Collaborative Filtering with Spark

Chris Johnson
@MrChrisJohnson

Spotify

May 9, 2014

Who am I??

- Chris Johnson
  - Machine Learning guy from NYC
  - Focused on music recommendations
  - Formerly a graduate student at UT Austin

What is MLlib?

Algorithms:

- **classification:** logistic regression, linear support vector machine (SVM), naive bayes
- **regression:** generalized linear regression
- **clustering:** k-means
- **decomposition:** singular value decomposition (SVD), principle component analysis (PCA
- **collaborative filtering:** alternating least squares (ALS)

http://spark.apache.org/docs/0.9.0/mllib-guide.html

What is MLlib?

Algorithms:

- **classification:** logistic regression, linear support vector machine (SVM), naive bayes
- **regression:** generalized linear regression
- **clustering:** k-means
- **decomposition:** singular value decomposition (SVD), principle component analysis (PCA
- **collaborative filtering:** alternating least squares (ALS)

http://spark.apache.org/docs/0.9.0/mllib-guide.html

Collaborative Filtering – "The Netflix Prize"

Collaborative Filtering

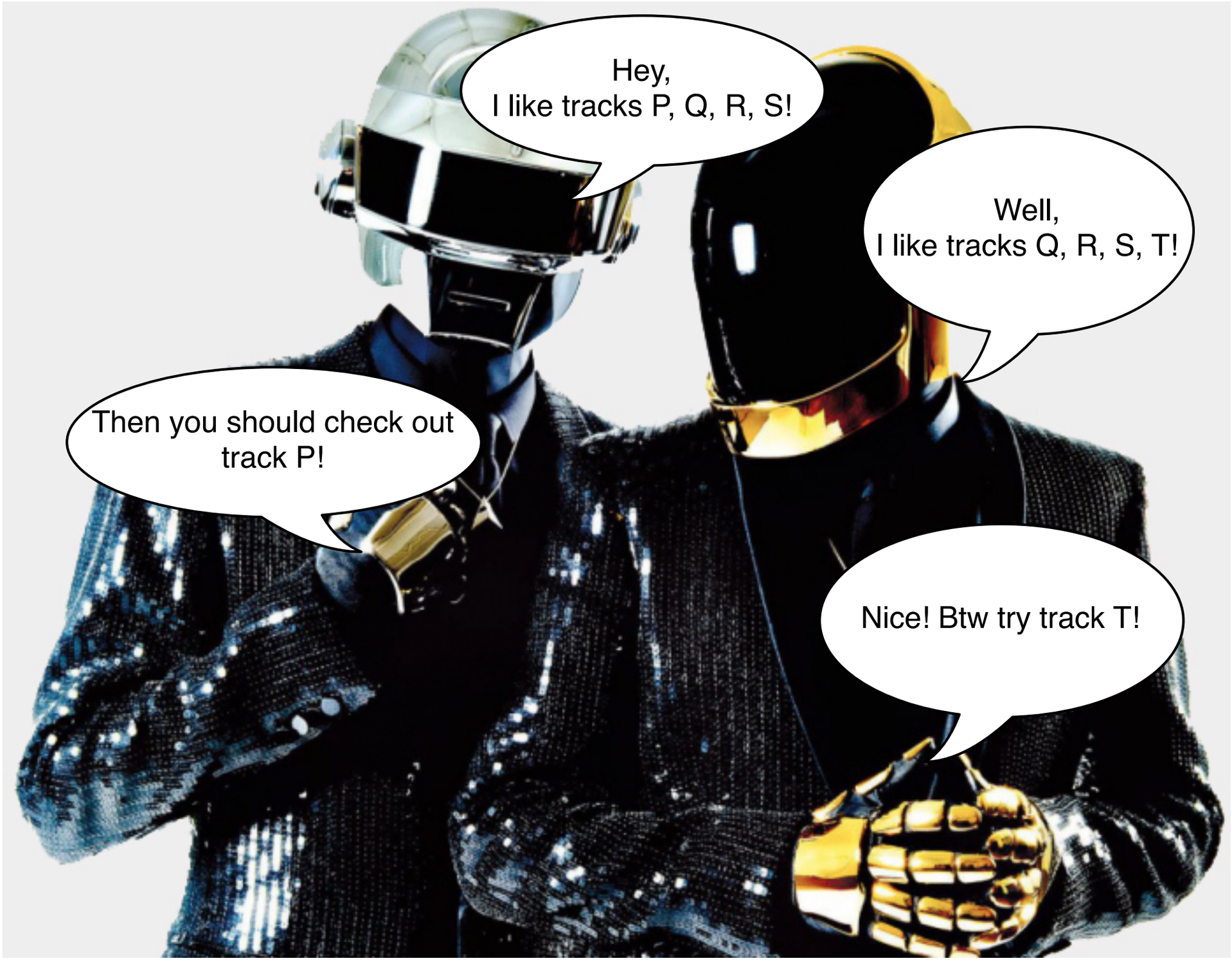
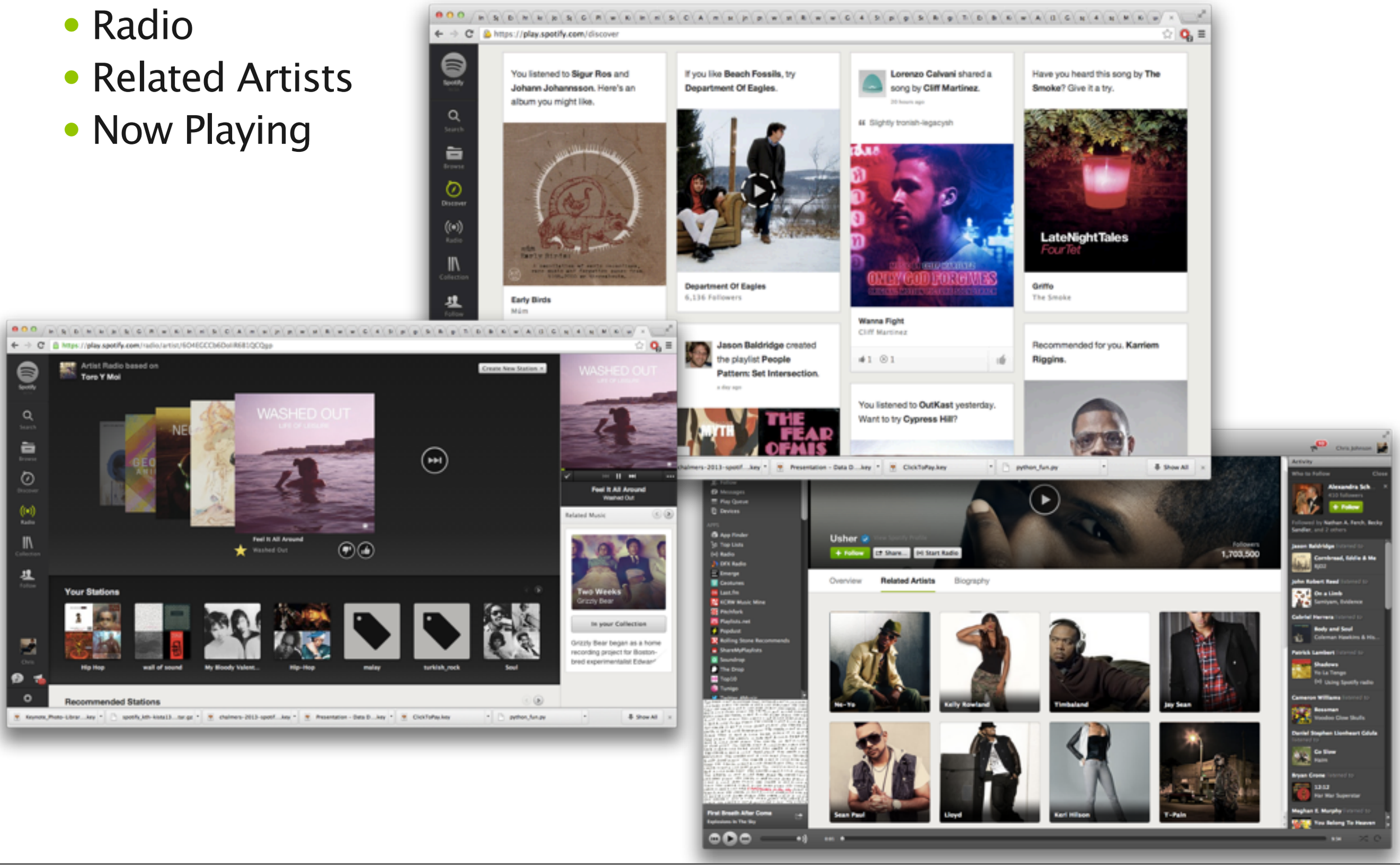


Image via Erik Bernhardsson

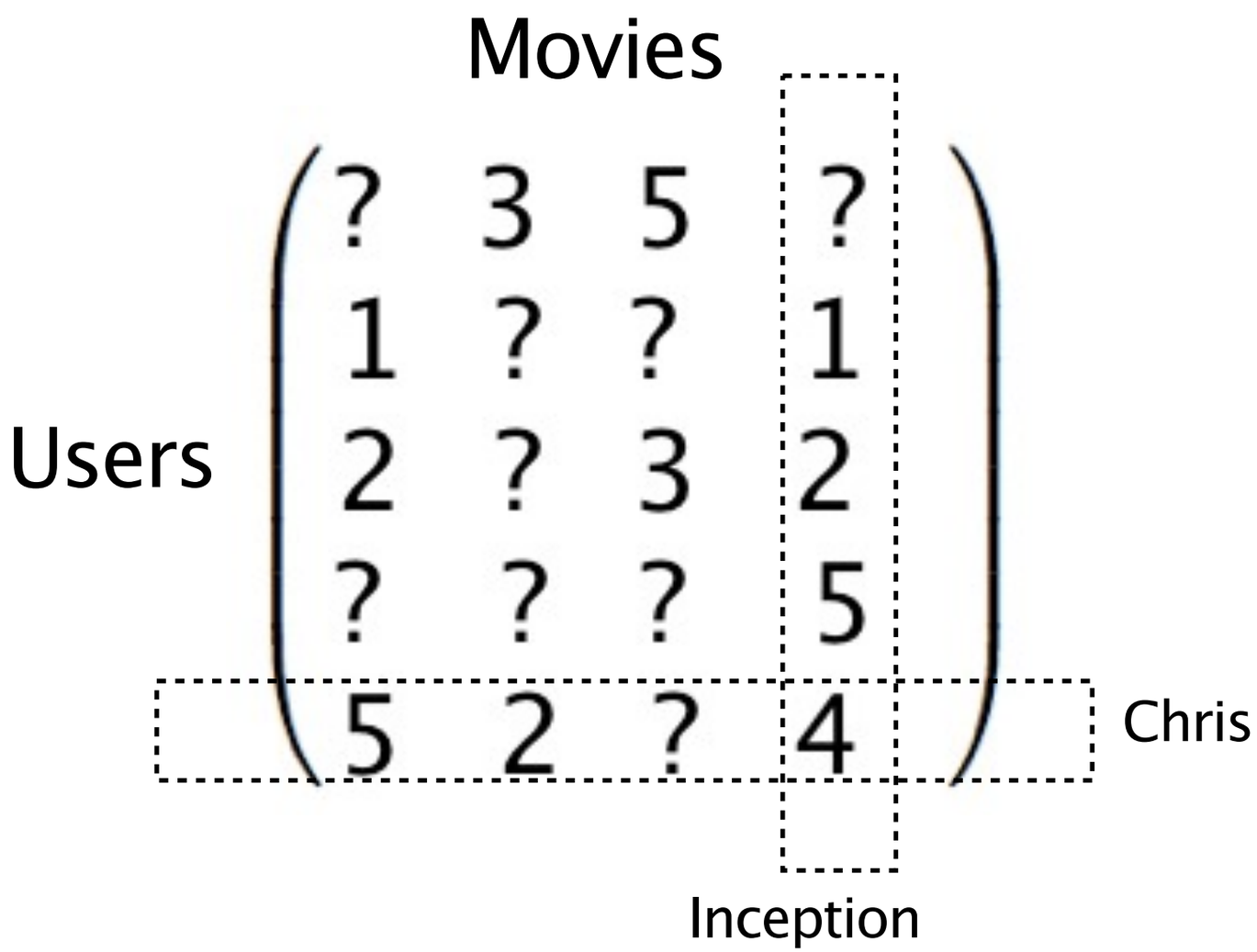
Collaborative Filtering at Spotify

- Discover (personalized recommendations)
- Radio
- Related Artists
- Now Playing



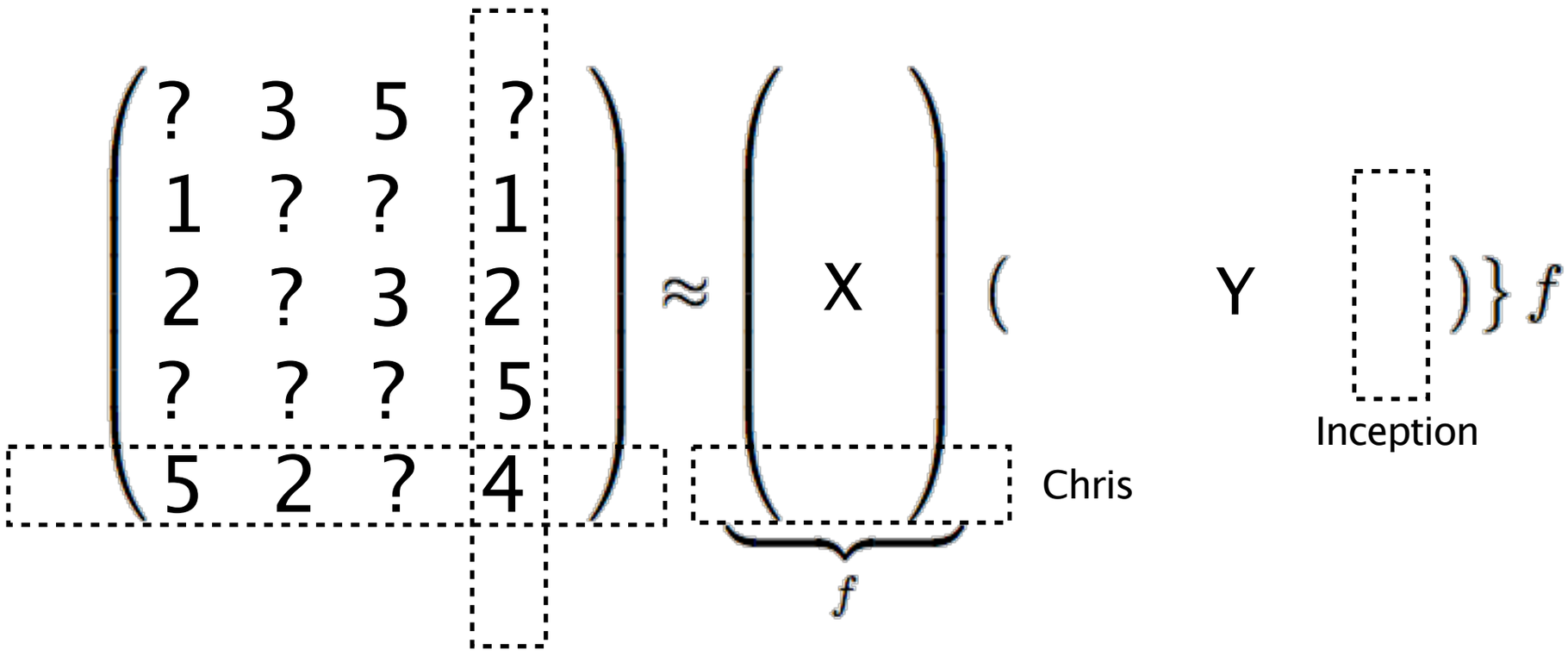
Explicit Matrix Factorization

- Users explicitly rate a subset of the movie catalog
- Goal: predict how users will rate new movies



Explicit Matrix Factorization

- Approximate ratings matrix by the product of low-dimensional user and movie matrices
- Minimize RMSE (root mean squared error)

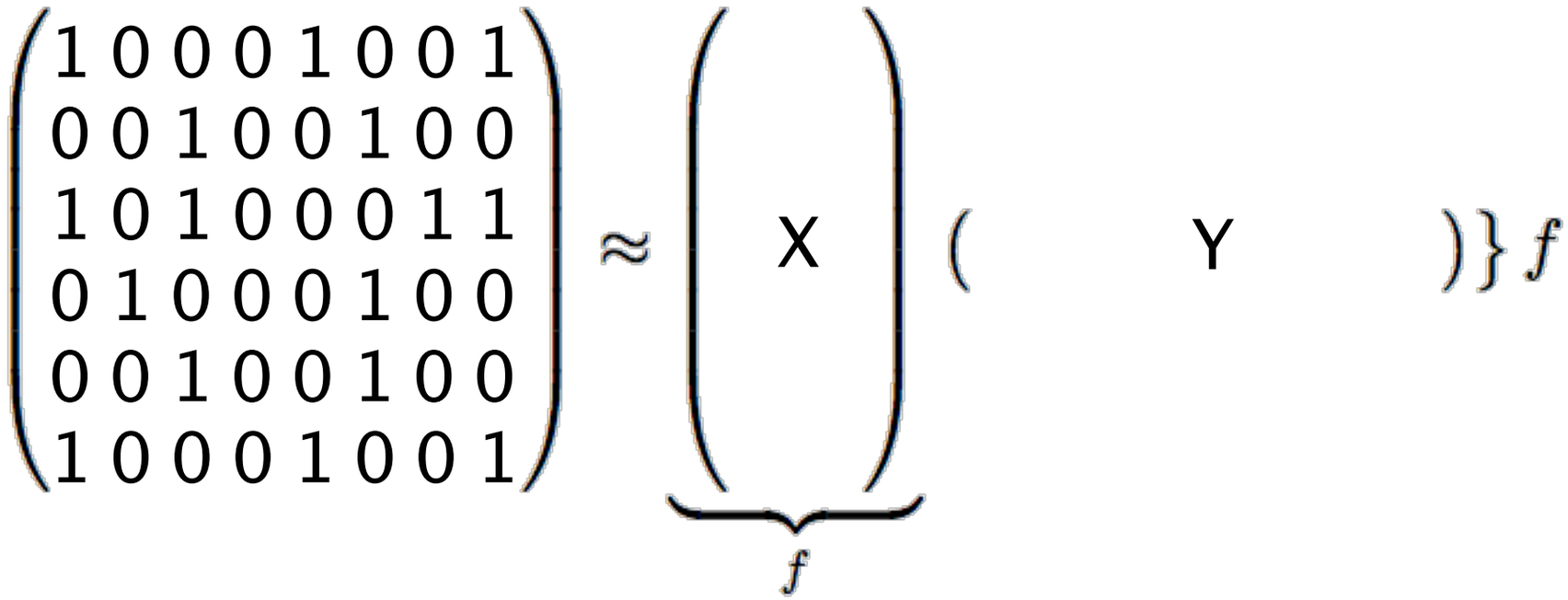


$$min_{x,y} \sum_{u,i} (r_{ui} - x_u^T y_i - b_u - b_i)^2 - \lambda \left(\sum_u ||x_u||^2 + \sum_i ||y_i||^2 \right)$$

- r_{ui} = user $u's$ rating for movie i
- x_u = user $u's$ latent factor vector
- x_i = item $i's$ latent factor vector
- b_u = bias for user u
- b_i = bias for item i
- λ = regularization parameter

Implicit Matrix Factorization

- Replace Stream counts with binary labels
  - 1 = streamed, 0 = never streamed
- Minimize weighted RMSE (root mean squared error) using a function of stream counts as weights



$$min_{x,y} \sum_{u,i} c_{ui}(p_{ui} - x_u^T y_i - b_u - b_i)^2 - \lambda \left(\sum_u ||x_u||^2 + \sum_i ||y_i||^2 \right)$$

- p_{ui} = 1 if user u streamed track i else 0
- $c_{ui} = 1 + \alpha r_{ui}$
- x_u = user $u's$ latent factor vector
- x_i =i tem $i's$ latent factor vector
- b_u = bias for user u
- b_i = bias for item i
- λ = regularization parameter

Alternating Least Squares

- Initialize user and item vectors to random noise

```
1  user_vectors = numpy.random.normal(size=(num_users,
2                                           num_factors))
3  item_vectors = numpy.random.normal(size=(num_items,
4                                           num_factors))
```

- Fix item vectors and solve for optimal user vectors
  - Take the derivative of loss function with respect to user's vector, set equal to 0, and solve
  - Results in a system of linear equations with closed form solution!

$$x_u = (Y^T C^u Y + \lambda I)^{-1} Y^T C^u p(u)$$

- Fix user vectors and solve for optimal item vectors
- Repeat until convergence

code: https://github.com/MrChrisJohnson/implicitMF

Alternating Least Squares

- Note that:

$$Y^T C^u Y = Y^T Y + Y^T (C^u - I) Y$$

- Then, we can pre-compute $Y^T Y$ once per iteration
  - $C^u - I$ and $C^u p(u)$ only contain non-zero elements for tracks that the user streamed
  - Using sparse matrix operations we can then compute each user's vector efficiently in $O(f^2 n_u + f^3)$ time where n_u is the number of tracks the user streamed

$$x_u = (Y^T Y + Y^T (C^u - I) Y)^{-1} Y^T C^u p(u)$$

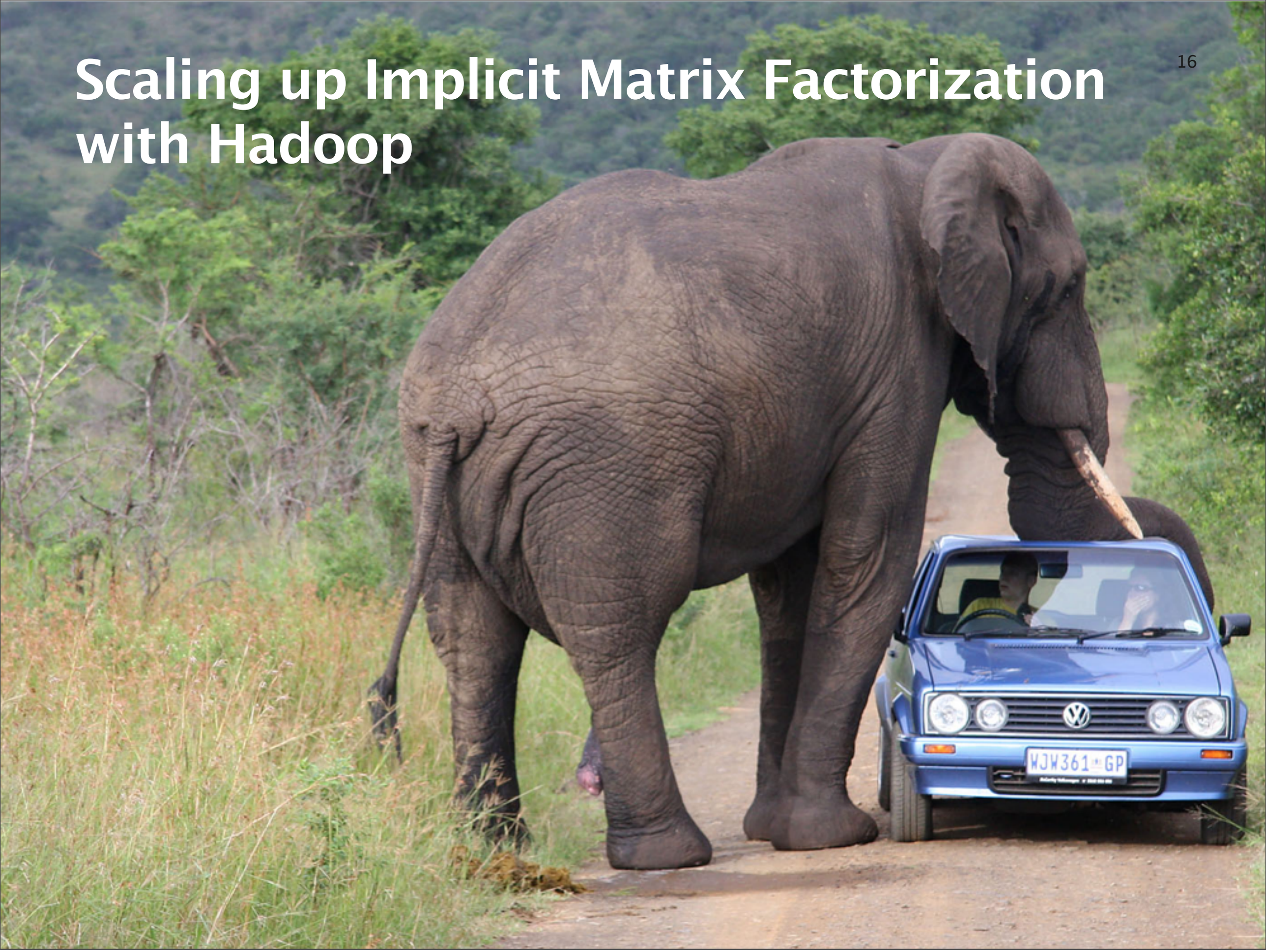
code: https://github.com/MrChrisJohnson/implicitMF

Alternating Least Squares


```python
def als_iteration(user, counts, solve_vecs, fixed_vecs, num_factors=40, reg_param=0.8):
    '''
        @param user:        True if solving for user vectors
        @param counts:      scipy.sparse matrix containing implicit
                                user-item counts * alpha
        @param solve_vecs:  scipy.sparse vector of latent factors you
                                wish to solve for
        @param fixed_vecs:  scipy.sparse vector of fixed latent factors
        @param reg_param:   regularization parameter (lambda)

    '''
    num_fixed = fixed_vecs.shape[0]
    YTY = fixed_vecs.T.dot(fixed_vecs)
    eye = scipy.sparse.eye(num_fixed)
    lambda_eye = reg_param * scipy.sparse.eye(num_factors)

    for i in xrange(solve_vecs.shape[0]):
        if user:
            counts_i = counts[i].toarray()
        else:
            counts_i = counts[:, i].T.toarray()
        CuI = scipy.sparse.diags(counts_i, [0])
        pu = counts_i.copy()
        pu[numpy.where(pu != 0)] = 1.0
        YTCuIY = fixed_vecs.T.dot(CuI).dot(fixed_vecs)
        YTCupu = fixed_vecs.T.dot(CuI + eye).dot(scipy.sparse.csr_matrix(pu).T)
        xu = scipy.sparse.linalg.spsolve(YTY + YTCuIY + lambda_eye, YTCupu)
        solve_vecs[i] = xu

    return solve_vecs
```

code: https://github.com/MrChrisJohnson/implicitMF

Scaling up Implicit Matrix Factorization with Hadoop



Hadoop at Spotify 2009

Hadoop at Spotify 2014

700 Nodes in our London data center

Implicit Matrix Factorization with Hadoop

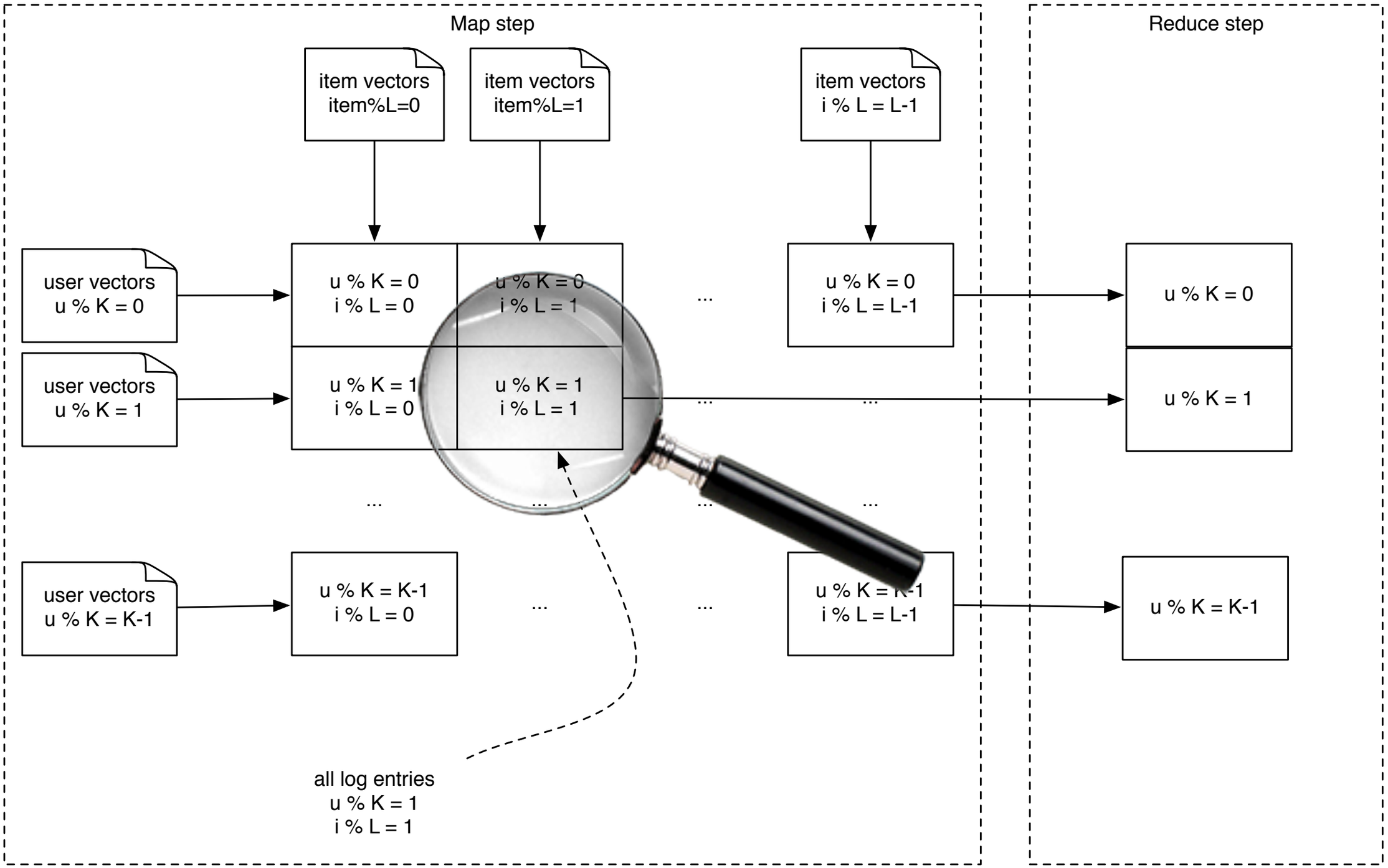


Figure via Erik Bernhardsson

Implicit Matrix Factorization with Hadoop

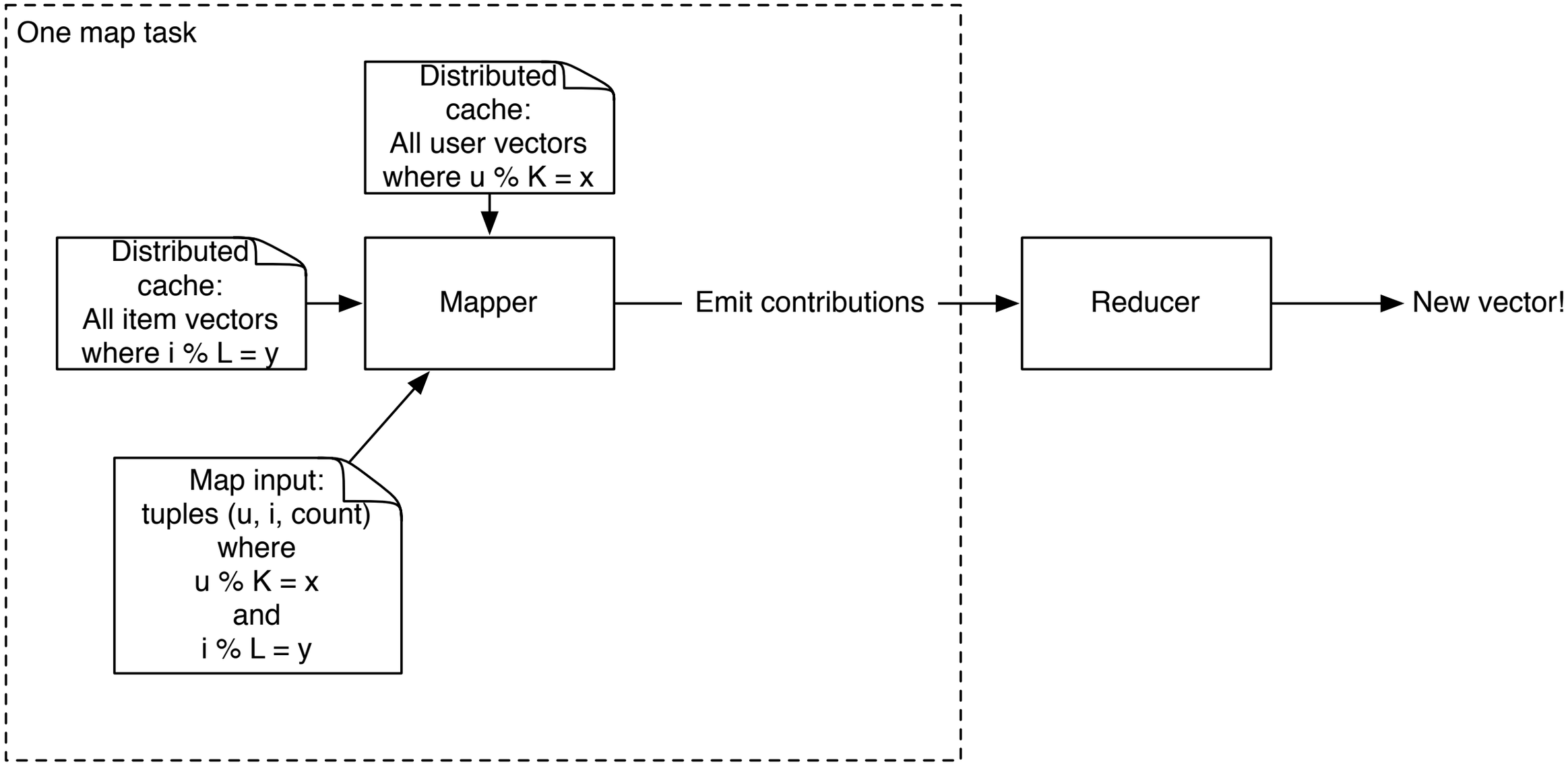
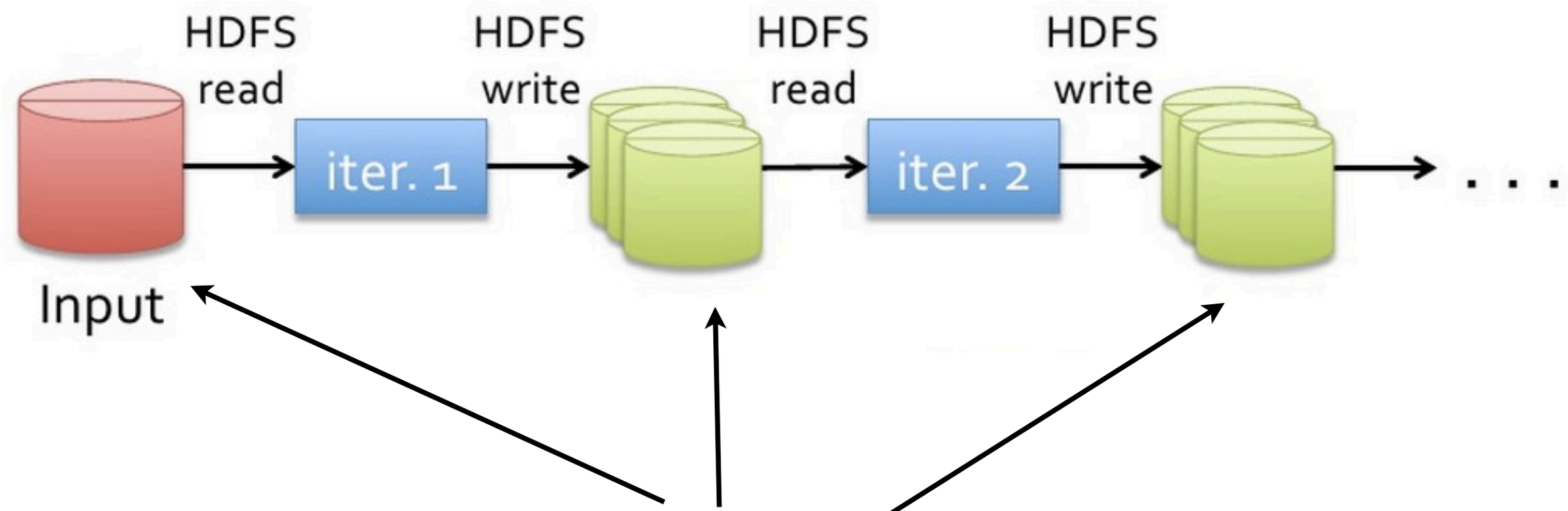
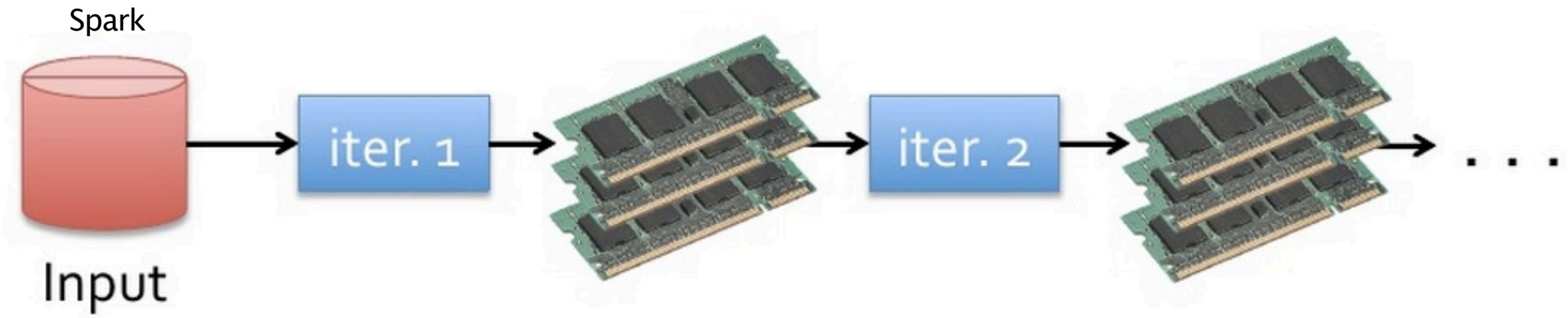


Figure via Erik Bernhardsson

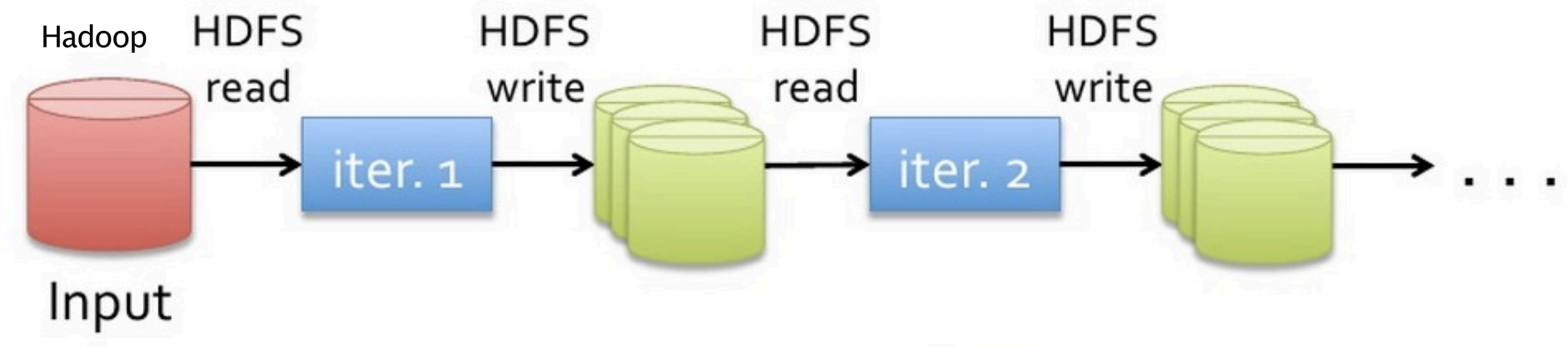
Hadoop suffers from I/O overhead



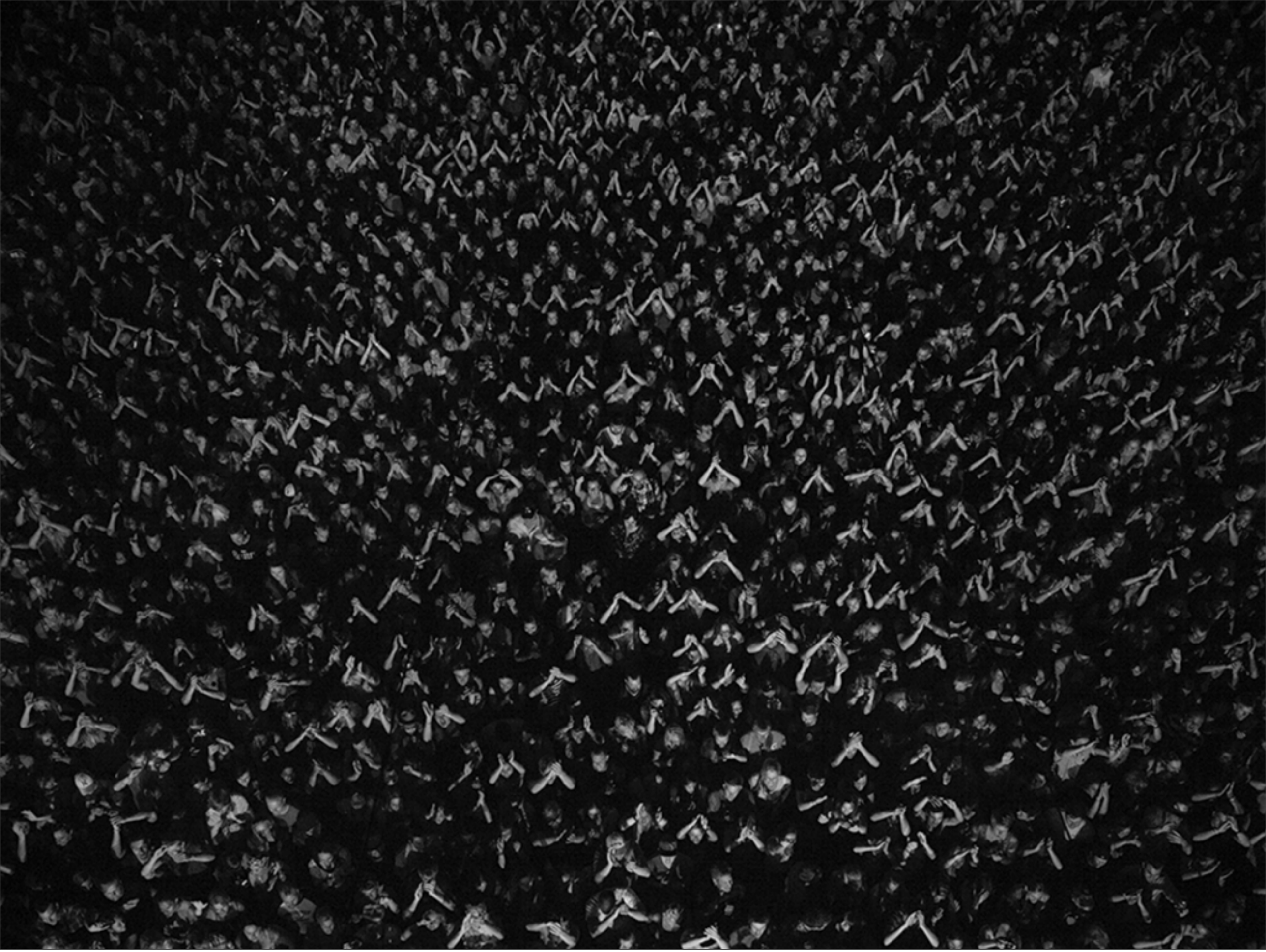
Spark to the rescue!!



Vs

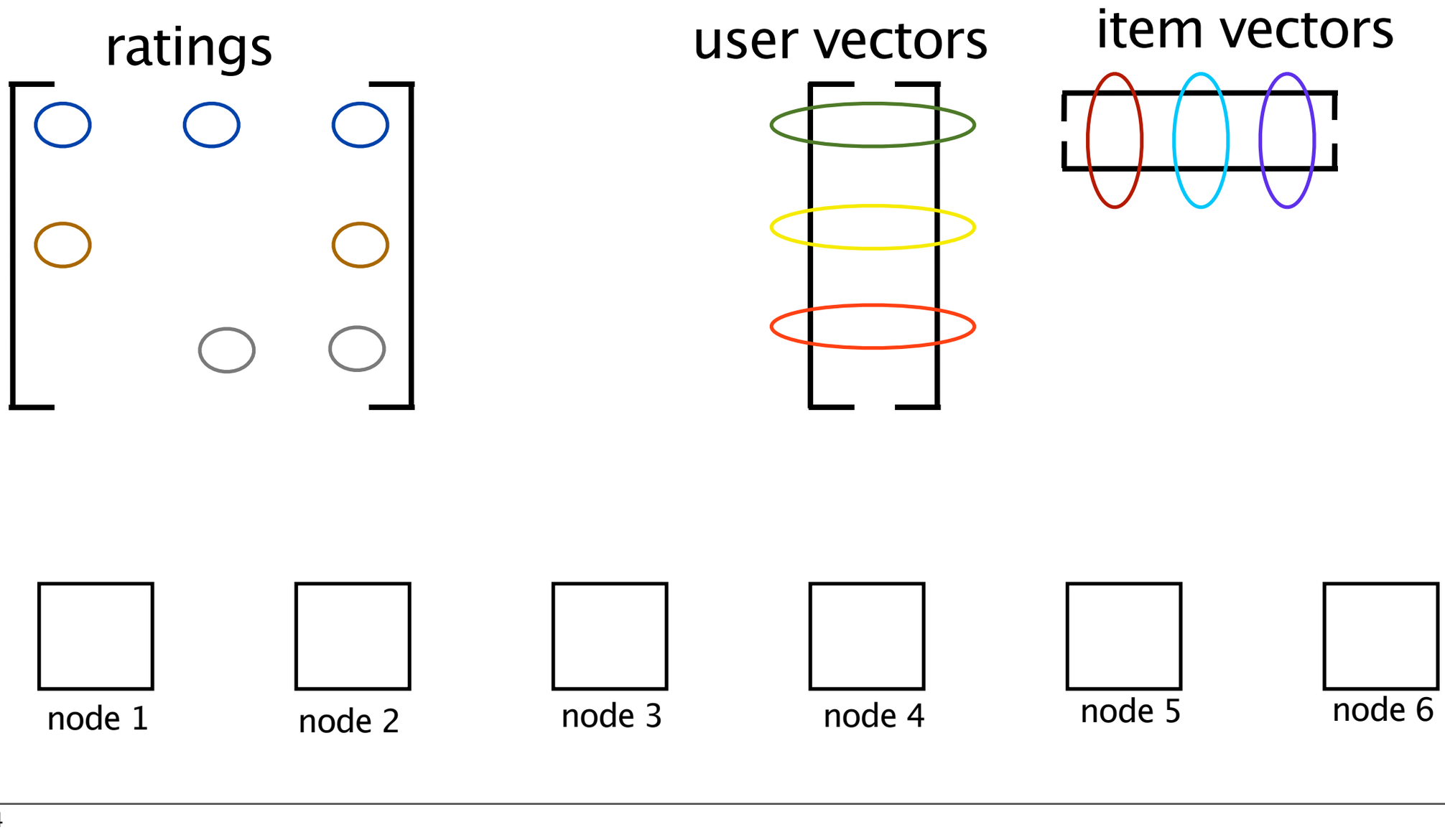


http://www.slideshare.net/Hadoop_Summit/spark-and-shark



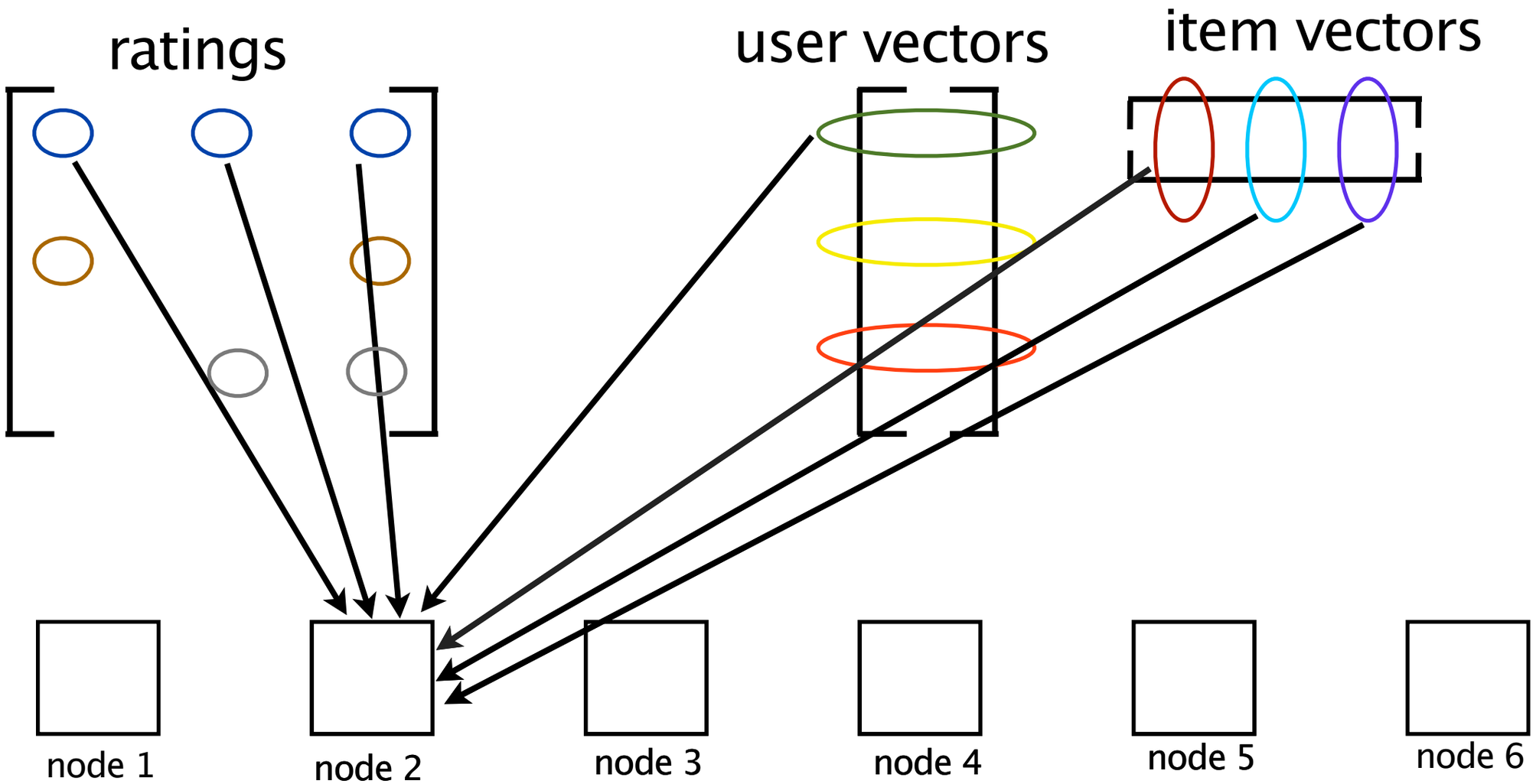
First Attempt

- For each iteration:
  - Compute YtY over item vectors and broadcast
  - Join user vectors along with all ratings for that user and all item vectors for which the user rated the item
  - Sum up YtCuIY and YtCuPu and solve for optimal user vectors



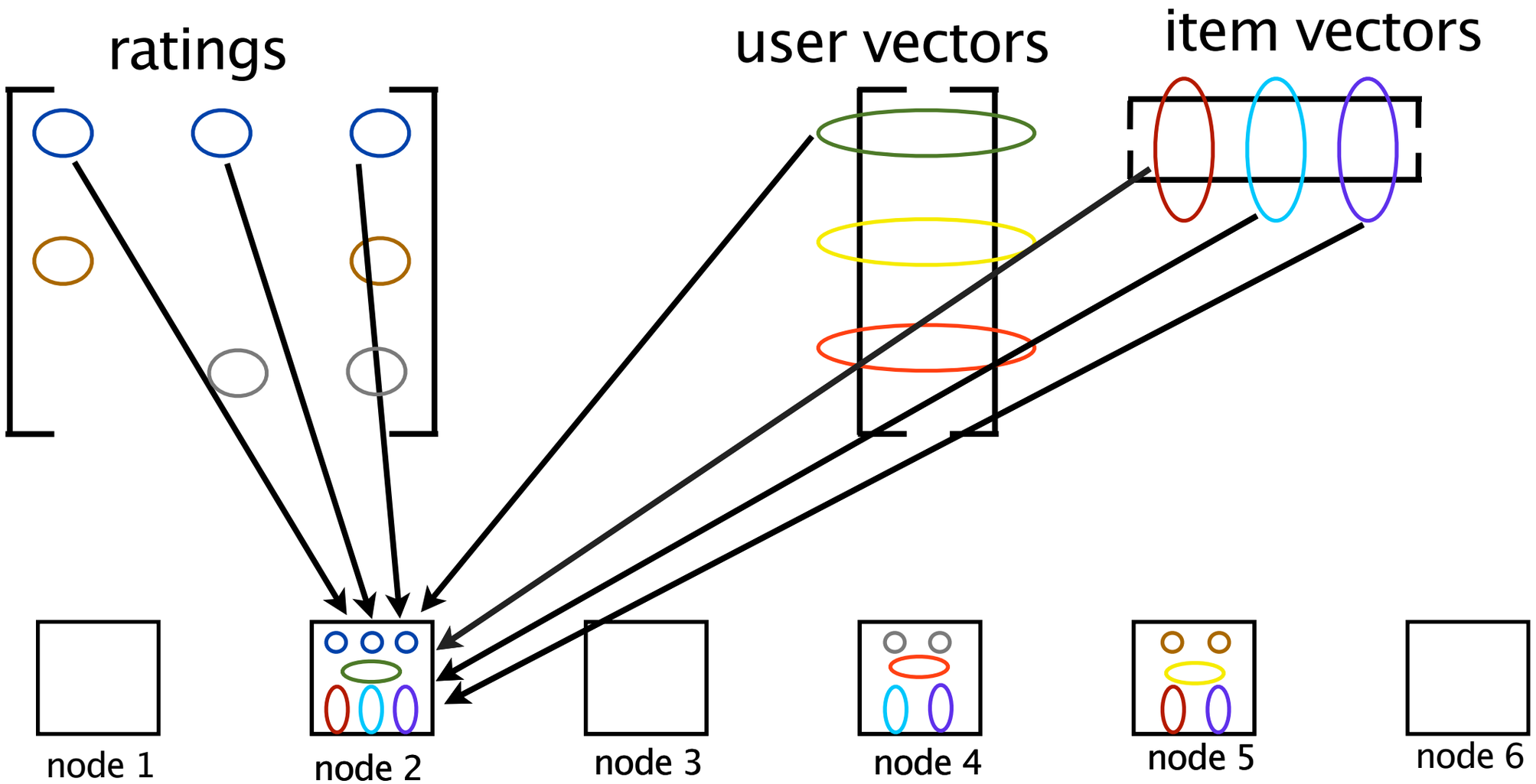
First Attempt

- For each iteration:
  - Compute YtY over item vectors and broadcast
  - Join user vectors along with all ratings for that user and all item vectors for which the user rated the item
  - Sum up YtCuIY and YtCuPu and solve for optimal user vectors



First Attempt

- For each iteration:
  - Compute YtY over item vectors and broadcast
  - Join user vectors along with all ratings for that user and all item vectors for which the user rated the item
  - Sum up YtCuIY and YtCuPu and solve for optimal user vectors



First Attempt

```
75    def ALSIteration(ratings: RDD[(Int, Int, Double)],
76                     users: RDD[(Int, DenseVector[Double])],
77                     items: RDD[(Int, DenseVector[Double])],
78                     rank: Int,
79                     lambda: Double,
80                     alpha: Double,
81                     YtY: Broadcast[DenseVector[Double]]){
82
83      users
84        .cartesian(items)
85        .groupByKey
86        .map{case((user, vector), items) =>
87          (user, (vector, items))}
88        .join(ratings.map{case (user, item, rating) =>
89          (user, (item, rating))}
90          .groupByKey)
91        .map{ case(user, ((vector, items), ratings)) =>
92          val itemsMap = Map ++ items
93          updateVector(user, vector, itemsMap, ratingsMap, rank, lambda, alpha, YtY)}
94    }
```

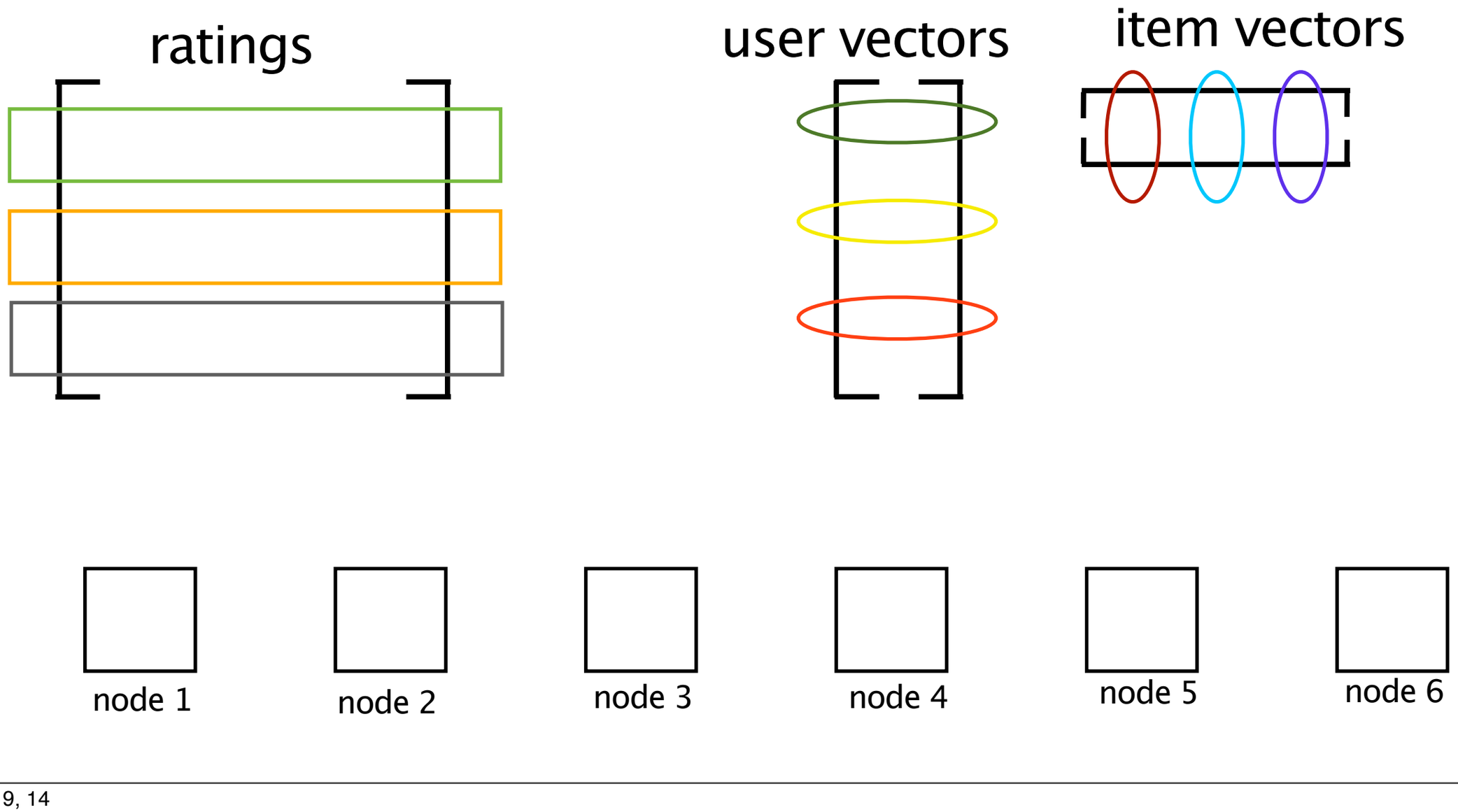
First Attempt

```
75    def ALSIteration(ratings: RDD[(Int, Int, Double)],
76                     users: RDD[(Int, DenseVector[Double])],
77                     items: RDD[(Int, DenseVector[Double])],
78                     rank: Int,
79                     lambda: Double,
80                     alpha: Double,
81                     YtY: Broadcast[DenseVector[Double]]){
82
83      users
84        .cartesian(items)
85        .groupByKey
86        .map{case((user, vector), items) =>
87          (user, (vector, items))}
88        .join(ratings.map{case (user, item, rating) =>
89          (user, (item, rating))}
90          .groupByKey)
91        .map{ case(user, ((vector, items), ratings)) =>
92          val itemsMap = Map ++ items
93          updateVector(user, vector, itemsMap, ratingsMap, rank, lambda, alpha, YtY)}
94    }
```

- Issues:
  - Unnecessarily sending multiple copies of item vector to each node
  - Unnecessarily shuffling data across cluster at each iteration
  - Not taking advantage of Spark's in memory capabilities!

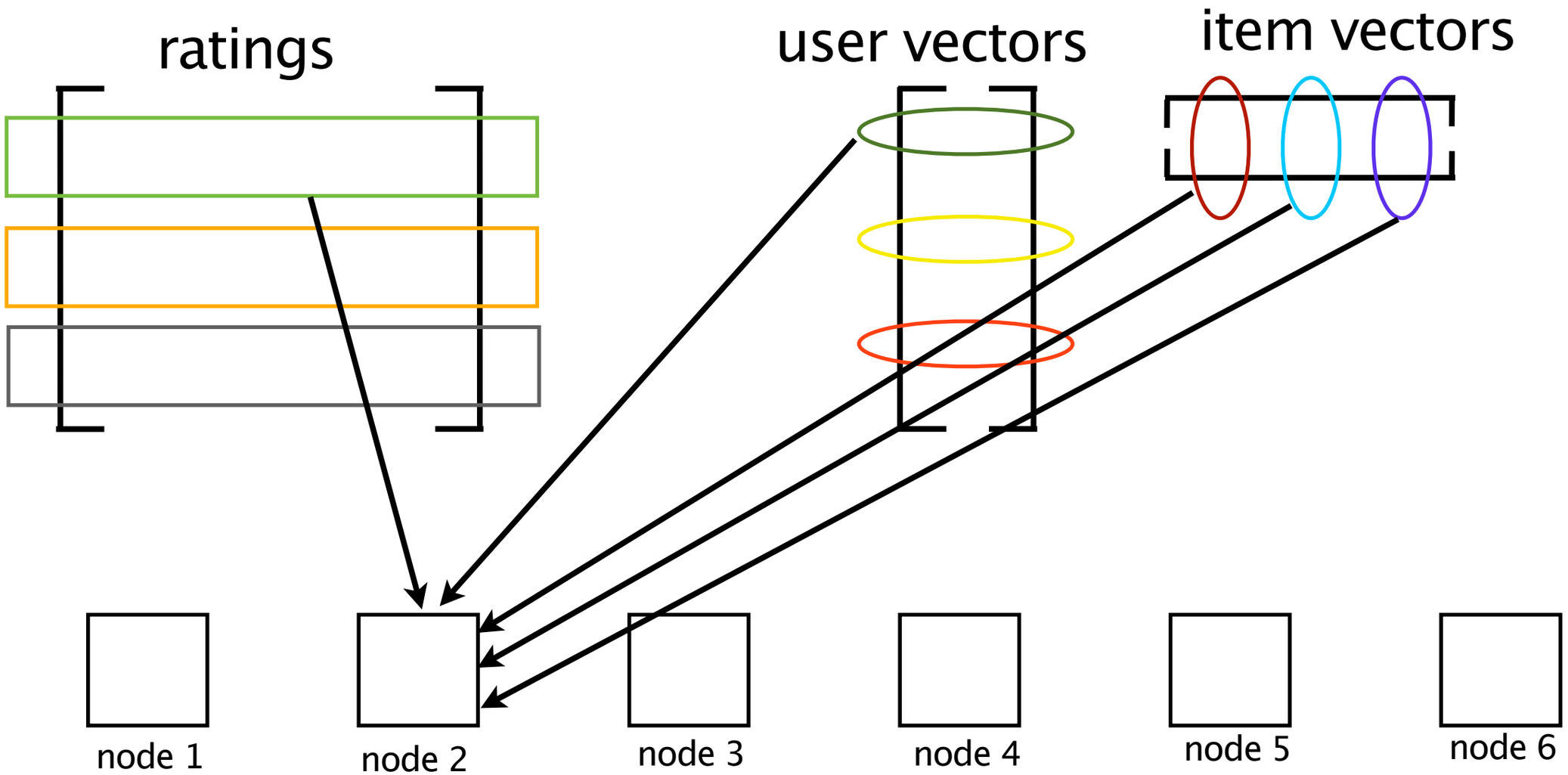
Second Attempt

- For each iteration:
  - Compute YtY over item vectors and broadcast
  - Group ratings matrix into blocks, and join blocks with necessary user and item vectors (to avoid multiple item vector copies at each node)
  - Sum up YtCuIY and YtCuPu and solve for optimal user vectors



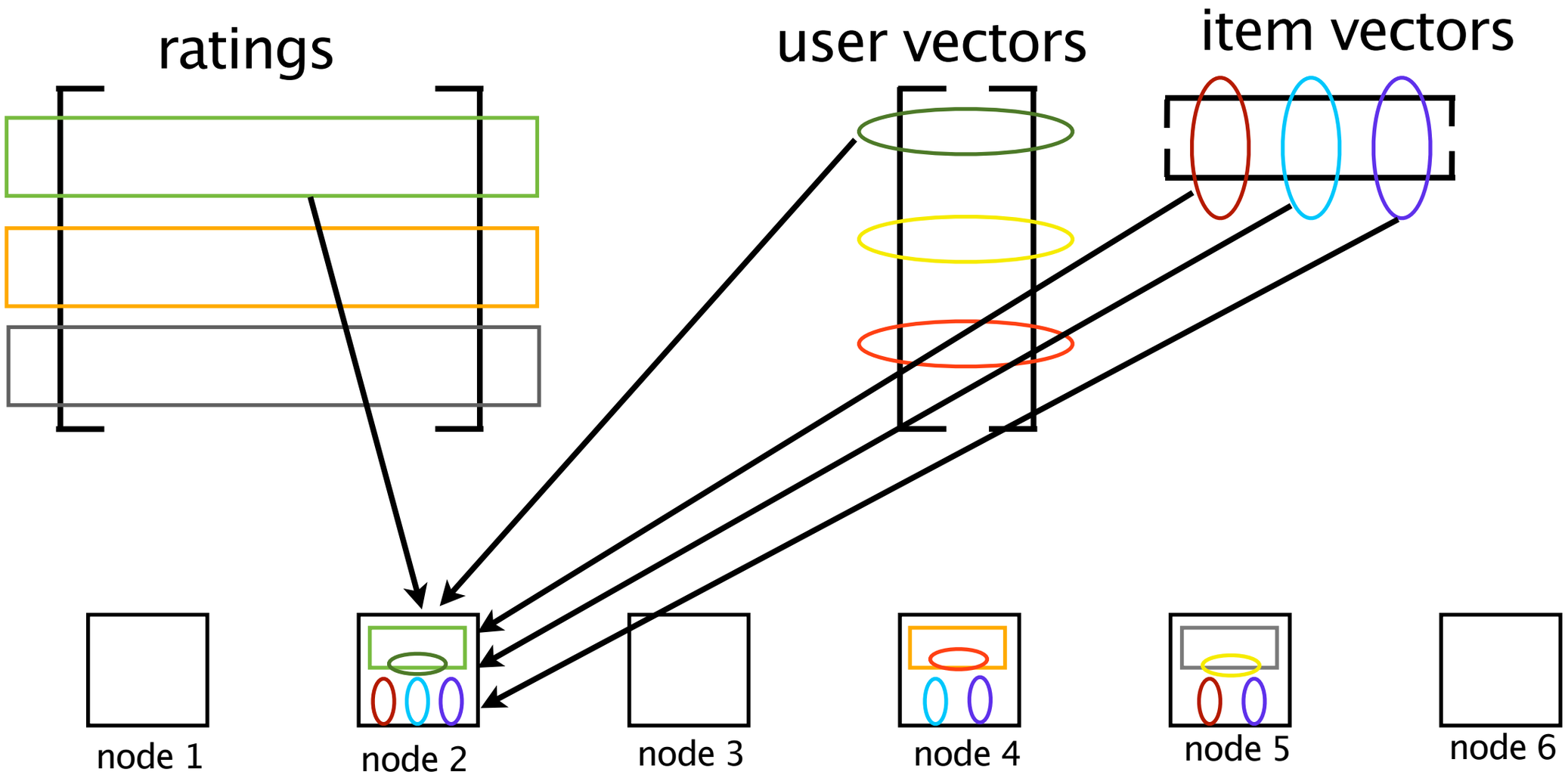
Second Attempt

- For each iteration:
  - Compute YtY over item vectors and broadcast
  - Group ratings matrix into blocks, and join blocks with necessary user and item vectors (to avoid multiple item vector copies at each node)
  - Sum up YtCuIY and YtCuPu and solve for optimal user vectors



Second Attempt

- For each iteration:
  - Compute YtY over item vectors and broadcast
  - Group ratings matrix into blocks, and join blocks with necessary user and item vectors (to avoid multiple item vector copies at each node)
  - Sum up YtCuIY and YtCuPu and solve for optimal user vectors



Second Attempt

```
35    def gridify(ratings: RDD[(Int, Int, Double)],
36                users: RDD[(Int, DenseVector[Double])],
37                items: RDD[(Int, DenseVector[Double])],
38                rank: Int,
39                blocks: Int) :
40                (RDD[(Int, (Int, Int, Double))],
41                  RDD[(Int, (Int, DenseVector[Double]))],
42                  RDD[(Int, (Int, DenseVector[Double]))]) = {
43
44      val ratingsByBlock = ratings
45        .map{case (user, item, rating)  =>
46          ((user % blocks), (user, item, rating))}
47
48      val usersByBlock = users
49        .map{case (user, vector) =>
50        ((user % blocks), (user, vector))}
51
52      val itemsByBlock = ratings
53        .map{case (user, item, rating) =>
54          (item, (user, rating))}
55        .groupByKey
56        .join(items)
57        .flatMap{case (item, (ratings, vector)) =>
58          ratings.map{case (user, rating) =>
59            ((user % blocks), (item, vector))
60          }
61        }.distinct
62
63      (ratingsByBlock, usersByBlock, itemsByBlock)
64    }
```

Second Attempt

```
21    def ALSIteration(ratingsByBlock: RDD[(Int, (Int, Int, Double))],
22                     usersByBlock: RDD[(Int, (Int, DenseVector[Double]))],
23                     itemsByBlock: RDD[(Int, (Int, DenseVector[Double]))],
24                     rank: Int,
25                     lambda: Double,
26                     alpha: Double,
27                     YtY: Broadcast[DenseVector[Double]]){
28
29      usersByBlock
30        .groupByKey
31        .join(ratingsByBlock
32                .groupByKey)
33        .join(itemsByBlock
34                .groupByKey)
35        .mapValues{case ((users, ratings), items) =>
36          val ratingsMap = Map ++ ratings.map{case (user, item, rating) =>
37            ((user, item), rating)
38          val itemsMap = Map ++ items
39          users.map{case (user, vector) =>
40            updateVector(user, vector, itemsMap, ratingsMap, rank, lambda, alpha, YtY)}
41
42    }
```

Second Attempt

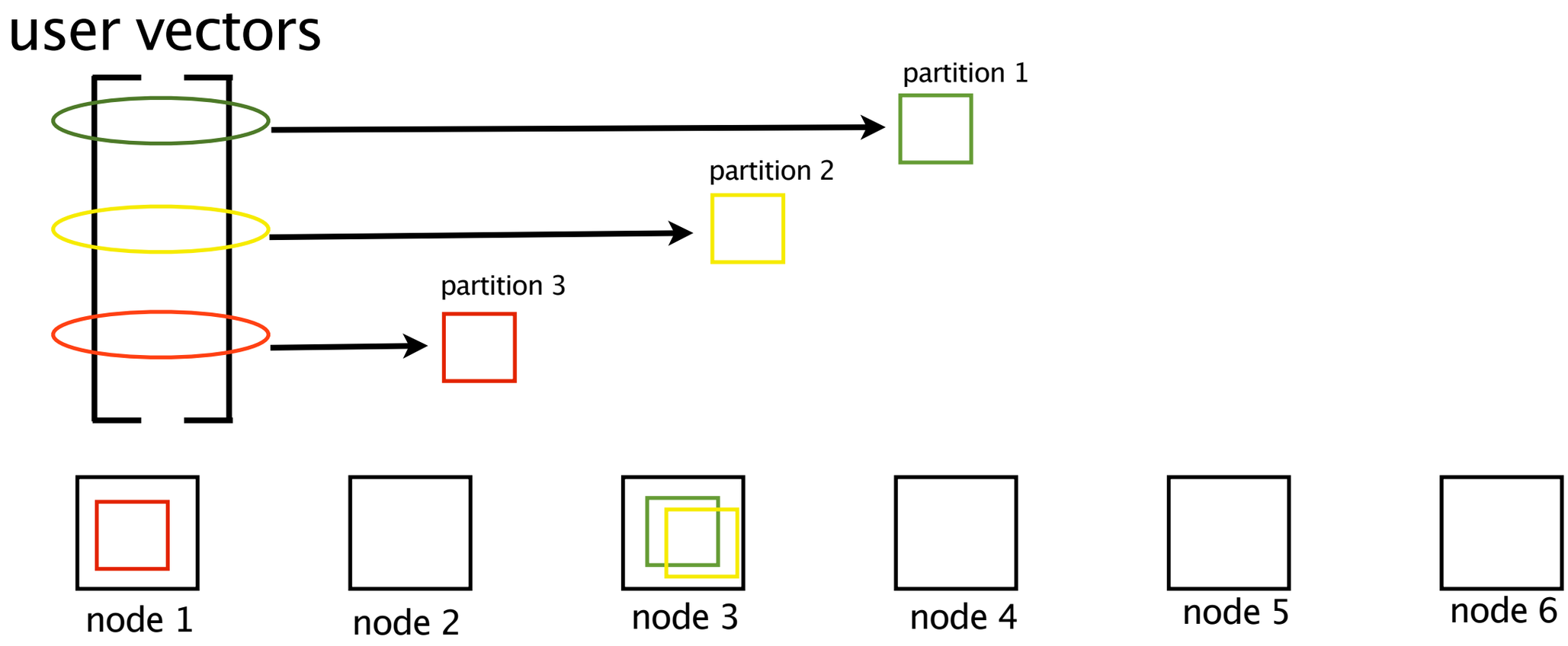
```
21      def ALSIteration(ratingsByBlock: RDD[(Int, (Int, Int, Double))],
22                       usersByBlock: RDD[(Int, (Int, DenseVector[Double]))],
23                       itemsByBlock: RDD[(Int, (Int, DenseVector[Double]))],
24                       rank: Int,
25                       lambda: Double,
26                       alpha: Double,
27                       YtY: Broadcast[DenseVector[Double]]){
28
29        usersByBlock
30          .groupByKey
31          .join(ratingsByBlock
32                  .groupByKey)
33          .join(itemsByBlock
34                  .groupByKey)
35          .mapValues{case ((users, ratings), items) =>
36            val ratingsMap = Map ++ ratings.map{case (user, item, rating) =>
37              ((user, item), rating)
38            val itemsMap = Map ++ items
39            users.map{case (user, vector) =>
40              updateVector(user, vector, itemsMap, ratingsMap, rank, lambda, alpha, YtY)}
41
42      }
```

- Issues:
  - Still Unnecessarily shuffling data across cluster at each iteration
  - Still not taking advantage of Spark's in memory capabilities!

So, what are we missing?...

- Partitioner: Defines how the elements in a key-value pair RDD are partitioned across the cluster.

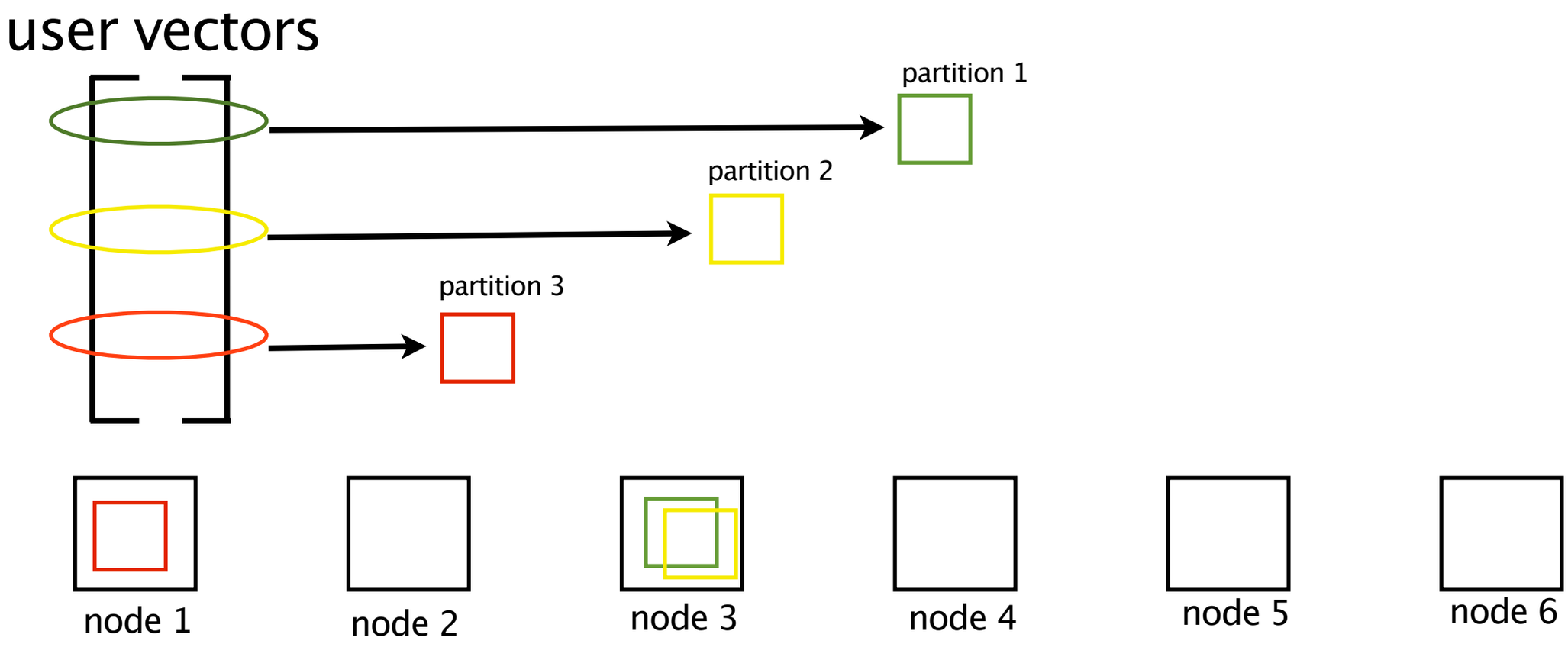
```
28      val partitioner = new HashPartitioner(blocks)
29
30      val userVectors = sc.parallelize(List((1, DenseVector((1,2))), (2, DenseVector((3,4))), (3, DenseVector((4,5)))))
```



So, what are we missing?...

- partitionBy(partitioner): Partitions all elements of the same key to the same node in the cluster, as defined by the partitioner.

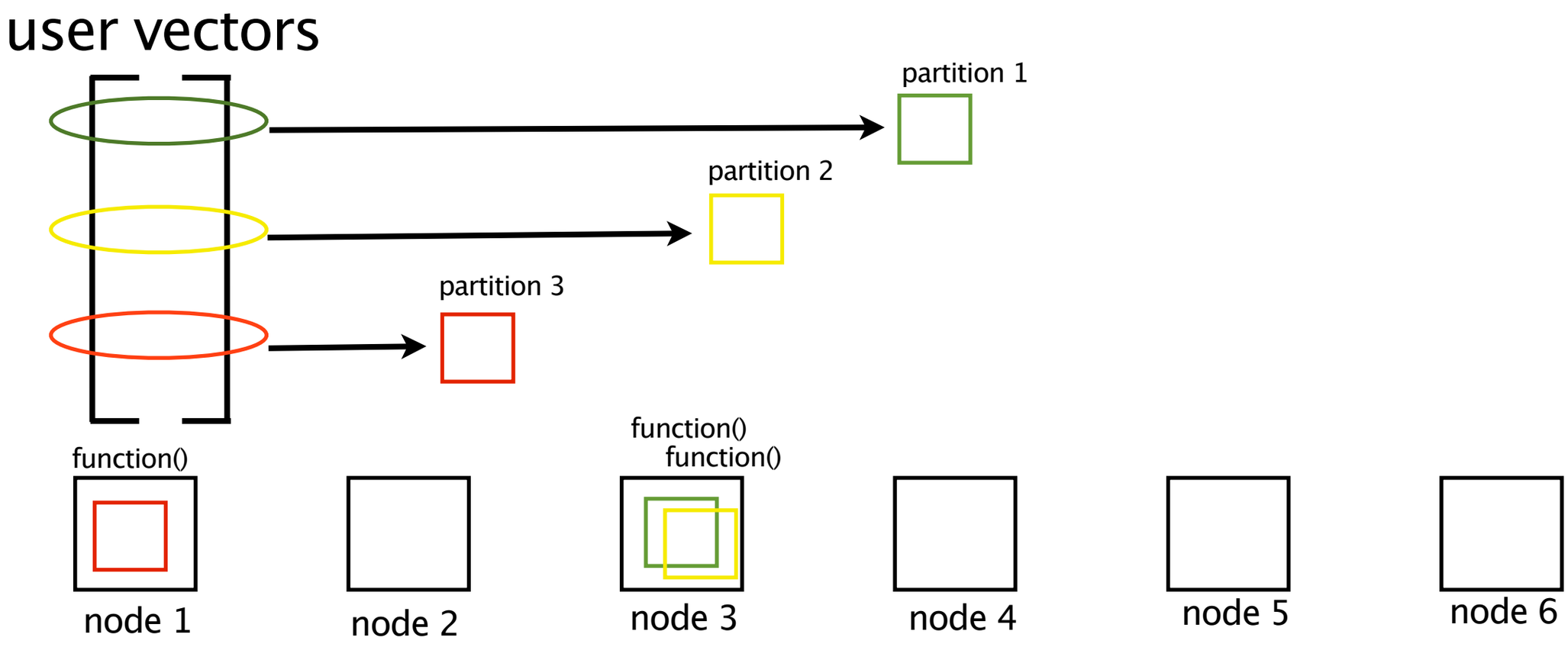
```
38      val partitioner = new HashPartitioner(blocks)
39
40      val userVectors = sc.parallelize(List((1, DenseVector((1,2))), (2, DenseVector((3,4))), (3, DenseVector((4,5)))))
41
42      userVectors
43        .partitionBy(partitioner)
```



So, what are we missing?...

- mapPartitions(func): Similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator[T] => Iterator[U] when running on an RDD of type T.

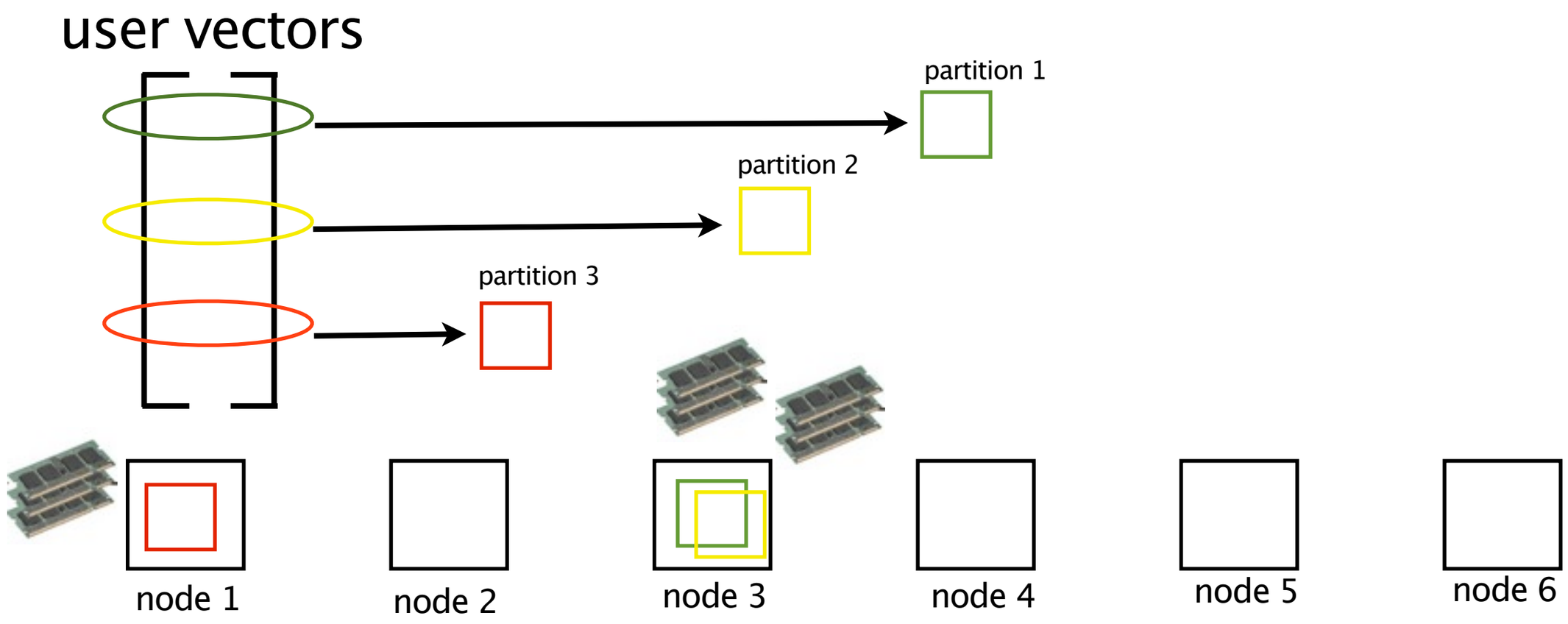
```
38      val partitioner = new HashPartitioner(blocks)
39
40      val userVectors = sc.parallelize(List((1, DenseVector((1,2))), (2, DenseVector((3,4))), (3, DenseVector((4,5)))))
41
42      userVectors
43        .partitionBy(partitioner)
44        .mapPartitions{case(user, vector) =>
45          Iterator.single(function(user, vector))}
```



So, what are we missing?...

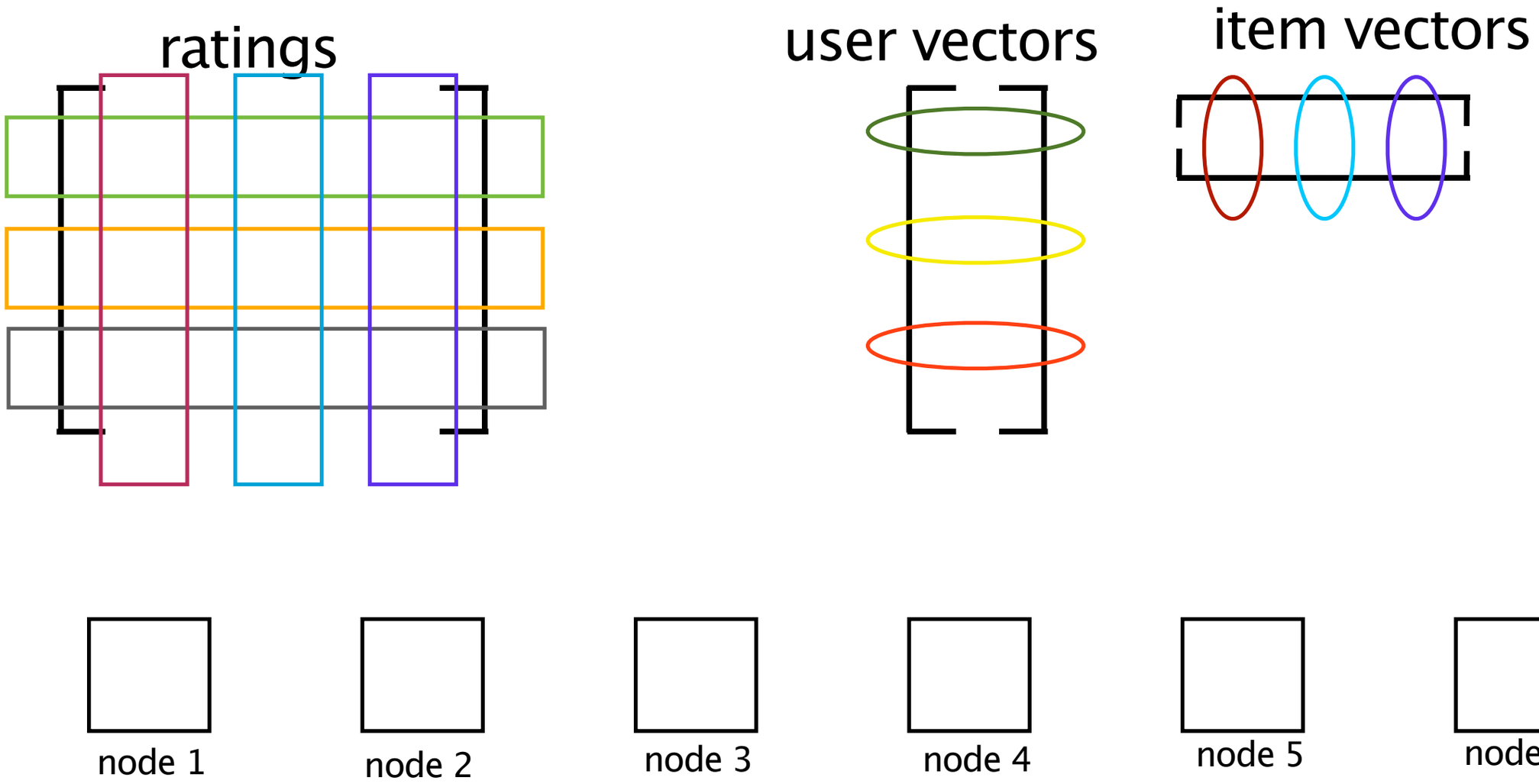
- persist(storageLevel): Set this RDD's storage level to persist (cache) its values across operations after the first time it is computed.

```
38      val partitioner = new HashPartitioner(blocks)
39
40      val userVectors = sc.parallelize(List((1, DenseVector((1,2))), (2, DenseVector((3,4))), (3, DenseVector((4,5)))))
41
42      userVectors
43        .partitionBy(partitioner)
44        .mapPartitions{case(user, vector) =>
45          Iterator.single(function(user, vector))}
46        .persist(StorageLevel.MEMORY_AND_DISK)
```



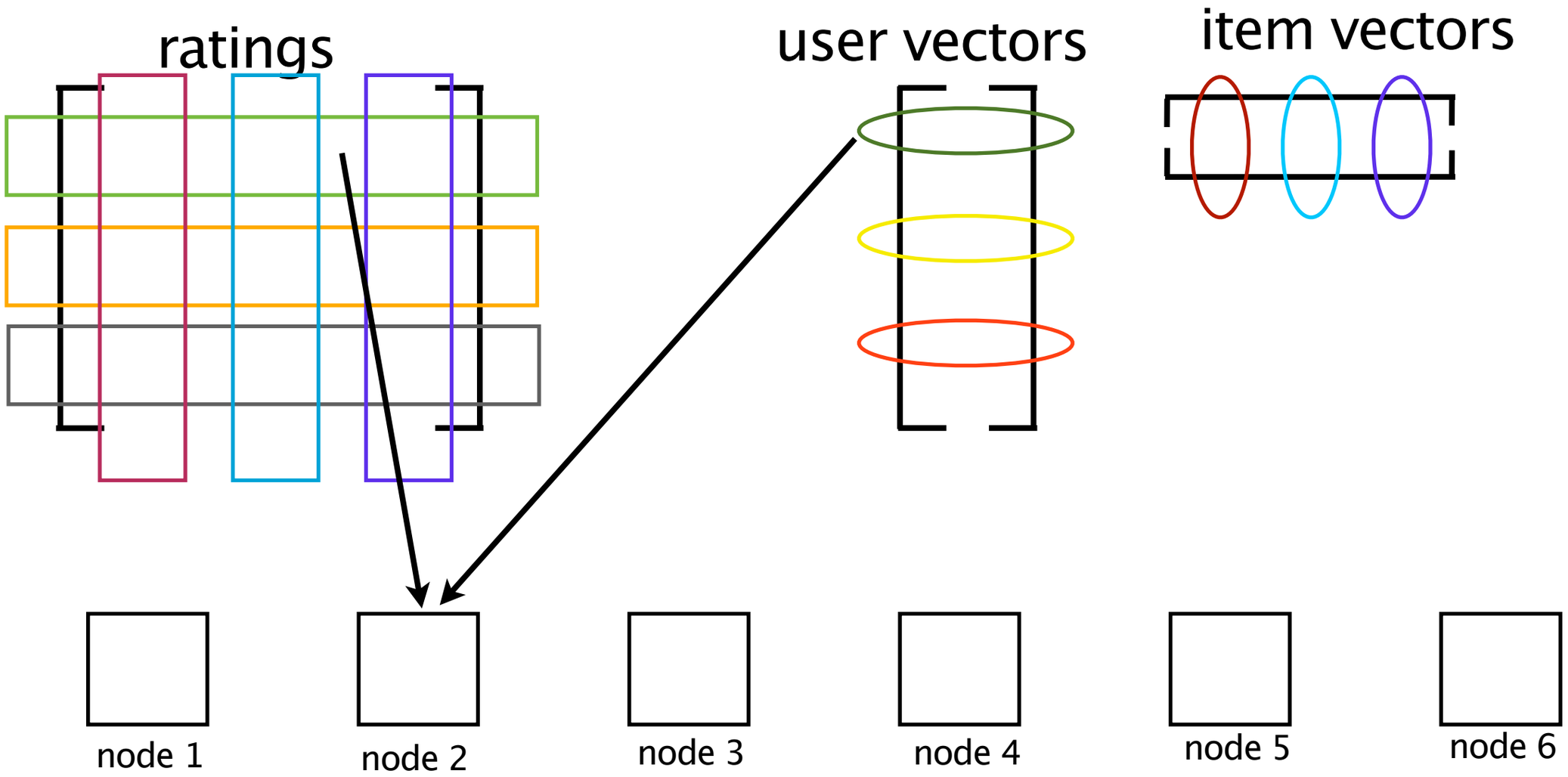
Third Attempt

- Partition ratings matrix, user vectors, and item vectors by user and item blocks and cache partitions in memory
- Build InLink and OutLink mappings for users and items
  - InLink Mapping: Includes the user IDs and vectors for a given block along with the ratings for each user in this block
  - OutLink Mapping: Includes the item IDs and vectors for a given block along with a list of destination blocks for which to send these vectors
- For each iteration:
  - Compute YtY over item vectors and broadcast
  - On each item block, use the OutLink mapping to send item vectors to the necessary user blocks
  - On each user block, use the InLink mapping along with the joined item vectors to update vectors



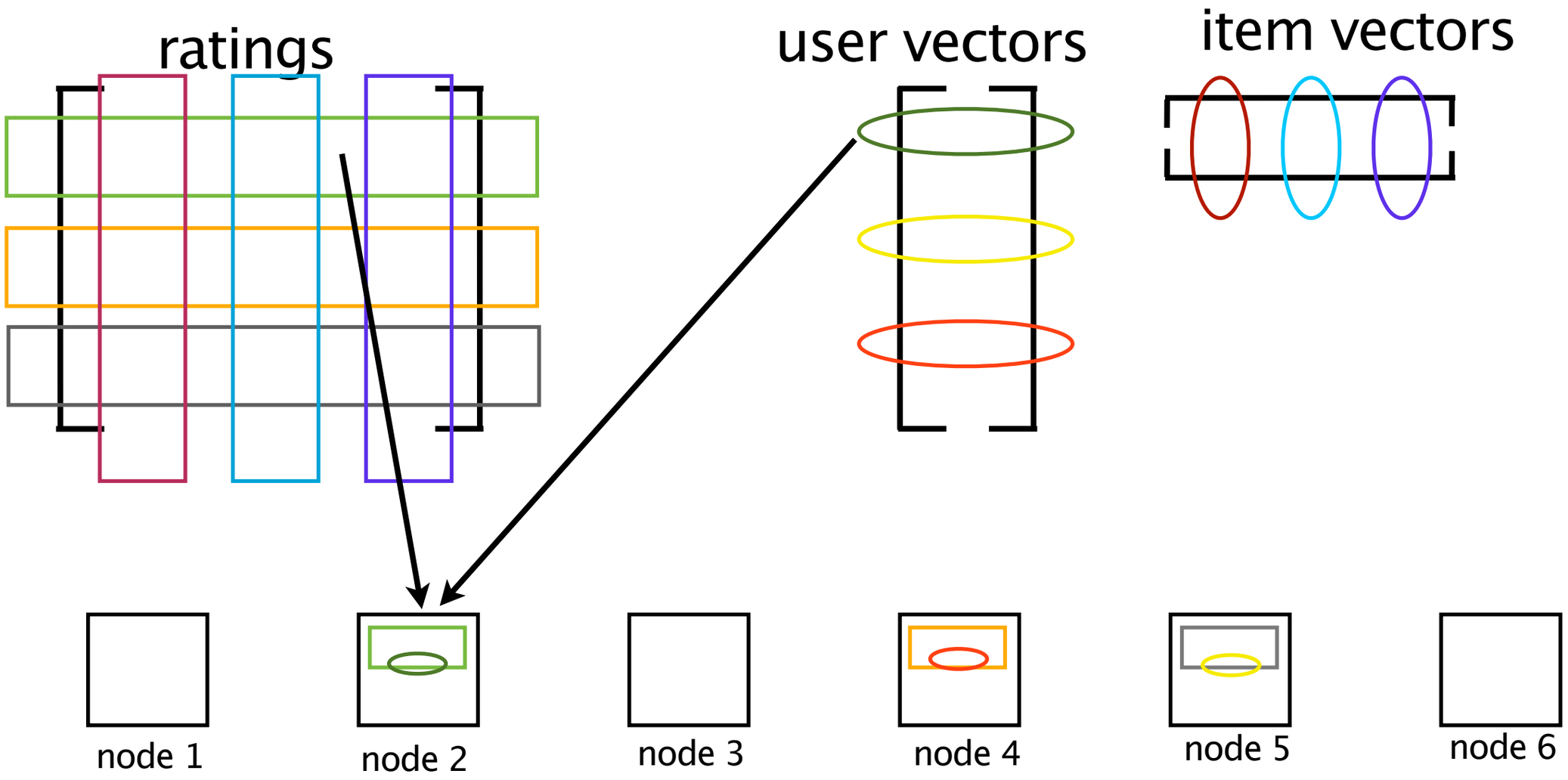
Third Attempt

- Partition ratings matrix, user vectors, and item vectors by user and item blocks and cache partitions in memory
- Build InLink and OutLink mappings for users and items
  - InLink Mapping: Includes the user IDs and vectors for a given block along with the ratings for each user in this block
  - OutLink Mapping: Includes the item IDs and vectors for a given block along with a list of destination blocks for which to send these vectors
- For each iteration:
  - Compute YtY over item vectors and broadcast
  - On each item block, use the OutLink mapping to send item vectors to the necessary user blocks
  - On each user block, use the InLink mapping along with the joined item vectors to update vectors



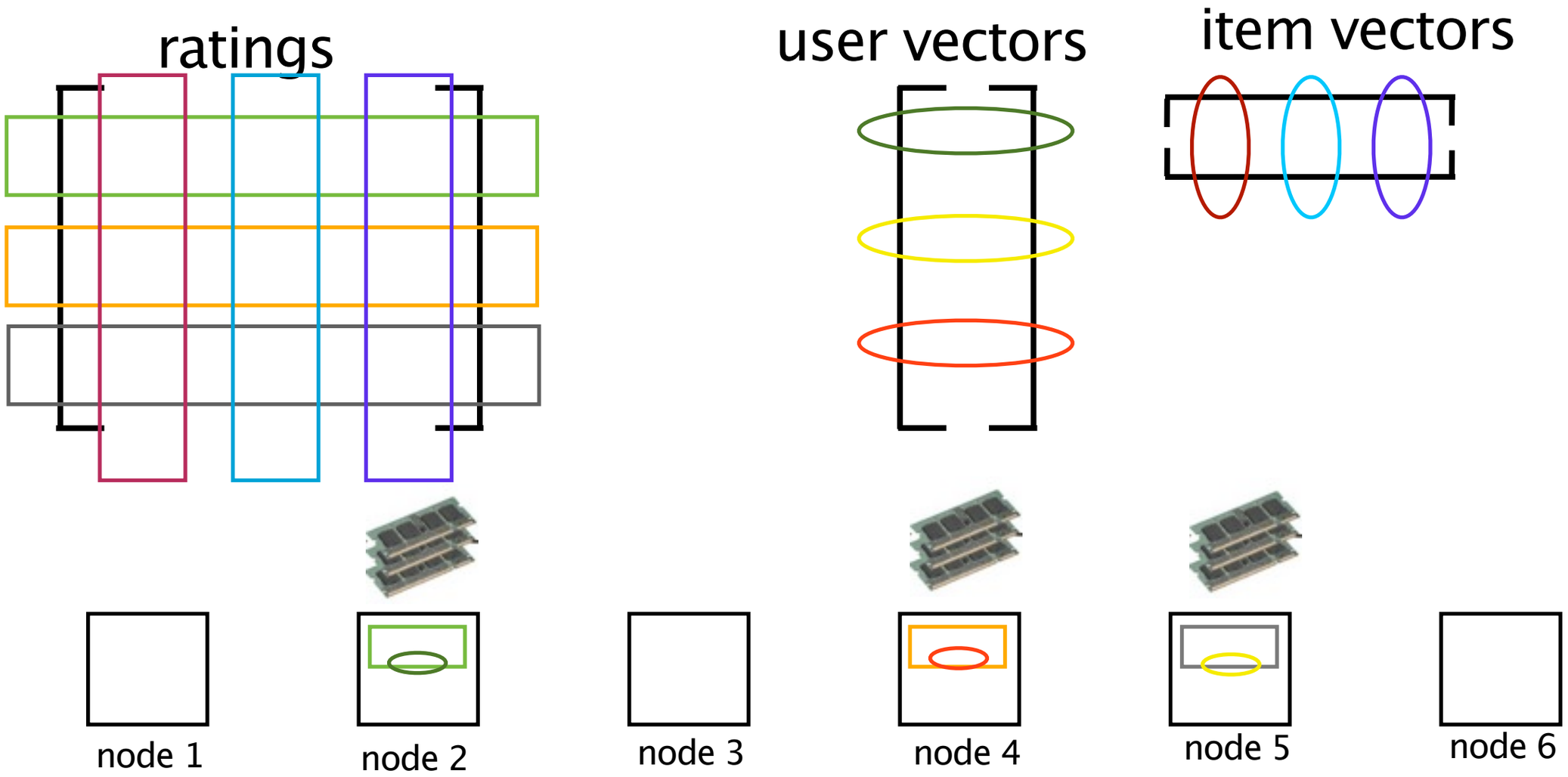
Third Attempt

- Partition ratings matrix, user vectors, and item vectors by user and item blocks and cache partitions in memory
- Build InLink and OutLink mappings for users and items
  - InLink Mapping: Includes the user IDs and vectors for a given block along with the ratings for each user in this block
  - OutLink Mapping: Includes the item IDs and vectors for a given block along with a list of destination blocks for which to send these vectors
- For each iteration:
  - Compute YtY over item vectors and broadcast
  - On each item block, use the OutLink mapping to send item vectors to the necessary user blocks
  - On each user block, use the InLink mapping along with the joined item vectors to update vectors



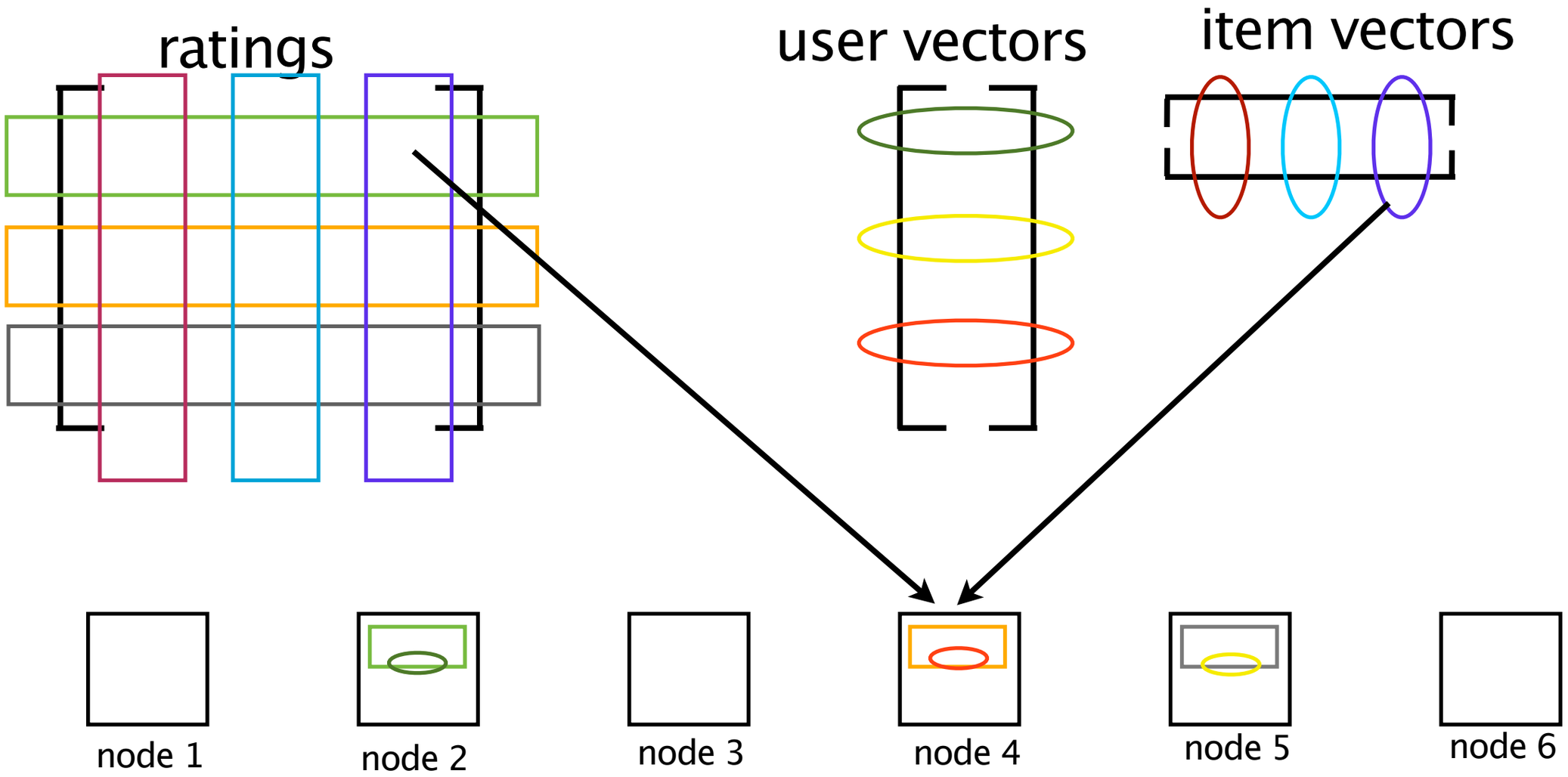
Third Attempt

- Partition ratings matrix, user vectors, and item vectors by user and item blocks and cache partitions in memory
- Build InLink and OutLink mappings for users and items
  - InLink Mapping: Includes the user IDs and vectors for a given block along with the ratings for each user in this block
  - OutLink Mapping: Includes the item IDs and vectors for a given block along with a list of destination blocks for which to send these vectors
- For each iteration:
  - Compute YtY over item vectors and broadcast
  - On each item block, use the OutLink mapping to send item vectors to the necessary user blocks
  - On each user block, use the InLink mapping along with the joined item vectors to update vectors



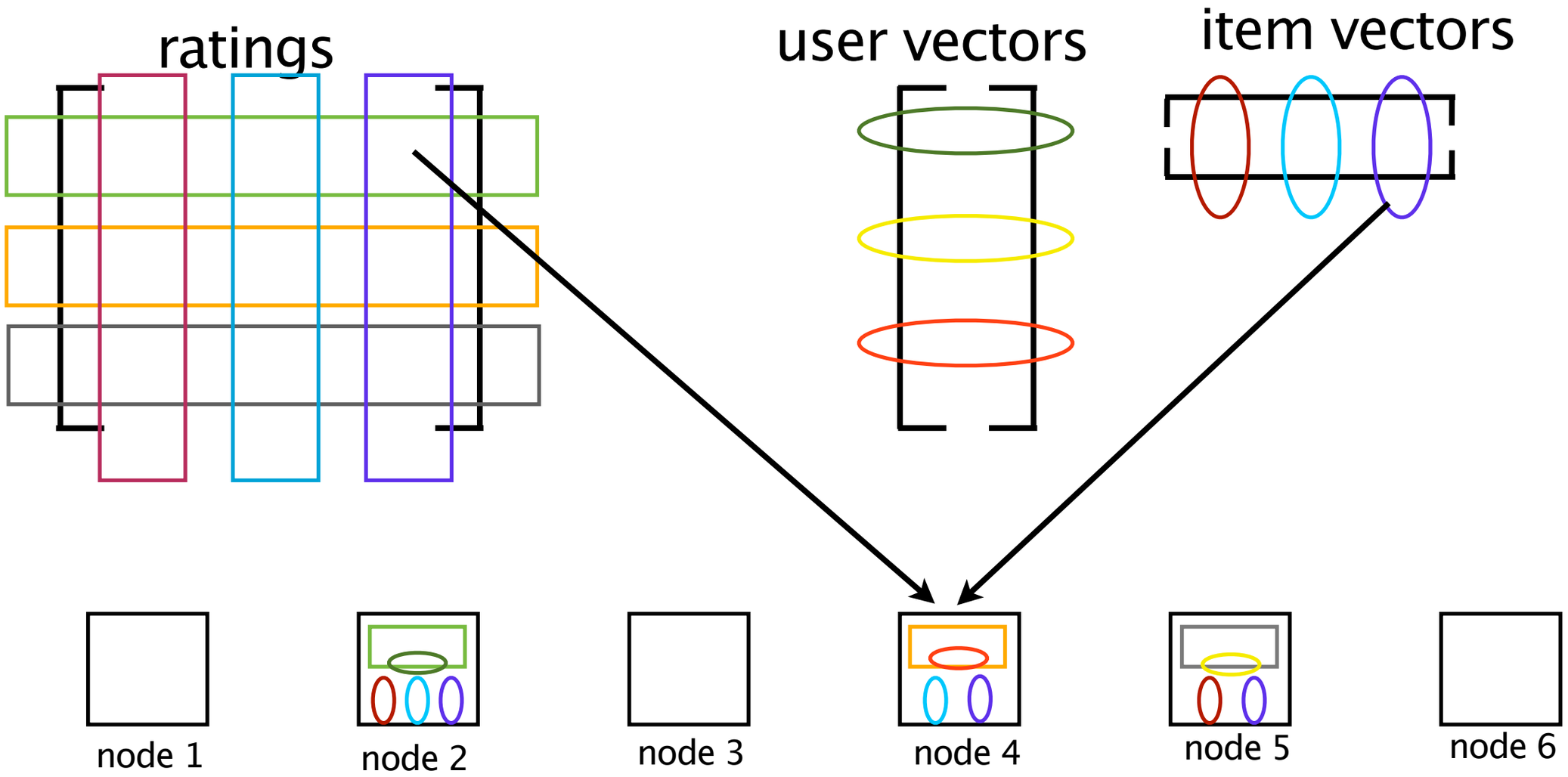
Third Attempt

- Partition ratings matrix, user vectors, and item vectors by user and item blocks and cache partitions in memory
- Build InLink and OutLink mappings for users and items
  - InLink Mapping: Includes the user IDs and vectors for a given block along with the ratings for each user in this block
  - OutLink Mapping: Includes the item IDs and vectors for a given block along with a list of destination blocks for which to send these vectors
- For each iteration:
  - Compute YtY over item vectors and broadcast
  - On each item block, use the OutLink mapping to send item vectors to the necessary user blocks
  - On each user block, use the InLink mapping along with the joined item vectors to update vectors



Third Attempt

- Partition ratings matrix, user vectors, and item vectors by user and item blocks and cache partitions in memory
- Build InLink and OutLink mappings for users and items
  - InLink Mapping: Includes the user IDs and vectors for a given block along with the ratings for each user in this block
  - OutLink Mapping: Includes the item IDs and vectors for a given block along with a list of destination blocks for which to send these vectors
- For each iteration:
  - Compute YtY over item vectors and broadcast
  - On each item block, use the OutLink mapping to send item vectors to the necessary user blocks
  - On each user block, use the InLink mapping along with the joined item vectors to update vectors



Third attempt

```
281   /**
282    * Make RDDs of InLinkBlocks and OutLinkBlocks given an RDD of (blockId, (u, p, r)) values for
283    * the users (or (blockId, (p, u, r)) for the products). We create these simultaneously to avoid
284    * having to shuffle the (blockId, (u, p, r)) RDD twice, or to cache it.
285    */
286   private def makeLinkRDDs(numBlocks: Int, ratings: RDD[(Int, Rating)])
287     : (RDD[(Int, InLinkBlock)], RDD[(Int, OutLinkBlock)]) =
288   {
289     val grouped = ratings.partitionBy(new HashPartitioner(numBlocks))
290     val links = grouped.mapPartitionsWithIndex((blockId, elements) => {
291       val ratings = elements.map{_._2}.toArray
292       val inLinkBlock = makeInLinkBlock(numBlocks, ratings)
293       val outLinkBlock = makeOutLinkBlock(numBlocks, ratings)
294       Iterator.single((blockId, (inLinkBlock, outLinkBlock)))
295     }, true)
296     links.persist(StorageLevel.MEMORY_AND_DISK)
297     (links.mapValues(_._1), links.mapValues(_._2))
298   }
```

Third attempt

```
307   /**
308    * Compute the user feature vectors given the current products (or vice-versa). This first joins
309    * the products with their out-links to generate a set of messages to each destination block
310    * (specifically, the features for the products that user block cares about), then groups these
311    * by destination and joins them with the in-link info to figure out how to update each user.
312    * It returns an RDD of new feature vectors for each user block.
313    */
314   private def updateFeatures(
315       products: RDD[(Int, Array[Array[Double]])],
316       productOutLinks: RDD[(Int, OutLinkBlock)],
317       userInLinks: RDD[(Int, InLinkBlock)],
318       partitioner: Partitioner,
319       rank: Int,
320       lambda: Double,
321       alpha: Double,
322       YtY: Broadcast[Option[DoubleMatrix]])
323     : RDD[(Int, Array[Array[Double]])] =
324   {
325     val numBlocks = products.partitions.size
326     productOutLinks.join(products).flatMap { case (bid, (outLinkBlock, factors)) =>
327         val toSend = Array.fill(numBlocks)(new ArrayBuffer[Array[Double]])
328         for (p <- 0 until outLinkBlock.elementIds.length; userBlock <- 0 until numBlocks) {
329           if (outLinkBlock.shouldSend(p)(userBlock)) {
330             toSend(userBlock) += factors(p)
331           }
332         }
333         toSend.zipWithIndex.map{ case (buf, idx) => (idx, (bid, buf.toArray)) }
334     }.groupByKey(partitioner)
335      .join(userInLinks)
336      .mapValues{ case (messages, inLinkBlock) =>
337        updateBlock(messages, inLinkBlock, rank, lambda, alpha, YtY)
338      }
339   }
```

Third attempt

```
307   /**
308    * Compute the user feature vectors given the current products (or vice-versa). This first joins
309    * the products with their out-links to generate a set of messages to each destination block
310    * (specifically, the features for the products that user block cares about), then groups these
311    * by destination and joins them with the in-link info to figure out how to update each user.
312    * It returns an RDD of new feature vectors for each user block.
313    */
314   private def updateFeatures(
315       products: RDD[(Int, Array[Array[Double]])],
316       productOutLinks: RDD[(Int, OutLinkBlock)],
317       userInLinks: RDD[(Int, InLinkBlock)],
318       partitioner: Partitioner,
319       rank: Int,
320       lambda: Double,
321       alpha: Double,
322       YtY: Broadcast[Option[DoubleMatrix]])
323     : RDD[(Int, Array[Array[Double]])] =
324   {
325     val numBlocks = products.partitions.size
326     productOutLinks.join(products).flatMap { case (bid, (outLinkBlock, factors)) =>
327         val toSend = Array.fill(numBlocks)(new ArrayBuffer[Array[Double]])
328         for (p <- 0 until outLinkBlock.elementIds.length; userBlock <- 0 until numBlocks) {
329           if (outLinkBlock.shouldSend(p)(userBlock)) {
330             toSend(userBlock) += factors(p)
331           }
332         }
333         toSend.zipWithIndex.map{ case (buf, idx) => (idx, (bid, buf.toArray)) }
334     }.groupByKey(partitioner)
335      .join(userInLinks)
336      .mapValues{ case (messages, inLinkBlock) =>
337        updateBlock(messages, inLinkBlock, rank, lambda, alpha, YtY)
338      }
339   }
```

ALS Running Times

System	Wall-clock time (seconds)
MATLAB	15443
Mahout	4206
GraphLab	291
MLlib	481

- Dataset: scaled version of Netflix data (9X in size).
- Cluster: 9 machines.
- MLlib is an order of magnitude faster than Mahout.
- MLlib is within factor of 2 of GraphLab.

Via Xiangrui Meng (Databricks) http://stanford.edu/~rezab/sparkworkshop/slides/xiangrui.pdf


Fin