# Programming Models and System Architectures

## P2, 2020

Animesh Trivedi
a.trivedi@vu.nl

# What are Distributed Systems?

How do you define them?

What are their examples?

What are their interesting properties?

How do we use them?

What kind of machines do they run on?

https://www.confluent.io/learn/distributed-systems/

*Tell me what do you think a distributed system is?*

# Distributed systems are really …

Distributed

- Scale: Internet (planet-scale), country-wide infrastructure (e.g., DAS), Datacenter, Supercomputers
- Application domains: storage, processing (batch, graph, streaming, ML), messaging, resource managers
- Heterogenous: grid, cloud, supercomputers

*How can you come up with a single programming model and architecture?*

*Hint: There is no one size fits all*

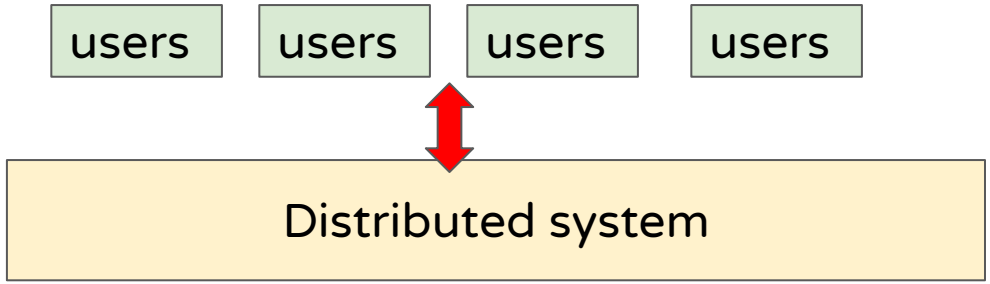
# For every distributed system

You can ask

- How is work parallelism and distribution achieved: how to define the work, who divides the work, who distributes it, bookkeeping …
- How is communication done between various entities: how are messages generated and when, to whom, delivered how…
- What happens when there is failure: detection, recovery, continuation …

*You need answers for these basic questions for a working distributed system*

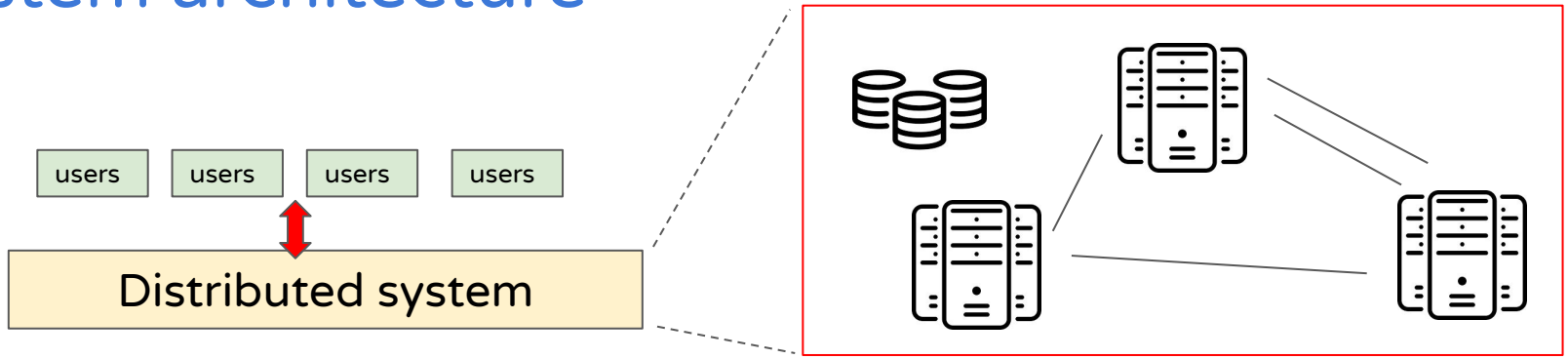
# What is a programming model




A **programming model** is **a contract** between a system and its users regarding work definition, distribution, communication, failure management, etc.

Often expressed in terms of APIs and abstractions provided by the system

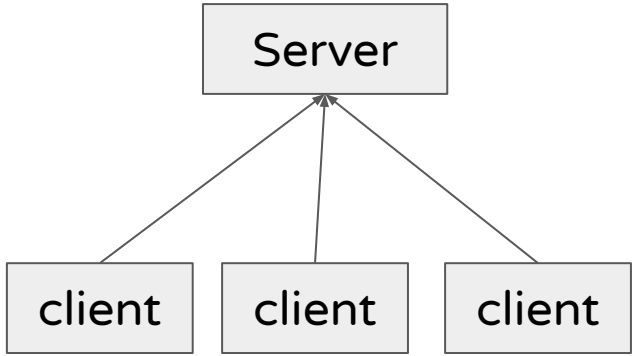
# System architecture



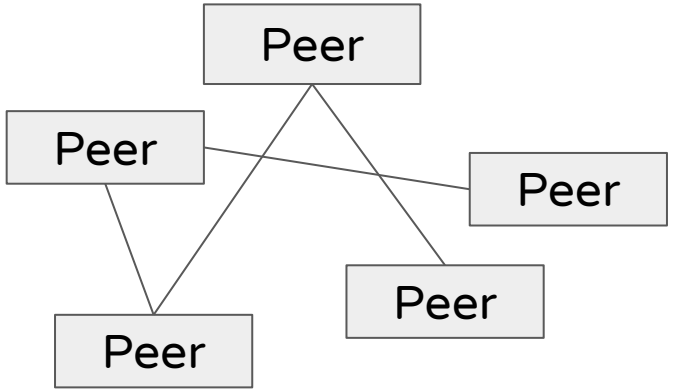

**System architecture:**

*how are resources arranged and connected to each other, what are the resources, how does execution look like, performance concerns, scheduling, resource management, security, etc.*

# System architecture




Peer = {server, client}*



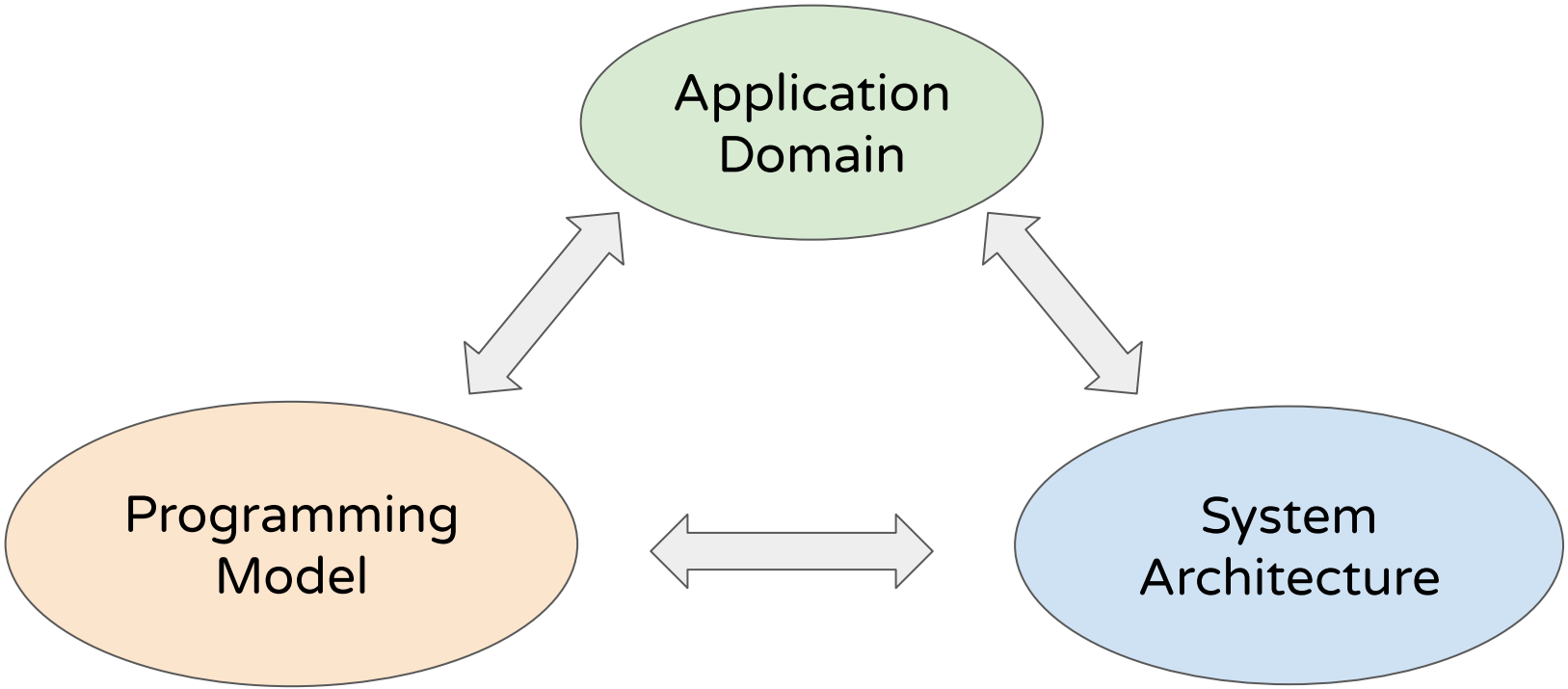

Serve-client or master-worker

+ Easy, simple to implement
- Scalability and single point of failure

Peer-to-Peer

+ Scalable, distributed state
- Difficult bookkeeping, QoS, performance

# Typically there are soft affinities (not just technical)



# Parallelism

Parallelism is an easy way to break and distribute work among multiple entities

- Task/process parallelism
- Data parallelism
- Event parallelism
- Object parallelism
- …

These different models define what is the basic “**abstraction**” (or unit) or work. Finer → more performance opportunities, but more overheads too!

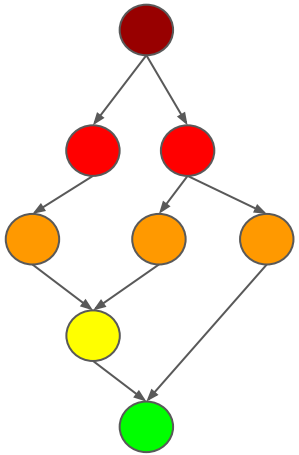
# Task and Data parallelism

**Task parallelism**

different vertex running on different machines

Each vertex represent different work (unit), and different input data

Common in multiprocessor programming

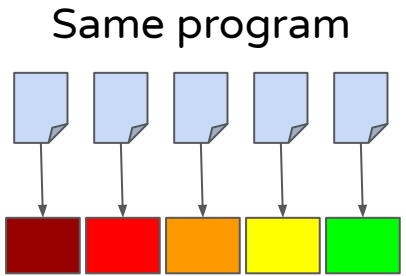


**Data parallelism**

Same program, different data

Data is chopped and distributed (unit)

Common in many data processing frameworks




Which to choose depends upon on the problem domain

# Fault tolerance

How to do detect failure? Slow machine vs failed machine

What state to keep track of?

If failed then


- Do nothing : simple and effective if work lost is not important
- Restart : just restart the failed work on another node
- Replication : keep states replicated
- Restore : checkpoint and restore from the last good state
- Lineage : The duality of compute and data
- …

# Distributed Storage

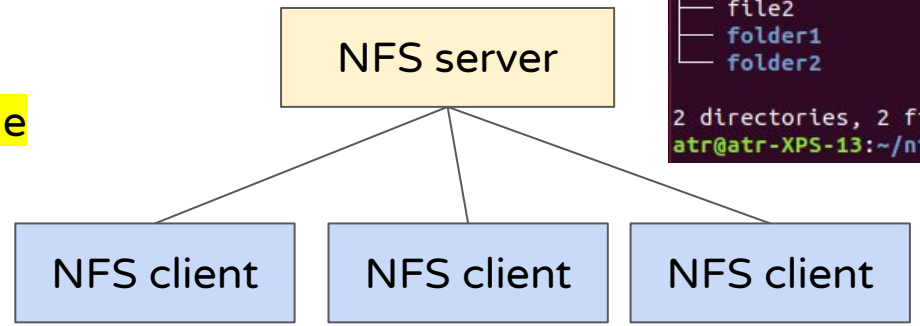
# Simple server storage

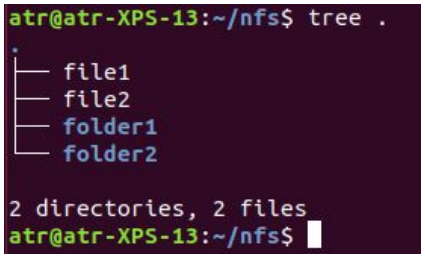
A **server** export storage resources for **clients** over the network (server-client model)

API example: Network file server (NFS), FTP servers, network block devices…

+ Very easy to deploy, immediately usable (files, or block disks)
- Availability, if server goes down OR if it is busy

Question: What happens if multiple clients update a file?



# Distributed storage

Distribute data over multiple servers

- Provides fault tolerance
- Storing entities in entirety (files, blocks, objects) on servers - example?
- Divide and distribute storage system specific “units” - example?

**Common concerns**:

- How to distribute data or how to look up data?
- How to keep data up-to date?
- How do servers communicate with each other?
- What happens in case of a failure?

# Chord: Structured peer-to-peer storage

Is a **structured** peer-to-peer system

There is a ring structure

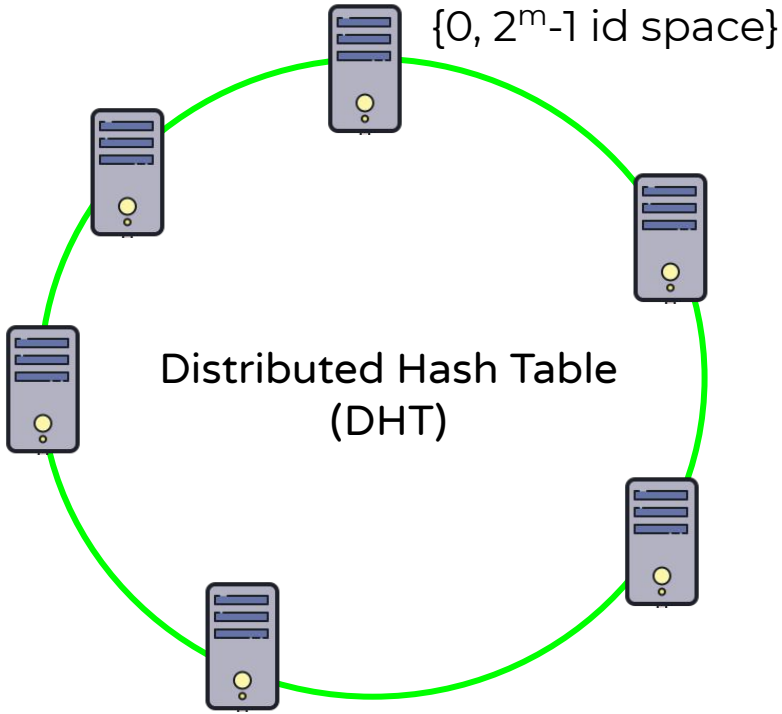
How did they got arranged in a ring?
using a consistent hashing function (SHA-1)

Data values are also hashed and mapped to the ring

**Programming model**: a simple get/put API

put(key, value)

value = get(key)




*Chord: A scalable peer-to-peer lookup service for internet applications.* ACM SIGCOMM 2001

# Chord: Structured peer-to-peer storage

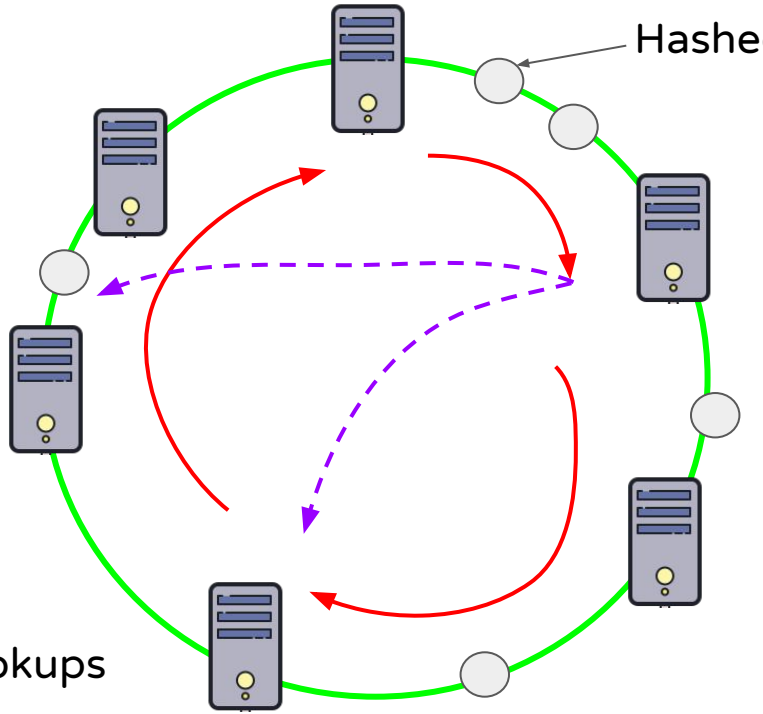
Each server maintains the successor and predecessor pointers

For a new value, the server following the hashed value on the ring is responsible for storing it

For a look up, a client can contact any server which will route the request in the ring by passing it to the successor pointer

Optimization: finger table with distant key lookups

Table [i] = my_id + $2^i$

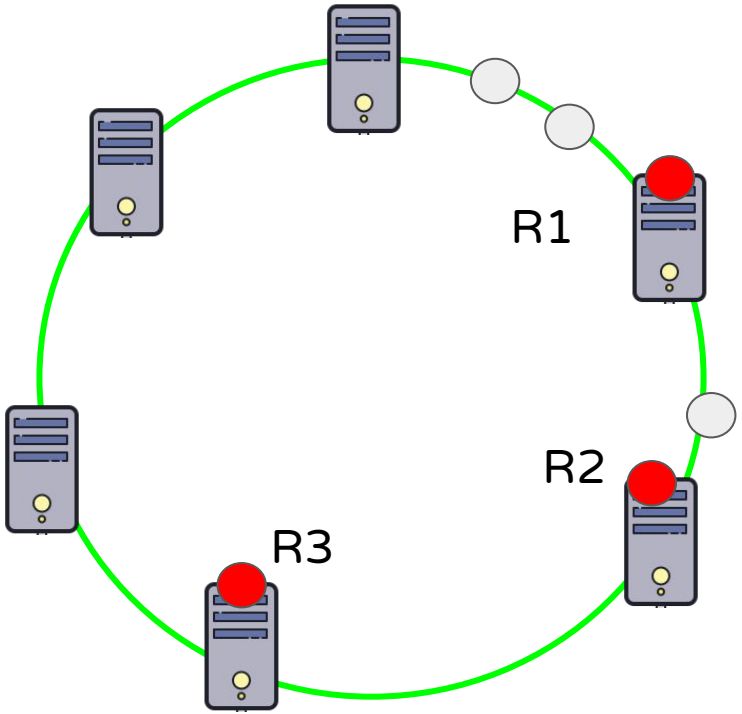


# Chord: Fault tolerance

Can use replication

Each value mapped to a server (X) is mapped to “r” following successors as well

If one fails, then others are used for providing service




A chord type system architecture has an affinity for large-scale decentralized storage (+ additional advantages for load-balancing, and fault tolerance - used in CDNs).

# LOCKSS: Unstructured peer-to-peer

Lots of Copies Keep Stuff Safe (https://www.lockss.org/)

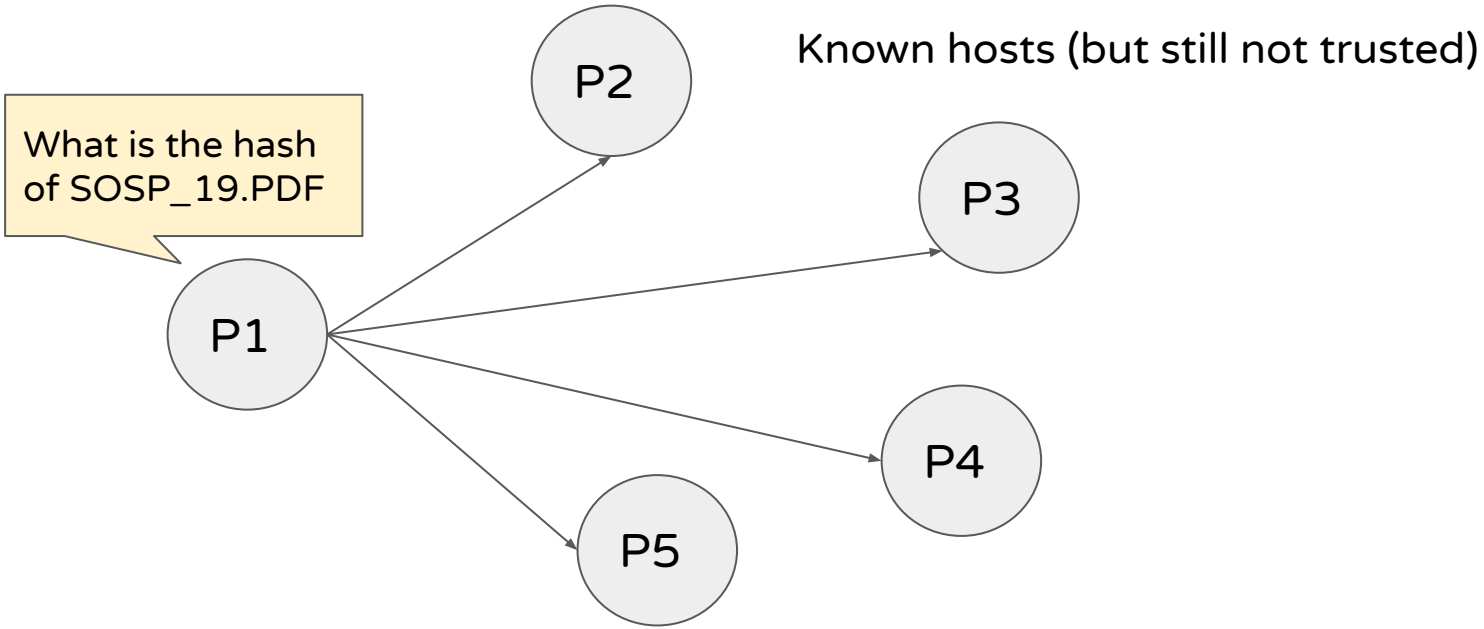
Assumes that everything (storage media, machines, users) will eventually have faults - only way to preserve **scientific data** is to make many many “authorized” copies

This is _NOT_ for general purpose, constantly changing storage: it is for long-term archival storage

Why can we do so?

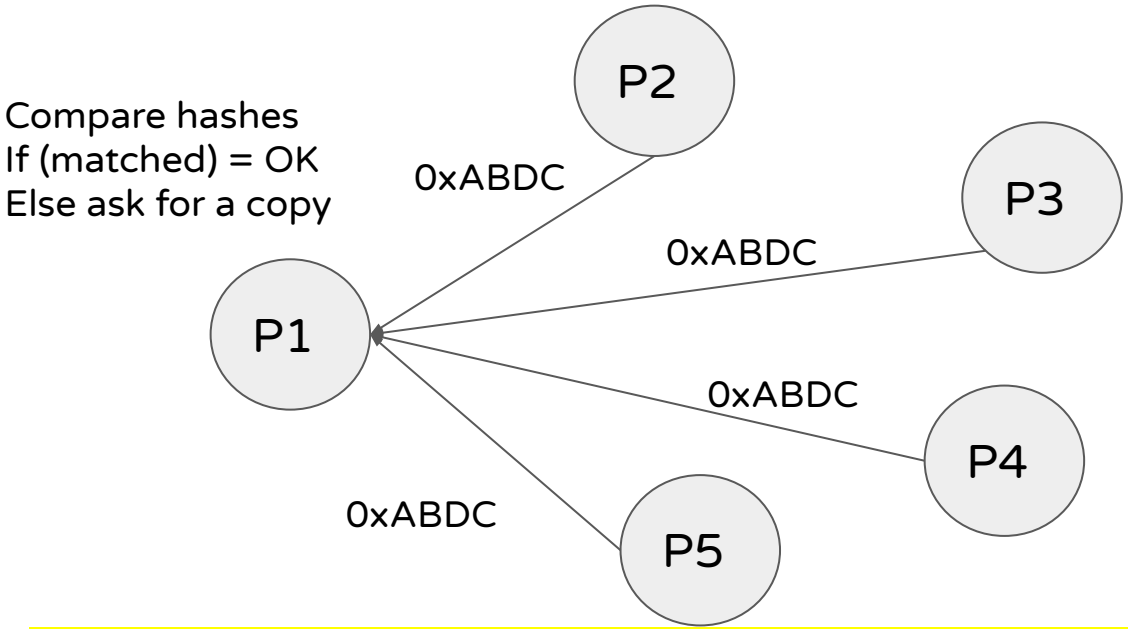
- Scientific data (archives, publications, books) are immutable
- Neatly documented  (catalog number, volume number, serial number)
- New releases at a regular frequency

# LOCKSS: Unstructured peer-to-peer




https://storageconference.us/2018/Presentations/LOCKSS-tutorial-3.pdf

# LOCKSS: Unstructured peer-to-peer



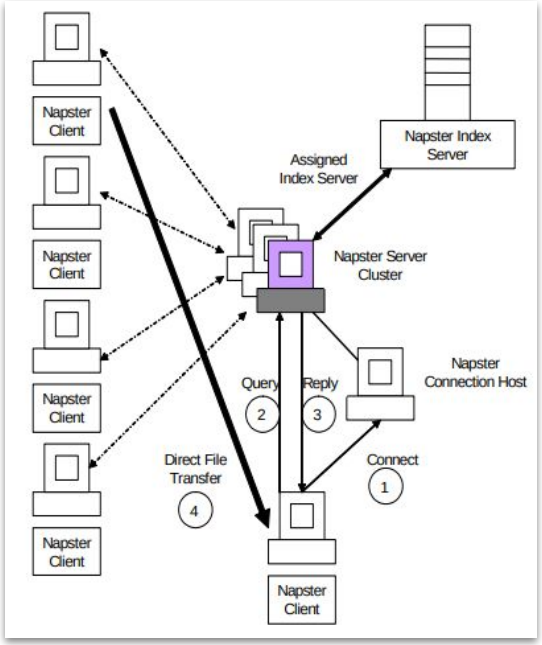

**Programming model**: saving whole files. LOCKSS has multiple frontends like objects, files, etc.

**System architecture**: no structure, general try to reach all (and run voting!)

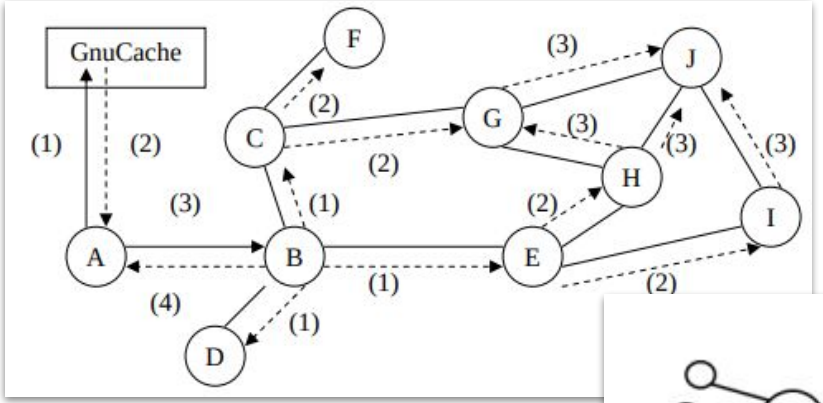
An internet-scale system like this is possible because unstructured peer-to-peer design with immutable data

https://storageconference.us/2018/Presentations/LOCKSS-tutorial-3.pdf

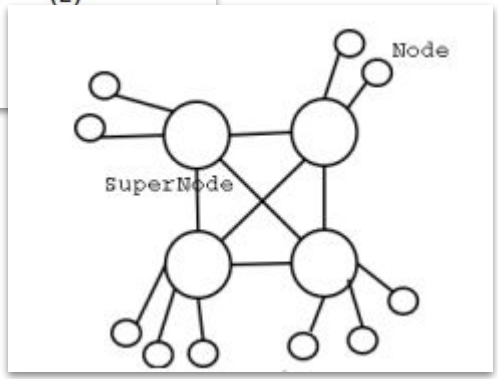
# File sharing systems: Napster, Gnutella, FastTrack




+ Centralized index, easy use
- Single point of failure (SPF)




+ No Single point of failure
- High-network load, no QoS




- Sensible trade-off

http://www.cloudbus.org/papers/P2PbasedContentSharing.pdf

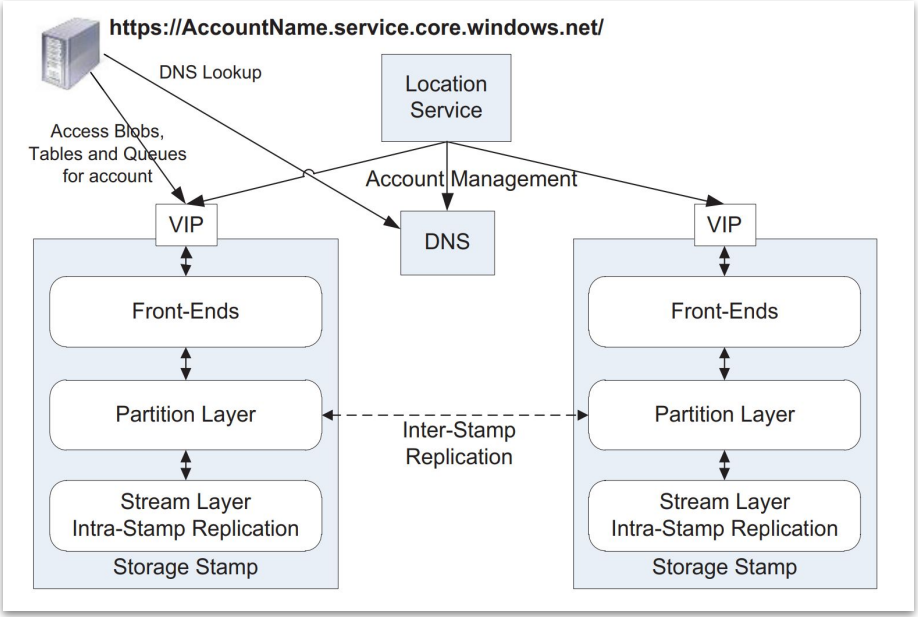
# Cloud storage: Windows Azure Storage (WAS)

**What is different:** a single administrative domain, nodes cannot join and leave arbitrarily, trusted nodes

**API:** object, file, stream storage

**Architecture**: multiple-layered design with each layer internally using **server-client** model

Often in DCs, the server-client model is preferred.




*Windows Azure Storage: a highly available cloud storage service with strong consistency. In SOSP 2011,* https://dl.acm.org/doi/10.1145/964723.383071

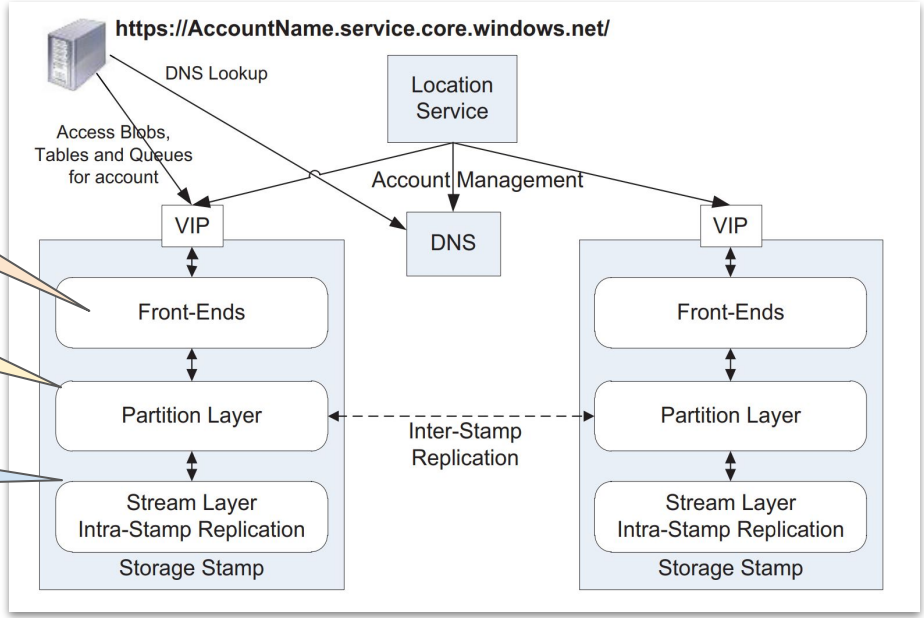
# Cloud storage: Windows Azure Storage (WAS)

More close to an ecosystem than a single system

- Permission checks and routing

- Blob, table, queue
- Scalable object namespace
- Transaction ordering/consistency
- Caching
- Storing on streams

- Low-level I/O management
- I/O scheduling
- Replication
- Disk management



# Recap So Far …

1. Chord - structured (the ring) peer-to-peer system
2. LOCKSS - unstructured peer-to-peer system
3. Server-client setup
   a. Napster - Centralized index server
   b. Gnutella - Completely decentralized
   c. Supernode architecture - a hybrid mix of Napster and Gnutella
4. Windows storage system - a centralized, server-client architecture with trusted nodes only

A variety and mix of systems out there … questions?

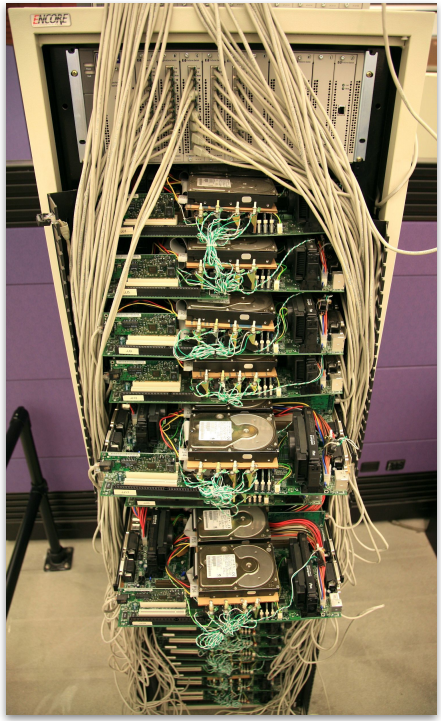
# Data Processing

# Imagine ...

You are one of the biggest data companies in early 2000s

You are gathering increasing large amounts of data to process

- Everyone is writing their own data processing framework
- There are certain “frequent” patterns in processing
- Mostly bulk-throughput, not latency sensitive
- Cannot rely on expensive fault-tolerant hardware



# Data-Parallel, Big-Data processing systems

- **Data management** became equally important as **compute** management
  - Data-parallel (partition and parallelize) model became a more natural fit
- The goal is to build a system architecture with a programming model to capture "many" data-processing applications and paradigms
  - Do not reinvent the wheel, reuse
- Multiple paradigms
  - Batch processing
  - Graph processing
  - Relational data processing (has its own history of parallel processing)
  - Stream data processing
  - Machine Learning
  - More …

# System architecture

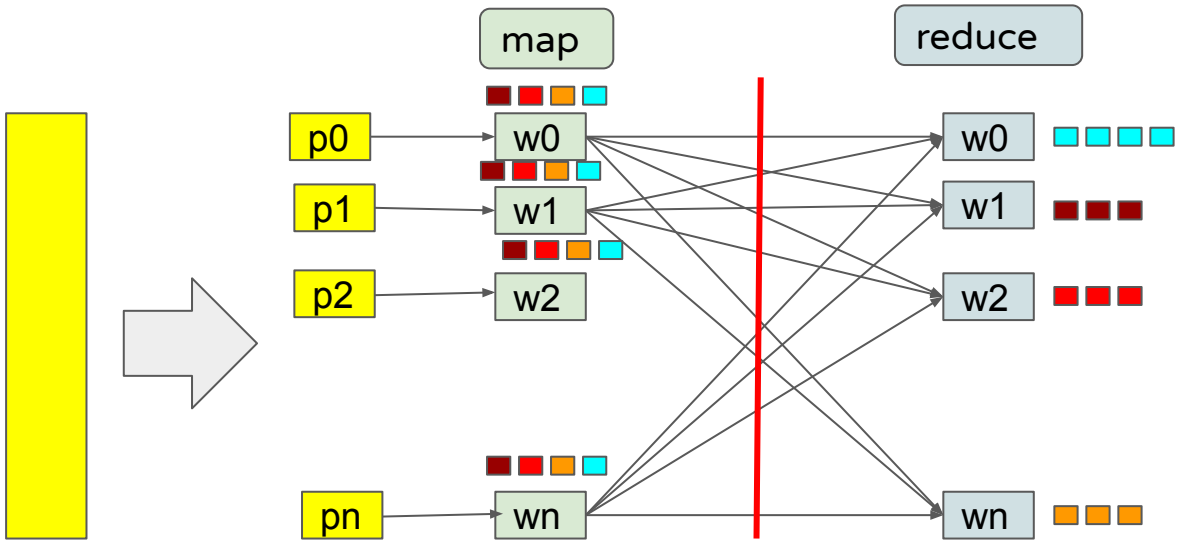
There are many practical concerns when running a real system, specifically:

- Scalability
  - How to manage shared state, if any?
  - Scheduling: which tasks to schedule next? And how fast?
  - Stragglers: What to do with a slow task, machine?
- Fault Tolerance
  - How to recover from a failed machine, task, or operation?
  - What is the cost in terms of energy, money, and performance?
- Performance / Efficiency
  - How to deliver performance? how is messaging done?
  - How to deal with locality? Data movement is expensive (~)
  - How to do load balancing so that no one machine/task is overloaded
  - How easy is for developers to write programs and run them

# MapReduce (2004)

Highly influential work from Google

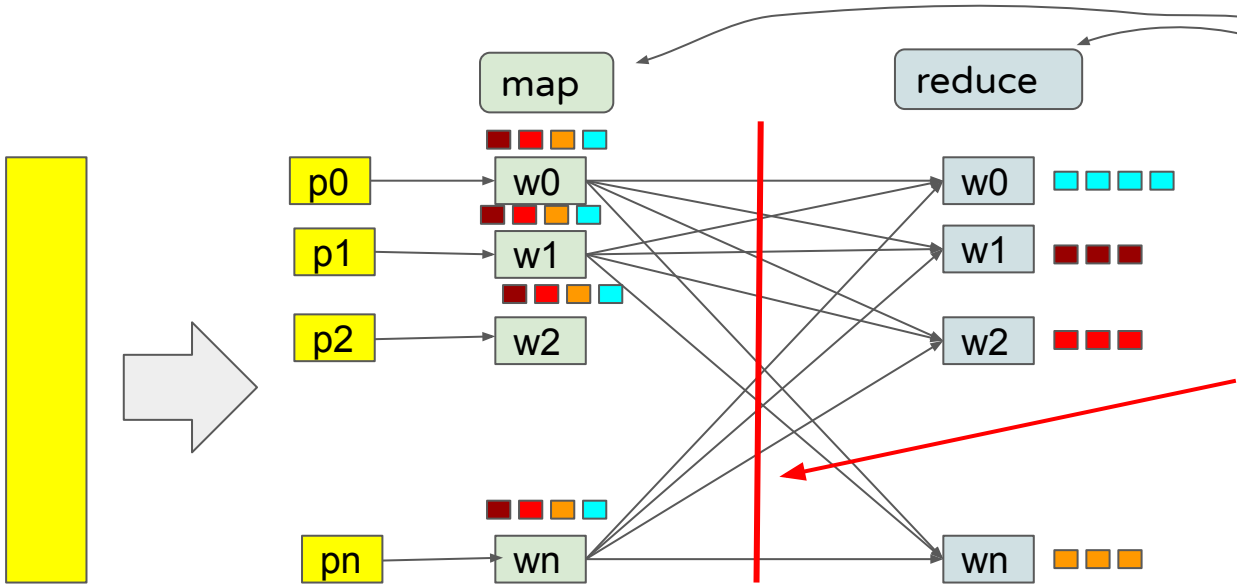
**Programming Model:** a template (map, reduce) based 2-stage bipartite graph



# MapReduce (2004)

Highly influential work from Google

**Programming Model:** a template (map, reduce) based 2-stage bipartite graph



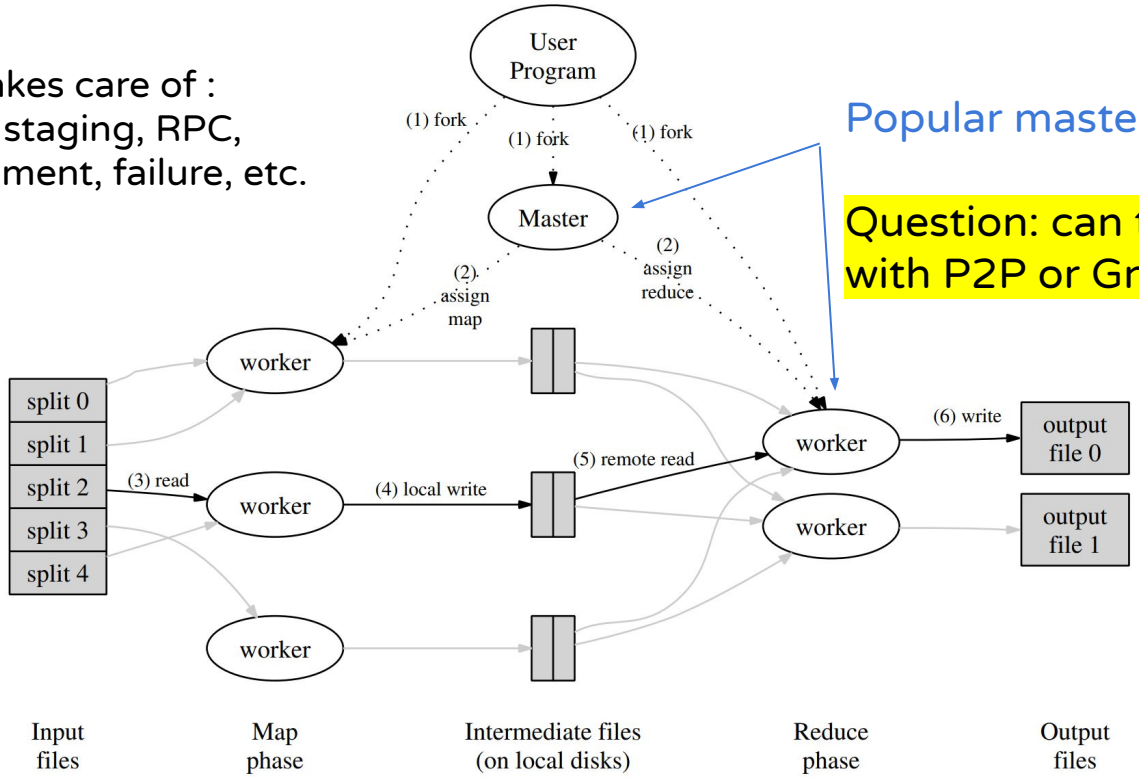

**Data-parallelism:** same map/reduce functions are invoked on different data segments

Bulk Synchronous Parallel (**BSP**) execution

+ Easy to implement and reason about
- Not all machines are the same

# MapReduce : System Architecture

MR framework takes care of :
Scheduling, data staging, RPC, machine management, failure, etc.




Popular master-worker model

Question: can this be done with P2P or Gnutella?

# MapReduce

In case of a fault: map failure => restart, reduce failure => recollect data, master failure (do periodic checkpointing)

- *It was all fine, these were batch jobs (hours, days of runtime)*

MapReduce is hugely influential, but its programing model

- Has limited expressibility (only 2-stages)
- No iteration support (cannot loop back)
- Performance: disk-based system

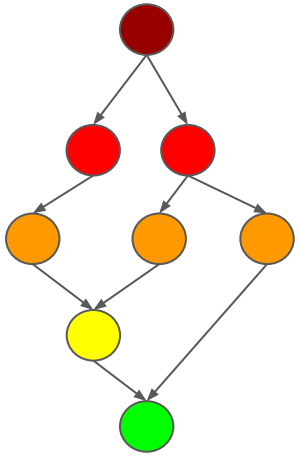
Some of these things can be achieved outside the framework (and people did: see HaLoop - end reference), **but at a cost of extra programming effort!**

→ **Needed: change of contract with the user ⇒ re-division of labour**

# Dryad (2007)

Microsoft's system for data processing

Offered a **programming API** (a new graph description language) to build any arbitrary DAG execution



- Can define computation inside vertices
- Input/output variables for these vertices
- How to connect them

Very powerful idea, but also cumbersome.

Though there were helpers to auto-generate these DAGs.
Multiple operators and optimizations were proposed

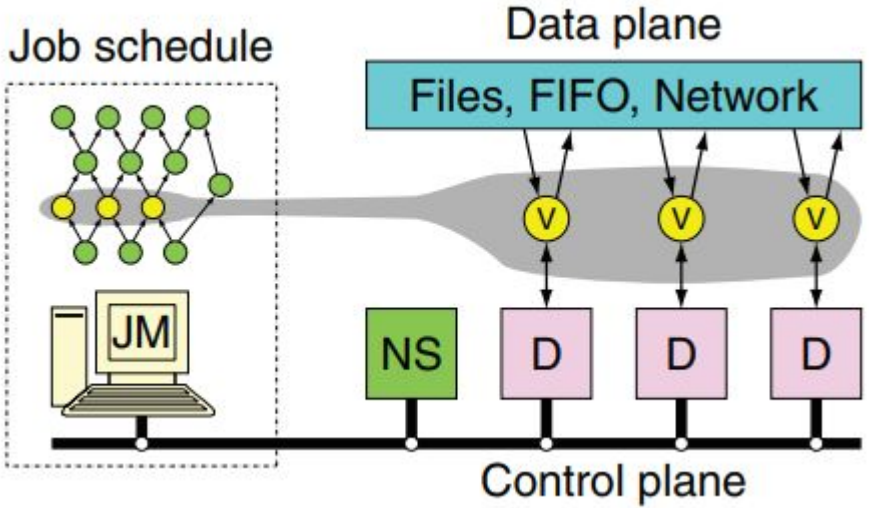
# Dryad : systems details

Similar system architecture

Jobs are represented as DAGs, and there is a JobManager (JM) that schedules vertices on worker machines

There is also a NameServer (resource manager)

Failed jobs/tasks are restarted (assumed deterministic execution)




- Still batch oriented, disk-based systems
- No intermediate data sharing
- No dynamic DAGs

# Spark (2012)

Uses Resilient Distributed Datasets or RDDs - a collection of highly reliable items distributed over multiple machines

**Key ideas:**

1. (Intermediate) Data **sharing** is important
2. Data is kept **in memory** as a immutable distributed collection - fast
    a. Fast sharing and processing in memory
3. Coarse-grained ***transformations*** on these collections - arbitrary DAG
    a. Automatic building of DAG as operations are applied
4. Integration with **modern languages** and frameworks - interactive
    a. Functional programming

# Spark: API

| | | | |
|---|---|---|---|
| **Transformations** | *map*(*f* : T ⇒ U) | : | RDD[T] ⇒ RDD[U] |
| | *filter*(*f* : T ⇒ Bool) | : | RDD[T] ⇒ RDD[T] |
| | *flatMap*(*f* : T ⇒ Seq[U]) | : | RDD[T] ⇒ RDD[U] |
| | *sample*(*fraction* : Float) | : | RDD[T] ⇒ RDD[T] (Deterministic sampling) |
| | *groupByKey*() | : | RDD[(K, V)] ⇒ RDD[(K, Seq[V])] |
| | *reduceByKey*(*f* : (V, V) ⇒ V) | : | RDD[(K, V)] ⇒ RDD[(K, V)] |
| | *union*() | : | (RDD[T], RDD[T]) ⇒ RDD[T] |
| | *join*() | : | (RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))] |
| | *cogroup*() | : | (RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))] |
| | *crossProduct*() | : | (RDD[T], RDD[U]) ⇒ RDD[(T, U)] |
| | *mapValues*(*f* : V ⇒ W) | : | RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning) |
| | *sort*(*c* : Comparator[K]) | : | RDD[(K, V)] ⇒ RDD[(K, V)] |
| | *partitionBy*(*p* : Partitioner[K]) | : | RDD[(K, V)] ⇒ RDD[(K, V)] |
| **Actions** | *count*() | : | RDD[T] ⇒ Long |
| | *collect*() | : | RDD[T] ⇒ Seq[T] |
| | *reduce*(*f* : (T, T) ⇒ T) | : | RDD[T] ⇒ T |
| | *lookup*(*k* : K) | : | RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs) |
| | *save*(*path* : String) | : | Outputs RDD to a storage system, *e.g.*, HDFS |

From the NSDI 2012 paper. There has been significant advancements since then but the overall idea remains!

# Spark: API

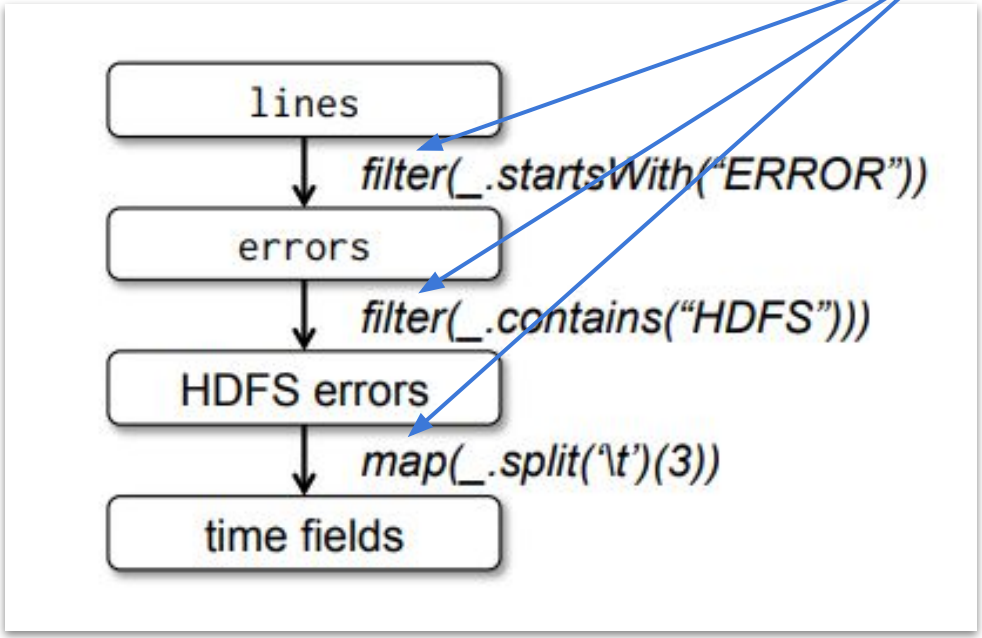
These high level operations determine how a computation can be expressed

The sequence of transformation calls : lineage

- RDDs are immutable, each calls on an RDD returns a new one
- **Advantage**: very simple reasoning about fault tolerance (otherwise: who is writing when and how)
- **Disadvantage**: tracking, and garbage collection overheads

Spark API was closely integrated with Scala. Writing a program for a cluster became as easy as writing for a single machine! (no need to build explicit DAGs like Dryad)

# Spark Examples




```
val points = spark.textFile(...)
                  .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```

# Spark Shell - where magic happens

```
atr@atr-XPS-13:~/Downloads/spark-3.0.0-preview-
19/11/26 14:38:09 WARN Utils: Your hostname, at
19/11/26 14:38:09 WARN Utils: Set SPARK_LOCAL_I
19/11/26 14:38:09 WARN NativeCodeLoader: Unable
Using Spark's default log4j profile: org/apache
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newL
Spark context Web UI available at http://130.37
Spark context available as 'sc' (master = local
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.0.0-pre
      /_/

Using Scala version 2.12.10 (Java HotSpot(TM) 6
Type in expressions to have them evaluated.
Type :help for more information.

scala> val local_seq = Seq(1,2,3,4,5,6,7,8,9,10)
local_seq: Seq[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val RDD_seq = sc.parallelize(local_seq)
RDD_seq: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:26

scala> val filter_rdd = RDD_seq.filter ( _ % 2 == 0)
filter_rdd: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[1] at filter at <console>:25

scala> filter_rdd.count()
res0: Long = 5

scala>
```

```
atr@atr-XPS-13:~$ scala
Welcome to Scala 2.11.12 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_221).
Type in expressions for evaluation. Or try :help.

scala> val local_seq = Seq(1,2,3,4,5,6,7,8,9,10)
local_seq: Seq[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val filter_seq = local_seq.filter( _ % 2 == 0)
filter_seq: Seq[Int] = List(2, 4, 6, 8, 10)

scala> filter_seq.size
res0: Int = 5

scala>
```

# Spark: system architecture

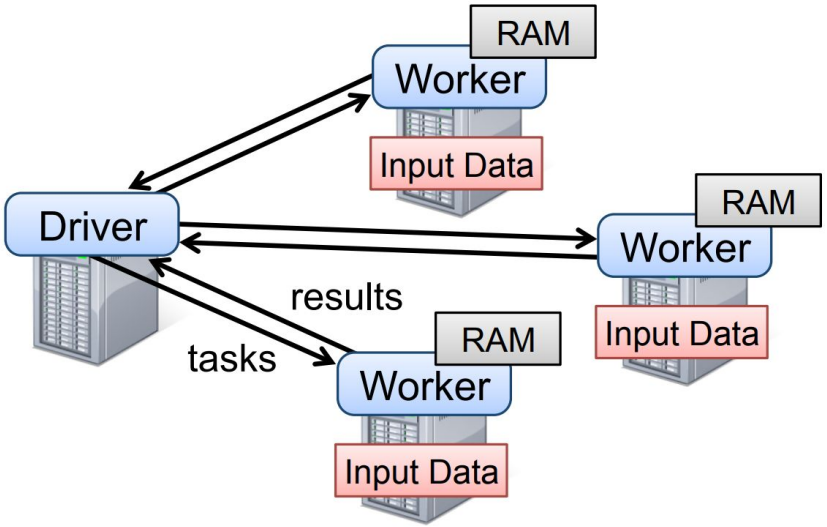
Follows the **driver (master) worker model**

The driver is responsible for all control setup and communicating with workers (stateful!)

Periodic updates on heartbeats

Locality-driven task scheduling

In case of **failures**: reconstruct the lost partition using lineage



# Spark today

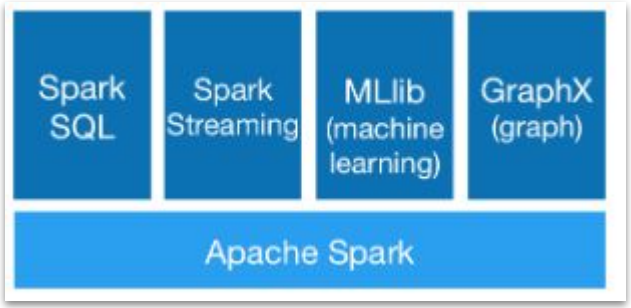
One of the most popular data processing engines

Combines multiple languages frontends
(Python, Java, Scala, R)

Combines several programming paradigms

New typed Dataset API

Mixing and matching ideas from RDBMS, parallel programming, and stream processing



# Some applications were bad fits

- Machine Learning
- Streaming
- Graphs …

They needed

- Fine-grained updates and state management (accesses and sharing)
- Real-time constraints on fresh data processing

What programming model and systems properties helps here?

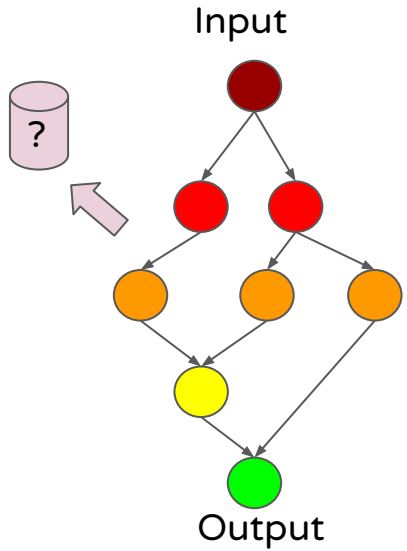
# Machine Learning

# Piccolo (2010)

The basic DAG driven computation does not have any abstraction for shared state

Shared states are common in ML workloads (training model, parameters) and graph algorithms (pagerank)

- Can spark do it? [Globally R/W shared variable?]
  - Make copy after each write (RDDs are immutable)




Piccolo : keep the DAG programming model, plus a distributed shared key-value store for storing shared state

- A user can also deploy memcached/redis, why bother offering it in the framework?

# Piccolo (2010) - KV API

Shared mutable KV store

- Enables sharing of intermediate stage
- Immediate access to updated values

How the shared state is managed?

- Single key operations are atomic
- “Some” total order between multiple users
- User-define accumulator and merge functions

```
Table<Key, Value>:
  clear()
  contains(Key)
  get(Key)
  put(Key, Value)

  # updates the existing entry via
  # user-defined accumulation.
  update(Key, Value)

  # Commit any buffered updates/puts
  flush()

  # Return an iterator on a table partition
  get_iterator(Partition)
```

Writing/Reading KV is part of the Piccolo user API

# Piccolo: System Architecture

Master-worker model,
program = control + loop("kernels")

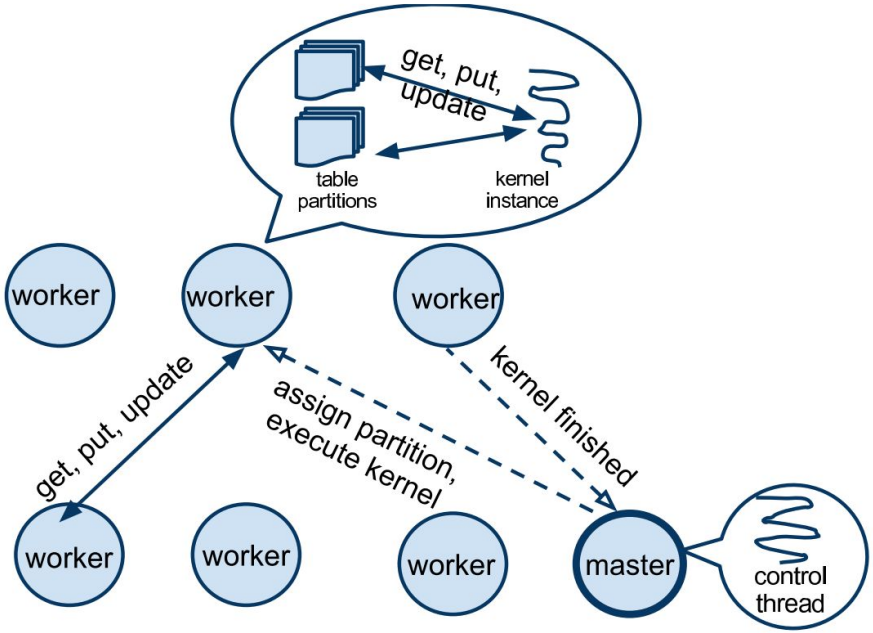
Control runs on the master

Kernels on the workers (who also store KV table partitions)

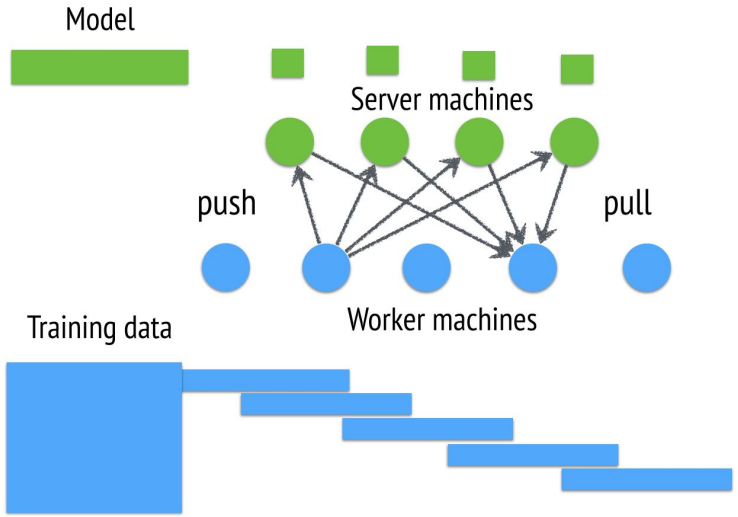
Global barrier for sync

Control thread decides the next step (hence, can do dynamic DAG!)

Load balancing via task stealing



# Distributed Parameter Server (2012)




Distributed Parameter Server (**PS**) is an influential work for ML training workloads

The PS idea has a long history (see the paper)

But here:

- ML model stored on multiple servers
- workers pull model and update with training data
- workers push the updated model back to the servers
- system has high-level operations associated with metrics, vectors etc. - Not just low level opaque get/put API

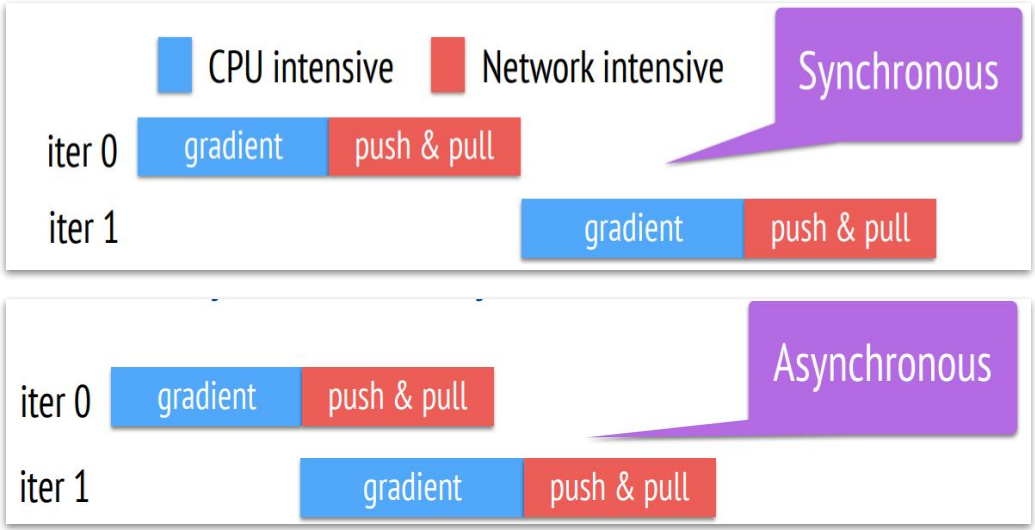
https://www.usenix.org/sites/default/files/conference/protected-files/osdi14_slides_li-mu.pdf

# Pull/Push Approach

A lot of network and CPU coordination

Flexible consistency models and API that lets users choose

- Synchronous (like BSP)
- Asynchronous
  - Bounded
  - Unbounded, eventual



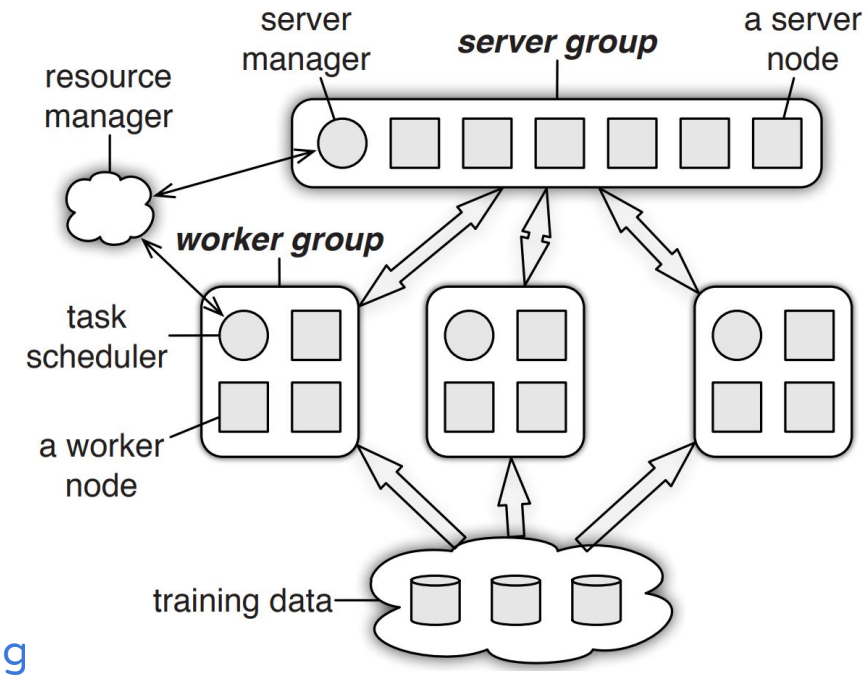

https://www.usenix.org/sites/default/files/conference/protected-files/osdi14_slides_li-mu.pdf

# System Architecture

Servers are managed with consistent hashing (recall: Chord), why?

- Similar mechanisms to handle server management (failure, replication)

Workers are left to the users -
if they are worth it or not. Insight: ML algorithms are stochastic/probabilistic, and OK with a bit of data inconsistency.
But this would not be a good design for storag



# Challenges so far

- General pattern of stateless workers and stateful servers with training data and model updates - does not give a general approach for ML, e.g., RNNs, or reinforcement learning

- Any fiddling around in update rules or testing algorithms required changing internals

- No single solution for training and deployment

- Multi-device deployments (accelerators, TPUs, GPUs, mobile,…)

Need for a general purpose ML framework

# ML: TensorFlow (2016)

Proposed by google in the 2015-2016 timeframe


Single training and deployment framework

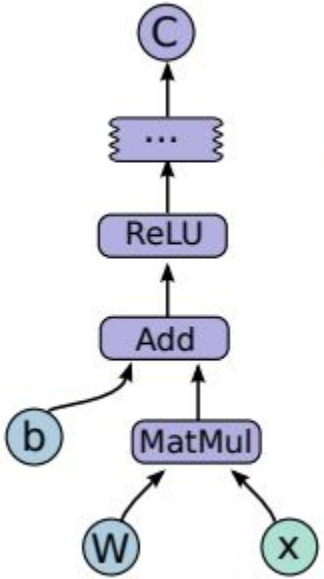
Expressive unified Dataflow Graphs (DFGs) abstraction with

- Express an ML numeric computation as a graph
- Stateful vertices with any number of inputs and outputs
- Edges defines computation flow that needs to happen on states
- Tensor data flows over the edges between vertices

Combines ideas from accelerator research, dataflow models, and PS

Session and sub-graph execution, synchronization, persistent variables…

# Example

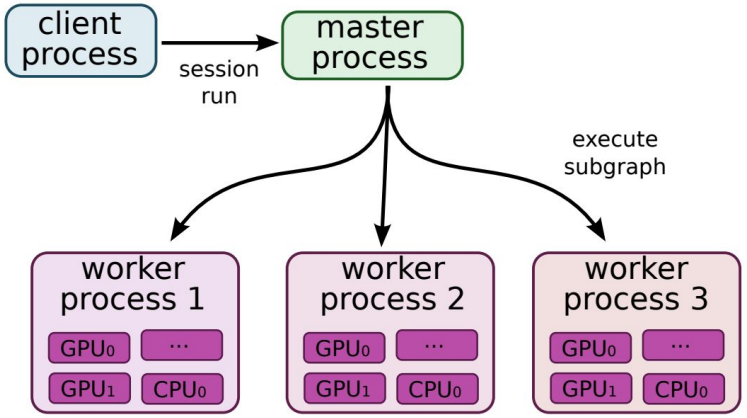



```
import tensorflow as tf

b = tf.Variable(tf.zeros([100]))                    # 100-d vector, init to zeroes
W = tf.Variable(tf.random_uniform([784,100],-1,1)) # 784x100 matrix w/rnd vals
x = tf.placeholder(name="x")                        # Placeholder for input
relu = tf.nn.relu(tf.matmul(W, x) + b)              # Relu(Wx+b)
C = [...]                                           # Cost computed as a function
                                                    # of Relu
```

| Category | Examples |
|---|---|
| Element-wise mathematical operations | Add, Sub, Mul, Div, Exp, Log, Greater, Less, Equal, ... |
| Array operations | Concat, Slice, Split, Constant, Rank, Shape, Shuffle, ... |
| Matrix operations | MatMul, MatrixInverse, MatrixDeterminant, ... |
| Stateful operations | Variable, Assign, AssignAdd, ... |
| Neural-net building blocks | SoftMax, Sigmoid, ReLU, Convolution2D, MaxPool, ... |
| Checkpointing operations | Save, Restore |
| Queue and synchronization operations | Enqueue, Dequeue, MutexAcquire, MutexRelease, ... |
| Control flow operations | Merge, Switch, Enter, Leave, NextIteration |

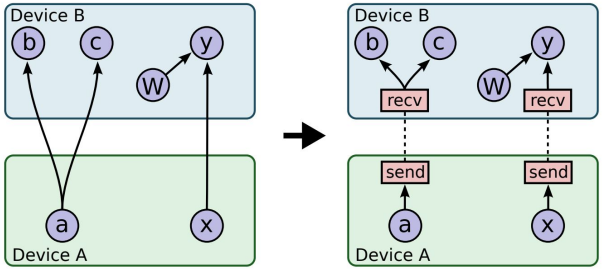
# TensorFlow: Architecture




**Master-worker model**, with deferred execution

1. Define the graph
2. Optimize and deploy the graph

Optimized implementation of operators on different devices (**kernels**)




Dynamic insertion of control/communication nodes (loop, if statements, rx/tx nodes)

Node placement (each accelerator device is abstracted with certain capabilities) : run kernel, allocate mem, access host mem

FT: A user-driven (periodic or “X” iteration) checkpointing and restore

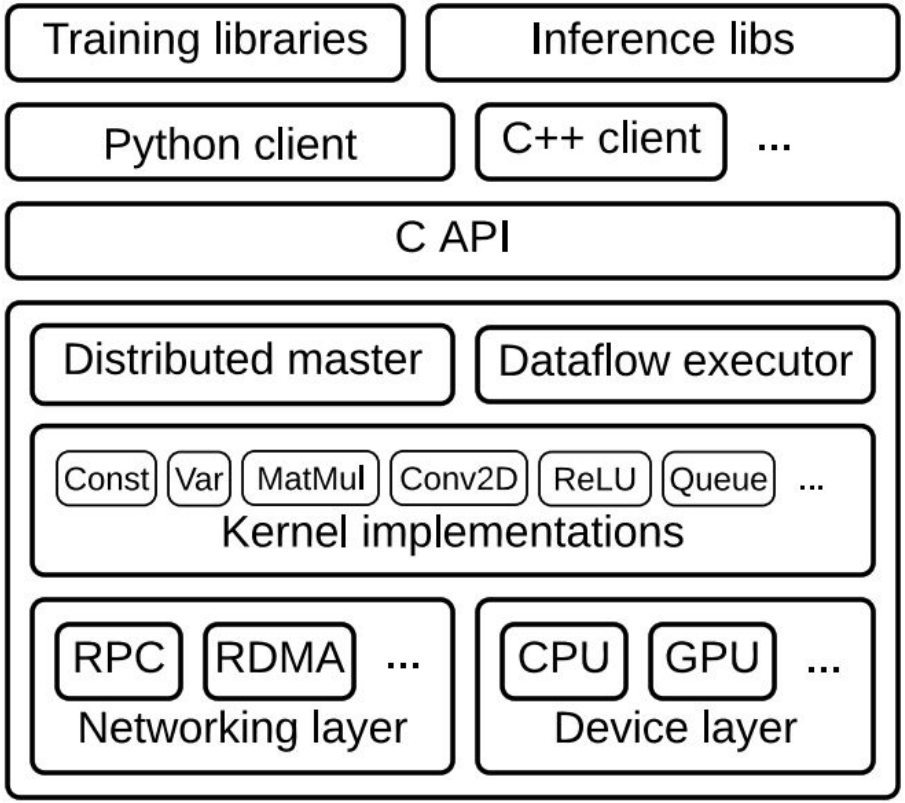
# TensorFlow: Today

One of the most popular frameworks

Supports multiple devices, libraries, operators, and devices

-- * --

- No dynamic graphs,
- New emerging (simulation, actor, reinforcement learning) models, in TF?



# Graph Processing

(see the lecture notes - similar evolution)

# Brief Overview

**Pregel :** "*think like a vertex*"

- user defines what each vertex should compute and how
- framework manages information flow (messages)
- uses BSP execution

**GraphLab :** BSP is not good, different vertex do different amount of work

- Asynchronous execution, scheduling (3 models for consistency and parallelism)
- Better graph partitioning (consider graph structure - PowerLaw Graphs)

**GraphX :** Deploying a separate graph processing framework is pain

- Builds on a common pattern of Gather-Apply-Scatter (GAS)
- Integrated graph processing with RDDs (joins, and groupBy)
- Optimized implementation

# Open Challenges and Trends

# Challenge - Inefficiency

My framework “X” is the best for solving “Y”

I can also solve my problem “Z” in “X”

But is “X” the best framework to solve “Z”?

- *Nobody ever got fired for using Hadoop on a cluster*, HotCDP '12

Often Big Data frameworks sell scalability to compensate for performance

- Underlying argument: add enough resources and you will get infinite performance scalability
- Is it really true?
- *Scalability! But at what COST?*, HotOS 2015

# COST - Configuration that Outperforms a Single Thread

| 20xPR | cores | twitter_rv | uk_2007_05 |
|---|---|---|---|
| Spark | 128 | 857s | 1759s |
| Giraph | 128 | 596s | 1235s |
| GraphLab | 128 | 249s | 833s |
| GraphX | 128 | 419s | 462s |

from Gonzalez *et al.*, OSDI 2014

```
fn pagerank<G: Graph>(graph: &G, nodes: usize, alpha: f32)
{
    let mut src = vec![0f32; nodes];
    let mut dst = vec![0f32; nodes];
    let mut deg = vec![0f32; nodes];

    graph.map_edges(|x, _| { deg[x] += 1f32 });

    for _iteration in (0 .. 20) {
        println!("Iteration: {}", _iteration);
        for node in (0 .. nodes) {
            src[node] = alpha * dst[node] / deg[node];
            dst[node] = 1f32 - alpha;
        }

        graph.map_edges(|x, y| { dst[y] += src[x]; });
    }
}
```

https://www.usenix.org/sites/default/files/conference/protected-files/hotos15_slides_mcsherry.pdf

# COST - Configuration that Outperforms a Single Thread

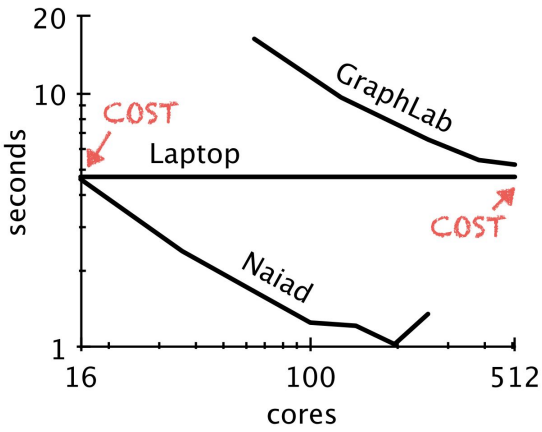
| 20xPR | cores | twitter_rv | uk_2007_05 |
|---|---|---|---|
| Spark | 128 | 857s | 1759s |
| Giraph | 128 | 596s | 1235s |
| GraphLab | 128 | 249s | 833s |
| GraphX | 128 | 419s | 462s |
| Laptop | 1 | | |

# COST - Configuration that Outperforms a Single Thread

| 20xPR | cores | twitter_rv | uk_2007_05 |
|---|---|---|---|
| Spark | 128 | 857s | 1759s |
| Giraph | 128 | 596s | 1235s |
| GraphLab | 128 | 249s | 833s |
| GraphX | 128 | 419s | 462s |
| Laptop | 1 | 300s | 651s |

# COST - Configuration that Outperforms a Single Thread

| 20xPR | cores | twitter_rv | uk_2007_05 |
|---|---|---|---|
| Spark | 128 | 857s | 1759s |
| Giraph | 128 | 596s | 1235s |
| GraphLab | 128 | 249s | 833s |
| GraphX | 128 | 419s | 462s |
| Laptop | 1 | ~~300s~~ 110s | ~~651s~~ 256s |



# Basic assumptions

CPU is very fast and I/O is slow

CPU performance is improving constantly

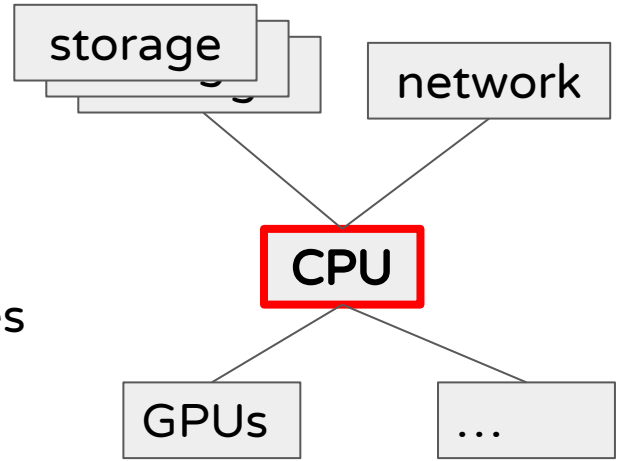
Small I/O is slower than the bulk sequential accesses

Random storage accesses are slow

Messages takes 100s of microsecond or milliseconds

Going to remote machine is slower than accessing local resources

…




**A New Golden Age for Computer Architecture**, Hennessy and Patterson Turing Lecture, 2018, https://cacm.acm.org/magazines/2019/2/234352-a-new-golden-age-for-computer-architecture/fulltext

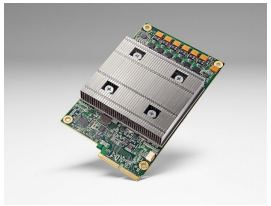
# The world is changing

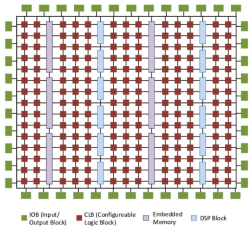
CPU performance is not improving

- Various thermal and manufacturing limits

Hardware/accelerator are now mainstream

- Powered by Moore's law
- Performance and capabilities (40, 100, 200 Gbps)
- RDMA networking - very impactful !
- Heterogeneity: GPU, TPU, FPGA, smartNICs, programmable storage

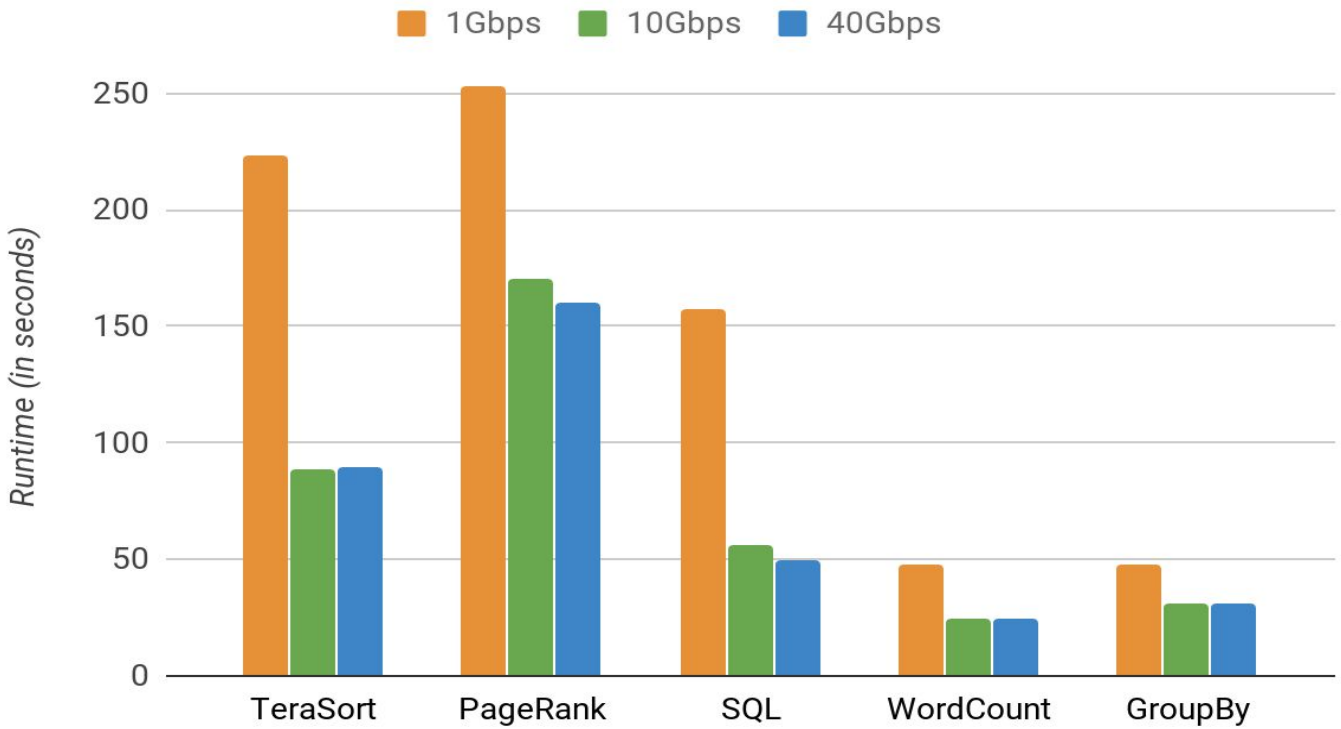






How should we build next generation of distributed systems keeping this hardware in mind?

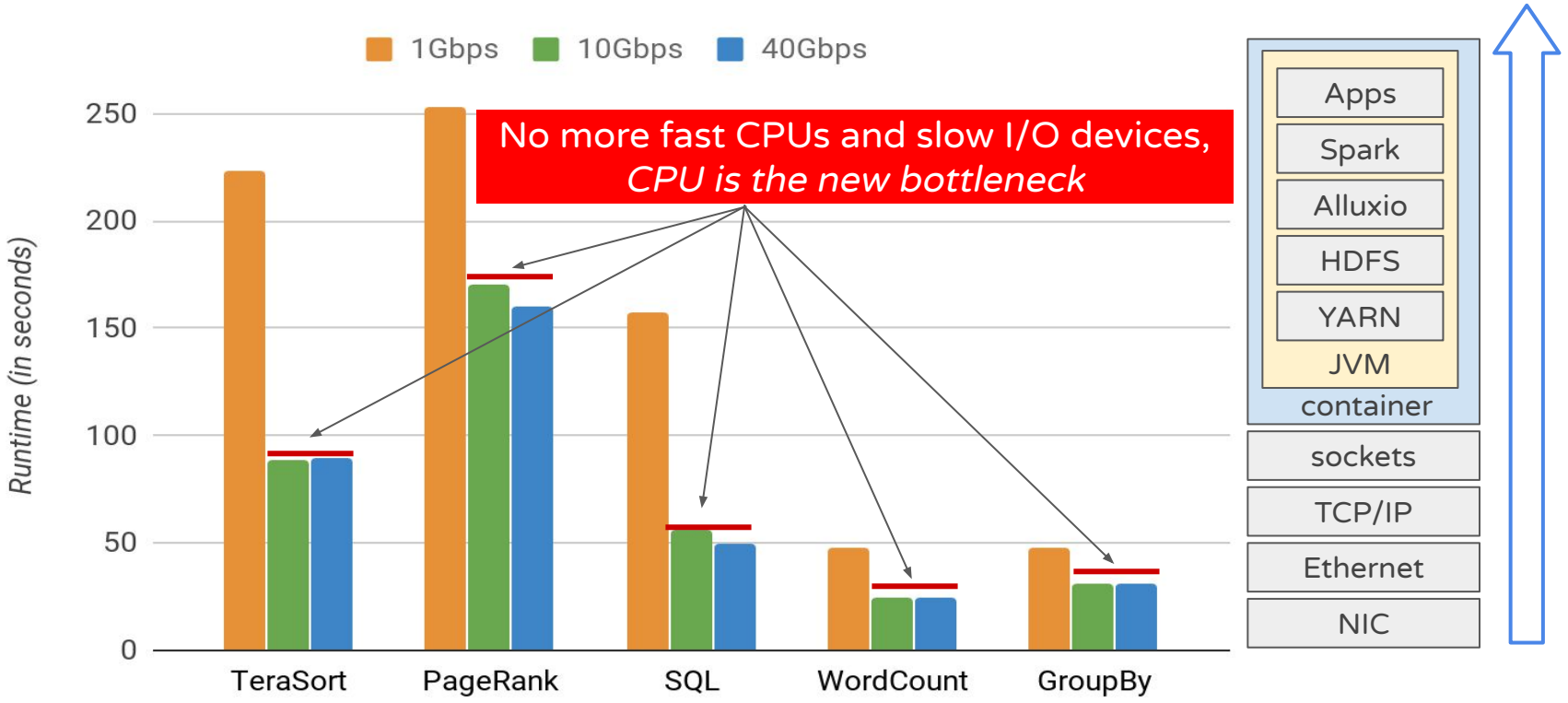
**A New Golden Age for Computer Architecture**, Hennessy and Patterson Turing Lecture, 2018, https://cacm.acm.org/magazines/2019/2/234352-a-new-golden-age-for-computer-architecture/fulltext

# What happens when we don't change




Trivedi et al., On The [Ir]relevance of Network Performance for Data Processing, published at USENIX HotCloud'16.

# What happens when we don't change




Trivedi et al., On The [Ir]relevance of Network Performance for Data Processing, published at USENIX HotCloud'16.

# What we are not covering in the lecture

- Integration of accelerators and advanced technologies
  - RDMA, programmable storage and networking elements, GPUs, FPGAs, Switches
- Detailed discussion on workloads-specific systems
  - Bioinformatics, Astronomy, eCommerce, Social computing, others
- Debugging, troubleshooting, and tracing frameworks
  - OpenTrace, Pivot Tracing, X-Trace
- Resource management
  - Borg, YARN, Mesos
- Performance variance and measurements
- Languages and runtime for distributed systems - JVM, JITing, new languages
- Security concerns

# Incredible Opportunity

Right now we are fundamentally re-thinking our systems designs …

1. What new kind of devices should we leverage in distributed systems
   a. FPGA, programmable storage and networking devices, compute accelerators like TPUs and GPUs
2. How a distributed storage systems should look like?
3. How a distributed data processing systems should look like?
4. What is the right system architecture?
5. What is the right programming model?
6. What is the resource requirement, performance scalability, energy efficiency, and cost of such systems?
7. And many more ideas for research projects and thesis works …

# Take away messages

1. There are no one single design/system that fits all
2. Application domain, scale, and APIs (and often developers' preferences) determine the structure of the system
    a. Data centers often prefer a simple master-worker model
    b. Independent ideas eventually converge in a single powerful system
    c. But then, powerful system becomes too general to deliver good performance for a specific domain/application (and the cycle continues)
3. MapReduce (data-parallel model) and its successors became popular because
    a. Easy to program, handled many concerns transparently esp. Fault tolerance
    b. Deliver performance by simply scaling out by adding more machines
    c. But, have poor efficiency and baseline performance (high COST!)
4. MapReduce-and-friends are only _ONE_ way of building distributed systems

# Further Reading


1. Saniya Ben Hassen, Henri E. Bal, and Ceriel J. H. Jacobs. 1998. A task- and data-parallel programming language based on shared objects. ACM Trans. Program. Lang. Syst. 20, 6 (Nov. 1998), 1131–1170. DOI:https://doi.org/10.1145/295656.295658
2. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '01). Association for Computing Machinery, New York, NY, USA, 149–160. DOI:https://doi.org/10.1145/383059.383071
3. Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. 2010. HaLoop: efficient iterative data processing on large clusters. Proc. VLDB Endow. 3, 1–2 (September 2010), 285–296. DOI:https://doi.org/10.14778/1920841.1920881
4. Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07). Association for Computing Machinery, New York, NY, USA, 59–72. DOI:https://doi.org/10.1145/1272996.1273005
5. Russell Power and Jinyang Li. 2010. Piccolo: building fast, distributed programs with partitioned tables. In Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI'10). USENIX Association, USA, 293–306.
6. Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI'14). USENIX Association, USA, 583–598.
7. Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: a system for large-scale machine learning. In Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI'16). USENIX Association, USA, 265–283.