PowerGraph and GraphX

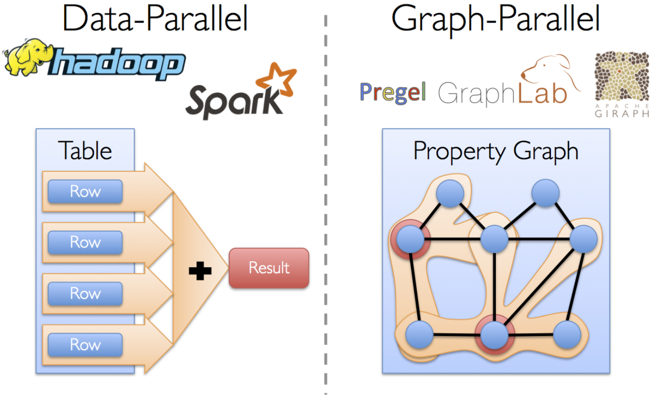
Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)

AMIRKABIR UNIVERSITY
OF TECHNOLOGY

Reminder

Data-Parallel vs. Graph-Parallel Computation



Pregel

- Vertex-centric

Pregel

- Vertex-centric
- Bulk Synchronous Parallel (BSP)

Pregel

- Vertex-centric
- Bulk Synchronous Parallel (BSP)
- Runs in sequence of iterations (supersteps)

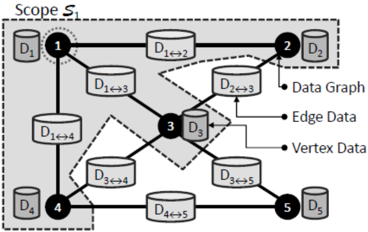
Pregel

- Vertex-centric
- Bulk Synchronous Parallel (BSP)
- Runs in sequence of iterations (supersteps)
- A vertex in superstep S can:
  - reads messages sent to it in superstep S-1.
  - sends messages to other vertices: receiving at superstep S+1.
  - modifies its state.

Pregel Limitations

- Inefficient if different regions of the graph converge at different speed.
- Can suffer if one task is more expensive than the others.
- Runtime of each phase is determined by the slowest machine.

GraphLab



Scope S_1
D_1
1
$D_{1\leftrightarrow 2}$
2
D_2
$D_{1\leftrightarrow 3}$
$D_{2\leftrightarrow 3}$
Data Graph
Edge Data
$D_{1\leftrightarrow 4}$
3
D_3
Vertex Data
$D_{3\leftrightarrow 4}$
$D_{3\leftrightarrow 5}$
D_4
4
$D_{4\leftrightarrow 5}$
5
D_5

GraphLab

Input: Data Graph $G = (V, E, D)$
Input: Initial task set $\mathcal{T} = \{(f, v_1), (g, v_2), ...\}$
while $\mathcal{T}$ *is not Empty* **do**

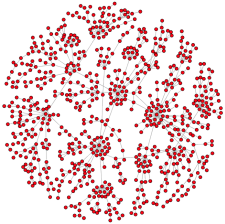
$\quad (f, v) \leftarrow \texttt{RemoveNext}(\mathcal{T})$
$\quad (\mathcal{T}', \mathcal{S}_v) \leftarrow f(v, \mathcal{S}_v)$
$\quad \mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$

Output: Modified Data Graph $G = (V, E, D')$

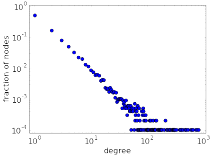
GraphLab Limitations

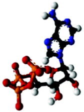
- Poor performance on Natural graphs.

Natural Graphs



- Graphs derived from natural phenomena.
- Skewed Power-Law degree distribution.

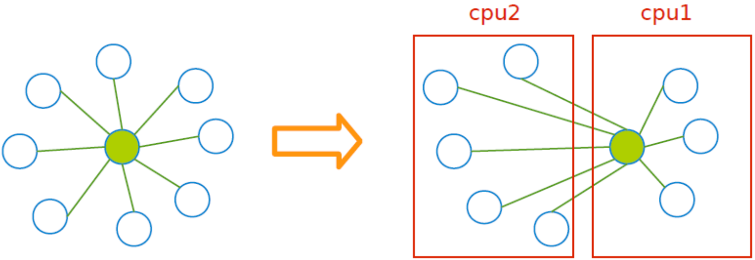




Natural Graphs Challenges

- Traditional graph-partitioning algorithms (edge-cut algorithms) perform poorly on Power-Law Graphs.

- Challenges of high-degree vertices.

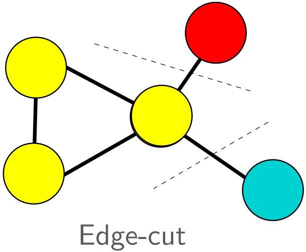


Proposed Solution

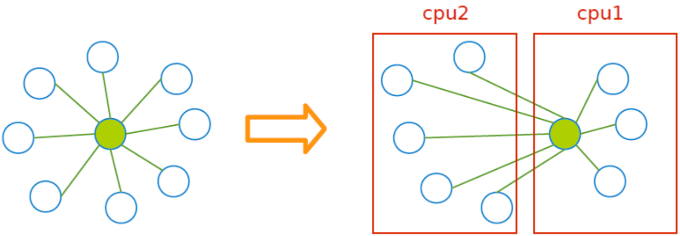
Vertex-Cut Partitioning

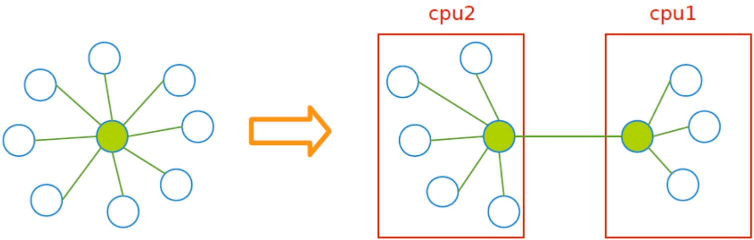
Proposed Solution

Vertex-Cut Partitioning

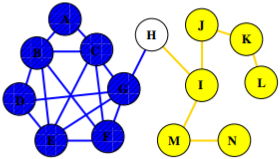


Edge-cut vs. Vertex-cut Partitioning

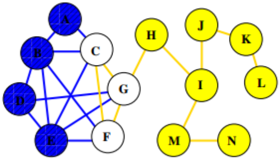




Edge-cut vs. Vertex-cut Partitioning



Edge-cut



Vertex-cut

PowerGraph

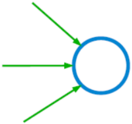
PowerGraph

- Vertex-cut partitioning of graphs.

- Factorizes the GraphLab update function into the Gather, Apply and Scatter phases (GAS).

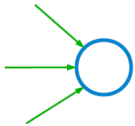
Gather-Apply-Scatter Programming Model

- Gather
  - Accumulate information about neighborhood through a generalized sum.



Gather-Apply-Scatter Programming Model

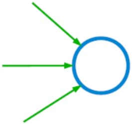
- Gather
  - Accumulate information about neighborhood through a generalized sum.



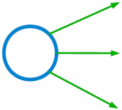
- Apply
  - Apply the accumulated value to center vertex.

Gather-Apply-Scatter Programming Model

- Gather
  - Accumulate information about neighborhood through a generalized sum.



- Apply
  - Apply the accumulated value to center vertex.
- Scatter
  - Update adjacent edges and vertices.



Data Model

- A directed graph that stores the program state, called data graph.

Execution Model (1/2)

- Vertex-centric programming: implementing the `GASVertexProgram` interface (vertex-program for short).

- Similar to `Comput` in Pregel, and update function in GraphLab.

```
interface GASVertexProgram(u) {
  // Run on gather_nbrs(u)
  gather(D_u, D_{u-v}, D_v) → Accum
  sum(Accum left, Accum right) → Accum
  apply(D_u,Accum) → D_u^new
  // Run on scatter_nbrs(u)
  scatter(D_u^new,D_{u-v},D_v) → (D_{u-v}^new, Accum)
}
```

Execution Model (2/2)

Input: Center vertex u

if *Cache Disabled* **then**

```
// Basic Gather-Apply-Scatter Model
```

  foreach *neighbor v in gather_nbrs(u)* **do**

    $a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{u-v}, D_v))$

  $D_u \leftarrow \text{apply}(D_u, a_u)$

  foreach *neighbor v scatter_nbrs(u)* **do**

    $(D_{u-v}) \leftarrow \text{scatter}(D_u, D_{u-v}, D_v)$

else if *Cache Enabled* **then**

```
// Faster GAS Model with Delta Caching
```

  if *cached accumulator a_u is empty* **then**

    foreach *neighbor v in gather_nbrs(u)* **do**

      $a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{u-v}, D_v))$

  $D_u \leftarrow \text{apply}(D_u, a_u)$

  foreach *neighbor v scatter_nbrs(u)* **do**

    $(D_{u-v}, \Delta a) \leftarrow \text{scatter}(D_u, D_{u-v}, D_v)$

    if *a_v and Δa are not Empty* **then** $a_v \leftarrow \text{sum}(a_v, \Delta a)$

    else $a_v \leftarrow$ Empty

Execution Model (2/2)

Input: Center vertex u

if *Cache Disabled* **then**

    // Basic Gather-Apply-Scatter Model

    foreach *neighbor v in gather_nbrs(u)* **do**

        $a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{u-v}, D_v))$

    $D_u \leftarrow \text{apply}(D_u, a_u)$

    foreach *neighbor v scatter_nbrs(u)* **do**

        $(D_{u-v}) \leftarrow \text{scatter}(D_u, D_{u-v}, D_v)$

else if *Cache Enabled* **then**

    // Faster GAS Model with Delta Caching

    if *cached accumulator a_u is empty* **then**

        foreach *neighbor v in gather_nbrs(u)* **do**

            $a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{u-v}, D_v))$

    $D_u \leftarrow \text{apply}(D_u, a_u)$

    foreach *neighbor v scatter_nbrs(u)* **do**

        $(D_{u-v}, \Delta a) \leftarrow \text{scatter}(D_u, D_{u-v}, D_v)$

        if *a_v and Δa are not Empty* **then** $a_v \leftarrow \text{sum}(a_v, \Delta a)$

        else $a_v \leftarrow \text{Empty}$

PageRank - Pregel

```
Pregel_PageRank(i, messages):
  // receive all the messages
  total = 0
  foreach(msg in messages):
    total = total + msg

  // update the rank of this vertex
  R[i] = 0.15 + total

  // send new messages to neighbors
  foreach(j in out_neighbors[i]):
    sendmsg(R[i] * wij) to vertex j
```

$$R[i] = 0.15 + \sum_{j \in Nbrs(i)} w_{ji} R[j]$$

PageRank - GraphLab

```
GraphLab_PageRank(i)
  // compute sum over neighbors
  total = 0
  foreach(j in in_neighbors(i)):
    total = total + R[j] * wji

  // update the PageRank
  R[i] = 0.15 + total

  // trigger neighbors to run again
  foreach(j in out_neighbors(i)):
    signal vertex-program on j
```

$$R[i] = 0.15 + \sum_{j \in Nbrs(i)} w_{ji} R[j]$$

PageRank - PowerGraph

```
PowerGraph_PageRank(i):
  Gather(j -> i):
    return wji * R[j]

  sum(a, b):
    return a + b

  // total: Gather and sum
  Apply(i, total):
    R[i] = 0.15 + total

  Scatter(i -> j):
    if R[i] changed then activate(j)
```

$$R[i] = 0.15 + \sum_{j \in Nbrs(i)} w_{ji} R[j]$$

Scheduling (1/5)

Input: Data Graph $G = (V, E, D)$
Input: Initial task set $\mathcal{T} = \{(f, v_1), (g, v_2), ...\}$
while $\mathcal{T}$ *is not Empty* **do**

```
    (f, v) ← RemoveNext (T)
    (T′, S_v) ← f(v, S_v)
    T ← T ∪ T′
```

Output: Modified Data Graph $G = (V, E, D')$

- PowerGraph inherits the dynamic scheduling of GraphLab.

Scheduling (2/5)

- Initially all vertices are active.

Input: Data Graph $G = (V, E, D)$
Input: Initial task set $\mathcal{T} = \{(f, v_1), (g, v_2), ...\}$
while $\mathcal{T}$ *is not Empty* **do**
    $(f, v) \leftarrow$ `RemoveNext` $(\mathcal{T})$
    $(\mathcal{T}', \mathcal{S}_v) \leftarrow f(v, \mathcal{S}_v)$
    $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$
Output: Modified Data Graph $G = (V, E, D')$

Scheduling (2/5)

- Initially all vertices are active.

Input: Data Graph $G = (V, E, D)$
Input: Initial task set $\mathcal{T} = \{(f, v_1), (g, v_2), ...\}$
while $\mathcal{T}$ *is not Empty* **do**
  $(f, v) \leftarrow$ `RemoveNext` $(\mathcal{T})$
  $(\mathcal{T}', \mathcal{S}_v) \leftarrow f(v, \mathcal{S}_v)$
  $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$
Output: Modified Data Graph $G = (V, E, D')$

- PowerGraph executes the vertex-program on the active vertices until none remain.

Scheduling (2/5)

- Initially all vertices are active.

Input: Data Graph $G = (V, E, D)$
Input: Initial task set $\mathcal{T} = \{(f, v_1), (g, v_2), ...\}$
while $\mathcal{T}$ *is not Empty* **do**
    $(f, v) \leftarrow$ `RemoveNext` $(\mathcal{T})$
    $(\mathcal{T}', \mathcal{S}_v) \leftarrow f(v, \mathcal{S}_v)$
    $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$
Output: Modified Data Graph $G = (V, E, D')$

- PowerGraph executes the vertex-program on the active vertices until none remain.

- The order of executing activated vertices is up to the PowerGraph execution engine.

Scheduling (2/5)

- Initially all vertices are active.

Input: Data Graph $G = (V, E, D)$
Input: Initial task set $\mathcal{T} = \{(f, v_1), (g, v_2), ...\}$
while $\mathcal{T}$ *is not Empty* **do**
    $(f, v) \leftarrow$ `RemoveNext` $(\mathcal{T})$
    $(\mathcal{T}', \mathcal{S}_v) \leftarrow f(v, \mathcal{S}_v)$
    $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$
Output: Modified Data Graph $G = (V, E, D')$

- PowerGraph executes the vertex-program on the active vertices until none remain.

- The order of executing activated vertices is up to the PowerGraph execution engine.

- Once a vertex-program completes the scatter phase it becomes inactive until it is reactivated.

Scheduling (2/5)

Input: Data Graph $G = (V, E, D)$
Input: Initial task set $\mathcal{T} = \{(f, v_1), (g, v_2), ...\}$
while $\mathcal{T}$ *is not Empty* **do**
    $(f, v) \leftarrow$ `RemoveNext` $(\mathcal{T})$
    $(\mathcal{T}', \mathcal{S}_v) \leftarrow f(v, \mathcal{S}_v)$
    $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$
Output: Modified Data Graph $G = (V, E, D')$

- Initially all vertices are active.
- PowerGraph executes the vertex-program on the active vertices until none remain.
- The order of executing activated vertices is up to the PowerGraph execution engine.
- Once a vertex-program completes the scatter phase it becomes inactive until it is reactivated.
- Vertices can activate themselves and neighboring vertices.

Scheduling (3/5)

- PowerGraph can execute both synchronously and asynchronously.
  - Bulk synchronous execution
  - Asynchronous execution

Scheduling - Bulk Synchronous Execution (4/5)

- Similar to Pregel.

Scheduling - Bulk Synchronous Execution (4/5)

- Similar to Pregel.
- Minor-step: executing the gather, apply, and scatter in order.
  - Runs synchronously on all active vertices with a barrier at the end.

Scheduling - Bulk Synchronous Execution (4/5)

- Similar to Pregel.
- Minor-step: executing the gather, apply, and scatter in order.
  - Runs synchronously on all active vertices with a barrier at the end.
- Super-step: a complete series of GAS minor-steps.

Scheduling - Bulk Synchronous Execution (4/5)

- Similar to Pregel.
- Minor-step: executing the gather, apply, and scatter in order.
  - Runs synchronously on all active vertices with a barrier at the end.
- Super-step: a complete series of GAS minor-steps.
- Changes made to the vertex/edge data are committed at the end of each minor-step and are visible in the subsequent minor-steps.

Scheduling - Asynchronous Execution (5/5)

- Changes made to the vertex/edge data during the apply and scatter functions are immediately committed to the graph.
  - Visible to subsequent computation on neighboring vertices.

Scheduling - Asynchronous Execution (5/5)

- Changes made to the vertex/edge data during the apply and scatter functions are immediately committed to the graph.
  - Visible to subsequent computation on neighboring vertices.

- Serializability: prevents adjacent vertex-programs from running concurrently using a fine-grained locking protocol.

Scheduling - Asynchronous Execution (5/5)

- Changes made to the vertex/edge data during the apply and scatter functions are immediately committed to the graph.
    - Visible to subsequent computation on neighboring vertices.

- Serializability: prevents adjacent vertex-programs from running concurrently using a fine-grained locking protocol.
    - Dining philosophers problem, where each vertex is a philosopher, and each edge is a fork.

Scheduling - Asynchronous Execution (5/5)

- Changes made to the vertex/edge data during the apply and scatter functions are immediately committed to the graph.
  - Visible to subsequent computation on neighboring vertices.

- Serializability: prevents adjacent vertex-programs from running concurrently using a fine-grained locking protocol.
  - Dining philosophers problem, where each vertex is a philosopher, and each edge is a fork.
  - GraphLab implements Dijkstras solution, where forks are acquired sequentially according to a total ordering.

Scheduling - Asynchronous Execution (5/5)

- Changes made to the vertex/edge data during the apply and scatter functions are immediately committed to the graph.
  - Visible to subsequent computation on neighboring vertices.

- Serializability: prevents adjacent vertex-programs from running concurrently using a fine-grained locking protocol.
  - Dining philosophers problem, where each vertex is a philosopher, and each edge is a fork.
  - GraphLab implements Dijkstras solution, where forks are acquired sequentially according to a total ordering.
  - PowerGraph implements Chandy-Misra solution, which acquires all forks simultaneously.

Delta Caching (1/2)

- Changes in a few of its neighbors → triggering a vertex-program
- The gather operation is invoked on all neighbors: wasting computation cycles

Delta Caching (1/2)

- Changes in a few of its neighbors $\rightarrow$ triggering a vertex-program
- The gather operation is invoked on all neighbors: wasting computation cycles
- Maintaining a cache of the accumulator a_v from the previous gather phase for each vertex.
- The scatter can return an additional Δa, which is added to the cached accumulator a_v.

Delta Caching (2/2)

Input: Center vertex u

if *Cache Disabled* **then**

    // Basic Gather-Apply-Scatter Model

    foreach *neighbor* v *in gather_nbrs(u)* **do**

        $a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{u-v}, D_v))$

    $D_u \leftarrow \text{apply}(D_u, a_u)$

    foreach *neighbor* v *scatter_nbrs(u)* **do**

        $(D_{u-v}) \leftarrow \text{scatter}(D_u, D_{u-v}, D_v)$

else if *Cache Enabled* **then**

    // Faster GAS Model with Delta Caching

    if *cached accumulator* a_u *is empty* **then**

        foreach *neighbor* v *in gather_nbrs(u)* **do**

            $a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{u-v}, D_v))$

    $D_u \leftarrow \text{apply}(D_u, a_u)$

    foreach *neighbor* v *scatter_nbrs(u)* **do**

        $(D_{u-v}, \Delta a) \leftarrow \text{scatter}(D_u, D_{u-v}, D_v)$

        if a_v *and* Δa *are not Empty* **then** $a_v \leftarrow \text{sum}(a_v, \Delta a)$

        else $a_v \leftarrow \text{Empty}$

Delta Caching (2/2)

```
Input: Center vertex u
if Cache Disabled then
    // Basic Gather-Apply-Scatter Model
    foreach neighbor v in gather_nbrs(u) do
        a_u ← sum(a_u, gather(D_u, D_{u−v}, D_v))
    D_u ← apply(D_u, a_u)
    foreach neighbor v scatter_nbrs(u) do
        (D_{u−v}) ← scatter(D_u, D_{u−v}, D_v)
else if Cache Enabled then
    // Faster GAS Model with Delta Caching
    if cached accumulator a_u is empty then
        foreach neighbor v in gather_nbrs(u) do
            a_u ← sum(a_u, gather(D_u, D_{u−v}, D_v))
    D_u ← apply(D_u, a_u)
    foreach neighbor v scatter_nbrs(u) do
        (D_{u−v}, Δa) ← scatter(D_u, D_{u−v}, D_v)
        if a_v and Δa are not Empty then  a_v ← sum(a_v, Δa)
        else a_v ← Empty
```

Example: PageRank (Delta-Caching)

```
PowerGraph_PageRank(i):
  Gather(j -> i):
    return wji * R[j]

  sum(a, b):
    return a + b

  // total: Gather and sum
  Apply(i, total):
    new = 0.15 + total
    R[i].delta = new - R[i]
    R[i] = new

  Scatter(i -> j):
    if R[i] changed then activate(j)
    return wij * R[i].delta
```

$$R[i] = 0.15 + \sum_{j \in Nbrs(i)} w_{ji} R[j]$$

Graph Partitioning

- Vertex-cut partitioning.

- Evenly assign edges to machines.
  - Minimize machines spanned by each vertex.

- Two proposed solutions:
  - Random edge placement.
  - Greedy edge placement.

Random Vertex-Cuts

- Randomly assign edges to machines.
- Completely parallel and easy to distribute.
- High replication factor.

Greedy Vertex-Cuts (1/2)

- A(v): set of machines that contain adjacent edges of v.

Greedy Vertex-Cuts (1/2)

- A(v): set of machines that contain adjacent edges of v.
- Case 1: If A(u) and A(v) intersect, then the edge should be assigned to a machine in the intersection.

Greedy Vertex-Cuts (1/2)

- A(v): set of machines that contain adjacent edges of v.
- Case 1: If A(u) and A(v) intersect, then the edge should be assigned to a machine in the intersection.
- Case 2: If A(u) and A(v) are not empty and do not intersect, then the edge should be assigned to one of the machines from the vertex with the most unassigned edges.

Greedy Vertex-Cuts (1/2)

- A(v): set of machines that contain adjacent edges of v.
- Case 1: If A(u) and A(v) intersect, then the edge should be assigned to a machine in the intersection.
- Case 2: If A(u) and A(v) are not empty and do not intersect, then the edge should be assigned to one of the machines from the vertex with the most unassigned edges.
- Case 3: If only one of the two vertices has been assigned, then choose a machine from the assigned vertex.

Greedy Vertex-Cuts (1/2)

- A(v): set of machines that contain adjacent edges of v.
- Case 1: If A(u) and A(v) intersect, then the edge should be assigned to a machine in the intersection.
- Case 2: If A(u) and A(v) are not empty and do not intersect, then the edge should be assigned to one of the machines from the vertex with the most unassigned edges.
- Case 3: If only one of the two vertices has been assigned, then choose a machine from the assigned vertex.
- Case 4: If neither vertex has been assigned, then assign the edge to the least loaded machine.

Greedy Vertex-Cuts (2/2)

- Coordinated edge placement:
  - Requires coordination to place each edge
  - Slower, but higher quality cuts

- Oblivious edge placement:
  - Approx. greedy objective without coordination
  - Faster, but lower quality cuts

PowerGraph Summary

- Gather-Apply-Scatter programming model
- Synchronous and Asynchronous models
- Vertex-cut graph partitioning

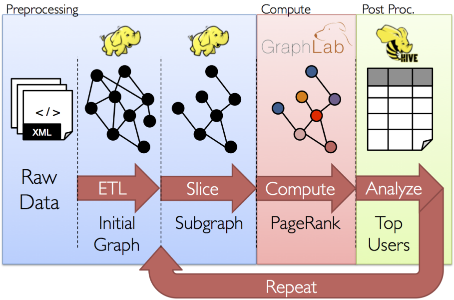
- Any limitations?

Data-Parallel vs. Graph-Parallel Computation

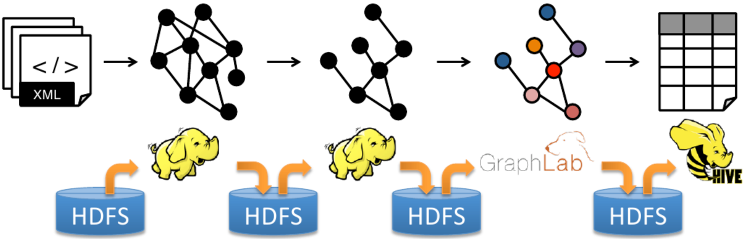
- Graph-parallel computation: restricting the types of computation to achieve performance.

Data-Parallel vs. Graph-Parallel Computation

- Graph-parallel computation: restricting the types of computation to achieve performance.
- But, the same restrictions make it difficult and inefficient to express many stages in a typical graph-analytics pipeline.

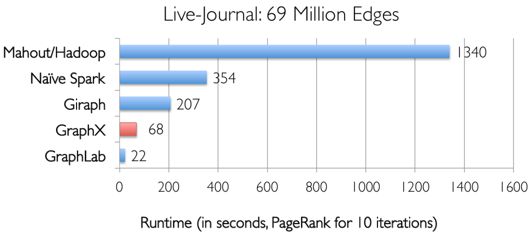


Data-Parallel and Graph-Parallel Pipeline

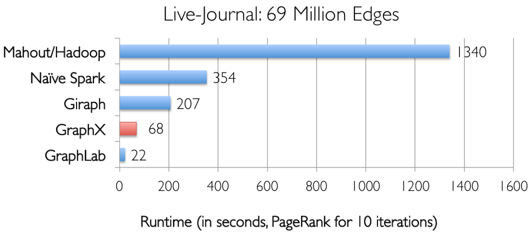


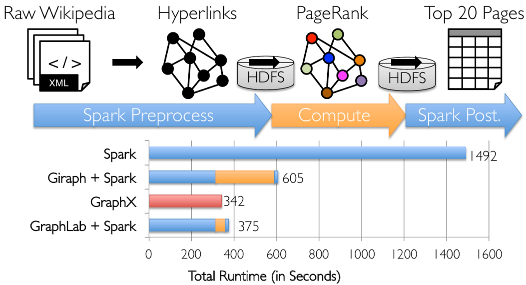
- Moving between table and graph views of the same physical data.
- Inefficient: extensive data movement and duplication across the network and file system.

GraphX vs. Data-Parallel/Graph-Parallel Systems



GraphX vs. Data-Parallel/Graph-Parallel Systems





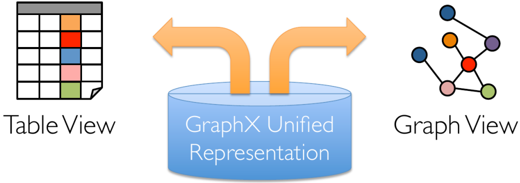
GraphX

GraphX

- New API that blurs the distinction between Tables and Graphs.
- New system that unifies Data-Parallel and Graph-Parallel systems.
- It is implemented on top of Spark.

Unifying Data-Parallel and Graph-Parallel Analytics

- Tables and Graphs are composable views of the same physical data.
- Each view has its own operators that exploit the semantics of the view to achieve efficient execution.

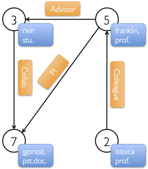


Data Model

- Property Graph: represented using two Spark RDDs:
  - Edge collection: VertexRDD
  - Vertex collection: EdgeRDD

```
// VD: the type of the vertex attribute
// ED: the type of the edge attribute
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```

Property Graph



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

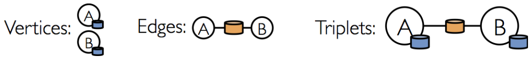
Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

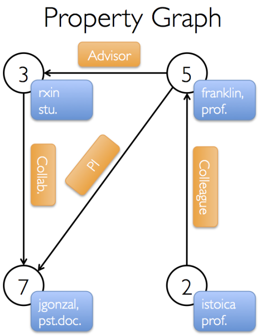
Primitive Data Types

```
// Vertex collection
class VertexRDD[VD] extends RDD[(VertexId, VD)]

// Edge collection
class EdgeRDD[ED] extends RDD[Edge[ED]]
case class Edge[ED](srcId: VertexId = 0, dstId: VertexId = 0,
                    attr: ED = null.asInstanceOf[ED])

// Edge Triple
class EdgeTriplet[VD, ED] extends Edge[ED]
```

- EdgeTriplet represents an edge along with the vertex attributes of its neighboring vertices.



Example (1/3)



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

Example (2/3)

```
val sc: SparkContext

// Create an RDD for the vertices
val users: VertexRDD[(String, String)] = sc.parallelize(
    Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
          (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))

// Create an RDD for edges
val relationships: EdgeRDD[String] = sc.parallelize(
    Array(Edge(3L, 7L, "collab"),    Edge(5L, 3L, "advisor"),
          Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))

// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")

// Build the initial Graph
val userGraph: Graph[(String, String), String] =
    Graph(users, relationships, defaultUser)
```

Example (3/3)

```
// Constructed from above
val userGraph: Graph[(String, String), String]

// Count all users which are postdocs
userGraph.vertices.filter((id, (name, pos)) => pos == "postdoc").count

// Count all the edges where src > dst
userGraph.edges.filter(e => e.srcId > e.dstId).count

// Use the triplets view to create an RDD of facts
val facts: RDD[String] = graph.triplets.map(triplet =>
    triplet.srcAttr._1 + " is the " +
    triplet.attr + " of " + triplet.dstAttr._1)

// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")

facts.collect.foreach(println(_))
```

Property Operators (1/2)

```
class Graph[VD, ED] {
  def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]

  def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]

  def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
}
```

- They yield new graphs with the vertex or edge properties modified by the map function.
- The graph structure is unaffected.

Property Operators (2/2)

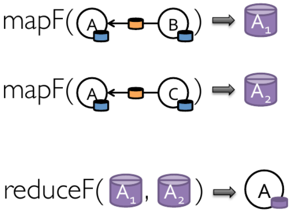
```
val newGraph = graph.mapVertices((id, attr) => mapUdf(id, attr))
```

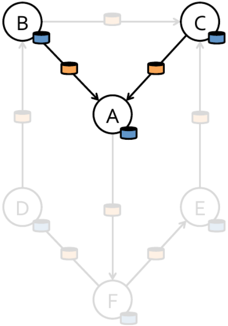
```
val newVertices = graph.vertices.map((id, attr) => (id, mapUdf(id, attr)))
val newGraph = Graph(newVertices, graph.edges)
```

- Both are logically equivalent, but the second one does not preserve the structural indices and would not benefit from the GraphX system optimizations.

Map Reduce Triplets

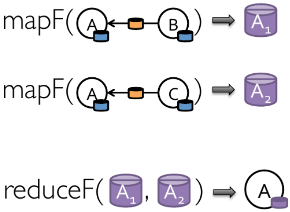
- Map-Reduce for each vertex

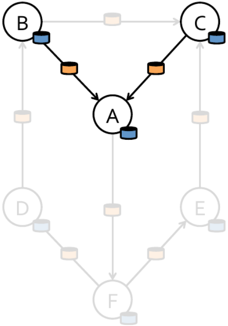




Map Reduce Triplets

- Map-Reduce for each vertex






```
// what is the age of the oldest follower for each user?
val oldestFollowerAge = graph.mapReduceTriplets(
  e => (e.dstAttr, e.srcAttr), // Map
  (a, b) => max(a, b) // Reduce
).vertices
```

Structural Operators

```scala
class Graph[VD, ED] {
  // returns a new graph with all the edge directions reversed
  def reverse: Graph[VD, ED]

  // returns the graph containing only the vertices and edges that satisfy
  // the vertex predicate
  def subgraph(epred: EdgeTriplet[VD,ED] => Boolean,
               vpred: (VertexId, VD) => Boolean): Graph[VD, ED]

  // a subgraph by returning a graph that contains the vertices and edges
  // that are also found in the input graph
  def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
}
```

Structural Operators Example

```scala
// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)

// Run Connected Components
val ccGraph = graph.connectedComponents()

// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")

// Restrict the answer to the valid subgraph
val validCCGraph = ccGraph.mask(validGraph)
```

Join Operators

- To join data from external collections (RDDs) with graphs.

```
class Graph[VD, ED] {
  // joins the vertices with the input RDD and returns a new graph
  // by applying the map function to the result of the joined vertices
  def joinVertices[U](table: RDD[(VertexId, U)])
      (map: (VertexId, VD, U) => VD): Graph[VD, ED]

  // similarly to joinVertices, but the map function is applied to
  // all vertices and can change the vertex property type
  def outerJoinVertices[U, VD2](table: RDD[(VertexId, U)])
      (map: (VertexId, VD, Option[U]) => VD2): Graph[VD2, ED]
}
```

Graph Builders

```
// load a graph from a list of edges on disk
object GraphLoader {
  def edgeListFile(
      sc: SparkContext,
      path: String,
      canonicalOrientation: Boolean = false,
      minEdgePartitions: Int = 1)
    : Graph[Int, Int]
}

// graph file
# This is a comment
2 1
4 1
1 2
```

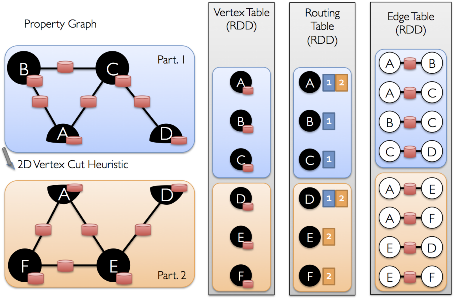
GraphX and Spark

- GraphX is implemented on top of Spark
- In-memory caching
- Lineage-based fault tolerance
- Programmable partitioning

Distributed Graph Representation (1/2)

- Representing graphs using two RDDs: edge-collection and vertex-collection

- Vertex-cut partitioning (like PowerGraph)



Distributed Graph Representation (2/2)

- Each vertex partition contains a bitmask and routing table.

- Routing table: a logical map from a vertex id to the set of edge partitions that contains adjacent edges.

- Bitmask: enables the set intersection and filtering.
  - Vertices bitmasks are updated after each operation (e.g., mapReduceTriplets).
  - Vertices hidden by the bitmask do not participate in the graph operations.

Summary

Summary

- PowerGraph
  - GAS programming model
  - Vertex-cut partitioning
- GraphX
  - Unifying data-parallel and graph-parallel analytics
  - Vertex-cut partitioning

Questions?

Acknowledgements

Some pictures were derived from the Spark web site (`http://spark.apache.org/`).

Questions?