An Introduction to Graph Analytics Platforms

(Very Short Version)

M. Tamer Özsu

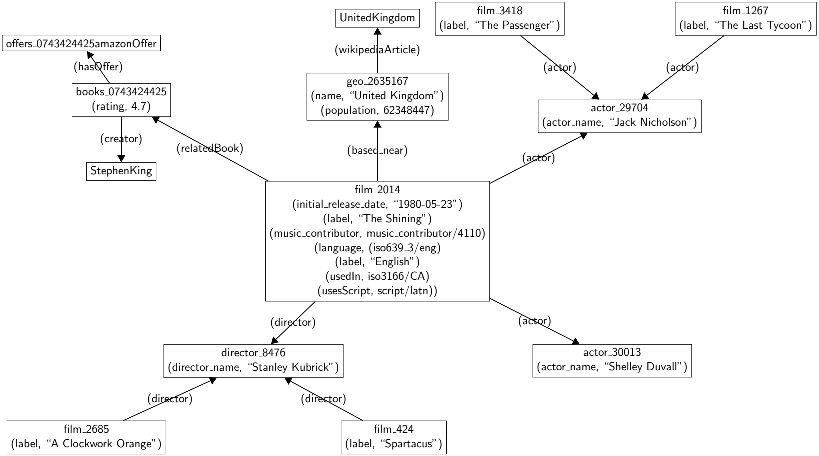
University of Waterloo
David R. Cheriton School of Computer Science

Outline

Outline

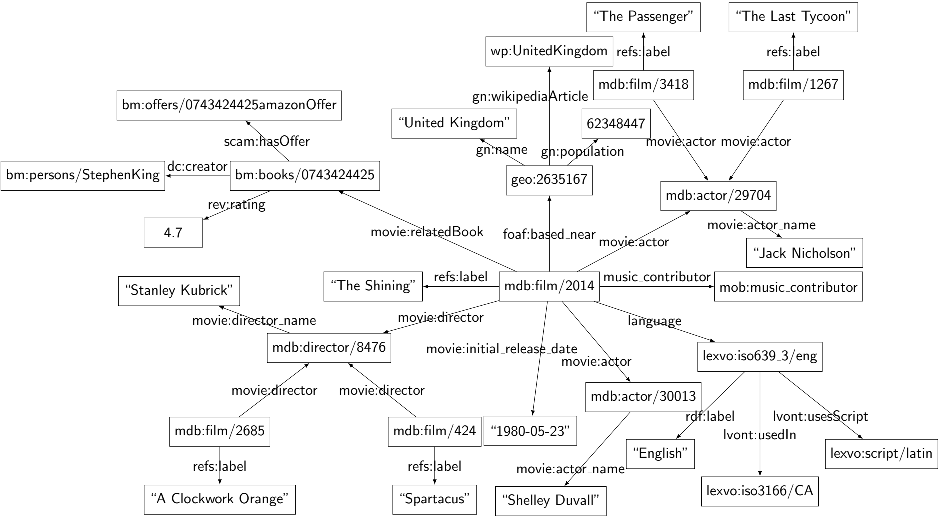
Graph Types

Property graph



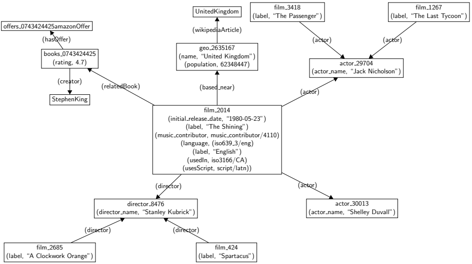
Graph Types

RDF graph

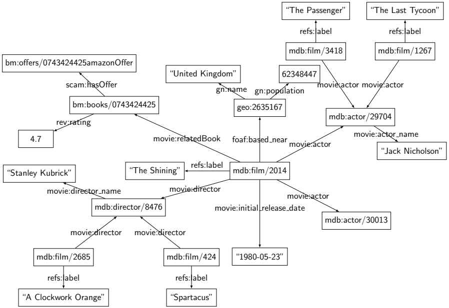


Graph Types

Property graph



RDF graph



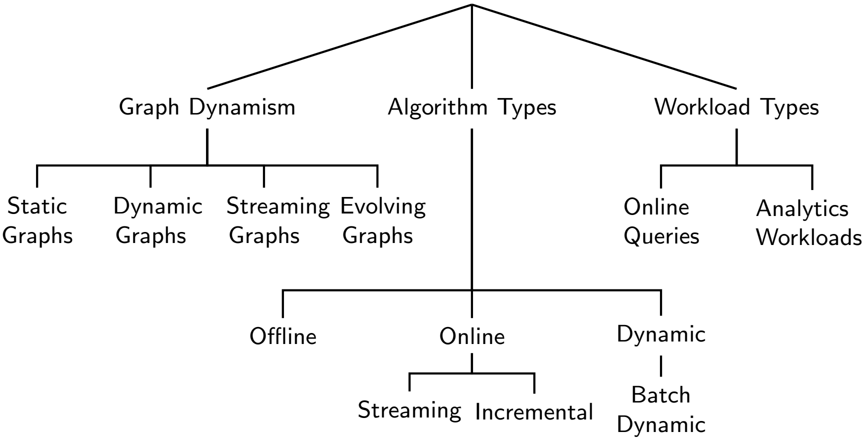
- Workload: Online queries and analytic workloads
- Query execution: Varies

- Workload: SPARQL queries
- Query execution: subgraph matching by homomorphism

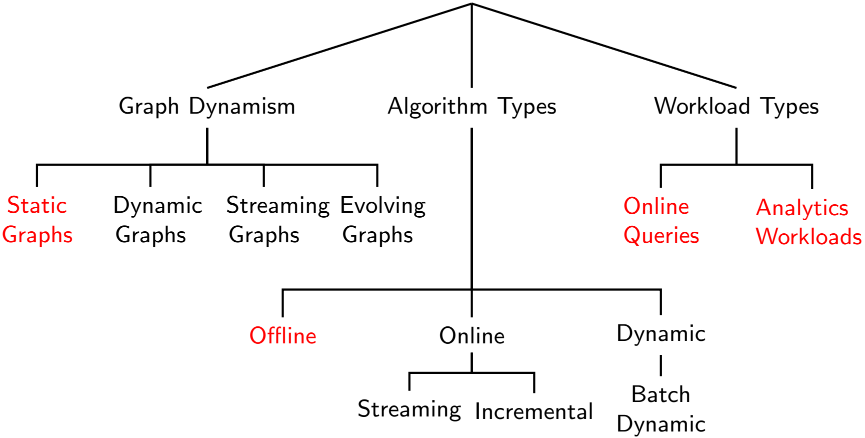
Outline

Outline

Classification Summary

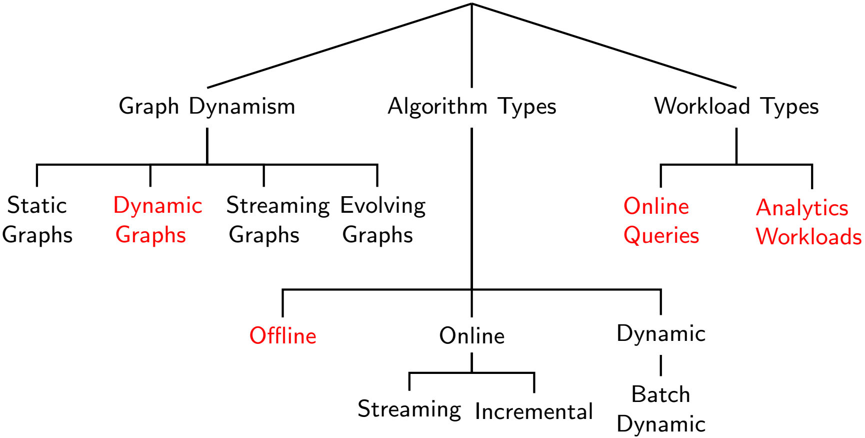


Example Design Points



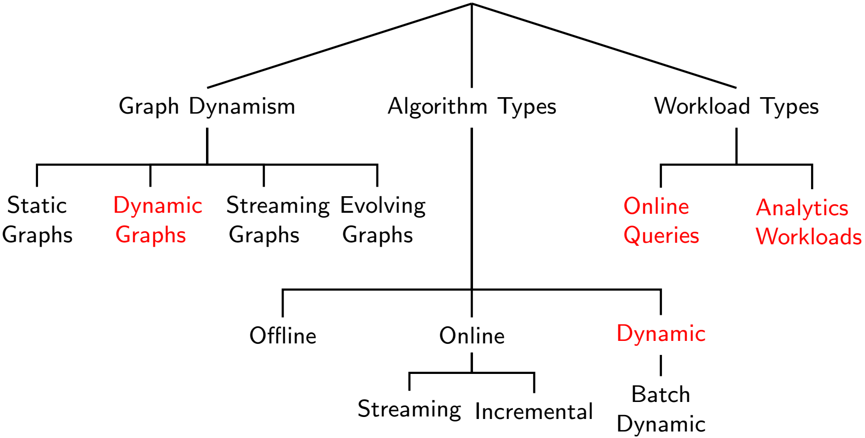
Compute the query result/perform analytic computation over the graph as it exists.

Example Design Points



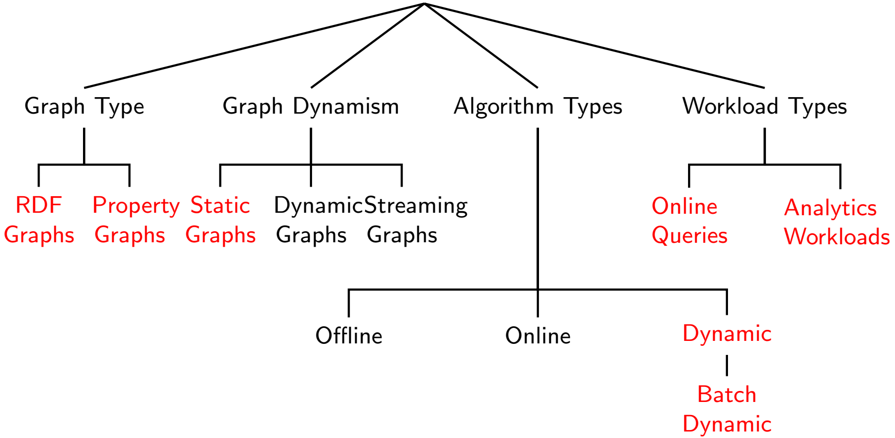
Compute the query result/perform analytic computation on each snapshot from scratch.

Example Design Points



Continuously compute the query result/perform analytic computation as the input changes.

Example Design Points – Not all alternatives make sense



Dynamic (or batch-dynamic) algorithms do not make sense for static graphs.

Scale-up or Scale-out? [Lin, 2018]

- Scale-up: Single machine execution
  - Graph datasets are small and can fit in a single machine – even in main memory
  - Single machine avoids parallel execution complexities

Scale-up or Scale-out?

[Lin, 2018]

- Scale-up: Single machine execution
  - Graph datasets are small and can fit in a single machine – even in main memory
  - Single machine avoids parallel execution complexities
- Scale-out: Parallel execution
  - Graph data sets grow when they are expanded to their storage formats

Dataset	$\|V\|$	$\|E\|$	Regular size	Single Machine*
Live Journal	4,847,571	68,993,773	1.08GB	6.3GB
USA Road	23,947,347	58,333,344	951MB	9.09GB
Twitter	41,652,230	1,468,365,182	26GB	128 GB
UK0705	82,240,700	2,829,101,180	48GB	247GB
World Road	682,496,072	717,016,716	15GB	194GB
CommonCrawl2014	1,727,000,000	64,422,000,000	1.3TB	Out of memory

* Using (PowerLyra)

Scale-up or Scale-out? [Lin, 2018]

- Scale-up: Single machine execution
  - Graph datasets are small and can fit in a single machine – even in main memory
  - Single machine avoids parallel execution complexities
- Scale-out: Parallel execution
  - Graph data sets grow when they are expanded to their storage formats
  - Workstations big enough to handle even smaller datasets are still expensive
  - Some graphs are very large: Alibaba: several billion vertices, >100 million edges
  - Dataset size may not be the determinant $\Rightarrow$ parallelizing computation is important

Scale-up or Scale-out? [Lin, 2018]

- Scale-up: Single machine execution
  - Graph datasets are small and can fit in a single machine – even in main memory
  - Single machine avoids parallel execution complexities
- Scale-out: Parallel execution
  - Graph data sets grow when they are expanded to their storage formats
  - Workstations big enough to handle even smaller datasets are still expensive
  - Some graphs are very large: Alibaba: several billion vertices, $>$ 100 million edges
  - Dataset size may not be the determinant $\Rightarrow$ parallelizing computation is important

We focus on parallel graph analytics systems

Graph Partitioning

- Edge-cut (vertex-disjoint)
  - Achieve disjoint partitions by allocating each vertex to a partition
  - Objective 1: Partitions should be balanced
  - Objective 2: Minimize edge-cuts (to reduce communication)
  - Good for graphs with low-degree vertices, not for power-law graphs
  - Examples: Hashing, METIS [Karypis and Kumar, 1995], label propagation algorithms [Ugander and Backstrom, 2013]

Graph Partitioning

- Edge-cut (vertex-disjoint)
  - Achieve disjoint partitions by allocating each vertex to a partition
  - Objective 1: Partitions should be balanced
  - Objective 2: Minimize edge-cuts (to reduce communication)
  - Good for graphs with low-degree vertices, not for power-law graphs
  - Examples: Hashing, METIS [Karypis and Kumar, 1995], label propagation algorithms [Ugander and Backstrom, 2013]
- Vertex-cut (edge-disjoint)
  - Achieve disjoint partitions by allocating each edge to a partition and replicating vertices as necessary
  - Objective 1: Partitions should be balanced
  - Objective 2: Minimize vertex-cuts (to reduce replica cost)
  - Perform better on power-law graphs
  - Examples: Hashing, greedy algorithms [Gonzalez et al., 2012]

Graph Partitioning

- Edge-cut (vertex-disjoint)
  - Achieve disjoint partitions by allocating each vertex to a partition
  - Objective 1: Partitions should be balanced
  - Objective 2: Minimize edge-cuts (to reduce communication)
  - Good for graphs with low-degree vertices, not for power-law graphs
  - Examples: Hashing, METIS [Karypis and Kumar, 1995], label propagation algorithms [Ugander and Backstrom, 2013]
- Vertex-cut (edge-disjoint)
  - Achieve disjoint partitions by allocating each edge to a partition and replicating vertices as necessary
  - Objective 1: Partitions should be balanced
  - Objective 2: Minimize vertex-cuts (to reduce replica cost)
  - Perform better on power-law graphs
  - Examples: Hashing, greedy algorithms [Gonzalez et al., 2012]
- Hybrid
  - Edge-cut for low-degree vertices/vertex-cut for high-degree ones
  - PowerLyra [Chen et al., 2015]

Graph Workloads

Online graph querying

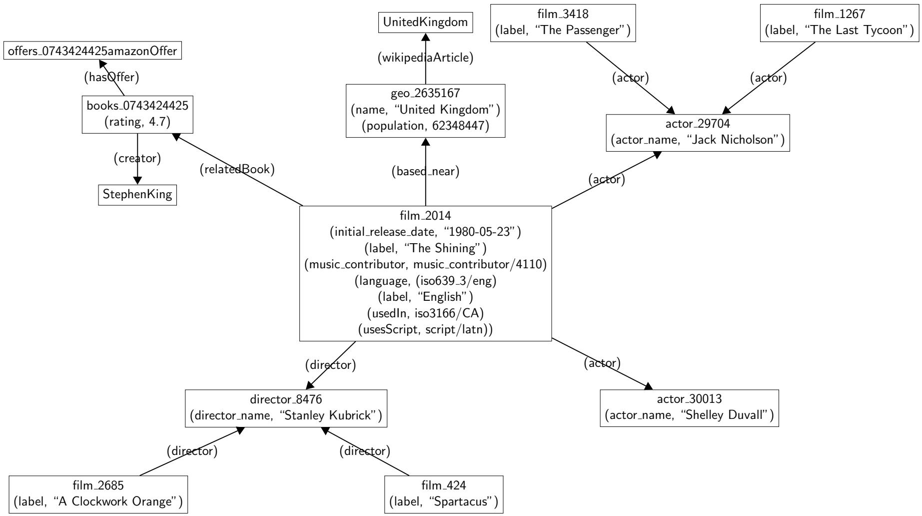
- Reachability
- Single source shortest-path
- Subgraph matching
- SPARQL queries

Offline graph analytics

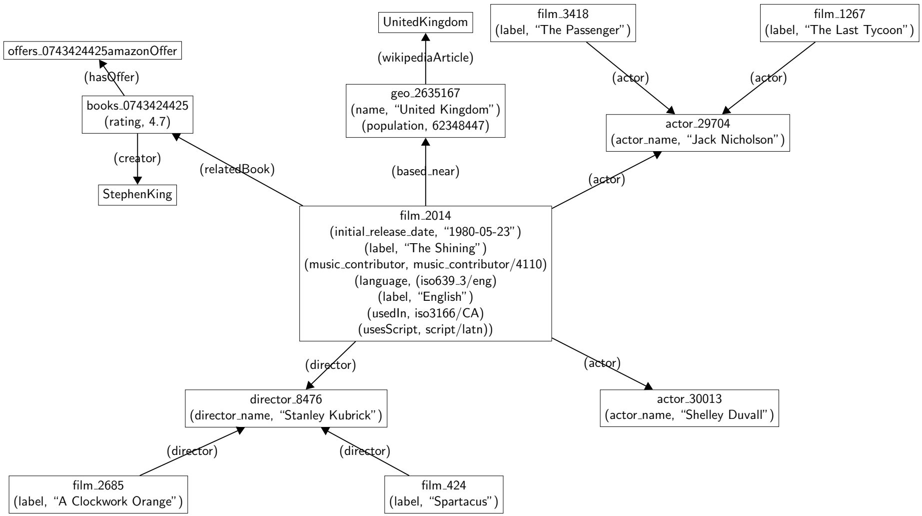
- PageRank
- Clustering
- Connected components
- Diameter finding
- Graph colouring
- All pairs shortest path
- Graph pattern mining
- Machine learning algorithms (Belief propagation, Gaussian non-negative matrix factorization)

Outline

Reachability Queries

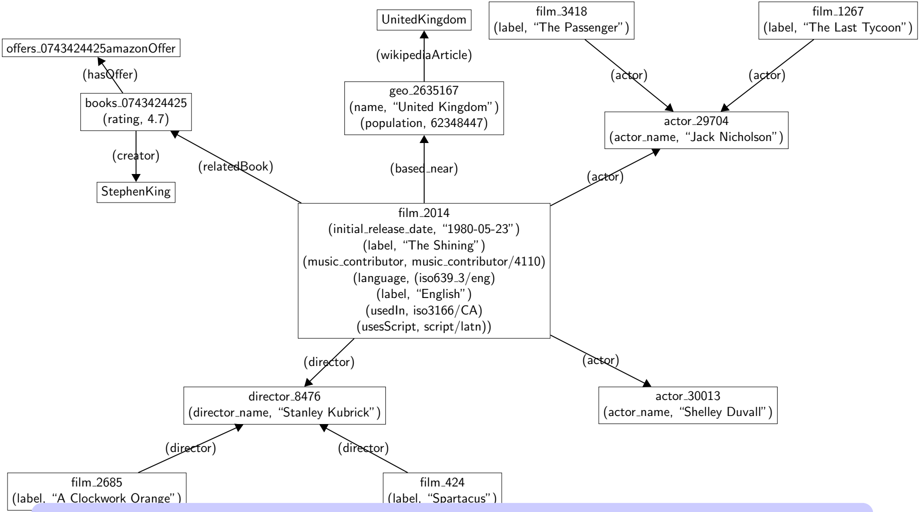


Reachability Queries



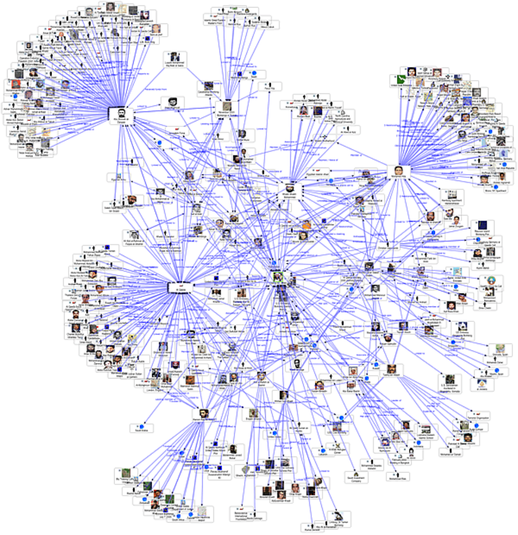
Can you reach film_1267 from film_2014?

Reachability Queries



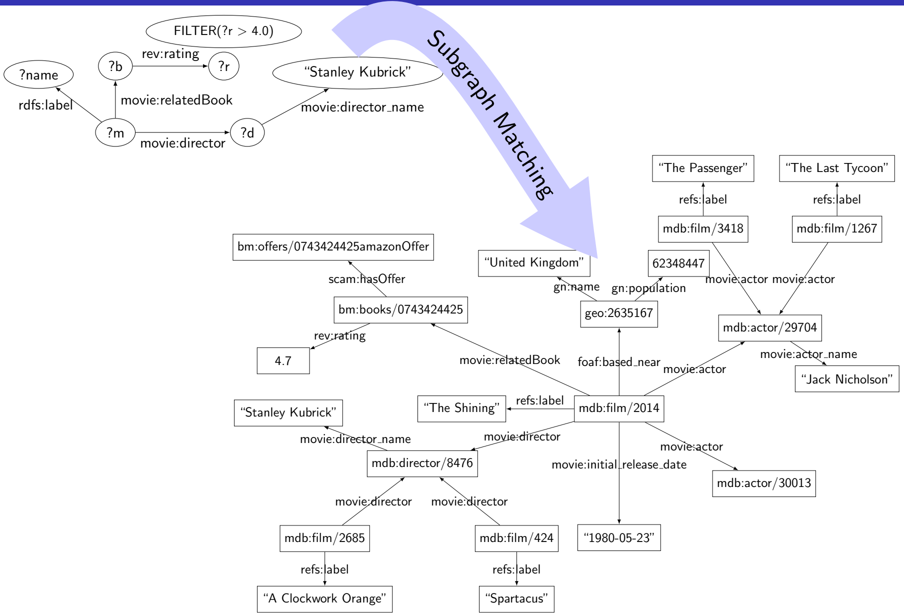
Is there a book whose rating is > 4.0 associated with a film that was directed by Stanley Kubrick?

Reachability Queries



Think of Facebook graph and finding friends of friends.

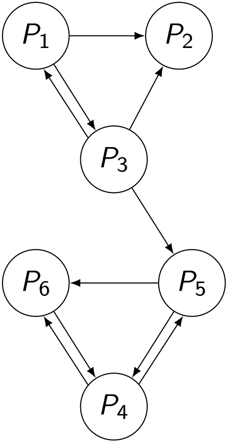
Subgraph Matching



Outline

PageRank Computation

A web page is important if it is pointed to by other important pages.



$$r(P_i) = (1-d) + d \sum_{P_j \in B_{P_i}} \frac{r(P_j)}{|F_{P_j}|}$$

(let $d = 1$)

$$r(P_2) = \frac{r(P_1)}{2} + \frac{r(P_3)}{3}$$

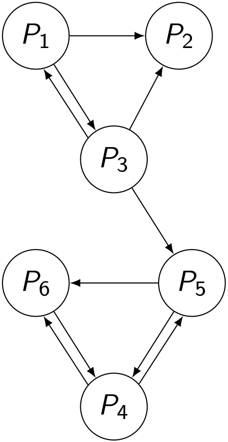
$$r_{k+1}(P_i) = \sum_{P_j \in B_{P_i}} \frac{r_k(P_j)}{|F_{P_j}|}$$

B_{P_i}: in-neighbours of P_i

F_{P_i}: out-neighbours of P_i

PageRank Computation

A web page is important if it is pointed to by other important pages.



$$r_{k+1}(P_i) = \sum_{P_j \in B_{P_i}} \frac{r_k(P_j)}{|F_{P_j}|}$$

Iteration 0	Iteration 1	Iteration 2	Rank at Iter. 2
$r_0(P_1) = 1/6$	$r_1(P_1) = 1/18$	$r_2(P_1) = 1/36$	5
$r_0(P_2) = 1/6$	$r_1(P_2) = 5/36$	$r_2(P_2) = 1/18$	4
$r_0(P_3) = 1/6$	$r_1(P_3) = 1/12$	$r_2(P_3) = 1/36$	5
$r_0(P_4) = 1/6$	$r_1(P_4) = 1/4$	$r_2(P_4) = 17/72$	1
$r_0(P_5) = 1/6$	$r_1(P_5) = 5/36$	$r_2(P_5) = 11/72$	3
$r_0(P_6) = 1/6$	$r_1(P_6) = 1/6$	$r_2(P_6) = 14/72$	2

Iterative processing
Touch each vertex

Outline

Outline

Can MapReduce be Used for Graph Analytics?

- Yes; `map` and `reduce` functions can be written for graph analytics workloads
  - Scalable Graph processing Class $\mathcal{SGC}$ [Qin et al., 2014]
  - Connected component computation [Kiveris et al., 2014; Rastogi et al., 2013]
- Not suitable for iterative processing due to data movement at each stage
  - No guarantee that computation will be assigned to the same worker nodes in the next round
- High I/O cost
  - Need to save in storage system (HDFS) intermediate results of each iteration

Can MapReduce be Used for Graph Analytics?

- Yes; `map` and `reduce` functions can be written for graph analytics workloads
  - Scalable Graph processing Class $\mathcal{SGC}$ [Qin et al., 2014]
  - Connected component computation [Kiveris et al., 2014; Rastogi et al., 2013]
- Not suitable for iterative processing due to data movement at each stage
  - No guarantee that computation will be assigned to the same worker nodes in the next round
- High I/O cost
  - Need to save in storage system (HDFS) intermediate results of each iteration
- There are systems that address these concerns
  - HaLoop [Bu et al., 2010, 2012]
  - GraphX over Spark [Gonzalez et al., 2014]

Spark System

- Spark objectives
  - Better support for iterative programs
  - Provide a complete ecosystem
  - Similar abstraction (to MapReduce) for programming
  - Maintain MapReduce fault-tolerance and scalability

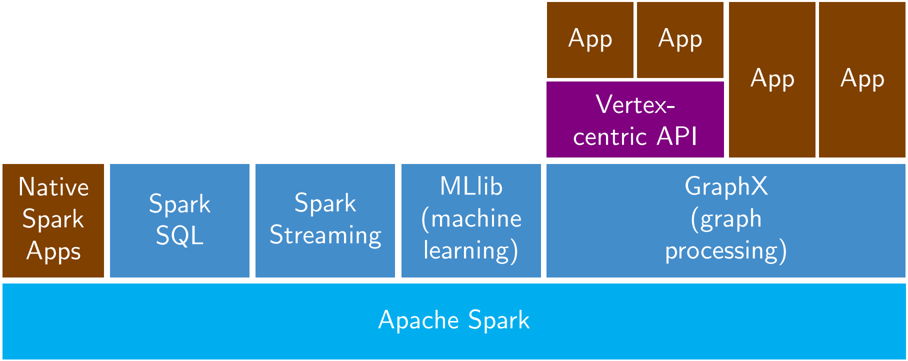
Spark System

- Spark objectives
  - Better support for iterative programs
  - Provide a complete ecosystem
  - Similar abstraction (to MapReduce) for programming
  - Maintain MapReduce fault-tolerance and scalability
- Fundamental concepts
  - RDD: Reliable Distributed Datasets
  - Caching of working set
  - Maintaining lineage for fault-tolerance

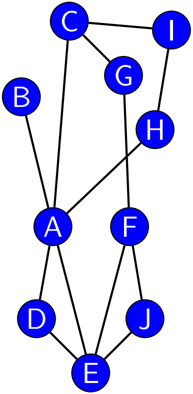
GraphX

[Gonzalez et al., 2014]

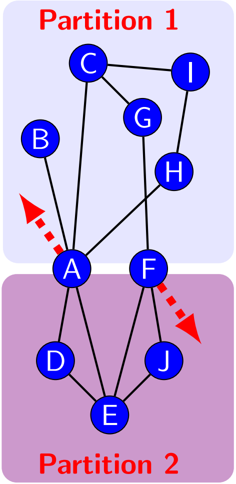
- Built on top of Spark
- Objective is to combine data analytics with graph processing
  - Unify computation on tables and graphs
- Carefully convert graph to tabular representation
- Native GraphX API or can accommodate vertex-centric computation



GraphX: Representation of Graphs as Tables

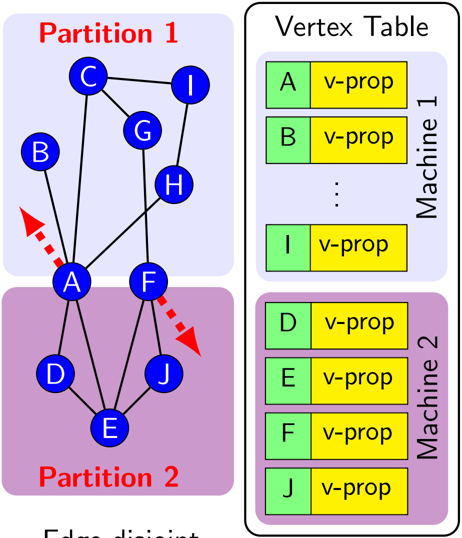


GraphX: Representation of Graphs as Tables



Edge-disjoint partitioning

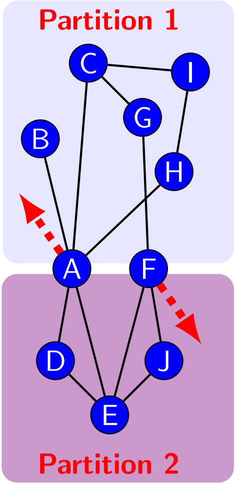
GraphX: Representation of Graphs as Tables



Edge-disjoint partitioning

(RDD)
v-prop:vertex prop.

GraphX: Representation of Graphs as Tables



Edge-disjoint partitioning

Vertex Table

A	v-prop	Machine 1
B	v-prop	
⋮		
I	v-prop	
D	v-prop	Machine 2
E	v-prop	
F	v-prop	
J	v-prop	

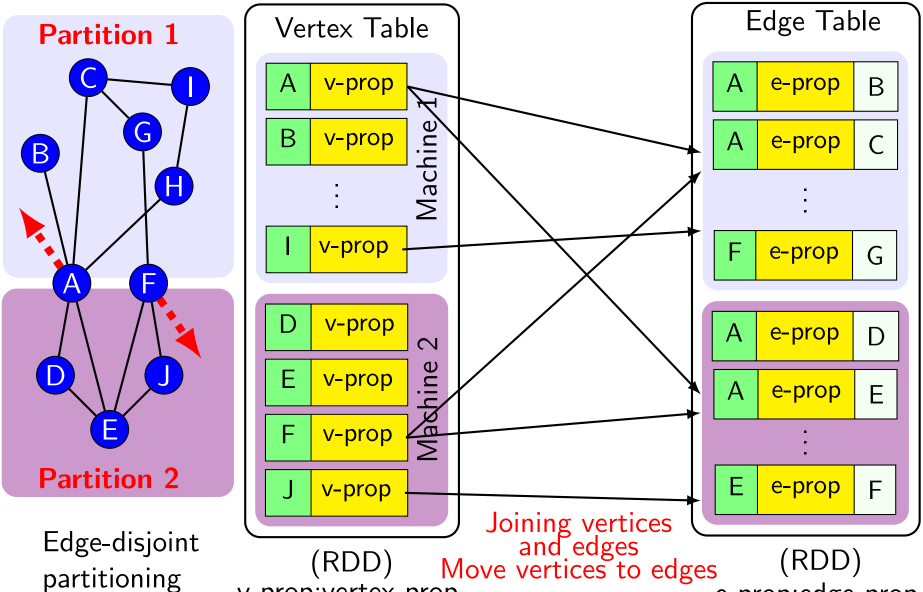
(RDD)
v-prop:vertex prop.

Edge Table

A	e-prop	B
A	e-prop	C
⋮		
F	e-prop	G
A	e-prop	D
A	e-prop	E
⋮		
E	e-prop	F

(RDD)
e-prop:edge prop.

GraphX: Representation of Graphs as Tables

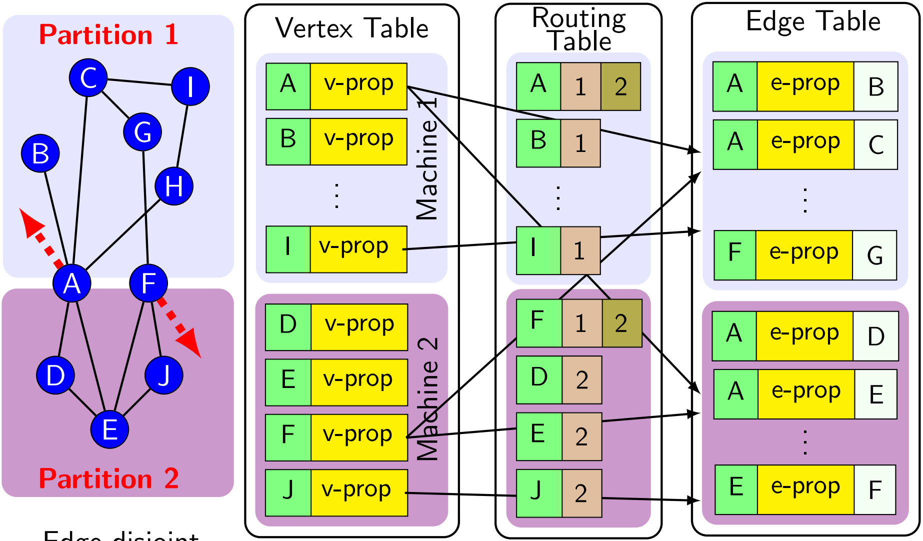


Edge-disjoint partitioning

(RDD)
v-prop:vertex prop.

(RDD)
e-prop:edge prop.

GraphX: Representation of Graphs as Tables



Edge-disjoint partitioning

(RDD)
v-prop:vertex prop.

(RDD)

(RDD)
e-prop:edge prop.

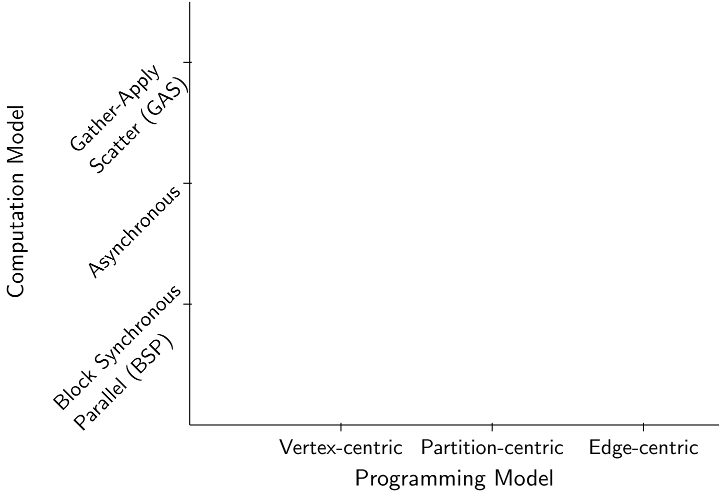
Outline

Classification of Graph Processing Systems [Han, 2015]

- Programming model
- Computation model

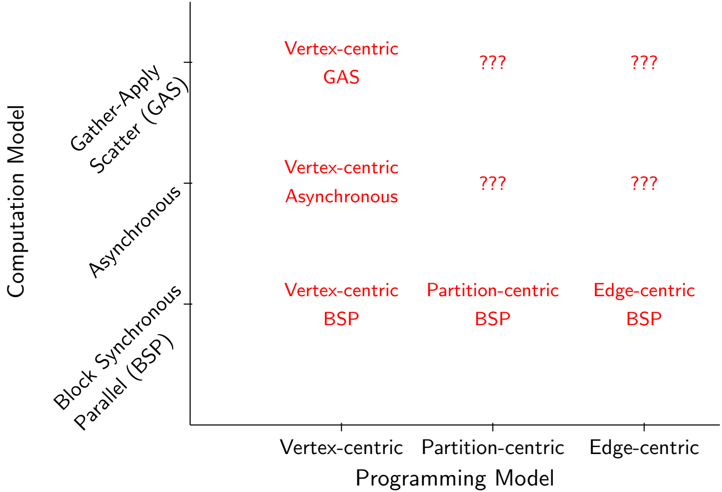
Classification of Graph Processing Systems [Han, 2015]

- Programming model
- Computation model



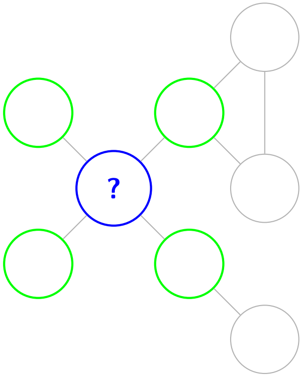
Classification of Graph Processing Systems [Han, 2015]

- Programming model
- Computation model



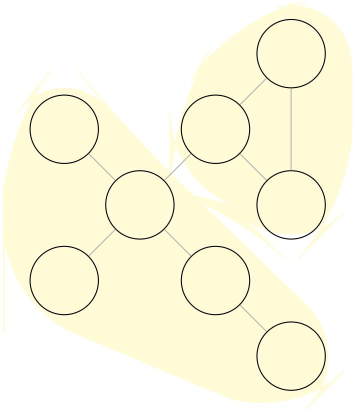
Programming Models

- Vertex-centric
  - Computation on a vertex is the focus
  - "Think like a vertex"
  - Vertex computation depends on its own state + states of its neighbors
  - `Compute(vertex v)`
  - `GetValue()`, `WriteValue()`



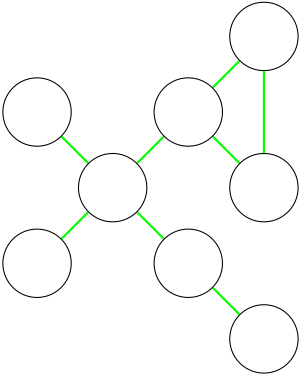
Programming Models

- Vertex-centric
- Partition-centric (Block-centric)
  - Computation on an entire partition is specified
  - "Think like a block" or "Think like a graph"
  - Aim is to reduce the communication cost among vertices



Programming Models

- Vertex-centric
- Partition-centric (Block-centric)
- Edge-centric
  - Computation is specified on each edge rather than on each vertex or block
  - `Compute(edge e)`



Computational Models

- Block Synchronous Parallel (BSP) [Valiant, 1990]

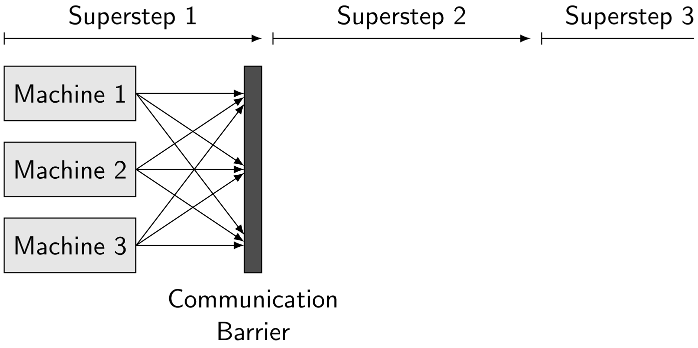
Computational Models

- Block Synchronous Parallel (BSP) [Valiant, 1990]

Computation

Computational Models

- Block Synchronous Parallel (BSP) [Valiant, 1990]

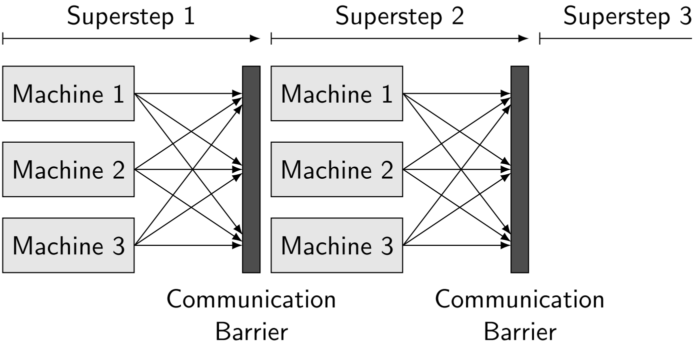


Each machine performs computation on its graph partition

At the end of each superstep results are pushed to other workers

Computational Models

- Block Synchronous Parallel (BSP) [Valiant, 1990]

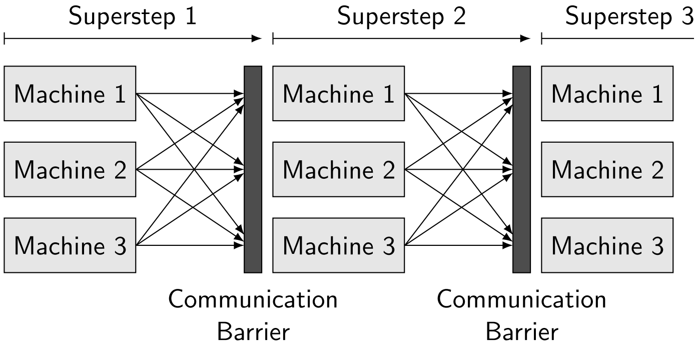


Each machine performs computation on its graph partition

At the end of each superstep results are pushed to other workers

Computational Models

- Block Synchronous Parallel (BSP) [Valiant, 1990]



Each machine performs computation on its graph partition

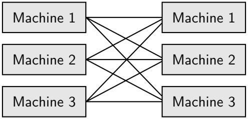
At the end of each superstep results are pushed to other workers

Computational Models

- Block Synchronous Parallel (BSP) [Valiant, 1990]
- Asynchronous Parallel

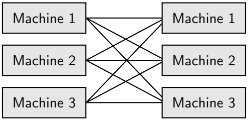
Computational Models

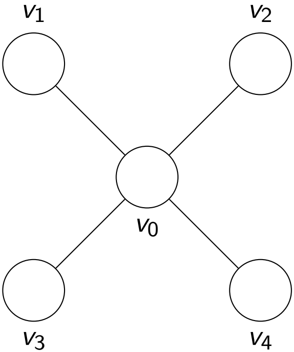
- Block Synchronous Parallel (BSP) [Valiant, 1990]
- Asynchronous Parallel
  - No communication barriers. ✓
  - Uses the *most recent* values. ✓
  - Implemented via distributed locking



Computational Models

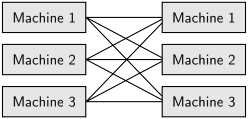
- Block Synchronous Parallel (BSP) [Valiant, 1990]
- Asynchronous Parallel
  - No communication barriers. ✓
  - Uses the *most recent* values. ✓
  - Implemented via distributed locking
  - Consider vertex-centric program

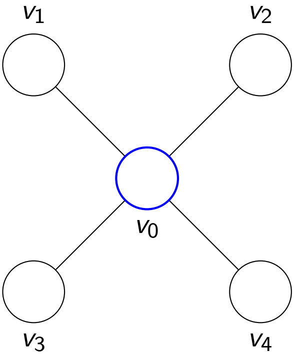




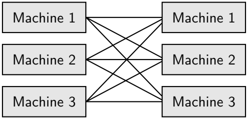
Computational Models

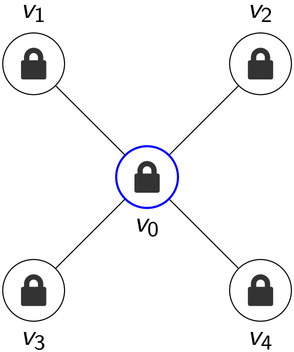
- Block Synchronous Parallel (BSP) [Valiant, 1990]
- Asynchronous Parallel
  - No communication barriers. ✓
  - Uses the *most recent* values. ✓
  - Implemented via distributed locking
  - Consider vertex-centric program





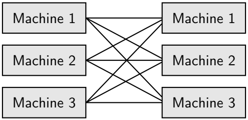
- Block Synchronous Parallel (BSP) [Valiant, 1990]
- Asynchronous Parallel
  - No communication barriers. ✓
  - Uses the *most recent* values. ✓
  - Implemented via distributed locking
  - Consider vertex-centric program

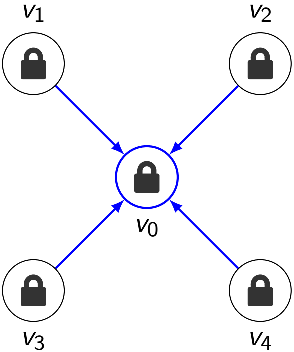




Computational Models

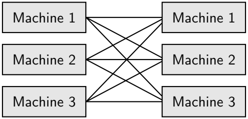
- Block Synchronous Parallel (BSP) [Valiant, 1990]
- Asynchronous Parallel
  - No communication barriers. ✓
  - Uses the *most recent* values. ✓
  - Implemented via distributed locking
  - Consider vertex-centric program

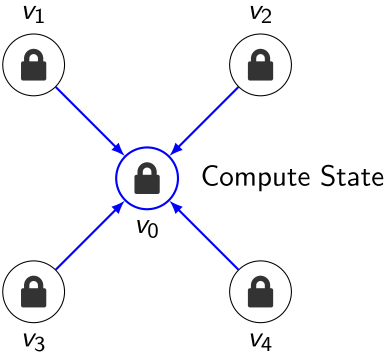




Computational Models

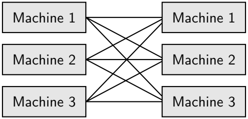
- Block Synchronous Parallel (BSP) [Valiant, 1990]
- Asynchronous Parallel
  - No communication barriers. ✓
  - Uses the *most recent* values. ✓
  - Implemented via distributed locking
  - Consider vertex-centric program

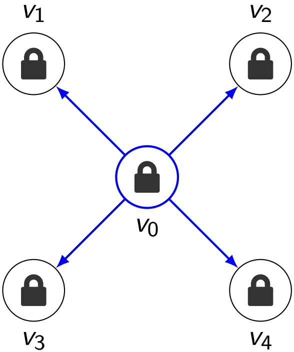




Computational Models

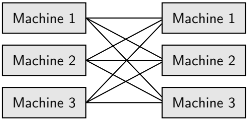
- Block Synchronous Parallel (BSP) [Valiant, 1990]
- Asynchronous Parallel
    - No communication barriers. ✓
    - Uses the *most recent* values. ✓
    - Implemented via distributed locking
    - Consider vertex-centric program

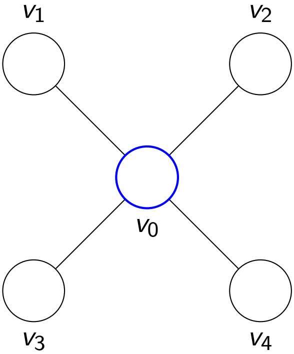




Computational Models

- Block Synchronous Parallel (BSP) [Valiant, 1990]
- Asynchronous Parallel
  - No communication barriers. ✓
  - Uses the *most recent* values. ✓
  - Implemented via distributed locking
  - Consider vertex-centric program





Computational Models

- Block Synchronous Parallel (BSP) [Valiant, 1990]
- Asynchronous Parallel
- Gather-Apply-Scatter (GAS)
  - Similar to BSP, but pull-based
  - Gather: pull state
  - Apply: `Compute` function
  - Scatter: Update state
  - Updates of states separated from scheduling

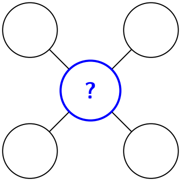
Real-World Graph Characteristics

- Read-world graphs have skewed vertex degree distribution
  - Common in power-law graphs
  - Problem: imbalanced communication workloads
- Real-world graphs have large diameters
  - Common in road networks, web graphs, terrain meshes
  - Problem: one superstep per hop $\Rightarrow$ too many supersteps
- Real-world graphs have high average vertex degree
  - Common in social networks, mobile communication networks
  - Problem: heavy average communication workloads

Outline

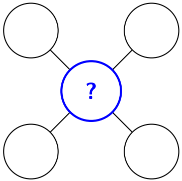
Vertex-Centric BSP Systems

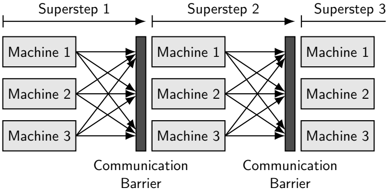
- "Think like a vertex"
- `Compute(vertex v)`



Vertex-Centric BSP Systems

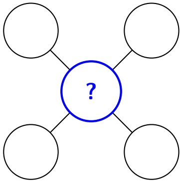
- "Think like a vertex"
- `Compute(vertex v)`
- BSP Computation – push state to neighbor vertices at the end of each superstep

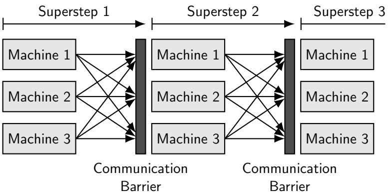


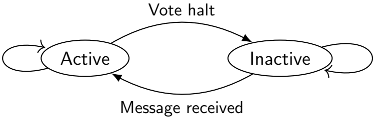


Vertex-Centric BSP Systems

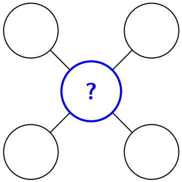
- "Think like a vertex"
- `Compute(vertex v)`
- BSP Computation – push state to neighbor vertices at the end of each superstep
- Continue until all vertices are inactive
- Vertex state machine

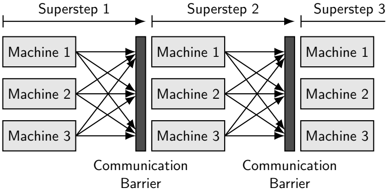


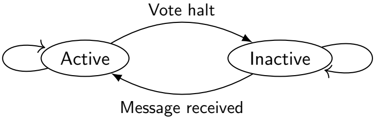




- "Think like a vertex"
- `Compute(vertex v)`
- BSP Computation – push state to neighbor vertices at the end of each superstep
- Continue until all vertices are inactive
- Vertex state machine
- Example systems: Pregel [Malewicz et al., 2010], Apache Giraph, GPS [Salihoglu and Widom, 2013], Mizan [Khayyat et al., 2013], Trinity [?]

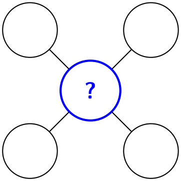






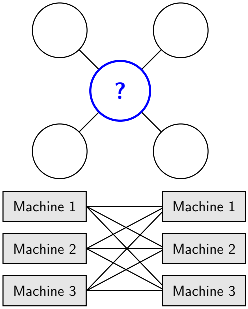
Vertex-Centric Asynchronous Systems

- "Think like a vertex"
- `Compute(vertex v)`



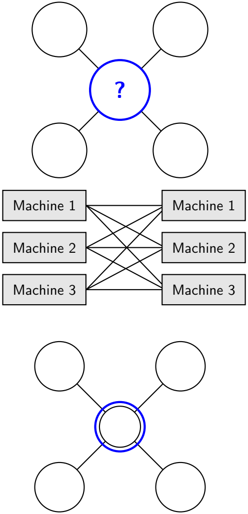
Vertex-Centric Asynchronous Systems

- “Think like a vertex”
- `Compute(vertex v)`
- Supersteps exist along with synchronization barriers, but ...
- `Compute(vertex v)` function can see messages it was sent in the same superstep as well as those that come at the end of the previous superstep



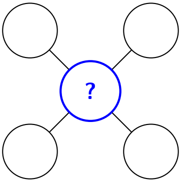
Vertex-Centric Asynchronous Systems

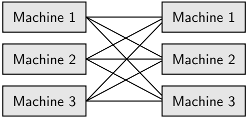
- "Think like a vertex"
- `Compute(vertex v)`
- Supersteps exist along with synchronization barriers, but ...
- `Compute(vertex v)` function can see messages it was sent in the same superstep as well as those that come at the end of the previous superstep
- Consistency of vertex states: distributed locking

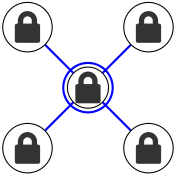


Vertex-Centric Asynchronous Systems

- “Think like a vertex”
- `Compute(vertex v)`
- Supersteps exist along with synchronization barriers, but ...
- `Compute(vertex v)` function can see messages it was sent in the same superstep as well as those that come at the end of the previous superstep
- Consistency of vertex states: distributed locking

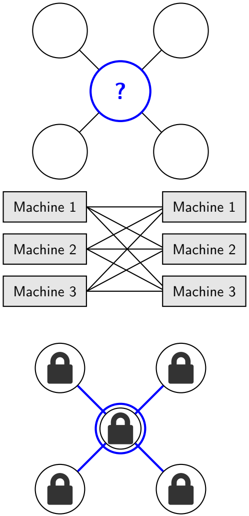






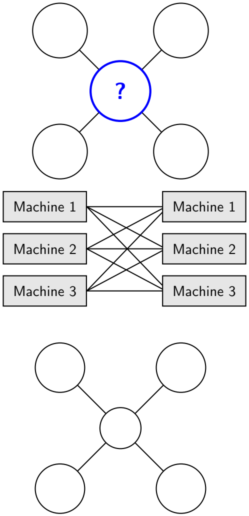
Vertex-Centric Asynchronous Systems

- "Think like a vertex"
- `Compute(vertex v)`
- Supersteps exist along with synchronization barriers, but ...
- `Compute(vertex v)` function can see messages it was sent in the same superstep as well as those that come at the end of the previous superstep
- Consistency of vertex states: distributed locking



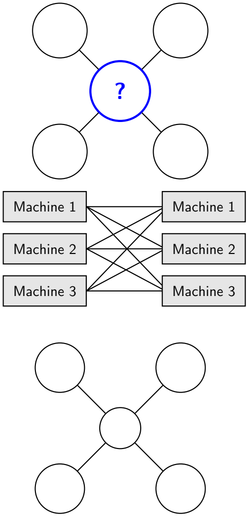
Vertex-Centric Asynchronous Systems

- "Think like a vertex"
- `Compute(vertex v)`
- Supersteps exist along with synchronization barriers, but ...
- `Compute(vertex v)` function can see messages it was sent in the same superstep as well as those that come at the end of the previous superstep
- Consistency of vertex states: distributed locking



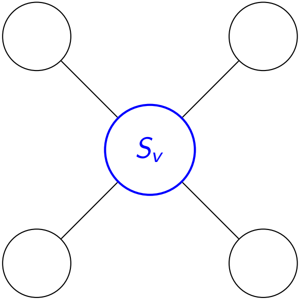
Vertex-Centric Asynchronous Systems

- "Think like a vertex"
- `Compute(vertex v)`
- Supersteps exist along with synchronization barriers, but ...
- `Compute(vertex v)` function can see messages it was sent in the same superstep as well as those that come at the end of the previous superstep
- Consistency of vertex states: distributed locking
- Consistency issues: no guarantee about input to `Compute()`
- Example systems: GRACE [Wang et al., 2013], GiraphCU [Han and Daudjee, 2015]



Vertex-Centric GAS Systems

- "Think like a vertex"
- Gather phase
  - Gather local computation from neighbours if vertex v: called *scope* S_v
- Apply phase
  - `Compute(v, Sv)`
- Scatter phase
  - `Compute(v, Sv)` produces $S_v^{'}$ (scattering state)
- Example: GraphLab [Low et al., 2012]
- Synchronous version
  - Similar to vertex-centric BSP, except pulling S_v rather than pushing
- Asynchronous version different

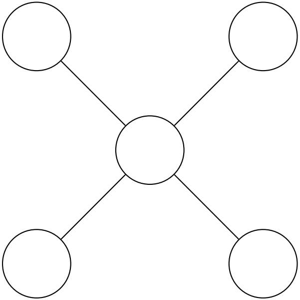


Vertex-Centric GAS Systems – Asynchronous

procedure $\textsc{GraphLab\_Async}(G = (V, E, D), \mathcal{T})$

return Modified $G = (V, E, D')$
end procedure

- Graph mutation restricted to vertex states

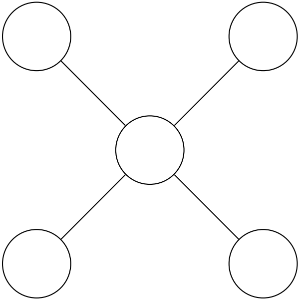


Vertex-Centric GAS Systems – Asynchronous

procedure $\textsc{GraphLab\_Async}(G = (V, E, D), \mathcal{T})$
while $\mathcal{T}$ is not empty **do**

end while
return Modified $G = (V, E, D')$
end procedure

- Graph mutation restricted to vertex states

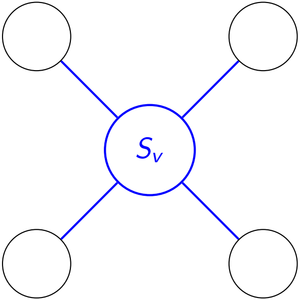


Vertex-Centric GAS Systems – Asynchronous

procedure $\textsc{GraphLab\_Async}(G = (V, E, D), \mathcal{T})$
while $\mathcal{T}$ is not empty **do**
    $v \leftarrow \texttt{RemoveNext}(\mathcal{T})$

end while
return Modified $G = (V, E, D^{'})$
end procedure

- Graph mutation restricted to vertex states

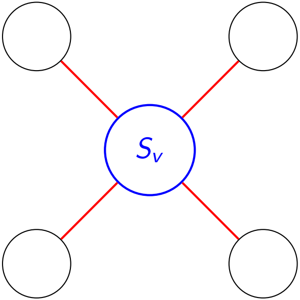


Vertex-Centric GAS Systems – Asynchronous

procedure $\textsc{GraphLab\_Async}(G = (V, E, D), \mathcal{T})$
while $\mathcal{T}$ is not empty **do**
  $v \leftarrow \texttt{RemoveNext}(\mathcal{T})$
  $\texttt{Compute}(v, S_v) \rightarrow (\mathcal{T}', S_v')$

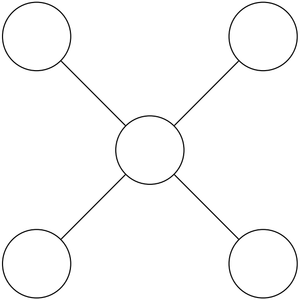
end while
return Modified $G = (V, E, D')$
end procedure

- Graph mutation restricted to vertex states
- Computing S_v' updates (scatters) states of the vertices in scope



Vertex-Centric GAS Systems – Asynchronous

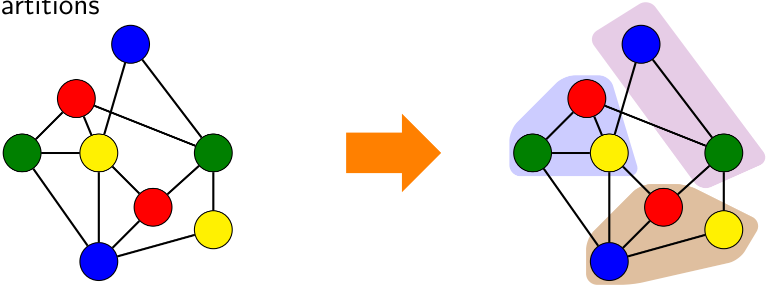
procedure GraphLab_Async($G = (V, E, D), \mathcal{T}$)
while $\mathcal{T}$ is not empty **do**
    $v \leftarrow$ `RemoveNext`$(\mathcal{T})$
    `Compute`$(v, S_v) \rightarrow (\mathcal{T}', S'_v)$
    $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$
end while
return Modified $G = (V, E, D')$
end procedure



- Graph mutation restricted to vertex states
- Computing S'_v updates (scatters) states of the vertices in scope
- Computation of new states S'_v separated from computation of new $\mathcal{T}'$
    - `RemoveNext()` can remove any vertex $\rightarrow$ scheduling separated from state scatter

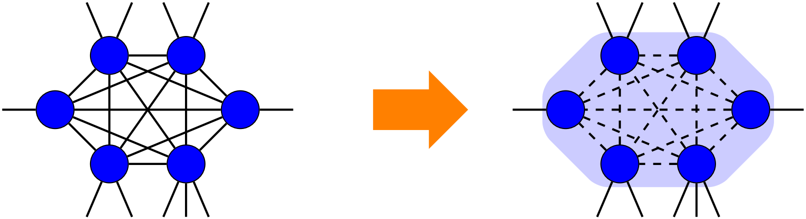
Partition- (Block-)Centric BSP Systems

- Blogel [Yan et al., 2014]: "Think like a block"; also "think like a graph" [Tian et al., 2013]
- Better handles the characteristics of real-world graphs by reducing communication
- Exploit the partitioning of the graph
- Message exchanges only among blocks
- Within a block, run a serial in-memory algorithm; BSP between partitions



Benefits of Partition- (Block-)Centric BSP

- High-degree vertices inside a block send no messages
- Fewer number of supersteps
- Fewer number of blocks than vertices



Edge-Centric BSP Systems

- "Think like an edge"
- `Compute(edge e)`
- Number of edges $\gg$ number of vertices
  - More computation but perhaps fewer messages
  - Operate on unsorted sequence of edges $\Rightarrow$ no random access
- X-Stream [Roy et al., 2013]

Outline

1. Introduction – Graph Types
2. Property Graph Processing
   - Classification
   - Online querying
   - Offline analytics
3. Graph Analytics Approaches
   - MapReduce & Variants
   - Classification of Native Approaches
4. Graph Analytics Systems
5. OLAP-Style Analytics
   - Graph Summarization
   - Snapshot-based Aggregation
   - Graph Cube
   - Pagrol
   - Gagg Model

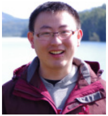
OLAP Over Graphs

- OLAP in RDBMS
  - Usage: Data Warehousing + Business Intelligence
  - Model: Multidimensional cube
  - Operations: Roll-up, drill-down, and slice and dice
- Analytics that we discussed over graphs is much different
- Can we do OLAP-style analytics over graphs?
  - There is some work
    - Graph summarization [Tian et al., 2008]
    - Snapshot-based Aggregation [Chen et al., 2008]
    - Graph Cube [Zhao et al., 2011]
    - Pagrol [?]
    - Gagg Model [Maali et al., 2015]

Acknowledgements

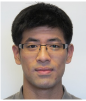
This presentation draws upon collaborative research and discussions with the following colleagues

Khaled Ammar, U. Waterloo



Xiaofei Zhang, U. Waterloo (U. Memphis)

Khuzaima Daudjee,U. Waterloo



Young Han, U. Waterloo (Google)

Thank you!

NSERC CRSNG

INNOVATION.CA
CANADA FOUNDATION FOR INNOVATION | FONDATION CANADIENNE POUR L'INNOVATION

Ontario
MINISTRY OF RESEARCH AND INNOVATION
MINISTÈRE DE LA RECHERCHE ET DE L'INNOVATION

HUAWEI

Google

BOSCH

IBM Canada Lab
CAS

References I

Bu, Y., Howe, B., Balazinska, M., and Ernst, M. D. (2010). HaLoop: efficient iterative data processing on large clusters. *Proc. VLDB Endowment*, 3(1):285–296. Available from: http://dl.acm.org/citation.cfm?id=1920841.1920881.

Bu, Y., Howe, B., Balazinska, M., and Ernst, M. D. (2012). The HaLoop approach to large-scale iterative data analysis. *VLDB J.*, 21(2):169–190.

Chen, C., Yan, X., Zhu, F., Han, J., and Yu, P. S. (2008). Graph OLAP: Towards online analytical processing on graphs. In *Proc. 8th IEEE Int. Conf. on Data Mining*, pages 103–112.

Chen, R., Shi, J., Chen, Y., and Chen, H. (2015). PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proc. 10th ACM SIGOPS/EuroSys European Conf. on Comp. Syst.*, pages 1:1–1:15. Available from: http://doi.acm.org/10.1145/2741948.2741970.

Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. (2012). PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. 10th USENIX Symp. on Operating System Design and Implementation*, pages 17–30. Available from: http://dl.acm.org/citation.cfm?id=2387880.2387883.

References II

Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I. (2014). GraphX: graph processing in a distributed dataflow framework graph processing in a distributed dataflow framework. In *Proc. 11th USENIX Symp. on Operating System Design and Implementation*, pages 599–613. Available from: `https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez`.

Han, M. (2015). On improving distributed Pregel-like graph processing systems. Master's thesis, University of Waterloo, David R. Cheriton School of Computer Science.

Han, M. and Daudjee, K. (2015). Giraph unchained: Barrierless asynchronous parallel execution in Pregel-like graph processing systems. *Proc. VLDB Endowment*, 8(9):950–961. Available from: `http://www.vldb.org/pvldb/vol8/p950-han.pdf`.

Karypis, G. and Kumar, V. (1995). Multilevel graph partitioning schemes. In *Proc. 1995 Int. Conf. on Parallel Processing*, pages 113–122.

Khayyat, Z., Awara, K., Alonazi, A., Jamjoom, H., Williams, D., and Kalnis, P. (2013). Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proc. 8th ACM SIGOPS/EuroSys European Conf. on Comp. Syst.*, pages 169–182. Available from: `http://doi.acm.org/10.1145/2465351.2465369`.

References III

Kiveris, R., Lattanzi, S., Mirrokni, V., Rastogi, V., and Vassilvitskii, S. (2014). Connected components in MapReduce and beyond. In *Proc. 5th ACM Symp. on Cloud Computing*, pages 18:1–18:13.

Lin, J. (2018). Scale up or scale out for graph processing? *IEEE Internet Comput.*, 22(3):72–78.

Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. (2012). Distributed graphlab: A framework for machine learning in the cloud. *Proc. VLDB Endowment*, 5(8):716–727.

Maali, F., Campinas, S., and Decker, S. (2015). Gagg: A graph aggregation operator. In *Proc. 14th Int. Semantic Web Conf.*, pages 491–504.

Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 135–146.

Qin, L., Yu, J. X., Chang, L., Cheng, H., Zhang, C., and Lin, X. (2014). Scalable big graph processing in mapreduce. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 827–838. Available from: `http://doi.acm.org/10.1145/2588555.2593661`.

References IV

Rastogi, V., Machanavajjhala, A., Chitnis, L., and Sarma, A. D. (2013). Finding connected components in map-reduce in logarithmic rounds. In *Proc. 29th Int. Conf. on Data Engineering*, pages 50–61.

Roy, A., Mihailovic, I., and Zwaenepoel, W. (2013). X-stream: edge-centric graph processing using streaming partitions. In *Proc. 24th ACM Symp. on Operating System Principles*, pages 472–488. Available from: `http://doi.acm.org/10.1145/2517349.2522740`.

Salihoglu, S. and Widom, J. (2013). GPS: a graph processing system. In *Proc. 25th Int. Conf. on Scientific and Statistical Database Management*, pages 22:1–22:12. Available from: `http://doi.acm.org/10.1145/2484838.2484843`.

Tian, Y., Balmin, A., Corsten, S. A., Tatikonda, S., and McPherson, J. (2013). From "think like a vertex" to "think like a graph". *Proc. VLDB Endowment*, 7(3):193–204. Available from: `http://www.vldb.org/pvldb/vol7/p193-tian.pdf`.

Tian, Y., Hankins, R. A., and Patel, J. M. (2008). Efficient aggregation for graph summarization. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 567–580.

References V

Ugander, J. and Backstrom, L. (2013). Balanced label propagation for partitioning massive graphs. In *Proc. 6th ACM Int. Conf. Web Search and Data Mining*, pages 507–516. Available from: `http://doi.acm.org/10.1145/2433396.2433461`.

Valiant, L. G. (1990). A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111.

Wang, G., Xie, W., Demers, A. J., and Gehrke, J. (2013). Asynchronous large-scale graph processing made easy. In *Proc. 6th Biennial Conf. on Innovative Data Systems Research*.

Yan, D., Cheng, J., Lu, Y., and Ng, W. (2014). Blogel: A block-centric framework for distributed computation on real-world graphs. *Proc. VLDB Endowment*, 7(14):1981–1992. Available from: `http://www.vldb.org/pvldb/vol7/p1981-yan.pdf`.

Zhao, P., Li, X., Xin, D., and Han, J. (2011). Graph cube: on warehousing and OLAP multidimensional networks. In *Proc. ACM International Conference on Management of Data*, pages 853–864.