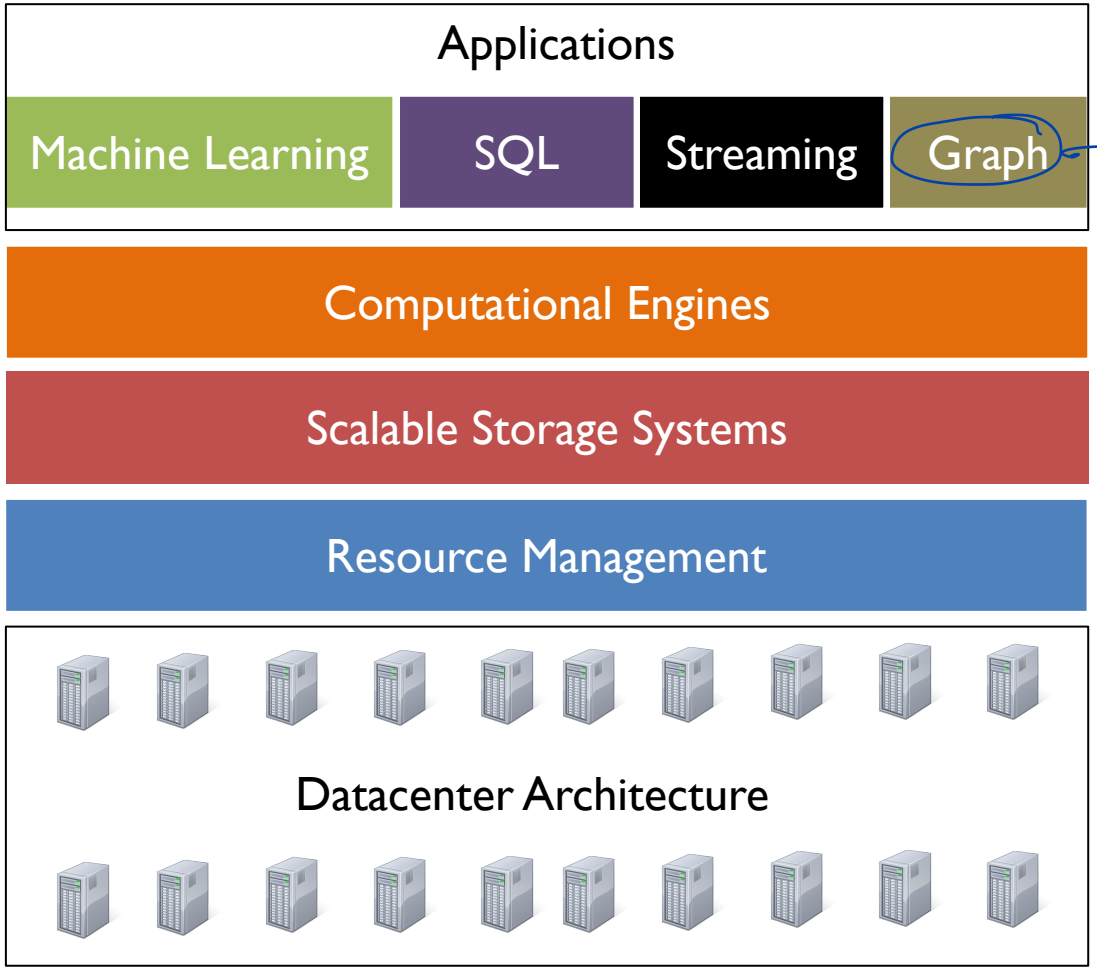
CS 744: GRAPHX

Shivaram Venkataraman

Fall 2019

ADMINISTRIVIA

- Midterm grades are up!
- Course Project: Check in meetings Thu, Mon



Analyze large graph data

Natural graphs

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?

- Fine-grained parallelism
- Less comm using vertex cuts
- Not all are activated

What are some shortcomings?

- Static graphs
- Fault Tolerance
-
- Cross-graph analytics

↑ Summary

THIS CLASS

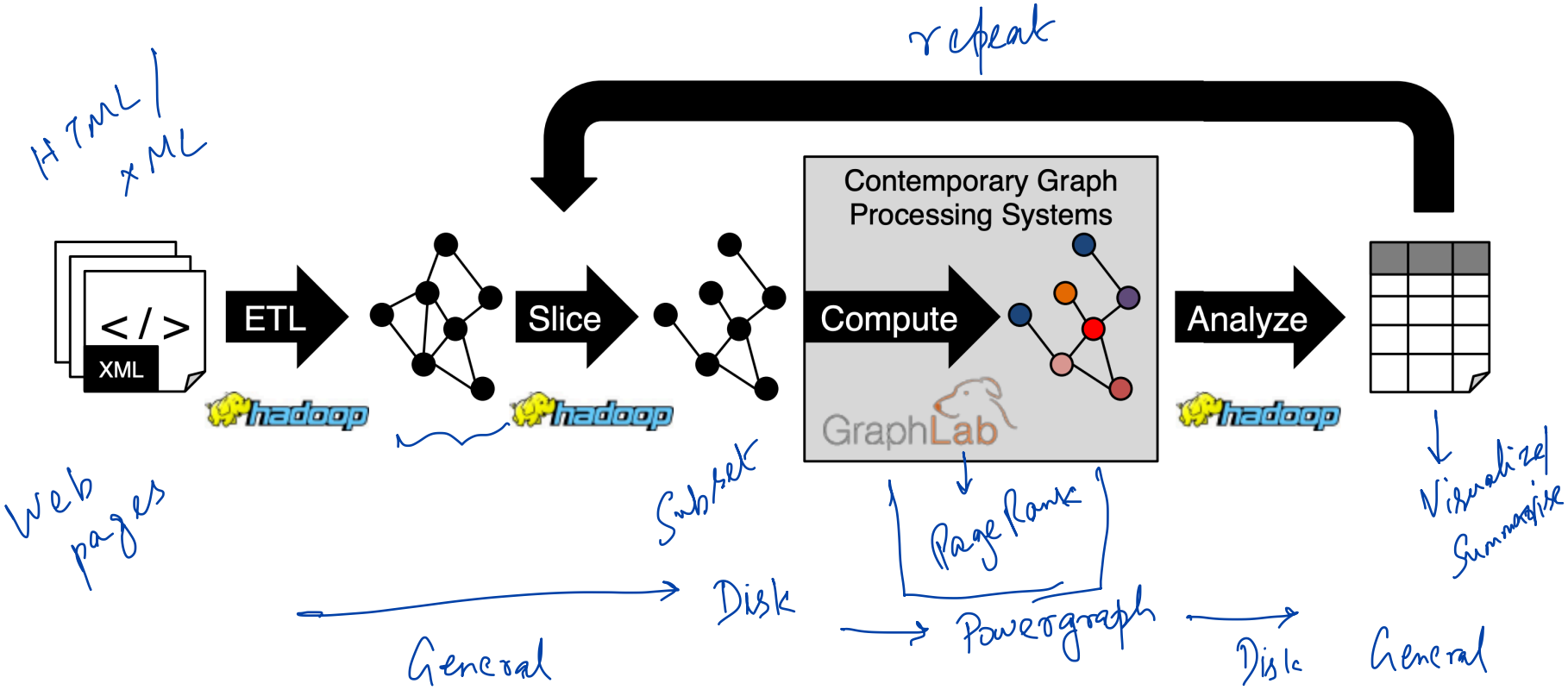
GraphX

Can we efficiently map graph abstractions to dataflow engines?

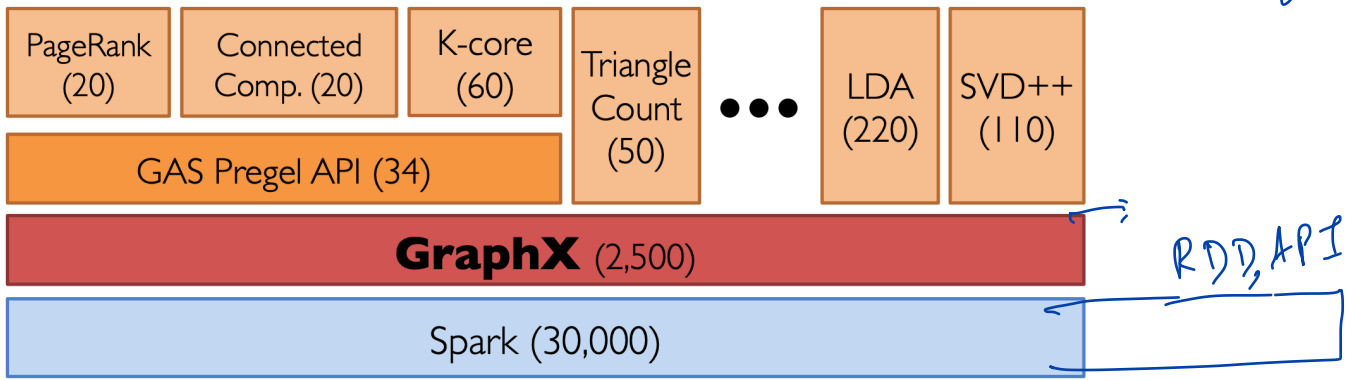
Scalability! But at what COST?

When should we distribute graph processing?

MOTIVATION


repeat
HTML / XML
Contemporary Graph Processing Systems
</>
XML
ETL
hadoop
Slice
hadoop
Compute
GraphLab
Analyze
hadoop
Web pages
Subset
Disk
General
PageRank
Powergraph
Disk
General
Visualize / Summarize

SYSTEM OVERVIEW



Layering

RDD API

Advantages?

↳ leverage lower-level

↳ Cluster reuse

Codebase sharing

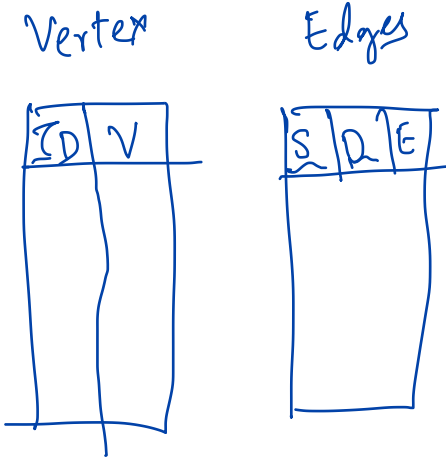
↳ View data graph / table

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

Vertex & Edge Collection



Triplets

Join vertex &

S.ID, D.Id, E, V.S, V.D

select from the edge table

and join vertex E.source = V.ID

and join vertex E.dest. V.ID

MR TRIPLETS

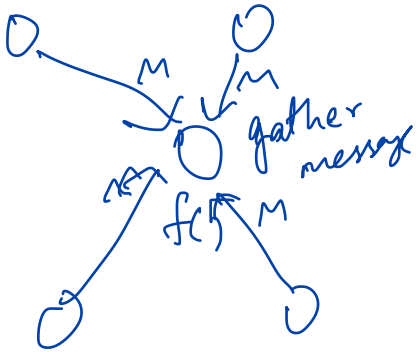
```
mrTriplets(f: (Triplet) => M, sum: (M, M) => M): Collection[(Id, M)]
```

map : Triplet → Message → Apply

sum : Combine → Sum Powergraph

Dest vertices → Final value of message associated with it

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
}
```

Limit Sync execution

Graph

Vertex Program

mrTriplet

All vertices are active

While there are active vertices

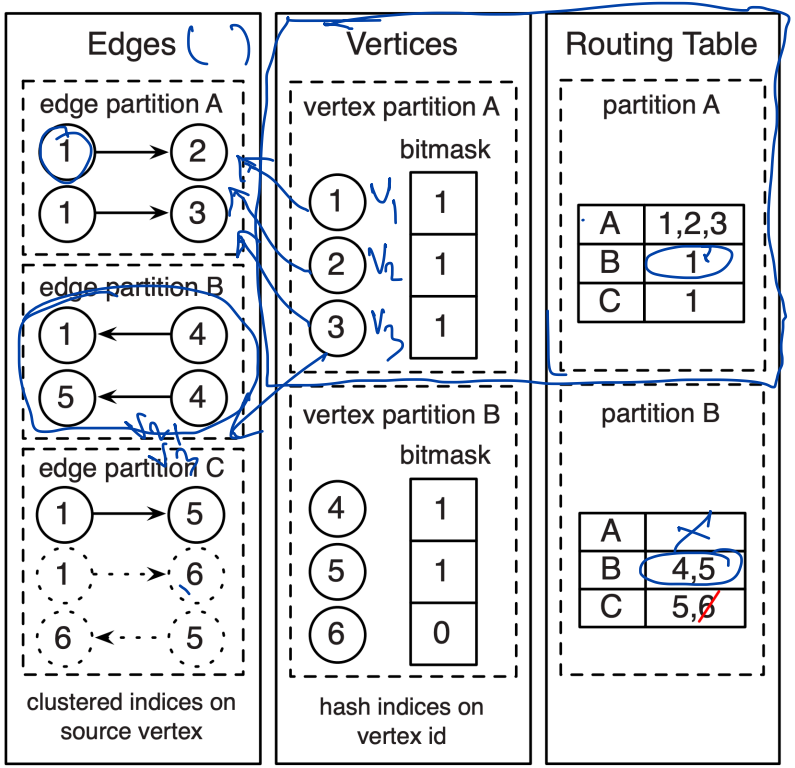
Filter to get active vertices

Derive new vertex property given old vertex and msgs

IMPLEMENTING TRIPLETS VIEW

RDD [Edge]

RDD[Vertex] → Hash partition



Join strategy

- Send vertices to the edge site

Default: Use FS partitions

- Number of vertices << |E|

Multicast join

Using routing table

A: 1, 2, 3 → to machine A

B: 1 → I will send 1, 2, 3

Incremental / Partial materialization

Edge cut → Put all of edges for vertex in same machine

Vertex → Greedy "locality"

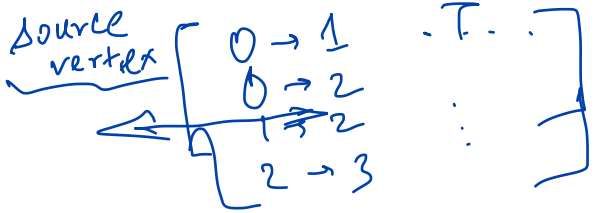
Broadcast → All vertices

OPTIMIZING MR TRIPLETS

Filtered Index Scanning → Only process active vertices

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

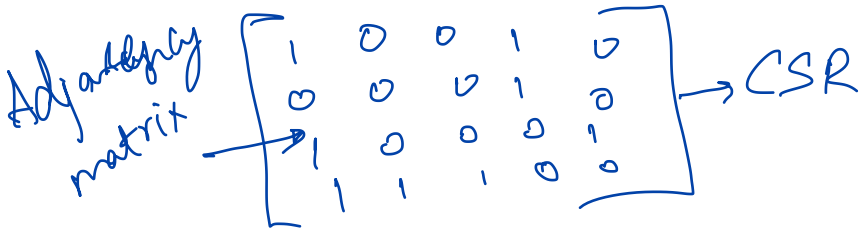
List of triplets/edges map only on active ones



Automatic Join Elimination

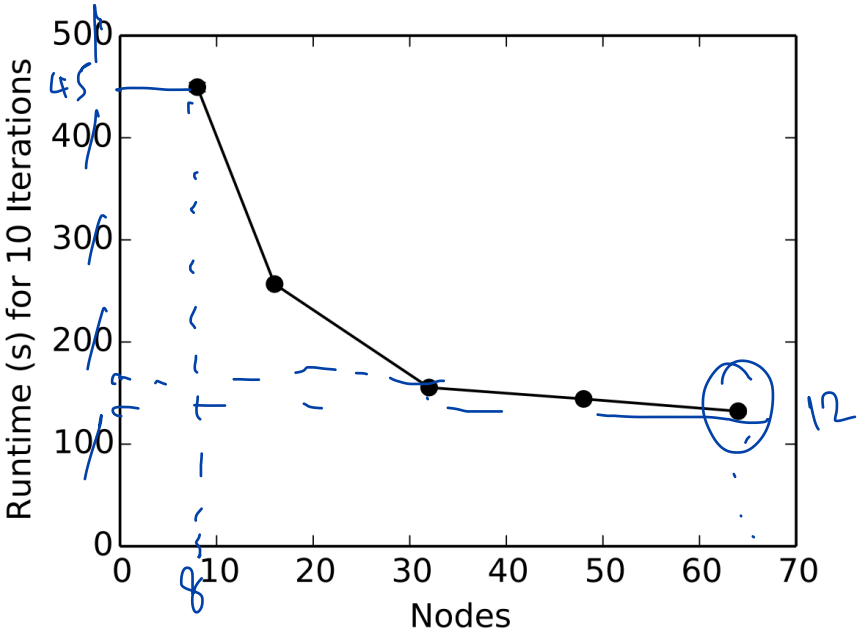
- Some UDFs don't access source or dest properties
- Inspect JVM byte code to avoid joins

```
f (t.Triple → M):
if(t.sP = 10)
    return 1;
return 0;
```

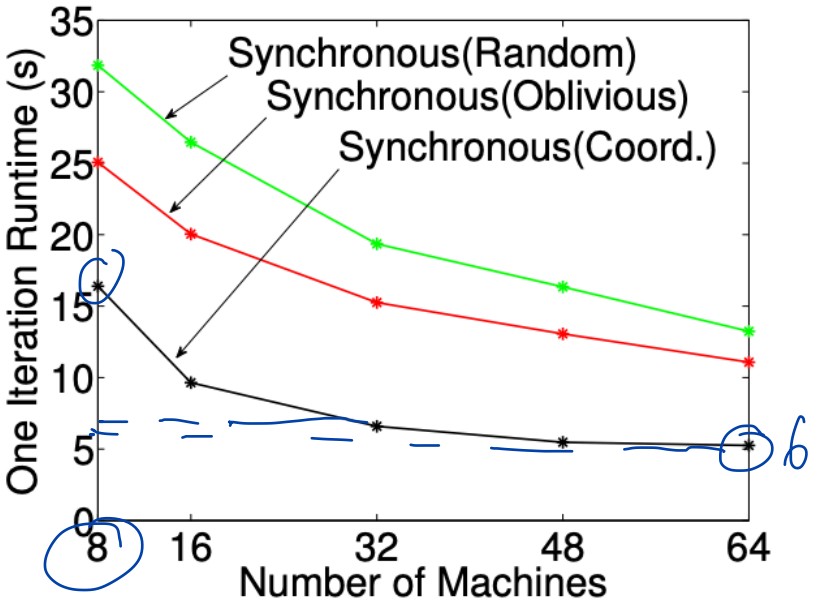


SCALABILITY VS. ABSOLUTE PERFORMANCE

Trend is similar, Twitter, PageRank



GraphX
3x from 8 to 32 machines (4x)



PowerGraph
2.6x from 8 to 32

DISCUSSION

https://forms.gle/ARaU8Ce9XCpkZznn6

Consider a single-threaded PageRank implementation as shown and the performance comparison shown in the corresponding table. What could be some reasons for this performance gap?

- Graph X slowest → immutable data lineage tracking
- Single threaded → Avoid SSD lookup → Graph fits in RAM memory!
- Overhead of "distribution" → High!
  - ↳ Sync in Pregel
    - ↳ Will it get better or worse every iteration?
    - Graphs are hard to split!

Twitter

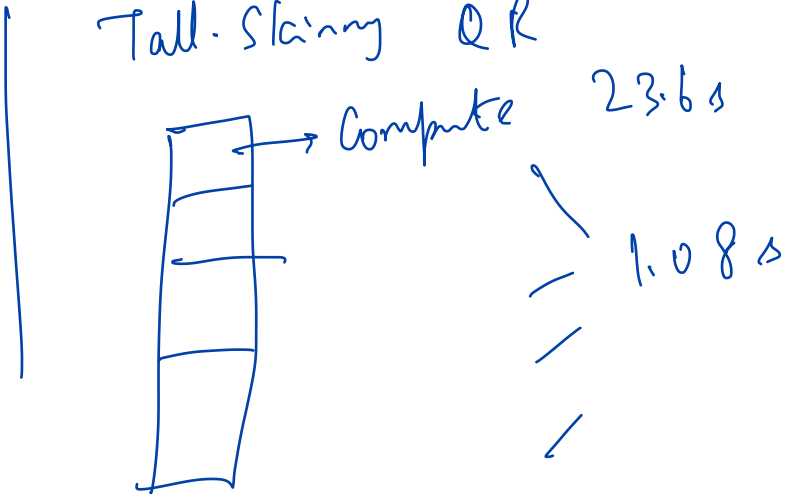
100M vertices	400MB
1B edges	4GB
	≃ 5GB

PAGERANK has not much compute

Now consider a distributed QR decomposition workload shown in Figure below with corresponding performance breakdown. How would you expect a single-thread implementation to perform here?

QR decomposition

A : QR



What are some workload properties that could explain the difference?

SUMMARY

GraphX: Combine graph processing with relational model

COST

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

NEXT STEPS

Next class: Weld

Project check-in meetings