Hi!

CS 744: GRAPHX

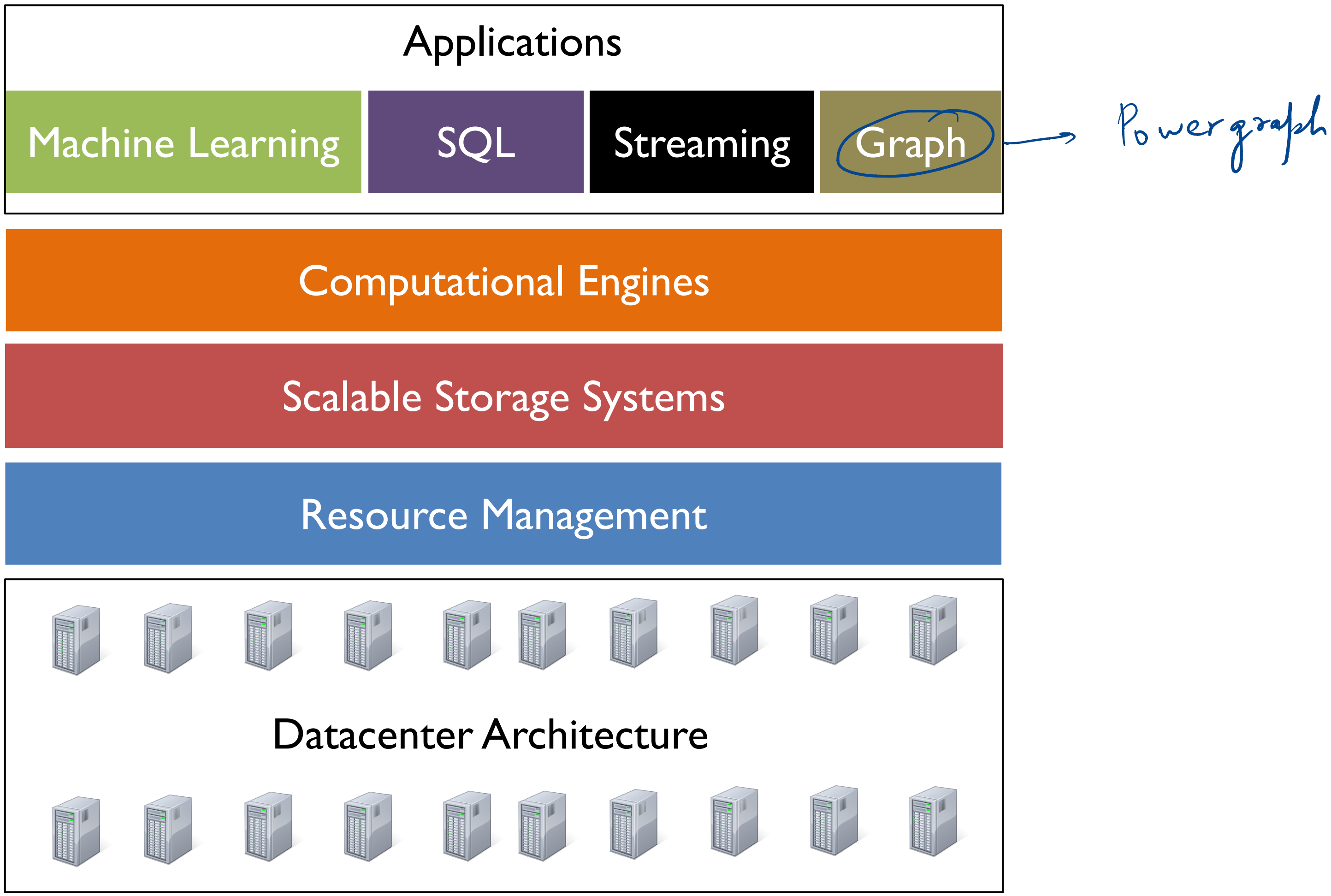
Shivaram Venkataraman

Fall 2021

ADMINISTRIVIA

- Midterm grades today? → Thu office hours ?!
- Course Project: Check in by Nov 30th

↓ Canvas ≈ 1 page update of
what you have done
what are road blocks / challenges


Applications
Machine Learning
SQL
Streaming
Graph
Powergraph
Computational Engines
Scalable Storage Systems
Resource Management
Datacenter Architecture

POWERGRAPH

Programming Model:
Gather-Apply-Scatter

Better Graph Partitioning
with vertex cuts

Distributed execution
(Sync, Async)

What is different from dataflow system e.g., Spark?

→ specialized partitioning
↳ lower communication

→ API was more graph specific
→ easy to express many algorithms

What are some shortcomings?

→ Fault tolerance
↳ checkpoint of all vertices

THIS CLASS

GraphX

Can we efficiently map graph abstractions to dataflow engines?

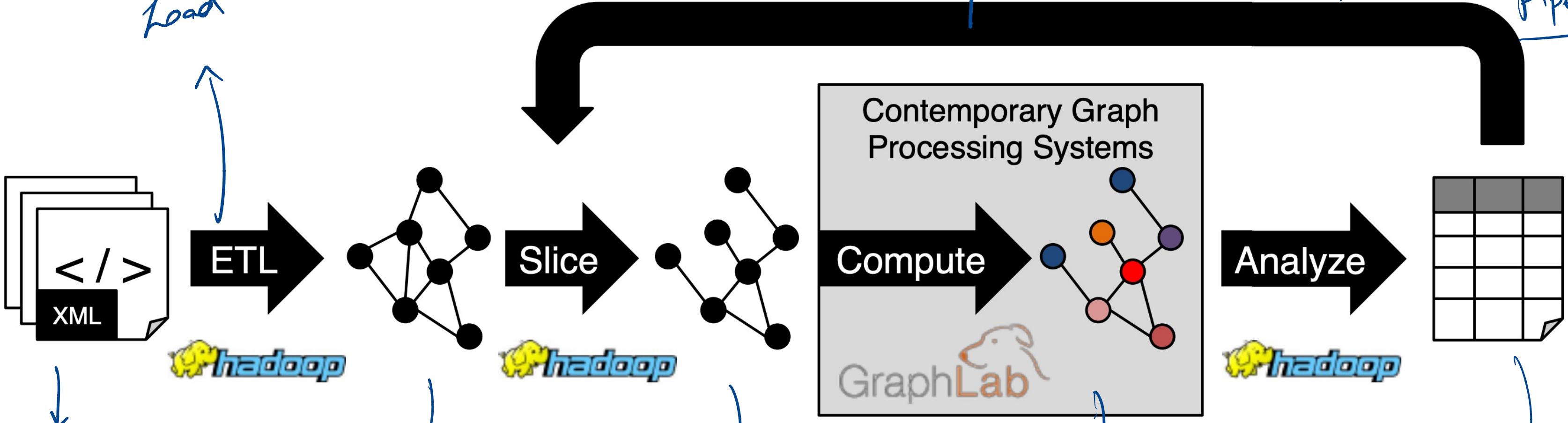
Scalability! But at what COST?

When should we distribute graph processing?

MOTIVATION

Specialized approach
↳ Write out to files in-between systems
→ Modularity → update parts of "pipeline"

Extract, Transform, Load



Input data

Adjacency list ...

users in the US

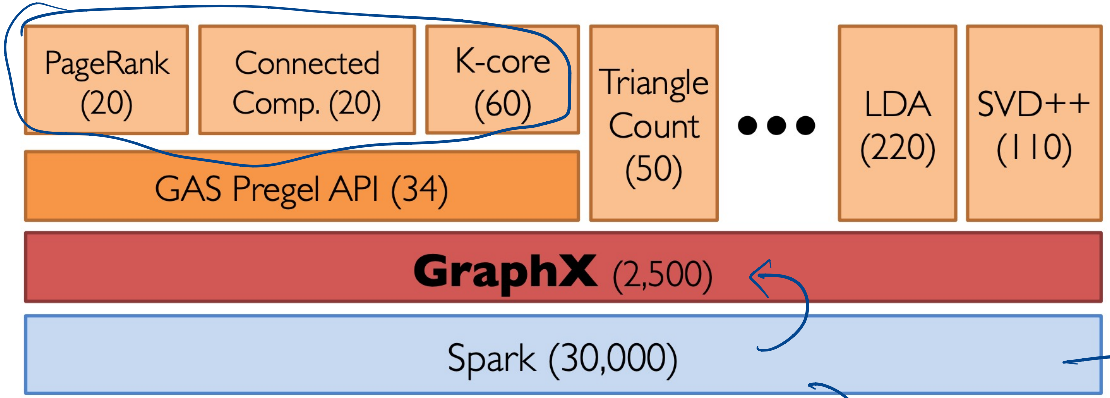
Connected components

visualise or evaluate

SYSTEM OVERVIEW

Unified approach

Advantages?



ETL

Evaluation/ Analysis

Straggler mitigation/ scheduling

→ Hierarchical implementation

↳ Simplifies implementation

→ reuse !

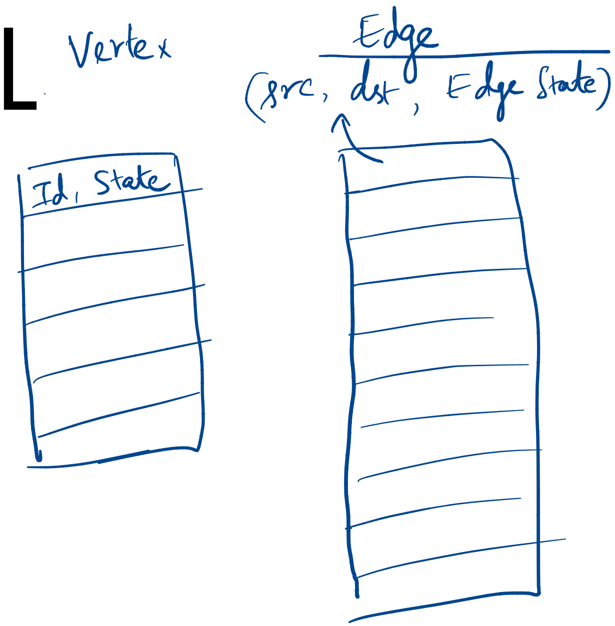
fault tolerance etc.

→ Base abstractions might not be suitable and need retrofit

PROGRAMMING MODEL

```
class Graph[V, E] {
  // Constructor
  def Graph(v: Collection[(Id, V)],
            e: Collection[(Id, Id, E)])
  // Collection views
  def vertices: Collection[(Id, V)]
  def edges: Collection[(Id, Id, E)]
  def triplets: Collection[Triplet]
  // Graph-parallel computation
  def mrTriplets(f: (Triplet) => M,
      sum: (M, M) => M): Collection[(Id, M)]
  // Convenience functions
  def mapV(f: (Id, V) => V): Graph[V, E]
  def mapE(f: (Id, Id, E) => E): Graph[V, E]
  def leftJoinV(v: Collection[(Id, V)],
      f: (Id, V, V) => V): Graph[V, E]
  def leftJoinE(e: Collection[(Id, Id, E)],
      f: (Id, Id, E, E) => E): Graph[V, E]
  def subgraph(vPred: (Id, V) => Boolean,
      ePred: (Triplet) => Boolean)
    : Graph[V, E]
  def reverse: Graph[V, E]
}
```

Constructor



Triplets

S : source
D : dest

(S.ID, D.ID, E, S.V, D.V)

select * from edges
JOIN vertex.ID = edges.source
AND JOIN VERTEX.ID = edges.Dest

MR TRIPLETS

```
mrTriplets(f: (Triplet) => M, sum: (M, M) => M): Collection[(Id, M)]
```

vertex ID ← Sum of all incoming msg

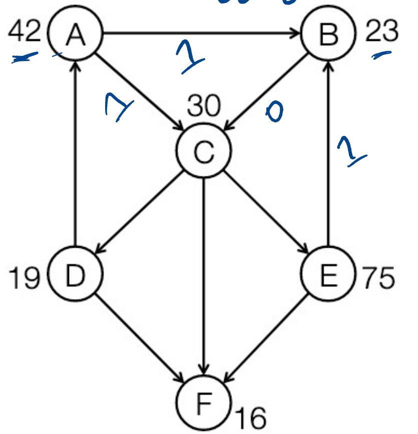
Gather: Edge (S, D, E) → Accumulator M

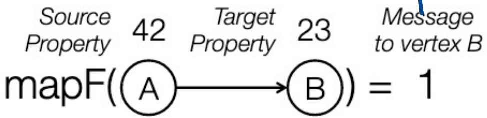
Sum : Sum

Return: Collection with messages aggregated at destination vertex

Joins to produce triplets
map applies f to triplets
groupBy group on dest vertex Id and calls sum on M

→ map operation after mrTriplets to Apply part.





Resulting Vertices

Vertex Id	Property
A	0
B	2
C	1
D	1
E	0
F	3

```
val graph: Graph[User, Double]
def mapUDF(t: Triplet[User, Double]) =
  if (t.src.age > t.dst.age) 1 else 0
def reduceUDF(a: Int, b: Int): Int = a + b
val seniors: Collection[(Id, Int)] =
  graph.mrTriplets(mapUDF, reduceUDF)
```

PREGEL USING GRAPHX

```
def Pregel(g: Graph[V, E],
      vprog: (Id, V, M) => V,
      sendMsg: (Triplet) => M,
      gather: (M, M) => M): = {

  g.mapV((id, v) => (v, halt=false))

  while (g.vertices.exists(v => !v.halt)) {
    val msgs: Collection[(Id, M)] =
        g.subgraph(ePred=(s,d,sP,eP,dP)=>!sP.halt)
         .mrTriplets(sendMsg, gather)

    g = g.leftJoinV(msgs).mapV(vprog)
  }

  return g.vertices
}
```

Think like a vertex

Apply

"Activate" vertex. All vertices are active

Super step

filter only active vertices

Gather

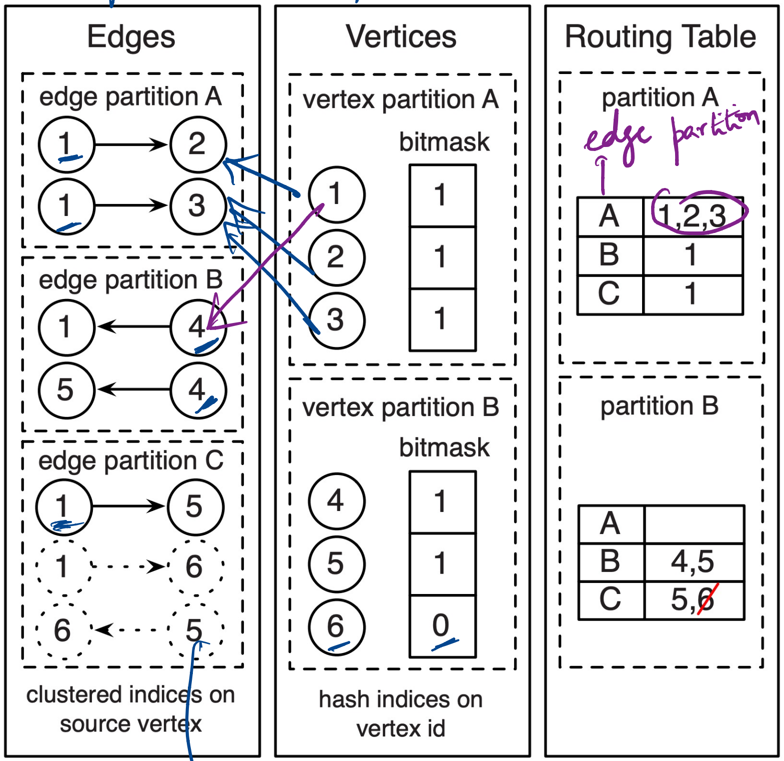
(Id, Vertex State)

msg: Vertex ID → Message

IMPLEMENTING TRIPLETS VIEW

Vertex Cut in Power graph

Hash Partitioner



not active anymore

→ join, map, groupBy for mrTriplets

Join strategy

Send vertices to the edge site

→ More edges than vertices, parallelize on edges

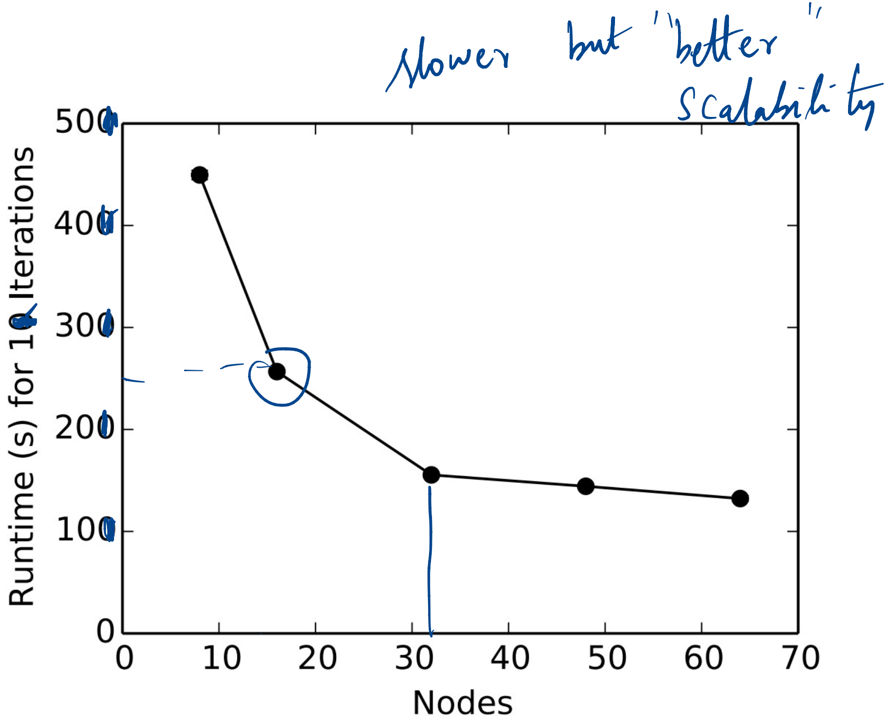
Bitmask tracks active vertices

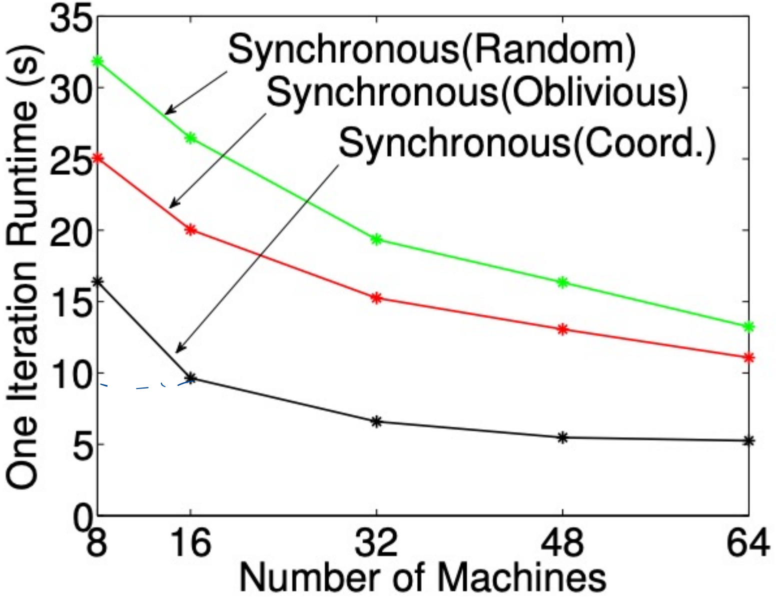
Multicast join

Using routing table

→ Routing table minimizes vertex state sent across the network

SCALABILITY VS. ABSOLUTE PERFORMANCE





GraphX

3x from 8 to 32 machines (4x)

PowerGraph

2.6x from 8 to 32 (4x)

COST: CONFIGURATION THAT OUT-PERFORMS SINGLE THREAD

C#, single threaded program

```
fn PageRank20(graph: GraphIterator, alpha: f32) {
  let mut a = vec![0f32; graph.nodes()];
  let mut b = vec![0f32; graph.nodes()];
  let mut d = vec![0f32; graph.nodes()];

  graph.map_edges(|x, y| { d[x] += 1; });

  for iter in 0..20 {
    for i in 0..graph.nodes() {
      b[i] = alpha * a[i] / d[i];
      a[i] = 1f32 - alpha;
    }

    graph.map_edges(|x, y| { a[y] += b[x]; });
  }
}
```

→ arrays

vertices

graph

scalable system	cores	twitter
GraphLab [10]	128	249s
GraphX [10]	128	419s
Single thread (SSD)	1	300s
Single thread (RAM)	1	275s

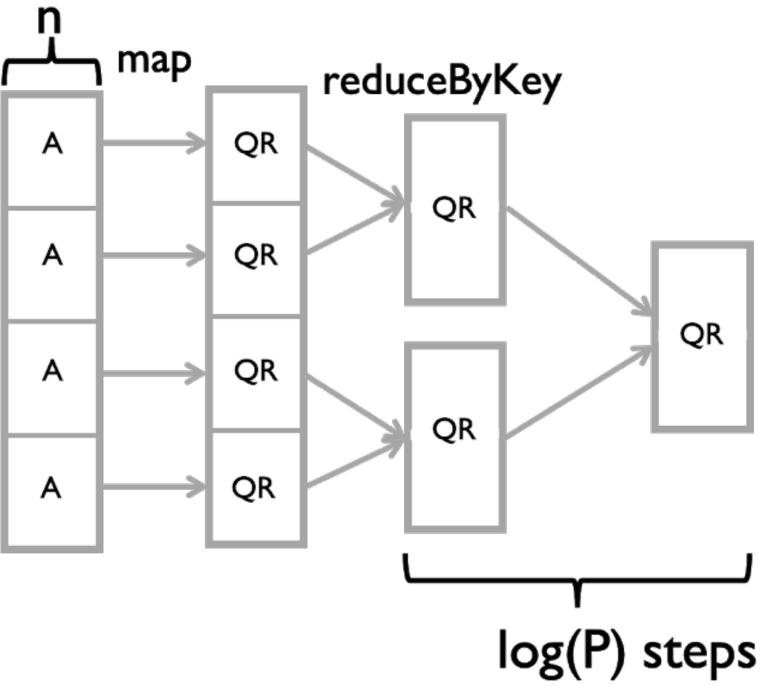
DISCUSSION

https://forms.gle/u4TvMumnH7yBHd3b8

What are some reasons why GraphX or GraphLab or Naiad might be slower than a single thread implementation of PageRank?

- Communication overhead between nodes
  - ↳ single thread = no communication
- Load balancing → one core / machine be slow and lead to overall slow down
- "Memory locality" → all of this data fits in memory of one machine (~ Billion vertex 4 bytes = 4GB)
  (Hilbert ordering)
- GraphX implemented in Scala / JVM / Python → overheads
  vs. having C++ / C

How would you expect a single-thread QR implementation to perform?



Configuration: 100K rows/core, 24 cores

Matrix: 2.4M x 1024, Dense matrix

	1st Stage	TSQR tree	Total
EC2+OpenBLAS	23.604 (s)	1.080 (s)	24.752 (s)

- What is being computed
  - Scalar addition / multiplication
  - low compute
- First map stage is compute intensive ⇒ distributing makes more sense

SUMMARY

GraphX: Combine graph processing with relational model

COST

- Configuration that outperforms single-thread
- Measure scalability AND absolute performance → utilization
    - Computation model of scalable frameworks might be limited
    - Hardware efficiency matters
    - System/Language overheads

NEXT STEPS

Next class: Marius

Project check-ins by Nov ~~20~~ 30 th

OPTIMIZING MR TRIPLETS

Filtered Index Scanning

- Store edges clustered on source vertex id
- Filter triplets using user-defined predicate

Automatic Join Elimination

- Some UDFs don't access source or dest properties
- Inspect JVM byte code to avoid joins