Large Scale Graph Processing
X-Stream and Chaos

Amir H. Payberah
amir@sics.se

KTH Royal Institute of Technology
2016/10/05

Graphs

Big Graphs

- Large graphs are a subset of the big data problem.
- Billions of vertices and edges, hundreds of gigabytes.
- Normally tackled on large clusters.
  - Pregel, Giraph, GraphLab, PowerGraph ...
  - Complexity, power consumption ...

Could we compute Big Graphs on a single machine?

Challenges

- Disk-based processing
  - Problem: graph traversal = random access
  - Random access is inefficient for storage

Challenges

- Disk-based processing
  - Problem: graph traversal = random access
  - Random access is inefficient for storage

Medium	Read (MB/s)		Write (MB/s)	
	Random	Sequential	Random	Sequential
RAM	567	2605	1057	2248
SSD	22.64	355	49.16	298
Disk	0.61	174	1.27	170

Note: 64 byte cachelines, 4K blocks (disk random), 16M chunks (disk sequential)

Eiko Y., and Roy A., "Scale-up Graph Processing: A Storage-centric View", 2013.

X-Stream

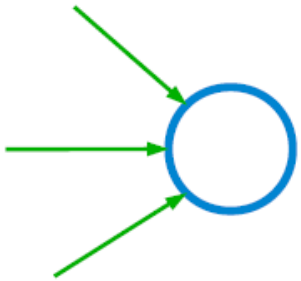
X-Stream

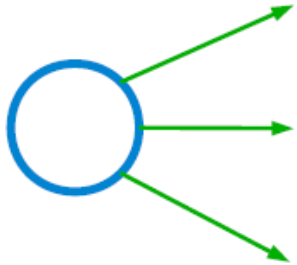
- X-Stream makes graph accesses sequential.
- Contribution:
  - Edge-centric scatter-gather model
  - Streaming partitions

Edge-Centric Programming Model

Vertex-Centric Programming Model

- Vertex-centric Programming model
  - Write a vertex program
  - State stored in vertices.
- Vertex operations:
  - Gather-Apply-Scatter (GAS)
  - Gather updates from incoming edges
  - Scatter updates along outgoing edges



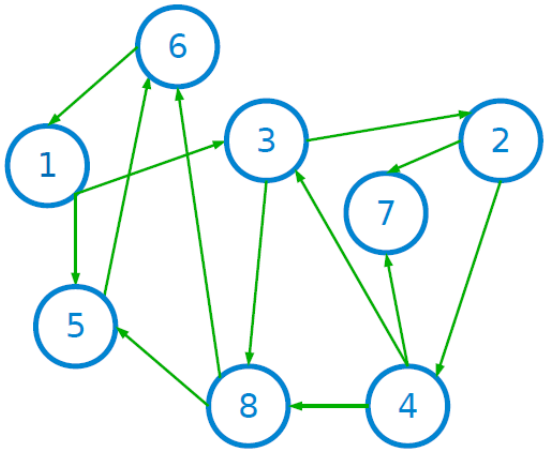


A Vertex-Centric Program

- Iterates over vertices

```
Until convergence {
  // the gather phase
  for all vertices v
    for all incoming edges to v: v.value = g(v.value, update)

  // the scatter phase
  for all vertices v
    for all outgoing edges from v: update = f(v.value)
}
```

Vertex-Centric Scatter-Gather (1/5)



vertices

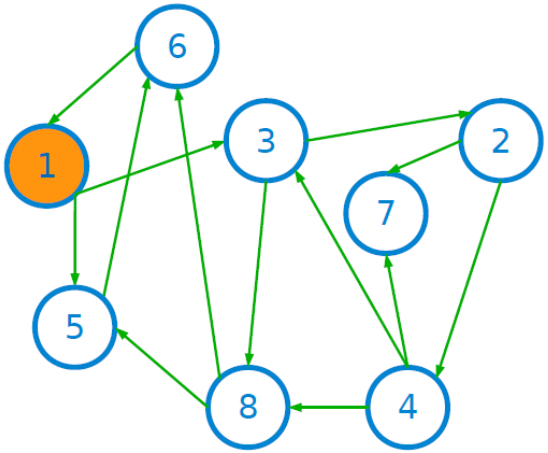
v
1
2
3
4
5
6
7
8

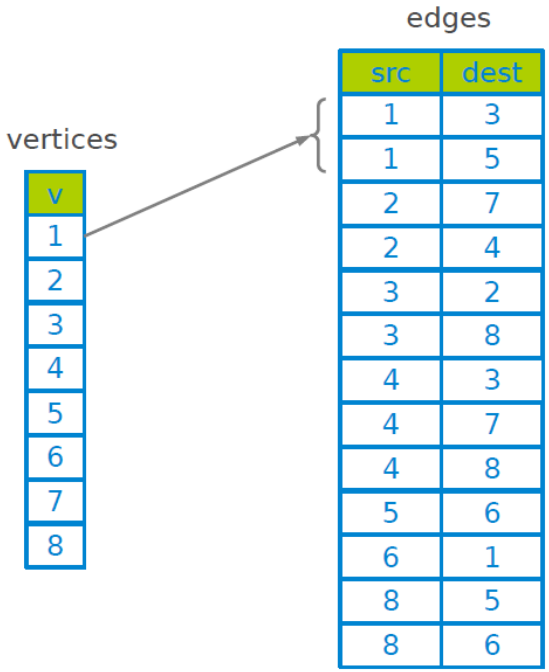
edges

src	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

```
Until convergence {
  // the gather phase
  for all vertices v
    for all incoming edges to v: v.value = g(v.value, update)

  // the scatter phase
  for all vertices v
    for all outgoing edges from v: update = f(v.value)
}
```

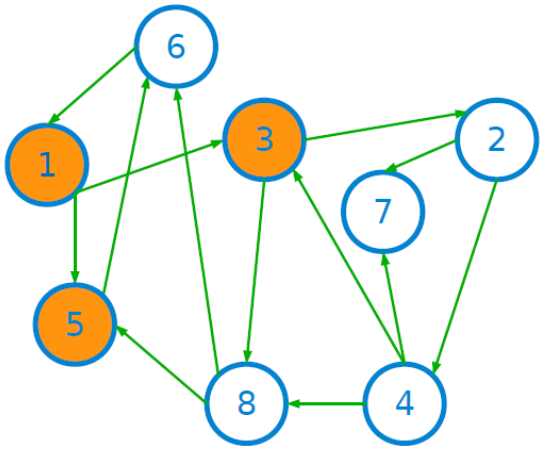
Vertex-Centric Scatter-Gather (2/5)

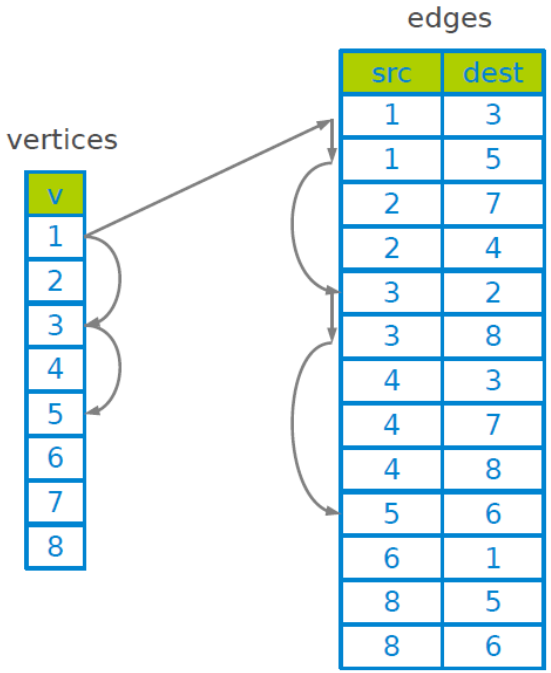





```
Until convergence {
  // the gather phase
  for all vertices v
    for all incoming edges to v: v.value = g(v.value, update)

  // the scatter phase
  for all vertices v
    for all outgoing edges from v: update = f(v.value)
}
```

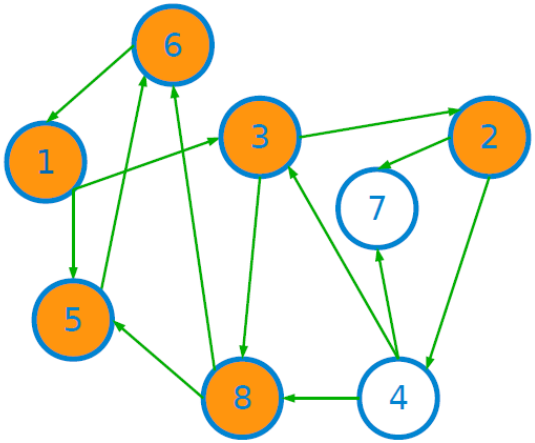
Vertex-Centric Scatter-Gather (3/5)

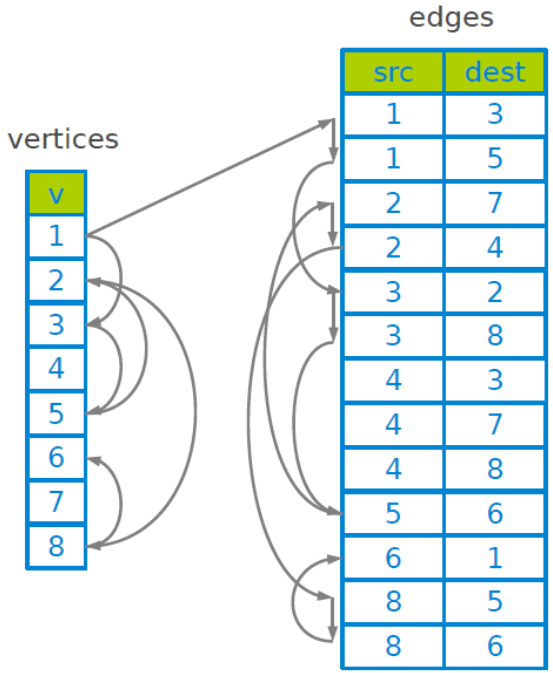





```
Until convergence {
  // the gather phase
  for all vertices v
    for all incoming edges to v: v.value = g(v.value, update)

  // the scatter phase
  for all vertices v
    for all outgoing edges from v: update = f(v.value)
}
```

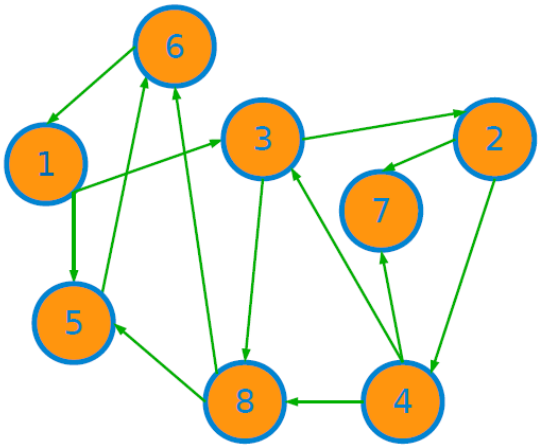
Vertex-Centric Scatter-Gather (4/5)

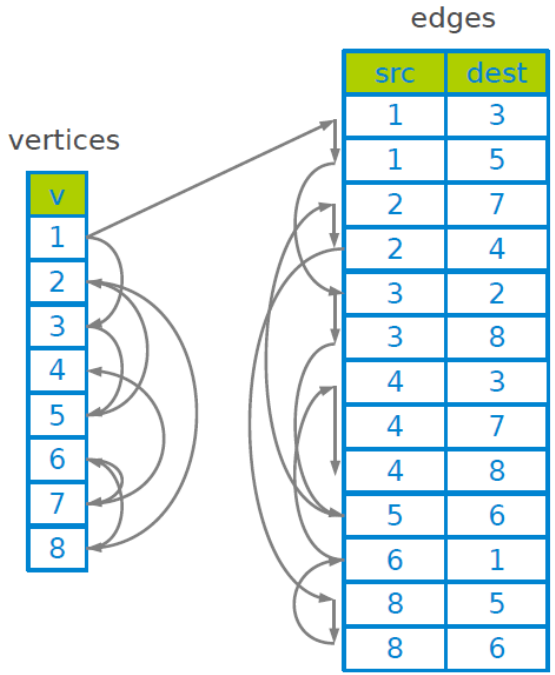





```
Until convergence {
  // the gather phase
  for all vertices v
    for all incoming edges to v: v.value = g(v.value, update)

  // the scatter phase
  for all vertices v
    for all outgoing edges from v: update = f(v.value)
}
```

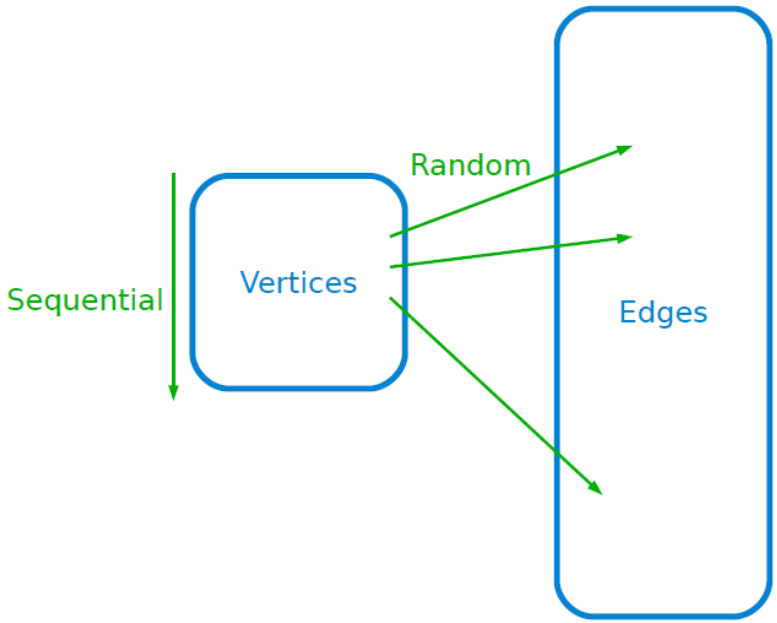
Vertex-Centric Scatter-Gather (5/5)



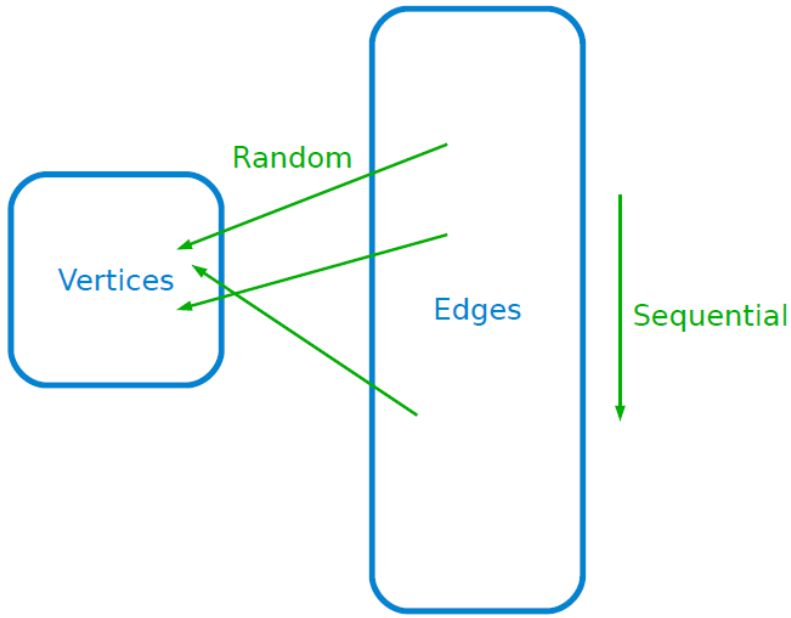



```
Until convergence {
  // the gather phase
  for all vertices v
    for all incoming edges to v: v.value = g(v.value, update)

  // the scatter phase
  for all vertices v
    for all outgoing edges from v: update = f(v.value)
}
```

Vertex-Centric vs. Edge-Centric Access



Vertex-centric



Edge-centric

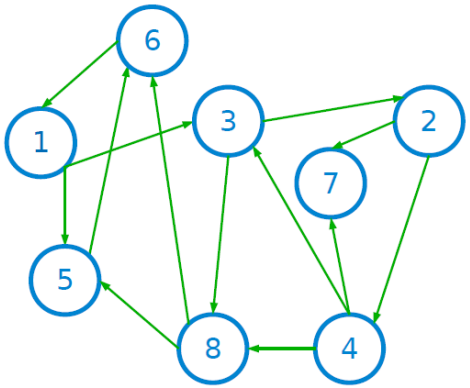
- Iterates over edges

Vertex-Centric to Edge-Centric Programming

```
Until convergence {
  // the gather phase
  for all vertices v
    for all incoming edges to v: v.value = g(v.value, update)

  // the scatter phase
  for all vertices v
    for all outgoing edges from v: update = f(v.value)
}
```

```
Until convergence {
  // the gather phase
  for all edges e
    e.dst.value = g(e.dst.value, u.value)

  // the scatter phase
  for all edges e
    u = new update
    u.dst = e.dst
    u.value = f(e.src.value)
}
```

Edge-Centric Scatter-Gather (1/5)



vertices

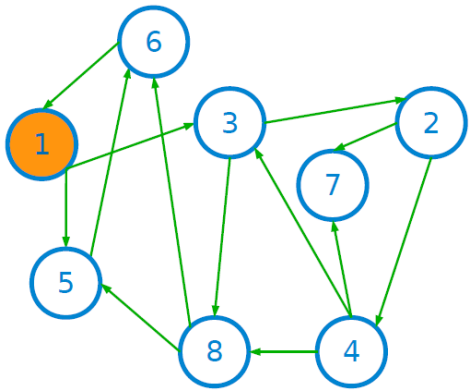
v
1
2
3
4
5
6
7
8

edges

src	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

```
Until convergence {
  // the gather phase
  for all edges e
    e.dst.value = g(e.dst.value, u.value)

  // the scatter phase
  for all edges e
    u = new update
    u.dst = e.dst
    u.value = f(e.src.value)
}
```

Edge-Centric Scatter-Gather (2/5)



edges

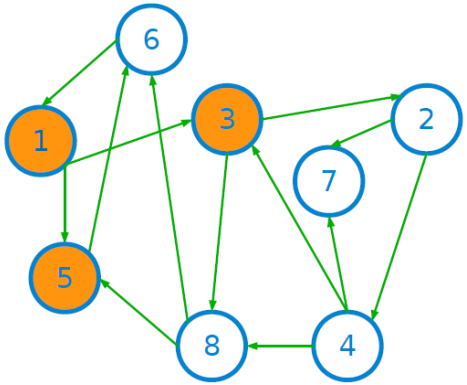
src	dest
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

vertices

v
1
2
3
4
5
6
7
8

```
Until convergence {
  // the gather phase
  for all edges e
    e.dst.value = g(e.dst.value, u.value)

  // the scatter phase
  for all edges e
    u = new update
    u.dst = e.dst
    u.value = f(e.src.value)
}
```

Edge-Centric Scatter-Gather (3/5)



vertices

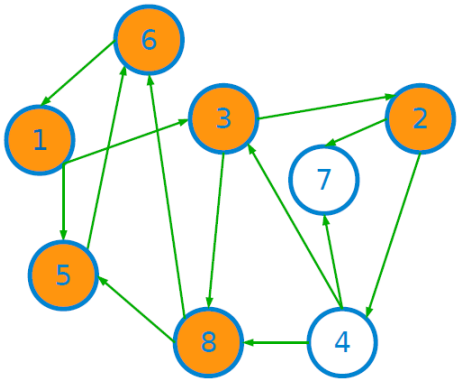
v
1
2
3
4
5
6
7
8

edges

src	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

```
Until convergence {
  // the gather phase
  for all edges e
    e.dst.value = g(e.dst.value, u.value)

  // the scatter phase
  for all edges e
    u = new update
    u.dst = e.dst
    u.value = f(e.src.value)
}
```

Edge-Centric Scatter-Gather (4/5)



vertices

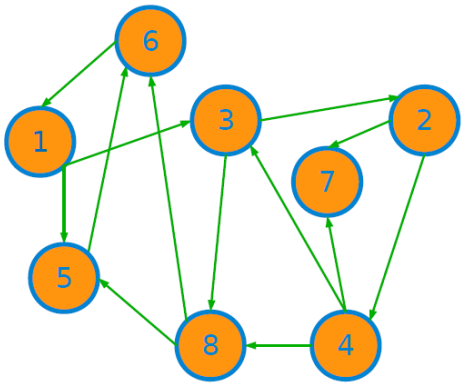
v
1
2
3
4
5
6
7
8

edges

src	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

```
Until convergence {
  // the gather phase
  for all edges e
    e.dst.value = g(e.dst.value, u.value)

  // the scatter phase
  for all edges e
    u = new update
    u.dst = e.dst
    u.value = f(e.src.value)
}
```

Edge-Centric Scatter-Gather (5/5)



vertices

v
1
2
3
4
5
6
7
8

edges

src	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

```
Until convergence {
  // the gather phase
  for all edges e
    e.dst.value = g(e.dst.value, u.value)

  // the scatter phase
  for all edges e
    u = new update
    u.dst = e.dst
    u.value = f(e.src.value)
}
```

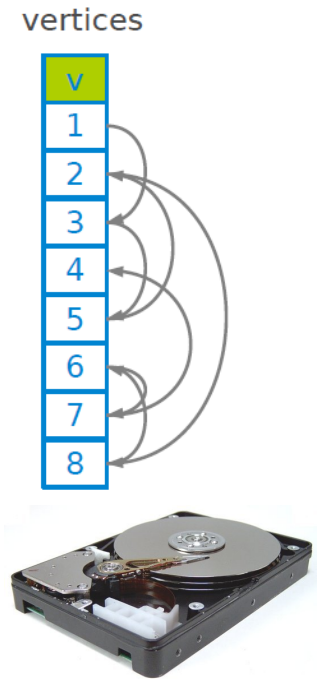
Vertex-Centric vs. Edge-Centric Tradeoff

- Vertex-centric scatter-gather: $\frac{EdgeData}{RandomAccessBandwidth}$
- Edge-centric scatter-gather: $\frac{Scatters \times EdgeData}{SequentialAccessBandwidth}$
- Sequential Access Bandwidth $\gg$ Random Access Bandwidth.
- Few scatter gather iterations for real world graphs.

Streaming Partitions

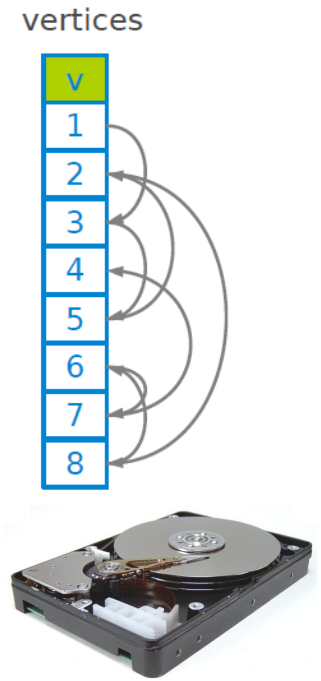
Problem

- Problem: still have random access to vertex set.



Problem

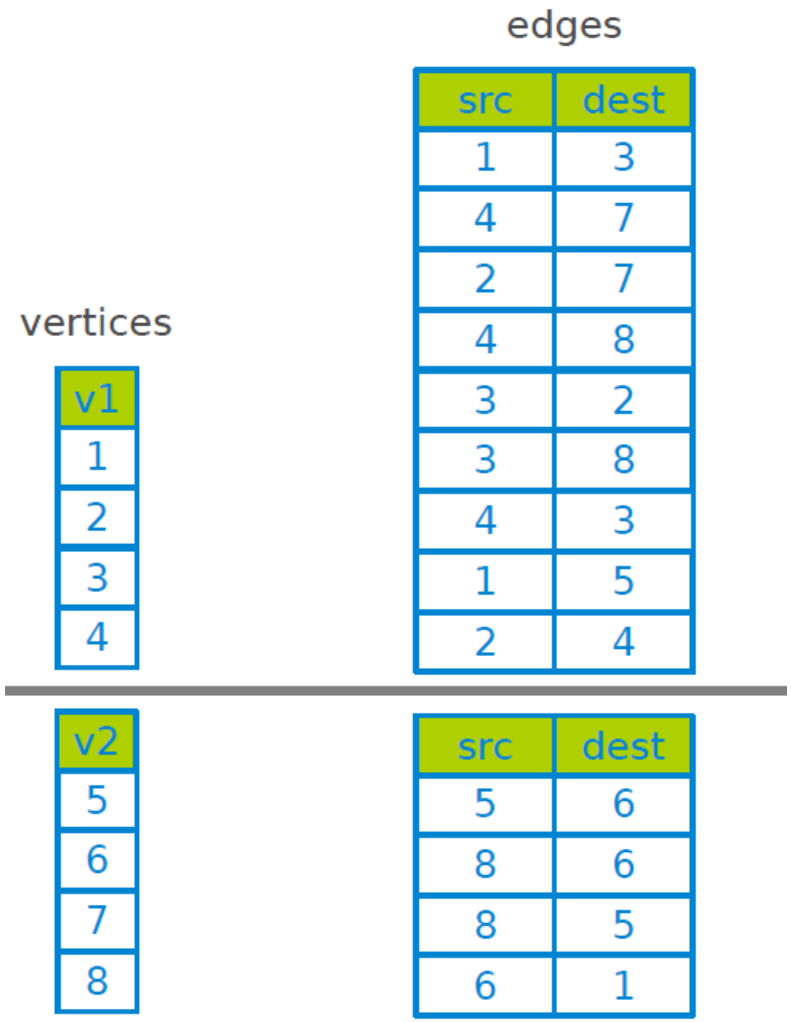
- Problem: still have random access to vertex set.



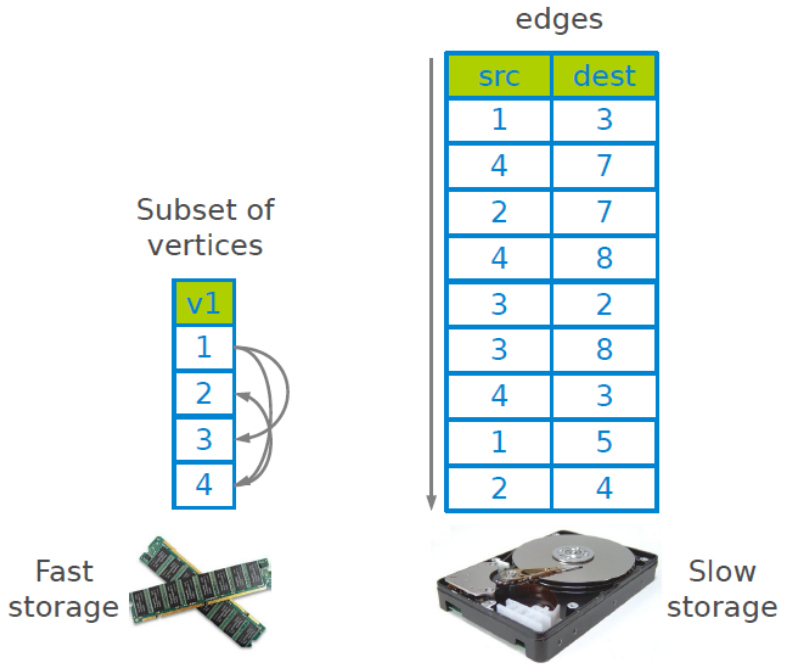
Solution

Partition the graph into streaming partitions.

Partitioning the Graph (1/2)



Partitioning the Graph (2/2)



Random access for free.

Streaming Partition

- A subset of the vertices that fits in RAM.
- All edges whose source vertex is in that subset.
- No requirement on quality of the partition, e.g., sorting edges.
- Consists of three sets: vertex set, edge list, and update list.

Streaming Partition Scatter-Gather

```
// the scatter phase
for each streaming_partition p {
  load Vertices(p)

  for each unprocessed e in Edges(P)
    u = new update
    u.dst = e.dst
    u.value = f(e.src.value)
    add u to Update(partition(u.dst))
}

// the gather phase
for each streaming-partition p {
  load Vertices(p)

  for each unprocessed u in Update(p)
    u.dst.value = g(u.dst.value, u.value)

  delete Update(p)
}
```

X-Stream Limitations

- Capacity: amount of storage on a single machine
- Bandwidth: storage bandwidth on a single machine

Chaos

Chaos

- Extend X-Stream to a cluster
- Goals:
  - Capacity: aggregate storage on all machines
  - Bandwidth: aggregate bandwidth on all machines

Contribtion

- X-Stream: iterates over partitions
- For all partitions:
  - Load vertex-set from storage into memory
  - Stream edge-set from storage

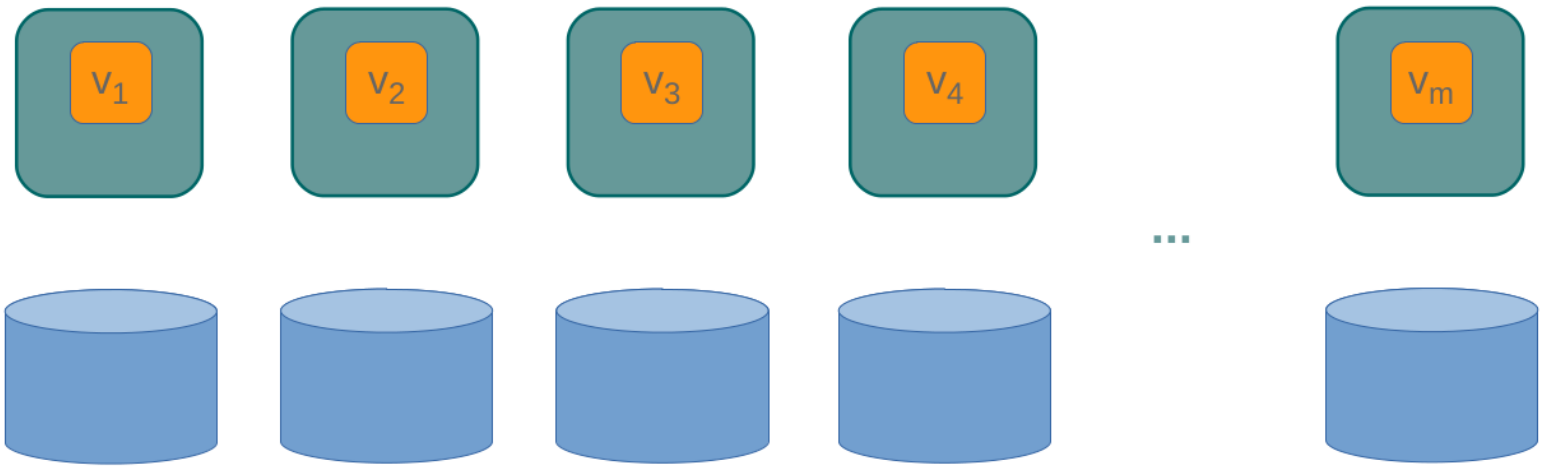
Contribtion

- X-Stream: iterates over partitions
- For all partitions:
  - Load vertex-set from storage into memory
  - Stream edge-set from storage
- Streaming partitions are independent

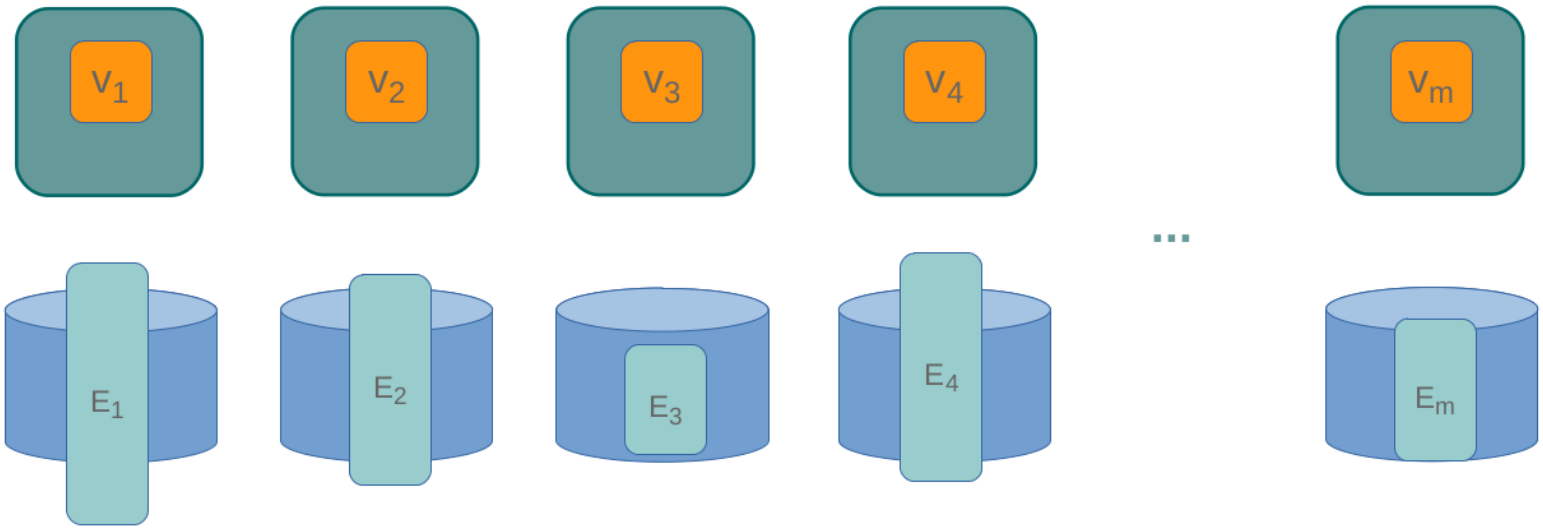
Contribtion

- X-Stream: iterates over partitions
- For all partitions:
  - Load vertex-set from storage into memory
  - Stream edge-set from storage
- Streaming partitions are independent
- Chaos: iterate in parallel over partitions

Vertex Distribution

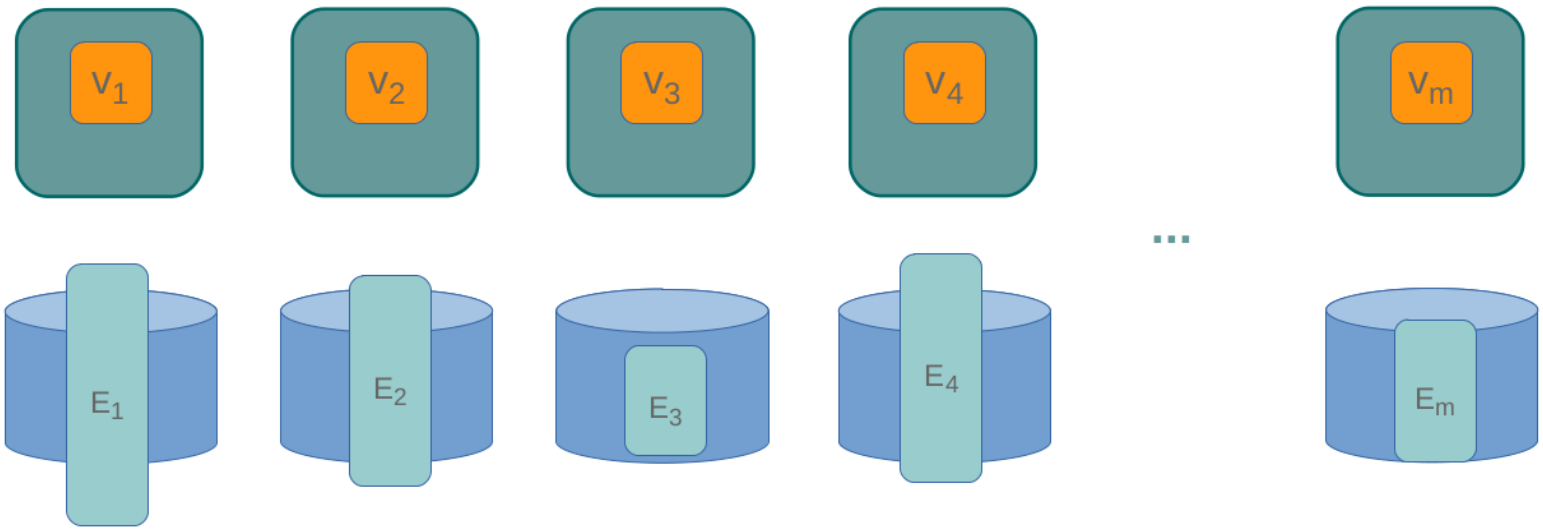


Edge Distribution



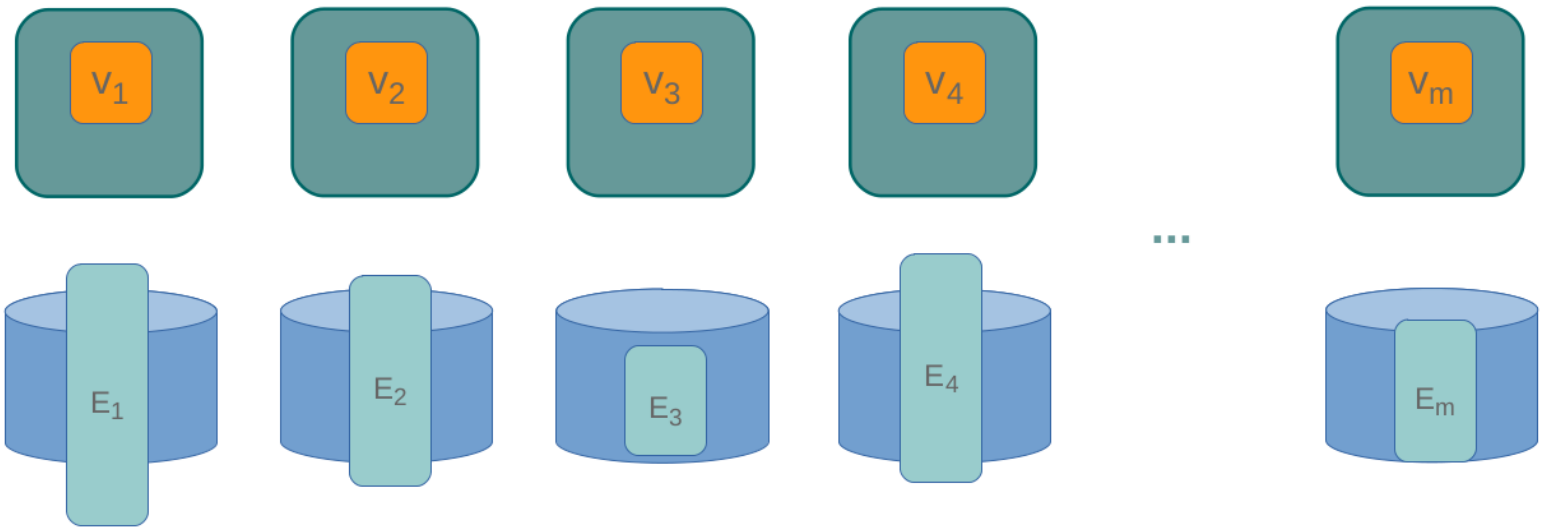
Load Imbalance

- How to deal with load imbalance?



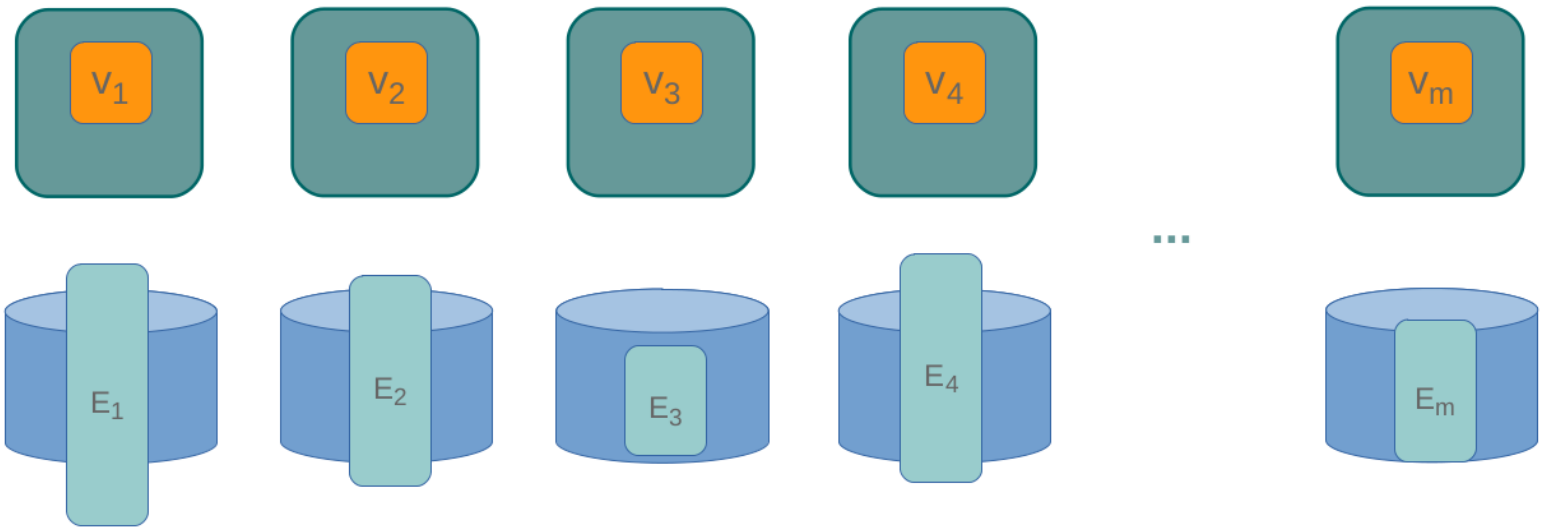
Load Imbalance

- How to deal with load imbalance?
- I/O imbalance: flat storage (storage sub-system)



Load Imbalance

- How to deal with load imbalance?
- I/O imbalance: flat storage (storage sub-system)
- Computational imbalance: work stealing (computation sub-system)



Chaos Components

- Storage sub-system
- Computation sub-system

Storage Sub-System

Storage Sub-System

- One storage engine on each machine.
- Supplies vertices, edges and updates of different partitions to the computation sub-system.
- The vertices, edges and updates of a partition are uniformly randomly spread over the different storage engines.

Streaming Partitions

- A streaming partition consists of:
  - Vertex lists: set of vertices that fits in memory
  - Edge lists: all of their outgoing edges
  - Update lists: all of their incoming updates

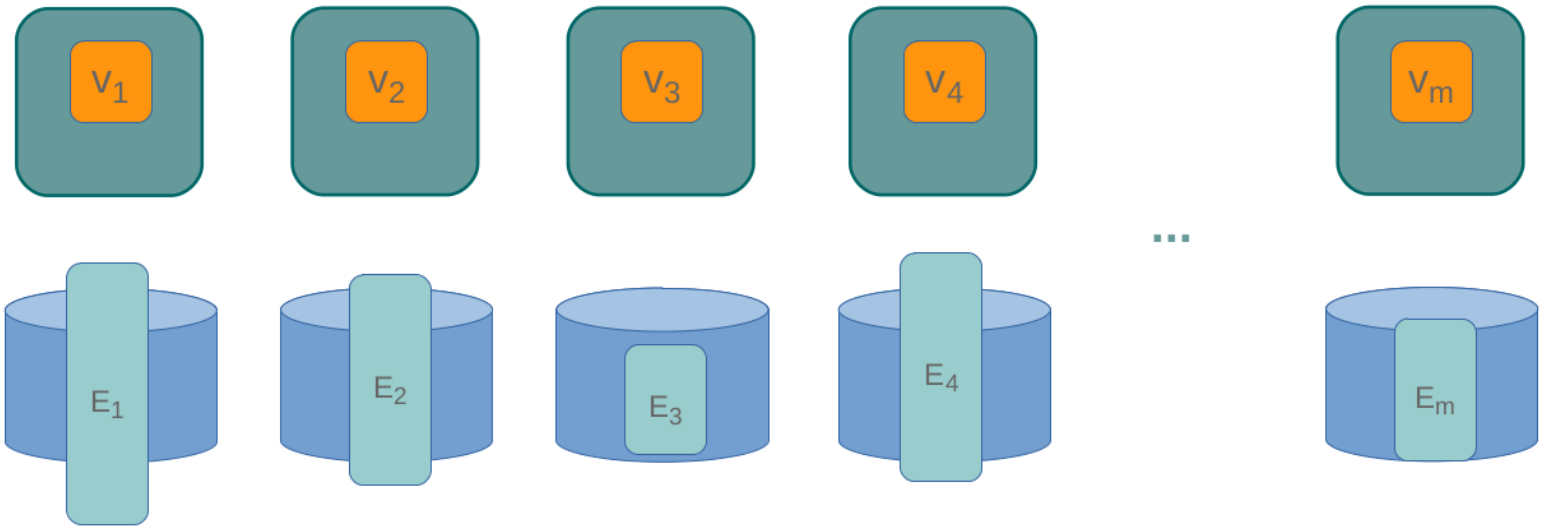
Streaming Partitions

- A streaming partition consists of:
  - Vertex lists: set of vertices that fits in memory
  - Edge lists: all of their outgoing edges
  - Update lists: all of their incoming updates
- This partitioning is the only pre-processing done in Chaos.

Stored Data Structures

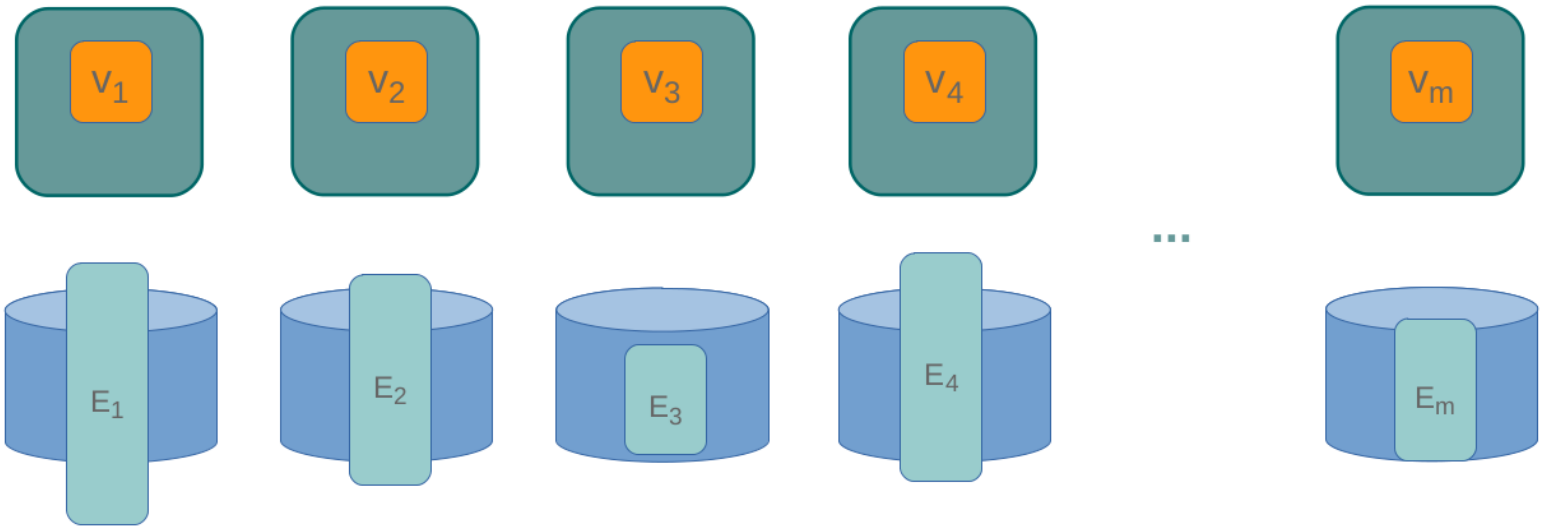
- For each partition, Chaos records three data structures on storage:
  - Vertex set: initialized during pre-processing, read during scatter and gather.
  - Edge set: created during pre-processing, read during scatter.
  - Update set: created and written to storage during scatter, read during gather.
- The accumulators are temporary structures, and are never written to storage.

I/O Imbalance (1/2)



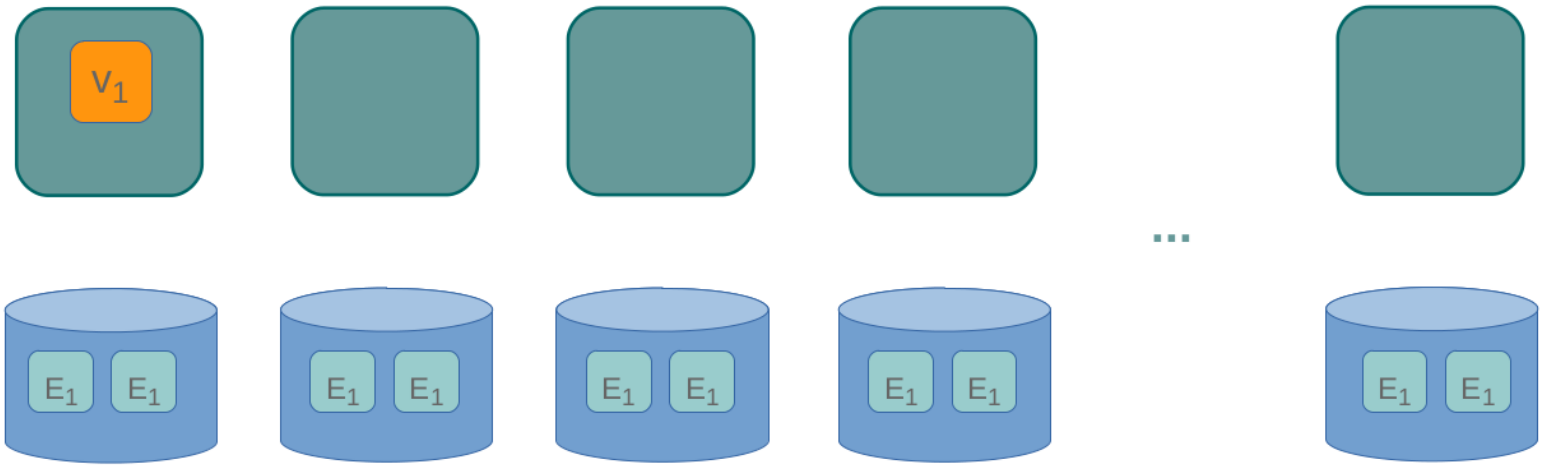
I/O Imbalance (2/2)

- Locality hardly matters
- There is no point in putting vertices and edges of a partition together



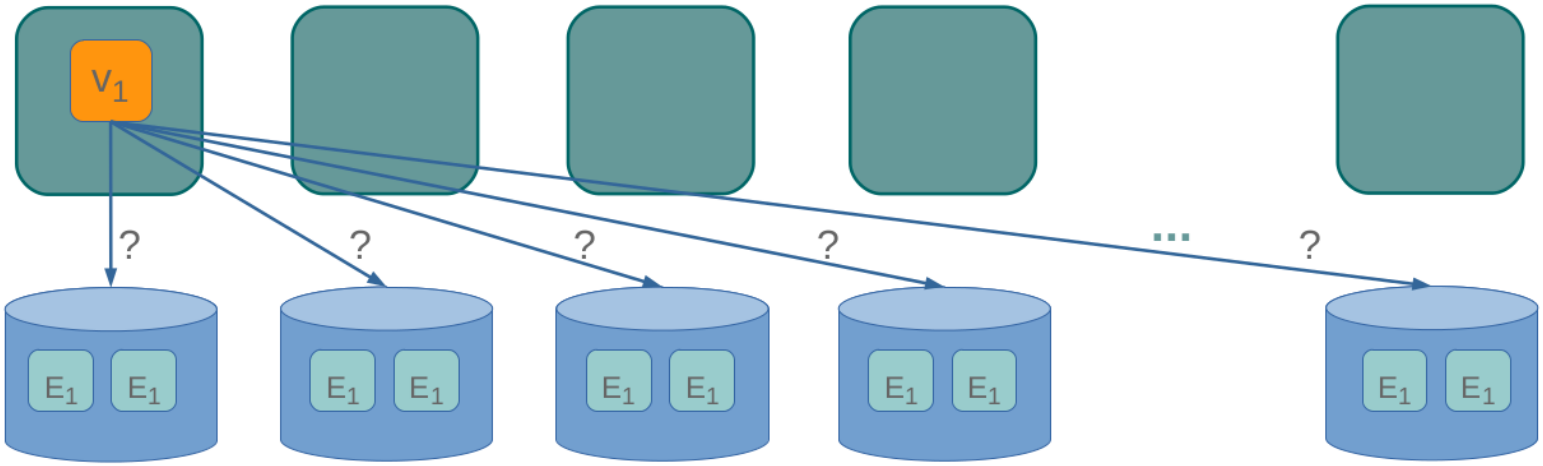
Chunks

- All data structures are accessed in units called chunks.
- Chaos spreads all data structures across the storage engines in an uniform random manner.



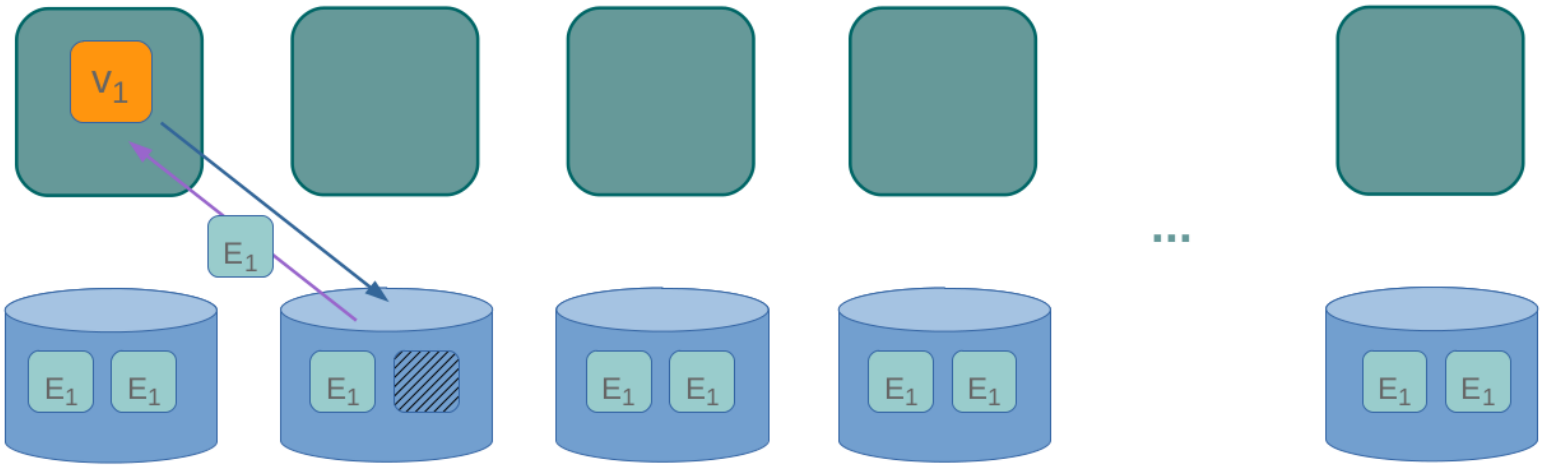
Reading Edge Chunks

- From where to read next edge chunk?



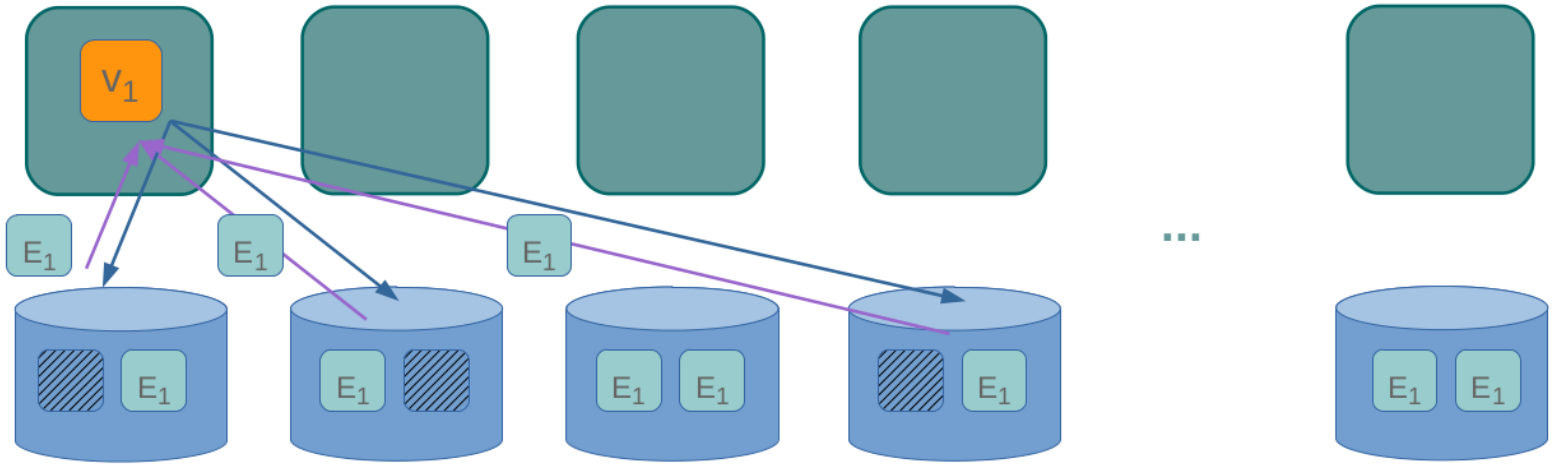
Reading Edge Chunks

- From where to read next edge chunk?
- It can read any random chunk (that has not been read).



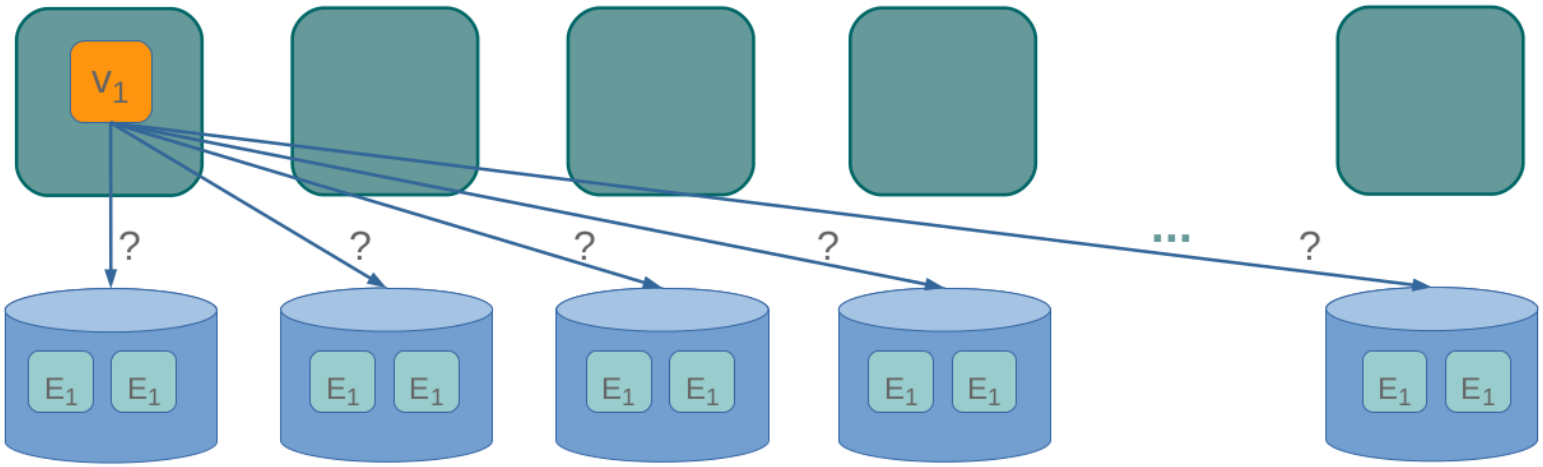
Reading Edge Chunks

- From where to read next edge chunk?
- It can read any random chunk (that has not been read).
- In fact, it reads several random chunks.



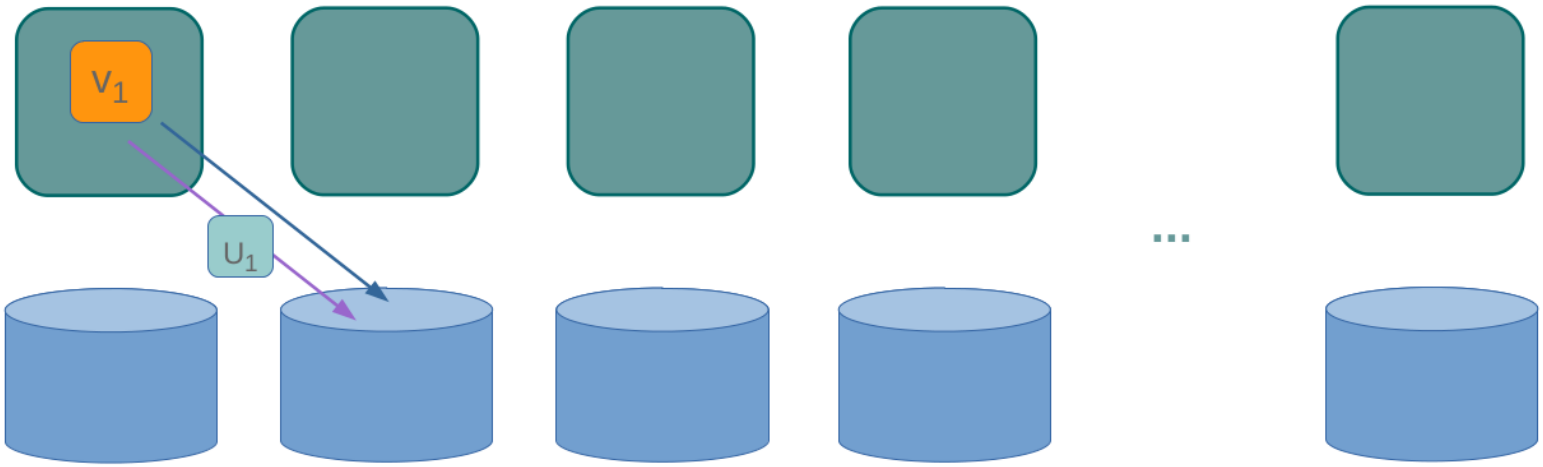
Writing Update Chunks

- Where to write update stripe?



Writing Update Chunks

- Where to write update stripe?
- Choose any device at random.



Computation Sub-System

Computation Sub-System

- One engine on each machine.
- They collectively implement the GAS model.

The Scatter Phase

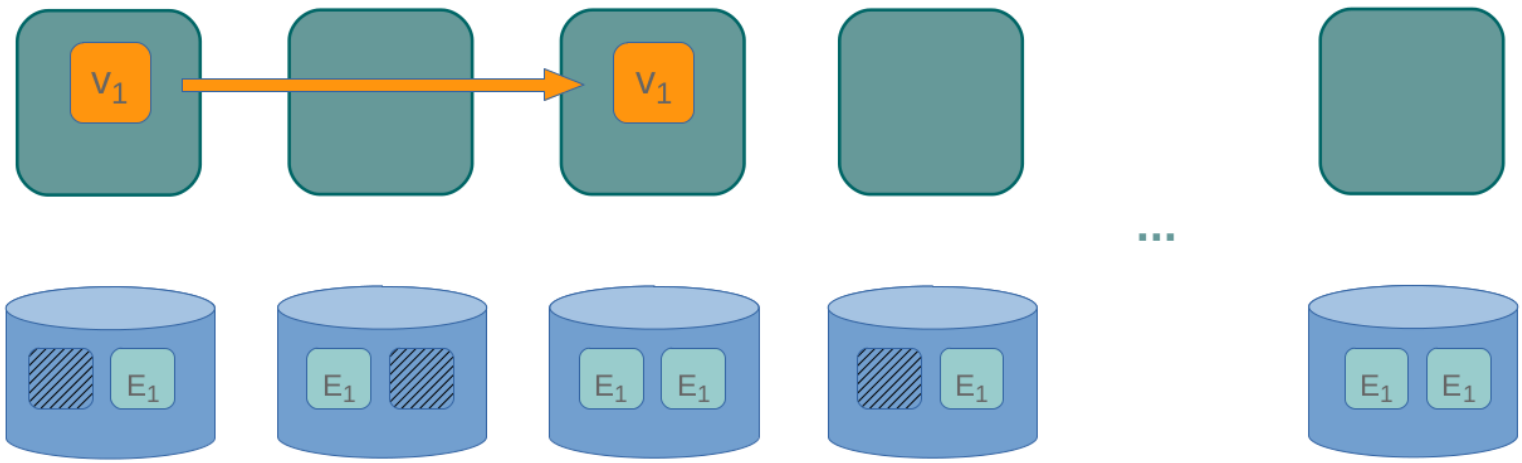
```
// the scatter phase
for each streaming_partition p {
  load Vertices(p)
  exec_scatter(p)
}

def exec_scatter(partition p) {
  for each unprocessed e in Edges(p) {
    u = new update
    u.dst = e.dst
    u.value = f(e.src.value)
    add u to Update(partition(u.dst))
  }
}
```

The Gather Phase

```
// the gather phase
for each streaming-partition p {
  load Vertices(p)
  exec_gather(p)

  // the apply phase
  for all v in Vertices(p)
    Apply(v.value, v.accum)

  delete Update(p)
}

def exec_gather(partition p) {
  for each unprocessed u in Update(p)
    u.dst.accume = g(u.dst.accume, u.value)
}
```

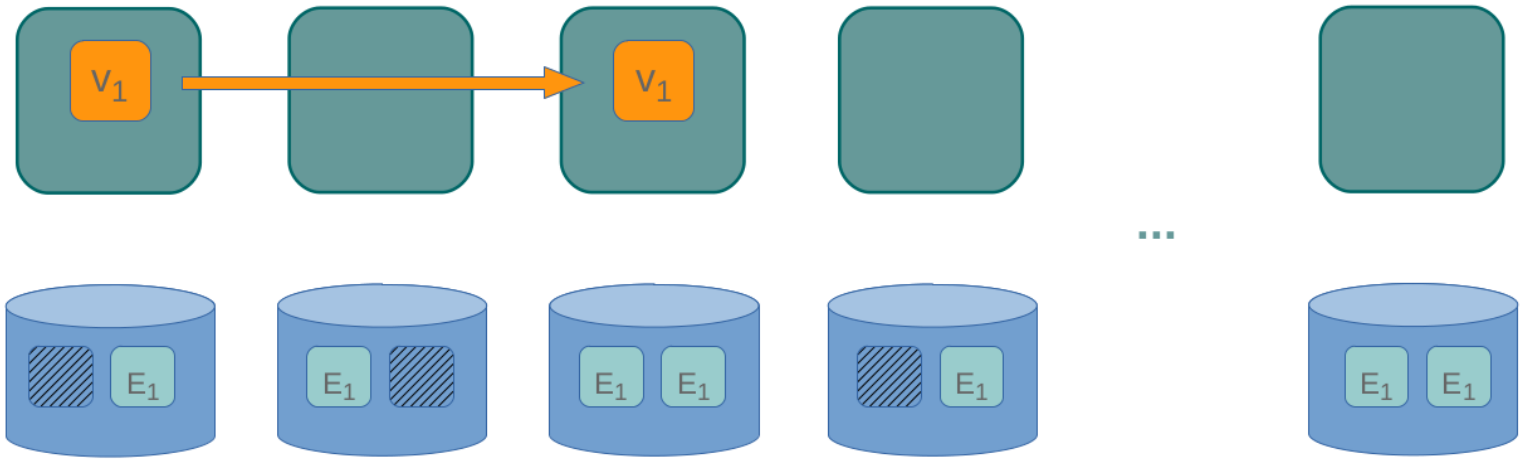
Work Stealing (1/2)

- Work stealing: copy vertex set
- One master for each replicated vertex.



Work Stealing (1/2)

- Work stealing: copy vertex set
- One master for each replicated vertex.
- Work stealing issues:
  - More than one machine work on a streaming partition.
  - More than one access the same edge list.
  - Need for synchronization?



Work Stealing (2/2)

- When computation engine `i` completes the work for its assigned partitions:
  - It goes through every partition `p` (for which it is not the master).
  - Sends a proposal to help out with `p` to its master `j`.
  - The proposal may be accepted or rejected.

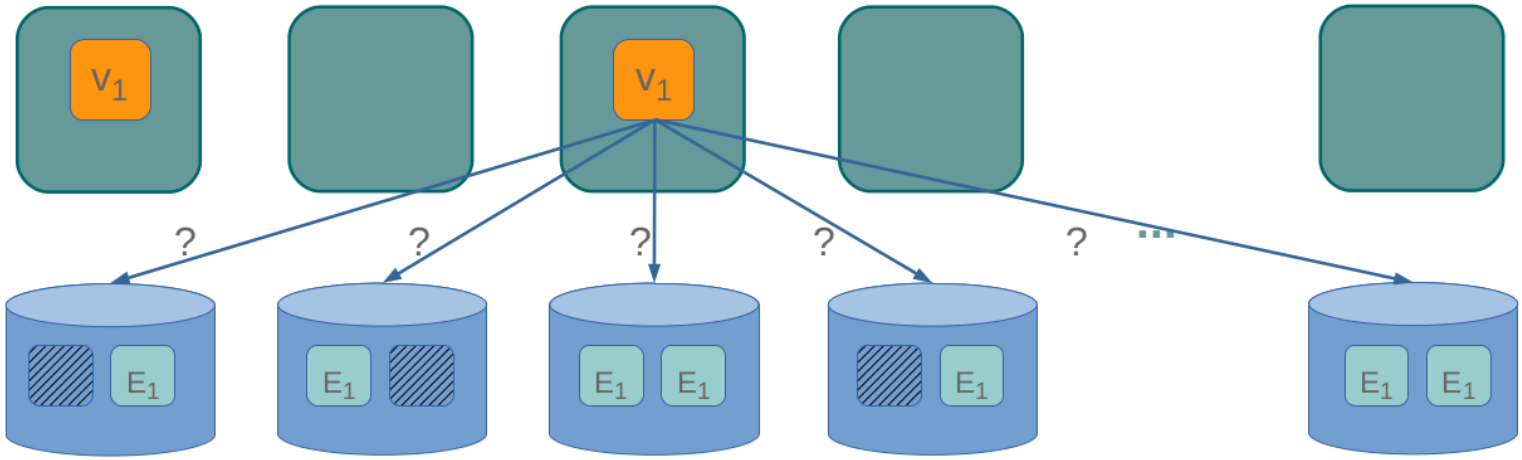
Work Stealing (2/2)

- When computation engine `i` completes the work for its assigned partitions:
  - It goes through every partition `p` (for which it is not the master).
  - Sends a proposal to help out with `p` to its master `j`.
  - The proposal may be accepted or rejected.
- Rejected proposal: it continues to iterate the partitons until it receives a positive response or until it has determined that no help is needed for any of the partitions.

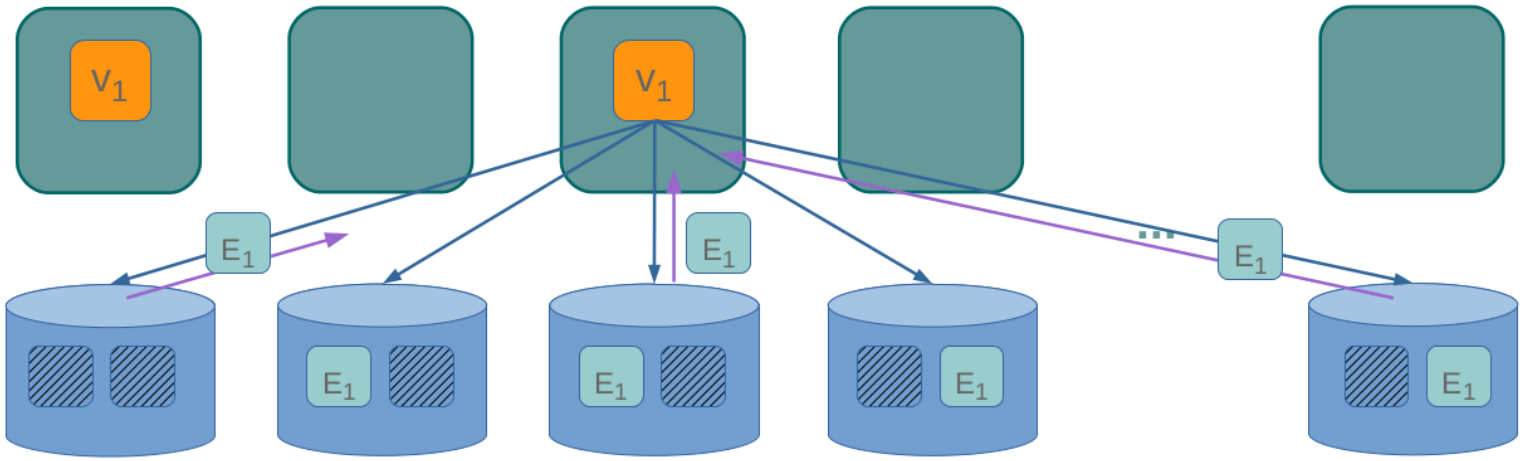
Work Stealing (2/2)

- When computation engine `i` completes the work for its assigned partitions:
  - It goes through every partition `p` (for which it is not the master).
  - Sends a proposal to help out with `p` to its master `j`.
  - The proposal may be accepted or rejected.
- Rejected proposal: it continues to iterate the partitons until it receives a positive response or until it has determined that no help is needed for any of the partitions.
- Accepted proposal: it reads the vertex set of that partition from storage into its memory, and starts working on it.

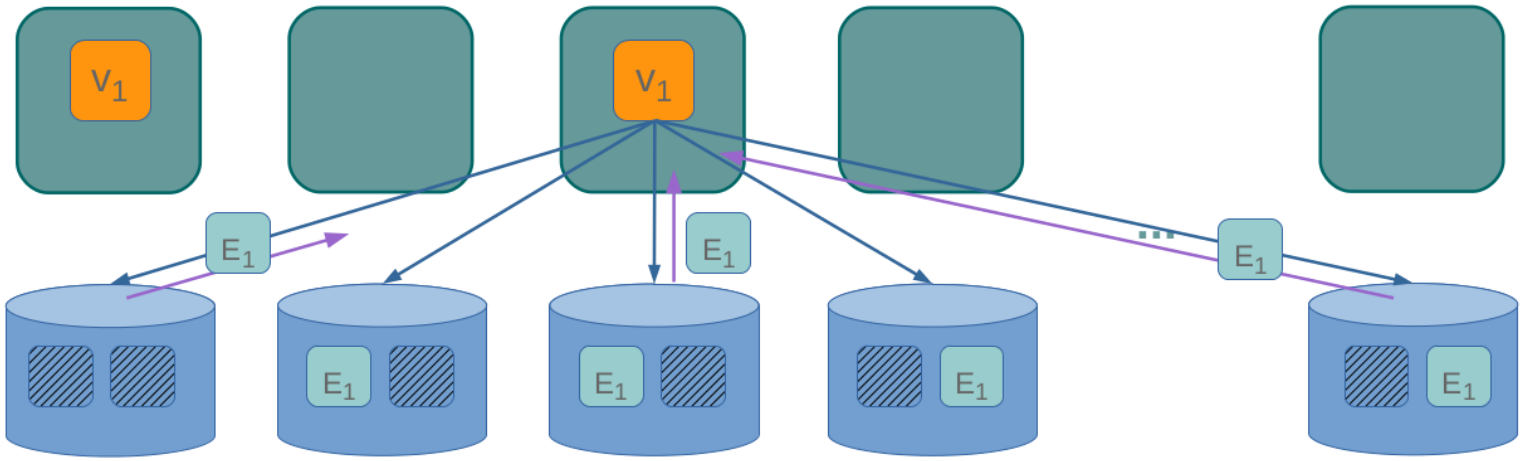
- Which edge chunk to read?



- Which edge chunk to read?
- It can read any chunk (that has not been read)



- Which edge chunk to read?
- It can read any chunk (that has not been read)
- Storage sub-system maintains what has and has not been read: no synchronization



The Scatter Phase With Work Stealing

```
// the scatter phase
for each streaming_partition p {
  load Vertices(p)
  exec_scatter(p)
}

// when done with my partitions, steal from others
for every partition p_stolen not belonging to me {
  if need_help(Master(p_stolen))
    load Vertices(p_stolen)
    exec_scatter(p_stolen)
}
```

The Gather Phase With Work Stealing

```
// the gather phase
for each streaming-partition p {
  load Vertices(p)
  exec_gather(p)

  // the apply phase
  for all stealers s
    accumulators = get_accums(s)
    for all v in Vertices(p)
      Apply(v.value, accumulators(v))

  delete Update(p)
}

// when done with my partitions, steal from others
for every partition p_stolen not belonging to me
  if need_help(Master(p_stolen))
    load Vertices(p_stolen)
    exec_gather(p_stolen)
    wait for get_accums(p_stolen)
}
```

Summary

Summary

- X-Stream
  - Single machine
  - GAS model
  - Edge-centric
  - Streaming partitioning
- Chaos
  - X-Stream on a cluster
  - I/O imbalance: flat storage (storage sub-system)
  - Computation imbalance: work strealing (computation sub-system)

Questions?

Acknowledgement

Some slides were derived from the slides of Amitabha Roy (EPFL)