https://hao-ai-lab.github.io/dsc204a-f25/

DSC 204A: Scalable Data Systems
Fall 2025

Staff
Instructor: Hao Zhang
TAs: Mingjia Huo, Yuxuan Zhang

@haozhangml @haoailab
haozhang@ucsd.edu

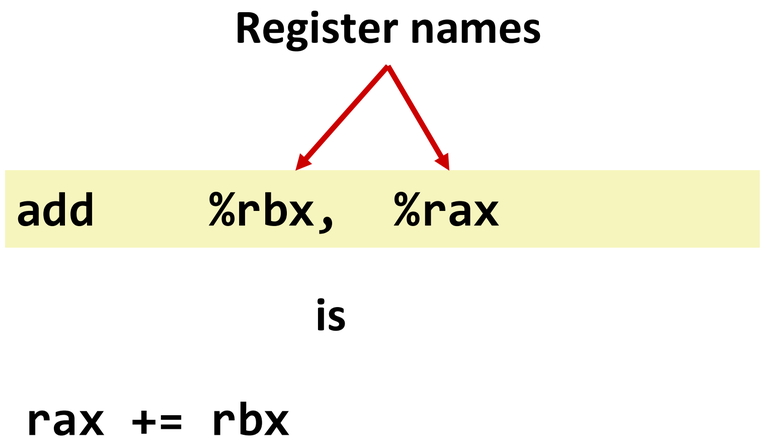
Summary: Batch Processing

- Batch Processing
  - Suitable for latency insensitive tasks
  - Map-reduce prog model: mapper, reducer, (combiner, partitioner)
  - Many Map-reduce jobs to compose dataflows
    - They communicate via disk I/O
  - Pros and Cons
    - Pros: expressive, scalable, and fault tolerant
    - Cons: low performance due to disk I/O, a bit less intuitive to think of?

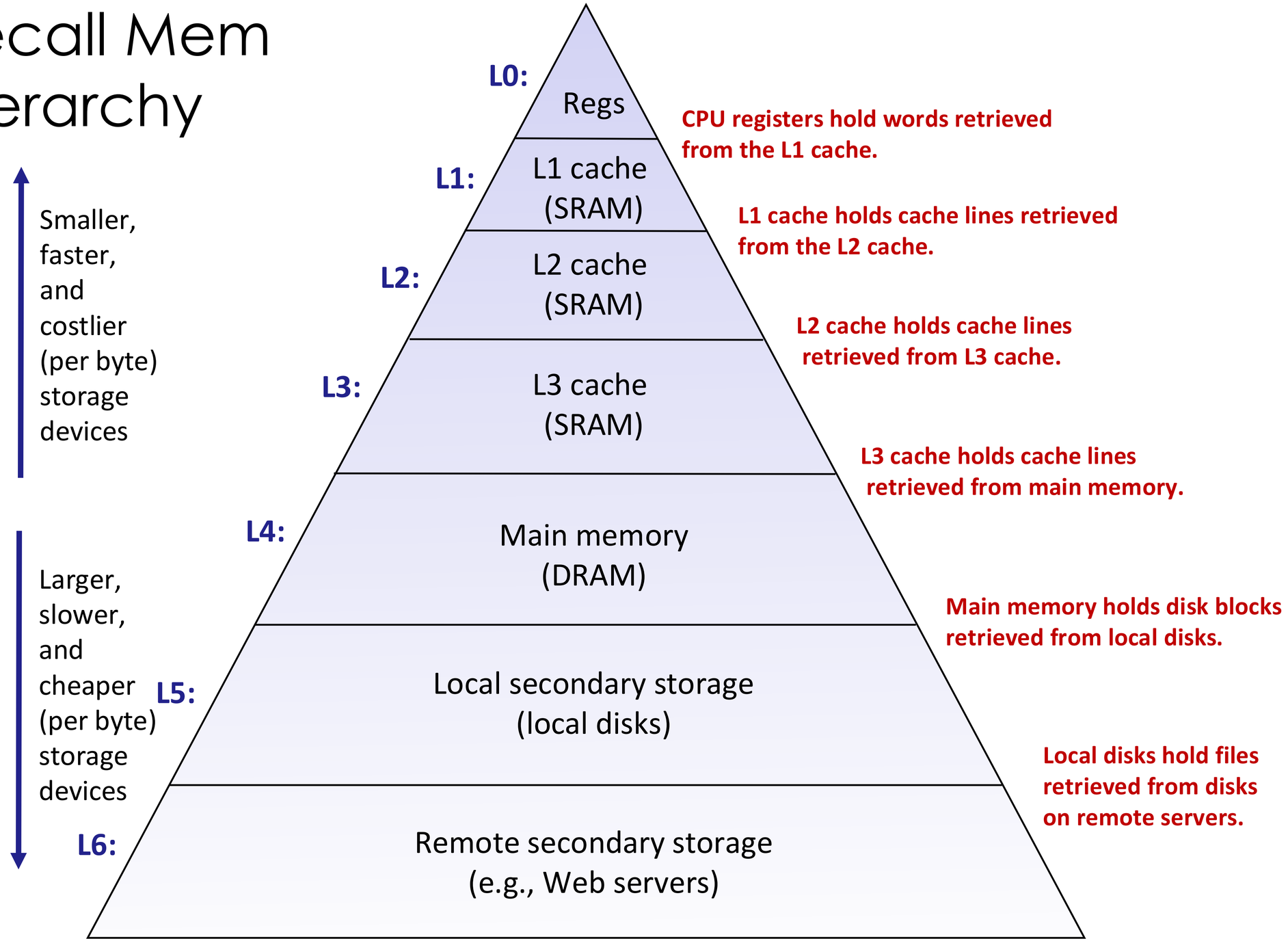
Next: Stream Processing

- Computation vs. I/O: Arithmetic intensity
  - Loop fusion
- When MapReduce fails
- Spark and RDD
- Why Spark succeeded

Recall: Instruction



Recall Mem Hierarchy



How to measure the impact of I/O

- I/O is the primary enemy of computer engineers/scientists: it will always slow down computation in every levels of the memory hierarchy
  - Processor reads/writes cache or memory
  - Map-reduce save and load results from distributed storage
- Q: how we measure such slowdown?
  - Arithmetic intensity

Arithmetic Intensity

$$AI = \frac{\#Compute\ Op}{\#\mathrm{I/O}\ op}$$

Arithmetic intensity

```
void add(int n, float*   A, float*   B, float*   C){
   for   (int   i=0; i<n; i++)
      C[i]  = A[i]  + B[i];
}
```

Two loads, one store per math op

1. Read A[i]
2. Read B[i]
3. Add A[i]+B[i]
4. Store C[i]

Which program performs better? Program 1

```
void add(int n, float* A, float* B, float* C){
   for (int i=0; i<n; i++)
      C[i] = A[i] + B[i];
}

void mul(int n, float* A, float* B, float* C) {
   for (int i=0; i<n; i++)
      C[i] = A[i] * B[i];
}

float* A, *B, *C, *D, *E, *tmp1, *tmp2;
//  assume arrays are allocated here
//  compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);
```

Two loads, one store per math op
(arithmetic intensity = 1/3)

Two loads, one store per math op
(arithmetic intensity = 1/3)

Overall arithmetic intensity = 1/3

Which program performs better? Program 2

```
float* A, *B, *C, *D, *E, *tmp1, *tmp2;
// assume arrays are allocated here
// compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);



void fused(int n, float* A, float* B, float* C, float* D,
   float* E) {
   for (int i=0; i<n; i++)
      E[i] = D[i] + (A[i] + B[i]) * C[i];
}
// compute E = D + (A + B) * C
fused(n, A, B,C, D, E);
```

Overall arithmetic intensity = 1/3

Four loads, one store per 3 math ops
arithmetic intensity = 3/5

computation fusion!

Core Problem of Map-reduce

Low arithmetic intensity due to Disk I/O

PageRank Computation

Initially

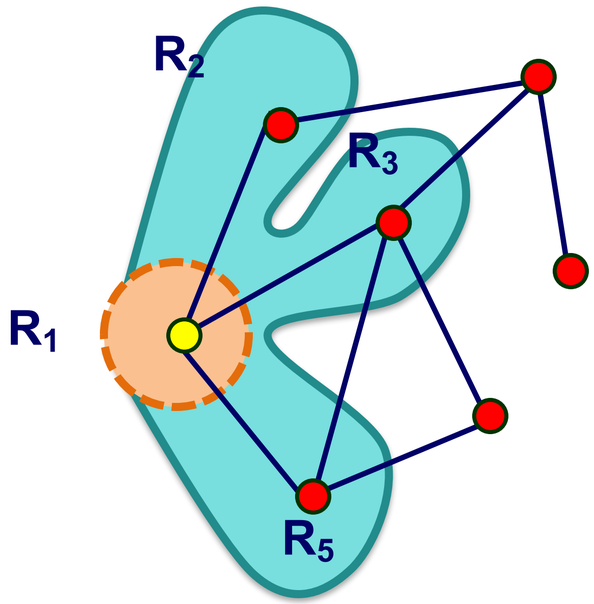
- Assign weight 1.0 to each page

Iteratively

- Select arbitrary node and update its value

Convergence

- Results unique, regardless of selection ordering



$R_1 \leftarrow \quad 0.1 + 0.9 * (\frac{1}{2} R_2 + \frac{1}{4} R_3 + \frac{1}{3} R_5)$

Iterative algorithms must load from disk each iteration

```
void pagerank_mapper(graphnode n, map<string,string> results) {
   float val = compute update value for n
   for (dst in outgoing links from n)
     results.add(dst.node, val);
}

void pagerank_reducer(graphnode n, list<float> values, float& result) {
    float sum = 0.0;
    for (v in values)
       sum += v;
    result = sum;
}

for (i = 0 to NUM_ITERATIONS) {
   input = load graph from last iteration
   output = file for this iteration output
   runMapReduceJob(pagerank_mapper, pagerank_reducer, result[i-1], result[i]);
}
```

Low
Arithmetic Intensity!

in-memory, fault-tolerant distributed computing
http://spark.apache.org/

Goals

- This guy thought UC GSR salary too low so he decided to make some money (roughly 1M) via the Netflix challenge.

Netflix Prize

2 languages

Article Talk | Read Edit View history Tools

From Wikipedia, the free encyclopedia

The **Netflix Prize** was an open competition for the best collaborative filtering algorithm to predict user ratings for films, based on previous ratings without any other information about the users or films, i.e. without the users being identified except by numbers assigned for the contest.

The competition was held by Netflix, a video streaming service, and was open to anyone who is neither connected with Netflix (current and former employees, agents, close relatives of Netflix employees, etc.) nor a resident of certain blocked countries (such as Cuba or North Korea).[1] On September 21, 2009, the grand prize of US$1,000,000 was given to the BellKor's Pragmatic Chaos team which bested Netflix's own algorithm for predicting ratings by 10.06%.[2]

Recommender systems

Concepts
Collective intelligence · Relevance · Star ratings · Long tail

Methods and challenges
Cold start · Collaborative filtering · Dimensionality reduction · Implicit data collection · Item-item collaborative filtering · Matrix factorization · Preference elicitation · Similarity search

Matei Zaharia

Associate Professor, Computer Science
matei@berkeley.edu
Google Scholar | LinkedIn | Twitter

I'm an associate professor at UC Berkeley (previously Stanford), where I work on computer systems and machine learning. I'm also co-founder and CTO of Databricks.

Interests: I'm interested in computer systems for large-scale workloads such as AI, data analytics

Goals

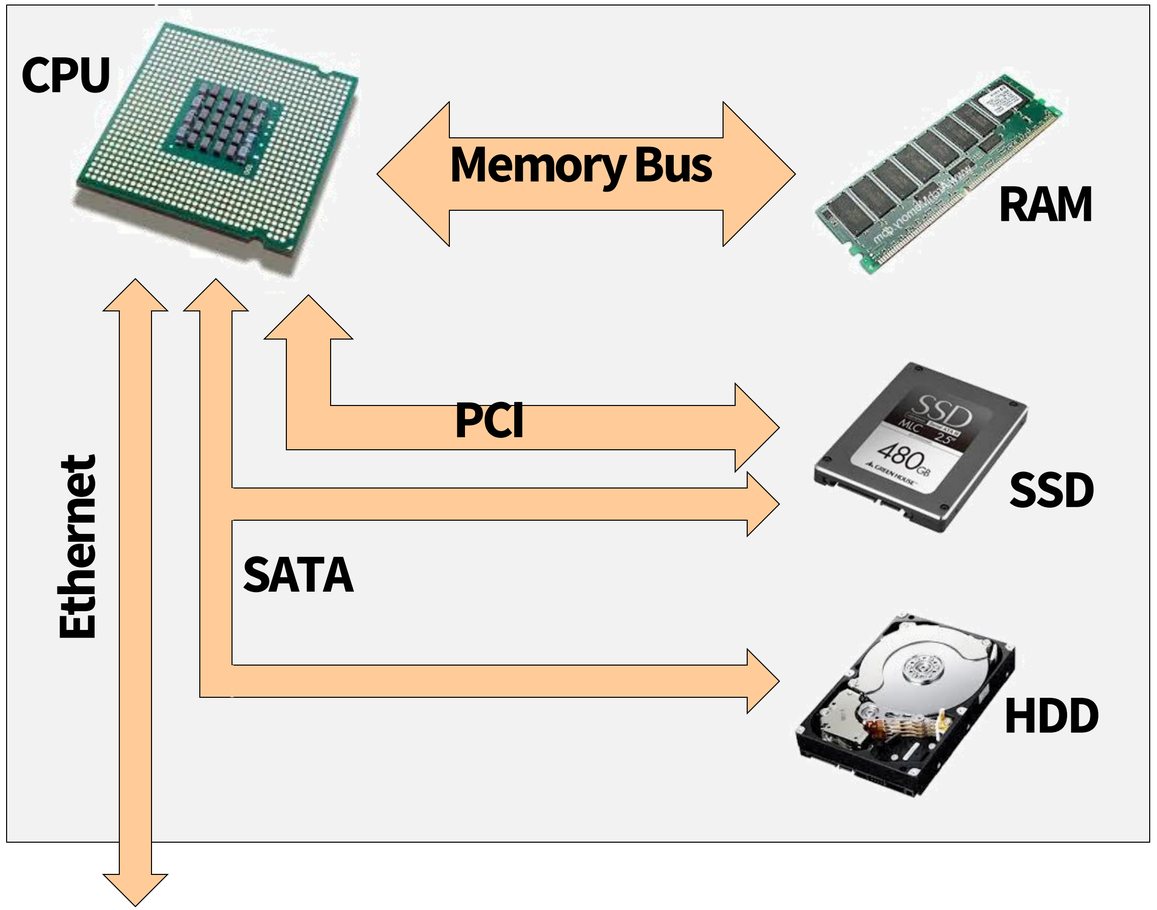
- Programming model for cluster-scale computations where there is **significant reuse** of intermediate datasets
  - Iterative machine learning and graph algorithms
  - Interactive data mining: load large dataset into aggregate memory of cluster and then perform multiple ad-hoc queries

- Don't want incur inefficiency of writing intermediates to persistent distributed file system (want to keep it in memory)
  - Challenge: efficiently implementing fault tolerance for large-scale distributed in-memory computations.

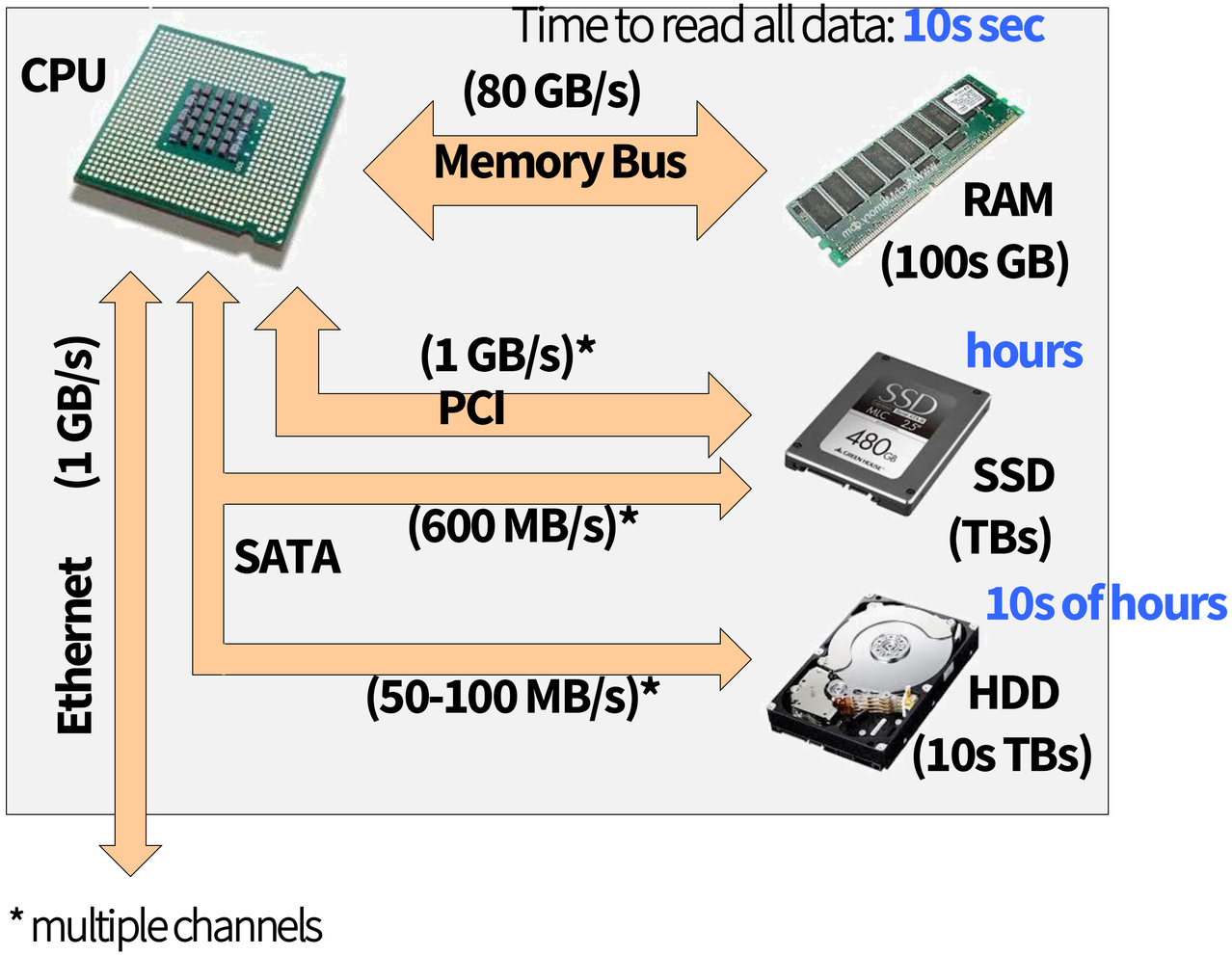
Three Necessary Conditions

- Memory: large (cheap) enough
- Network: fast (cheap) enough
- fault tolerance: at least as good as map-reduce

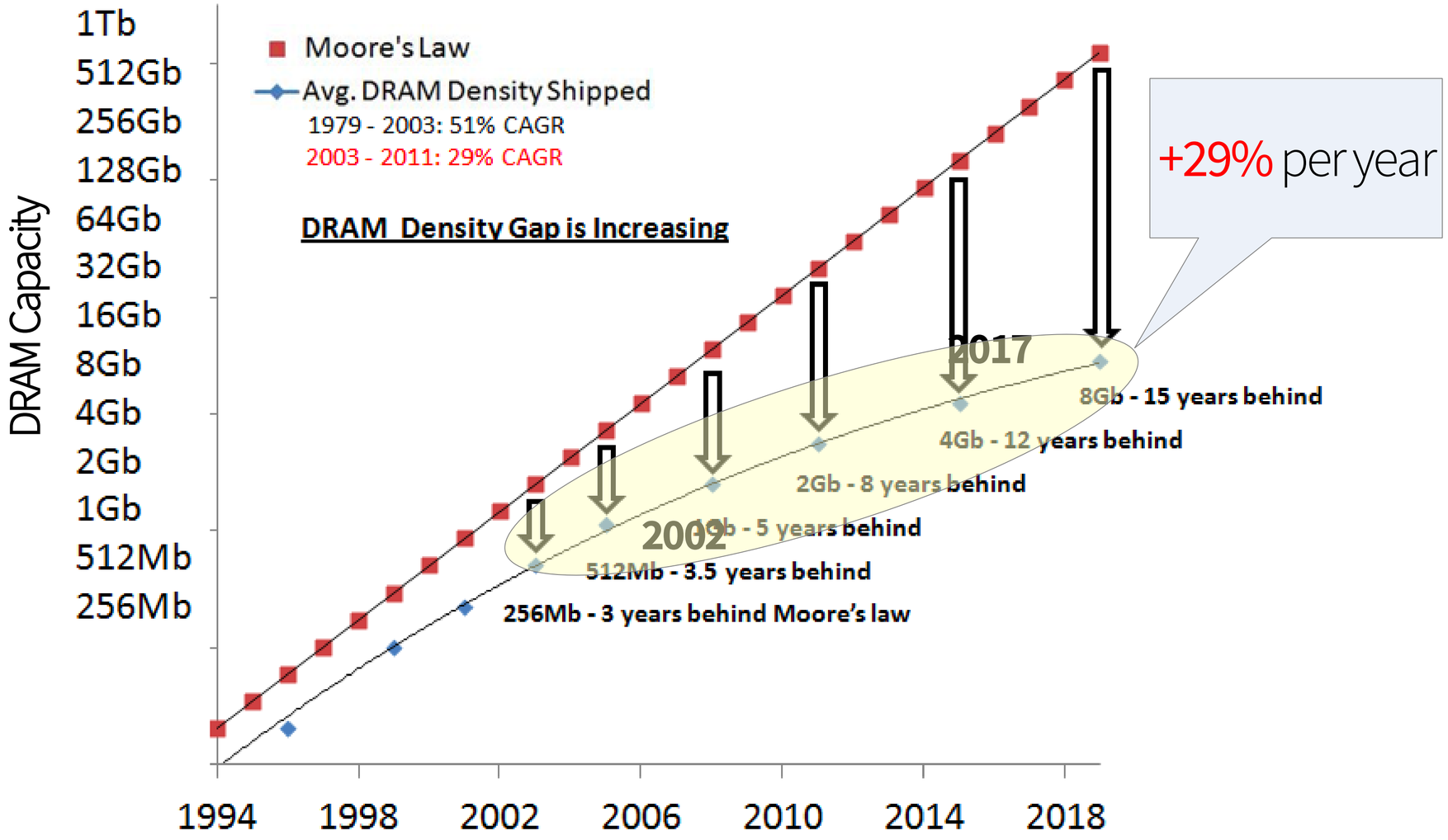
Typical Server Node



Typical Server Node



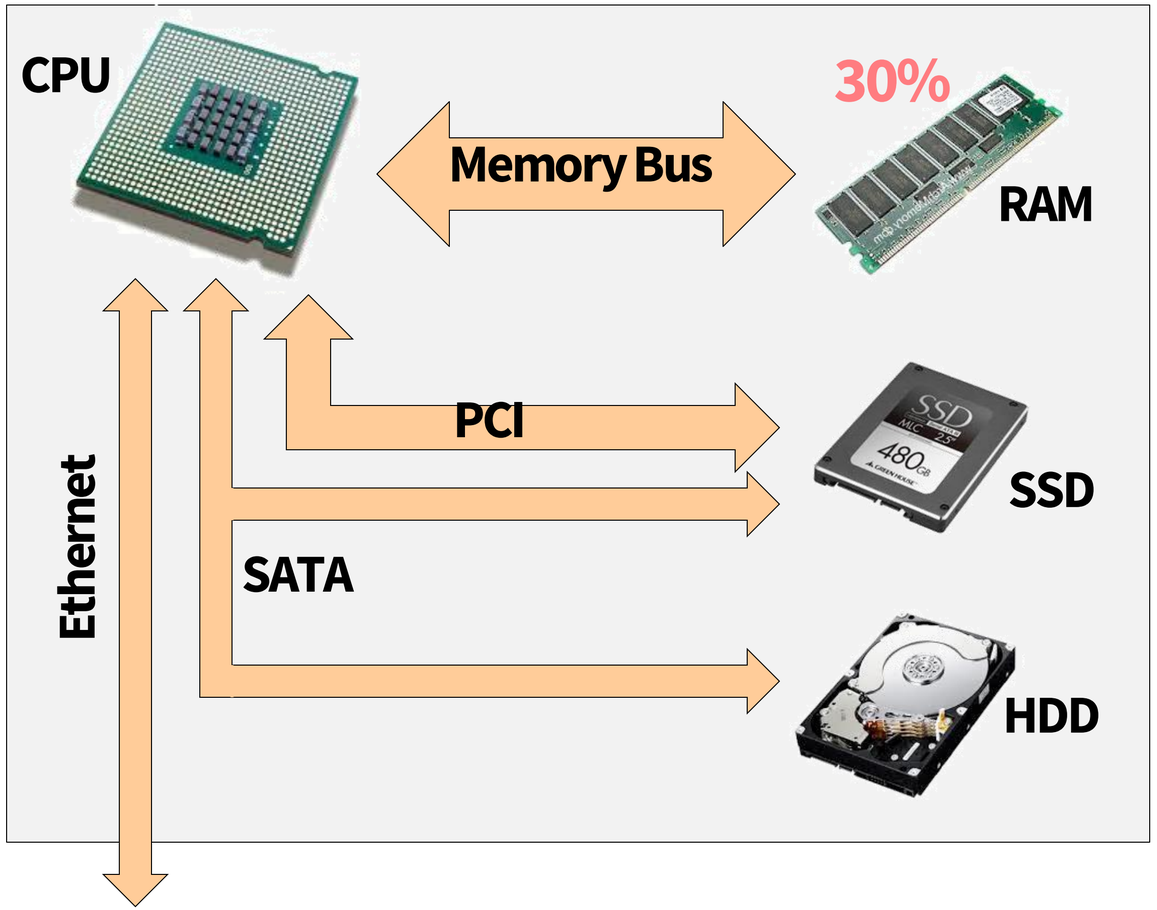
Memory Capacity

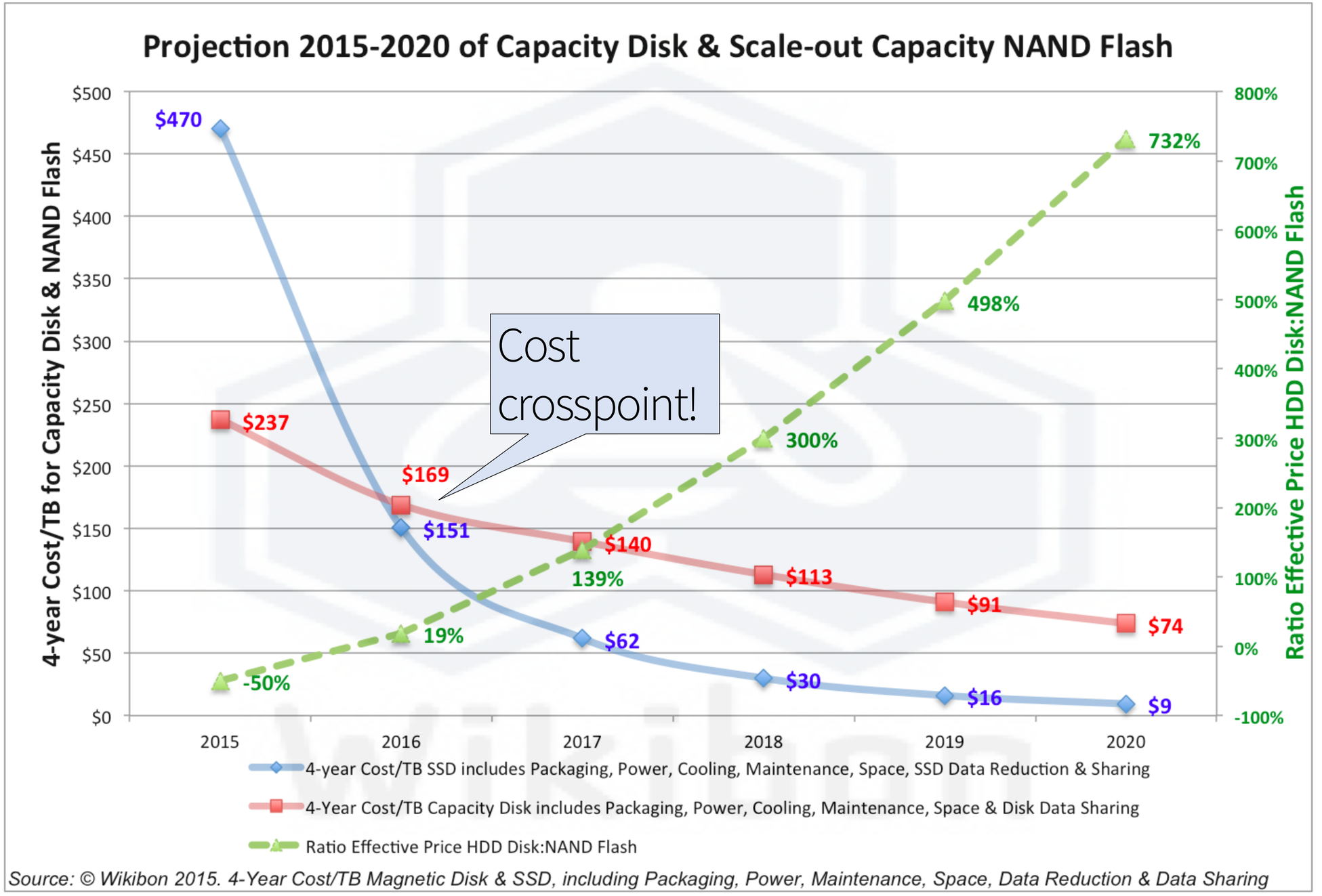


Memory Price/Byte Evolution

- 1990-2000: -54% per year
- 2000-2010: -51% per year
- 2010-2015: -32% per year
- (http://www.jcmit.com/memoryprice.htm)

Typical Server Node





Source: © Wikibon 2015. 4-Year Cost/TB Magnetic Disk & SSD, including Packaging, Power, Maintenance, Space, Data Reduction & Data Sharing

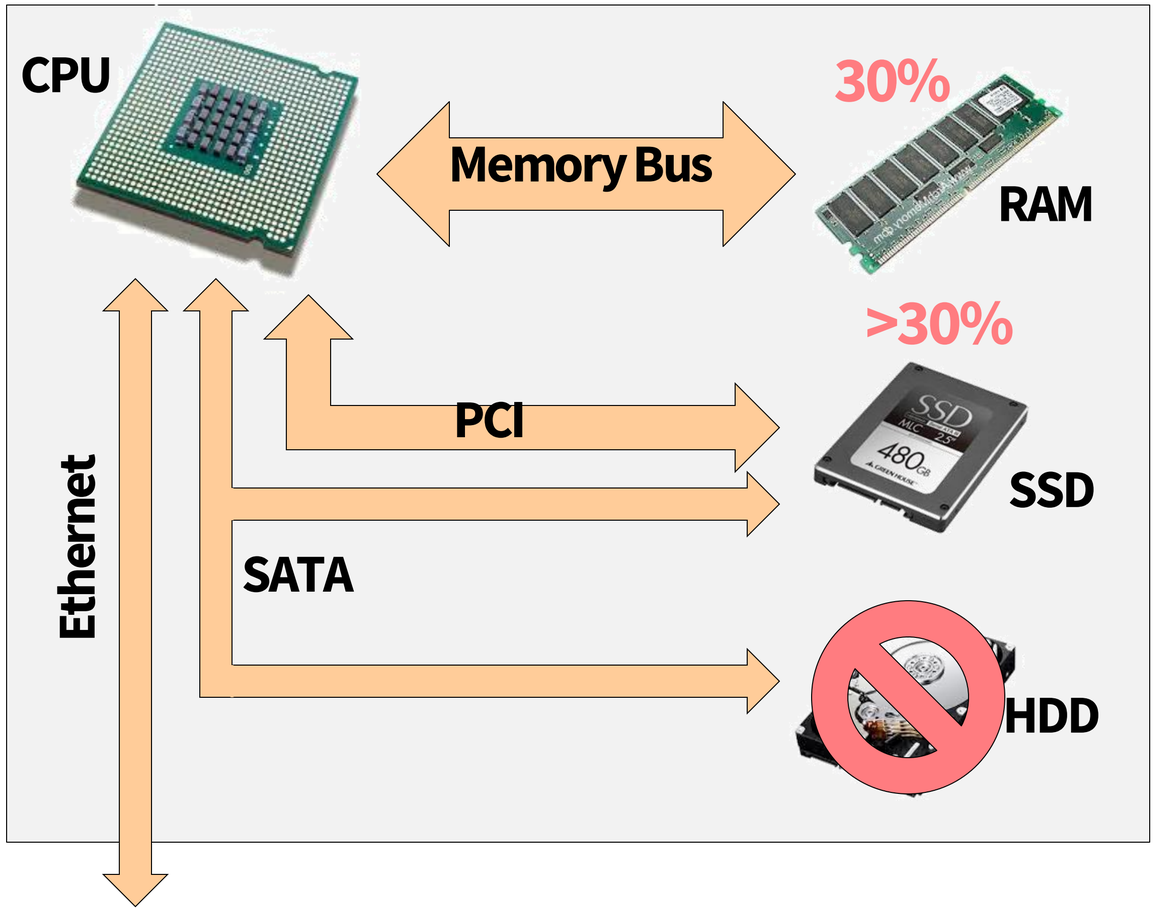
SSDs vs. HDDs

- SSDs has become cheaper than (or as cheap as to) HDDs

- Transition from HDDs to SSDs has accelerate
  - Already most instances in AWS have SSDs
  - Digital Ocean instances are SSD only

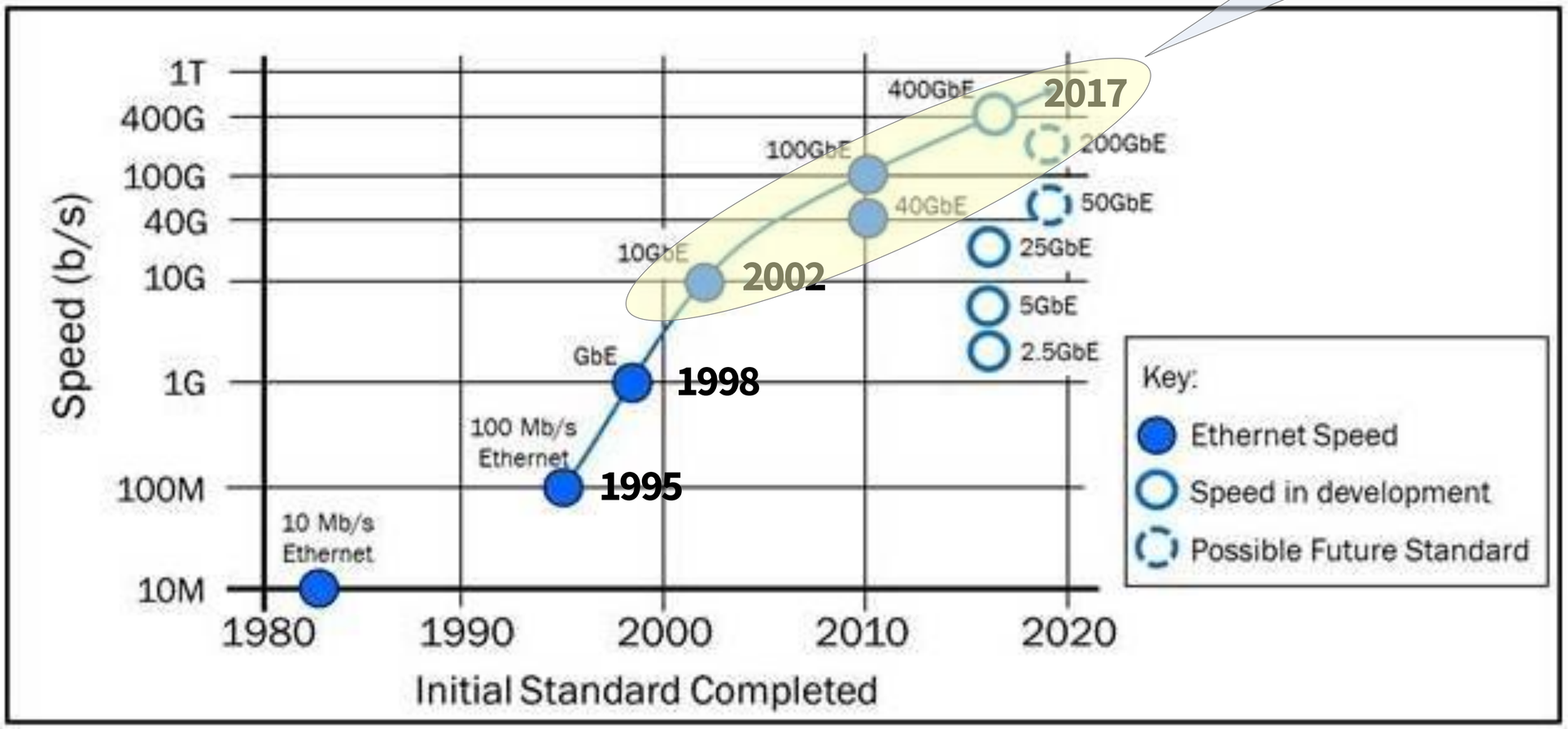
- Going forward we can assume SSD only clusters

Typical Server Node

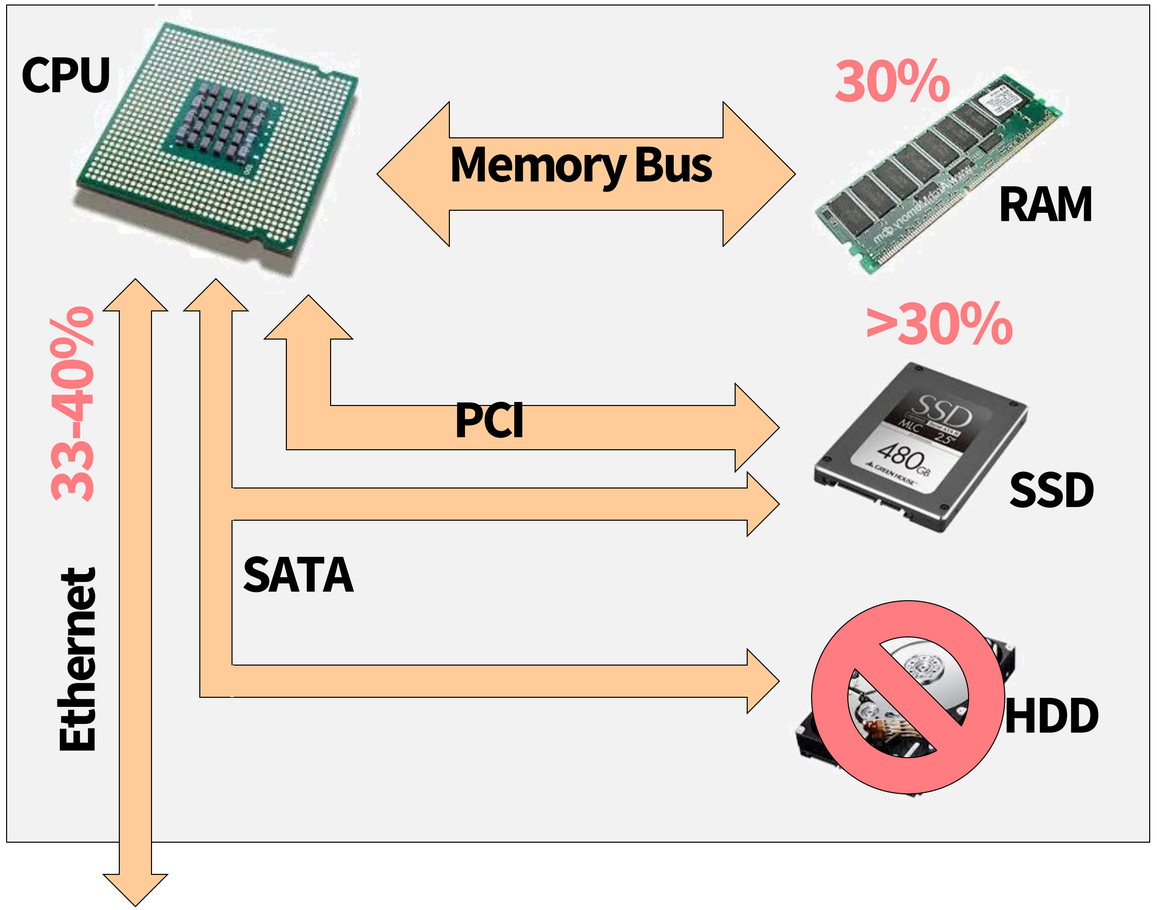


Ethernet Bandwidth

33-40% per year



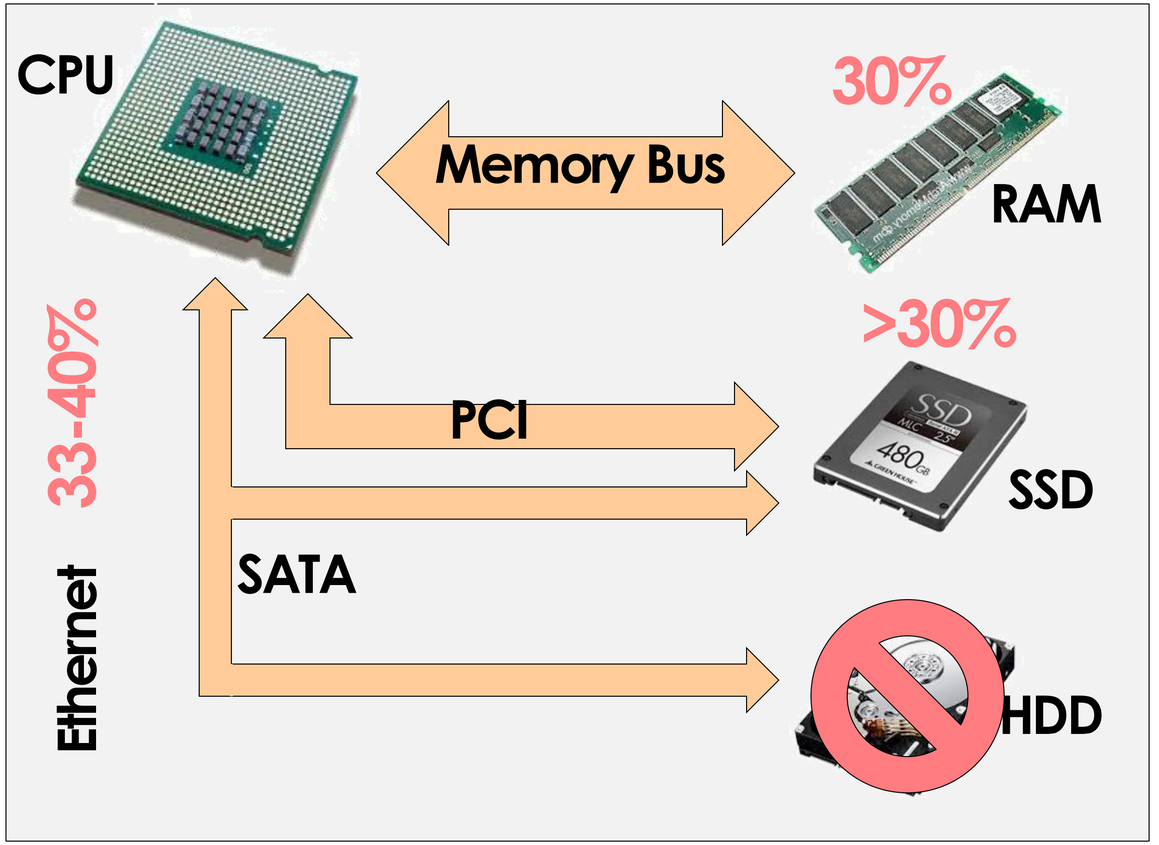
Typical Server Node



What Does This Mean?

- Memory hierarchy has shift one layer up

- HDD is virtually dead
- We have unlimited space of SSD
- Today's RAM space = yesterday's SSD space
- Today's SSD space = yesterday's HDD space
- Ethernet may become faster than PCI/SATA bandwidth



Three Necessary Conditions

- Memory: large (cheap) enough ✅
- Network: fast (cheap) enough ✅
- Fault tolerance: at least as good as map-reduce

Stream Processing

- Computation vs. I/O: Arithmetic intensity
  - Loop fusion
- When MapReduce fails
- **Spark and RDD**
- Spark Ecosystem and Beyond
- Early ML systems: parameter server

Resilient distributed dataset (RDD)

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

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma,
Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

1 Introduction

Cluster computing frameworks like MapReduce [10] and Dryad [19] have been widely adopted for large-scale data analytics. These systems let users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications: those that *reuse* intermediate results across multiple computations. Data reuse is common in many *iterative* machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression. Another compelling use case is *interactive* data mining, where a user runs multiple ad-hoc queries on the same subset of the data. Unfortunately, in most current frameworks, the only way to reuse data between computations (*e.g.,* between two MapReduce jobs) is to write it to an external stable storage system, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (*e.g.,* looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, *e.g.,* to let a user load several datasets into memory and run ad-hoc queries across them.

In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance *efficiently*. Existing abstractions for in-memory storage on clusters, such as distributed shared memory [24], key-value stores [25], databases, and Piccolo [27], offer an interface based on fine-grained updates to mutable state (*e.g.,* cells in a table). With this interface, the only ways to provide fault tolerance are to replicate the data across machines or to log updates across machines. Both approaches are expensive for data-intensive workloads, as they require copying large amounts of data over the cluster network, whose bandwidth is far lower than that of RAM, and they incur substantial storage overhead.

In contrast to these systems, RDDs provide an interface based on *coarse-grained* transformations (*e.g.,* map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its *lineage*) rather than the actual data.[1] If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute

RDD: Spark's key programming abstraction:

- Read-only collection of records (immutable)
- RDDs can only be created by deterministic transformations on data in persistent storage or on existing RDDs

RDDs

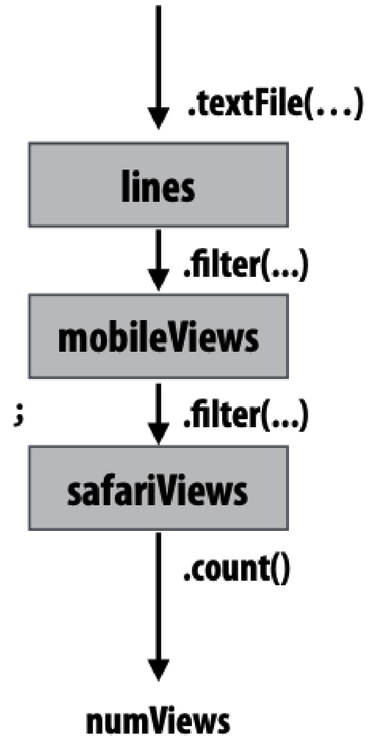
```
// create RDD from file system data
var lines = spark.textFile("hdfs://15418log.txt");

// create RDD using filter() transformation on lines
var mobileViews = lines.filter((x: String) => isMobileClient(x));

// another filter() transformation
var safariViews = mobileViews.filter((x: String) => x.contains("Safari"));

// then count number of elements in RDD via count() action
var numViews = safariViews.count();
```

int



Predefined Set of Operators

Transformation

Action

RDD transformations and actions

Transformations: (data parallel operators taking an input RDD to a new RDD)

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

Actions: (provide data back to the "host" application)

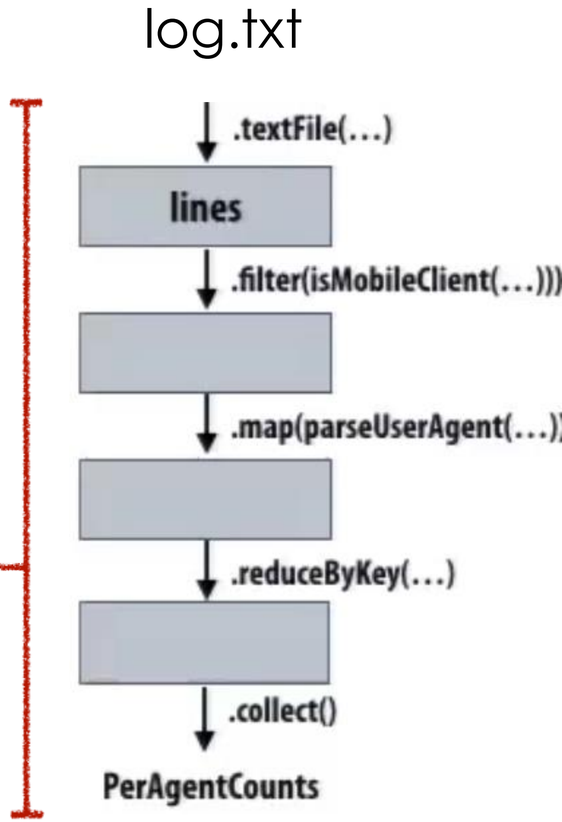
$$
\begin{aligned}
count() &: \mathrm{RDD[T]} \Rightarrow \mathrm{Long} \\
collect() &: \mathrm{RDD[T]} \Rightarrow \mathrm{Seq[T]} \\
reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T}) &: \mathrm{RDD[T]} \Rightarrow \mathrm{T} \\
lookup(k : \mathrm{K}) &: \mathrm{RDD[(K, V)]} \Rightarrow \mathrm{Seq[V]}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

Repeating the map-reduce example

```
// 1. create RDD from file system data
// 2. create RDD with only lines from mobile clients
// 3. create RDD with elements of type (String,Int) from line string
// 4. group elements by key
// 5. call provided reduction function on all keys to count views
var perAgentCounts = spark.textFile("hdfs://log.txt")
                          .filter(x => isMobileClient(x))
                          .map(x => (parseUserAgent(x),1));
                          .reduceByKey((x,y) => x+y)
                          .collect();
```

Array [String,int]

"Lineage": Sequence of RDD operations needed to compute output



Another Spark program

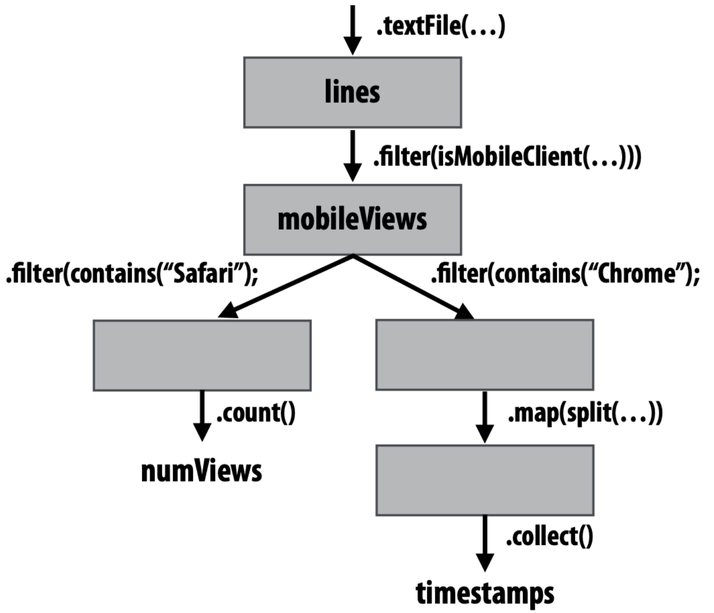
```
// create RDD from file system data
var lines = spark.textFile("hdfs://log.txt");

// create RDD using filter() transformation on lines
var mobileViews = lines.filter((x: String) => isMobileClient(x));

// instruct Spark runtime to try to keep mobileViews in memory
mobileViews.persist();

// create a new RDD by filtering mobileViews
// then count number of elements in new RDD via count() action
var numViews = mobileViews.filter(_.contains("Safari")).count();

// 1. create new RDD by filtering only Chrome views
// 2. for each element, split string and take timestamp of // page view
// 3. convert RDD to a scalar sequence (collect() action)
var timestamps = mobileViews.filter(_.contains("Chrome"))
                            .map(_.split(" ")(0))
                            .collect();
```



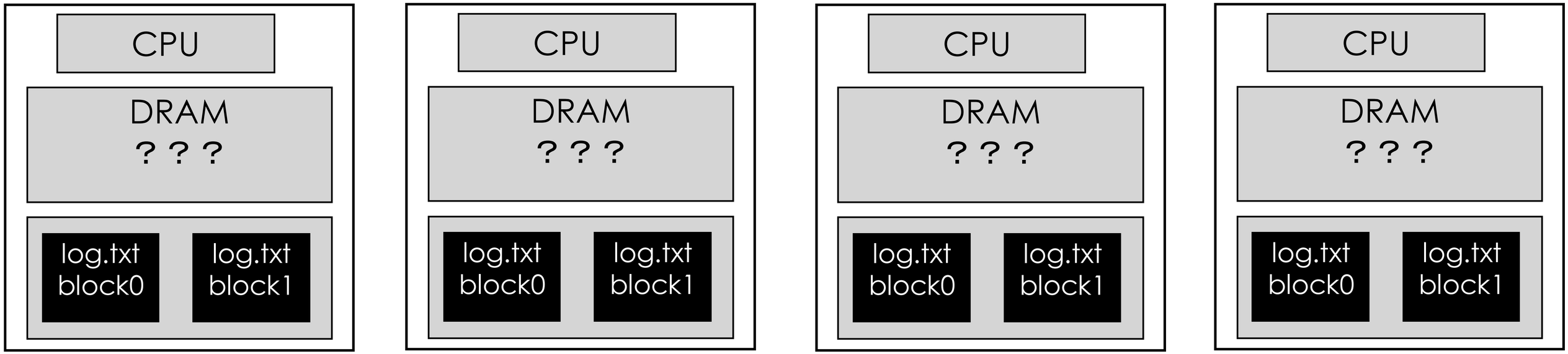
Discussion

- How do you like this programming model?
- v.s. map reduce
  - Flexibility and Expressiveness?
  - Simplicity?

How do we implement RDDs?

- In particular, how should they be stored?
  - var lines = spark.textFile("hdfs://log.txt");
  - var lower = lines.map(_.toLower());
  - var mobileViews = lower.filter(x => isMobileClient(x));
  - var howMany = mobileViews.count();

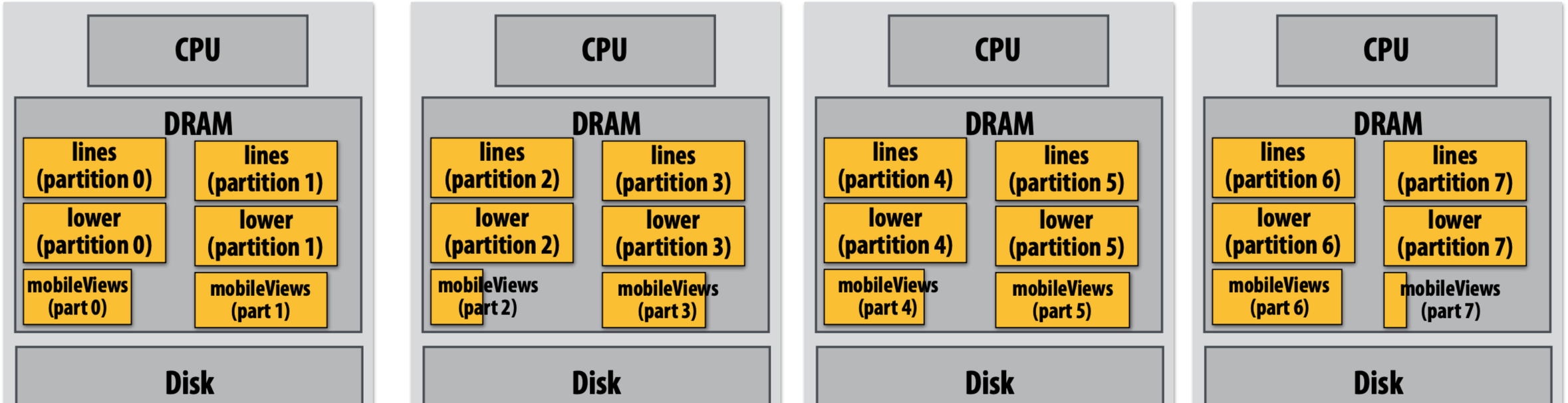
Question: should we think of RDD's like arrays?



How do we implement RDDs?

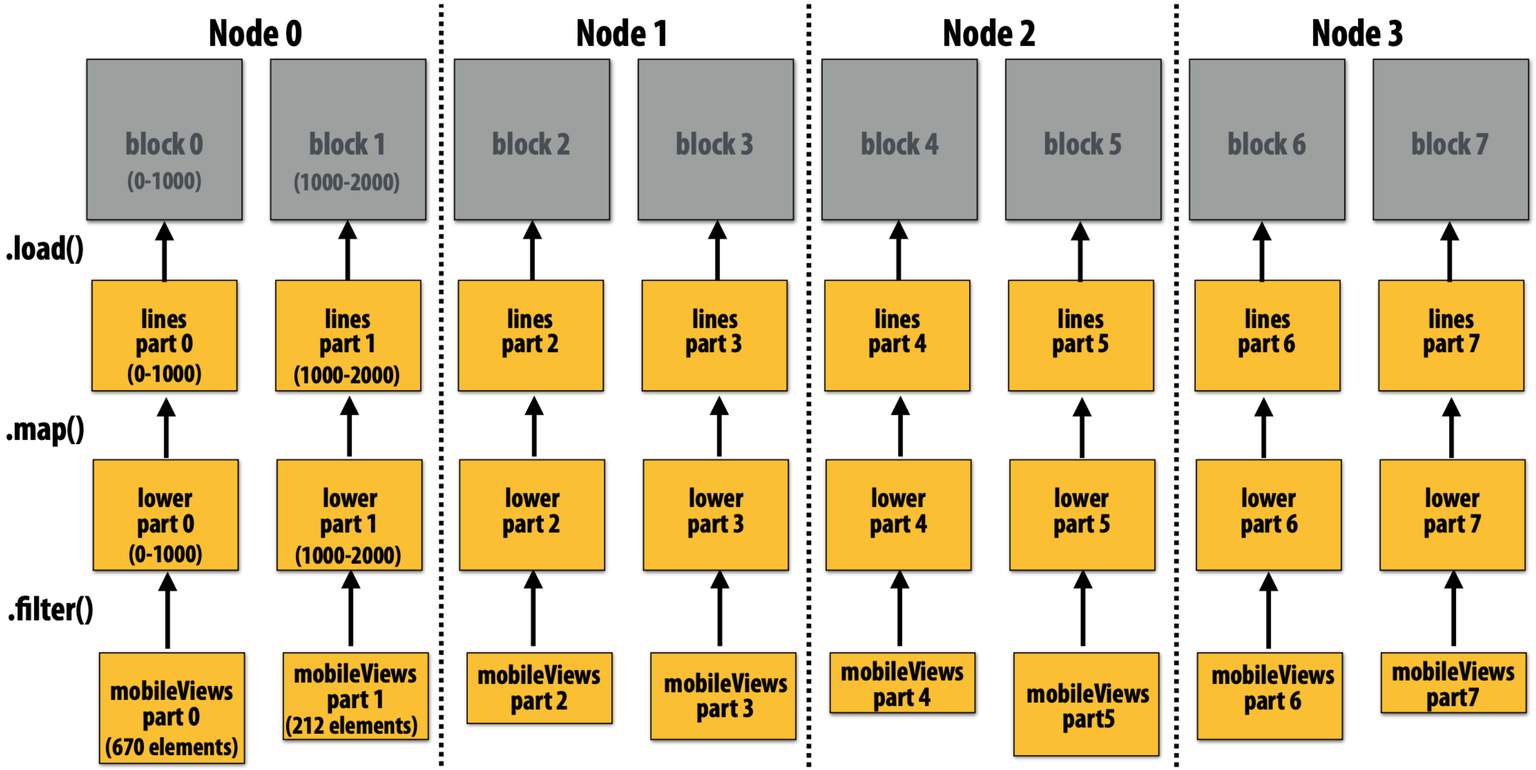
- In particular, how should they be stored?
  - var lines = spark.textFile("hdfs://log.txt");
  - var lower = lines.map(_.toLower());
  - var mobileViews = lower.filter(x => isMobileClient(x));
  - var howMany = mobileViews.count();

Question: Array -> In-memory representation would be huge! (larger than original flie on disk)



RDD partitioning and dependencies

```
var lines = spark.textFile("hdfs://log.txt");
var lower = lines.map(_.toLower());
var mobileViews = lower.filter(x => isMobileClient(x));
var howMany = mobileViews.count();
```



Black lines show dependencies between RDD partitions.

Implementing sequence of RDD ops efficiently

```
var lines = spark.textFile(“hdfs://log.txt”);
var lower = lines.map(_.toLower());
var mobileViews = lower.filter(x => isMobileClient(x));
var howMany = mobileViews.count();
```

- Recall “loop fusion” from start of lecture
- The following code stores only a line of the log file in memory, and only reads input data from disk once (“streaming” solution)

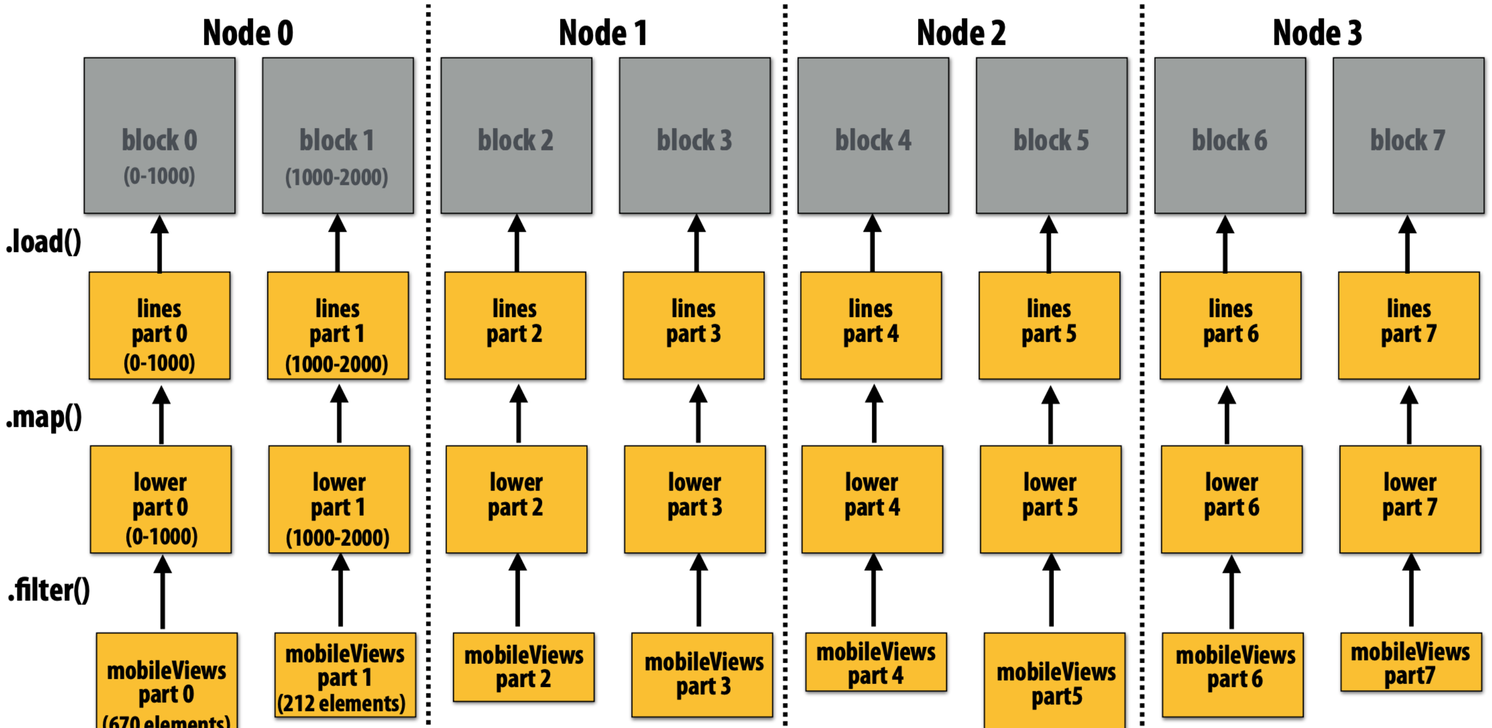
```
int count = 0;
while (inputFile.eof()) {
   string line = inputFile.readLine();
   string lower = line.toLower;
   if (isMobileClient(lower))
      count++;
}
```

Narrow dependencies

```
var lines = spark.textFile("hdfs://log.txt");
var lower = lines.map(_.toLower());
var mobileViews = lower.filter(x => isMobileClient(x));
var howMany = mobileViews.count();
```

"Narrow dependencies" = each partition of parent RDD referenced by at most one child RDD partition

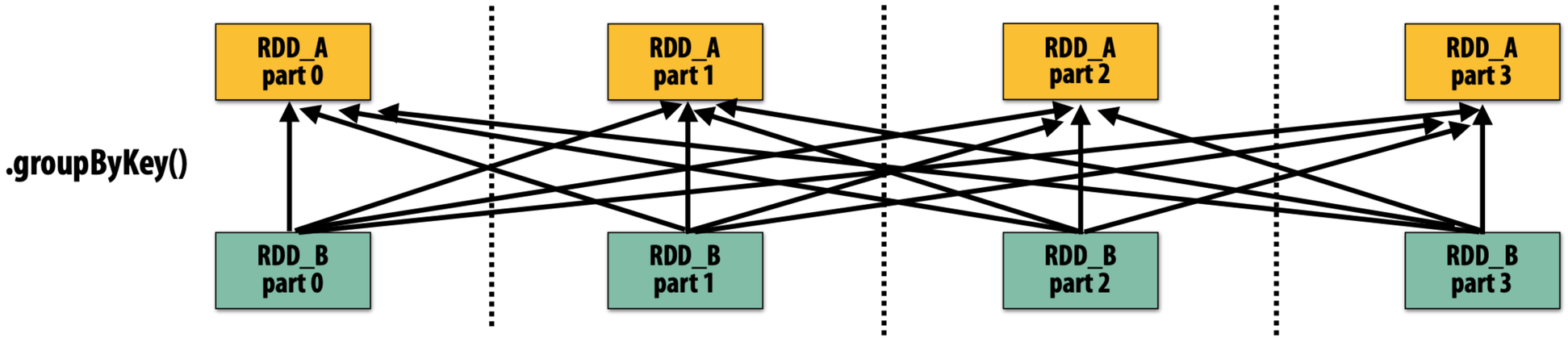
- Allows for fusing of operations
  (here: can apply map and then filter all at once on input element)
- In this example: no communication between nodes of cluster
  (communication of one int at end to perform count() reduction)



Wide dependencies

groupByKey: RDD[(K,V)] → RDD[(K,Seq[V])]

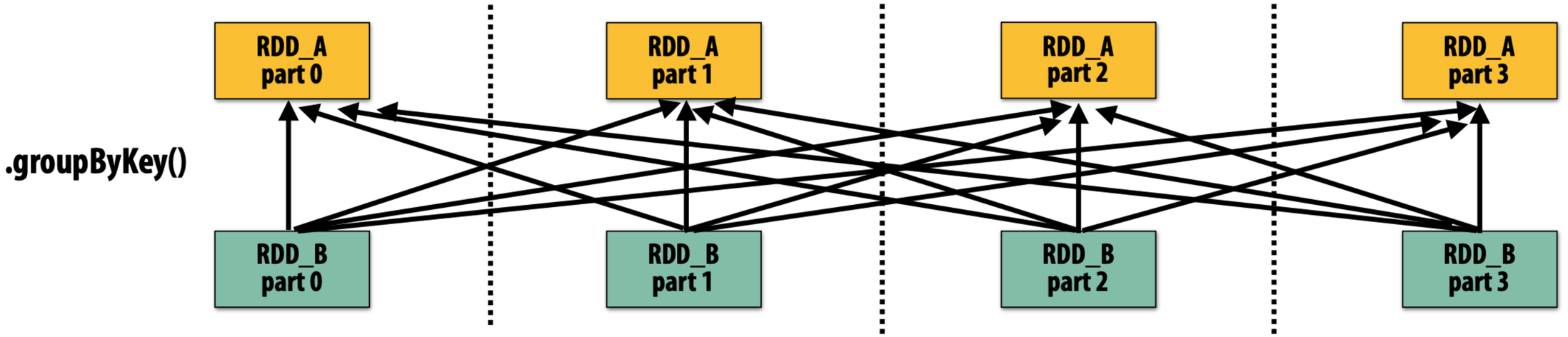
"Make a new RDD where each element is a sequence containing all values from the parent RDD with the same key."



Wide dependencies = each partition of parent RDD referenced by multiple child RDD partitions

Wide dependencies

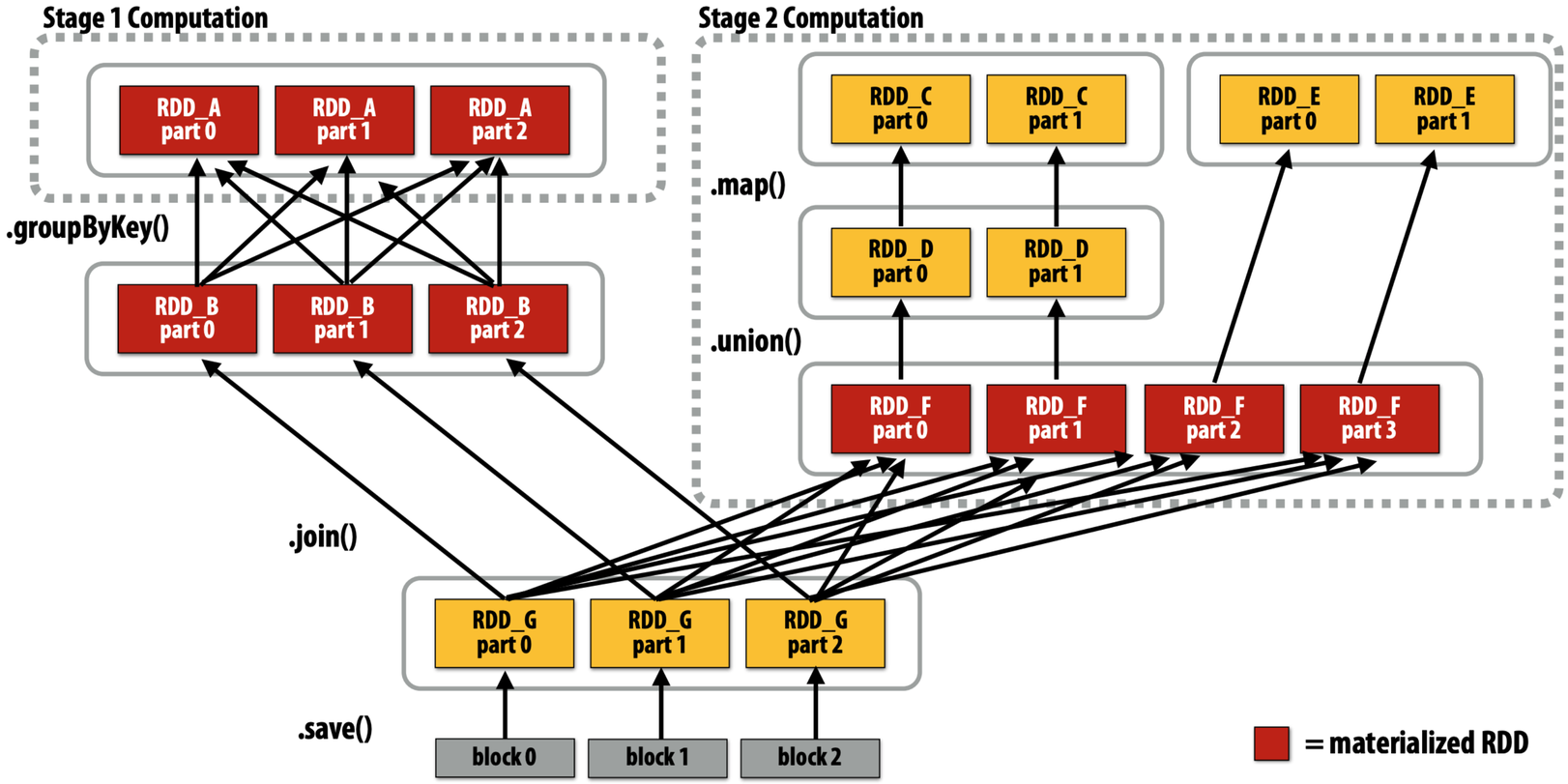
Wide dependencies = each partition of parent RDD referenced by multiple child RDD partitions



Challenges:

- Must compute all of RDD_A before computing RDD_B
  - Example: groupByKey() may induce all-to-all communication as shown above
- May trigger significant recompilation of ancestor lineage upon node failure

Scheduling Spark computations

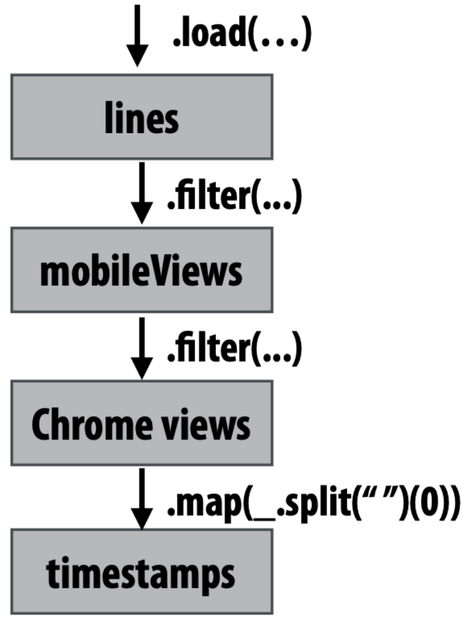


- Actions (e.g., save()) trigger evaluation of Spark lineage graph.
  - Stage 1 Computation: do nothing since input already materialized in memory
  - Stage 2 Computation: evaluate map in fused manner, only actually materialize RDD F
  - Stage 3 Computation: execute join (could stream the operation to disk, do not need to materialize)

Implementing resilience via lineage

- RDD transformations are bulk, deterministic, and functional
  - Implication: runtime can always reconstruct contents of RDD from its lineage (the sequence of transformations used to create it)
  - Lineage is a log of transformations
  - Efficient: since log records bulk data-parallel operations, overhead of logging is low (compared to logging fine-grained operations, like in a database)

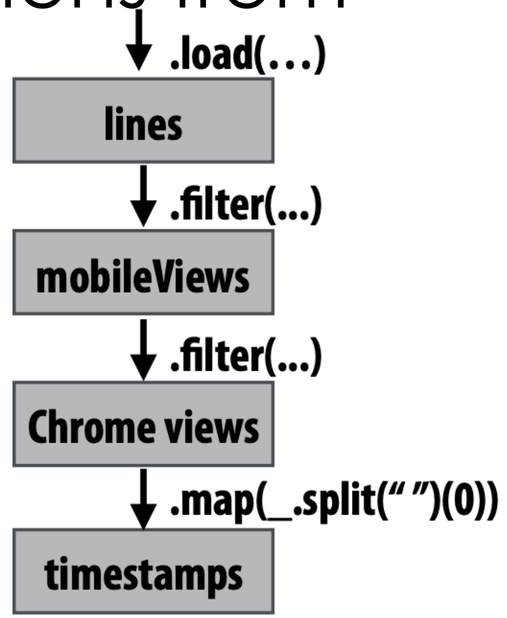
```
// create RDD from file system data
var lines = spark.textFile("hdfs://15418log.txt");
// create RDD using filter() transformation on lines
var mobileViews = lines.filter((x: String) => isMobileClient(x));
// 1. create new RDD by filtering only Chrome views
// 2. for each element, split string and take timestamp of // page view (first element)
// 3. convert RDD To a scalar sequence (collect() action)
var timestamps = mobileView.filter(_.contains("Chrome")) .map(_.split(" ")(0));
```

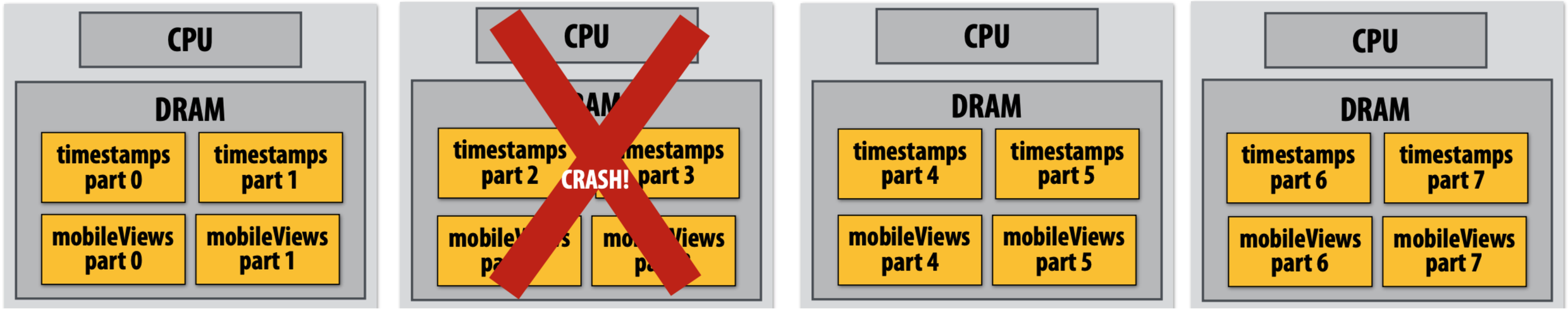


Upon node failure: recompute lost RDD partitions from lineage

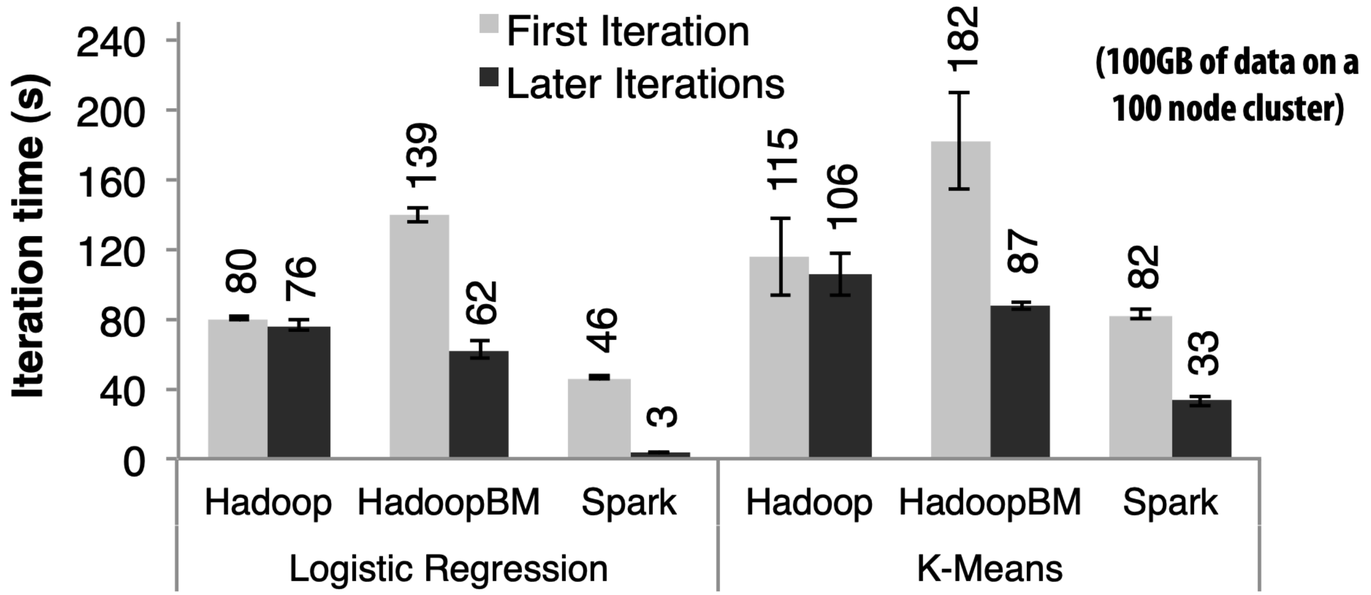
Must reload required subset of data from disk and recompute entire sequence of operations given by lineage to regenerate partitions 2 and 3 of RDD timestamps.

Note: (not shown): file system data is replicated so assume blocks 2 and 3 remain accessible to all nodes





Spark Performance



Spark Improves MapReduce Over

- Easy for programmers because you express your computation by chaining atomic operators
- Much fewer I/O -> very improved AI

Spark Cons?

- Debuggability
- Bulky
  - Map-reduce is not bulky as it works well if you only have one worker. That’s why now every PL has a “map” function

Modern Spark ecosystem

Compelling feature: enables integration/composition of multiple domain-specific frameworks (since all collections implemented under the hood with RDDs and scheduled using Spark scheduler)


```
sqlCtx = new HiveContext(sc)
results = sqlCtx.sql(
  "SELECT * FROM people")
names = results.map(lambda p: p.name)
```

Interleave computation and database query
Can apply transformations to RDDs produced by SQL queries

Machine learning library build on top of Spark abstractions.

```
points = spark.textFile("hdfs://...")
              .map(parsePoint)

model = KMeans.train(points, k=10)
```

GraphLab-like library built on top of Spark abstractions.

```
graph = Graph(vertices, edges)
messages = spark.textFile("hdfs://...")
graph2 = graph.joinVertices(messages) {
  (id, vertex, msg) => ...
}
```

Story time: Spark and Databricks

- Initially just an open-source project by a few students
- The community grows because of advantages over Hadoop and Map-reduce
- Students were about to graduate and could not commit time to those projects, what's next?
- "We asked Hortonworks if they wanted to take over Spark…They were not willing… We started Databricks."
- Hortonworks -> later merged with Cloudera at 2019

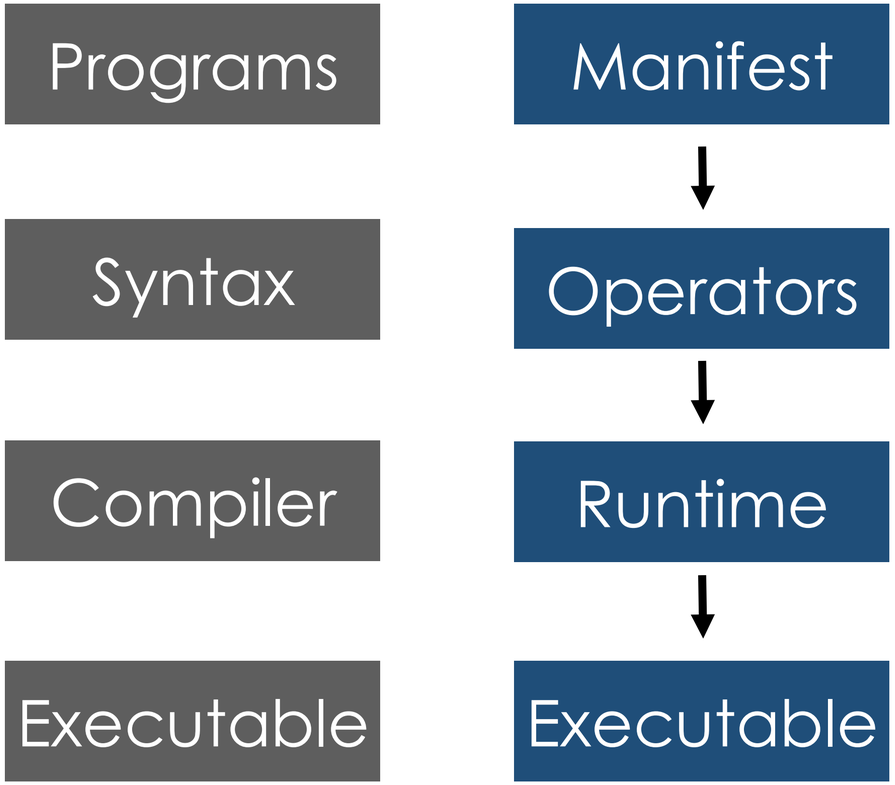
Spark and Databricks

- Cloudera: data platform company, founded by Hadoop authors
  - Used to be a unicorn / high-profile / high-tech company
  - Was beat hard by Databricks / Snowflake
  - Went to public 2017, stock price keeps declining…, merged with Hortonworks in 2018, went to private in 2021 after being acquired by investment companies.
- Databricks: 7 cofounders, Initial CEO is Prof. Ion Stoica.
  - They tried to sell Spark but were unsuccessful
  - Switched to Ali Ghodsi: Iranian-Swedish, visitor to UC Berkeley, no US-born nor US-educated

Spark and Databricks

- Databricks struggled for quite a few years
  - Raised up to Series I (Seed, A, B, C, D, E, F, G, H, I)
  - Almost failed during 2018 – 2020
  - Data warehousing and OLAP gradually become a business, why?
    - Competitors all failed
    - Customer Education
    - Data indeed bigger and bigger
  - Intended to go public in 2022, but hit covid
  - Valued at 100B today (is there any bubble?)
  - Create 7 billionaires
  - Competitions with Snowflake are intense

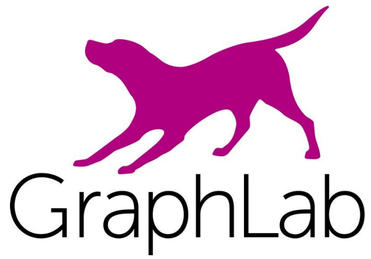
After Spark:
All Modern Data/ML Systems follow a similar architecture



A fixed set of operators

A trusted runtime with a small set of pre-loaded implementations

After Spark: Many new systems



Naiad