Graphs are Everywhere

- Transportation networks
- Bibliographic networks
- Computer networks
- Social networks
- Topic maps
- Knowledge bases
- Protein interactions
- Biological food chains
- etc.
Categories of Graph Data Systems

- **Triple stores**
  - Typically, pattern matching queries
  - Data model: RDF

- **Graph databases**
  - Typically, navigational queries
  - Prevalent data model: property graphs

- **Graph processing systems**
  - Typically, complex graph analysis tasks
  - Prevalent data model: generic graphs

Remember my earlier lecture on RDF and SPARQL
Graph Data Models
RDF Data Model

- Data comes as a set of triples \((s, p, o)\)
  - subject: URI
  - predicate: URI
  - object: URI or literal

- Such a set may be understood as a graph
  - Triples as directed edges
  - Subjects and objects as vertexes
  - Edges labeled by predicate

Remember my earlier lecture on RDF and SPARQL
Property Graph

1. person
   - name: marko
   - age: 29
   created
   - weight: 0.4

2. person
   - name: vadas
   - age: 27
   knows
   - weight: 0.5

3. software
   - name: lop
   - lang: java
   created
   - weight: 0.4

4. person
   - name: josh
   - age: 32
   created
   - weight: 0.4

5. software
   - name: ripple
   - lang: java
   created
   - weight: 1.0

6. person
   - name: peter
   - age: 35

10. created
   - weight: 1.0
11. created
   - weight: 0.4
12. created
   - weight: 0.2
Property Graph (cont'd)

- Directed multigraph
  - multiple edges between the same pair of nodes
- Any node and any edge may have a label
- Additionally, any node and any edge may have an arbitrary set of key-value pairs ("properties")
Property Graphs versus RDF Graphs

• Both data models have a lot of similarities:
  – Directed multigraphs
  – Labels on edges and on vertexes
  – Attributes with values on vertexes

• However, there are some subtle differences:
  – No edge properties in RDF graphs
  – Edge labels cannot appear as nodes in a PG (in RDF we may have <s1,p1,o1> and <p1,p2,o2>)
  – No multivalued (vertex) properties in a PG (unless we use a collection object as the value)
  – Node and edge identifiers in a PG are local to the PG, whereas URIs are globally unique identifiers (important for data integration)
Generic Graphs

• Data model
  – Directed multigraphs
  – Arbitrary user-defined data structure can be used as value of a vertex or an edge (e.g., a Java object)

• Example (Flink Gelly API):

```java
// create new vertexes with a Long ID and a String value
Vertex<Long, String> v1 = new Vertex<Long, String>(1L, "foo");
Vertex<Long, String> v2 = new Vertex<Long, String>(2L, "bar");

Edge<Long, Double> e = new Edge<Long, Double>(1L, 2L, 0.5);
```

• Advantage: give users maximum flexibility
• Drawback: systems cannot provide built-in operators related to vertex data or edge data
Graph Databases
Categories of Graph Data Systems

• **Triple stores**
  - Typically, pattern matching queries
  - Data model: RDF

• **Graph databases**
  - Typically, navigational queries
  - Prevalent data model: property graphs

• **Graph processing systems**
  - Typically, complex graph analysis tasks
  - Prevalent data model: generic graphs
Examples of Graph DB Systems

- Systems that focus on graph databases
  - Neo4j
  - Sparksee
  - Titan
  - InfiniteGraph

- Multi-model NoSQL stores with support for graphs:
  - OrientDB
  - ArangoDB

- Triple stores with TinkerPop support
  - Blazegraph
  - Stardog
  - IBM System G
Apache TinkerPop

- **Graph computing framework**
  - Vendor-agnostic
- **Includes a graph structure API**
  - Formerly known as Blueprints API
  - For creating and modifying Property Graphs
  - Example:

  ```java
  Graph graph = ...
  Vertex marko = graph.addVertex(T.label, "person", T.id, 1, "name", "marko", "age", 29);
  Vertex vadas = graph.addVertex(T.label, "person", T.id, 2, "name", "vadas", "age", 27);
  marko.addEdge("knows", vadas, T.id, 7, "weight", 0.5f);
  ```

- **Also includes a process API**
  - Graph-parallel engine
  - Graph traversal, based on a language called Gremlin
Gremlin Graph Traversal Language

• Part of the TinkerPop framework
• Powerful domain-specific language (DSL) with embeddings in various programming languages
• Expressions specify a concatenation of traversal steps
Gremlin Example

g.V().has('name','marko').out('knows').values('name')

Result:
===>vadas
===>josh
Gremlin Example

g.V().has('name', 'marko').out('knows').values('name').path()

Result:

==>[v[1], v[2], vadas]
==>[v[1], v[4], josh]
Gremlin Example

g.V().has('name','marko').repeat(out()).times(2).values('name')

Result:
===>ripple
===>lop
Cypher

- Declarative graph database query language
- Proprietary (used by Neo4j)
- The OpenCypher project aims to deliver an open specification
- Example
  - Recall our initial Gremlin example:
    
    ```
    g.V().has('name','marko').out('knows').values('name')
    ```
  - In Cypher we could express this query as follows:
    
    ```
    MATCH ( {name:'marko'} )-[[:knows]]->( x )
    RETURN x.name
    ```
Possible Clauses in Cypher Queries

CREATE - creates nodes and edges
DELETE - removes nodes, edges, properties
SET - sets values of properties
MATCH - specifies a pattern to match in the data graph
WHERE - filters pattern matching results
RETURN - which nodes / edges / properties in the matched data should be returned
UNION - merges results from two or more queries
WITH - chains subsequent query parts (like piping in Unix commands)
Node Patterns in Cypher

- Node patterns may have different forms:
  - `( )` - matches any node
  - `(:person)` - matches nodes whose label is `person`
  - `( {name:'marko'} )` - matches nodes that have a property `name='marko'`
  - `(:person {name:'marko'})` - matches nodes that have both the label `person` and a property `name='marko'`

- Every node pattern can be assigned a variable
  - Can be used to refer to the matching node in another query clause or to express joins
  - For instance, `(x)`, `(x:person)`
Relationship Patterns in Cypher

- Relationship pattern must be placed between two node patterns and it may have different forms

  -`--->` or `<---` - matches any edge (with the given direction)
  -`[-[:knows]-]` - matches edges whose label is `knows`
  -`-[ {weight:0.5} ]->` - matches edges that have a property `weight=0.5`
  -`[:knows {weight:0.5} ]->` - matches edges that have both the label `knows` and a property `weight=0.5`
  -`[:knows*..4]->` - matches paths of `knows` edges of up to length 4

- Every relationship pattern can be assigned a `variable`
  - For instance, `<-[x:knows]-`
More Complex Cypher Patterns

• Node patterns and relationship patterns are just basic building blocks that can be combined into more complex patterns

• Examples:
  - MATCH (a)-[:knows]->()-[:knows]->(a)
    RETURN a

  - MATCH p = shortestPath(
        (:person {name:'marko'})-[*]->(:person {name:'josh'})
    )
    RETURN p
Filtering in Cypher

• Pattern matching results can be filtered out by using the WHERE clause (similar to SQL)

• Examples:
  
  - \textbf{MATCH} (a:person)-[x:knows]->(b:person) \textbf{WHERE} x.weight > 0.5 \text{ AND } x.weight < 0.9 \textbf{RETURN} a, b
  
  - \textbf{MATCH} ()-[x:knows]->() \textbf{WHERE} exists(x.weight) \textbf{RETURN} x
  
  - \textbf{MATCH} (a)-[:knows]->(b)-[x:knows]->(c) \textbf{WHERE} \text{ NOT } (a)-[:knows]->(c) \textbf{RETURN} *
Graph Processing Systems
Categories of Graph Data Systems

- **Triple stores**
  - Typically, pattern matching queries
  - Data model: RDF

- **Graph databases**
  - Typically, navigational queries
  - Prevalent data model: property graphs

- **Graph processing systems**
  - Typically, complex graph analysis tasks
  - Prevalent data model: generic graphs
Complex Graph Analysis Tasks???

• Tasks that require an *iterative processing* of the *entire graph* or large portions thereof

• Examples:
  - Centrality analysis (e.g., PageRank)
  - Clustering, connected components
  - Graph coloring
  - Diameter finding
  - All-pairs shortest path
  - Graph pattern mining (e.g., frequent subgraphs, community detection)
  - Machine learning (e.g., belief propagation, Gaussian non-negative matrix factorization)
Example: PageRank

\[
PR_{k+1}(v) = \frac{\sum_{v \text{IN}} PR_k(v_{\text{IN}})}{|Out(v_{\text{IN}})|}
\]

### Table

<table>
<thead>
<tr>
<th></th>
<th>(k=0)</th>
<th>(k=1)</th>
<th>(k=2)</th>
<th>(k=3)</th>
<th>(k=4)</th>
<th>(k=5)</th>
<th>(k=6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR(v1)</td>
<td>0.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR(v2)</td>
<td>0.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR(v3)</td>
<td>0.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR(v4)</td>
<td>0.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: PageRank

\[
PR_{k+1}(v) = \sum_{v_{\text{IN}}} PR_k(v_{\text{IN}}) / |\text{Out}(v_{\text{IN}})|
\]

\[
PR_2(v1) = \frac{PR_1(v3)}{1} + \frac{PR_1(v4)}{2} \\
= \frac{0.25}{1} + \frac{0.25}{2} \\
= 0.25 + 0.125 \\
= 0.375
\]

<table>
<thead>
<tr>
<th></th>
<th>k=0</th>
<th>k=1</th>
<th>k=2</th>
<th>k=3</th>
<th>k=4</th>
<th>k=5</th>
<th>k=6</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR(v1)</td>
<td>0.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR(v2)</td>
<td>0.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR(v3)</td>
<td>0.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR(v4)</td>
<td>0.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: PageRank

\[ PR_{k+1}(v) = \sum_{v_{IN}} PR_k(v_{IN}) / \lvert \text{Out}(v_{IN}) \rvert \]

\[ PR_2(v1) = PR_1(v3)/1 + PR_1(v4)/2 \]
\[ = 0.25/1 + 0.25/2 \]
\[ = 0.25 + 0.125 \]
\[ = 0.375 \]

<table>
<thead>
<tr>
<th></th>
<th>k=0</th>
<th>k=1</th>
<th>k=2</th>
<th>k=3</th>
<th>k=4</th>
<th>k=5</th>
<th>k=6</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR(v1)</td>
<td>0.25</td>
<td>0.37</td>
<td>0.43</td>
<td>0.35</td>
<td>0.39</td>
<td>0.38</td>
<td>0.38</td>
</tr>
<tr>
<td>PR(v2)</td>
<td>0.25</td>
<td>0.08</td>
<td>0.12</td>
<td>0.14</td>
<td>0.11</td>
<td>0.13</td>
<td>0.13</td>
</tr>
<tr>
<td>PR(v3)</td>
<td>0.25</td>
<td>0.33</td>
<td>0.27</td>
<td>0.29</td>
<td>0.29</td>
<td>0.28</td>
<td>0.28</td>
</tr>
<tr>
<td>PR(v4)</td>
<td>0.25</td>
<td>0.20</td>
<td>0.16</td>
<td>0.20</td>
<td>0.19</td>
<td>0.19</td>
<td>0.19</td>
</tr>
</tbody>
</table>

Convergence
Observation

• Many such algorithms iteratively propagate data along the graph structure by transforming intermediate vertex and edge values
Can we use MapReduce for this?
Can we use MapReduce for this?

- M/R does not directly support iterative algorithms
- Materializing intermediate results at each M/R iteration harms performance
- Extra M/R job on each iteration for checking whether a fixed point has been reached
- Additional issue for graph algorithms
  - Invariant graph-topology data reloaded and reprocessed at each iteration
  - Wastes I/O, CPU, and network bandwidth
Graph Processing Systems

Pregel Family
- Pregel
- Giraph
- Giraph++
- Mizan
- GPS
- Pregelix
- Pregel+

GraphLab Family
- GraphLab
- PowerGraph
- GraphChi (centralized)

Other Systems
- Trinity
- TurboGraph (centralized)
- Signal/Collect
Vertex-Centric Abstraction

- Many such algorithms iteratively propagate data along the graph structure by transforming intermediate vertex and edge values
  - These transformations are defined in terms of functions on the values of adjacent vertexes and edges
  - Hence, such algorithms can be expressed by specifying a function that can be applied to any vertex separately
- “Think like a vertex”
Vertex-Centric Abstraction (cont'd)

- Vertex compute function consists of three steps:
  1. Read all incoming messages from neighbors
  2. Update the value of the vertex
  3. Send messages to neighbors
- Additionally, function may “vote to halt” if a local convergence criterion is met
- Overall execution can be parallelized
  - Terminates when all vertexes have halted and no messages in transit
Example: Vertex-Centric PageRank

- Vertex compute function consists of three steps:
  1. Read all incoming messages from neighbors
  2. Update the value of the vertex
  3. Send messages to neighbors

- Additionally, function may “vote to halt” if a local convergence criterion is met

\[
PR_{k+1}(v) = \sum_{v_{in}} PR_k(v_{in}) / |Out(v_{in})|
\]

<table>
<thead>
<tr>
<th></th>
<th>PR(v1)</th>
<th>PR(v2)</th>
<th>PR(v3)</th>
<th>PR(v4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>k=0</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>k=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k=2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k=3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k=4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k=5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k=6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Vertex-Centric PageRank

- Vertex compute function consists of three steps:
  1. Read all incoming messages from neighbors
  2. Update the value of the vertex
  3. Send messages to neighbors
- Additionally, function may “vote to halt” if a local convergence criterion is met

\[ PR_{k+1}(v) = \sum_{v_{\text{in}}} PR_k(v_{\text{in}}) / |\text{Out}(v_{\text{in}})| \]

<table>
<thead>
<tr>
<th></th>
<th>PR(v1)</th>
<th>PR(v2)</th>
<th>PR(v3)</th>
<th>PR(v4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>k=0</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>k=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k=2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k=3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k=4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k=5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k=6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Vertex-Centric PageRank

- Vertex compute function consists of three steps:
  1. Read all incoming messages from neighbors
  2. Update the value of the vertex
  3. Send messages to neighbors
- Additionally, function may “vote to halt” if a local convergence criterion is met

\[ PR_{k+1}(v) = \sum_{v_{IN}} PR_k(v_{IN}) / |Out(v_{IN})| \]

<table>
<thead>
<tr>
<th></th>
<th>(k=0)</th>
<th>(k=1)</th>
<th>(k=2)</th>
<th>(k=3)</th>
<th>(k=4)</th>
<th>(k=5)</th>
<th>(k=6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(PR(v1))</td>
<td>0.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(PR(v2))</td>
<td>0.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(PR(v3))</td>
<td>0.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(PR(v4))</td>
<td>0.25</td>
<td>0.20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Vertex-Centric PageRank

- Vertex compute function consists of three steps:
  1. Read all incoming messages from neighbors
  2. Update the value of the vertex
  3. Send messages to neighbors
- Additionally, function may “vote to halt” if a local convergence criterion is met

\[
PR_{k+1}(v) = \sum_{v_{\text{IN}}} PR_k(v_{\text{IN}}) / |\text{Out}(v_{\text{IN}})|
\]

<table>
<thead>
<tr>
<th></th>
<th>(k=0)</th>
<th>(k=1)</th>
<th>(k=2)</th>
<th>(k=3)</th>
<th>(k=4)</th>
<th>(k=5)</th>
<th>(k=6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR(v1)</td>
<td>0.25</td>
<td>0.37</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR(v2)</td>
<td>0.25</td>
<td>0.08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR(v3)</td>
<td>0.25</td>
<td>0.33</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR(v4)</td>
<td>0.25</td>
<td>0.20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Vertex-Centric PageRank

- Vertex compute function consists of three steps:
  1. Read all incoming messages from neighbors
  2. Update the value of the vertex
  3. Send messages to neighbors

- Additionally, function may “vote to halt” if a local convergence criterion is met

\[
PR_{k+1}(v) = \sum_{v_{in}} PR_k(v_{in}) / |\text{Out}(v_{in})|
\]

<table>
<thead>
<tr>
<th></th>
<th>(k=0)</th>
<th>(k=1)</th>
<th>(k=2)</th>
<th>(k=3)</th>
<th>(k=4)</th>
<th>(k=5)</th>
<th>(k=6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR(v1)</td>
<td>0.25</td>
<td>0.37</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR(v2)</td>
<td>0.25</td>
<td>0.08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR(v3)</td>
<td>0.25</td>
<td>0.33</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR(v4)</td>
<td>0.25</td>
<td>0.20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Vertex-Centric PageRank

- Vertex compute function consists of three steps:
  1. Read all incoming messages from neighbors
  2. Update the value of the vertex
  3. Send messages to neighbors
- Additionally, function may “vote to halt” if a local convergence criterion is met

\[ PR_{k+1}(v) = \sum_{v_{in}} PR_k(v_{in}) / |Out(v_{in})| \]

<table>
<thead>
<tr>
<th></th>
<th>( k=0 )</th>
<th>( k=1 )</th>
<th>( k=2 )</th>
<th>( k=3 )</th>
<th>( k=4 )</th>
<th>( k=5 )</th>
<th>( k=6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR(v1)</td>
<td>0.25</td>
<td>0.37</td>
<td>0.43</td>
<td>0.35</td>
<td>0.39</td>
<td>0.38</td>
<td></td>
</tr>
<tr>
<td>PR(v2)</td>
<td>0.25</td>
<td>0.08</td>
<td>0.12</td>
<td>0.14</td>
<td>0.11</td>
<td>0.13</td>
<td></td>
</tr>
<tr>
<td>PR(v3)</td>
<td>0.25</td>
<td>0.33</td>
<td>0.27</td>
<td>0.29</td>
<td>0.29</td>
<td>0.28</td>
<td></td>
</tr>
<tr>
<td>PR(v4)</td>
<td>0.25</td>
<td>0.20</td>
<td>0.16</td>
<td>0.20</td>
<td>0.19</td>
<td>0.19</td>
<td></td>
</tr>
</tbody>
</table>
Example: Vertex-Centric PageRank

• Vertex compute function consists of three steps:
  1. Read all incoming messages from neighbors
  2. Update the value of the vertex
  3. Send messages to neighbors
• Additionally, function may “vote to halt” if a local convergence criterion is met

\[
PR_{k+1}(v) = \sum_{v_{\text{in}}} PR_k(v_{\text{in}}) / |\text{Out}(v_{\text{in}})|
\]

<table>
<thead>
<tr>
<th></th>
<th>k=0</th>
<th>k=1</th>
<th>k=2</th>
<th>k=3</th>
<th>k=4</th>
<th>k=5</th>
<th>k=6</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR(v1)</td>
<td>0.25</td>
<td>0.37</td>
<td>0.43</td>
<td>0.35</td>
<td>0.39</td>
<td>0.38</td>
<td>0.38</td>
</tr>
<tr>
<td>PR(v2)</td>
<td>0.25</td>
<td>0.08</td>
<td>0.12</td>
<td>0.14</td>
<td>0.11</td>
<td>0.13</td>
<td>0.13</td>
</tr>
<tr>
<td>PR(v3)</td>
<td>0.25</td>
<td>0.33</td>
<td>0.27</td>
<td>0.29</td>
<td>0.29</td>
<td>0.28</td>
<td>0.28</td>
</tr>
<tr>
<td>PR(v4)</td>
<td>0.25</td>
<td>0.20</td>
<td>0.16</td>
<td>0.20</td>
<td>0.19</td>
<td>0.19</td>
<td>0.19</td>
</tr>
</tbody>
</table>
Google Pregel

- First system that implemented vertex-centric computation for shared-nothing clusters
  - Communication through message passing

- Based on the *bulk synchronous parallel (BSP)* programming model
  - Supersteps with synchronization barriers

- Apache Giraph was a first open source implementation of Pregel
MapReduce versus Pregel

- Requires passing of entire graph topology from one iteration to the next
  - Graph topology is not passed across iterations, vertexes only send their state to their neighbors

- Intermediate result after each iteration is stored on disk and then read again from disk
  - Main memory based

- Programmer needs to write a driver program to support iterations, and another M/R job to check for fixed point
  - Usage of supersteps and master-client architecture makes programming easy
Limitation of Pregel

- In the BSP model, performance is limited by slowest worker machine
  - Many real-world graphs have power-law degree distribution, which may lead to few highly-loaded workers
Limitation of Pregel

- In the BSP model, performance is limited by slowest worker machine
  - Many real-world graphs have power-law degree distribution, which may lead to few highly-loaded workers

- Possible optimizations to balance the workload:
  - Decompose the vertex program
  - Sophisticated graph partitioning
  - Graph-centric abstraction

- Another possibility: asynchronous execution (instead of BSP)
Combiner

• Takes two messages and combines them into one
  - Associative, commutative function

• Can be used to aggregate messages before sending them to the worker node that has the target vertex

• Example:
  - In the vertex-centric PageRank, messages are values $m_{IN} = (Pr_k(v_{IN}) / |Out(v_{IN})|)$ of each incoming neighbor $v_{IN}$
  - In the vertex function these values are summed up:
    $$(Pr_k(v_{IN1}) / |Out(v_{IN1})|) + (Pr_k(v_{IN2}) / |Out(v_{IN2})|) + ...$$
  - Parts of this sum may be computed by worker nodes that have some of the incoming neighbor vertexes
Signal/Collect Model

- **Signaling (edge function):**
  - Every edge uses the value of its source vertex to compute a message ("signal") for the target vertex
  - Executed on the worker that has the source vertex

- **Collecting (vertex function):**
  - Every vertex computes its new value based on the messages received from its incoming edges
  - Executed on the worker that has the target vertex
Gather, Apply, Scatter (GAS) Model

- **Gather:**
  - Accumulate incoming messages, i.e., same purpose as a combiner

- **Apply:**
  - Update the vertex value based on the accumulated information
  - Operates only on the vertex

- **Scatter:**
  - Computes outgoing messages
  - Can be executed in parallel for each adjacent edge
Limitation of Pregel

- In the BSP model, performance is limited by slowest worker machine
  - Many real-world graphs have power-law degree distribution, which may lead to few highly-loaded workers

- Possible optimizations to balance the workload:
  - Decompose the vertex program
  - Sophisticated graph partitioning
  - Graph-centric abstraction

- Another possibility: asynchronous execution (instead of BSP)
Partitioning

- **Goal:** distribute the vertexes to achieve a balanced workload while minimizing inter-partition edges to avoid costly network traffic

  - For instance, hash-based (random) partitioning has extremely poor locality

- Unfortunately, the problem is NP-complete
  - k-way graph partitioning problem

- Various heuristics and approximation algorithms
Vertex-Cut

- PowerGraph introduced a partitioning scheme that “cuts” vertexes such that the edges of high-degree vertexes are handled by multiple workers
  - improved work balance

- Power-law graphs have good vertex cuts
  - Communication is linear in the number of machines each vertex spans
  - Vertex-cut minimizes this number
  - Hence, reduced network traffic
Limitation of Pregel

- In the BSP model, performance is limited by slowest worker machine
  - Many real-world graphs have power-law degree distribution, which may lead to few highly-loaded workers

- Possible optimizations to balance the workload:
  - Decompose the vertex program
  - Sophisticated graph partitioning
  - Graph-centric abstraction

- Another possibility: asynchronous execution (instead of BSP)
Acknowledgements:
- Some of the slides about graph processing systems are from a slideset of Sherif Sakr. Thanks Sherif!

Image sources:
- Example Property Graph http://tinkerpop.apache.org/docs/current/tutorials/getting-started/
- BSP Illustration https://en.wikipedia.org/wiki/Bulk_synchronous_parallel
- Smiley https://commons.wikimedia.org/wiki/File:Face-smile.svg
- Frowny https://commons.wikimedia.org/wiki/File:Face-sad.svg