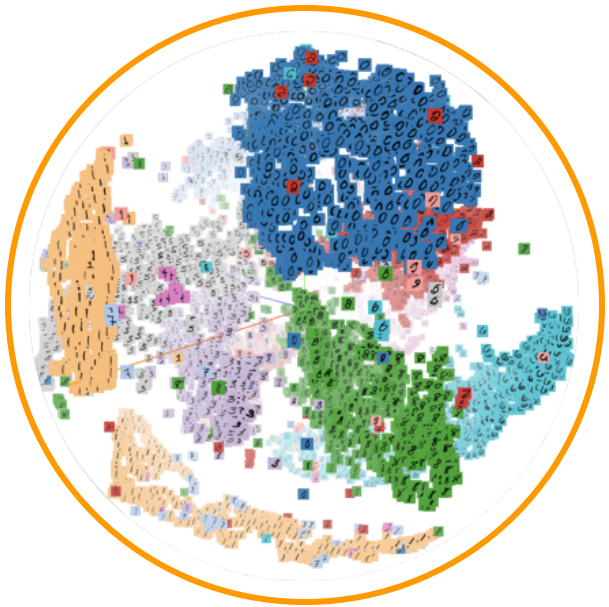
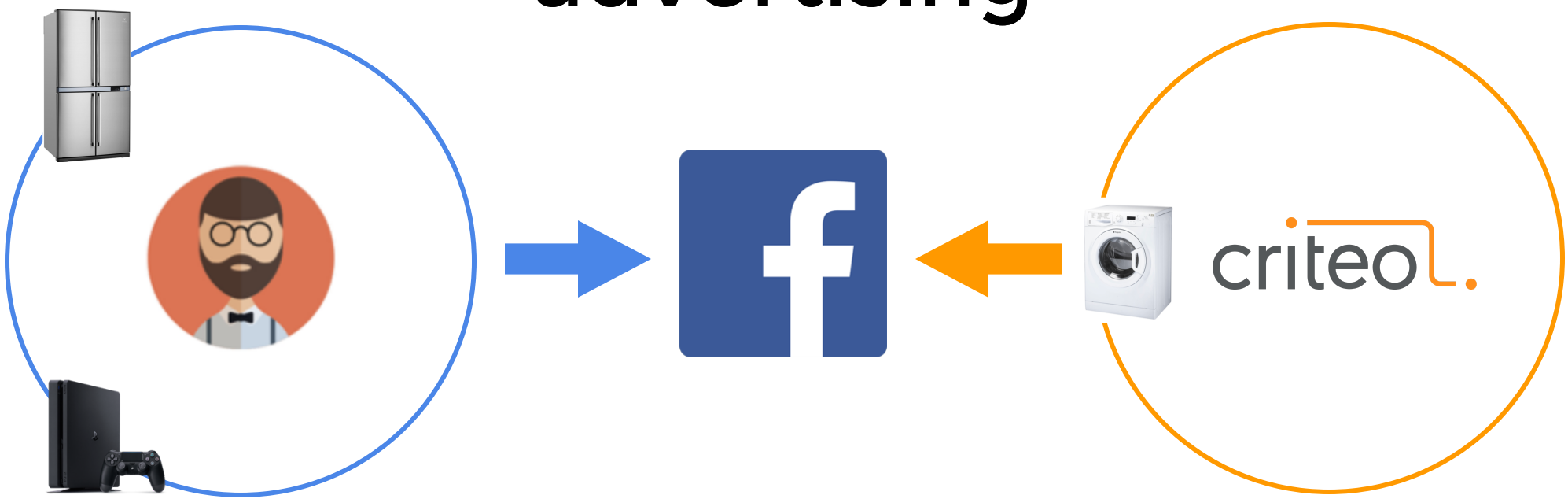
criteo



Billion-scale recommendations on Spark

Motivation

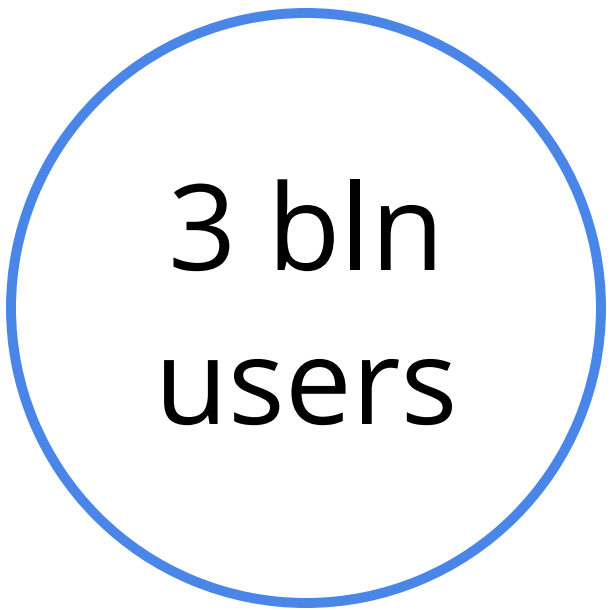
Criteo business is personalized advertising



* User icons designed by Freepik

criteo

At tera-scale



worldwide

We want to know...

criteo

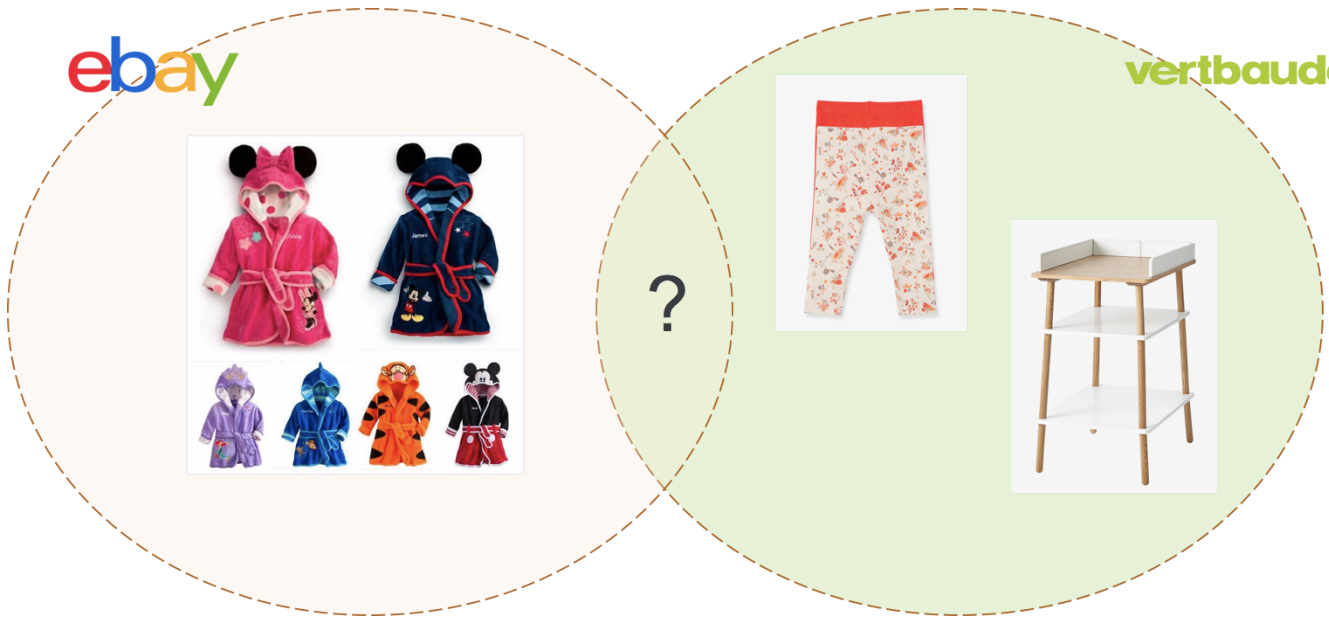
Which products are more similar?

?

Which users have similar product history?

* User icons designed by Freepik

Which products from Ebay can we recommend to those you have history at VertBaudet?



The standard approach

Underlying data => user timelines







* User icons designed by Freepik

criteo

Implicit feedback personalized recommendations

u_i - user i

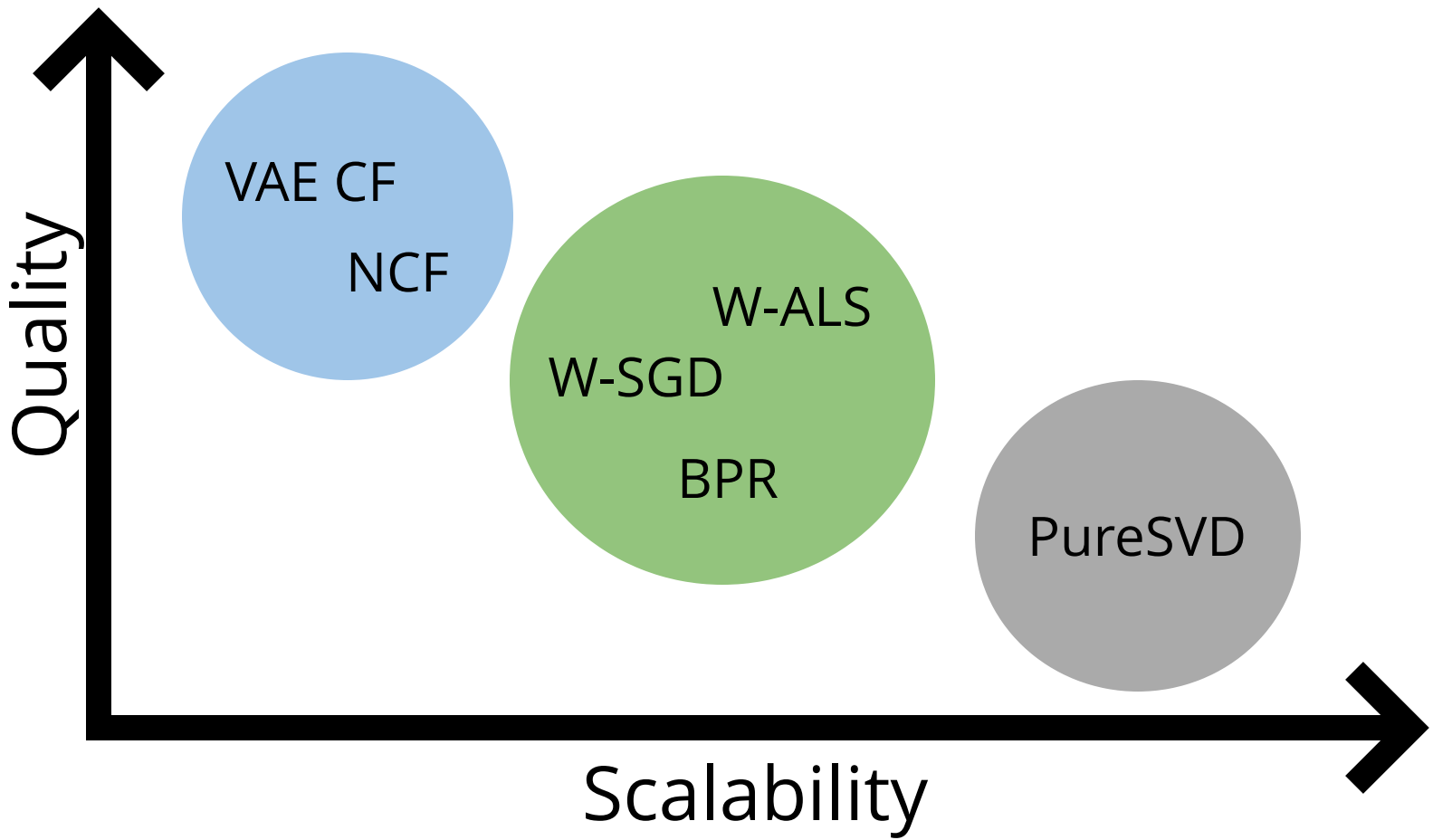
v_j - item j

No explicit feedback, but interactions of users with items (views, clicks, sales, etc.)

Factorize user-item matrix

		Items					
		1	*2*	...	*i*	...	*m*
Users	*1*	5	3		1	2	
	2		2				4
	:			5			
	u	3	4		2	1	
	:					4	
	n			3	2		

Plethora of methods



Mostly tiny datasets used

	ML-20M	Netflix	MSD
# of users	136,677	463,435	571,355
# of items	20,108	17,769	41,140
# of interactions	10.0M	56.9M	33.6M
% of interactions	0.36%	0.69%	0.14%

3 bln users

1 bln items

Similarity SVD approach

1/ Build the similarity matrix

For example, PMI, pointwise mutual information, is a measure of association between a pair of discrete variable values

$$PMI_{i,j} := \log \frac{P(u_i,v_j)}{P(u_i)P(v_j)} - \log k$$

$PMI_{i,j} := 0 \text{ when } P(u_i, v_j) = 0$

2/ Truncated SVD and kNN

$$PMI = USV^T$$

U is a user factor matrix

V is an item factor matrix

$U_{i,*}SV^T$ is scores of all items for a particular user

Comparison between methods

Netflix dataset, ratings 4+

Method	**NDCG@100**	**Training time, secs**
Most popular	0.158	0
PureSVD (2012)	0.340	3
PMI SVD (ours)	**0.348 (CI ~0.002)**	**3**
W-ALS (2008)	0.352	84
VAE-CF (2018)	0.386	5 580

criteo

Scalable truncated SVD

criteo

Step 1 - tall-and-skinny projection that captures most of the action

$$||A - USV^T|| \approx ||A - QQ^T A||$$

Where $||\cdot||$ is spectral norm (the largest eigenvalue)

And Q is low-rank and orthonormal

criteo

Step 2 - SVD of k x k matrix gives us everything

$$\text{We can decompose } Q^T A \text{ part of } QQ^T A = Q(\hat{U} S V^T)$$

criteo

Step 2 - SVD of k x k matrix gives us everything

We can decompose $Q^T A$ part of $QQ^T A = Q(\hat{U} S V^T)$

notice that by uniqueness of $U = Q\hat{U}$

criteo

Step 2 - SVD of k x k matrix gives us everything

We can decompose $Q^T A$ part of $QQ^T A = Q(\hat{U} S V^T)$

notice that by uniqueness of $U = Q\hat{U}$

and finding $\hat{U}$ is cheap!

criteo

Step 2 - SVD of k x k matrix gives us everything

We can decompose $Q^T A$ part of $QQ^T A = Q(\hat{U} S V^T)$

notice that by uniqueness of $U = Q\hat{U}$

and finding $\hat{U}$ is cheap!

$(Q^T A)^T = \tilde{Q} R = \tilde{Q}(\hat{V} S \hat{U}^T)$

Algorithm

criteo

Algorithm

1/ Generate random matrix $G \in R^{m \times (k+p)}$

with values drawn independently from gaussian distribution

where k - target approximation rank, p - oversampling

Algorithm

criteo

1/ Generate random matrix $G \in R^{m \times (k+p)}$

with values drawn independently from gaussian distribution

where k - target approximation rank, p - oversampling

2/ Multiply by A several times: $\hat{Q} = (AA^T)^q AG$

orthogonalizing columns after every multiplication $\hat{Q} = QR$

Algorithm

criteo

1/ Generate random matrix $G \in R^{m \times (k+p)}$

with values drawn independently from gaussian distribution

where k - target approximation rank, p - oversampling

2/ Multiply by A several times: $\hat{Q} = (AA^T)^q AG$

orthogonalizing columns after every multiplication $\hat{Q} = QR$

3/ Find $\hat{U}$ by $B := Q^T A$; $B^T = \tilde{Q}R = \tilde{Q}(\hat{V}S\hat{U}^T)$

Algorithm

criteo

1/ Generate random matrix $G \in R^{m \times (k+p)}$

with values drawn independently from gaussian distribution

where k - target approximation rank, p - oversampling

2/ Multiply by A several times: $\hat{Q} = (AA^T)^q AG$

orthogonalizing columns after every multiplication $\hat{Q} = QR$

3/ Find $\hat{U}$ by $B := Q^T A$; $B^T = \tilde{Q}R = \tilde{Q}(\hat{V}S\hat{U}^T)$

4/ Return $U := Q\hat{U}$

Approximation error bound

With n=10^9, k=100, p=30, q=3:

$$||A - QQ^T A|| \leq 4.19 \times \sigma_{k+1}$$

Meaning that R-SVD with k=105 will be as good as full SVD with k=100 even if singular values do not decay

Witten et al., 2013, Randomized Algorithms for Low-Rank Matrix Factorizations: Sharp Performance Bounds

Implementation

criteo

Distributed operations

1/ Generate random matrix $G \in R^{m \times k}$

2/ Mutltiply dense B by single block $C \in R^{k \times k}$

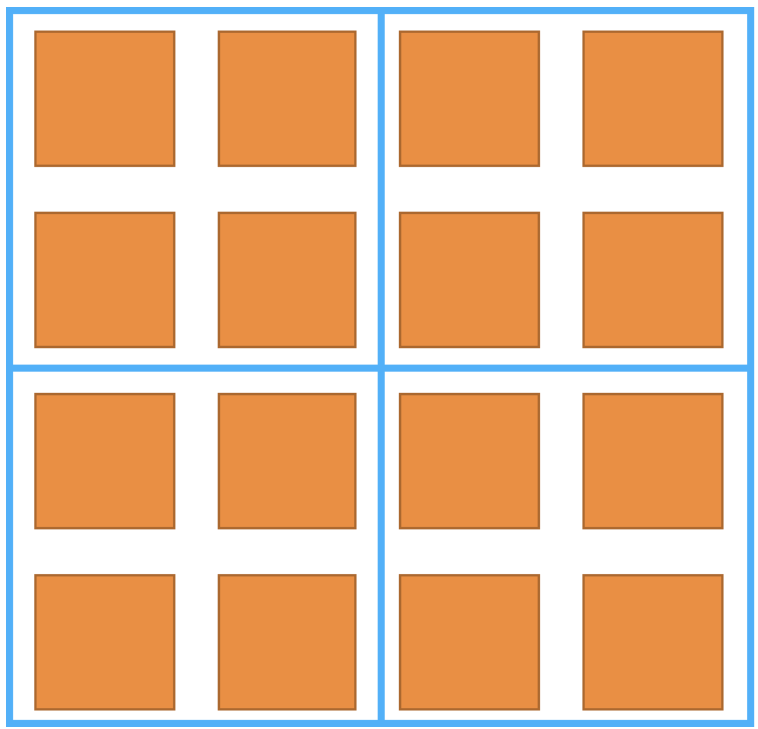
3/ QR-decomposition of dense $B \in R^{m \times k}$

4/ Multiply sparse $A \in R^{n \times m}$ by dense $B \in R^{m \times k}$

criteo

Distributed block-matrix

$A =$



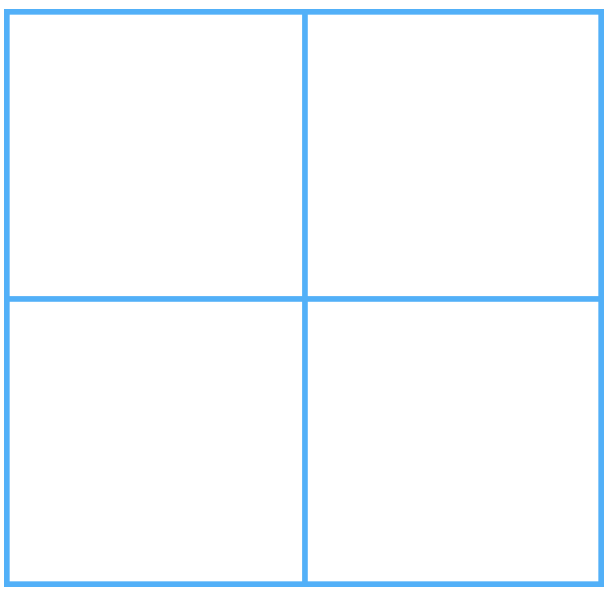
Block-matrix

Partition

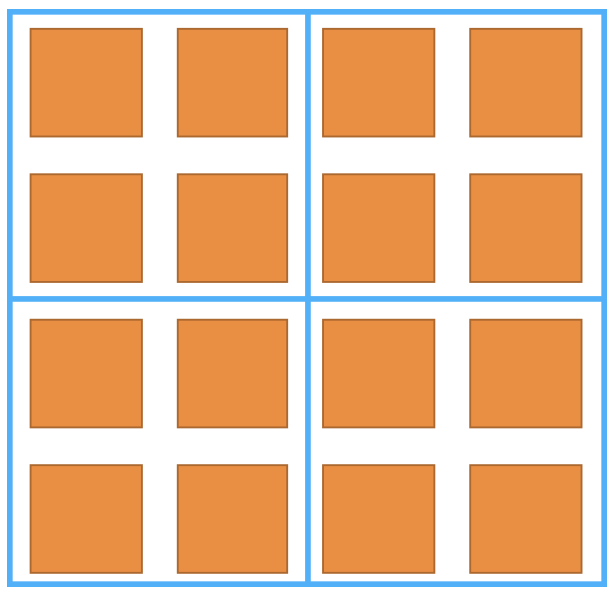
criteo

Generating random matrix

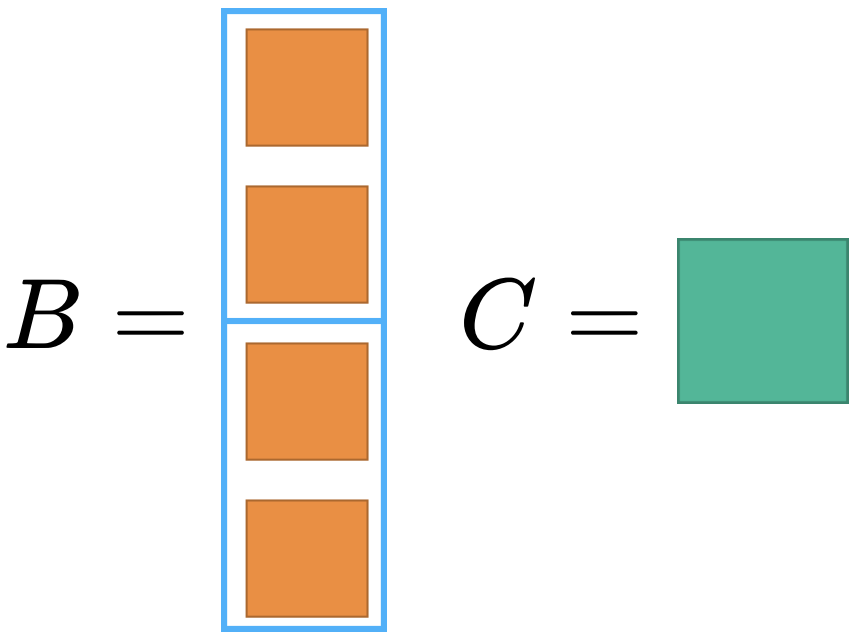
Create empty partitions



Fill-in with random values



Tall-and-skinny multiplication by a single block

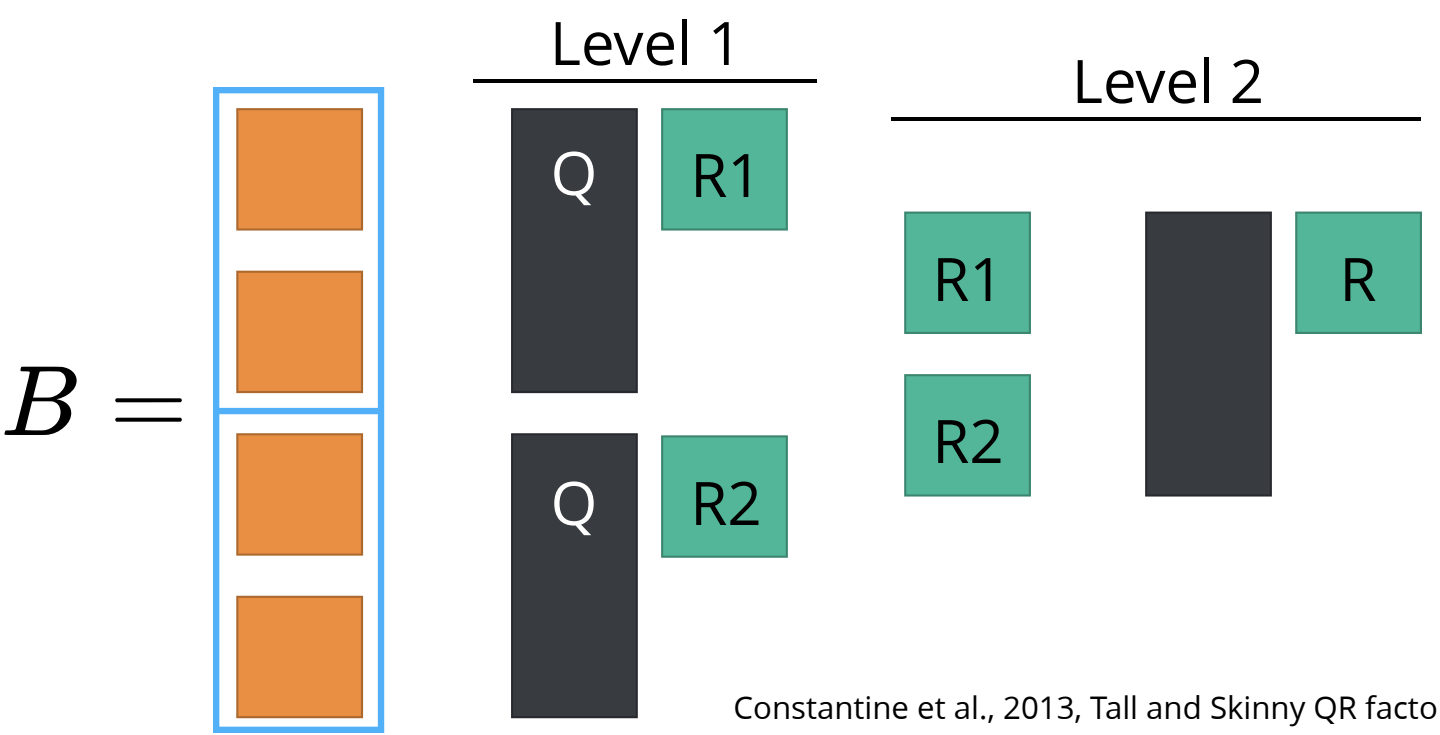


Use efficient broadcasting

```
broadcastedC = sc.broadcast(C)

blocks_B.map {
    block =>
        block * broadcastedC.value
}
```

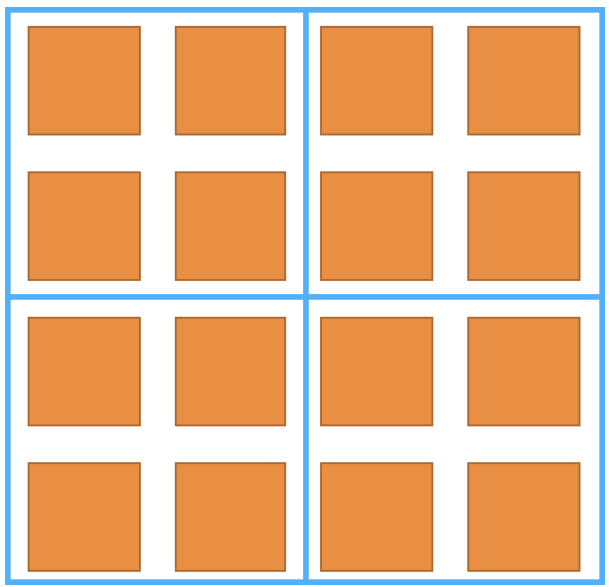
Tall-and-skinny QR-decomposition

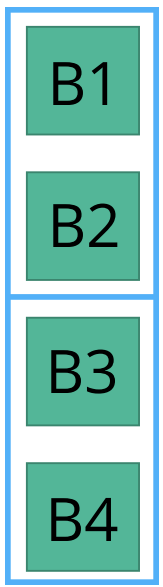


Constantine et al., 2013, Tall and Skinny QR factorizations in MapReduce architectures

criteo

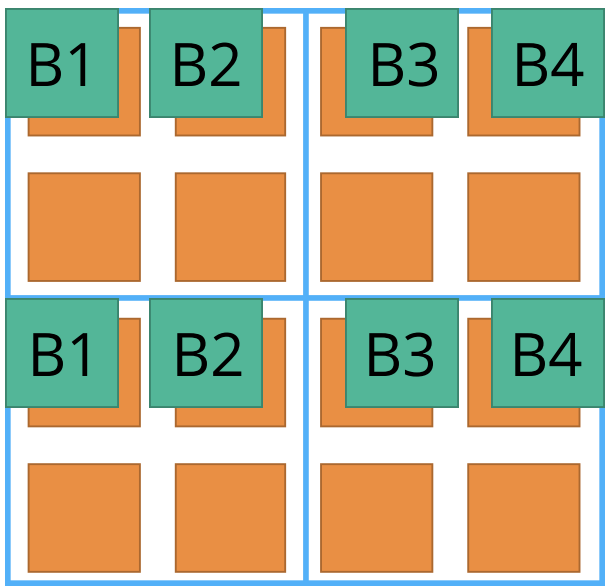
Square sparse by tall-and-skinny dense multiplication

$A =$ 

$B =$ 

1/ Shuffle pattern

Send required blocks of B to every partition of A

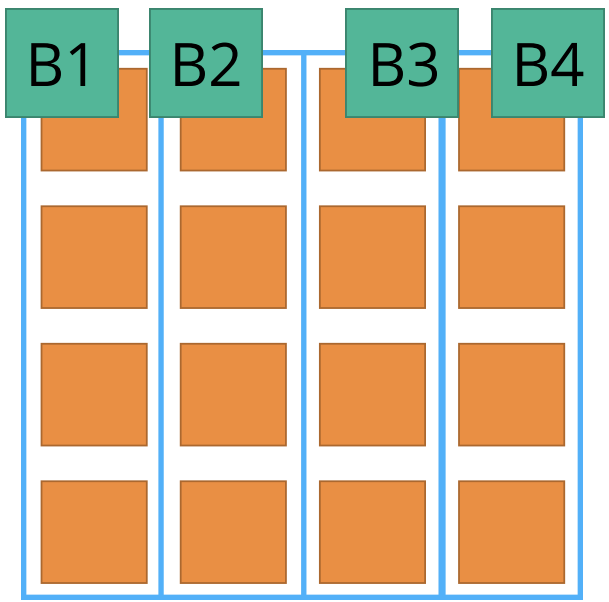


Pros: easy to implement

Cons: shuffle-size is B times number of row partitions in A

2/ Column-partition zipping

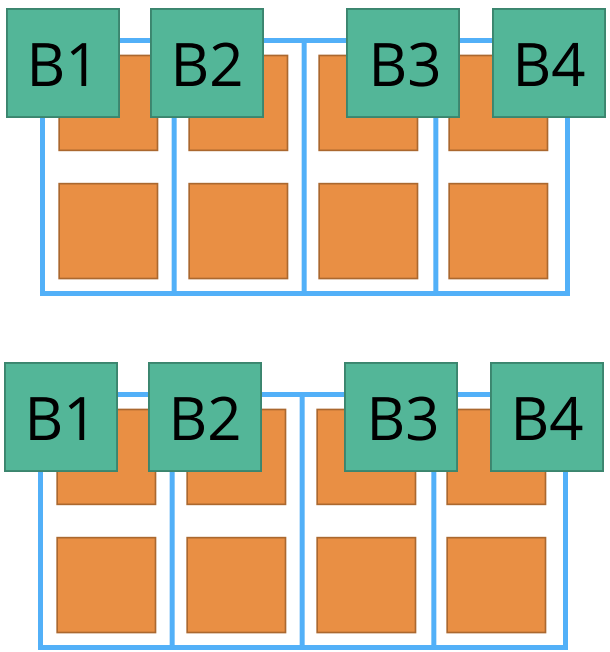
A is column-partitioned, B is row-partitioned. Use zipPartitions method



Pros: no shuffle

Cons: even with single column in partition, we need to store in memory equivalent to the size of B

3/ Multiplexed RDDs column zipping



Split A into multiple RDDs by rows.
Do column-zipping on every RDD

Pros: no shuffle

Cons: overhead on splitting of A

criteo

Properties of our solution

1. Can be expressed in MapReduce / Spark API

2. Memory requirements independent of size

3. Deterministic results

criteo

Clone, fork, send PRs

github.com/criteo/Spark-RSVD

Questions?

We are hiring!

1/ Randomized methods to capture the main action of a matrix:
FINDING STRUCTURE WITH RANDOMNESS:
PROBABILISTIC ALGORITHMS FOR CONSTRUCTING
APPROXIMATE MATRIX DECOMPOSITIONS
https://arxiv.org/pdf/0909.4061.pdf

2/ Sharp bounds on randomized projection error (Corollary 1.5):
Randomized Algorithms for Low-Rank Matrix Factorizations:
Sharp Performance Bounds
https://arxiv.org/pdf/1308.5697.pdf

3/ Indirect tall-and-skinny QR algorithm (the one implemented):
Tall and Skinny QR factorizations
in MapReduce architectures
http://inside.mines.edu/~pconstan/docs/constantine-mrtsqr.pdf

4/ Direct tall-and-skinny QR algorithm (not the one implemented, but good analysis):
Direct QR factorizations for tall-and-skinny
matrices in MapReduce architectures
https://arxiv.org/pdf/1301.1071.pdf

5/ Randomized PCA algorithms (with implementation in Spark):
Randomized algorithms for distributed computation
of principal component analysis and singular value
decomposition
https://arxiv.org/pdf/1612.08709.pdf
https://github.com/hl475/svd (branch testSVD)