Distributed Graph Snapshot Placement and Query Performance in a Data Center Environment

Alan G. Labouseur
School of Computer Science and Mathematics
Marist College
Poughkeepsie, NY 12601
Alan.Labouseur@Marist.edu

Justin Svegliato
School of Computer Science and Mathematics
Marist College
Poughkeepsie, NY 12601
Justin.Svegliato1@Marist.edu

Jeong-Hyon Hwang
Dept. of Computer Science
University at Albany
State University of New York
Albany, NY 12222
jhh@cs.albany.edu

Daily Deluge of Data

IoT is always **Growing**

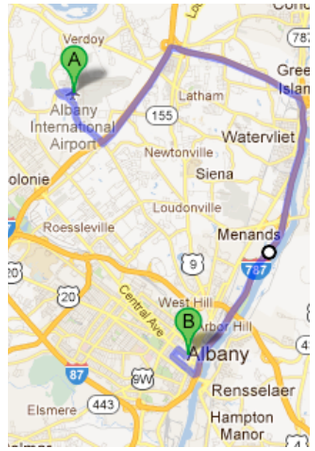
- more sources and sensors
- more products and services
- more messages
- more transactions

IoT is always **Changing**

- adding, removing, and modifying connections

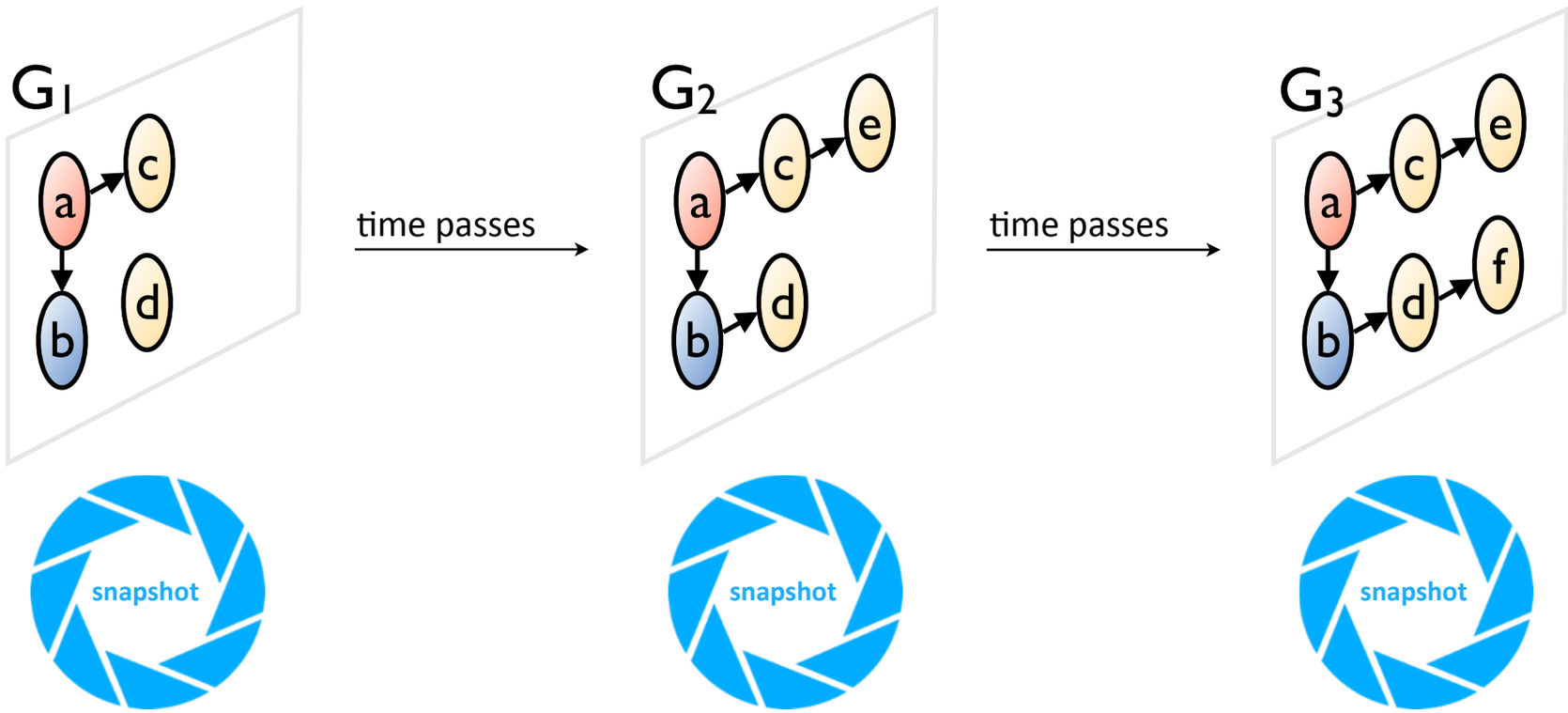
In other words . . .
IoT is always **Evolving**

Q: How do we gain insight from evolving networks?



Dynamic Graphs

A: We gain insight from evolving networks by treating them like Dynamic Graphs where vertices (dots) model entities and edges (lines) model relationships between entities.



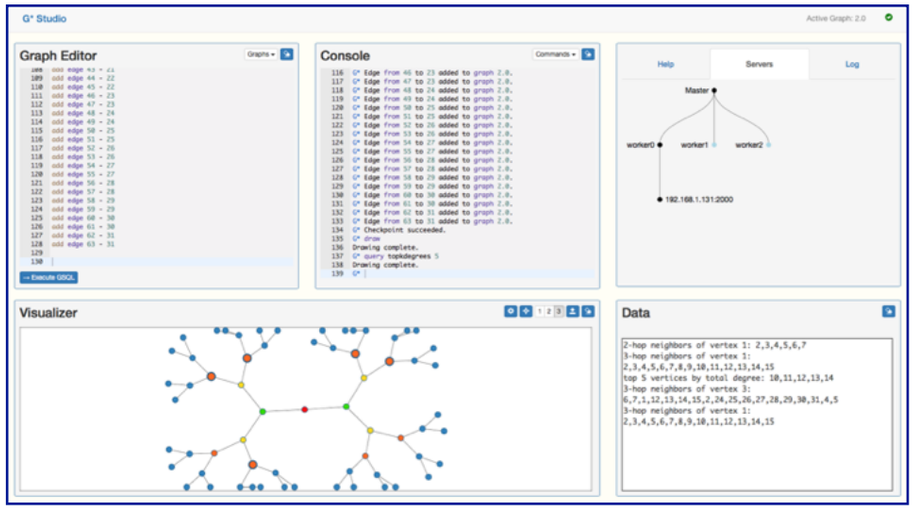
The evolution of a network can be modeled as a **series of graph snapshots** that represent that network at different points in time.

G* Overview

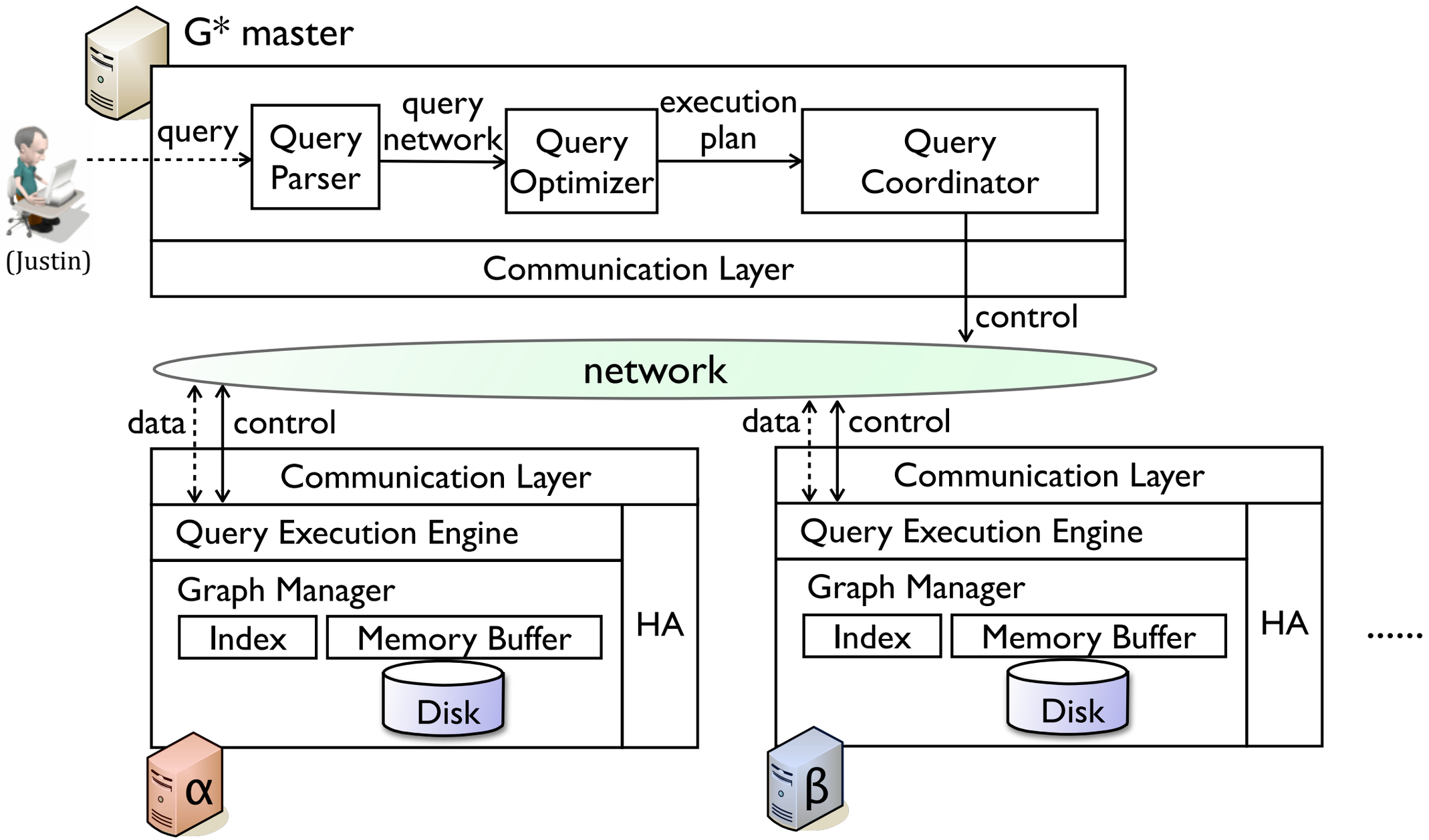
National Science Foundation
WHERE DISCOVERIES BEGIN

G* is our dynamic graph database system.

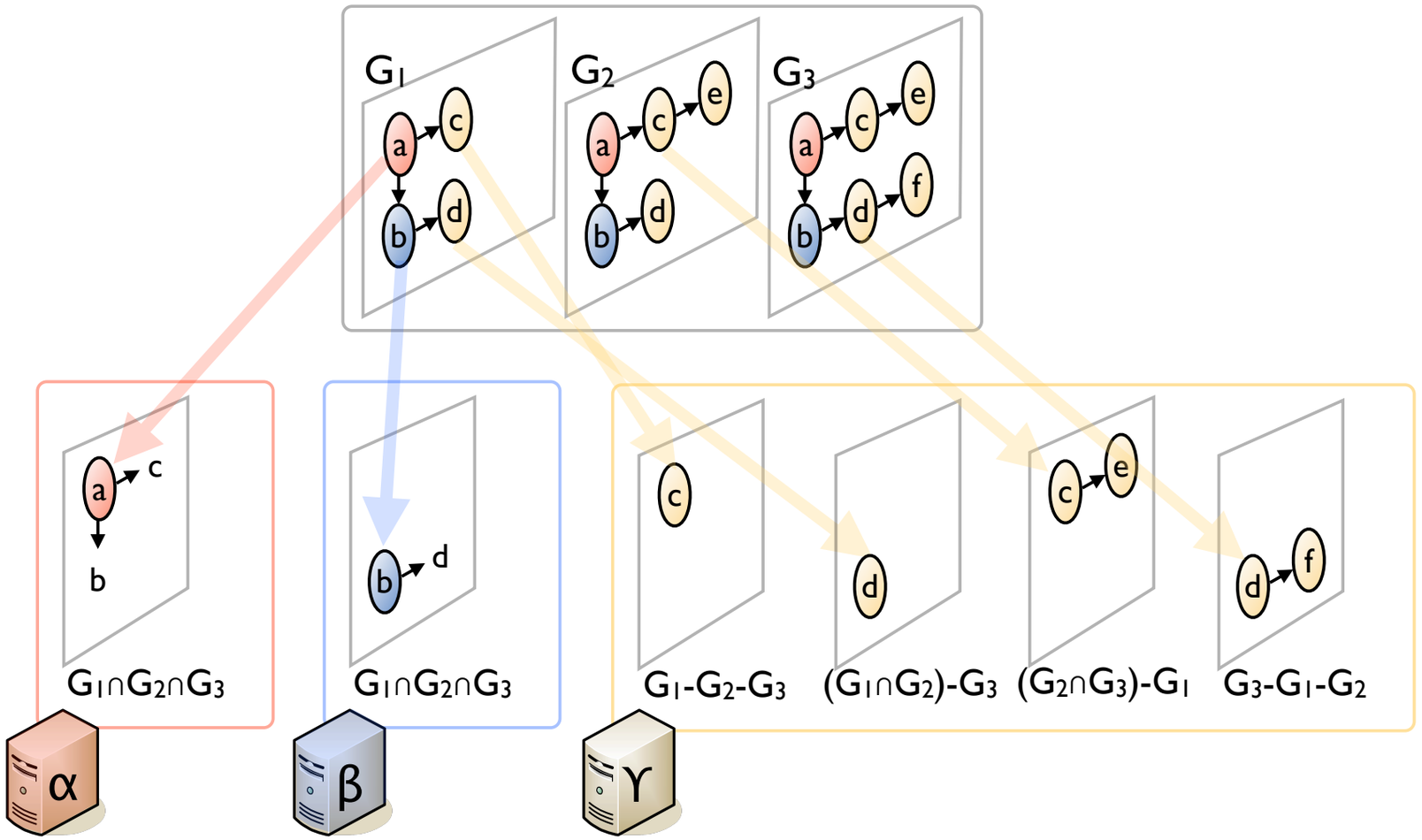
- graph snapshot distribution
- multi-core scale up
- multi-server scale out
- deduplicated storage for large graphs
- in-memory compact indexing
- shared computation
- sophisticated parallel graph queries
- integration with Relational databases and other stores
- a convenient easy-to-use GUI
- a distribution dashboard for analysis



G* Details: Distributed Architecture



G* Details: Deduplicated Snapshot Storage

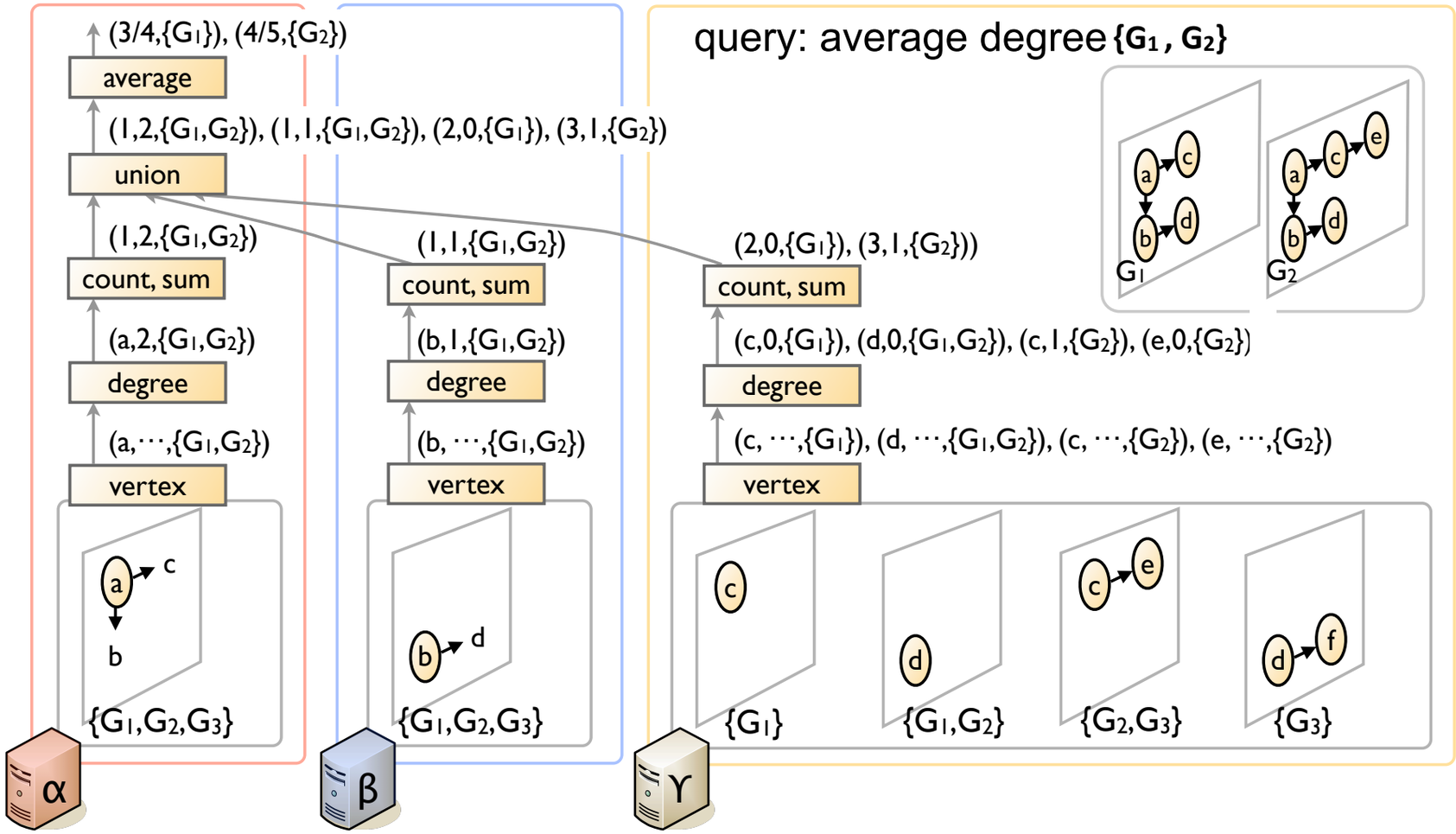


commonality-based deduplication

G* Details: Distributed Query Processing

Average Degree Query (BSP)

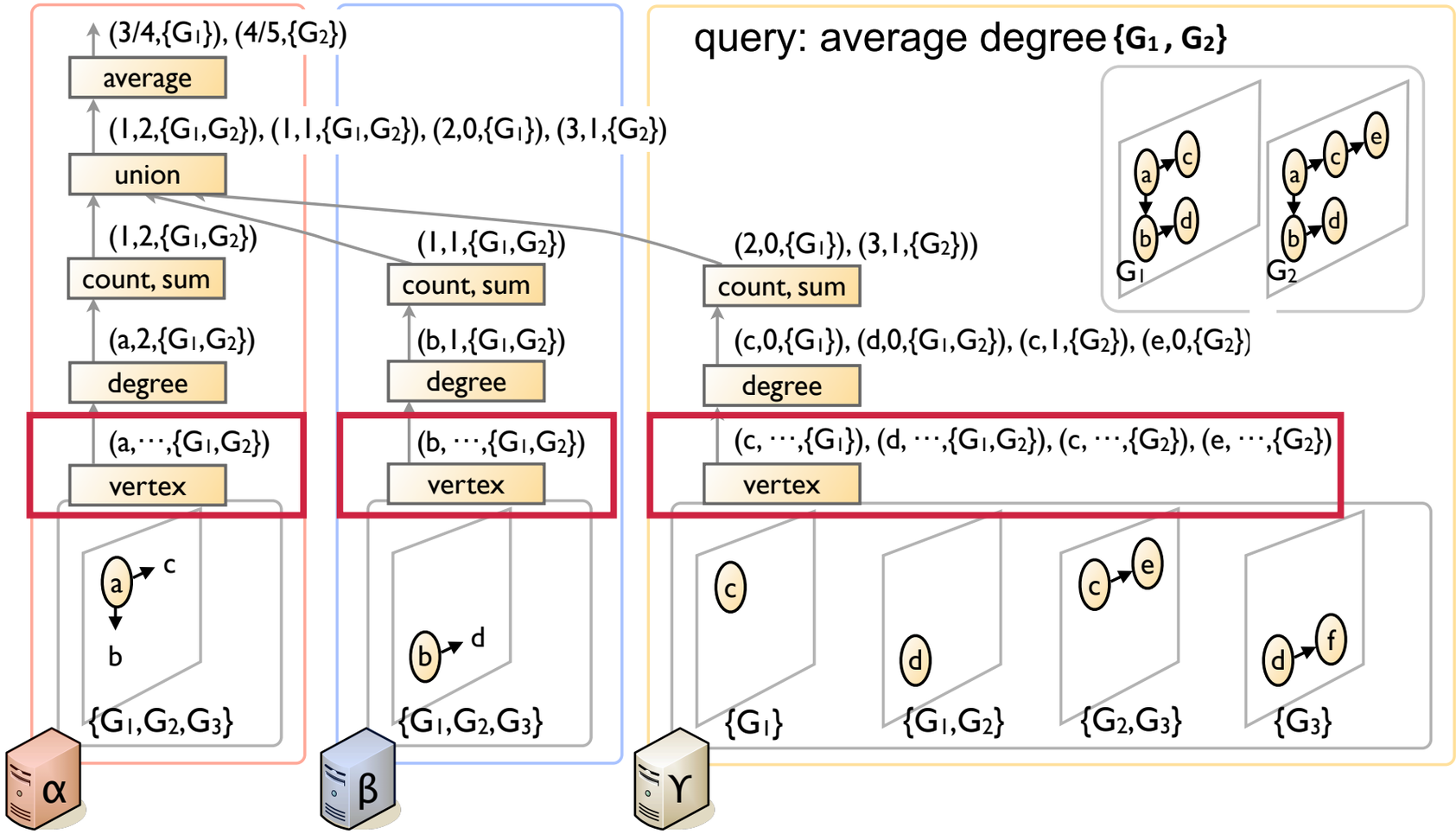
```
operator vertex@*  = VertexOperator([], [α,β,γ]);
operator degree@*  = ProjectionOperator([vertex@local],
                        [id, graph.id, cardinality(outgoing_edges)+cardinality(incoming_edges)],
                        [id, graph.id, degree]);
operator partial@* = AggregateOperator([degree@local], [count, sum], [*, degree], [count, sum], [graph.id]);
operator union@α   = UnionOperator([partial@*]);
operator total@α   = AggregateOperator([union@α], [sum, sum], [count, sum], [count, sum], [graph.id]);
operator avg@α     = ProjectionOperator([total@local], [graph.id, 1.0*sum/count], [graph.id, avg]);
```



G* Details: Distributed Query Processing

Average Degree Query (BSP)

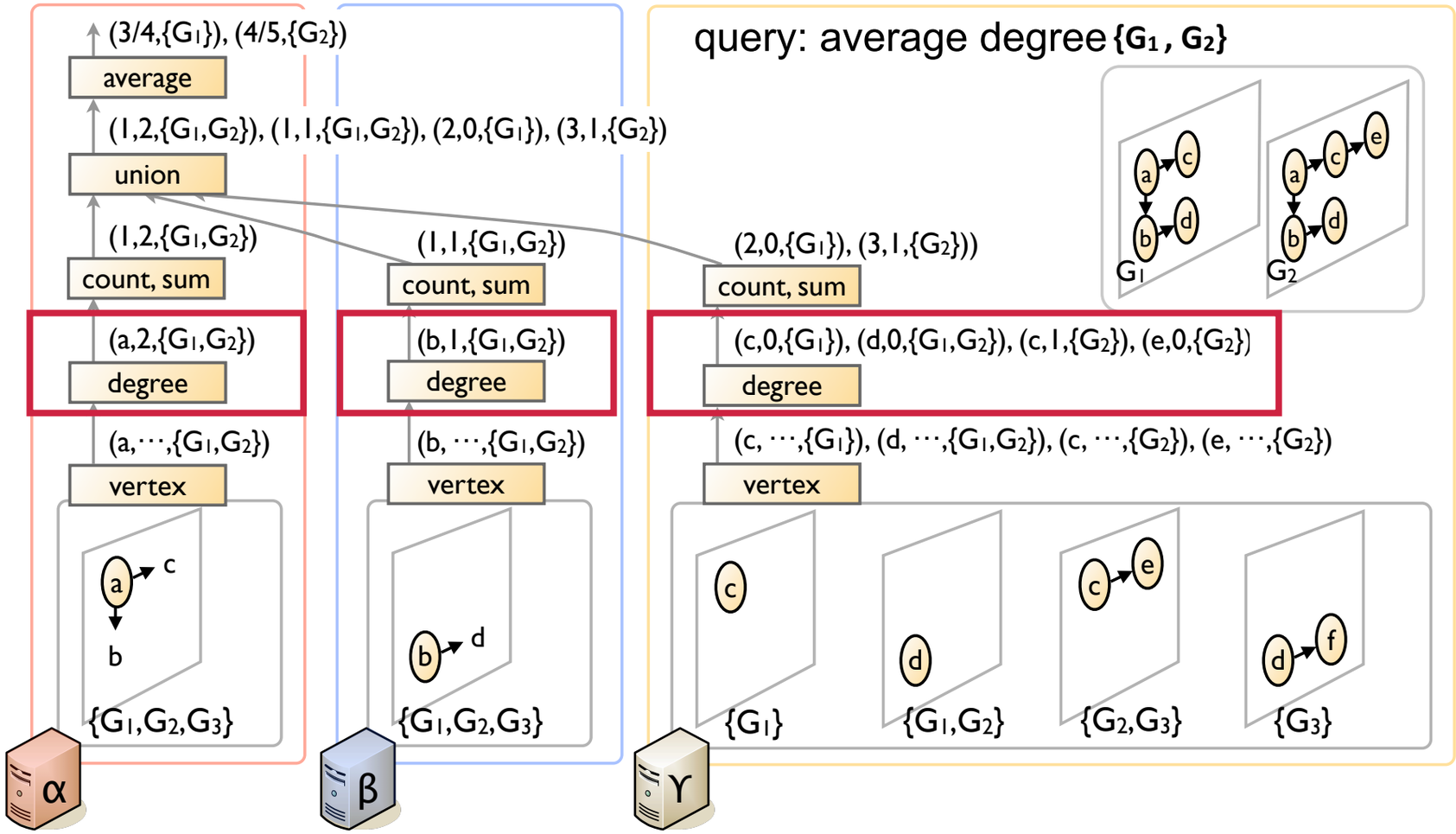
```
operator vertex@*   = VertexOperator([], [α,β,γ]);
operator degree@*   = ProjectionOperator([vertex@local],
                         [id, graph.id, cardinality(outgoing_edges)+cardinality(incoming_edges)],
                         [id, graph.id, degree]);
operator partial@* = AggregateOperator([degree@local], [count, sum], [*, degree], [count, sum], [graph.id]);
operator union@α    = UnionOperator([partial@*]);
operator total@α    = AggregateOperator([union@α], [sum, sum], [count, sum], [count, sum], [graph.id]);
operator avg@α      = ProjectionOperator([total@local], [graph.id, 1.0*sum/count], [graph.id, avg]);
```



G* Details: Distributed Query Processing

Average Degree Query (BSP)

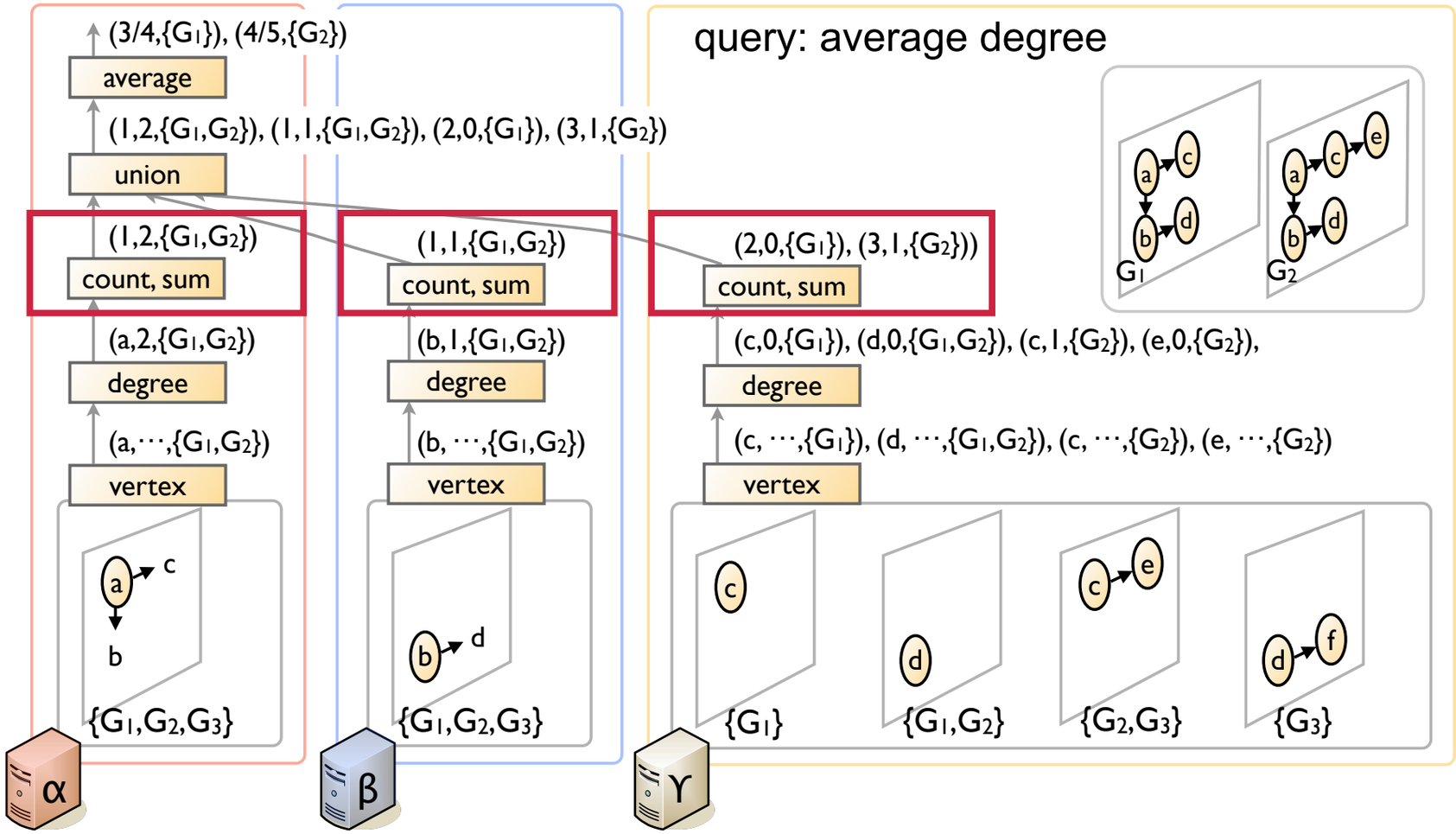
```
operator vertex@*   = VertexOperator([], [α,β,γ]);
operator degree@*   = ProjectionOperator([vertex@local],
                        [id, graph.id, cardinality(outgoing_edges)+cardinality(incoming_edges)],
                        [id, graph.id, degree]);
operator partial@* = AggregateOperator([degree@local], [count, sum], [*, degree], [count, sum], [graph.id]);
operator union@α    = UnionOperator([partial@*]);
operator total@α    = AggregateOperator([union@α], [sum, sum], [count, sum], [count, sum], [graph.id]);
operator avg@α      = ProjectionOperator([total@local], [graph.id, 1.0*sum/count], [graph.id, avg]);
```



G* Details: Distributed Query Processing

Average Degree Query (BSP)

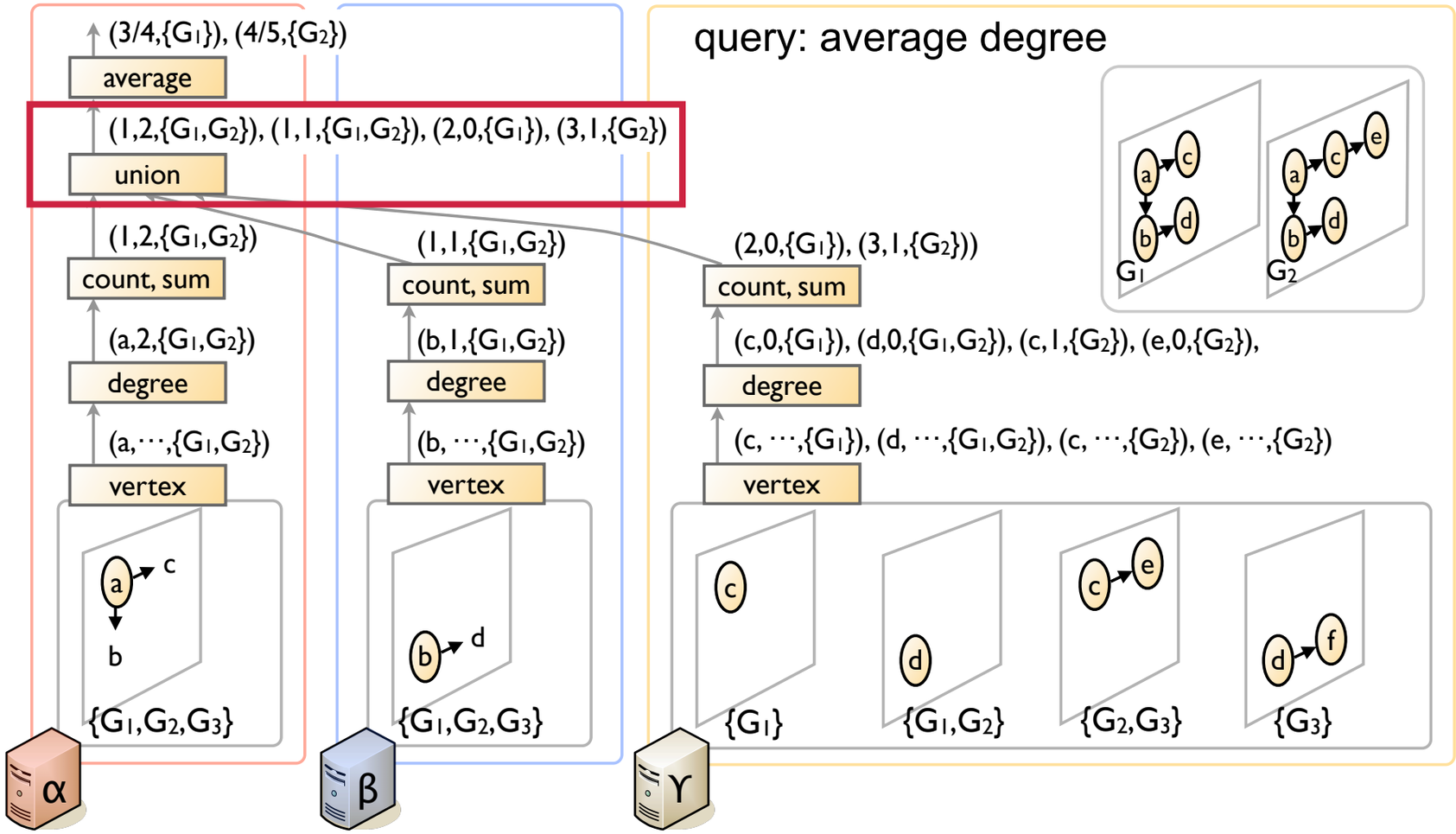
```
operator vertex@*   = VertexOperator([], [α,β,γ]);
operator degree@*   = ProjectionOperator([vertex@local],
                        [id, graph.id, cardinality(outgoing_edges)+cardinality(incoming_edges)],
                        [id, graph.id, degree]);
operator partial@* = AggregateOperator([degree@local], [count, sum], [*, degree], [count, sum], [graph.id]);
operator union@α    = UnionOperator([partial@*]);
operator total@α    = AggregateOperator([union@α], [sum, sum], [count, sum], [count, sum], [graph.id]);
operator avg@α      = ProjectionOperator([total@local], [graph.id, 1.0*sum/count], [graph.id, avg]);
```



G* Details: Distributed Query Processing

Average Degree Query (BSP)

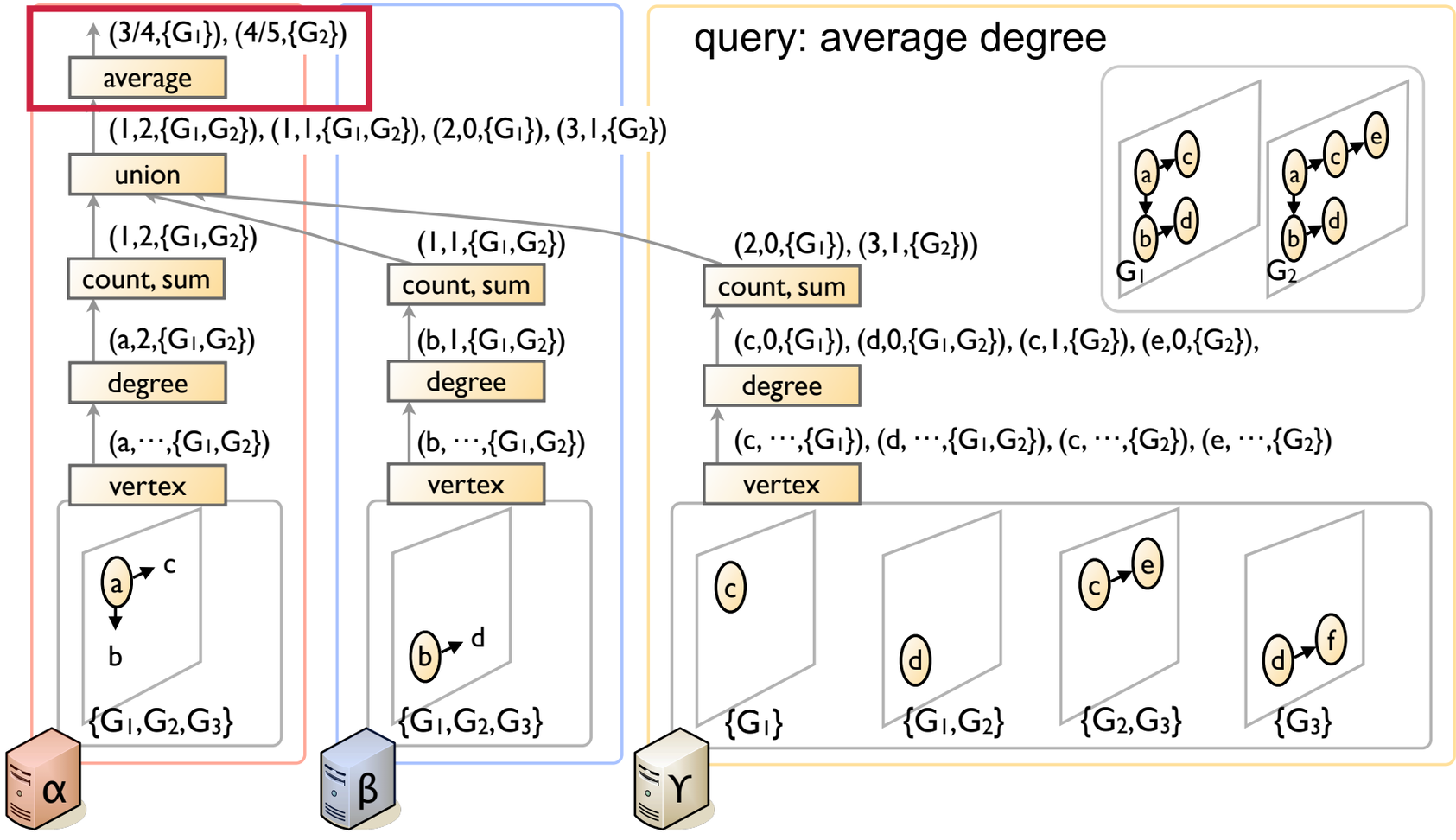
```
operator vertex@*   = VertexOperator([], [α,β,γ]);
operator degree@*   = ProjectionOperator([vertex@local],
                         [id, graph.id, cardinality(outgoing_edges)+cardinality(incoming_edges)],
                         [id, graph.id, degree]);
operator partial@* = AggregateOperator([degree@local], [count, sum], [*, degree], [count, sum], [graph.id]);
operator union@α    = UnionOperator([partial@*]);
operator total@α    = AggregateOperator([union@α], [sum, sum], [count, sum], [count, sum], [graph.id]);
operator avg@α      = ProjectionOperator([total@local], [graph.id, 1.0*sum/count], [graph.id, avg]);
```



G* Details: Distributed Query Processing

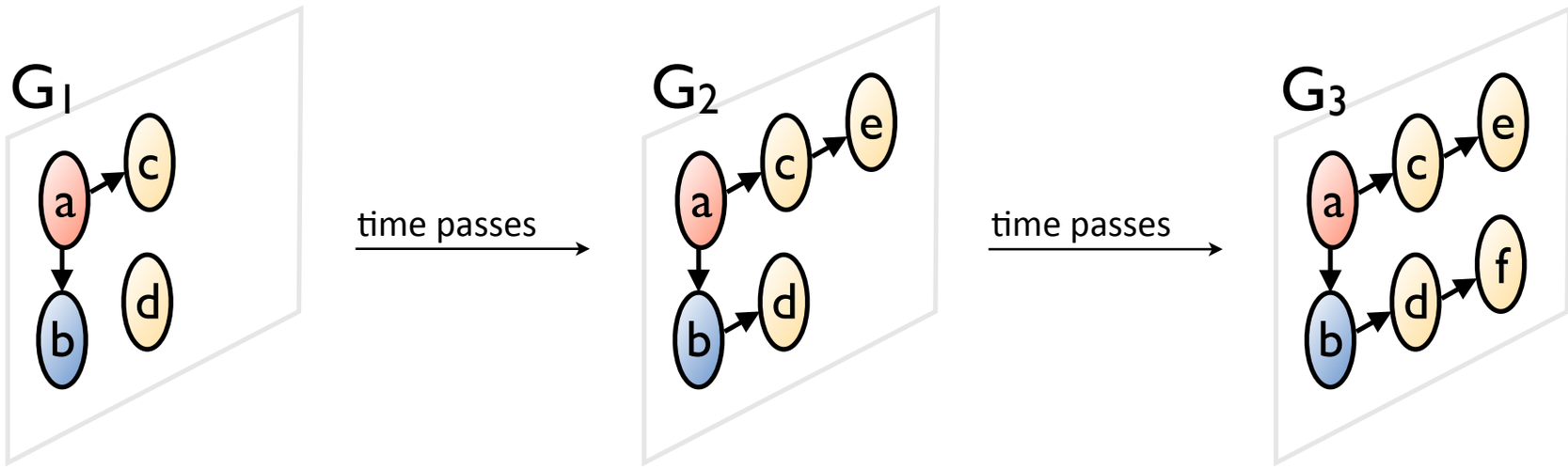
Average Degree Query (BSP)

```
operator vertex@*   = VertexOperator([], [α,β,γ]);
operator degree@*   = ProjectionOperator([vertex@local],
                         [id, graph.id, cardinality(outgoing_edges)+cardinality(incoming_edges)],
                         [id, graph.id, degree]);
operator partial@* = AggregateOperator([degree@local], [count, sum], [*, degree], [count, sum], [graph.id]);
operator union@α    = UnionOperator([partial@*]);
operator total@α    = AggregateOperator([union@α], [sum, sum], [count, sum], [count, sum], [graph.id]);
operator avg@α      = ProjectionOperator([total@local], [graph.id, 1.0*sum/count], [graph.id, avg]);
```



Dynamic Graph Analytics

Using distributed queries, we can analyze network evolution to discover trends and insights crucial to many fields within the Internet of Things:



- networks of electronic health records
- geolocation trackers
- diagnostic data from connected devices (e.g., smart watches)
- and more…

Wait. Isn't this already solved?

So far . . .

- Accelerating queries by distributing **static data** over multiple workers has been well explored.
- Techniques for partitioning **individual graphs** to facilitate parallel computation have been developed.

However . . .

- Accelerating queries by distributing **continuously generated** data presents new challenges.
- Techniques for partitioning **series of graphs** (snapshots) to facilitate parallel computation raise new challenges.
  - cannot consider only one graph at a time
  - cannot distribute every snapshot to all workers

So there's a problem.

The Problem

The snapshot distribution problem:

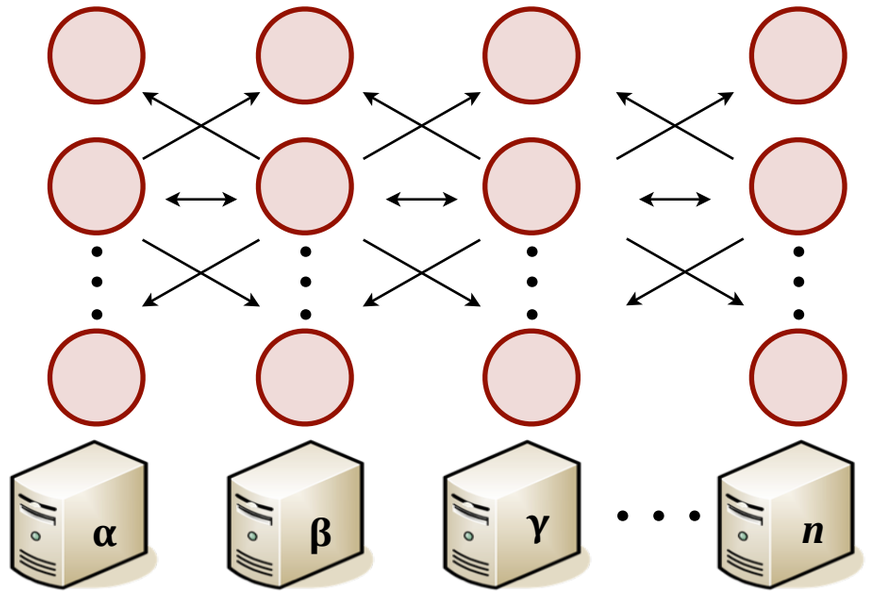
> *How do we distribute snapshots of a continuously evolving network among our available workers in a way that's efficient, scalable, and optimized to process various types of queries*?

Formally:

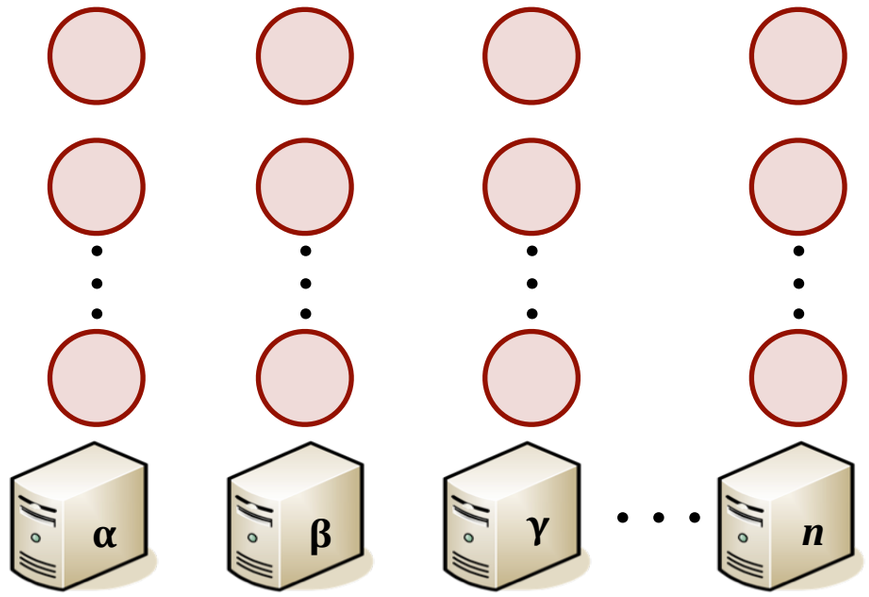
> **Definition**: Given a series of graph snapshots $\{G_i(V_i, E_i) : i = 1, 2, \cdots\}$, n workers, and a set of queries Q on some or all of the snapshots, find a distribution $\{V_{i,w} : i = 1, 2, \cdots \wedge w = 1, 2, \cdots, n\}$ that minimizes $\sum_{q \in Q} time(q, \{V_{i,w}\})$ where $V_{i,w}$ denotes the set of vertices that are from snapshot $G_i(V_i, E_i)$ and that are assigned to worker w, and $time(q, \{V_{i,w}\})$ represents the execution time of query $q \in Q$ on the distributed snapshots $\{V_{i,w}\}$ satisfying (1) $\cup_{w=1}^{n} V_{i,w} = V_i$ (i.e., the parts of a snapshot on all workers cover the original snapshot) and (2) $V_{i,w} \cap V_{i,w'} = \emptyset$ if $w \neq w'$ (i.e., workers are assigned disjoint parts of a snapshot).

Queries

PageRank
requires many vertices to be examined to compute the value for each vertex.

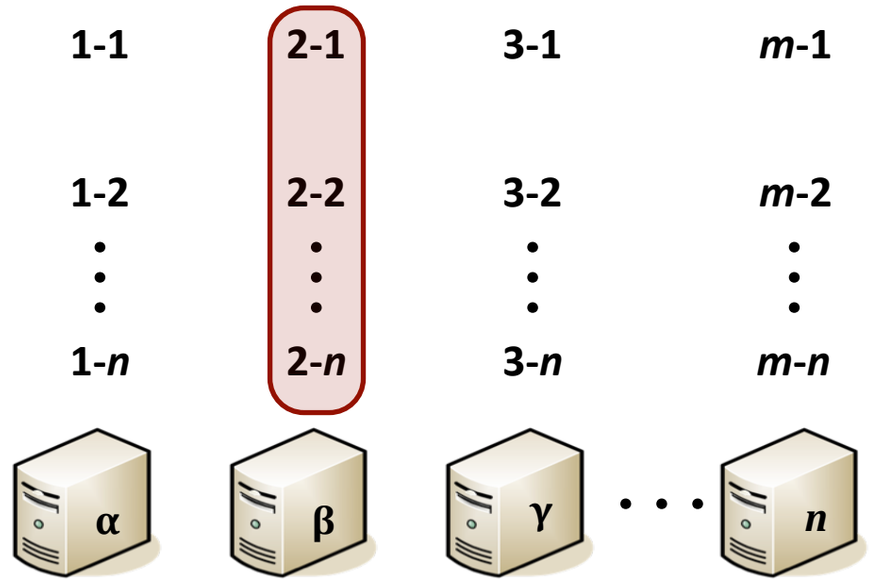


Average Degree
requires only one vertex to be examined to compute the value for each vertex.

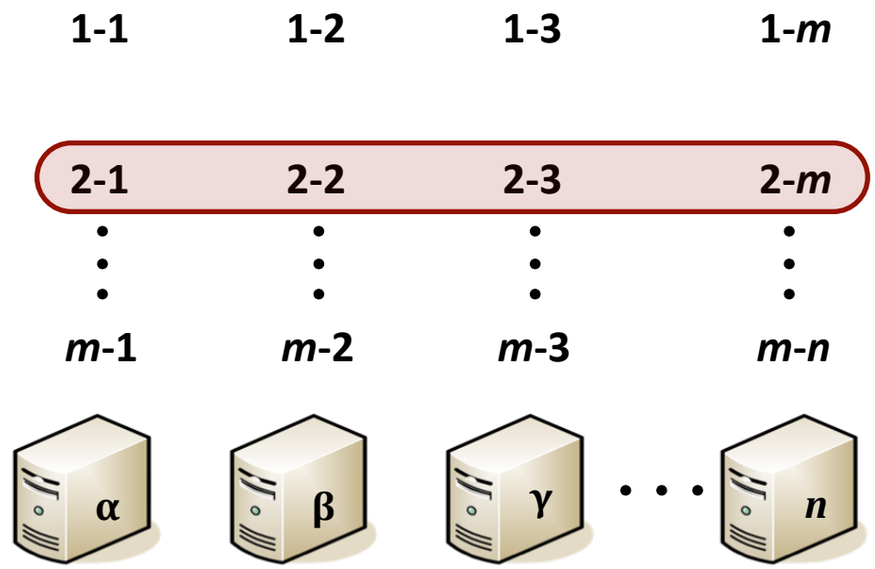


Distributions

Shared Nothing
stores all of the vertices comprising each snapshot on **one** of the ***n*** workers. Snapshot G_2 is stored entirely on worker β.



Shared Everything
stores the vertices comprising each snapshot such that they are shared among **all** of the ***n*** workers. Snapshot G_2 is shared across all workers.



Data Center Environment

IBM PureFlex hardware

- three 2.9 GHz Intel Xeon E5-2690 CPUs of 8 cores each (for 24 cores total)
- 131GB RAM
- dedicated 20TB storage area network

Software

- Ubuntu 14.04.2 LTS (`GNU/Linux 3.13.0-57-generic x86 64`) virtual machines on each of our three blades configured with access to all 8 cores and a 70GB SAN partition
- OpenJDK Runtime Environment (IcedTea 2.5.5).
- G* Database installed on one master and 23 workers, each taskset to a single core

Experiments

We computed the relative **speedup** for

- **PageRank** and **Average Degree** queries
- on **one snapshot** and **all snapshots** (23 evolutionary graphs)
- placed in **Shared Nothing** and **Shared Everything** distributions
- with vertex counts of **1K**, **2K**, **4K**, and **8K**

PageRank Queries	Snapshots	Shared Everything	Shared Nothing	Speedup
1K vertices	one			
1K vertices	all			
2K vertices	one			
2K vertices	all			
4K vertices	one			
4K vertices	all			
8K vertices	one			
8K vertices	all			

Avg. Degree Queries	Snapshots	Shared Everything	Shared Nothing	Speedup
1K vertices	one			
1K vertices	all			
2K vertices	one			
2K vertices	all			
4K vertices	one			
4K vertices	all			
8K vertices	one			
8K vertices	all			

Relative Speedup

Relative **speedup** is the average query execution time for a given query in Shared Everything placement (τ_{se}) divided by the average query execution time for Shared Nothing placement (τ_{sn}).

$$Speedup = \tau_{se} \div \tau_{sn}$$

Both **PageRank** and **Average Degree** queries were executed five times and the resulting execution times averaged.

PageRank Queries	Snapshots	Shared Everything	Shared Nothing	Speedup
1K vertices	one			
1K vertices	all			
2K vertices	one			
2K vertices	all			
4K vertices	one			
4K vertices	all			
8K vertices	one			
8K vertices	all			

Avg. Degree Queries	Snapshots	Shared Everything	Shared Nothing	Speedup
1K vertices	one			
1K vertices	all			
2K vertices	one			
2K vertices	all			
4K vertices	one			
4K vertices	all			
8K vertices	one			
8K vertices	all			

Results: PageRank on all snapshots

PageRank Queries	Snapshots	Shared Everything	Shared Nothing	Speedup
1K vertices	one	7184.8ms	5849.6ms	1.23
1K vertices	all	9263.2ms	7872.4ms	1.18
2K vertices	one	9267.2ms	7619.6ms	1.22
2K vertices	all	11395.8ms	9319.4ms	1.22
4K vertices	one	9305.0ms	7938.4ms	1.17
4K vertices	all	13637.0ms	9408.8ms	1.45
8K vertices	one	9823.8ms	9857.4ms	1.00
8K vertices	all	17795.2ms	11091.0ms	1.60

Common case:
Analysis of changing influence as networks evolve.

Remember: PageRank queries involve significant communication among vertices.

Note the **increasing** speedup of Shared Nothing over Shared Everything for PageRank queries on **all snapshots**.

Speedup grows as data set grows.

Shows
- benefit of parallelism vs. overhead of communication
- importance of snapshot placement

Results: Average Degree on all snapshots

Avg. Degree Queries	Snapshots	Shared Everything	Shared Nothing	Speedup
1K vertices	one	4640.6ms	2977.2ms	1.56
1K vertices	all	4877.6ms	4809.4ms	1.01
2K vertices	one	5892.4ms	4308.8ms	1.37
2K vertices	all	6343.6ms	6291.2ms	1.01
4K vertices	one	5497.4ms	4617.0ms	1.19
4K vertices	all	6369.8ms	6284.4ms	1.01
8K vertices	one	5787.6ms	4721.4ms	1.23
8K vertices	all	6605.4ms	6636.6ms	1.00

Common case:
Analysis of changing popularity as networks evolve.

Remember: Average Degree queries involve no communication among vertices.

Note the **consistency** in Shared Nothing and Shared Everything for Average Degree queries on **all shapshots**.

Speedup is unchanged as data set grows.

Shows
- benefits of "embarrassing" parallelism (no communication overhead)
- snapshot placement does not matter

Results: PageRank on one snapshot

PageRank Queries	Snapshots	Shared Everything	Shared Nothing	Speedup
1K vertices	one	7184.8ms	5849.6ms	1.23
1K vertices	all	9263.2ms	7872.4ms	1.18
2K vertices	one	9267.2ms	7619.6ms	1.22
2K vertices	all	11395.8ms	9319.4ms	1.22
4K vertices	one	9305.0ms	7938.4ms	1.17
4K vertices	all	13637.0ms	9408.8ms	1.45
8K vertices	one	9823.8ms	9857.4ms	1.00
8K vertices	all	17795.2ms	11091.0ms	1.60

Note the **decreasing** speedup of Shared Nothing over Shared Everything for PageRank queries on **one snapshot**.

Speedup shrinks as data set grows.

Shows

- as the data set grows, the benefit of parallelism decreases due to increasing communication overhead

Remember: PageRank queries involve significant communication among vertices.

Results: Average Degree on one snapshot

Avg. Degree Queries	Snapshots	Shared Everything	Shared Nothing	Speedup
1K vertices	one	4640.6ms	2977.2ms	1.56
1K vertices	all	4877.6ms	4809.4ms	1.01
2K vertices	one	5892.4ms	4308.8ms	1.37
2K vertices	all	6343.6ms	6291.2ms	1.01
4K vertices	one	5497.4ms	4617.0ms	1.19
4K vertices	all	6369.8ms	6284.4ms	1.01
8K vertices	one	5787.6ms	4721.4ms	1.23
8K vertices	all	6605.4ms	6636.6ms	1.00

Remember: Average Degree queries involve no communication among vertices.

Note the **decreasing** speedup of Shared Nothing over Shared Everything for Average Degree queries on **one shapshot**.

Speedup shrinks as data set grows.

Shows
- impact of communication overhead on parallelism
- importance of snapshot placement

Conclusions So Far

Even in a data center environment (with fast, shared resources), there are **significant benefits of careful snapshot placement** in the most common and important cases.

- When multiple snapshots are queried together and those queries involve significant communication (i.e., PageRank on all snapshots) it is advantageous to store each snapshot on fewer servers.
- But… it is important that the overall queried data be balanced over all servers to avoid hurting the performance of queries that do not involve significant communication among workers (i.e., Average Degree on one snapshot).

Next Steps

Our work continues. We're . . .

- experimenting with larger and different data sets
  - long-tail Barabasi-Albert graphs
  - social graphs generated by the Linked Data Benchmark Council benchmarking tool
- increasing the resolution of our execution time measurements to record data at the BSP superstep level.
- comparing our data center results with those obtained from experimenting on the 64-core server cluster we used in our prior work
  - This may suggest insight into graph query performance on clusters of connected computers with no shared resources as compared to data center environments with many shared resources.

Thank you. Questions?

Distributed Graph Snapshot Placement and Query Performance in a Data Center Environment

Alan G. Labouseur
School of Computer Science and Mathematics
Marist College
Poughkeepsie, NY 12601
Alan.Labouseur@Marist.edu

Justin Svegliato
School of Computer Science and Mathematics
Marist College
Poughkeepsie, NY 12601
Justin.Svegliato1@Marist.edu

Jeong-Hyon Hwang
Dept. of Computer Science
University at Albany
State University of New York
Albany, NY 12222
jhh@cs.albany.edu