Music Recommendations at Scale with Spark

Chris Johnson
@MrChrisJohnson

Spotify

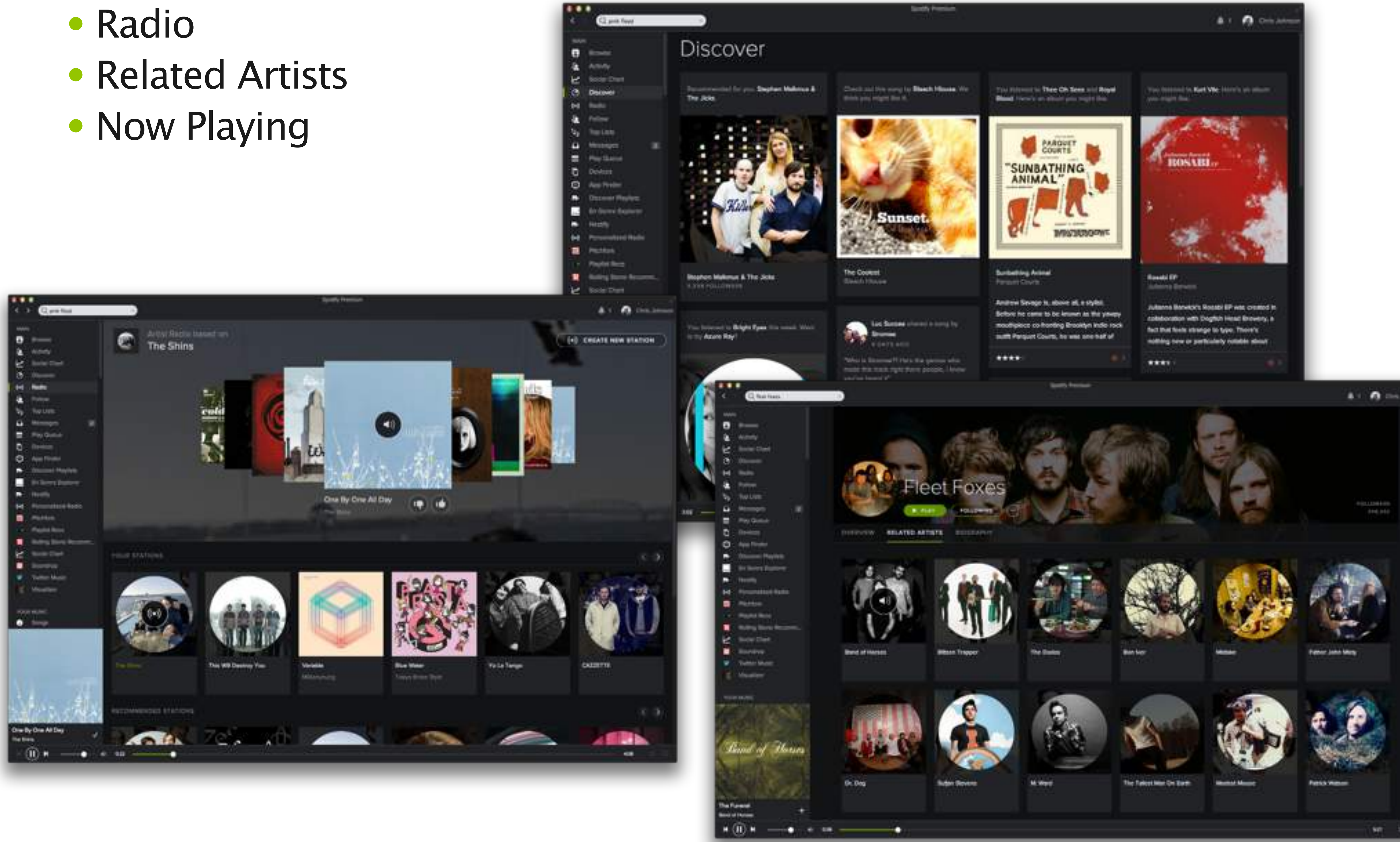
June 27, 2014

Who am I??

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

Recommendations at Spotify

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



How can we find good recommendations?

- Manual Curation
- Manually Tag Attributes
- Audio Content, Metadata, Text Analysis
- Collaborative Filtering

How can we find good recommendations?

- Manual Curation
- Manually Tag Attributes
- Audio Content, Metadata, Text Analysis
- Collaborative Filtering

Collaborative Filtering – “The Netflix Prize”

Collaborative Filtering

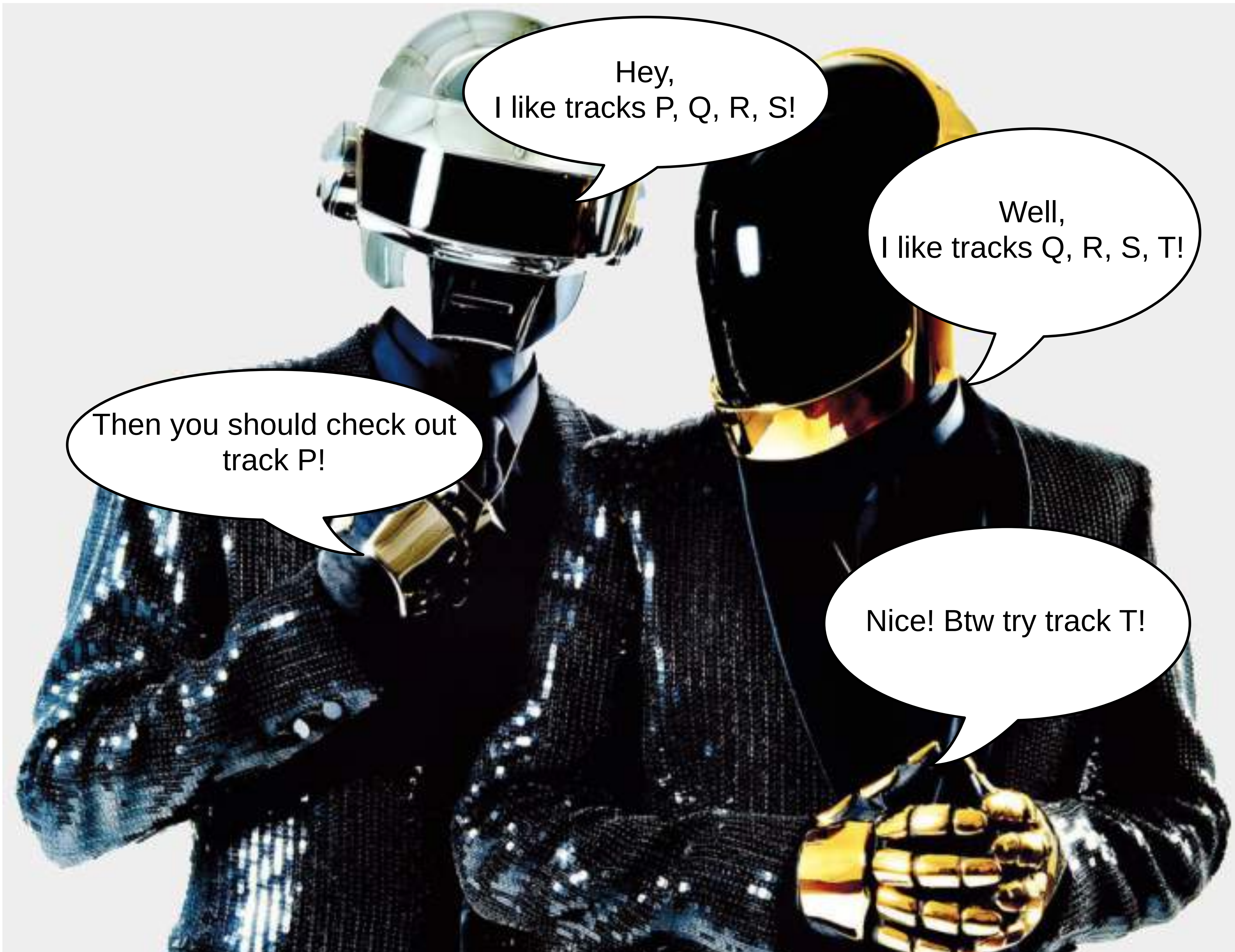


Image via Erik Bernhardsson

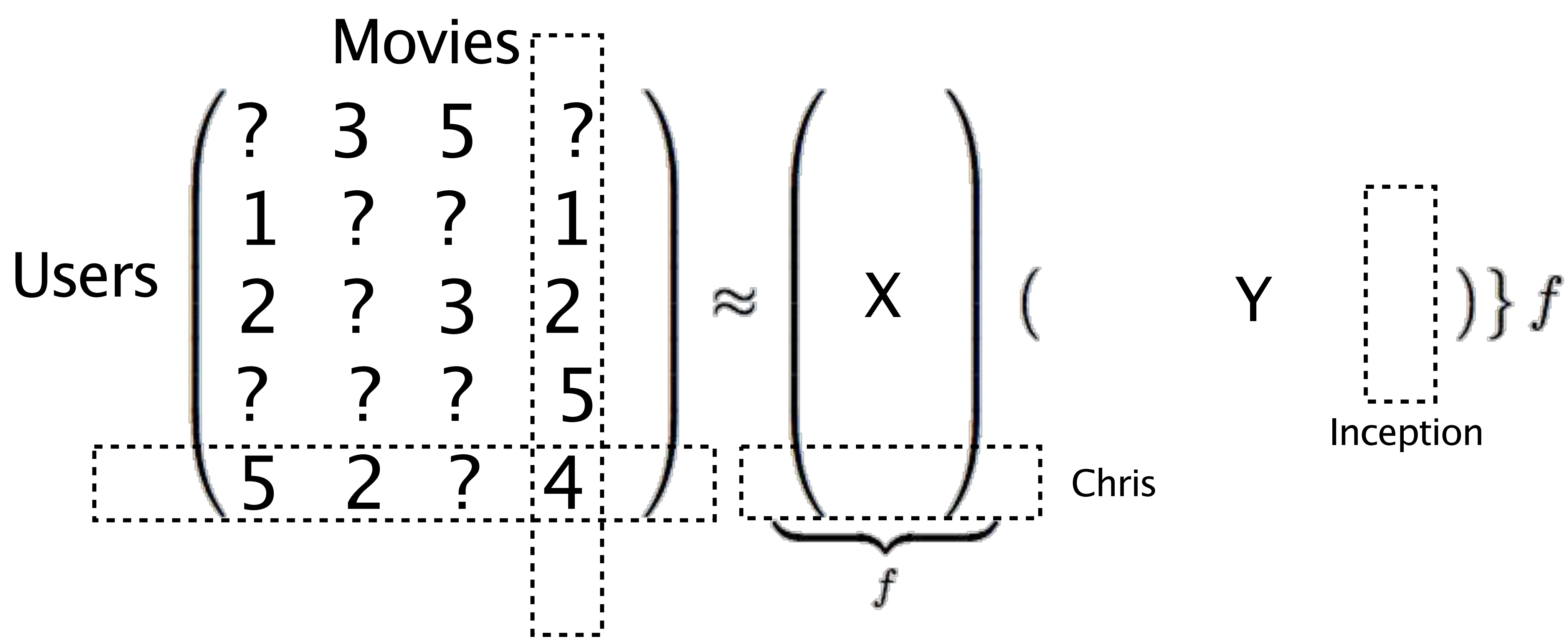
Explicit Matrix Factorization

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

$$\begin{array}{r c l}
 & \text{Movies} & \\
\text{Users} & \begin{pmatrix} ? & 3 & 5 & ? \\ 1 & ? & ? & 1 \\ 2 & ? & 3 & 2 \\ ? & ? & ? & 5 \\ 5 & 2 & ? & 4 \end{pmatrix} & \begin{array}{l} \\ \\ \\ \\ \text{Chris} \end{array} \\
 & \text{Inception} &
\end{array}$$

Explicit Matrix Factorization

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



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

- 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
- β_u = bias for user u
- β_i = bias for item i
- λ = regularization parameter

Implicit Matrix Factorization

- Instead of explicit ratings use binary labels
  - 1 = streamed, 0 = never streamed
- Minimize weighted RMSE (root mean squared error) using a function of total streams as weights

$$\begin{array}{cc} \text{Users} & \begin{pmatrix} 1&0&0&0&1&0&0&1 \\ 0&0&1&0&0&1&0&0 \\ 1&0&1&0&0&0&1&1 \\ 0&1&0&0&0&1&0&0 \\ 0&0&1&0&0&1&0&0 \\ 1&0&0&0&1&0&0&1 \end{pmatrix} \\ & \text{Songs} \end{array} \approx \underbrace{\begin{pmatrix} \\ \\ X \\ \\ \\ \end{pmatrix}}_{f} \left(\qquad Y \qquad \right)\} f$$

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

- 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
- β_u = bias for user u
- β_i = bias for item i
- λ = regularization parameter

Alternating Least Squares (ALS)

- Instead of explicit ratings use binary labels
  - 1 = streamed, 0 = never streamed
- Minimize weighted RMSE (root mean squared error) using a function of total streams as weights

$$\text{Users}\ \underbrace{\begin{pmatrix} 1&0&0&0&1&0&0&1\\ 0&0&1&0&0&1&0&0\\ 1&0&1&0&0&0&1&1\\ 0&1&0&0&0&1&0&0\\ 0&0&1&0&0&1&0&0\\ 1&0&0&0&1&0&0&1 \end{pmatrix}}_{\text{Songs}} \approx \underbrace{\begin{pmatrix} \\ \\ X \\ \\ \\ \end{pmatrix}}_{f} \left.\begin{pmatrix} \quad & Y & \quad \end{pmatrix}\right\} f$$

Fix songs

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

- 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
- β_u = bias for user u
- β_i = bias for item i
- λ = regularization parameter

Alternating Least Squares (ALS)

- Instead of explicit ratings use binary labels
  - 1 = streamed, 0 = never streamed
- Minimize weighted RMSE (root mean squared error) using a function of total streams as weights

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

Solve for users

$$\text{Users}\begin{pmatrix} 1&0&0&0&1&0&0&1\\ 0&0&1&0&0&1&0&0\\ 1&0&1&0&0&0&1&1\\ 0&1&0&0&0&1&0&0\\ 0&0&1&0&0&1&0&0\\ 1&0&0&0&1&0&0&1 \end{pmatrix} \approx \underbrace{\begin{pmatrix} \\ \\ X \\ \\ \\ \end{pmatrix}}_{f} \left(\quad Y \quad\right)\Big\}\, f$$

Songs

Fix songs

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

- 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
- β_u = bias for user u
- β_i = bias for item i
- λ = regularization parameter

Alternating Least Squares (ALS)

- Instead of explicit ratings use binary labels
  - 1 = streamed, 0 = never streamed
- Minimize weighted RMSE (root mean squared error) using a function of total streams as weights

$$\begin{array}{cc} \text{Users} & \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \\ & \text{Songs} \end{array} \approx \underbrace{\begin{pmatrix} \\ \\ \mathrm{X} \\ \\ \\ \end{pmatrix}}_{f} \left.\begin{pmatrix} \quad & \mathrm{Y} & \quad \end{pmatrix}\right\} f$$

Fix users

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

- 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
- β_u = bias for user u
- β_i = bias for item i
- λ = regularization parameter

Alternating Least Squares (ALS)

- Instead of explicit ratings use binary labels
  - 1 = streamed, 0 = never streamed
- Minimize weighted RMSE (root mean squared error) using a function of total streams as weights

$$\begin{array}{cc} \text{Users} & \begin{pmatrix} 1&0&0&0&1&0&0&1\\ 0&0&1&0&0&1&0&0\\ 1&0&1&0&0&0&1&1\\ 0&1&0&0&0&1&0&0\\ 0&0&1&0&0&1&0&0\\ 1&0&0&0&1&0&0&1 \end{pmatrix} \\ & \text{Songs} \end{array} \approx \underbrace{\begin{pmatrix} \\ \\ X \\ \\ \\ \end{pmatrix}}_{f} \left.\begin{pmatrix} \quad Y \quad \end{pmatrix}\right\} f$$

Fix users

Solve for songs

$$y_i = (X^T C^i X + \lambda I)^{-1} X^T C^i p(i)$$

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

- 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
- β_u = bias for user u
- β_i = bias for item i
- λ = regularization parameter

Alternating Least Squares (ALS)

- Instead of explicit ratings use binary labels
  - 1 = streamed, 0 = never streamed
- Minimize weighted RMSE (root mean squared error) using a function of total streams as weights

Repeat until convergence…

$$\text{Users}\ \underset{\text{Songs}}{\begin{pmatrix} 1&0&0&0&1&0&0&1\\ 0&0&1&0&0&1&0&0\\ 1&0&1&0&0&0&1&1\\ 0&1&0&0&0&1&0&0\\ 0&0&1&0&0&1&0&0\\ 1&0&0&0&1&0&0&1 \end{pmatrix}} \approx \underbrace{\begin{pmatrix} \\ \\ X \\ \\ \\ \end{pmatrix}}_{f} \left.\begin{pmatrix} \quad Y \quad \end{pmatrix}\right\} f$$

Solve for songs

Fix users

$$y_i = (X^T C^i X + \lambda I)^{-1} X^T C^i p(i)$$

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

- 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
- β_u = bias for user u
- β_i = bias for item i
- λ = regularization parameter

Alternating Least Squares (ALS)

- Instead of explicit ratings use binary labels
  - 1 = streamed, 0 = never streamed
- Minimize weighted RMSE (root mean squared error) using a function of total streams as weights

Repeat until convergence…

$$\text{Users}\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \approx \underbrace{\begin{pmatrix} \\ \\ X \\ \\ \\ \end{pmatrix}}_{f} \left.\begin{pmatrix} \quad Y \quad \end{pmatrix}\right\} f$$

Songs

Solve for songs

Fix users

$$y_i = (X^T C^i X + \lambda I)^{-1} X^T C^i p(i)$$

$$y_i = (X^T X + X^T (C^i - I) X + \lambda I)^{-1} X^T C^i p(i)$$

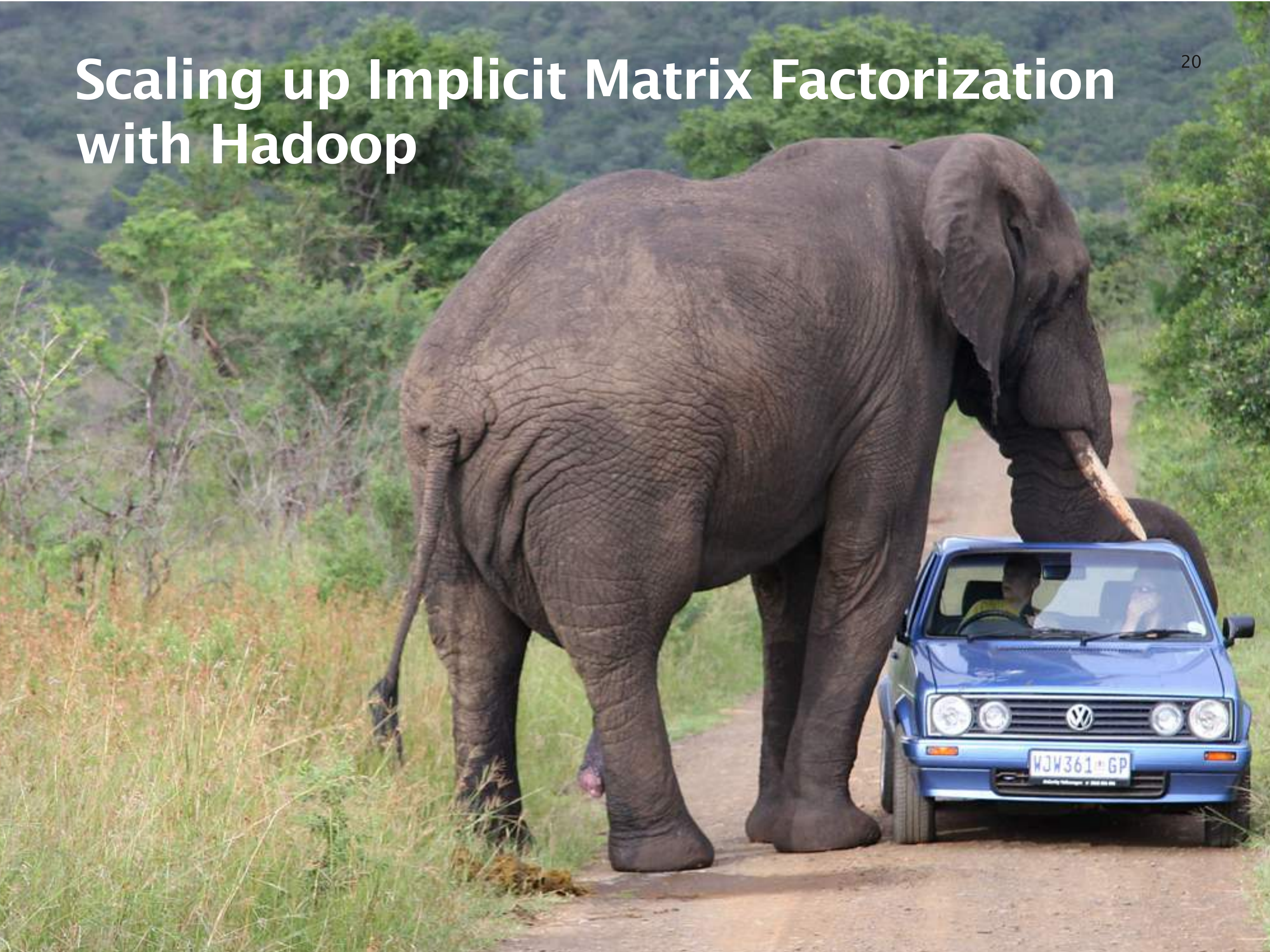
- p_{ui} = 1 if user u streamed track i else 0
- $c_{ui} = 1 + \alpha r_{ui}$
- x_u = user $u's$ latent factor vector
- x_i =i tem $i's$ latent factor vector
- β_u = bias for user u
- β_i = bias for item i
- λ = regularization parameter

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

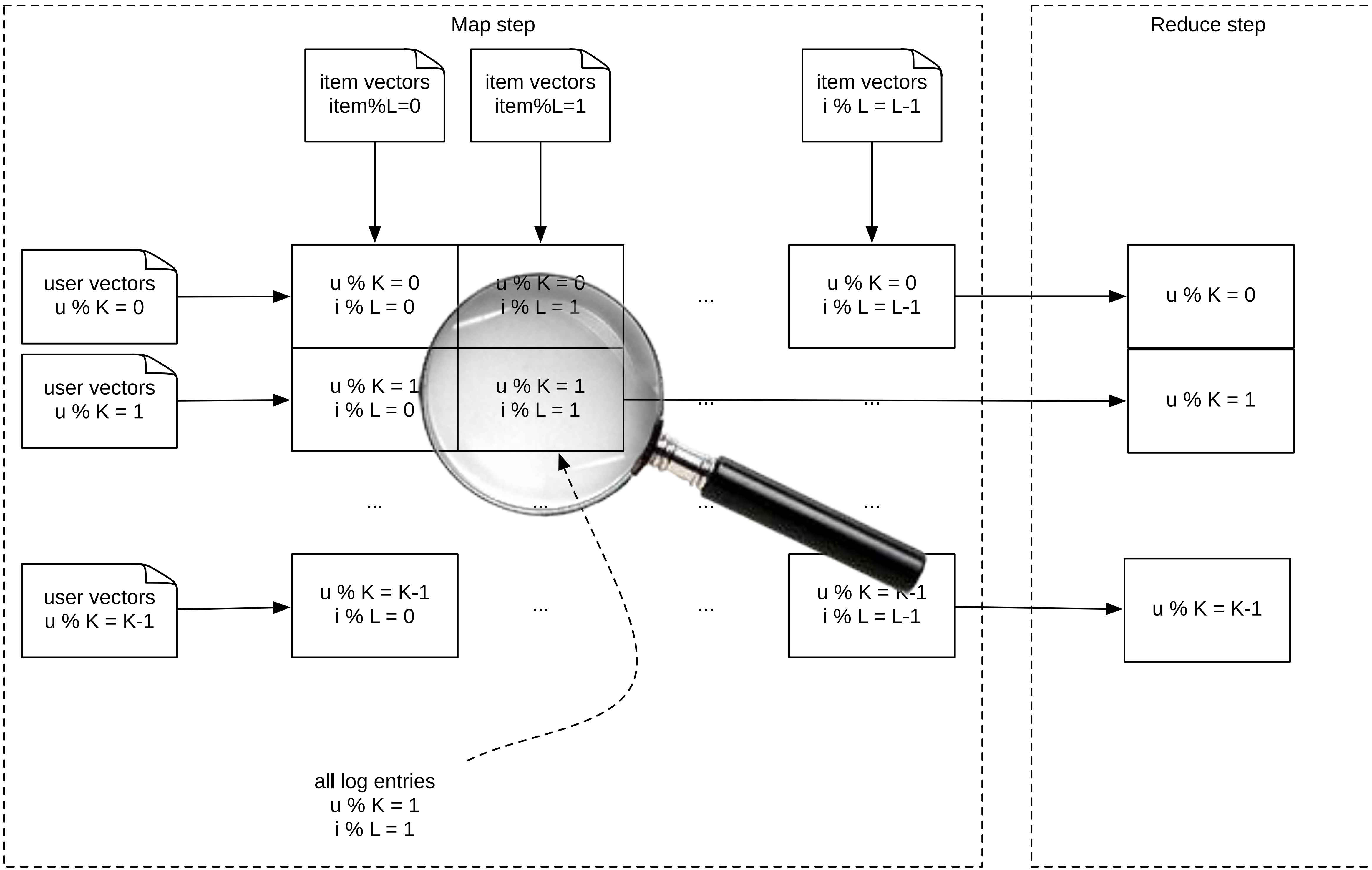


Figure via Erik Bernhardsson

Implicit Matrix Factorization with Hadoop

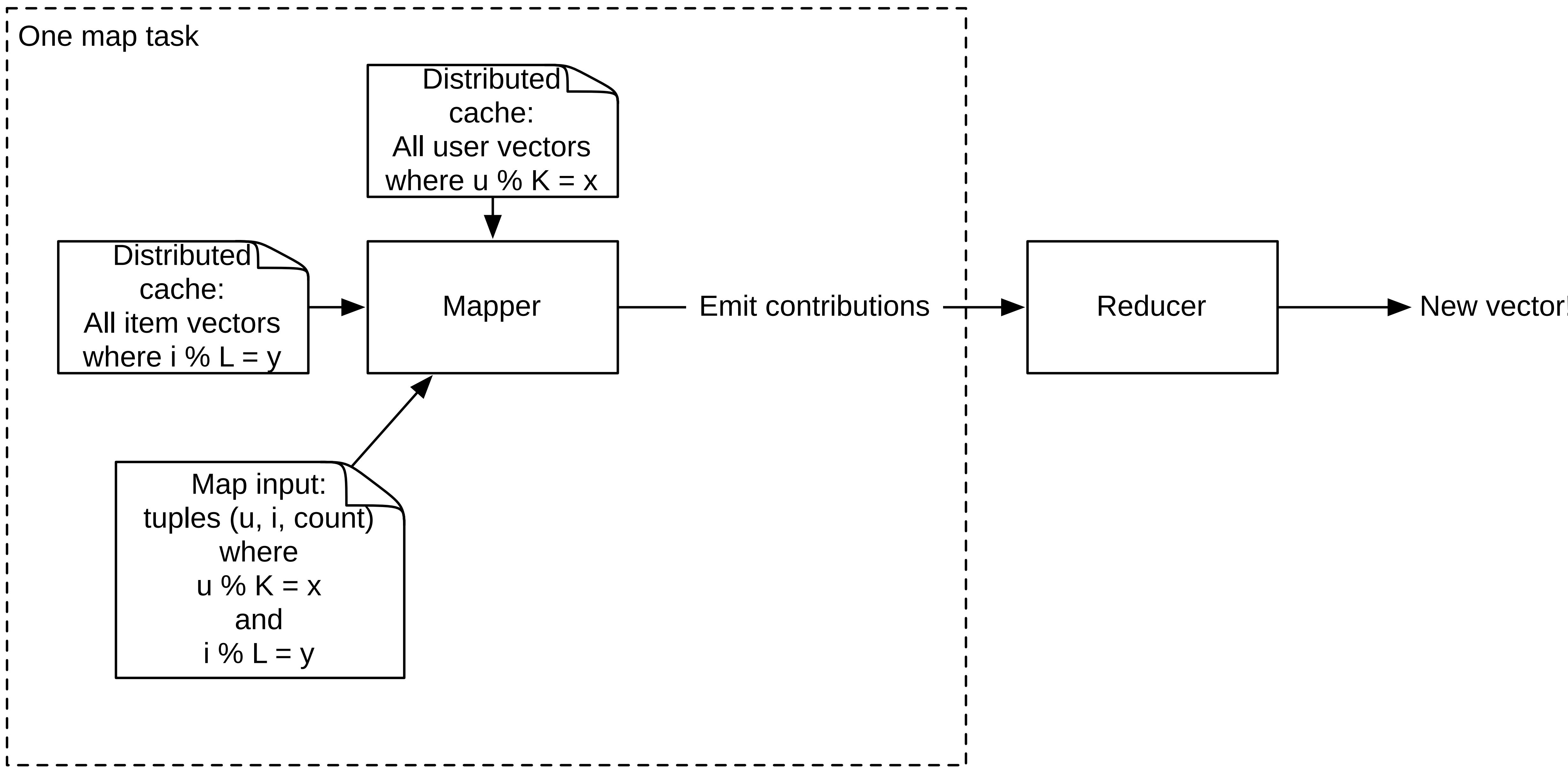
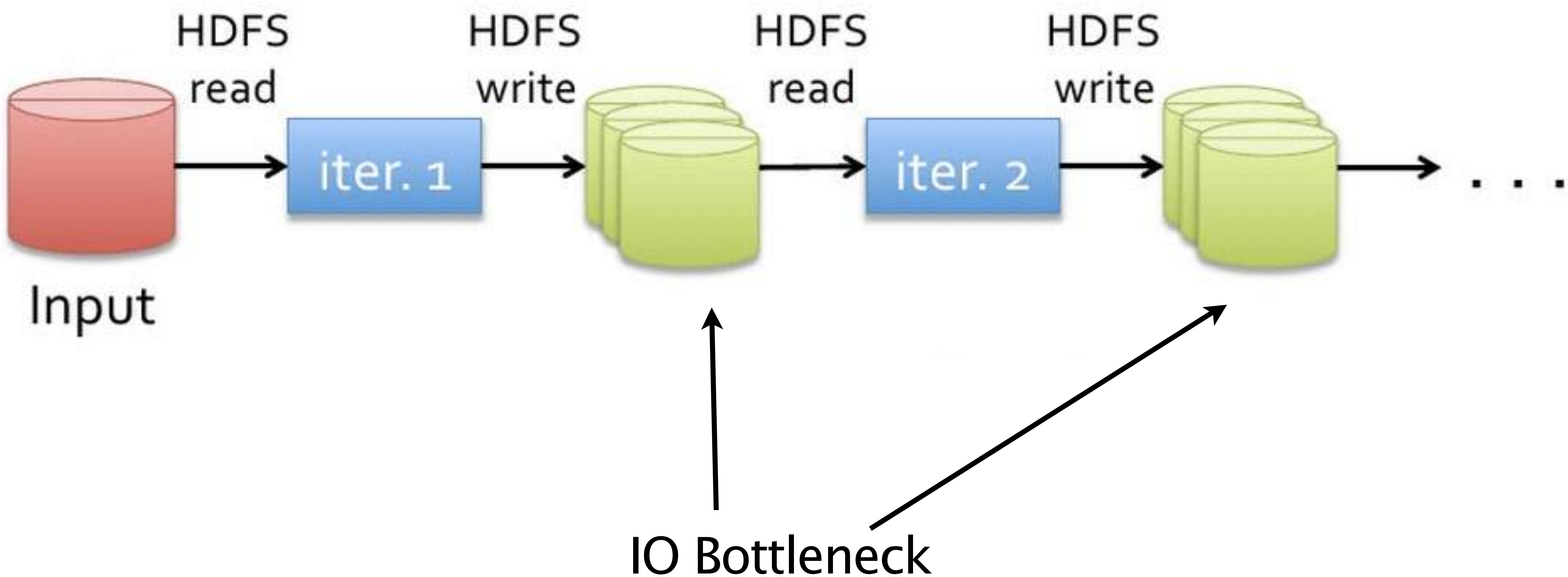
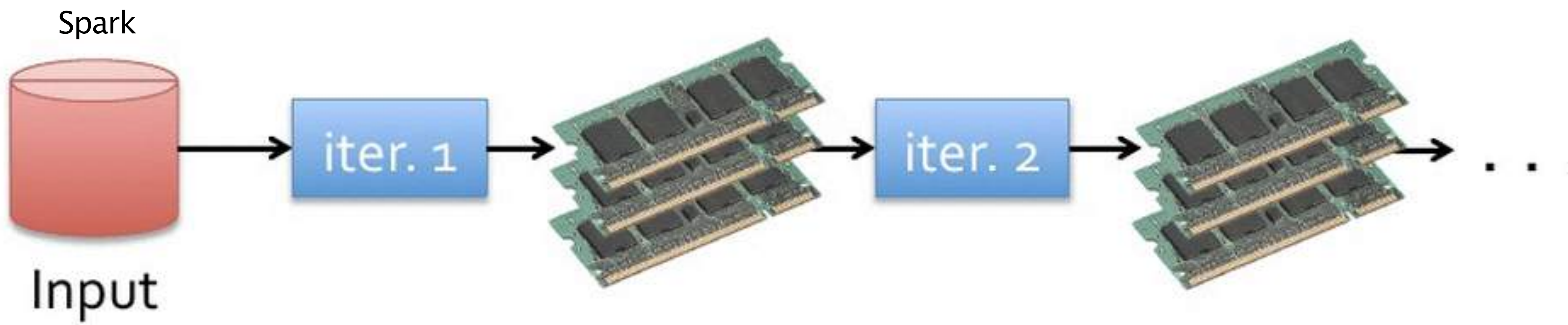


Figure via Erik Bernhardsson

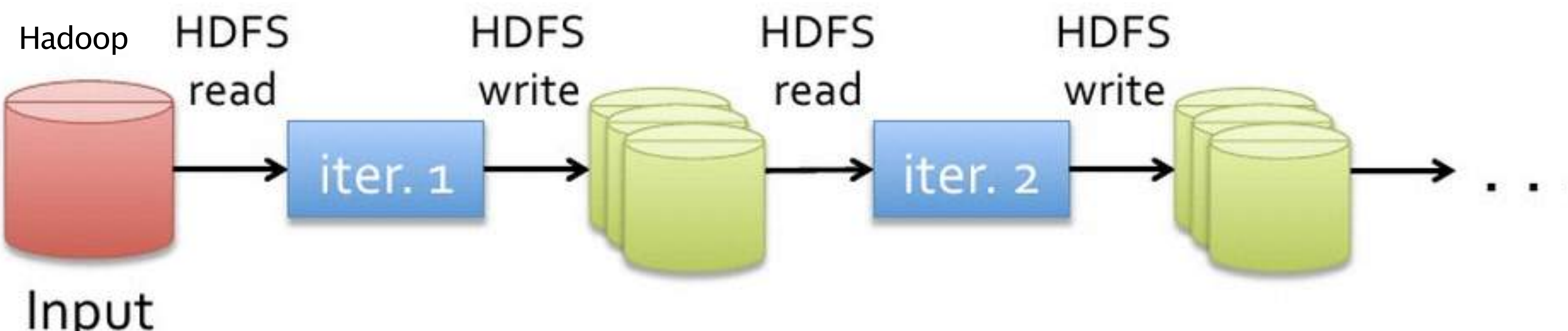
Hadoop suffers from I/O overhead



Spark to the rescue!!



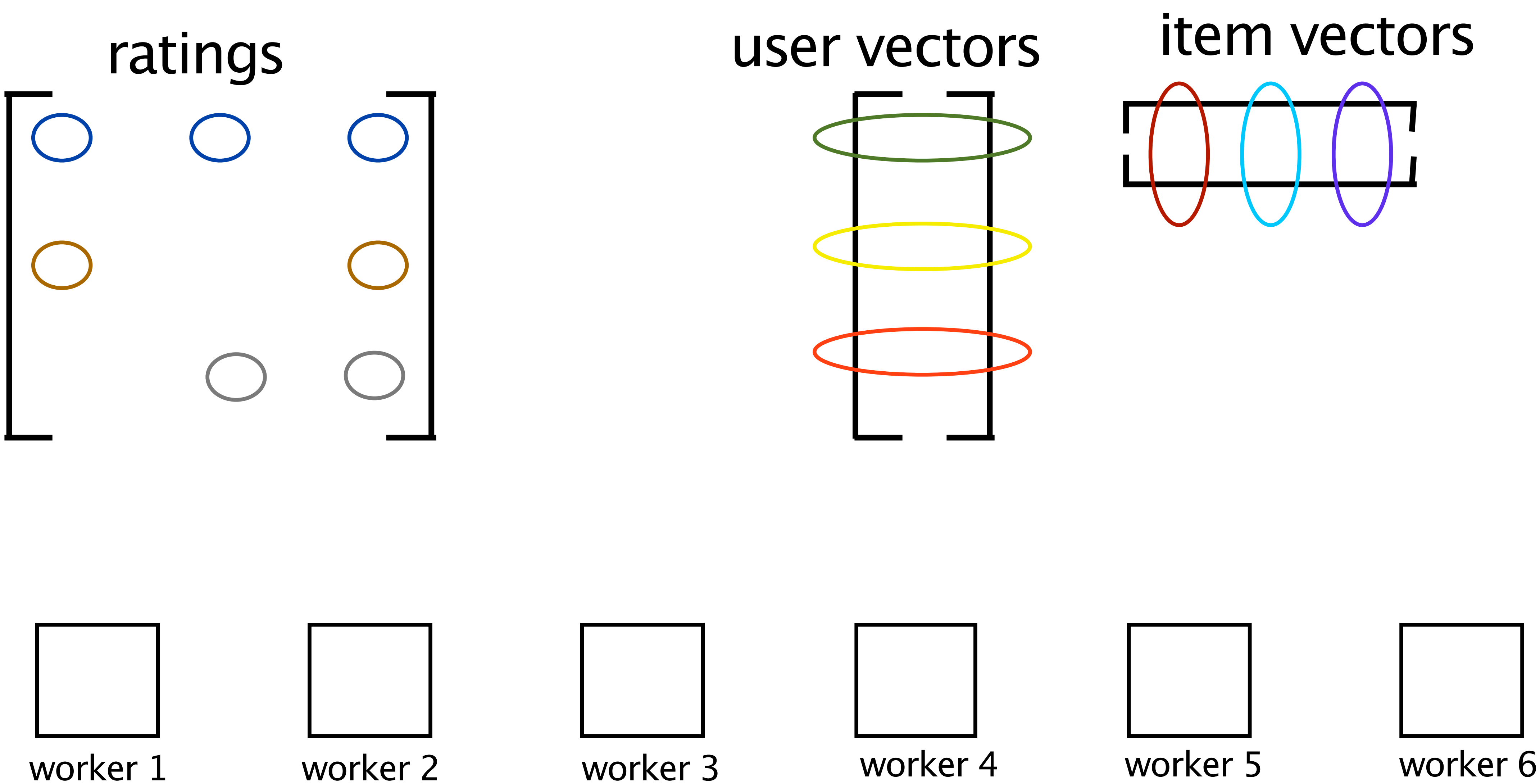
Vs



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

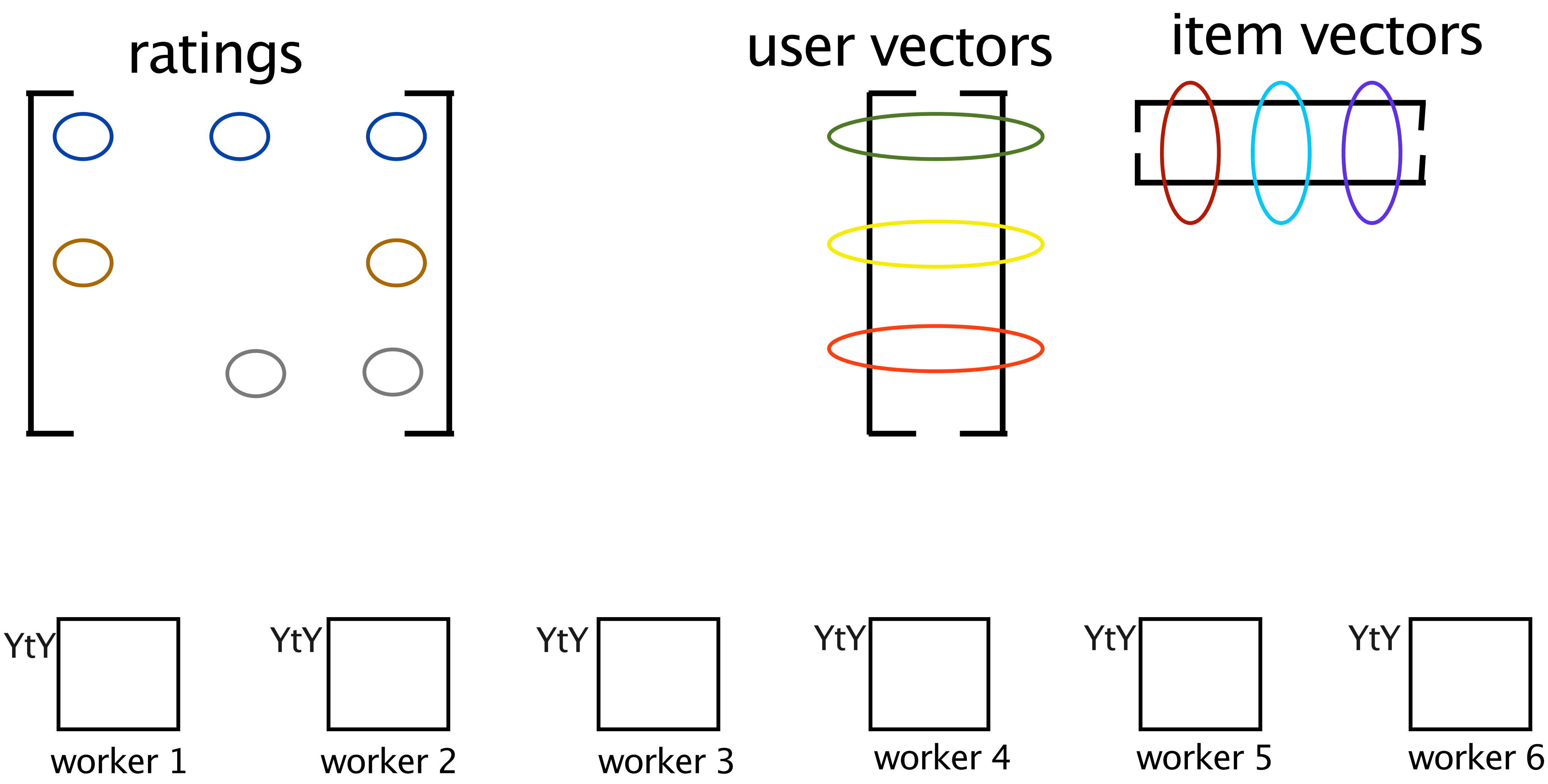
First Attempt (broadcast everything)

- For each iteration:
  1. Compute YtY over item vectors and broadcast
  2. Broadcast item vectors
  3. Group ratings by user
  4. Solve for optimal user vector



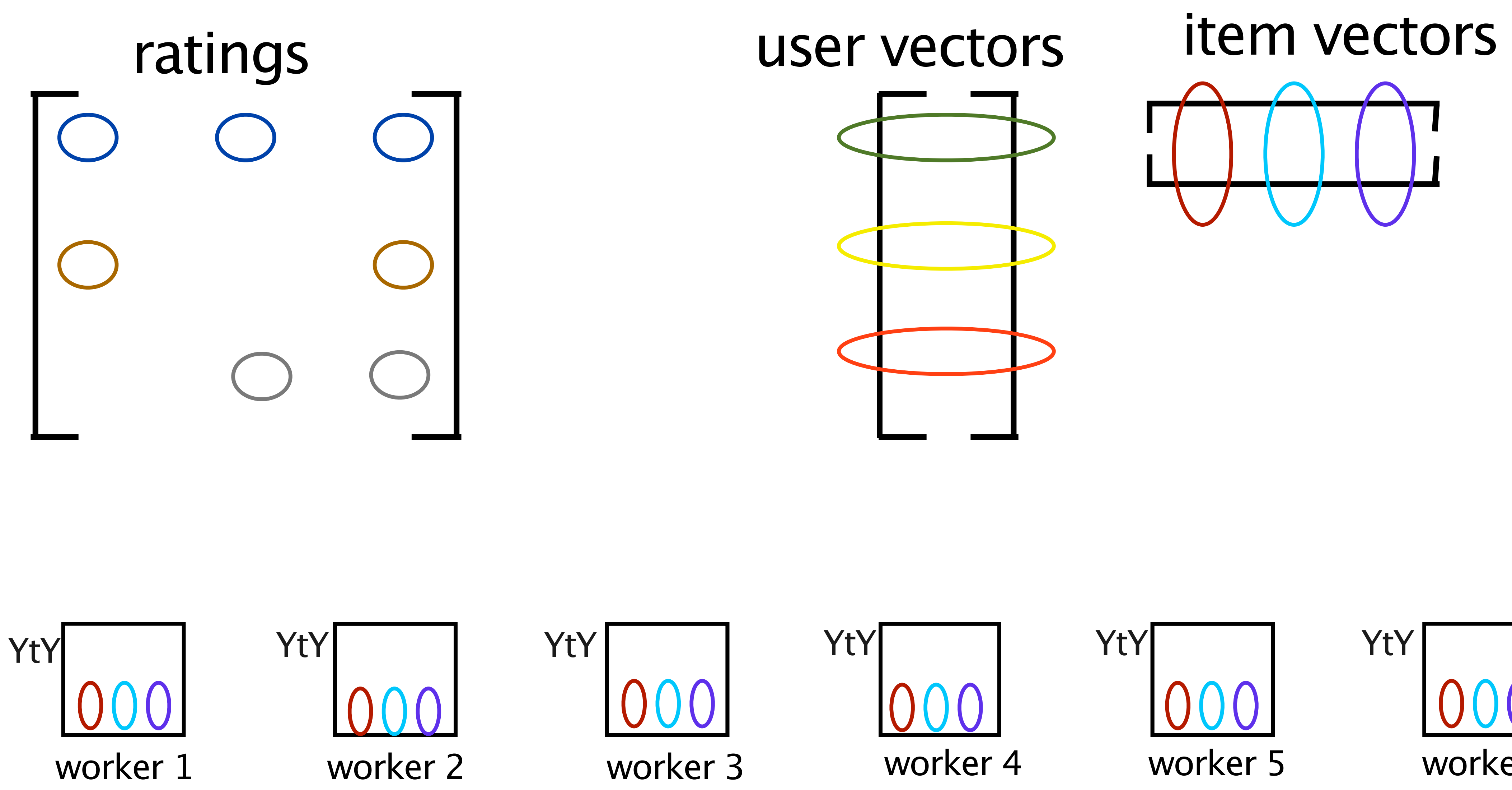
First Attempt (broadcast everything)

- For each iteration:
  1. Compute YtY over item vectors and broadcast
  2. Broadcast item vectors
  3. Group ratings by user
  4. Solve for optimal user vector



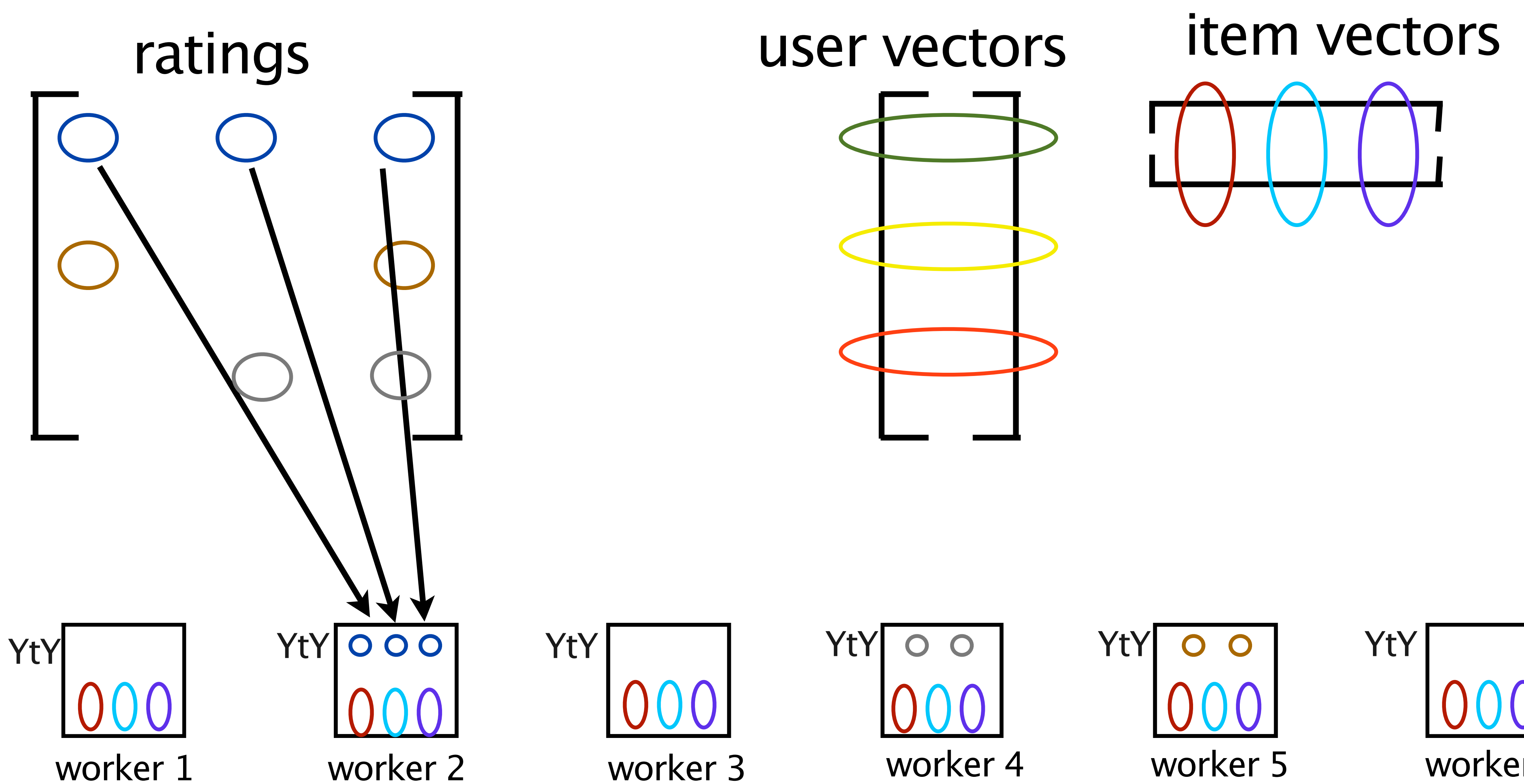
First Attempt (broadcast everything)

- For each iteration:
  1. Compute YtY over item vectors and broadcast
  2. Broadcast item vectors
  3. Group ratings by user
  4. Solve for optimal user vector



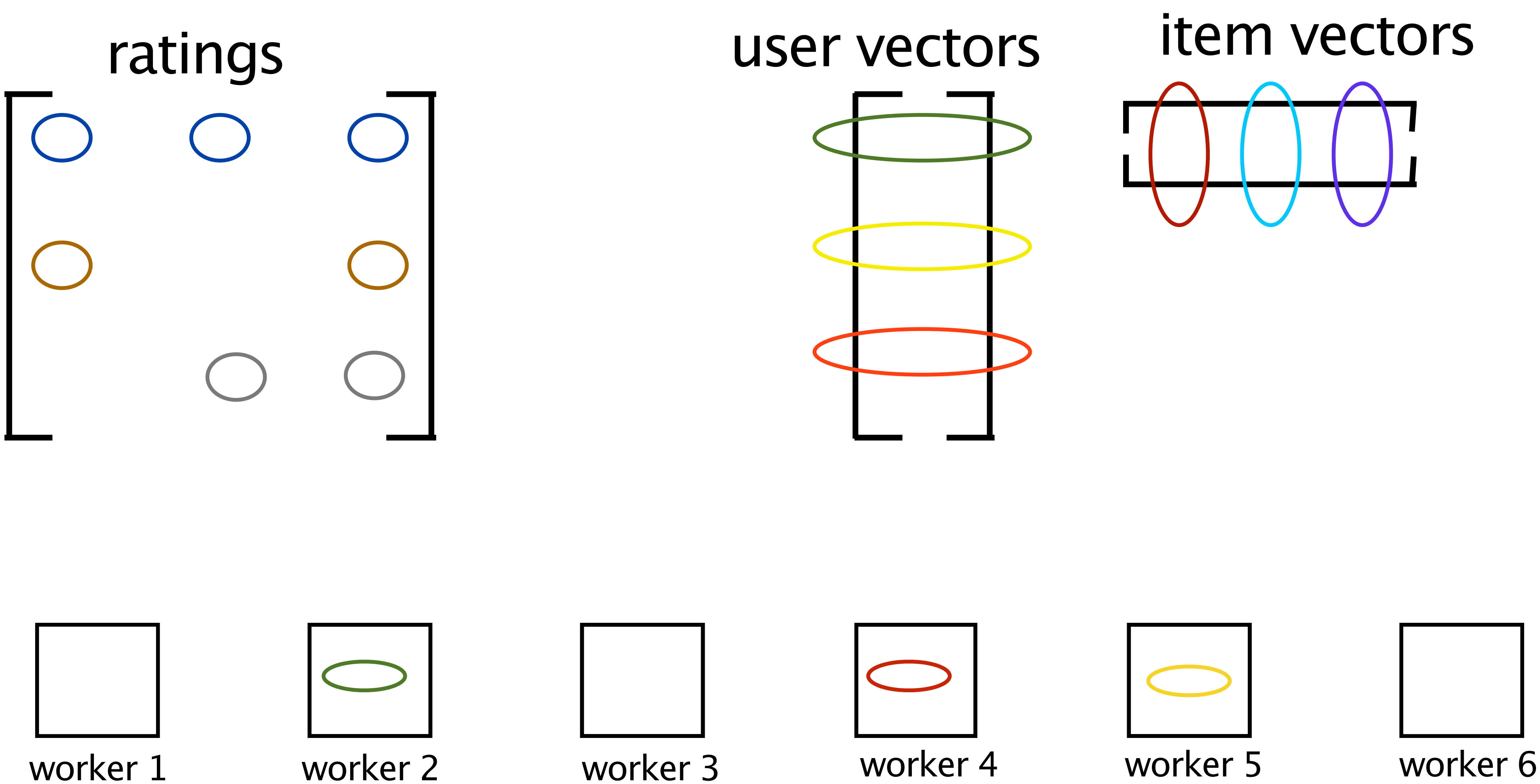
First Attempt (broadcast everything)

- For each iteration:
  1. Compute YtY over item vectors and broadcast
  2. Broadcast item vectors
  3. Group ratings by user
  4. Solve for optimal user vector



First Attempt (broadcast everything)

- For each iteration:
  1. Compute YtY over item vectors and broadcast
  2. Broadcast item vectors
  3. Group ratings by user
  4. Solve for optimal user vector



First Attempt (broadcast everything)

```scala
def ALSIteration(ratings: RDD[(Int, Int, Double)],
                 users: RDD[(Int, DenseVector[Double])],
                 items: RDD[(Int, DenseVector[Double])]) = {

  val YtY: Broadcast[DenseMatrix[Double]] = sc.broadcast(
    items
    .map{case(item: Int, vector: DenseVector[Double]) =>
      vector * vector.t
    }.reduce{(m1: DenseMatrix[Double], m2: DenseMatrix[Double]) =>
      m1 + m2}
  )

  val itemMap = sc.broadcast(
    items
    .toLocalIterator
    .toMap
  )

  ratings
    .map{case(user: Int, item: Int, rating: Double) =>
      (user, (item, rating))}
    .groupByKey
    .map{case(user: Int, ratings: Iterable[(Int, Double)]) =>
      solveVectors(user, ratings, itemMap, YtY)
    }
}
```

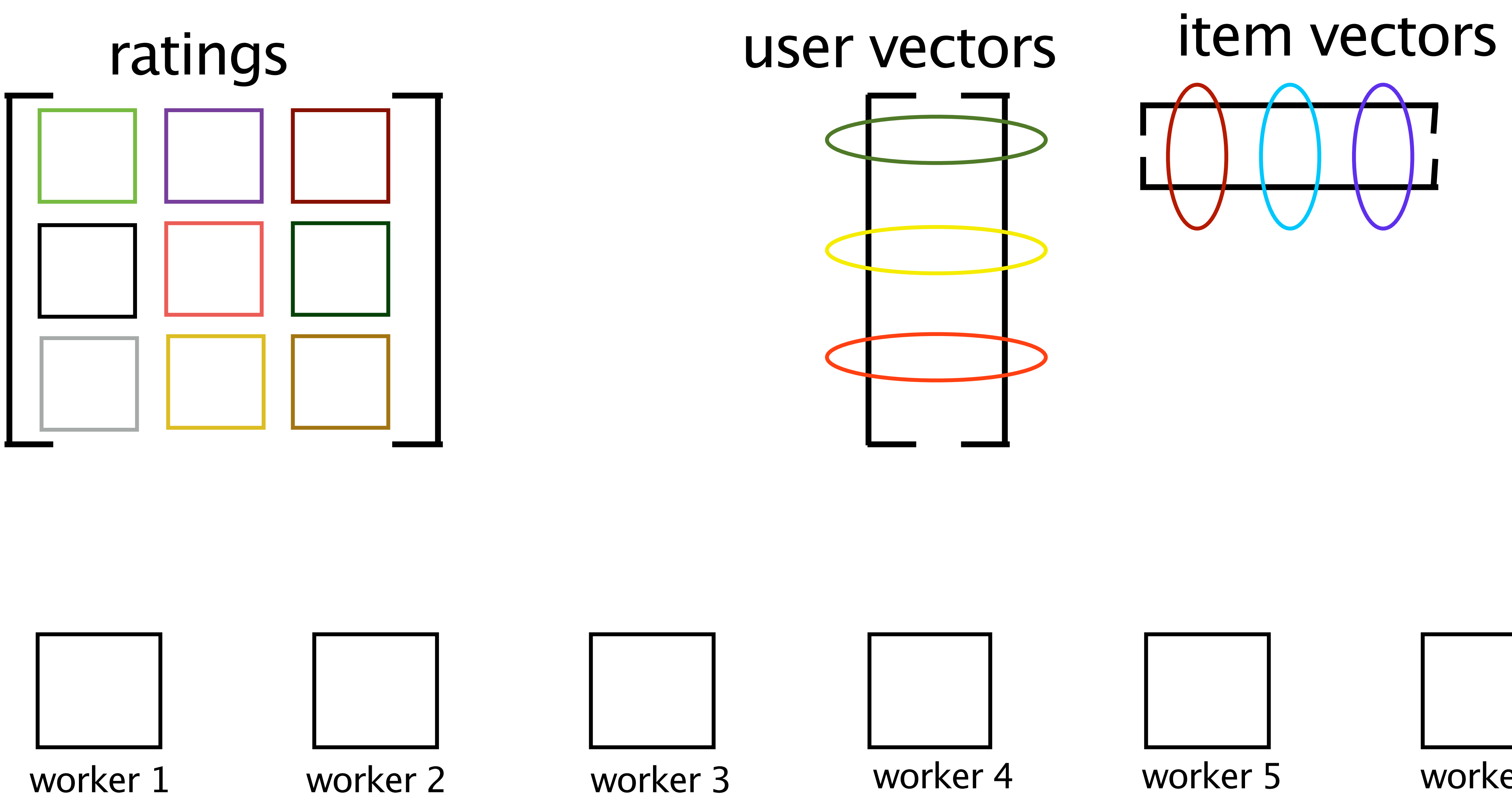
First Attempt (broadcast everything)

```scala
def ALSIteration(ratings: RDD[(Int, Int, Double)],
                 users: RDD[(Int, DenseVector[Double])],
                 items: RDD[(Int, DenseVector[Double])]) = {

  val YtY: Broadcast[DenseMatrix[Double]] = sc.broadcast(
    items
    .map{case(item: Int, vector: DenseVector[Double]) =>
      vector * vector.t
    }.reduce{(m1: DenseMatrix[Double], m2: DenseMatrix[Double]) =>
      m1 + m2}
  )

  val itemMap = sc.broadcast(
    items
    .toLocalIterator
    .toMap
  )

  ratings
    .map{case(user: Int, item: Int, rating: Double) =>
      (user, (item, rating))}
    .groupByKey
    .map{case(user: Int, ratings: Iterable[(Int, Double)]) =>
      solveVectors(user, ratings, itemMap, YtY)
    }
}
```

- Cons:
  - Unnecessarily shuffling all data across wire each iteration.
  - Not caching ratings data
  - Unnecessarily sending a full copy of user/item vectors to all workers.

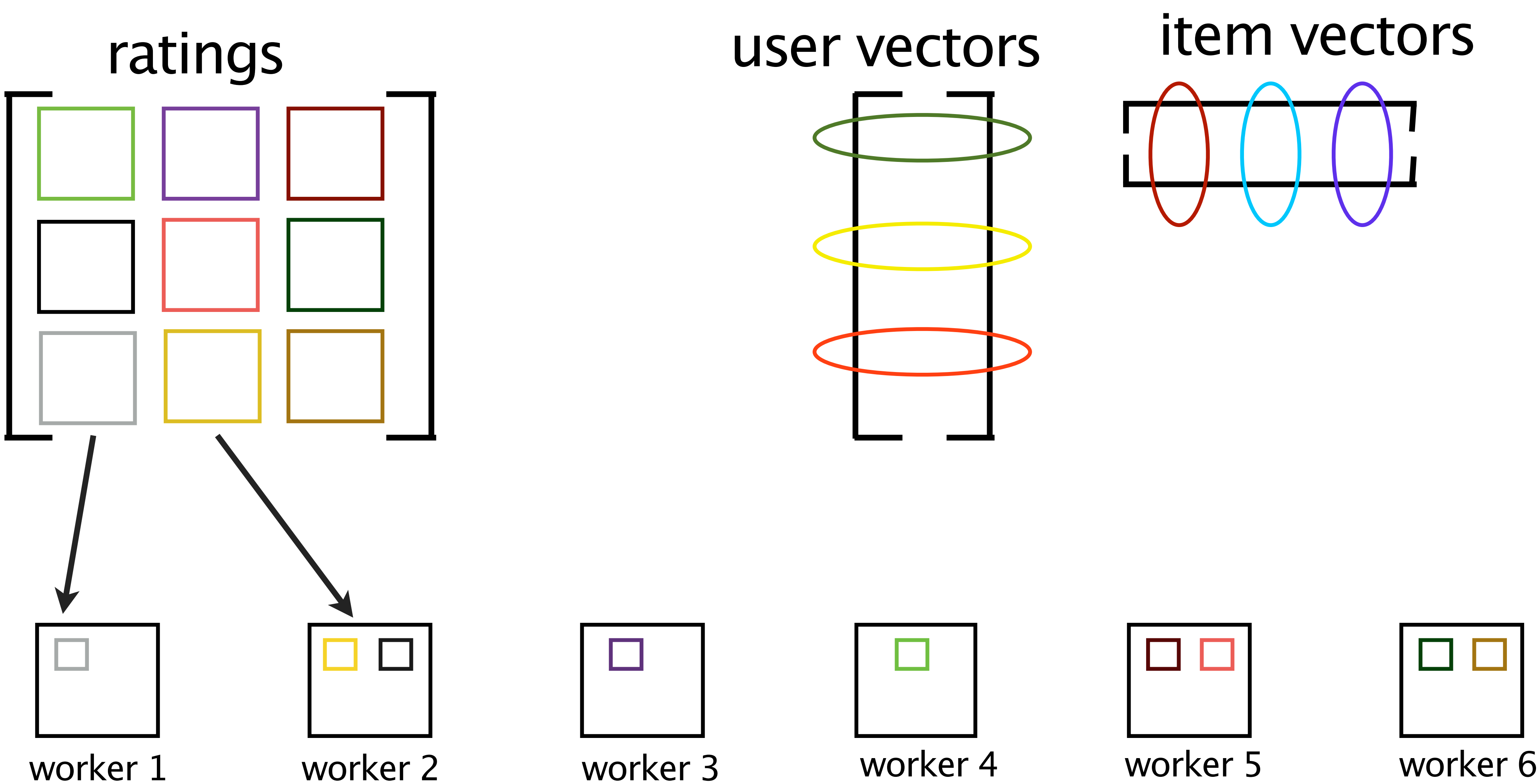
Second Attempt (full gridify)

- Group ratings matrix into K x L, partition, and cache
- For each iteration:
  1. Compute YtY over item vectors and broadcast
  2. For each item vector send a copy to each rating block in the item % L column
  3. Compute intermediate terms for each block (partition)
  4. Group by user, aggregate intermediate terms, and solve for optimal user vector



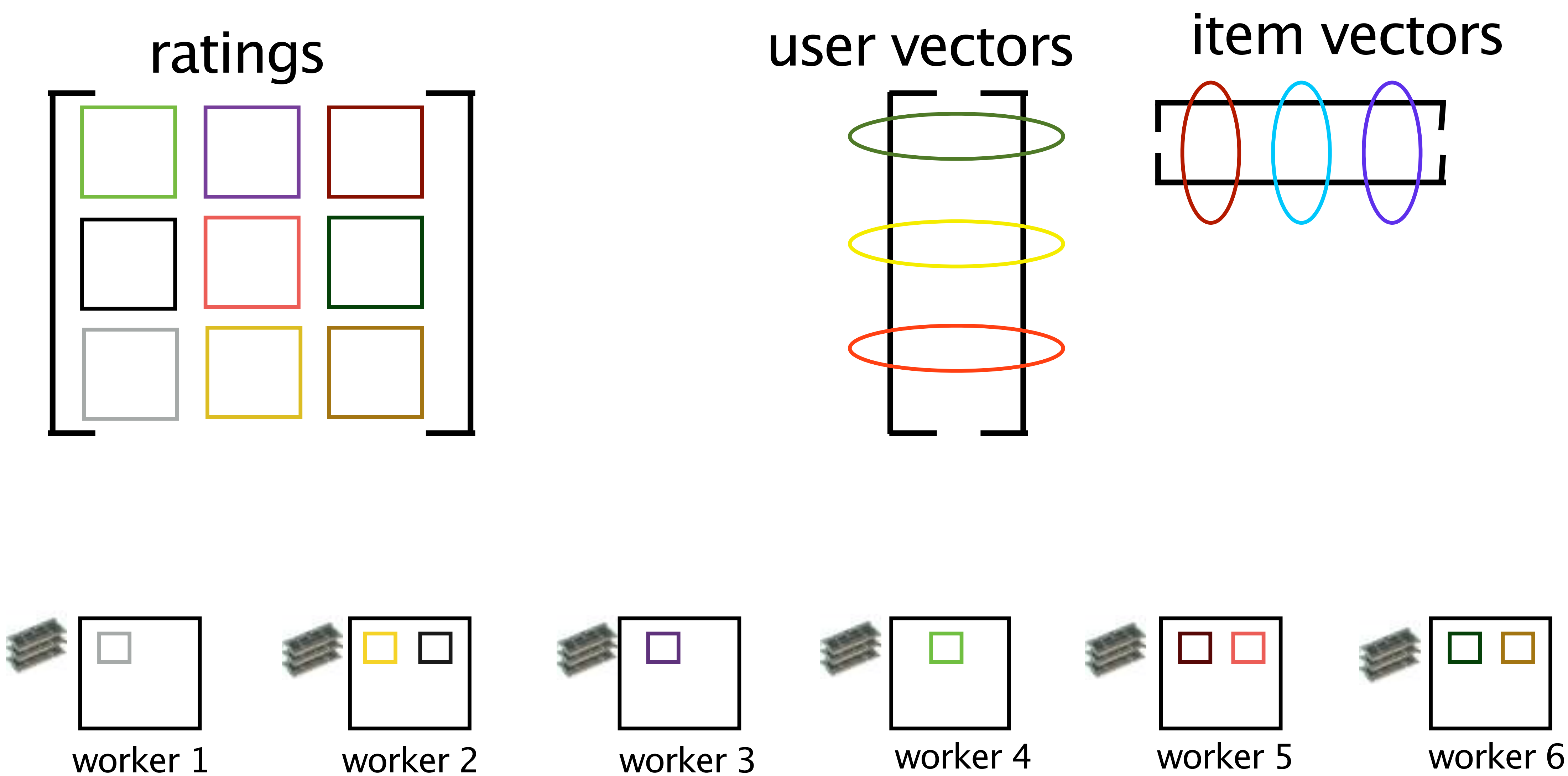
Second Attempt (full gridify)

- Group ratings matrix into K x L, partition, and cache
- For each iteration:
  1. Compute YtY over item vectors and broadcast
  2. For each item vector send a copy to each rating block in the item % L column
  3. Compute intermediate terms for each block (partition)
  4. Group by user, aggregate intermediate terms, and solve for optimal user vector



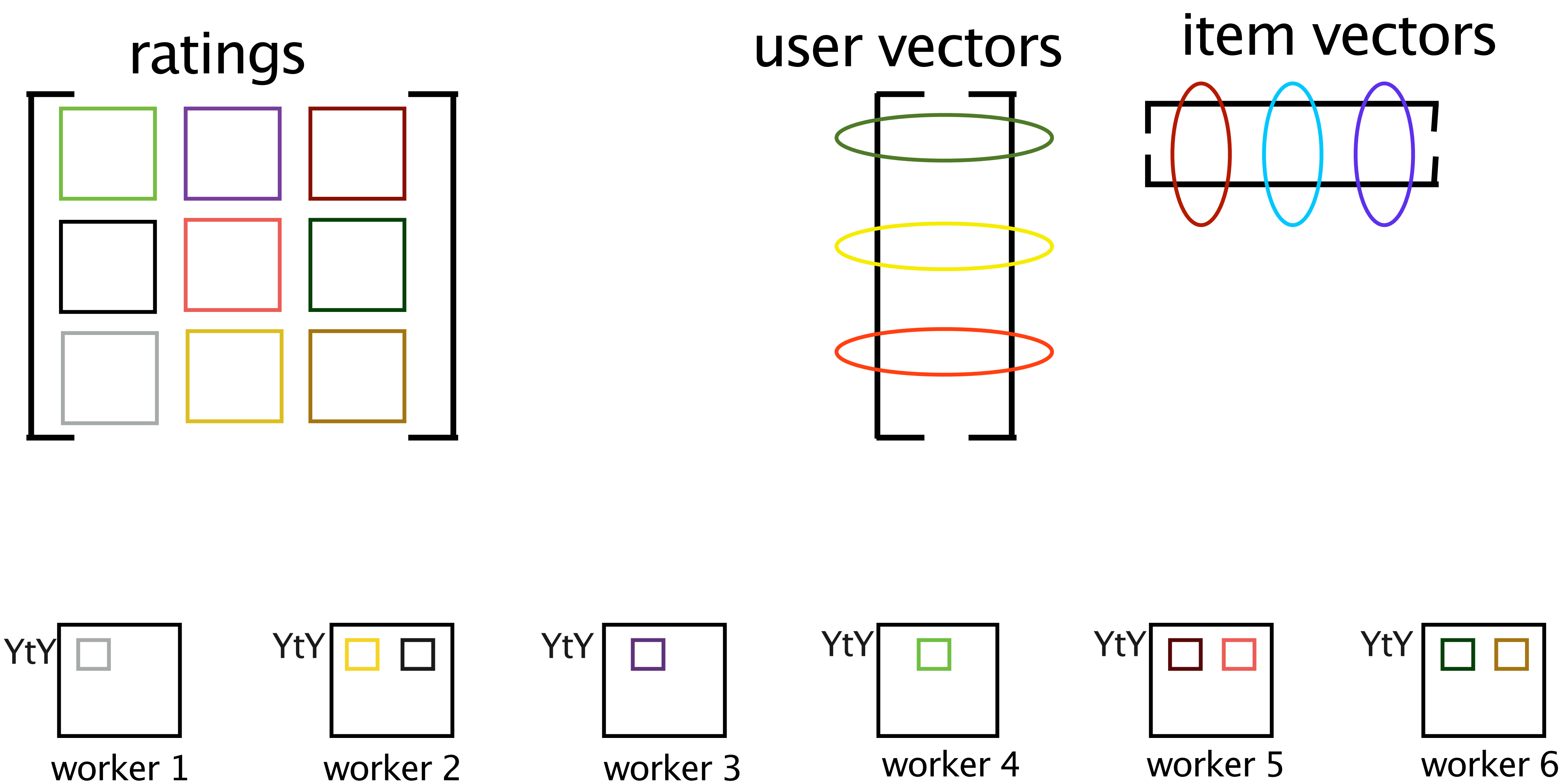
Second Attempt (full gridify)

- Group ratings matrix into K x L, partition, and cache
- For each iteration:
  1. Compute YtY over item vectors and broadcast
  2. For each item vector send a copy to each rating block in the item % L column
  3. Compute intermediate terms for each block (partition)
  4. Group by user, aggregate intermediate terms, and solve for optimal user vector



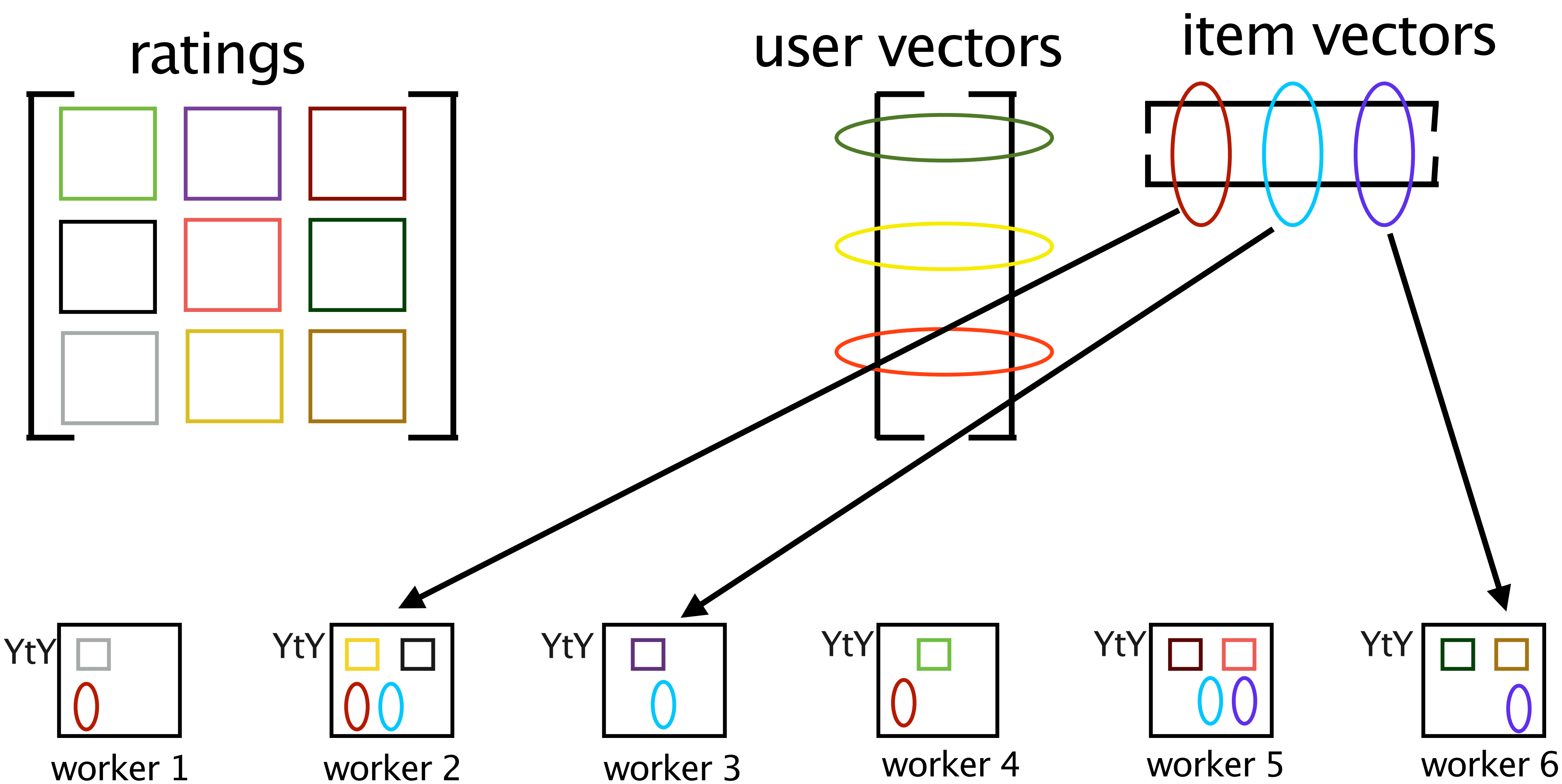
Second Attempt (full gridify)

- Group ratings matrix into K x L, partition, and cache
- For each iteration:
  1. Compute YtY over item vectors and broadcast
  2. For each item vector send a copy to each rating block in the item % L column
  3. Compute intermediate terms for each block (partition)
  4. Group by user, aggregate intermediate terms, and solve for optimal user vector



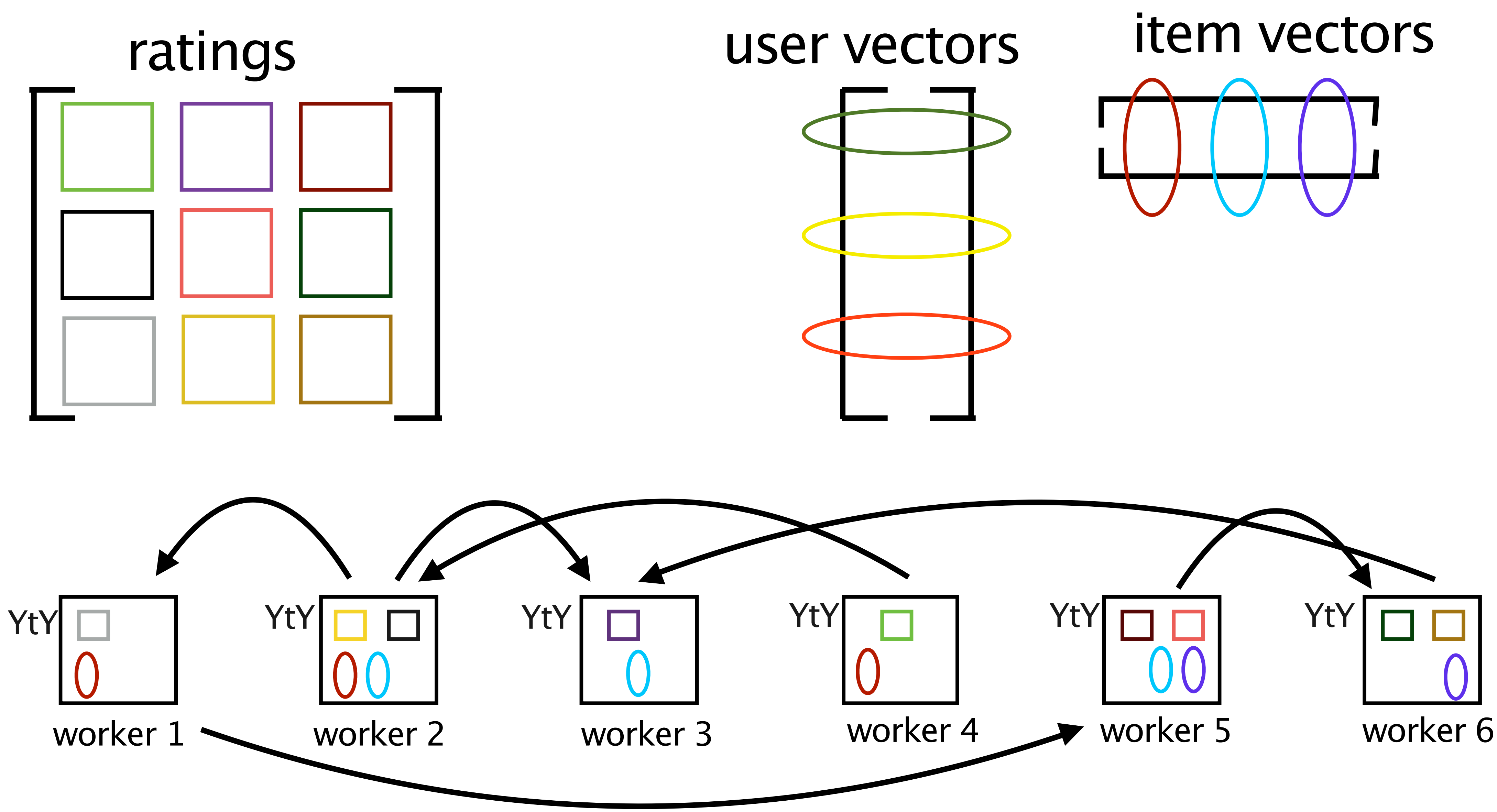
Second Attempt (full gridify)

- Group ratings matrix into K x L, partition, and cache
- For each iteration:
  1. Compute YtY over item vectors and broadcast
  2. For each item vector send a copy to each rating block in the item % L column
  3. Compute intermediate terms for each block (partition)
  4. Group by user, aggregate intermediate terms, and solve for optimal user vector



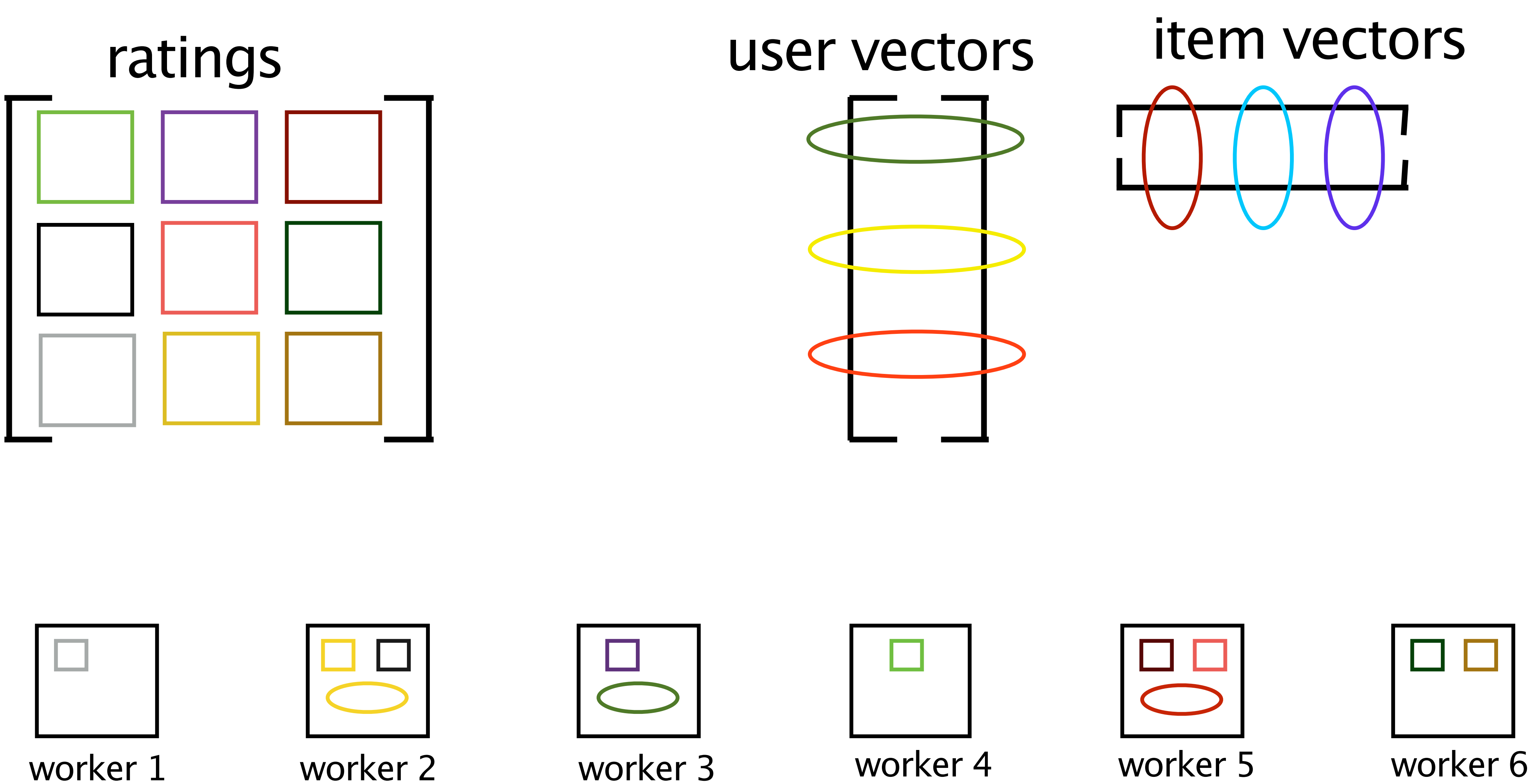
Second Attempt (full gridify)

- Group ratings matrix into K x L, partition, and cache
- For each iteration:
  1. Compute YtY over item vectors and broadcast
  2. For each item vector send a copy to each rating block in the item % L column
  3. Compute intermediate terms for each block (partition)
  4. Group by user, aggregate intermediate terms, and solve for optimal user vector



Second Attempt (full gridify)

- Group ratings matrix into K x L, partition, and cache
- For each iteration:
  1. Compute YtY over item vectors and broadcast
  2. For each item vector send a copy to each rating block in the item % L column
  3. Compute intermediate terms for each block (partition)
  4. Group by user, aggregate intermediate terms, and solve for optimal user vector



Second Attempt

```scala
def fullGridify(ratings: RDD[Rating],
                k: Int,
                l: Int,
                partitioner: Partitioner)= {

  ratings
    .map{r: Rating =>
      val row = r.user % k
      val column = r.item % l
      (((row * l) + column), r)
    }.groupByKey(partitioner)
    .mapValues{itr: Iterable[Rating] =>
      itr.toList.groupBy(_.user)
    }.persist(StorageLevel.MEMORY_AND_DISK)
}

def updateVectors(ratingsByBlock: RDD[(Int, Map[Int, List[Rating]])],
                  itemsByBlock: RDD[(Int, Map[Int, VectorData])],
                  partitioner: Partitioner) = {

  val yty: DenseMatrix[Double] = computeYtY(itemsByBlock)

  val joinedVectorsRatings = joinVectorsRatings(ratingsByBlock, itemsByBlock, k, l, user, partitioner)

  val aggregatedTerms: RDD[(Int, Iterable[(Int, DenseMatrix[Double], DenseVector[Double])])] =
    aggregateTerms(joinedVectorsRatings, alpha, k, user, rank, partitioner)

  solveVectors(aggregatedTerms, yty, lambda, user)
}
```

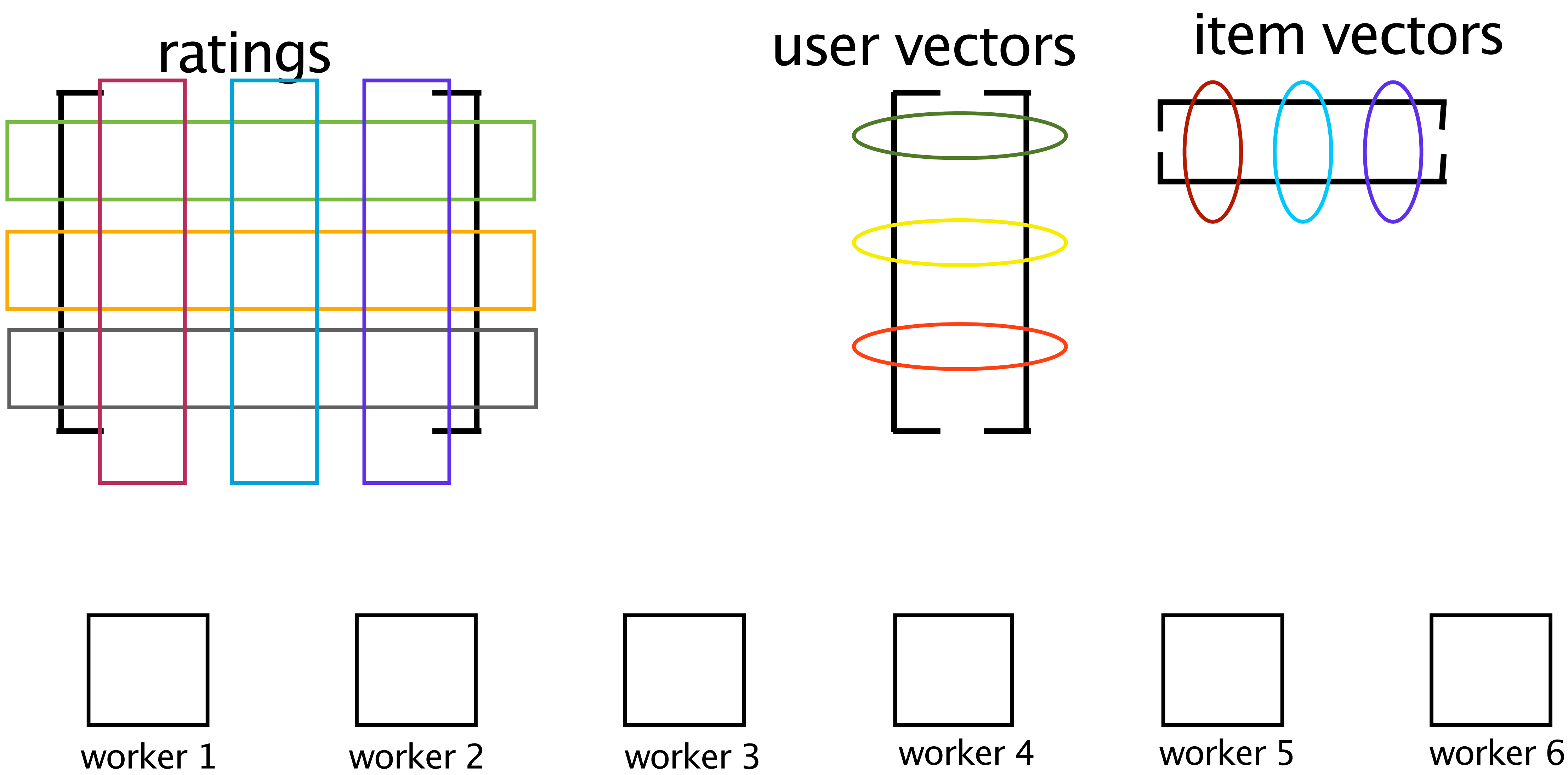
Second Attempt

```scala
def fullGridify(ratings: RDD[Rating],
                k: Int,
                l: Int,
                partitioner: Partitioner)= {

  ratings
    .map{r: Rating =>
      val row = r.user % k
      val column = r.item % l
      (((row * l) + column), r)
    }.groupByKey(partitioner)
    .mapValues{itr: Iterable[Rating] =>
      itr.toList.groupBy(_.user)
    }.persist(StorageLevel.MEMORY_AND_DISK)
}

def updateVectors(ratingsByBlock: RDD[(Int, Map[Int, List[Rating]])],
                  itemsByBlock: RDD[(Int, Map[Int, VectorData])],
                  partitioner: Partitioner) = {

  val yty: DenseMatrix[Double] = computeYtY(itemsByBlock)

  val joinedVectorsRatings = joinVectorsRatings(ratingsByBlock, itemsByBlock, k, l, user, partitioner)

  val aggregatedTerms: RDD[(Int, Iterable[(Int, DenseMatrix[Double], DenseVector[Double])])] =
    aggregateTerms(joinedVectorsRatings, alpha, k, user, rank, partitioner)

  solveVectors(aggregatedTerms, yty, lambda, user)
}
```

- Pros
  - Ratings get cached and never shuffled
  - Each partition only requires a subset of item (or user) vectors in memory each iteration
  - Potentially requires less local memory than a “half gridify” scheme
- Cons
  - Sending lots of intermediate data over wire each iteration in order to aggregate and solve for optimal vectors
  - More IO overhead than a “half gridify” scheme

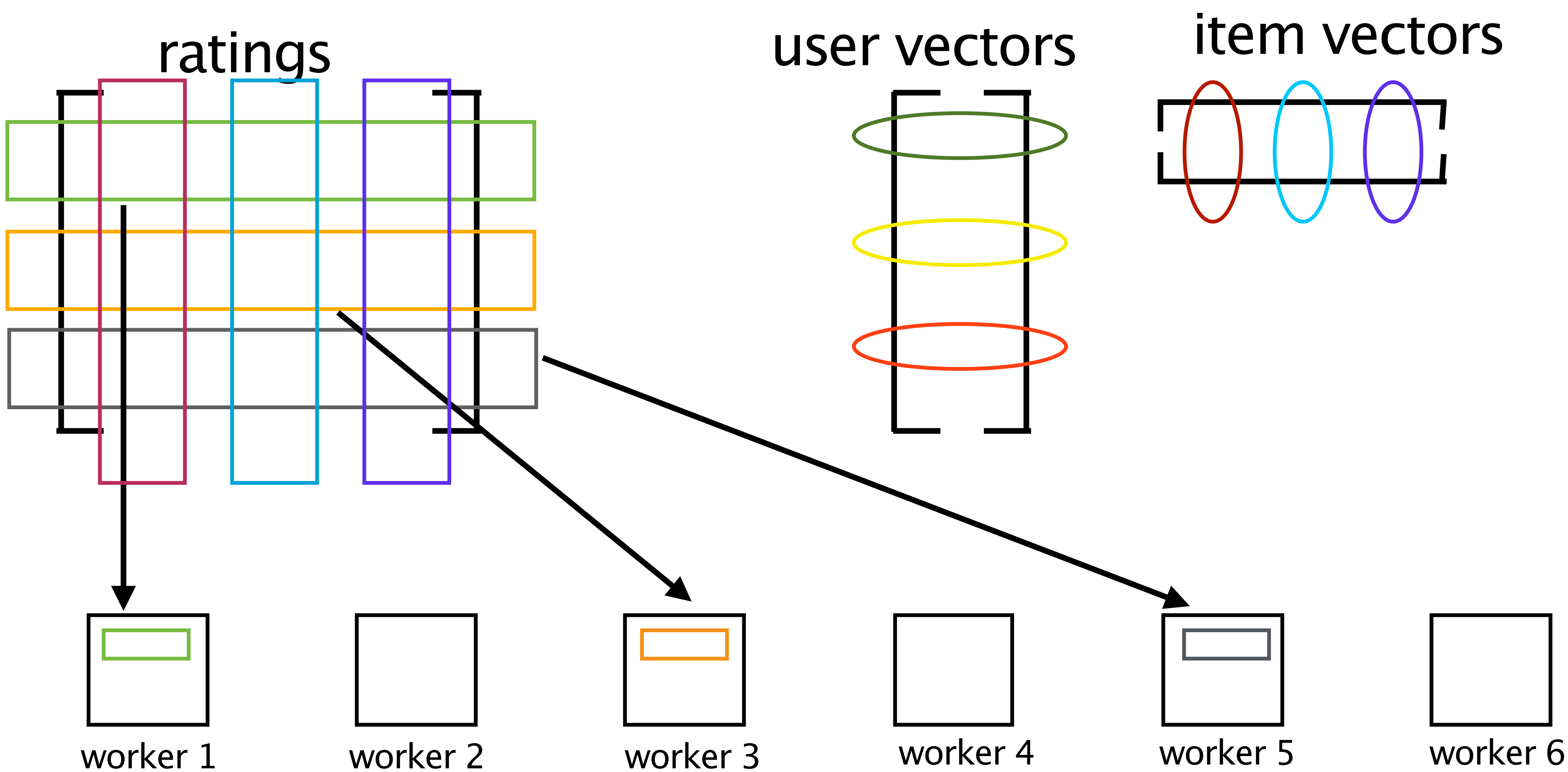
Third Attempt (half gridify)

- Partition ratings matrix into K user (row) and item (column) blocks, partition, and cache
- For each iteration:
  1. Compute YtY over item vectors and broadcast
  2. For each item vector, send a copy to each user rating partition that requires it (potentially all partitions)
  3. Each partition aggregates intermediate terms and solves for optimal user vectors



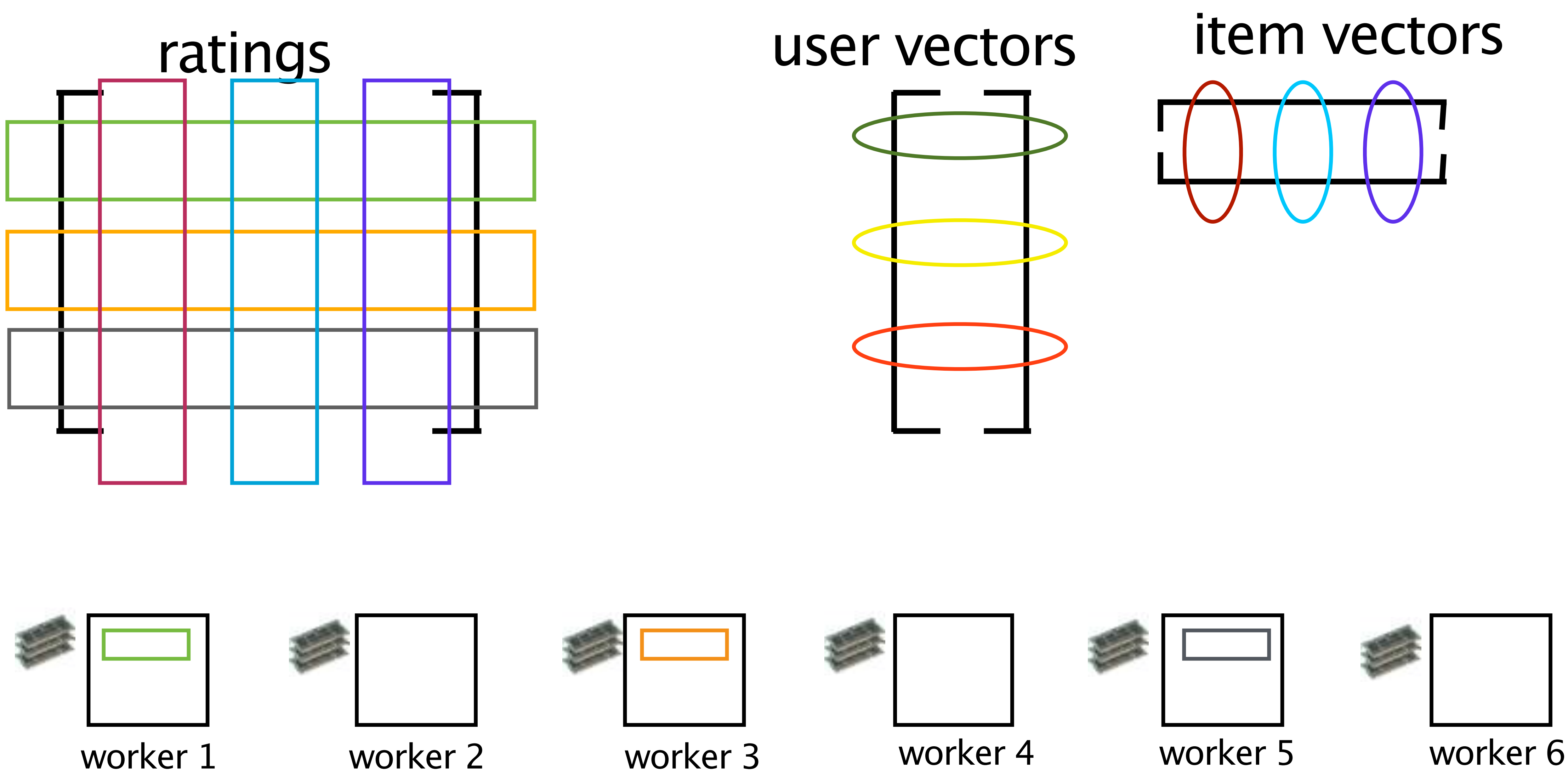
Third Attempt (half gridify)

- Partition ratings matrix into K user (row) and item (column) blocks, partition, and cache
- For each iteration:
  1. Compute YtY over item vectors and broadcast
  2. For each item vector, send a copy to each user rating partition that requires it (potentially all partitions)
  3. Each partition aggregates intermediate terms and solves for optimal user vectors



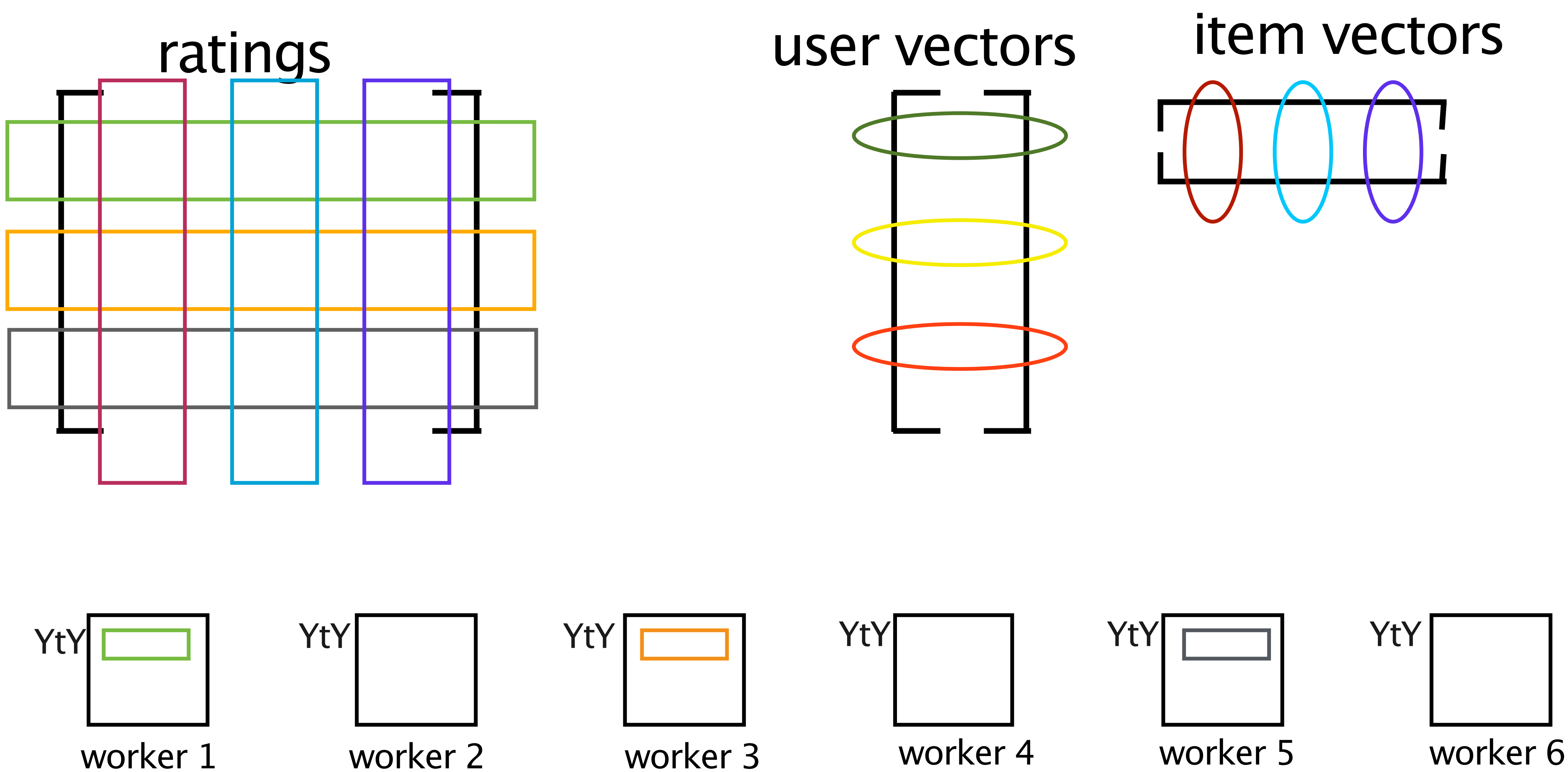
Third Attempt (half gridify)

- Partition ratings matrix into K user (row) and item (column) blocks, partition, and cache
- For each iteration:
  1. Compute YtY over item vectors and broadcast
  2. For each item vector, send a copy to each user rating partition that requires it (potentially all partitions)
  3. Each partition aggregates intermediate terms and solves for optimal user vectors



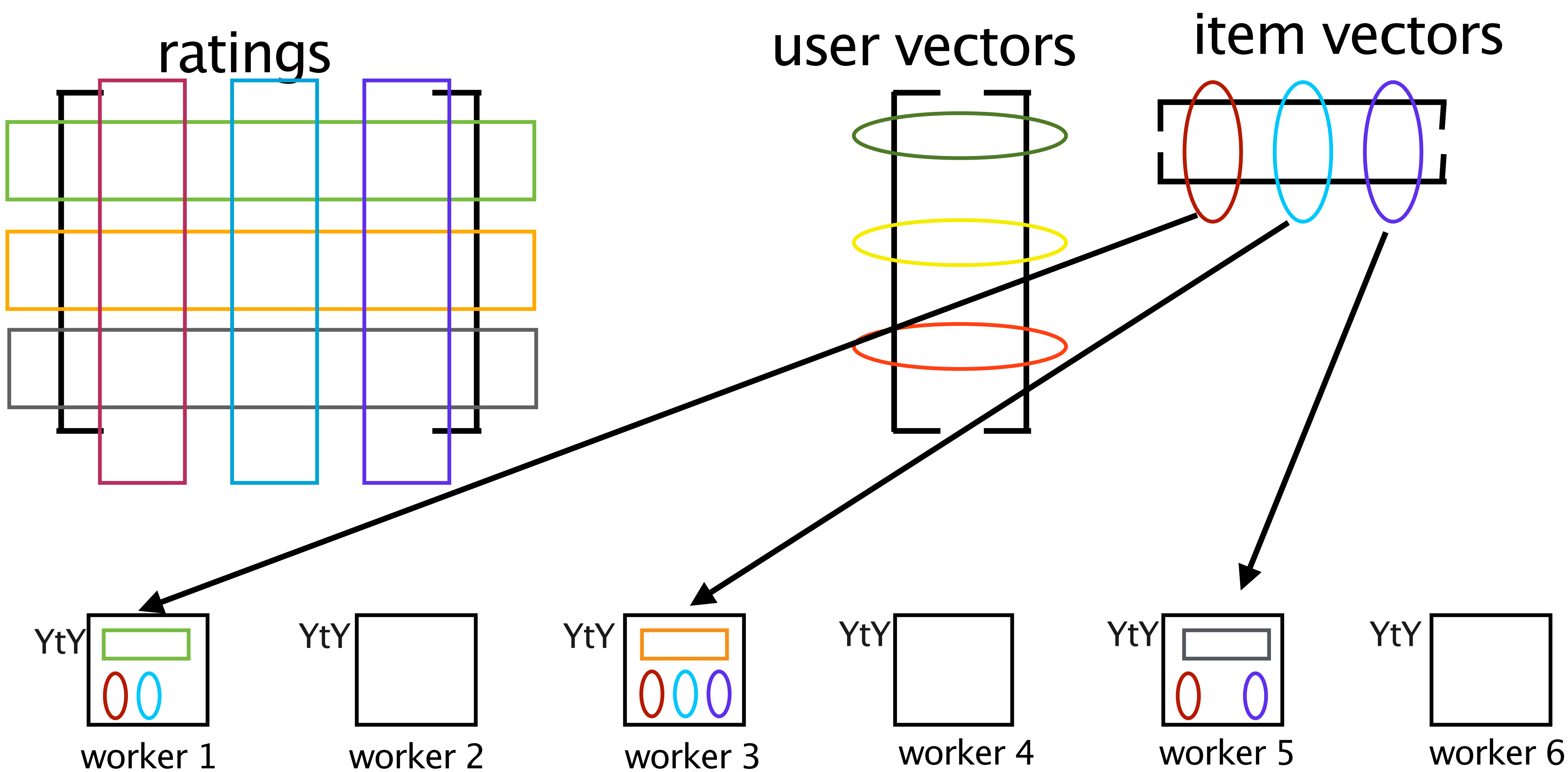
Third Attempt (half gridify)

- Partition ratings matrix into K user (row) and item (column) blocks, partition, and cache
- For each iteration:
  1. Compute YtY over item vectors and broadcast
  2. For each item vector, send a copy to each user rating partition that requires it (potentially all partitions)
  3. Each partition aggregates intermediate terms and solves for optimal user vectors



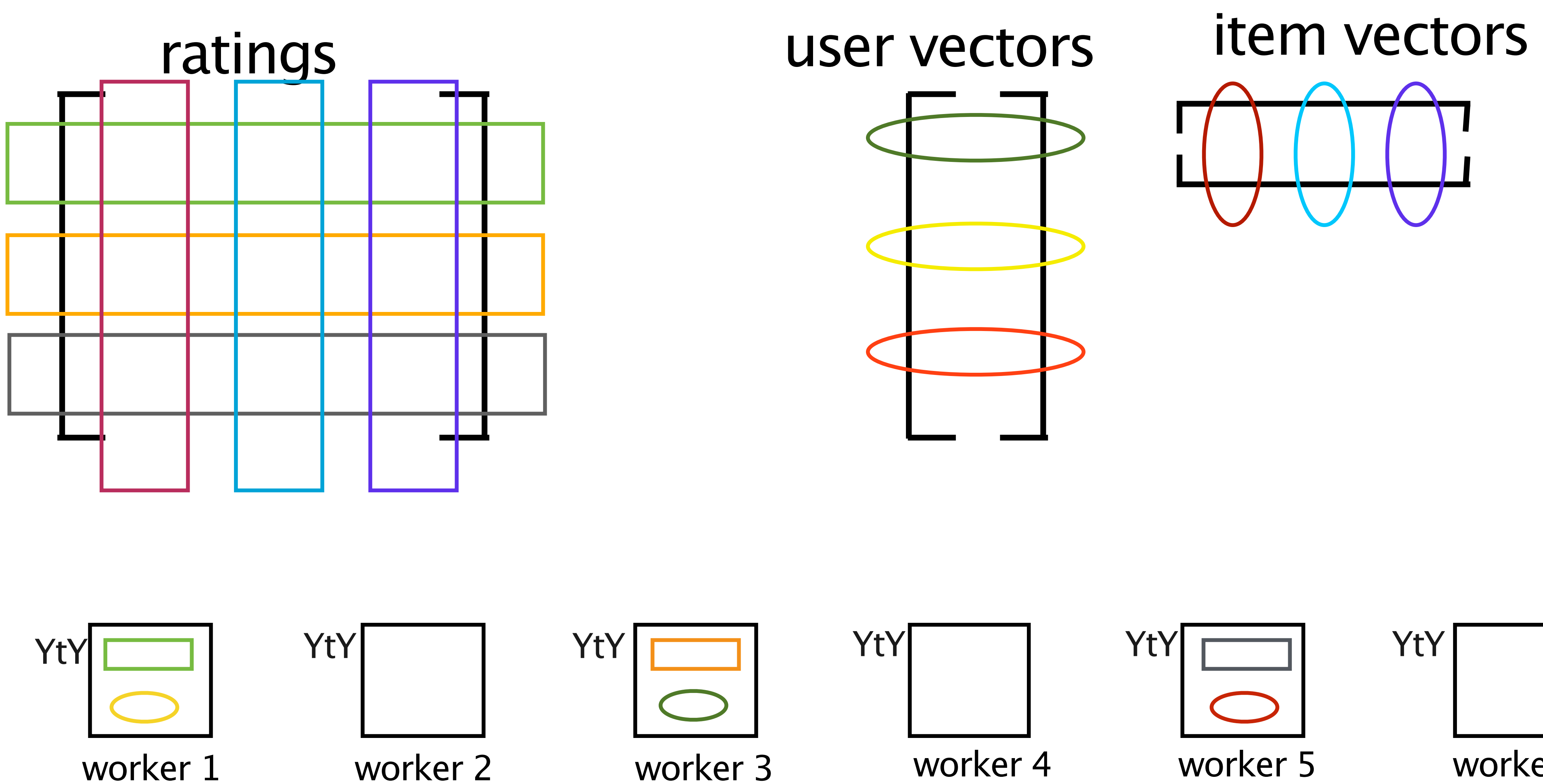
Third Attempt (half gridify)

- Partition ratings matrix into K user (row) and item (column) blocks, partition, and cache
- For each iteration:
  1. Compute YtY over item vectors and broadcast
  2. For each item vector, send a copy to each user rating partition that requires it (potentially all partitions)
  3. Each partition aggregates intermediate terms and solves for optimal user vectors



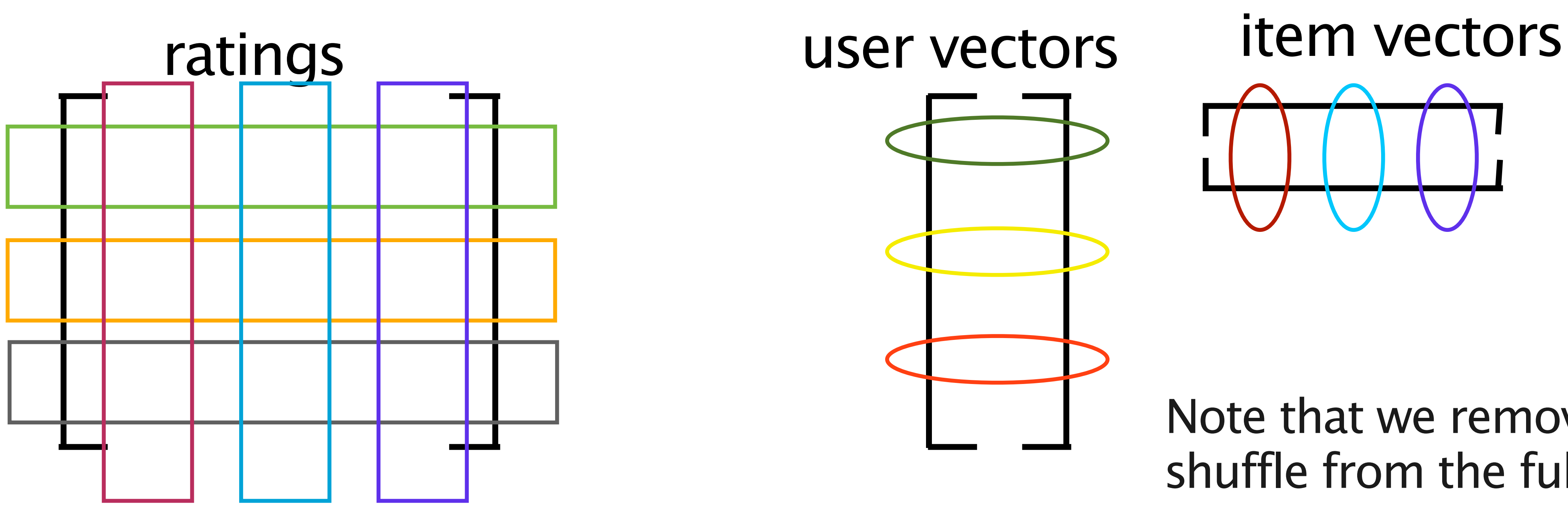
Third Attempt (half gridify)

- Partition ratings matrix into K user (row) and item (column) blocks, partition, and cache
- For each iteration:
  1. Compute YtY over item vectors and broadcast
  2. For each item vector, send a copy to each user rating partition that requires it (potentially all partitions)
  3. Each partition aggregates intermediate terms and solves for optimal user vectors

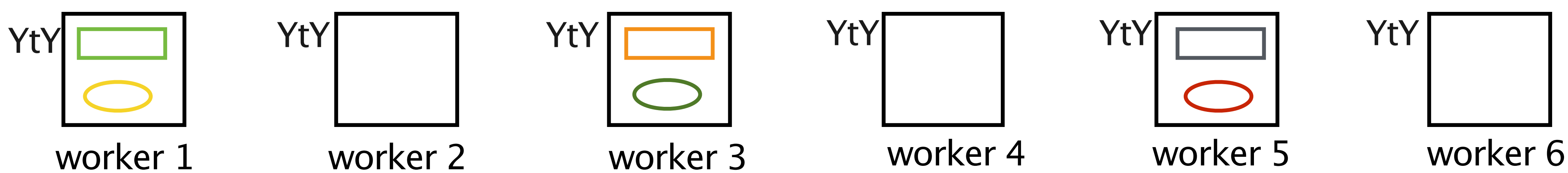


Third Attempt (half gridify)

- Partition ratings matrix into K user (row) and item (column) blocks, partition, and cache
- For each iteration:
  1. Compute YtY over item vectors and broadcast
  2. For each item vector, send a copy to each user rating partition that requires it (potentially all partitions)
  3. Each partition aggregates intermediate terms and solves for optimal user vectors



Note that we removed the extra shuffle from the full gridify approach.



Third Attempt (half gridify)

```
private def updateFeatures(
    products: RDD[(Int, Array[Array[Double]])],
    productOutLinks: RDD[(Int, OutLinkBlock)],
    userInLinks: RDD[(Int, InLinkBlock)],
    partitioner: Partitioner,
    rank: Int,
    lambda: Double,
    alpha: Double,
    YtY: Option[Broadcast[DoubleMatrix]])
  : RDD[(Int, Array[Array[Double]])] =
{
  val numBlocks = products.partitions.size
  productOutLinks.join(products).flatMap { case (bid, (outLinkBlock, factors)) =>
      val toSend = Array.fill(numBlocks)(new ArrayBuffer[Array[Double]])
      for (p <- 0 until outLinkBlock.elementIds.length; userBlock <- 0 until numBlocks) {
        if (outLinkBlock.shouldSend(p)(userBlock)) {
          toSend(userBlock) += factors(p)
        }
      }
      toSend.zipWithIndex.map{ case (buf, idx) => (idx, (bid, buf.toArray)) }
  }.groupByKey(partitioner)
   .join(userInLinks)
   .mapValues{ case (messages, inLinkBlock) =>
      updateBlock(messages, inLinkBlock, rank, lambda, alpha, YtY)
    }
}
```

Actual MLLib code!

- Pros
  - Ratings get cached and never shuffled
  - Once item vectors are joined with ratings partitions each partition has enough information to solve optimal user vectors without any additional shuffling/aggregation (which occurs with the “full gridify” scheme)
- Cons
  - Each partition could potentially require a copy of each item vector (which may not all fit in memory)
  - Potentially requires more local memory than “full gridify” scheme

ALS Running Times

- Dataset consisting of Spotify streaming data for 2 Million users and 500k artists
  - Note: full dataset consists of 40M users and 20M songs but we haven't yet successfully run with Spark
- All jobs run using 40 latent factors
- Spark jobs used 200 executors with 8G containers
- Hadoop job used 1k mappers, 300 reducers

Hadoop	Spark (full gridify)	Spark (half gridify)
10 hours	3.5 hours	1.5 hours

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

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