Slides Credit: Matthias Boehm

SCIENCE
PASSION
TECHNOLOGY

Data Integration and Large Scale Analysis
10 Distributed Data-Parallel Computation

Shafaq Siddiqi

Graz University of Technology, Austria

Last update: Jan 12, 2024

Announcements/Org

- **Course Evaluation and Exam**
  - Evaluation period: **till Feb 15**
  - Exam date: Feb 02, 3:00pm (90 min written exam)
  - Oral Exam for Erasmus Students
    - Schedule available in TeachCenter

Course Outline Part B:
Large-Scale Data Management and Analysis

11 Distributed Stream Processing

12 Distributed Machine Learning Systems

Compute/ Storage

10 Distributed Data-Parallel Computation

09 Distributed Data Storage

Infra

08 Cloud Resource Management and Scheduling

07 Cloud Computing Fundamentals

Agenda

- **Motivation and Terminology**
- **Data-Parallel Collection Processing**
- **Data-Parallel DataFrame Operations**
- **Data-Parallel Computation in SystemDS**

Motivation and Terminology

Recap: Central Data Abstractions

- **#1 Files and Objects**
  - **File:** Arbitrarily large sequential data in specific file format (CSV, binary, etc)
  - **Object:** binary large object, with certain meta data

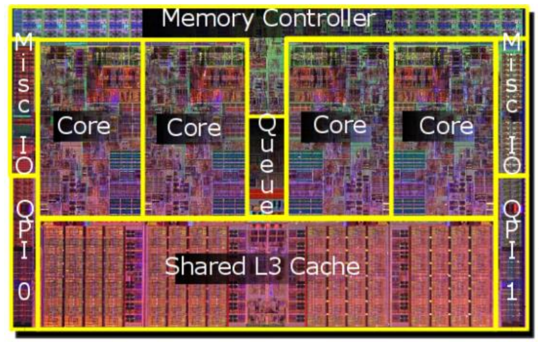
- **#2 Distributed Collections**
  - Logical multi-set (**bag**) of **key-value pairs** (**unsorted collection**)
  - Different physical representations
  - **Easy distribution** of pairs via horizontal partitioning (aka shards, partitions)
  - Can be created from single file, or directory of files (unsorted)

Key	Value
4	Delta
2	Bravo
1	Alfa
3	Charlie
5	Echo
6	Foxtrot
7	Golf
1	Alfa

Excursus: Nehalem Architecture

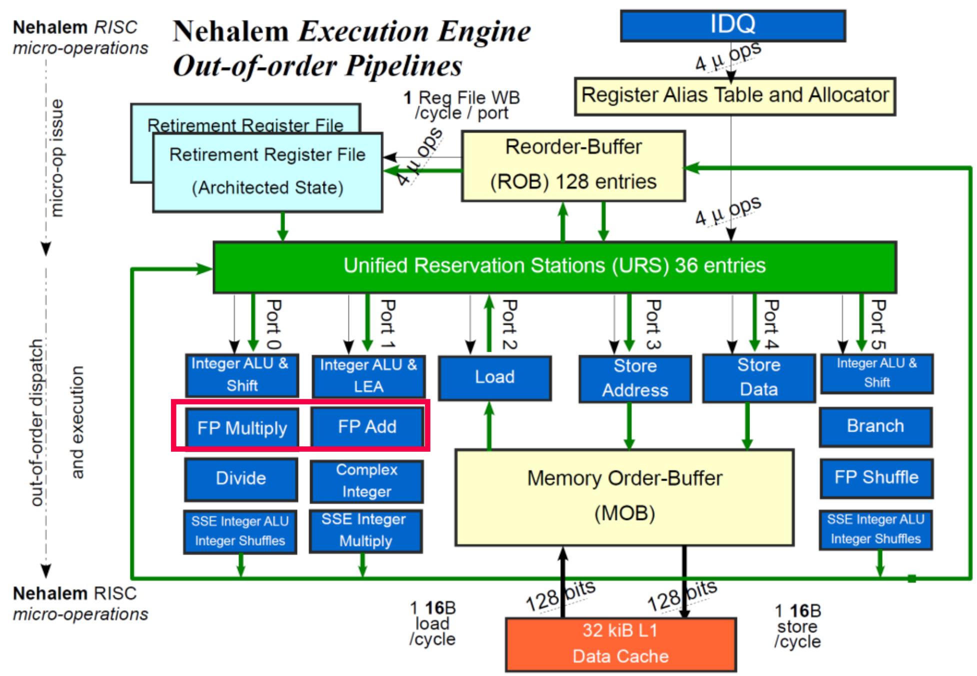
[Michael E. Thomadakis: The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms, Report, 2010]

- **Multi-core CPU**
  - 4 core w/ hyper-threading
  - **Per core:** L1i/L1d, L2 cache
  - **Per CPU:** L3 cache (8MB)
  - 3 memory channels (8B width, max 1.333Ghz)



QPI … Quick Path Interconnect

- **Pipeline**
  - **Frontend:** Instruction Fetch, Pre-Decode, and Decode
  - **Backend:** Rename/Allocate, Scheduler, Execute
- **Out-of-Order Execution Engine** (IPC=4)
  - 128b FP Multiply
  - 128b FP Add



Terminology

- **Flynn's Classification**
  - SISD, SIMD
  - (MISD), MIMD

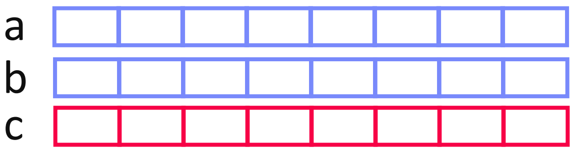
[Michael J. Flynn, Kevin W. Rudd: Parallel Architectures. **ACM Comput. Surv. 28(1) 1996**]

	Single Data	Multiple Data
Single Instruction	**SISD** (uni-core)	**SIMD** (vector)
Multiple Instruction	**MISD** (pipelining)	**MIMD** (multi-core)

- **Example: SIMD Processing**
  - Streaming SIMD Extensions (SSE)
  - Process the same operation on multiple elements at a time (**packed** vs scalar SSE instructions)
  - Data parallelism (aka: instruction-level parallelism)
  - Example: **VFMADD132PD**

2009 Nehalem: **128b** (2xFP64)
2012 Sandy Bridge: **256b** (4xFP64)
2017 Skylake: **512b** (8xFP64)

```
c = _mm512_fmadd_pd(a, b);
```



Terminology cont.

- **Distributed, Data-Parallel Computation**

```
Y = X.map(x -> foo(x))
```

  - Parallel computation of function foo() ➔ **single instruction**
  - Collection X of data items (key-value pairs) ➔ **multiple data**
  - Data parallelism similar to **SIMD** but more coarse-grained notion of "instruction" and "data" ➔ **SPMD** (single program, multiple data)

[Frederica Darema: The SPMD Model : Past, Present and Future. **PVM/MPI 2001**]

- **Additional Terminology**
  - **BSP:** Bulk Synchronous Parallel (global barriers)
  - **ASP:** Asynchronous Parallel (no barriers, often with accuracy impact)
  - **SSP:** Stale-synchronous parallel (staleness constraint on fastest-slowest)
  - Other: Fork&Join, Hogwild!, event-based, decentralized

- **Beware:** **data parallelism** used in very different contexts (e.g., Param Server)

Data-Parallel Collection Processing

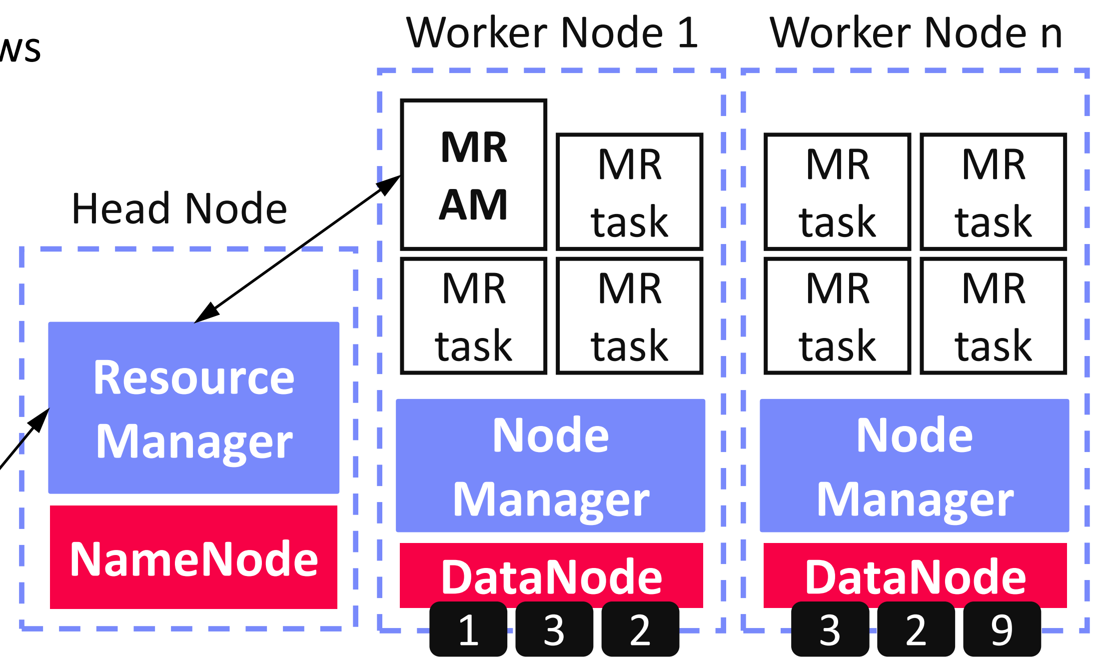
Hadoop History and Architecture

[Jeffrey Dean, Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters. **OSDI 2004**]

- **Recap: Brief History**
  - Google's GFS [SOSP'03] + MapReduce → Apache Hadoop (2006)
  - Apache Hive (SQL), Pig (ETL), Mahout (ML), Giraph (Graph)

- **Hadoop Architecture / Eco System**
  - Management (Ambari)
  - Coordination / workflows (Zookeeper, Oozie)
  - Storage (**HDFS**)
  - Resources (**YARN**) [SoCC'13]
  - Processing (**MapReduce**)

MR Client



MapReduce – Programming Model

- **Overview Programming Model**
  - Inspired by functional programming languages
  - **Implicit parallelism** (abstracts distributed storage and processing)
  - Map function: key/value pair → set of intermediate key/value pairs
  - Reduce function: merge all intermediate values by key

- **Example** `SELECT Dep, count(*) FROM csv_files GROUP BY Dep`

Name	Dep
X	CS
Y	CS
A	EE
Z	CS

Collection of key/value pairs

```
map(Long pos, String line) {
   parts ← line.split(“,”)
   emit(parts[1], 1)
}
```

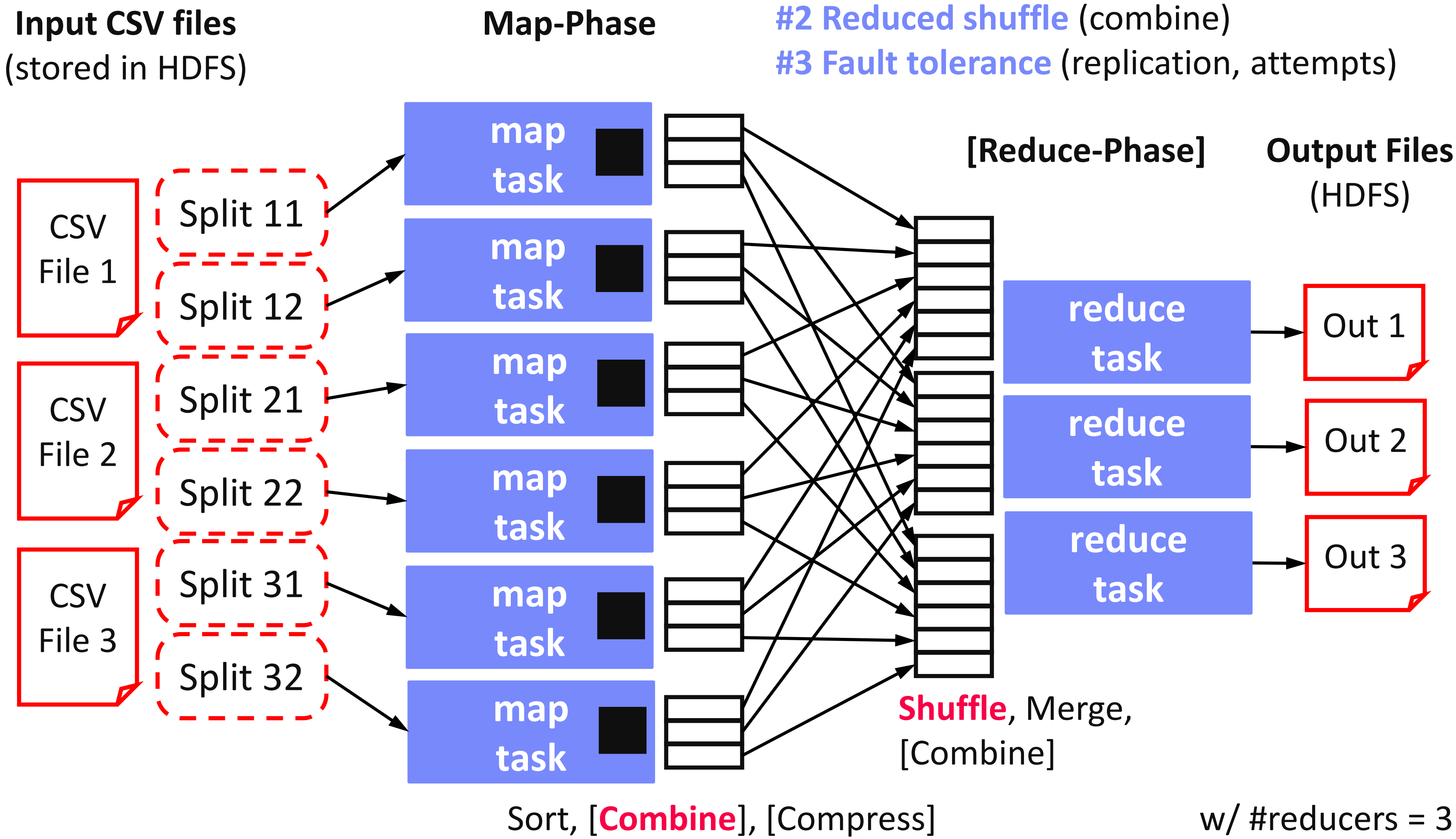
CS	1
CS	1
EE	1
CS	1

```
reduce(String dep,
       Iterator<Long> iter) {
   total ← iter.sum();
   emit(dep, total)
}
```

CS	3
EE	1

MapReduce – Execution Model

#1 Data Locality (delay sched., write affinity)
#2 Reduced shuffle (combine)
#3 Fault tolerance (replication, attempts)



MapReduce – Query Processing

- **Basic Unary Operations**
  - Selections (brute-force), projections
  - Ordering (e.g., **TeraSort**): Sample, pick k quantiles; shuffle-based partition sort
  - Additive and semi-additive aggregation with grouping, distinct

- **Binary Operations**
  - Set operations
    (union, intersect, difference) and joins
  - Different physical operators for R ⋈ S
    - **Broadcast join**: broadcast S, build HT S, map-side HJOIN
    - **Repartition join**: shuffle (repartition) R and S, reduce-side MJOIN
    - **Improved repartition join:** avoid buffering via key-tag sorting
    - **Directed join** (pre/co-partitioned): map-only, R input, S read side-ways

[Spyros Blanas et al.: A comparison of join algorithms for log processing in MapReduce. **SIGMOD 2010**]

- **Hybrid SQL-on-Hadoop Systems** [VLDB'15]
  - E.g.: Hadapt (HadoopDB), Impala, IBM BigSQL, Presto, Drill, Actian

Spark History and Architecture

- **Summary MapReduce**
  - Large-scale & fault-tolerant processing w/ UDFs and files → **Flexibility**
  - Restricted functional APIs → **Implicit parallelism and fault tolerance**
  - **Criticism**: #1 **Performance**, #2 **Low-level APIs**, #3 **Many different systems**

- **Evolution to Spark** (and Flink)
  - Spark [HotCloud'10] + RDDs [NSDI'12] → **Apache Spark** (2014)
  - **Design: standing executors with in-memory storage**, lazy evaluation, and fault-tolerance via RDD lineage
  - **Performance:** In-memory storage and fast job scheduling (100ms vs 10s)
  - **APIs:** Richer functional APIs and general computation DAGs, high-level APIs (e.g., DataFrame/Dataset), unified platform

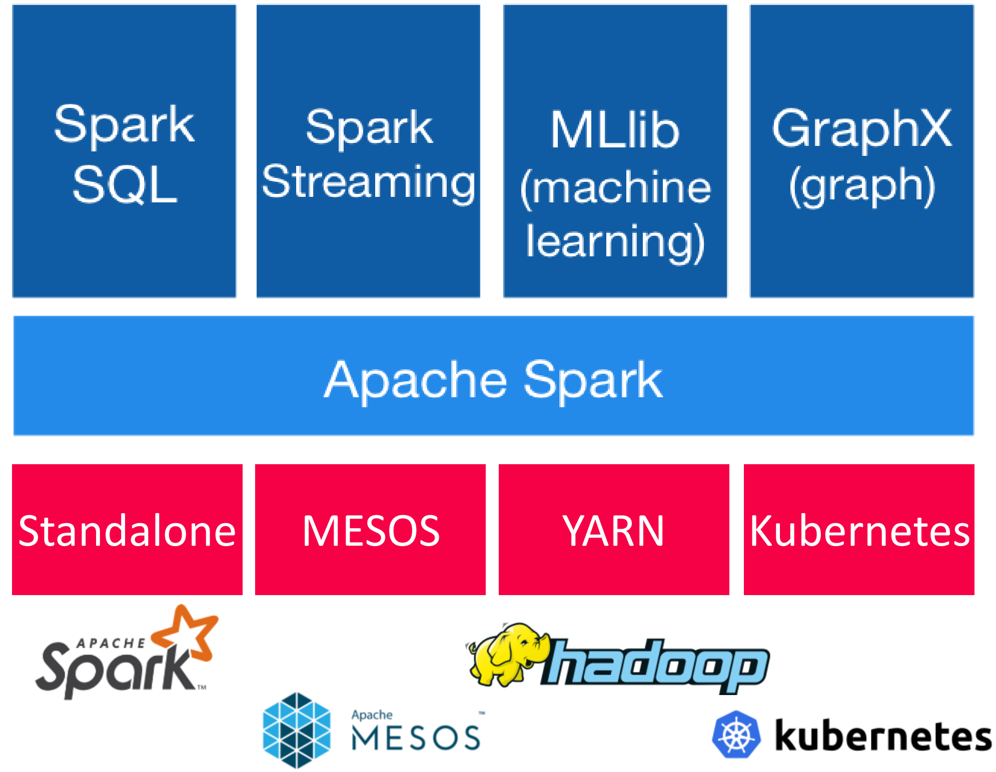
→ **But many shared concepts/infrastructure**

  - **Implicit parallelism through dist. collections** (data access, fault tolerance)
  - Resource negotiators (YARN, Mesos, Kubernetes)
  - HDFS and object store connectors (e.g., Swift, S3)

Spark History and Architecture, cont.

- **High-Level Architecture**
  - **Different language bindings**: Scala, Java, Python, R
  - **Different libraries**: SQL, ML, Stream, Graph
  - Spark core (incl RDDs)
  - **Different cluster managers**: Standalone, Mesos, **Yarn**, **Kubernetes**
  - Different file systems/ formats, and data sources: **HDFS**, **S3**, SWIFT, **DBs**, **NoSQL**

[https://spark.apache.org/]



- **Focus on a unified platform for data-parallel computation** (**Apache Flink** w/ similar goals)

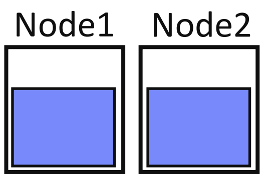
Spark Resilient Distributed Datasets (RDDs)

- **RDD Abstraction**

  JavaPairRDD<MatrixIndexes,MatrixBlock>
  - **Immutable**, partitioned **collections of key-value pairs**
  - **Coarse-grained** deterministic operations (transformations/actions)
  - Fault tolerance via lineage-based re-computation
- **Operations**
  - Transformations: define new RDDs
  - Actions: return result to driver

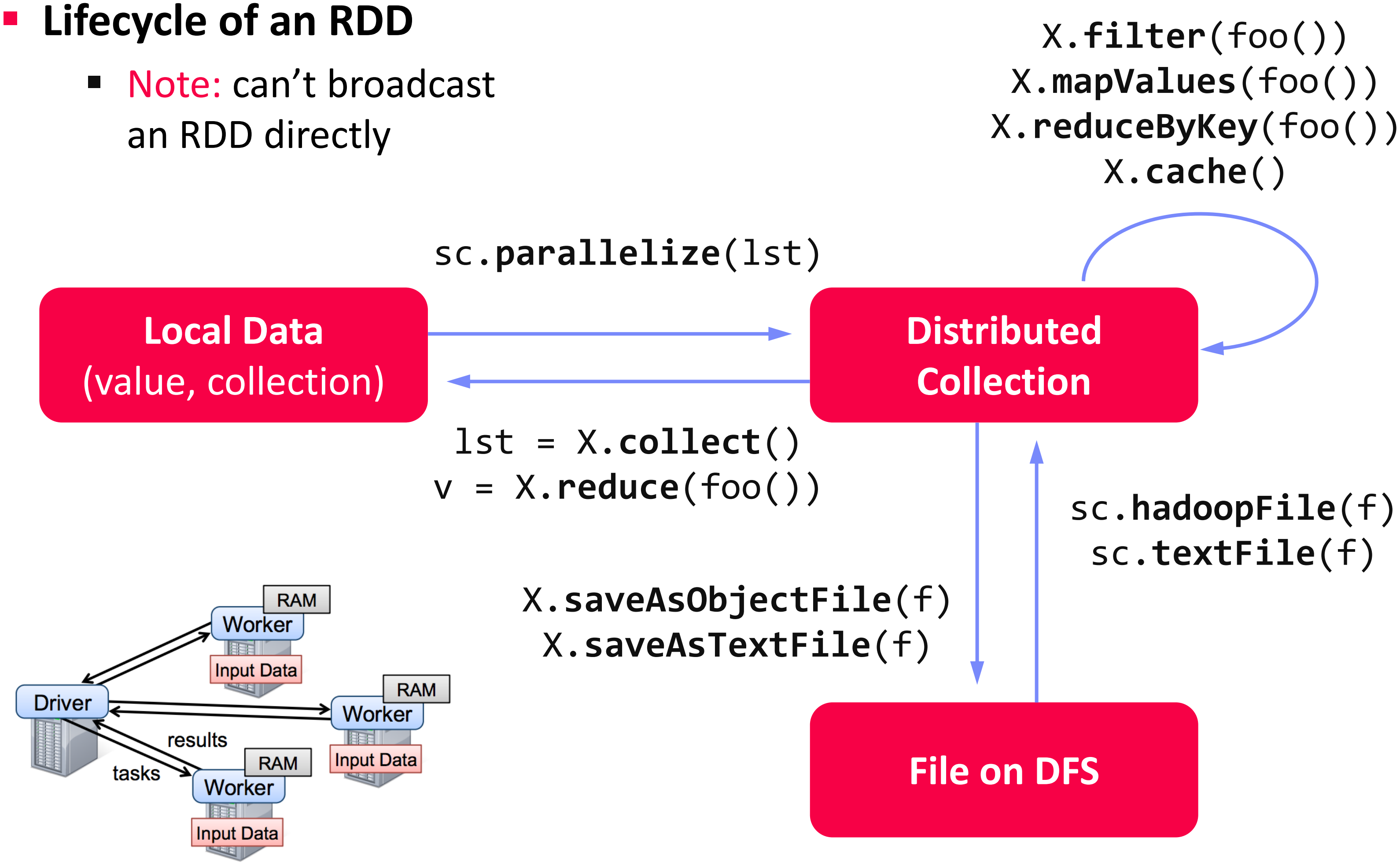
Type	Examples
Transformation (**lazy**)	**map**, hadoopFile, textFile, flatMap, filter, sample, join, groupByKey, cogroup, reduceByKey, cross, sortByKey, mapValues
Action	**reduce**, save, collect, count, lookupKey

- **Distributed Caching**
  - Use fraction of worker **memory for caching**
  - Eviction at granularity of individual partitions
  - **Different storage levels** (e.g., mem/disk x serialization x compression)



Spark Resilient Distributed Datasets (RDDs), cont.

- **Lifecycle of an RDD**
  - Note: can't broadcast an RDD directly



Spark Partitions and Implicit/Explicit Partitioning

- **Spark Partitions**
  - Logical key-value collections are split into **physical partitions**
  - Partitions are granularity of **tasks, I/O, shuffling, evictions**

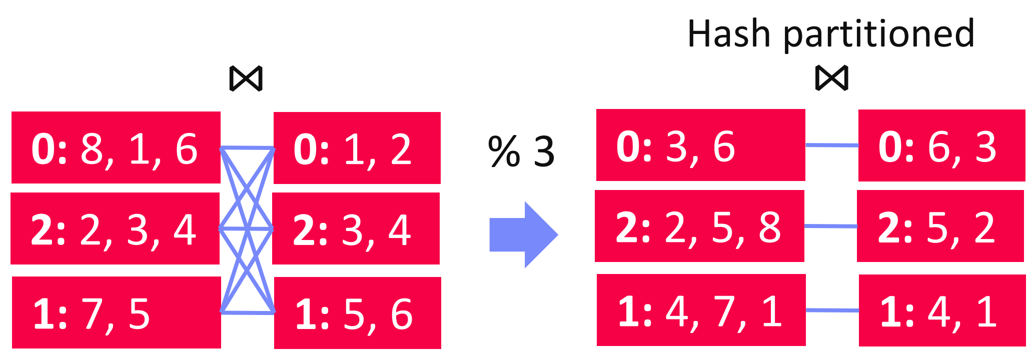
~128MB

- **Partitioning via Partitioners**
  - Implicitly on every data shuffling
  - Explicitly via `R.repartition(n)`

Example Hash Partitioning:
For all (k,v) of R:
pid = hash(k) % n

- **Partitioning-Preserving**
  - All operations that are guaranteed to keep keys unchanged (e.g. `mapValues()`, `mapPartitions()` w/ preservesPart flag)

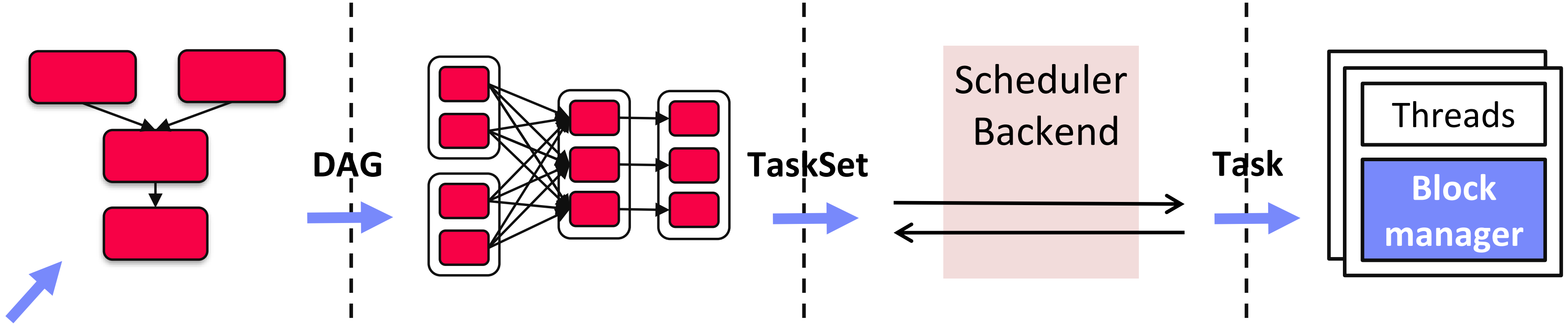
- **Partitioning-Exploiting**
  - Join: R3 = R1.join(R2)
  - Lookups: `v = C.lookup(k)`



Spark Scheduling Process

[Tilmann Rabl: Big Data Systems, HPI **WS2019/20**]

RDD Objects | **DAGScheduler** | **TaskScheduler** | **Workers**




```
rdd1.join(rdd2)
    .reduceByKey(…)
    .filter(…)
```

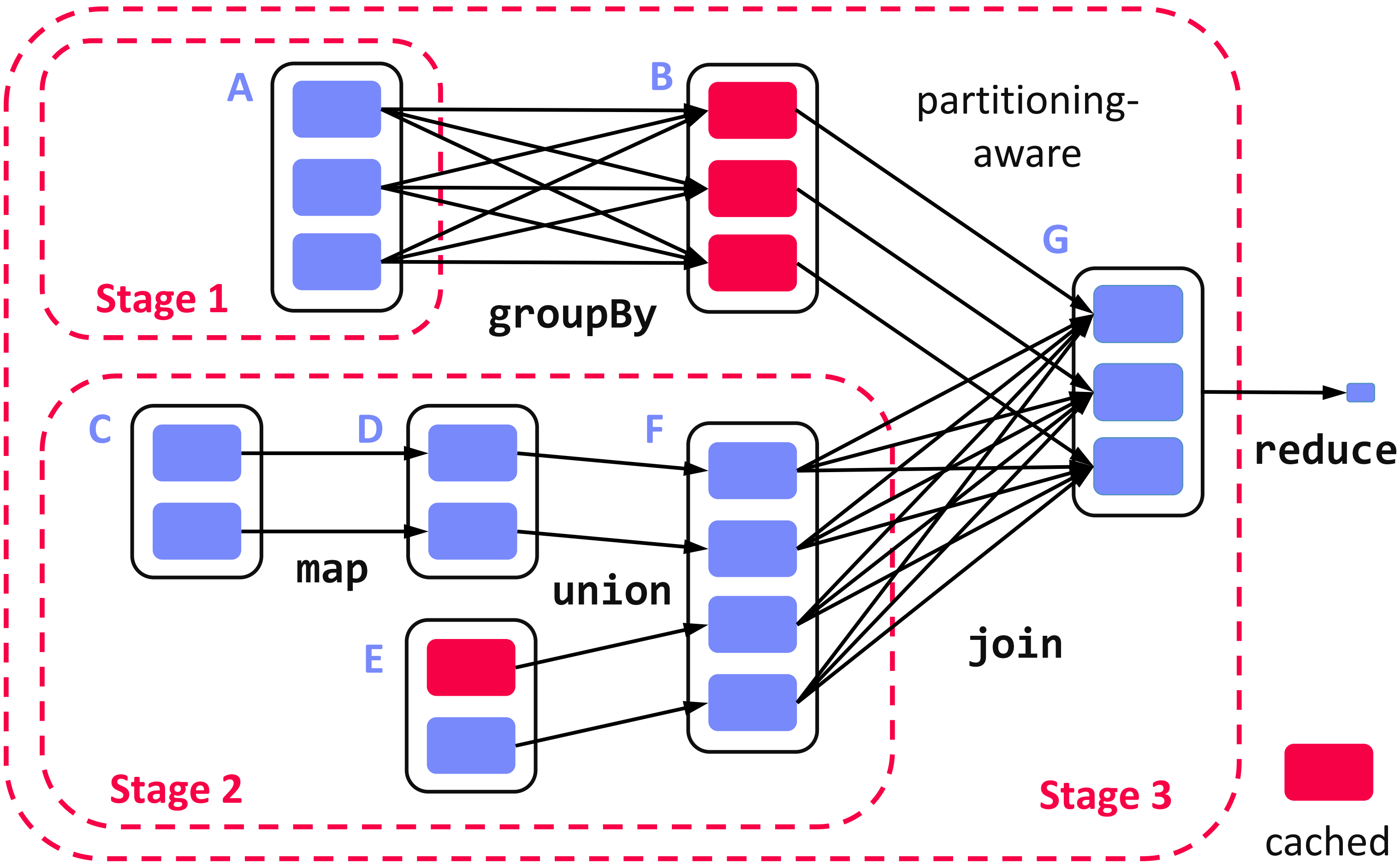
RDD Objects
- build operator DAG

DAGScheduler
- split graph into stages of tasks
- submit each stage as ready

TaskScheduler
- launch tasks at workers
- retry failed or straggling tasks

Workers
- execute tasks
- store and serve blocks

Spark Lazy Evaluation, Caching, and Lineage

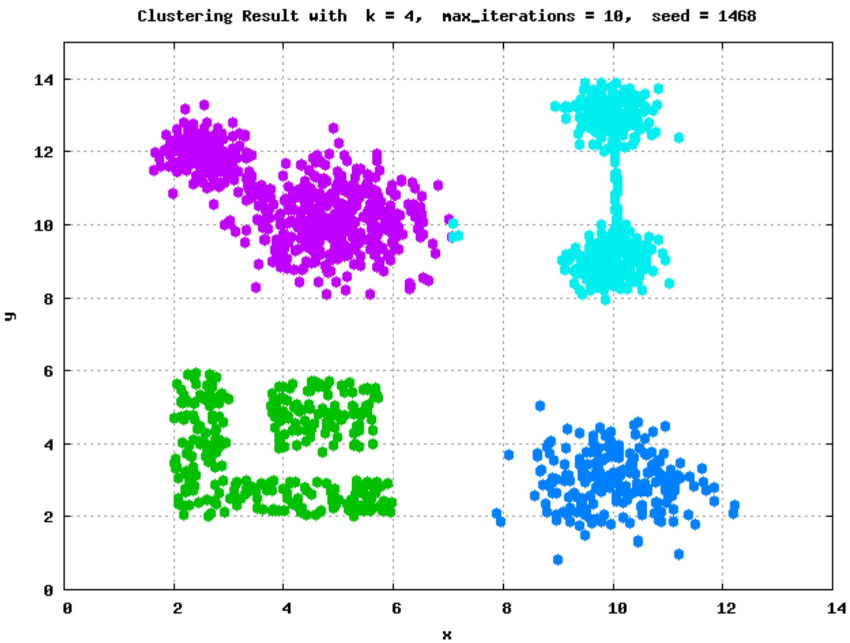


[Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, Ion Stoica: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. **NSDI 2012**]

Example: k-Means Clustering

- **k-Means Algorithm**
  - Given dataset D and number of clusters k, find cluster centroids ("mean" of assigned points) that minimize within-cluster variance
  - Euclidean distance: **sqrt**(**sum**((**a**-**b**)^2))
- **Pseudo Code**

```
function Kmeans(D, k, maxiter) {
  C‘ = randCentroids(D, k);
  C = {};
  i = 0; //until convergence
  while( C‘ != C & i<=maxiter ) {
    C = C‘;
    i = i + 1;
    A = getAssignments(D, C);
    C‘ = getCentroids(D, A, k);
  }
  return C‘
}
```



Example: K-Means Clustering in Spark

```
// create spark context (allocate configured executors)
JavaSparkContext sc = new JavaSparkContext();

// read and cache data, initialize centroids
JavaRDD<Row> D = sc.textFile("hdfs:/user/mboehm/data/D.csv")
  .map(new ParseRow()).cache(); // cache data in spark executors
Map<Integer,Mean> C = asCentroidMap(D.takeSample(false, k));

// until convergence
while( !equals(C, C2) & i<=maxiter ) {
  C2 = C; i++;
  // assign points to closest centroid, recompute centroid
  Broadcast<Map<Integer,Row>> bC = sc.broadcast(C)
  C = D.mapToPair(new NearestAssignment(bC))
       .foldByKey(new Mean(0), new IncComputeCentroids())
       .collectAsMap();
}

return C;
```

Note: Existing library algorithm
[https://github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spark/mllib/clustering/KMeans.scala]

Data-Parallel DataFrame Operations

Origins of DataFrames

- **Recap: Data Preparation Problem**
  - **80% Argument:** 80-90% time for finding, integrating, cleaning data
  - Data scientists prefer scripting languages and in-memory libraries

- **R and Python DataFrames**
  - R `data.frame/dplyr` and Python pandas DataFrame for seamless data manipulations (most popular packages/features)
  - DataFrame: **table with a schema**
  - Descriptive stats and basic math, reorganization, joins, grouping, windowing
  - **Limitation:** Only in-memory, single-node operations

- **Example Pandas**

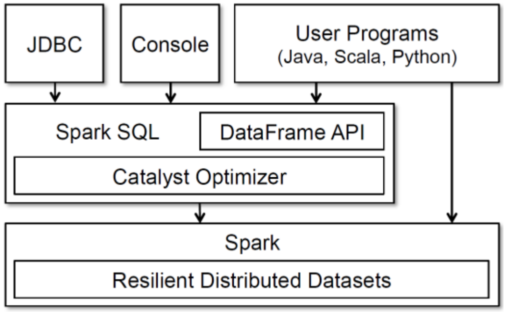
```
import pandas as pd

df = pd.read_csv('data/tmp1.csv', index_col=2)
df.head() # df w/ indexes A-Z

df = pd.concat(df, df[['A','C']], axis=0)
```

Spark DataFrames and DataSets

- **Overview Spark DataFrame**
  - DataFrame is **distributed collection of rows** with named/typed columns
  - Relational operations (e.g., projection, selection, joins, grouping, aggregation)
  - DataSources (e.g., json, jdbc, parquet, hdfs, s3, avro, hbase, csv, cassandra)



- **DataFrame and Dataset APIs** `DataFrame = Dataset[Row]`
  - DataFrame was introduced as basis for Spark SQL
  - DataSets allow **more customization** and compile-time analysis errors (Spark 2)

- **Example DataFrame**

```
logs = spark.read.format("json").open("s3://logs")
logs.groupBy(logs.user_id).agg(sum(logs.time))
  .write.format("jdbc").save("jdbc:mysql//...")
```

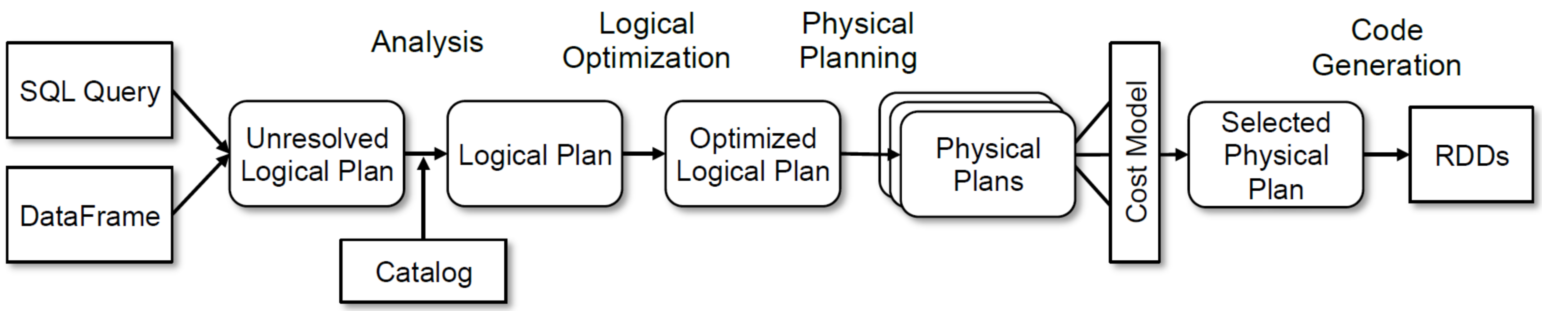
[Michael Armbrust: Structuring Apache Spark – SQL, DataFrames, Datasets, and Streaming, **Spark Summit 2016**]

→ **PySpark**

SparkSQL and DataFrame/Dataset

[Michael Armbrust et al.: Spark SQL: Relational Data Processing in Spark. **SIGMOD 2015**]

- **Overview SparkSQL**
  - Shark (~2013): academic prototype for SQL on Spark
  - **SparkSQL** (~2015): reimplementation from scratch
  - Common IR and compilation of SQL and DataFrame operations
- **Catalyst: Query Planning**

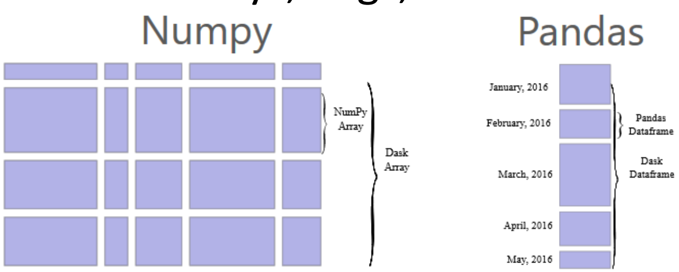


- **Performance features**
  - **#1 Whole-stage code generation** via Janino
  - **#2 Off-heap memory** (`sun.misc.Unsafe`) for caching and certain operations
  - **#3 Pushdown** of selection, projection, joins into data sources (+ join ordering)

Dask

[Matthew Rocklin: Dask: Parallel Computation with Blocked algorithms and Task Scheduling, **Python in Science 2015**]
[Dask Development Team: Dask: Library for dynamic task scheduling, 2016, https://dask.org]

- **Overview Dask**
  - Multi-threaded and distributed operations for arrays, bags, and dataframes
  - `dask.array:` list of numpy n-dim arrays
  - `dask.dataframe:` list of pandas data frames
  - `dask.bag:` unordered list of tuples (second order functions)
  - Local and distributed schedulers: threads, processes, YARN, Kubernetes, containers, HPC, and cloud, GPUs



- **Execution**
  - Lazy evaluation
  - Limitation: requires **static size inference**
  - Triggered via `compute()`

```
import dask.array as da

x = da.random.random(
  (10000,10000), chunks=(1000,1000))
y = x + x.T
y.persist() # cache in memory
z = y[::2, 5000:].mean(axis=1) # colMeans
ret = z.compute() # returns NumPy array
```

Ray

RAY

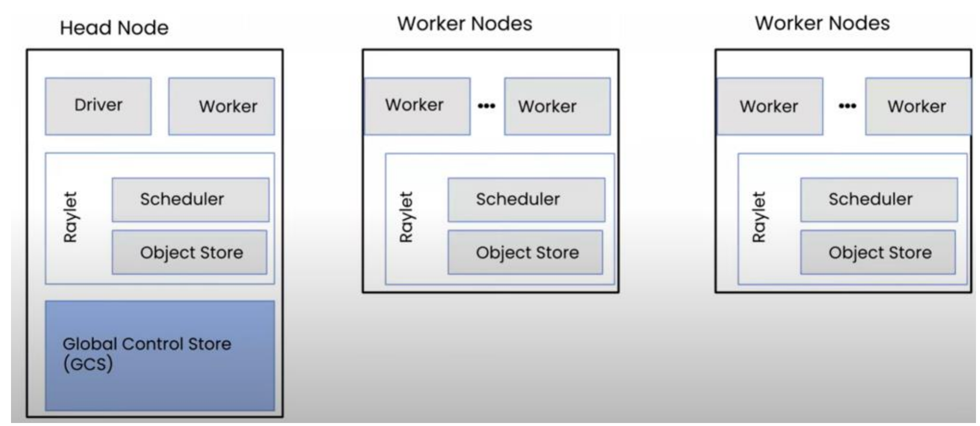
[Philipp Moritz et al.: Ray: A Distributed Framework for Emerging AI Applications, **OSDI 2018**]

- **Overview Ray**
  - Universal framework for distributed computing
    - Scale AI and Python workloads
  - **AIR-**AI runtime, set of ML libraries
    - Ray Data, RLib, Ray Train,
    Ray Batch Predictor, Ray Tune, Ray Serve
  - Architecture
    - GCS, Raylet, Object Store (Redis)
  - Parallel Task
    - Stateless units
  - Objects or Futures
    - Distributed objects
  - Actors
    - Stateful services
    - message passing

```python
@ray.remote
def read_array(file):
    # read ndarray "a"
    # from "file"
    return a

@ray.remote
def add(a, b):
    return np.add(a, b)

id1 = read_array.remote(file1)
id2 = read_array.remote(file2)
id = add.remote(id1, id2)
sum = ray.get(id)
```



Data-Parallel Operations in SystemDS / DAPHNE

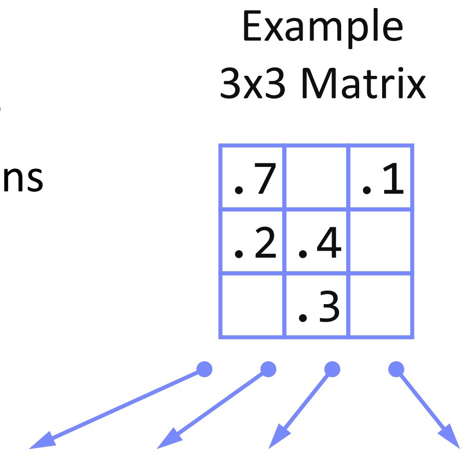
[Matthias Boehm et al.: **SystemDS**: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. **CIDR 2020**]

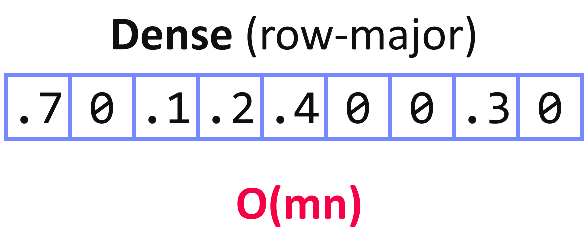
[Matthias Boehm et al.: **SystemML**: Declarative Machine Learning on Spark. **PVLDB 9(13) 2016**]

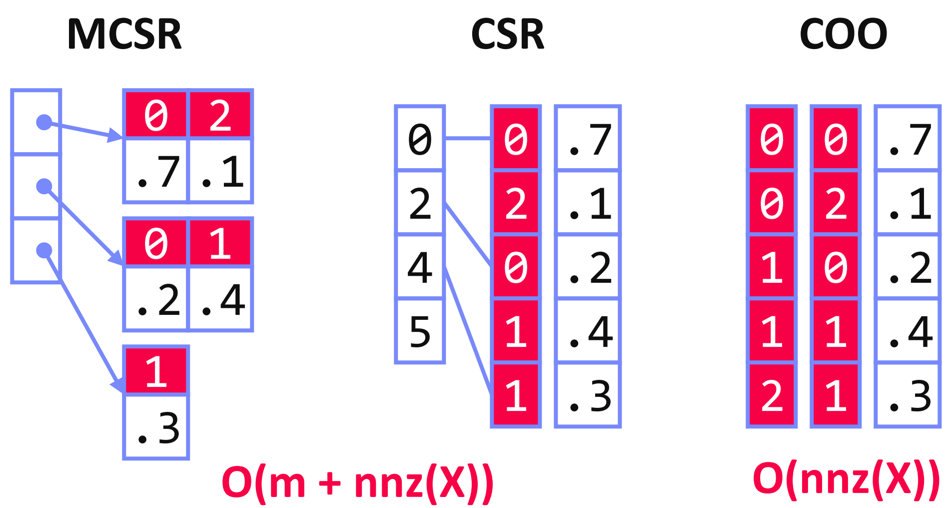
[Amol Ghoting et al.: **SystemML**: Declarative Machine Learning on MapReduce. **ICDE 2011**]

Background: Matrix Formats

- **Matrix Block** (m x n)
  - A.k.a. tiles/chunks, most operations defined here
  - Local matrix: single block, different representations
- **Common Block Representations**
  - Dense (linearized arrays)
  - MCSR (modified CSR)
  - CSR (compressed sparse rows), CSC
  - COO (Coordinate matrix)



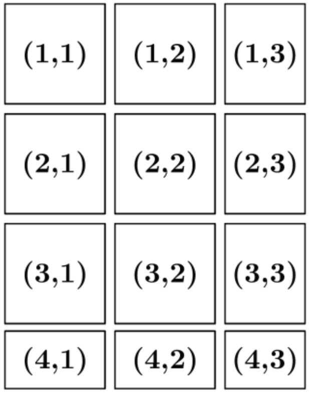




Distributed Matrix Representations

- **Collection of "Matrix Blocks" (and keys)**
  - **Bag semantics** (duplicates, unordered)
  - Logical (Fixed-Size) Blocking
    + join processing / independence
    - (sparsity skew)
  - E.g., SystemML/SystemDS on Spark:
    `JavaPairRDD<MatrixIndexes,MatrixBlock>`
  - Blocks encoded independently (dense/sparse)

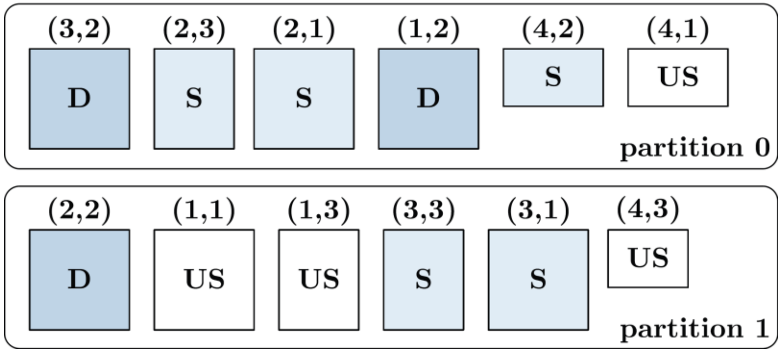
Logical Blocking
3,400x2,700 Matrix
(w/ B_c=1,000)



- **Partitioning**
  - Logical Partitioning
    (e.g., row-/column-wise)
  - Physical Partitioning
    (e.g., hash / grid)
  - PartitionPruning for Indexing

Physical Blocking and Partitioning

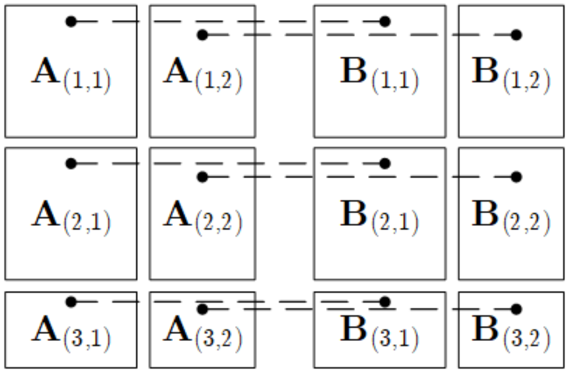
hash partitioned: e.g., hash(3,2) → 99,994 % 2 = 0



Distributed Matrix Operations

Elementwise Multiplication
(Hadamard Product)

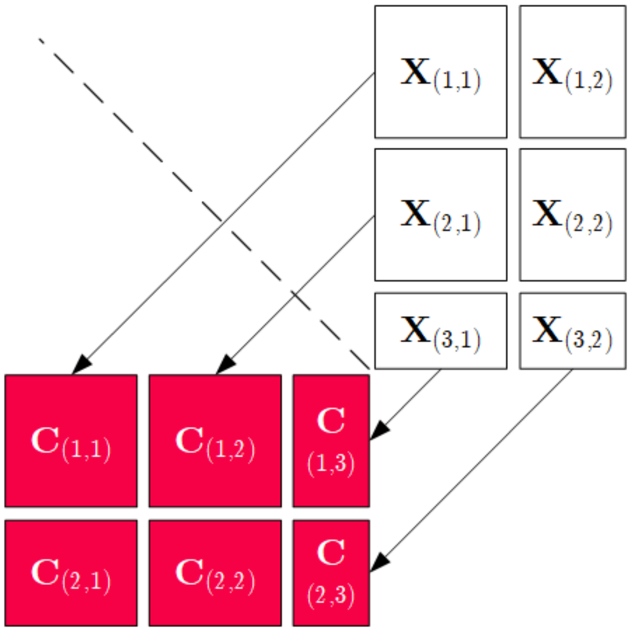
`C = A * B`



Note: also with row/column vector rhs

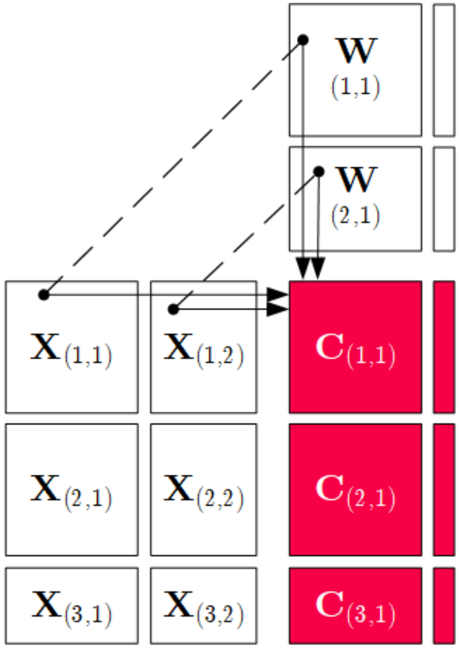
Transposition

`C = t(X)`



Matrix Multiplication

`C = X %*% W`



Note: 1:N join

Summary and Q&A

- **Motivation and Terminology**
- **Data-Parallel Collection Processing**
- **Data-Parallel DataFrame Operations**
- **Data-Parallel Computation in SystemDS / DAPHNE**

- **Next Lectures**
  - **12 Distributed Stream Processing** [Jan 19]
  - **13 Distributed Machine Learning Systems** [Jan 19]
  - Q&A Session including sample exam questions