IR: Information Retrieval

FIB, Master in Innovation and Research in Informatics

Slides by Marta Arias, José Balcázar, Ricard Gavaldá
Department of Computer Science, UPC

Fall 2016

http://www.cs.upc.edu/~ir

6. Architecture of large-scale systems. Mapreduce. Big Data

Architecture of Web Search & Towards Big Data

Outline:

1. Scaling the architecture: Google cluster, BigFile, Mapreduce/Hadoop
2. Big Data and NoSQL databases
3. The Apache ecosystem for Big Data

Google 1998. Some figures

- 24 million pages
- 259 million anchors
- 147 Gb of text
- 256 Mb main memory per machine
- 14 million terms in lexicon
- 3 crawlers, 300 connection per crawler
- 100 webpages crawled / second, 600 Kb/second
- 41 Gb inverted index
- 55 Gb info to answer queries; 7Gb if doc index compressed
- Anticipate hitting O.S. limits at about 100 million pages

Google today?

- Current figures = $\times$ 1,000 to $\times$ 10,000

- 100 thousands Gb transferred per day
- 100 million Gb of storage
- Several 10s of copies of the accessible web
- 1 million machines?

Google in 2003

- More applications, not just web search
- Many machines, many data centers, many programmers
- Huge & complex data
- Need for abstraction layers

Three influential proposals:

- Hardware abstraction: The Google Cluster
- Data abstraction: The Google File System

  BigFile (2003), BigTable (2006)
- Programming model: MapReduce

Google cluster, 2003: Design criteria

Use more cheap machines, not expensive servers

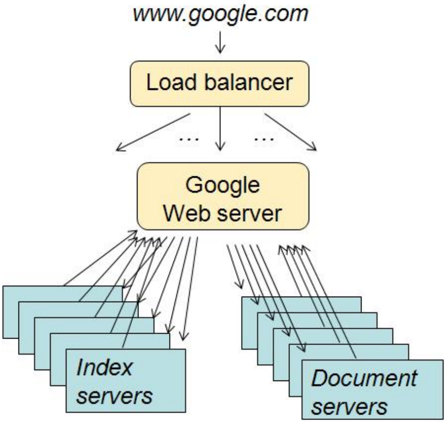
- High task parallelism; Little instruction parallelism

  (e.g., process posting lists, summarize docs)
- Peak processor performance less important than price/performance

  price is superlinear in performance!
- Commodity-class PCs. Cheap, easy to make redundant
- Redundancy for high throughput
- Reliability for free given redundancy. Managed by soft
- Short-lived anyway (< 3 years)

L.A. Barroso, J. Dean, U. Hölzle: “Web Search for a Planet: The Google Cluster Architecture”, 2003

Google cluster for web search



- Load balancer chooses freest / closest GWS
- GWS asks several index servers
- They compute hit lists for query terms, intersect them, and rank them
- Answer (docid list) returned to GWS
- GWS then asks several document servers
- They compute query-specific summary, url, etc.
- GWS formats an html page & returns to user

Index “shards”

- Documents randomly distributed into “index shards”
- Several replicas (index servers) for each indexshard
- Queries routed through local load balancer
- For speed & fault tolerance
- Updates are infrequent, unlike traditional DB’s
- Server can be temporally disconnected while updated

The Google File System, 2003

- System made of cheap PC's that fail often
- Must constantly monitor itself and recover from failures transparently and routinely
- Modest number of large files (GB's and more)
- Supports small files but not optimized for it
- Mix of large streaming reads + small random reads
- Occasionally large continuous writes
- Extremely high concurrency (on same files)

S. Ghemawat, H. Gobioff, Sh.-T. Leung: “The Google File System”, 2003

The Google File System, 2003

- One GFS cluster = 1 master process + several chunkservers
- BigFile broken up in chunks
- Each chunk replicated (in different racks, for safety)
- Master knows mapping chunks $\rightarrow$ chunkservers
- Each chunk unique 64-bit identifier
- Master does not serve data: points clients to right chunkserver
- Chunkservers are stateless; master state replicated
- Heartbeat algorithm: detect & put aside failed chunkservers

MapReduce and Hadoop

- Mapreduce: Large-scale programming model developed at Google (2004)
  - Proprietary implementation
  - Implements old ideas from functional programming, distributed systems, DB's ...
- Hadoop: Open source (Apache) implementation at Yahoo! (2006 and on)
  - HDFS: Open Source *Hadoop Distributed File System;* analog of BigFile
  - Pig: Yahoo! Script-like language for data analysis tasks on Hadoop
  - Hive: Facebook SQL-like language / datawarehouse on Hadoop
  - ...

MapReduce and Hadoop

Design goals:

- Scalability to large data volumes and number of machines
  - 1000's of machines, 10,000's disks
  - Abstract hardware & distribution (compare MPI: explicit flow)
  - Easy to use: good learning curve for programmers
- Cost-efficiency:
  - Commodity machines: cheap, but unreliable
  - Commodity network
  - Automatic fault-tolerance and tuning. Fewer administrators

HDFS

- Optimized for large files, large sequential reads
- Optimized for “write once, read many”
- Large blocks (64MB). Few seeks, long transfers
- Takes care of replication & failures
- Rack aware (for locality, for fault-tolerant replication)
- Own types (`IntWritable`, `LongWritable`, `Text`, ...)
  - Serialized for network transfer and system & language interoperability

The MapReduce Programming Model

- Data type: (key, value) records
- Three (key, value) spaces
- `Map` function:

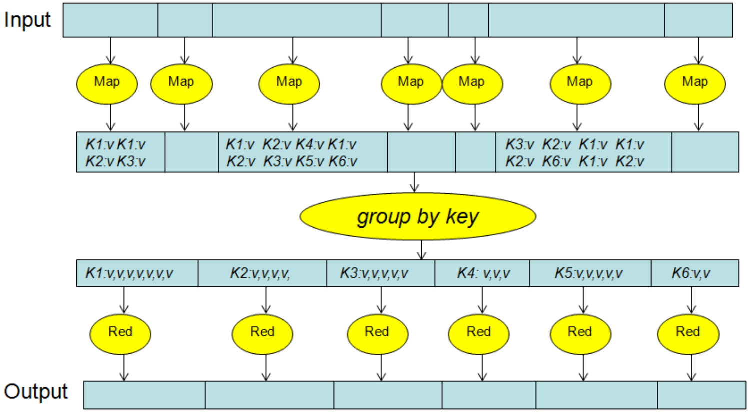
$$(K_{ini}, V_{ini}) \rightarrow \textsf{list}\langle (K_{inter}, V_{inter}) \rangle$$

- `Reduce` function:

$$(K_{inter}, \textsf{list}\langle V_{inter} \rangle) \rightarrow \textsf{list}\langle (K_{out}, V_{out}) \rangle$$

Semantics

Key step, handled by the platform: group by or shuffle by key

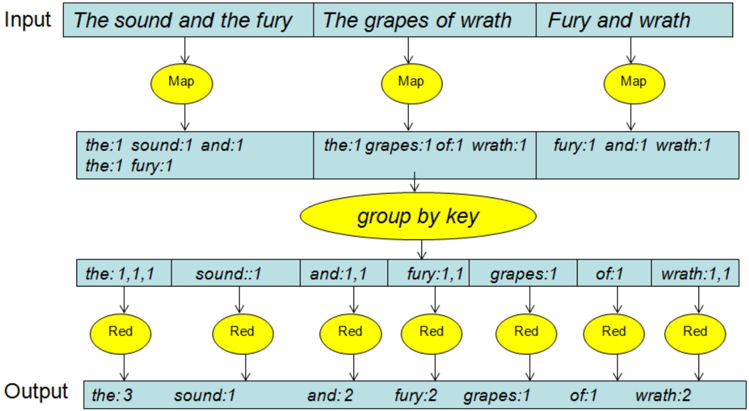


Example 1: Word Count

Input: A big file with many lines of text
Output: For each word, times that it appears in the file

```
map(line):
    foreach word in line.split() do
        output (word,1)

reduce(word,L):
    output (word,sum(L))
```

Example 1: Word Count



Example 2: Temperature statistics

Input: Set of files with records `(time,place,temperature)`
Output: For each place, report maximum, minimum, and average temperature

```
map(file):
    foreach record (time,place,temp) in file do
        output (place,temp)

reduce(p,L):
    output (p,(max(L),min(L),sum(L)/length(L)))
```

Example 3: Numerical integration

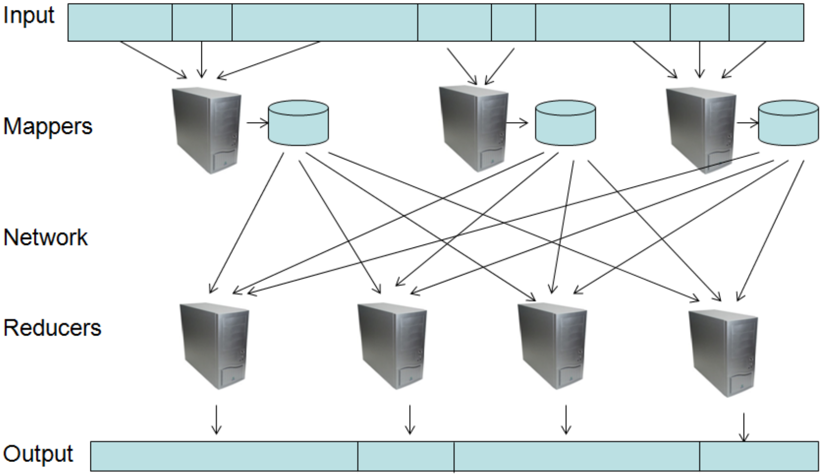
Input: A function $f : R \to R$, an interval $[a, b]$
Output: An approximation of the integral of f in $[a, b]$

```
map(start,end):
    sum = 0;
    for (x = start; x < end; x += step)
        sum += f(x)*step;
    output (0,sum)

reduce(key,L):
    output (0,sum(L))
```

Implementation

- Some *mapper* machines, some *reducer* machines
- Instances of *map* distributed to mappers
- Instances of *reduce* distributed to reduce
- Platform takes care of shuffling through network
- Dynamic load balancing
- Mappers write their output to local disk (not HDFS)
- If a map or reduce instance fails, automatically reexecuted
- Incidentally, information may be sent compressed

Implementation



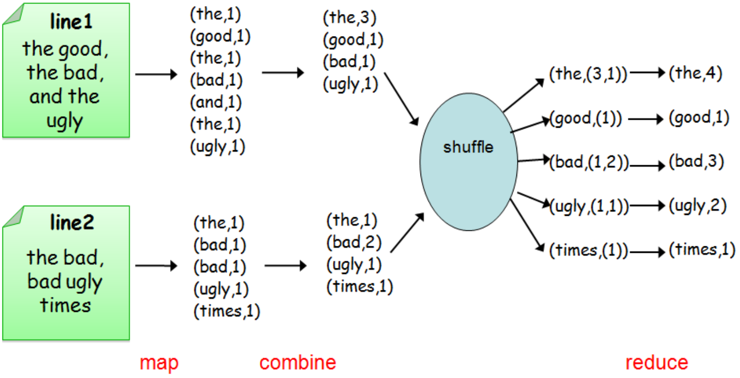
An Optimization: Combiner

- `map` outputs pairs `(key,value)`
- `reduce` receives pair `(key,list-of-values)`
- `combiner(key,list-of-values)` is applied to mapper output, *before* shuffling
- may help sending much less information
- must be associative and commutative

Example 1: Word Count, revisited

```
map(line):
    foreach word in line.split() do
        output (word,1)

combine(word,L):
    output (word,sum(L))

reduce(word,L):
    output (word,sum(L))
```

Example 1: Word Count,revisited



Example 4: Inverted Index

Input: A set of text files
Output: For each word, the list of files that contain it

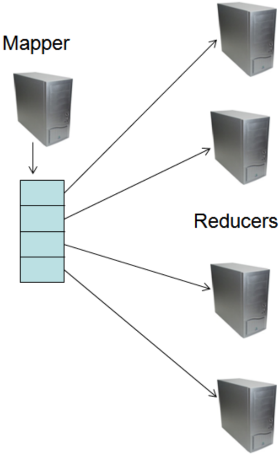
```
map(filename):
    foreach word in the file text do
        output (word, filename)

combine(word,L):
    remove duplicates in L;
    output (word,L)

reduce(word,L):
    //want sorted posting lists
    output (word,sort(L))
```

This replaces all the barrel stuff we saw in the last session

Can also keep pairs (filename,frequency)

Implementation, more

- A mapper writes to local disk
- In fact, makes as many partitions as reducers
- Keys are distributed to partitions by `Partition` function
- By default, hash
- Can be user defined too



Example 5. Sorting

Input: A set S of elements of a type T with a $<$ relation
Output: The set S, sorted

1. `map(x): output x`
2. `Partition`: any such that `k` $<$ `k'` $\rightarrow$ `Partition(k)` $\leq$ `Partition(k')`
3. Now each reducer gets an interval of T according to $<$ (e.g., 'A'..'F', 'G'..'M', 'N'..'S','T'..'Z')
4. Each reducer sorts its list

Note: In fact Hadoop guarantees that the list sent to each reducer is sorted by key, so step 4 may not be needed

Implementation, even more

- A user submits a `job` or a sequence of jobs
- User submits a class implementing map, reduce, combiner, partitioner, ...
- ... plus several configuration files (machines & roles, clusters, file system, permissions...)
- Input partitioned into equal size splits, one per mapper
- A running jobs consists of a *jobtracker* process and *tasktracker* processes
- Jobtracker orchestrates everything
- Tasktrackers execute either `map` or `reduce` instances
- `map` executed on each record of each split
- Number of reducers specified by users

Implementation, even more

```
public class C {

  static class CMapper
     extends Mapper<KeyType,ValueType> {
     ....
     public void map(KeyType k, ValueType v, Context context)
         .... code of map function ...
           ... context.write(k',v');
    }

  static class CReducer
    extends Reducer<KeyType,ValueType> {
    ....
    public void reduce(KeyType k, Iterable<ValueType> values,
               Context context) {
         .... code of reduce function ...
           .... context.write(k',v');
    }
  }

}
```

Example 6: Entropy of a distribution

Input: A multiset S
Output: The entropy of S:

$$H(S) = \sum_i p_i \log(1/p_i), \text{ where } p_i = \#(S,i)/\#S$$

Job 1: For each i, compute p_i:

- `map(i): output (i,1)`
- `combiner(i,L) = reduce(i,L):`
  ```
  output (i,sum(L))
  ```

Job 2: Given a vector p, compute $H(p)$:

- `map(p(i)): output (0,p(i))`
- `combiner(k,L) = reduce(k,L) :`
  ```
  output sum( p(i)*log(1/p(i)) )
  ```

Mapreduce/Hadoop: Conclusion

- De-facto standard for open-source big data distributed processing
  - Though far from universally optimal solution. Think twice.
- Abstracts from cluster details
- Missing features can be externally added
  - Data storage and retrieval components (e.g. HDFS in Hadoop), scripting languages, workflow management, SQL-like languages. . . enditemize

Cons:

- Complex to setup, lengthy to program
- Input and output of each job goes to disk (e.g. HDFS); slow
- No support for online, streaming processing
- Often, performance bottlenecks

Big Data and NoSQL: Outline

1. Big Data
2. NoSQL: Generalities
3. NoSQL: Some Systems
4. Key-value DB's: Dynamo and Cassandra
5. A document-oriented DB: MongoDB
6. The Apache ecosystem for Big Data

Big Data

- Sets of data whose size surpasses what data storage tools can typically handle
- Figure that grows concurrently with technology
- The problem has always existed
- In fact, it has always driven innovation

Big Data

Yes, but:

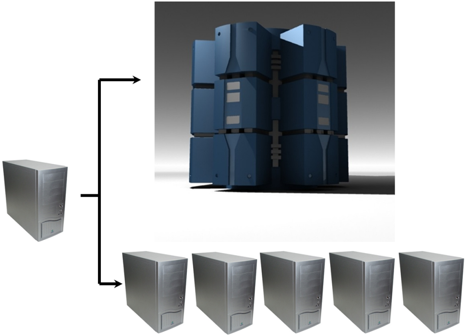
- Planet-scale applications do exist today
  - Tb of data per day
  - 2 billion Internet users
- 5 billion cellphones
- Internet of things, sensorized environments
- Open Data initiatives (science, government)
- The Cloud
- ...

Big Data

- Technological problem: how to store, use & analyze
- Business problem:
  - what to look for in the data?
  - how to model the data?
  - where to start???

The problem with Relational DBs

- The relational DB has ruled for 2-3 decades
- Superb capabilities, superb implementations
- One of the ingredients of the web revolution
  - LAMP = Linux + Apache HTTP server + MySQL + PHP
- Main problem: scalability



Scaling UP

- Price superlinear in performance
- Performance ceiling

Scaling OUT

- No performance ceiling, but
- More complex management
- More complex programming
- Problems keeping ACID properties

The problem with Relational DBs

- RDBMS scale *up* well (single node). Don't scale *out* well
- Vertical partitioning: Different tables in different servers
- Horizontal partitioning: Rows of same table in different servers

Apparent solution: Replication and caches

- Good for fault-tolerance, for sure
- OK for many concurrent reads
- Not much help with writes, if we want to keep ACID

There's a reason: The CAP theorem

Three desirable properties:

- Consistency: After an update to the object, every access to the object will return the updated value
- Availability: At all times, all DB clients are able to access *some* version of the data. Equivalently, every request receives an answer
- Partition tolerance: The DB is split over multiple servers communicating over a network. Messages among nodes may be lost arbitrarily

The CAP theorem [Brewer 00, Gilbert-Lynch 02] says:

No distributed system can have these three properties

In other words: In a system made up of nonreliable nodes and network, it is impossible to implement atomic reads & writes and ensure that every request has an answer.

CAP theorem: Proof

- Two nodes, A, B
- A gets request “read(x)”
- To be consistent, A must check whether some “write(x,value)” performed on B
- …so sends a message to B
- If A doesn’t hear from B, either A answers (inconsistently)
- or else A does not answer (not available)

The problem with RDBMS

- A truly distributed, truly relational DBMS should have Consistency, Availability, and Partition Tolerance
- …which is impossible

- The NoSQL trend obtains scalability by *not aiming* at all three C, A, P
- Ensuring at least two: Often, A+P; sometimes, C+P
- Taking into account that none of C, A, P is none-or-all

NoSQL: Generalities

Properties of most NoSQL DB's:

1. BASE instead of ACID
2. Simple queries. No joins
3. No schema
4. Decentralized, partitioned (even multi data center)
5. Linearly scalable using commodity hardware
6. Fault tolerance
7. Not for online (complex) transaction processing
8. Not for datawarehousing

NOSQL: Generalities

Not without critics
An example (Mapreduce, but most apply to NoSQL):

- No schemas? No data modelling process!
- Not novel. Represents implementation of well known techniques developed 25 years ago
- Missing common DBMS utilities and features: Transactions, updates, integrity constraints, views, ...
- Incompatible with DBMS tools
- Brute force
- ...

MapReduce-A major step backwards, D. DeWitt and M. Stonebraker (2008)
(`http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/`)

BASE, eventual consistency

- Basically Available, Soft state, Eventual consistency
- Eventual consistency: If no new updates are made to an object, eventually all accesses will return the last updated value.
- ACID is pessimistic. BASE is optimistic. Accepts that DB consistency will be in a state of flux
- Surprisingly, OK with many applications
- And allows *far* more scalability than ACID

Some names, by Data Model

Table: BigTable, Hbase, Hypertable

Key-Value: Dynamo, Riak, Voldemort, Cassandra, CouchBase, Redis

Column-Oriented: Cassandra

Document: MongoDB, CouchDB

Graph Oriented: Neo4j, Sparksee (formerly DEX), Pregel, FlockDB

Some names, by CAP properties

- Consistency + Partitioning
  BigTable, Hypertable, Hbase, Redis
- Availability + Partionining
  Dynamo, Voldemort, Cassandra, Riak, MongoDB, CouchDB

Some names, by data size

RAM-based: CouchBase, Qlikview

Big Data: MongoDB, Neo4j, Hypergraph, Redis, Couchdb

BIG DATA: BigTable, Hbase, Riak, Voldemort, Cassandra, Hypertable

Some names, by genealogy & language

- Google's BigTable (C++): HBase (Java), Hypertable (C++)
- Amazon's Dynamo (?): Cassandra (Java), Riak (Erlang), Voldemort (Java)
- CouchDB (Erlang), MongoDB (C++)

Dynamo

- Amazon's propietary system
- Very influential: Voldemort, Riak, Cassandra
- Goal: system where ALL customers have a good experience, not just the majority
- I.e., very high availability

Dynamo

- Queries: simple objects reads and writes
- Objects: unique key + binary object (blob)
- Key implementation idea: Distributed Hash Tables (DHT)
- Client tunable tradeoff latency vs. consistency vs. durability

Dynamo

Interesting feature:

- In most rdbms, conflicts resolved at write time, so read remains simple.
- That's why lock before write. "Syntactic" resolution
- In Dynamo, conflict resolution at reads – "semantic" – solved by client with business logic

Example:

- Client gets several versions of end-user's shopping cart
- Knowing their business, decides to merge; no item ever added to cart is lost, but deleted items may reappear
- Final purchase we want to do in full consistency

Cassandra

- Key-value pairs, like Dynamo, Riak, Voldemort
- But also richer data model: Columns and Supercolumns
- Write-optimized

  Choice if you write more than you read, such as logging

A document-oriented DB: MongoDB

- Richer data model than most NoSQL DB's
- More flexible queries than most NoSQL DB's
- No schemas, allowing for dynamically changing data
- Indexing
- MapReduce & other aggregations
- Stored JavaScript functions on server side
- Automatic sharding and load balancing
- Javascript shell

MongoDB Data model

- Document: Set of key-value pairs and embedded documents
- Collection: Group of documents
- Database: A set of collections + permissions + ...

Relational analogy:

Collection = table; Document = row

Example Document

```
{
    "name" : "Anna Rose",
    "profession" : "lawyer",
    "address" : {
        "street" : "Champs Elisees 652",
        "city" : "Paris",
        "country" : "France"
    }
}
```

Always an extra field `_id` with unique value

Managing documents: Examples

```
> anna = db.people.findOne({ "name" : "Anna Rose" });
> anna.age = 25
> anna.address = { "Corrientes 348", "city" :
                   "Buenos Aires", "country" : "Argentina" }
>
> db.people.insert({ "name" : "Gilles Oiseau", "age" : 30 })
> ...
> db.people.update({ "name" : "Gilles Oiseau"},
                     $set : { "age" : 31 })
>
> db.people.update({ "name" : "Gabor Kun" },
                     $set : { "age" : 18 }, true)
```

Last parameter `true` indicates *upsert*:
update if it alredy exists, insert if it doesn't

`find`

- `db.find(condition)` returns a collection
- `condition` may contain boolean combinations of key-value pairs,
- also =, <, >, `$where`, `$group`, `$sort`, ...

Common queries can be sped-up by creating indices
Geospatial indices built-in

Consistency

- By default, all operations are "fire-and-forget": client does not wait until finished
- Allows for very fast reads and writes
- Price: possible inconsistencies

- Operations can be made *safe*: wait until completed
- Price: client slowdown

Sharding

- With a shard key, a user tells how to split DB into shards
- E.g. `"name"` as a shard key may split `db.people` into 3 shards A-G, H-R, S-Z, sent to 3 machines
- Random shard keys good idea

- Shards themselves may vary over time to balance load
- E.g., if many A's arrive the above may turn into A-D, E-P, Q-Z

The Apache ecosystem for Big Data

Lucene, text search system

Tomcat, web server

Hadoop, mapreduce platform

The Apache ecosystem for Big Data

Solr, text search on Lucene+hadoop
Can run as a Tomcat Servlet

ElasticSearch, text search on Lucene+hadoop

The Apache ecosystem for Big Data

Cassandra, write-optimized key-value noSQL DB

HBase, Hadoop-based BigTable NoSQL DB
Handles billion rows x million column tables

Pig, Hadoop scripting language

Hive, SQL-like language over Hadoop

The Apache ecosystem for Big Data

Nutch: crawler + web search system

"Relatively feature-rich crawler,
polite (obeys robots.txt rules), robust, and highly scalable:
- you can run Nutch on a cluster of 100 machines
- you can bias the crawling to fetch "important" pages first "

The Apache ecosystem for Big Data

Mahout: scalable machine learning

Many algorithms parallelized on top of Hadoop

k-means, frequent pattern mining, random forests, collaborative filtering, latent Dirichlet allocation, regression, perceptron, SVM, boosting, EM, PCA, SVD, ...

The Apache ecosystem for Big Data

Online, real-time:

Samza: Streaming, distributed processing

samza

Kafka: Massive scale message ditributing systems

Spark: In-memory, interactive, real-time

Mlib: Apache Spark's scalable machine learning library
(Scala, Java, Python, R)