SystemML: Declarative Machine Learning on Spark

Nathan Smith, Razieh Nabi, Alex Gain


Spark

Data comes in many forms

- Volume: terabytes to exabytes
- Velocity: speed of incoming data
- Variety: structured,unstructured data, text, image, video

Data Science Challenges

- Need for sophisticated, custom machine learning algorithms
- Growing amount of data through the Internet of Things expansion in manufacturing, healthcare, and more
- Need to run custom ML algorithms on distributed, data-parallel systems such as MapReduce and Spark for scalability and performance on low-cost commodity clusters

Evolution of Distributed Systems

- First generation:Proprietary, custom hardware and software,centralized data, hardware based fault recovery (Teradata)
- Second generation:Open source, commodity hardware, distributed data, software based fault recovery (Hadoop)
- Then: disks were cheap, primary source of data, networks costly, data locality, RAM costly, single core machines dominant
- Now: RAM is primary source of data, disk used as fallback, networks are faster and multi-core machines are common

History Lesson

- 2002: Google creates MapReduce out of the Google File System
- 2004: Google releases the MapReduce paper
- 2006: Hadoop 0.1.0 released, first used at Yahoo
- 2009: Spark created at UC Berkeley AMPLab
- 2012: Apache Hadoop 1.0 released
- 2013: Spark donated to the Apache Software Foundation

The Apache Software Foundation

- Decentralized open source community of developers, officially incorporated in 1999
- All software produced is free and open source
- Apache projects include: Hadoop, HTTP server, Maven, Spark, OpenNLP, Groovy, and many more!

MapReduce

- Processing and generating large data sets with a parallel, distributed algorithm on a cluster
- Map procedure:performs filtering and sorting(sorting students by first name into queues)
- Reduce method: summary operation (counting the number of students in each queue)
- Marshals distributed servers, runs tasks in parallel, and manages communications and data transfers between various parts of the system

Apache Hadoop

- Started from the Google File System and MapReduce papers in 2003
- Doug Cutting, an Apache developer and Yahoo employee, named it after his son's toy elephant
- Core consists of storage through the Hadoop Distributed File System (HDFS), and processing through MapReduce
- Splits files into large blocks and distributes them across nodes in a cluster
- Transfers packages code for nodes to process in parallel based on the data that needs to be processed

MapReduce and Hadoop fall short

- Lacks one thing to succeed for iterative queries and interactive queries: fast data sharing
- Hadoop needed to give fault tolerance, and to achieve this it needed to go to disk
- MapReduce was not a general compute engine, so many specialized systems emerged: Hive, HBase, Pig, Zookeeper, Mahout, and more
- We needed fault tolerance with speed

What is Spark?

- Open Source
- Alternative to MapReduce for certain applications
- A low latency cluster computing system
- For very large data sets, focused on iterative programs and interactive querying
- Used with Hadoop/HDFS
- May be 100 times faster than MapReduce for
  - Iterative algorithms
  - Interactive data mining

How does it work?

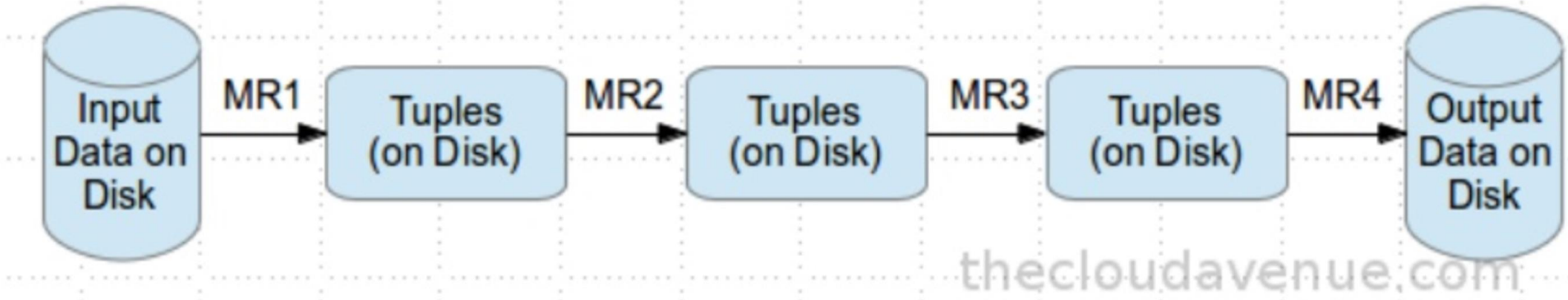
- Uses in-memory cluster computing: data analysis performed on cached, live data, rather than stored, historical data
- Memory access faster than disk access
- Can be accessed from Scala and Python shells
- Has APIs written in
  - Scala
  - Python
  - Java

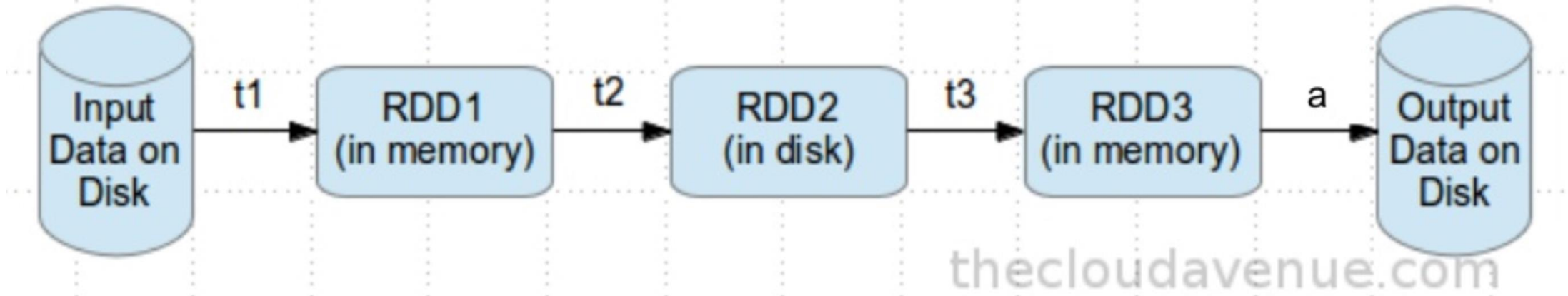
But how?

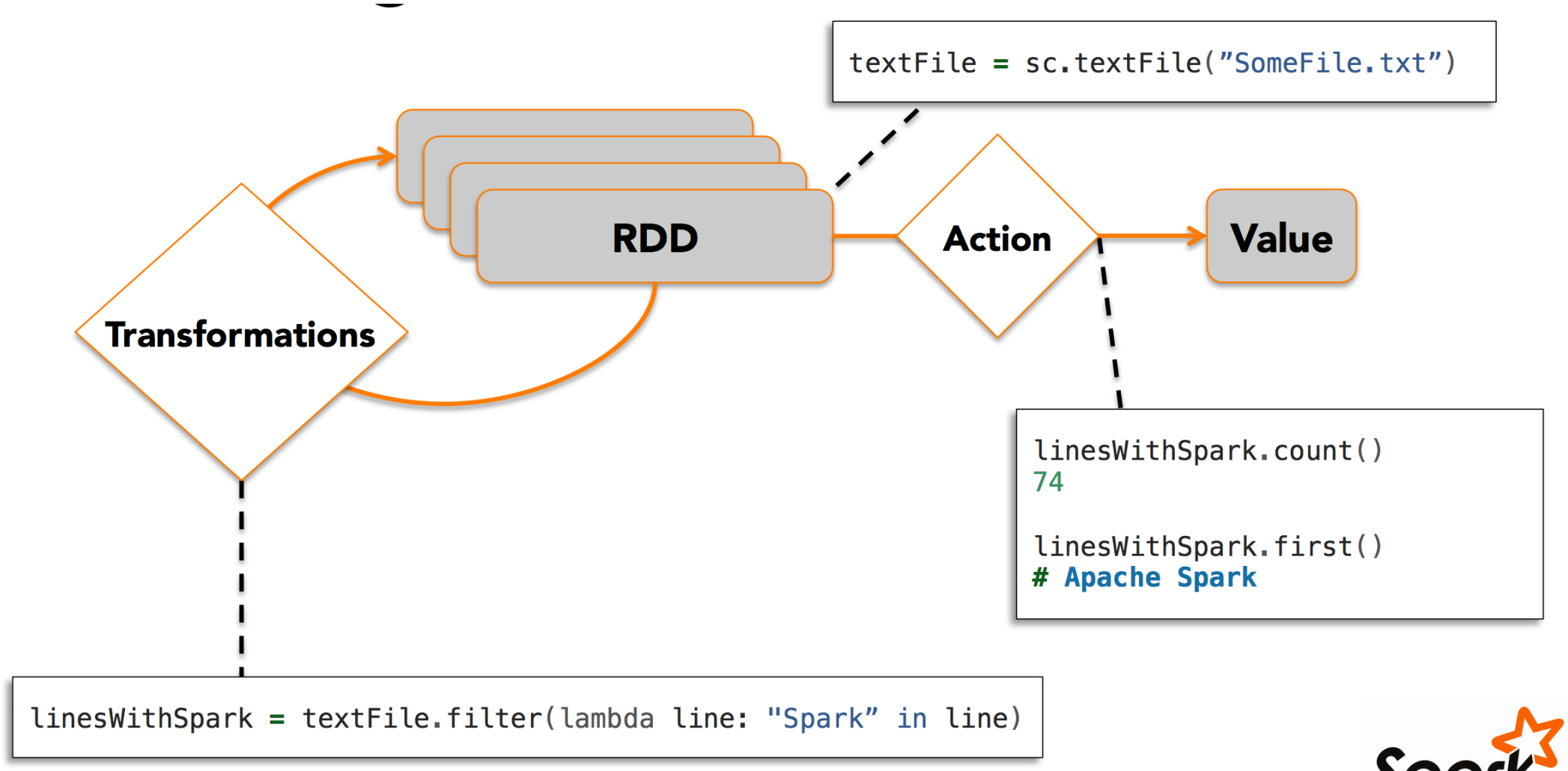
- To understand Spark, we need to understand 3 big concepts:
- RDDs:resilient distributed datasets, read-only multiset of data items distributed over a cluster of machines
- Transformations: what you do to RDDs to get other resultant RDDs, such as filtering
- Actions: asking for an answer,such as count or first line
- Lazy evaluation: RDDs are not loaded into the system every time the system encounters an RDD, but only when actions are performed
- Example: we can realize that a dataset created through map will be used in a reduce, and return only the result of the reduce

Motivation

- Complex apps and interactive queries both need one thing MapReduce lacks: efficient primitives for data sharing
- In MapReduce, the only way to share data across jobs is stable storage, slow due to replication and disk I/O!
- RDDs track the graph of transformations that built them (their lineage)to rebuild lost data


Input Data on Disk
MR1
Tuples (on Disk)
MR2
Tuples (on Disk)
MR3
Tuples (on Disk)
MR4
Output Data on Disk
thecloudavenue.com


Input Data on Disk
t1
RDD1 (in memory)
t2
RDD2 (in disk)
t3
RDD3 (in memory)
a
Output Data on Disk
thecloudavenue.com


textFile = sc.textFile("SomeFile.txt")
RDD
Action
Value
Transformations
linesWithSpark.count()
74
linesWithSpark.first()
Apache Spark
linesWithSpark = textFile.filter(lambda line: "Spark" in line)

Does Spark Replace Hadoop?

- No! Spark provides an application framework
- Still needs to run on a storage system or on a no-SQL system

MapReduce vs Spark

- MapReduce restricted to map and reduce stages
- Spark optimized for more complex, multi-staged applications including machine learning and graph processing
- An RDD can be cached in memory for frequently cached data
- RDD graph can be placed in the most optimized manner on the Hadoop cluster
- In 2013,using MapReduce with Hadoop, 100TB of data was sorted with 2100 machines in 72 minutes, setting a new record
- In 2014, spark did the same with 207 machines in 23 minutes

RDD vs Distributed Shared Memory

- Writes in RDD are coarse grained(applies to all elements of dataset), vs fine grained in DSM, requires replicating data across nodes
- RDDs build a lineage graph: if something goes wrong they can go back and recompute based on lineage
- DSM uses checkpointing at predefined intervals
- Consistency is trivial in RDDs, as the data is assumed to be immutable
- DSM makes no assumptions about immutability, leaving consistency assumptions to the specific application

Spark Benefits

- Fault recovery using lineage
- Optimal computation placement using the direct acyclic graph
- Easy programming paradigm using transformations and actions on RDDs
- Rich library support for machine learning, data streaming, and more
- Spark Modules:Spark SQL, Spark Streaming, MLib, GraphX
- Concise and simple API compared to MapReduce APIs


SYSTEM
ML

History of SystemML

- Work started in 2007 at IBM Research – Almaden
- Through engagements with customers, researchers observed how data scientists made machine learning algorithms
- Created in 2010 by IBM researchers
- Previously data scientists would often prototype ML algorithms in R, Python, or other domain languages for small data
- When scaled to big data,a systems programmer would be brought in to scale the algorithm in a language such as Scala
- Goal to automatically scale an algorithm written in languages like R and Python to operate on big data

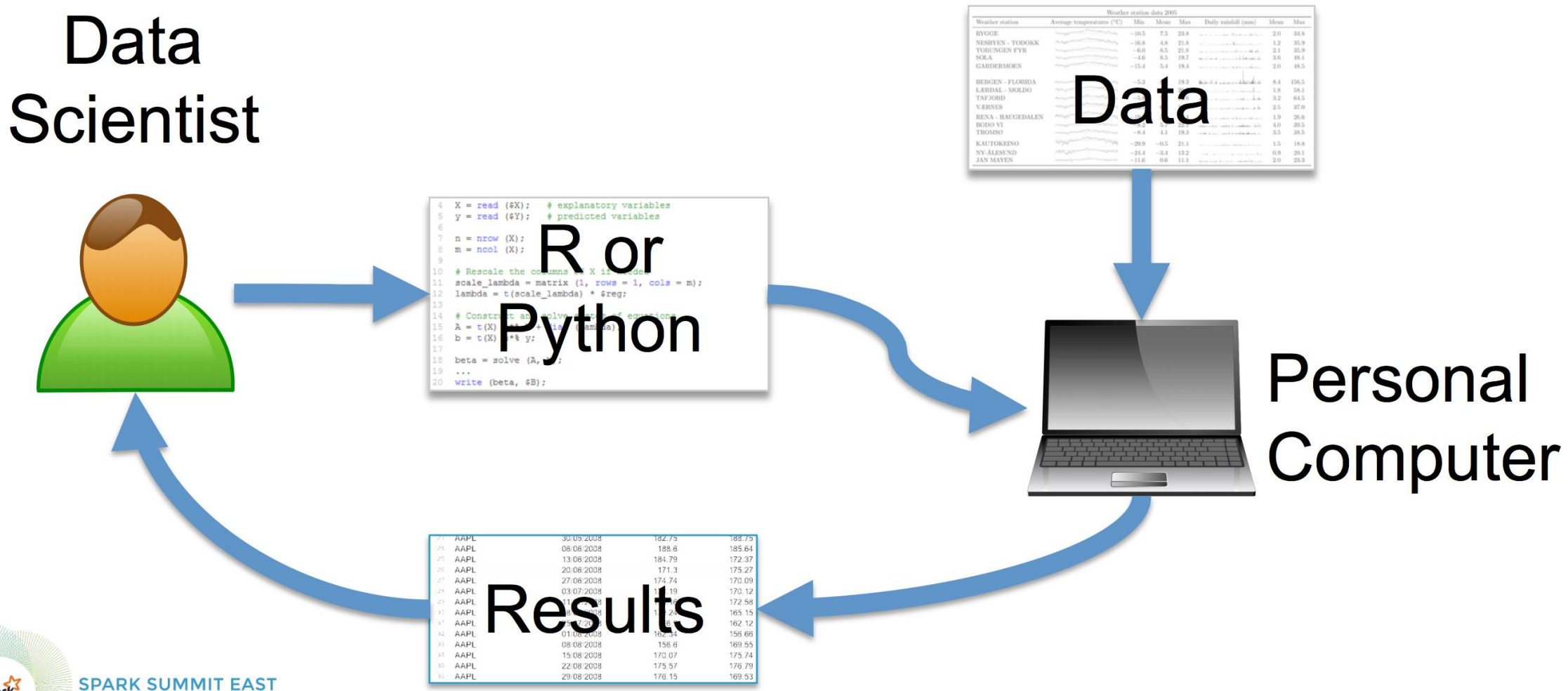
SystemML Development

- Began as an implementation on MapReduce as a data-parallel framework to share cluster resources with other MR-based systems
- Later evolved to utilize YARN(Yet Another Resource Negotiator): a Hadoop distributed operating system for big data applications
- YARN became a sub-project of Hadoop in 2012, aiming to decouple MR's resource management and scheduling capabilities from the data processing component, enabling Hadoop to support more varied processing approaches

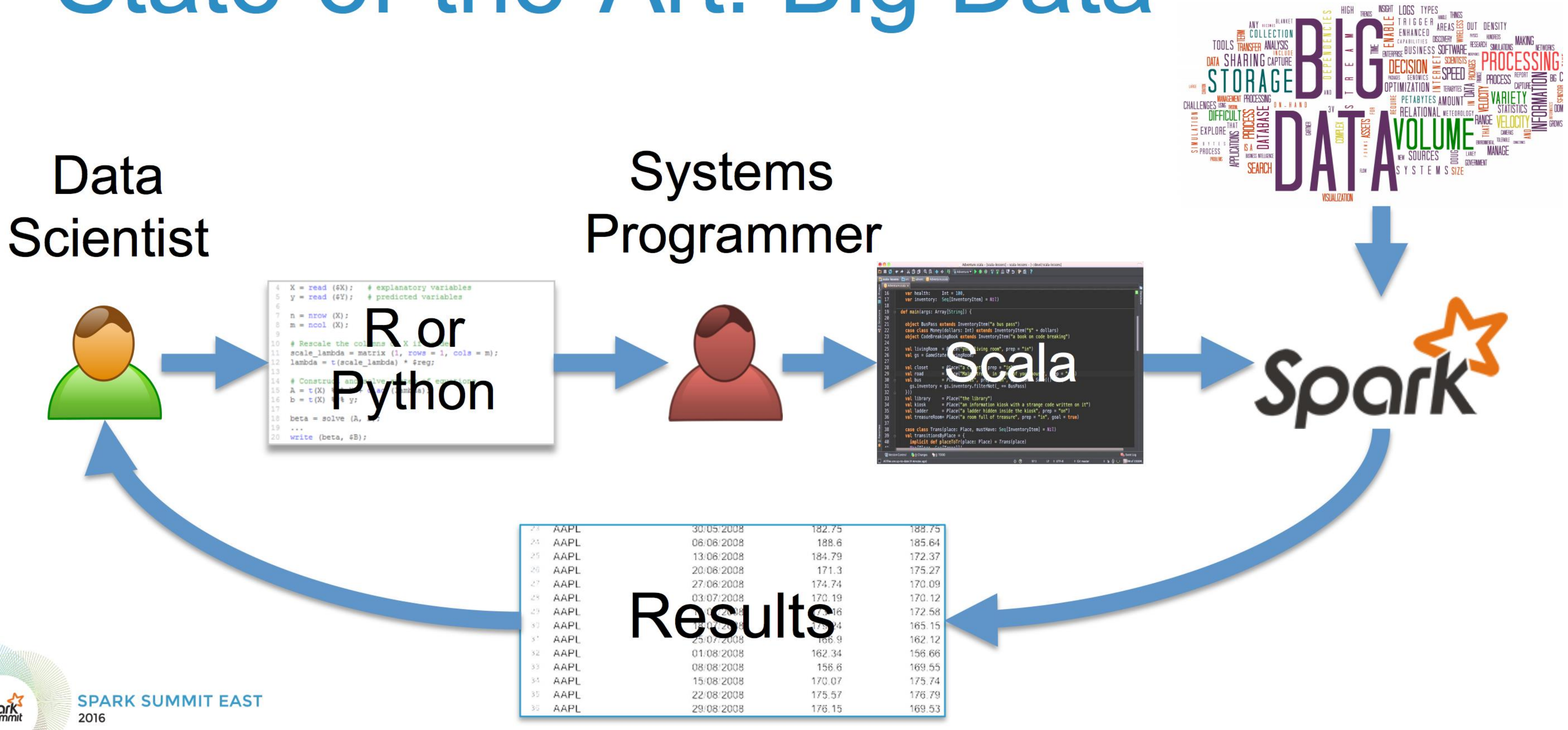
Requirements to Support Domain-Specific Analytics

- High-level semantics: understand, debug and control algorithm behavior
- Flexibility: specify new/customize existing ML algorithms
- Data independence: hide physical data representation (sparse/dense,row/column-major, blocking, partitioning, caching, compression)
- Scale independence: use the same code during small-scale and large-scale evaluation
- Pure in-memory, single node computation provides high efficiency on "small" use cases

State-of-the-Art: Small Data

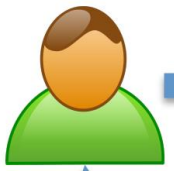


State-of-the-Art: Big Data



State-of-the-Art: Big Data

Data Scientist



- Days or weeks per iteration
- Errors while translating algorithms

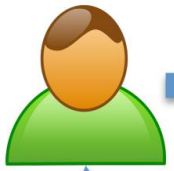
Spark

21	AAPL	30/05/2008	182.75	188.75
23	AAPL	06/06/2008	188.6	185.64
25	AAPL	13/06/2008	184.79	172.37
26	AAPL	20/06/2008	171.3	175.27
27	AAPL	27/06/2008	174.74	170.09
28	AAPL	03/07/2008	170.19	170.12
29	AAPL	[illegible]	[illegible]16	172.58
32	AAPL	[illegible]	[illegible]4	165.15
31	AAPL	25/07/2008	156.9	162.12
32	AAPL	01/08/2008	162.34	156.66
33	AAPL	08/08/2008	156.6	169.55
34	AAPL	15/08/2008	170.07	175.74
35	AAPL	22/08/2008	175.57	176.79
36	AAPL	29/08/2008	176.15	169.53

Results

The SystemML Vision

Data Scientist



```
X = read ($X);   # explanatory variables
y = read ($Y);   # predicted variables

n = nrow (X);
m = ncol (X);

# Rescale the columns
scale_lambda = matrix (1, rows = 1, cols = m);
lambda = t(scale_lambda) * $reg;

# Construct and solve the linear equations
A = t(X)
b = t(X)

beta = solve (A,
...
write (beta, $B);
```

R or Python

SystemML

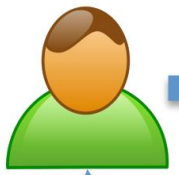
Spark

24	AAPL	30/05/2008	182.75	188.75
25	AAPL	06/06/2008	188.6	185.64
26	AAPL	13/06/2008	184.79	172.37
27	AAPL	20/06/2008	171.3	175.27
28	AAPL	27/06/2008	174.74	170.09
29	AAPL	03/07/2008	170.19	170.12
30	AAPL	[illegible]	[illegible]	172.58
31	AAPL	[illegible]	[illegible]	165.15
32	AAPL	25/07/2008	166.9	162.12
33	AAPL	01/08/2008	162.34	156.66
34	AAPL	08/08/2008	156.6	169.55
35	AAPL	15/08/2008	170.07	175.74
36	AAPL	22/08/2008	175.57	176.79
37	AAPL	29/08/2008	176.15	169.53

Results

The SystemML Vision

Data Scientist



```
X = read ($X);   # explanatory variables
y = read ($Y);   # predicted variables

n = nrow (X);
m = ncol (X);

# Rescale the columns of X if ...
scale_lambda = matrix (1, rows = 1, cols = m);
lambda = t(scale_lambda) * $reg;

# Construct ... solve ... equations
A = t(X) ...
b = t(X) ... y;

beta = solve (A, ...
...
write (beta, $B);
```

R or Python

😀 Fast iteration

😀 Same answer

SystemML

Spark

AAPL	30/05/2008	182.75	188.75
AAPL	06/06/2008	188.6	185.64
AAPL	13/06/2008	184.79	172.37
AAPL	20/06/2008	171.3	175.27
AAPL	27/06/2008	174.74	170.09
AAPL	03/07/2008	170.19	170.12
AAPL	[illegible]	[illegible]16	172.58
AAPL	[illegible]	[illegible]4	165.15
AAPL	25/07/2008	166.9	162.12
AAPL	01/08/2008	162.34	156.66
AAPL	08/08/2008	156.6	169.55
AAPL	15/08/2008	170.07	175.74
AAPL	22/08/2008	175.57	176.79
AAPL	29/08/2008	176.15	169.53

Results

SystemML Benefits

- Focuses on separating algorithm semantics from underlying data representations and runtime execution plans
- Leads to data scientist centric algorithm specification which improves productivity of data scientists
- Algorithm reusability and simplified deployment for varying data characteristics and runtime environments
- Automatic optimization of runtime execution plans

SystemML Language

- DML (Declarative ML): features R or Python-like syntax, provides many linear algebra and statistical functions
- DML scripts are parsed into a hierarchy of statement blocks and statements defined by control structures, and performs analysis
- Constructs directed acyclic graphs of high-level operators per statement block

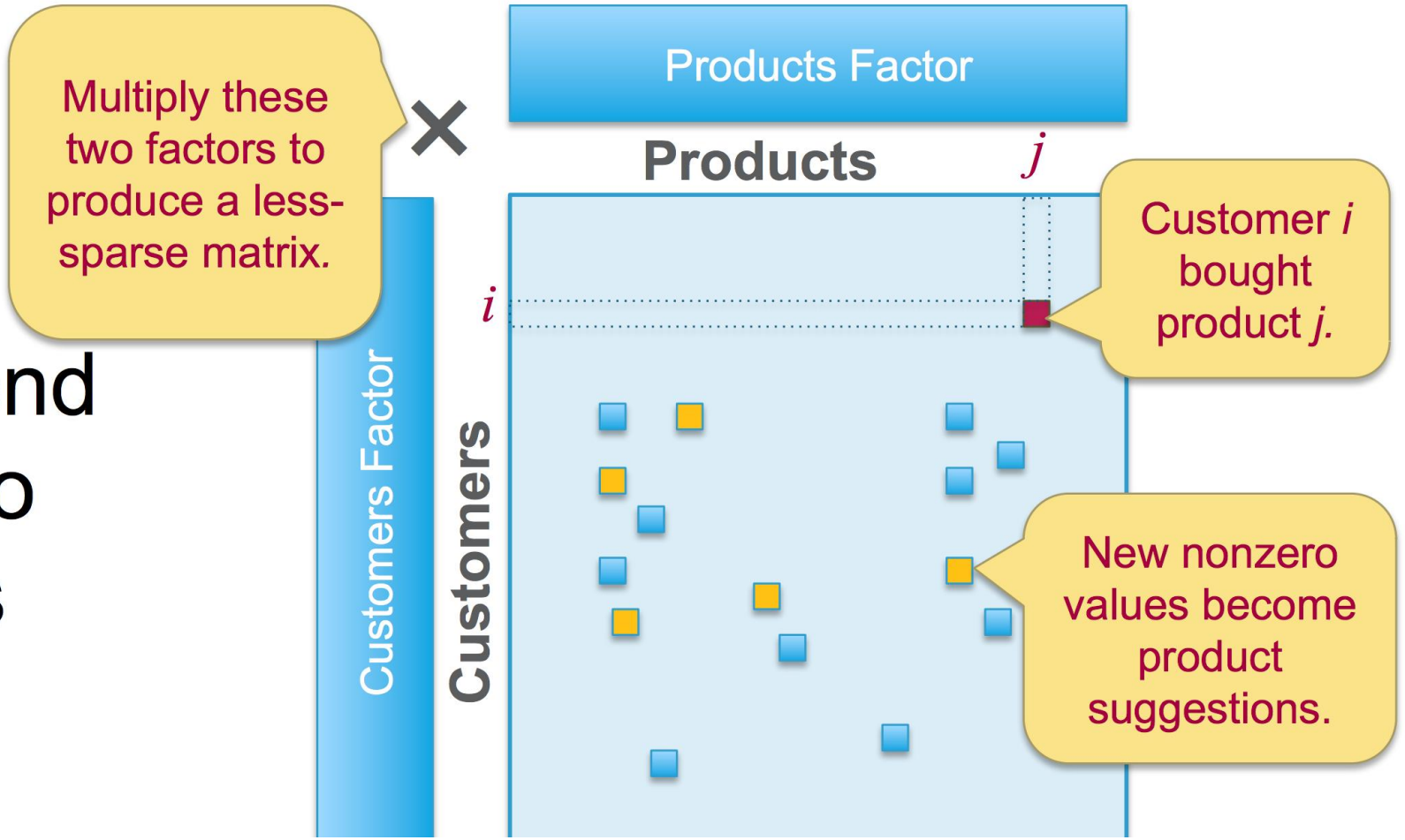
Running Example

- DML script of the alternating least squares (ALS)algorithm for matrix completion
- ALS tries to decompose a matrix X into two factor matrices U and V of low rank such that X = UV

Running Example:

Alternating Least Squares

- Problem: Recommend products to customers



Users

Alice	0.9	-0.1
Bob	-0.8	0.5
Corey	0.1	0.8

Items

			
0.8	0.9	-0.7	-0.8
0.9	0.1	0.7	-0.5

score[Corey, ] = 0.1*0.9 + 0.8*0.1 = 0.17

Alternating Least Squares (spark.ml)

```scala
/**
 * :: DeveloperApi ::
 * Implementation of the ALS algorithm.
 */
@DeveloperApi
def train[ID: ClassTag]( // scalastyle:ignore
    ratings: RDD[Rating[ID]],
    rank: Int = 10,
    numUserBlocks: Int = 10,
    numItemBlocks: Int = 10,
    maxIter: Int = 10,
    regParam: Double = 1.0,
    implicitPrefs: Boolean = false,
    alpha: Double = 1.0,
    nonnegative: Boolean = false,
    intermediateRDDStorageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK,
    finalRDDStorageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK,
    checkpointInterval: Int = 10,
    seed: Long = 0L)(
    implicit ord: Ordering[ID]): (RDD[(ID, Array[Float])], RDD[(ID, Array[Float])]) = {
  require(intermediateRDDStorageLevel != StorageLevel.NONE,
    "ALS is not designed to run without persisting intermediate RDDs.")
  val sc = ratings.sparkContext
  val userPart = new ALSPartitioner(numUserBlocks)
  val itemPart = new ALSPartitioner(numItemBlocks)
  val userLocalIndexEncoder = new LocalIndexEncoder(userPart.numPartitions)
  val itemLocalIndexEncoder = new LocalIndexEncoder(itemPart.numPartitions)
  val solver = if (nonnegative) new NNLSSolver else new CholeskySolver
  val blockRatings = partitionRatings(ratings, userPart, itemPart)
    .persist(intermediateRDDStorageLevel)
  val (userInBlocks, userOutBlocks) =
    makeBlocks("user", blockRatings, userPart, itemPart, intermediateRDDStorageLevel)
  // materialize blockRatings and user blocks
  userOutBlocks.count()
  val swappedBlockRatings = blockRatings.map {
    case ((userBlockId, itemBlockId), RatingBlock(userIds, itemIds, localRatings)) =>
      ((itemBlockId, userBlockId), RatingBlock(itemIds, userIds, localRatings))
  }
  val (itemInBlocks, itemOutBlocks) =
    makeBlocks("item", swappedBlockRatings, itemPart, userPart, intermediateRDDStorageLevel)
  // materialize item blocks
  itemOutBlocks.count()
  val seedGen = new XORShiftRandom(seed)
  var userFactors = initialize(userInBlocks, rank, seedGen.nextLong())
  var itemFactors = initialize(itemInBlocks, rank, seedGen.nextLong())
  var previousCheckpointFile: Option[String] = None
  val shouldCheckpoint: Int => Boolean = (iter) =>
    sc.checkpointDir.isDefined && checkpointInterval != -1 && (iter % checkpointInterval == 0)
  val deletePreviousCheckpointFile: () => Unit = () =>
    previousCheckpointFile.foreach { file =>
      try {
        FileSystem.get(sc.hadoopConfiguration).delete(new Path(file), true)
      } catch {
        case e: IOException =>
          logWarning(s"Cannot delete checkpoint file $file:", e)
      }
    }
```

```scala
      srcIds += r.user
      dstIds += r.item
      ratings += r.rating
      this
    }

    /** Merges another [[RatingBlockBuilder]]. */
    def merge(other: RatingBlock[ID]): this.type = {
      size += other.srcIds.length
      srcIds ++= other.srcIds
      dstIds ++= other.dstIds
      ratings ++= other.ratings
      this
    }

    /** Builds a [[RatingBlock]]. */
    def build(): RatingBlock[ID] = {
      RatingBlock[ID](srcIds.result(), dstIds.result(), ratings.result())
    }
  }

  /**
   * Partitions raw ratings into blocks.
   *
   * @param ratings raw ratings
   * @param srcPart partitioner for src IDs
   * @param dstPart partitioner for dst IDs
   *
   * @return an RDD of rating blocks in the form of ((srcBlockId, dstBlockId), ratingBlock)
   */
  private def partitionRatings[ID: ClassTag](
      ratings: RDD[Rating[ID]],
      srcPart: Partitioner,
      dstPart: Partitioner): RDD[((Int, Int), RatingBlock[ID])] = {

    /* The implementation produces the same result as the following but generates less objec

    ratings.map { r =>
      ((srcPart.getPartition(r.user), dstPart.getPartition(r.item)), r)
    }.aggregateByKey(new RatingBlockBuilder)(
        seqOp = (b, r) => b.add(r),
        combOp = (b0, b1) => b0.merge(b1.build()))
      .mapValues(_.build())
    */

    val numPartitions = srcPart.numPartitions * dstPart.numPartitions
    ratings.mapPartitions { iter =>
      val builders = Array.fill(numPartitions)(new RatingBlockBuilder[ID])
      iter.flatMap { r =>
        val srcBlockId = srcPart.getPartition(r.user)
        val dstBlockId = dstPart.getPartition(r.item)
        val idx = srcBlockId + srcPart.numPartitions * dstBlockId
        val builder = builders(idx)
        builder.add(r)
        if (builder.size >= 2048) { // 2048 * (3 * 4) = 24k
          builders(idx) = new RatingBlockBuilder
          Iterator.single(((srcBlockId, dstBlockId), builder.build()))
```

```scala
        } else {
          Iterator.empty
        }
      } ++ {
        builders.view.zipWithIndex.filter(_._1.size > 0).map { case (block, idx) =>
          val srcBlockId = idx % srcPart.numPartitions
          val dstBlockId = idx / srcPart.numPartitions
          ((srcBlockId, dstBlockId), block.build())
        }
      }
    }.groupByKey().mapValues { blocks =>
      val builder = new RatingBlockBuilder[ID]
      blocks.foreach(builder.merge)
      builder.build()
    }.setName("ratingBlocks")
  }

  /**
   * Builder for uncompressed in-blocks of (srcId, dstEncodedIndex, rating) tuples.
   * @param encoder encoder for dst indices
   */
  private[recommendation] class UncompressedInBlockBuilder[@specialized(Int, Long) ID: ClassTag](
      encoder: LocalIndexEncoder)(
      implicit ord: Ordering[ID]) {

    private val srcIds = mutable.ArrayBuilder.make[ID]
    private val dstEncodedIndices = mutable.ArrayBuilder.make[Int]
    private val ratings = mutable.ArrayBuilder.make[Float]

    /**
     * Adds a dst block of (srcId, dstLocalIndex, rating) tuples.
     *
     * @param dstBlockId dst block ID
     * @param srcIds original src IDs
     * @param dstLocalIndices dst local indices
     * @param ratings ratings
     */
    def add(
        dstBlockId: Int,
        srcIds: Array[ID],
        dstLocalIndices: Array[Int],
        ratings: Array[Float]): this.type = {
      val sz = srcIds.length
      require(dstLocalIndices.length == sz)
      require(ratings.length == sz)
      this.srcIds ++= srcIds
      this.ratings ++= ratings
      var j = 0
      while (j < sz) {
        this.dstEncodedIndices += encoder.encode(dstBlockId, dstLocalIndices(j))
        j += 1
      }
      this
    }

    /** Builds a [[UncompressedInBlock]]. */
    def build(): UncompressedInBlock[ID] = {
```

Alternating Least Squares (spark.ml)

```
if (implicitPrefs) {
  for (iter <- 1 to maxIter) {
    userFactors.setName(s"userFactors-$iter").persist(intermediateRDDStorageLevel)
    val previousItemFactors = itemFactors
    itemFactors = computeFactors(userFactors, userOutBlocks, itemInBlocks, rank, regPara
      userLocalIndexEncoder, implicitPrefs, alpha, solver)
    previousItemFactors.unpersist()
    itemFactors.setName(s"itemFactors-$iter").persist(intermediateRDDStorageLevel)
    // TODO: Generalize PeriodicGraphCheckpointer and use it here.
    if (shouldCheckpoint(iter)) {
      itemFactors.checkpoint() // itemFactors gets materialized in computeFactors.
    }
    val previousUserFactors = userFactors
    userFactors = computeFactors(itemFactors, itemOutBlocks, userInBlocks, rank, regPara
      itemLocalIndexEncoder, implicitPrefs, alpha, solver)
    if (shouldCheckpoint(iter)) {
      deletePreviousCheckpointFile()
      previousCheckpointFile = itemFactors.getCheckpointFile
    }
    previousUserFactors.unpersist()
  }
} else {
  for (iter <- 0 until maxIter) {
    itemFactors = computeFactors(userFactors, userOutBlocks, itemInBlocks, rank, regPara
      userLocalIndexEncoder, solver = solver)
    if (shouldCheckpoint(iter)) {
      itemFactors.checkpoint()
      itemFactors.count() // checkpoint item factors and cut lineage
      deletePreviousCheckpointFile()
      previousCheckpointFile = itemFactors.getCheckpointFile
    }
    userFactors = computeFactors(itemFactors, itemOutBlocks, userInBlocks, rank, regPara
      itemLocalIndexEncoder, solver = solver)
  }
}
val userIdAndFactors = userInBlocks
  .mapValues(_.srcIds)
  .join(userFactors)
  .mapPartitions({ items =>
    items.flatMap { case (_, (ids, factors)) =>
      ids.view.zip(factors)
    }
  // Preserve the partitioning because IDs are consistent with the partitioners in userIn
  // and userFactors.
  }, preservesPartitioning = true)
  .setName("userFactors")
  .persist(finalRDDStorageLevel)
val itemIdAndFactors = itemInBlocks
  .mapValues(_.srcIds)
  .join(itemFactors)
  .mapPartitions({ items =>
    items.flatMap { case (_, (ids, factors)) =>
      ids.view.zip(factors)
    }
  }, preservesPartitioning = true)
  .setName("itemFactors")
  .persist(finalRDDStorageLevel)
if (finalRDDStorageLevel != StorageLevel.NONE) {
  userIdAndFactors.count()
  itemFactors.unpersist()
  itemIdAndFactors.count()
  userInBlocks.unpersist()
  userOutBlocks.unpersist()
  itemInBlocks.unpersist()
  itemOutBlocks.unpersist()
  blockRatings.unpersist()
}
(userIdAndFactors, itemIdAndFactors)
}

/**
 * Factor block that stores factors (Array[Float]) in an Array.
 */
private type FactorBlock = Array[Array[Float]]

/**
 * Out-link block that stores, for each dst (item/user) block, which src (user/item) factors to
 * send. For example, outLinkBlock(0) contains the local indices (not the original src IDs) of t
 * src factors in this block to send to dst block 0.
 */
private type OutBlock = Array[Array[Int]]

/**
 * In-link block for computing src (user/item) factors. This includes the original src IDs
 * of the elements within this block as well as encoded dst (item/user) indices and correspondi
 * ratings. The dst indices are in the form of (blockId, localIndex), which are not the original
 * dst IDs. To compute src factors, we expect receiving dst factors that match the dst indices.
 * For example, if we have an in-link record
 *
 * {srcId: 0, dstBlockId: 2, dstLocalIndex: 3, rating: 5.0},
 *
 * and assume that the dst factors are stored as dstFactors: Map[Int, Array[Array[Float]]], whic
 * is a blockId to dst factors map, the corresponding dst factor of the record is dstFactor(2)(3
 *
 * We use a CSC-like (compressed sparse column) format to store the in-link information. So we c
 * compute src factors one after another using only one normal equation instance.
 *
 * @param srcIds src ids (ordered)
 * @param dstPtrs dst pointers. Elements in range [dstPtrs(i), dstPtrs(i+1)) of dst indices and
 *                ratings are associated with srcIds(i).
 * @param dstEncodedIndices encoded dst indices
 * @param ratings ratings
 *
 * @see [[LocalIndexEncoder]]
 */
private[recommendation] case class InBlock[@specialized(Int, Long) ID: ClassTag](
    srcIds: Array[ID],
    dstPtrs: Array[Int],
    dstEncodedIndices: Array[Int],
    ratings: Array[Float]) {
  /** Size of the block. */
  def size: Int = ratings.length
  require(dstEncodedIndices.length == size)
  require(dstPtrs.length == srcIds.length + 1)
}

/**
 * Initializes factors randomly given the in-link blocks.
 *
 * @param inBlocks in-link blocks
 * @param rank rank
 * @return initialized factor blocks
 */
private def initialize[ID](
    inBlocks: RDD[(Int, InBlock[ID])],
    rank: Int,
    seed: Long): RDD[(Int, FactorBlock)] = {
  // Choose a unit vector uniformly at random from the unit sphere, but from the
  // "first quadrant" where all elements are nonnegative. This can be done by choosing
  // elements distributed as Normal(0,1) and taking the absolute value, and then normalizing.
  // This appears to create factorizations that have a slightly better reconstruction
  // (<1%) compared picking elements uniformly at random in [0,1].
  inBlocks.map { case (srcBlockId, inBlock) =>
    val random = new XORShiftRandom(byteswap64(seed ^ srcBlockId))
    val factors = Array.fill(inBlock.srcIds.length) {
      val factor = Array.fill(rank)(random.nextGaussian().toFloat)
      val nrm = blas.snrm2(rank, factor, 1)
      blas.sscal(rank, 1.0f / nrm, factor, 1)
      factor
    }
    (srcBlockId, factors)
  }
}

/**
 * A rating block that contains src IDs, dst IDs, and ratings, stored in primitive arrays.
 */
private[recommendation] case class RatingBlock[@specialized(Int, Long) ID: ClassTag](
    srcIds: Array[ID],
    dstIds: Array[ID],
    ratings: Array[Float]) {
  /** Size of the block. */
  def size: Int = srcIds.length
  require(dstIds.length == srcIds.length)
  require(ratings.length == srcIds.length)
}

/**
 * Builder for [[RatingBlock]]. [[mutable.ArrayBuilder]] is used to avoid boxing/unboxing.
 */
private[recommendation] class RatingBlockBuilder[@specialized(Int, Long) ID: ClassTag]
  extends Serializable {

  private val srcIds = mutable.ArrayBuilder.make[ID]
  private val dstIds = mutable.ArrayBuilder.make[ID]
  private val ratings = mutable.ArrayBuilder.make[Float]
  var size = 0

  /** Adds a rating. */
  def add(r: Rating[ID]): this.type = {
    size += 1
```

Alternating Least Squares (spark.ml)

```
    new UncompressedInBlock(srcIds.result(), dstEncodedIndices.result(), ratings.result())
  }
}

/**
 * A block of (srcId, dstEncodedIndex, rating) tuples stored in primitive arrays.
 */
private[recommendation] class UncompressedInBlock[@specialized(Int, Long) ID: ClassTag](
    val srcIds: Array[ID],
    val dstEncodedIndices: Array[Int],
    val ratings: Array[Float])(
    implicit ord: Ordering[ID]) {

  /** Size the of block. */
  def length: Int = srcIds.length

  /**
   * Compresses the block into an [[InBlock]]. The algorithm is the same as converting a
   * sparse matrix from coordinate list (COO) format into compressed sparse column (CSC) format.
   * Sorting is done using Spark's built-in Timsort to avoid generating too many objects.
   */
  def compress(): InBlock[ID] = {
    val sz = length
    assert(sz > 0, "Empty in-link block should not exist.")
    sort()
    val uniqueSrcIdsBuilder = mutable.ArrayBuilder.make[ID]
    val dstCountsBuilder = mutable.ArrayBuilder.make[Int]
    var preSrcId = srcIds(0)
    uniqueSrcIdsBuilder += preSrcId
    var curCount = 1
    var i = 1
    var j = 0
    while (i < sz) {
      val srcId = srcIds(i)
      if (srcId != preSrcId) {
        uniqueSrcIdsBuilder += srcId
        dstCountsBuilder += curCount
        preSrcId = srcId
        j += 1
        curCount = 0
      }
      curCount += 1
      i += 1
    }
    dstCountsBuilder += curCount
    val uniqueSrcIds = uniqueSrcIdsBuilder.result()
    val numUniqueSrdIds = uniqueSrcIds.length
    val dstCounts = dstCountsBuilder.result()
    val dstPtrs = new Array[Int](numUniqueSrdIds + 1)
    var sum = 0
    i = 0
    while (i < numUniqueSrdIds) {
      sum += dstCounts(i)
      i += 1
      dstPtrs(i) = sum
    }
    InBlock(uniqueSrcIds, dstPtrs, dstEncodedIndices, ratings)
  }

  private def sort(): Unit = {
    val sz = length
    // Since there might be interleaved log messages, we insert a unique id for easy pairing.
    val sortId = Utils.random.nextInt()
    logDebug(s"Start sorting an uncompressed in-block of size $sz. (sortId = $sortId)")
    val start = System.nanoTime()
    val sorter = new Sorter(new UncompressedInBlockSort[ID])
    sorter.sort(this, 0, length, Ordering[KeyWrapper[ID]])
    val duration = (System.nanoTime() - start) / 1e9
    logDebug(s"Sorting took $duration seconds. (sortId = $sortId)")
  }
}

/**
 * A wrapper that holds a primitive key.
 *
 * @see [[UncompressedInBlockSort]]
 */
private class KeyWrapper[@specialized(Int, Long) ID: ClassTag](
    implicit ord: Ordering[ID]) extends Ordered[KeyWrapper[ID]] {

  var key: ID = _

  override def compare(that: KeyWrapper[ID]): Int = {
    ord.compare(key, that.key)
  }

  def setKey(key: ID): this.type = {
    this.key = key
    this
  }
}

/**
 * [[SortDataFormat]] of [[UncompressedInBlock]] used by [[Sorter]].
 */
private class UncompressedInBlockSort[@specialized(Int, Long) ID: ClassTag](
    implicit ord: Ordering[ID])
  extends SortDataFormat[KeyWrapper[ID], UncompressedInBlock[ID]] {

  override def newKey(): KeyWrapper[ID] = new KeyWrapper()

  override def getKey(
      data: UncompressedInBlock[ID],
      pos: Int,
      reuse: KeyWrapper[ID]): KeyWrapper[ID] = {
    if (reuse == null) {
      new KeyWrapper().setKey(data.srcIds(pos))
    } else {
      reuse.setKey(data.srcIds(pos))
    }
  }

  override def getKey(
      data: UncompressedInBlock[ID],
      pos: Int): KeyWrapper[ID] = {
    getKey(data, pos, null)
  }

  private def swapElements[@specialized(Int, Float) T](
      data: Array[T],
      pos0: Int,
      pos1: Int): Unit = {
    val tmp = data(pos0)
    data(pos0) = data(pos1)
    data(pos1) = tmp
  }

  override def swap(data: UncompressedInBlock[ID], pos0: Int, pos1: Int): Unit = {
    swapElements(data.srcIds, pos0, pos1)
    swapElements(data.dstEncodedIndices, pos0, pos1)
    swapElements(data.ratings, pos0, pos1)
  }

  override def copyRange(
      src: UncompressedInBlock[ID],
      srcPos: Int,
      dst: UncompressedInBlock[ID],
      dstPos: Int,
      length: Int): Unit = {
    System.arraycopy(src.srcIds, srcPos, dst.srcIds, dstPos, length)
    System.arraycopy(src.dstEncodedIndices, srcPos, dst.dstEncodedIndices, dstPos, length)
    System.arraycopy(src.ratings, srcPos, dst.ratings, dstPos, length)
  }

  override def allocate(length: Int): UncompressedInBlock[ID] = {
    new UncompressedInBlock(
      new Array[ID](length), new Array[Int](length), new Array[Float](length))
  }

  override def copyElement(
      src: UncompressedInBlock[ID],
      srcPos: Int,
      dst: UncompressedInBlock[ID],
      dstPos: Int): Unit = {
    dst.srcIds(dstPos) = src.srcIds(srcPos)
    dst.dstEncodedIndices(dstPos) = src.dstEncodedIndices(srcPos)
    dst.ratings(dstPos) = src.ratings(srcPos)
  }
}

/**
 * Creates in-blocks and out-blocks from rating blocks.
 * @param prefix prefix for in/out-block names
 * @param ratingBlocks rating blocks
 * @param srcPart partitioner for src IDs
 * @param dstPart partitioner for dst IDs
 * @return (in-blocks, out-blocks)
 */
private def makeBlocks[ID: ClassTag](
    prefix: String,
    ratingBlocks: RDD[((Int, Int), RatingBlock[ID])],
```

Alternating Least Squares (spark.ml)

```
    srcPart: Partitioner,
    dstPart: Partitioner,
    storageLevel: StorageLevel)(
    implicit srcOrd: Ordering[ID]): (RDD[(Int, InBlock[ID])], RDD[(Int, OutBlock)]) = {
  val inBlocks = ratingBlocks.map {
    case ((srcBlockId, dstBlockId), RatingBlock(srcIds, dstIds, ratings)) =>
      // The implementation is a faster version of
      // val dstIdToLocalIndex = dstIds.toSet.toSeq.sorted.zipWithIndex.toMap
      val start = System.nanoTime()
      val dstIdSet = new OpenHashSet[ID](1 << 20)
      dstIds.foreach(dstIdSet.add)
      val sortedDstIds = new Array[ID](dstIdSet.size)
      var i = 0
      var pos = dstIdSet.nextPos(0)
      while (pos != -1) {
        sortedDstIds(i) = dstIdSet.getValue(pos)
        pos = dstIdSet.nextPos(pos + 1)
        i += 1
      }
      assert(i == dstIdSet.size)
      Sorting.quickSort(sortedDstIds)
      val dstIdToLocalIndex = new OpenHashMap[ID, Int](sortedDstIds.length)
      i = 0
      while (i < sortedDstIds.length) {
        dstIdToLocalIndex.update(sortedDstIds(i), i)
        i += 1
      }
      logDebug(
        "Converting to local indices took " + (System.nanoTime() - start) / 1e9 + " seco
      val dstLocalIndices = dstIds.map(dstIdToLocalIndex.apply)
      (srcBlockId, (dstBlockId, srcIds, dstLocalIndices, ratings))
  }.groupByKey(new ALSPartitioner(srcPart.numPartitions))
    .mapValues { iter =>
      val builder =
        new UncompressedInBlockBuilder[ID](new LocalIndexEncoder(dstPart.numPartitions))
      iter.foreach { case (dstBlockId, srcIds, dstLocalIndices, ratings) =>
        builder.add(dstBlockId, srcIds, dstLocalIndices, ratings)
      }
      builder.build().compress()
    }.setName(prefix + "InBlocks")
    .persist(storageLevel)
  val outBlocks = inBlocks.mapValues { case InBlock(srcIds, dstPtrs, dstEncodedIndices, _
    val encoder = new LocalIndexEncoder(dstPart.numPartitions)
    val activeIds = Array.fill(dstPart.numPartitions)(mutable.ArrayBuilder.make[Int])
    var i = 0
    val seen = new Array[Boolean](dstPart.numPartitions)
    while (i < srcIds.length) {
      var j = dstPtrs(i)
      ju.Arrays.fill(seen, false)
      while (j < dstPtrs(i + 1)) {
        val dstBlockId = encoder.blockId(dstEncodedIndices(j))
        if (!seen(dstBlockId)) {
          activeIds(dstBlockId) += i // add the local index in this out-block
          seen(dstBlockId) = true
        }
        j += 1
      }
      i += 1
    }
    activeIds.map { x =>
      x.result()
    }
  }.setName(prefix + "OutBlocks")
    .persist(storageLevel)
  (inBlocks, outBlocks)
}

/**
 * Compute dst factors by constructing and solving least square problems.
 *
 * @param srcFactorBlocks src factors
 * @param srcOutBlocks src out-blocks
 * @param dstInBlocks dst in-blocks
 * @param rank rank
 * @param regParam regularization constant
 * @param srcEncoder encoder for src local indices
 * @param implicitPrefs whether to use implicit preference
 * @param alpha the alpha constant in the implicit preference formulation
 * @param solver solver for least squares problems
 *
 * @return dst factors
 */
private def computeFactors[ID](
    srcFactorBlocks: RDD[(Int, FactorBlock)],
    srcOutBlocks: RDD[(Int, OutBlock)],
    dstInBlocks: RDD[(Int, InBlock[ID])],
    rank: Int,
    regParam: Double,
    srcEncoder: LocalIndexEncoder,
    implicitPrefs: Boolean = false,
    alpha: Double = 1.0,
    solver: LeastSquaresNESolver): RDD[(Int, FactorBlock)] = {
  val numSrcBlocks = srcFactorBlocks.partitions.length
  val YtY = if (implicitPrefs) Some(computeYtY(srcFactorBlocks, rank)) else None
  val srcOut = srcOutBlocks.join(srcFactorBlocks).flatMap {
    case (srcBlockId, (srcOutBlock, srcFactors)) =>
      srcOutBlock.view.zipWithIndex.map { case (activeIndices, dstBlockId) =>
        (dstBlockId, (srcBlockId, activeIndices.map(idx => srcFactors(idx))))
      }
  }
  val merged = srcOut.groupByKey(new ALSPartitioner(dstInBlocks.partitions.length))
  dstInBlocks.join(merged).mapValues {
    case (InBlock(dstIds, srcPtrs, srcEncodedIndices, ratings), srcFactors) =>
      val sortedSrcFactors = new Array[FactorBlock](numSrcBlocks)
      srcFactors.foreach { case (srcBlockId, factors) =>
        sortedSrcFactors(srcBlockId) = factors
      }
      val dstFactors = new Array[Array[Float]](dstIds.length)
      var j = 0
      val ls = new NormalEquation(rank)
      while (j < dstIds.length) {
        ls.reset()
        if (implicitPrefs) {
          ls.merge(YtY.get)
        }
        var i = srcPtrs(j)
        var numExplicits = 0
        while (i < srcPtrs(j + 1)) {
          val encoded = srcEncodedIndices(i)
          val blockId = srcEncoder.blockId(encoded)
          val localIndex = srcEncoder.localIndex(encoded)
          val srcFactor = sortedSrcFactors(blockId)(localIndex)
          val rating = ratings(i)
          if (implicitPrefs) {
            // Extension to the original paper to handle b < 0. confidence is a function of |b|
            // instead so that it is never negative. c1 is confidence - 1.0.
            val c1 = alpha * math.abs(rating)
            // For rating <= 0, the corresponding preference is 0. So the term below is only added
            // for rating > 0. Because YtY is already added, we need to adjust the scaling here.
            if (rating > 0) {
              numExplicits += 1
              ls.add(srcFactor, (c1 + 1.0) / c1, c1)
            }
          } else {
            ls.add(srcFactor, rating)
            numExplicits += 1
          }
          i += 1
        }
        // Weight lambda by the number of explicit ratings based on the ALS-WR paper.
        dstFactors(j) = solver.solve(ls, numExplicits * regParam)
        j += 1
      }
      dstFactors
  }
}

/**
 * Computes the Gramian matrix of user or item factors, which is only used in implicit preference.
 * Caching of the input factors is handled in [[ALS#train]].
 */
private def computeYtY(factorBlocks: RDD[(Int, FactorBlock)], rank: Int): NormalEquation = {
  factorBlocks.values.aggregate(new NormalEquation(rank))(
    seqOp = (ne, factors) => {
      factors.foreach(ne.add(_, 0.0))
      ne
    },
    combOp = (ne1, ne2) => ne1.merge(ne2))
}

/**
 * Encoder for storing (blockId, localIndex) into a single integer.
 *
 * We use the leading bits (including the sign bit) to store the block id and the rest to store
 * the local index. This is based on the assumption that users/items are approximately evenly
 * partitioned. With this assumption, we should be able to encode two billion distinct values.
 *
 * @param numBlocks number of blocks
 */
private[recommendation] class LocalIndexEncoder(numBlocks: Int) extends Serializable {
  require(numBlocks > 0, s"numBlocks must be positive but found $numBlocks.")

  private[this] final val numLocalIndexBits =
    math.min(java.lang.Integer.numberOfLeadingZeros(numBlocks - 1), 31)
  private[this] final val localIndexMask = (1 << numLocalIndexBits) - 1
```

```scala
  /** Encodes a (blockId, localIndex) into a single integer. */
  def encode(blockId: Int, localIndex: Int): Int = {
    require(blockId < numBlocks)
    require((localIndex & ~localIndexMask) == 0)
    (blockId << numLocalIndexBits) | localIndex
  }

  /** Gets the block id from an encoded index. */
  @inline
  def blockId(encoded: Int): Int = {
    encoded >>> numLocalIndexBits
  }

  /** Gets the local index from an encoded index. */
  @inline
  def localIndex(encoded: Int): Int = {
    encoded & localIndexMask
  }
}

/**
 * Partitioner used by ALS. We requires that getPartition is a projection. That is, for any key k,
 * we have getPartition(getPartition(k)) = getPartition(k). Since the the default HashPartitioner
 * satisfies this requirement, we simply use a type alias here.
 */
private[recommendation] type ALSPartitioner = org.apache.spark.HashPartitioner
```

- 25 lines' worth of algorithm…
- …mixed with 800 lines of performance code

Alternating Least Squares(in R)

```
U = rand(nrow(X), r, min = -1.0, max = 1.0);
V = rand(r, ncol(X), min = -1.0, max = 1.0);
while(i < mi) {
   i = i + 1; ii = 1;
   if (is_U)
      G = (W * (U %*% V - X)) %*% t(V) + lambda * U;
   else
      G = t(U) %*% (W * (U %*% V - X)) + lambda * V;
   norm_G2 = sum(G ^ 2); norm_R2 = norm_G2;
   R = -G; S = R;
   while(norm_R2 > 10E-9 * norm_G2 & ii <= mii) {
     if (is_U) {
       HS = (W * (S %*% V)) %*% t(V) + lambda * S;
       alpha = norm_R2 / sum (S * HS);
       U = U + alpha * S;
     } else {
       HS = t(U) %*% (W * (U %*% S)) + lambda * S;
       alpha = norm_R2 / sum (S * HS);
       V = V + alpha * S;
     }
     R = R - alpha * HS;
     old_norm_R2 = norm_R2; norm_R2 = sum(R ^ 2);
     S = R + (norm_R2 / old_norm_R2) * S;
     ii = ii + 1;
   }
   is_U = ! is_U;
}
```

Alternating Least Squares (in SystemML's subset of R)

```
U = rand(nrow(X), r, min = -1.0, max = 1.0);
V = rand(r, ncol(X), min = -1.0, max = 1.0);
while(i < mi) {
   i = i + 1; ii = 1;
   if (is_U)
      G = (W * (U %*% V - X)) %*% t(V) + lambda * U;
   else
      G = t(U) %*% (W * (U %*% V - X)) + lambda * V;
   norm_G2 = sum(G ^ 2); norm_R2 = norm_G2;
   R = -G; S = R;
   while(norm_R2 > 10E-9 * norm_G2 & ii <= mii) {
     if (is_U) {
       HS = (W * (S %*% V)) %*% t(V) + lambda * S;
       alpha = norm_R2 / sum (S * HS);
       U = U + alpha * S;
     } else {
       HS = t(U) %*% (W * (U %*% S)) + lambda * S;
       alpha = norm_R2 / sum (S * HS);
       V = V + alpha * S;
     }
     R = R - alpha * HS;
     old_norm_R2 = norm_R2; norm_R2 = sum(R ^ 2);
     S = R + (norm_R2 / old_norm_R2) * S;
     ii = ii + 1;
   }
   is_U = ! is_U;
}
```

SystemML Optimization

- Automatically generates hybrid runtime execution plans that are composed of single-node and distributed operations depending on data and cluster characteristics such as data size, sparsity, cluster size, memory configurations
- Exploits the capabilities of underlying data-parallel frameworks such as MR or Spark

SystemML Runtime

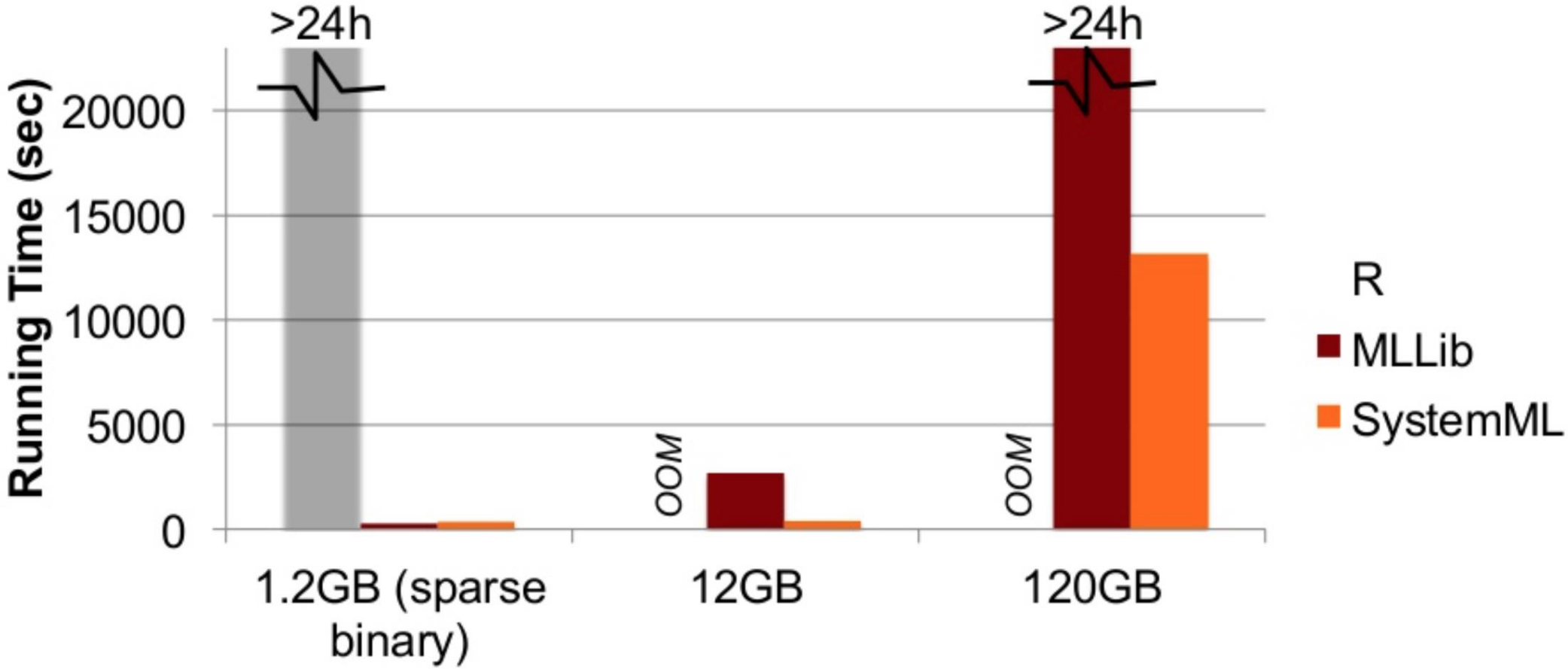
- For MR backend,SystemML compiler groups LOPs into a minimal number of MR jobs
- In Spark backend, we rely on Spark's lazy evaluation and stage construction
- Adaptive matrix block library,sparsity-aware and operates on entire matrix
- Other key features: parallel for-loops,dynamic recompilation

MLContext API

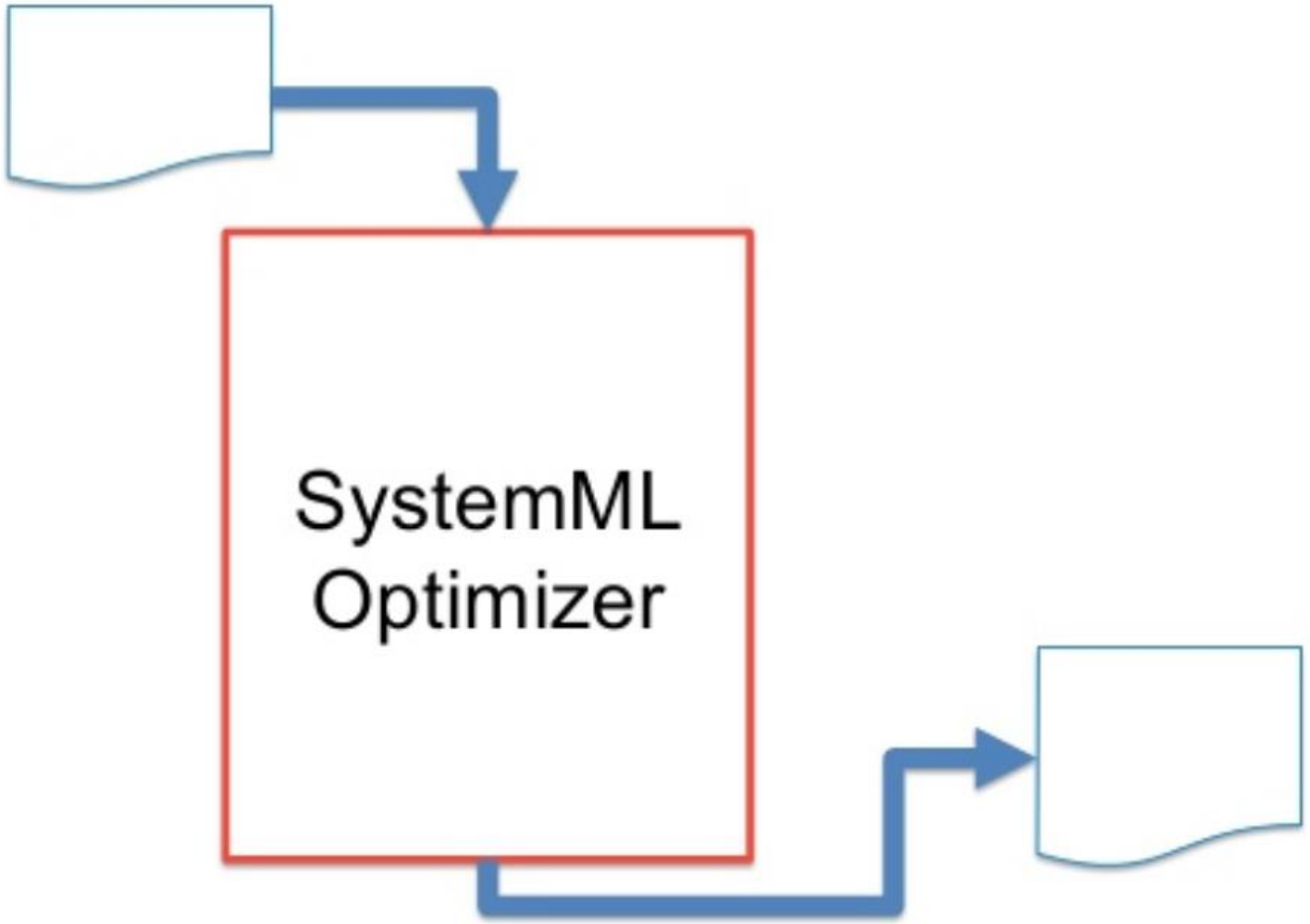
- MLContext works with Scala, Java, and Python, allowing the user to register RDDs and DataFrames as input and output variables of a DML script
- Thus, SystemML can seamlessly integrate into the Spark ecosystem
- SystemML provides many APIs to simplify deployment

Optimizer Integration

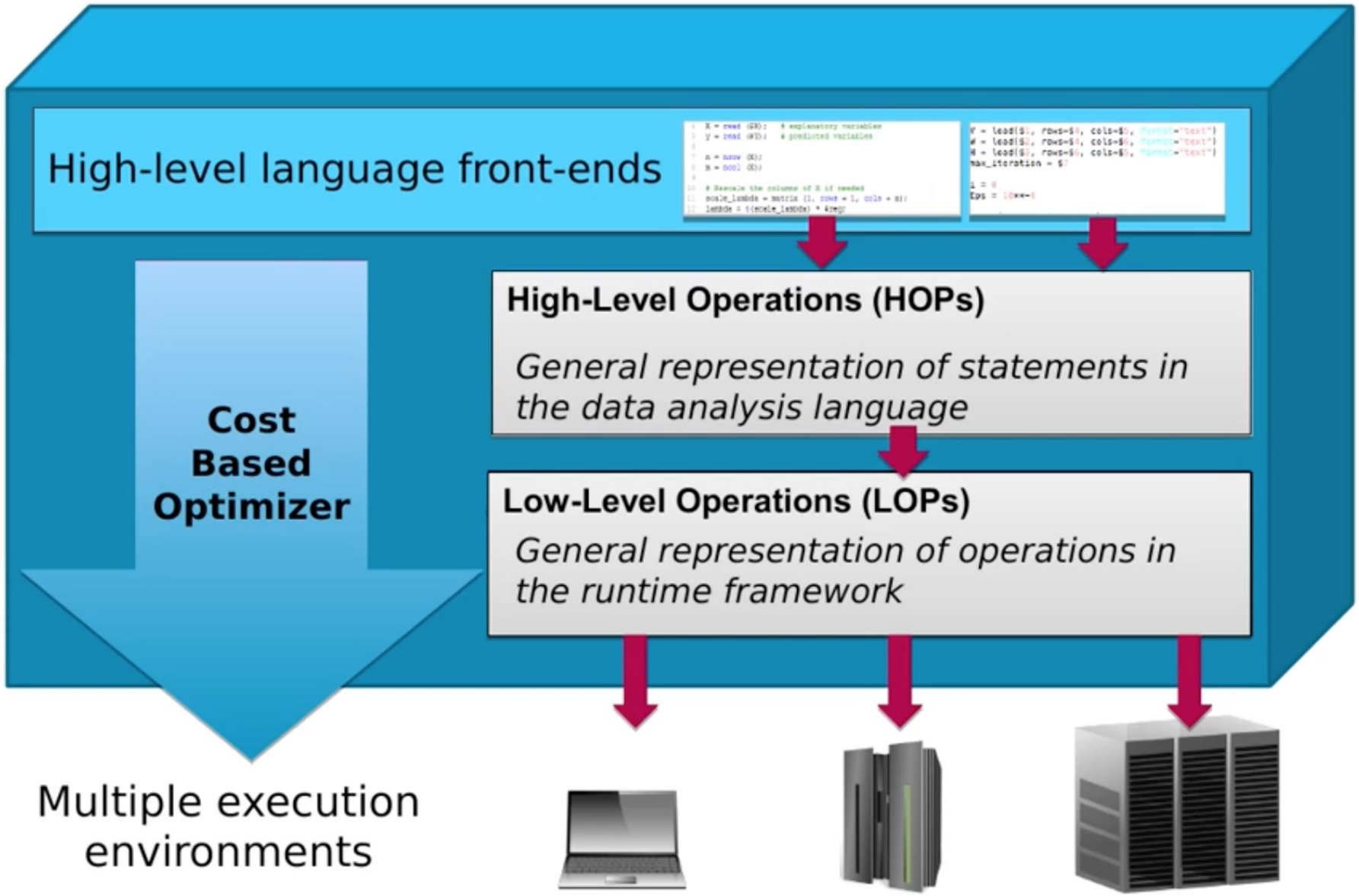
Performance Comparison: ALS



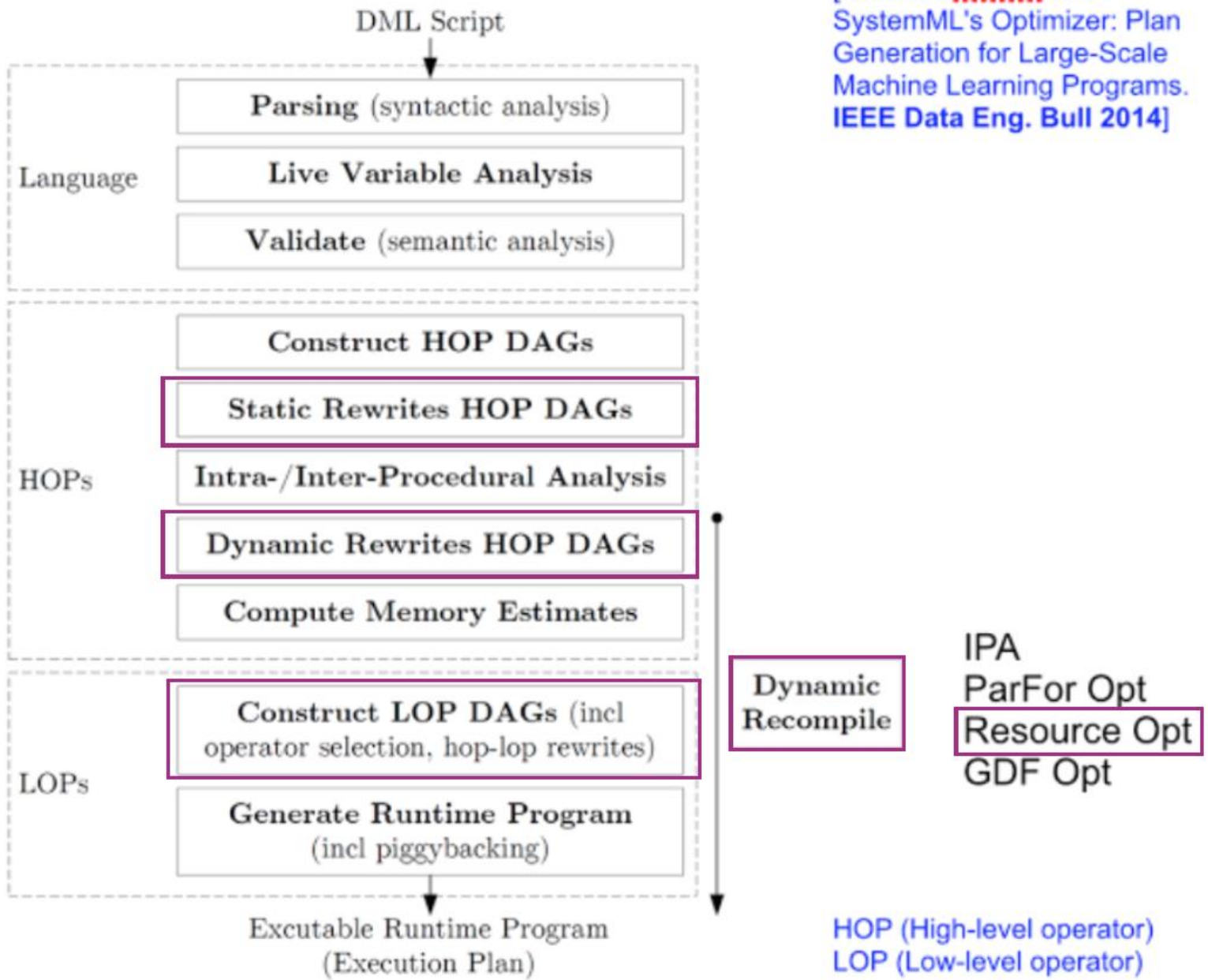
How SystemMl is able to achieve these results?!



What's in the SystemML box



[Matthias Boehm et al: SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. **IEEE Data Eng. Bull 2014**]



HOP (High-level operator)
LOP (Low-level operator)

Basic HOP and LOP DAG Compilation

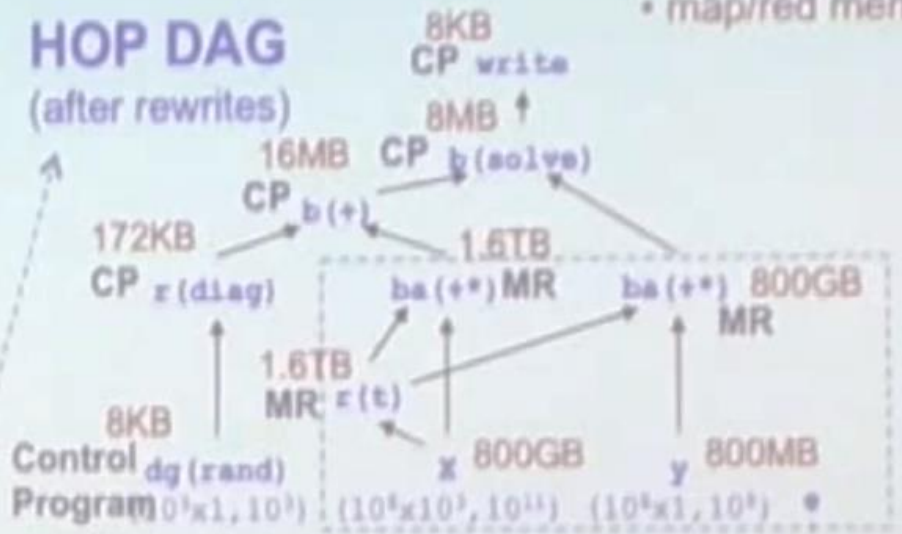
Cluster Config:
- client mem: 4 GB
- map/red mem: 2 GB

Example Linreg DS

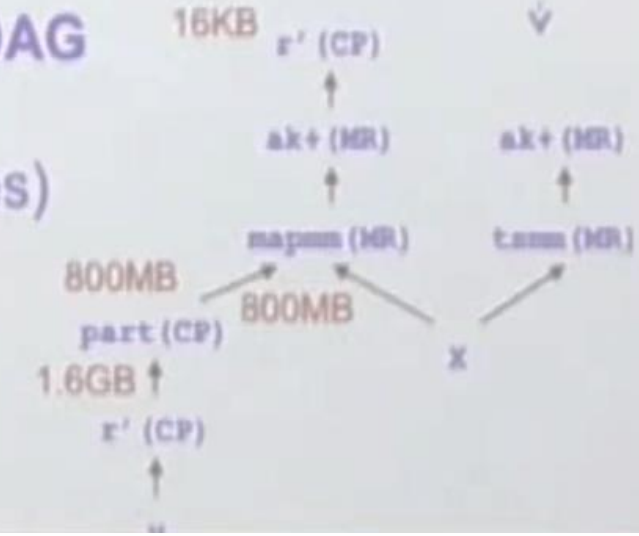
```
X = read($1);
y = read($2);
intercept = $3;
lambda = 0.001;
...
if( intercept == 1 ) {
    ones = matrix(1,nrow(X),1);
    X = append(X, ones);
}
I = matrix(1, ncol(X), 1);
A = t(X) %*% X + diag(I)*lambda;
b = t(X) %*% y;
beta = solve(A, b);
...
write(beta, $4);
```

Scenario:
X: $10^8 \times 10^3$, 10^{11}
y: $10^8 \times 1$, 10^8

HOP DAG (after rewrites)



LOP DAG (after rewrites)



Optimizer Extension

Optimizer extensions at different levels are required to exploit Spark in a robust and effective way:

- Spark-specific Rewrites
- Handling of memory budgets and constraints
- Physical Operator Selection
- Parfor Optimizer Extension

Rewrites

- Types of rewrites
  - Static size-independent rewrites
  - Dynamic size-dependent rewrites

- Examples of Static Rewrites
  - Constant Folding
  - Static Algebraic Simplification Rewrites
  - For-loop vectorization
  - Checkpoint injection (Catching)
  - Repartition Injection
- Examples of Dynamic Rewrites
  - Matrix Multiplication Rewrites
  - Dynamic Algebraic Simplification Rewrites

Selected Algebraic Simplification Rewrites

Name	Static Pattern
Remove Unnecessary Operations	`t(t(X)), X/1, X*1, X-0 → X  matrix(1,)/X → 1/X` `rand(,min=-1,max=1)*7 → rand(,min=-7,max=7)`
Binary to Unary	`X+X → 2*X  X*X → X^2  X-X*Y → X*(1-Y)`
Simplify Diag Aggregates	`trace(X%*%Y)→sum(X*t(Y))`

Name	Pattern
Remove Unnecessary Indexing	`X[a:b,c:d] = Y → X = Y` **iff** `dims(X)=dims(Y)` `X = Y[, 1] → X = Y` **iff** `ncol(Y)=1`
Remove Empty Matrix Multiply	`X%*%Y → matrix(0,nrow(X),ncol(Y))` **iff** `nnz(X)=0|nnz(Y)=0`
Removed Unnecessary Outer Product	`X*(Y%*%matrix(1,...)) → X*Y` **iff** `ncol(Y)=1`
Simplify Diag Aggregates	`sum(diag(X))→trace(X)` **iff** `ncol(X)=1`
Simplify Matrix Mult Diag	`diag(X)%*%Y → X*Y` **iff** `ncol(X)=1&ncol(Y)=1`
Simplify Diag Matrix Mult	`diag(X%*%Y) → rowSums(X*t(Y))` **iff** `ncol(Y)>1`
Simplify Dot Product Sum	`sum(X^2) → t(X)%*%X` **iff** `ncol(X)=1`

Checkpoint Injection

- Spark allows an RDD to be persisted into various storage levels for distributed caching
- Inject checkpoints after every persistent read or reblock to prevent repeated read from HDFS
- Inject checkpoints before loops for all read-only variables in the loop body
- Cleanup: remove unnecessary checkpoints

Repartition Injection

- **Join** or **reduceByKey** operations that cause shuffle are very expensive operations on Spark because shuffle significantly dominates execution time compared to reads from distributed cache.

- Spark avoids unnecessary shuffle if, for example, the inputs to a **join** are partitioned with the same partitioning function because it guarantees co-partitioning but not necessarily co- location.

- This optimization feature in Spark is exploited with partitioning-preserving operations like **zipmm** that avoids key changes to retain co-partitioning.

SystemML Memory Budget

MD: spark memory drive, **ME**: executor memory,

|E|: number of executors, **Δ:**data fraction, **Σ:**shuffle fraction

M_CP = α*MD(default α=0.7)

M_ B = β*Σ*ME(default β=0.3)

total data memory = |E|*Δ* ME

Dynamic memory management—introduced in Spark 1.6—increases our memory budgets by removing **Σ** and thus, increasing **Δ.**

Objective is to minimize total program execution time subject to memory constraints over all operations and execution context

Operator Selection

- Example Fused Operator: WSLoss
  - Exploit sparsity via selective computation

$$sum(\mathbf{W} \odot (\mathbf{U}\,\mathbf{V} - \mathbf{X})^2).$$

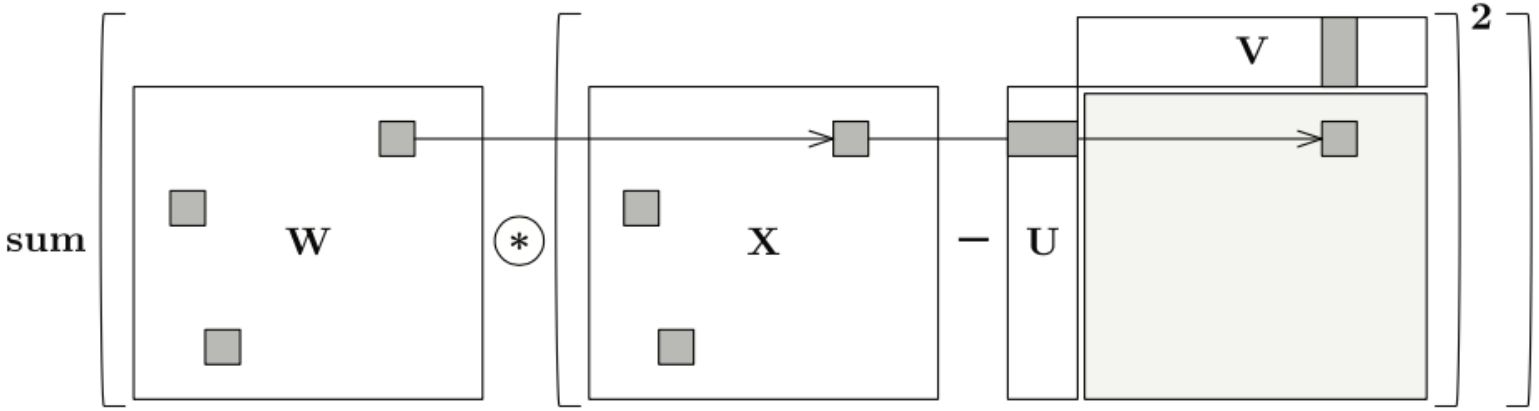


Figure 1: Example Weighted Squared Loss.

Parallel For Loops

- SystemML supports task-parallel computation via parfor loops
- Examples:descriptive statistics, ensemble learning, KMeans, Multiclass SVM
- **Local parfor** executes multi-threaded workers in the drive, allowing multi-thread execution and concurrent Spark jobs
- **Remote parfor** executes the entire loops as a single map-side Spark job
- **Remote_dp parfor** partitions a given input matrix into disjoint slices and executes the parfor body per slice

Runtime Integration

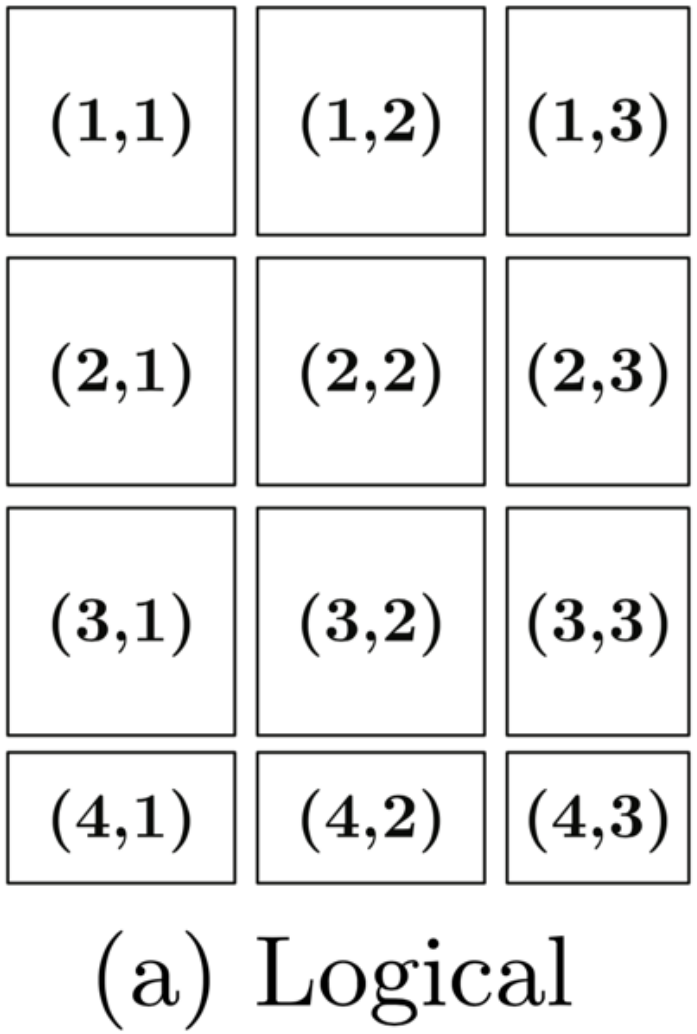
- Overview SystemML's backend and runtime plans:
- Distributed matrix representations
  - How to represent and operate on distributed matrices across nodes?
- Runtime Plans via Buffer Pool Integration
  - How to handle hybrid runtime plans (operations involving both single node and distributed system)
- Dynamic Recompilation
  - Adapt runtimes plan to changing or initially unknown data characteristics
- Other optimizations

Binary Block Matrices

- Distributed matrices are partitioned into fixed size blocks
- Allows for matrix operations in a distributed manner
  - e.g. transpose operation can be done separately for each sub-matrix rather than on the whole matrix
- Keeps partitioned matrices at fixed sizes
- Fixed size leads to variable physical sizes, but simplifies join processing for binary operations

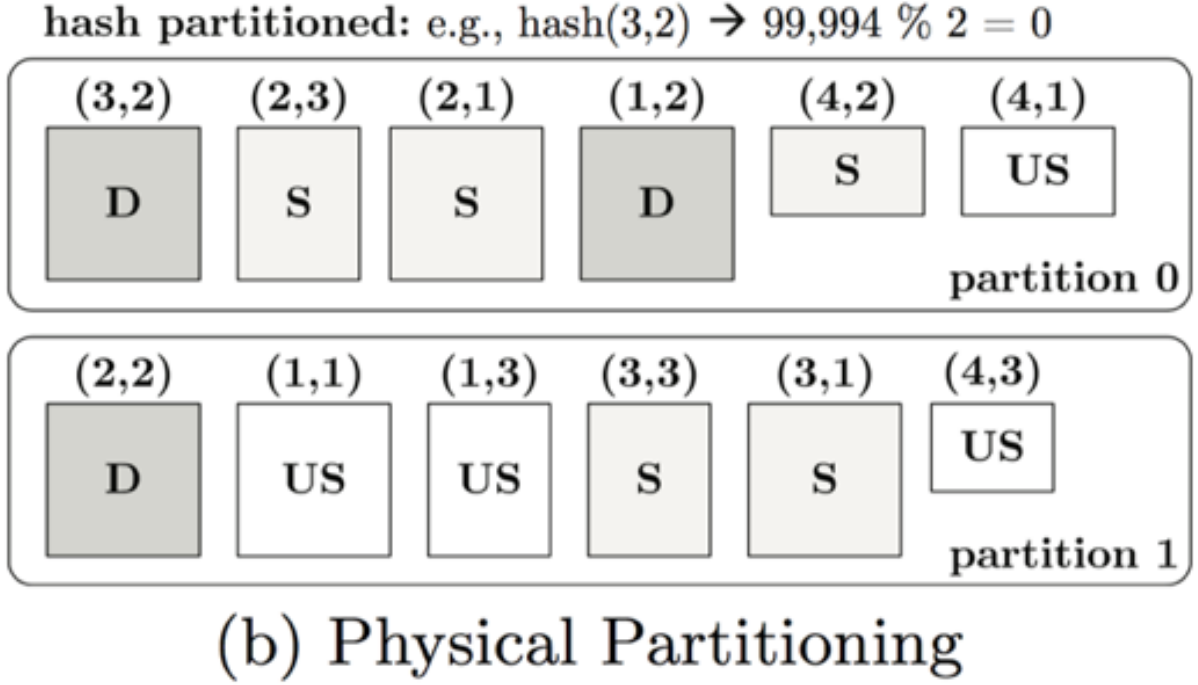
Binary Block Matrices

- Represented as pairs of row/column block indexes
- Figure 2(a): Distributed matrix,logical view



Binary Block Matrices

- Fixed size leads to variable physical sizes, but simplifies join processing for binary operations
- Figure 2(b): Physical RDD partitions with hash partitioning



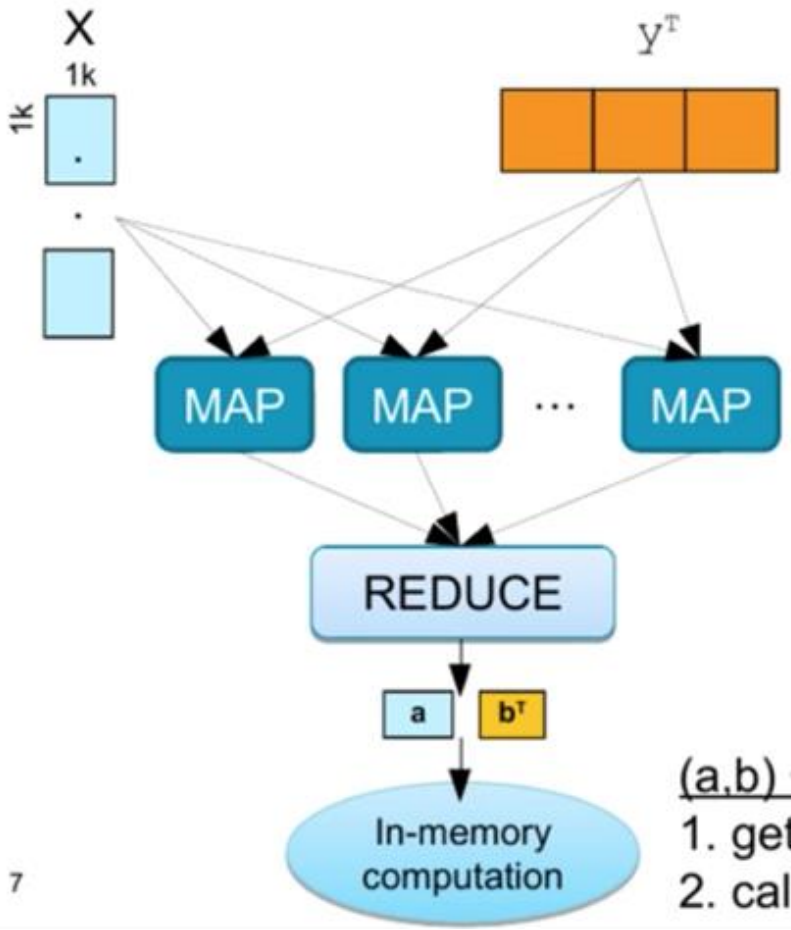
(b) Physical Partitioning

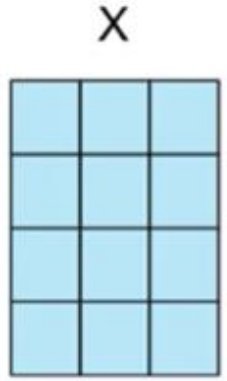
Blocks

- Square blocks simplifies operations, such as a transpose by flipping block indexes per block, avoiding the need for data shuffle across blocks
- Shuffle is a major bottleneck for SystemML
- Aim to reduce shuffling overhead by minimizing the serialized size of shuffled RDDs and avoiding shuffle via co-partitioning
- Use square blocks of size Bc = 1K
  - Benefits:
  - (1) Small dimensions lead to a maximum size of 8 MB for dense blocks; leads to cache friendly behavior, especially for long chains of operations
  - (2) Square blocks simplifies some operations such as transpose

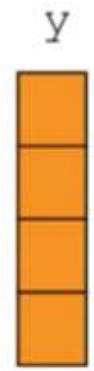
Linear Regression - Execution

```
a = t(X) %*% X + diag(lambda);
b = t(X) %*% y;
theta = solve(a,b);
```



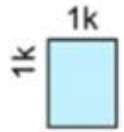


500 features
300M observations
4TB text file



300M observations
9GB text file

X^TX for each
y^TX for each



(a,b) < 2 MB
1. get b
2. call `solve(a,b)`

Cluster Configuration
3.5 GB Map Task JVM
7 GB In-memory Master JVM
128 MB HDFS block size

Buffer Pool Integration

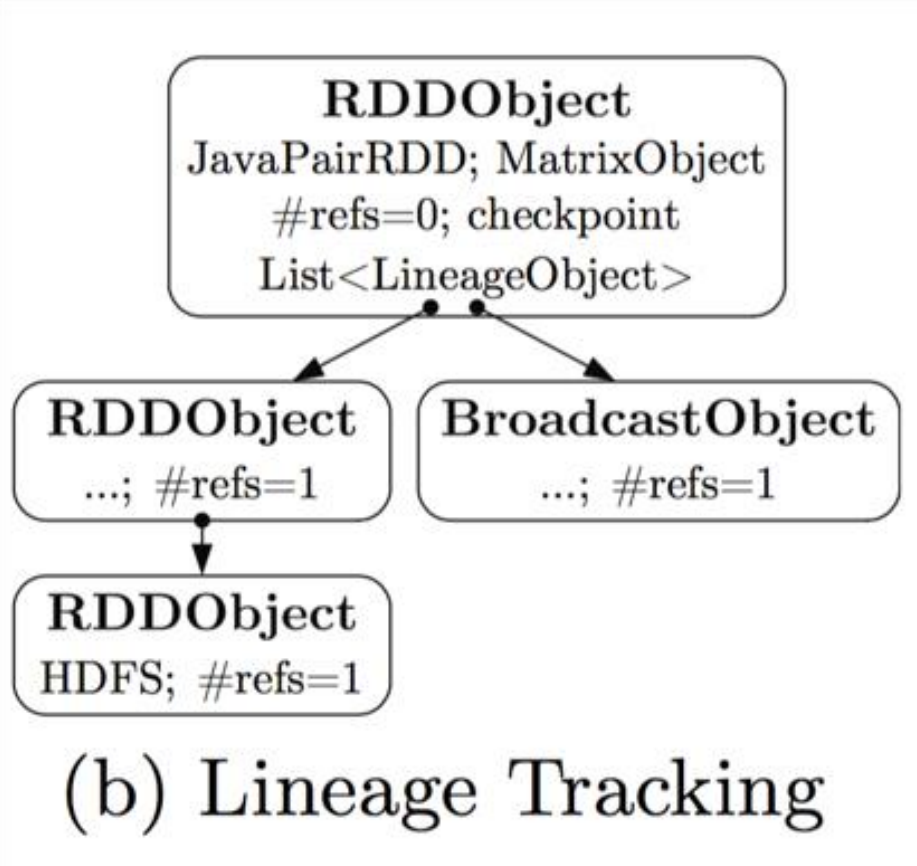
- Purpose of buffer pools is to cache table and index data
- Makes management of live matrix objects faster and organized
- Compiled blocks are executed by making use of the symbol table of currently live matrix objects and scalars which is passed along
- Matrix objects handles to in-memory or distributed matrices and interact with the buffer pool

---

- Hybrid runtime plans involve both single node use and distributed system use
- Hybrid runtime plans can lead to unique challenges
- BPI can overcome challenges related to hybrid runtime plans:
  - Challenges:
  - (1) Exchanging intermediates between runtime backends
  - (2) Robustness with regard to lazy evaluation

Lineage Tracking

- When cleanup of variables occurs, lazy RDD evaluation can lead to the problem where there may be some RDDs that will be refenced in pending/future RDD operations operations
- To address this, during runtime, build a lineage graph of RDD and broadcast objects and their data dependencies
- Every outputted RDD carries the lineage to input RDDs and broadcasts
- Ensures robustness with lazily evaluated RDDs and provides access to all data-dependent RDDs



(b) Lineage Tracking

Dynamic Recompilation

- The goal is to adapt the runtime plan to changing or initially unknown data characteristics
- At recompilation time, sizes of intermediates may be unknown due to lazy evaluation
- Linear algebra operations (element-wise computations, transpose, etc.) allow for inferring the exact output dimensions
- Operations like transpose or sort, even preserve all input characteristics
- Exploit this and other techniques

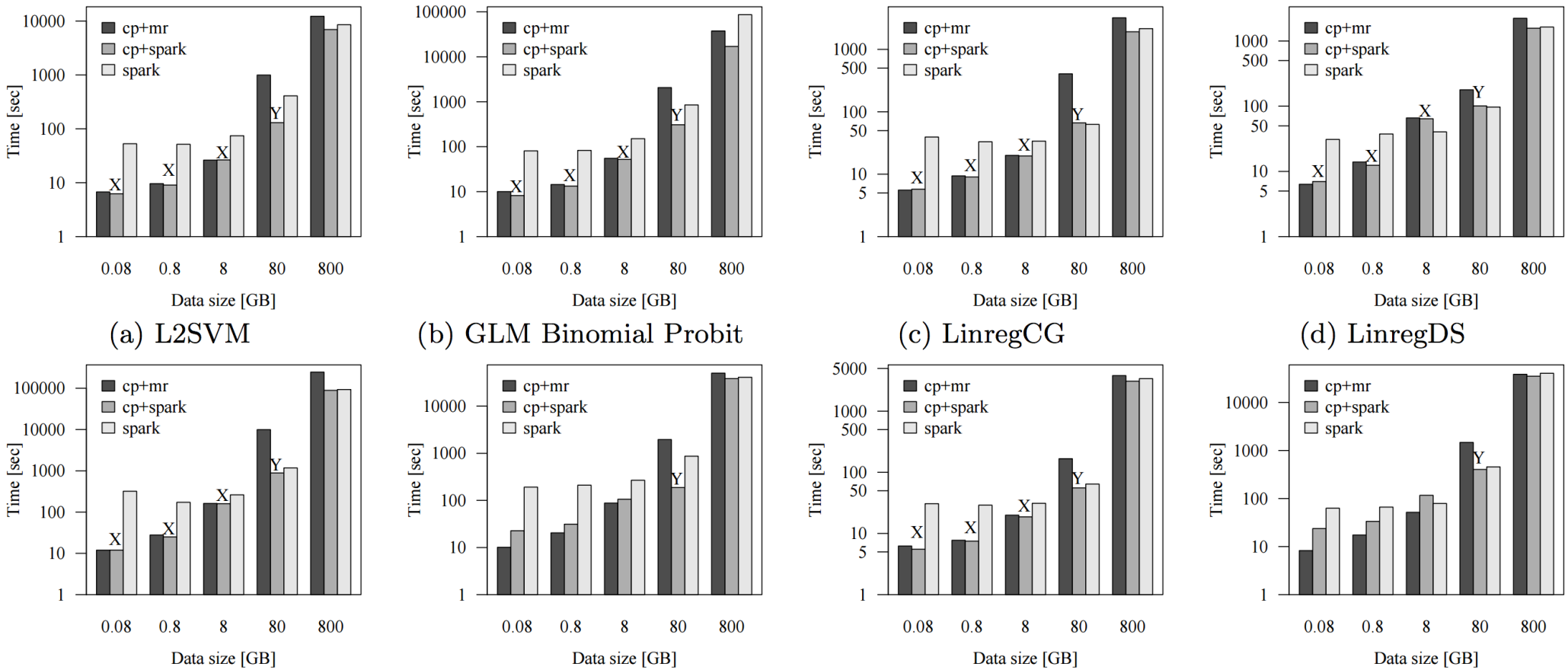
Specific Runtime Optimizations

- Lazy Spark-Context Creation:
- Spark context creation may lead to unnecessary overhead when all operations are single-node
- On creation of the Spark context, executors are allocated and initialized, which can take up to 20 s
- Lazily allocate the context on demand when it is first accessed (e.g., when creating an RDD)
- Avoids context creation for pure single-node computation

Experiments

Table 3: Characteristics of Used ML Algorithms.

Algorithm	Maxi	ϵ	λ	Icpt	#C	ParFor
L2SVM	$20/\infty$	1e-6	1e-2	N	2	N
GLM	$20/\infty$	1e-6	1e-2	N	2	N
LinregCG	20	1e-6	1e-2	N	N/A	N
LinregDS	N/A	N/A	1e-2	N	N/A	N
MLogreg	$20/\infty$	1e-6	1e-2	N	5	N
MSVM	$5 \times 20/\infty$	1e-6	1e-2	N	5	Y
Naïve Bayes	N/A	N/A	N/A	N	5	Y
KMeans	10×20	1e-4	N/A	N	10	Y
ALS	6/50*	0	1	N/A	50	N



(a) L2SVM (b) GLM Binomial Probit (c) LinregCG (d) LinregDS

(e) MLogreg (f) MSVM (g) Naïve Bayes (h) KMeans

Figure 4: End-to-End Performance of Different Algorithms with Different Execution Modes.

Comparison

- In all cases except LinregDS and Kmeans, cp+spark outperforms spark on average by 6.1x for small datasets and 1.9x for larger datasets
- When many operations over small input data, in-memory single-node computation is faster than distributed
- With LinregDS, cp+spark compiles an in-memory transpose while spark is distributed. With 8GB data size, 63 HDFS partitions and thus higher parallelism for distributed computation,vs single-node with 16 threads.

Comparison

- Even in large datasets, cp+spark is 5-10x faster than cp+mr
- Due to: automatic checkpoint injection by SystemML to avoid repeated reads, and smaller Spark latency vs MapReduce
- On the largest 800GB datasets,cp+spark is 1.1-2.8x faster than cp+mr
- Here, data does not fit into aggregated memory, limiting the time improvements due to spark RDDs
- Improvement due to faster task scheduling with Spark

ALS Experiment

Table 6: ALS End-to-End Performance.

Scenario	cp+mr	cp+spark	spark
S ($10^5 \times 10^5$, 0.01, 1.2 GB)	**131** s	136 s	135 s
M ($10^6 \times 10^5$, 0.01, 12 GB)	1,088 s	**342** s	432 s
L ($10^7 \times 10^5$, 0.01, 120 GB)	>24 h	**10,537** s	15,487 s

- Running ALS on cp+spark is clearly beneficial across all datasets
- For M and L scenarios, cp+spark generates hybrid execution plans including CP and Spark intructions and outperforms the other execution modes
- For dataset L, user-factors did not fit into the memory of map/reduce tasks, thus replication and shuffling led to repeated spilling

Any Questions?