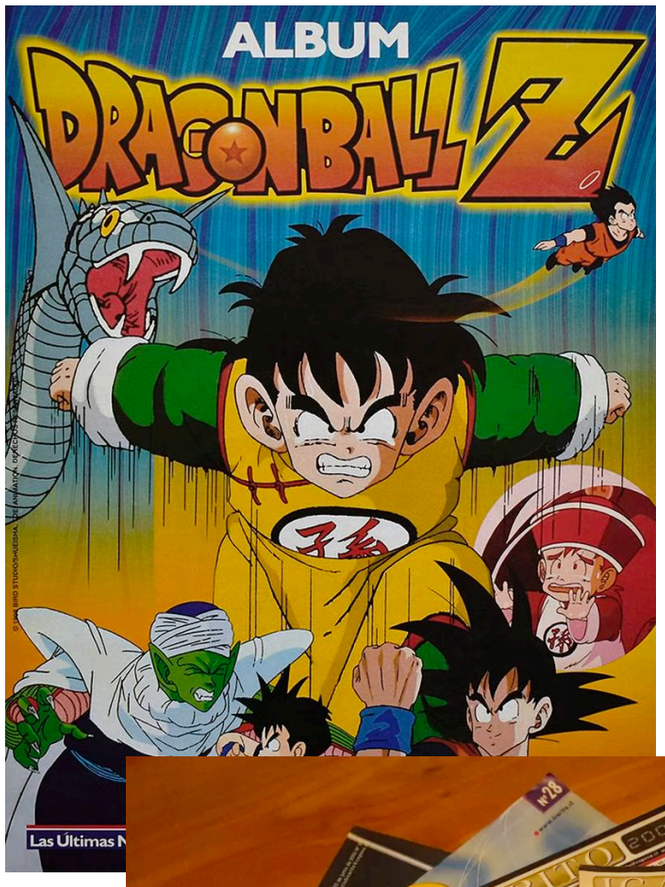
Procesamiento de Datos Masivos

Clase 09 - Map-Reduce y Apache Spark

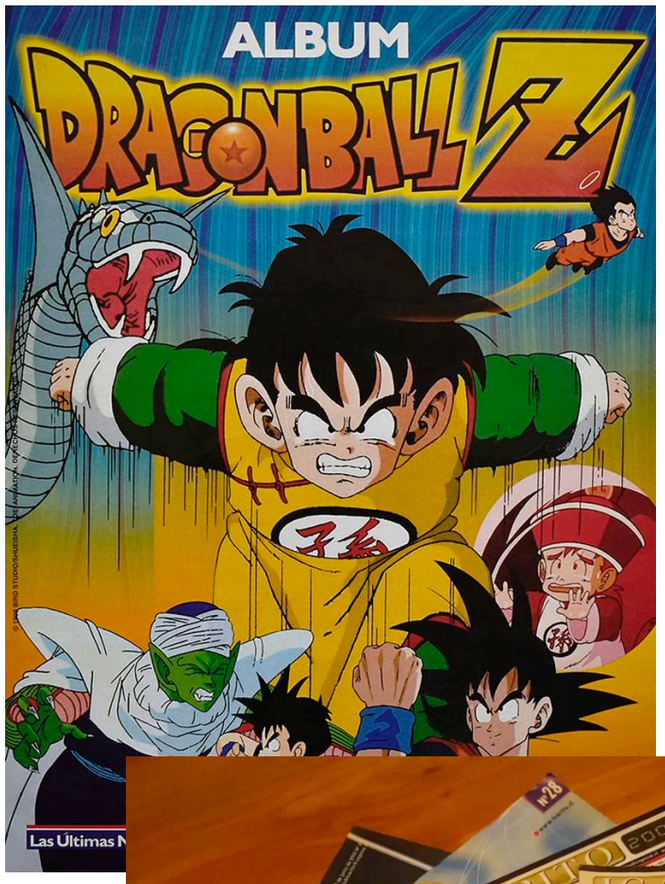
Hace mucho tiempo

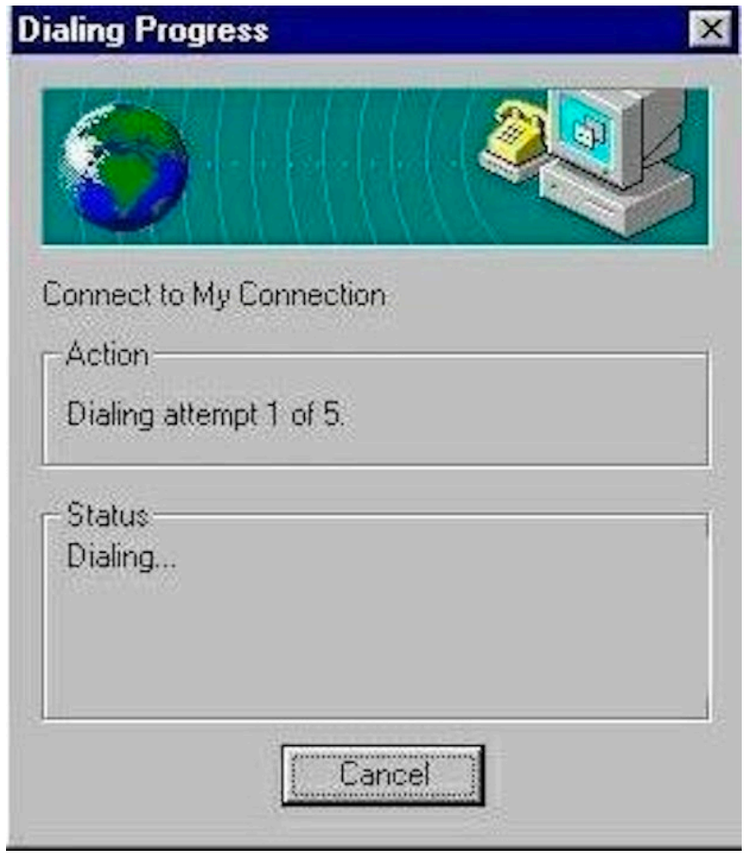
Hace mucho tiempo

Hace mucho tiempo

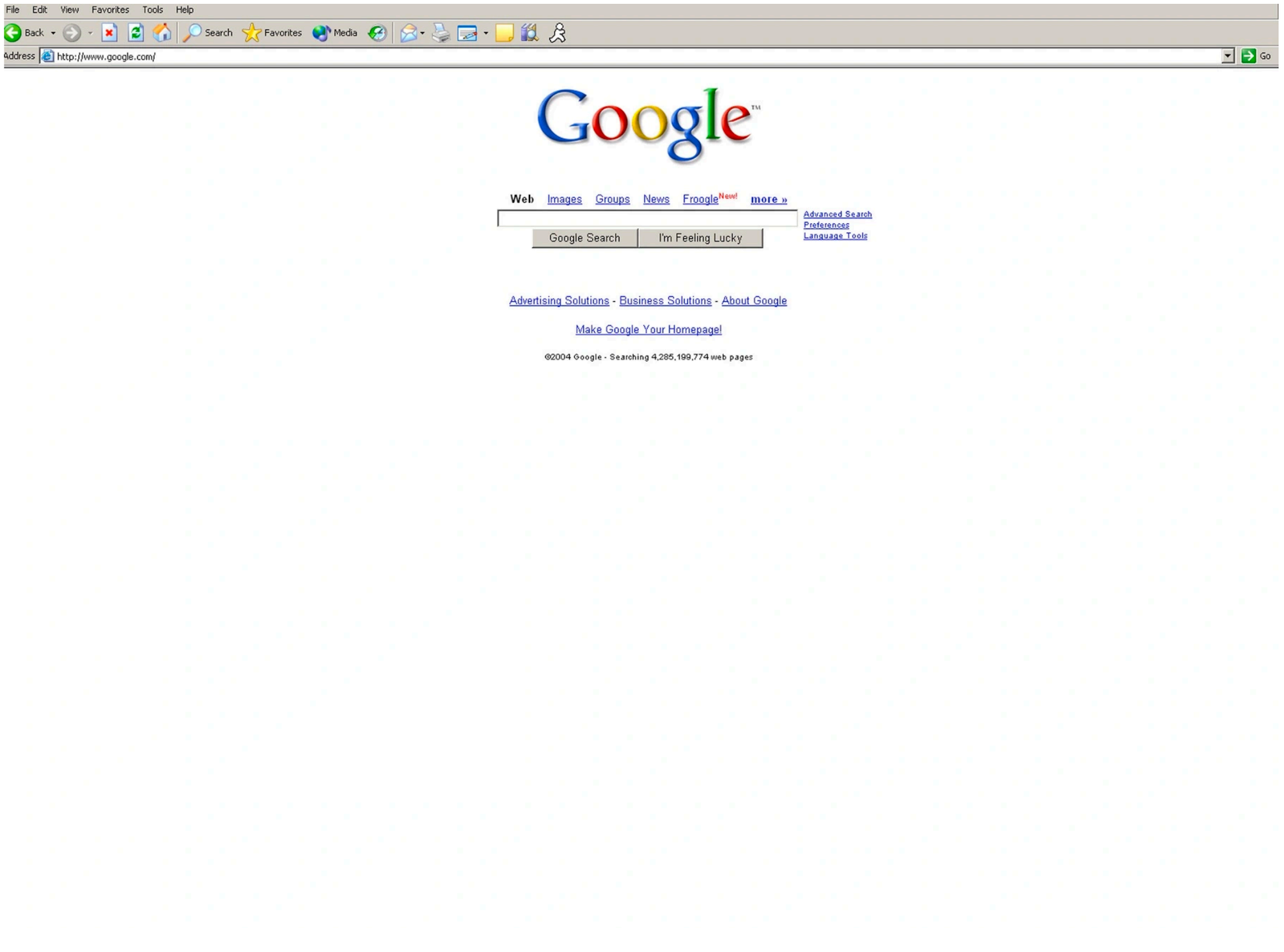


Hace mucho tiempo





Hace mucho tiempo


File Edit View Favorites Tools Help
Back
Search
Favorites
Media
Address
http://www.google.com/
Go
Google
Web
Images
Groups
News
Froogle
New!
more »
Advanced Search
Preferences
Language Tools
Google Search
I'm Feeling Lucky
Advertising Solutions - Business Solutions - About Google
Make Google Your Homepage!
©2004 Google - Searching 4,285,199,774 web pages

El inicio de Google

Google fue el primer buscador realmente exitoso, permitía buscar sitios en toda la web donde los resultados eran sobresalientes

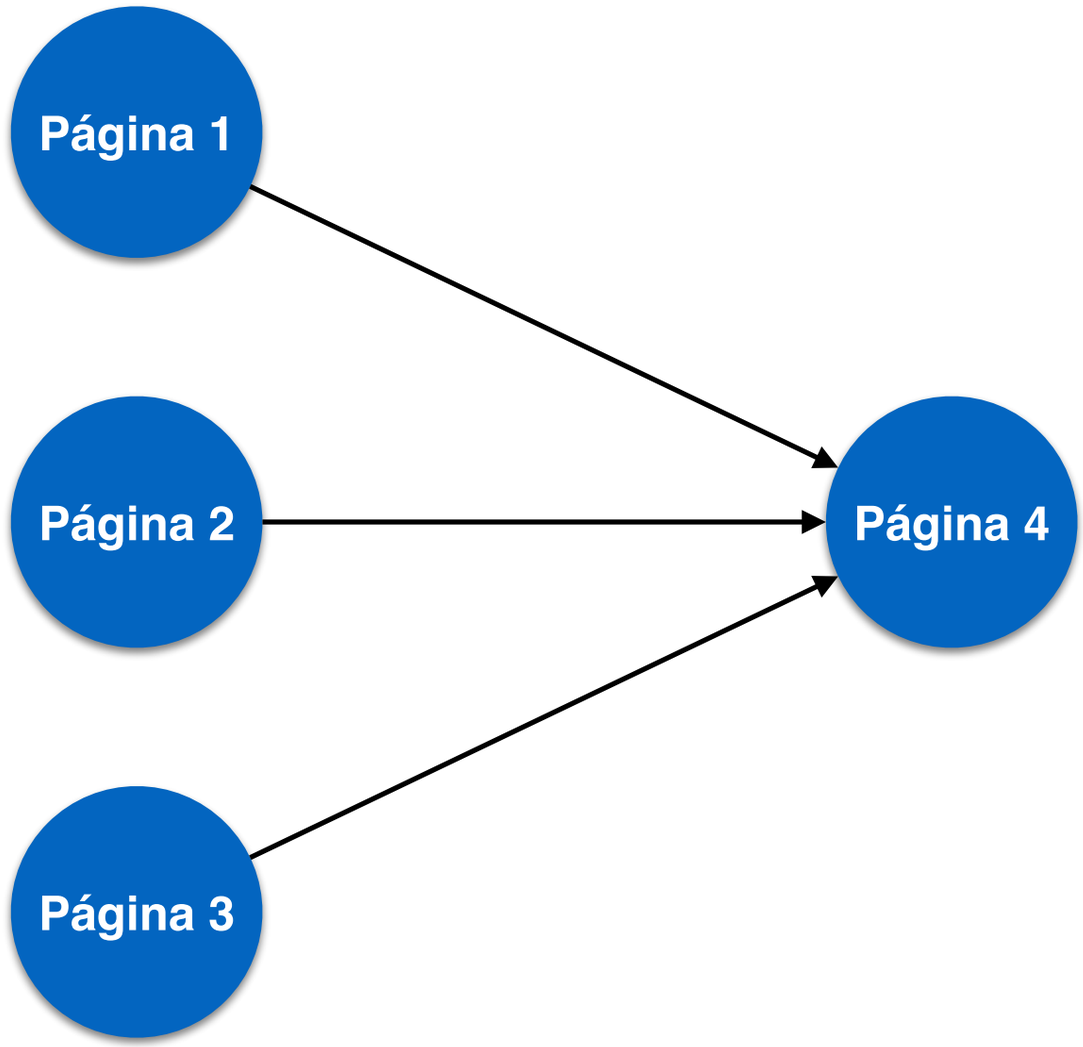
El secreto detrás de Google es el algoritmo de PageRank

Page Rank

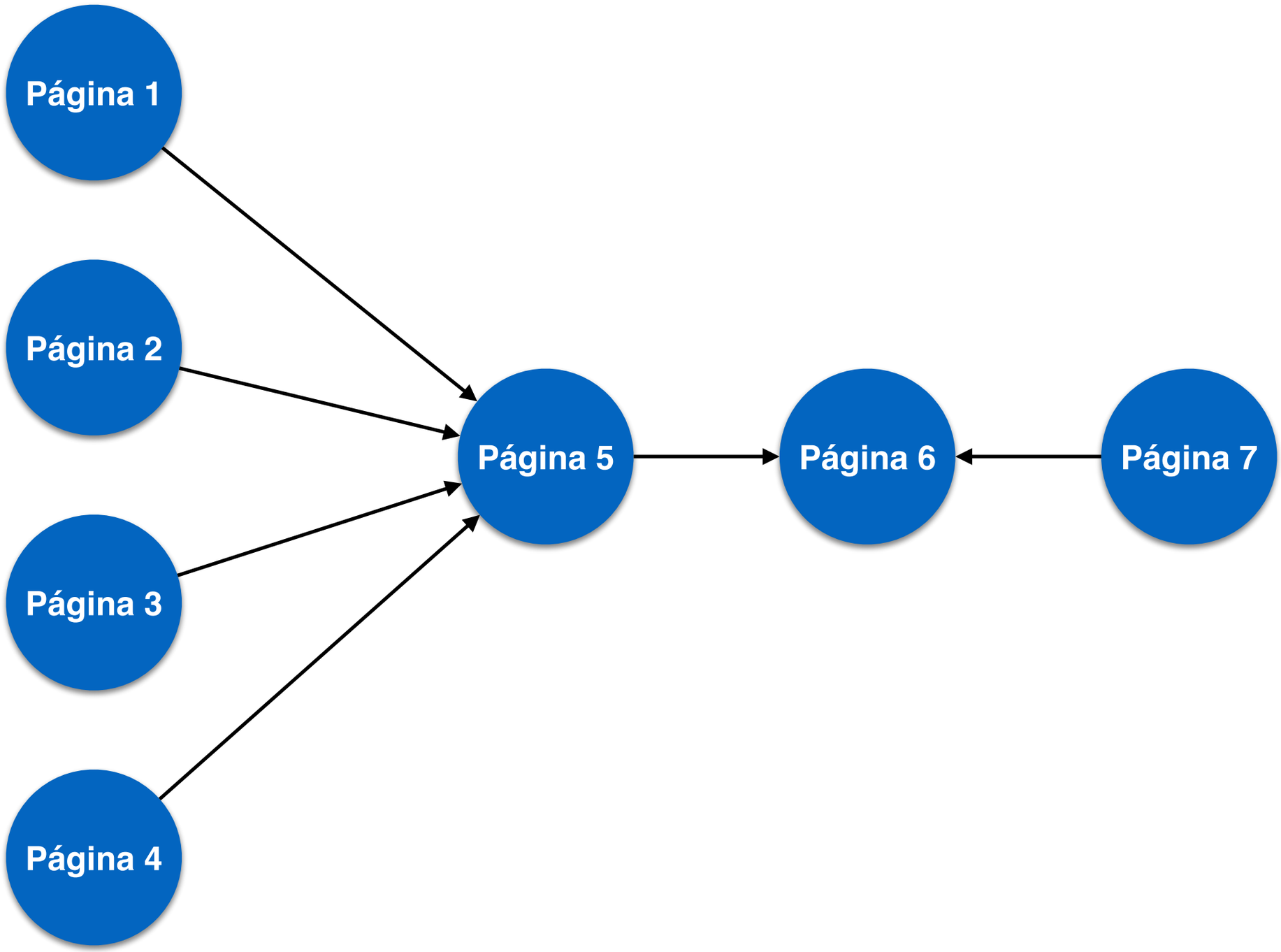
Page Rank es un algoritmo para calcular centralidad en grafos, esto es **obtener los nodos más importantes de una red**

Se basa en el hecho de que un nodo es más importante si es apuntado por nodos importantes

¿Qué nodo es más importante en esta red?



¿Qué nodo es más importante en esta red?



Page Rank

El algoritmo simula un caminante aleatorio por toda la web, y calcula cuál es la página en la que es más probable que finalice su camino

El algoritmo se basa en que todos los nodos parten con un Page Rank inicial de 1/n

En cada iteración, los nodos regalan en partes iguales el Page Rank a los vecinos a los que apuntan

Page Rank

El algoritmo tiene más detalles pero esa es la idea principial

¿Pero este algoritmo es realmente novedoso como para justificar el éxito de Google?

Lo realmente novedoso fue poder hacer este calculo para **toda la web**

En esta clase

- Vamos a estudiar el paradigma de computación Map-Reduce y su historia
- Vamos a entender cómo los algoritmos clásicos de las bases de datos se pueden traducir a algoritmos en entornos distribuidos
- Vamos a crear código pensado para funcionar en un entorno distribuido con Apache Spark

Repaso: map-reduce para iterables

La función **map** nos sirve para aplicar la misma función a todos los elementos del iterable

```
def cuadrado(numero):
    return numero ** 2

numeros = [1, 2, 3, 4, 5]
cuadrados = map(cuadrado, numeros)

# map() retorna un objeto iterable, así que podemos convertirlo a una
# lista para visualizar los resultados
cuadrados_lista = list(cuadrados)

print(cuadrados_lista)

# >> [1, 4, 9, 16, 25]
```

Repaso: map-reduce para iterables

También es común usar **map** con funciones **lambda**

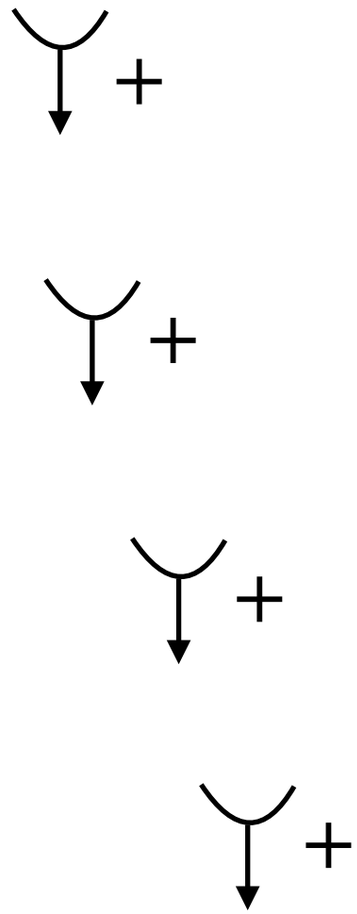
```
numeros = [1, 2, 3, 4, 5]
cuadrados = map(lambda x: x ** 2, numeros)

cuadrados_lista = list(cuadrados)

print(cuadrados_lista)
```

Repaso: map-reduce para iterables

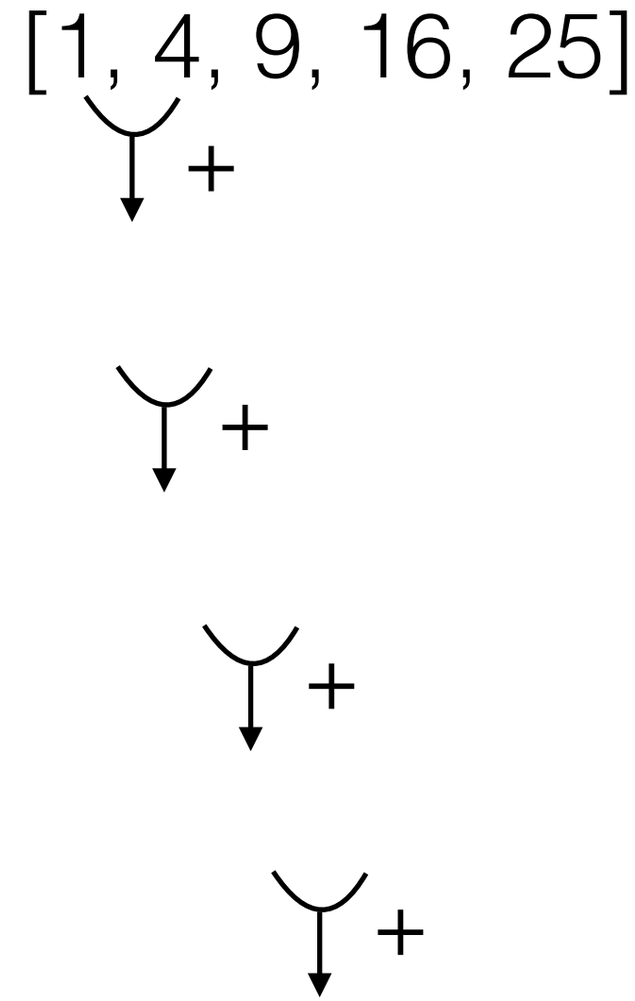
La función **reduce** nos permite pasar de un iterable a un único elemento

```
import functools

def suma(x, y):
    return x + y

numeros = [1, 4, 9, 16, 25]
total = functools.reduce(suma, numeros)

print(total)
```

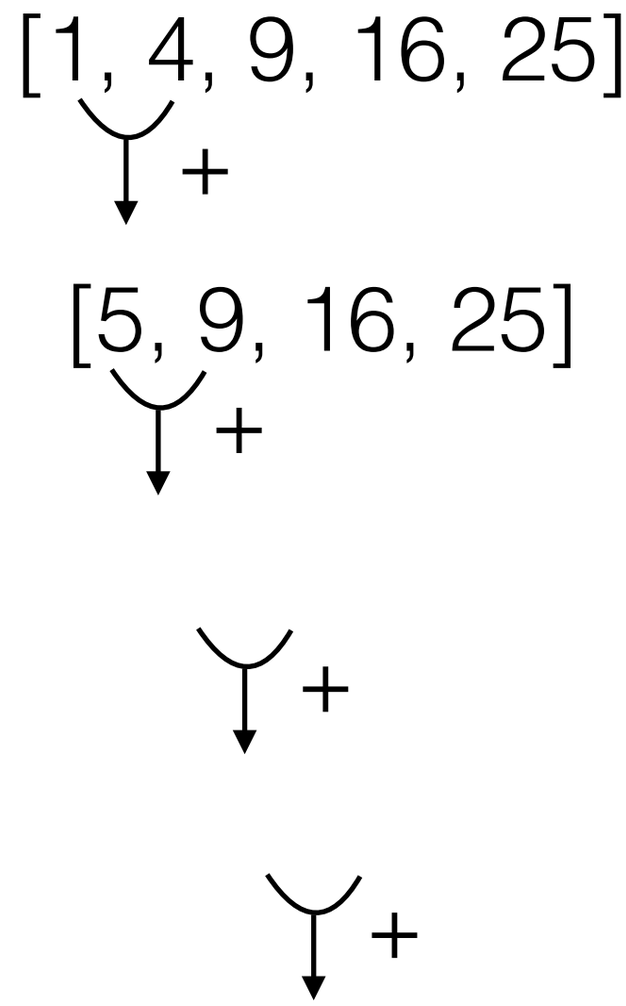
Ejemplo: función reduce



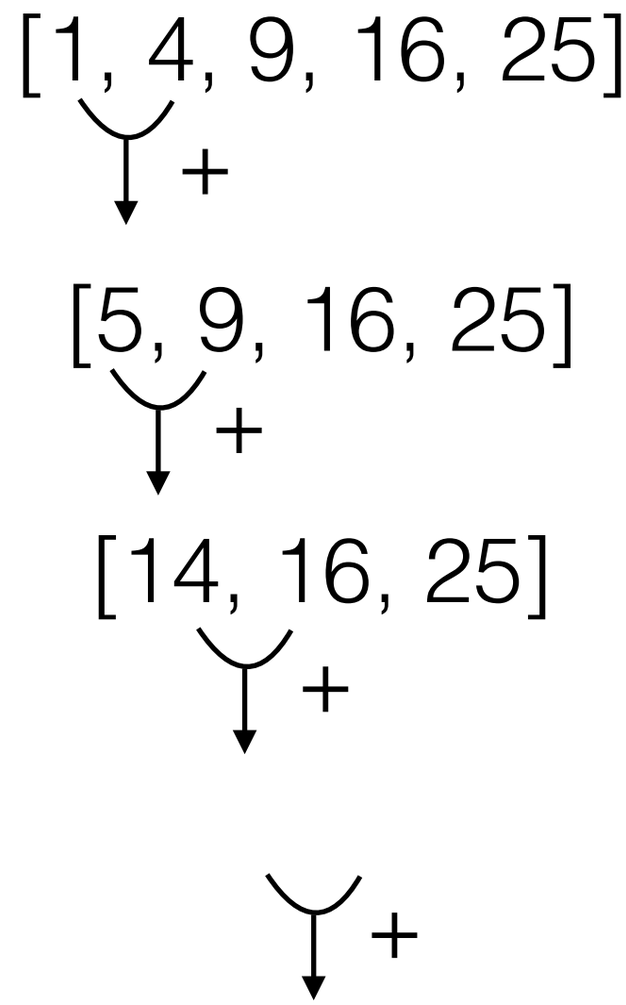
Ejemplo: función reduce



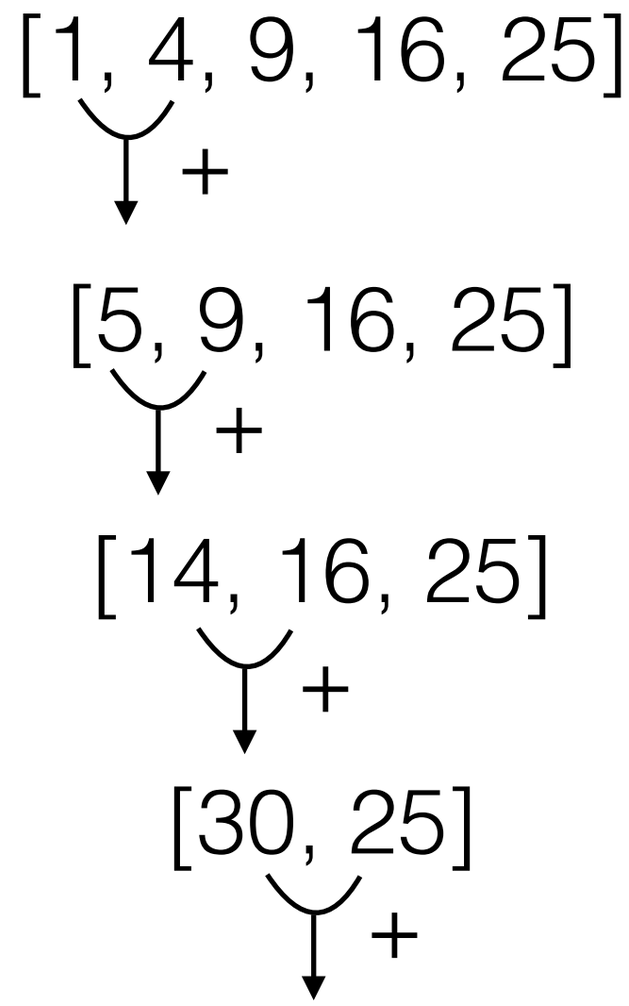
Ejemplo: función reduce



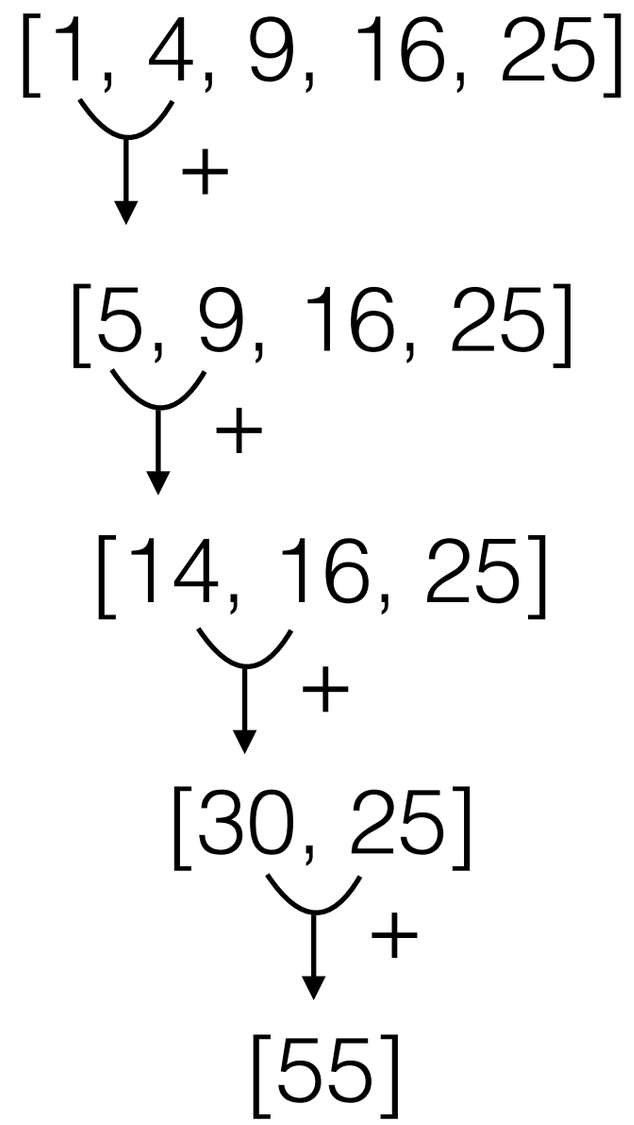
Ejemplo: función reduce



Ejemplo: función reduce



Ejemplo: función reduce



Map-Reduce distribuido

El paradigma Map-Reduce se basa en que la función **map** emita un par key-value

Todos los pares con la misma llave van al mismo nodo de cómputo

Por cada key, se forma un iterable con todos los valores asociados; la función **reduce** se aplica sobre ese iterable

Map-Reduce distribuido

Ejemplo: contar palabras de un archivo

Supongamos el siguiente archivo

```
Deer Bear River
Car Car River
Deer Car Bear
```

Cuando hacemos map-reduce en un entorno distribuido, cada línea del archivo puede estar en un computador distinto

Map-Reduce distribuido

Ejemplo: contar palabras de un archivo

A cada línea le hacemos split por espacio

```
["Deer", "Bear", "River"]
["Car", "Car", "River"]
["Deer", "Car", "Bear"]
```

Y luego aplicamos la función **map**, ejecutando la siguiente función: **lambda x: (x, 1)**

Map-Reduce distribuido

Ejemplo: contar palabras de un archivo

Nuestras listas se ven así:

```
[("Deer", 1), ("Bear", 1), ("River", 1)]
[("Car", 1), ("Car", 1), ("River", 1)]
[("Deer", 1), ("Car", 1), ("Bear", 1)]
```

Luego hacemos **reduce** (o **reduceByKey**), en el que nos aseguramos que si dos llaves son iguales queden en el mismo nodo de cómputo

Nodo 1

"Bear": [1, 1]
"Car": [1, 1, 1]

Nodo 2

"Deer": [1, 1]

Nodo 3

"River": [1, 1]

Map-Reduce distribuido

Ejemplo: contar palabras de un archivo

La función reduce que usamos es:

```
lambda x, y: x + y
```

Y se aplica a cada uno de las listas

Nodo 1

"Bear": [2]
"Car": [3]

Nodo 2

"Deer": [2]

Nodo 3

"River": [2]

Joins con Map-Reduce

Supongamos que ahora este es nuestro archivo

```
T1,1,a
T1,1,b
T2,1,c
T2,2,d
```

Donde representamos dos tablas y queremos hacer un join por el campo numérico, ¿cómo podemos hacer un join con **map-reduce**?

Historia de Map-Reduce

Historia de Map-Reduce

- Map-Reduce y sistemas de archivos distribuidos
- Hadoop y HDFS
- Apache Spark
- Serverless Data Warehouse

Procesamiento de Datos Masivos

Clase 09 - Map-Reduce y Apache Spark