TU Graz

SCIENCE
PASSION
TECHNOLOGY

Architecture of ML Systems
05 Execution Strategies

Matthias Boehm

Graz University of Technology, Austria
Computer Science and Biomedical Engineering
Institute of Interactive Systems and Data Science
BMVIT endowed chair for Data Management

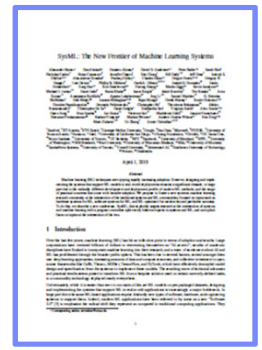
Last update: Apr 12, 2019

ISDS

Announcements/Org

- **#1 Programming/Analysis Projects**
  - **#1 Auto Differentiation**
  - **#5 LLVM Code Generator**
  - **#12 Information Extraction from Unstructured PDF/HTML**
  - ➔ Individual meetings in next two weeks (if needed)

- **#2 Recommended Reading**
  - SysML whitepaper (building the ML systems community)
  - **Alexander Ratner et al: SysML: The New Frontier of Machine Learning Systems, SysML 2019**



Agenda

- **Overview Execution Strategies**
- **Background MapReduce and Spark**
- **Data-Parallel Execution**
- **Task-Parallel Execution**

Overview Execution Strategies

Categories of Execution Strategies

- **#1 Data-parallel Execution**
  - Run the same operations over data partitions in parallel
  - ML focus: batch algorithms, hybrid batch/mini-batch algorithms
- **#2 Task-parallel Execution**
  - Run different tasks (e.g., iterations of parfor) in parallel
  - Custom parallelization of independent subtasks
  - ML focus: meta learning, batch and mini-batch algorithms

This lecture

- **#3 Parameter Servers**
  - Compute partial or full model updates over data partitions, with periodic model synchronization
  - Compute parts of neural networks on different nodes w/ pipelining
  - Also know as **data-parallel learning vs model-parallel learning**
  - ML focus: mini-batch algorithms

Next lecture

Categories of Execution Strategies, cont.

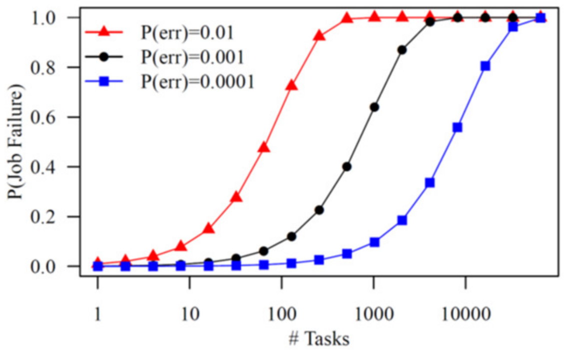
- **Example Systems**
  - Local computation
  - Distributed computation

Category	System	Data Par	Task Par	Param Serv	Accelerators
Numerical Computing	R		X / X		(GPU)*
	Julia		X / X		(GPU)*
Batch ML	SystemML	X / X	X / X	(X) / (X)	(GPU)
	Mahout S	X / X	(- / X)		
Mini-batch ML	TensorFlow		X / -	X / X	GPU / TPU
	PyTorch			X / X	GPU

Recap: Fault Tolerance & Resilience

[Google Data Center: https://www.youtube.com/watch?v=XZmGGAbHqa0]

- **Resilience Problem**
  - Increasing error rates at scale (soft/hard mem/disk/net errors)
  - Robustness for preemption
  - **Need for cost-effective resilience**
- **Fault Tolerance in Large-Scale Computation**
  - Block replication in distributed file systems
  - ECC; checksums for blocks, broadcast, shuffle
  - Checkpointing (all task outputs / on request)
  - Lineage-based recomputation for recovery in Spark



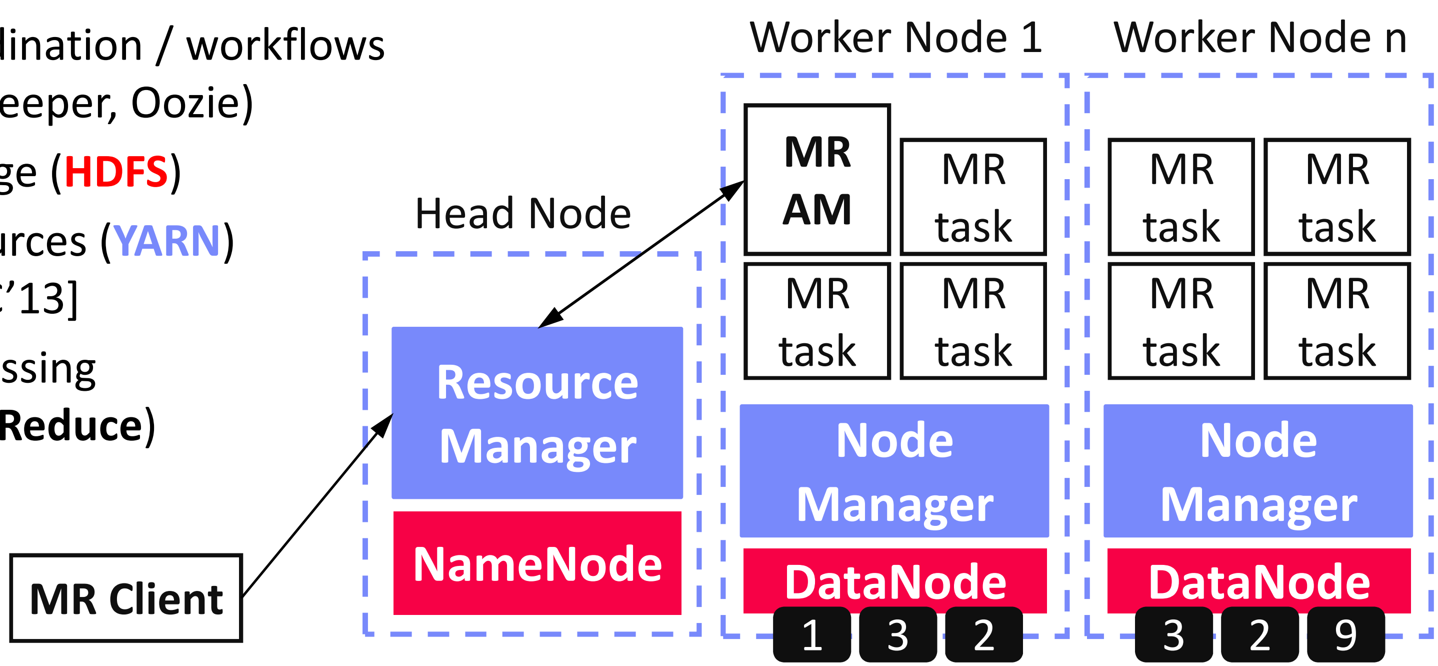
- **ML-specific Approaches** (exploit app characteristics)
  - Estimate contribution from lost partition to avoid strugglers
  - Example: user-defined "compensation" functions

Background MapReduce and Spark

Abstractions for Fault-tolerant, Distributed Storage and Computation

Hadoop History and Architecture

- **Brief History**
  - Google's GFS [SOSP'03] + MapReduce [ODSI'04] → 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**)



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

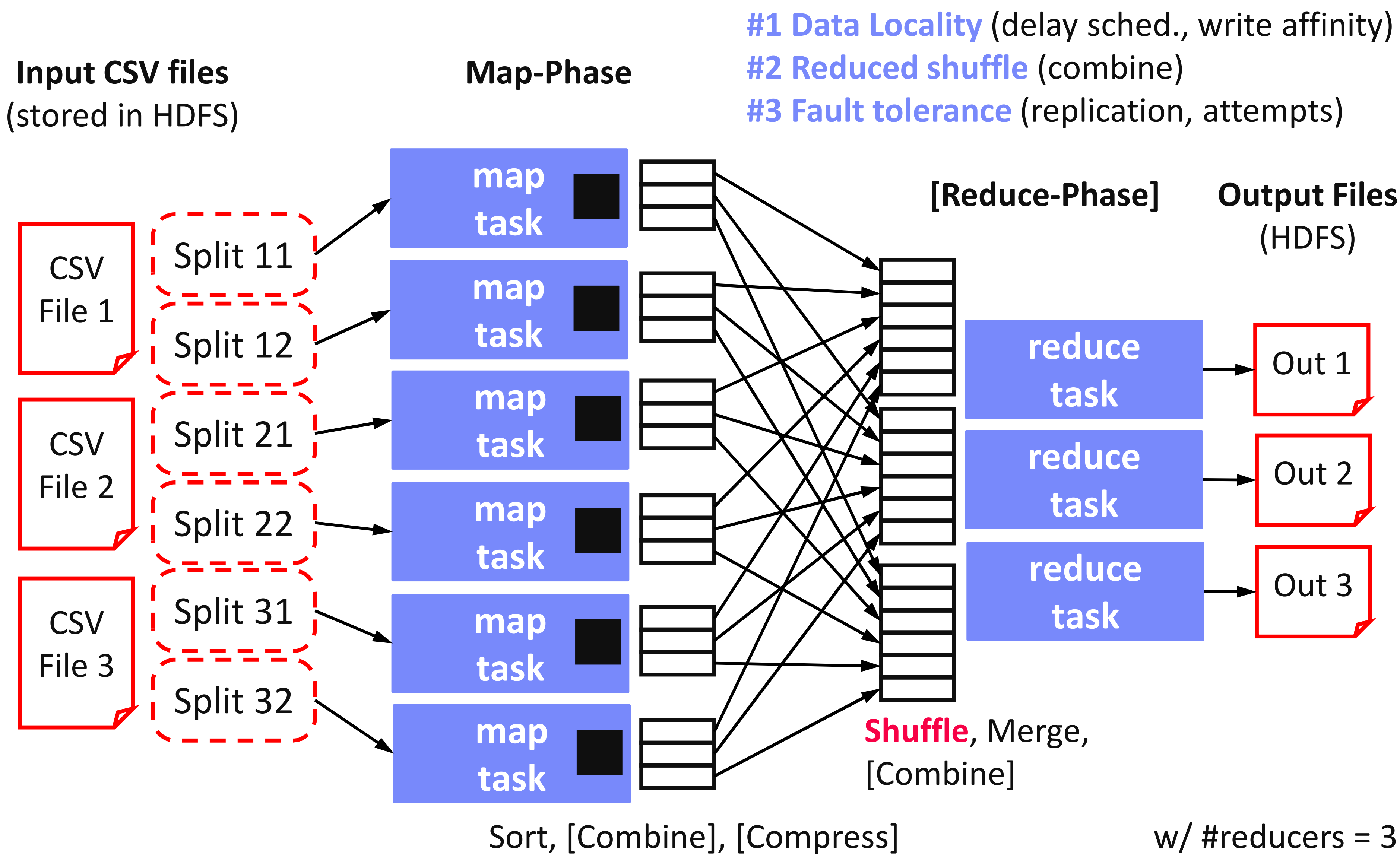
```
map(Long pos, String line) {
  parts ← line.split(“,”)
  emit(parts[1], 1)
}
```

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



MapReduce – Query Processing

- **Basic Unary Operations**
  - Selections (brute-force), projections, ordering
  - Additive and semi-additive aggregation with grouping
- **Binary Operations**
  - Set operations (union, intersect, difference) and joins
  - Different physical operators for R $\bowtie$ S (comparison [SIGMOD'10], [TODS'16])
    - **Broadcast join**: broadcast S, build HT S, map-side HJOIN
    - **Repartition join**: shuffle (repartition) R and S, reduce-side MJOIN
    - Improved repartition join, map-side/directed join (co-partitioned)
- **Criticism on MR for Query Processing** [SIGMOD'09] and ML
  - Lacks **high-level language/APIs**, **performance** (caching, indexing, compression)
- **Hybrid SQL-on-Hadoop Systems** [VLDB'15]
  - Examples: 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

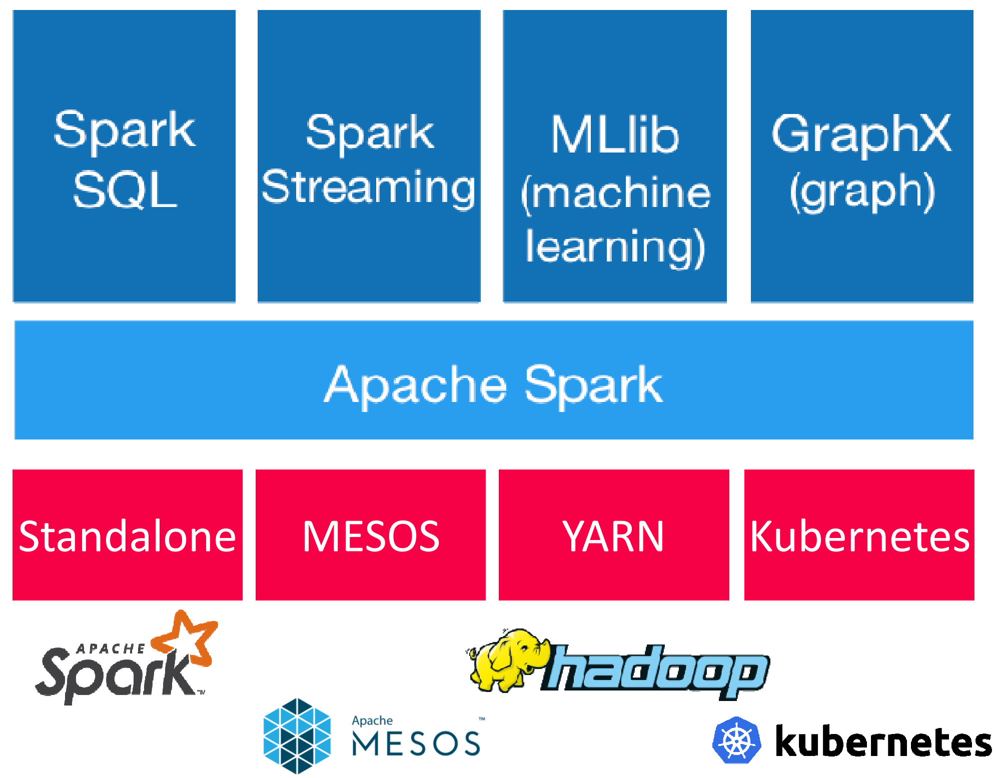
➔ **But many shared concepts/infrastructure**

- **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.

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

- **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



- **Focus on a unified platform for data-parallel computation**

How about the integration of **specialized parameter severs**?
➔ [SPARK-24375]

Resilient Distributed Datasets (RDDs)

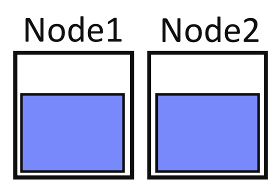
- **RDD Abstraction**
  - Immutable, partitioned **collections of key-value pairs**
  - Coarse-grained deterministic operations (transformations/actions)
  - Fault tolerance via lineage-based recomputation

```
JavaPairRDD
  <MatrixIndexes,MatrixBlock>
```

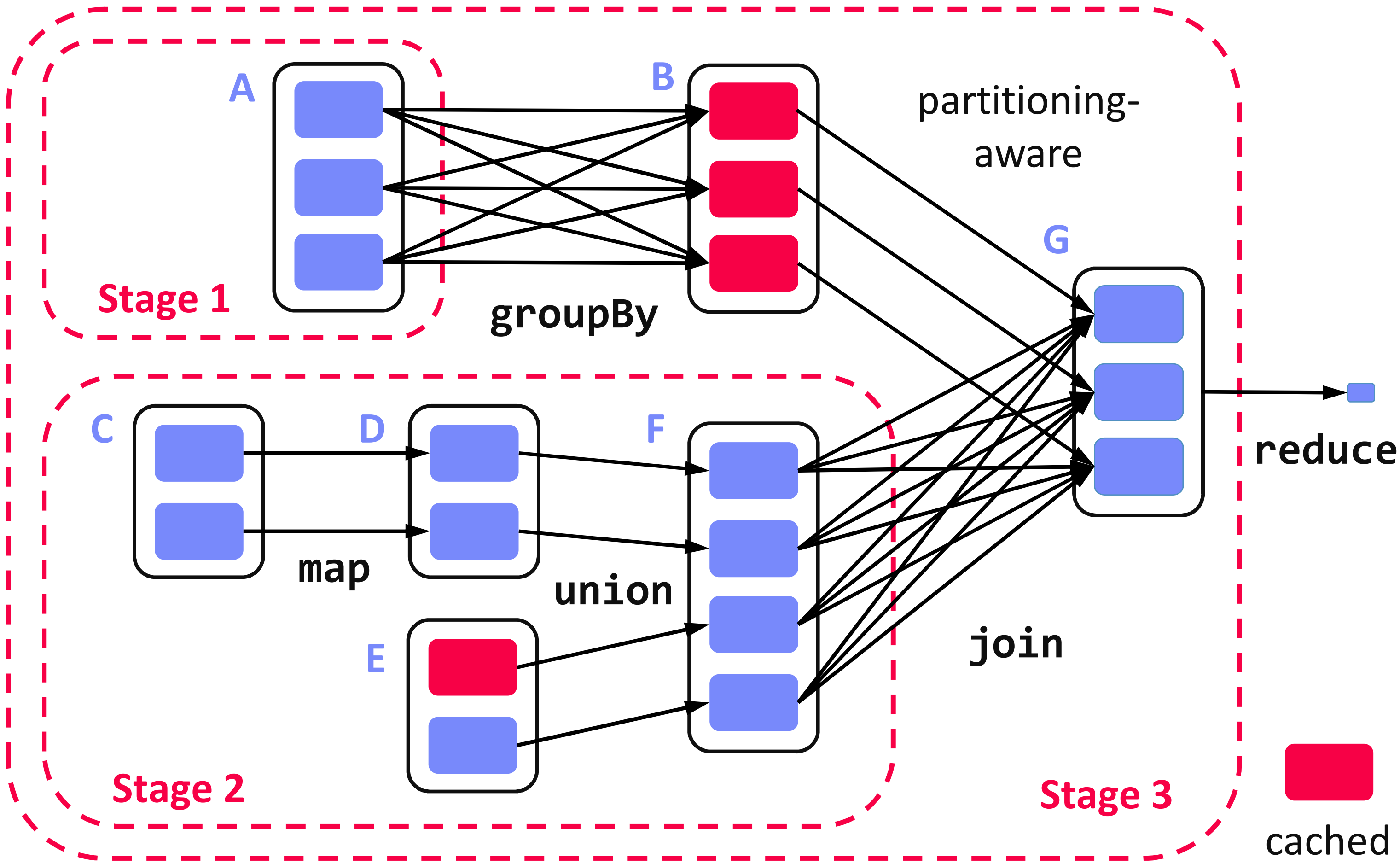
- **Operations**
  - Transformations: define new RDDs
  - Actions: return result to driver

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)



Lazy Evaluation, Caching, and Lineage

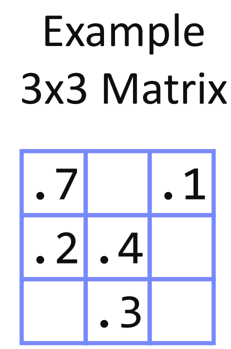


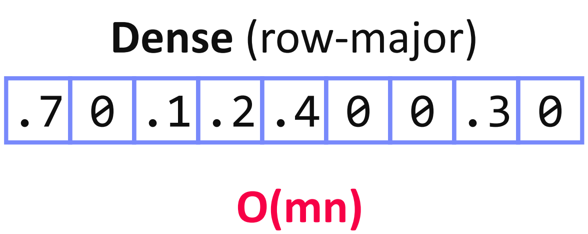
[Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, Ion Stoica: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. **NSDI 2012**]

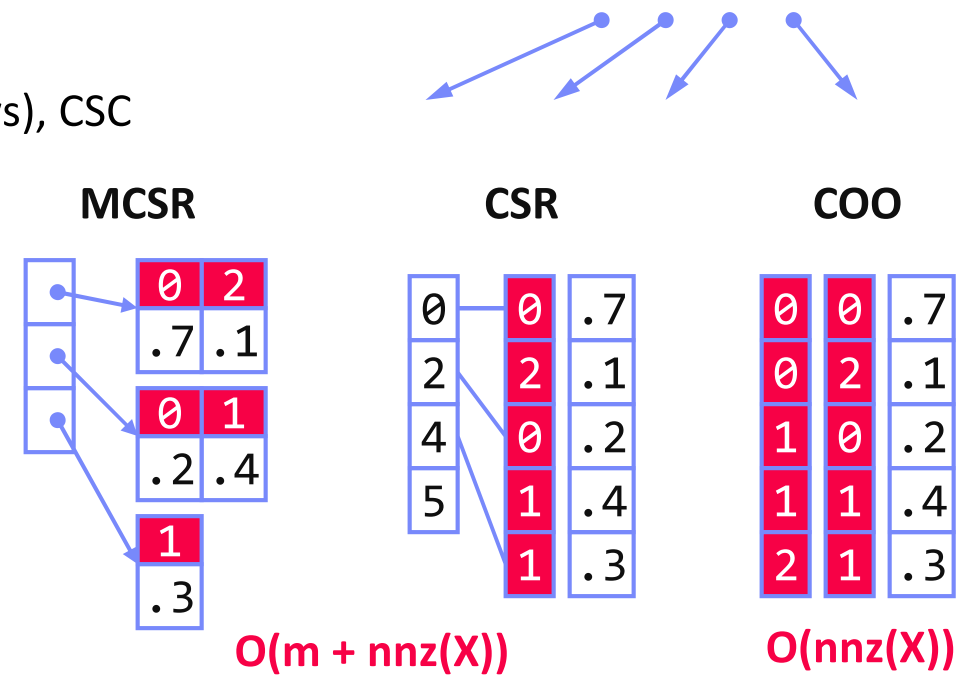
Data-Parallel Execution

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)







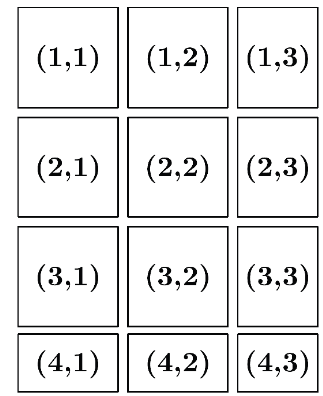
Distributed Matrix Representations

- **Collection of "Matrix Blocks" (and keys)**
  - Bag semantics (duplicates, unordered)
  - Logical (Fixed-Size) Blocking

    + join processing / independence

    - (sparsity skew)
  - E.g., SystemML on Spark:
    JavaPairRDD<MatrixIndexes,MatrixBlock>
  - Blocks encoded independently (dense/sparse)

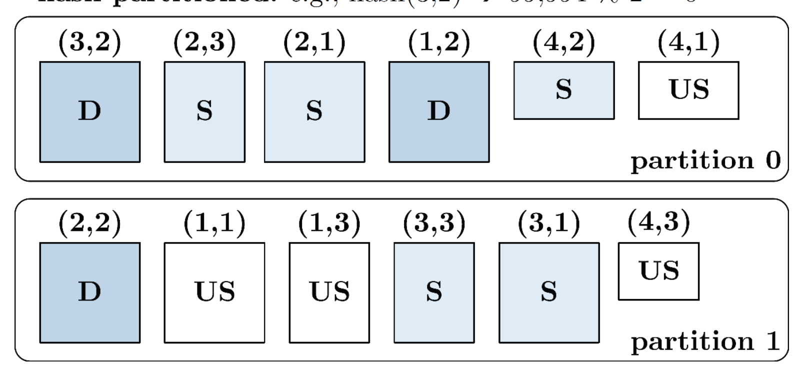
Logical Blocking
3,400x2,700 Matrix
(w/ B_c=1,000)



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

Physical Blocking and Partitioning

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



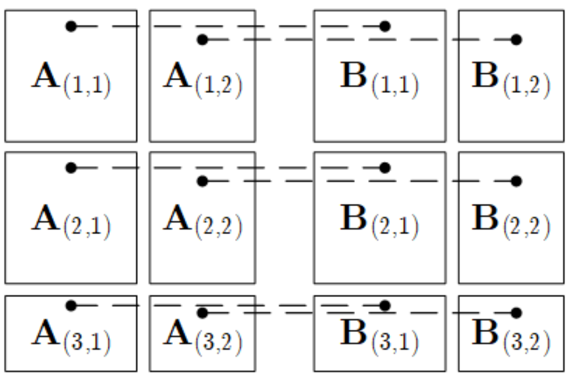
Distributed Matrix Representations, cont.

- **#1 Block-partitioned Matrices**
  - Fixed-size, square or rectangular blocks
  - **Pros:** Input/output alignment, block-local transpose, amortize block overheads, bounded memory requirements, cache-conscious block ops
  - **Cons:** Converting row-wise inputs (e.g., text) into blocks requires shuffle
  - Examples: **RIOT**, **PEGASUS**, **SystemML**, **SciDB**, **Cumulon**, **Distributed R**, **DMac**, **Spark Mllib**, **Gilbert**, **MatFast**, and **SimSQL**
- **#2 Row/Column-partitioned Matrices**
  - Collection of row indexes and rows (or columns respectively)
  - **Pros:** Seamless data conversion and access to entire rows
  - **Cons:** Storage overhead in Java, and cache unfriendly operations
  - Examples: **Spark MLlib**, **Mahout Samsara**, **Emma**, **SimSQL**
- **#3 Algorithm-specific Partitioning**
  - Operation and algorithm-centric data representations
  - Examples: matrix **inverse**, matrix **factorization**

Distributed Matrix Operations

Elementwise Multiplication
(Hadamard Product)

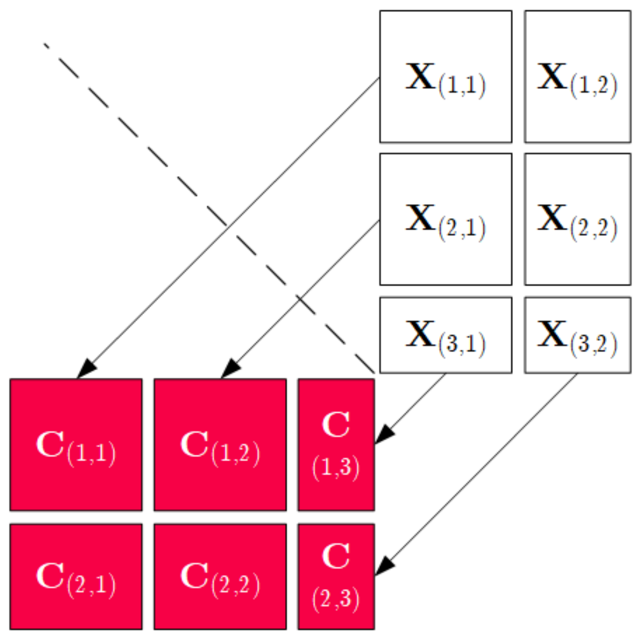
`C = A * B`



Note: also with row/column vector rhs

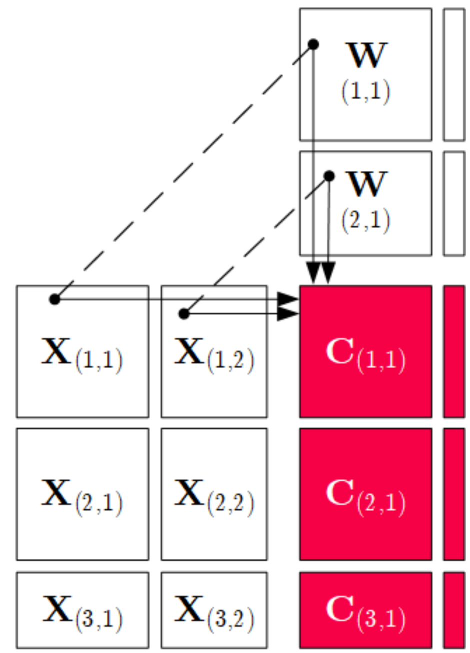
Transposition

`C = t(X)`



Matrix Multiplication

`C = X %*% W`



Note: 1:N join

Partitioning-Preserving Operations

- **Shuffle is major bottleneck for ML on Spark**
- **Preserve Partitioning**
  - Op is partitioning-preserving if keys unchanged (guaranteed)
  - Implicit: Use restrictive APIs (`mapValues()` vs `mapToPair()`)
  - Explicit: Partition computation w/ declaration of partitioning-preserving
- **Exploit Partitioning**
  - Implicit: Operations based on `join`, `cogroup`, etc
  - Explicit: Custom operators (e.g., zipmm)
- **Example: Multiclass SVM**
  - Vectors fit neither into driver nor broadcast
  - $ncol(X) \leq B_c$

```
parfor(iter_class in 1:num_classes) {        <- repart, chkpt X MEM_DISK
   Y_local = 2 * (Y == iter_class) - 1
   g_old = t(X) %*% Y_local
   ...                                        <- chkpt y_local MEM_DISK
   while( continue ) {
      Xd = X %*% s                            <- chkpt Xd, Xw MEM_DISK
      ... inner while loop (compute step_sz)
      Xw = Xw + step_sz * Xd;
      out = 1 - Y_local * Xw;
      out = (out > 0) * out;
      g_new = t(X) %*% (out * Y_local) ...       zipmm
```

Single Instruction Multiple Data (SIMD)

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

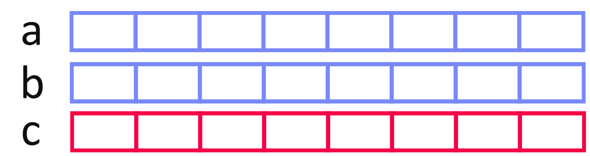
Increasing Vector Lengths

2009 Nehalem: **128b** (2xFP64)

2012 Sandy Bridge: **256b** (4xFP64)

2017 Skylake: **512b** (8xFP64)

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



- **SIMD vs Multi-threading in ML Systems**
  - ML systems in native programming languages focus primarily on **SIMD**
    ➔ Essential for **mini-batch algorithms and compute-intensive kernels**
  - SIMD very good for **dense operations**, gather/scatter required for sparse
  - **Multi-threading** additionally applied via reused thread pools
    ➔ Even without SIMD: quickly **saturate peak memory bandwidth**
  - ML systems in Java use JNI to call native **BLAS to exploit SIMD**

Task-Parallel Execution

Parallel For Loops (parfor)

[M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. Burdick, S. Vaithyanathan: Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. **PVLDB 2014**]

- **Motivation**
  - **Use cases:** ensemble learning, cross validation, hyper-parameter tuning, complex models with disjoint/overlapping/all data per task
  - **Hybrid parallelization strategies** (combined data- and task-parallel)

- **Key Ideas:**
  - Dependency Analysis
  - Task partitioning
  - Data partitioning, scan sharing, various rewrites
  - Execution strategies
  - Result agg strategies
  - **ParFor optimizer**

```
reg = 10^(seq(-1,-10))
B_all = matrix(0, nrow(reg), n)

parfor( i in 1:nrow(reg) ) {
   B = linregCG(X, y, reg[i,1]);
   B_all[i,] = t(B);
}
```

- **Local ParFor** (multi-threaded), w/ local ops
- **Remote ParFor** (distributed Spark job)
- **Local ParFor**, w/ concurrent distributed ops

- **Example Systems**
  - **SystemML**, **R**, **Matlab**

Additional Examples

#1 Pairwise Pearson Correlation

(in practice, bivariate statistics: Pearson's R, Anova F, Chi-squared, Degree of freedom, P-value, Cramers V, Spearman, etc)

```
D = read("./input/D");
m = nrow(D);
n = ncol(D);
R = matrix(0, rows=n, cols=n);
parfor( i in 1:(n-1) ) {
   X = D[ ,i];
   m2X = centralMoment(X,2);
   sigmaX = sqrt( m2X*(m/(m-1.0)) );
   parfor( j in (i+1):n ) {
      Y = D[ ,j];
      m2Y = centralMoment(Y,2);
      sigmaY = sqrt( m2Y*(m/(m-1.0)) );
      R[i,j] = cov(X,Y) / (sigmaX*sigmaY);
   }
}
write(R, "./output/R");
```

#2 Batch-wise CNN Scoring

```
prob = matrix(0, Ni, Nc)
parfor( i in 1:ceil(Ni/B) ) {
   Xb = X[((i-1)*B+1):min(i*B,Ni),];
   prob[((i-1)*B+1):min(i*B,Ni),] =
      ... # CNN scoring
}
```

➔ **Conceptual Design: Master/worker**
(task: group of parfor iterations)

ParFor Execution Strategies

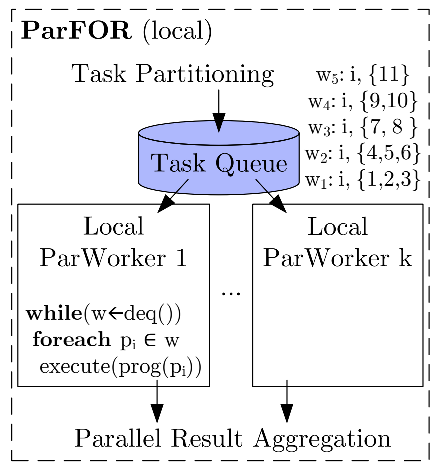
- **#1 Task Partitioning**
  - Fixed-size schemes:
    naive (1) , static (n/k), fixed (m)
  - Self-scheduling: e.g.,
    guided self scheduling, factoring

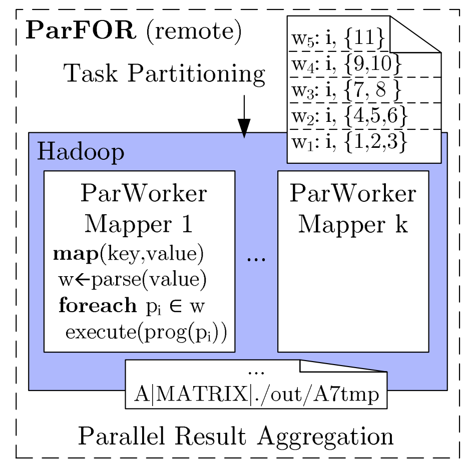
Factoring (n=101, k=4)

$$R_0 = N, \quad R_{i+1} = R_i - k \cdot l_i, \qquad l_i = \left\lceil \frac{R_i}{x_i \cdot k} \right\rceil = \left\lceil \left(\frac{1}{x_i}\right)^{i+1} \frac{N}{k} \right\rceil$$

(13,13,13,13, 7,7,7,7, 3,3,3,3, 2,2,2,2, 1)

- **#2 Data Partitioning**
  - Local or remote row/column partitioning (incl locality)
- **#3 Task Execution**
  - Local (multi-core) execution
  - Remote (MR/Spark) execution

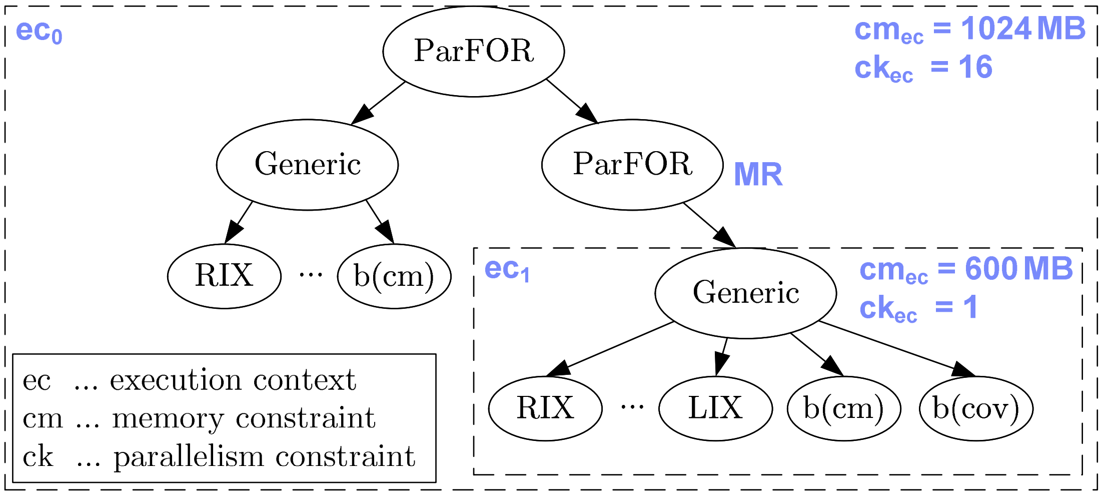




- **#4 Result Aggregation**
  - With and without compare (non-empty output variable)
  - Local in-memory / remote MR/Spark result aggregation

ParFor Optimizer Framework

- **Design: Runtime optimization for each top-level parfor**
- **Plan Tree P**
  - Nodes N_P
    - Exec type et
    - Parallelism k
    - Attributes A
  - Height h
  - Exec contexts EC_P



- **Plan Tree Optimization Objective**

$$\phi_2: \min \quad \hat{T}(r(P))$$
$$s.t. \quad \forall ec \in \mathcal{EC}_P : \hat{M}(r(ec)) \leq cm_{ec} \;\wedge\; K(r(ec)) \leq ck_{ec}.$$

- **Heuristic optimizer w/ transformation-based search strategy**
  - Cost and memory estimates w/ plan tree aggregate statistics

Summary and Conclusions

- **Categories of Execution Strategies**
  - Data-parallel execution for batch ML algorithms
  - Task-parallel execution for custom parallelization of independent tasks
  - Parameter servers (data-parallel vs model-parallel) for mini-batch ML algorithms

- **#1 Different strategies (and systems) for different ML workloads**
  ➔ Specialization and abstraction

- **#2 Awareness of underlying execution frameworks**

- **#3 Awareness of effective compilation and runtime techniques**

- **Next Lecture**
  - 06 Parameter Servers [May 03]