

SCALING FACTORIZATION MACHINES ON APACHE SPARK WITH PARAMETER SERVERS

Nick Pentreath

Principal Engineer, IBM



About

- About me
 - @MLnick
 - Principal Engineer at IBM working on machine learning & Spark
 - Apache Spark PMC
 - Author of *Machine Learning with Spark*

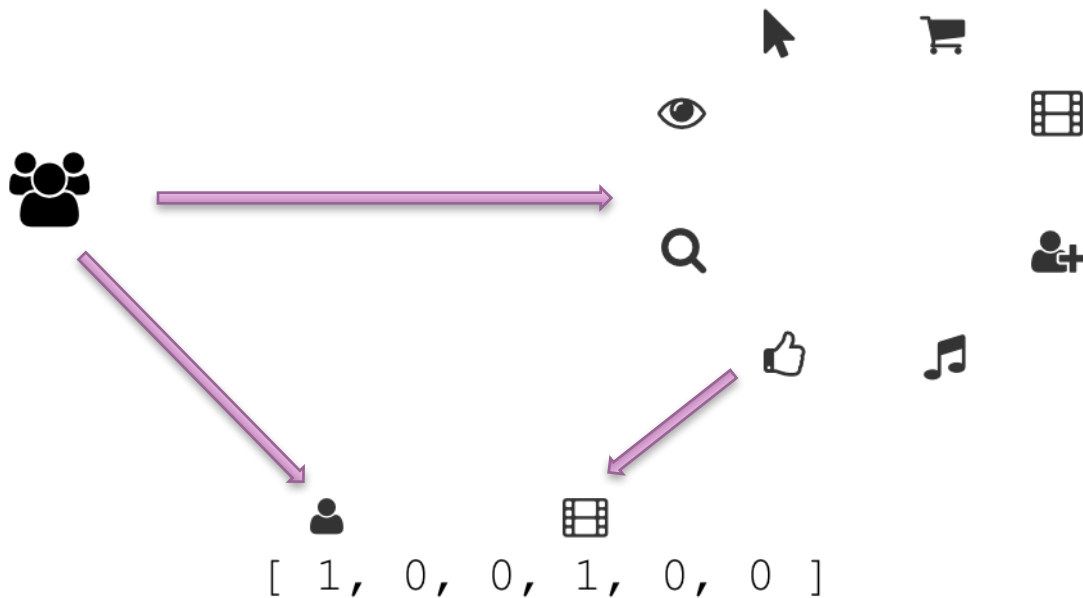
Agenda

- Brief Intro to Factorization Machines
- Distributed FMs with Spark and Glint
- Results
- Challenges
- Future Work

FACTORIZATION MACHINES

Factorization Machines

Feature interactions



Factorization Machines

Linear Models

$$w_0 + \sum_{i=1}^n w_i x_i \quad \longrightarrow \quad w_0 + \underbrace{[w_u, 0, 0, w_i, \dots]}_{\text{Bias terms}}$$

The diagram illustrates the transformation of a standard linear model into a form that includes bias terms. On the left, the linear model is represented as $w_0 + \sum_{i=1}^n w_i x_i$. A large purple arrow points to the right, where the model is shown as $w_0 + [w_u, 0, 0, w_i, \dots]$. Above the first element w_u is a user icon, and above the fourth element w_i is a film strip icon. A purple bracket underneath the vector $[w_u, 0, 0, w_i, \dots]$ is labeled "Bias terms".

Not expressive
enough!

Factorization Machines

Polynomial Regression

$$w_0 + \sum_{i=1}^n w_i x_i + \sum_{i=1}^n \sum_{j=i+1}^n w_{ij} x_i x_j \quad \Rightarrow \quad w_0 + \underbrace{[w_u, 0, 0, w_i, \dots, w_{ui}, \dots]}_{\text{Bias terms} \quad \text{Interaction term}}$$

The diagram illustrates the mapping from a polynomial regression model to a Factorization Machine model. The polynomial regression model on the left consists of a bias term w_0 , a linear term $\sum_{i=1}^n w_i x_i$, and a quadratic interaction term $\sum_{i=1}^n \sum_{j=i+1}^n w_{ij} x_i x_j$. A purple arrow points to the Factorization Machine model on the right, which is represented as $w_0 + [w_u, 0, 0, w_i, \dots, w_{ui}, \dots]$. Above the vector components are icons: a person icon above w_u , a film strip icon above the first zero, and a person-film strip icon above w_{ui} . Brackets below the vector identify the first three components as "Bias terms" and the subsequent components as the "Interaction term".

$n \ll d$

$O(d^2)$

Factorization Machines

Factorization Machine

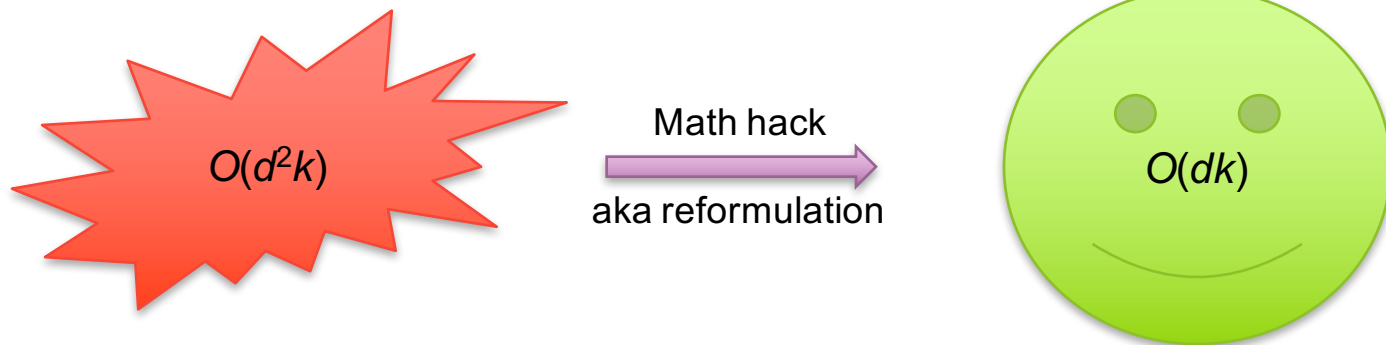
$$w_0 + \sum_{i=1}^n w_i x_i + \sum_{i=1}^n \sum_{j=i+1}^n \langle \vec{v}_i \vec{v}_j \rangle x_i x_j \Rightarrow w_0 + \underbrace{[w_u, 0, 0, w_i, \dots]}_{\text{Bias terms}} + \underbrace{[\dots, \langle \mathbf{v}_u \mathbf{v}_i \rangle, \dots]}_{\text{Factorized interaction term}}$$

The diagram illustrates the transformation of a standard linear model with pairwise interactions into a Factorization Machine model. The left side shows the traditional form with bias terms w_i and interaction terms $\langle \vec{v}_i \vec{v}_j \rangle$. The right side shows the Factorization Machine form, where the bias terms are separated from the factorized interaction terms. The interaction term $\langle \mathbf{v}_u \mathbf{v}_i \rangle$ is represented by a user icon and a movie icon, indicating a user-item interaction.

$O(d^2k)$

Factorization Machines

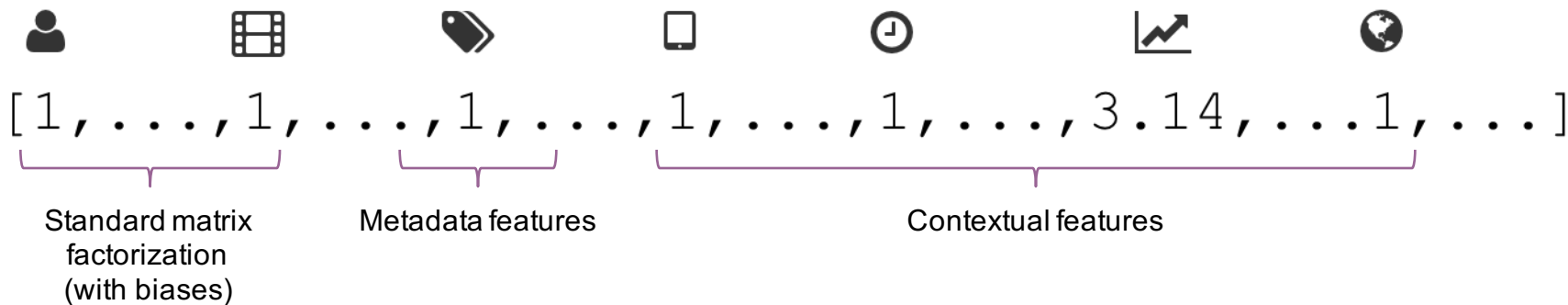
Factorization Machine



Not convex, but efficient to train using SGD, coordinate descent, MCMC

Factorization Machines

Factorization Machine

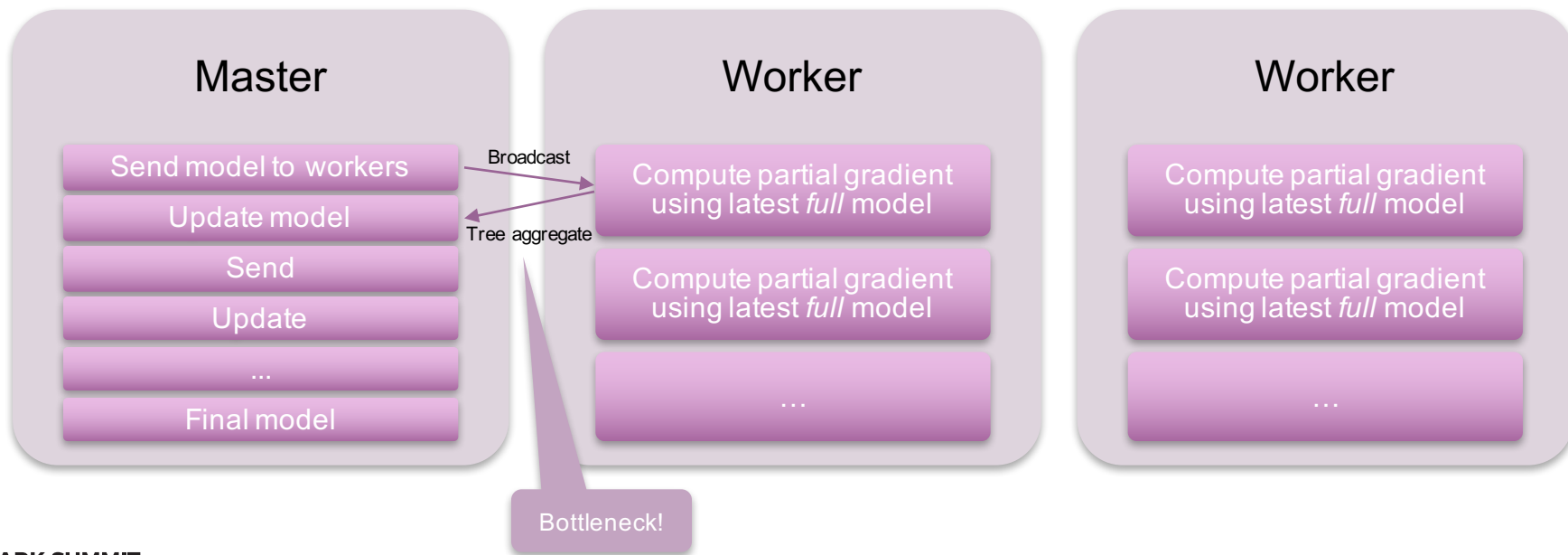


Model size can still be very large! e.g. video sharing, online ads, social networks

DISTRIBUTED FM MODELS ON SPARK

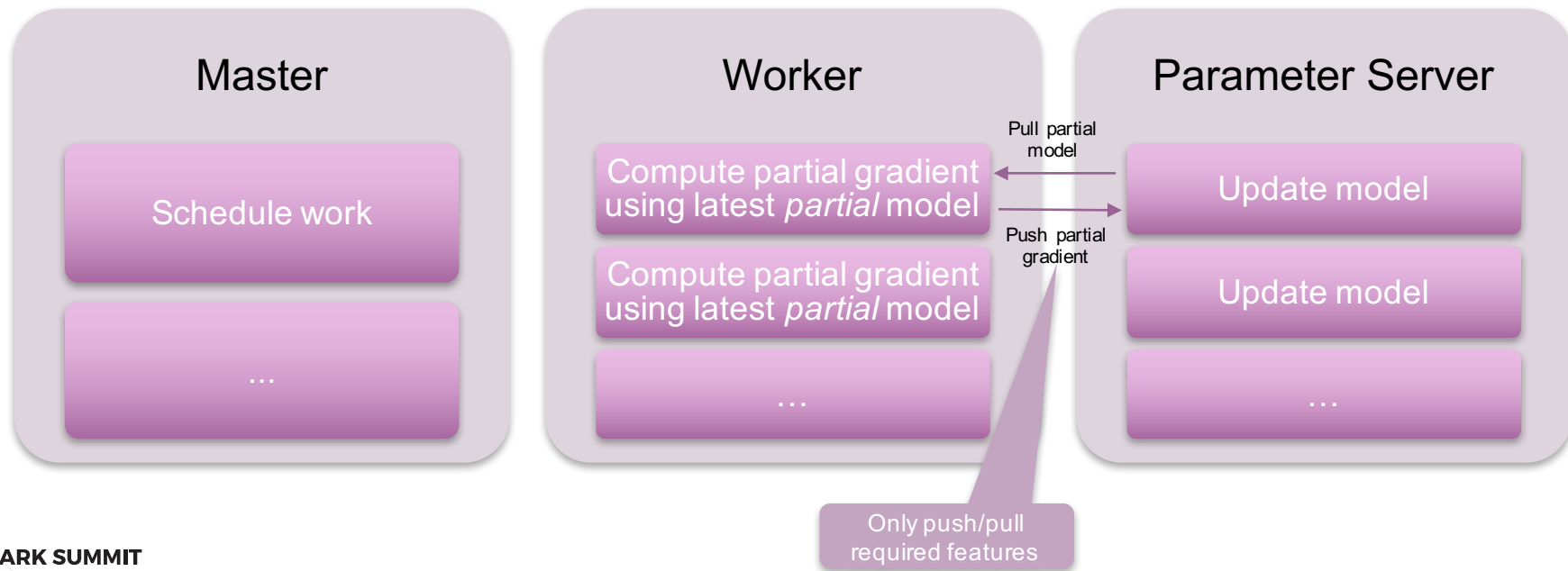
Linear Models on Spark

- Data parallel



Parameter Servers

- Model & data parallel



Distributed FMs

- spark-libFM
 - Uses old MLlib `GradientDescent` and `LBFGS` interfaces
- DiFacto
 - Async SGD implementation using parameter server (ps-lite)
 - Adagrad, L1 regularization, frequency-adaptive model-size
- Key is that most real-world datasets are *highly sparse* (especially high-cardinality categorical data), e.g. online ads, social network, recommender systems
- Workers only need access to a small piece of the model

Glint is a parameter server built using Akka
For more info, see Rolf's talk at 17:15!

GlintFM

- Procedure:

1. Construct Glint Client
2. Create distributed parameters
3. Pre-compute required feature indices (per partition)
4. Iterate:
 - Pull partial model (blocking)
 - Compute partial gradient & update
 - Push partial update to parameter servers (can be async)
5. Done!

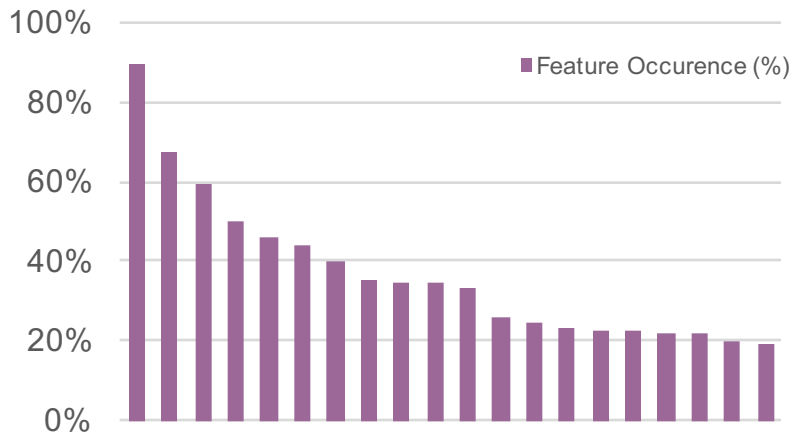
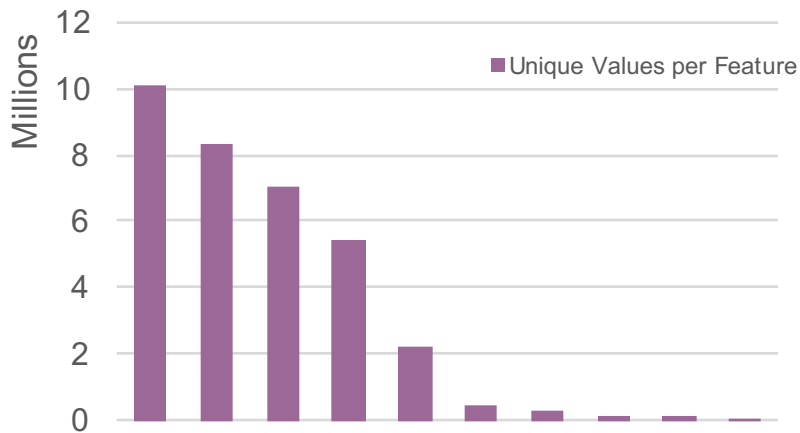
```
val client = Client(config)
val w = client.vector[Double](d)
val V = client.matrix[Double](d, k)
// training
train.foreachPartition { iter =>
  // compute partition statistics
  val localKeys = { ... }
  // iterate
  for (i <- 1 to numIterations) {
    // pull latest model for l local keys
    val localW = w.pull(localKeys) // 1 x l vector
    val localV = V.pull(localKeys) // l x k matrix
    // compute gradient
    partitionData.foldLeft(new FMAggregator(...)) { case (agg, features, label) =>
      agg.add(features, label, localW, localV)
    }
    // compute and push update
    val updates = ...
    w.push(...)
    V.push(...)
  }
}
```

RESULTS

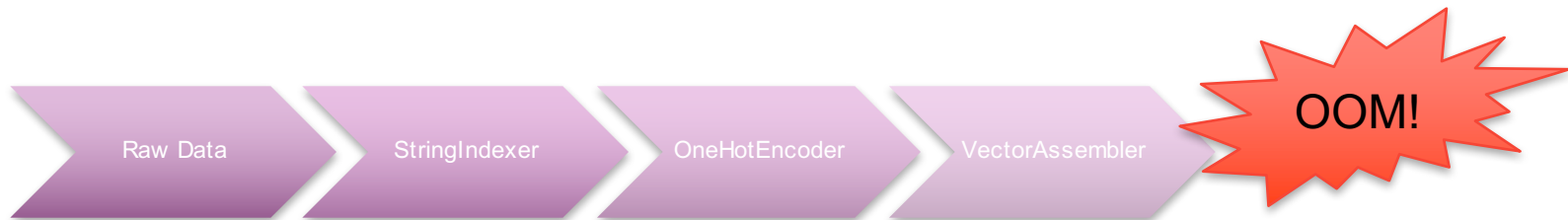
Data

Criteo Display Advertising Challenge Dataset

- 45m examples, 34m unique features, 48 nnz /example



Feature Extraction



label	i1	i2	i3	i4	i5	i6	i7	i8	i9
0	1	1	5	0	1382	4	15	2	181
0	2	0	44	1	102	8	2	2	4
0	2	0	1	14	767	89	4	2	245
0	NULL	893	NULL	NULL	4392	NULL	0	0	0
0	3	-1	NULL	0	2	0	3	0	0



label	i1_idx	i1_ohc	features
0	2.0	(152,[2],[1.0])	(273492,[2,153,28...
0	3.0	(152,[3],[1.0])	(273492,[3,152,28...
0	3.0	(152,[3],[1.0])	(273492,[3,152,28...
0	0.0	(152,[0],[1.0])	(273492,[0,923,28...
0	4.0	(152,[4],[1.0])	(273492,[4,154,28...

Feature Extraction

Solution? “Stringify” + CountVectorizer

```
from pyspark.sql import Row
```

```
cols = df.columns
```

```
def convert_row(row):
```

```
    l = row.label
```

```
    i = 1
```

```
    v = []
```

```
    for c in cols[1:]:
```

```
        if row[i] is not None:
```

```
            v.append("%s=%s" % (c, row[i]))
```

```
            i += 1
```

```
    return Row(label=l, raw=v)
```

```
Row(i1=u'1', i2=u'1', i3=u'5', i4=u'0', i5=u'1382',... )
```

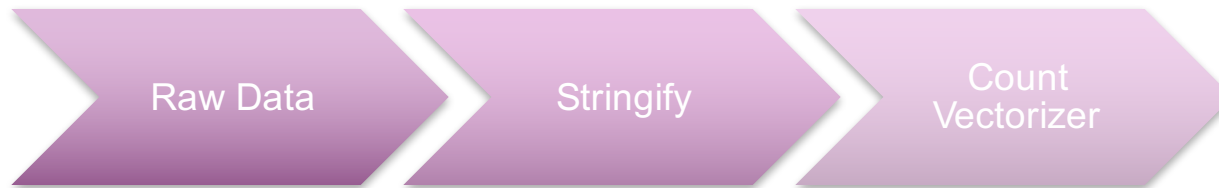


Convert set of
String features into
Seq[String]

```
Row(raw=[u'i1=1', u'i2=1', u'i3=5', u'i4=0', u'i5=1382', ...])
```

```
df_stringified = spark.createDataFrame(df.rdd.map(lambda row: convert_row(row)))
```

Feature Extraction

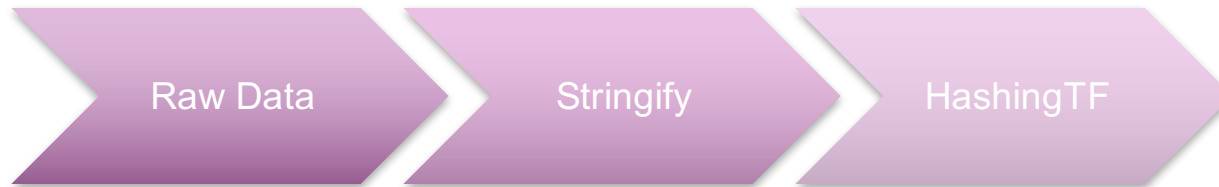


label	i1	i2	i3	i4	i5	i6	i7	i8	i9
0	1	1	5	0	1382	4	15	2	181
0	2	0	44	1	102	8	2	2	4
0	2	0	1	14	767	89	4	2	245
0	NULL	893	NULL	NULL	4392	NULL	0	0	0
0	3	-1	NULL	0	2	0	3	0	0



label	raw	features
0	[i1=1, i2=1, i3=5...]	(273531,[0,1,2,3,...]
0	[i1=2, i2=0, i3=4...]	(273531,[0,1,2,3,...]
0	[i1=2, i2=0, i3=1...]	(273531,[0,3,4,6,...]
0	[i1=NULL, i2=893,...]	(273531,[0,1,2,3,...]
0	[i1=3, i2=-1, i3=...]	(273531,[0,1,2,3,...]

Feature Extraction

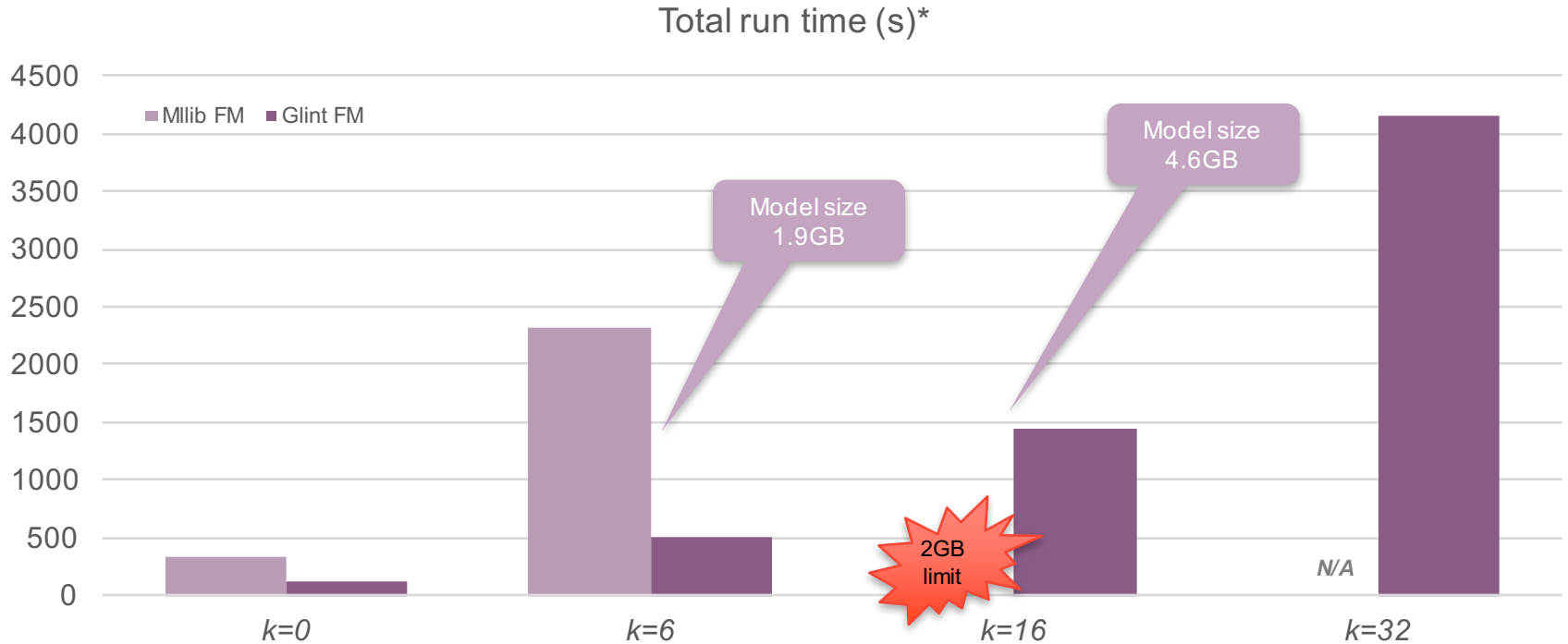


label	i1	i2	i3	i4	i5	i6	i7	i8	i9
0	1	1	5	0	1382	4	15	2	181
0	2	0	44	1	102	8	2	2	4
0	2	0	1	14	767	89	4	2	245
0	NULL	893	NULL	NULL	4392	NULL	0	0	0
0	3	-1	NULL	0	2	0	3	0	0



label	raw	features
0	[i1=1, i2=1, i3=5...]	(262144,[2411,726...
0	[i1=2, i2=0, i3=4...]	(262144,[5352,934...
0	[i1=2, i2=0, i3=1...]	(262144,[14069,15...
0	[i1=NULL, i2=893,...]	(262144,[4201,693...
0	[i1=3, i2=-1, i3=...]	(262144,[6935,140...

Performance



Performance

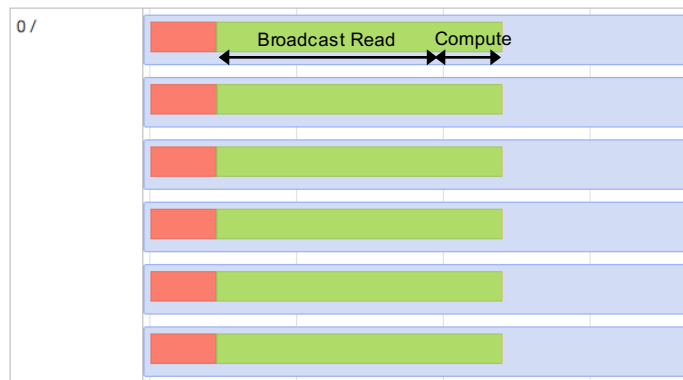
Details for Stage 23 (Attempt 0)

Total Time Across All Tasks: 41 min
Locality Level Summary: Process local: 48
Input Size / Records: 18.9 GB / 36671573
Shuffle Write: 4.1 GB / 48

Gradient
computation

- ▶ DAG Visualization
- ▶ Show Additional Metrics
- ▼ Event Timeline
- ☐ Enable zooming

Scheduler Delay Executor Computing Time Getting Result Time
Task Deserialization Time Shuffle Write Time
Shuffle Read Time Result Serialization Time



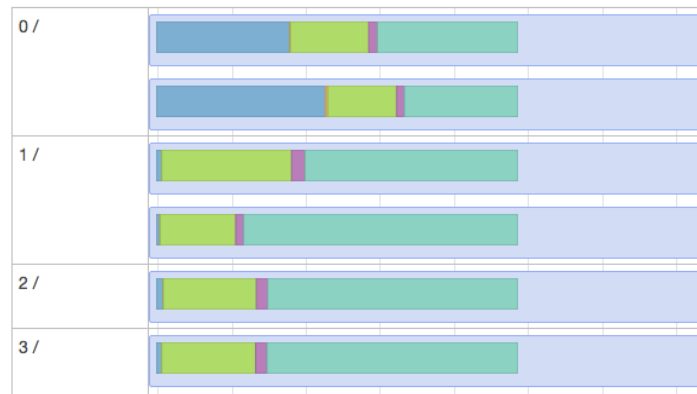
Details for Stage 24 (Attempt 0)

Total Time Across All Tasks: 2.6 min
Locality Level Summary: Node local: 6
Shuffle Read: 4.1 GB / 48

Aggregation
& collect

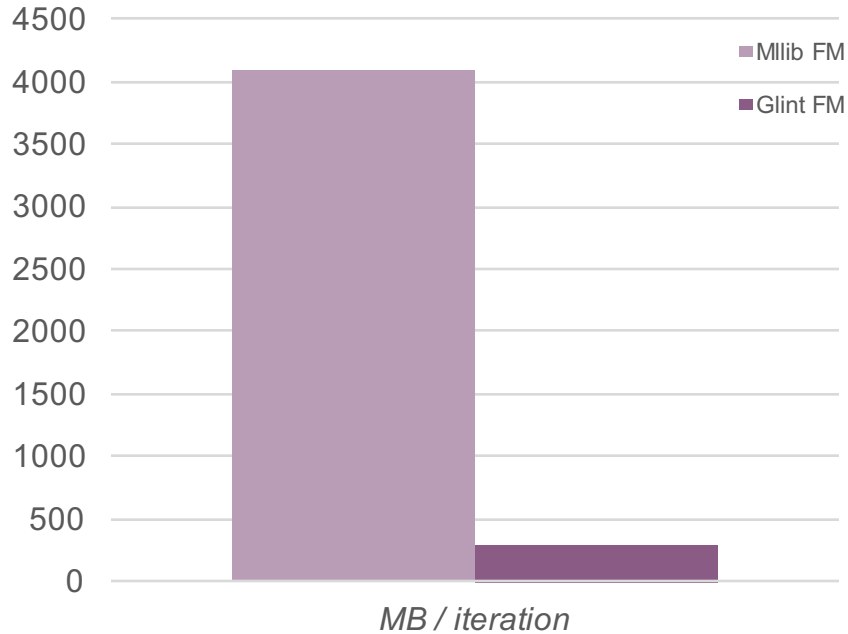
- ▶ DAG Visualization
- ▶ Show Additional Metrics
- ▼ Event Timeline
- ☐ Enable zooming

Scheduler Delay Executor Computing Time Getting Result Time
Task Deserialization Time Shuffle Write Time
Shuffle Read Time Result Serialization Time

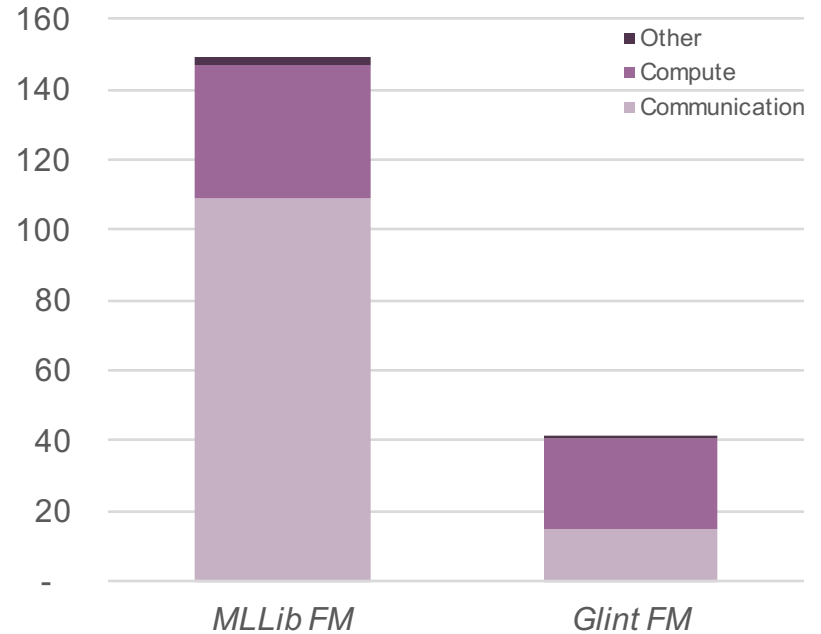


Performance

Data Transfer



Median time per iteration (s)



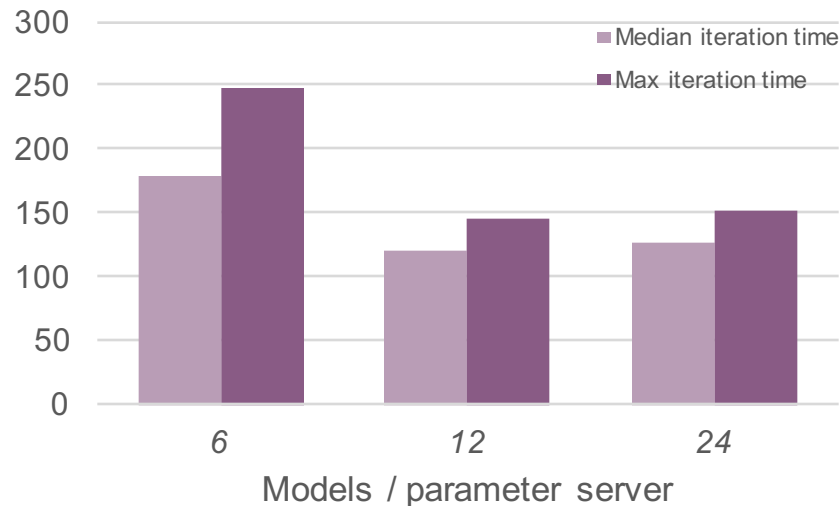
CHALLENGES & FUTURE WORK

Challenges

- Tuning configuration
 - Glint - models/server, message size, Akka frame size
 - Spark - data partitioning (can be seen as “mini-batch” size)
- Lack of server-side processing in Glint
 - For L1 regularization, adaptive sparsity, Adagrad
 - These result in better performance, faster execution
- Backpressure / concurrency control

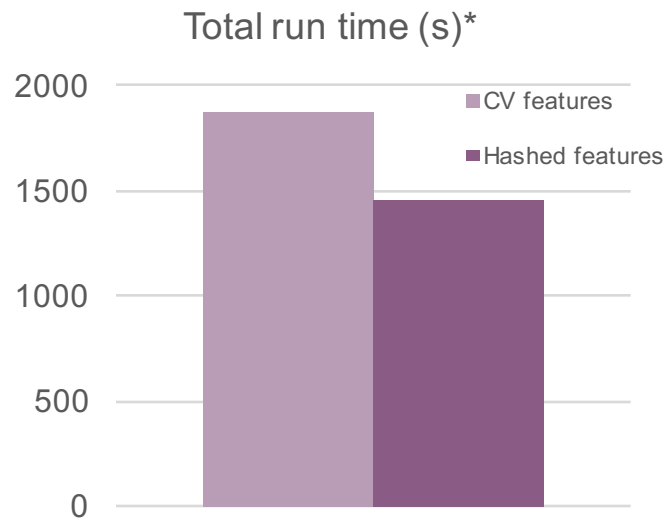
Challenges

- Tuning models / server



Challenges

- Index partitioning for “hot features”
 - `CountVectorizer` for features leads to hot spots & straggler tasks due to sorting by occurrence
 - `OneHotEncoder` OOMed... but can also face this problem
 - Spreading out features is critical (used feature hashing)



* $k = 16$, 10 iterations, 48 partitions, fit intercept

Future Work

- Glint enhancements
 - Add features from DiFacto, i.e. L1 regularization, Adagrad & memory-adaptive k
 - Requires support for UDFs on the server
 - Built-in backpressure (Akka Artery / Streams?)
 - Key caching – 2x decrease in message size
- Mini-batch SGD within partitions
- Distributed solvers for ALS, MCMC, CD
- Relational data / block structure formulation
 - www.vldb.org/pvldb/vol6/p337-rendle.pdf

References

- Factorization Machines
 - <http://www.csie.ntu.edu.tw/~b97053/paper/Rendle2010FM.pdf>
 - <https://github.com/ibayer/fastFM>
 - www.libfm.org
 - <https://github.com/zhengruifeng/spark-libFM>
 - <https://github.com/scikit-learn-contrib/polylearn>
- DiFacto
 - <https://github.com/dmlc/difacto>
 - www.cs.cmu.edu/~yuxiangw/docs/fm.pdf
- Glint / parameter servers
 - <https://github.com/rjagerman/glint>
 - <https://github.com/dmlc/ps-lite>

THANK YOU.

@MLnick

github.com/MLnick/glint-fm
spark.tc

