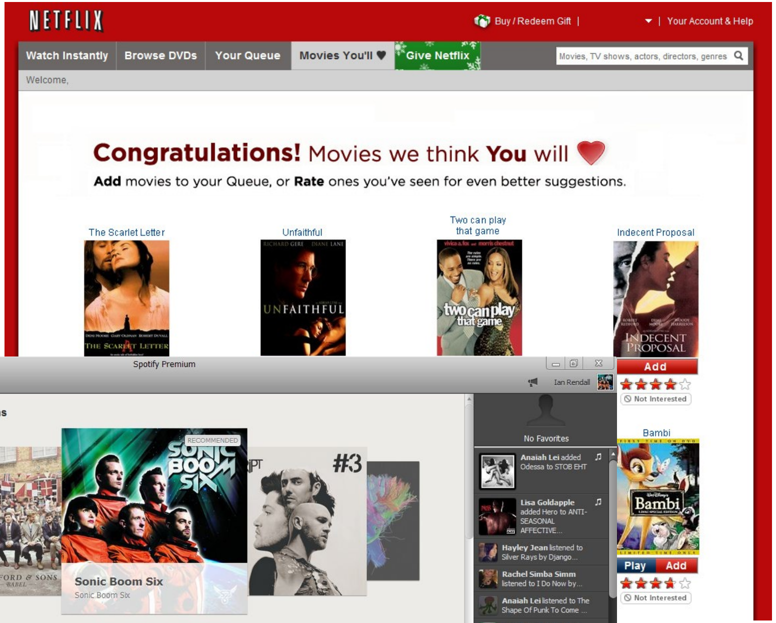
Computing Recommendations at Extreme Scale with Apache Flink

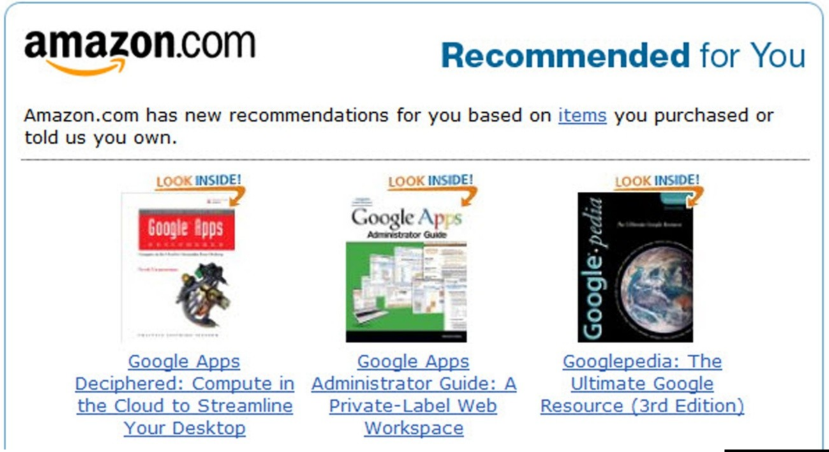
Till Rohrmann
Flink committer / data Artisans
trohrmann@apache.org
@stsffap

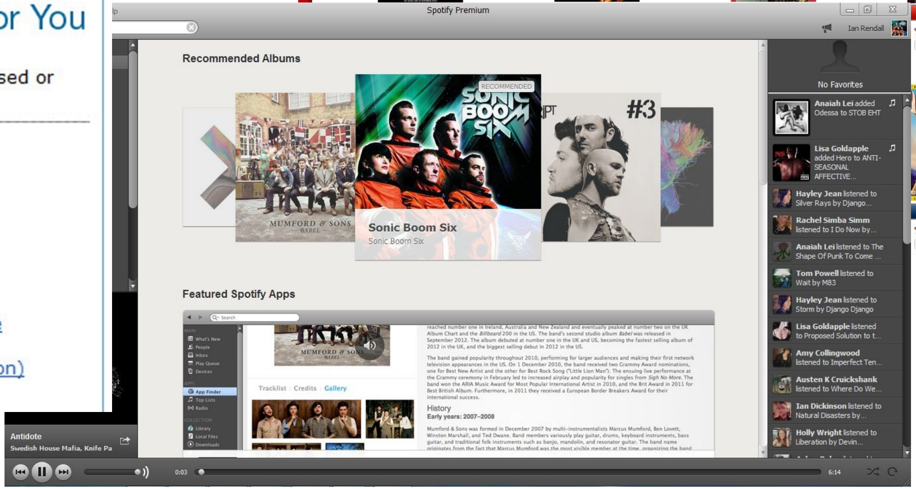
Recommendations: Collaborative Filtering

Recommendations

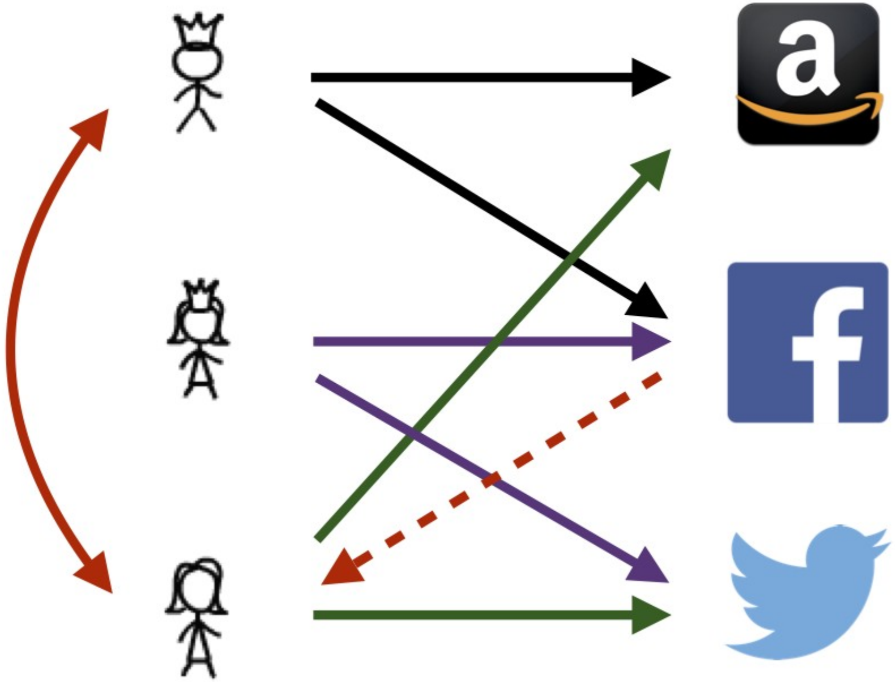
- Omnipresent nowadays
- Important for user experience and sales







Recommending Websites



Collaborative Filtering

- Recommend items based on users with similar preferences
- Latent factor models capture underlying characteristics of items and preferences of user
- Predicted preference: $\hat{r}_{u,i} = x_u^T y_i$

Rating Matrix

- Explicit or implicit ratings
- Prediction goal: Rating for unseen items

$$\begin{array}{cc} & \textit{Items} \\ \textit{Users} & \begin{pmatrix} 10 & 5 & ? & \vdots \\ ? & 2 & 10 & \vdots \\ 7 & ? & 5 & \vdots \end{pmatrix} \end{array}$$

Princess

Facebook

Matrix Factorization

- Calculate low rank approximation to obtain latent factors

$$\begin{array}{cc} & \textit{Items} \\ \textit{Users} & \begin{pmatrix} 10 & 5 & ? \\ ? & 2 & 10 \\ 7 & ? & 5 \end{pmatrix} \end{array} \approx \begin{pmatrix} X \end{pmatrix} \bullet \begin{pmatrix} Y \end{pmatrix}$$

Facebook

Princess

$$\min_{X,Y} \sum_{r_{u,i} \neq 0} \left(r_{u,i} - x_u^T y_i \right)^2 + \lambda \left(\sum_u n_u \left\| x_u \right\|^2 + \sum_i n_i \left\| y_i \right\|^2 \right)$$

- $r_{u,i}$ = rating of user u for item i
- x_u = latent factors of user u
- y_i = latent factors of item i
- λ = regularization constant
- n_u = number of rated items for user u
- n_i = number of ratings for item i

Alternating Least Squares

- Hard to optimize since we have two variables X, Y
- Fixing one variables gives quadratic problem

$$x_u = \left(Y S^u Y^T + \lambda n_u \mathrm{I} \right)^{-1} Y r_u^T$$

$$S_{ii}^u = \begin{cases} 1 & if\ r_{u,i} \neq 0 \\ 0 & else \end{cases}$$

We only need the item vectors rated by user u

ALS Algorithm

- Update user matrix

$$\begin{array}{cc} & \text{Items} \\ \text{Users} & \begin{pmatrix} 10 & 5 & ? \\ ? & 2 & 10 \\ 7 & ? & 5 \end{pmatrix} \end{array} \approx \overset{\text{Calculate update}}{\boxed{\begin{pmatrix} \\ X \\ \\ \end{pmatrix}}} \bullet \overset{\text{Keep fixed}}{\boxed{\begin{pmatrix} & Y & \end{pmatrix}}}$$

ALS Algorithm contd.

- Update item matrix

$$\begin{array}{cc} & \text{Items} \\ \text{Users} & \begin{pmatrix} 10 & 5 & ? \\ ? & 2 & 10 \\ 7 & ? & 5 \end{pmatrix} \end{array} \approx \overset{\text{Keep fixed}}{\boxed{\begin{pmatrix} \\ X \\ \\ \end{pmatrix}}} \bullet \overset{\text{Calculate update}}{\boxed{\begin{pmatrix} \quad Y \quad \end{pmatrix}}}$$

ALS Algorithm contd.

- Repeat update step until convergence

$$\begin{array}{cc} & \textit{Items} \\ \textit{Users} & \begin{pmatrix} 10 & 5 & ? \\ ? & 2 & 10 \\ 7 & ? & 5 \end{pmatrix} \end{array} \approx \overset{\text{Calculate update}}{\boxed{\begin{pmatrix} \\ X \\ \\ \end{pmatrix}}} \bullet \overset{\text{Keep fixed}}{\boxed{\begin{pmatrix} \quad Y \quad \end{pmatrix}}}$$

Apache Flink

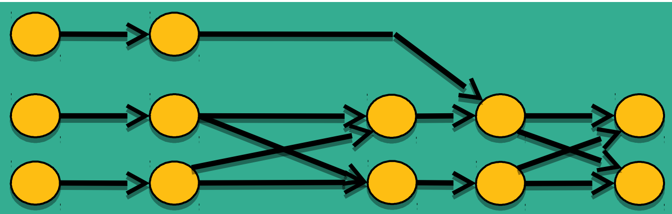
What is Apache Flink?

Hadoop M/R
Table
Gelly
FlinkML
Dataflow
MRQL
Cascading (WiP)
Table
SAMOA
Dataflow (WiP)

Apache Flink deep-dive
by Stephan Ewen
Tomorrow, 12:20 – 13:00
on Stage 3

DataSet (Java/Scala/Python)

DataStream (Java/Scala)



Streaming dataflow runtime

Local
Remote
Yarn
Tez
Embedded

Why Using Flink for ALS?

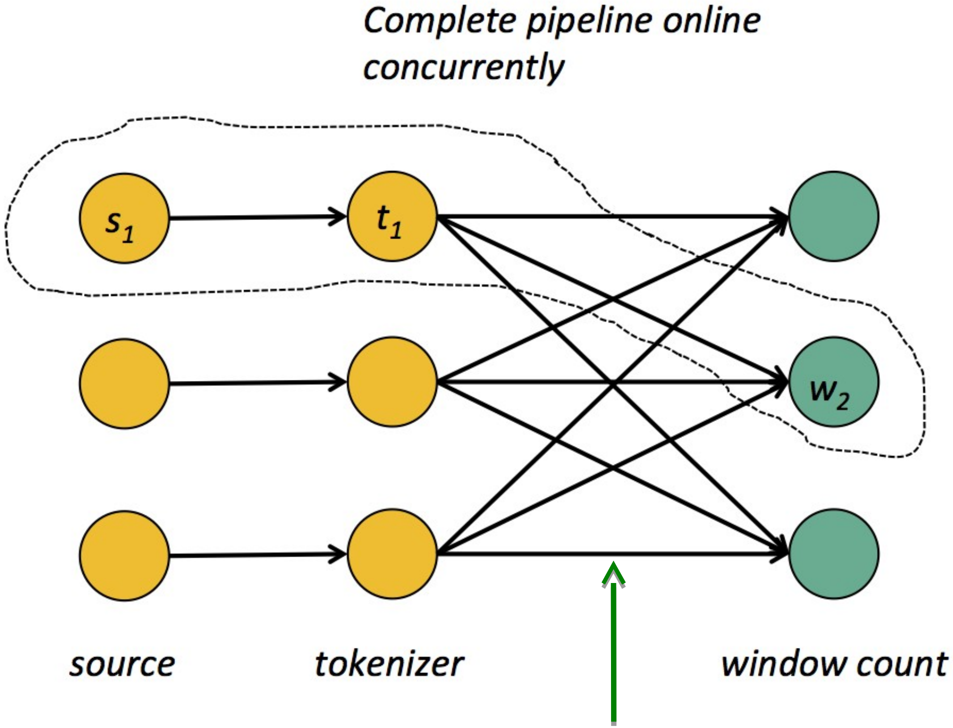
- Expressive API
- Pipelined stream processor
- Closed loop iterations
- Operations on managed memory

Expressive APIs

- DataSet: Abstraction for distributed data
- Computation specified as sequence of lazily evaluated transformations

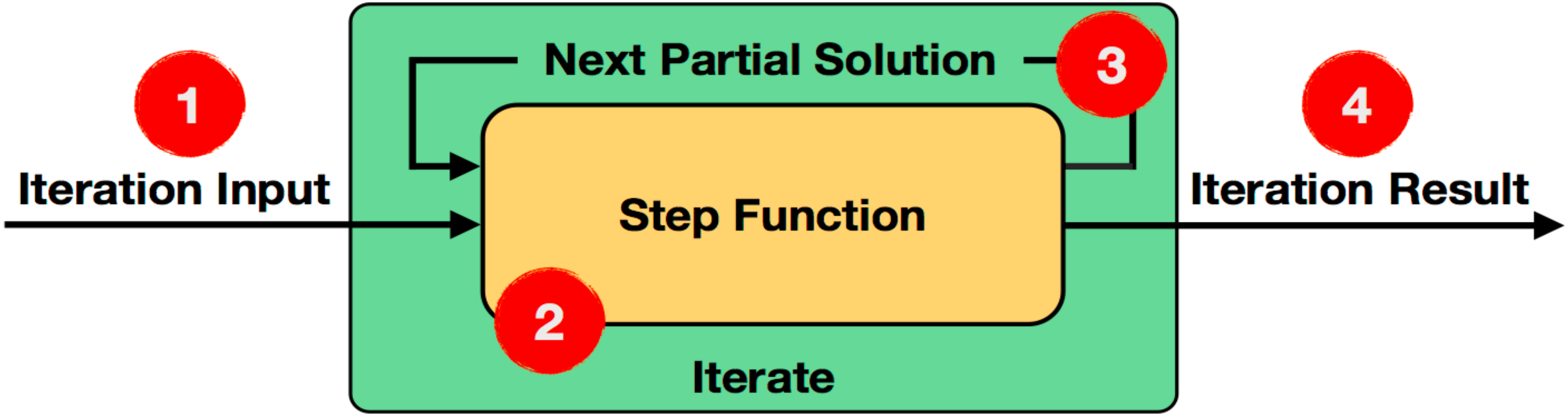
```
case class Word(word: String, frequency: Int)

val lines: DataSet[String] = env.readTextFile(…)

lines.flatMap(line => line.split(" ").map(word => Word(word,1))
     .groupBy("word").sum("frequency")
     .print()
```

Pipelined Stream Processor

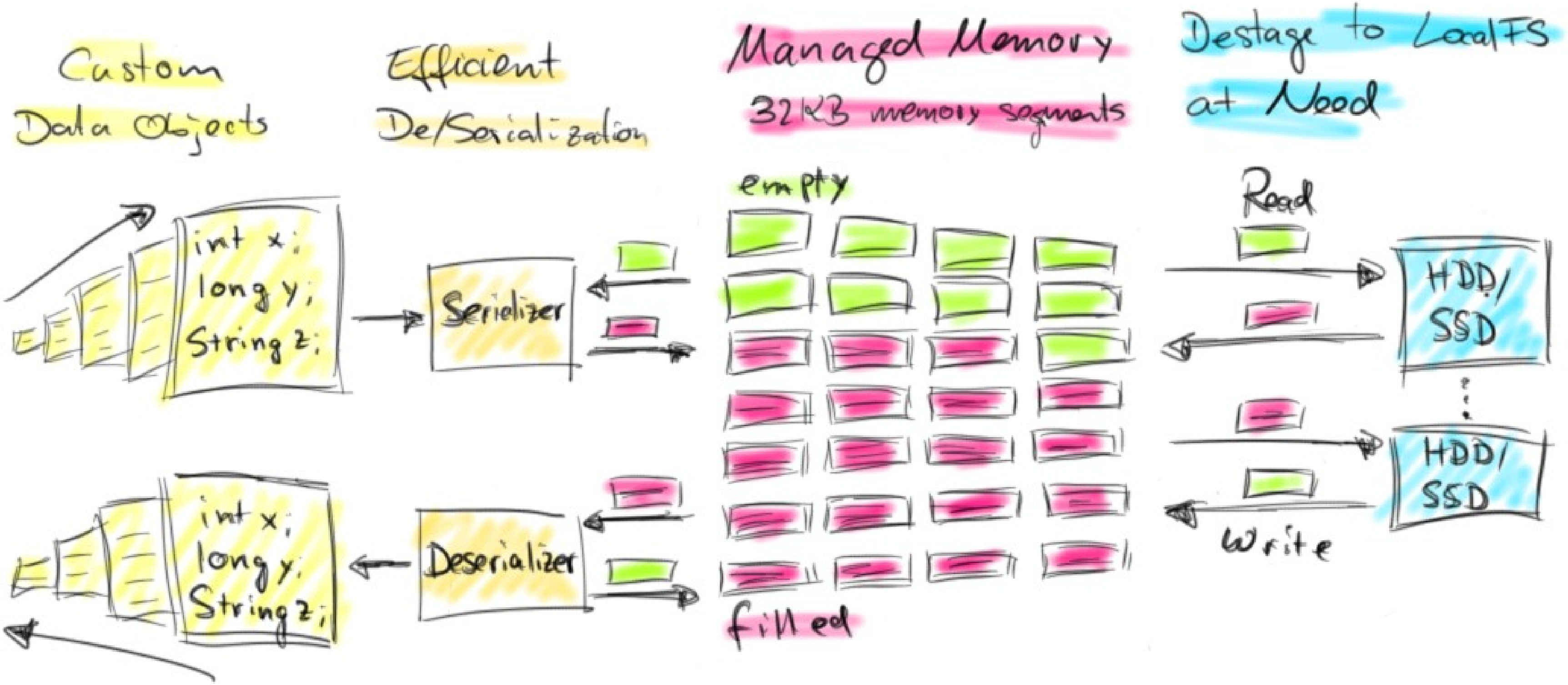


Avoiding materialization of intermediate results

Iterate in the Dataflow



Memory Management

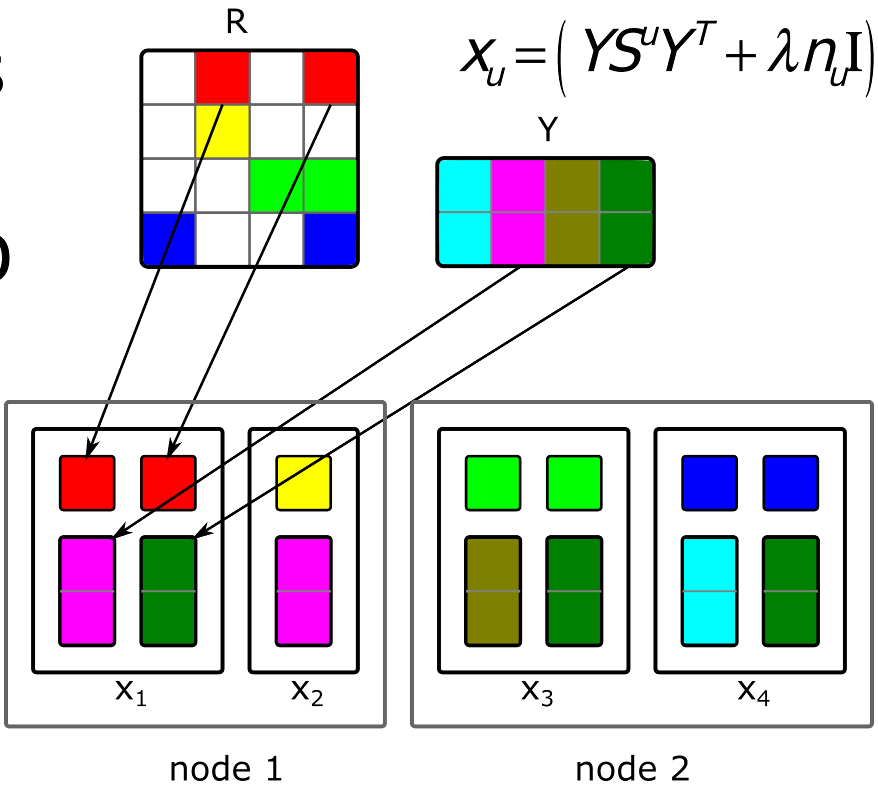


ALS implementations with Apache Flink

Naïve Implementation

1. Join item vectors with ratings
2. Group on user ID
3. Compute new user vectors

$$x_u = \left(Y S^u Y^T + \lambda n_u \mathrm{I} \right)^{-1} Y r_u^T$$

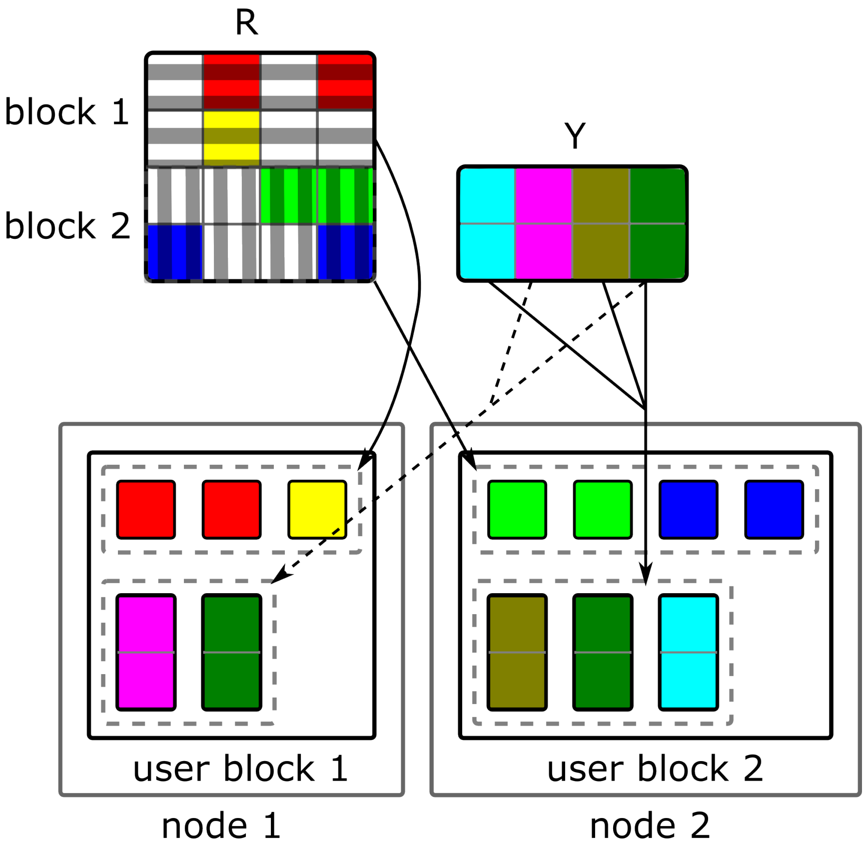


Pros and Cons of Naïve ALS

- Pros
  - Easy to implement
- Cons
  - Item vectors are sent redundantly to network nodes
  - Two shuffle steps make execution expensive

Blocked ALS Implementation

1. Create user and item rating blocks
2. Cache them on worker nodes
3. Send all item vectors needed by user rating block bundled
4. Compute block of item vectors

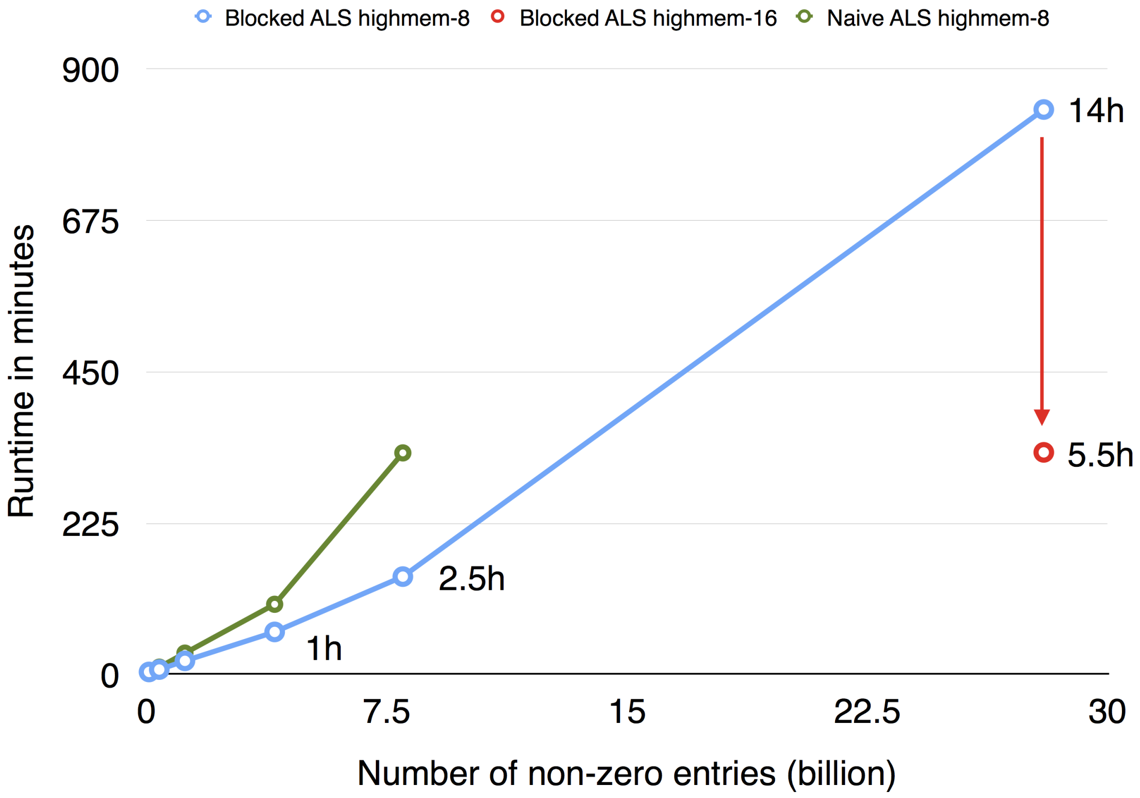


Based on Spark's MLlib implementation

Pros and Cons of Blocked ALS

- Pros
  - Reduces network load by avoiding data duplication
  - Caching ratings: Only one shuffle step needed
- Cons
  - Duplicates the rating matrix (user block/item block partitioning)

Performance Comparison



- 40 node GCE cluster, highmem-8
- 10 ALS iterations with 50 latent factors
- Rating matrix has 28 billion non zero entries: **Scale of Netflix or Spotify**

Machine Learning with FlinkML

- FlinkML contains blocked ALS
- Support for many other tasks
  - Clustering
  - Regression
  - Classification
- scikit-learn like pipeline support

```
val als = ALS()

val ratingDS =
  env.readCsvFile[(Int,Int,Double)](ratingData)

val parameters = ParameterMap()
  .add(ALS.Iterations,10)
  .add(ALS.NumFactors,50)
  .add(ALS.Lambda,1.5)

als.fit(ratingDS,parameters)

val testingDS = env.readCsvFile[(Int,Int)](testingData)

val predictions = als.predict(testingDS)
```

Closing

What Have You Seen?

- How to use collaborative filtering to make recommendations
- Apache Flink, a powerful parallel stream processing engine
- How to use Apache Flink and alternating least squares to factorize really large matrices

Flink Forward

BERLIN 12/13 OCT 2015

flink.apache.org
@ApacheFlink