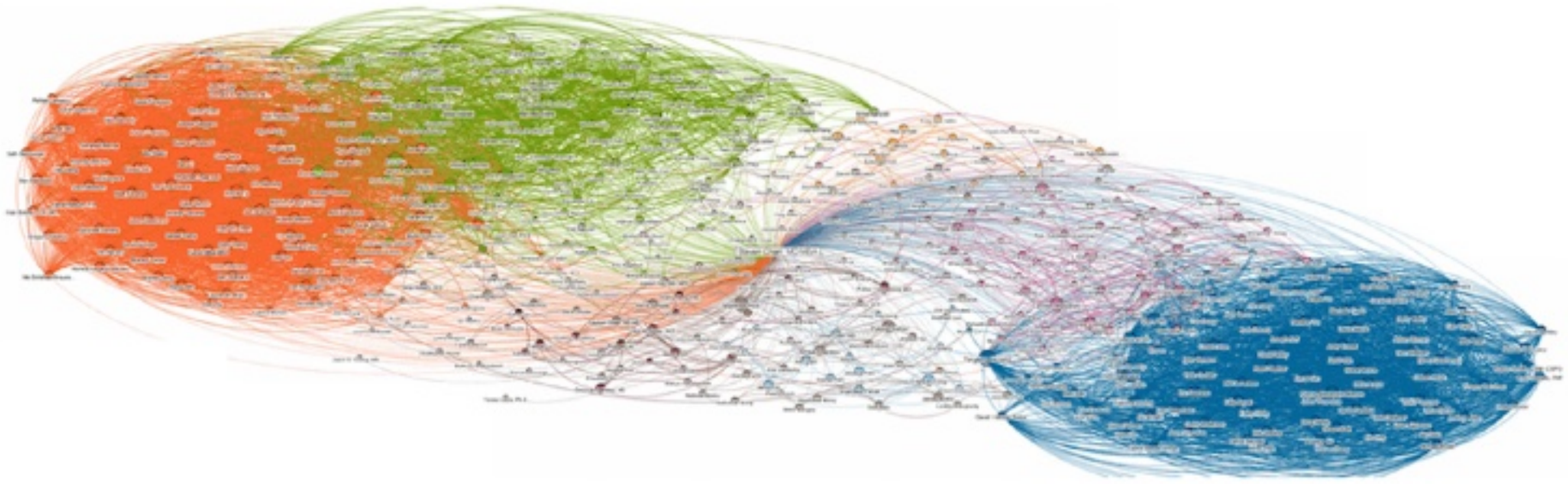
Random Walk on Large-Scale Graphs with Spark

Presenter:
Min Shen (LinkedIn)

Collaborators:
Jeremy Simpson (LinkedIn)
Myunghwan Kim (LinkedIn)
Maneesh Varshney (LinkedIn)

Graph @ LinkedIn

- Connections between 500M LinkedIn members form a large-scale graph
- Useful information can be extracted from this graph
- Raises the requirement for scalable distributed graph algorithms



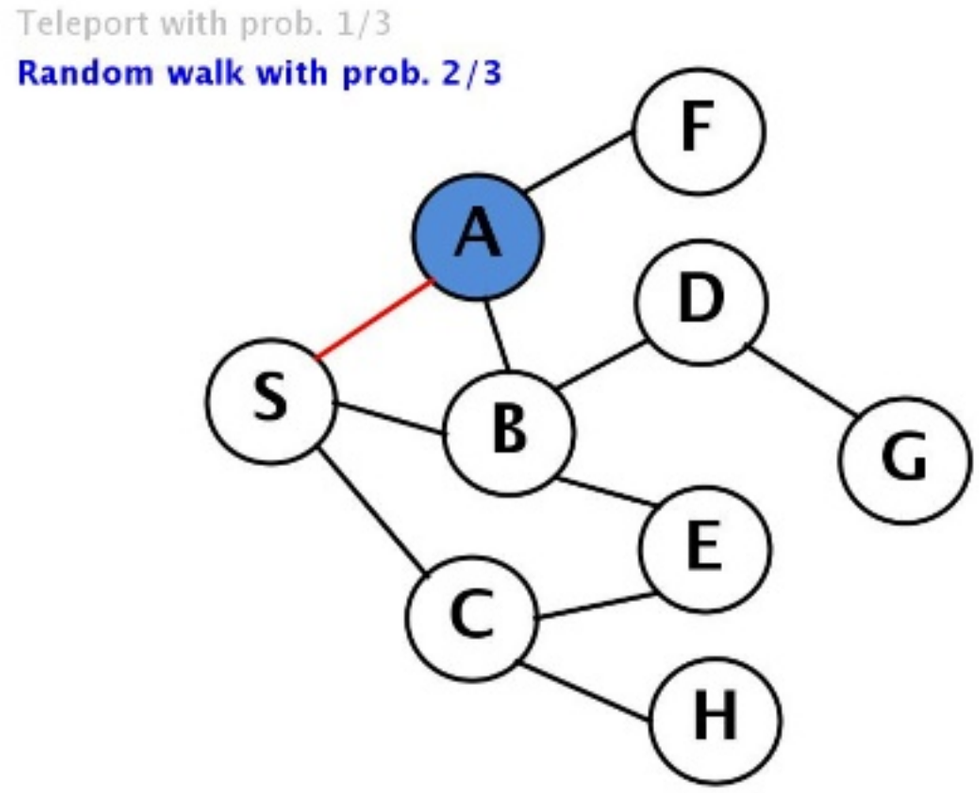
SPARK SUMMIT 2017

Big Graph Challenges

- To make a distributed graph algorithm scale:
  - Minimize data transfer between vertices
  - Reduce cost of transferring data between vertices
- Challenges we face implementing scalable graph algorithms
  - Inefficient to implement on MapReduce
  - Specialized graph engines could miss optimization opportunities for certain algorithms

Random Walk Algorithm

- Random-Walk algorithm is one such example.



Random Walk

- Random walk with personalized restart has applications such as Personalized PageRank (PPR).
- PPR identifies important nodes in the graph from the perspective of individual nodes.
- Well suited for generating personalized recommendations of candidate members for our members to connect to.

Problem Statement

- Input:
  - Graph G = <V, E>
  - N: Number of random walkers starting from each vertex
  - α: Probability to restart/terminate the walk
  - L: Max walk length before the random walker terminates the walk
- Output:
  - Collections of (s, t) pairs: Each represents a sample of the position t of a random walker starting from s.

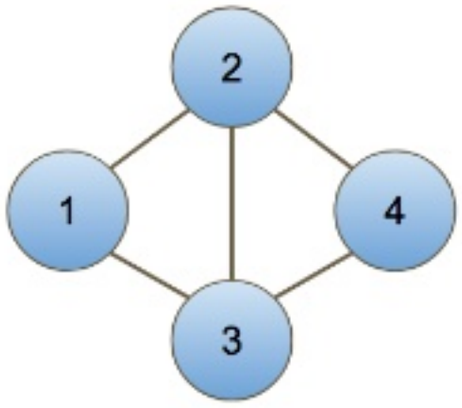
Algorithm Overview

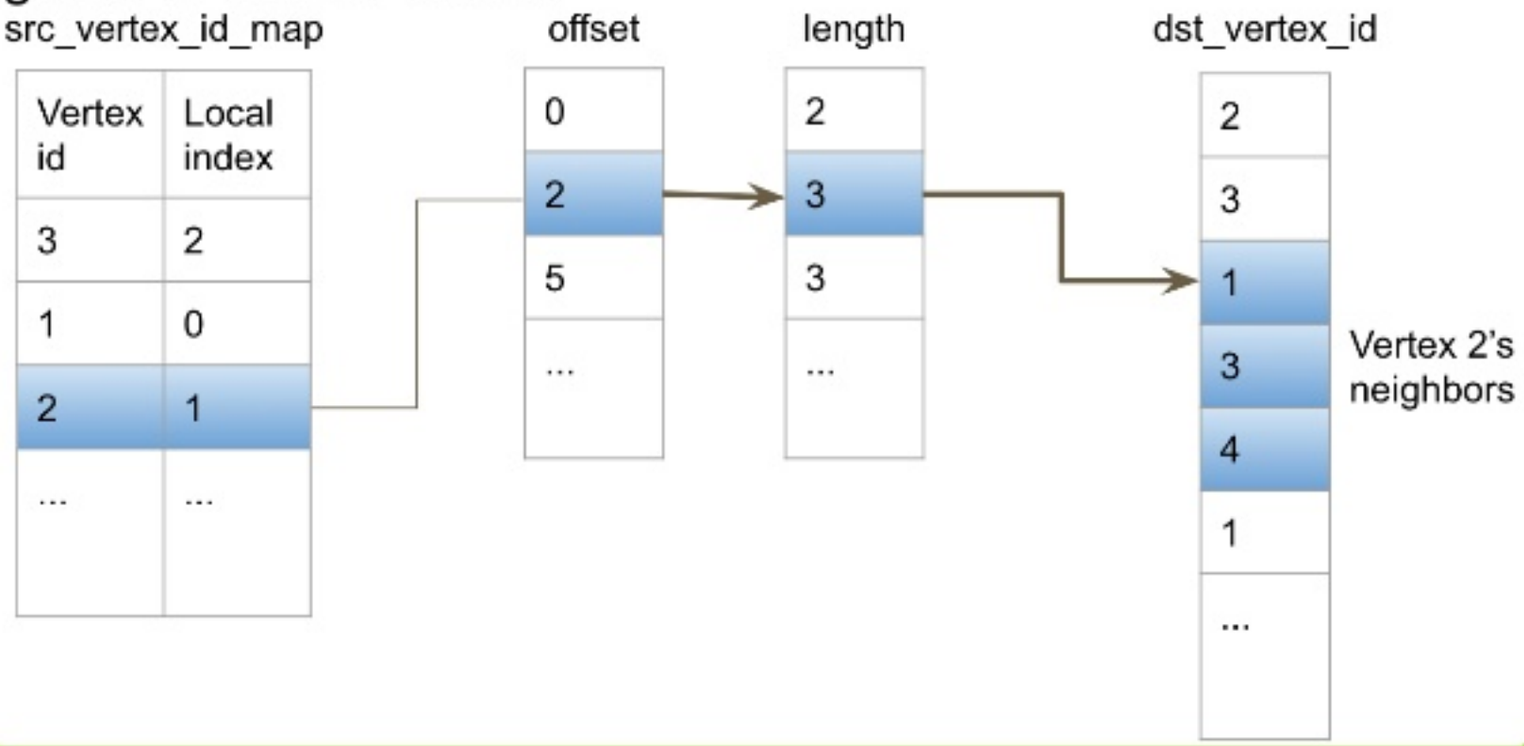
Algorithm pseudo-code on a single processor:

```
random_walk {
  // (s, t): A walker starting vertex s, currently at vertex t
  for each local random walker (s, t) {
    do {
      if (rand(0, 1) < α) {
        break
      } else {
        t = a random neighbor of t
      }
    } while (t is in local graph partition && walker hasn't reached max walk length)
    if (walker (s, t) not terminated) {
      send (s, t) to vertex t
    } else {
      emit (s, t)
    }
  }
}
```

Algorithm Design – Graph Partition

- Graph represented as partitioned adjacency list.
- Each graph partition stored in memory-efficient format supporting:
  - Query if a vertex is inside a partition
  - Fetch all the neighbor of a local vertex





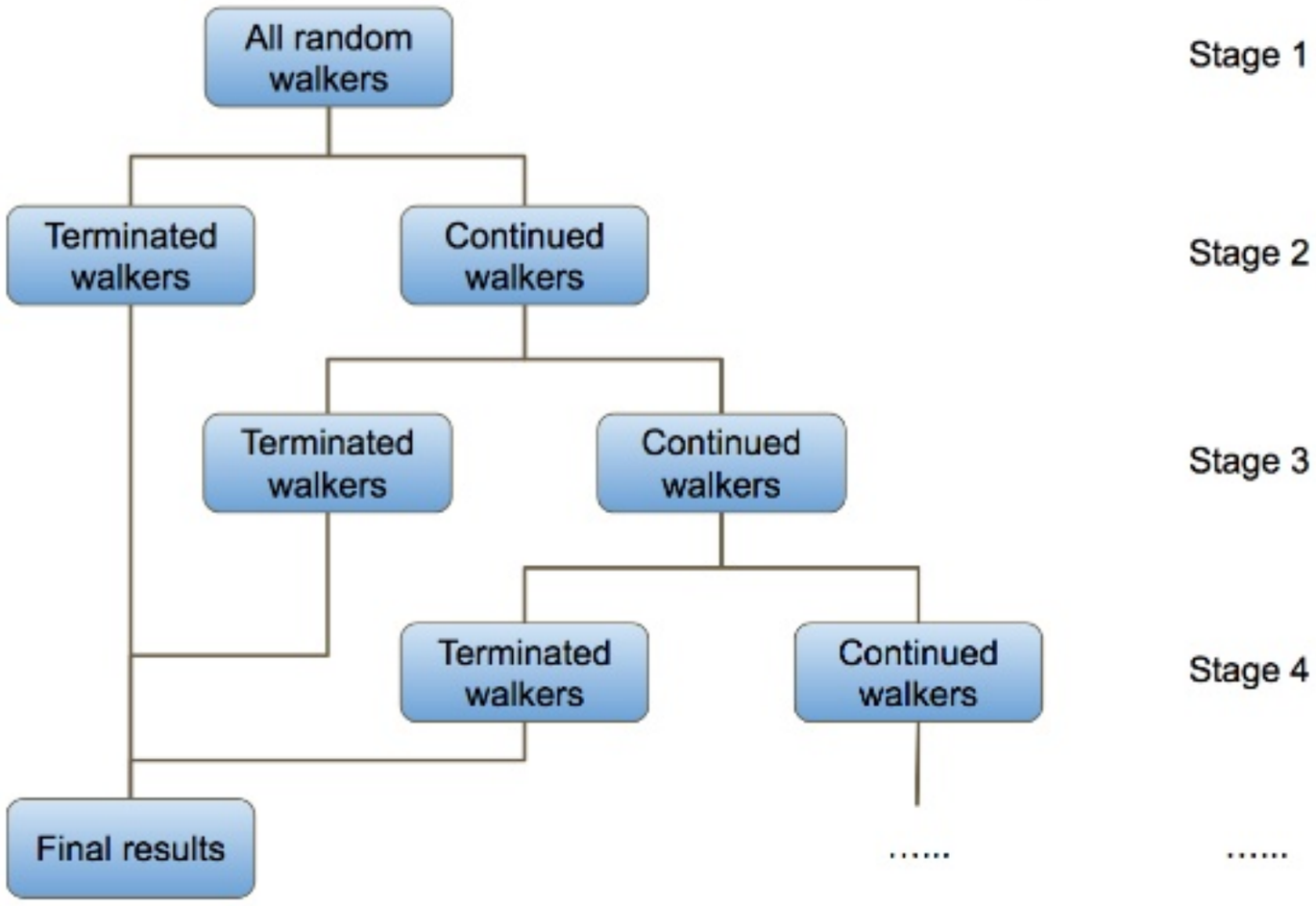
SPARK SUMMIT 2017

How Spark makes it better

- Spark's multi-core execution model
  - By making the graph partition data structures JVM-static and thread-safe, they can be shared between tasks running in the same executor.
  - Graph is only physically partitioned across executors.
  - With executors of larger number of vcores, the number of physical graph partitions can be small while the execution parallelism can be high.

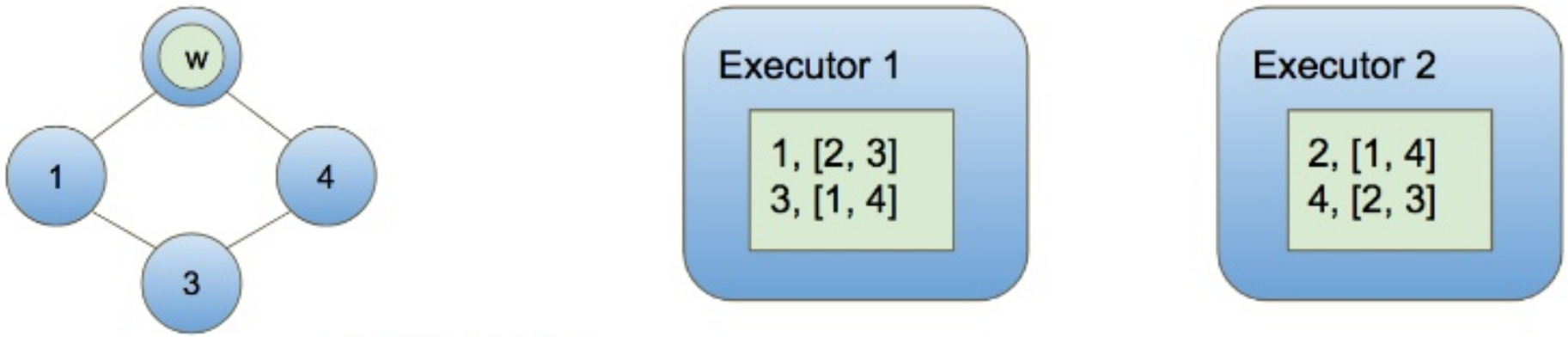
Algorithm Design – Stage execution

- Execution of the algorithm is divided into stages:



Algorithm Design – Stage execution

- Example of a single random walker

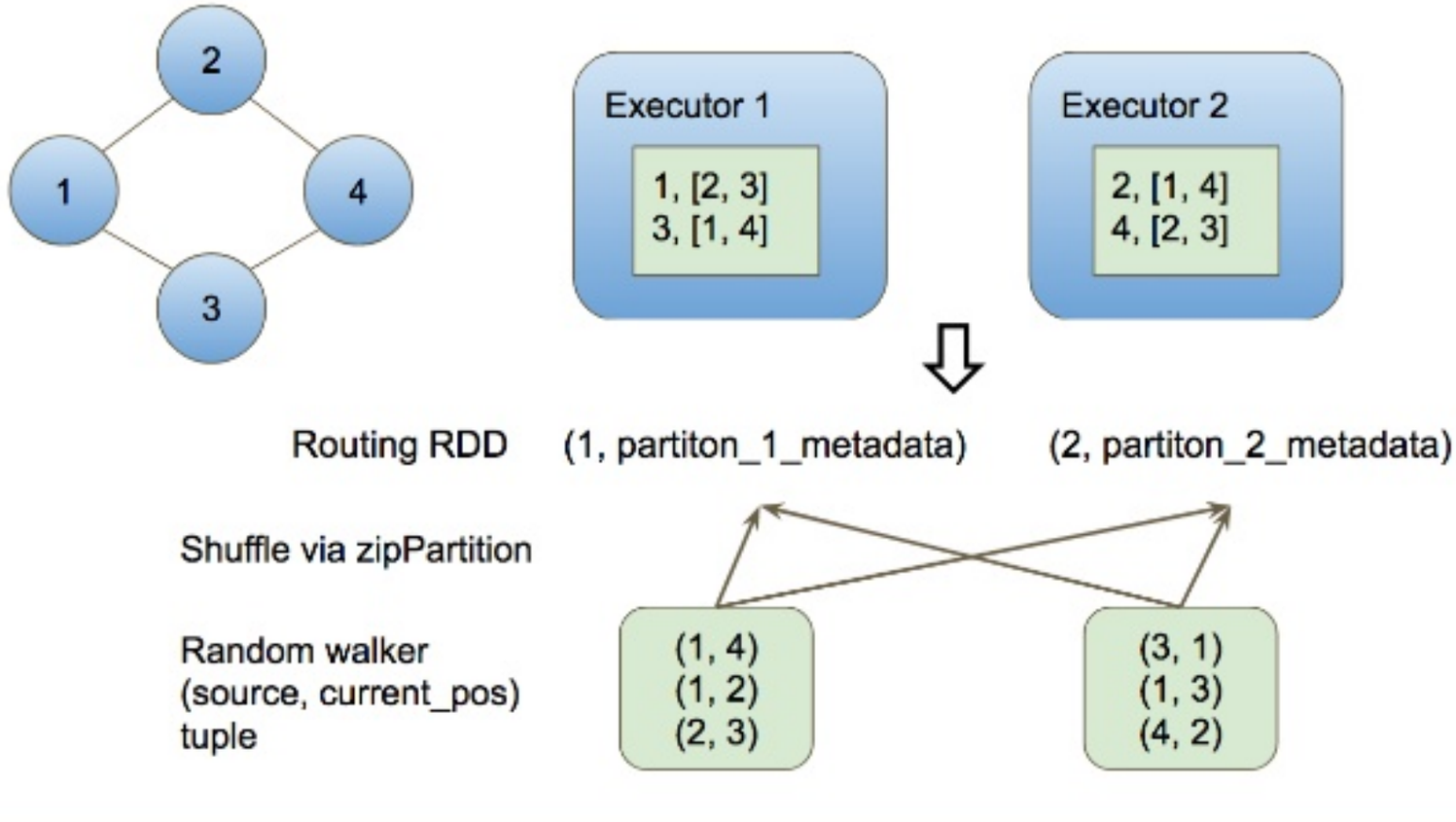


Random walker
(source, current_pos)
tuple

(1, 2)

How Spark makes it better

- Generate routing table from graph partitions as a routing RDD
- Spark's zipPartition operator efficiently leverages the routing table



How Spark makes it better

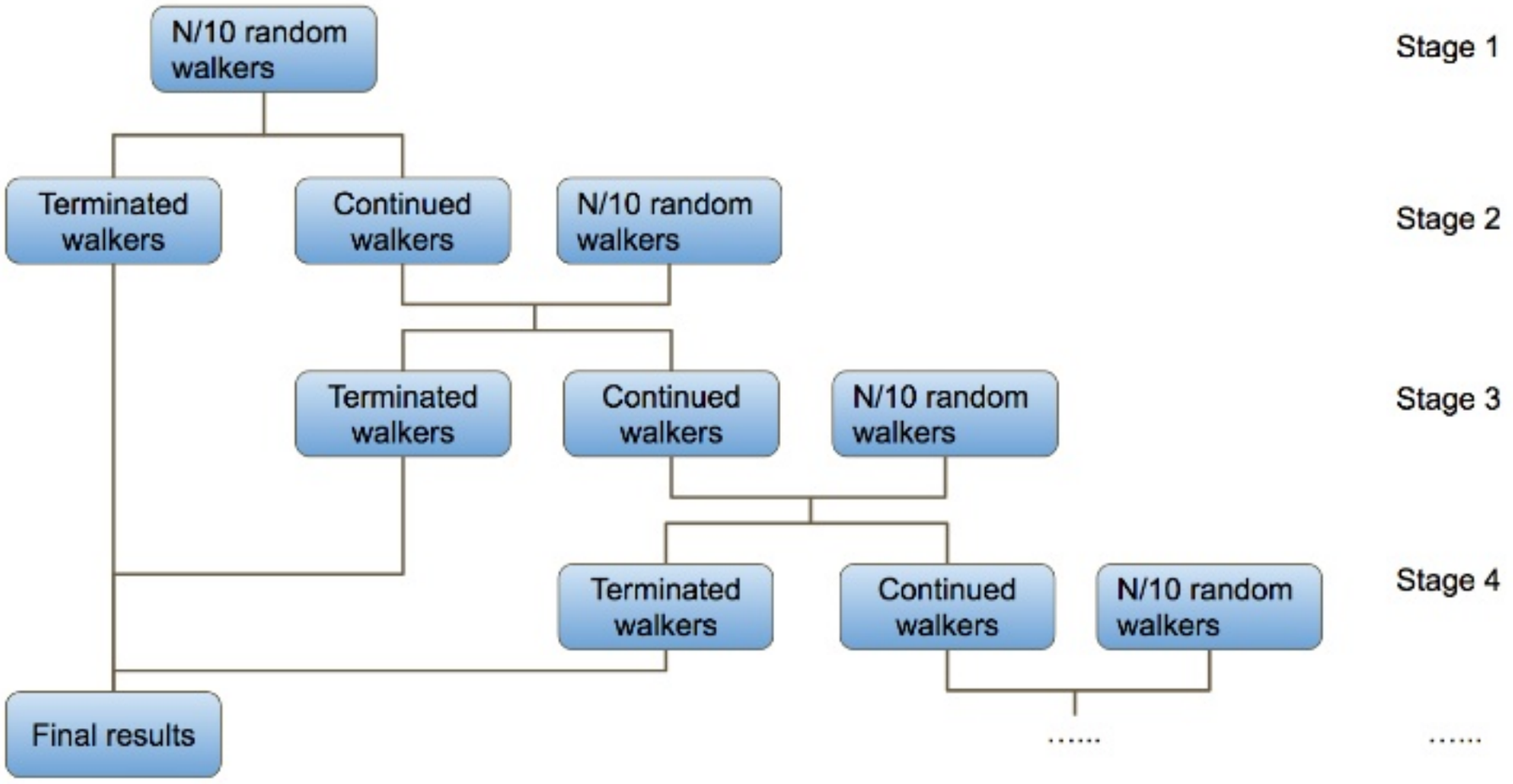
- Advantages brought by the zipPartitions operator
  - Minimize the size of the routing table
  - Graph partition still stored in memory-efficient format
  - Only need to shuffle random walkers once per stage
  - Bring fault-tolerance to the customized graph partition data structure

Algorithm Design – Reduce Shuffle

- Starting all random walkers at once is not scalable
  - Too much data shuffled in the initial stages
- Incremental computation
  - Start *p%* of the random walkers in each stage
  - Some old walkers terminate in each stage
  - Max number of walkers to process in each stage is bounded by N * p% / α
  - We select max number of random walkers to process in each stage vs. total number of stages

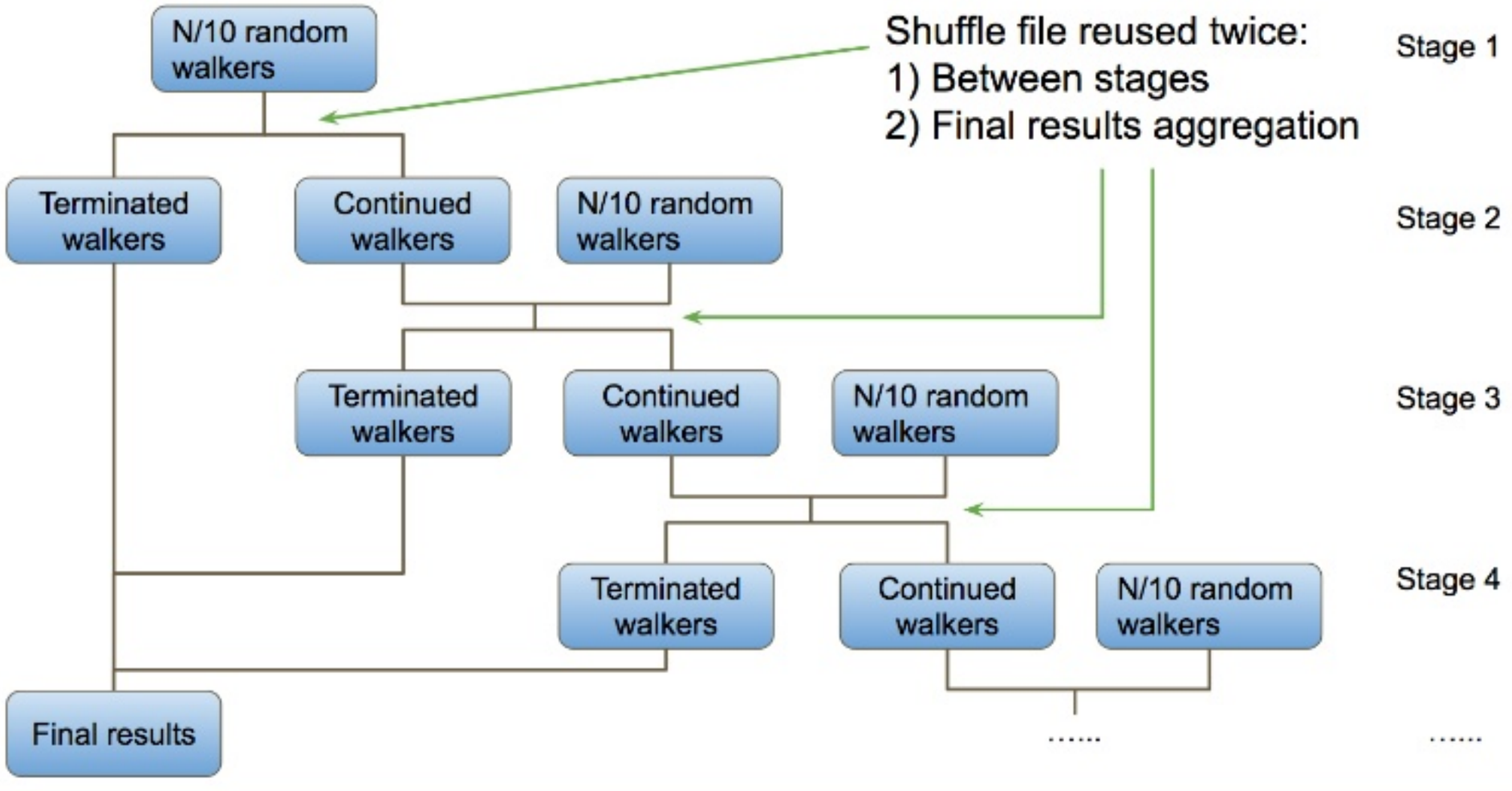
Algorithm Design – Reduce Shuffle

- Illustration of incremental computation



How Spark makes it better

- Illustration of shuffle file reuse



How Spark makes it better

- Spark's shuffle file management
  - Enables us to process random walkers incrementally and effectively gather the final results
  - Each shuffle file is reused twice: between stages and when gathering the final results
  - Leverage external shuffle server to manage the shuffle files so the cost is not incurred inside Spark executors

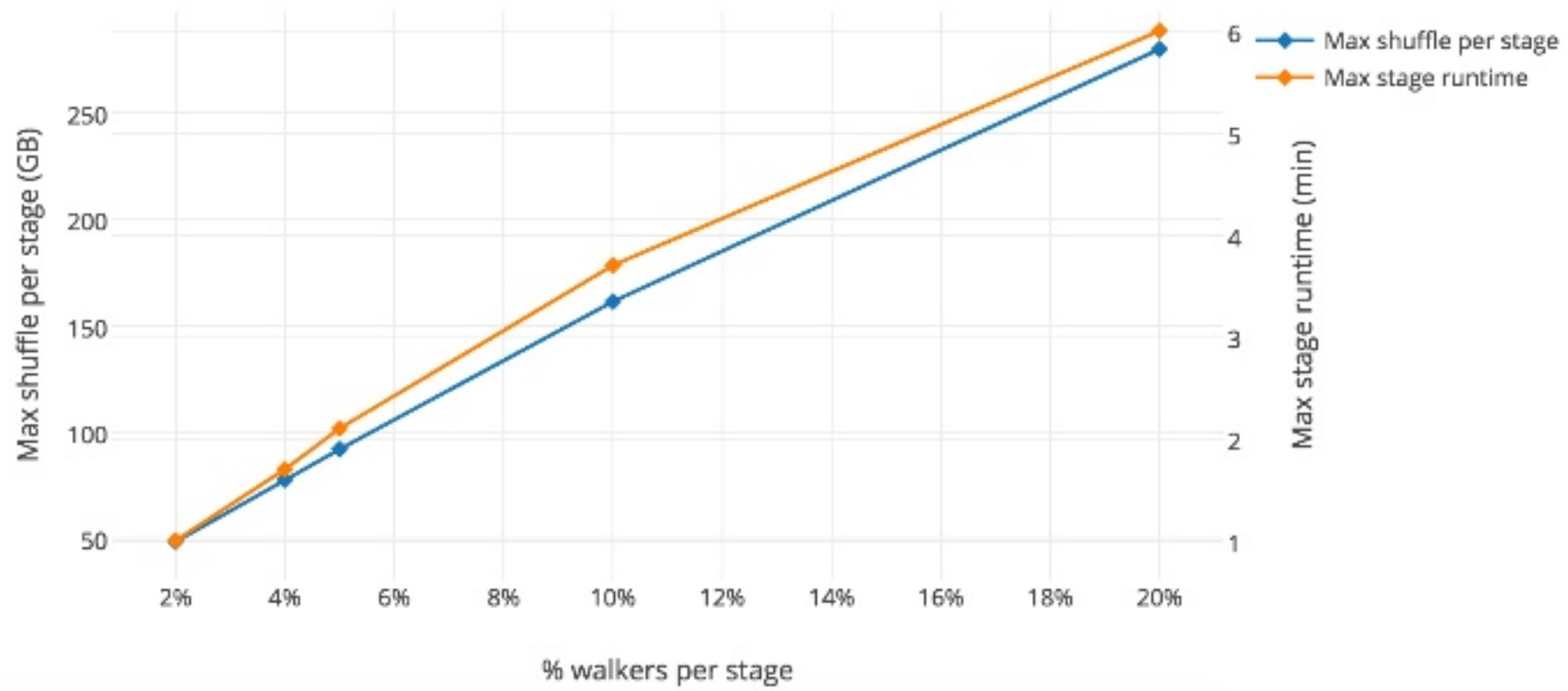
Benchmark

- Setup
  - 200 executors requested from a shared YARN cluster
  - 10G memory and 10 vcores for each executor
  - Take the entire LinkedIn member connection graph as input
  - $N = 500$, $\alpha = 0.4$, $L = 20$
- Memory foot print of the custom graph partition data structure

Store the graph as an RDD of adjacency lists	Store the graph using the custom data structure
300G	260G

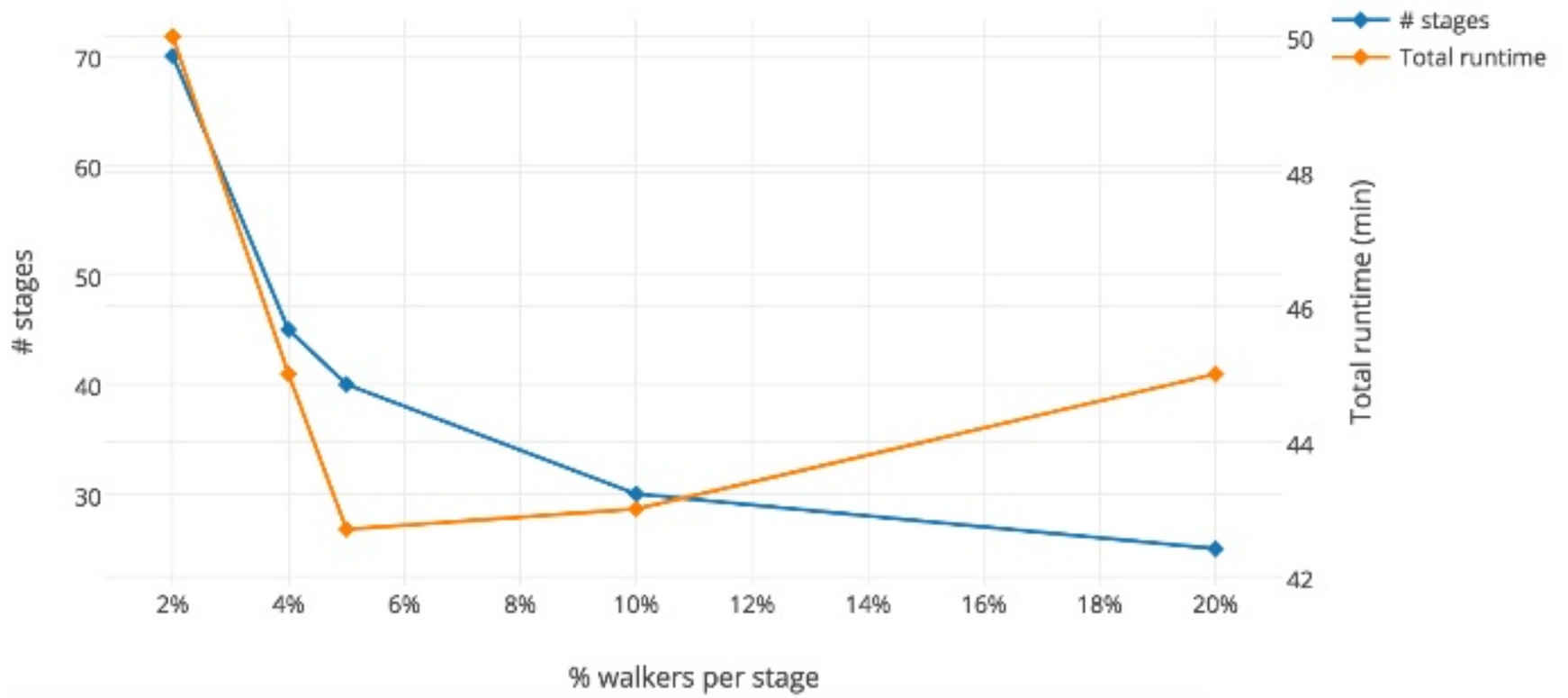
Benchmark

- Trade off between workload per stage vs. total number of stages
  - % walkers per stage vs. max shuffle size and runtime per stage



Benchmark

- Trade off between workload per stage vs. total number of stages
  - % walkers per stage vs. # of stages and total runtime



Conclusion

- Very convenient to implement scalable graph algorithms using Spark's programming interface
- Sophisticated algorithm optimization + Spark's performance = Efficient scalable graph algorithm implementation

Thanks!

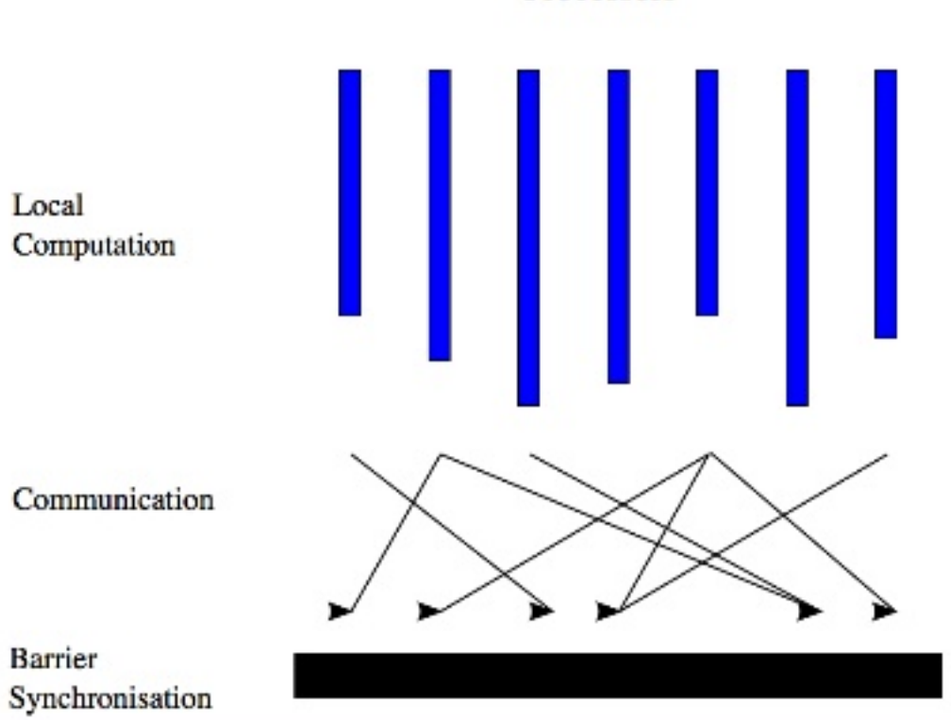
Contact:

Min Shen (mshen@linkedin.com)

Backup slides

What is BSP

- Random walk algorithm can be fit into bulk synchronous parallel (BSP) computation model.
- BSP proceeds in a series of global supersteps.



Why BSP is not a good fit

- However, for personalized random-walk algorithm:
  - Individual walks are independent to each other. There's no need to synchronize between the walks.
  - The random walker only needs local information when determining which node to move to. Information about the state/attribute of the previous nodes in the walk does not need to be piggybacked with the message being sent.
- These observations lead us to designing this algorithm on top of Spark to leverage:
  - Its capability of defining complex computation paradigm beyond simple MapReduce
  - Its flexibility compared with specialized graph processing engines

Algorithm Design – Reduce Shuffle

- Local aggregation of random walkers
  - Random walkers generated in each stage could be aggregated

(src_node_4, target node_5)
(src_node_4, target node_5)
(src_node_4, target node_6)
(src_node_4, target node_6)
(src_node_5, target node_7)
(src_node_5, target node_7)
(src_node_5, target node_7)
...

(src_node_4, target node_5, 2)
(src_node_4, target node_6, 2)
(src_node_5, target node_7, 3)
...