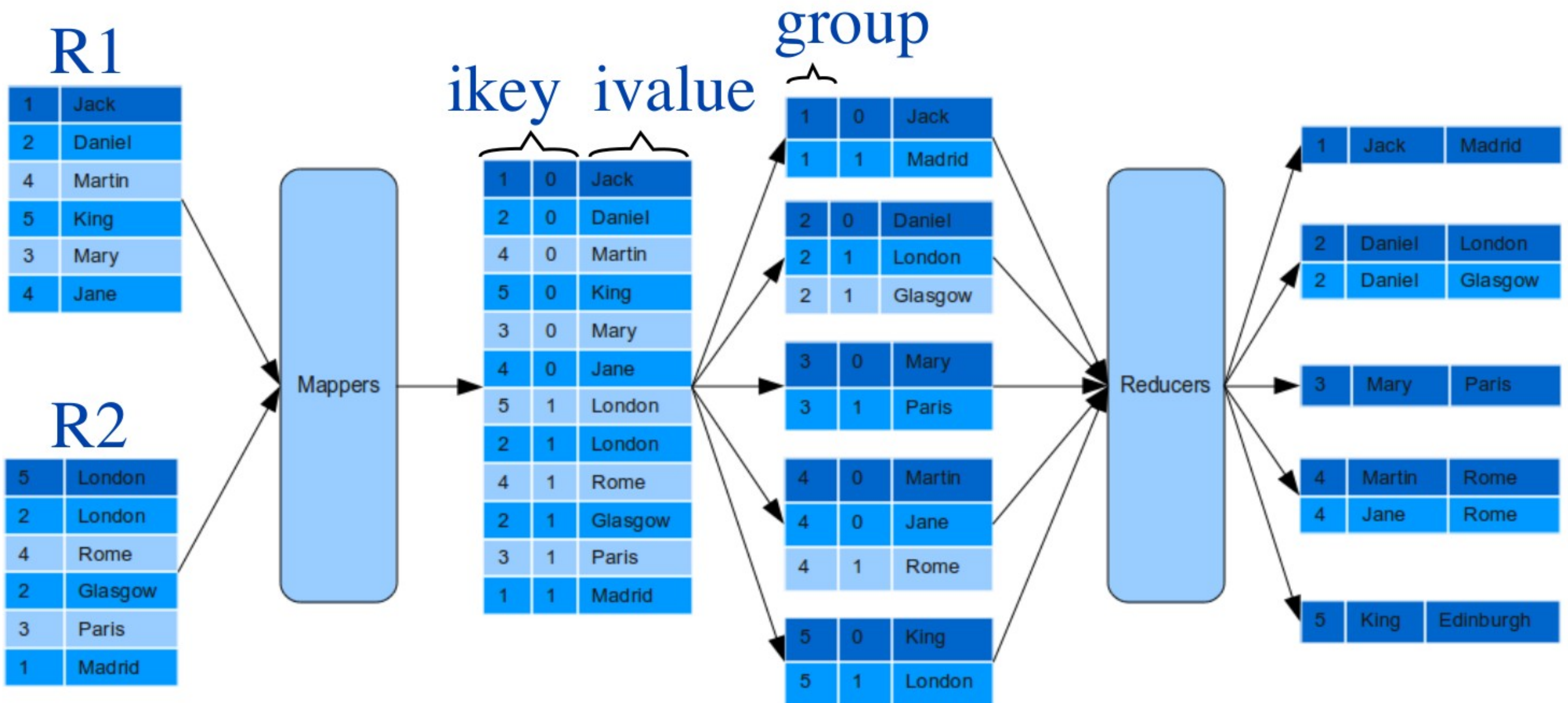
- Fue el caso de uso que dio origen a MapReduce en Google
- Construir una lista de URLs que contienen cierta palabra



Filtrado: top - k



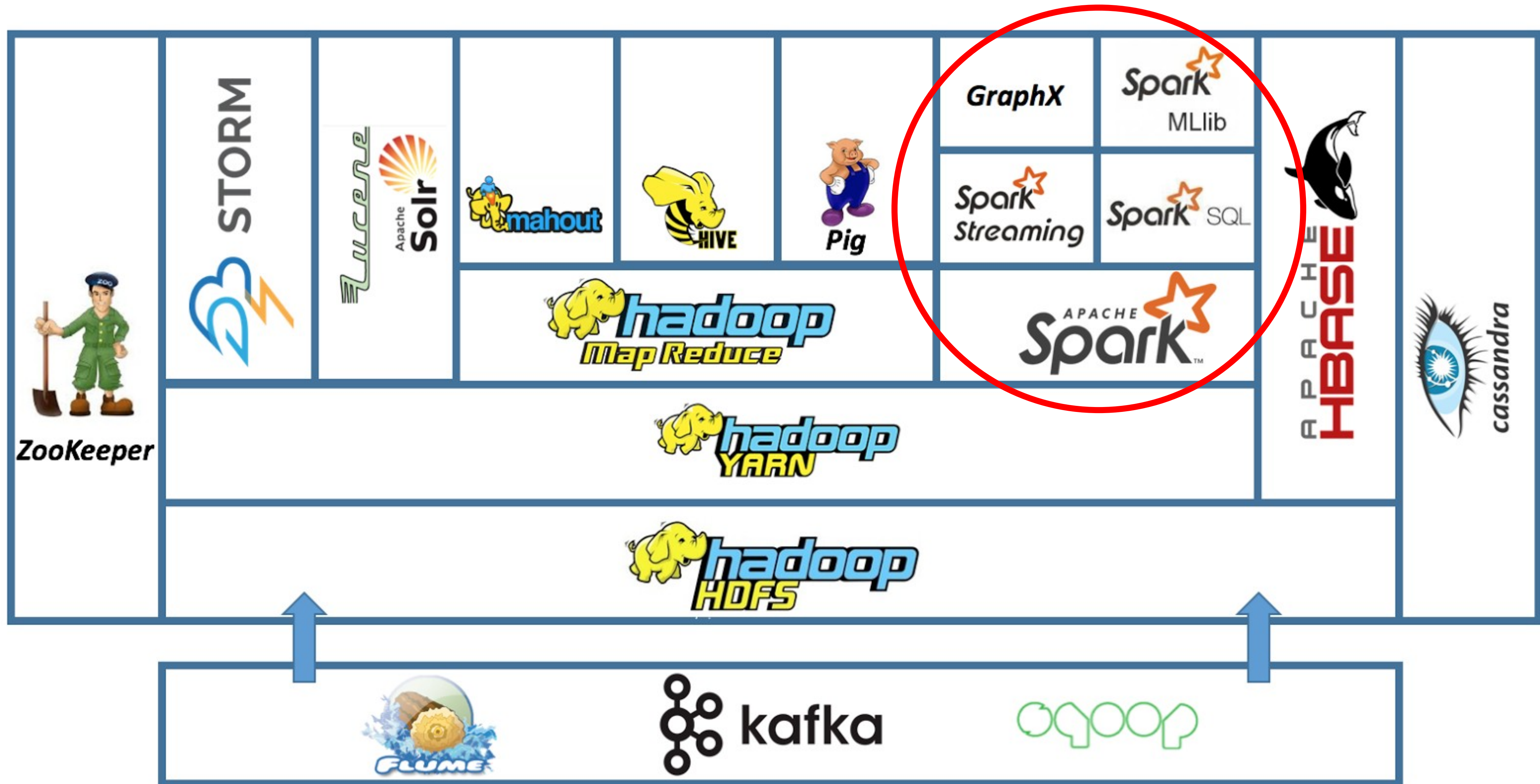
Patrones de Join (reduce side join)



Programar directamente sobre MapReduce no es sencillo :(

- Aparecen abstracciones
- Pig es un lenguaje de alto nivel que permite realizar consultas
  - B = ORDER A BY col4 DESC;
  - C = LIMIT B 10;
- Hive provee una abstracción sobre Hadoop.
  - Modelo de datos: tablas, vistas, etc
  - HiveQL como lenguaje de consultas

Spark: la "evolución" de MapReduce

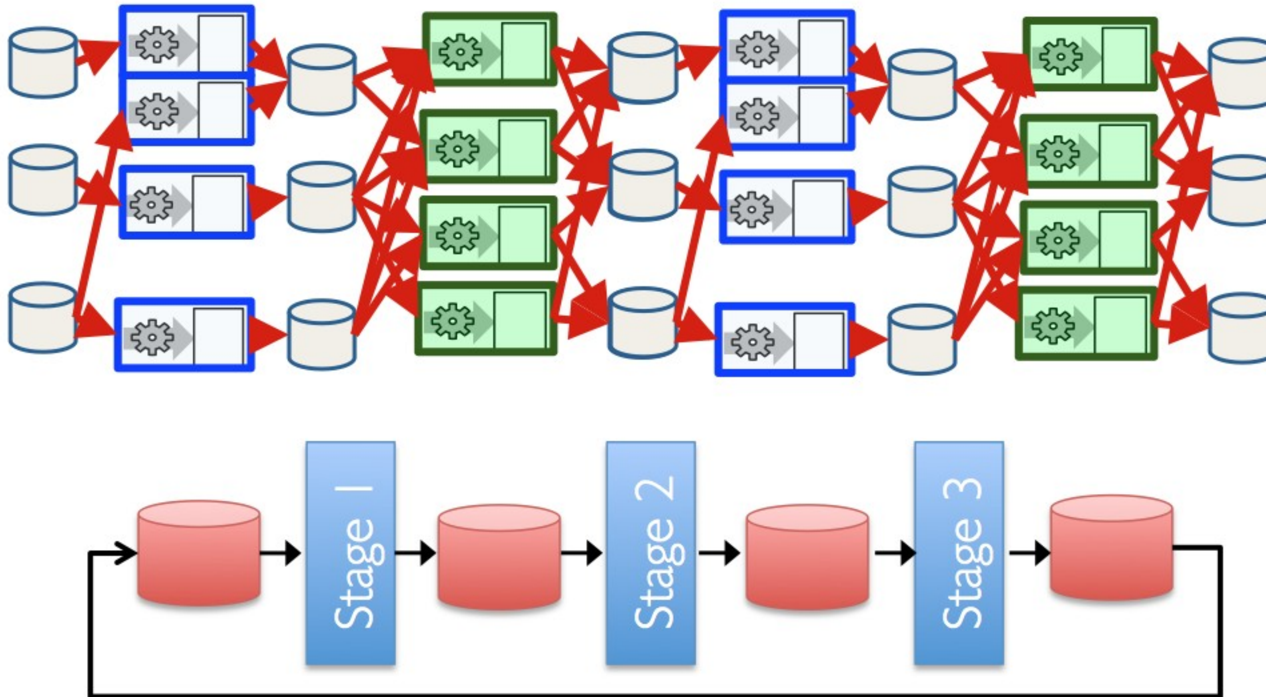


¿Qué es Apache Spark ?

Es un sistema de cómputo distribuído, **en memoria**, tolerante a fallas y escalable.

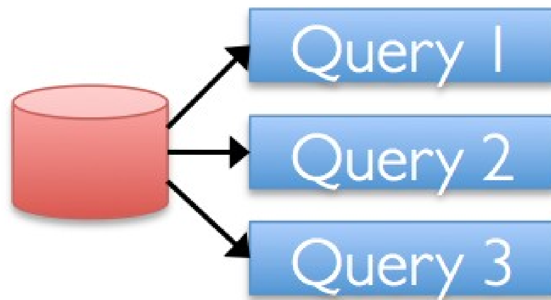
Puede pensarse como un simil o evolución de Hadoop, pero en memoria

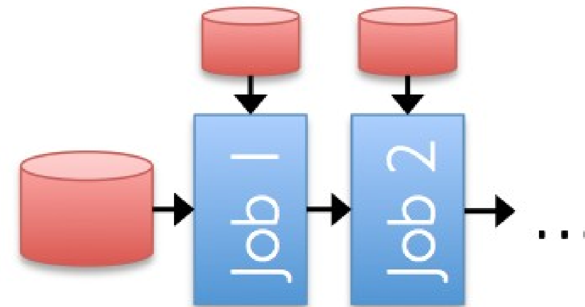
Motivación



El proceso iteritivo

Motivación (ii)





Realizar análisis interactivo sobre conjuntos de datos, o procesar datos tipo *stream* también son escenarios **intensivos en acceso a disco**

Apache Spark

Surge como un proyecto de UC Berkeley en 2009

Se transforma en proyecto de Apache en 2013

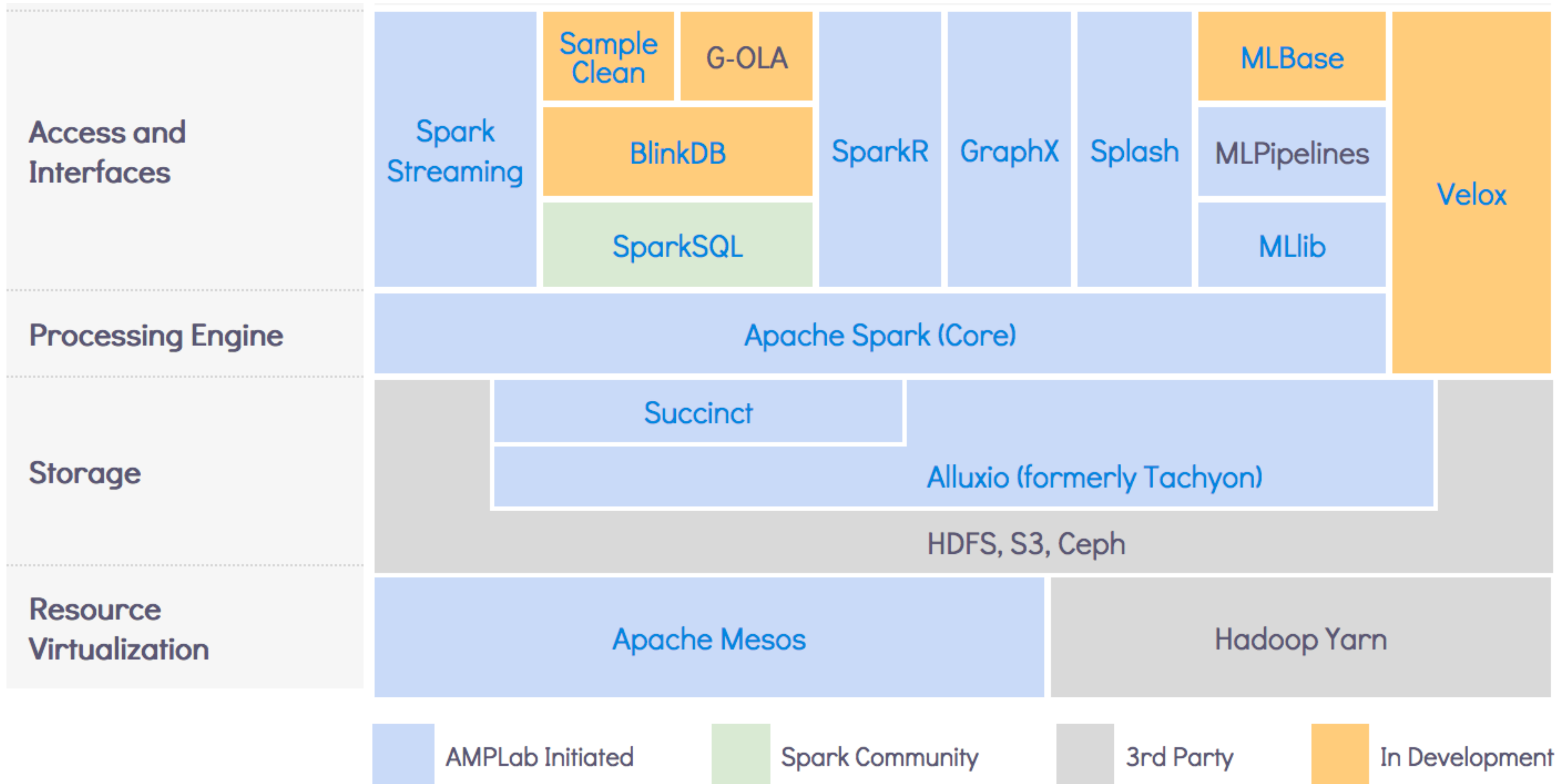
Spark: Cluster Computing with Working Sets.

Matei et al.. HotCloud 2010.

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.

Matei et al.. NSDI 2012.

Berkeley Data Analytics Stack

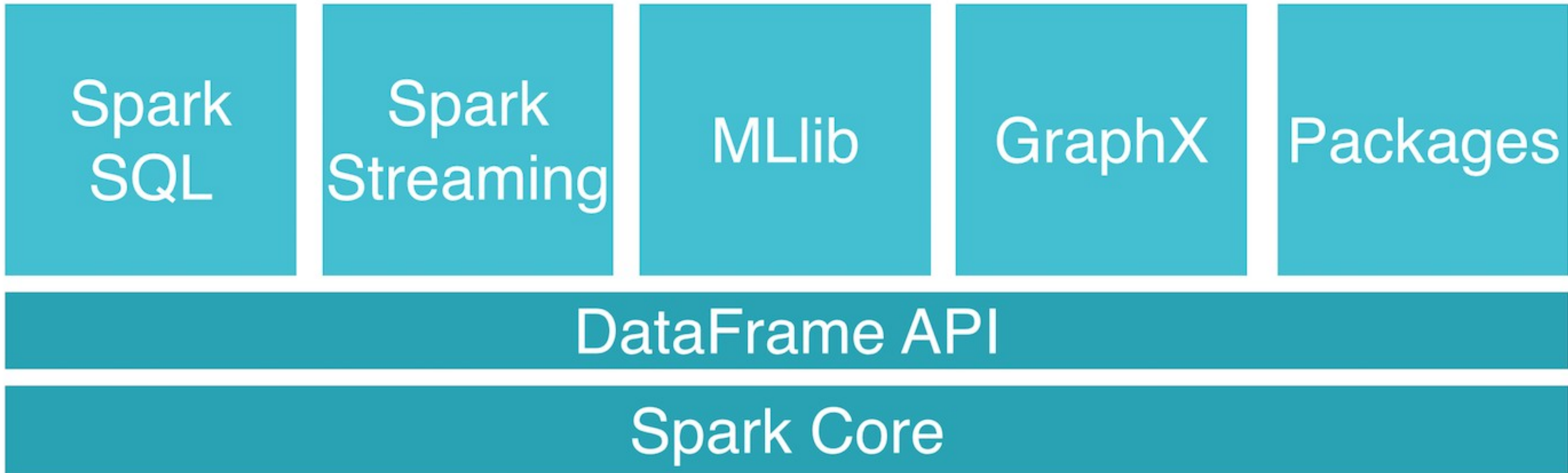


Fuente: https://amplab.cs.berkeley.edu/software/

Scala

Java

python

R

Spark SQL
Spark Streaming
MLlib
GraphX
Packages
DataFrame API
Spark Core

Data Source

hadoop
cassandra
HIVE
APACHE HBASE
PostgreSQL
CSV
{JSON}
MySQL
elasticsearch.

Modelo de Programación

Resilient Distributed Datasets (RDDs)

- Colecciones *read-only* de objetos
- Operaciones en paralelo sobre los RDDs
- Variables compartidas

Dataframes

- Similares a los RDDs pero para datos estructurados
- Infiere un esquema a partir de los datos
- Luego puedo usar sparkSQL

Algunos aspectos sobre los RDDs

Son colecciones de **objetos** que
se particionan en diferentes máquinas.

Por defecto son *lazy* y efímeras.

¿cómo se resuelve la **tolerancia a fallas**?

Se guarda suficiente información de *lineage* o *provenance* como para poder recomputar cualquier RDD

¿cómo se crean los RDDs?

1 . Desde **archivos**

2 . **Particionando** (*"parallelizing")* una colección Scala

3 . **Transformando** un RDD existente (via *flatMap* y funciones)

4 . Cambiando el modo de **persistencia** de un RDD existente: *cache* y *save*

¿cómo se manipulan los RDDs?

Transformaciones

$map(f : T \Rightarrow U)$	:	$RDD[T] \Rightarrow RDD[U]$
$filter(f : T \Rightarrow Bool)$	:	$RDD[T] \Rightarrow RDD[T]$
$flatMap(f : T \Rightarrow Seq[U])$	:	$RDD[T] \Rightarrow RDD[U]$
$sample(fraction : Float)$	:	$RDD[T] \Rightarrow RDD[T]$ (Deterministic sampling)
$groupByKey()$	:	$RDD[(K, V)] \Rightarrow RDD[(K, Seq[V])]$
$reduceByKey(f : (V, V) \Rightarrow V)$	:	$RDD[(K, V)] \Rightarrow RDD[(K, V)]$
$union()$	:	$(RDD[T], RDD[T]) \Rightarrow RDD[T]$
$join()$	:	$(RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (V, W))]$
$cogroup()$	:	$(RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (Seq[V], Seq[W]))]$
$crossProduct()$	:	$(RDD[T], RDD[U]) \Rightarrow RDD[(T, U)]$
$mapValues(f : V \Rightarrow W)$	:	$RDD[(K, V)] \Rightarrow RDD[(K, W)]$ (Preserves partitioning)
$sort(c : Comparator[K])$	:	$RDD[(K, V)] \Rightarrow RDD[(K, V)]$
$partitionBy(p : Partitioner[K])$	:	$RDD[(K, V)] \Rightarrow RDD[(K, V)]$

ATENCIÓN!

map es un mapping 1-1

flatMap es similar al map de MapReduce

Acciones

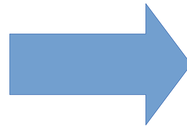
$count()$	:	$RDD[T] \Rightarrow Long$
$collect()$	:	$RDD[T] \Rightarrow Seq[T]$
$reduce(f : (T, T) \Rightarrow T)$	:	$RDD[T] \Rightarrow T$
$lookup(k : K)$	:	$RDD[(K, V)] \Rightarrow Seq[V]$ (On hash/range partitioned RDDs)
$save(path : String)$	:	Outputs RDD to a storage system, *e.g.*, HDFS

Ejemplo: conteo de palabras

```
public static class WordCountMapClass extends MapReduceBase
implements Mapper<LongWritable, Text, Text, IntWritable> {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();
  }

public void map(LongWritable key, Text value,
OutputCollector<Text, IntWritable> output,  Reporter
reporter) throws IOException {
  String line = value.toString();
  StringTokenizer itr = new StringTokenizer(line);
  while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    output.collect(word, one);
    }
  }

public static class WorkdCountReduce extends MapReduceBase
implements Reducer<Text, IntWritable, Text, IntWritable> {
  public void reduce(Text key, Iterator<IntWritable> values,
OutputCollector<Text, IntWritable> output, Reporter
reporter) throws IOException {
    int sum = 0;
    while (values.hasNext()) {
      sum += values.next().get();
      }
    output.collect(key, new IntWritable(sum));
    }
  }
```



```
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Fuente: Introduction to Spark with Scala, Himanshu Gupta

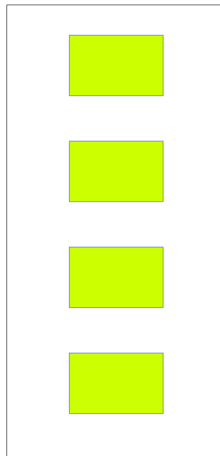
Ejemplo: conteo de palabras en Scala (cont)

```scala
val master = "local"
val conf = new SparkConf().setMaster(master)
val sc = new SparkContext(conf)
```

Ejemplo: conteo de palabras en Scala (cont)

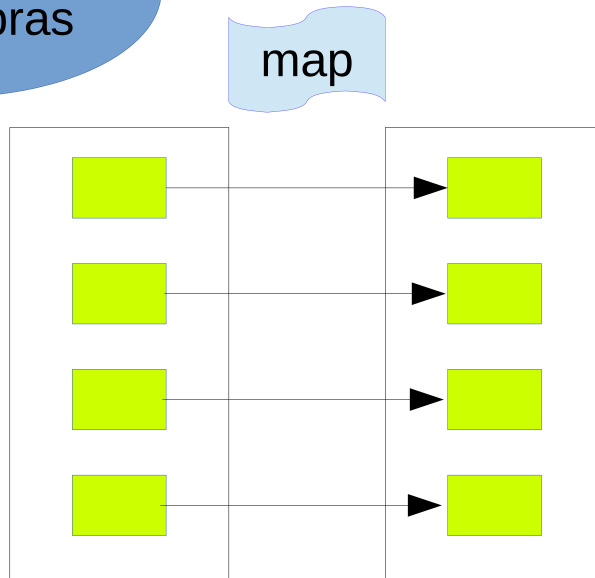
```
val master = "local"
val conf = new SparkConf().setMaster(master)
val sc = new SparkContext(conf)
val lines = sc.textFile("data.txt")
```

Creación de
un RDD desde
archivo



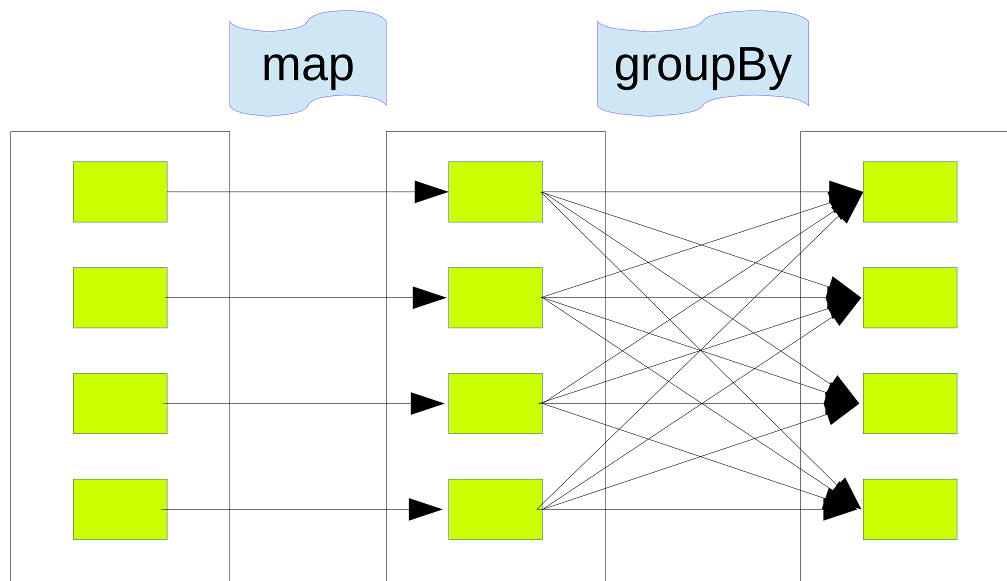
Ejemplo: conteo de palabras en Scala(cont)

```
val master = "local"
val conf = new SparkConf().setMaster(master)
val sc = new SparkContext(conf)
val lines = sc.textFile("demo.txt")
val words = lines.flatMap(_.split(" ")).map((_,1))
```



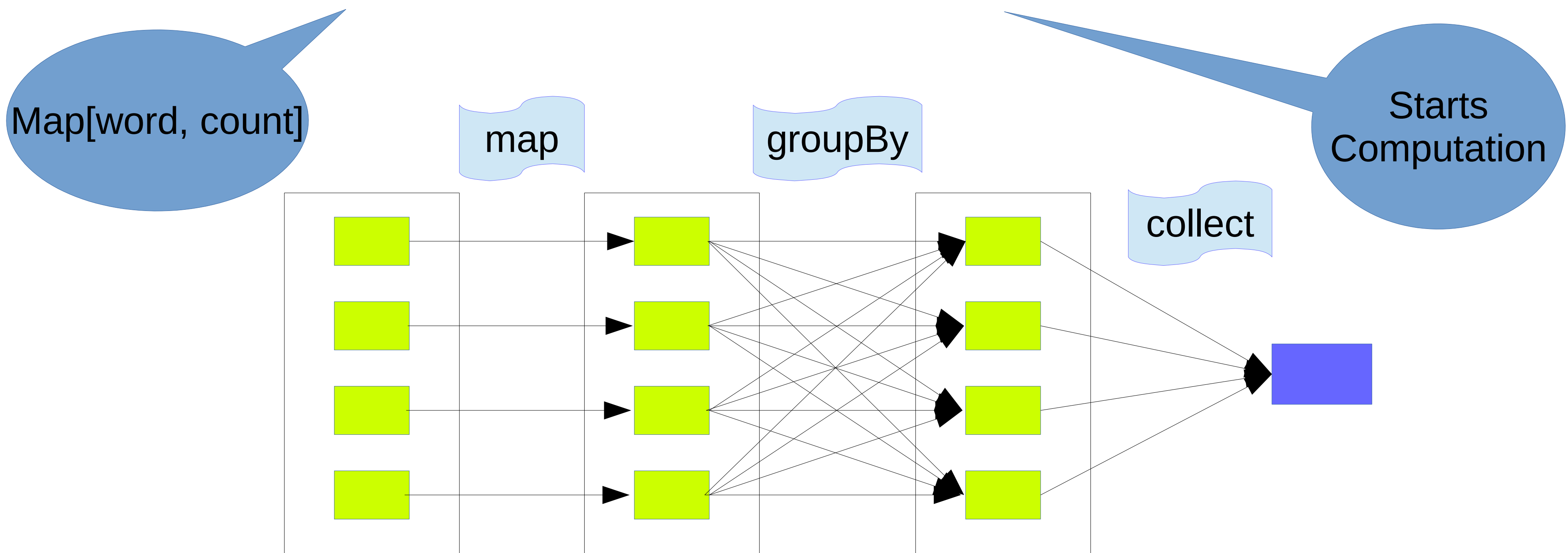
Ejemplo: conteo de palabras en Scala (cont)

```
val master = "local"
val conf = new SparkConf().setMaster(master)
val sc = new SparkContext(conf)
val lines = sc.textFile("demo.txt")
val words = lines.flatMap(_.split(" ")).map((_,1))
val wordCountRDD = words.reduceByKey(_ + _)
```



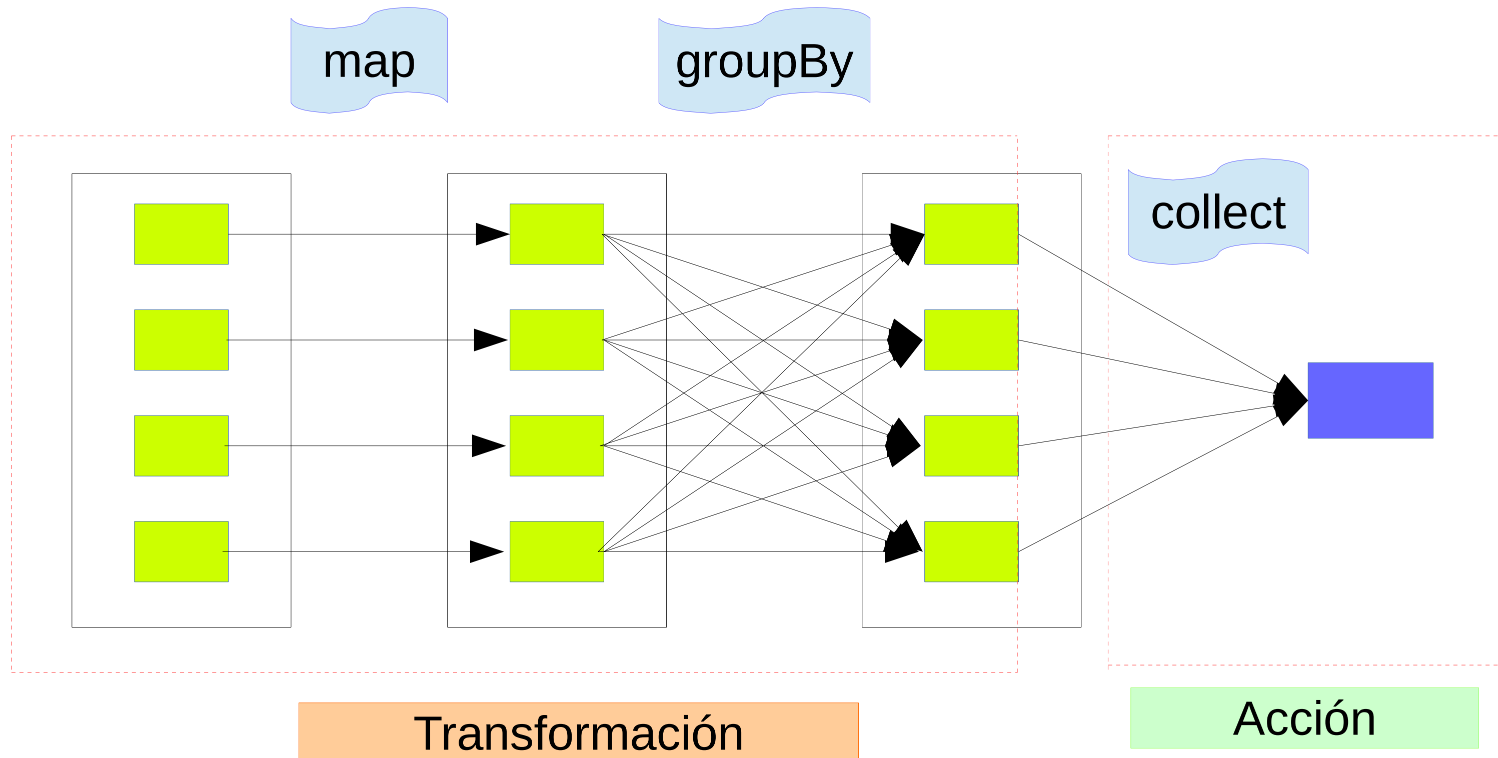
Ejemplo: conteo de palabras en Scala (cont)

```scala
val master = "local"
val conf = new SparkConf().setMaster(master)
val sc = new SparkContext(conf)
val lines = sc.textFile("demo.txt")
val words = lines.flatMap(_.split(" ")).map((_,1))
val wordCountRDD = words.reduceByKey(_ + _)
val wordCount = wordCountRDD.collect
```

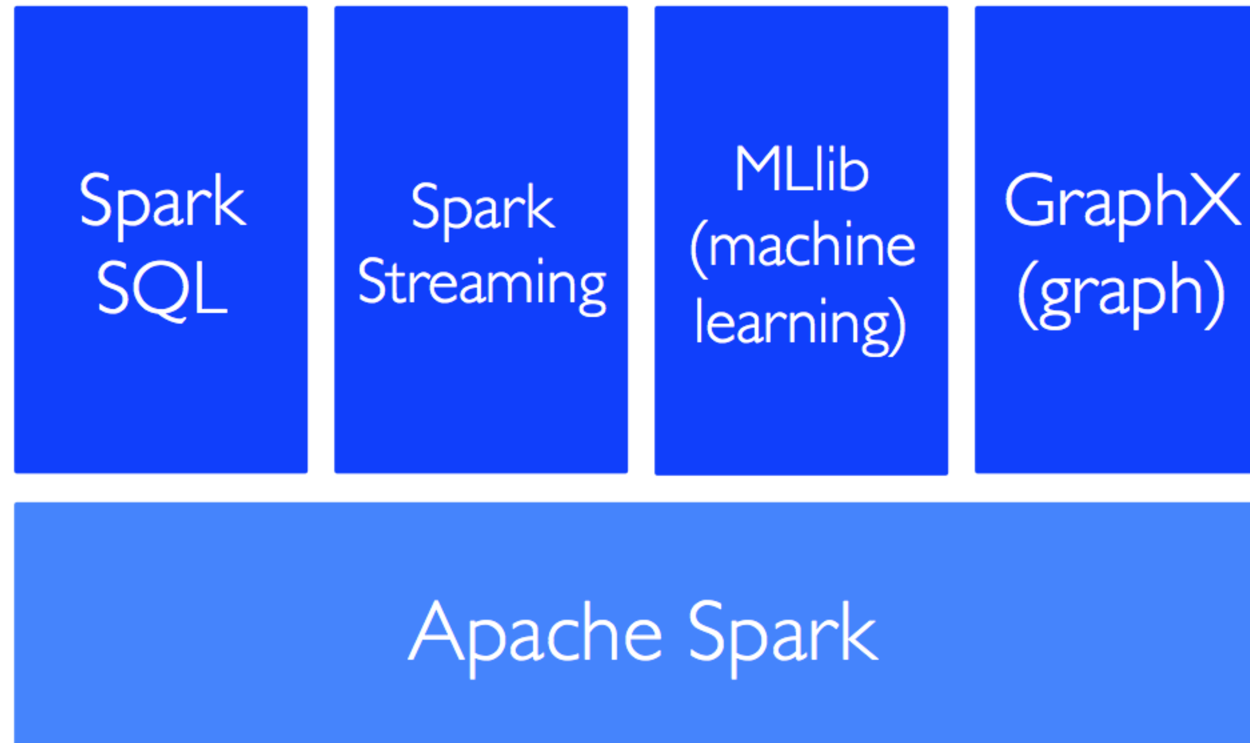


Ejemplo: conteo de palabras en pySpark

```
input_file = sc.textFile("demo.txt")
map = input_file.flatMap(lambda line: line.split("")).map(lambda word: (word, 1))
counts = map.reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("/path/to/output/")
```



Herramientas sobre Spark



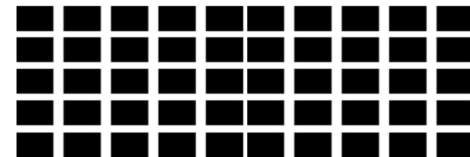
Algunas diferencias entre Hadoop y Spark

	Hadoop Map Reduce	Spark
Storage	Disk only	In-memory or on disk
Operations	Map and Reduce	Map, Reduce, Join, Sample, etc…
Execution model	Batch	Batch, interactive, streaming
Programming environments	Java	Scala, Java, R, and Python

Desafío *Daytona GraySort*

La tarea: ordenar 100 TB de datos!!

Hadoop (2013): 2100 nodos



72 minutos

Spark (2014): 206 nodos

23 minutos

Más info sobre el experimento :https://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html

¿quién gana?
Hadoop vs Spark

Hadoop MapReduce is meant for data that does not fit in the memory whereas Apache **Spark has a better performance for the data that fits in the memory**, particularly on dedicated clusters.

Hadoop MapReduce can be an economical option because of Hadoop as a service offering(HaaS) and availability of more personnel. According to the benchmarks, Apache Spark is more cost effective but **staffing would be expensive in case of Spark**.

Apache Spark and Hadoop MapReduce both are failure tolerant but comparatively **Hadoop MapReduce is more failure tolerant than Spark**.

Spark and Hadoop MapReduce both have similar compatibility in terms of data types and data sources.

Programming in Apache Spark is easier as it has an interactive mode whereas Hadoop MapReduce requires core java programming skills,however there are several utilities that make programming in Hadoop MapReduce easier.

https://www.dezyre.com/article/hadoop-mapreduce-vs-apache-spark-who-wins-the-battle/83

Referencias y material adicional

- Artículos sobre Spark y proyectos asociados https://spark.apache.org/research.html
- Big Data Analytics with Spark *A Practitioner's Guide to Using Spark for Large-Scale Data Processing, Machine Learning, and Graph Analytics, and High-Velocity Data Stream Processing*, Guller Apress 2015. (Disponible en el portal Timbo)