A More Scalable Way of Making Recommendations with MLlib

Xiangrui Meng

Spark Summit 2015

databricks™

More interested in application than implementation?

iRIS: A Large-Scale Food and Recipe Recommendation System Using Spark

Joohyun Kim (MyFitnessPal, Under Armour)

3:30 – 4:00 PM

Imperial Ballroom (Level 2)

About Databricks

- Founded by Apache spark creators
- Largest contributor to Spark project, committed to keeping Spark 100% open source
- End-to-end hosted platform

  https://www.databricks.com/product/databricks

Spark MLlib

Large-scale machine learning on Apache Spark

databricks™

About MLlib

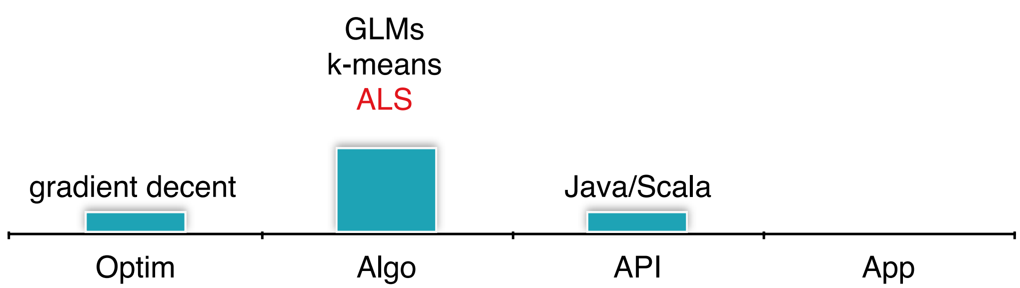
- Started in UC Berkeley AMPLab
  - Shipped with Spark 0.8
- Currently (Spark 1.4)
  - Contributions from 50+ organizations, 150+ individuals
  - Good coverage of algorithms

MLlib’s Mission

MLlib’s mission is to make practical machine learning easy and scalable.

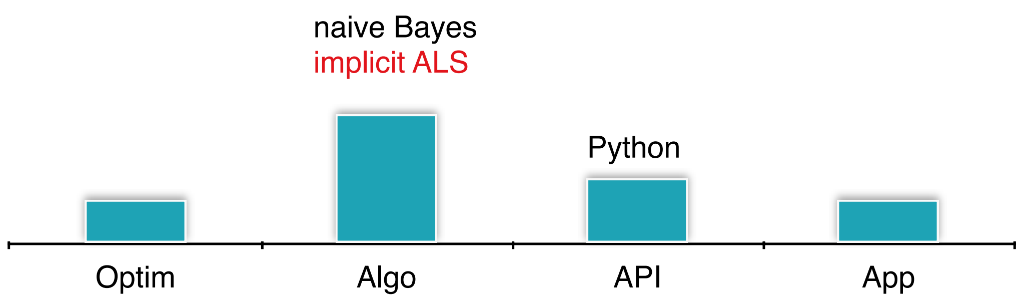
- Easy to build machine learning applications
- Capable of learning from large-scale datasets

A Brief History of MLlib



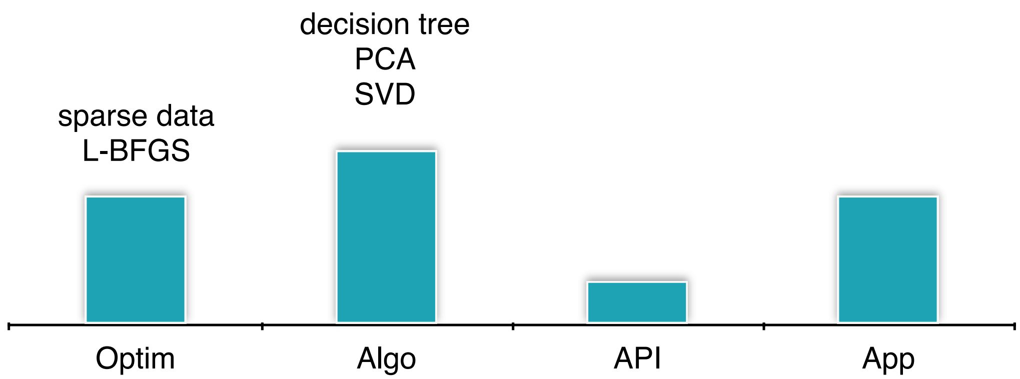
v0.8

A Brief History of MLlib



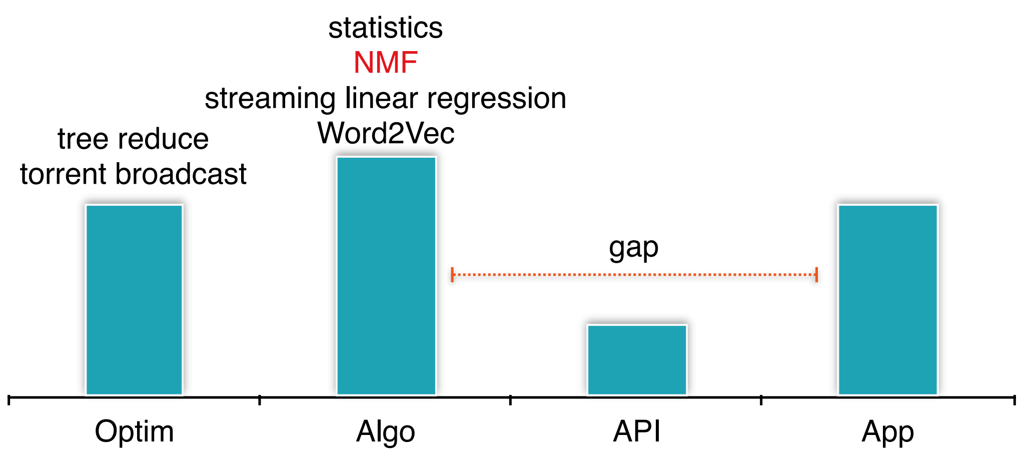
v0.9

A Brief History of MLlib



v1.0

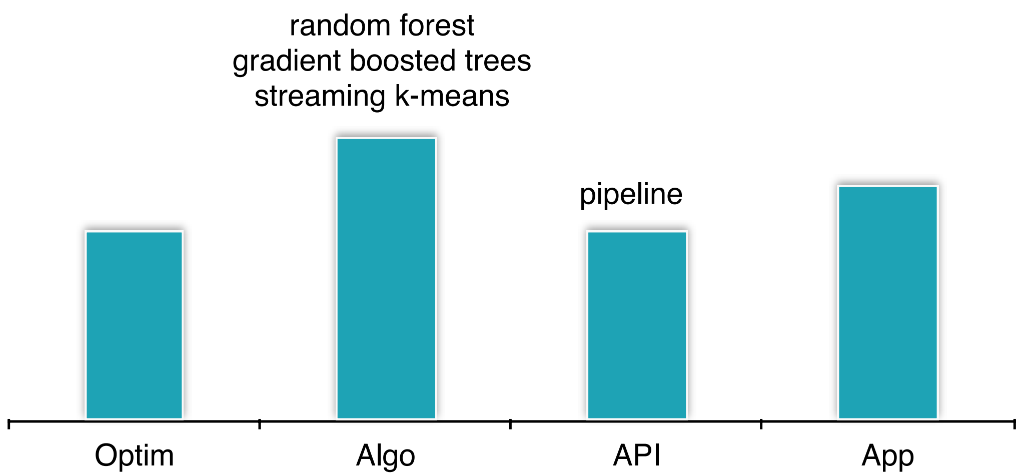
A Brief History of MLlib



v1.1

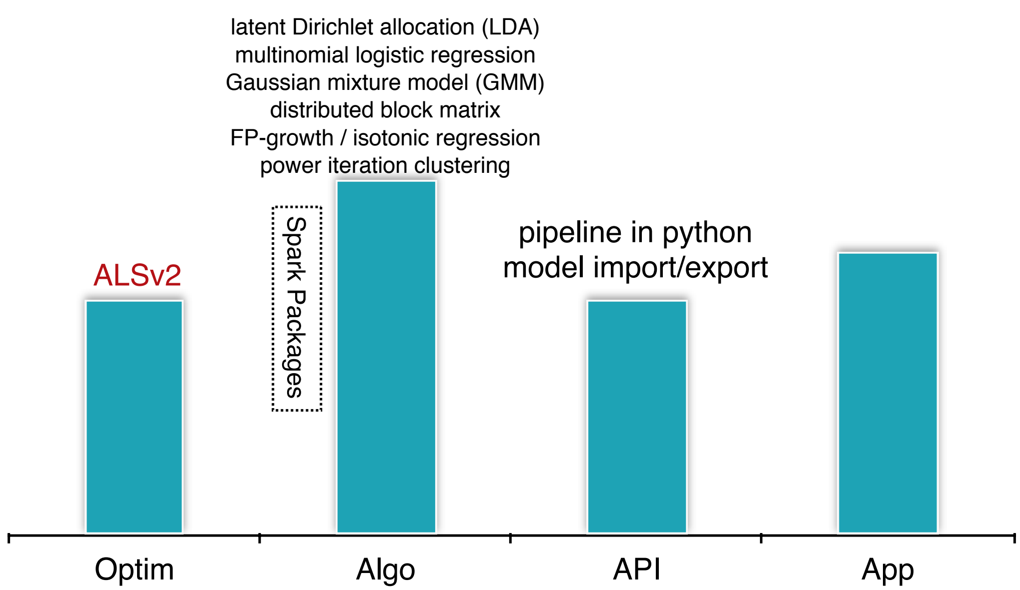
databricks

A Brief History of MLlib



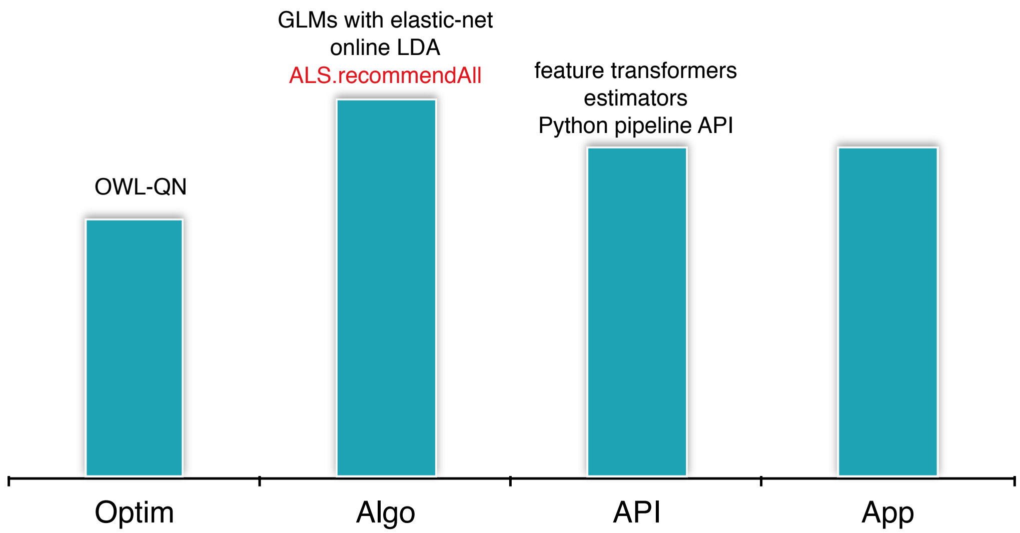
v1.2

A Brief History of MLlib



v1.3

A Brief History of MLlib



v1.4

Alternating Least Squares (ALS)

Collaborative filtering via matrix factorization

databricks™

Collaborative Filtering

items

users

					4		8	
		6		1		7		
	4		3					5
		5	2					3
			?	7		1		
9					5			
7					3	5		
	3		8				2	
		9		6				

A: a rating matrix

databricks

Low-Rank Assumption

- What kind of movies do you like?
- sci-fi / crime / action

Perception of preferences usually takes place in a low dimensional latent space.

$$a_{ij} \approx u_i^T v_j$$

So the rating matrix is approximately low-rank.

$$A \approx UV^T, \quad U \in \mathbb{R}^{m \times k}, V \in \mathbb{R}^{n \times k}$$

databricks

Objective Function

- minimize the reconstruction error

$$\text{minimize } \frac{1}{2}\|A - UV^T\|_F^2$$

- only check observed ratings

$$\text{minimize } \frac{1}{2}\sum_{(i,j)\in\Omega}(a_{ij} - u_i^T v_j)^2$$

databricks

Alternating Least Squares (ALS)

- If we fix U, the objective becomes convex and separable:

$$\text{minimize } \frac{1}{2}\sum_{j}\left(\sum_{i,(i,j)\in\Omega}(a_{ij}-u_i^T v_j)^2\right)$$

- Each sub-problem is a least squares problem, which can be solved in parallel. So we take alternating directions to minimize the objective:
- fix U, solve for V;
- fix V, solve for U.

Complexity

- To solve a least squares problem of size n-by-k, we need $O(n\,k^2)$ time. So the total computation cost is $O(nnz\,k^2)$, where nnz is the total number of ratings.

- We take the normal equation approach in ALS

$$A^T A x = A^T b$$

- Solving each subproblem requires $O(k^2)$ storage. We call LAPACK's routine to solve this problem.

ALS Implementation in MLlib

How to scale to 100,000,000,000 ratings?

databricks™

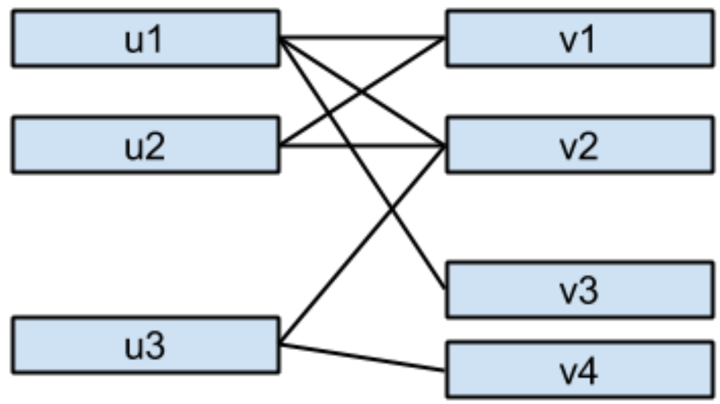
Communication Cost

The most important factor of implementing an algorithm in parallel is the communication cost.

To make ALS scale to billions of ratings, millions of users/items, we have to distribute ratings (A), user factors (U), and item factors (V). How?

- all-to-all
- block-to-block
- …

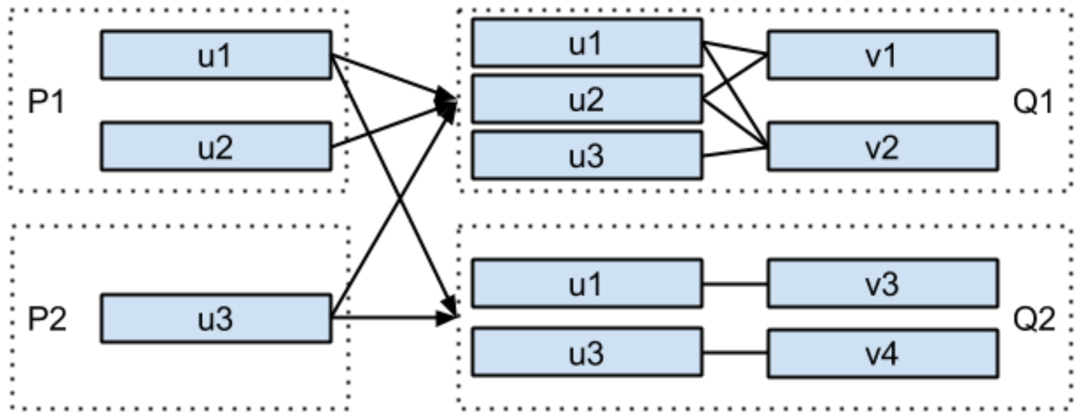
Communication: All-to-All



- users: u1, u2, u3; items: v1, v2, v3, v4
- shuffle size: O(nnz k) (nnz: number of nonzeros, i.e., ratings)
- sending the same factor multiple times

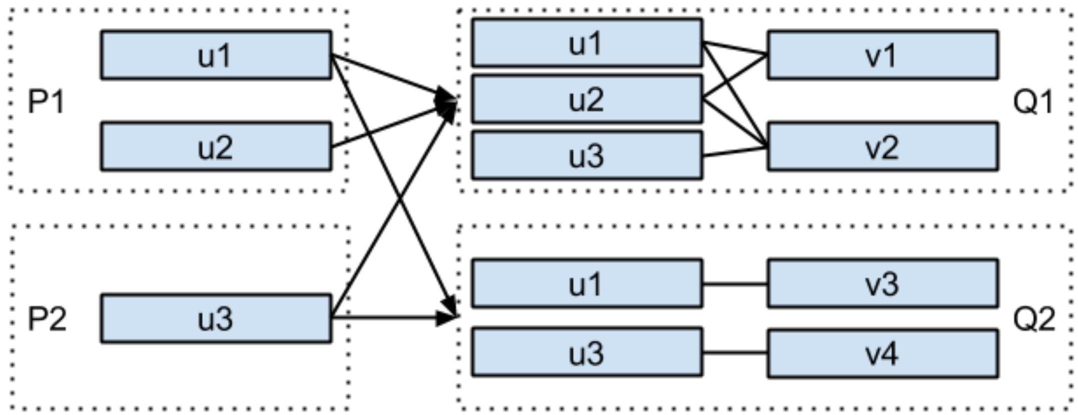
databricks

Communication: Block-to-Block



- OutBlocks (P1, P2)
  - for each item block, which user factors to send
- InBlocks (Q1, Q2)
  - for each item, which user factors to use

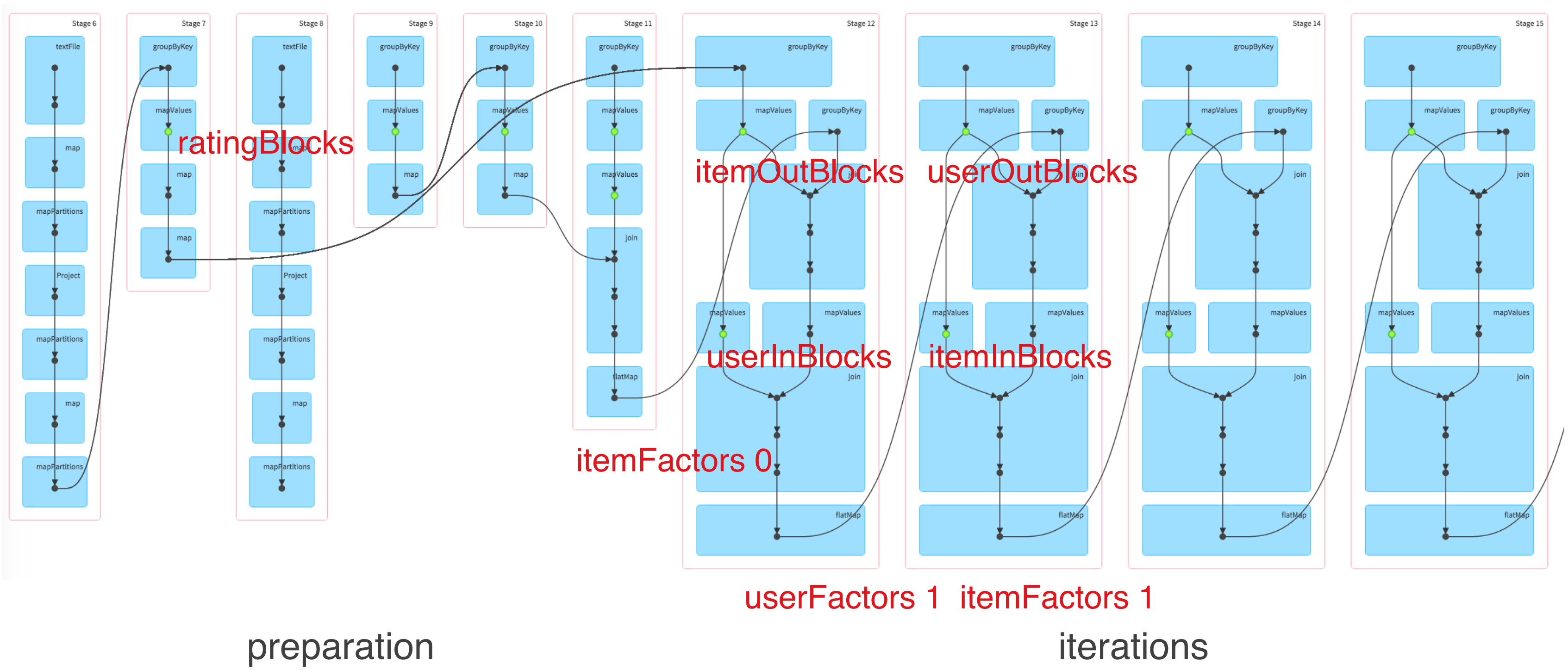
Communication: Block-to-Block



- Shuffle size is significantly reduced.
- We cache two copies of ratings — InBlocks for users and InBlocks for items.

databricks

DAG Visualization of an ALS Job



Compressed Storage for InBlocks

$$[(v_1, u_1, a_{11}), (v_2, u_1, a_{12}), (v_1, u_2, a_{21}), (v_2, u_2, a_{22}), (v_2, u_3, a_{32})]$$

Array of rating tuples

- huge storage overhead
- high garbage collection (GC) pressure

databricks

Compressed Storage for InBlocks

$$([v_1, v_2, v_1, v_2, v_2], [u_1, u_1, u_2, u_2, u_3], [a_{11}, a_{12}, a_{21}, a_{22}, a_{32}])$$

Three primitive arrays

- low GC pressure
- constructing all sub-problems together
  - $O(n_j k^2)$ storage

Compressed Storage for InBlocks

$$([v_1, v_1, v_2, v_2, v_2], [u_1, u_2, u_1, u_2, u_3], [a_{11}, a_{21}, a_{12}, a_{22}, a_{32}])$$

Primitive arrays with items ordered:

- solving sub-problems in sequence:
  - $O(k^2)$ storage
- TimSort

databricks

Compressed Storage for InBlocks

$$([v_1, v_2], [0, 2, 5], [u_1, u_2, u_1, u_2, u_3], [a_{11}, a_{21}, a_{12}, a_{22}, a_{32}])$$

Compressed items:

- no duplicated items
- map lookup for user factors

Compressed Storage for InBlocks

$$([v1, v2], [0, 2, 5], [0|0, 0|1, 0|0, 0|1, 1|0], [a_{11}, a_{21}, a_{12}, a_{22}, a_{32}])$$

Store block IDs and local indices instead of user IDs. For example, u3 is the first vector sent from P2.

Encode (block ID, local index) into an integer

- use higher bits for block ID
- use lower bits for local index
- works for ~4 billions of unique users/items

01 | 00 0000 0000 0000

Avoid Garbage Collection

We use specialized code to replace the following:

- initial partitioning of ratings

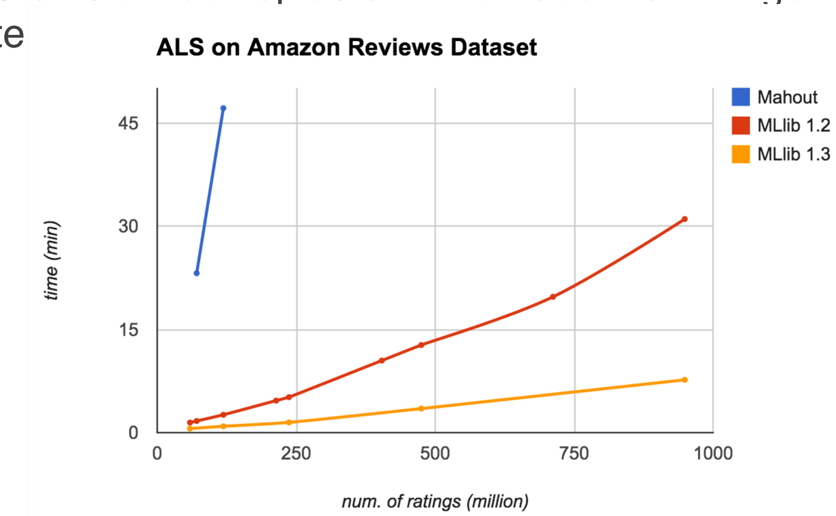
```
ratings.map { r =>
  ((srcPart.getPartition(r.user), dstPart.getPartition(r.item)), r)
}.aggregateByKey(new RatingBlockBuilder)(
    seqOp = (b, r) => b.add(r),
    combOp = (b0, b1) => b0.merge(b1.build()))
  .mapValues(_.build())
```

- map IDs to local indices

```
dstIds.toSet.toSeq.sorted.zipWithIndex.toMap
```

Amazon Reviews Dataset

- Amazon Reviews: ~6.6 million users, ~2.2 million items, and ~30 million ratings
- Tested ALS on stacked copies on a 16-node m3.2xlarge cluster with rank=10, ite



Storage Comparison

	1.2	1.3/1.4
userInBlock	941MB	277MB
userOutBlock	355MB	65MB
itemInBlock	1380MB	243MB
itemOutBlock	119MB	37MB

databricks

Spotify Dataset

- Spotify: 75+ million users and 30+ million songs
- Tested ALS on a subset with ~50 million users, ~5 million songs, and ~50 billion ratings.
  - thanks to Chris Johnson and Anders Arpteg
- 32 r3.8xlarge nodes (~$10/hr with spot instances)
- It took 1 hour to finish 10 iterations with rank 10.
  - 10 mins to prepare in/out blocks
  - 5 mins per iteration

ALS Implementation in MLlib

- Save communication by duplicating data
- Efficient storage format
- Watch out for GC
- Native LAPACK calls

Future Directions

- Leverage on Project Tungsten to save some specialized code that avoids GC.
- Solve issues with really popular items.
- Explore other recommendation algorithms, e.g., factorization machine.

Thank you.

- Spark: http://spark.apache.org
- Databricks: http://databricks.com

databricks™