Exam Preparation

Guido Salvaneschi

Lecture Material

Lectures

- Intro to dist. systems
- MapReduce
- HDFS
- Hive, HBase, Yarn
- Futures, Promises, Actors
- Spark
- Spark streaming

Exercises

- MapReduce
- Futures, Actors
- Spark

Papers

- MapReduce
- GFS
- Spark

Warning!

- These are just examples of the kind of questions that can appear in the exam.
- They are not supposed to be complete (of course).
- They are not representative of the coverage of the course topics in the exam.
- They do not cover questions about coding
  (but “simple” exercises provide good examples for that).

Explain 3 reasons that motivate building a system in a distributed way

Why Distributed Systems

- Functional distribution
  - Computers have different functional capabilities (e.g., File server, printer) yet may need to share resources
    - Client / server
    - Data gathering / data processing

- Incremental growth
  - Easier to evolve the system
  - Modular expandability

- Inherent distribution in application domain
  - Banks, reservation services, distributed games, mobile apps
  - physically or across administrative domains
  - cash register and inventory systems for supermarket chains
  - computer supported collaborative work

Why Distributed Systems

- Economics
  - collections of microprocessors offer a better price/ performance ratio than large mainframes.
  - Low price/performance ratio: cost effective way to increase computing power.
- Better performance
  - Load balancing
  - Replication of processing power
  - A distributed system may have more total computing power than a mainframe. Ex. 10,000 CPU chips, each running at 50 MIPS. Not possible to build 500,000 MIPS single processor since it would require 0.002 nsec instruction cycle. Enhanced performance through load distributing.
- Increased Reliability
  - Exploit independent failures property
  - If one machine crashes, the system as a whole can still survive.
- Another driving force: the existence of large number of personal computers, the need for people to collaborate and share information.

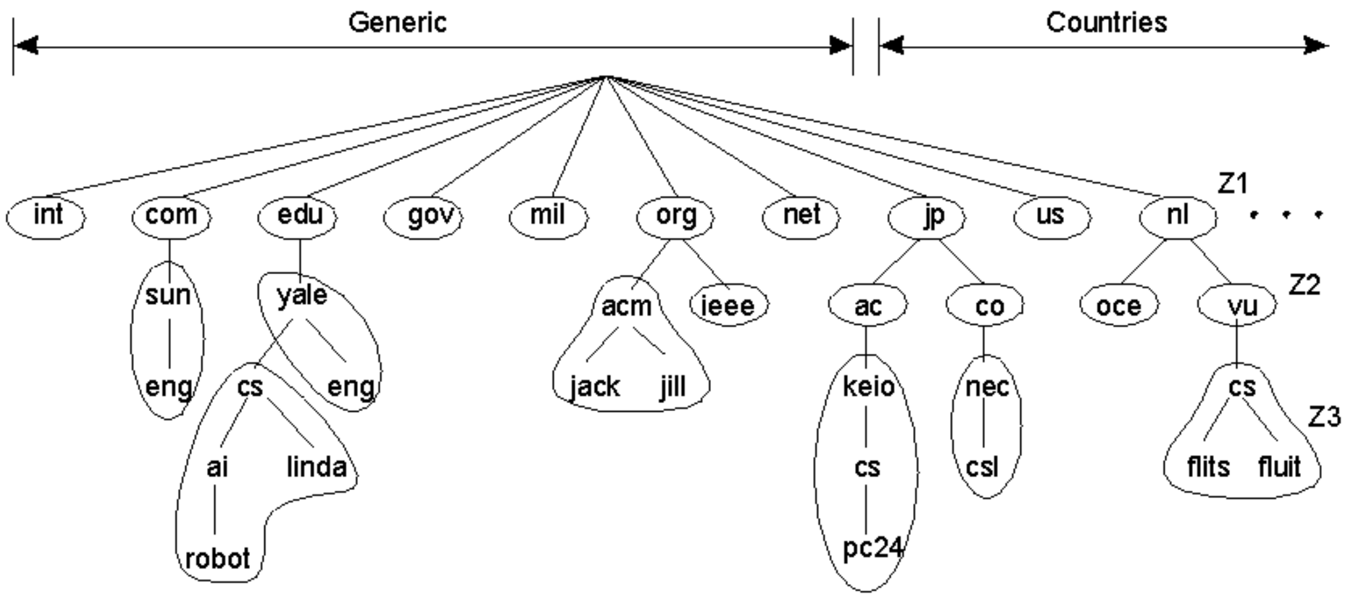
Explain 3 goals (and challenges) of distributed systems

Goals and challenges of distributed systems

- Transparency
  - How to achieve the single-system image
- Performance
  - The system provides high (computing, storage, ..) performance
- Scalability
  - The ability to serve more users, provide acceptable response times with increased amount of data
- Openness
  - An open distributed system can be extended and improved incrementally
  - Requires publication of component interfaces and standards protocols for accessing interfaces
- Reliability / fault tolerance
  - Maintain availability even when individual components fail
- Heterogeneity
  - Network, hardware, operating system, programming languages, different developers
- Security
  - Confidentiality, integrity and availability

Which techniques can be used to make a system scalable? Briefly explain them.

Scaling techniques



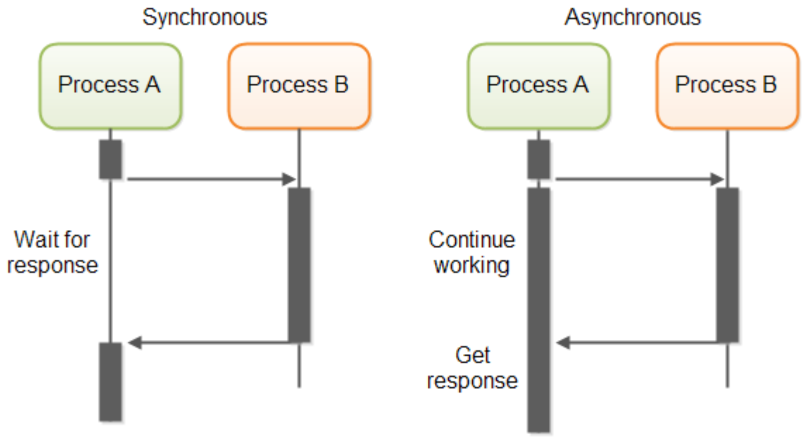
Distribution

- Splitting a resource (such as data) into smaller parts, and spreading the parts across the system (cf DNS)

Scaling techniques

- Replication
  - Replicate resources (services, data) across the system, can access them in multiple places
  - Caching to avoid recomputation
  - Increased availability reduces the probability that a bigger system breaks

- Hiding communication latencies
  - Avoid waiting for responses to remote service requests
  - Use asynchronous communication

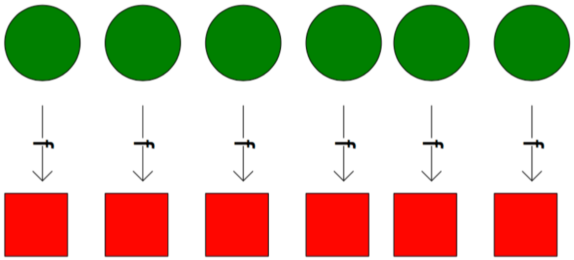


Show the signature of the Map function and the Reduce function in MapReduce.
What is the Map phase and what are the Reduce phase responsible for?

Functional programming "foundations"

Note: There is no precise 1-1 correspondence. Please take this just as an analogy.

- map in MapReduce ↔ map in FP
  - map::(a→b)→[a]→[b]
  - Example: Double all numbers in a list.
  - > map ((*) 2) [1, 2, 3]
    > [2, 4, 6]



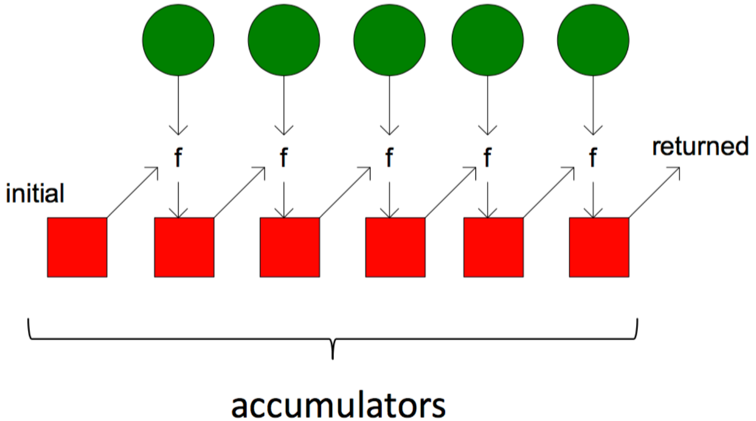
- In a purely functional setting, an element of a list being computed by map **cannot see the effects** of the computations on other elements.
- If the order of application of a function f to elements in a list is commutative, then we can **reorder or parallelize** execution.

Note: There is no precise 1-1 correspondence. Please take this just as an analogy.

Functional programming "foundations"

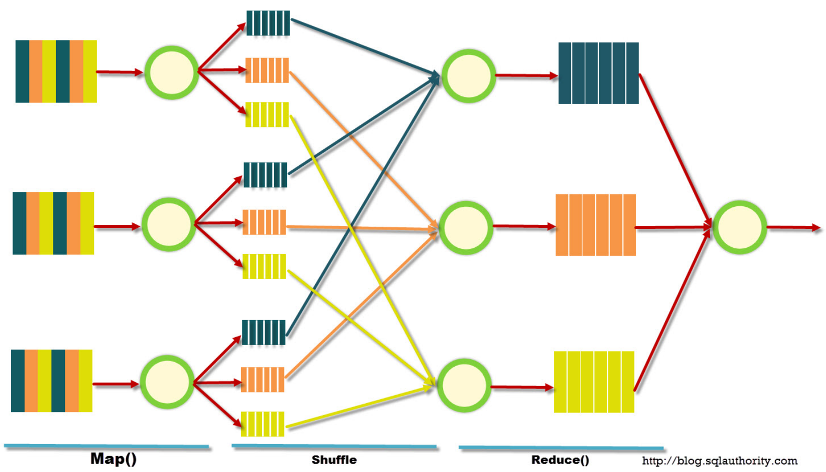
- Move over the list, apply **f** to each element and an **accumulator**. f returns the next accumulator value, which is combined with the next element.

- reduce in MapReduce ↔ fold in FP
  - foldl :: (b → a → b) → b → [a] → b
  - Example: Sum of all numbers in a list.
  - > foldl (+) 0 [1, 2, 3] foldl (+) 0 [1, 2, 3]
    > 6



MapReduce Basic Programming Model

- Transform a set of input key-value pairs to a set of output values:
  - Map: (k1, v1) → list(k2, v2)
  - MapReduce library groups all intermediate pairs with same key together.
  - Reduce: (k2, list(v2)) → list(v2)



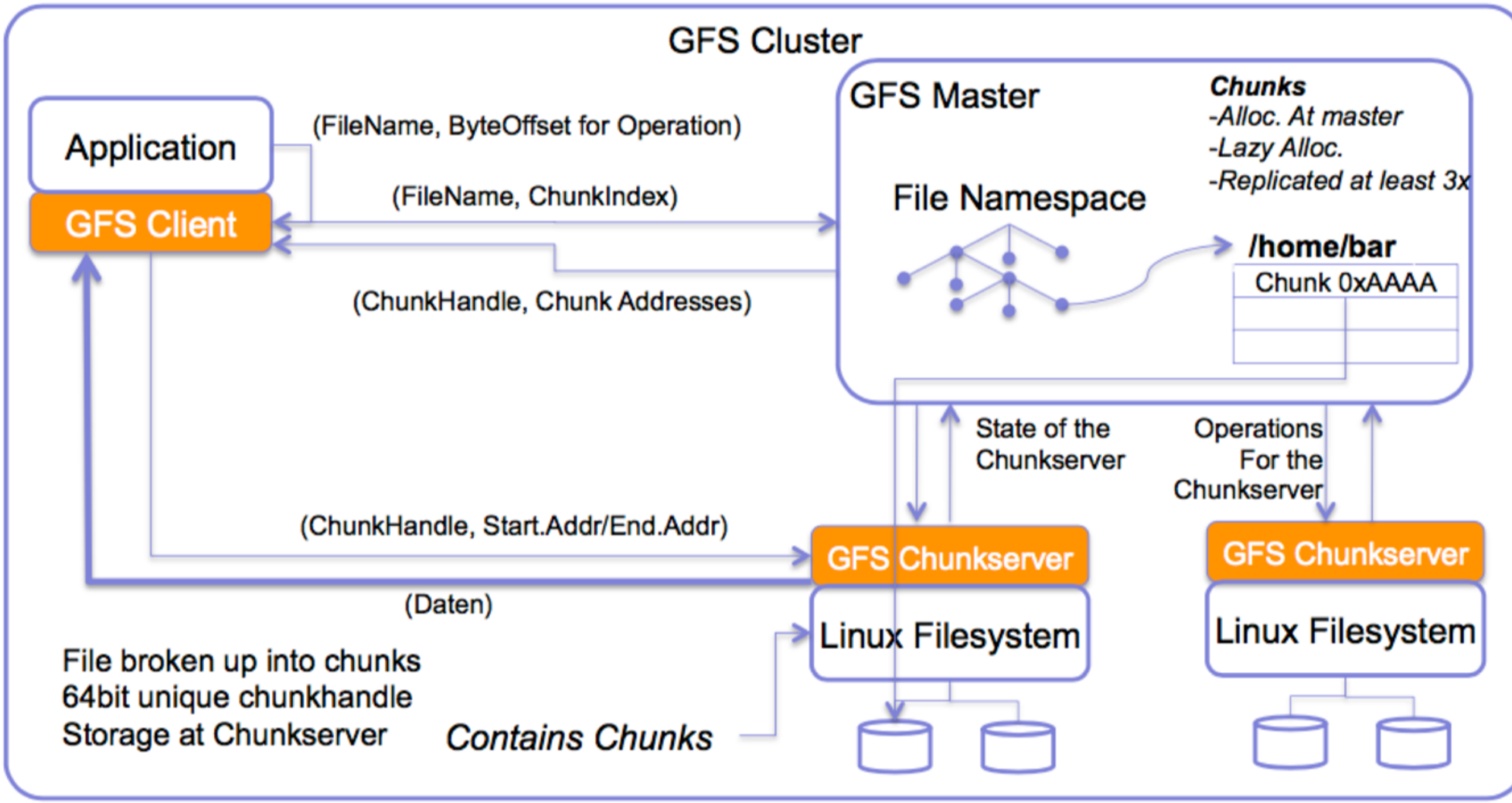
What is the problem with “stragglers” (slow workers) and what can be done to solve this problem?

Stragglers & Backup Tasks

- Problem: “Stragglers” (i.e., slow workers) significantly lengthen the completion time.
- Solution: Close to completion, spawn backup copies of the remaining in-progress tasks.
  - Whichever one finishes first, “wins”.
- Additional cost: a few percent more resource usage.
- Example: A sort program without backup = 44% longer.

Sketch the GFS architecture presenting the components that constitutes it and the main interactions.

GFS - Overview



Explain what a future is

Explain what a future is

- Placeholder object for a value that may not yet exist
- The value of the Future is supplied concurrently and can subsequently be used

Which underlying data structure is used by Apache Spark? Show a *minimal* example and indicate where such data structure is used.

RDD (Resilient Distributed Datasets)

- Restricted form of distributed shared memory
  - immutable, partitioned collection of records
  - can only be built through coarse-grained deterministic transformations (map, filter, join...)

- Efficient fault-tolerance using lineage
  - Log coarse-grained operations instead of fine-grained data updates
  - An RDD has enough information about how it’s derived from other dataset
  - Recompute lost partitions on failure

Spark and RDDs

- Implements Resilient Distributed Datasets (RDDs)

- Operations on RDDs
  - **Transformations**: defines new dataset based on previous ones
  - **Actions**: starts a job to execute on cluster

- Well-designed interface to represent RDDs
  - Makes it very easy to implement transformations
  - Most Spark transformation implementation < 20 LoC

Operation	**Meaning**
partitions()	Return a list of Partition objects
preferredLocations(*p*)	List nodes where partition *p* can be accessed faster due to data locality
dependencies()	Return a list of dependencies
iterator(*p, parentIters*)	Compute the elements of partition *p* given iterators for its parent partitions
partitioner()	Return metadata specifying whether the RDD is hash/range partitioned

Table 3: Interface used to represent RDDs in Spark.

More on RDDs

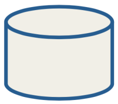
Work with distributed collections as you would with local ones

- Resilient distributed datasets (RDDs)
  - Immutable collections of objects spread across a cluster
  - Built through parallel transformations (map, filter, etc)
  - Automatically rebuilt on failure
  - Controllable persistence (e.g., caching in RAM)
    - Different storage levels available, fallback to disk possible

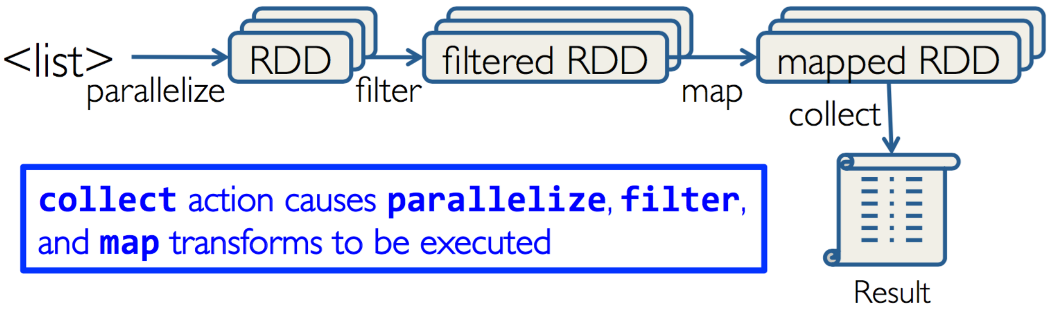
- Operations
  - **Transformations** (e.g. map, filter, groupBy, join)
    - Lazy operations to build RDDs from other RDDs
  - **Actions** (e.g. count, collect, save)
    - Return a result or write it to storage

Workflow with RDDs

- Create an RDD from a data source: <list>



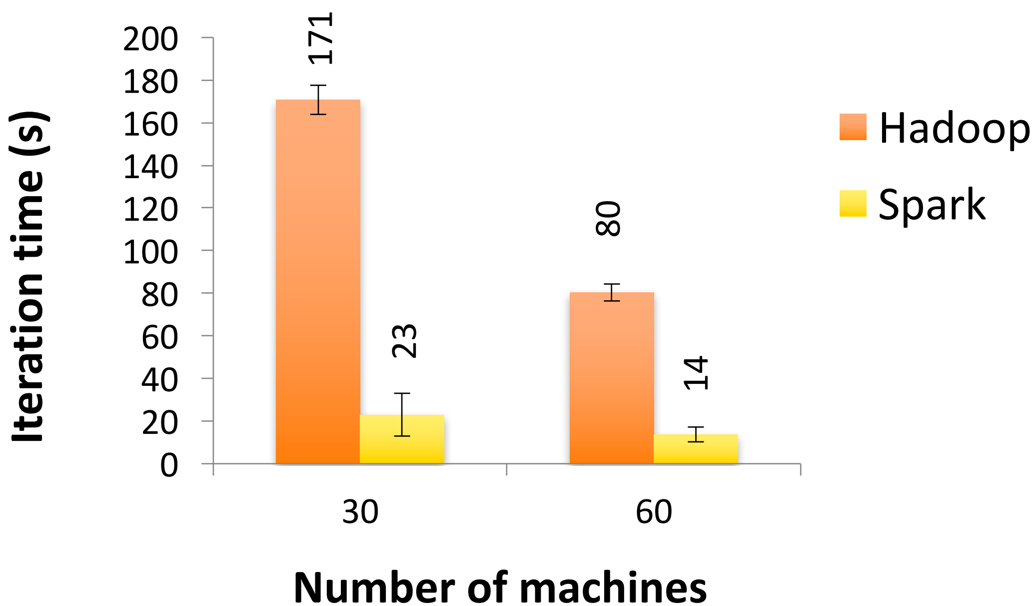
- Apply transformations to an RDD: map filter
- Apply actions to an RDD: collect count




```
distFile = sc.textFile("...", 4)
```

- RDD distributed in 4 partitions
- Elements are lines of input
- *Lazy evaluation* means no execution happens now

Give a possible explanation why the computation of the Page Rank is significantly different between Hadoop and Spark



Spark

- Fast, expressive cluster computing system compatible with Apache Hadoop
  - Works with any Hadoop-supported storage system (HDFS, S3, Avro, …)

- Improves **efficiency** through:
  - In-memory computing primitives
  - General computation graphs

→ Up to 100× faster

- Improves **usability** through:
  - Rich APIs in Java, **Scala**, Python
  - Interactive shell

→ Often 2-10× less code

Page Rank

- Give pages ranks (scores) based on links to them
  - Links from many pages ➔ high rank
  - Link from a high-rank page ➔ high rank

- Good example of a more complex algorithm
  - Multiple stages of map & reduce
- Benefits from Spark's in-memory caching
  - Multiple iterations over the same data

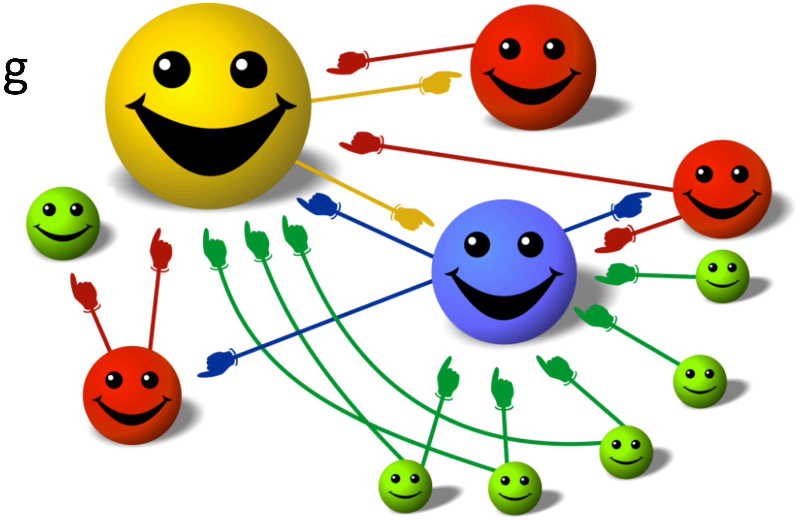
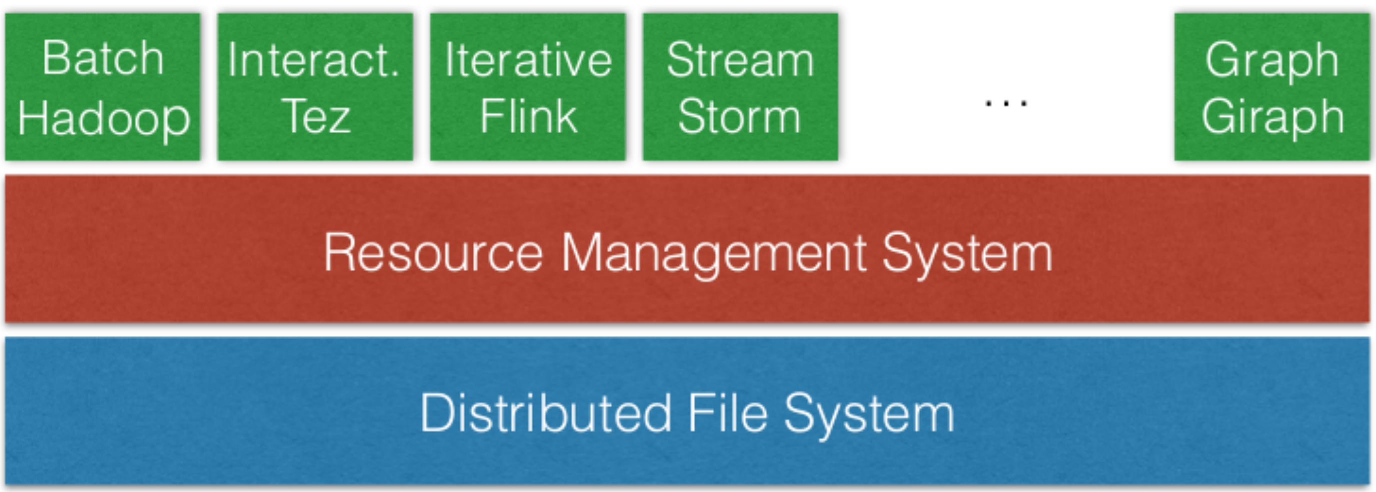


Image: en.wikipedia.org/wiki/File:PageRank-hi-res-2.png

What is a resource management system, e.g., Apache YARN?

Resource Management

- Typically implemented by a system deployed across nodes of a cluster
  - Layer below “frameworks” like Hadoop
  - On any node, the system keeps track of availabilities
  - Applications on top use information and estimations of own requirements to choose where to deploy something
    - RM systems (RMSs) differ in abstractions/interface provided and actual scheduling decisions



Given the scenario X, what is the technology/approach that you would recommend for solving problem Y ?

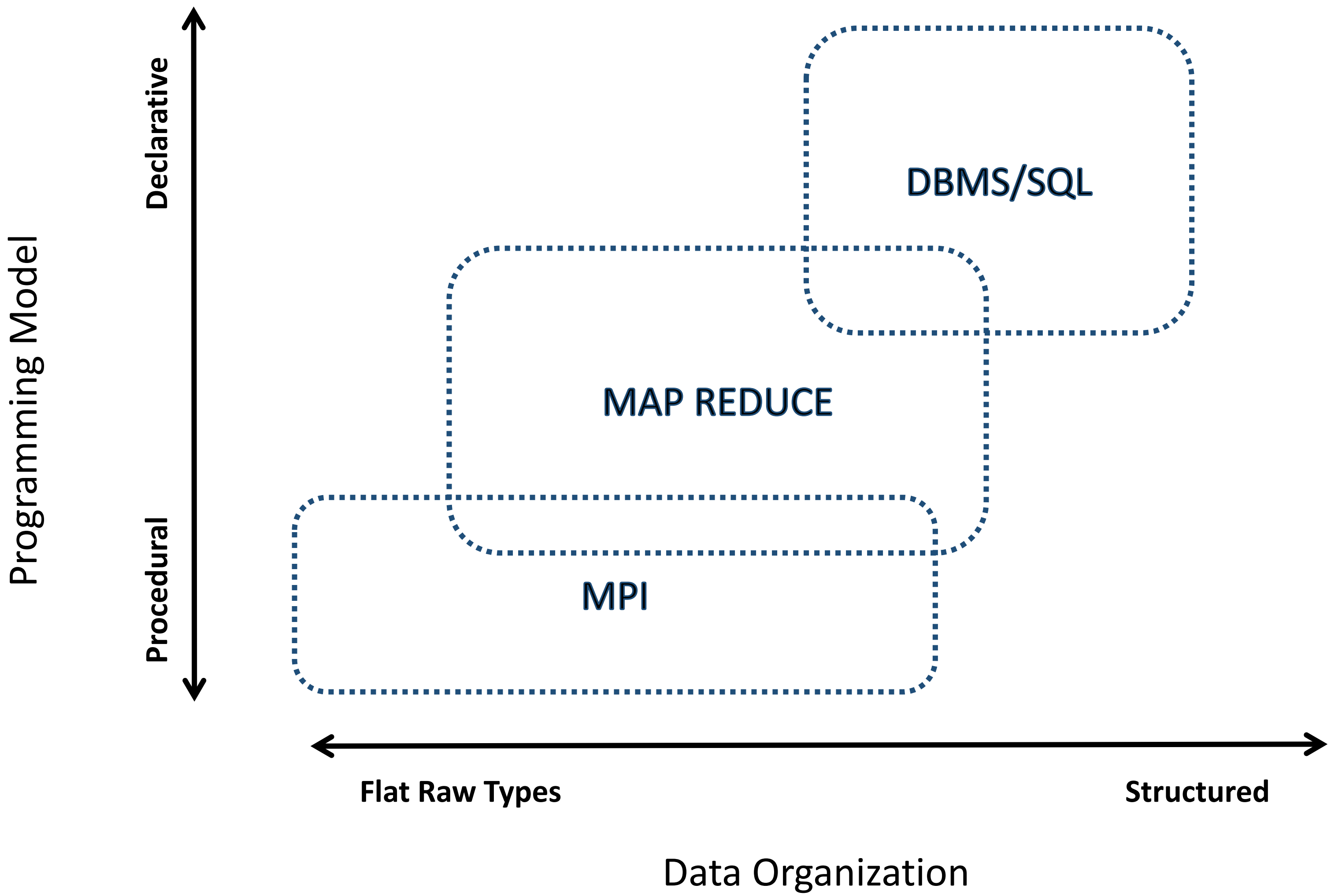
- MapReduce
- HDFS
- A database
- HBase
- Apache Spark
- Spark streaming
- ...

MapReduce vs. Traditional RDBMS

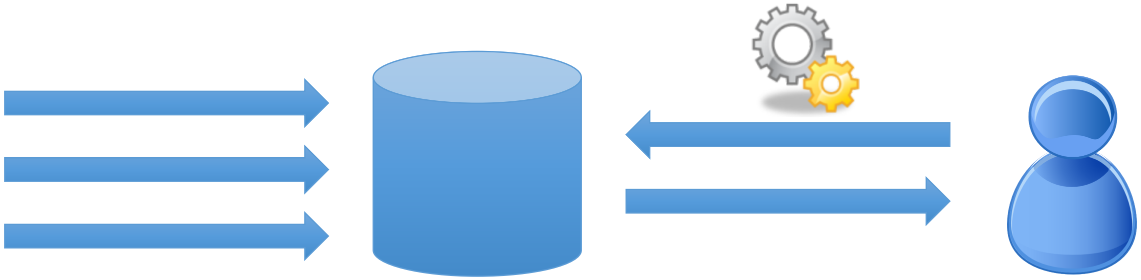
	MapReduce	Traditional RDBMS
Data size	Petabytes	Gigabytes
Access	Batch	Interactive and batch
Updates	Write once, read many times	Read and write many times
Structure	Dynamic schema	Static schema
Integrity	Low	High (normalized data)
Scaling	Linear	Non-linear (general SQL)

A Summary

	MPI	MapReduce	DBMS/SQL
What they are	A general parrellel programming paradigm	A programming paradigm and its associated execution system	A system to store, manipulate and serve data.
Programming Model	Messages passing between nodes	Restricted to Map/Reduce operations	Declarative on data query/retrieving; Stored procedures
Data organization	No assumption	"files" can be sharded	Organized datastructures
Data to be manipulated	Any	k,v pairs: string	Tables with rich types
Execution model	Nodes are independent	Map/Shuffle/Reduce Checkpointing/Backup Physical data locality	Transaction Query/operation optimization Materialized view
Usability	Steep learning curve*; difficult to debug	Simple concept Could be hard to optimize	Declarative interface; Could be hard to debug in runtime
Key selling point	Flexible to accommodate various applications	Plow through large amount of data with commodity hardware	Interactive querying the data; Maintain a consistent view across clients


Programming Model
Declarative
Procedural
DBMS/SQL
MAP REDUCE
MPI
Flat Raw Types
Structured
Data Organization

Event-driven applications



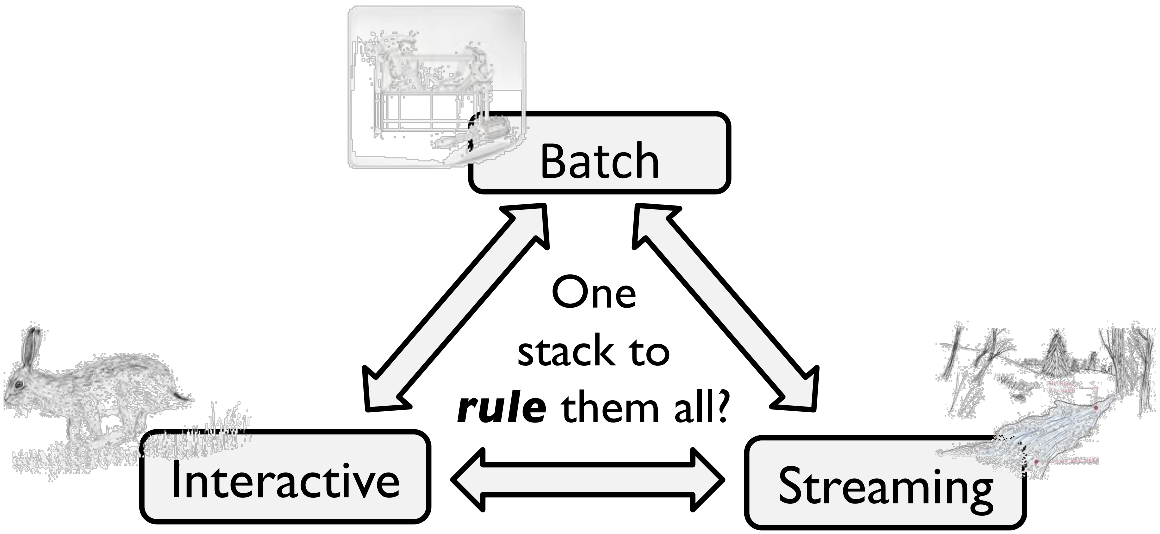
- Can we use existing technologies for batch processing?
  - They are not designed to minimize latency
  - We need a whole new model!

Esper in a nutshell

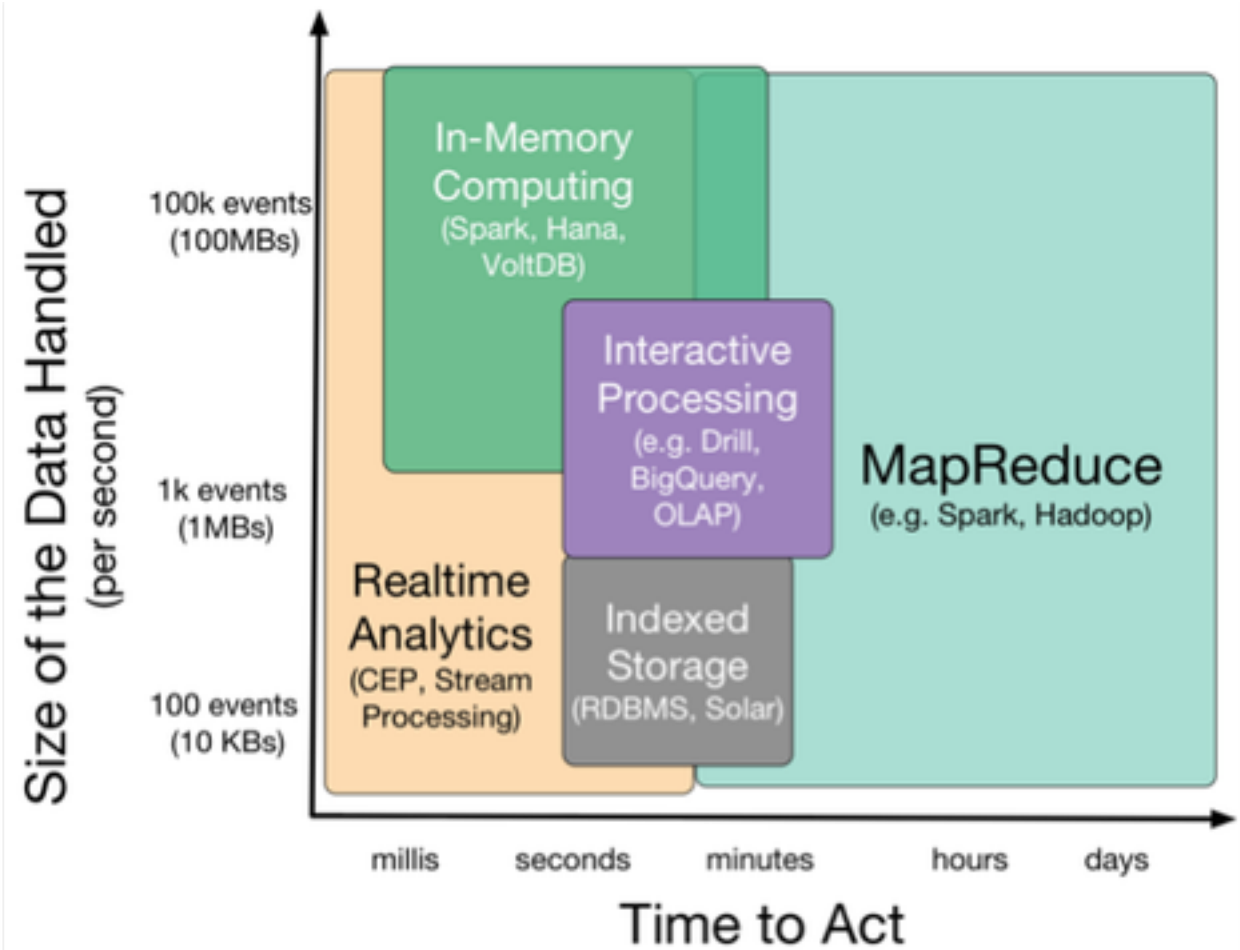
- EPL: rich language to express rules
  - Grounded on the DSMS approach
    - Windowing
    - Relational select, join, aggregate, …
    - Relation-to-stream operators to produce output
    - Sub-queries
  - Queries can be combined to form a graph
  - Introduces some features of CEP languages
    - Pattern detection

- Designed for performance
  - High throughput
  - Low latency

Goals



- ***Easy*** to combine ***batch***, ***streaming***, and ***interactiv***e computations
- ***Easy*** to develop ***sophisticated*** algorithms
- ***Compatible*** with existing open source ecosystem (Hadoop/HDFS)



Size of the Data Handled
(per second)
100k events
(100MBs)
1k events
(1MBs)
100 events
(10 KBs)
In-Memory
Computing
(Spark, Hana,
VoltDB)
Interactive
Processing
(e.g. Drill,
BigQuery,
OLAP)
MapReduce
(e.g. Spark, Hadoop)
Realtime
Analytics
(CEP, Stream
Processing)
Indexed
Storage
(RDBMS, Solar)
millis
seconds
minutes
hours
days
Time to Act