Programming for Big Data Processing

Hai Jin

National Engineering Research Center for Big Data Technology and System (NERC-BDTS)

School of Computer Science and Technology
Huazhong University of Science and Technology
Wuhan, 430074, China

What is big data?

A massive volume of both structured and unstructured data that is too large to process with traditional database and software techniques.

- 34K likes every minute
- It deals with 3-4 PB data every day
- 1 billion active user

- It generates 12TB data daily
- 200 million users generate 230 million tweets daily

- 2 million search every minute
- It deals with 20 PB data every day

Application of big data analytics

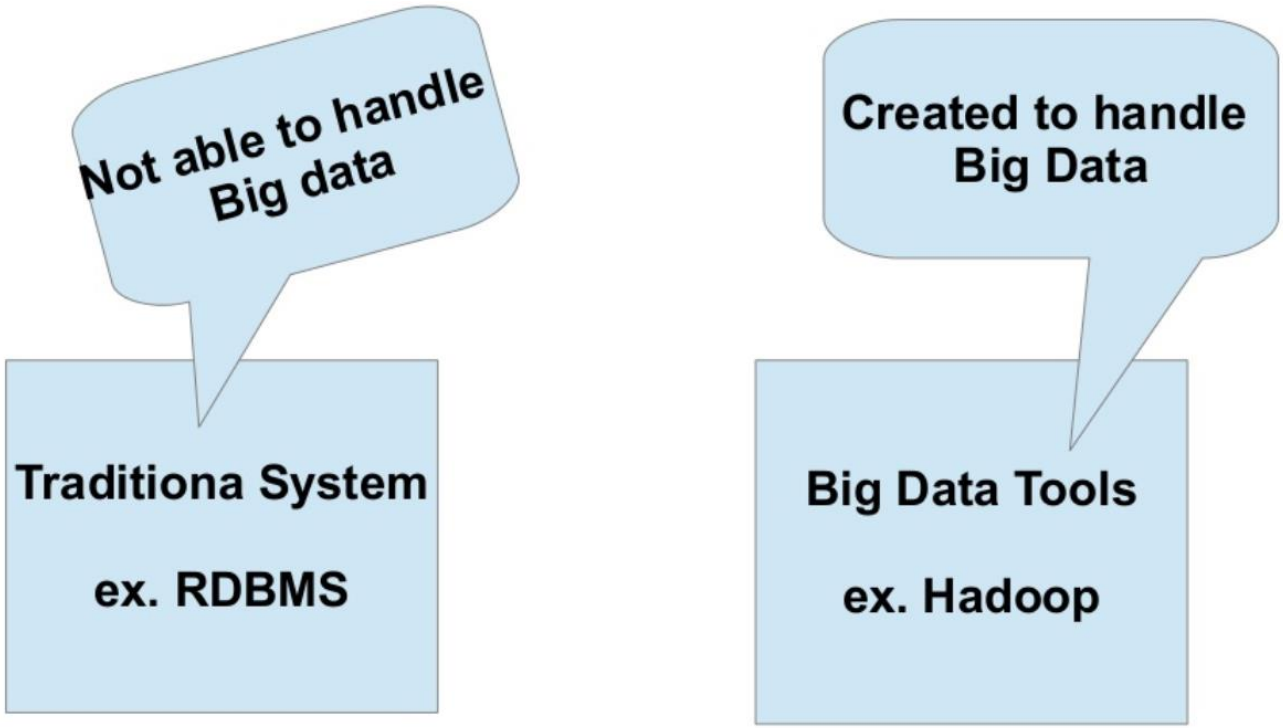
Smart healthcare

Manufacturing

Traffic control

Trading analytics

Tools for handling big data analytics



Outline

- Programming for batch data processing
- Programming for graph processing
- Programming for stream processing

PROGRAMMING FOR BATCH DATA PROCESSING

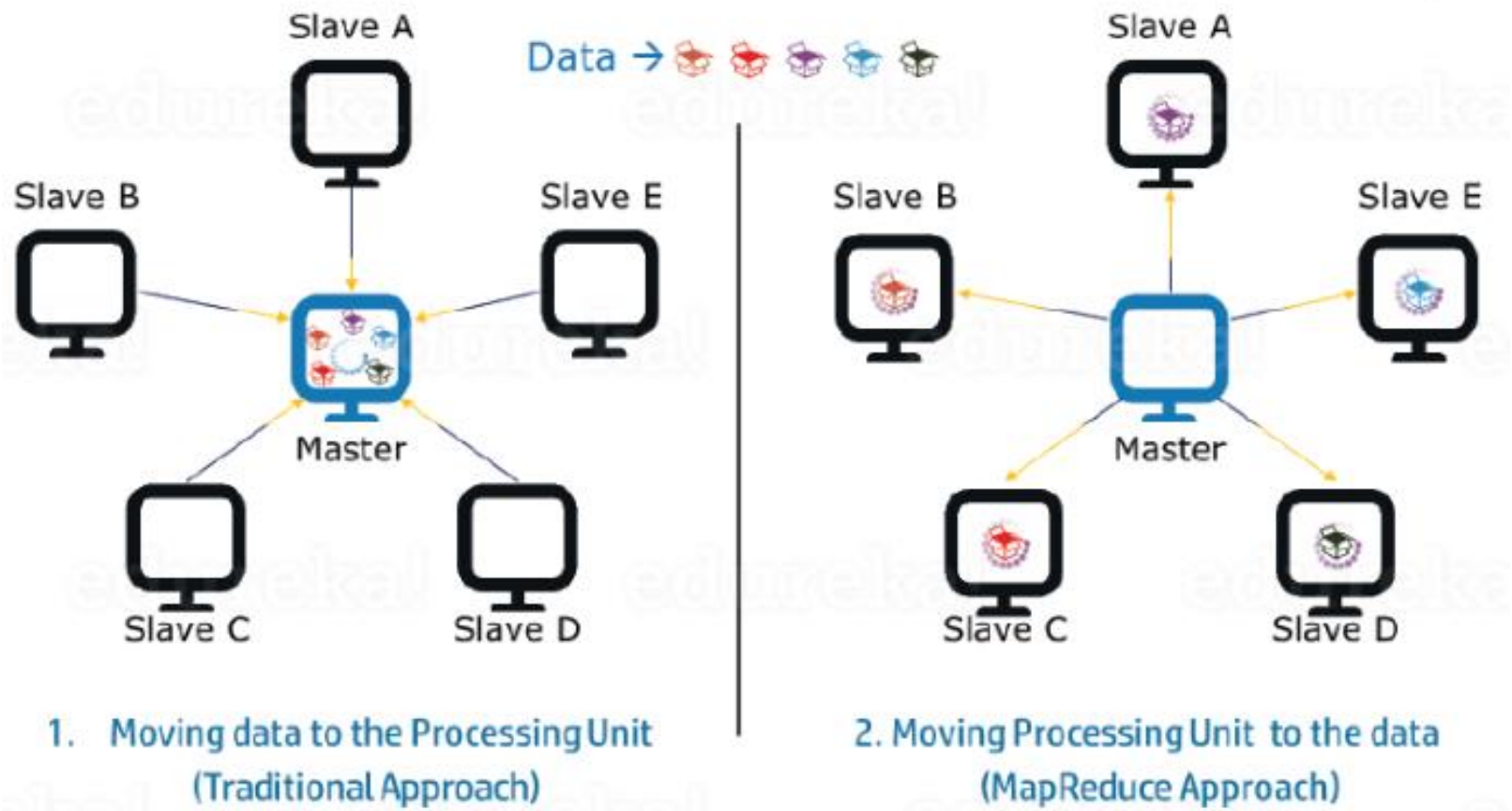
Outline

- Background of batch data processing
- Programming model for batch data processing
- MapReduce framework

Batch data processing

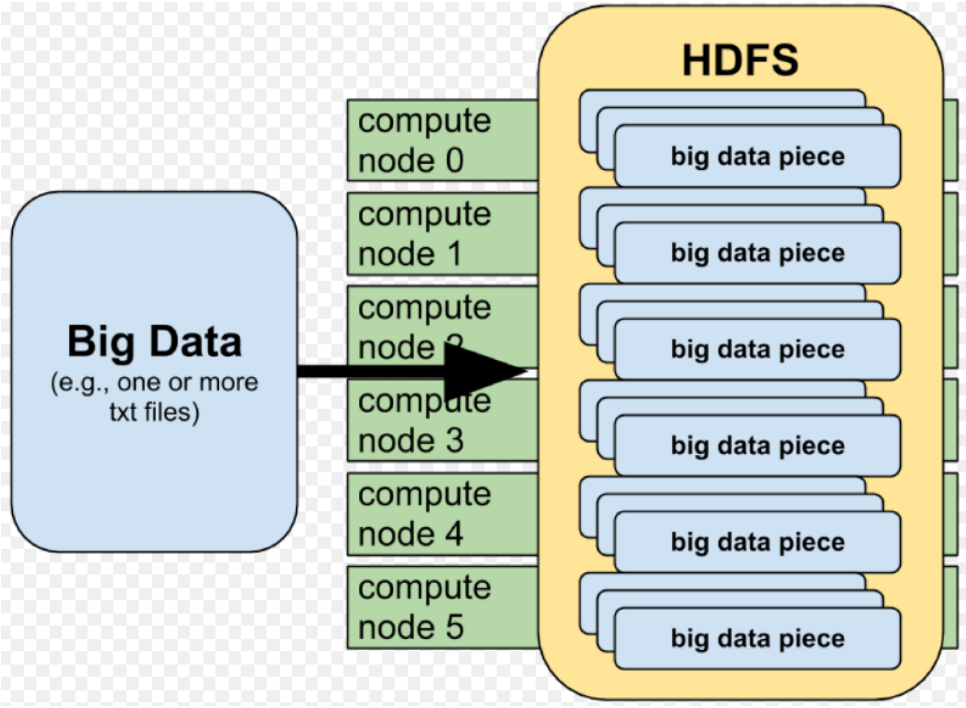
- Big data analytics usually contain tasks that perform batch data processing
  - Tasks that can be executed offline without user interaction
  - It process all data records in batch
  - Tools for batch processing of big data
- Challenge
  - Data placement, computation overhead
- Solution
  - Parallel execution on clusters with thousands of machines
  - Moving processing unit to the data

Big data placement



Parallel batch data processing

- Divide the input data into pieces
- Distribute the pieces on nodes and manage the data with distributed file system
- Parallel process of data piece on each node
- Aggregate the results



Challenges for programmers

- Data placement
  - Data division, redundancy
- Computing parallelization
  - Synchronization between processes, consistency
- Communication
- Fault tolerant
- Load balancing

Solution

- A general abstraction for special-purpose applications
  - Express the various simple computation
  - Hide the details of parallelization, such as fault-tolerance, data distribution, and load balancing
  - Provide powerful interfaces to users
  - Programmers can implement the application code without considering the messy parallelization details
  - Automatic parallelization and distribution of large-scale computation

A general programming interface for various special-purpose applications is required!

Outline

- Background of batch data processing
- Programming model for batch data processing
- MapReduce framework

Programming model for batch data processing

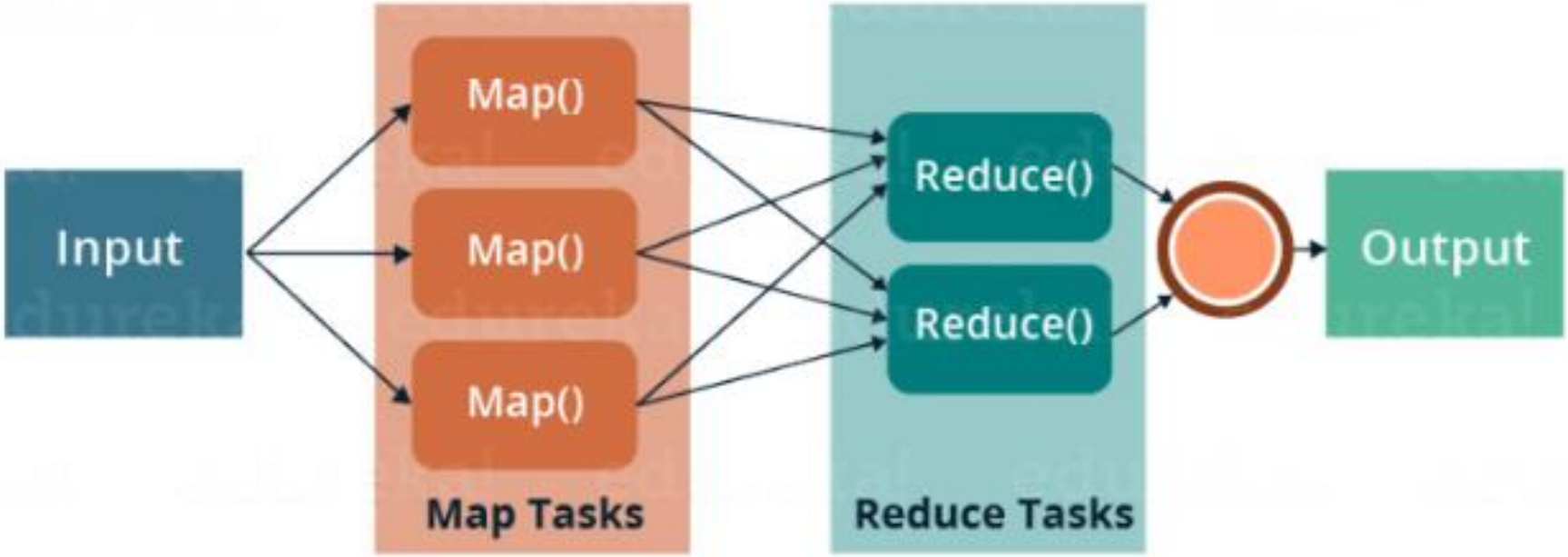
- Inspired by primitives of Lisp and other functional languages
- Most of our computations can be represented by a map and reduce operator to each logical record in the input dataset
  - The map and reduce operator are interfaces for programmers
- We can develop a general framework **MapReduce** for batch data processing
  - Provide the powerful map/reduce interface for programmers
  - Deal with parallelization details transparently and automatically

Program model

- Read a lot of data
- **Map**: extract something you care about from each data record
- Shuffle and sort the data
- **Reduce**: aggregate, summarize, filter, or transform the intermediate data
- Write the final results

Users can change the definition of map and reduce to fit different problems

Program model



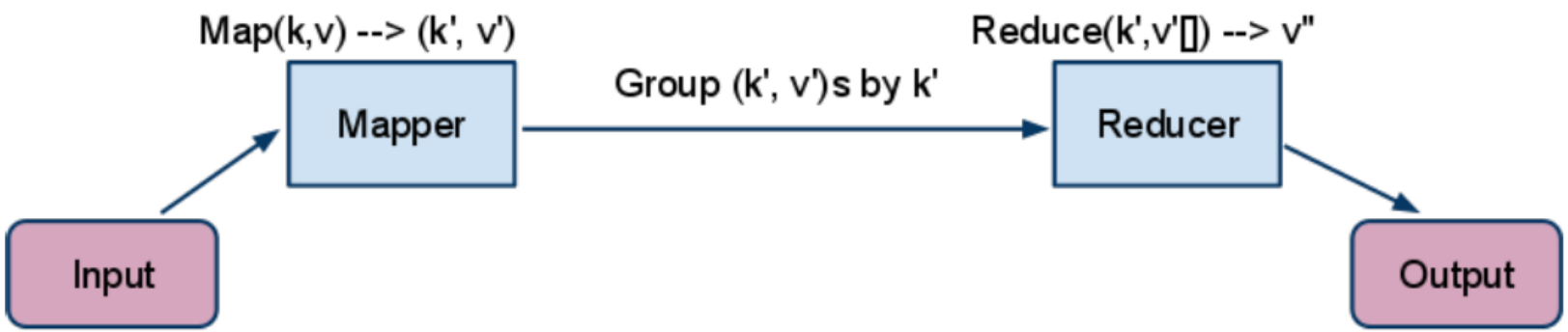
Map/reduce abstraction for large-scale data

- Divide the large-scale input dataset into splits
- Distribute all splits on machines
- Perform map operator on each split
- Shuffle and sort intermediate results
- Perform reduce operator on each sorted results
  - Aggregate the intermediate results
- Generate final output

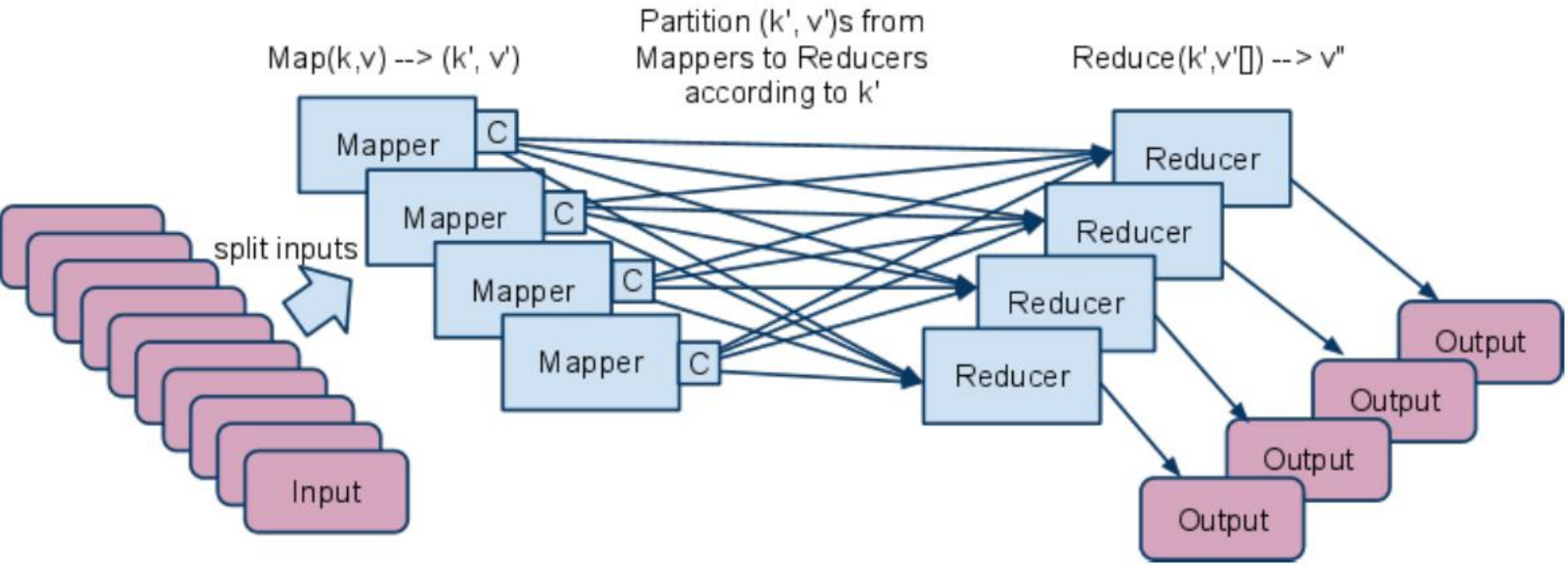
Programmers only need to specify the map and reduce operator without considering other details.

Map/reduce operators

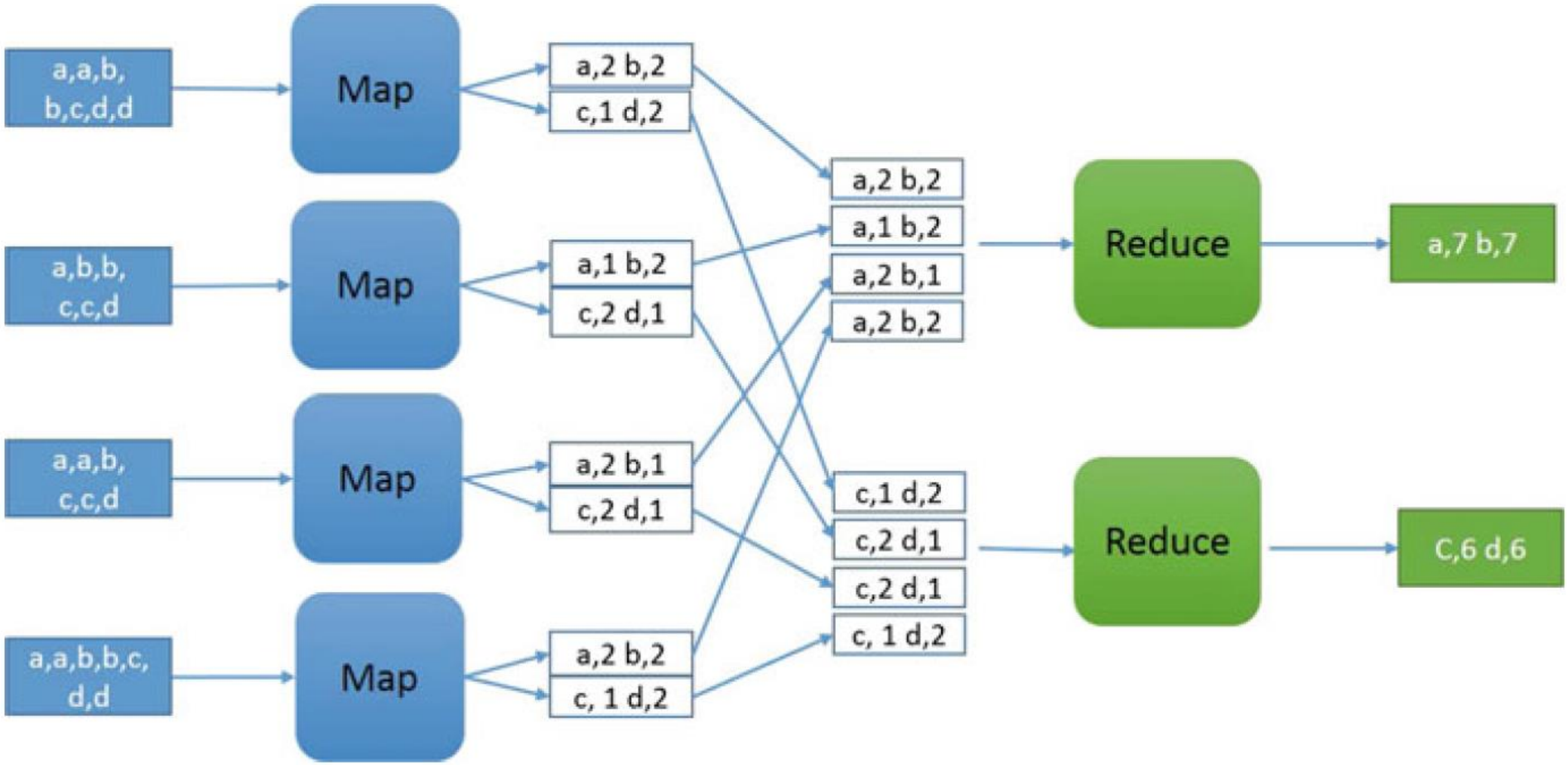
- Programmers specify two primary functions
  - map (k, v) → <k', v'>
  - reduce (k', [v']) → <k'', v''>
  - All v' with the same k' are sent to the same reducer (shuffle)



Map/reduce abstraction for large-scale data



Word count solution



Word count solution

```
//Pseudo-code for "word counting"
map(String key, String value):
  // key: document name,
  // value: document contents
    for each word w in value:
      EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
    int word_count = 0;
    for each v in values:
      word_count += ParseInt(v);
    Emit(key, AsString(word_count));
```

Outline

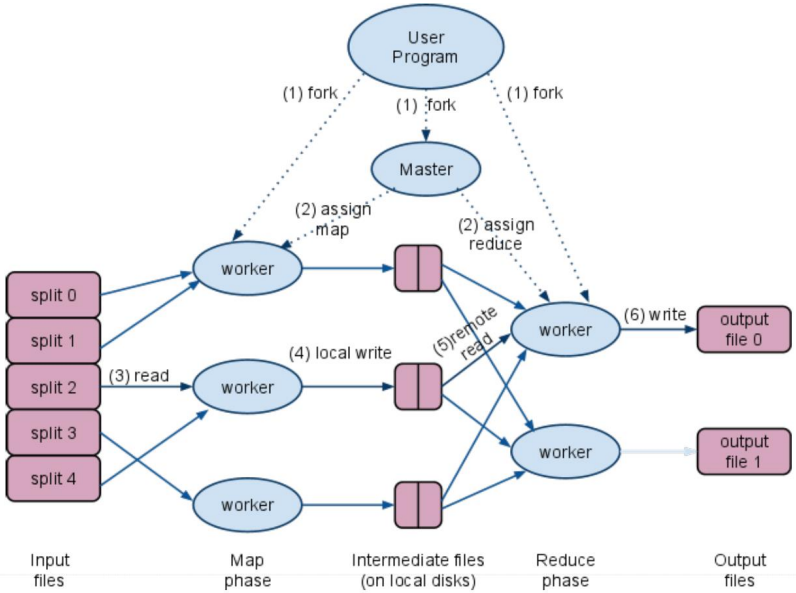
- Batch data processing
- Programming model for batch data processing
- MapReduce framework

Functions of MapReduce framework

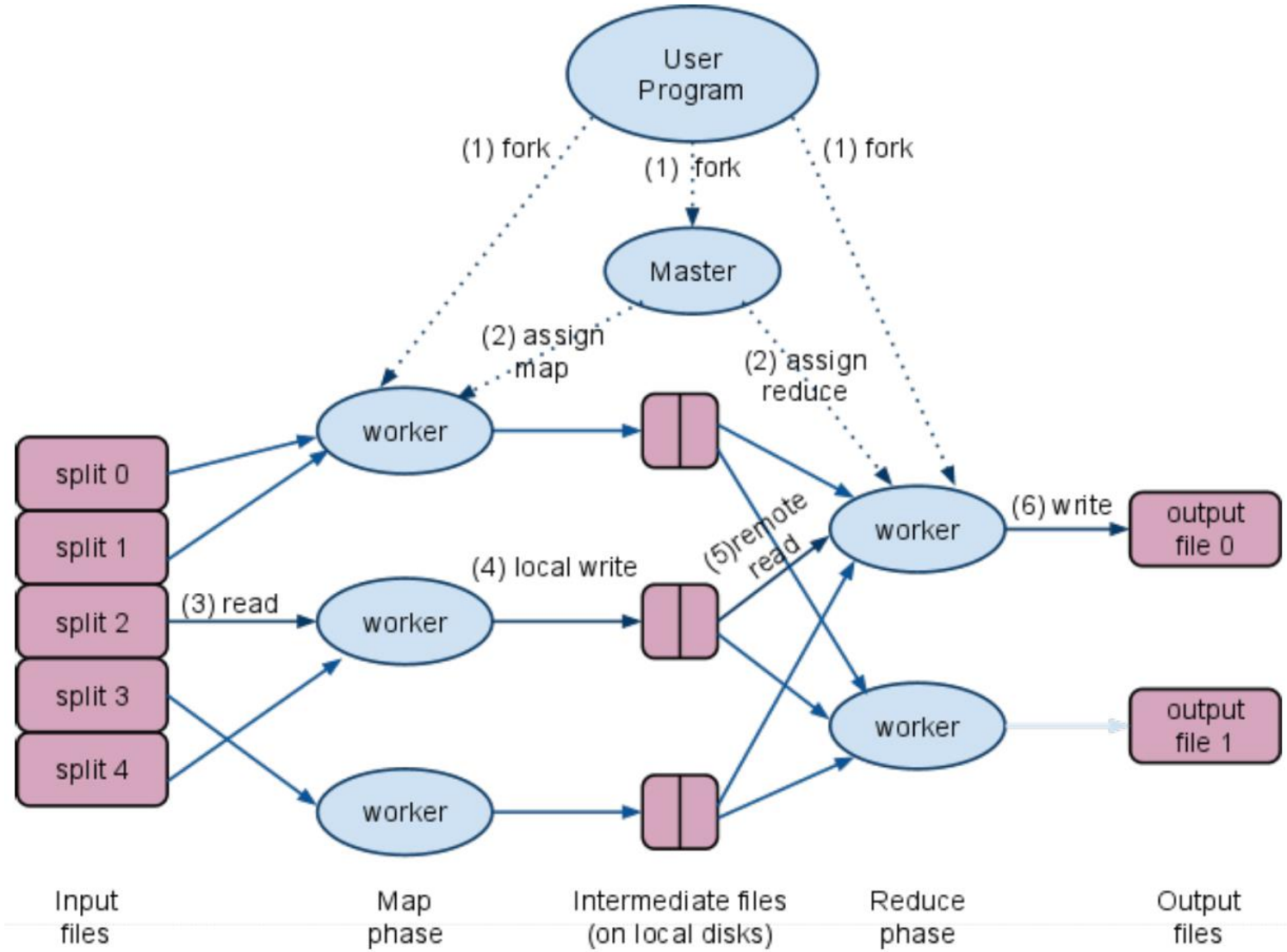
- Programmers specify the map/reduce operators, MapReduce framework should handle
  - Data distribution
    - Split data for multiple workers, move processes to data
  - Synchronization
    - Shuffle intermediate data
  - Fault tolerance
    - Detect failed machines and automatically restart, recover the failed tasks
  - Scheduling
    - Load balancing, detect stragglers
  - Scalability
    - Fast growing dataset, extension of machines

Architecture of MapReduce framework

- One master, many workers
- Master assigns map/reduce task to a free worker
- Workers perform the assigned tasks with corresponding input data
- Master schedules the tasks, handles the failed workers

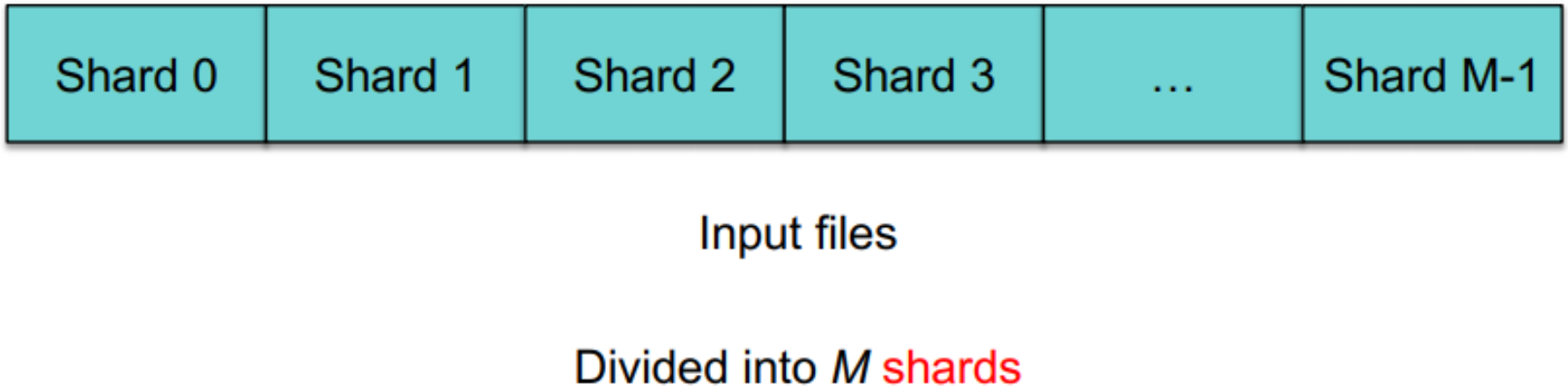


Execution overview of MapReduce framework



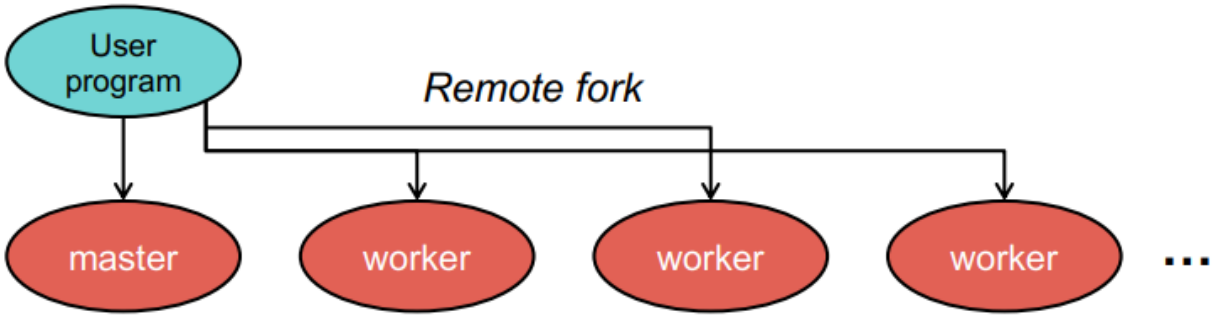
Step 1: split input files into shards

- Break up the input data into M pieces (typically 64MB)



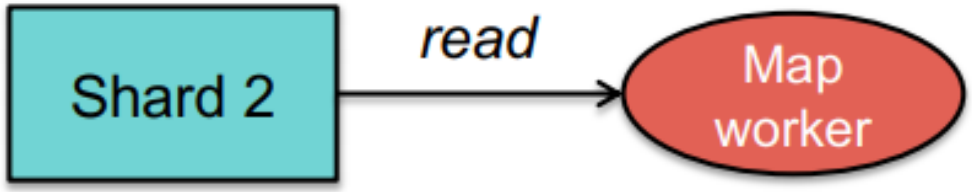
Step 2: fork processes

- Start up many copies of the program on a cluster of machines
  - One master: scheduler and coordinator
  - Lots of workers
- Idle workers are assigned
  - Map tasks (each works on a shard) – there are M map tasks
  - Reduce tasks (each works on intermediate pairs) – there are R reduce tasks
    - R is defined by the user



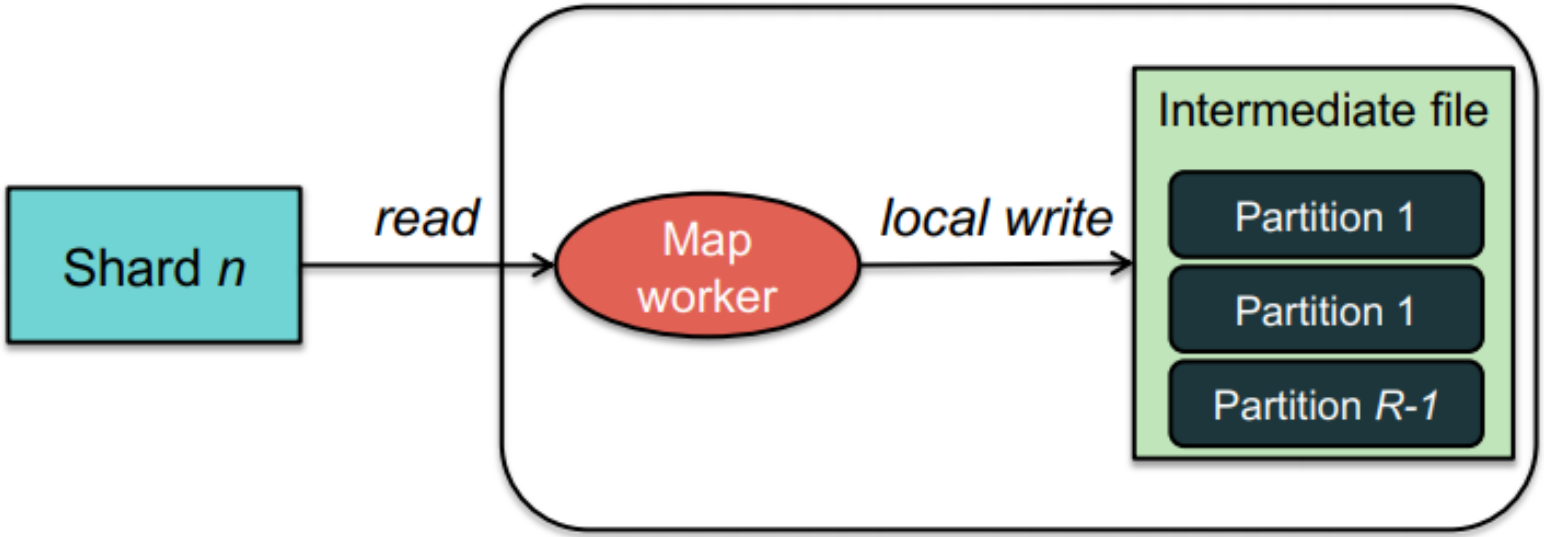
Step 3: run map tasks

- Map worker reads the input shard assigned to it
- Parse key/value pairs out of the input shard
- Pass each pair to the user-specified map function
  - Produce intermediate key/value pairs
  - Intermediate pairs are buffered in memory



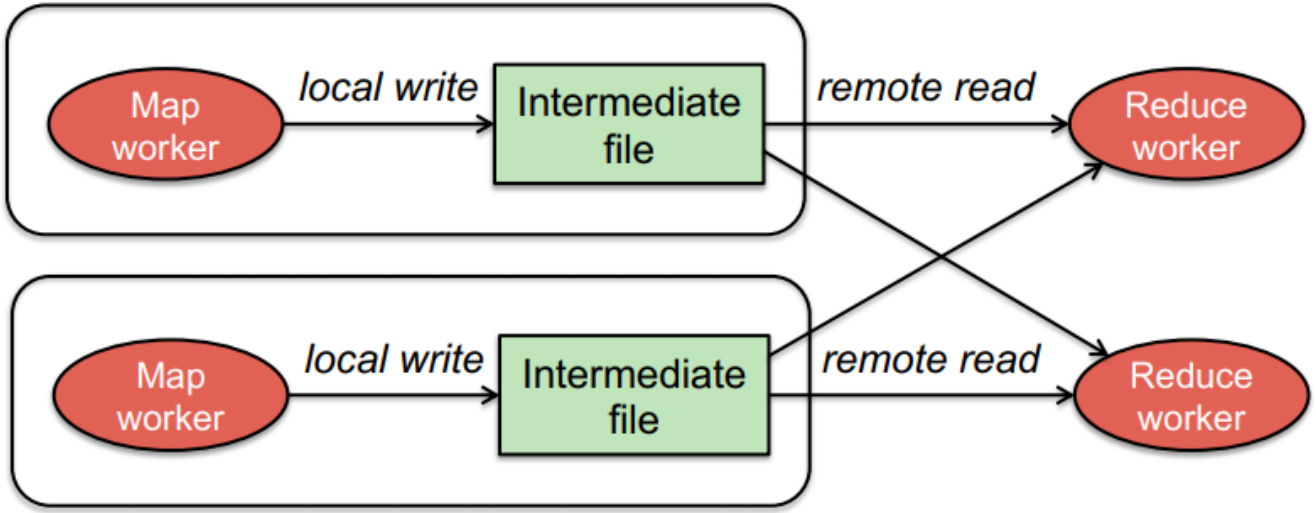
Step 4: create intermediate files

- Intermediate pairs produced by map worker are periodically written to the local disk
  - Partitioned into R regions by a partitioning function
  - Sort all intermediate pairs by keys



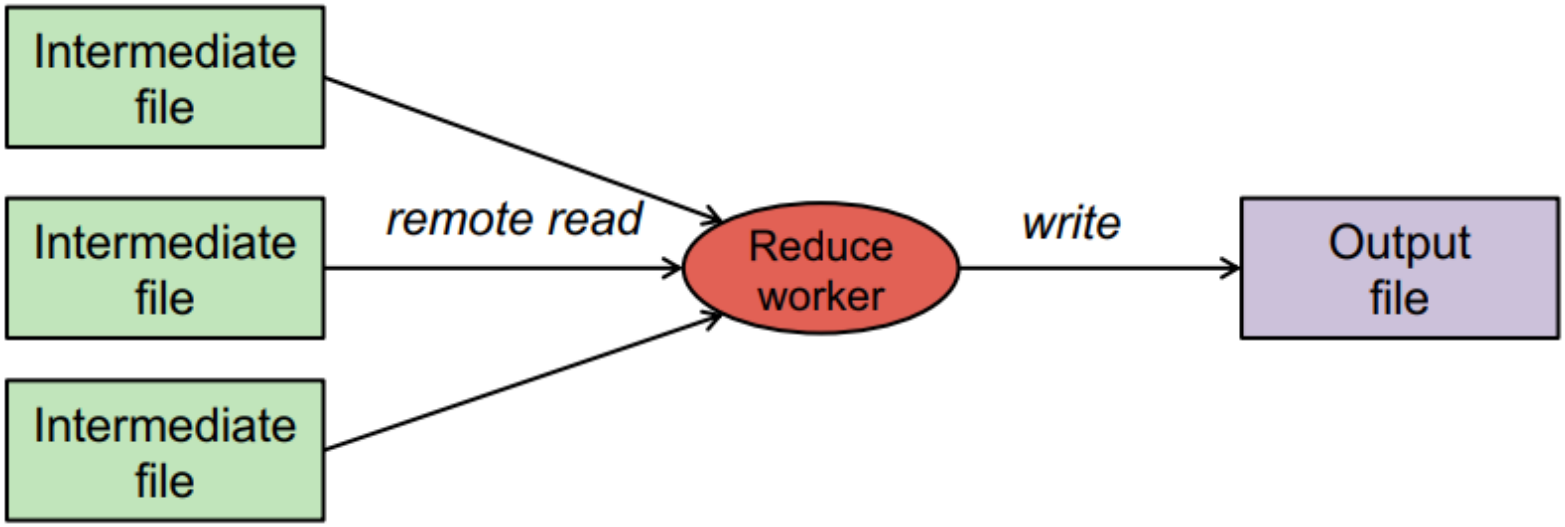
Step 5: sorting

- Reduce worker gets notified by the master about the location of intermediate files for its partition
- Read the data from the local disk of the map workers
- Sorts the data by the intermediate keys
- All pairs with the same key are grouped together



Step 6: run reduce tasks

- Pass the sorted and grouped pairs to the user-specified reduce function
- The output of reduce function is appended to an output file

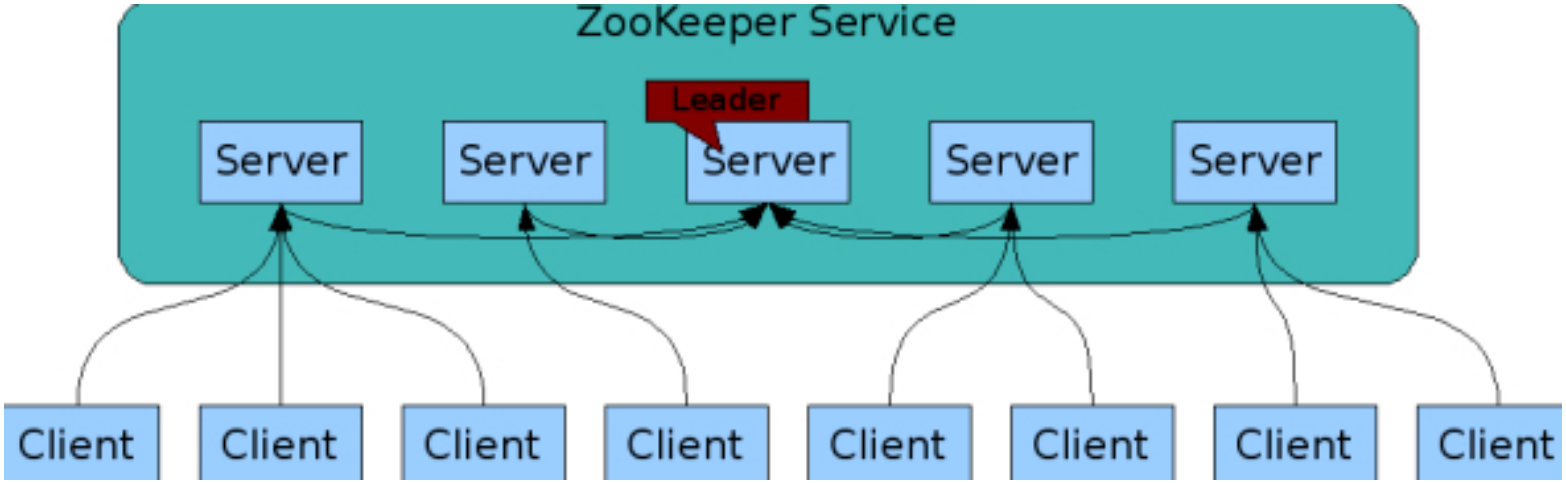


Step 7: return to user

- When all map and reduce tasks have completed, the master wakes up the user program
  - The output of MapReduce is saved in R files
- The MapReduce call in the user program returns

Coordinate

- A centralized service for MapReduce framework
  - Maintain configuration information of the framework
  - Provide distributed synchronization between processes
  - Guarantee the consistency of the framework
  - Apache Zookeeper, Google Chubby, etc.



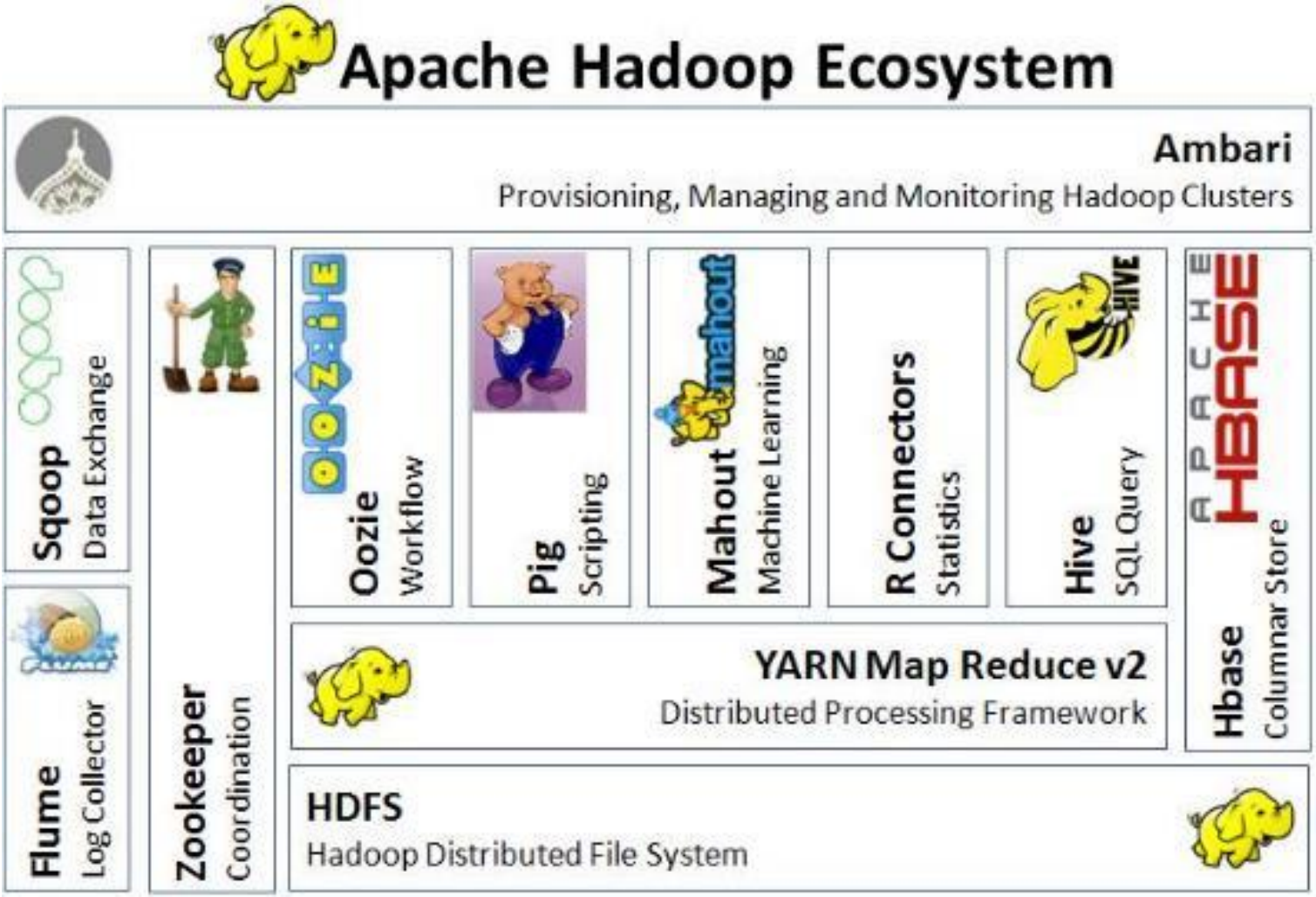
Fault tolerance

- Worker failure
  - Detect failure via periodic heartbeats
  - Master reschedules the tasks to other workers
  - Re-execute completed and in-progress map tasks
  - Re-execute in-progress reduce tasks
  - Task completion committed through master
- Master failure
  - Select a new master
  - Recover the master according to the checkpoint on distribute filesystem

MapReduce implementations

- Google has a proprietary implementation in C++
- Hadoop is an open-source implementation in Java
  - Developed by Yahoo, used in production
  - An Apache project
  - Rapidly expanding software ecosystem
- Lots of custom research implementations
  - For GPUs, FPGAs, etc.

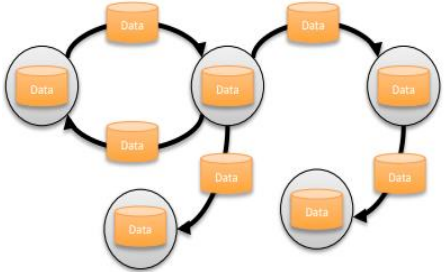
Ecosystem of Hadoop



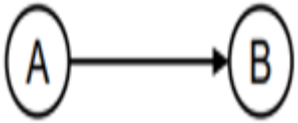
PROGRAMMING FOR GRAPH PROCESSING

What is graph?

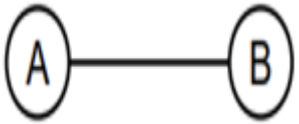
- A graph is a collection of vertices connected by edges (G = (V, E))



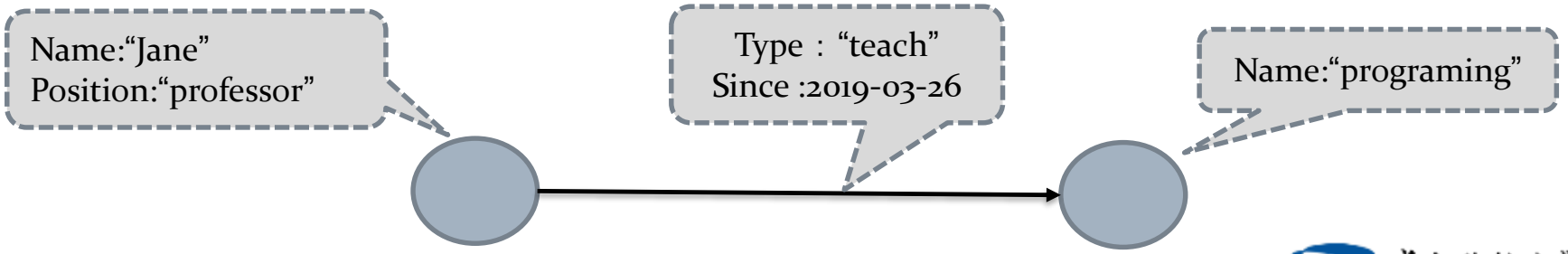
- Edge may be directed (from A to B)



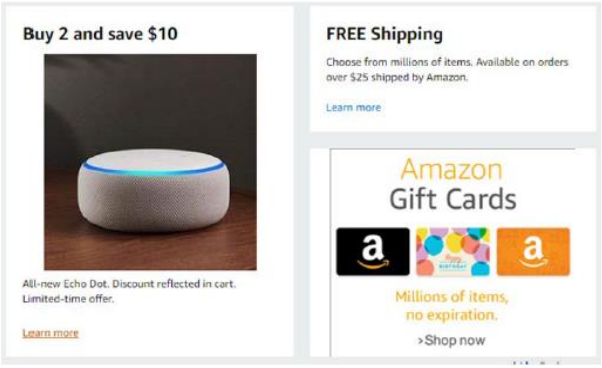
or undirected (between A and B)



- Vertices and edges may have properties



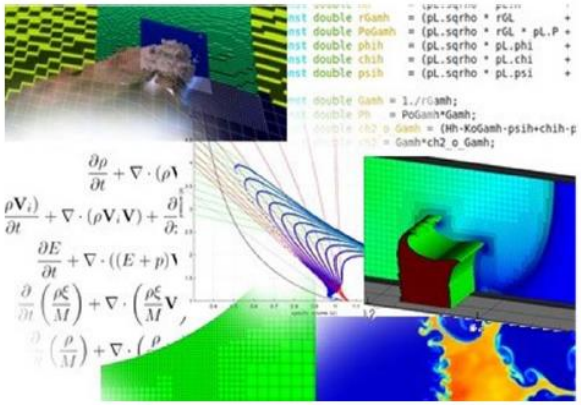
Graph processing is universal



advertising recommendation

social network analysis

network ranking



scientific calculation

Graph size is increasing

> 1 M vertices
> 100 M edges
*Distributed graphLab-2012

advertising network

> 1 B vertices
> 1 T edges
*Facebook Engineering Blog

social media

> 50 B vertices
> 1 T edges
*NSA Big Graph Experiment - 2013

internet

> 100 B vertices
> 100 T edges
*NSA Big graph Experiment - 2013

scientific calculation

Challenges of graph processing

Data-driven computation

- Computation for graph task is closely related to vertex and edge
- Computation structure is hard to predict before running, hard to parallel

Irregular structure

- Hard to divide the irregular structure of graph with good quality

Poor access locality

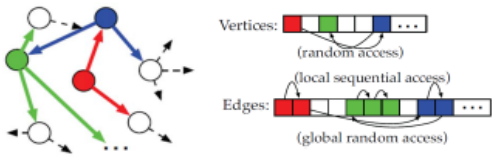
- Non-contiguous access for vertex/edge
- Existing caching mechanisms can only accelerate access with good locality

Access/computation ratio is high

- Large number of access lead to CPU waiting

Graph processing vs traditional processing

- **Compute intensive**
  Computation complexity is high, easy to cover memory latency
- **Sequential memory access**
  Date is stored in memory sequentially
- **High data parallelism**
  No complex dependency between different data, convenient for parallel processing
- **Good locality**
  With good spatial and time locality



- **Low computation/access ratio**
  Little computation in each node, hard to cover latency
- **Large random access**
  There is a large number of random access requests across the region
- **Complex data dependency**
  Difficult to explore parallelism, a lot of data conflicts
- **Unstructured distribution**
  Uneven distribution of vertex degree, serious load balancing problems and communication overhead

Graph algorithm

- **Traversal-centric algorithms**
  - Require a specific way to traverse the graph from a particular vertex
  - Have a large number of random access
- **Computation-centric algorithms**
  - A large number of operations in each iteration
  - All nodes participate in each iteration

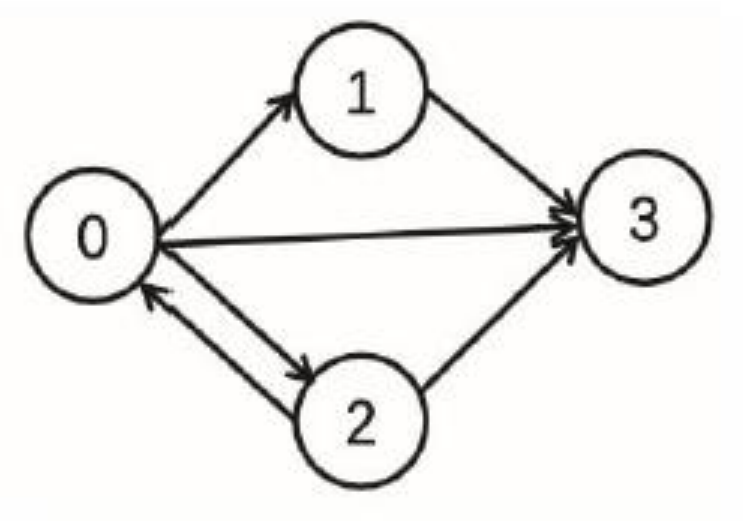
Traversal-centric algorithms

BFS	Dijkstra	Prim	SSSP	Betweenness Centrality	Radii
DFS	MST	Kruskal	Bellman Ford	Floyd Warshall	SPFA

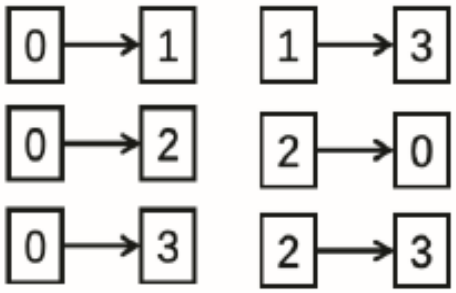
Computation-centric algorithms

A*搜索	PageRank	Connected Component	Ford Fulkerson
Union Find	Graph Color	MIS	Triangle Count

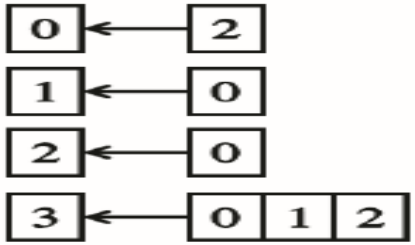
Graph storage



Graph Topology

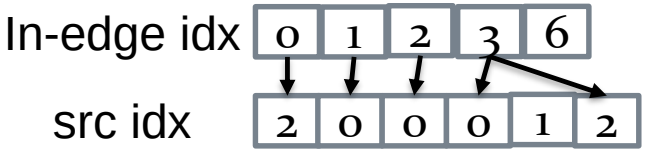


Edge list



Adjacent list

Access efficiency vs Compact ratio



CSR (Compressed Sparse Row)

Graph processing frameworks

Programming Model

Vertex Centric: Pregel[SIGMOD'10]
Edge Centric: X-Stream[SOSP'13]
Path Centric: PathGraph[SC'14]
Subgraph centric: Blogel[VLDB'14]

Graph Partition Strategy

Vertex Cut: Pregel[SIGMOD '2010]
Edge cut: PowerGraph[OSDI'12]
Hybrid Cut: PowerLyra[EuroSys '15]

Computation Strategy

Pull vs push

Synchronous vs Asynchronous

Typical System

Pregel : SIGMOD 2010
PowerGraph: OSDI 2012
GraphChi: OSDI 2012

Graph programming model

Programming model

- Easy and concise to use
- Flexible to express different graph tasks
- Efficient to fit in computation architecture

Exiting programming model

- Vertex-centric
- Edge-centric
- Path-centric
- Subgraph-centric

Programming model: vertex-centric

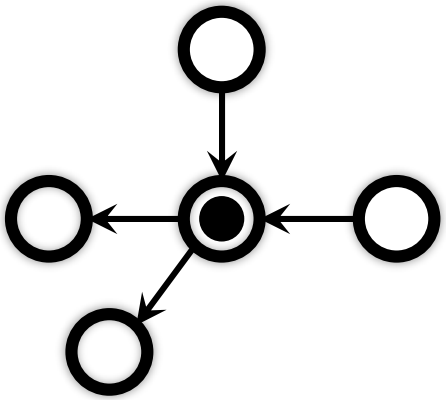
Coding graph algorithms as vertex-centric programs to process vertices in parallel and communicate along edges

-- "*Think as a Vertex*" philosophy

Example: *PageRank*

A centrality analysis algorithm to iteratively rank each vertex as a weighted sum of neighbors' ranks

$$R_i = 0.15 + 0.85 \sum_{(j,i) \in E} \omega_{ij} R_j$$



```
COMPUTE(v)
  foreach n in v.in_nbrs
    sum += n.rank / n.nedges
  v.rank = 0.15 + 0.85 * sum
  if !converged(v)
    foreach n in v.out_nbrs
      activate(n)
```

Programming model: vertex-centric

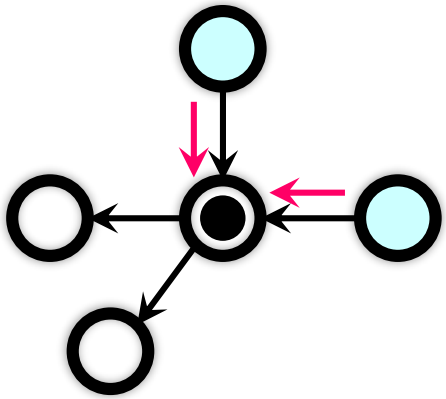
Coding graph algorithms as vertex-centric programs to process vertices in parallel and communicate along edges

-- "*Think as a Vertex*" philosophy

Example: *PageRank*

A centrality analysis algorithm to iteratively rank each vertex as a weighted sum of neighbors' ranks

$$R_i = 0.15 + 0.85 \sum_{(j,i) \in E} \omega_{ij} R_j$$



Gather data from neighbors via in-edges

```
COMPUTE(v)
  foreach n in v.in_nbrs
    sum += n.rank / n.nedges
  v.rank = 0.15 + 0.85 * sum
  if !converged(v)
    foreach n in v.out_nbrs
      activate(n)
```

Programming model: vertex-centric

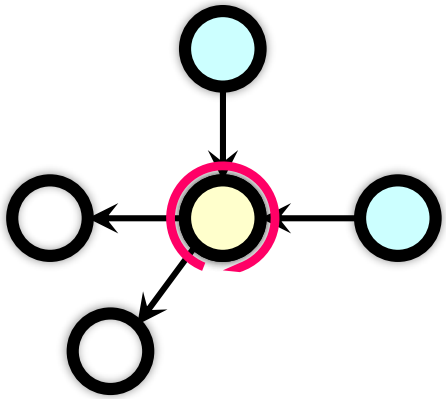
Coding graph algorithms as vertex-centric programs to process vertices in parallel and communicate along edges

-- "*Think as a Vertex*" philosophy

Example: *PageRank*

A centrality analysis algorithm to iteratively rank each vertex as a weighted sum of neighbors' ranks

$$R_i = 0.15 + 0.85 \sum_{(j,i) \in E} \omega_{ij} R_j$$



Update vertex with new data

```
COMPUTE(v)
  foreach n in v.in_nbrs
    sum += n.rank / n.nedges
  v.rank = 0.15 + 0.85 * sum
  if !converged(v)
    foreach n in v.out_nbrs
      activate(n)
```

Programming model: vertex-centric

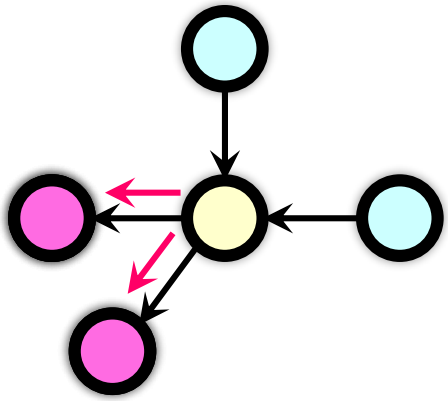
Coding graph algorithms as vertex-centric programs to process vertices in parallel and communicate along edges

-- "*Think as a Vertex*" philosophy

Example: *PageRank*

A centrality analysis algorithm to iteratively rank each vertex as a weighted sum of neighbors' ranks

$$R_i = 0.15 + 0.85 \sum_{(j,i) \in E} \omega_{ij} R_j$$



Scatter data to neighbors via out-edges

```
COMPUTE(v)
  foreach n in v.in_nbrs
    sum += n.rank / n.nedges
  v.rank = 0.15 + 0.85 * sum
  if !converged(v)
    foreach n in v.out_nbrs
      activate(n)
```

Programming model: edge-centric

Coding graph algorithms as edge-centric programs to process edges in parallel, where edge is the minimum parallel processing unit.

-- "*Think as an Edge*" philosophy

```
for each vertex v                    Scatter
  if v has update
    for each edge e from v
        scatter update along e
```

Vertex-Centric

Transformation

```
for each edge e                      Scatter
    If e.src has update
        scatter update along e
```

Edge-Centric

--Balanced load for fine-grained parallel processing unit
--Better locality for sequential access of the edges

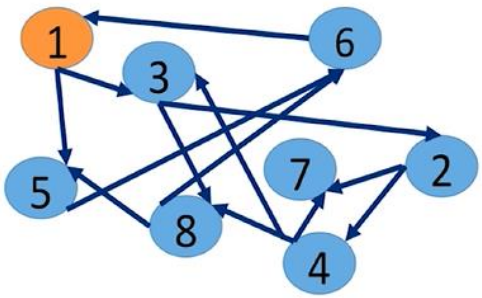
Programming model: edge-centric

Coding graph algorithms as edge-centric programs to process edges in parallel, where edge is the minimum parallel processing unit.

-- "*Think as an Edge*" philosophy

Example:BFS

Edge-Centric Scatter Gather



BFS

V
1
2
3
4
5
6
7
8

SOURCE	DEST
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

Edge-Centric Scatter Gather for BFS

Edge-centric Scatter-Gather

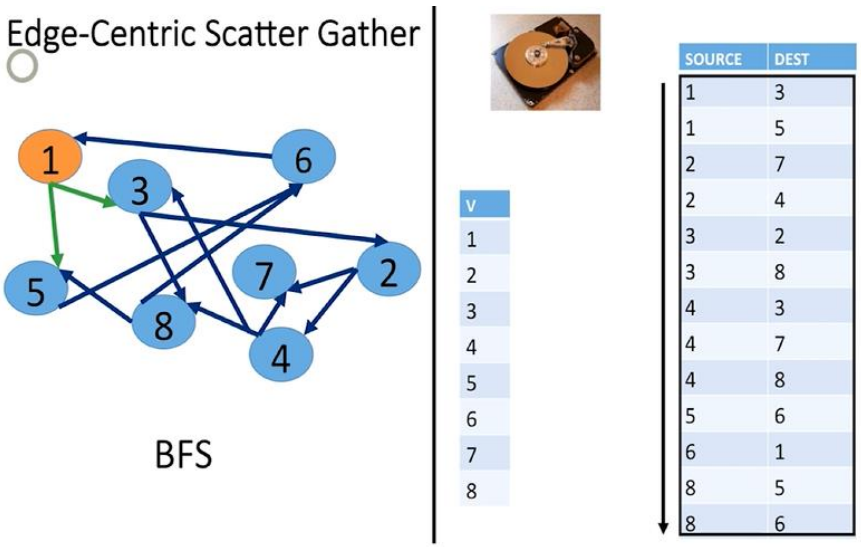
```
edge_scatter(edge e)
 send update over e

update_gather(update u)
 apply update u to u.destination

while not done
  for all edges e
    edge_scatter(e)
  for all updates u
    update_gather(u)
```

Programming model: edge-centric

Coding graph algorithms as edge-centric programs to process edges in parallel, where edge is the minimum parallel processing unit.

-- "*Think as an Edge*" philosophy

Example:BFS



Edge-Centric Scatter Gather for BFS

Edge-centric Scatter-Gather

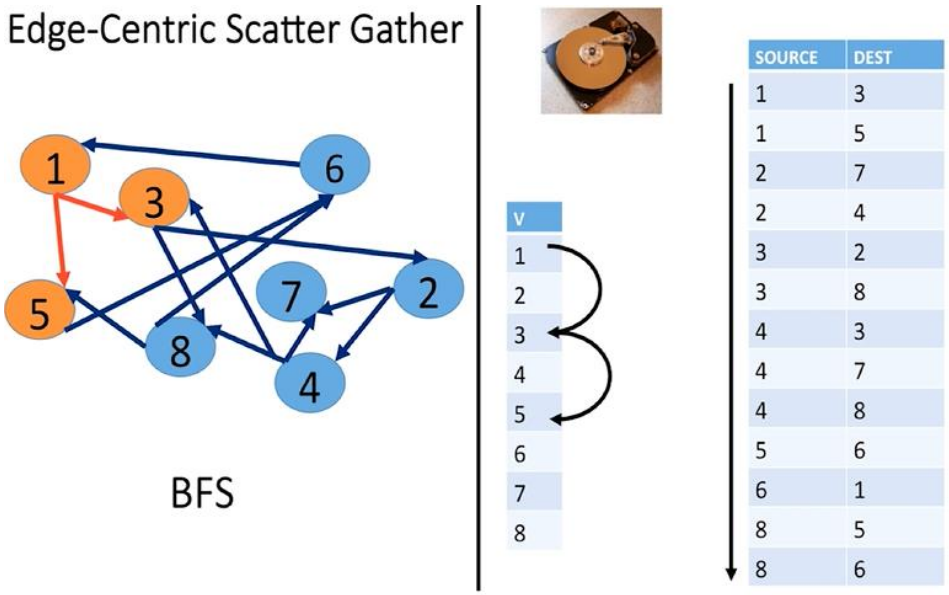
```
edge_scatter(edge e)
 send update over e

update_gather(update u)
 apply update u to u.destination

while not done
  for all edges e
    edge_scatter(e)
  for all updates u
    update_gather(u)
```

Programming model: edge-centric

Coding graph algorithms as edge-centric programs to process edges in parallel, where edge is the minimum parallel processing unit.

-- "*Think as an Edge*" philosophy

Example:BFS



Edge-Centric Scatter Gather for BFS

Edge-centric Scatter-Gather

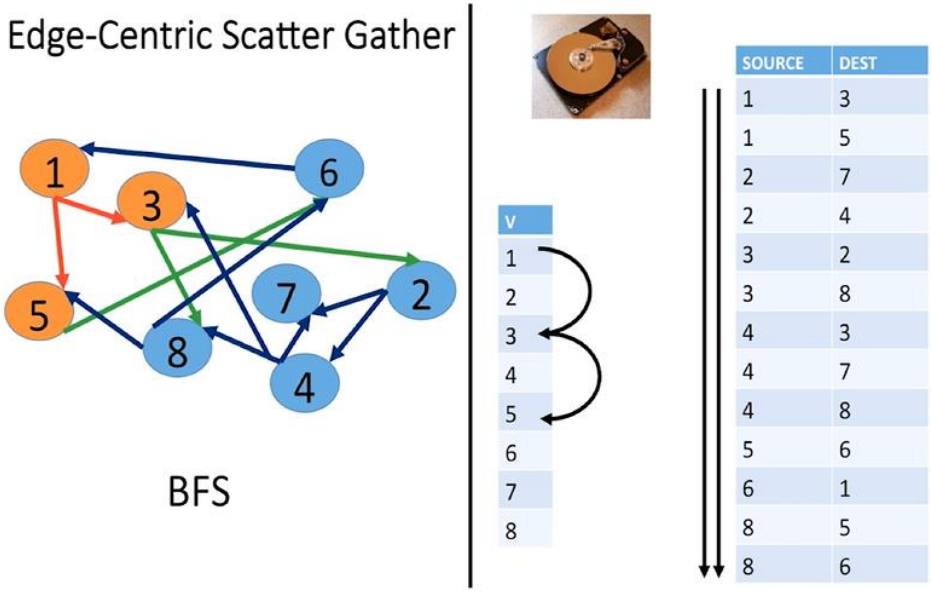
```
edge_scatter(edge e)
 send update over e

update_gather(update u)
 apply update u to u.destination

while not done
  for all edges e
    edge_scatter(e)
  for all updates u
    update_gather(u)
```

Programming model: edge-centric

Coding graph algorithms as edge-centric programs to process edges in parallel, where edge is the minimum parallel processing unit.

-- "*Think as an Edge*" philosophy

Example:BFS



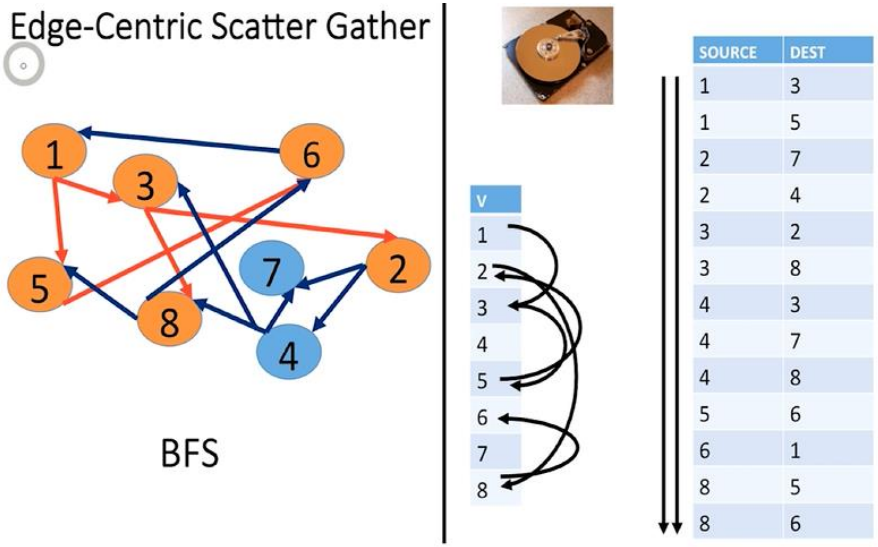
Edge-Centric Scatter Gather for BFS

Programming model: edge-centric

Coding graph algorithms as edge-centric programs to process edges in parallel, where edge is the minimum parallel processing unit.

-- "*Think as an Edge*" philosophy

Example:BFS

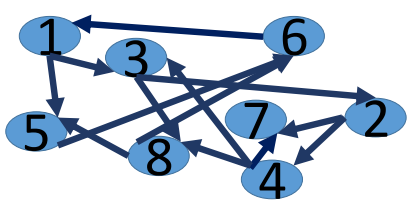


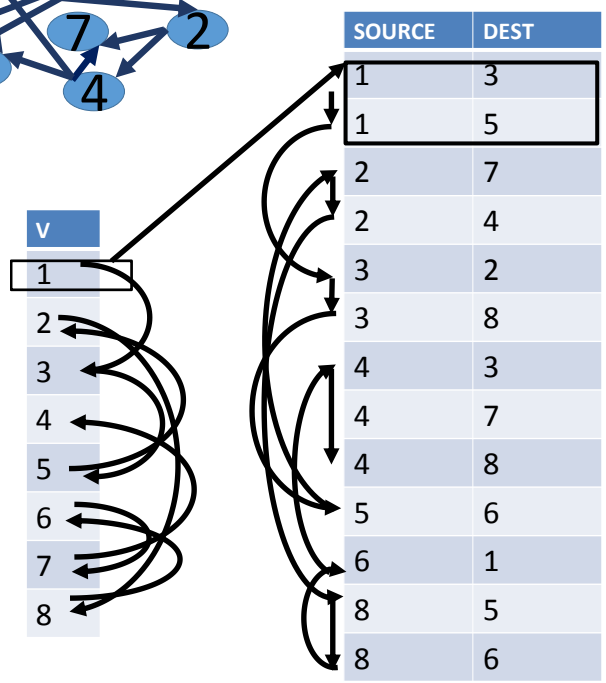
Edge-Centric Scatter Gather for BFS

Programming model: edge-centric

Coding graph algorithms as edge-centric programs to process edges in parallel, where edge is the minimum parallel processing unit.

-- "*Think as an Edge*" philosophy





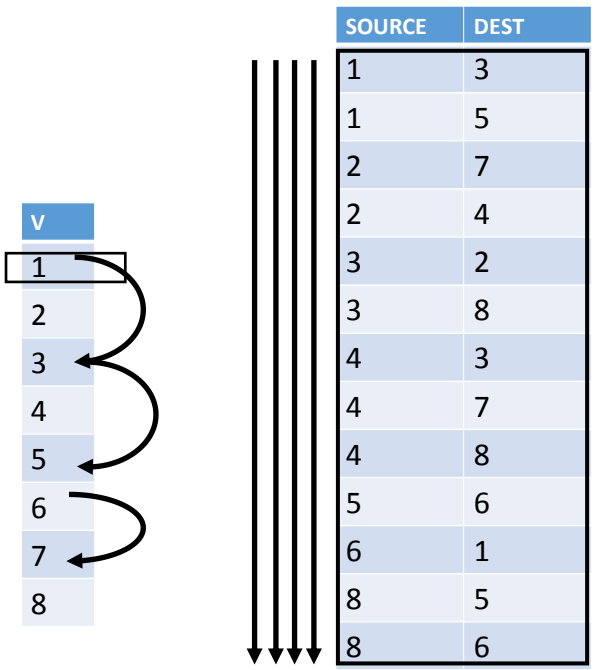
V
1
2
3
4
5
6
7
8

SOURCE	DEST
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

Random accesses over edges

Vertex-Centric Scatter Gather for BFS

Transformation



V
1
2
3
4
5
6
7
8

SOURCE	DEST
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

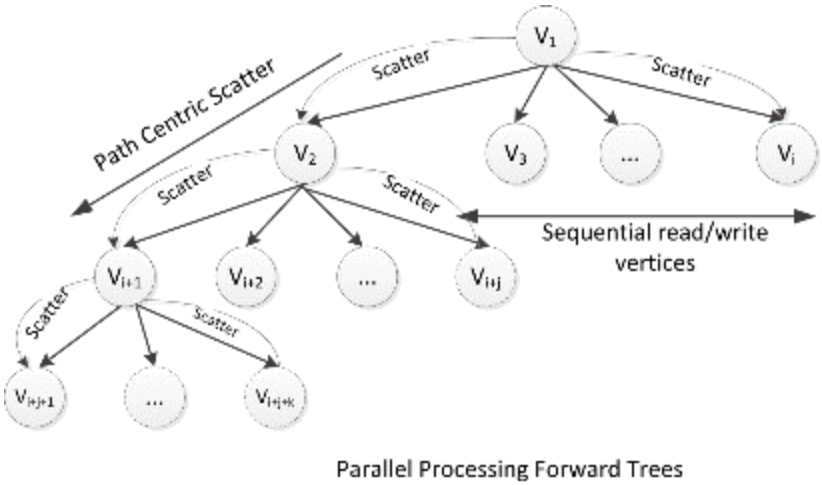
Sequential accesses over edges

Edge-Centric Scatter Gather for BFS

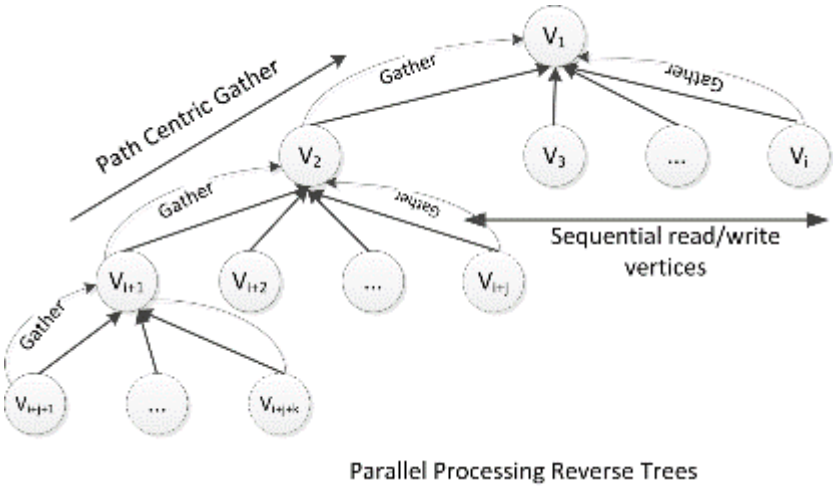
Programming model: path-centric

Computation model

- Scatter or gather: faster method
- Process graph following paths: improve locality



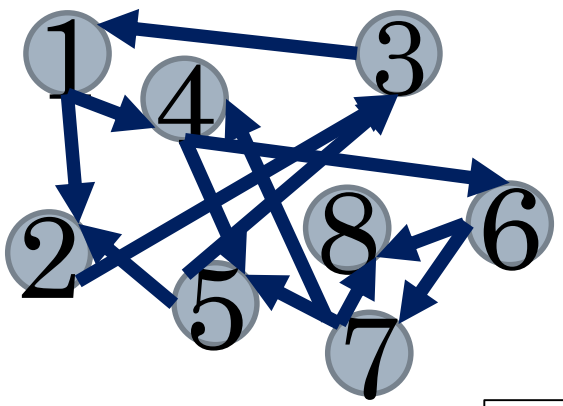
Path Centric Scatter

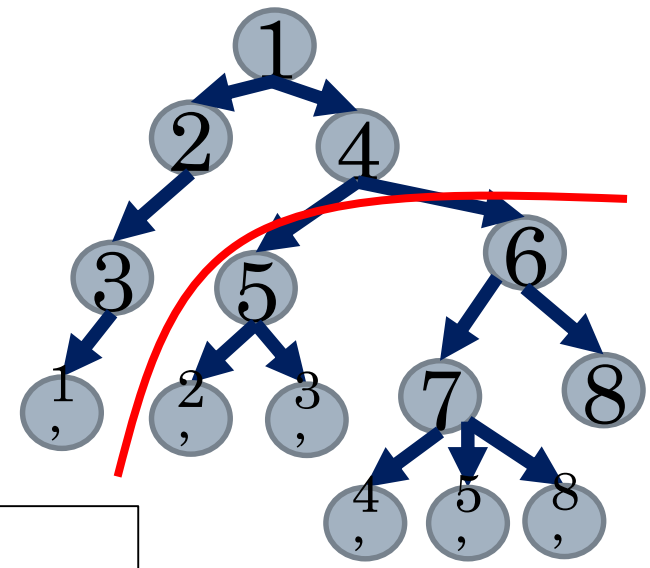


Path Centric Gather

Programming model: path-centric

- In order to achieve it, generate edge travel tree for each graph, then partition tree into multiple parts
  - Each partition with forward-visited tree and backward-tree
  - Each partition can parallelly executed





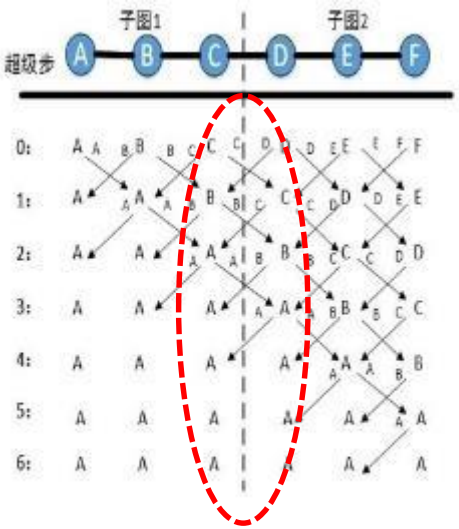

```
1: for each iteration do
        2: parfor each p of Partitions do
        3:      Gather(p);
        4: end parfor
        5: sync;
6: end for
```

Programming model: subgraph-centric

Coding graph algorithms as subgraph-centric programs to process vertices/edges via taking subgraph as the minimum parallel processing unit.

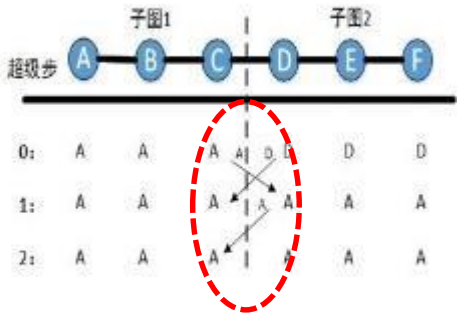
-- "*Think as a Subgraph*" philosophy

- First, it divides the graph data into different subgraphs
- Then, it updates all vertices of each subgraph until the subgraph converges
- Finally, it transmits the updated state information of the subgraph to other subgraphs



Many high cost global synchronization operations

Vertex-Centric Scatter Gather for CC



Less global synchronization operations

Subgraph-centric Scatter Gather for CC

--It not only ensures better data locality, but also can transform many high cost global synchronization operations to low overhead local ones.

- The status information can only be propagated one-hop in each super-step for vertex/edge centric
- 7 super-step to converge for vertex/edge-centric
- 3 super-step to converge for subgraph-centric

Programming model: subgraph-centric

Coding graph algorithms as subgraph-centric programs to process vertices/edges via taking subgraph as the minimum parallel processing unit.

-- "*Think as a Subgraph*" philosophy

- First, it divides the graph data into different subgraphs

examples of subgraphs

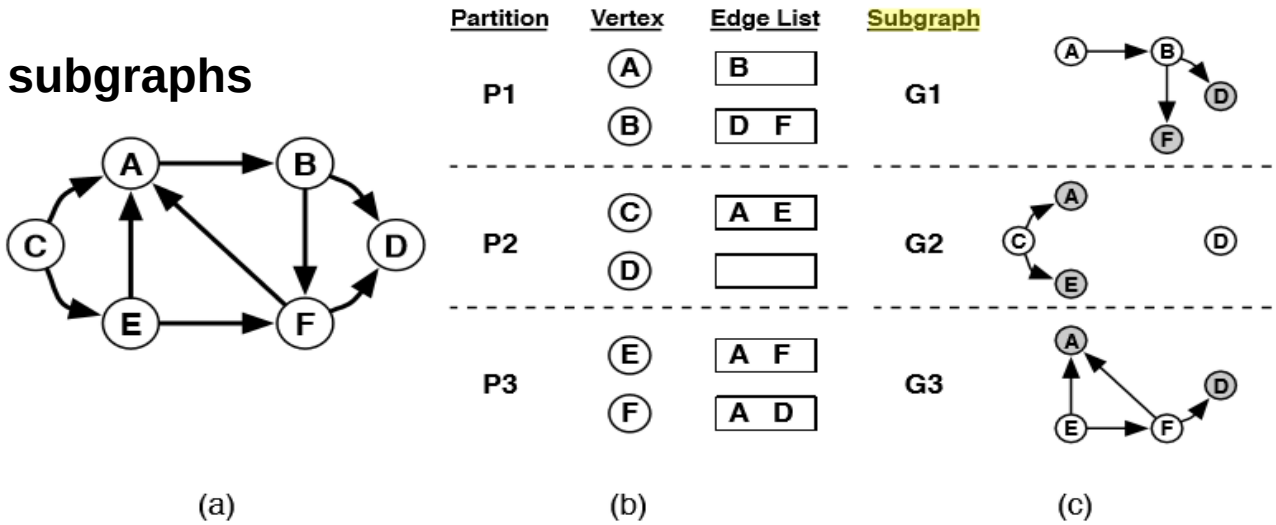


Figure (a) shows the original graph.

Figure (b) divides the set of vertices in the original graph into partitions.

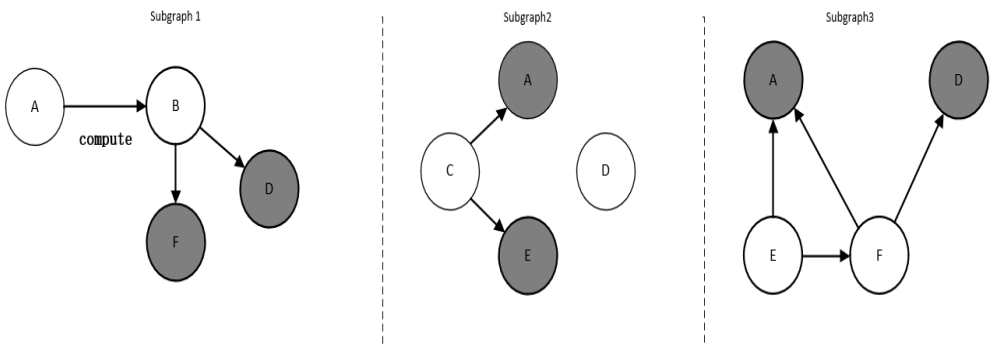
Figure (c) shows examples of subgraphs.

Programming model: subgraph-centric

Coding graph algorithms as subgraph-centric programs to process vertices/edges via taking subgraph as the minimum parallel processing unit.

-- "*Think as a Subgraph*" philosophy

- Then, it updates all vertices of each subgraph until the subgraph converges.



TLASG model

TLASG model

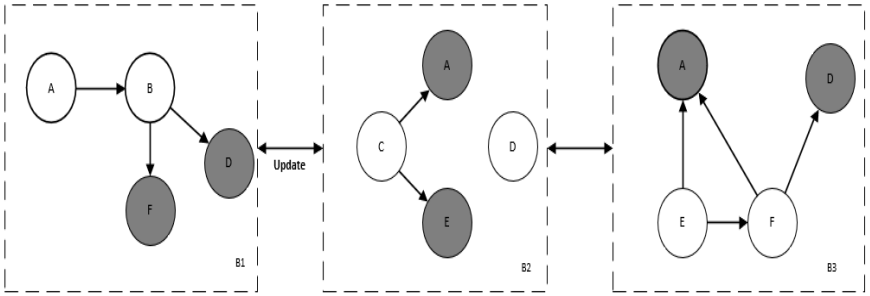
```
Compute(vertex v)
 update all vertices of subgraph

block_update(subgraph subg)
  while not done
    for all vertices v that have   updates
      compute(vertex v)
  apply updates from inbound edges of subgraph

while  not  done
  for all subg that have updates
    block_upddate(subg)
```

Programming model: subgraph-centric

Coding graph algorithms as subgraph-centric programs to process vertices/edges via taking subgraph as the minimum parallel processing unit.

-- "*Think as a Subgraph*" philosophy

- Finally, it transmits the updated state information of the subgraph to other subgraphs.



TLASG model

```
Compute(vertex v)
 update all vertices of subgraph

block_update(subgraph subg)
  while not done
    for all vertices v that have   updates
      compute(vertex v)
  apply updates from inbound edges of subgraph

while not done
  for all subg that have updates
    block_upddate(subg)
```

Graph processing frameworks

Programming Model

Vertex Centric: Pregel[SIGMOD'10]
Edge Centric: X-Stream[SOSP'13]
Path Centric: PathGraph[SC'14]
Subgraph centric: Blogel[VLDB'14]

Graph Partition Strategy

Vertex Cut: Pregel[SIGMOD '2010]
Edge cut: PowerGraph[OSDI'12]
Hybrid Cut: PowerLyra[EuroSys '15]

Computation Strategy

Pull vs push

Synchronous vs Asynchronous

Typical System

Pregel : SIGMOD 2010
PowerGraph: OSDI 2012
GraphChi: OSDI 2012

Graph partition

Graph is too large to hold in single node memory.

Goal of graph partition

- Balance workload between machines
- Minimize communication
- Maximize the computation efficiency within each machine

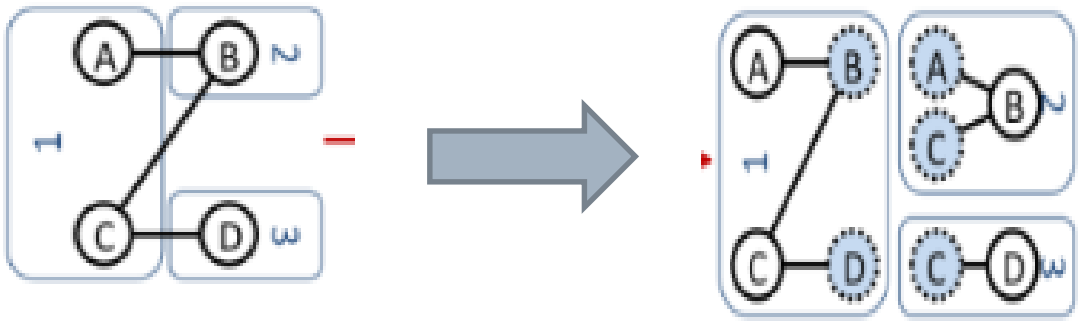
Graph partition method

- Edge-cut
- Vertex-cut
- Hybrid-cut

Graph partition: edge-cut

Pregel[SIGMOD'08] and **GraphLab**[VLDB'12]

- Focus on exploiting locality
- Partitioning: use edge-cut to evenly assign vertices along with all edges
- Computation: aggregate all resources (i.e., messages or replicas) of a vertex on local machine



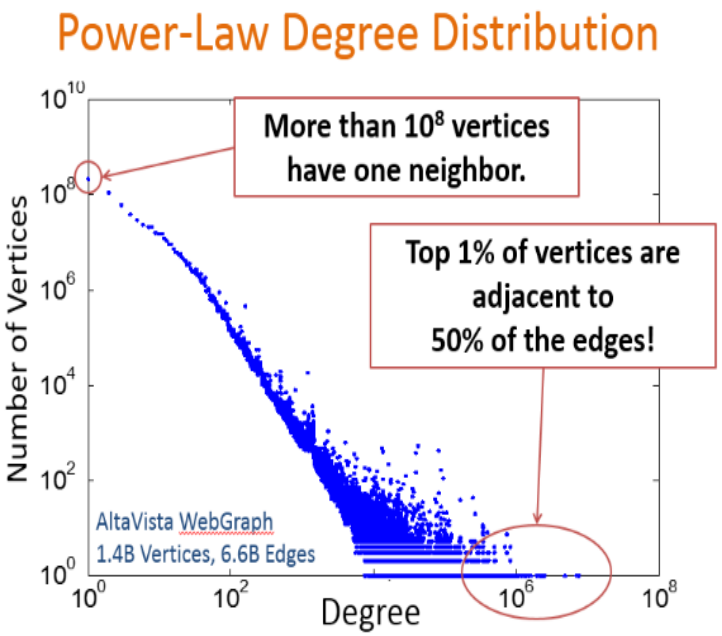
Graph partition: edge-cut

Natural Graphs are Skewed

Graphs in real-world

- *Power-law* degree distribution (aka Zipf's law or Pareto)

"most vertices have relatively few neighbors while a few have many neighbors" -- *PowerGraph*[OSDI'12]



Case: SINA *Weibo*

- 167 million active users

#1 

#2 

#100 

#Follower : 77,971,093 76,946,782 ··· 14,723,818 ····· 397 ····· 69 ·····

Disadvantages of edge-cut

Load balance

- Work imbalance for high degree vertices, as computation, storage, and communication are linear with the degree of vertices

Difficult to partition

- Natural graphs are difficult to partition to minimize the communication, and maximize work balance

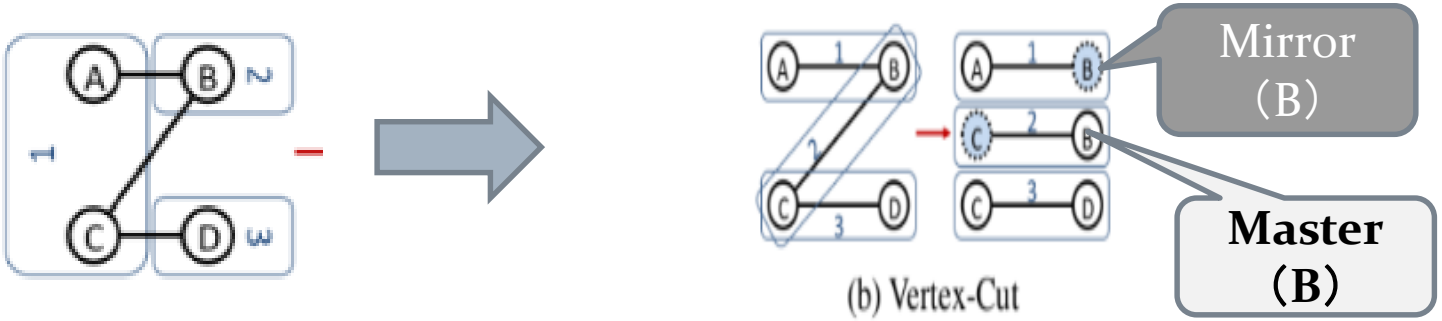
Limited parallelism

- No parallelism possible within individual vertices

Graph partition: vertex-cut

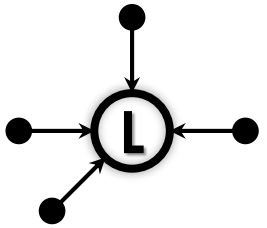
PowerGraph [OSDI'12] and **GraphX**[OSDI'14]

- Focus on exploiting parallelism
- Partitioning: use vertex-cut to evenly assign edges with replicated vertices
- Computation: decompose the workload of a vertex into multiple machines



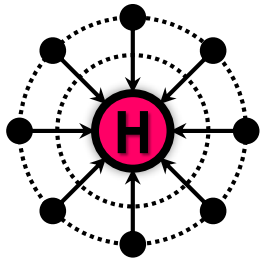
Is vertex–cut perfect for power-law graph ?

Low-degree vertex



397

High-degree vertex



77,971,093

Locality

make resource locally accessible to hidden network latency

⇦ Conflict ⇨

Parallelism

evenly parallelize the workloads to avoid load imbalance

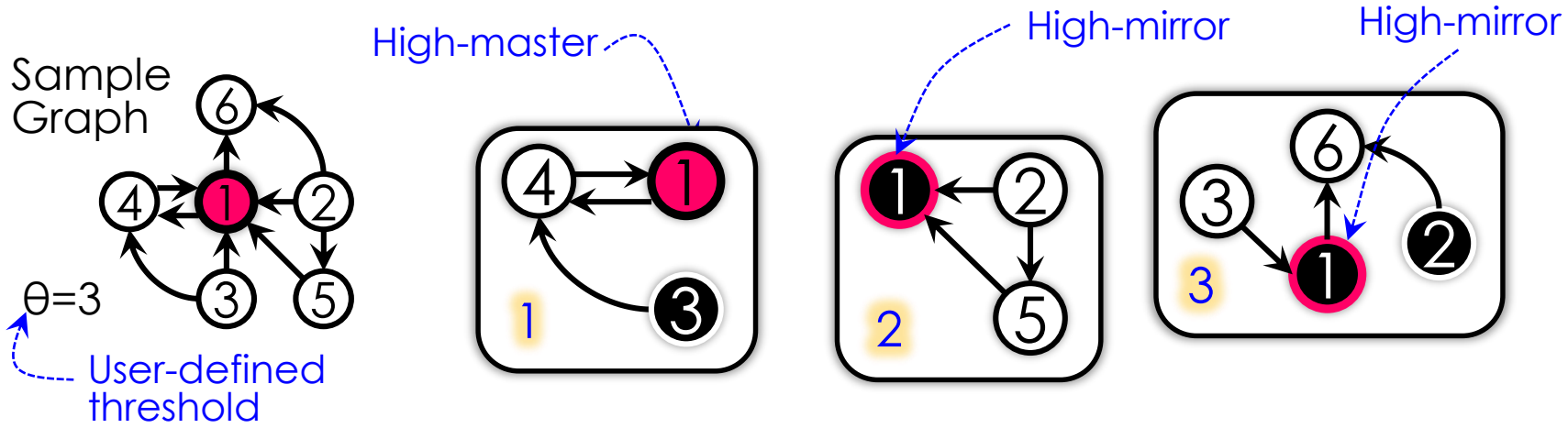
100-million users each with 100 followers

100 users each with 100-million followers

Graph partition: hybrid-cut

Hybrid-cut: differentiate the graph partitioning for low-degree and high-degree vertices

- Low-cut (inspired by *edge-cut*): reduce mirrors and exploit locality for low-degree vertices
- High-cut (inspired by *vertex-cut*): provide balance and restrict the impact of high-degree vertices
- Efficiently combine low-cut and high-cut



Graph processing frameworks

Programming Model

Vertex Centric: Pregel[SIGMOD'10]
Edge Centric: X-Stream[SOSP'13]
Path Centric: PathGraph[SC'14]
Subgraph centric: Blogel[VLDB'14]

Graph Partition Strategy

Vertex Cut: Pregel[SIGMOD '2010]
Edge cut: PowerGraph[OSDI'12]
Hybrid Cut: PowerLyra[EuroSys '15]

Computation Strategy

Pull vs push

Synchronous vs Asynchronous

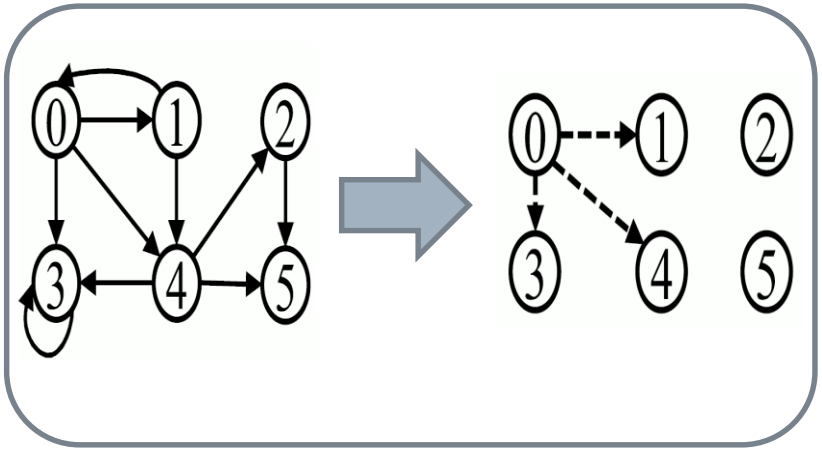
Typical System

Pregel : SIGMOD 2010
PowerGraph: OSDI 2012
GraphChi: OSDI 2012

Message passing: push model

Push updates to the neighbors

- Selectively schedule to reduce computation
- Lead to the write-write conflict on a destination vertex from multiple sources




```
parallel_for (int vSrc = 0; vSrc < numVertices; ++vSrc) {
    if (!frontier.contains(vSrc)) continue;
    for (int d = 0; d < vertex[vSrc].outdegree; ++d) {
        const int vDst = vertex[vSrc].outneighbor[d];
        if (converged.contains(vDst)) continue;
        atomicCAS(vertex[vDst].value,
            compute(vertex[vSrc].value, vertex[vDst].value)); } }
```

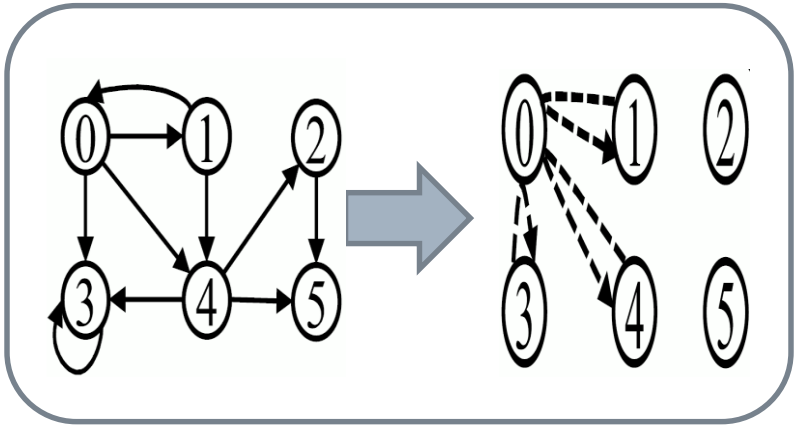
Push-based message implementation

Ref. Genimi (OSDI 2016) , Grazella (PPoPP 2018)

Message passing: pull model

Get update from incoming neighbors

- Do not have the write-write conflict
- Scan all incoming neighbors can incur redundant computation since some nodes may not active



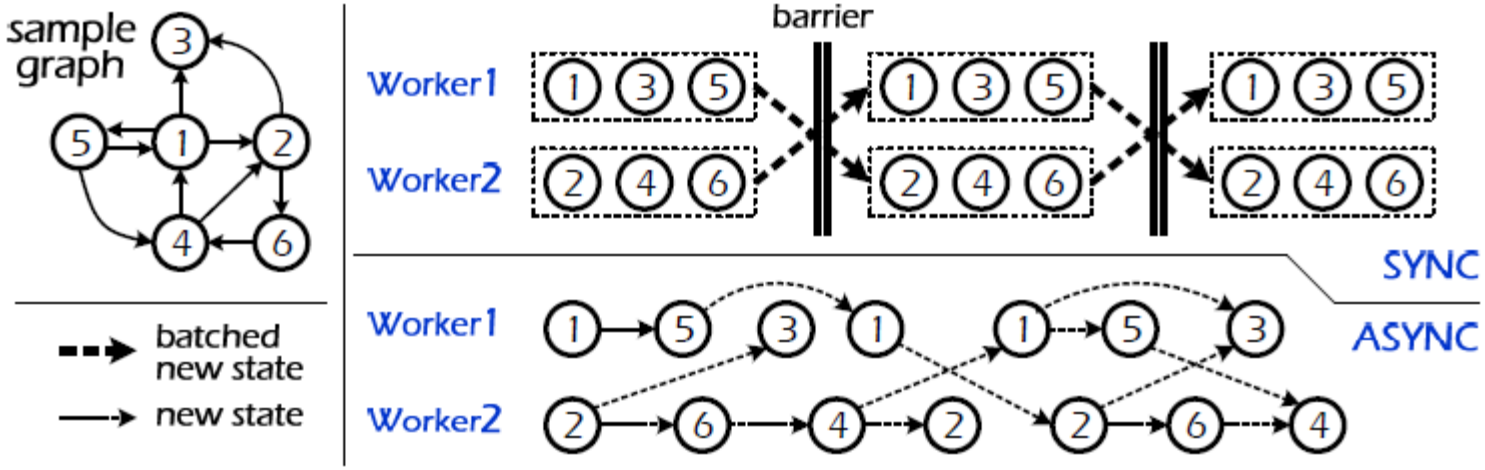

```
parallel_for (int vDst = 0; vDst < numVertices; ++vDst) {
    if (converged.contains(vDst)) continue;
    for (int s = 0; s < vertex[vDst].indegree; ++s) {
        const int vSrc = vertex[vDst].inneighbor[s];
        if (!frontier.contains(vSrc)) continue;
        vertex[vDst].value =
            compute(vertex[vSrc].value, vertex[vDst].value); } }
```

Pull-based message implementation

Ref. Genimi (OSDI 2016), Grazella (PPoPP 2018)

Schedule model

Synchronous vs Asynchronous

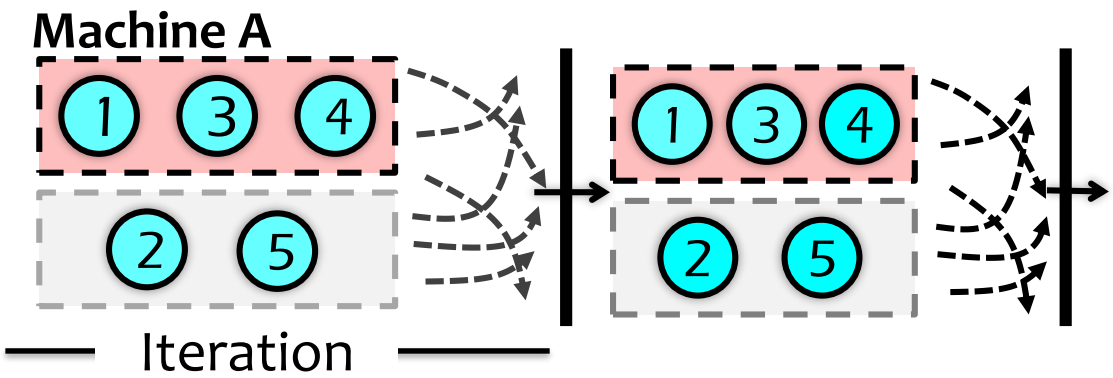


	Sync	*Async*
Properties		
Communication	Regular	Irregular
Convergence	Slow	Fast
Favorites		
Algorithm	I/O-intensive	CPU-intensive
Execution Stage	High Workload	Low Workload
Scalability	Graph Size	Cluster Size

Ref. **PoweSwitch PPoPP 2015**

Schedule model: sync model

Scheduling:



Current iteration active vertex

next iteration active vertex

Pseudocode

while (*iteration* ≤ *max*) *do*

if V_a== Ø then break

$V'_a \leftarrow$ Ø

foreach v ∈ V_a do

A ← **compute**(v)

$V'_a \leftarrow V'_a \cup A$

barrier to update

$V_a \leftarrow V'_a$

iteration ++

Ref. **PoweSwitch PPoPP 2015**

Schedule model: sync model

Advantages of synchronous model

1. Batching massages which improves of network utilization
2. Better for algorithm with much communication needed
3. With light computation for each node

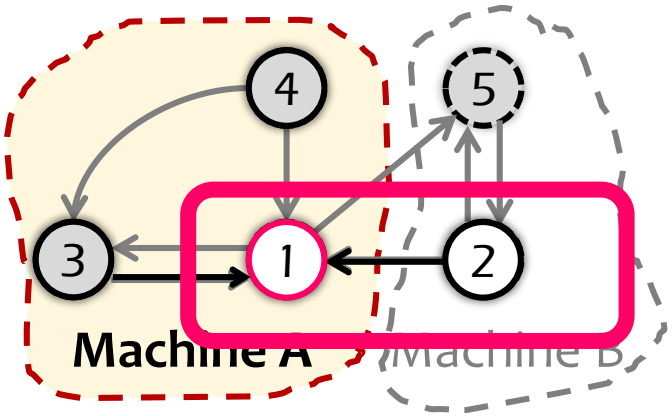
Disadvantages of synchronous model

1. Uneven convergence problem
2. Straggler problem
3. Not suitable for some algorithms, eg. Graph Coloring and Clustering based on Gibbs Sampling

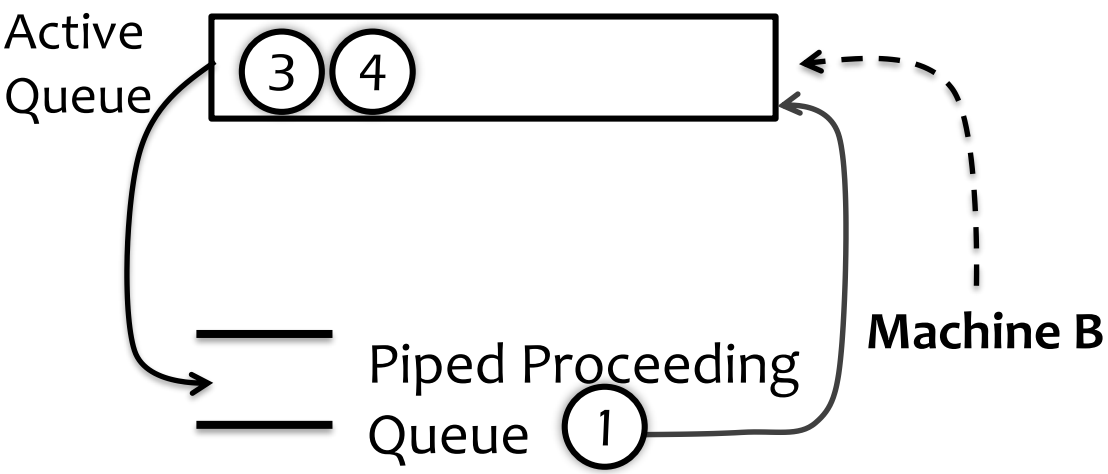
Ref. **PoweSwitch PPoPP 2015**

Schedule model: async model

Scheduling:



Internal State (e.g. Machine A):



Pseudocode

while (V_a != ∅) *do*

- v = **dequeue**(V_a)
- A ← **compute**(v)
- $V'_a \leftarrow V'_a \cup A$
- **signal across machines**

Ref. **PoweSwitch PPOPP 2015**

Schedule model: async model

Advantages of asynchronous

1. Speedup convergence
2. Better for CPU-bound algorithm

Disadvantages of asynchronous

1. Lock cost caused by vertex contention
2. Asynchronous model sends messages frequently, and CPU frequently encapsulates TCP/IP packets, which wastes the utilization of CPU

Ref. **PoweSwitch PPoPP 2015**

Graph processing frameworks

Programming Model

Vertex Centric: Pregel[SIGMOD'10]
Edge Centric: X-Stream[SOSP'13]
Path Centric: PathGraph[SC'14]
Subgraph centric: Blogel[VLDB'14]

Graph Partition Strategy

Vertex Cut: Pregel[SIGMOD '2010]
Edge cut: PowerGraph[OSDI'12]
Hybrid Cut: PowerLyra[EuroSys '15]

Computation Strategy

Pull vs push

Synchronous vs Asynchronous

Typical System

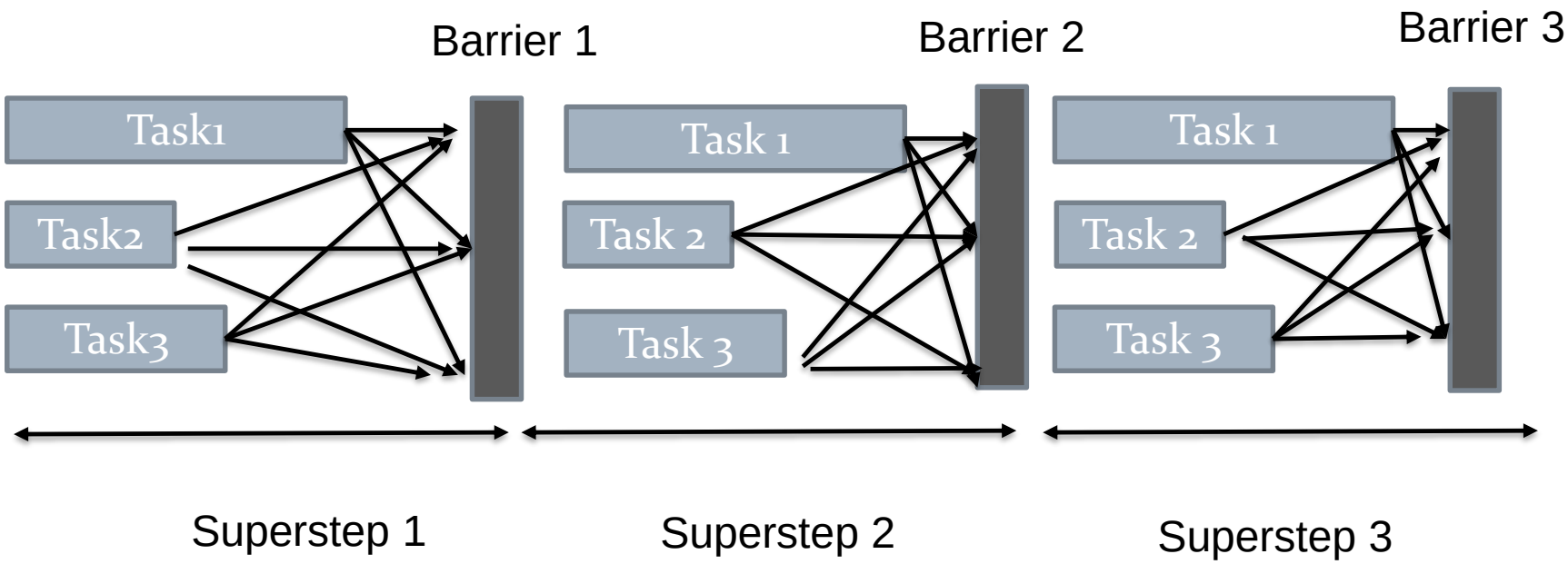
Pregel : SIGMOD 2010
PowerGraph: OSDI 2012
GraphChi: OSDI 2012

Graph processing system

- **In-memory single-node graph system**
  - Ligra, GraphMat, Polymer
- **Out-of core single-node graph system**
  - GraphChi, TurboGraph, X-Stream, PathGraph, GridGraph, FlashGraph
- **Distributed graph system**
  - Pregel, PowerGraph, GraphLab, GraphX, PowerSwitch, PowerLyra, Gemini
- **Graph system on accelerator**
  - GPU based, e.g. CuSha, WS, Gunrock, IrGL, Groute
  - FPGA based
  - ASIC based
  - NVM based

Pregel

- Large scale graph-parallel processing platform developed by Google
- Utilize BSP (bulk synchronous parallel) model

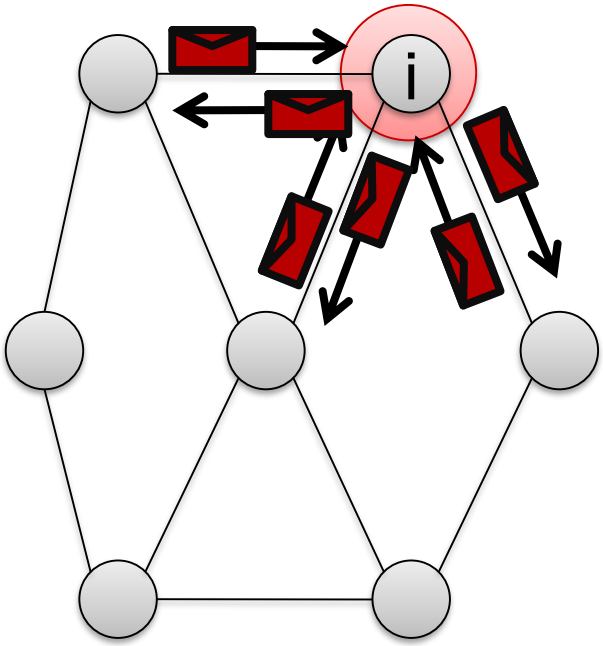


BSP model

Pregel

- Large scale graph-parallel processing platform developed by Google
- Utilize BSP (bulk synchronous parallel) model
- Vertex-Programs interact by sending messages

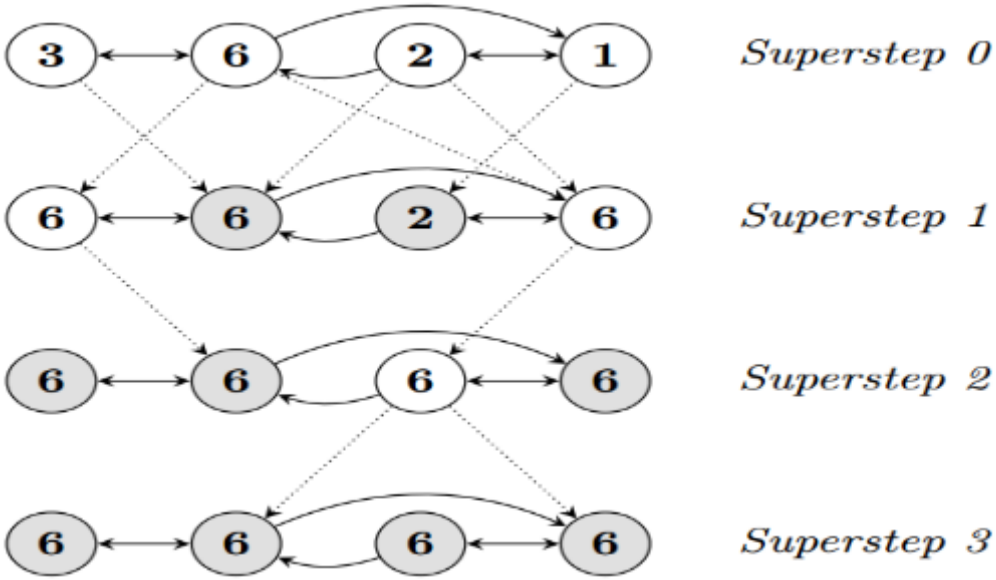
```
Pregel_PageRank(i, messages) :
  // Receive all the messages
  total = 0
  foreach( msg in messages) :
    total = total + msg

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

  // Send new messages to neighbors
  foreach(j in out_neighbors[i]) :
    Send  msg(R[i] * w_ij) to vertex j
```



Ref : Malewicz et al. [PODC'09, SIGMOD'10]

Pregel

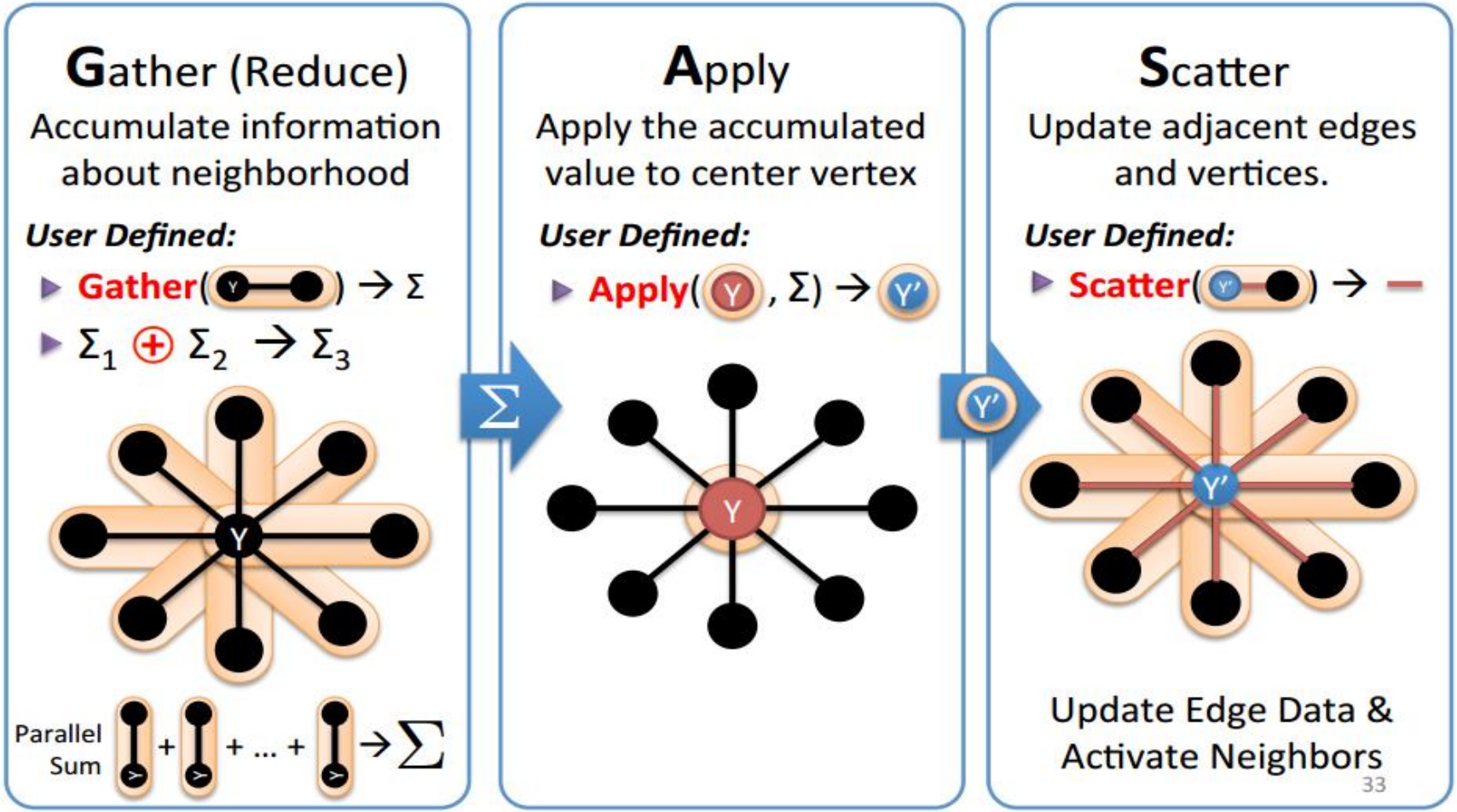
- Large scale graph-parallel processing platform developed by Google
- Utilize BSP (bulk synchronous parallel) model
- Vertex-Programs interact by sending messages.
- Iteration until convergence



Maximum value example for Pregel

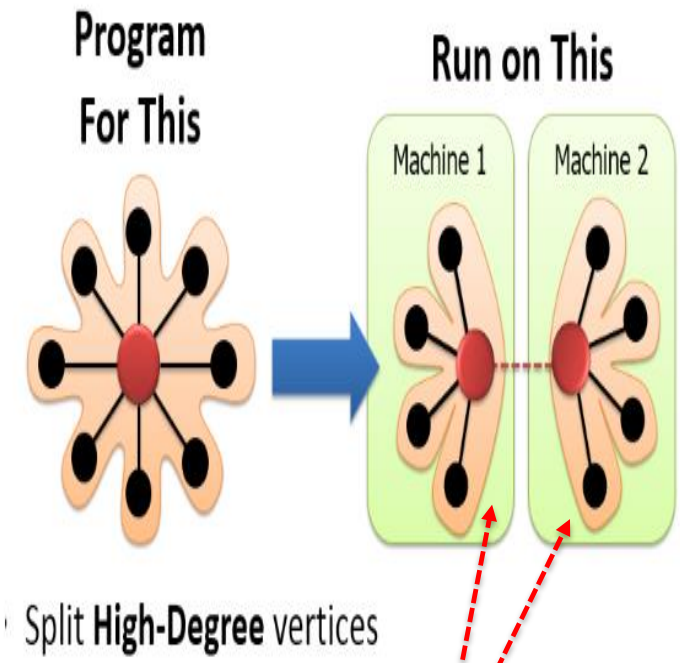
PowerGraph

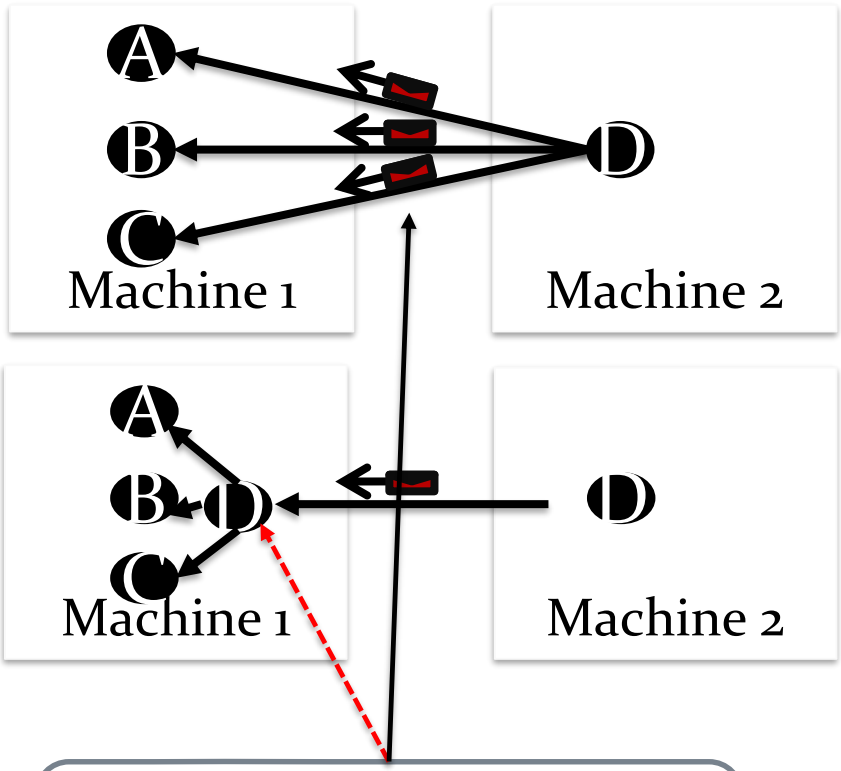
Introduce GAS programming model



PowerGraph

Edge-cut for power-law graph





Cut high-degree vertex can balance workload

Image node can reduce communication between node

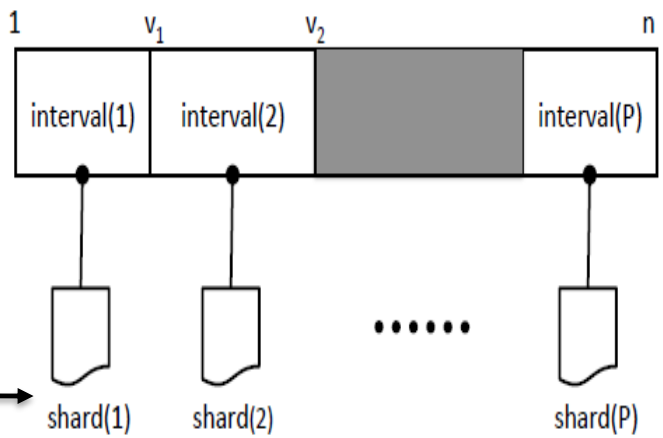
GraphChi

- Challenges for distributed graph computation
  - Partitioning a graph is difficult (especially for power law graphs)
  - Distributed graph processing incurs too much communication cost
- Could we compute big graphs on a single machine?
- **GraphChi :** Single-node out-of-core graph processing system
  - Goal: maximize sequential disk access while loading graph into memory (500x speedup for disk sequential vs random)
  - Method: Execute on individual subgraphs once a time with parallel sliding window (PSW) , load subgraphs efficiently from disk

GraphChi

PSW: parallel sliding window

- Vertices are partitioned into interval, each corresponds to a shard on disk
- **A shard on disk** include all in-edges for all vertices in corresponding interval



- Edges in a shard are sorted by source vertices
- Bring benefit for date access from disk
- Incur much preprocessing cost

Shard 1

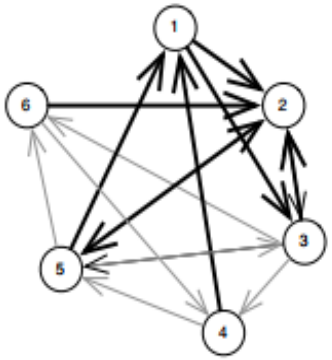
src	dst	value
1		
	2	0.3
3		
	2	0.2
4		
	1	1.4
5		
	1	0.5
	2	0.6
6		
	2	0.8

Shard 2

src	dst	value
1		
	3	0.4
2		
	3	0.3
3		
	4	0.8
5		
	3	0.2
6		
	4	1.9

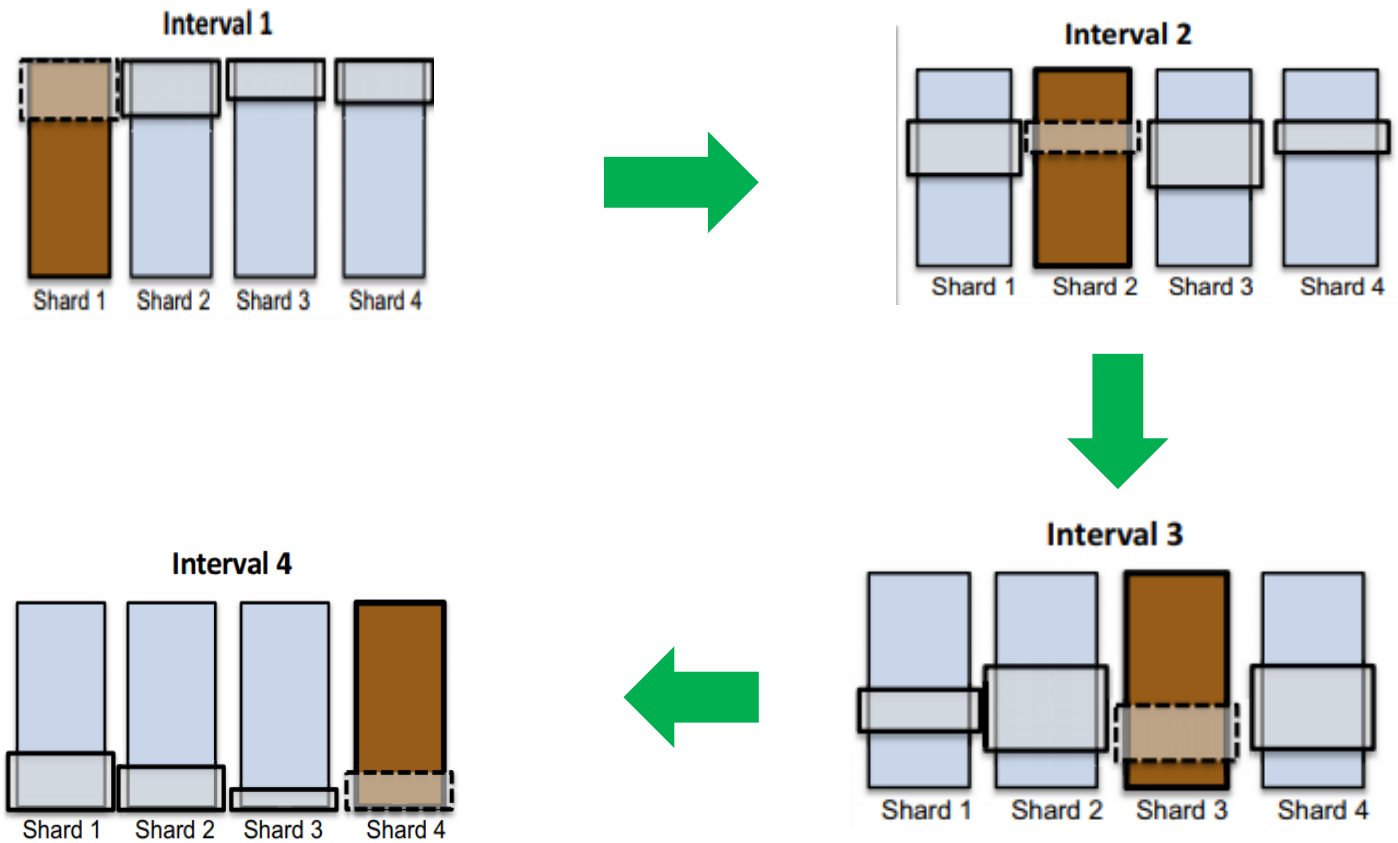
Shard 3

src	dst	value
2		
	5	0.6
3		
	5	0.9
	6	1.2
4		
	5	0.3
5		
	6	1.1



GraphChi

- Slide window to process one intervals with shard one by one



GraphChi

- Processing workflow for one interval
  (eg. Processing Interval 1)

Step 1: Load all in-edges from shard 1 into memory for vertices of interval 1

(1 sequential disk read)

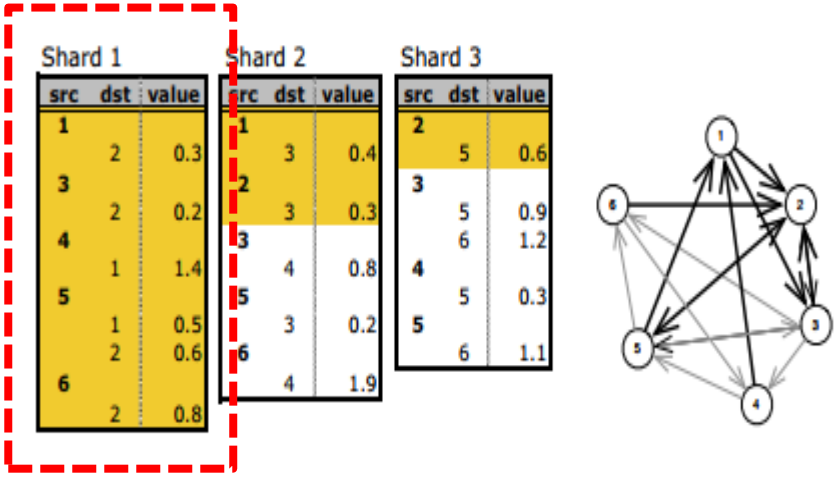


Fig. Execution interval 1 (with vertices 1, 2)

GraphChi

- Processing workflow for one interval (eg. Processing Interval 1)

Step 1: Load all in-edges from shard 1 into memory for vertices of interval 1

Step 2: Load all out-edges in memory form other shards for vertices of interval 1

(P-1 sequential reads)

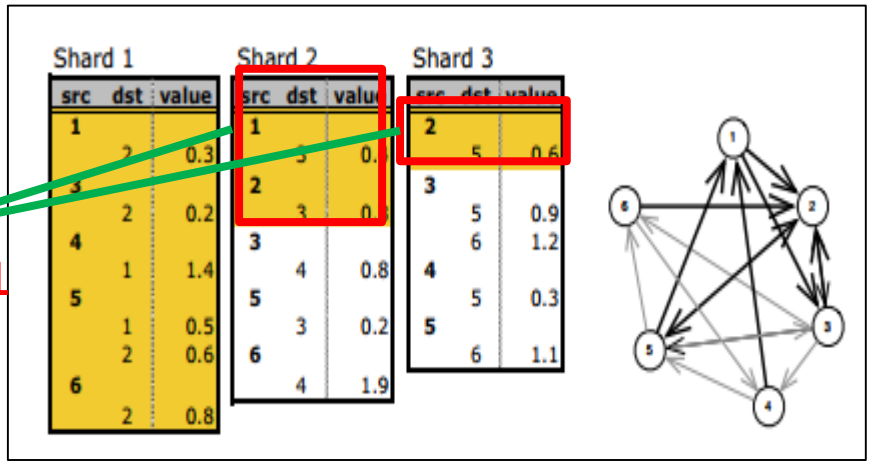


Fig. Execution interval (vertices 1, 2)

GraphChi

- Processing workflow for one interval (eg. Processing Interval 1)

Step 1: Load all in-edges from shard 1 into memory for vertices of interval 1

Step 2: Load all out-edges in memory form other shard for vertices of interval 1

Step 3: Update-function is executed on interval's vertices (vertex 1, 2).

(vertices-centric processing)

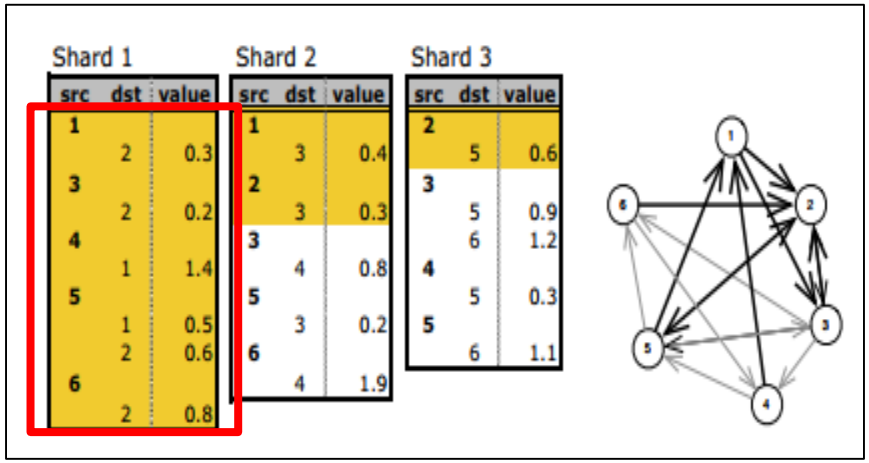


Fig. Execution interval 1 (with vertices 1, 2)

GraphChi

- Processing workflow for one interval (eg. Processing Interval 1)

Step 1: Load all in-edges from shard 1 into memory for vertices of interval 1

Step 2: Load all out-edges in memory form other shard for vertices of interval 1

Step 3: Update-function is executed on interval's vertices (vertex 1, 2)

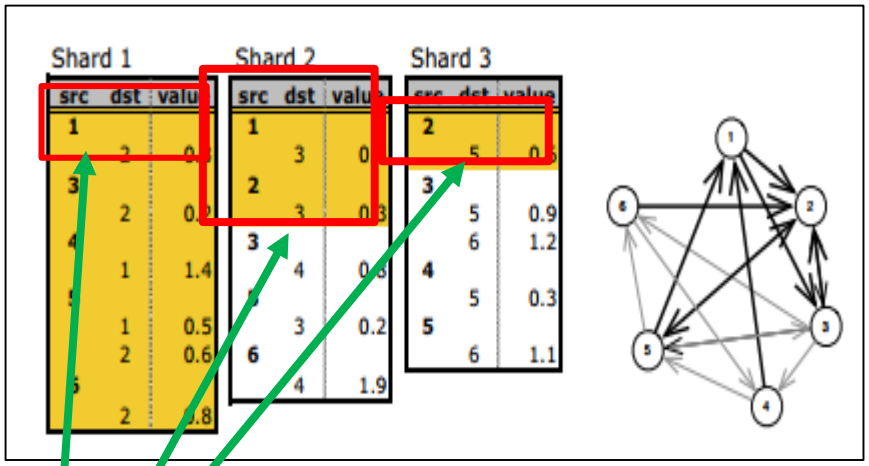


Fig. Execution interval 1 (with vertices 1, 2)

Step 4: **Commit to disk, updates are written back to disk (p sequential disk write)**

GraphChi

- Processing workflow for one interval (eg. Processing Interval 2)

Step 1: Load all in-edges from shard 2 into memory for vertices of interval 2

Step 2: Load all out-edges in memory form other shard for vertices of interval 2

Step 3: Update-function is executed on interval's vertices (vertex 3, 4)

Step 4: Commit to disk, update are written back to disk
(p sequential disk write)

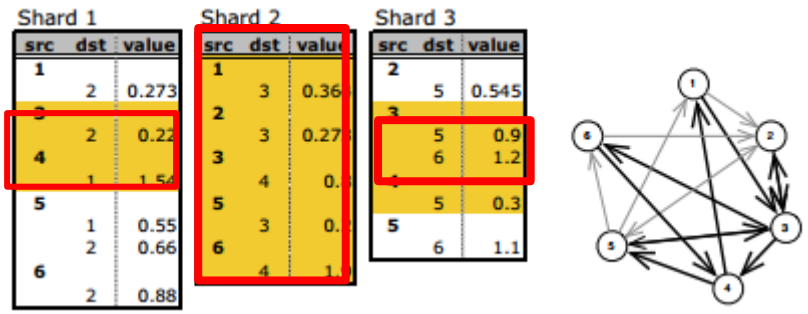


Fig. Execution interval 2 (vertices 3, 4)

PROGRAMMING FOR STREAM PROCESSING

Outline

- Stream data and stream applications
- Real-time stream processing
- Ecosystems and programming models

What is stream?

What is stream?

- A stream is a continuous and unbounded sequence of data elements made available over time

- Stream data in big data era
  - Volume, continuous and unbounded
  - Variety, changes quickly
  - Velocity, high speed incoming
  - Value, needs to be processed in near real-rime




Stream applications

“ 11 - 11 ”

(Didi) Real-time Traffic Monitoring

Hot Search Topics Ranking

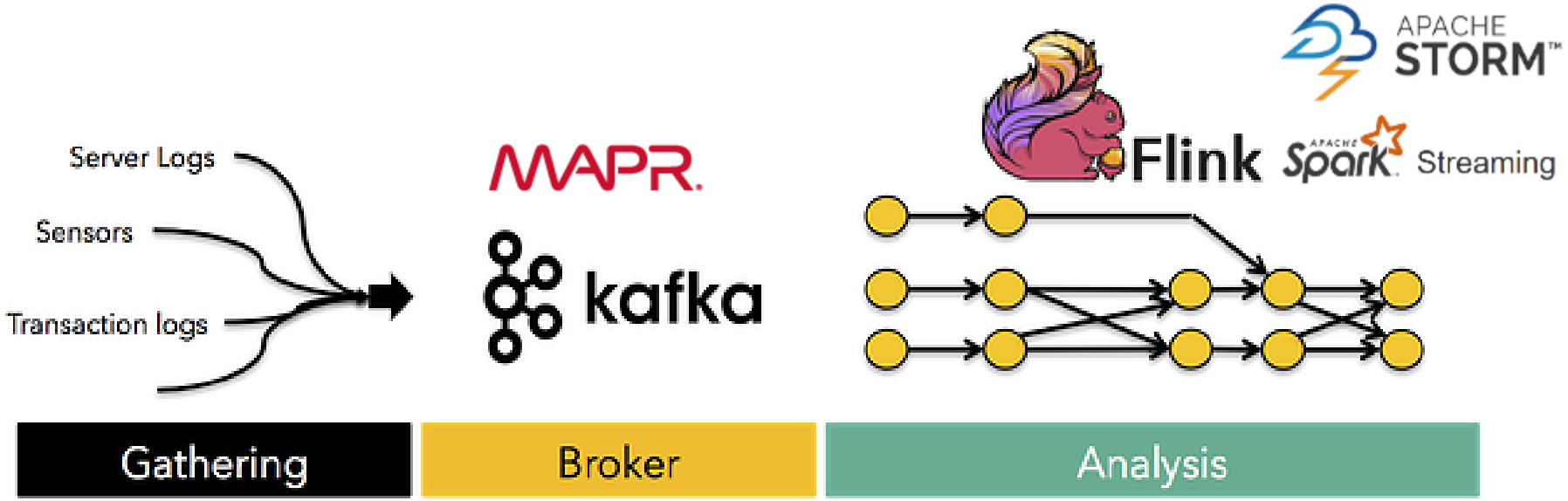
Market Transaction

Outline

- Stream data and stream applications
- Real-time stream processing
- Ecosystems and programming models

Common stream architecture

- Collect data from a variety of data sources
- Make data streams available for consumption
- Real-time stream processing



Data model - Tuples

- A tuple is the basic data element that can be processed independently

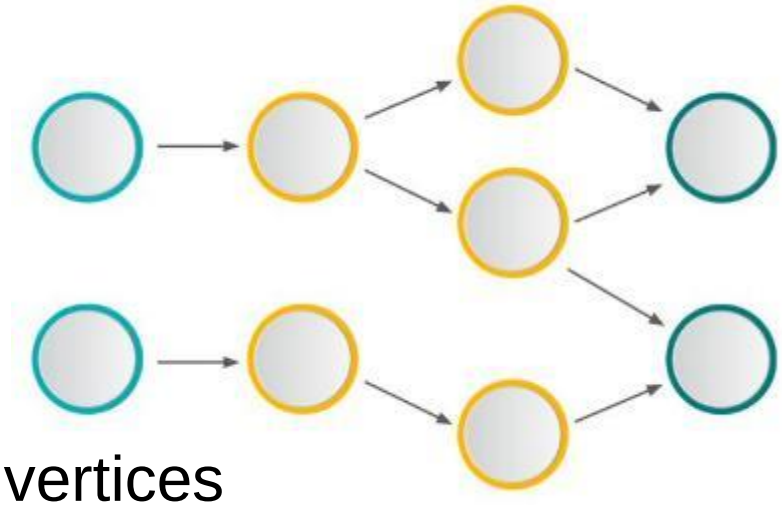
- Each tuple can be represented as a list of values. (t = < $value_1$, $value_2$, $value_3$,...>)


```
{
    "user_id": 17055506,
    "timestamp": 1421777578,
    "status":"Hello world!"
}
```

Application model - DAG

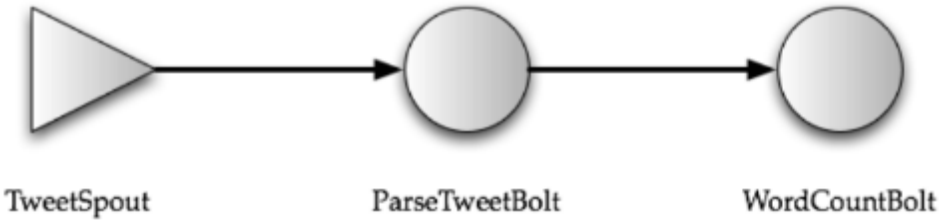
- Stream application can be modeled as a Directed Acyclic Graph (DAG)
  - Vertex: a simple operator

    executes common user defined function, e.g., map, filter, reduce, join,…



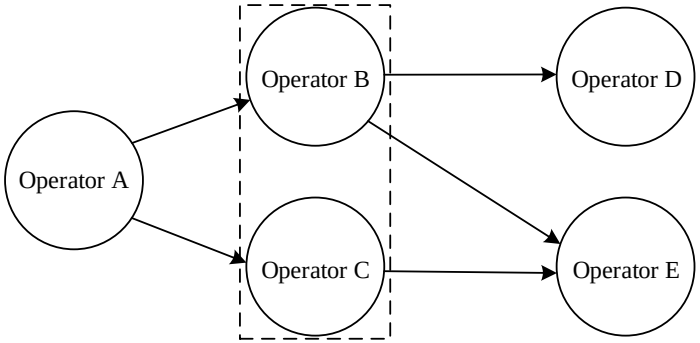
  - Edge: routing tuples between vertices

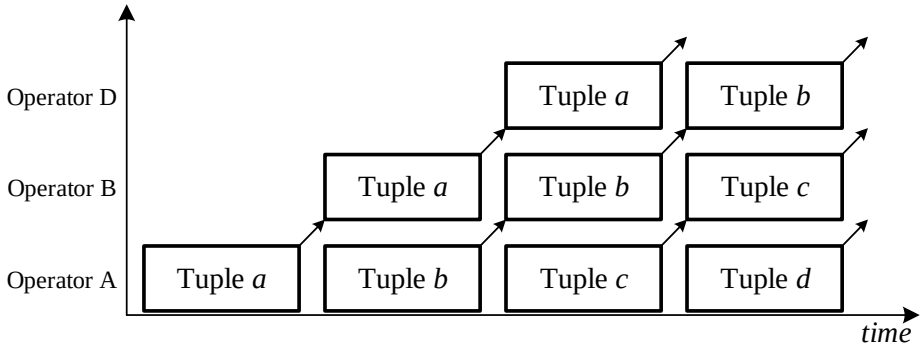
- A simple example: Twitter Word Count



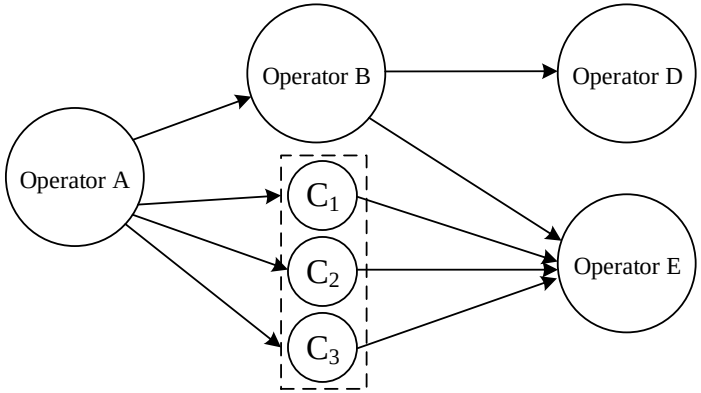
Parallisms in stream processing

- Stream processing exploits pipeline parallism and task parallism by using DAG model





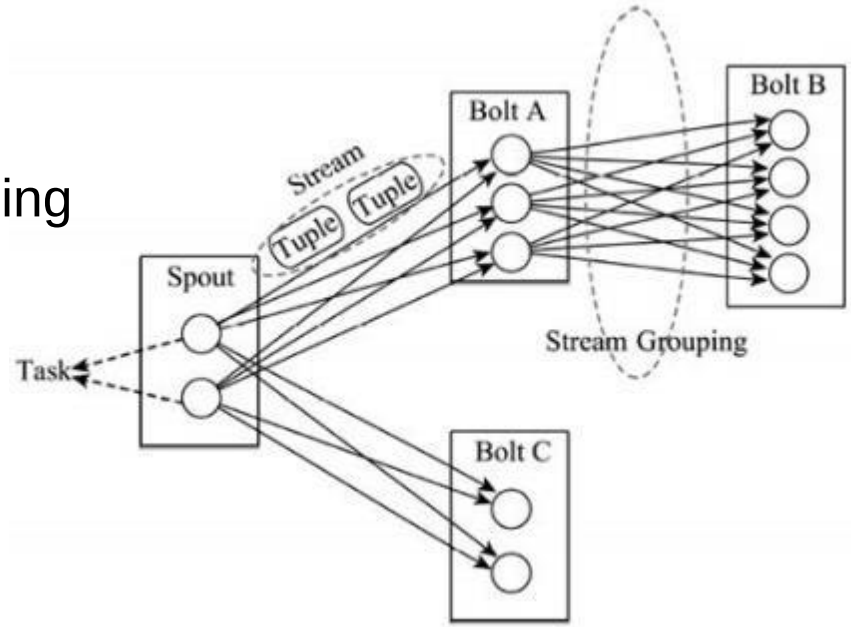
- Stream processing system can generate multiple instances for each operator to achieve high data parallism



Tuples grouping and dispatching

- Tuples are partitioned into different groups and dispatched to different instances of operators

- Grouping strategies
  - Key Grouping: hash based partitioning
  - Shuffle Grouping: round-robin
  - All Grouping: broadcast
  - Global Grouping: all to one
  - Others



Programming: step by step

1. Read input streams from data sources and generate a sequence of tuples
   - Kafka, HDFS, Flume, …
2. Build the DAG for a stream application
3. Specify the degree of parallisms for each operator and the data grouping strategies
   - Depending on different platform APIs

Development part for programmers

4. Submit the program to the cluster
   - Deployment
   - Fault Tolerant
   - Dashboard
   - …

Managed by platforms

Transparent to programmers

Outline

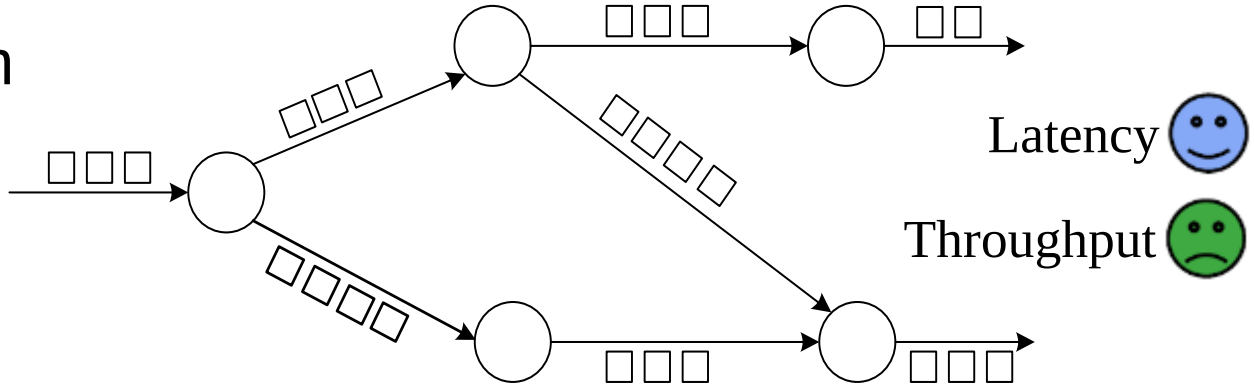
- Stream data and stream applications
- Real-time stream processing
- Ecosystems and programming models

Ecosystems

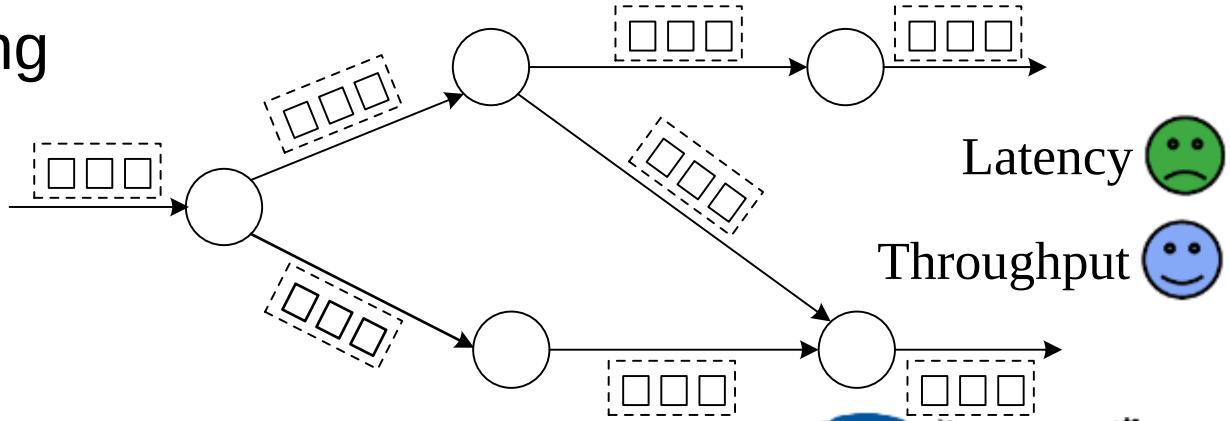
Open Source Stream Processing Frameworks

Two different programming models

- Continuous dataflow: process tuples one by one
  - Apache Storm
  - Apache Flink

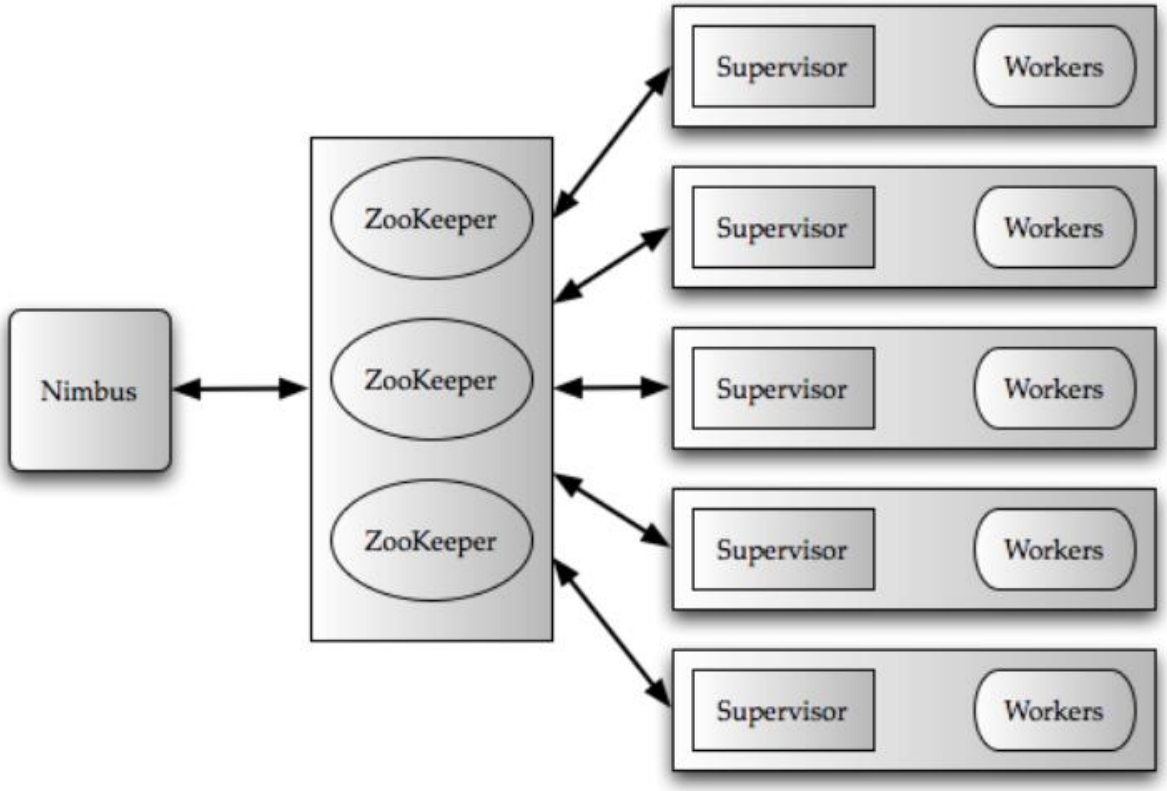


- Micro-batch: process tuples in small batches
  - Spark Streaming



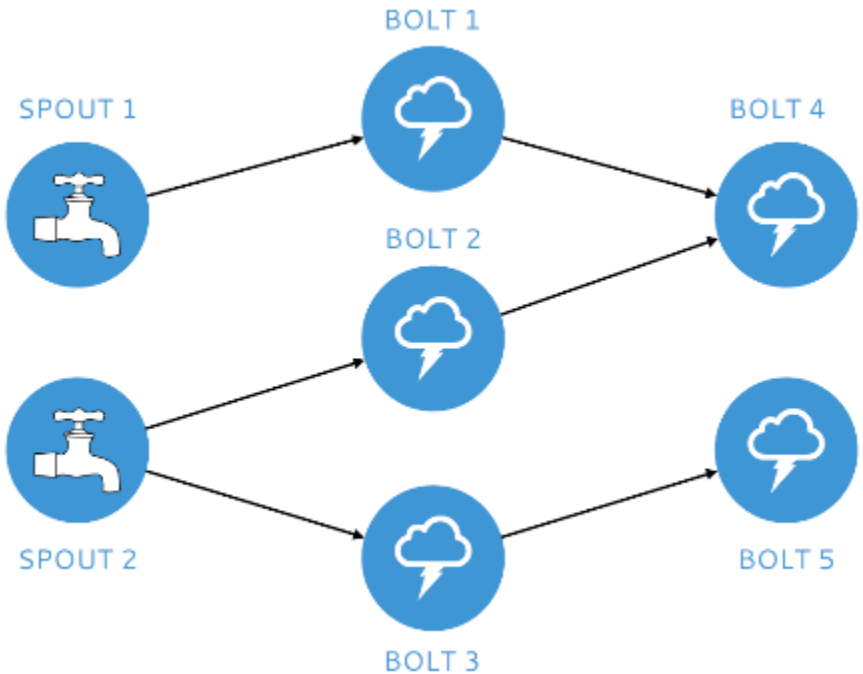
Apache Storm

- Storm is a classic continuous dataflow based distributed stream processing system developed by Twitter. It has many variations such as JStorm in Alibaba.



DAG model: topology

- Two kinds of components
  - Spout: read streams from sources and generates tuples
  - Bolt: execute a simple user defined function for tuples
- Each spout and bolt can generate multiple instances as multiple tasks and partition them in different nodes



Use case: word count on Storm

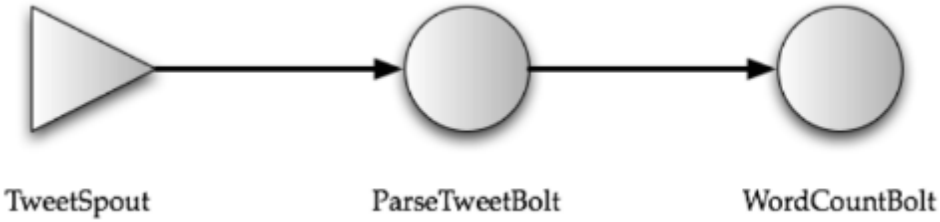
- Topology Building

```
TopologyBuilder tb = new TopologyBuilder();

tb.setSpout("TweetSpout", kafkaSpout(), 4);

tb.setBolt("ParseTweetBolt", new ParseBolt(), 4).setNumTasks(8).shuffleGrouping("TweetSpout");

tb.setBolt("WordCountBolt", new CountBolt(), 4).fieldsGrouping("Split", new Fields("ParseTweetBolt"));
```

Specify the degree of parallism

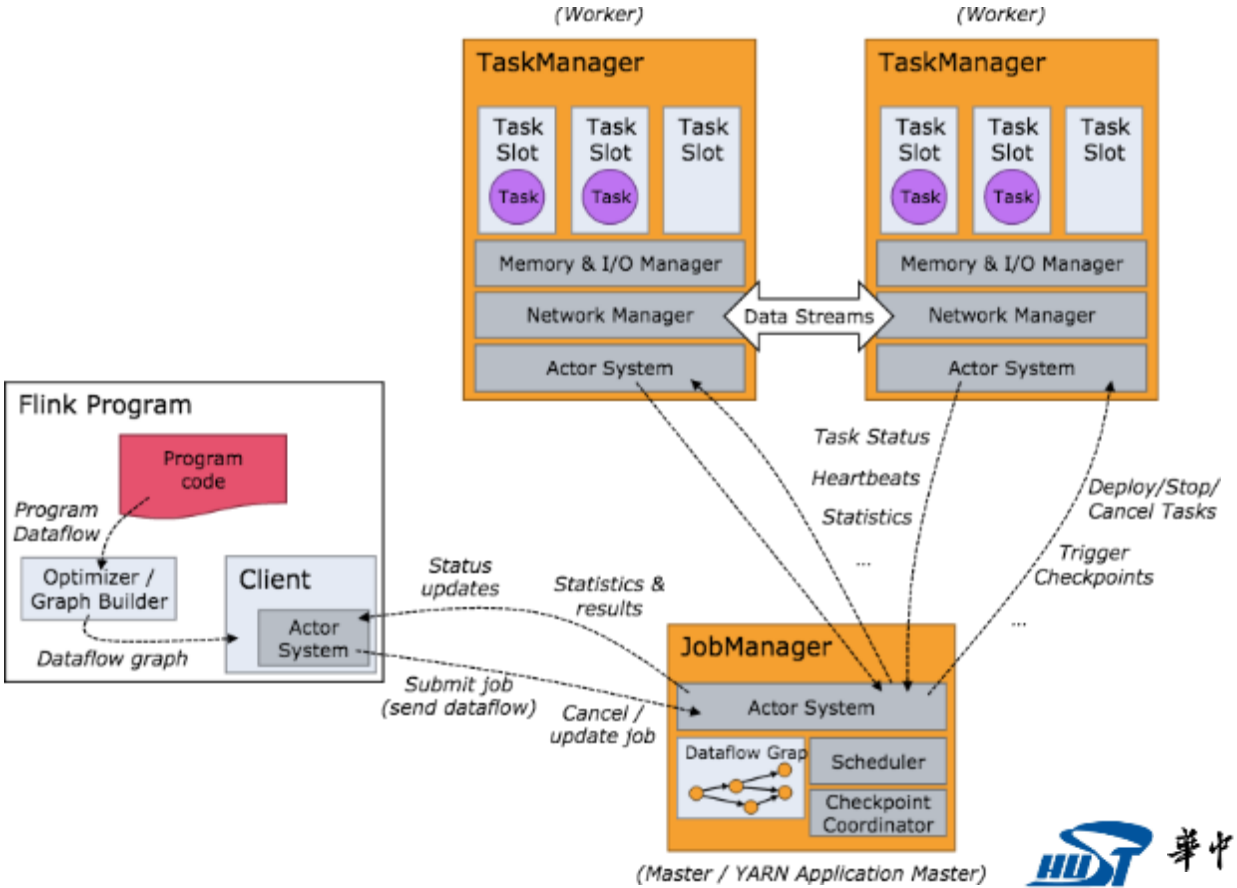
User defined functions

Specify the grouping strategies



Apache Flink

- Flink is one of the most popular continuous dataflow based distributed stream processing system, which can also support big data batch processing



DAG model: stream & transformation

- Flink's DAG model is almost like that in Storm, which contains one source operator, one sink operator and mulitple transformation operators
  - Transformation can be user defined functions or many pre-defined operations provied by Flink API, e.g., flatmap, map, reduce, windows, ...

Use case: word count on Flink

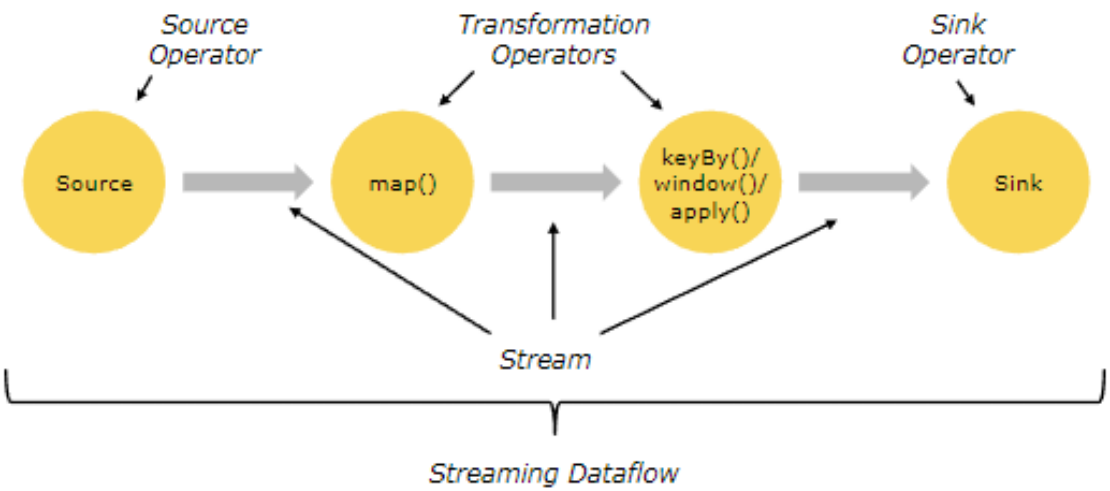
```
DataStream<String> lines = env.addSource(
                                new FlinkKafkaConsumer<>(…));

DataStream<Event> events = lines.map((line) -> parse(line));

DataStream<Statistics> stats = events
        .keyBy("id")
        .timeWindow(Time.seconds(10))
        .apply(new MyWindowAggregationFunction());

stats.addSink(new RollingSink(path));
```

Source

Transformation

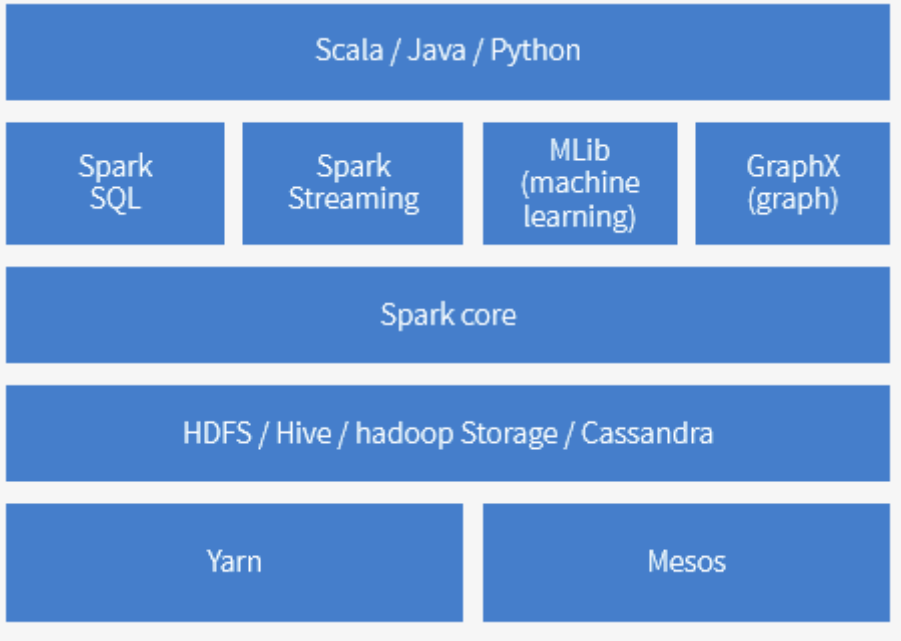
Transformation

Sink



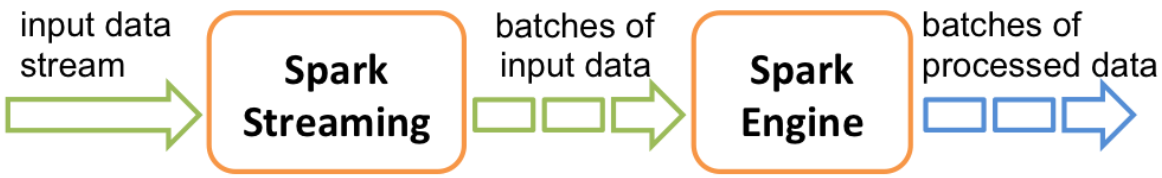
Spark Streaming

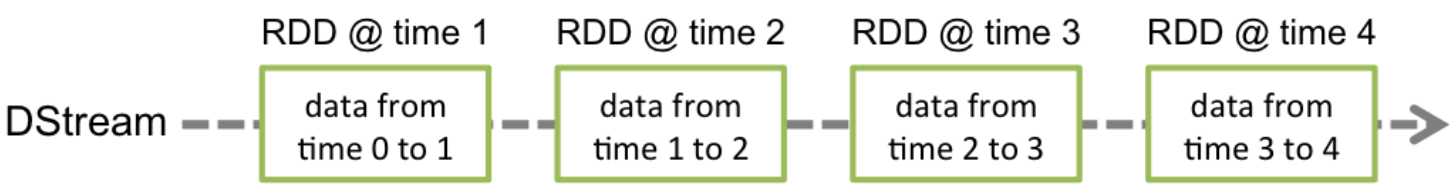
- Spark Streaming is an extension of the core Spark API, it is a classical micro-batch based distributed stream processing system



Discretized streams and RDDs

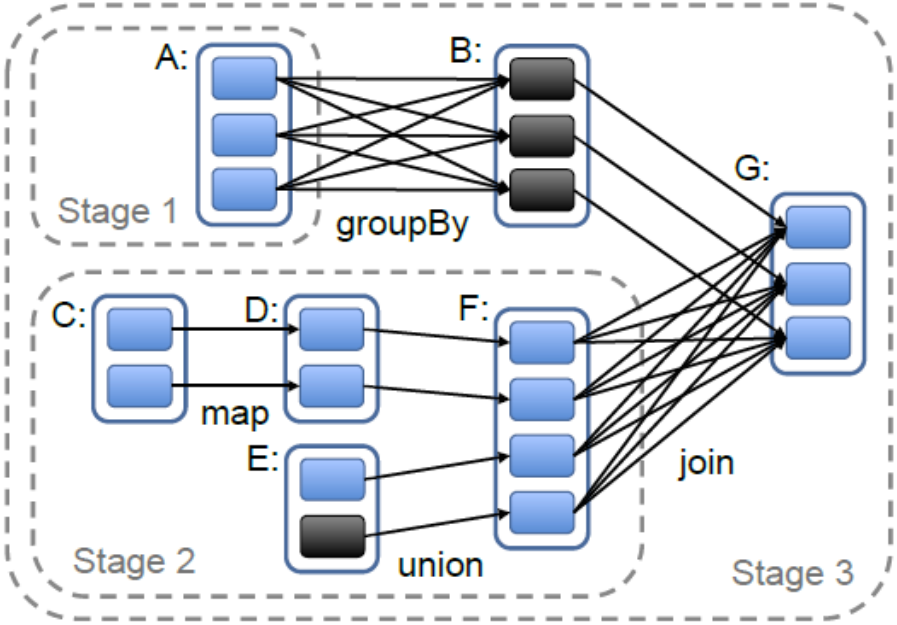
- The basic data abstraction of Spark Streaming is discretized streams
  - Tuples in a fixed interval (e.g., 1s) are partitioned into a small batch, and transformed to RDD by using Spark engine

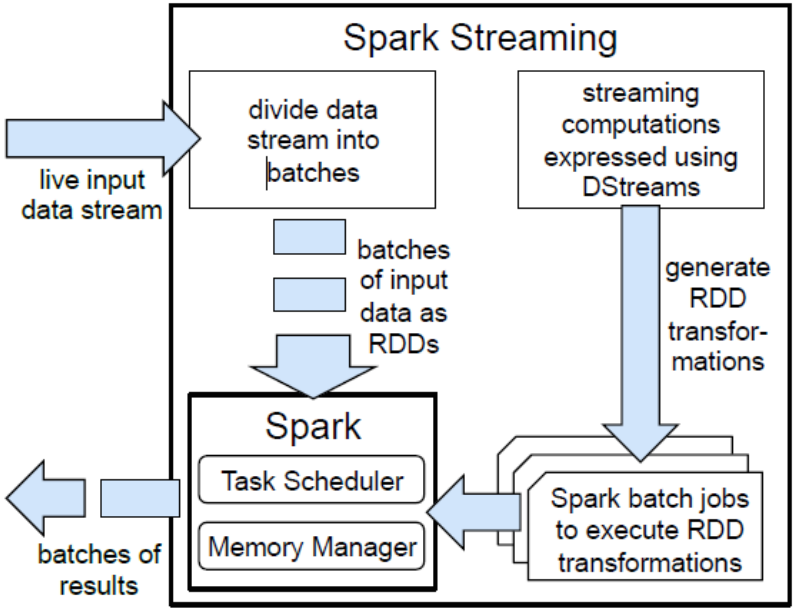




DAG model: RDD transformations

- Spark Streaming executes RDD transformations for each small batch of tuples and generates batches of results





Word count on Spark Streaming

- Essentially the same as batch processing

```
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaPairRDD<String, Integer> counts = textFile
    .flatMap(s -> Arrays.asList(s.split(" ")).iterator())
    .mapToPair(word -> new Tuple2<>(word, 1))
    .reduceByKey((a, b) -> a + b);
counts.saveAsTextFile("hdfs://...");
```