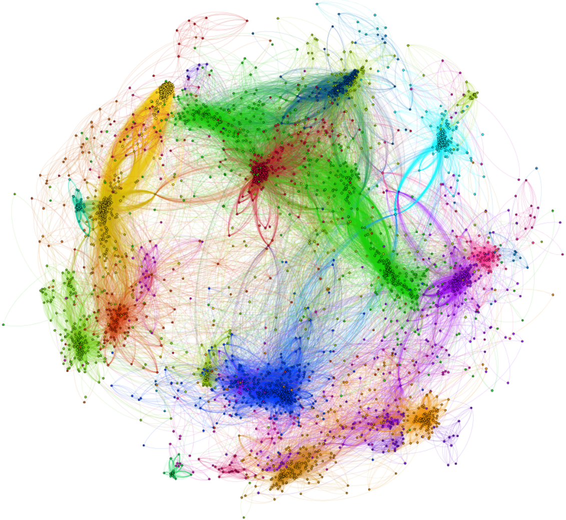
# Large Scale Graph Processing
# Pregel, GraphLab and GraphX

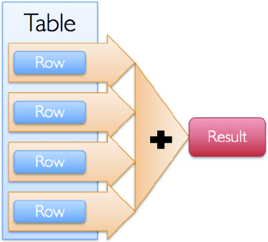
Amir H. Payberah
amir@sics.se

KTH Royal Institute of Technology

# Large Graph



Can we use platforms like MapReduce or Spark, which are based on data-parallel model, for large-scale graph proceeding?

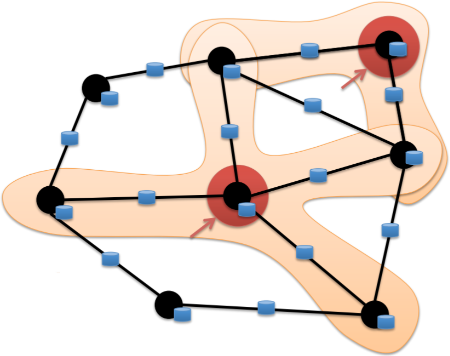


# Graph Algorithms Characteristics

- Difficult to extract parallelism based on partitioning of the data.
- Difficult to express parallelism based on partitioning of computation.

# Proposed Solution

Graph-Parallel Processing

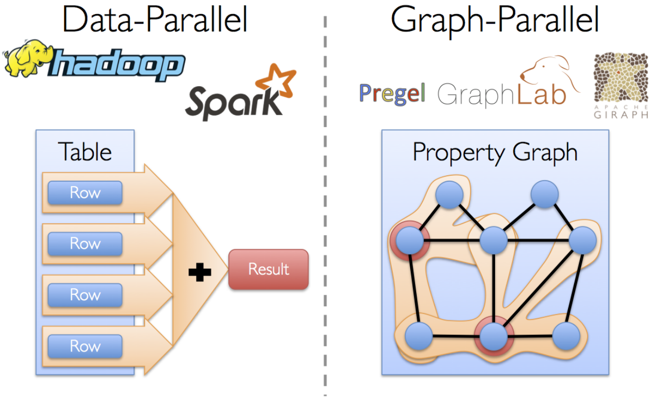


- Computation typically depends on the neighbors.

# Graph-Parallel Processing

- Expose specialized APIs to simplify graph programming.
- Restricts the types of computation.
- New techniques to partition and distribute graphs.
- Exploit graph structure.

# Data-Parallel vs. Graph-Parallel Computation



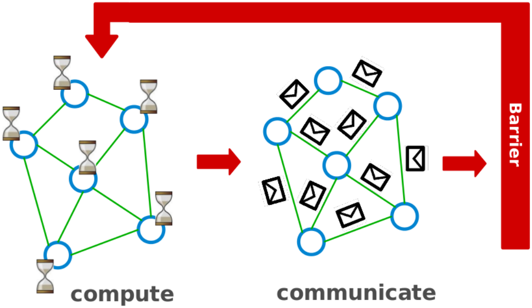
# Pregel

# Pregel

- Large-scale graph-parallel processing platform developed at Google.

- Inspired by bulk synchronous parallel (BSP) model.

# Bulk Synchronous Parallel




All vertices update in parallel (at the same time).

# Programming Model

- Vertex-centric programming: Think as a vertex.
- Each vertex computes individually its value: in parallel
- Each vertex can see its local context and updates its value.
- Input data: a directed graph that stores the program state, e.g., the current value.

# Execution Model (1/2)

- Applications run in sequence of iterations: supersteps
  - Invoking method `Compute()` in parallel in all vertices

# Execution Model (1/2)

- Applications run in sequence of iterations: supersteps
  - Invoking method `Compute()` in parallel in all vertices

- 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.

# Execution Model (1/2)

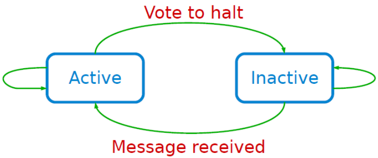
- Applications run in sequence of iterations: supersteps
  - Invoking method `Compute()` in parallel in all vertices

- 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.

- Vertices communicate directly with one another by sending messages.

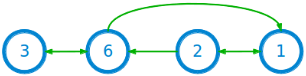
# Execution Model (2/2)

- Superstep 0: all vertices are in the active state.
- A vertex deactivates itself by voting to halt: no further work to do.
- A halted vertex can be active if it receives a message.
- The whole algorithm terminates when:
  - All vertices are simultaneously inactive.
  - There are no messages in transit.



# Example: Max Value (1/4)

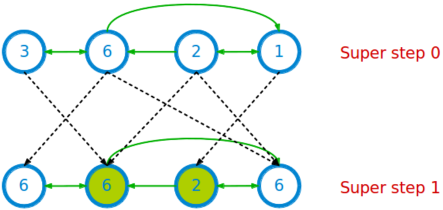
```
i_val := val

for each message m
  if m > val then val := m

if i_val == val then
  vote_to_halt
else
  for each neighbor v
    send_message(v, val)
```




Super step 0

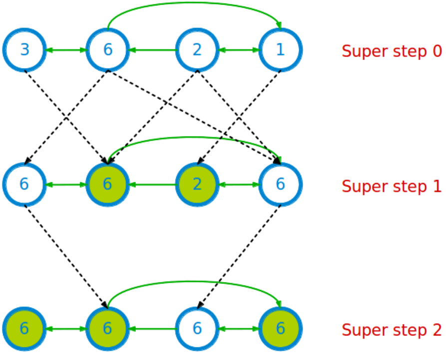
# Example: Max Value (2/4)

```
i_val := val

for each message m
  if m > val then val := m

if i_val == val then
  vote_to_halt
else
  for each neighbor v
    send_message(v, val)
```



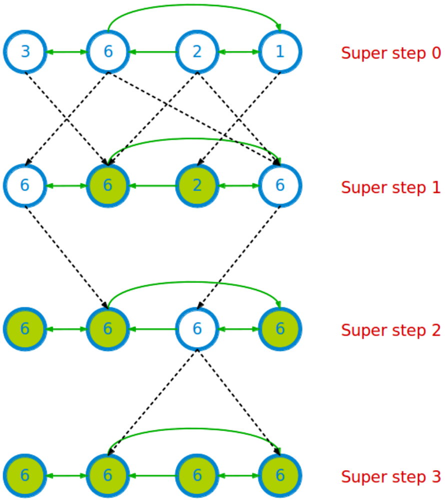
# Example: Max Value (3/4)

```
i_val := val

for each message m
  if m > val then val := m

if i_val == val then
  vote_to_halt
else
  for each neighbor v
    send_message(v, val)
```



# Example: Max Value (4/4)

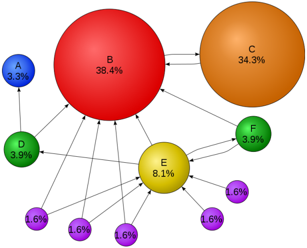
```
i_val := val

for each message m
  if m > val then val := m

if i_val == val then
  vote_to_halt
else
  for each neighbor v
    send_message(v, val)
```



# Example: PageRank

- Update ranks in parallel.
- Iterate until convergence.

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



# Example: PageRank

```
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
```

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

# Implementation

# System Model (1/2)

- Master-worker model.
- The master is responsible for
  - Coordinating workers.
  - Determining the number of partitions.
- Each worker is responsible for
  - Executing the user's `Compute()` method on its vertices.
  - Maintaining the state of its partitions.
  - Managing messages to and from other workers.

# System Model (2/2)

- The master assigns one or more partitions to each worker.
- The master assigns a portion of user input to each worker.
  - Set of records containing a number of vertices and edges.
  - If a worker loads a vertex that belongs to that worker's partitions, the appropriate data structures are immediately updated.
  - Otherwise, the worker enqueues a message to the remote peer that owns the vertex.
- After loading the graph, the master instructs each worker to perform a superstep.

# Graph Partitioning

- The pregel library divides a graph into a number of partitions.
- Each consisting of a set of vertices and all of those vertices' outgoing edges.
- Vertices are assigned to partitions based on their vertex-ID (e.g., hash(ID)).

# Fault Tolerance (1/2)

- Fault tolerance is achieved through checkpointing.
  - Saved to persistent storage
- At start of each superstep, master tells workers to save their state:
  - Vertex values, edge values, incoming messages
- Master saves aggregator values (if any).
- This is not necessarily done at every superstep: costly

# Fault Tolerance (2/2)

- When master detects one or more worker failures:
  - All workers revert to last checkpoint.
  - Continue from there.
  - That is a lot of repeated work.
  - At least it is better than redoing the whole job.

# 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.

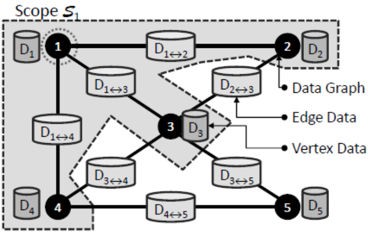
# GraphLab

# GraphLab

- GraphLab allows asynchronous iterative computation.

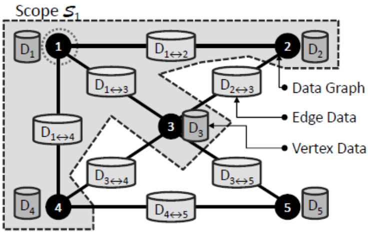
# GraphLab

- GraphLab allows asynchronous iterative computation.
- Vertex scope of vertex $v$: the data stored in $v$, in all adjacent vertices and edges.



# Programming Model

- Vertex-centric programming
- A vertex can read and modify any of the data in its scope.
  - Calling the `Update` function, similar to `Compute` in Pregel.
- Input data: a directed graph that stores the program state.



# Execution Model

**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 $G = (V, E, D')$

- Each task in the set of tasks $\mathcal{T}$, is a tuple $(f, v)$ consisting of an update function $f$ and a vertex $v$.

# Execution Model

**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 $G = (V, E, D')$

- Each task in the set of tasks $\mathcal{T}$, is a tuple $(f, v)$ consisting of an update function $f$ and a vertex $v$.
- After executing an update function $(f, g, \cdots)$ the modified scope data in $S_v$ is written back to the data graph.

# Example: 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
```

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

# Example: 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
```

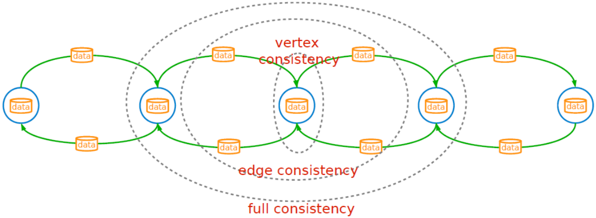
$$R[i] = 0.15 + \sum_{j \in Nbrs(i)} w_{ji} R[j]$$

# Consistency (1/4)

- Overlapped scopes: race-condition in simultaneous execution of two update functions.

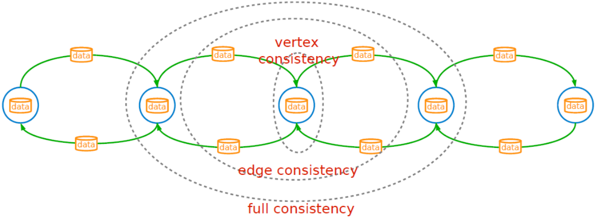
# Consistency (1/4)

- Overlapped scopes: race-condition in simultaneous execution of two update functions.



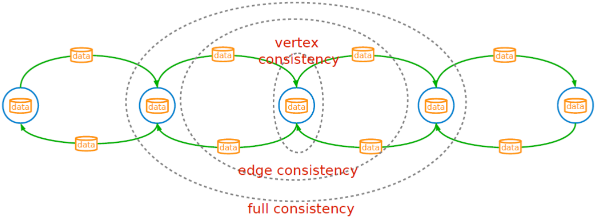

- Full consistency: during the execution $f(v)$, no other function reads or modifies data within the $v$ scope.

# Consistency (2/4)



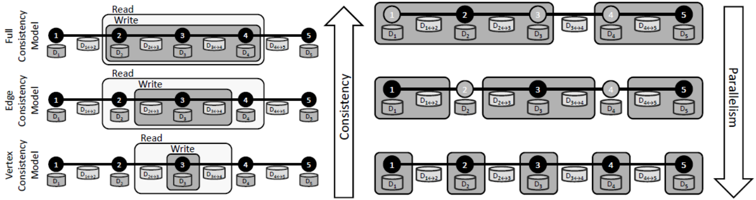

- Edge consistency: during the execution $f(v)$, no other function reads or modifies any of the data on $v$ or any of the edges adjacent to $v$.

# Consistency (3/4)




- Vertex consistency: during the execution $f(v)$, no other function will be applied to $v$.

# Consistency (4/4)




Consistency vs. Parallelism

[Low, Y., GraphLab: A Distributed Abstraction for Large Scale Machine Learning (Doctoral dissertation, University of California), 2013.]

# Implementation

# Consistency Implementation

- Distributed locking: associating a readers-writer lock with each vertex.

# Consistency Implementation

- Distributed locking: associating a readers-writer lock with each vertex.
- Vertex consistency
  - Central vertex (write-lock)

# Consistency Implementation

- Distributed locking: associating a readers-writer lock with each vertex.
- Vertex consistency
  - Central vertex (write-lock)
- Edge consistency
  - Central vertex (write-lock), Adjacent vertices (read-locks)

# Consistency Implementation

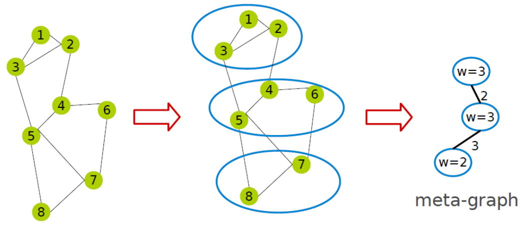
- Distributed locking: associating a readers-writer lock with each vertex.
- Vertex consistency
  - Central vertex (write-lock)
- Edge consistency
  - Central vertex (write-lock), Adjacent vertices (read-locks)
- Full consistency
  - Central vertex (write-locks), Adjacent vertices (write-locks)

# Consistency Implementation

- **Distributed locking**: associating a readers-writer lock with each vertex.
- Vertex consistency
  - Central vertex (write-lock)
- Edge consistency
  - Central vertex (write-lock), Adjacent vertices (read-locks)
- Full consistency
  - Central vertex (write-locks), Adjacent vertices (write-locks)
- **Deadlocks** are avoided by acquiring locks sequentially following a canonical order.

# Graph Partitioning (1/3)

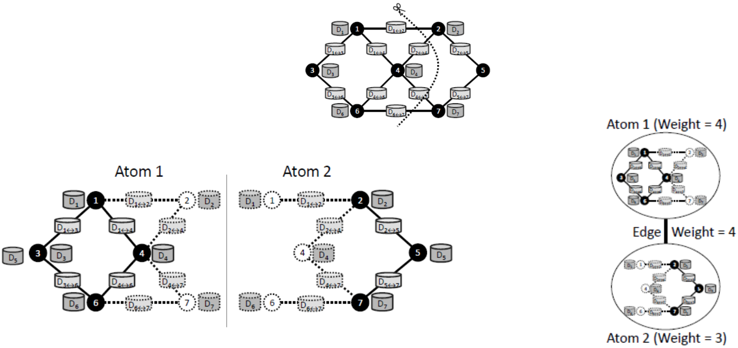
- Two-phase partitioning.
- Partitioning the data graph into k parts, called atom.
- Meta-graph: the graph of atoms (one vertex for each atom).
- Atom weight: the amount of data it stores.
- Edge weight: the number of edges crossing the atoms.



# Graph Partitioning (2/3)

- Meta-graph is very small.
- A fast balanced partition of the meta-graph over the physical machines.
- Assigning graph atoms to machines.

## Graph Partitioning (3/3)




- Each atom file stores interior and the ghosts of the partition.
  - Ghost is set of vertices and edges adjacent to the partition boundary.
- Each atom is stored as a separate file on HDFS.

# Fault Tolerance (1/2)

- Synchronous fault tolerance.
- The systems periodically signals all computation activity to halt.
- Then synchronizes all caches (ghosts) and saves to disk all data which has been modified since the last snapshot.

# Fault Tolerance (1/2)

- Synchronous fault tolerance.
- The systems periodically signals all computation activity to halt.
- Then synchronizes all caches (ghosts) and saves to disk all data which has been modified since the last snapshot.
- Simple, but eliminates the systems advantage of asynchronous computation.

# Fault Tolerance (2/2)

- Asynchronous fault tolerance: based on the Chandy-Lamport algorithm.
- The snapshot function is implemented as an `update` function in vertices.
  - It takes priority over all other update functions.

```
if v was already snapshotted then
    Quit
Save D_v // Save current vertex
// Save all edges connected to un-snapshotted vertices
foreach u ∈ N[v] do                                   // Loop over neighbors
    if u was not snapshotted then
        Save D_{u→v} if edge u → v exists
        Save D_{v→u} if edge v → u exists
        Reschedule u for a Snapshot Update
Mark v as snapshotted
```

# PowerGraph (GraphLab2)

# PowerGraph

- Factorizes the `update` function into the Gather, Apply and Scatter phases.
- Vertx-cut partitioning.

# Programming Model

- Gather-Apply-Scatter (GAS)
- Gather: accumulate information about neighborhood through a generalized sum.
- Apply: apply the accumulated value to center vertex.
- Scatter: update adjacent edges and vertices.

```
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 (1/2)

- Initially all vertices are active.
- It executes the vertex-program on the active vertices until none remain.
  - Once a vertex-program completes the scatter phase it becomes inactive until it is reactivated.
  - Vertices can activate themselves and neighboring vertices.

# Execution Model (1/2)

- Initially all vertices are active.
- It executes the vertex-program on the active vertices until none remain.
  - Once a vertex-program completes the scatter phase it becomes inactive until it is reactivated.
  - Vertices can activate themselves and neighboring vertices.
- PowerGraph can execute both synchronously and asynchronously.

# Scheduling (2/2)

- Synchronous scheduling like Pregel.
  - Executing the gather, apply, and scatter in order.
  - Changes made to the vertex/edge data are committed at the end of each step.

# Scheduling (2/2)

- Synchronous scheduling like Pregel.
  - Executing the gather, apply, and scatter in order.
  - Changes made to the vertex/edge data are committed at the end of each step.

- Asynchronous scheduling like GraphLab.
  - 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.

# Example: 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
```

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

# Example: 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
```

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

# Example: PageRank (PowerGraph)

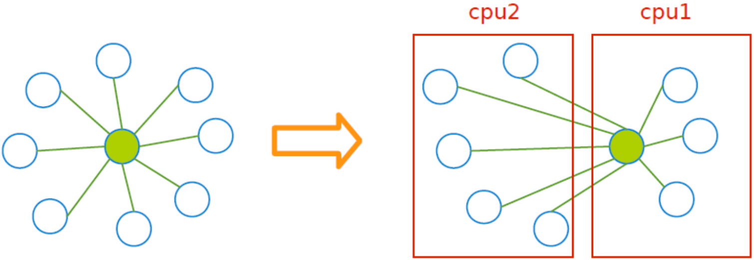
```
PowerGraph_PageRank(i):
  Gather(j -> i):
    return wji * R[j]

  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)
```

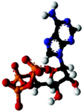
$$R[i] = 0.15 + \sum_{j \in Nbrs(i)} w_{ji} R[j]$$

# Implementation

# Graph Partitioning (1/4)

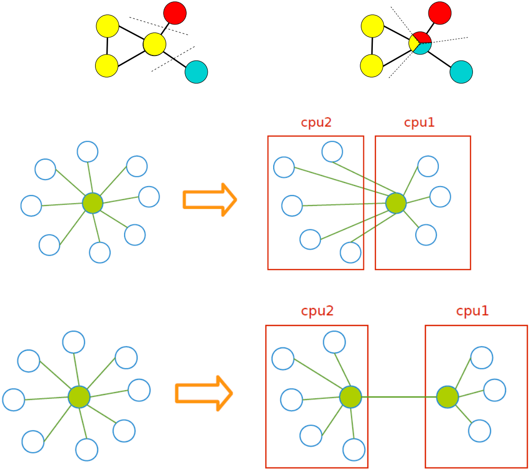
- Natural graphs: skewed Power-Law degree distribution.
- Edge-cut algorithms perform poorly on Power-Law Graphs.



# Graph Partitioning (2/4)

Vertex-Cut partitioning



# Graph Partitioning (3/4)

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

# Graph Partitioning (4/4)

- Greedy vertex-cuts
- $A(v)$: set of machines that vertex v spans.
- Case 1: If $A(u) \cap A(v) \neq \emptyset$, then the edge should be assigned to a machine in the intersection.
- Case 2: If $A(u) \cap A(v) = \emptyset$, 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 $A(u) = A(v) = \emptyset$, then assign the edge to the least loaded machine.

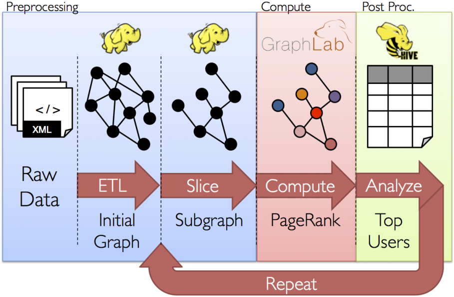
# GraphX

# Data-Parallel vs. Graph-Parallel Computation

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

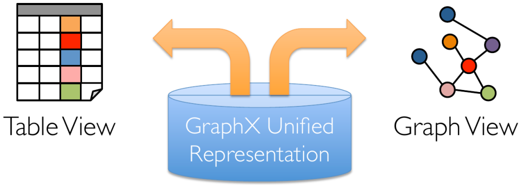
# Data-Parallel vs. Graph-Parallel Computation

- 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.

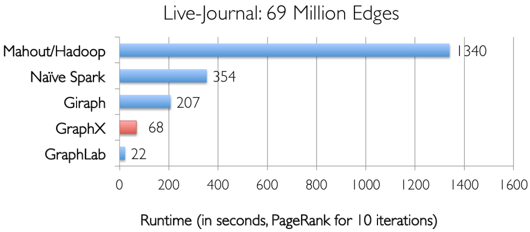


# GraphX

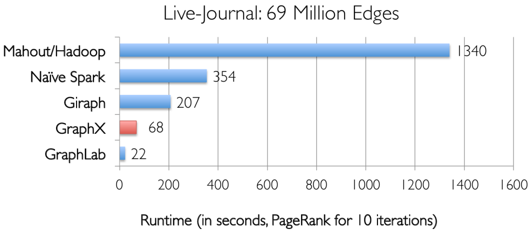
- Unifies data-parallel and graph-parallel systems.
- Tables and Graphs are composable views of the same physical data.
- Implemented on top of Spark.

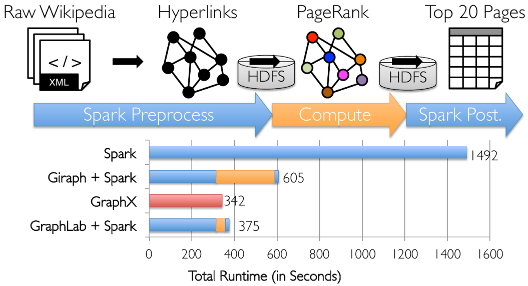


# GraphX vs. Data-Parallel/Graph-Parallel Systems



# GraphX vs. Data-Parallel/Graph-Parallel Systems



# Programming Model

- Gather-Apply-Scatter (GAS)

- Input data (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]
}
```

# Execution Model

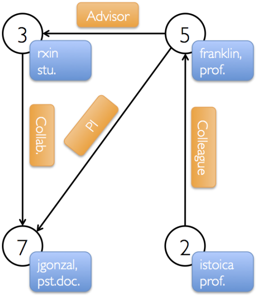
- GAS decomposition
- Gather: the `groupBy` stage gathers messages destined to the same vertex.
- Apply: an intervening `map` operation applies the message sum to update the vertex property.
- Scatter: the `join` stage scatters the new vertex property to all adjacent vertices.

# GraphX Operators

```
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]
}
```

# Example (1/3)

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 |

# 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(_))
```

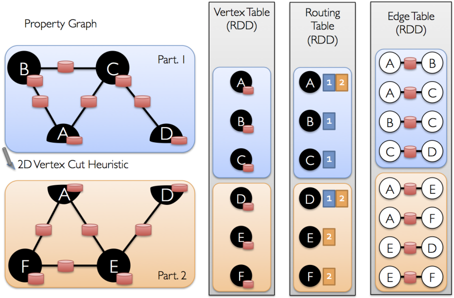
# Implementation

# Implementation

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

# Graph Representation

- Vertex-cut partitioning
- Representing graphs using two RDDs: edge-collection and vertex-collection
- Routing table: a logical map from a vertex id to the set of edge partitions that contains adjacent edges.



# Summary

# Pregel Summary

- Bulk synchronous parallel model
- Vertex-centric
- Superstep: sequence of iterations

# GraphLab Summary

- Asynchronous model
- Vertex-centric
- Three consistency levels: full, edge-level, and vertex-level
- Partitioning: two-phase partitioning
- Consistency: chromatic engine (graph coloring), distributed lock engine (reader-writer lock)
- Fault tolerance: synchronous, asynchronous (chandy-lamport)

# PowerGraph Summary

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

# GraphX Summary

- Unifies graph-parallel and data-prallel models
- Gather-Apply-Scatter programming model
- Vertex-cut graph partitioning
- On top of Spark

Questions?