Large-Scale Graph Processing on Emerging Storage Devices

Nima Elyasi[1], Changho Choi[2], Anand Sivasubramaniam[1]

[1]Pennsylvania State University

[2]Samsung Semiconductor Inc.

Graph Processing is Commonplace

Search Engines

Google

bing

Social Media

facebook

Recommendations and Ads

amazon

NETFLIX

Map and Navigation

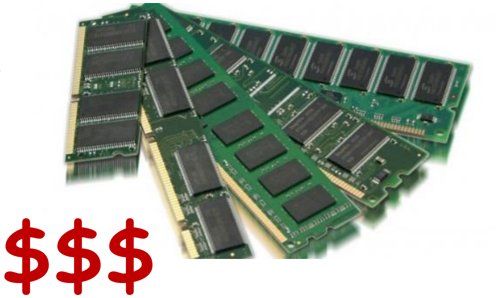
Large-Scale Graph Processing Challenges

Huge Datasets

Irregular Accesses

High cost of DRAM

DRAM



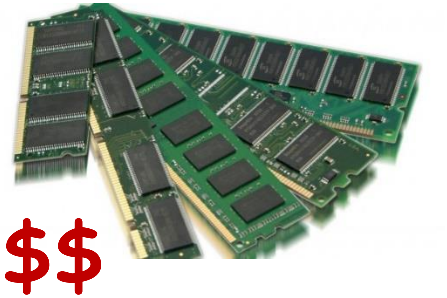
$$$$

Large-Scale Graph Processing Challenges

Huge Datasets

High cost of DRAM

DRAM



$$$$

External Graph Processing is Desirable

$

NVMe SSD

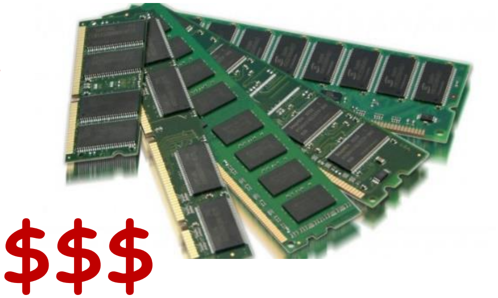
Irregular Accesses

Large-Scale Graph Processing Challenges

Huge Datasets

High cost of DRAM

DRAM



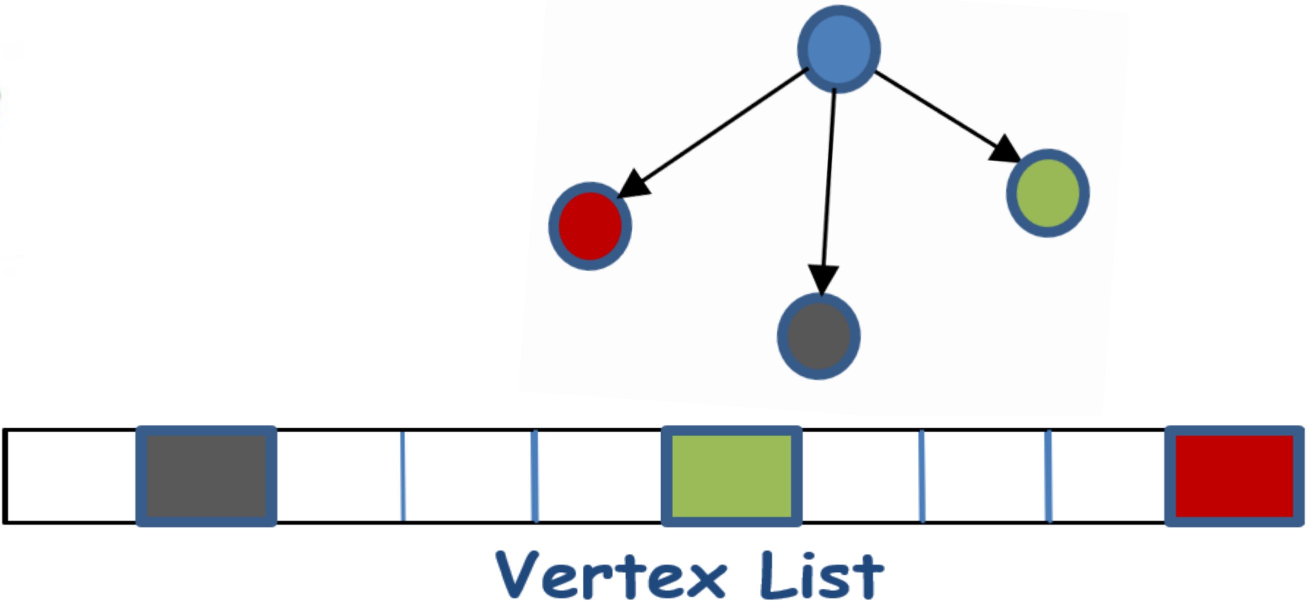
$$$$

External Graph Processing is Desirable

$

NVMe SSD

Irregular Accesses

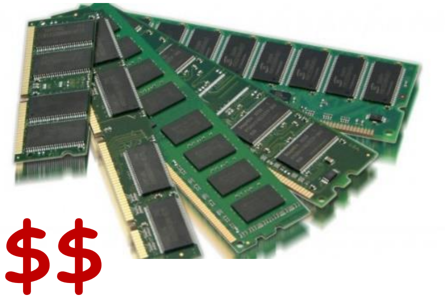


Large-Scale Graph Processing Challenges

Huge Datasets

High cost of DRAM

DRAM



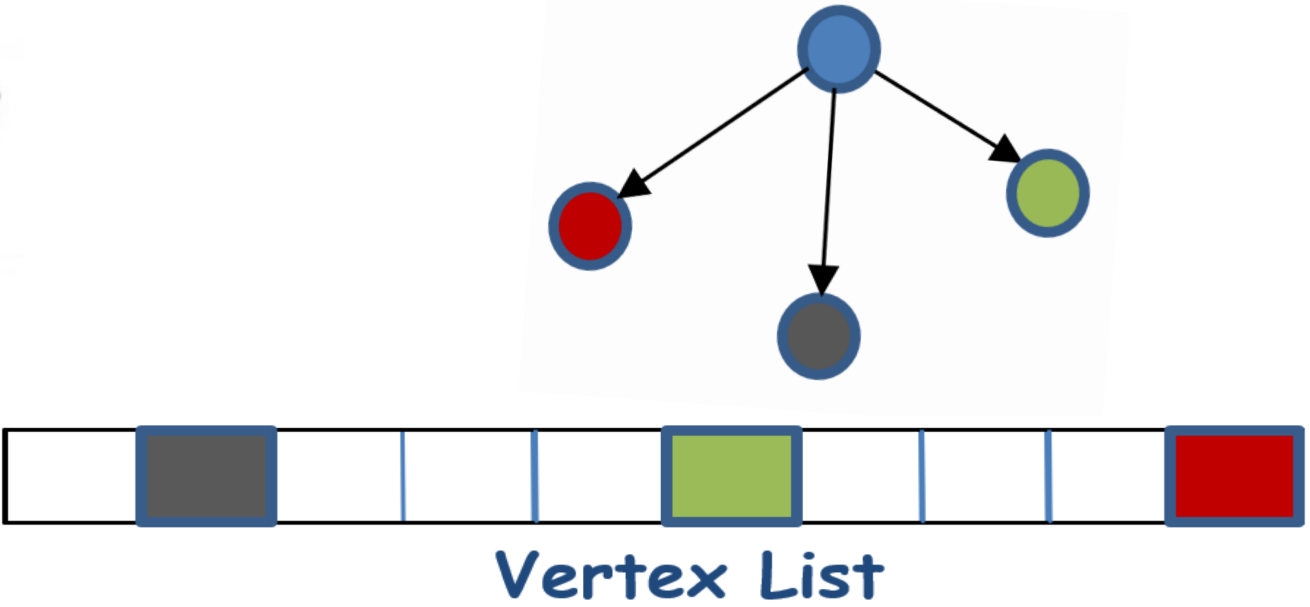
$$$$

External Graph Processing is Desirable

$

NVMe SSD

Irregular Accesses

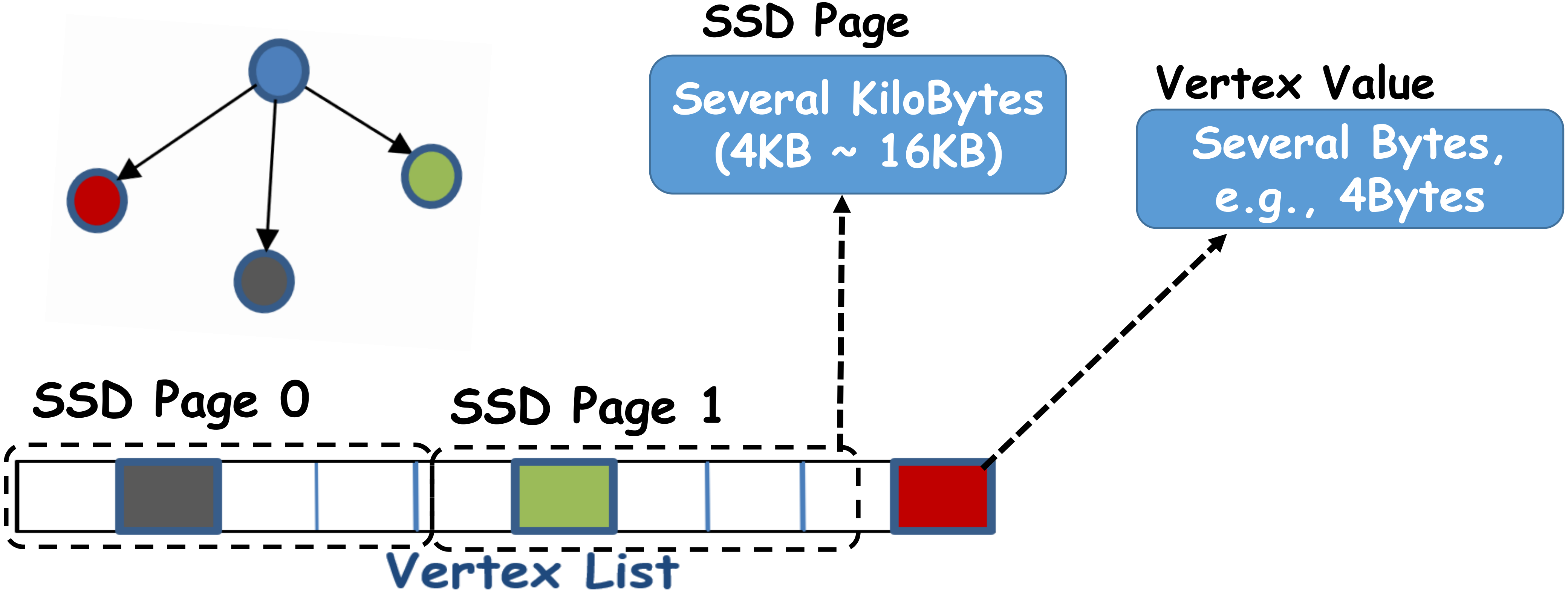


Fine-Grained and Random Accesses

Fine-Grained Access in External Graph Processing

Irregular Accesses

SSD Page Size and Vertex Accesses Don't Match!



SSD Page

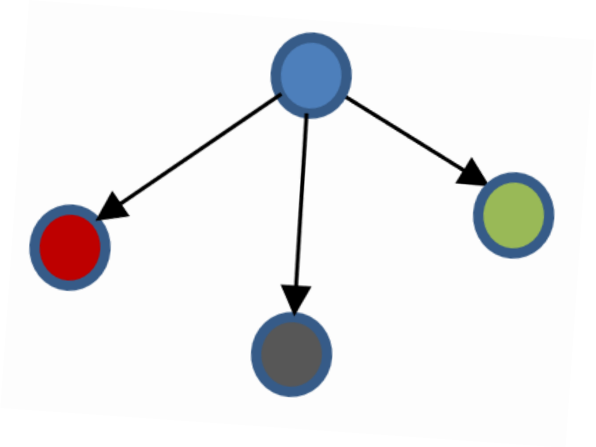
Several KiloBytes (4KB ~ 16KB)

Vertex Value

Several Bytes, e.g., 4Bytes

Fine-Grained Access in External Graph Processing

Irregular Accesses



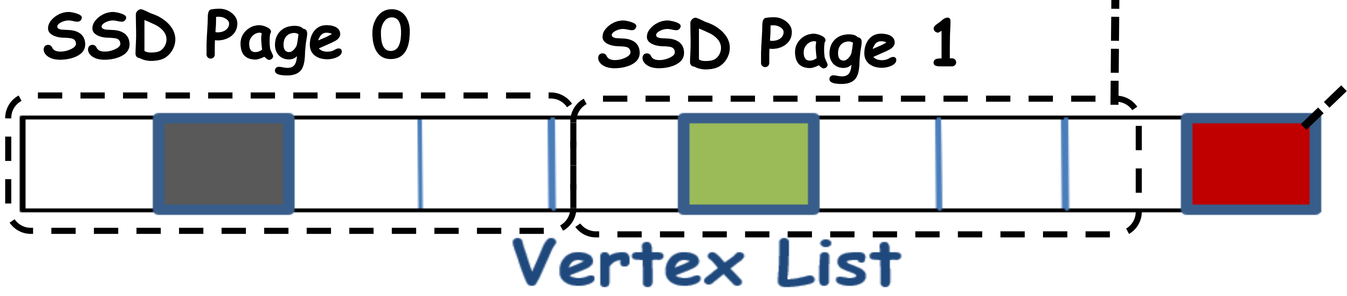
SSD Page Size and Vertex Accesses Don't Match!

SSD Page

Several KiloBytes (4KB ~ 16KB)

Vertex Value

Several Bytes, e.g., 4Bytes



Vertex updates are detrimental to:

Performance

Device Endurance

Providing Perfect Sequentiality as a Remedy

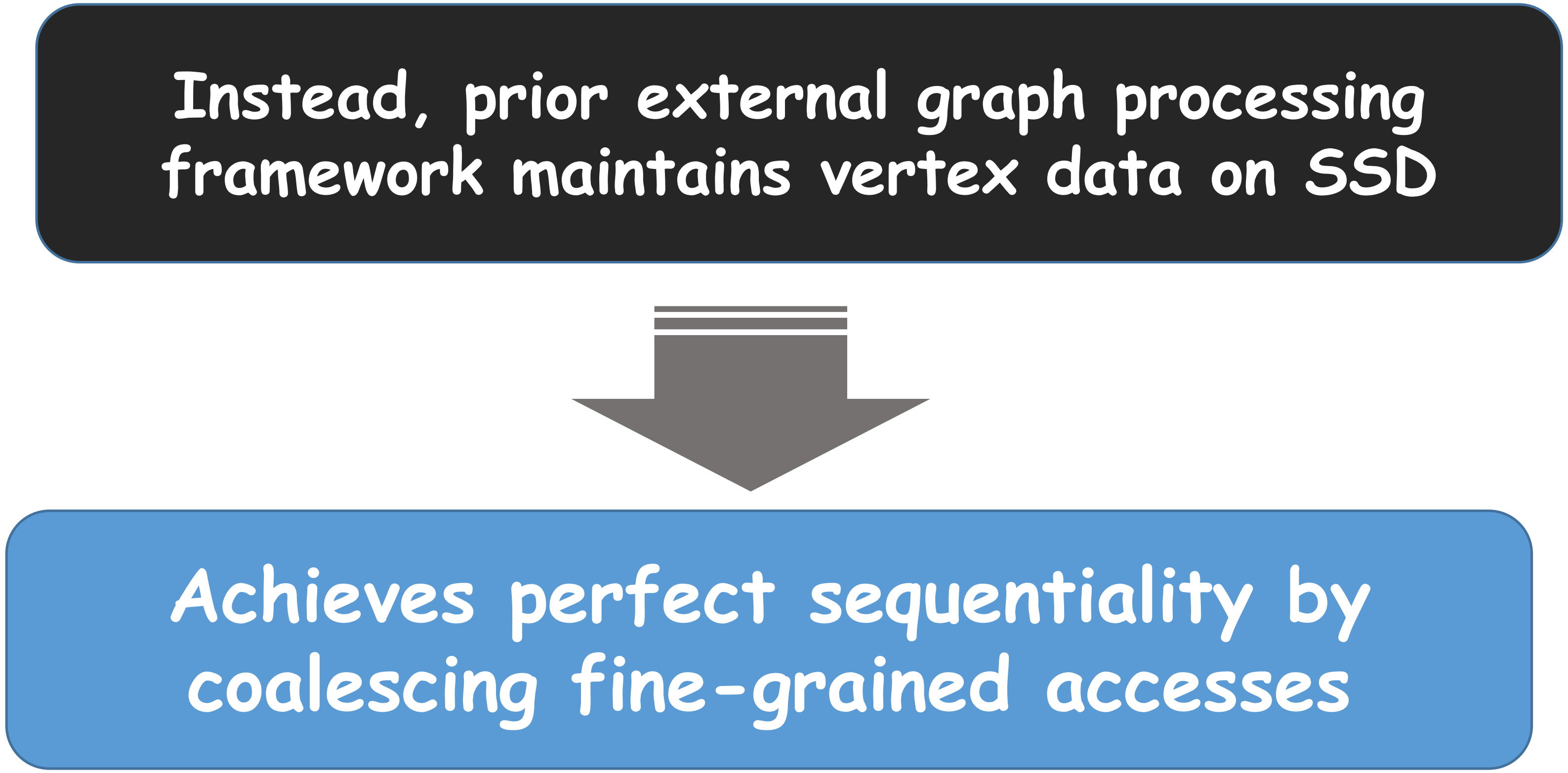
- If vertex data could be stored on DRAM
    - Fine-grained accesses was less of an issue

Instead, prior external graph processing framework maintains vertex data on SSD

GraFBoost, ISCA'18

Providing Perfect Sequentiality as a Remedy

- **If vertex data could be stored on DRAM**
    - Fine-grained accesses was less of an issue



Instead, prior external graph processing framework maintains vertex data on SSD

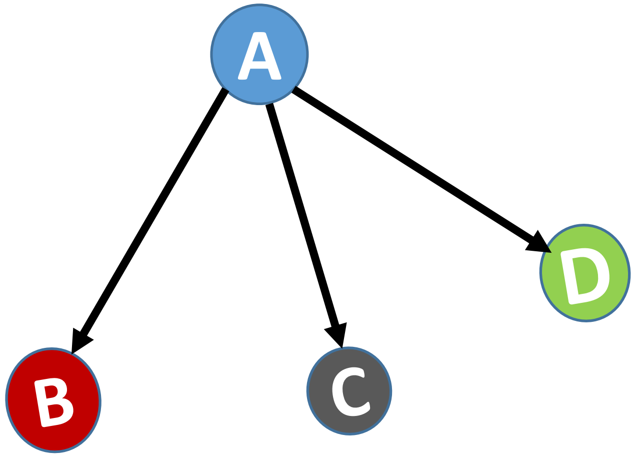
Achieves perfect sequentiality by coalescing fine-grained accesses

GraFBoost, ISCA'18

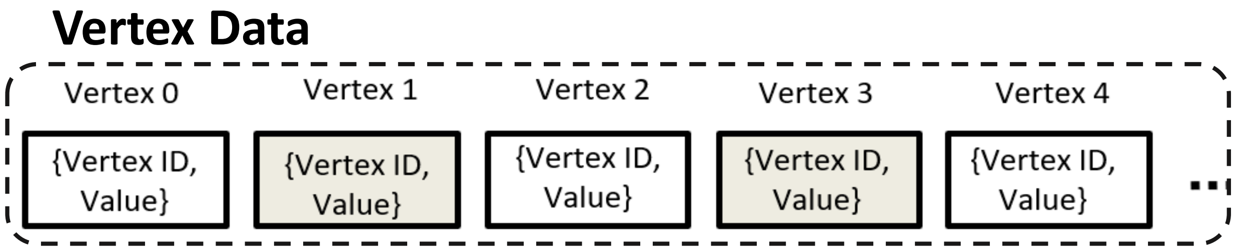
Programming Model

Vertex-centric Programming Model

- Iterative programming model
- Each vertex runs a user-defined program
- Sending updates to neighbors along outgoing edges



Prior External Graph Processing -- GraFBoost



Vertex Data

Vertex 0 {Vertex ID, Value}

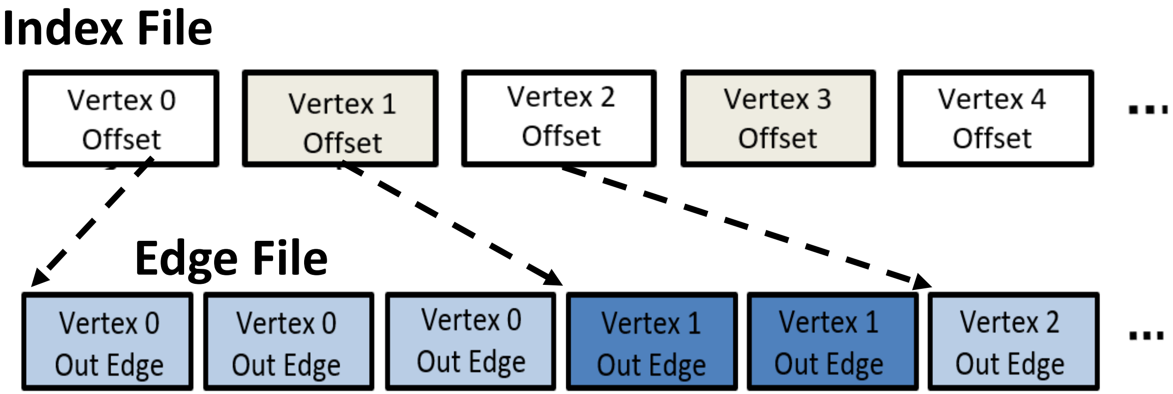
Vertex 1 {Vertex ID, Value}

Vertex 2 {Vertex ID, Value}

Vertex 3 {Vertex ID, Value}

Vertex 4 {Vertex ID, Value}

…



Index File

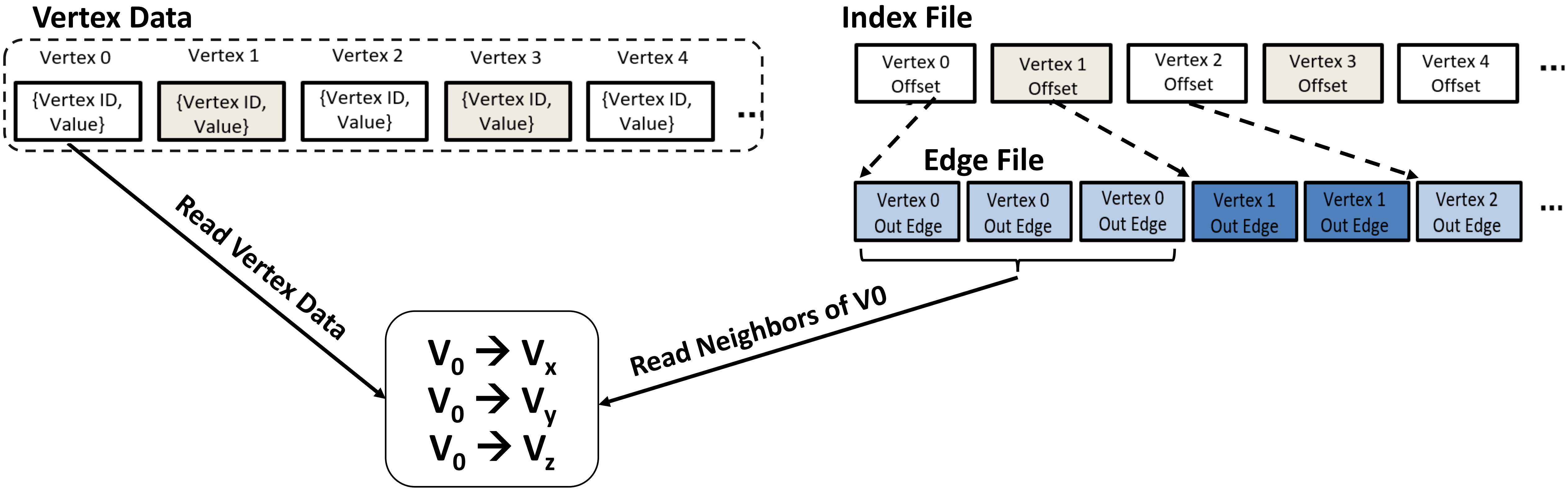
Vertex 0 Offset | Vertex 1 Offset | Vertex 2 Offset | Vertex 3 Offset | Vertex 4 Offset …

Edge File

Vertex 0 Out Edge | Vertex 0 Out Edge | Vertex 0 Out Edge | Vertex 1 Out Edge | Vertex 1 Out Edge | Vertex 2 Out Edge …

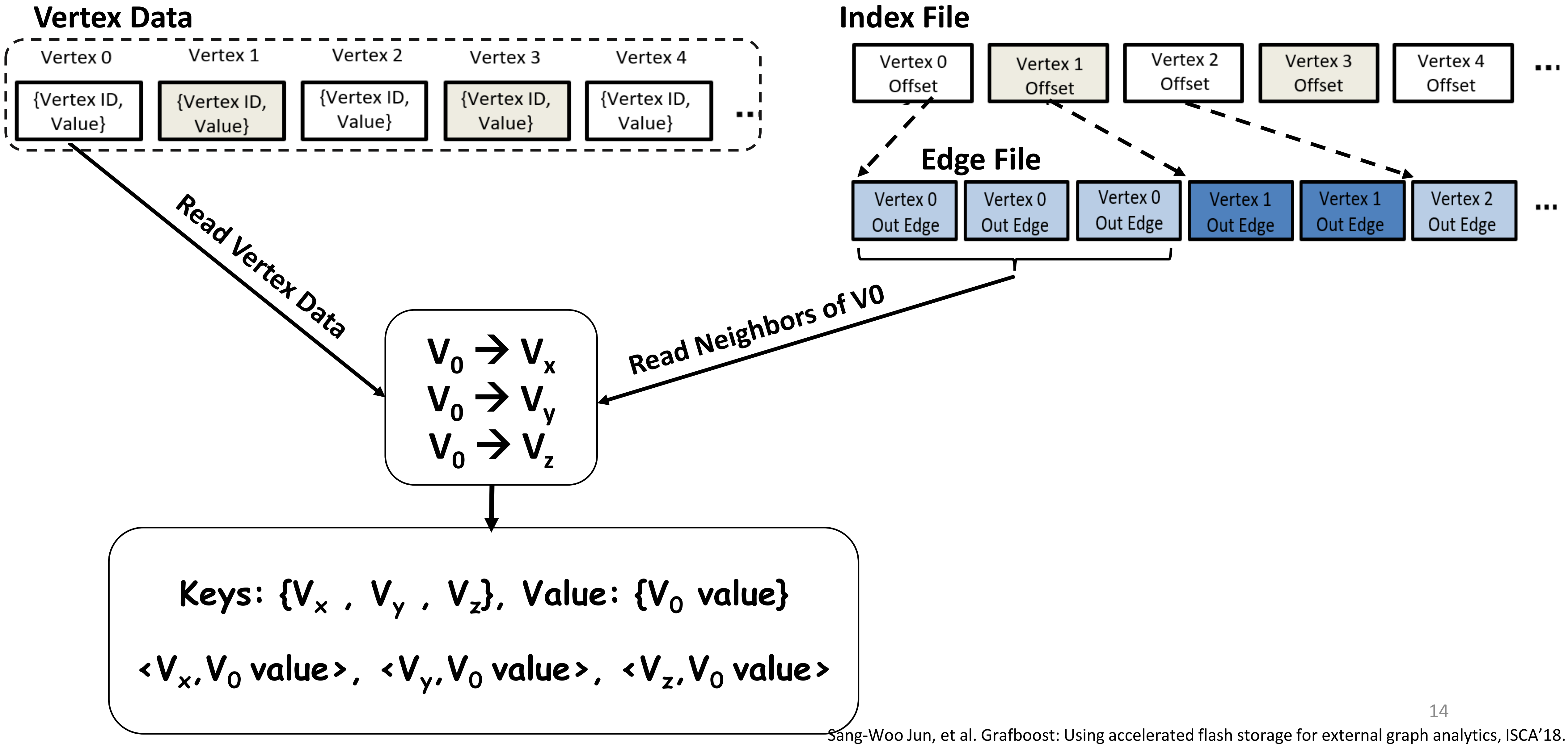
Sang-Woo Jun, et al. Grafboost: Using accelerated flash storage for external graph analytics, ISCA’18.

Prior External Graph Processing -- GraFBoost



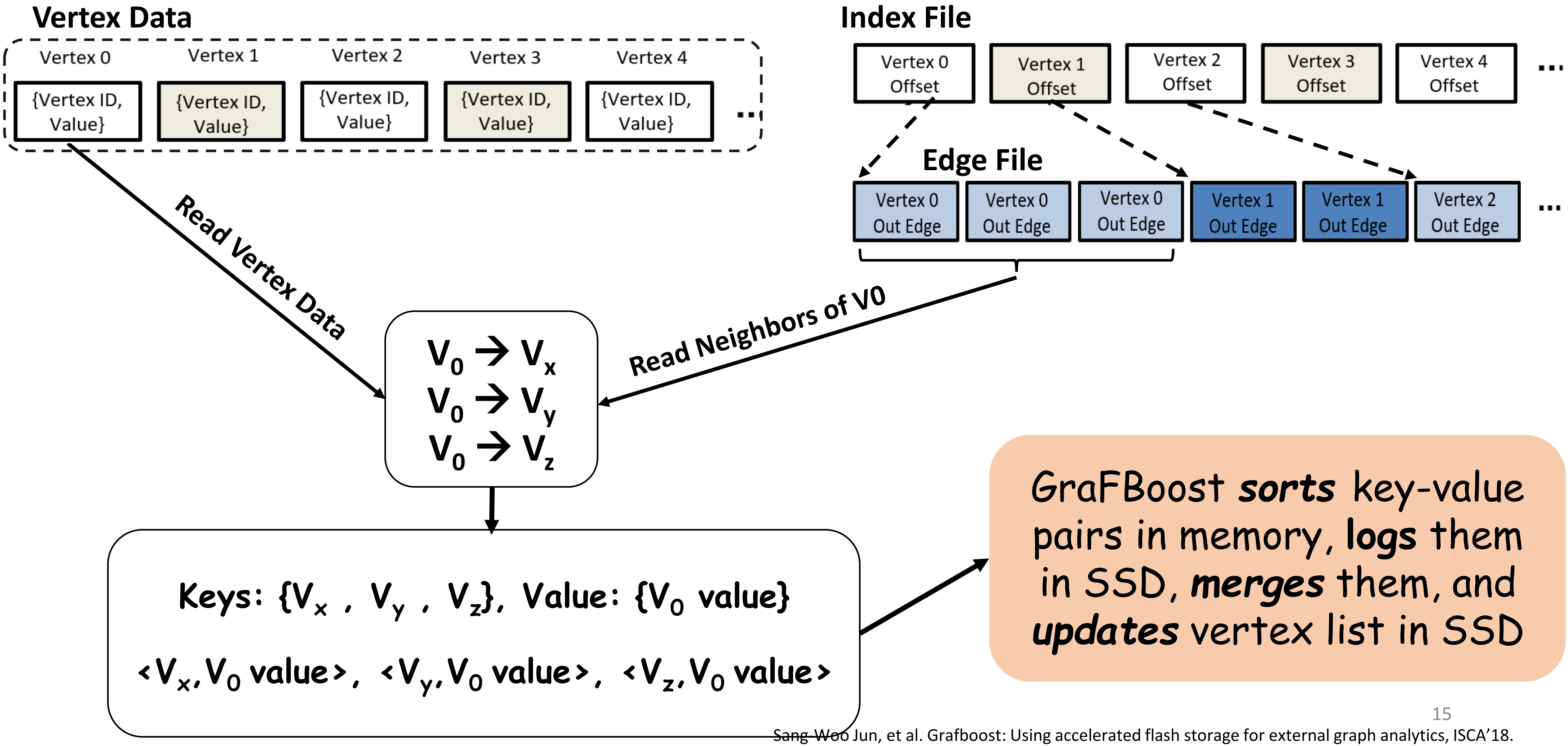
Sang-Woo Jun, et al. Grafboost: Using accelerated flash storage for external graph analytics, ISCA’18.

Prior External Graph Processing -- GraFBoost



Sang-Woo Jun, et al. Grafboost: Using accelerated flash storage for external graph analytics, ISCA'18.

Prior External Graph Processing -- GraFBoost



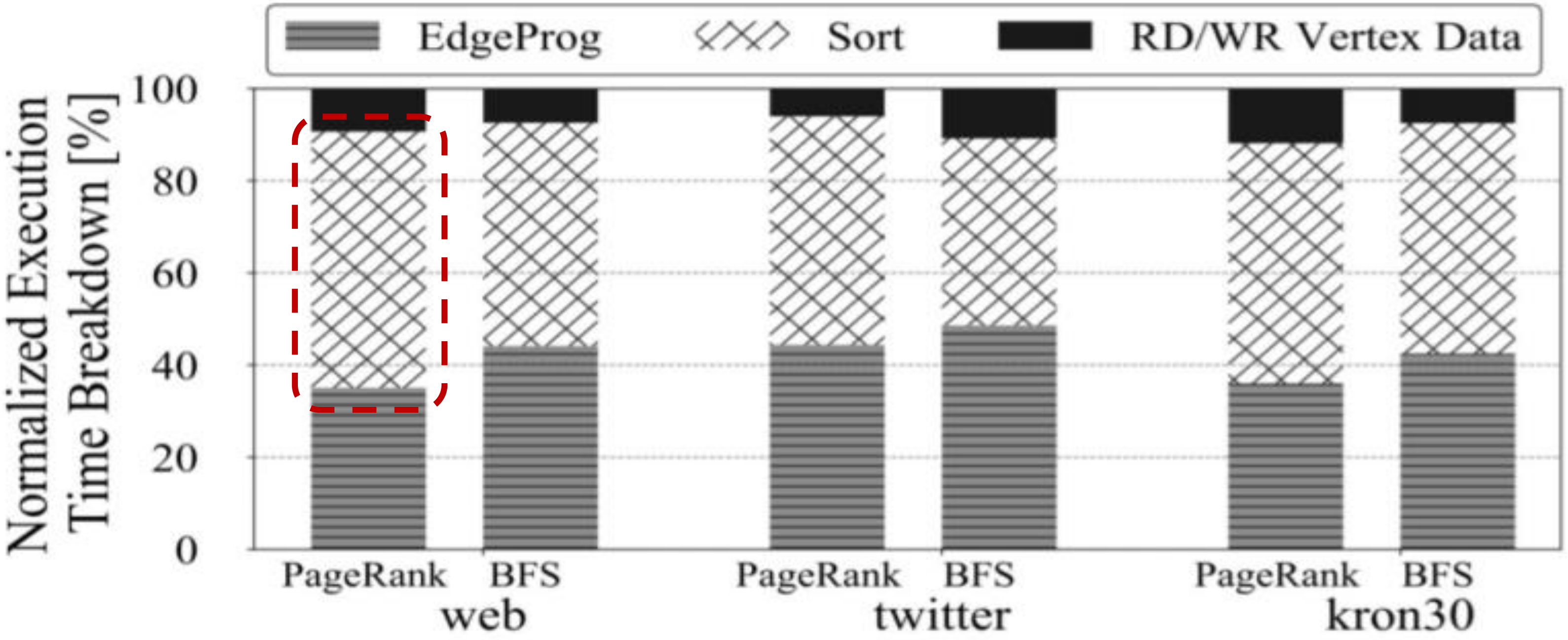
Keys: $\{V_x , V_y , V_z\}$, Value: $\{V_0$ value$\}$

$\langle V_x, V_0$ value$\rangle$, $\langle V_y, V_0$ value$\rangle$, $\langle V_z, V_0$ value$\rangle$

GraFBoost ***sorts*** key-value pairs in memory, **logs** them in SSD, ***merges*** them, and ***updates*** vertex list in SSD

Sang-Woo Jun, et al. Grafboost: Using accelerated flash storage for external graph analytics, ISCA'18.

Computation Overhead of Sort!

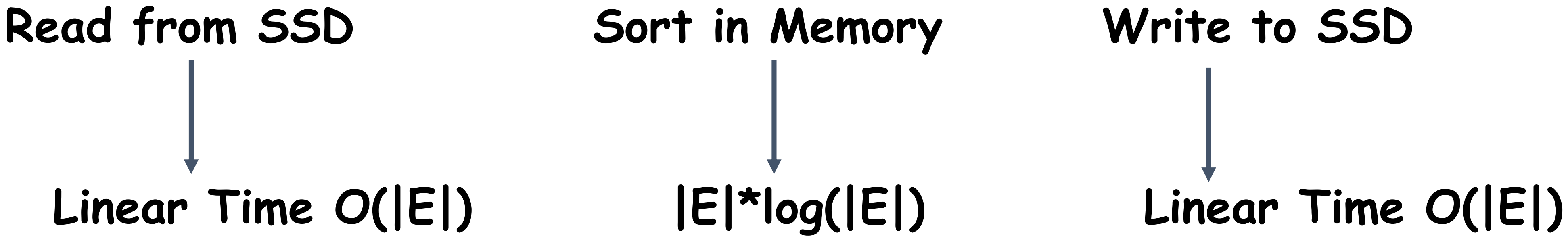


- Up to 60% sort overhead (web graph)

- Higher sort overhead for PageRank
  - Processes all vertices in each iteration and generates more updates

Scalability Issue

Current External Graph Processing:



Read from SSD

Linear Time O(|E|)

Sort in Memory

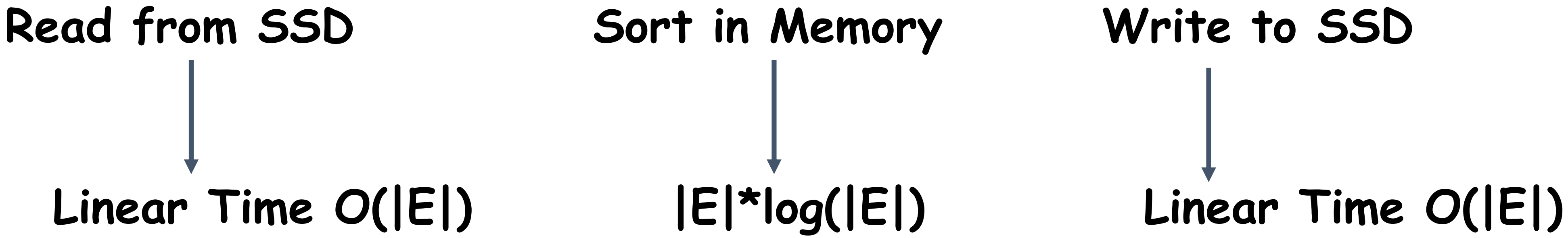
|E|*log(|E|)

Write to SSD

Linear Time O(|E|)

Scalability Issue

Current External Graph Processing:



Read from SSD → Linear Time O(|E|)

Sort in Memory → |E|*log(|E|)

Write to SSD → Linear Time O(|E|)

Assuming DRAM "k" times faster than SSD (e.g., k=30):

When k < log(|E|) → Sorting can become bottleneck

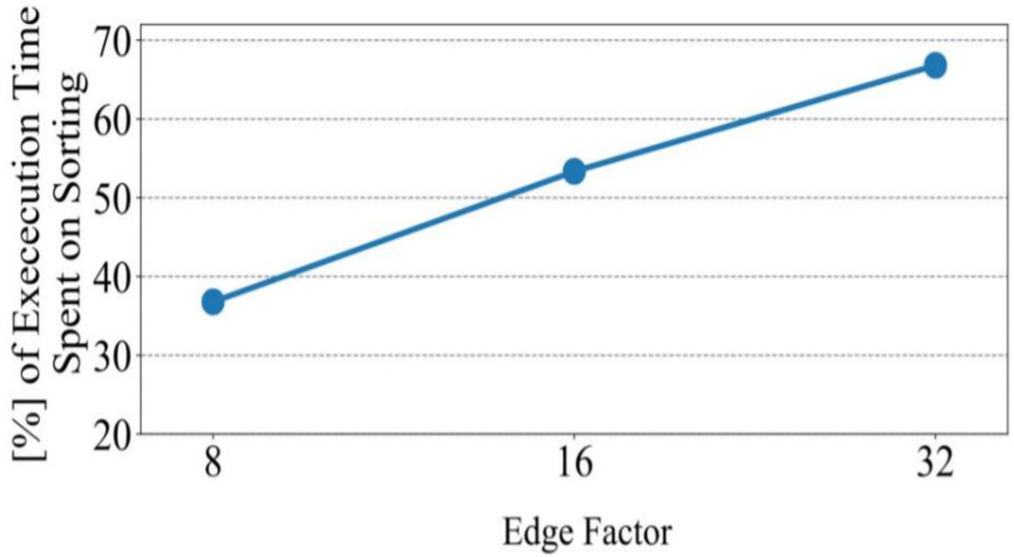
Scalability Issue

Current External Graph Processing:

Read from SSD	Sort in Memory	Write to SSD
↓	↓	↓
Linear Time O(|E|)	|E|*log(|E|)	Linear Time O(|E|)

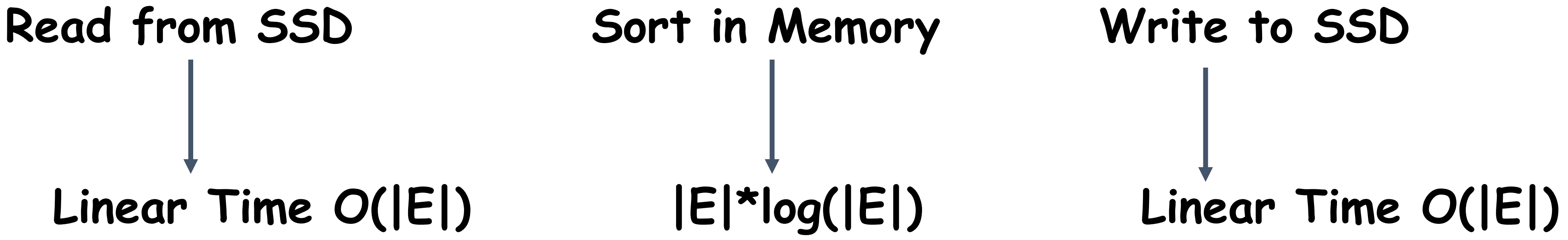
Assuming DRAM "k" times faster than SSD (e.g., k=30):

When k < log(|E|) → Sorting can become bottleneck



Scalability Issue

Current External Graph Processing:



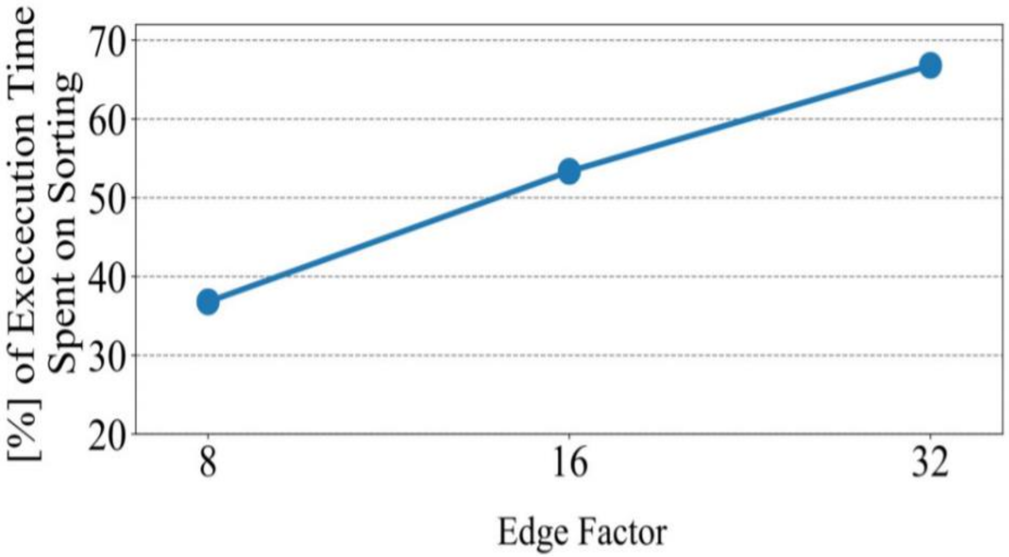
Read from SSD → Linear Time O(|E|)

Sort in Memory → |E|*log(|E|)

Write to SSD → Linear Time O(|E|)

Assuming DRAM "k" times faster than SSD (e.g., k=30):

When k < log(|E|) → Sorting can become bottleneck

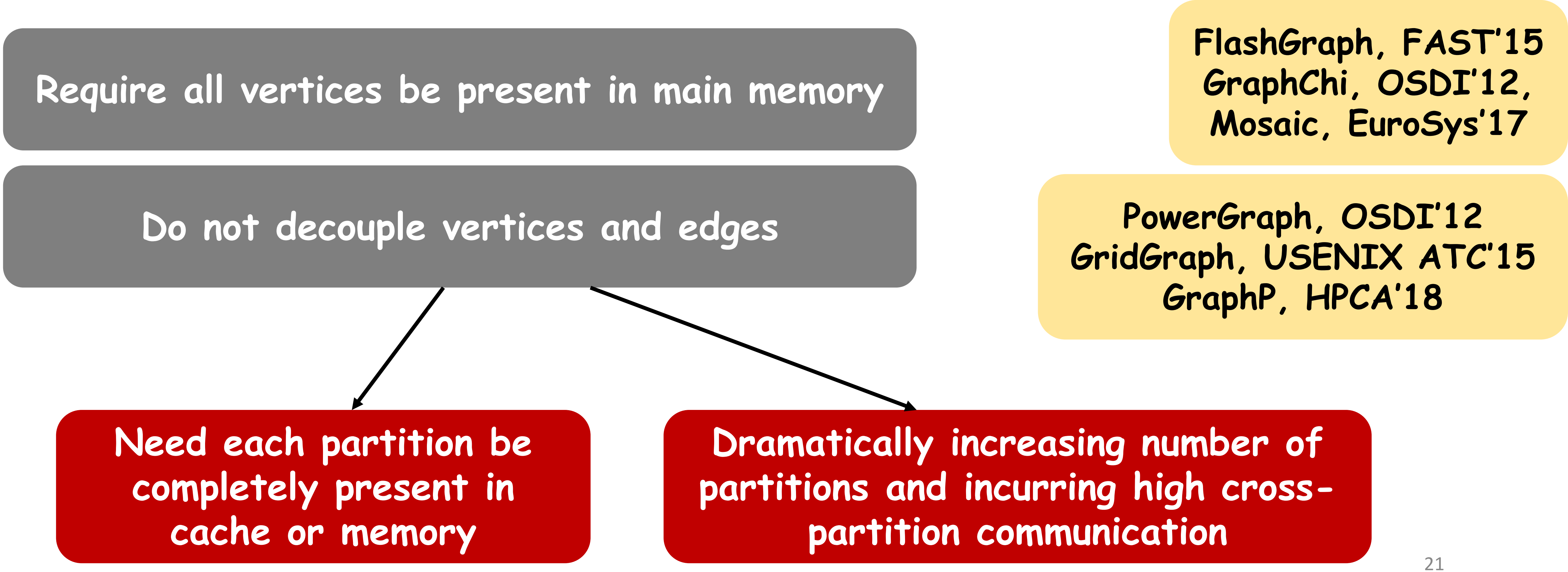


Instead, we propose a vertex partitioning to eliminate the sorting

Partitioning Graph Data

Extensive Prior Efforts on Partitioning Graph Data:

- Not well suited for fully external graph processing



Require all vertices be present in main memory

FlashGraph, FAST'15
GraphChi, OSDI'12,
Mosaic, EuroSys'17

Do not decouple vertices and edges

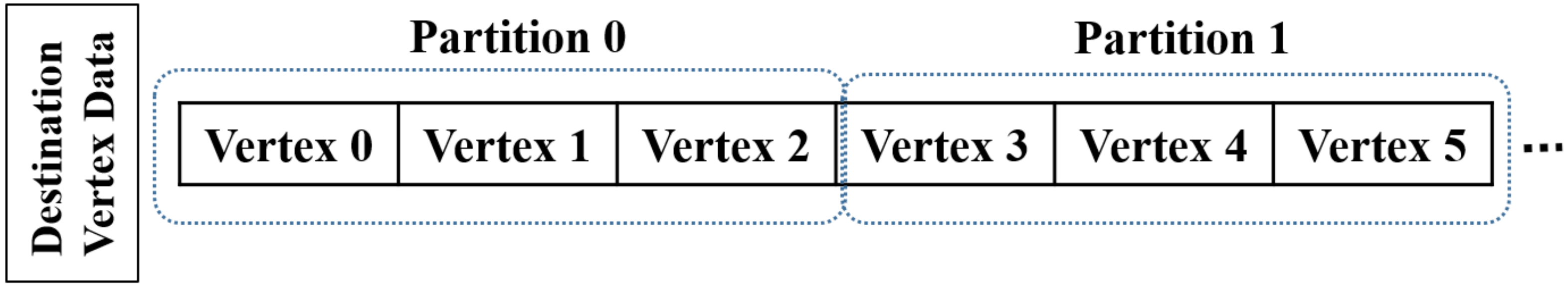
PowerGraph, OSDI'12
GridGraph, USENIX ATC'15
GraphP, HPCA'18

Need each partition be completely present in cache or memory

Dramatically increasing number of partitions and incurring high cross-partition communication

Instead, We Propose a Partitioning for Vertex Data

Reorganizing graph data so that vertices associated with each partition can fit in main memory

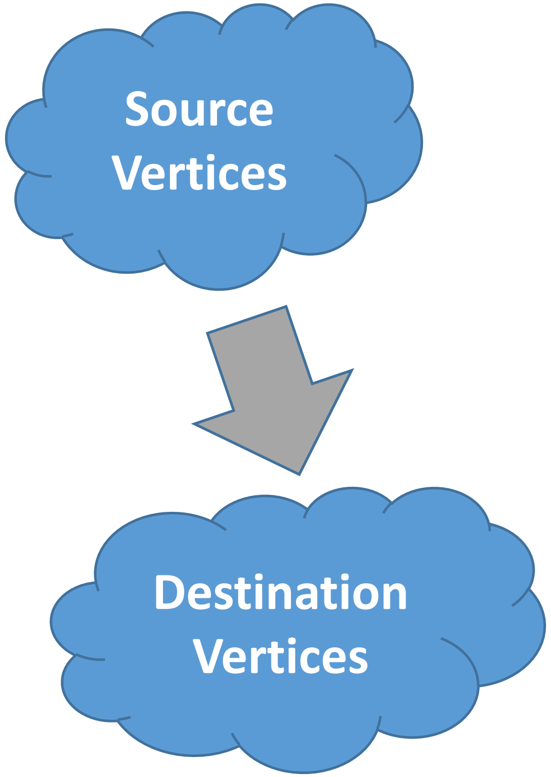


Destination Vertex Data

Partition 0

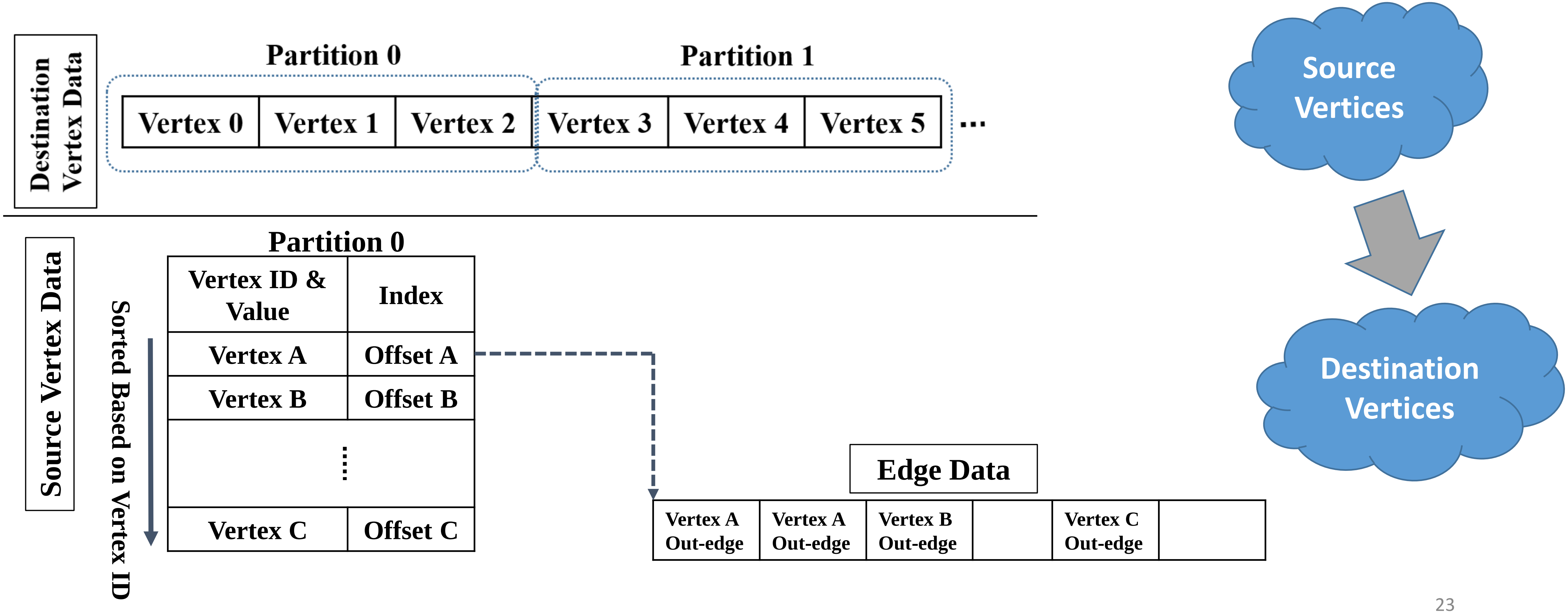
Partition 1

Vertex 0	Vertex 1	Vertex 2	Vertex 3	Vertex 4	Vertex 5	…



Instead, We Propose a Partitioning for Vertex Data

Reorganizing graph data so that vertices associated with each partition can fit in main memory



Destination Vertex Data

Partition 0			Partition 1			
Vertex 0	Vertex 1	Vertex 2	Vertex 3	Vertex 4	Vertex 5	…

Source Vertex Data

Sorted Based on Vertex ID

Partition 0

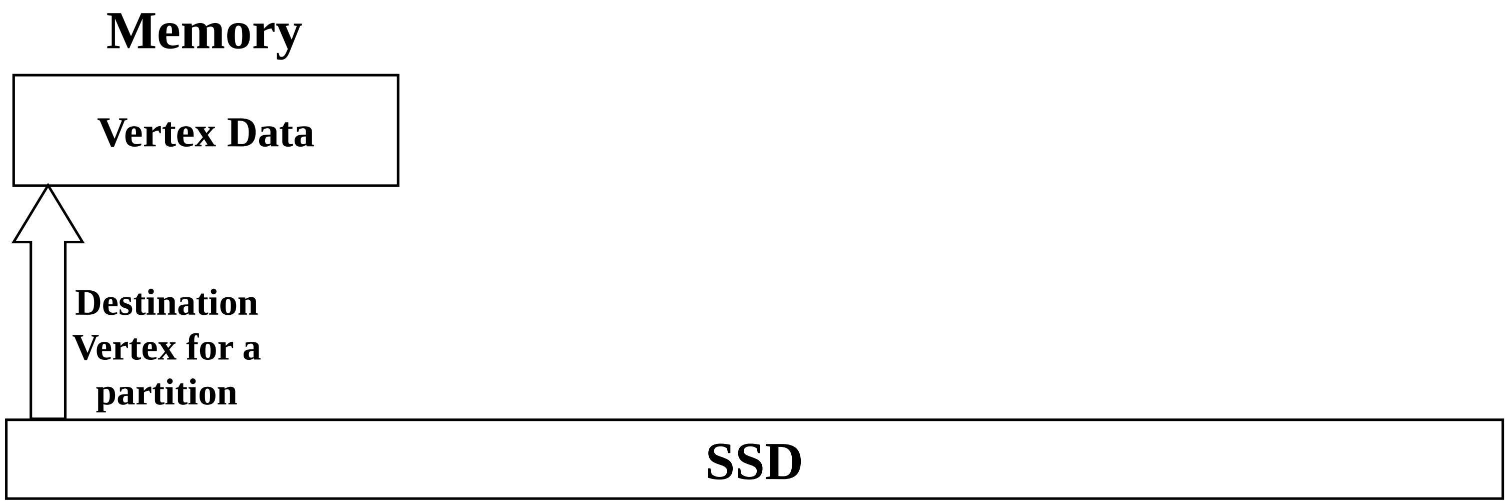
Vertex ID & Value	Index
Vertex A	Offset A
Vertex B	Offset B
⋮	
Vertex C	Offset C

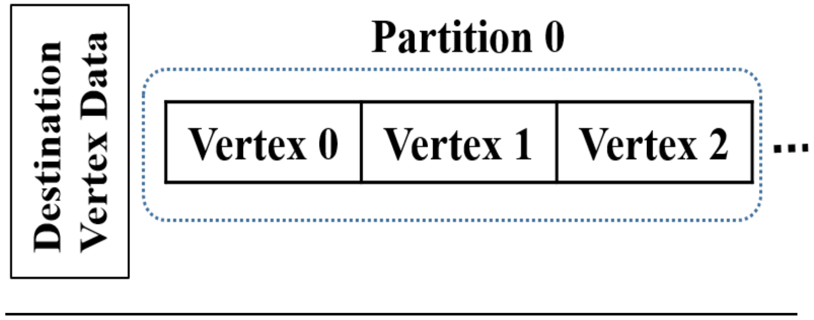
Edge Data

Vertex A Out-edge	Vertex A Out-edge	Vertex B Out-edge		Vertex C Out-edge	

Execution Flow

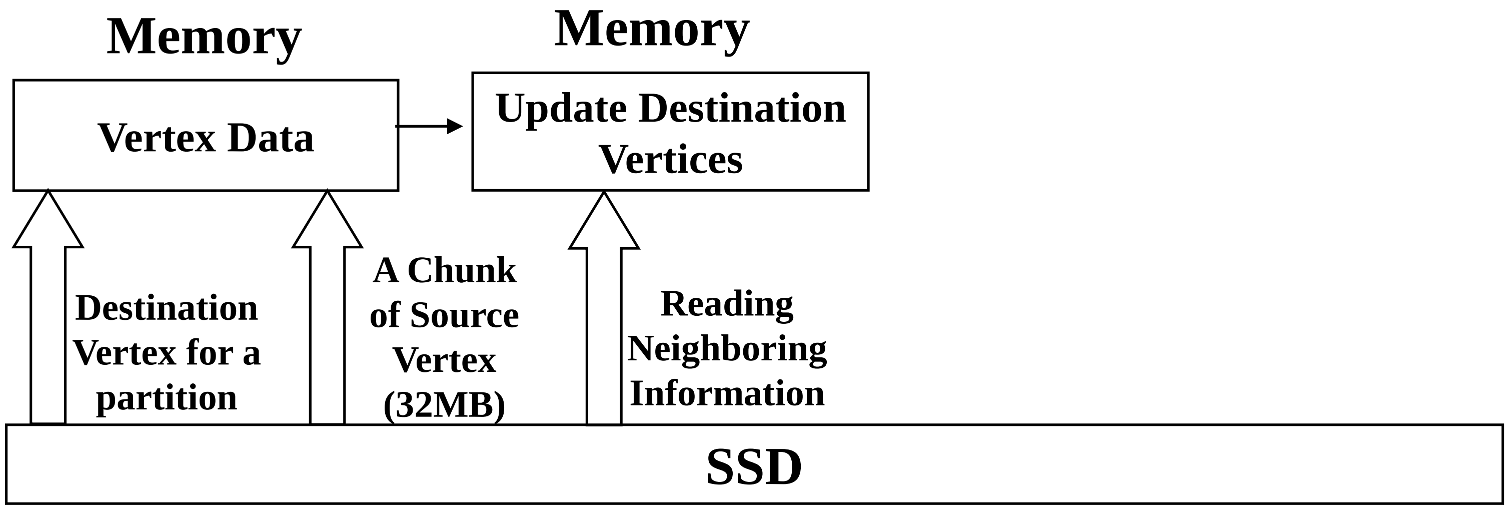
In each iteration:

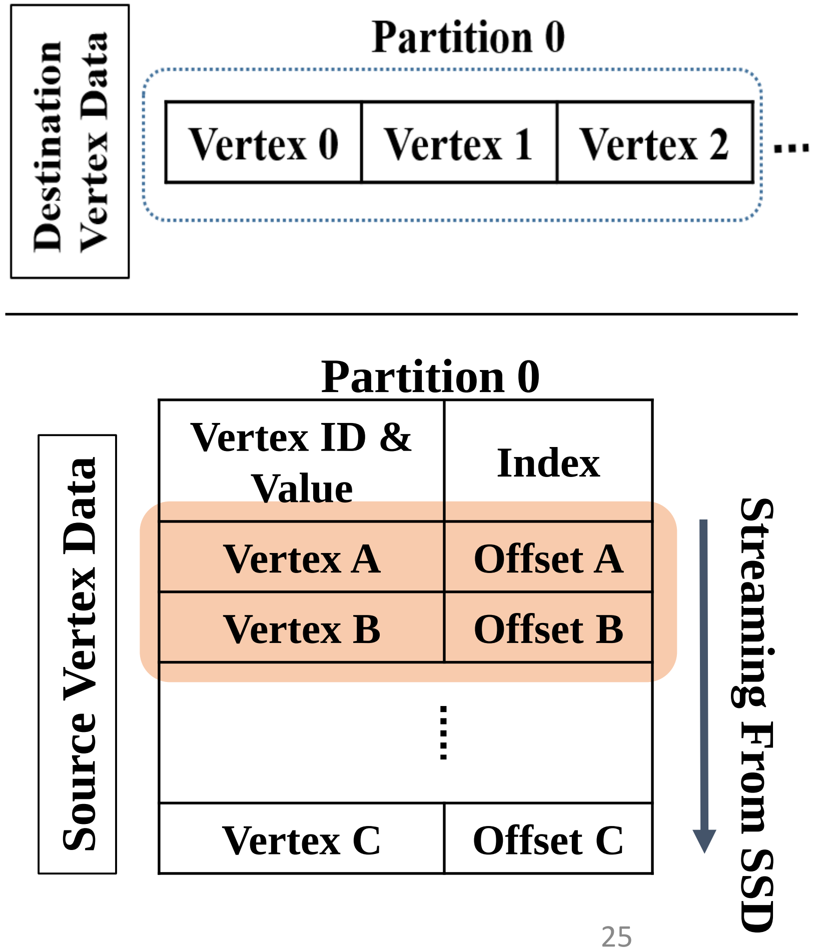




Execution Flow

In each iteration:


Memory
Vertex Data
Memory
Update Destination Vertices
Destination Vertex for a partition
A Chunk of Source Vertex (32MB)
Reading Neighboring Information
SSD

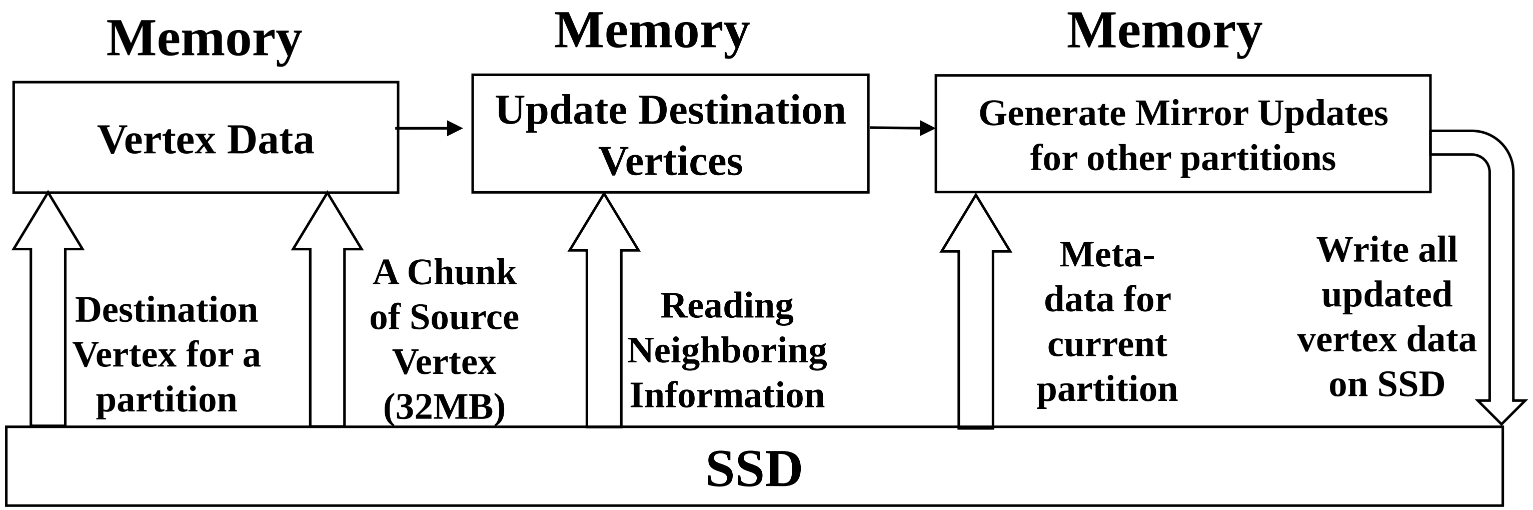


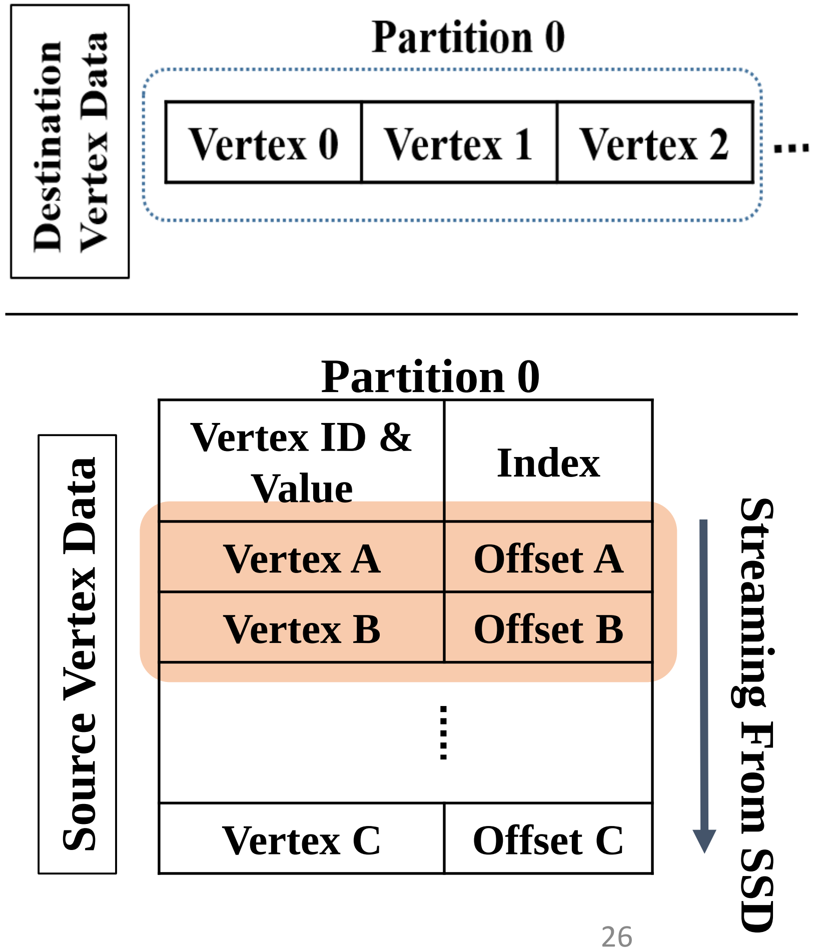
Vertex 0	Vertex 1	Vertex 2	...

Vertex ID & Value	Index
Vertex A	Offset A
Vertex B	Offset B
⋮	
Vertex C	Offset C

Execution Flow

In each iteration:


Memory
Vertex Data
Memory
Update Destination Vertices
Memory
Generate Mirror Updates for other partitions
Destination Vertex for a partition
A Chunk of Source Vertex (32MB)
Reading Neighboring Information
Meta-data for current partition
Write all updated vertex data on SSD
SSD

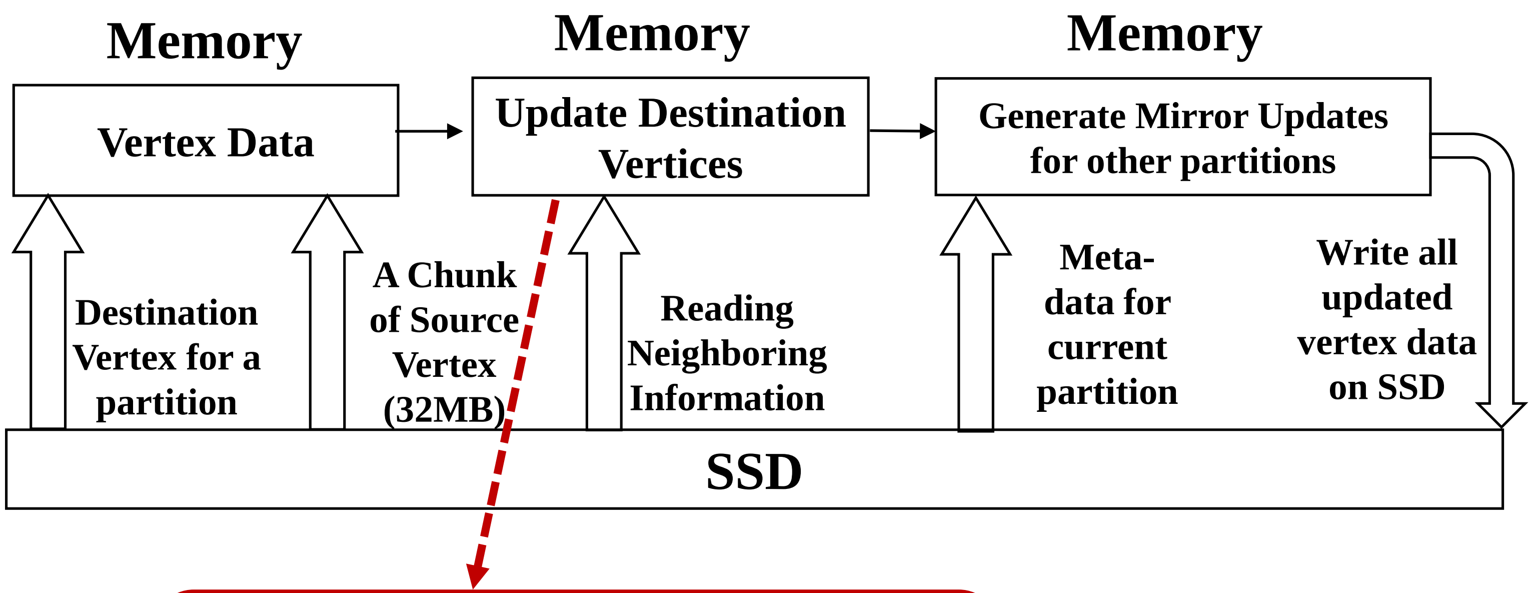


Vertex 0	Vertex 1	Vertex 2

Vertex ID & Value	Index
Vertex A	Offset A
Vertex B	Offset B
⋮	
Vertex C	Offset C

Execution Flow

In each iteration:



Memory

Vertex Data

Destination Vertex for a partition

A Chunk of Source Vertex (32MB)

Memory

Update Destination Vertices

Reading Neighboring Information

Memory

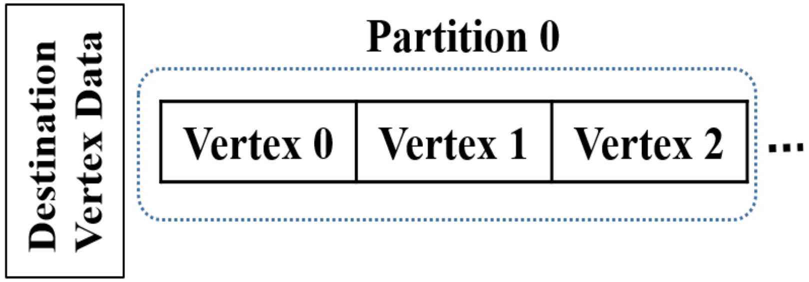
Generate Mirror Updates for other partitions

Meta-data for current partition

Write all updated vertex data on SSD

SSD

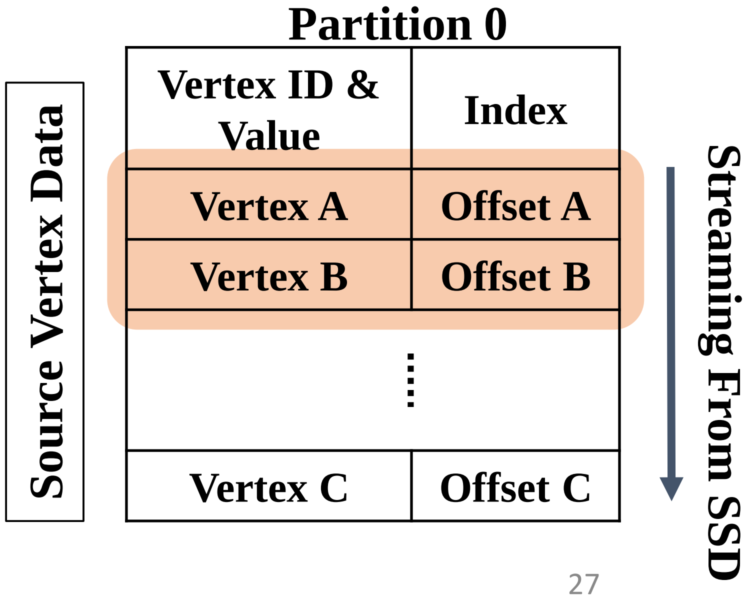
How to Update Vertex List in Main Memory?



Destination Vertex Data

Partition 0

Vertex 0	Vertex 1	Vertex 2	…



Source Vertex Data

Partition 0

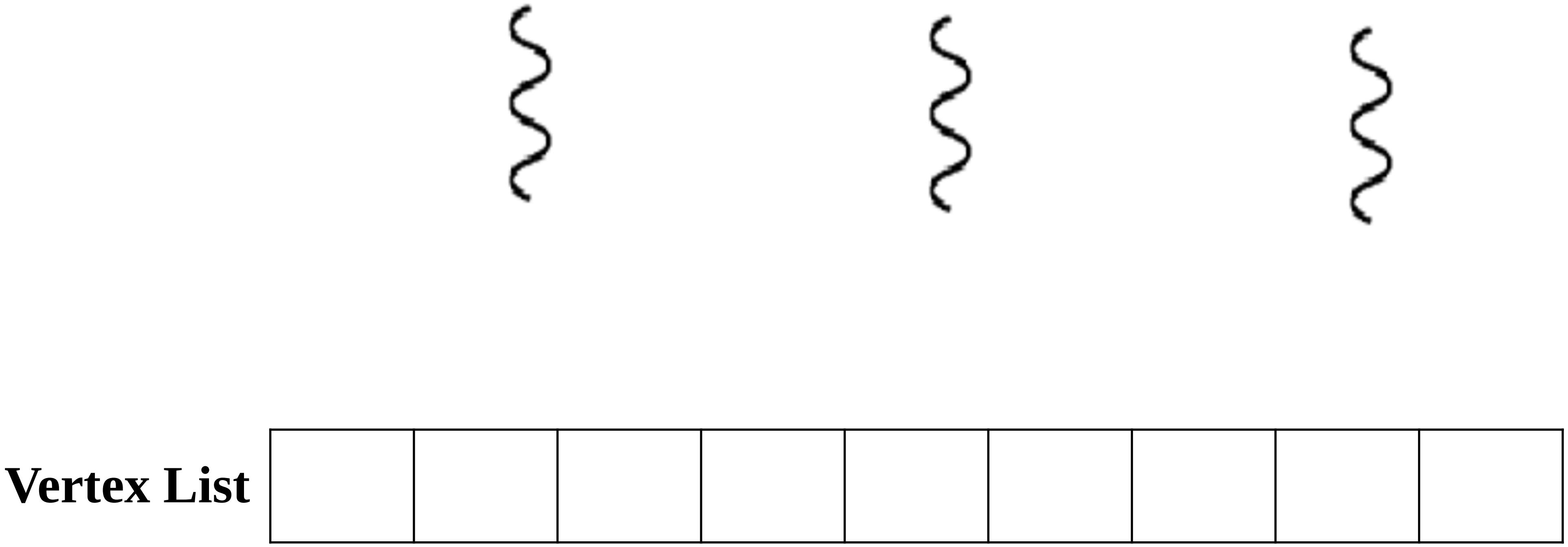
Vertex ID & Value	Index
Vertex A	Offset A
Vertex B	Offset B
⋮	
Vertex C	Offset C

Streaming From SSD

Updating Vertices in Memory

Multiple threads are updating elements of the same vertex list

- High synchronization cost

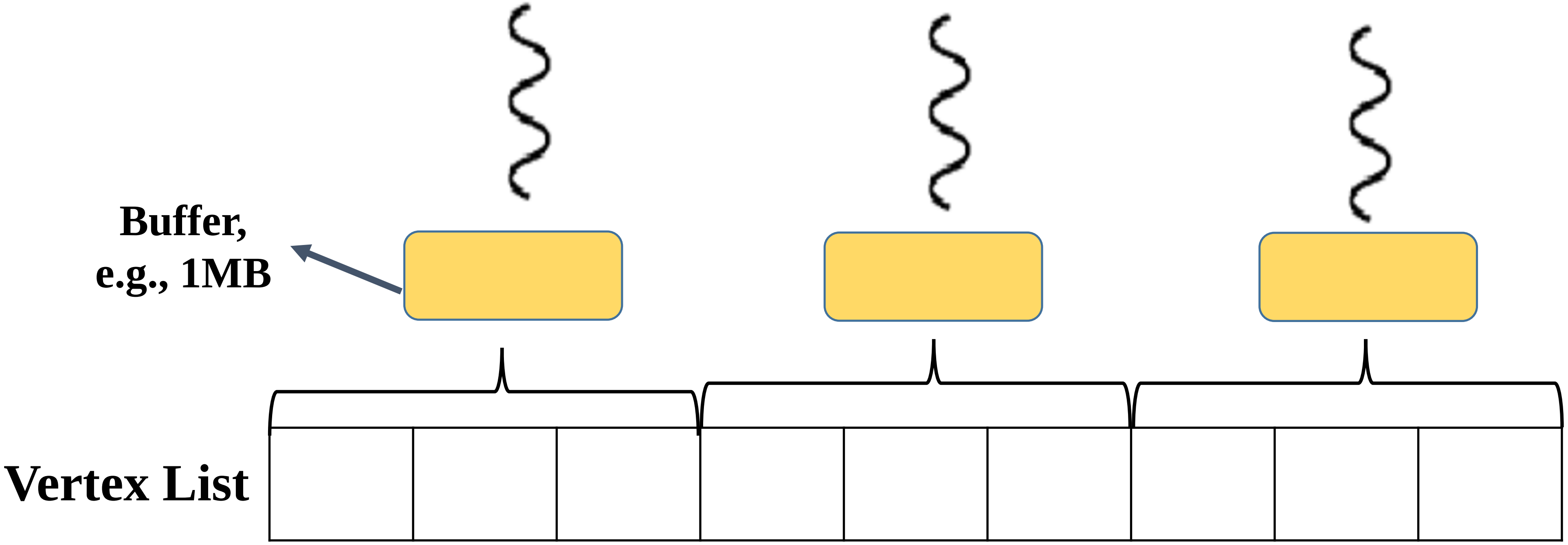


Vertex List

Updating Vertices in Memory

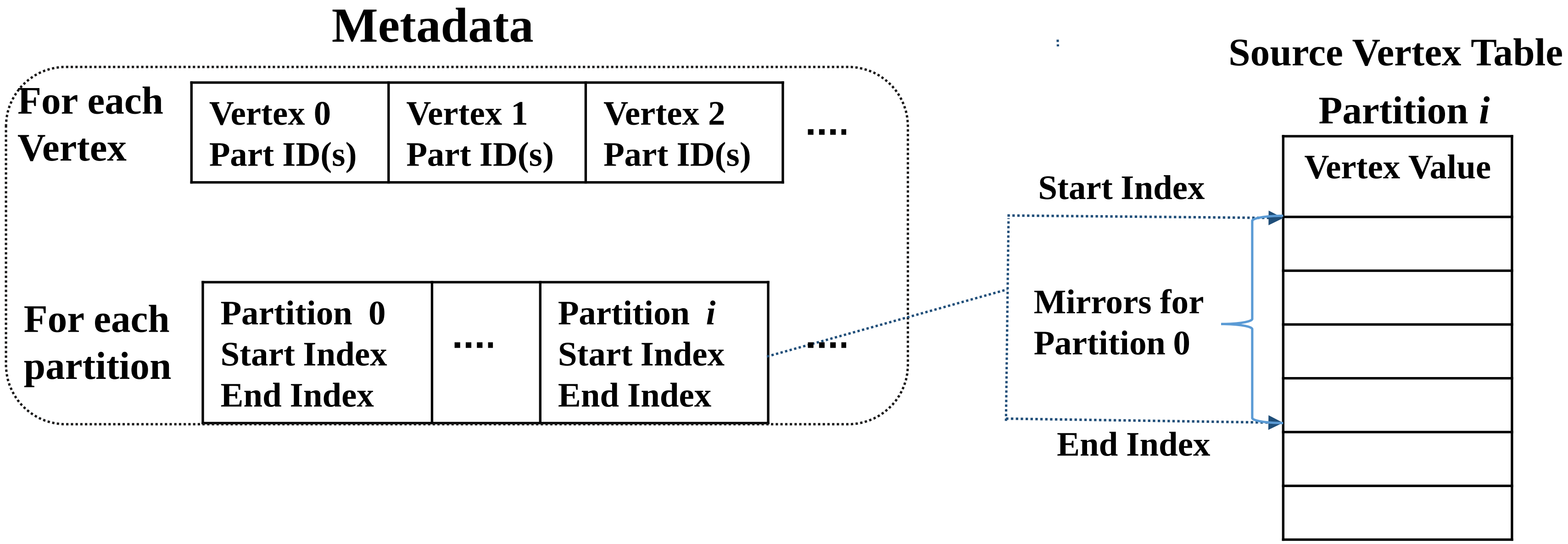
Multiple threads are updating elements of the same vertex list

- High synchronization cost



Updating Vertex Mirrors on Different Partitions

Required Meta-Data for Mirror Updates

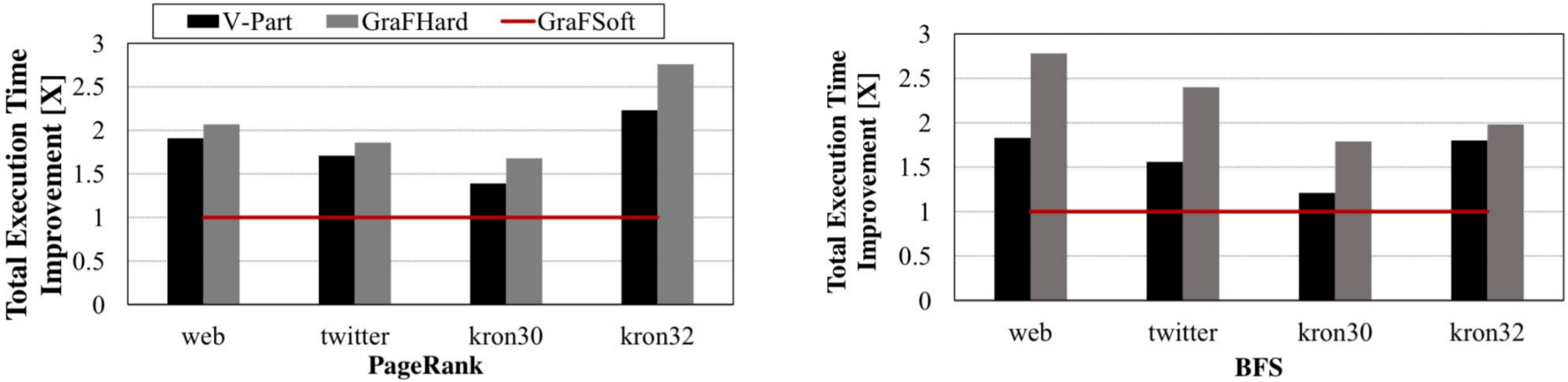


O(|V|) running time for updating mirrors

Experimental Setup

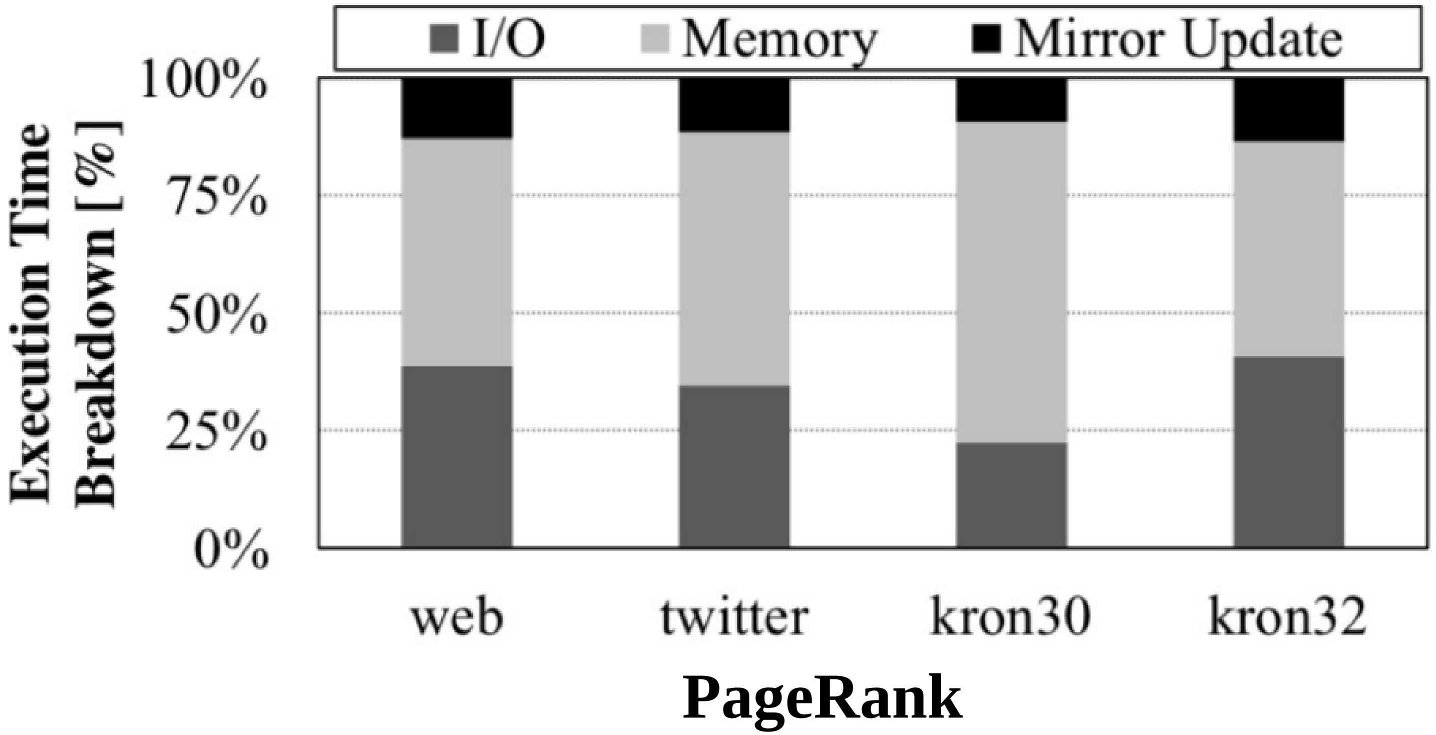
- Processor: Intel Xeon -- 48 Cores
- Memory: DRAM – 256 GB
- SSD: Two Samsung NVMe SSDs
  - 3.2 TB capacity in total, and 6.4 GB/s Sequential Read Speed
- Graph Algorithms:
  - PageRank and Breadth-First-Search (BFS)
- Input Graphs:
  - Web, Twitter, Synthetic (Kron)

Performance Evaluation



- More than 2X Improvement Compared to GrafSoft
- Providing Higher Benefits for larger graphs (Web, Kron32)
- Incurring around 10% space overhead for partitioning

Execution Time Breakdown



- Mirror updates account for 8-12% of execution time
- I/O does not remain the main contributor to the total execution time

Concluding Remarks

- Large-scale graph processing suffers from random updates to vertices
- State-of-the-art provides perfect sequentiality by sorting all updates
    - High computation overhead
- A partitioning for vertex data is proposed to eliminate the need for perfect sequentiality
- In Future: Addressing timely evolving graphs
- Thanks to GraFboost authors (Sang-Woo Jun) !

Thanks!