CSE 234
Data Systems for Machine Learning

Arun Kumar

Topic 1: Classical ML Training at Scale

Chapters 2, 5, and 6 of ML Sys book
Academic ML 101

“Classical” ML

**Generalized Linear Models** (GLMs); from statistics

**Bayesian Networks**; inspired by causal reasoning

**Decision Tree-based**: CART, Random Forest, Gradient-Boosted Trees (GBT), etc.; inspired by symbolic logic

**Support Vector Machines** (SVMs); inspired by psychology

**Artificial Neural Networks** (ANNs): Multi-Layer Perceptrons (MLPs), Convolutional NNs (CNNs), Recurrent NNs (RNNs), Transformers, etc.; inspired by brain neuroscience
# Real-World ML 101

The vast majority of ML applications use off-the-shelf ML methods!

**Tree learners**

- Linear or Logistic Regression: 80.3%
- Decision Trees or Random Forests: 74.1%
- Gradient Boosting Machines (xgboost, lightgbm, etc): 59.5%
- Convolutional Neural Networks: 39.6%
- Bayesian Approaches: 28.7%
- Dense Neural Networks (MLPs, etc): 27.6%
- Recurrent Neural Networks: 26.7%
- Transformer Networks (BERT, gpt-3, etc): 17.1%
- Generative Adversarial Networks: 7.6%
- Evolutionary Approaches: 5.8%
- Other: 3.3%
- None: 2.4%

Scalable ML Training in the Lifecycle

Data acquisition
Data preparation
Training & Inference
Model Selection
Serving
Monitoring

ML/Al + Data Systems Infrastructure

Source → Build → Deploy

Data Scientist/ML Engineer
Figure 1: Only a small fraction of real-world ML systems is composed of the ML code, as shown by the small black box in the middle. The required surrounding infrastructure is vast and complex.
What is a Machine Learning (ML) System?

A data processing system (aka data system) for mathematically advanced data analysis operations, i.e., inferential, predictive, or generative:

- Statistical analysis; ML, deep learning (DL); data mining (domain-specific applied ML + feature eng.)
- High-level APIs to express ML computations over (large) datasets
- **Execution engine** to run ML computations efficiently and in a scalable manner
But what exactly does it mean for an ML system to be “scalable”?
Outline

- Basics of Scaling ML Computations
- Scaling ML to On-Disk Files
- Layering ML on Scalable Data Systems
- Custom Scalable ML Systems
- Advanced Issues in ML Scalability
Background: Memory Hierarchy

- **CPU**
  - Access Speed: ~100GB/s
  - Cycles: 100s
  - Price: ~$2/MB

- **Cache**
  - Access Speed: ~10GB/s
  - Cycles: 10^5 - 10^6
  - Price: ~$5/GB

- **Main Memory**
  - Access Speed: ~GB/s
  - Cycles: 10^7 - 10^8
  - Price: ~$200/TB

- **Flash Storage**
  - Access Speed: ~200MB/s
  - Capacity: ~10TBs
  - Price: ~$40/TB

- **Magnetic Hard Disk Drive (HDD)**
  - Access Speed: ~GB/s
  - Capacity: ~10TBs
  - Price: ~$200/TB
Q: What does this program do when run with ‘python’?
(Assume tmp.csv is in current working directory)

```python
import pandas as p
m = p.read_csv('tmp.csv', header=None)
s = m.sum().sum()
print(s)
```

`1,2,3`  
`4,5,6`
Memory Hierarchy in Action

Rough sequence of events when program is executed

Processor

- CU
- ALU

- Registers
  - ‘21’

Caches

- CSV
  - ‘21’

Commands interpreted

Bus

I/O for Display

Monitor

I/O for code
- tmp.py

Store; Retrieve

I/O for data
- tmp.csv

Computations done by Processor

DRAM

- CSV
  - ‘21’

Store; Retrieve

Q: What if this does not fit in DRAM?
ML systems that do not require the (training) dataset to fit entirely in main memory (DRAM) of one node

Conversely, if the system *thrashes* when data file does not fit in RAM, it is not scalable

**Basic Idea:** Split data file (virtually or physically) and *stage reads* (and writes) of pages to DRAM and processor
Scalable ML Systems

4 main approaches to scale ML to large data:

- **Single-node disk**: Paged access from file on local disk
- **Remote read**: Paged access from disk(s) over network
- **Distributed memory**: Fits on a cluster’s total DRAM
- **Distributed disk**: Fits on a cluster’s full set of disks
Evolution of Scalable ML Systems

- **1980s**
  - Sas
  - In-RDBMS ML Systems
  - Scalability
- **Mid 1990s**
  - R
  - ML on Dataflow Systems
  - Mid 1990s to Late 2000s
  - Manageability
- **Late 1990s to Mid 2000s**
  - Parameter Server
  - Early 2010s
  - Developability
- **Mid 2010s**
  - Tree Learning Systems
  - Late 2000s to Mid 2010s
  - Usability
- **Late 2000s to Early 2010s**
  - TensorFlow
  - ML System Abstractions
  - Late 2010s to Onward
- **Late 2010s Onward**
  - Cloud ML/AI Services
  - ML Platforms and Feature Stores
Major Existing ML Systems

General ML Libraries:

In-memory:  
- scikit-learn
- R

Disk-based files:  
- S, SAS
- DASK

Layered on RDBMS/Spark:  
- MADlib
- Apache Spark
- MLlib

Cloud-native:  
- Amazon SageMaker

“AutoML” platforms:  
- DataRobot
- H2O.ai

Tree Learning Systems:  
- XGBoost
- LightGBM

Deep Learning Systems:  
- TensorFlow
- PyTorch
Outline

❖ Basics of Scaling ML Computations
❖ Scaling ML to On-Disk Files
❖ Layering ML on Scalable Data Systems
❖ Custom Scalable ML Systems
❖ Advanced Issues in ML Scalability
Basic Idea: Split data file (virtually or physically) and stage reads (and writes) of pages to DRAM and processor

- To scale an ML program’s computations, split them up to operate over “chunks” of data at a time
- *How* to split up an ML program this way can be non-trivial!
  - Depends on *data access pattern* of the algorithm
  - A large class of ML algorithms do just *sequential scans* for iterative numerical optimization
Numerical Optimization in ML

❖ Many regression and classification models in ML are formulated as a (constrained) minimization problem
❖ E.g., logistic and linear regression, linear SVM, etc.
❖ Aka “Empirical Risk Minimization” (ERM)

\[ \mathbf{w}^* = \arg \min_{\mathbf{w}} \sum_{i=1}^{n} l(y_i, f(\mathbf{w}, x_i)) \]

❖ GLMs define hyperplanes and use f() that is a scalar function of distances:

\[ \mathbf{w}^T x_i \]
Batch Gradient Descent for ML

\[ L(w) = \sum_{i=1}^{n} l(y_i, f(w, x_i)) \]

- For many ML models, loss function \( l() \) is **convex**; so is \( L() \)
- But closed-form minimization is typically infeasible
- **Batch Gradient Descent:**
  - Iterative numerical procedure to find an optimal \( w \)
  - Initialize \( w \) to some value \( w^{(0)} \)
  - Compute **gradient**:
    \[ \nabla L(w^{(k)}) = \sum_{i=1}^{n} \nabla l(y_i, f(w^{(k)}, x_i)) \]
  - Descend along gradient:
    (Aka **Update Rule**) \[ w^{(k+1)} \leftarrow w^{(k)} - \eta \nabla L(w^{(k)}) \]
  - Repeat until we get close to \( w^* \), aka **convergence**
Batch Gradient Descent for ML

\[ L(w) \]

\[ w^{(1)} \leftarrow w^{(0)} - \eta \nabla L(w^{(0)}) \]
\[ w^{(2)} \leftarrow w^{(1)} - \eta \nabla L(w^{(1)}) \]

- Learning rate is a **hyper-parameter** selected by user or “AutoML” tuning procedures
- Number of iterations/epochs of BGD also hyper-parameter
The data-intensive computation in BGD is the gradient.

In scalable ML, dataset D may not fit in DRAM.

Model \( w \) is typically small and DRAM-resident.

Gradient is like SQL SUM over vectors (one per example).

At each epoch, 1 filescan over D to get gradient.

Update of \( w \) happens normally in DRAM.

Monitoring across epochs for convergence needed.

Loss function \( L() \) is also just a SUM in a similar manner.

\[
\nabla L(w^{(k)}) = \sum_{i=1}^{n} \nabla l(y_i, f(w^{(k)}, x_i))
\]

Q: What SQL op is this reminiscent of?

Gradient is like SQL SUM over vectors (one per example).

At each epoch, 1 filescan over D to get gradient.

Update of \( w \) happens normally in DRAM.

Monitoring across epochs for convergence needed.

Loss function \( L() \) is also just a SUM in a similar manner.
Basic Idea: Split data file (virtually or physically) and *stage reads* (and writes) of pages to DRAM and processor.

Suppose OS Cache can hold 4 pages of file.

Process wants to read file’s pages one by one and then discard: aka “*filescan*” access pattern.

- Read P1
- Read P2
- Read P3
- Read P4
- Read P5
- Read P6

Cache is full; replace old pages.
Scaling BGD to Disk

- Sequential scan to read pages from disk to DRAM
- Modern DRAM sizes can be 10s of GBs; so we read a “chunk” of file at a time (say, 1000s of pages)
- Compute partial gradient on each chunk and add all up

\[ \nabla L(w^{(k)}) = \sum_{i=1}^{n} \nabla l(y_i, f(w^{(k)}, x_i)) \]

<table>
<thead>
<tr>
<th></th>
<th>Y</th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1b</td>
<td>1c</td>
<td>1d</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2b</td>
<td>2c</td>
<td>2d</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3b</td>
<td>3c</td>
<td>3d</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4b</td>
<td>4c</td>
<td>4d</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
DaskML’s Scalable DataFrame

**Basic Idea:** Split data file (virtually or physically) and **stage reads** (and writes) of pages to DRAM and processor

- Dask DF scales to on-disk files by splitting it as a bunch of Pandas DF under the hood
- Dask API is a wrapper around Pandas API to scale ops to splits and put all results together

https://docs.dask.org/en/latest/dataframe.html
Scaling with Remote Reads

**Basic Idea:** Split data file (virtually or physically) and **stage reads** (and writes) of pages to DRAM and processor

- Similar to scaling to disk but instead read pages/chunks over the network from remote disk/disks (e.g., from S3)
- Good in practice for a one-shot *filesccan* access pattern
- For iterative ML, repeated reads over network
- Can combine with caching on local disk / DRAM
- Increasingly popular for cloud-native ML workloads
Stochastic Gradient Descent for ML

❖ Two key cons of BGD:
  ❖ Slow to converge to optimal (too many epochs)
  ❖ Costly full scan of D for each update of $w$
❖ Stochastic GD (SGD) mitigates both issues
❖ **Basic Idea:** Use a *sample* (called **mini-batch**) of D to approximate gradient instead of “full batch” gradient
  ❖ *Without replacement* sampling
  ❖ Randomly shuffle D before each epoch
  ❖ One pass = sequence of mini-batches
❖ SGD works well for *non-convex* loss functions too, unlike BGD; “workhorse” of scalable ML
Data Access Pattern of Scalable SGD

\[ W^{(t+1)} \leftarrow W^{(t)} - \eta \nabla \tilde{L}(W^{(t)}) \]

\[ \nabla \tilde{L}(w^{(k)}) = \sum_{(y_i, x_i) \in B \subset D} \nabla l(y_i, f(w^{(k)}, x_i)) \]

Sample mini-batch from dataset without replacement

Original dataset

Randomized dataset

Epoch 1

Seq. scan

Epoch 2 ...

ORDER BY RAND()
Mini-batch gradient computations: 1 filescan per epoch
Update of $w$ happens in DRAM
During filescan, count number of examples seen and update per mini-batch
  Typical Mini-batch sizes: 10s to 1000s
  Orders of magnitude more updates than BGD!
Random shuffle is not trivial to scale; requires “external
merge sort” (roughly 2 scans of file)
ML practitioners often shuffle dataset only once up
front; good enough in most cases in practice
Handling pages directly is so low-level! Is there a higher-level way to scale ML?
Outline

❖ Basics of Scaling ML Computations
❖ Scaling ML to On-Disk Files
❖ Layering ML on Scalable Data Systems
❖ Custom Scalable ML Systems
❖ Advanced Issues in ML Scalability
Outline

❖ Basics of Scaling ML Computations
❖ Scaling ML to On-Disk Files
❖ Layering ML on Scalable Data Systems
❖ In-RDBMS ML
❖ ML on Dataflow Systems
❖ Custom Scalable ML Systems
❖ Advanced Issues in ML Scalability
Mature software systems to scale to larger-than-DRAM data
Invoking ML in SQL

- In-RDBMS ML tools enable invocation of ML training and inference from within SQL console
  - Both datasets and models are stored as relational tables
  - CREATE MODEL analogue to CREATE TABLE
  - Data/business analysts use it (but not data scientists)
- 3 main approaches to “in-RDBMS” ML:
  
  **ML in Pure SQL**
  
  **ML as User Defined Functions**
  
  **Call out to Custom ML System**
ML in Pure SQL

❖ Possible to implement a few ML algorithms in pure SQL
  ❖ OLS linear regression, logistic reg., k-means clustering
  ❖ Technically, need to add auxiliary library in SQL for math functions and feature engineering (e.g., one-hot encoding)

CREATE VIEW ols AS
SELECT pseudo_inverse(A) * b as beta_star,
  (transpose(b) * (pseudo_inverse(A) * b) - sum_y2/count) -- SSR
  / (sum_yy - sumy2/n) -- TSS
  as r_squared
FROM (SELECT sum(transpose(d.vector) * d.vector) as A,
  sum(d.vector * y) as b,
  sum(y)^2 as sum_y2, sum(y^2) as sum_yy,
  count(*) as n
  FROM design d)
  ols_aggs;

❖ Pros: All in SQL
❖ Cons: Very few ML algorithms fit in SQL; tuple-level overhead too high vs. other approaches

https://dsf.berkeley.edu/papers/vldb09-madskills.pdf
Sort of “cheating” because ML computations do NOT run in RDBMS process space

**Pros:** Utility of SQL console; efficiency of custom ML system

**Cons:** Data duplicated by export from DB
RDBMS User-Defined Aggregates

- Most industrial and open source RDBMSs allow extensibility to their SQL dialect with User-Defined Functions (UDFs)
- User-Defined Aggregate (UDA) is a UDF API to specify custom aggregates over the whole dataset

Example with SQL AVG:

1. **Initialize**
   - Start by setting up “agg. state” in DRAM
   - (S, C): (Partial sum, partial count)

2. **Transition**
   - RDBMS gives a tuple from table; update agg. state
   - \((S, C) \leftarrow (S, C) + (v_i, 1)\)

3. **Merge**
   - (Optional: In parallel RDBMS, combine agg. states of workers)
   - \((S', C') \leftarrow \sum_{worker \ j} (S_j, C_j)\)

4. **Finalize**
   - Post-process agg. state and return result
   - Return \(S'/C'\)
### Bismarck: SGD as RDBMS UDA

- **SGD epoch implemented as UDA for in-RDBMS execution**
  - **Initialize**: Allocate memory for model $W(t)$ and mini-batch gradient stats
  - **Transition**: Given tuple with $(y,x)$, compute gradient and update stats
    - If mini-batch limit hit, update model and reset stats
  - **Merge**: (Optional: applies only to parallel RDBMS)
    - “Combine” model parameters from indep. workers
  - **Finalize**: Return model parameters

- **Data-intensive computation scaled automatically by RDBMS**

- **Commands for shuffling, running multiple epochs, checking convergence, and validation/test error measurements issued from an external controller written in Python**
Many ML algorithms fit within Bismarck’s template

<table>
<thead>
<tr>
<th>Analytics Task</th>
<th>Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logistic Regression (LR)</td>
<td>$\sum_i \log(1 + \exp(-y_i w^T x_i)) + \mu</td>
</tr>
<tr>
<td>Classification (SVM)</td>
<td>$\sum_i (1 - y_i w^T x_i)^+ + \mu</td>
</tr>
<tr>
<td>Recommendation (LMF)</td>
<td>$\sum_{(i,j) \in \Omega} (L^T_i R_j - M_{ij})^2 + \mu</td>
</tr>
<tr>
<td>Labeling (CRF) [48]</td>
<td>$\sum_k \left[ \sum_j w_j F_j(y_k, x_k) - \log Z(x_k) \right]$</td>
</tr>
<tr>
<td>Kalman Filters</td>
<td>$\sum_{t=1}^T</td>
</tr>
<tr>
<td>Portfolio Optimization</td>
<td>$p^T w + w^T \Sigma w \text{ s.t. } w \in \Delta$</td>
</tr>
</tbody>
</table>

LR_Transition(ModelCoef \* w, Example e) { ...  
  wx = Dot_Product(w, e.x);  
  sig = Sigmoid(-wx \* e.y);  
  c = stepsize \* e.y \* sig;  
  Scale_And_Add(w, e.x, c); ... }  

Transition function is where the bulk of the ML logic is; a few lines of code for SGD updates
Distributed SGD via RDBMS UDA

**Q:** How does the RDBMS parallelize the (SGD) UDA?

- Data is pre-sharded across workers in parallel RDBMSs
- Initialize and Transition run independently on shard
- Merge “combines” model params from workers; tricky since SGD epoch is not an *algebraic agg.* like SUM/AVG
- Common heuristic: “Model Averaging”

\[
W^{(t)} = \frac{1}{n} \sum_{i=1}^{n} W_{i}^{(t)}
\]

- Affects convergence
- Works OK for GLMs
Model Averaging for distributed SGD has poor convergence for non-convex/ANN models.

- Too many epochs, typically poor ML accuracy
- Model sizes can be too large (even 10s of GBs) for UDA’s aggregation state
- UDA’s Merge step is choke point at scale (100s of workers)
  - Bulk Synchronous Parallelism (BSP) parallel RDBMSs
The MADlib Library

❖ A decade-old library of scalable statistical and ML procedures on PostgreSQL and Greenplum (parallel RDBMS)
❖ Many procedures are UDAs; some are written in pure SQL
❖ All can be invoked from SQL console
❖ RDBMS can be used for ETL (extract, transform, load) of features

https://madlib.apache.org/docs/latest/index.html
<table>
<thead>
<tr>
<th>Pros:</th>
<th>Cons:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usability:</strong></td>
<td>Some data analysts like ML from SQL console</td>
</tr>
<tr>
<td><strong>Manageability:</strong></td>
<td>In-situ data governance and security/auth. of RDBMSs</td>
</tr>
<tr>
<td><strong>Efficiency:</strong></td>
<td>Faster in some cases; typically slower due to API restrictions</td>
</tr>
<tr>
<td><strong>Scalability:</strong></td>
<td>Massively parallel processing of RDBMSs like Greenplum</td>
</tr>
<tr>
<td><strong>Developability:</strong></td>
<td>SQL-based ETL</td>
</tr>
</tbody>
</table>
Outline

❖ Basics of Scaling ML Computations
❖ Scaling ML to On-Disk Files
❖ Layering ML on Scalable Data Systems
   ❖ In-RDBMS ML
   ❖ ML on Dataflow Systems
❖ Custom Scalable ML Systems
❖ Advanced Issues in ML Scalability
The MapReduce Abstraction

- Programming model for writing programs on sharded data + distributed system architecture for processing large data
- **Map and Reduce** are terms/ideas from functional PL
- Developer only implements the logic of Map and Reduce
- System implementation handles orchestration of data distribution, parallelization, etc. under the hood
The MapReduce Abstraction

- **Standard example**: count word occurrences in a doc corpus
- **Input**: A set of text documents (say, webpages)
- **Output**: A dictionary of unique words and their counts

```
function map (String docname, String doctext):
    "Hmmm, sounds suspiciously familiar ..."
    for each word w in doctext :
        emit (w, 1)

function reduce (String word, Iterator partialCounts) :
    sum = 0
    for each pc in partialCounts :
        sum += pc
    emit (word, sum)
```

Part of MapReduce API
How MapReduce Works

Parallel flow of control and data during MapReduce execution:

The overall MapReduce word count process

Under the hood, each **Mapper** and **Reducer** is a separate process; Reducers face barrier synchronization (BSP). Fault tolerance achieved using **data replication**
What is Hadoop then?

❖ Open-source system implementation with MapReduce as prog. model and HDFS as distr. filesystem

❖ Map() and Reduce() functions in API; input splitting, data distribution, shuffling, fault tolerance, etc. all handled by the Hadoop library under the hood

❖ Mostly superseded by the Spark ecosystem these days although HDFS is still the base
Abstract Semantics of MapReduce

- **Map()**: Operates *independently* on one “record” at a time
  - Can *batch* multiple data examples on to one record
  - Dependencies across Mappers *not* allowed
  - Can *emit* 1 or more key-value pairs as output
  - Data types of inputs and outputs can be different!

- **Reduce()**: Gathers all Map output pairs across machines with same key into an Iterator (list)
  - *Aggregation* function applied on Iterator and output final

- **Input Split**:  
  - Physical-level split/shard of dataset that batches multiple examples to one file “block” (~128MB default on HDFS) 
  - Custom Input Splits can be written by appl. user
Benefits of MapReduce

- **Goal:** Higher level abstraction for parallel data processing

- **Key Benefits:**
  - Out-of-the-box scalability and fault tolerance
  - Map() and Reduce() can be highly general; no restrictions on data types; easier ETL
  - Free and OSS (Hadoop)

- **New burden on users:** Cast data-intensive computation to the Map() + Reduce() API
  - But libraries exist in many PLs to mitigate coding pains: Java, C++, Python, R, Scala, etc.
Q: How would you do the word counting in RDBMS / in SQL?

❖ **First step:** Transform text docs into relations and load:
   Part of the Extract-Transform-Load (ETL) stage
   Suppose we pre-divide each document into words and have the schema: **DocWords** (DocName, Word)

❖ **Second step:** a single, simple SQL query!

```
SELECT   Word, COUNT (*)
FROM      DocWords
GROUP BY  Word
[ORDER BY Word]
```

Parallelism, scaling, etc. done by RDBMS under the hood
RDBMS UDA vs. MapReduce

- **Aggregation state**: data structure computed (independently) by workers and unified by manager

- **Initialize**: Set up info./initialize RAM for agg. state; runs independently on each worker

- **Transition**: Per-tuple function run by worker to update its agg. state; analogous to Map() in MapReduce

- **Merge**: Function that combines agg. states from workers; run by manager after workers done; analogous to Reduce()

- **Finalize**: Run once at end by manager to return final result
BDG via MapReduce

- Assume data is sharded; a map partition has many examples
- Initial model $W$ read from file on HDFS by each mapper

```java
function map (String datafile) :
    Read $W$ from known file on HDFS
    Initialize $G = 0$
    For each tuple in datafile:
        $G +=$ per-example gradient on (tuple, $W$)
    emit ($G$)

function reduce (Iterator partialGradients) :
    $FullG = 0$
    for each $G$ in partialGradients :
        $FullG += G$
    emit ($FullG$)
```
SGD (Averaging) via MapReduce

- Similar to BGD but Model Averaging across map partitions

```java
function map (String datafile) :
Read Wj from known file on HDFS
For each mini-batch in datafile:
    Initialize G = 0
    For each example in that mini-batch:
        G += gradient on that example with Wj
        Wj = Update Wj with G
    emit (Wj)

function reduce (Iterator workerWeights) :
    AvgW = 0
    for each Wj in workerWeights :
        AvgW += Wj
        AvgW /= workerWeights.length()
    emit (AvgW)
```
Apache Spark

- Extended *dataflow programming* model to subsume MapReduce, most relational operators
  - Inspired by Python Pandas style of function calls for ops
- Unified system to handle relations, text, etc.; support more general distributed data processing
- Uses distributed memory for caching; faster than Hadoop
- New fault tolerance mechanism using lineage, not replication
- From UC Berkeley AMPLab; commercialized as Databricks
Spark’s Dataflow Programming Model

Transformations are relational ops, MR, etc. as functions

Actions are what force computation; aka *lazy evaluation*

<table>
<thead>
<tr>
<th>Transformations</th>
<th>Operation</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(f : T ⇒ U)</code></td>
<td><code>RDD[T] ⇒ RDD[U]</code></td>
<td></td>
</tr>
<tr>
<td><code>filter(f : T ⇒ Bool)</code></td>
<td><code>RDD[T] ⇒ RDD[T]</code></td>
<td></td>
</tr>
<tr>
<td><code>flatMap(f : T ⇒ Seq[U])</code></td>
<td><code>RDD[T] ⇒ RDD[U]</code></td>
<td></td>
</tr>
<tr>
<td><code>sample(fraction : Float)</code></td>
<td><code>RDD[T] ⇒ RDD[T]</code> (Deterministic sampling)</td>
<td></td>
</tr>
<tr>
<td><code>groupByKey()</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, Seq[V])]</code></td>
<td></td>
</tr>
<tr>
<td><code>reduceByKey(f : (V, V) ⇒ V)</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, V)]</code></td>
<td></td>
</tr>
<tr>
<td><code>union()</code></td>
<td><code>(RDD[T],RDD[T]) ⇒ RDD[T]</code></td>
<td></td>
</tr>
<tr>
<td><code>join()</code></td>
<td><code>(RDD[(K, V)],RDD[(K, W)]) ⇒ RDD[(K, (V, W))]</code></td>
<td></td>
</tr>
<tr>
<td><code>cogroup()</code></td>
<td><code>(RDD[(K, V)],RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]</code></td>
<td></td>
</tr>
<tr>
<td><code>crossProduct()</code></td>
<td><code>(RDD[T],RDD[U]) ⇒ RDD[(T, U)]</code></td>
<td></td>
</tr>
<tr>
<td><code>mapValues(f : V ⇒ W)</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, W)]</code> (Preserves partitioning)</td>
<td></td>
</tr>
<tr>
<td><code>sort(c : Comparator[K])</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, V)]</code></td>
<td></td>
</tr>
<tr>
<td><code>partitionBy(p : Partitioner[K])</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, V)]</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actions</th>
<th>Operation</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count()</code></td>
<td><code>RDD[T] ⇒ Long</code></td>
<td></td>
</tr>
<tr>
<td><code>collect()</code></td>
<td><code>RDD[T] ⇒ Seq[T]</code></td>
<td></td>
</tr>
<tr>
<td><code>reduce(f : (T, T) ⇒ T)</code></td>
<td><code>RDD[T] ⇒ T</code></td>
<td></td>
</tr>
<tr>
<td><code>lookup(k : K)</code></td>
<td><code>RDD[(K, V)] ⇒ Seq[V]</code> (On hash/range partitioned RDDs)</td>
<td></td>
</tr>
<tr>
<td><code>save(path : String)</code></td>
<td>Outputs RDD to a storage system, e.g., HDFS</td>
<td></td>
</tr>
</tbody>
</table>

Word Count Example in Spark

Spark RDD API available in Python, Scala, Java, and R

```python
text_file = sc.textFile("hdfs://...")
counts = text_file.flatMap(lambda line: line.split(" "))
  .map(lambda word: (word, 1))
  .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

```scala
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

```scala
JavaRDD[String> textFile = sc.textFile("hdfs://...");
JavaPairRDD<String, Integer> counts = textFile
  .flatMap(s -> Arrays.asList(s.split(" ")).iterator())
  .mapToPair(word -> new Tuple2<>(word, 1))
  .reduceByKey((a, b) -> a + b);
counts.saveAsTextFile("hdfs://...");
```
Programming ML in MapReduce/Spark

❖ All ML procedures that can be cast as RDBMS UDAs can be cast to MapReduce API of Hadoop or Spark RDD
❖ Most developers find MapReduce easier to use than UDA
❖ Spark integrates better with PyData ecosystem

❖ Apache Mahout is a library of classical ML algorithms written as MapReduce programs for Hadoop; expanded later to DSL
❖ SparkML is a library of classical ML algorithms written using Spark RDD API; common in enterprises for scalable ML

### Tradeoffs of ML on Dataflow Systems

<table>
<thead>
<tr>
<th></th>
<th><strong>Pros:</strong></th>
<th><strong>Cons:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usability:</strong></td>
<td>SparkML integrates well with Python stacks</td>
<td>Not all ML algorithms scaled</td>
</tr>
<tr>
<td><strong>Manageability:</strong></td>
<td>In-situ data governance and security/auth. of</td>
<td>Many ML users may not be familiar with Spark data ETL</td>
</tr>
<tr>
<td></td>
<td>“data lakes”</td>
<td></td>
</tr>
<tr>
<td><strong>Efficiency:</strong></td>
<td>Comparable to in-RDBMS ML; no OLTP interference</td>
<td>Custom ML systems typically faster still</td>
</tr>
<tr>
<td><strong>Scalability:</strong></td>
<td>Massively parallel processing</td>
<td>BSP is a bottleneck for 100+ nodes (asynchrony needed)</td>
</tr>
<tr>
<td><strong>Developability:</strong></td>
<td>SQL- &amp; MR-based ETL</td>
<td>Still somewhat unnatural APIs to write ML</td>
</tr>
<tr>
<td></td>
<td>Less code to write ML</td>
<td></td>
</tr>
</tbody>
</table>
Outline

❖ Basics of Scaling ML Computations
❖ Scaling ML to On-Disk Files
❖ Layering ML on Scalable Data Systems
❖ Custom Scalable ML Systems
❖ Advanced Issues in ML Scalability
Outline

❖ Basics of Scaling ML Computations
❖ Scaling ML to On-Disk Files
❖ Layering ML on Scalable Data Systems
❖ Custom Scalable ML Systems
❖ Parameter Server
❖ GraphLab
❖ XGBoost
❖ Advanced Issues in ML Scalability
Parameter Server for Distributed SGD

- Recall bottlenecks of Model Averaging-based SGD in RDBMS UDA or with MapReduce:
  - BSP becomes a choke point (Merge / Reduce stage)
  - Often poor convergence, especially for non-convex
  - Hard to handle large models
- Parameter Server (PS) mitigates all these issues:
  - Breaks the synchronization barrier for merging: allows *asynchronous updates* from workers to manager
  - Flexible communication frequency: can send updates at every mini-batch or a set of few mini-batches
ParameterServer for Distributed SGD

Multi-server Manager; each server manages a part of $\mathbf{W}^{(t)}$

No sync. for workers or servers

$\nabla \tilde{L}(\mathbf{W}_1^{(t)})$  $\nabla \tilde{L}(\mathbf{W}_2^{(t-1)})$  $\nabla \tilde{L}(\mathbf{W}_n^{(t+1)})$

Push / Pull when ready/needed

Workers send gradients to Manager for updates at each mini-batch (or lower frequency)

❖ Model params may get out-of-sync or stale; but SGD turns out to be remarkably robust—multiple updates/epoch really helps

❖ Communication cost per epoch is higher (per mini-batch)
Programming ML using PS

- Designed mainly for *sparse* feature vectors/updates
  - Easy to parallelize updates to model params
- ML developer recasts ML procedure into two pars: worker-side updates and server-side aggregation
  - Loosely analogous to Map and Reduce, respectively
  - But *more complex* due to flexible update schedules
- Supports 3 consistency models for staleness of updates, with different tradeoffs on efficiency vs accuracy

![Diagram](image)
Systems-level Advances in PS

- Workers and Servers can both be multi-nodes/multi-process; fine-grained task scheduling on nodes
- Range-based partitioning of params for servers
- Timestamped and compressed updates exchanged
- Replication and fault tolerance
# Tradeoffs of Parameter Server

<table>
<thead>
<tr>
<th></th>
<th>Pros:</th>
<th>Cons:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usability:</strong></td>
<td>Supports billions of features and params</td>
<td>Not reproducible; not well integrated with ETL stacks</td>
</tr>
<tr>
<td><strong>Manageability:</strong></td>
<td>In-built fault tolerance</td>
<td>TensorFlow offers it natively; o/w, hard to operate/govern</td>
</tr>
<tr>
<td><strong>Efficiency:</strong></td>
<td>Faster than in-RDBMS ML and ML-on-Dataflow</td>
<td>Not suitable for dense updates; high comm. cost</td>
</tr>
<tr>
<td><strong>Scalability:</strong></td>
<td>Highest; can work with 1000s of nodes</td>
<td>Not suitable for smaller scales due to overheads</td>
</tr>
<tr>
<td><strong>Developability:</strong></td>
<td>Abstracts away many systems scaling issues</td>
<td>Reasoning about ML (in)consistency is hard</td>
</tr>
</tbody>
</table>
Your Strong/Weak Points on PS

❖ (Walked through in class)
Innovativeness and Depth Ratings
Outline

❖ Basics of Scaling ML Computations
❖ Scaling ML to On-Disk Files
❖ Layering ML on Scalable Data Systems
❖ Custom Scalable ML Systems
   ❖ Parameter Server
   ❖ GraphLab
   ❖ XGBoost
❖ Advanced Issues in ML Scalability
Graph-Parallel Algorithms

❖ Some data analytics algorithms (not just ML) operate on graph data and have complex update dependencies

Example: PageRank

\[ R(v) = \frac{\alpha}{n} + (1 - \alpha) \sum_{u \text{ links to } v} w_{u,v} \times R(u) \]

❖ Not a simple sequential access pattern like SGD
❖ If viewed as a table: reads and writes to tuples, with each write depending on all neighboring tuples
❖ Does not scale well with RDBMS/UDA or MapReduce/Spark
The GraphLab Abstraction

❖ “Think like a vertex” paradigm over data graph (V, E, D)
❖ Arbitrary data state associated with vertices and edges
❖ 3-function API: Gather-Apply-Scatter
  ❖ Gather: Collect latest states from neighbors
  ❖ Apply: Vertex-local state update function
  ❖ Scatter: Send local state to neighbors

❖ Original single-node GraphLab assumed shared-memory
❖ Scaled to single-node disk with careful sharding & caching of graph

Distributed GraphLab Execution

- Enabled *asynchronous* updates of vertex and edge states
- Consistency-parallelism tradeoff
- Some algorithms seem robust to such inconsistency
- Sophisticated distributed locking protocols

(b) Consistency Models

(c) Consistency and Parallelism

Outline

❖ Basics of Scaling ML Computations
❖ Scaling ML to On-Disk Files
❖ Layering ML on Scalable Data Systems
❖ Custom Scalable ML Systems
   ❖ Parameter Server
   ❖ GraphLab
      ❖ XGBoost
❖ Advanced Issues in ML Scalability
Decision Tree Data Access Pattern

- CART has complex non-sequential data access patterns
  - Compare candidate splits on each feature at each node
  - Class-conditional aggregates needed per candidate
  - Repartition data for sub-tree growth

- Does not scale well with RDBMS/UDA or MapReduce/Spark

http://docs.h2o.ai/h2o/latest-stable/h2o-docs/variable-importance.html
Decision Tree Ensembles

❖ RandomForest is very popular
  ❖ Just a bunch of independent trees on column subsets
❖ Tree Boosting is a popular adaptive ensemble
  ❖ Construct trees sequentially and weigh them
  ❖ “Weak” learners aggregated to a strong learner
❖ Gradient-Boosted Decision Tree (GBDT) is very popular
  ❖ Also adds trees to ensemble sequentially
  ❖ Real-valued prediction; convex differentiable loss

\[ \hat{y}_i = \phi(x_i) = \sum_{k=1}^{K} f_k(x_i), \quad f_k \in F, \]

\[ \mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k) \]

where \( \Omega(f) = \gamma T + \frac{1}{2} \lambda \|w\|^2 \)
GBDT Data Access Pattern

- More complex non-sequential access pattern than single tree!
  - An “iteration” adds a tree by exploring candidate splits
  - Still needs recursive data re-partitioning
- Key difference: *scoring function* has more statistics (1st and 2nd deriv.); this needs *read-write access* per example

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Stats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>Gi</td>
</tr>
<tr>
<td>X1</td>
<td>Hi</td>
</tr>
<tr>
<td>X2</td>
<td></td>
</tr>
<tr>
<td>X3</td>
<td></td>
</tr>
</tbody>
</table>

- Access pattern over per-example stats is random, depends on split location
- Ideal if whole stats and a whole column (at a time) can fit in RAM
XGBoost

- Custom ML system to scale GBDT to larger-than-RAM data, both single-node disk and on a cluster
- Very popular on tabular data; won many Kaggle contests
- **Key philosophy**: Algorithm-system “co-design”:
  - Make system implementation memory hierarchy-aware based on algorithm’s data access patterns
  - Modify ML algorithmics to better suit system scale
XGBoost: Algorithm-level Ideas

- 2 key changes to make GBDT more scalable
- **Bottleneck**: Computing candidate split stats at scale
  - **Idea**: Approximate stats with weighted quantile sketch
- **Bottleneck**: Sparse feature vectors and missing data
  - **Idea**: Bake in “default direction” for child during learning
XGBoost: Systems-level Ideas

❖ 4 key choices to ensure efficiency at scale
❖ **Goal**: Reduce overhead of evaluating candidates
  ❖ **Idea**: Pre-sort all columns independently
❖ **Goal**: Exploit parallelism to raise throughput
  ❖ **Idea**: Shard data into a column “block” per worker; workers compute local columns’ stats independently

![Diagram showing layout transformation and calculation of $G_L$ and $G_R$](image)
XGBoost: Systems-level Ideas

❖ 4 key choices to ensure efficiency at scale
❖ **Goal**: Mitigate read-write randomness to stats in RAM
   ❖ **Idea**: CPU cache-aware staging of stats to reduce stalls

❖ **Goal**: Scale to on-disk and cluster data
   ❖ **Idea**: Shard blocks further on disk; stage reads to RAM; block-level compression to reduce I/O latency
XGBoost: Scalability Gains

❖ Gains from both algorithmic changes and systems ideas
# Tradeoffs of Custom Distr. ML Sys.

<table>
<thead>
<tr>
<th></th>
<th><strong>Pros:</strong></th>
<th><strong>Cons:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usability:</strong></td>
<td>Suitable for hyper-specialized ML use case</td>
<td>Need to (re)learn new system APIs again</td>
</tr>
<tr>
<td><strong>Manageability:</strong></td>
<td>More feasible in cloud-native / managed env.</td>
<td>Extra overhead to add and maintain in tools ecosystem</td>
</tr>
<tr>
<td><strong>Efficiency:</strong></td>
<td>Often much lower runtimes and costs</td>
<td>Debugging runtime issues needs specialized knowhow</td>
</tr>
<tr>
<td><strong>Scalability:</strong></td>
<td>Often more scalable to larger datasets</td>
<td>Arcane scalability issues may arise, e.g., inconsistency</td>
</tr>
<tr>
<td><strong>Developability:</strong></td>
<td>More amenable if ML algorithm is familiar; new open source/startup communities</td>
<td>Need to (re)learn new implementation APIs and consistency tradeoffs; risk of lower technical support</td>
</tr>
</tbody>
</table>
Your Strong/Weak Points on XGBoost

- (Walked through in class)
Innovativeness and Depth Ratings
Outline

❖ Basics of Scaling ML Computations
❖ Scaling ML to On-Disk Files
❖ Layering ML on Scalable Data Systems
❖ Custom Scalable ML Systems
❖ Advanced Issues in ML Scalability
Advanced Issues in ML Scalability

❖ Streaming/Incremental ML at scale
❖ Scaling Massive Task Parallelism in ML
❖ Pushing ML Through Joins
❖ Larger-than-Memory Models
❖ Models with More Complex Data Access Patterns
❖ Delay-Tolerant / Geo-Distributed / Federated ML
❖ Scaling end-to-end ML Pipelines
Scalable Incremental ML

- Datasets keep growing in many real-world ML applications
- **Incremental ML**: update a learned model using only new data
  - Streaming ML is one variant (near real-time)
- Non-trivial to make all ML algorithms incremental
  - SGD-based procedures are “online” by default; just resume gradient descent on new data
  - ML/data mining folks have studied how to make other ML algorithms incremental; accuracy-runtime tradeoffs
Industrial cloud-native ML requirements:

- Incremental training and model freshness
- Predictable training runtime
- Elasticity and pause-resume
- Trainable on ephemeral (non-archived) data
- Automatic model/hyper-parameter tuning

**Design:** Streaming ML algorithms that fit into a 3-function API of *Initialize-Update-Finalize* (akin to UDA)

- All SGD-based procedures (GLMs, fact. Machines)
- Variants of K-Means, PCA, topic models, forecasting

Scalable Incremental ML: SageMaker

- Parameter Server-based architecture with streaming workers

Massive Task Parallelism: Ray

- Advanced ML applications that use reinforcement learning produce large numbers of short-running tasks
  - Training robots, self-driving cars, etc.
  - ML-based cluster resource management

- Ray is an ML system that automatically scales such tasks from single-node to large clusters
- Tasks can have shared state for control
- Data is replicated/broadcast

Pushing ML Through Joins

- Most real-world tabular/relational datasets are multi-table
- A central fact table with target; many dimension tables

- Key-foreign key joins are ubiquitous to denormalize such data into single table before ML training
- Single table has a lot of data redundancy
- In turn, many ML algorithms end up having a lot of computational redundancy

- Pushing ML computations through joins enables computations directly over the normalized database
Pushing ML Through Joins

❖ Orion first showed how to rewrite ML computations to operate on join input rather than output, aka “factorized ML”
  ❖ GLMs with BGD, L-BFGS, Newton methods
  ❖ Rewritten impl. fits within UDA / MapReduce abstractions

❖ Morpheus offered a unified abstraction based on linear algebra to factorize many ML algorithms in one formalism
  ❖ K-Means clustering, matrix factorization, etc.

https://adalabucsd.github.io/papers/2017_Morpheus_VLDB.pdf
Larger-than-Memory Models

❖ Some ML algorithms may have state that does not fit in RAM
❖ Need to *shard* model too (not just data) to stage updates
❖ Specific to the ML algorithm’s data access patterns

**Example:** Matrix factorization trained with SGD

---

http://www.cs.cmu.edu/~kijungs/etc/10-405.pdf
Outline

❖ Basics of Scaling ML Computations
❖ Scaling ML to On-Disk Files
❖ Layering ML on Scalable Data Systems
❖ Custom Scalable ML Systems
❖ Advanced Issues in ML Scalability
Evaluating “Scalability”

Message: Not just scalability but efficiency at scale matters too; not just speedup curve but time vs. strong [single-node] baselines