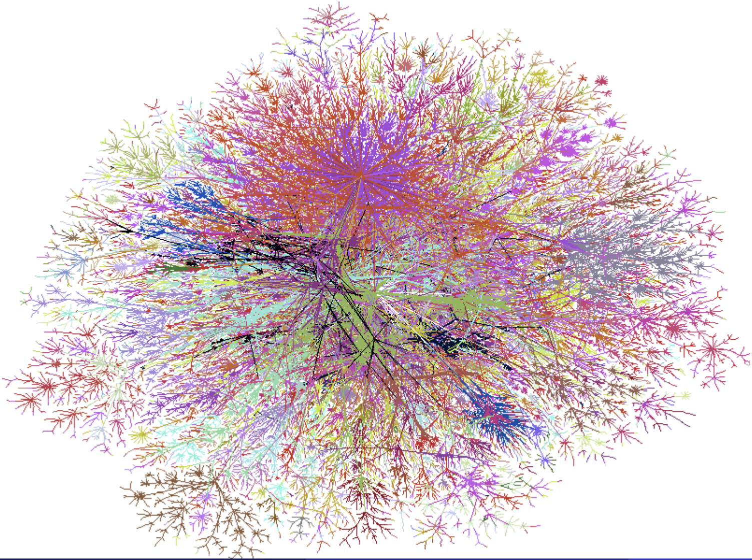
An Introduction to Graph Analytics Platforms

M. Tamer Özsu

University of Waterloo
David R. Cheriton School of Computer Science

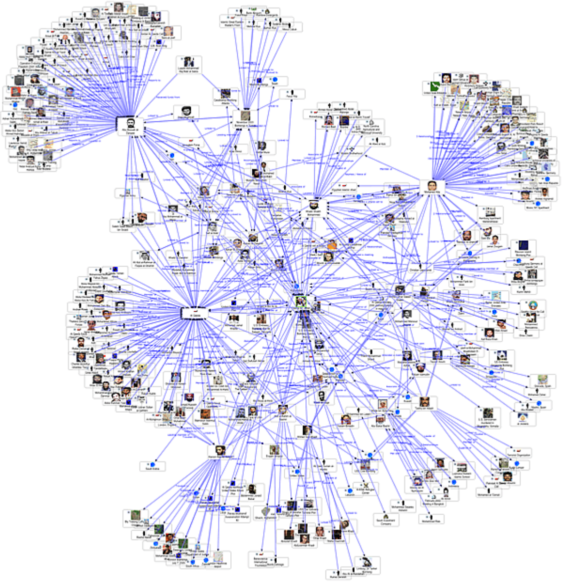
UNIVERSITY OF
Waterloo

Graph Data are Very Common



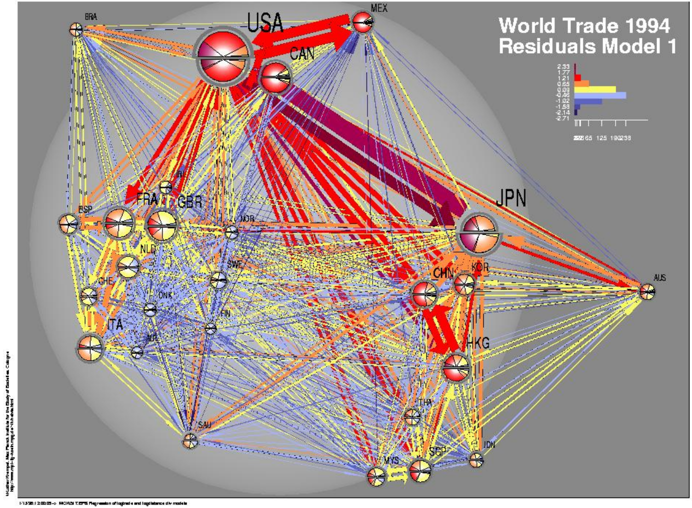
Internet

Graph Data are Very Common



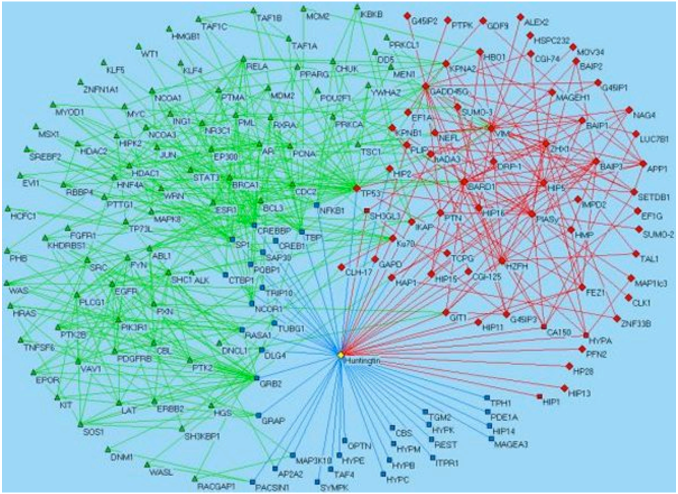
Social networks

Graph Data are Very Common



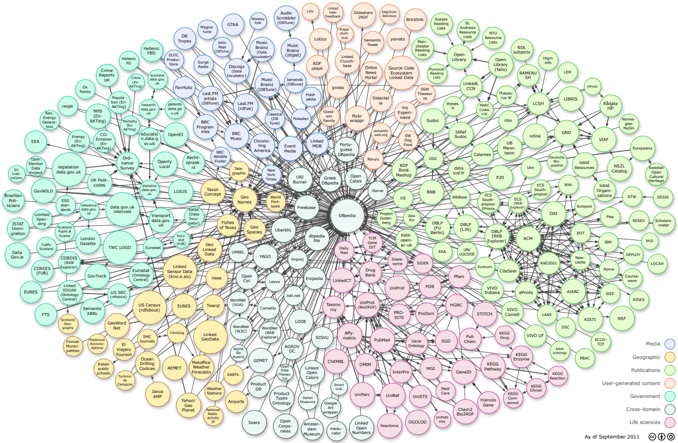
Trade volumes and connections

Graph Data are Very Common



Biological networks

Graph Data are Very Common



Linked data

Linking Open Data cloud diagram, by Richard Cyganiak and Anja Jentzsch.
http://lod-cloud.net/

Outline

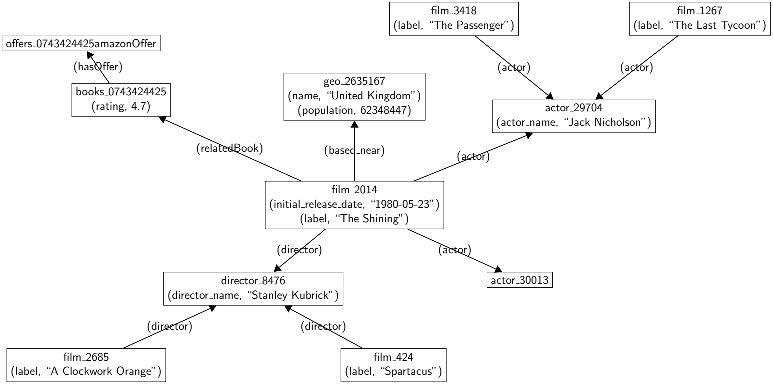
1. Introduction – Graph Types
2. Property Graph Processing
   - Classification
   - Online querying
   - Offline analytics
3. Graph Analytics Computational Models
   - Vertex-Centric
   - Block-Centric
   - MapReduce-Based
   - Modified MapReduce

Outline

1. Introduction – Graph Types
2. Property Graph Processing
   - Classification
   - Online querying
   - Offline analytics
3. Graph Analytics Computational Models
   - Vertex-Centric
   - Block-Centric
   - MapReduce-Based
   - Modified MapReduce

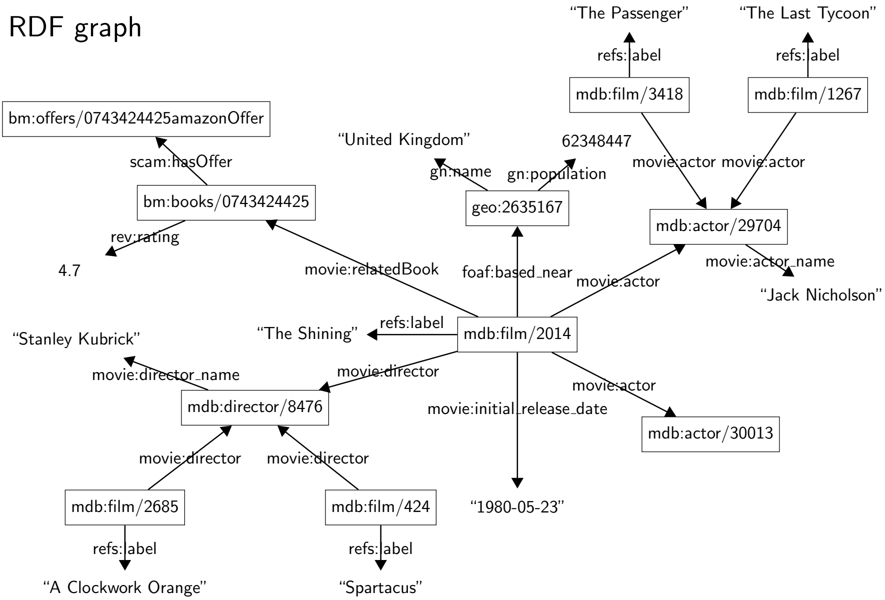
Graph Types

Property graph



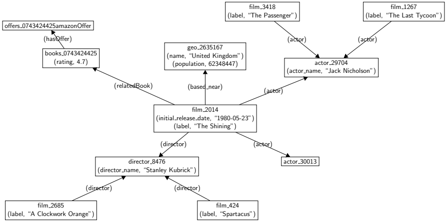
Graph Types

RDF graph



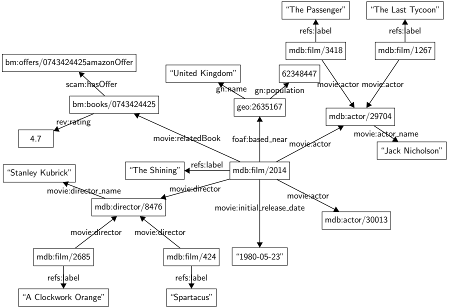
Graph Types

Property graph



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

RDF graph



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

RDF Introduction

http://data.linkedmdb.org/resource/actor/JN29704

- Everything is an uniquely named resource

RDF Introduction

xmlns:y=http://data.linkedmdb.org/resource/actor/
y:JN29704

- Everything is an uniquely named resource
- Prefixes can be used to shorten the names

RDF Introduction

xmlns:y=http://data.linkedmdb.org/resource/actor/

y:JN29704

y:JN29704:hasName "Jack Nicholson"
y:JN29704:BornOnDate "1937-04-22"

- Everything is an uniquely named resource
- Prefixes can be used to shorten the names
- Properties of resources can be defined

RDF Introduction

- Everything is an uniquely named resource
- Prefixes can be used to shorten the names
- Properties of resources can be defined
- Relationships with other resources can be defined

xmlns:y=http://data.linkedmdb.org/resource/actor/

y:JN29704

y:JN29704:hasName "Jack Nicholson"
y:JN29704:BornOnDate "1937-04-22"

JN29704:movieActor

y:TS2014

y:TS2014:title "The Shining"
y:TS2014:releaseDate "1980-05-23"

RDF Introduction

- Everything is an uniquely named resource
- Prefixes can be used to shorten the names
- Properties of resources can be defined
- Relationships with other resources can be defined
- Resource descriptions can be contributed by different people/groups and can be located anywhere in the web
    - Integrated web "database"

xmlns:y=http://data.linkedmdb.org/resource/actor/

y:JN29704

y:JN29704:hasName "Jack Nicholson"
y:JN29704:BornOnDate "1937-04-22"

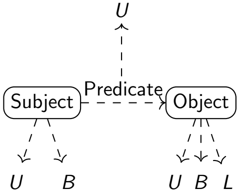
JN29704:movieActor

y:TS2014

y:TS2014:title "The Shining"
y:TS2014:releaseDate "1980-05-23"

RDF Data Model

- Triple: Subject, Predicate (Property), Object
  (s, p, o)
  - Subject: the entity that is described (URI or blank node)
  - Predicate: a feature of the entity (URI)
  - Object: value of the feature (URI, blank node or literal)
- $(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$
- Set of RDF triples is called an RDF graph



U: set of URIs
B: set of blank nodes
L: set of literals

Subject	**Predicate**	**Object**
http://...imdb.../film/2014	rdfs:label	"The Shining"
http://...imdb.../film/2014	movie:releaseDate	"1980-05-23"
http://...imdb.../29704	movie:actor_name	"Jack Nicholson"
...	...	...

RDF Example Instance

Prefixes: mdb=http://data.linkedmdb.org/resource/; geo=http://sws.geonames.org/
bm=http://wifo5-03.informatik.uni-mannheim.de/bookmashup/
lexvo=http://lexvo.org/id/;wp=http://en.wikipedia.org/wiki/

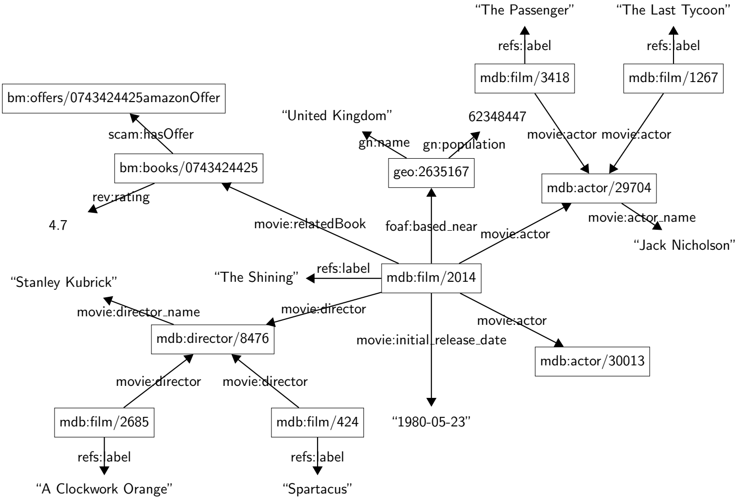
URI

Literal

URI

Subject	Predicate	Object
mdb: film/2014	rdfs:label	“The Shining”
mdb:film/2014	movie:initial_release_date	“1980-05-23”’
mdb:film/2014	movie:director	mdb:director/8476
mdb:film/2014	movie:actor	mdb:actor/29704
mdb:film/2014	movie:actor	mdb: actor/30013
mdb:film/2014	movie:music_contributor	mdb: music_contributor/4110
mdb:film/2014	foaf:based_near	geo:2635167
mdb:film/2014	movie:relatedBook	bm:0743424425
mdb:film/2014	movie:language	lexvo:iso639-3/eng
mdb:director/8476	movie:director_name	“Stanley Kubrick”
mdb:film/2685	movie:director	mdb:director/8476
mdb:film/2685	rdfs:label	“A Clockwork Orange”
mdb:film/424	movie:director	mdb:director/8476
mdb:film/424	rdfs:label	“Spartacus”
mdb:actor/29704	movie:actor_name	“Jack Nicholson”
mdb:film/1267	movie:actor	mdb:actor/29704
mdb:film/1267	rdfs:label	“The Last Tycoon”
mdb:film/3418	movie:actor	mdb:actor/29704
mdb:film/3418	rdfs:label	“The Passenger”
geo:2635167	gn:name	“United Kingdom”
geo:2635167	gn:population	62348447
geo:2635167	gn:wikipediaArticle	wp:United_Kingdom
bm:books/0743424425	dc:creator	bm:persons/Stephen+King
bm:books/0743424425	rev:rating	4.7
bm:books/0743424425	scom:hasOffer	bm:offers/0743424425amazonOffer
lexvo:iso639-3/eng	rdfs:label	“English”
lexvo:iso639-3/eng	lvont:usedIn	lexvo:iso3166/CA
lexvo:iso639-3/eng	lvont:usesScript	lexvo:script/Latn

RDF Graph


"The Passenger"
"The Last Tycoon"
refs:label
refs:label
mdb:film/3418
mdb:film/1267
bm:offers/0743424425amazonOffer
"United Kingdom"
62348447
scam:hasOffer
movie:actor
movie:actor
gn:name
gn:population
bm:books/0743424425
geo:2635167
mdb:actor/29704
rev:rating
movie:actor_name
4.7
movie:relatedBook
foaf:based_near
movie:actor
"Jack Nicholson"
refs:label
"Stanley Kubrick"
"The Shining"
mdb:film/2014
movie:director_name
movie:director
movie:actor
mdb:director/8476
movie:initial_release_date
mdb:actor/30013
movie:director
movie:director
mdb:film/2685
mdb:film/424
"1980-05-23"
refs:label
refs:label
"A Clockwork Orange"
"Spartacus"

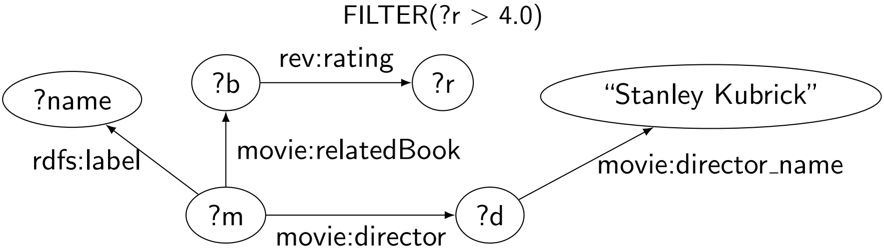
RDF Query Model – SPARQL

- Query Model - **S**PARQL **P**rotocol **a**nd **R**DF **Q**uery **L**anguage
- Given U (set of URIs), L (set of literals), and V (set of variables), a SPARQL expression is defined recursively:
  - an atomic triple pattern, which is an element of
  $$(U \cup V) \times (U \cup V) \times (U \cup V \cup L)$$
    - ?x rdfs:label "The Shining"
  - P FILTER R, where P is a graph pattern expression and R is a built-in SPARQL condition (i.e., analogous to a SQL predicate)
    - ?x rev:rating ?p FILTER(?p $>$ 3.0)
  - $P1$ AND/OPT/UNION $P2$, where $P1$ and $P2$ are graph pattern expressions
- Example:

```
SELECT ?name
WHERE {
  ?m rdfs:label ?name. ?m movie:director ?d.
  ?d movie:director_name "Stanley Kubrick".
  ?m movie:relatedBook ?b. ?b rev:rating ?r.
  FILTER(?r > 4.0)
}
```

SPARQL Queries

```
SELECT ?name
WHERE {
  ?m rdfs:label ?name. ?m movie:director ?d.
  ?d movie:director_name "Stanley Kubrick".
  ?m movie:relatedBook ?b. ?b rev:rating ?r.
  FILTER(?r > 4.0)
}
```



Outline

1. Introduction – Graph Types

2. Property Graph Processing
   - Classification
   - Online querying
   - Offline analytics

3. Graph Analytics Computational Models
   - Vertex-Centric
   - Block-Centric
   - MapReduce-Based
   - Modified MapReduce

Outline

1. Introduction – Graph Types
2. Property Graph Processing
   - Classification
   - Online querying
   - Offline analytics
3. Graph Analytics Computational Models
   - Vertex-Centric
   - Block-Centric
   - MapReduce-Based
   - Modified MapReduce

Classification

- Graph Dynamism
- Algorithm Types
- Workload Types

Classification

[Ammar and Özsu, 2016]

Graph Dynamism

Algorithm Types

Workload Types

Focus here is on the dynamism of the graphs in whether or not they change and how they change.

Classification

[Ammar and Özsu, 2016]

Graph Dynamism

Algorithm Types

Workload Types

Focus here is on the dynamism of the graphs in whether or not they change and how they change.

Focus here is on how algorithms behave as their input changes.

Classification

[Ammar and Özsu, 2016]

Graph Dynamism

Focus here is on the dynamism of the graphs in whether or not they change and how they change.

Algorithm Types

Focus here is on how algorithms behave as their input changes.

Workload Types

The types of workloads that the approaches are designed to handle.

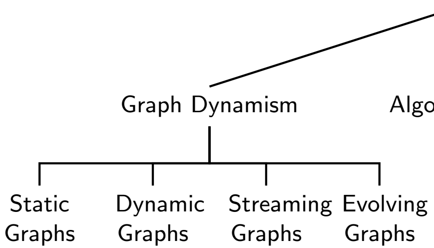
Classification

[Ammar and Özsu, 2016]

- Graph Dynamism
  - Static Graphs
  - Dynamic Graphs
  - Streaming Graphs
  - Evolving Graphs
- Algorithm Types
- Workload Types

Classification

[Ammar and Özsu, 2016]



Graphs do not change or we are not interested in their changes – only a *snapshot* is considered.

- Graph Dynamism
  - Static Graphs
  - Dynamic Graphs
  - Streaming Graphs
  - Evolving Graphs
- Algorithm Types
- Workload Types

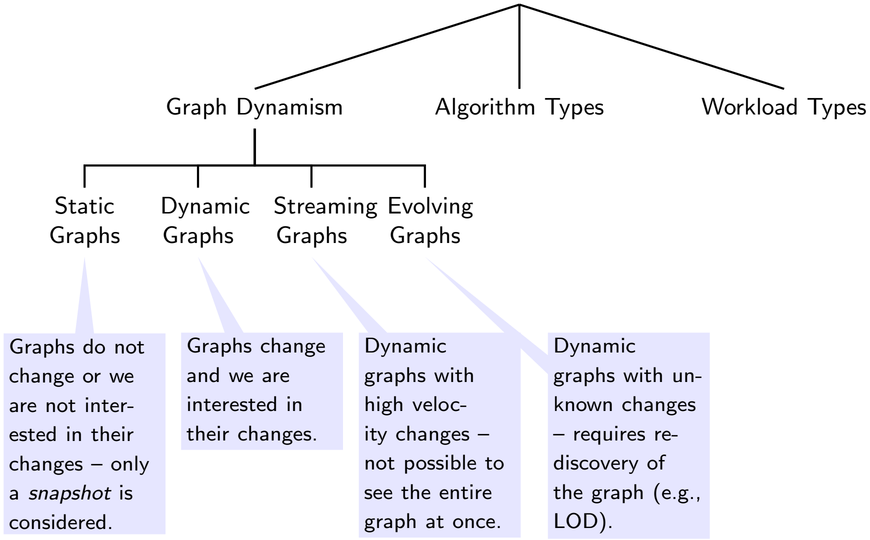
Static Graphs: Graphs do not change or we are not interested in their changes – only a *snapshot* is considered.

Dynamic Graphs: Graphs change and we are interested in their changes.

Classification

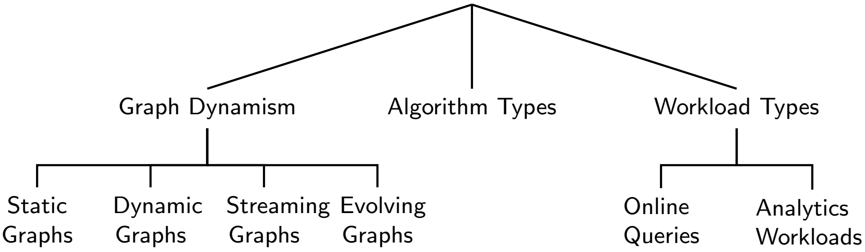
[Ammar and Özsu, 2016]

- Graph Dynamism
  - Static Graphs
    - Graphs do not change or we are not interested in their changes – only a *snapshot* is considered.
  - Dynamic Graphs
    - Graphs change and we are interested in their changes.
  - Streaming Graphs
    - Dynamic graphs with high velocity changes – not possible to see the entire graph at once.
  - Evolving Graphs
- Algorithm Types
- Workload Types


Graph Dynamism
Algorithm Types
Workload Types
Static Graphs
Dynamic Graphs
Streaming Graphs
Evolving Graphs
Graphs do not change or we are not interested in their changes – only a *snapshot* is considered.
Graphs change and we are interested in their changes.
Dynamic graphs with high velocity changes – not possible to see the entire graph at once.
Dynamic graphs with unknown changes – requires re-discovery of the graph (e.g., LOD).

Classification

[Ammar and Özsu, 2016]



- Graph Dynamism
  - Static Graphs
  - Dynamic Graphs
  - Streaming Graphs
  - Evolving Graphs
- Algorithm Types
- Workload Types
  - Online Queries
  - Analytics Workloads

Computation accesses a portion of the graph and the results are computed for a subset of vertices; e.g., point-to-point shortest path, subgraph matching, reachability, SPARQL.

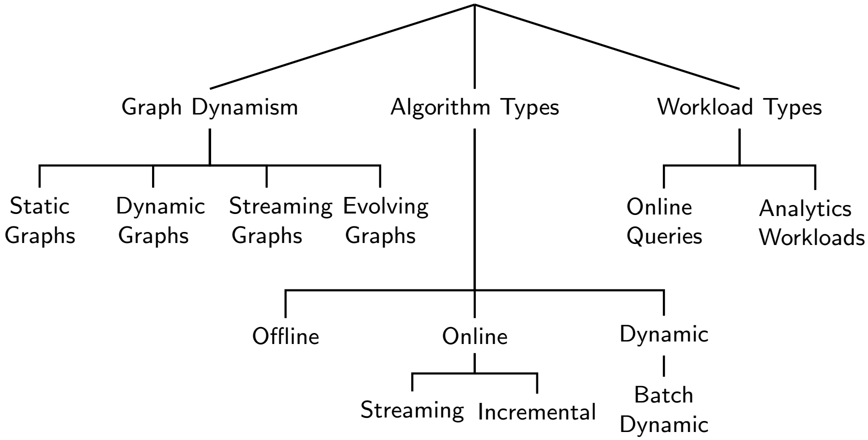
- Graph Dynamism
  - Static Graphs
  - Dynamic Graphs
  - Streaming Graphs
  - Evolving Graphs
- Algorithm Types
- Workload Types
  - Online Queries
  - Analytics Workloads

Online Queries: Computation accesses a portion of the graph and the results are computed for a subset of vertices; e.g., point-to-point shortest path, subgraph matching, reachability, SPARQL.

Analytics Workloads: Computation accesses the entire graph and may require multiple iterations; e.g., PageRank, clustering, graph colouring, all pairs shortest path.

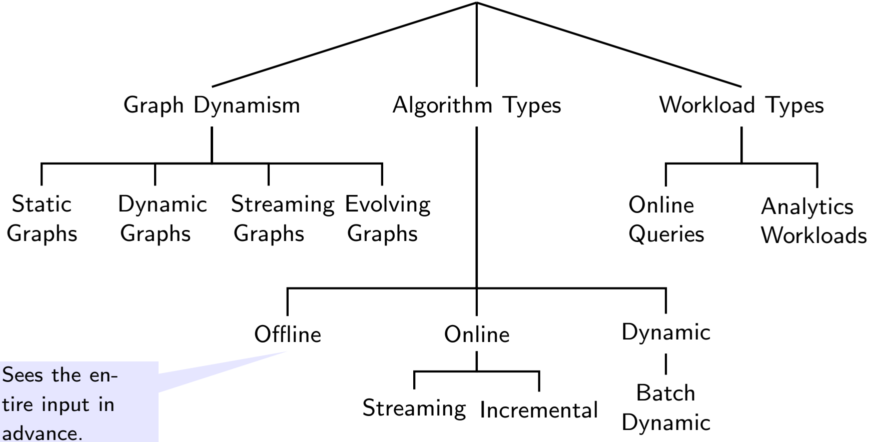
Classification

[Ammar and Özsu, 2016]



Classification

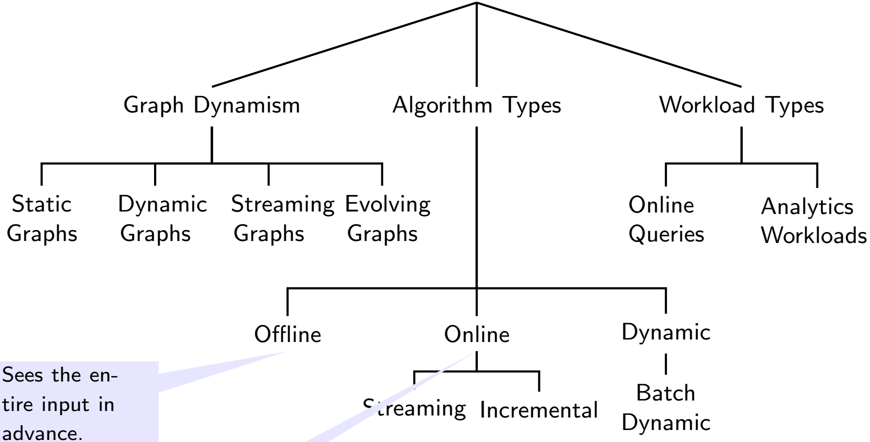
[Ammar and Özsu, 2016]



Sees the entire input in advance.

Classification

[Ammar and Özsu, 2016]

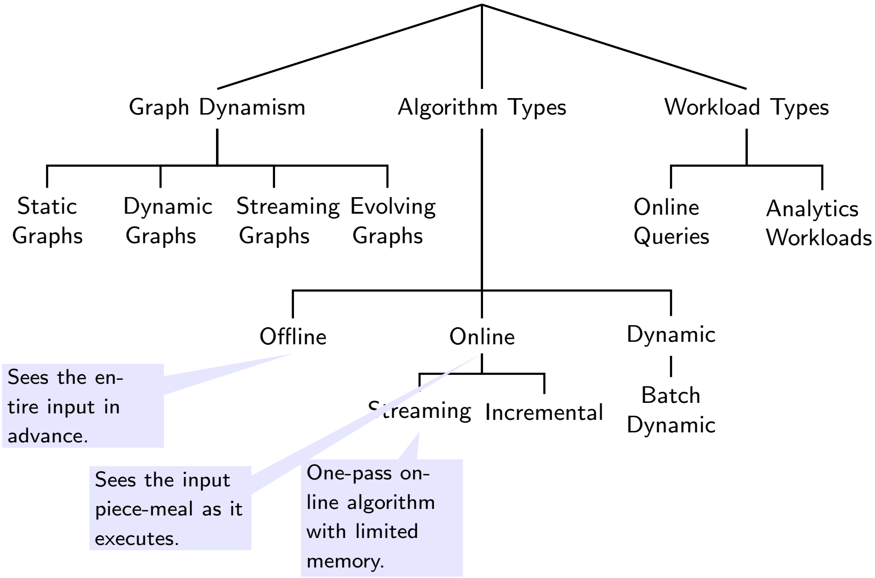


Sees the entire input in advance.

Sees the input piece-meal as it executes.

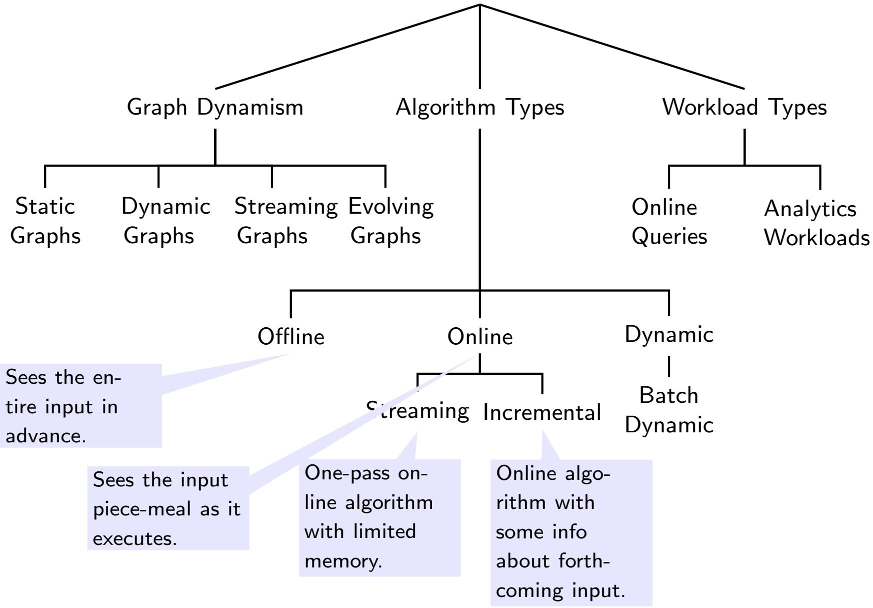
Classification

[Ammar and Özsu, 2016]



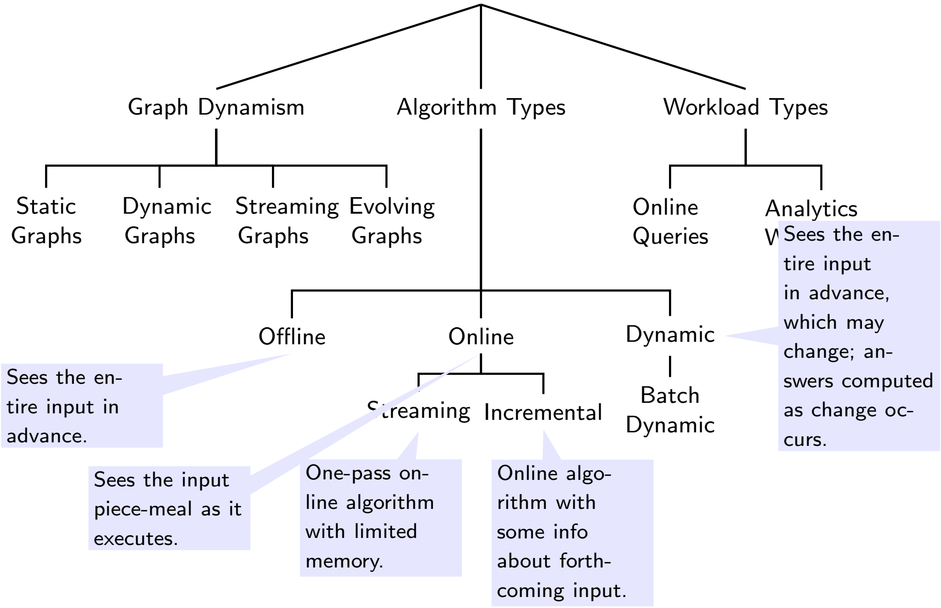
Classification

[Ammar and Özsu, 2016]



Classification

[Ammar and Özsu, 2016]

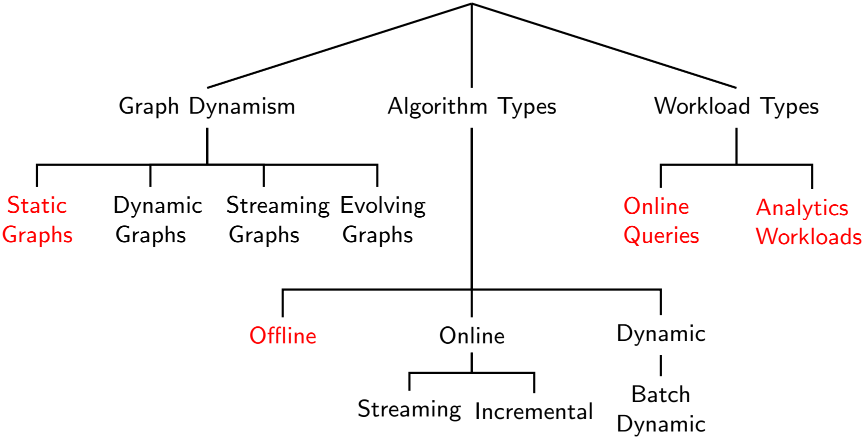


Classification

[Ammar and Özsu, 2016]

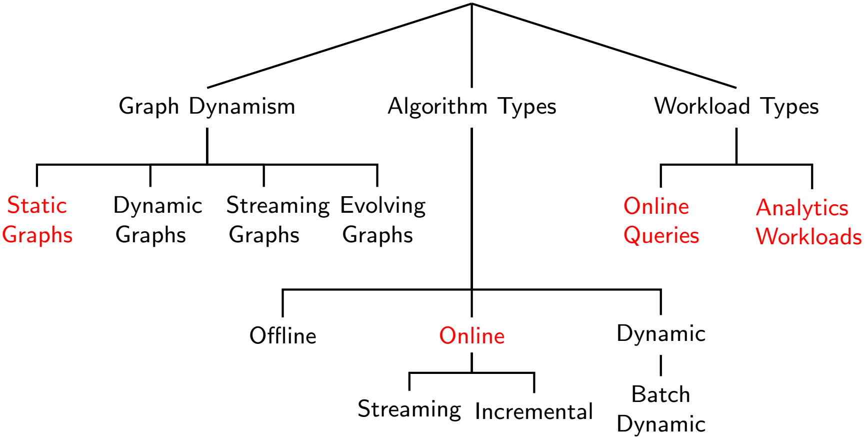
- Graph Dynamism
  - Static Graphs
  - Dynamic Graphs
  - Streaming Graphs
  - Evolving Graphs
- Algorithm Types
  - Offline
    - Sees the entire input in advance.
  - Online
    - Sees the input piece-meal as it executes.
    - Streaming
      - One-pass online algorithm with limited memory.
    - Incremental
      - Online algorithm with some info about forthcoming input.
  - Dynamic
    - Sees the entire input in advance, which may change; answers computed as change occurs.
    - Batch Dynamic
      - Similar to dynamic, but computation happens in batches of changes.
- Workload Types
  - Online Queries
  - Analytics W

Example Design Points



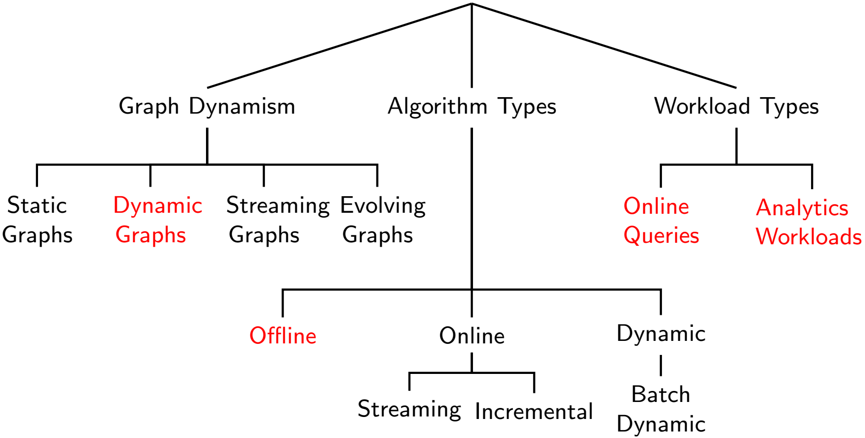
Compute the query result/perform analytic computation over the graph as it exists.

Example Design Points



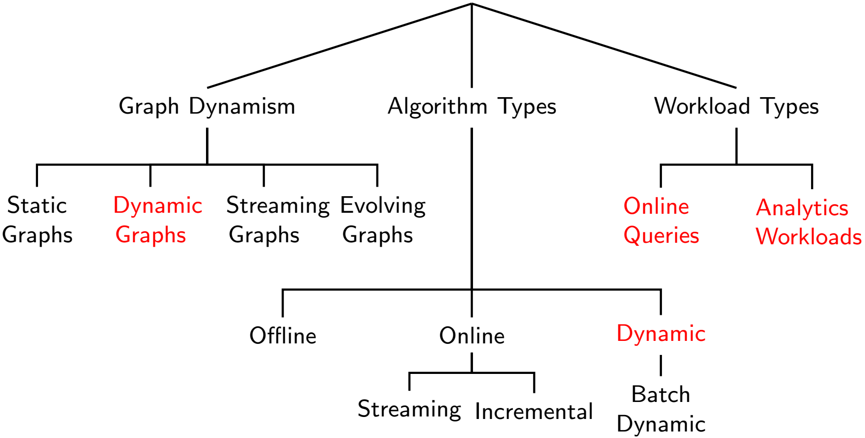
Compute the query result/perform analytic computation over the graph as it is revealed.

Example Design Points



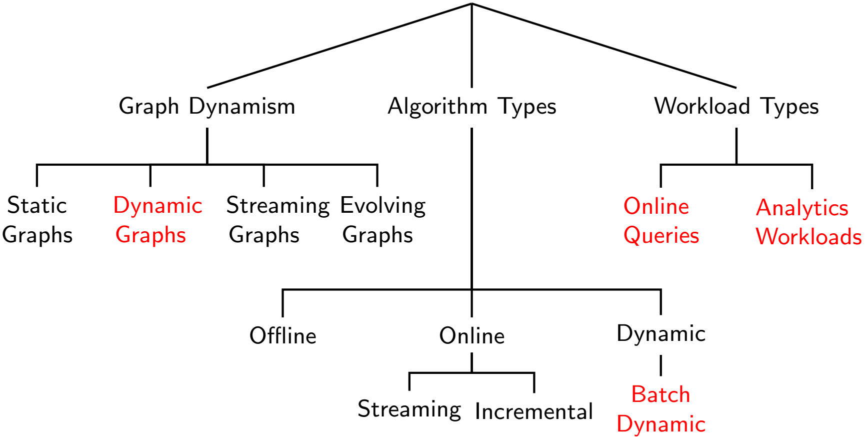
Compute the query result/perform analytic computation on each snapshot from scratch.

Example Design Points



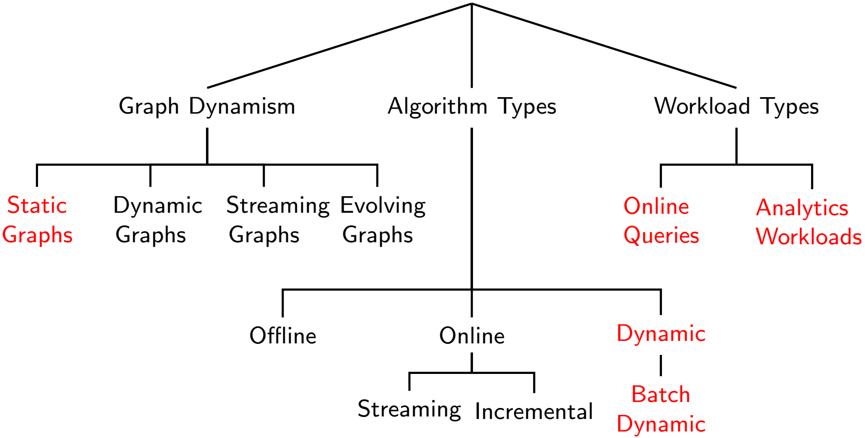
Continuously compute the query result/perform analytic computation as the input changes.

Example Design Points



Compute the query result/perform analytic computation after a batch of input changes.

Example Design Points – Not all alternatives make sense



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

Graph Processing Systems

System	Memory/ Disk	Architecture	Computing paradigm	Supported Workloads
Hadoop	Disk	Parallel/Distributed	MapReduce	Analytical
Haloop	Disk	Parallel/Distributed	MapReduce	Analytical
Pegasus	Disk	Parallel/Distributed	MapReduce	Analytical
GraphX	Disk	Parallel/Distributed	MapReduce (Spark)	Analytical
Pregel/Giraph	Memory	Parallel/Distributed	Vertex-Centric	Analytical
GraphLab	Memory	Parallel/Distributed	Vertex-Centric	Analytical
GraphChi	Disk	Single machine	Vertex-Centric	Analytical
Stream	Disk	Single machine	Edge-Centric	Analytical
Trinity	Memory	Parallel/Distributed	Flexible using K-V store on DSM	Online & Analytical
Titan	Disk	Parallel/Distributed	K-V store (Cassandra)	Online
Neo4J	Disk	Single machine	Procedural/ Linked-list	Online

Graph Workloads

Online graph querying

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

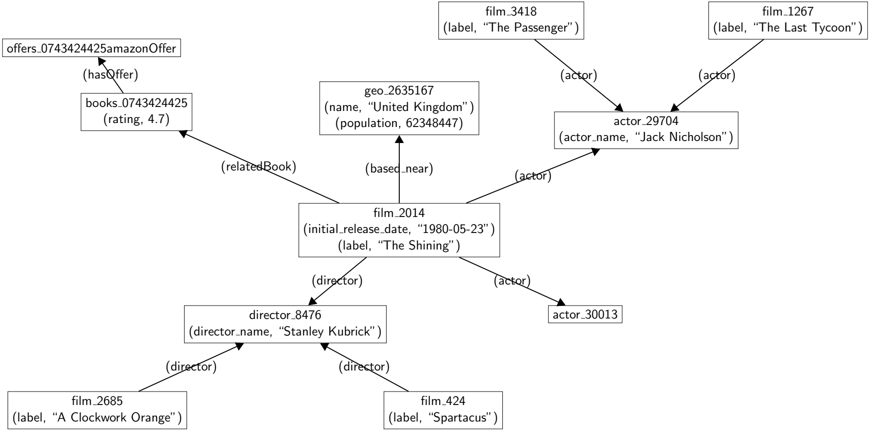
Offline graph analytics

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

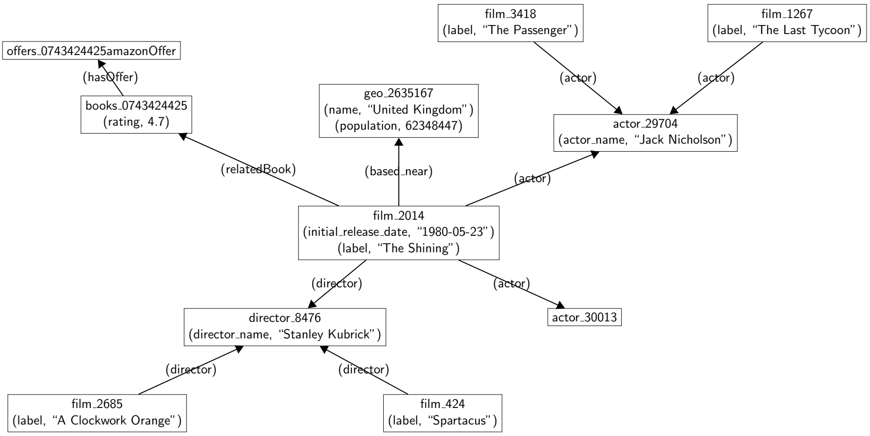
Outline

1. Introduction – Graph Types
2. Property Graph Processing
   - Classification
   - Online querying
   - Offline analytics
3. Graph Analytics Computational Models
   - Vertex-Centric
   - Block-Centric
   - MapReduce-Based
   - Modified MapReduce

Reachability Queries

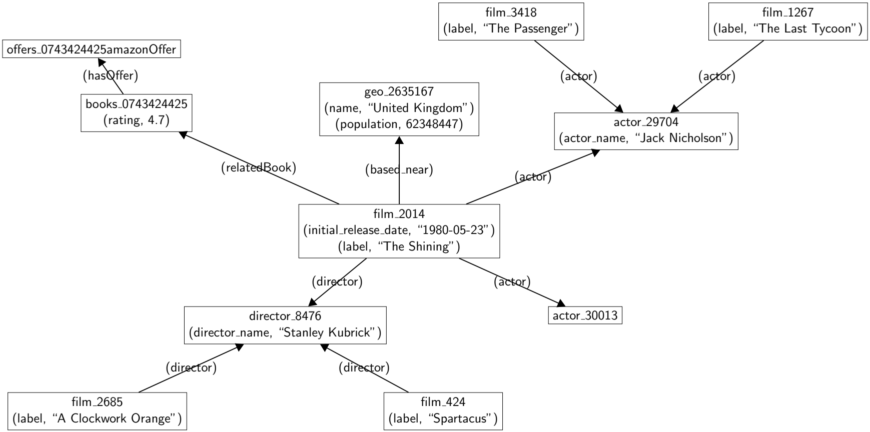


Reachability Queries



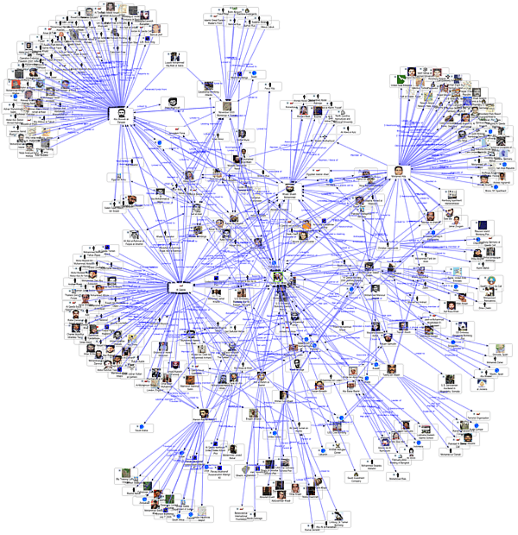
Can you reach film_1267 from film_2014?

Reachability Queries



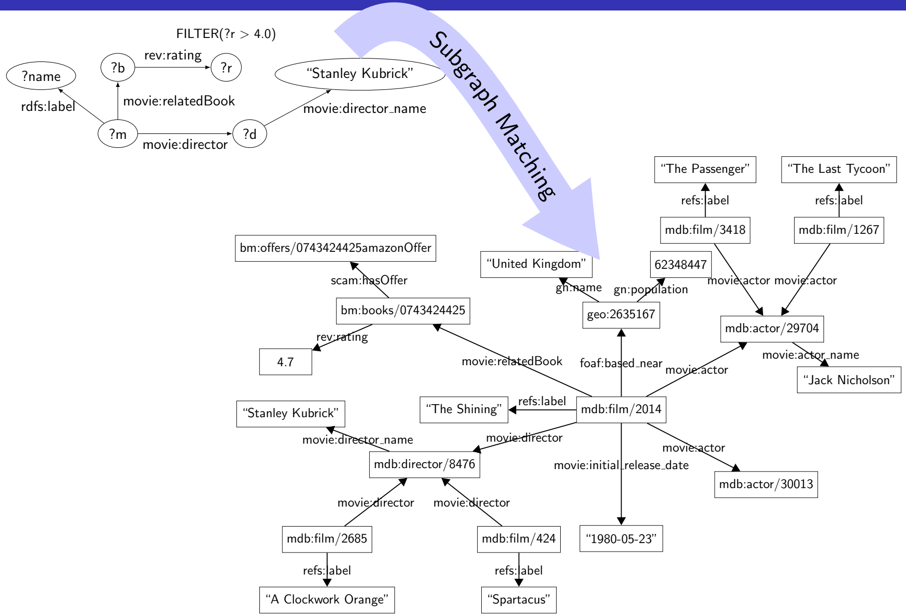
Is there a book whose rating is > 4.0 associated with a film that was directed by Stanley Kubrick?

Reachability Queries



Think of Facebook graph and finding friends of friends.

Subgraph Matching

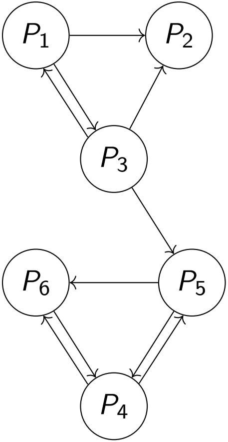


Outline

1. Introduction – Graph Types
2. Property Graph Processing
   - Classification
   - Online querying
   - Offline analytics
3. Graph Analytics Computational Models
   - Vertex-Centric
   - Block-Centric
   - MapReduce-Based
   - Modified MapReduce

PageRank Computation

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



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

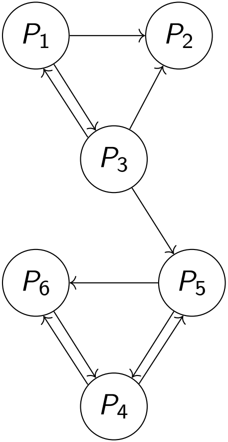
$$r(P_2) = \frac{r(P_1)}{2} + \frac{r(P_3)}{3}$$

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

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.

Outline

1. Introduction – Graph Types
2. Property Graph Processing
   - Classification
   - Online querying
   - Offline analytics
3. Graph Analytics Computational Models
   - Vertex-Centric
   - Block-Centric
   - MapReduce-Based
   - Modified MapReduce

Some Alternative Computational Models for Offline Analytics

- Vertex-centric (Scatter-Gather)
  - Specify (a) computation at each vertex, and (b) communication with neighbour vertices
  - Synchronous – Pregel [Malewicz et al., 2010], Giraph
  - Asynchronous – GraphLab [Low et al., 2012]

Some Alternative Computational Models for Offline Analytics

- Vertex-centric (Scatter-Gather)
  - Specify (a) computation at each vertex, and (b) communication with neighbour vertices
  - Synchronous – Pregel [Malewicz et al., 2010], Giraph
  - Asynchronous – GraphLab [Low et al., 2012]
- Block-centric
  - Similar to vertex-centric but on *blocks* for communication
    - Connected subgraph of the graph
  - Blogel [Yan et al., 2014]

Some Alternative Computational Models for Offline Analytics

- Vertex-centric (Scatter-Gather)
  - Specify (a) computation at each vertex, and (b) communication with neighbour vertices
  - Synchronous – Pregel [Malewicz et al., 2010], Giraph
  - Asynchronous – GraphLab [Low et al., 2012]
- Block-centric
  - Similar to vertex-centric but on *blocks* for communication
    - Connected subgraph of the graph
  - Blogel [Yan et al., 2014]
- MapReduce
  - Need to save in HDFS intermediate results of each iteration – both good and bad
  - Hadoop, Haloop [Bu et al., 2012]

Some Alternative Computational Models for Offline Analytics

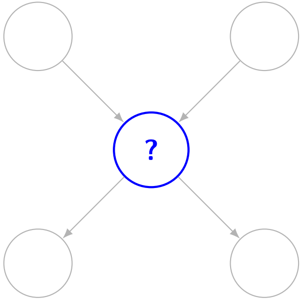
- Vertex-centric (Scatter-Gather)
  - Specify (a) computation at each vertex, and (b) communication with neighbour vertices
  - Synchronous – Pregel [Malewicz et al., 2010], Giraph
  - Asynchronous – GraphLab [Low et al., 2012]
- Block-centric
  - Similar to vertex-centric but on *blocks* for communication
    - Connected subgraph of the graph
  - Blogel [Yan et al., 2014]
- MapReduce
  - Need to save in HDFS intermediate results of each iteration – both good and bad
  - Hadoop, Haloop [Bu et al., 2012]
- Modified MapReduce
  - Based on Spark [Zaharia et al., 2010; Zaharia, 2016]
    - Keep intermediate states in memory
    - Provide fault-tolerance by keeping lineage
  - GraphX [Gonzalez et al., 2014]

Outline

1. Introduction – Graph Types
2. Property Graph Processing
   - Classification
   - Online querying
   - Offline analytics
3. Graph Analytics Computational Models
   - Vertex-Centric
   - Block-Centric
   - MapReduce-Based
   - Modified MapReduce

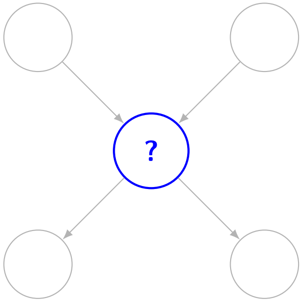
Vertex-Centric Computation

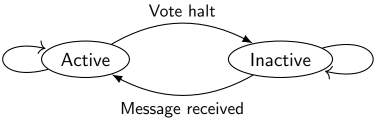
- "Think like a vertex"
- `vertex_scatter(vertex v)`
  - Push local computation to neighbours on the out-bound edges
- `vertex_gather(vertex v)`
  - Gather local computation from neighbours on the in-bound edges
- Continue until all vertices are inactive



Vertex-Centric Computation

- "Think like a vertex"
- `vertex_scatter(vertex v)`
  - Push local computation to neighbours on the out-bound edges
- `vertex_gather(vertex v)`
  - Gather local computation from neighbours on the in-bound edges
- Continue until all vertices are inactive
- Vertex state machine





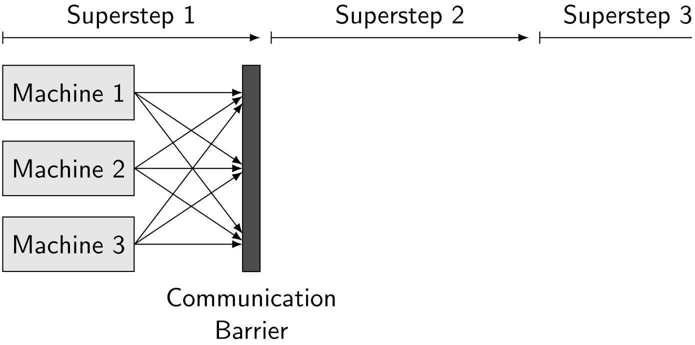
Synchronous Vertex-Centric Computation

Computation

Synchronous Vertex-Centric Computation

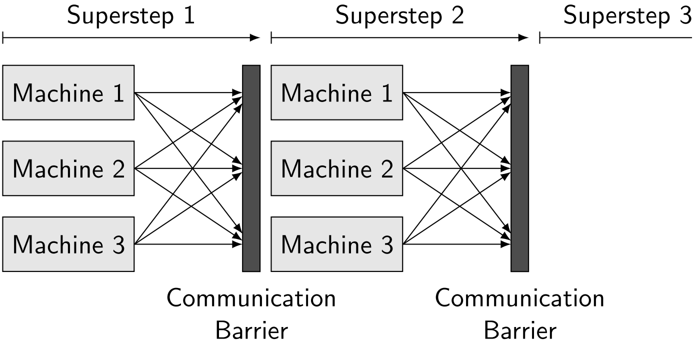
Superstep 1 → Superstep 2 → Superstep 3

Synchronous Vertex-Centric Computation



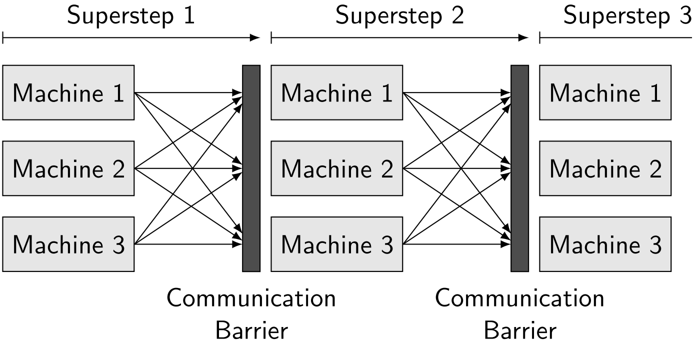
Each machine performs
vertex-centric computation
on its graph partition

Synchronous Vertex-Centric Computation



Each machine performs vertex-centric computation on its graph partition

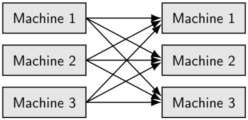
Synchronous Vertex-Centric Computation



Each machine performs vertex-centric computation on its graph partition

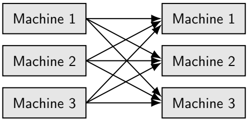
Asynchronous Vertex-Centric Computation

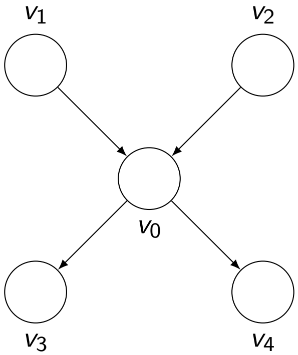
- No communication barriers. ✓
- Uses the *most recent* vertex values. ✓



Asynchronous Vertex-Centric Computation

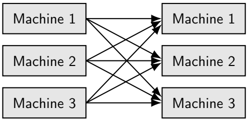
- No communication barriers. ✓
- Uses the *most recent* vertex values. ✓
- Implemented via distributed locking

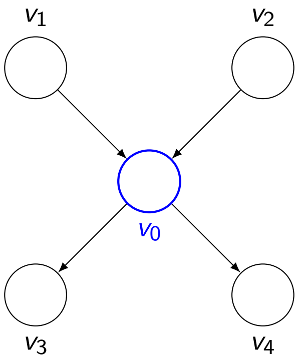




Asynchronous Vertex-Centric Computation

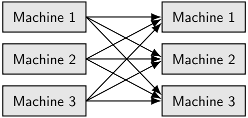
- No communication barriers. ✓
- Uses the *most recent* vertex values. ✓
- Implemented via distributed locking

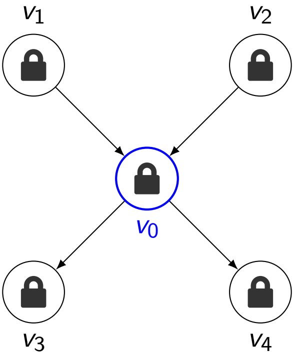




Asynchronous Vertex-Centric Computation

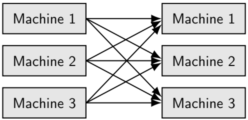
- No communication barriers. ✓
- Uses the *most recent* vertex values. ✓
- Implemented via distributed locking

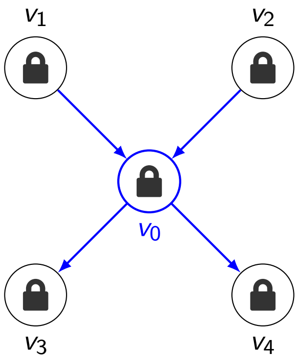




Asynchronous Vertex-Centric Computation

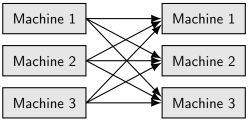
- No communication barriers. ✓
- Uses the *most recent* vertex values. ✓
- Implemented via distributed locking

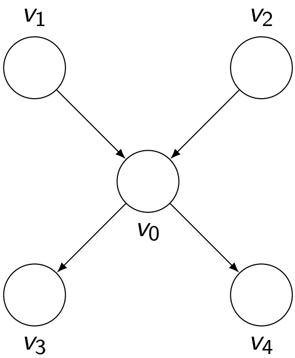




Asynchronous Vertex-Centric Computation

- No communication barriers. ✓
- Uses the *most recent* vertex values. ✓
- Implemented via distributed locking





Summary of an Experiment [Han et al., 2014]

A large study comparing Giraph, GraphLab, GPS, Mizan.

1. Giraph scales better across graphs;
   GraphLab scales better across more machines.

Summary of an Experiment

[Han et al., 2014]

A large study comparing Giraph, GraphLab, GPS, Mizan.

1. Giraph scales better across graphs;
   GraphLab scales better across more machines.

64 machines	TW	UK
Giraph (byte array)	5.8GB	7.0GB
GraphLab (sync)	4.5GB	14GB

TW	16 machines	128 machines
Giraph (byte array)	8.5GB	5.8GB
GraphLab (sync)	11GB	3.3GB

Summary of an Experiment

[Han et al., 2014]

A large study comparing Giraph, GraphLab, GPS, Mizan.

1. Giraph scales better across graphs;
   GraphLab scales better across more machines.
2. Distributed locking for asynchronous execution is not scalable – Performance degrades as more machines are used due to lock contention, termination scheme, lack of message batching

Summary of an Experiment

[Han et al., 2014]

A large study comparing Giraph, GraphLab, GPS, Mizan.

1. Giraph scales better across graphs;
   GraphLab scales better across more machines.
2. Distributed locking for asynchronous execution is not scalable – Performance degrades as more machines are used due to lock contention, termination scheme, lack of message batching
3. Graph storage should be memory and mutation efficient.

Summary of an Experiment

[Han et al., 2014]

A large study comparing Giraph, GraphLab, GPS, Mizan.

1. Giraph scales better across graphs;
   GraphLab scales better across more machines.
2. Distributed locking for asynchronous execution is not scalable – Performance degrades as more machines are used due to lock contention, termination scheme, lack of message batching
3. Graph storage should be memory and mutation efficient.

No Mutations

	Time	**Memory**
Byte array	✓	✓
Hash map	✗	✗

With Mutations (DMST)

	Time	**Memory**
Byte array	✗✗	✓
Hash map	✓	✗

Summary of an Experiment

[Han et al., 2014]

A large study comparing Giraph, GraphLab, GPS, Mizan.

1. Giraph scales better across graphs;
   GraphLab scales better across more machines.
2. Distributed locking for asynchronous execution is not scalable – Performance degrades as more machines are used due to lock contention, termination scheme, lack of message batching
3. Graph storage should be memory and mutation efficient.
4. Message *processing* optimizations are very important.

Summary of an Experiment

[Han et al., 2014]

A large study comparing Giraph, GraphLab, GPS, Mizan.

1. Giraph scales better across graphs;
   GraphLab scales better across more machines.
2. Distributed locking for asynchronous execution is not scalable – Performance degrades as more machines are used due to lock contention, termination scheme, lack of message batching
3. Graph storage should be memory and mutation efficient.
4. Message *processing* optimizations are very important.
5. Workloads have different resource demands

Algorithm	**CPU**	**Memory**	**Network**
PageRank	Medium	Medium	High
SSSP	Low	Low	Low
WCC	Low	Medium	Medium
DMST	High	High	Medium

Outline

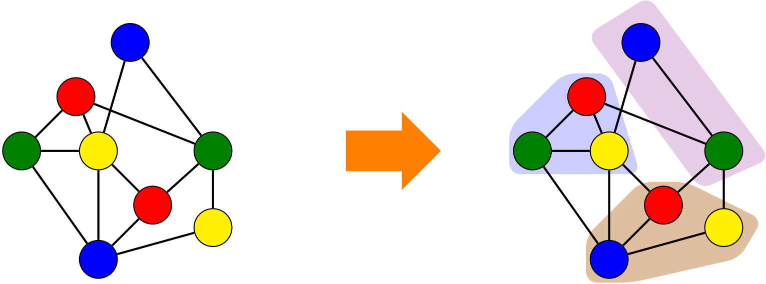
1. Introduction – Graph Types
2. Property Graph Processing
   - Classification
   - Online querying
   - Offline analytics
3. Graph Analytics Computational Models
   - Vertex-Centric
   - Block-Centric
   - MapReduce-Based
   - Modified MapReduce

Block-Centric Computation

- Blogel [Yan et al., 2014]: “Think like a block”; also “think like a graph” [Tian et al., 2013]
- Vertex-centric assumes *all* vertices communicate over the network; this is not efficient
  - 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

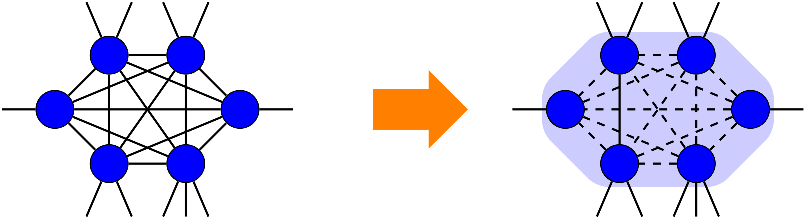
Blogel Principles

- Exploit the partitioning of the graph
- Message exchanges only among blocks
- Block: a connected subgraph of the graph
- Within a block, run a serial in-memory algorithm; no need to follow a BSP model



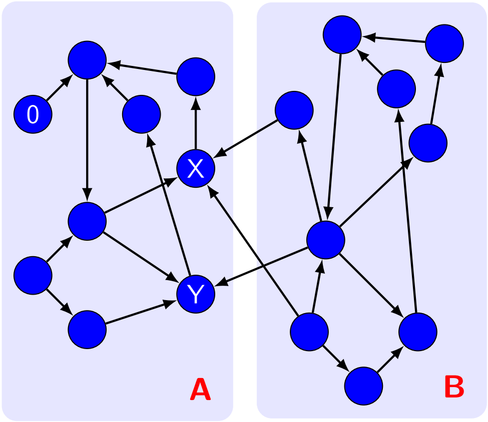
Benefits of Block-Centric Computation

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



Example: Weakly Connected Component

- Algorithm exchanges vertex id's with neighbours
- $\mathrm{id}(v_i) \leftarrow \min\{v_i, v_j, \ldots, v_k\}$ where $v_j, \ldots, v_k$ are neighbours of v_i
- Vertex-centric requires every vertex sends to its neighbours until every vertex is reached
- Block-centric needs two iterations:
  1. All vertices in partition A exchange ids; X and Y send ids to neighbours in partition B
  2. All vertices in partition B exchange ids



Block Construction

- The partitioning algorithm needs to maximize number of vertices that have all their edges in the same partition
- Hash partitioning is not suitable because many vertices will probably have at least one cut-edge
- URL partitioner
  - For web graphs: based on domain names of web page nodes
- 2D partitioner
  - For spatial networks: based on coordinates of node
- Graph Voronoi diagram partitioner
  - For general graphs

Outline

1 Introduction – Graph Types

2 Property Graph Processing
- Classification
- Online querying
- Offline analytics

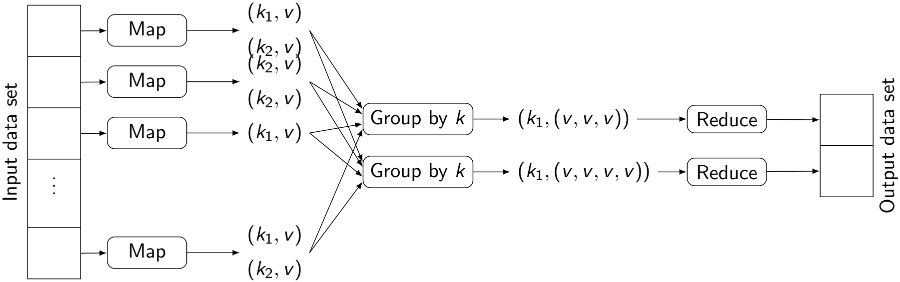
3 Graph Analytics Computational Models
- Vertex-Centric
- Block-Centric
- MapReduce-Based
- Modified MapReduce

MapReduce Basics

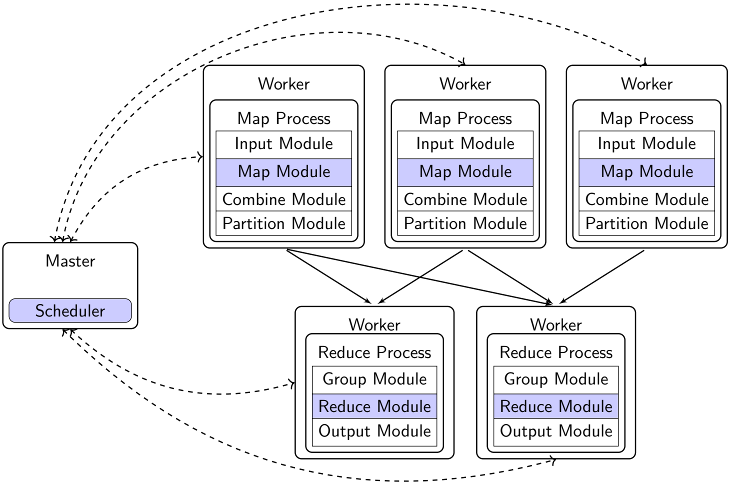
[Li et al., 2014]

- For data analysis of very large data sets
  - Highly dynamic, irregular, schemaless, etc.
  - SQL too heavy
- "Embarrassingly parallel problems"
- New, simple parallel programming model
  - Data structured as (key, value) pairs
    - E.g. (doc-id, content), (word, count), etc.
  - Functional programming style with two functions to be given:
    - `Map(k1,v1)` → `list(k2,v2)`
    - `Reduce(k2, list (v2))` → `list(v3)`
- Implemented on a distributed file system (e.g., Google File System) on very large clusters

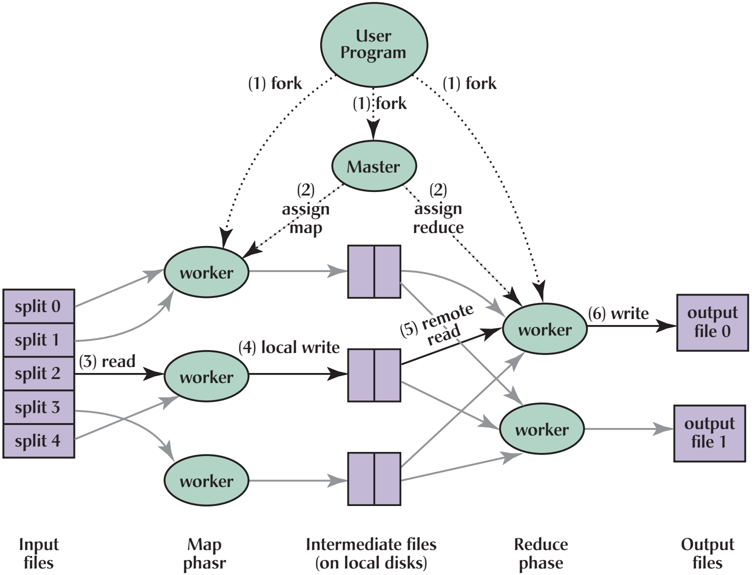
MapReduce Processing



MapReduce Architecture

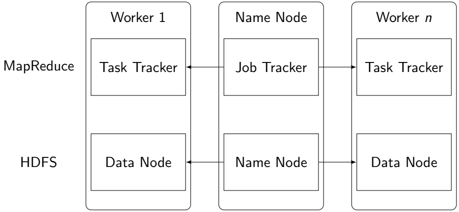


Execution Flow with Architecture [Dean and Ghemawat, 2008]



Hadoop

- Most popular MapReduce implementation – developed by Yahoo!
- Two components
  - Processing engine
  - HDFS: Hadoop Distributed Storage System – others possible
  - Can be deployed on the same machine or on different machines
- Processes
  - Job tracker: hosted on the master node and implements the schedule
  - Task tracker: hosted on the worker nodes and accepts tasks from job tracker and executes them
- HDFS
  - Name node: stores how data are partitioned, monitors the status of data nodes, and data dictionary
  - Data node: Stores and manages *data chunks* assigned to it



HaLoop

[Bu et al., 2012]

- Overcome MapReduce shortcomings for iterative jobs
  - Having to save data in HDFS in between each iteration
  - Checking the fixpoint requires a new job at each iteration
- Scheduler change: assign to the same machine the map & reduce tasks that occur in different iterations but access the same data
- Cache invariant data
- Cache reduce output to easily check for fixpoint

Outline

1. Introduction – Graph Types
2. Property Graph Processing
   - Classification
   - Online querying
   - Offline analytics
3. Graph Analytics Computational Models
   - Vertex-Centric
   - Block-Centric
   - MapReduce-Based
   - Modified MapReduce

Spark System

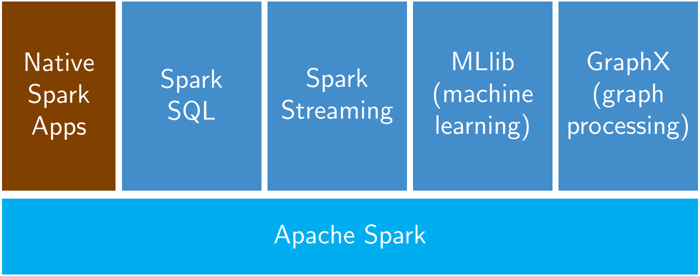
- MapReduce does not perform well in iterative computations
  - Workflow model is acyclic
  - Have to write to HDFS after each iteration and have to read from HDFS at the beginning of next iteration

Spark System

- MapReduce does not perform well in iterative computations
  - Workflow model is acyclic
  - Have to write to HDFS after each iteration and have to read from HDFS at the beginning of next iteration
- Spark objectives
  - Better support for iterative programs
  - Provide a complete ecosystem
  - Similar abstraction (to MapReduce) for programming
  - Maintain MapReduce fault-tolerance and scalability

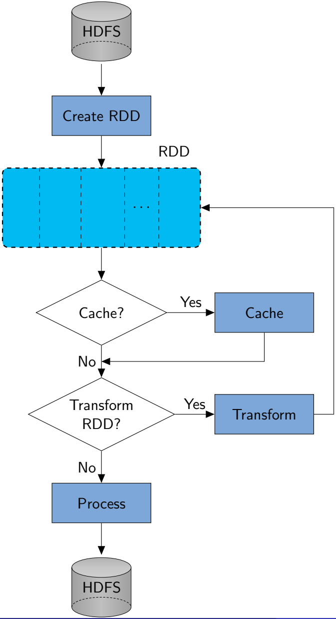
Spark System

- MapReduce does not perform well in iterative computations
  - Workflow model is acyclic
  - Have to write to HDFS after each iteration and have to read from HDFS at the beginning of next iteration
- 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


Native Spark Apps
Spark SQL
Spark Streaming
MLlib (machine learning)
GraphX (graph processing)
Apache Spark

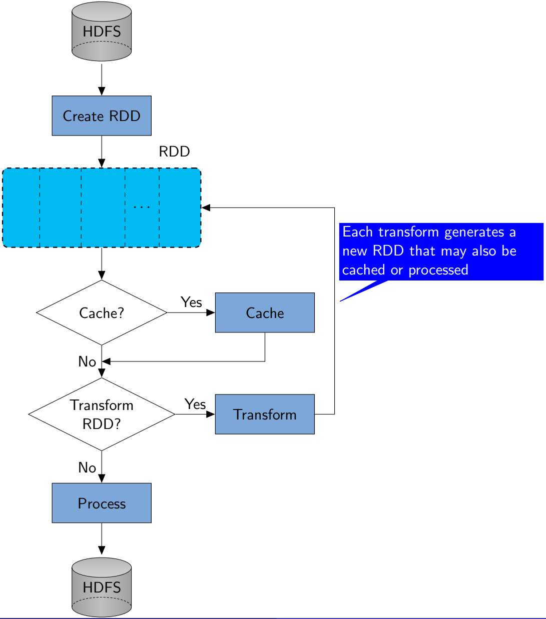
Spark Programming Model

[Zaharia et al., 2010, 2012]



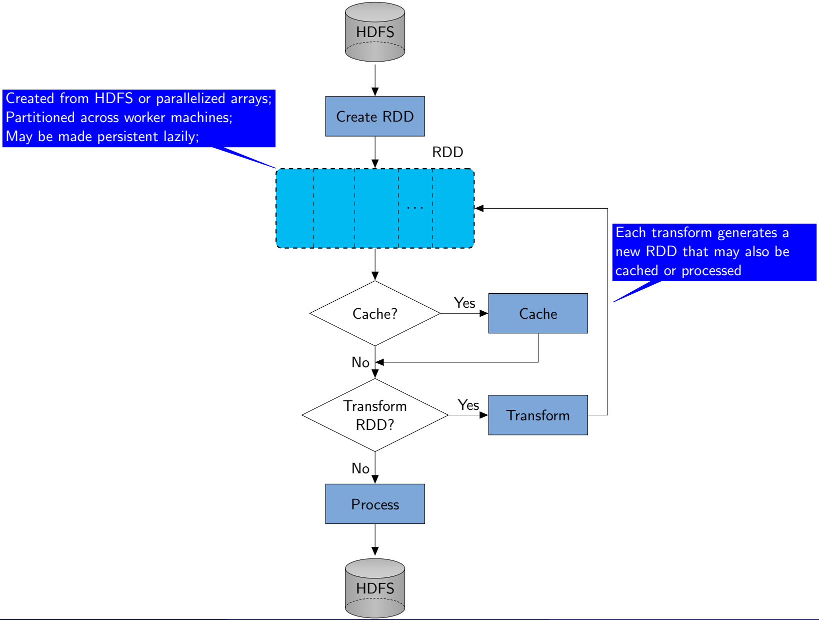
Spark Programming Model

[Zaharia et al., 2010, 2012]



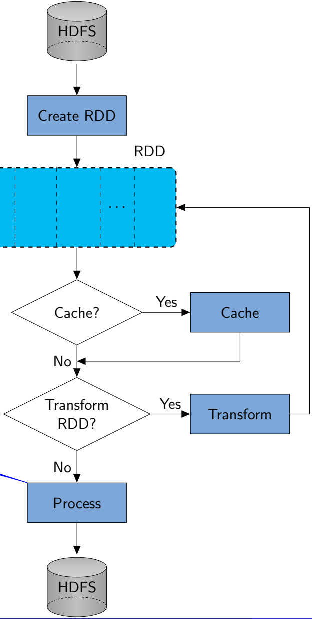
Spark Programming Model

[Zaharia et al., 2010, 2012]



Spark Programming Model

[Zaharia et al., 2010, 2012]

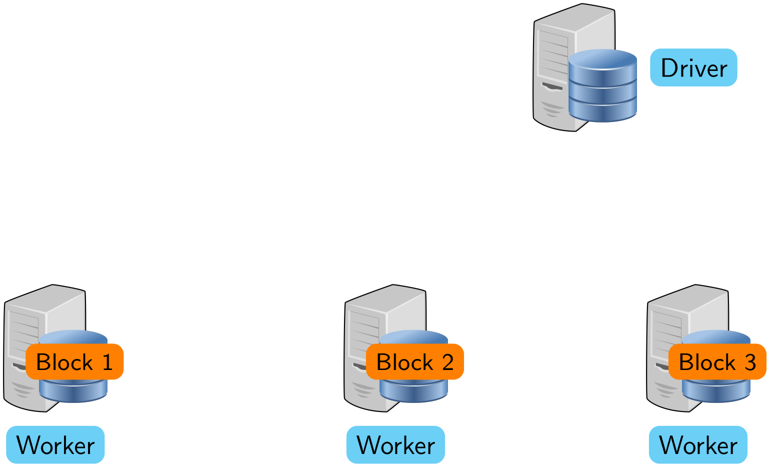


Created from HDFS or parallelized arrays;
Partitioned across worker machines;
May be made persistent lazily;

Each transform generates a new RDD that may also be cached or processed

Processing done on one of the RDDs;
Done in parallel across workers;
First processing on a RDD is from disk;
Subsequent processing of the same RDD from cache

Load log messages from a file system, create a new file by filtering the error messages, read this file into memory, then interactively search for various patterns



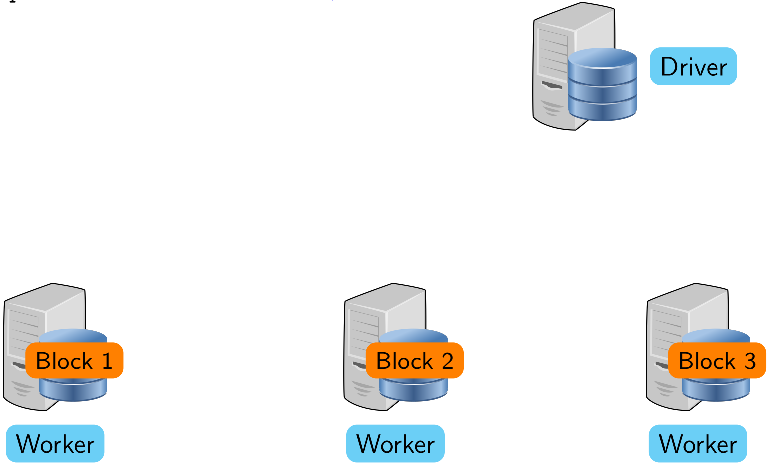
Example – Log Mining

[Zaharia et al., 2010, 2012]

Load log messages from a file system, create a new file by filtering the error messages, read this file into memory, then interactively search for various patterns

CreateRDD

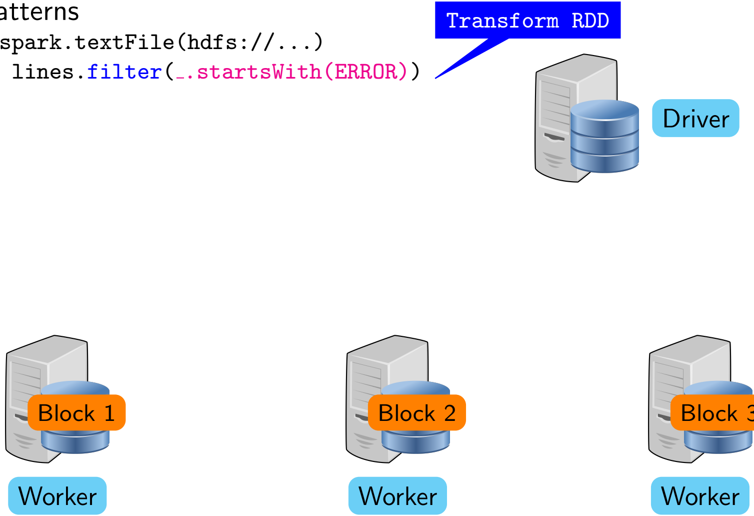
```
lines = spark.textFile(hdfs://...)
```



Example – Log Mining

Load log messages from a file system, create a new file by filtering the error messages, read this file into memory, then interactively search for various patterns

```
lines = spark.textFile(hdfs://...)
errors = lines.filter(_.startsWith(ERROR))
```



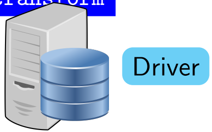
Example – Log Mining

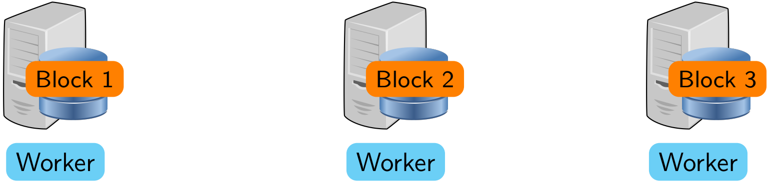
[Zaharia et al., 2010, 2012]

Load log messages from a file system, create a new file by filtering the error messages, read this file into memory, then interactively search for various patterns

```
lines = spark.textFile(hdfs://...)
errors = lines.filter(_.startsWith(ERROR))
messages = errors.map(_.split('\t ')(2))
```

Another transform



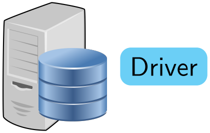


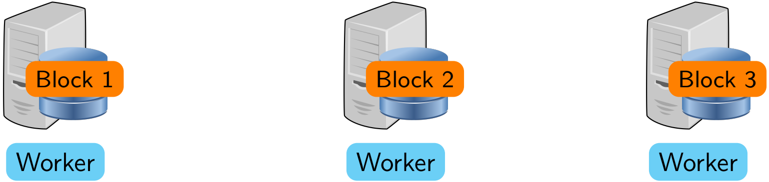
Example – Log Mining [Zaharia et al., 2010, 2012]

Load log messages from a file system, create a new file by filtering the error messages, read this file into memory, then interactively search for various patterns

```
lines = spark.textFile(hdfs://...)
errors = lines.filter(_.starts
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()
```

Cache results





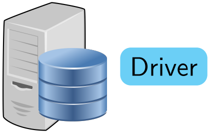
Example – Log Mining

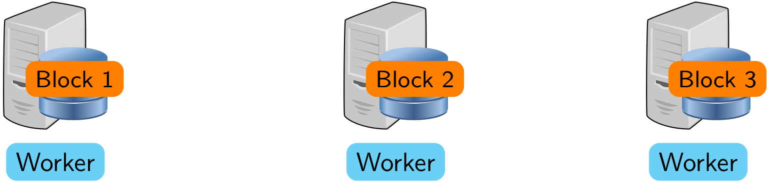
[Zaharia et al., 2010, 2012]

Load log messages from a file system, create a new file by filtering the error messages, read this file into memory, then interactively search for various patterns

```
lines = spark.textFile(hdfs://...)
errors = lines.filter(_.startsWith(ERROR))
messages = errors.map(_.split('\t ')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains(foo)).count
```

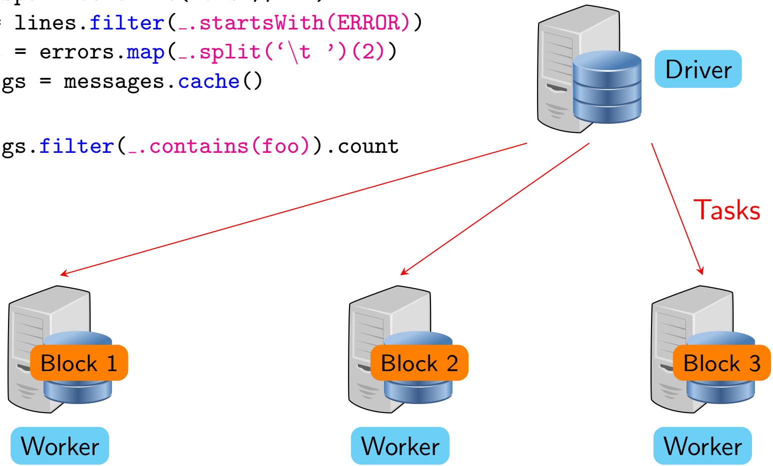
Action





Load log messages from a file system, create a new file by filtering the error messages, read this file into memory, then interactively search for various patterns

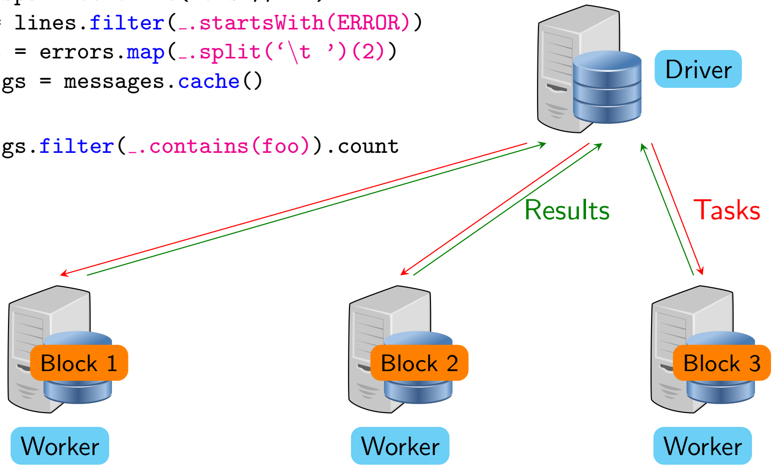
```
lines = spark.textFile(hdfs://...)
errors = lines.filter(_.startsWith(ERROR))
messages = errors.map(_.split('\t ')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains(foo)).count
```



Example – Log Mining [Zaharia et al., 2010, 2012]

Load log messages from a file system, create a new file by filtering the error messages, read this file into memory, then interactively search for various patterns

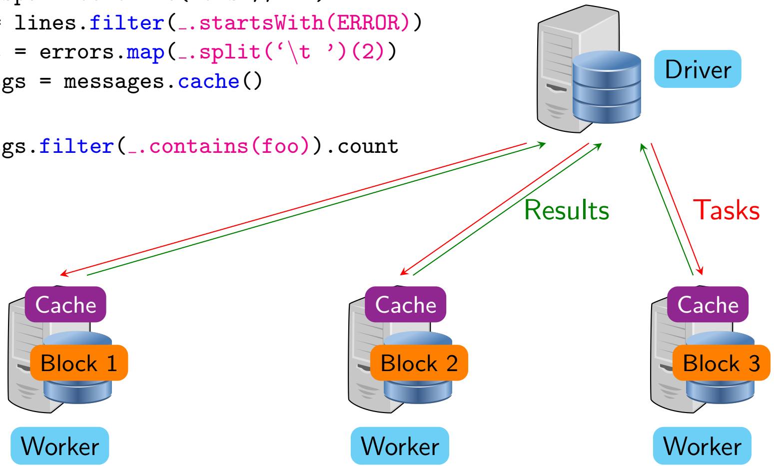
```
lines = spark.textFile(hdfs://...)
errors = lines.filter(_.startsWith(ERROR))
messages = errors.map(_.split('\t ')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains(foo)).count
```



Example – Log Mining [Zaharia et al., 2010, 2012]

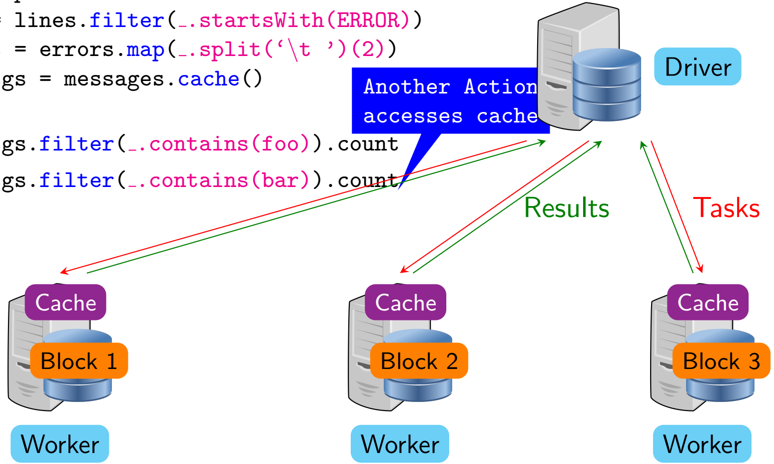
Load log messages from a file system, create a new file by filtering the error messages, read this file into memory, then interactively search for various patterns

```
lines = spark.textFile(hdfs://...)
errors = lines.filter(_.startsWith(ERROR))
messages = errors.map(_.split('\t ')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains(foo)).count
```

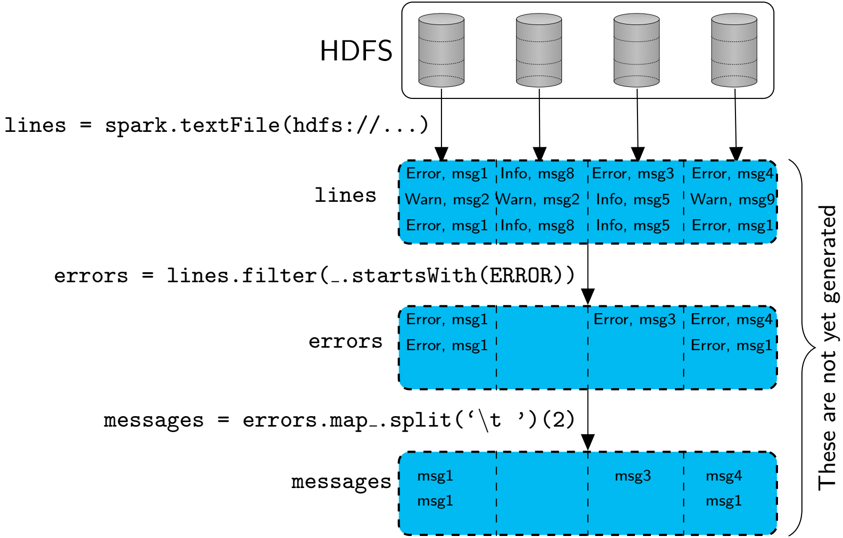


Load log messages from a file system, create a new file by filtering the error messages, read this file into memory, then interactively search for various patterns

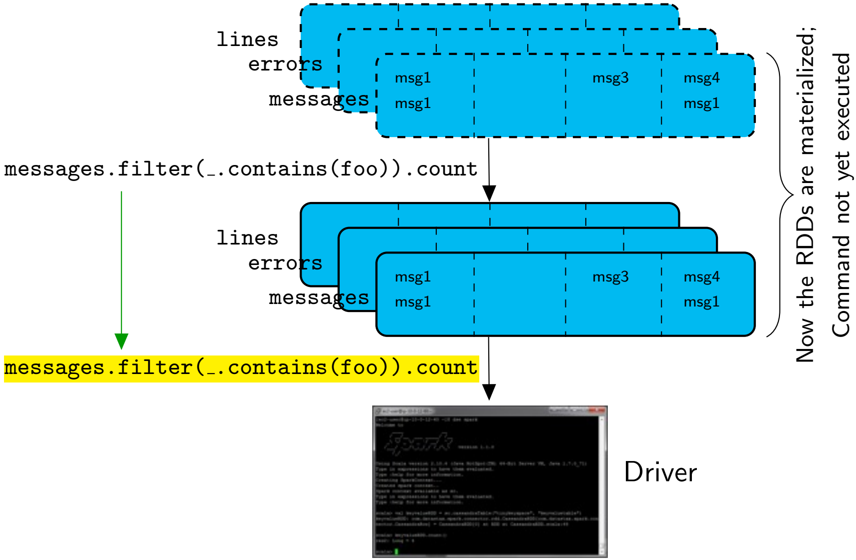
```
lines = spark.textFile(hdfs://...)
errors = lines.filter(_.startsWith(ERROR))
messages = errors.map(_.split(‘\t ’)(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains(foo)).count
cachedMsgs.filter(_.contains(bar)).count
```



RDD and Processing



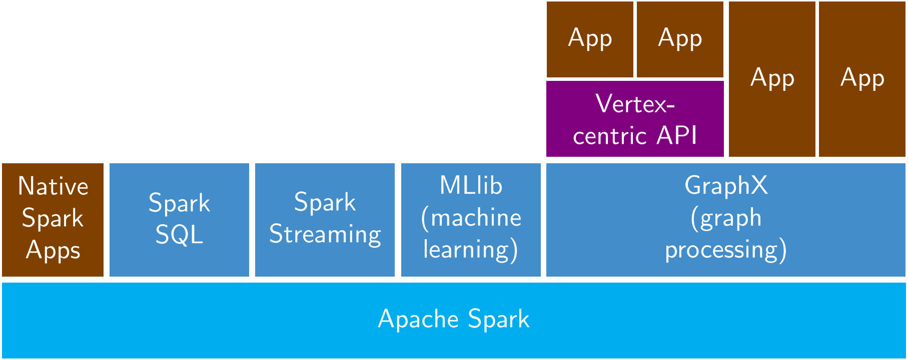
RDD and Processing



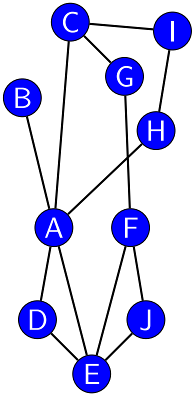
GraphX

[Gonzalez et al., 2014]

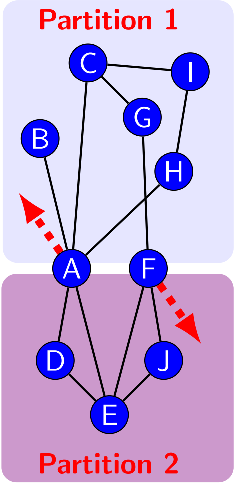
- Built on top of Spark
- Objective is to combine data analytics with graph processing
  - Unify computation on tables and graphs
- Carefully convert graph to tabular representation
- Native GraphX API or can accommodate vertex-centric computation



GraphX: Representation of Graphs as Tables

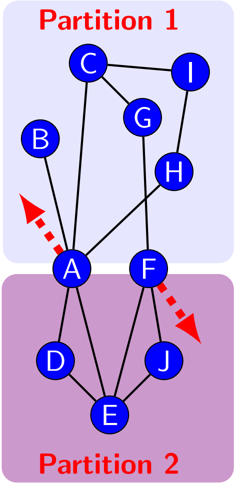


GraphX: Representation of Graphs as Tables

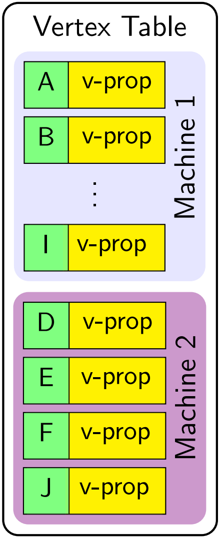


Edge-disjoint partitioning

GraphX: Representation of Graphs as Tables

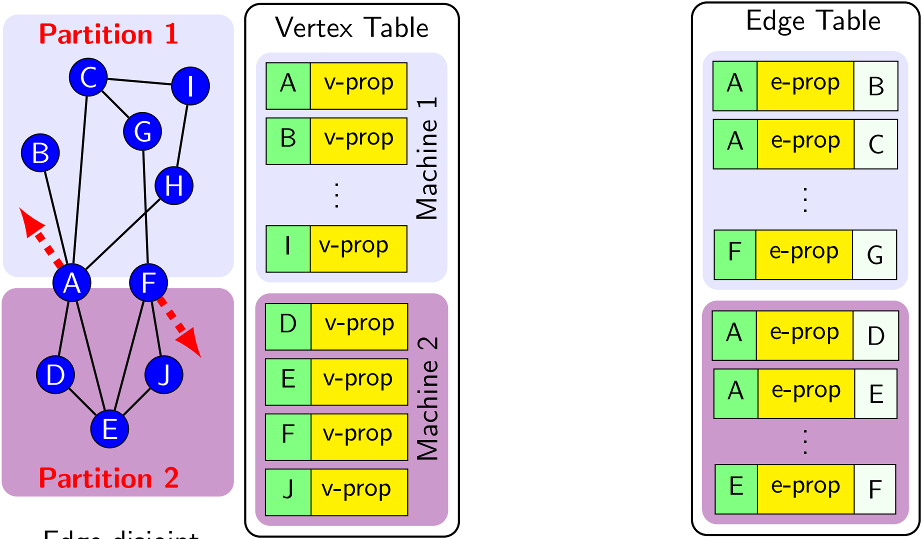


Edge-disjoint partitioning



(RDD)
v-prop:vertex prop.

GraphX: Representation of Graphs as Tables

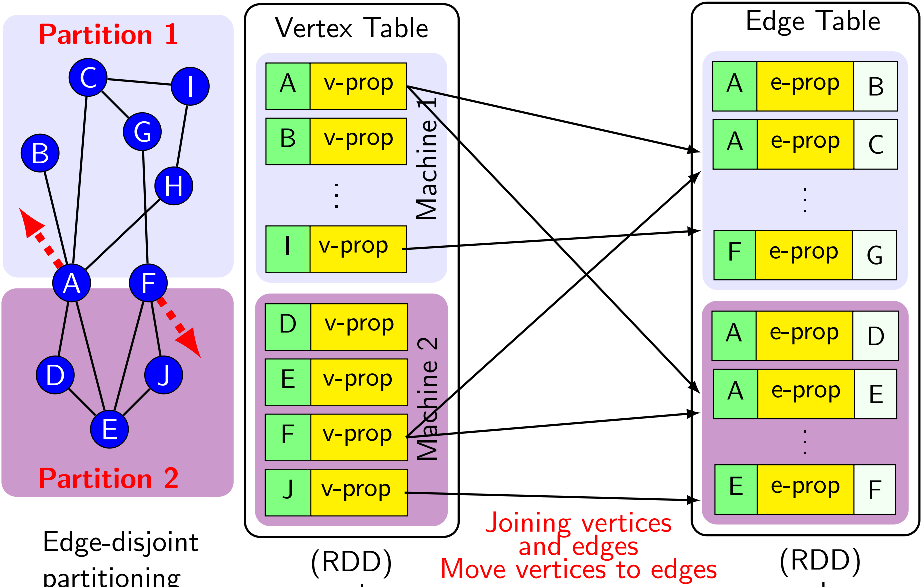


Edge-disjoint partitioning

(RDD)
v-prop:vertex prop.

(RDD)
e-prop:edge prop.

GraphX: Representation of Graphs as Tables

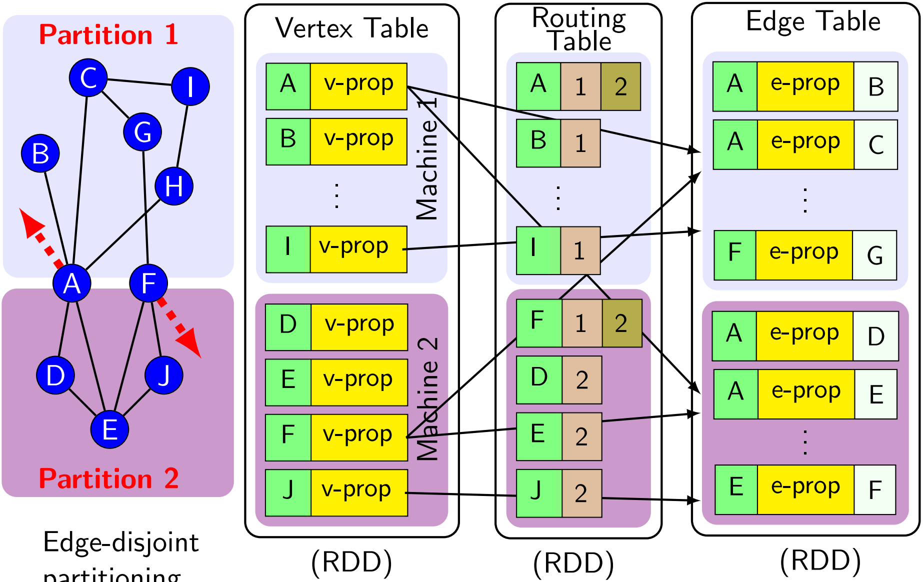


Edge-disjoint partitioning

(RDD)
v-prop:vertex prop.

(RDD)
e-prop:edge prop.

GraphX: Representation of Graphs as Tables

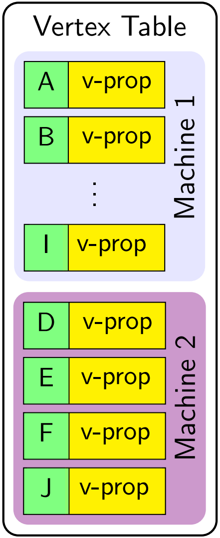


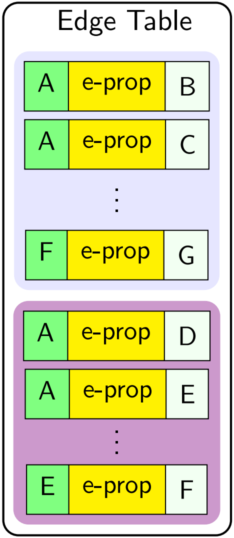
Edge-disjoint partitioning

v-prop:vertex prop.

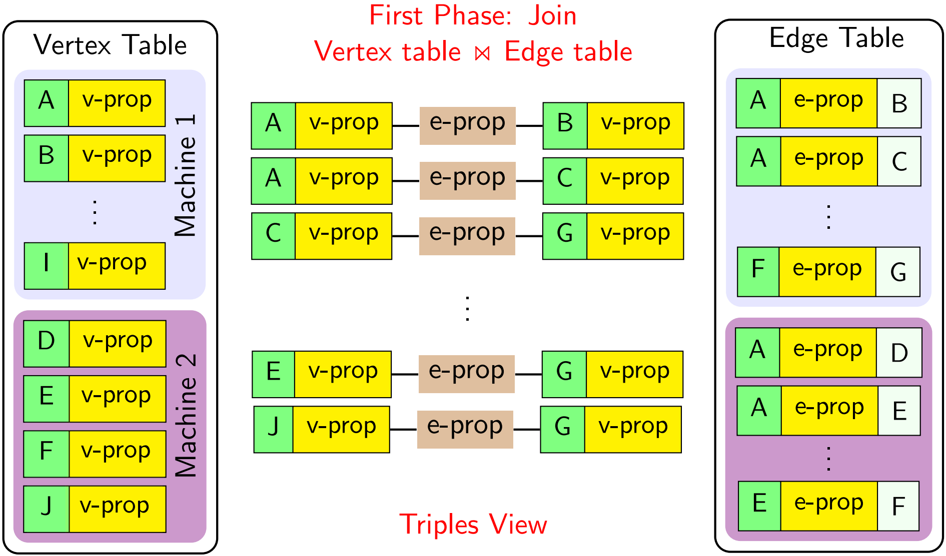
e-prop:edge prop.

GraphX: Computation Model

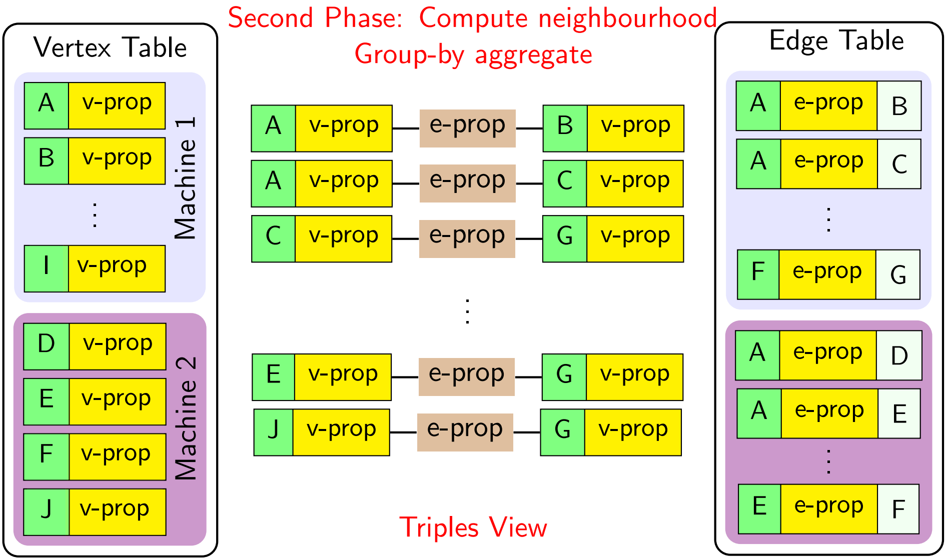




GraphX: Computation Model



GraphX: Computation Model



GraphX: Operators

- Table transform operators – inherited from Spark

map(*func*)	Return a new RDD formed by passing each element of the source through a function *func*
filter(*func*)	Return a new RDD formed by selecting those elements of the source on which *func* returns true
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator
sample(*repl*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement (set *repl* accordingly), using a given random number generator seed
union(*otherDataset*) intersection()	Return a new RDD containing the union/intersection of the elements in the source RDD and the argument
groupByKey()	Operates on a RDD of (K, V) pairs, returns a RDD of (K, Iterable<V>) pairs
reduceByKey(*func*, . . .)	Operates on a RDD of (K, V) pairs, returns a RDD of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*

GraphX: Operators

- Table transform operators – inherited from Spark
- Graph operators

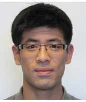
Graph(*vertex coll*, *edge coll*)	Logically binds together a pair of vertex and edge property collections into a property graph; verifies that each vertex occurs only once and edges connect existing vertices
triplets(*vertex coll*, *vertex coll*, *edge coll*)	Returns the triplets view of the graph
mrTriplets(map,reduce)	MapReduce triplets - encodes the two-stage process of join to create triplets and group by

Acknowledgements

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

Khaled Ammar, U. Waterloo

Khuzaima Daudjee, U. Waterloo



Young Han, U. Waterloo

Thank you!

Research supported by

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

CAS Research
Technology Incubation Lab

IBM Canada Lab

References I

Ammar, K. and Özsu, M. T. (2016). Approaches to graph processing – an overview. In preparation.

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.

Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

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. In *Proc. 11th USENIX Symp. on Operating System Design and Implementation*, pages 599–613.

Han, M., Daudjee, K., Ammar, K., Özsu, M. T., Wang, X., and Jin, T. (2014). An experimental comparison of Pregel-like graph processing systems. *Proc. VLDB Endowment*, 7(12):1047–1058.

Li, F., Ooi, B. C., Özsu, M. T., and Wu, S. (2014). Distributed data management using MapReduce. *ACM Comput. Surv.*, 46(3):Article No. 31.

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.

References II

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.

Michiardi, P. (2015). Introduction to spark internals. Slideshare. Available from: `http://www.slideshare.net/michiard/introduction-to-spark-internals?qid=511145e7-79d7-41d8-a133-9e705d4933c3&v=qf1&b=&from_search=11` [Last retrieved: 9 July 2015].

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.

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.

Zaharia, M. (2016). *An Architecture for Fast and General Data Processing on Large Clusters*. ACM Books. Forthcoming.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. 9th USENIX Symp. on Networked Systems Design & Implementation*, pages 2–2.

References III

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. In *Proc. 2nd USENIX Workshop on Hot Topics in Cloud Computing*, pages 10–10.