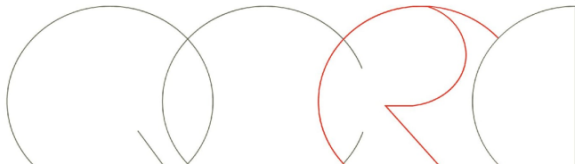
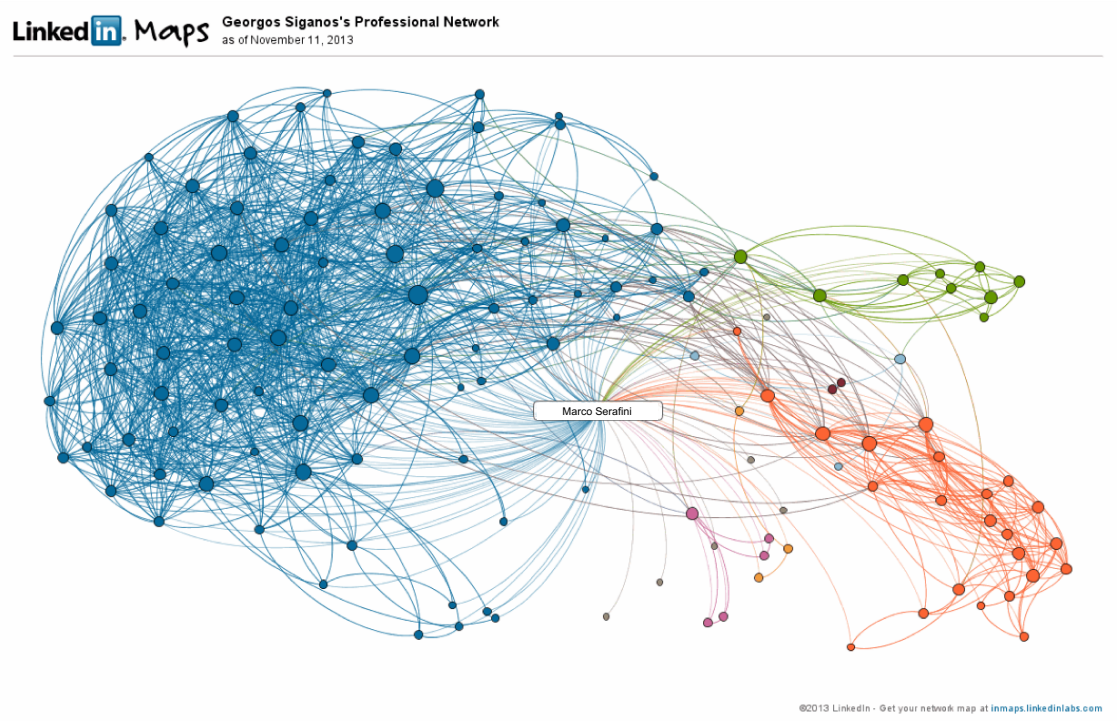
Graph Analytics

Graph Processing and Graph Mining

Marco Serafini, QCRI



Graphs are Ubiquitous



Classes of Graph Systems

- **Graph Computation**
  - Think like a vertex
  - Linear algebra

- **Graph Search**
  - Find instances of path expressions

- **Graph Mining**
  - Mine patterns of interest and their matches

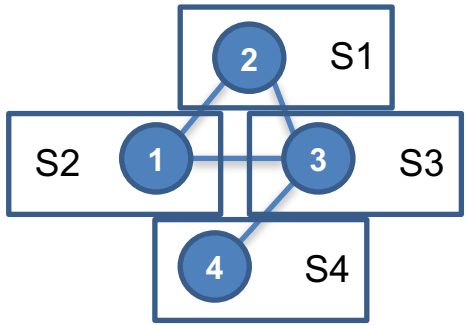
Graph Computation

Examples of Graph Computations

- PageRank
  - Used to rank vertices (e.g., web pages) in a graph (e.g., of hyperlinks)
  - Each vertex is given a relevance score based on its centrality
- Shortest path
  - Used for example for routing
- Bipartite matching
  - Match vertices on different sizes of a bipartite graph
  - Use for example for scheduling tasks based on capability

Google's Pregel

- Input graph as computational structure
  - Vertex → Computing thread with local state
  - Edges → Adjacency lists stored with to the vertex
  - Communication → Push messages to (neighboring) vertices
- "Think Like a Vertex (TLV)" programming
- Easy to scale to large graphs: partition by vertex



Bulk Synchronous Programming

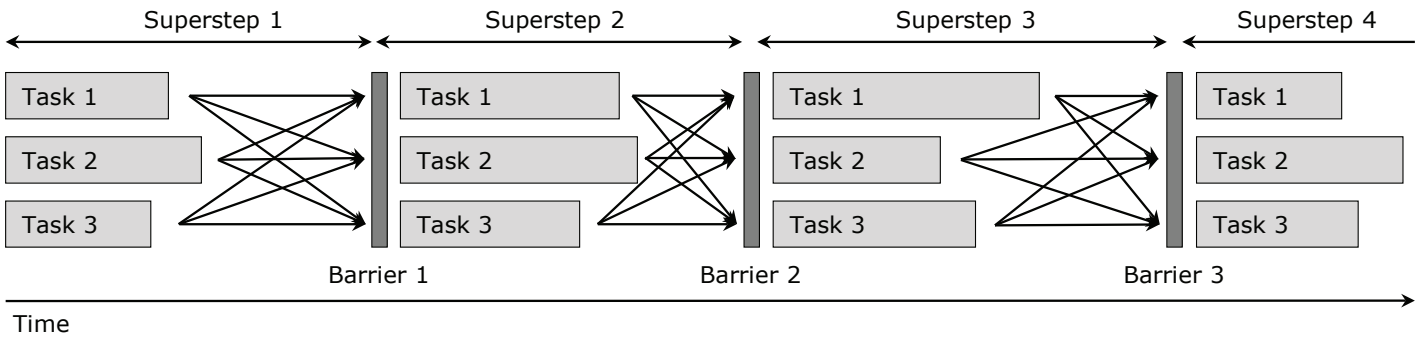


Figure taken from: [McCune *et al.*, Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing, ACM Comp. Surveys (2015)]

Termination

- Terminate when no more active vertices
- A vertex can locally vote to **halt** and deactivate
- If halting vertex receives a message → activates

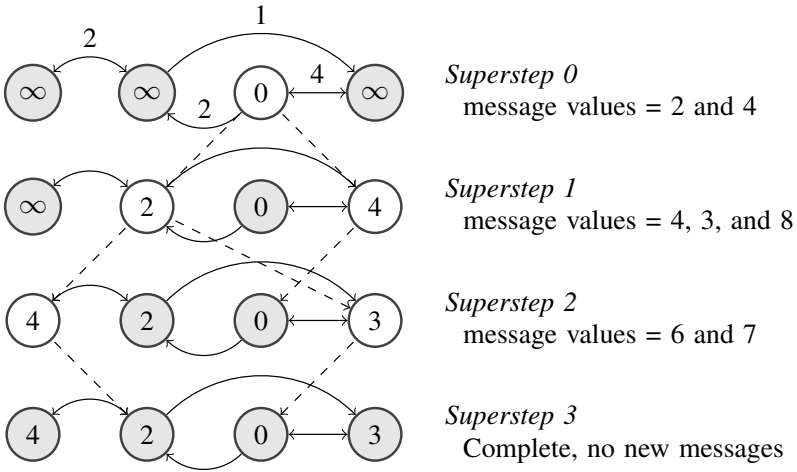
Example: S.S. Shortest Path

- **Input:** Graph (weighted edges), source vertex
- **Output:** Min distance between the source and all other vertices
- TLV implementation

```
vertex code:
    Receive distances from neighbors, extract minimum
    If minimum is smaller than current distance
        Replace current distance with minimum
        For each edge
            Send current distance + edge weight
    Halt
```

Example of SSSP Run

```
vertex code:
    Receive distances from neighbors, extract minimum
    If minimum is smaller than current distance
        Replace current distance with minimum
        For each edge
            Send current distance + edge weight
    Halt
```



How to execute this algorithm in MapReduce?

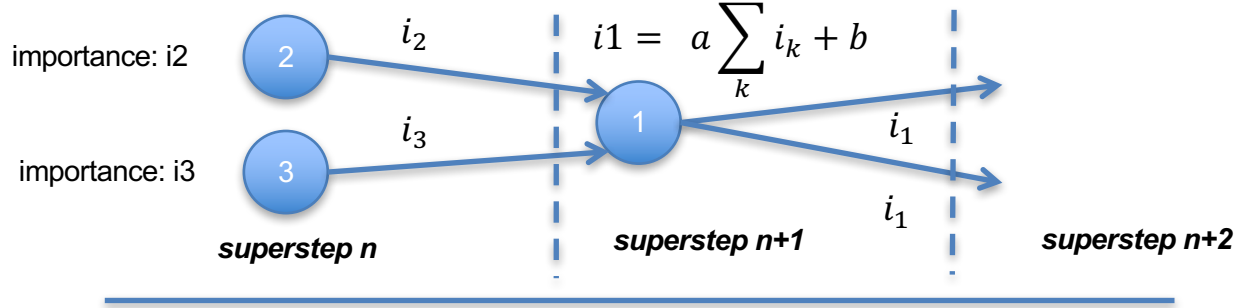
Figure taken from: [McCune *et al.*, Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing, ACM Comp. Surveys (2015)]

Combiners and Aggregators

- Combiners
  - Workers merge local messages to the same vertex
  - Example in SSSP?
- Aggregators
  - Similar to map-reduce: vertices map, aggregators reduce
  - Aggregation occurs in between supersteps
    - Each worker executes local aggregation
    - Aggregation tree
  - Example: compute the distribution of distances in SSSP

Matrix-Vector Multiplication in TLV

- E.g. Page-Rank style computation



links to v1

0	a12	a13
…	…	…

adjacency matrix
(transposed)

*

i1
i2
i3

importance

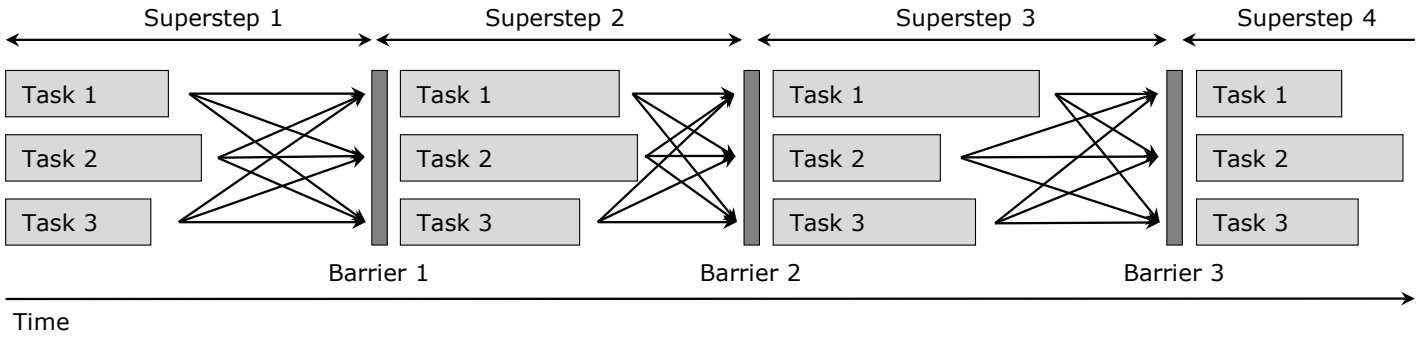
=

i2 + i3
…
…

new importance

→ *a (i2+i3) + b*

Problems with BSP?

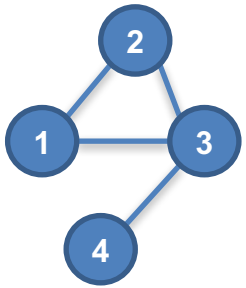


- Stragglers (workers that take longer to complete task)
- Need to balance load carefully. But how?

Asynchrony

GraphLab [2]

- Shared memory, asynchronous
- Vertices and edges have state
- Update functions
  - Modify a vertex, its incident edges, its neighbors
  - Schedule future updates on other vertices
- Serializable update functions using locking



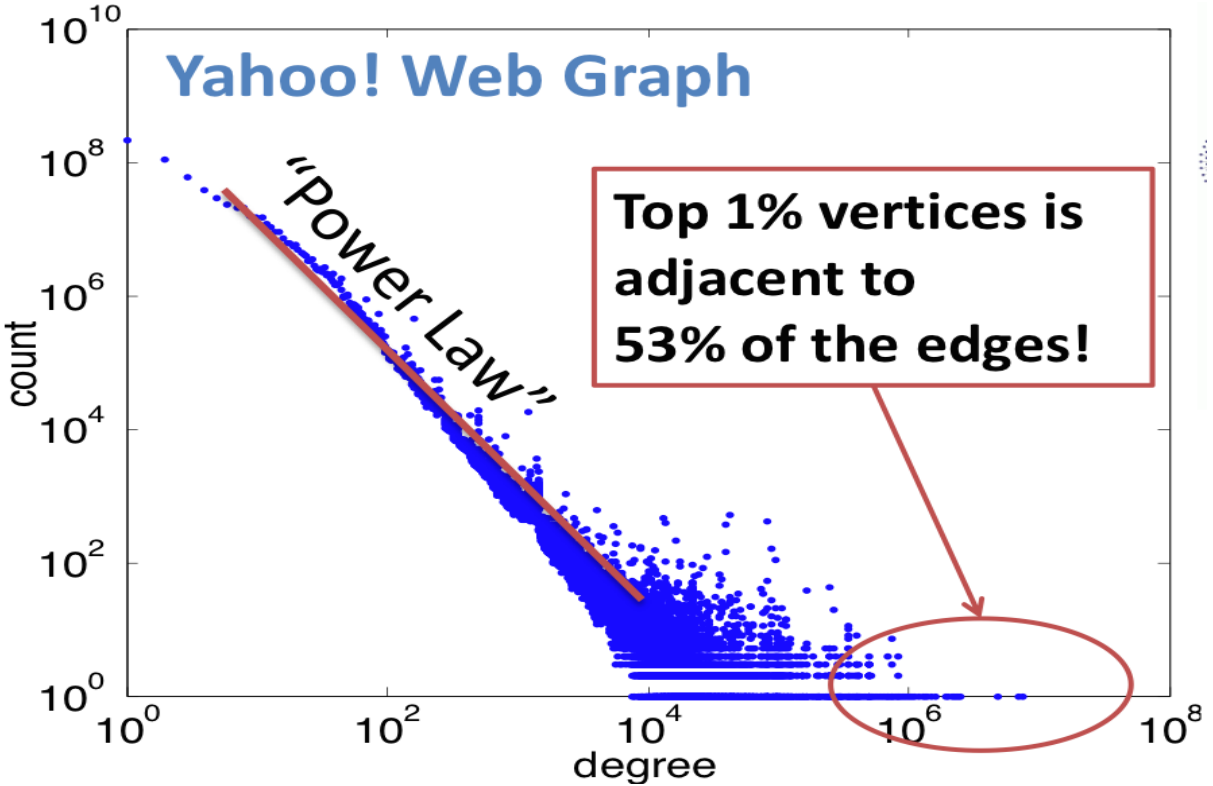
Reference: [2] Low et al, arXiv:1006.4990, (2010)

Is Asynchrony Better?

- In practice, benefits are not (yet) evident
  - Locking is expensive
  - Stragglers can still lock out other vertices
  - Application behavior is harder to understand
  - Not any longer supported even by GraphLab

HotSpots

Hotspot Vertices in Natural Graphs

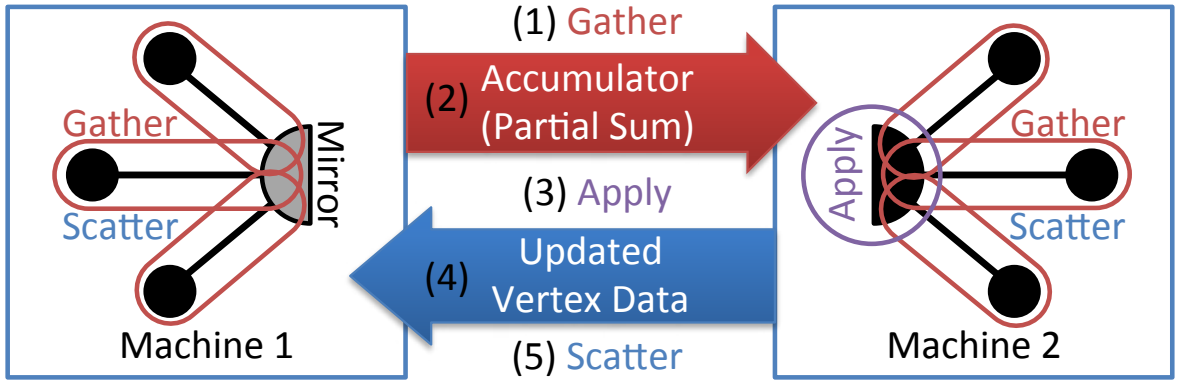


What are the issues with Hotspots?

- Long time to process all incoming messages
- Lots of output messages
- Lots of edge metadata to keep
- Delay overall execution
  - Hotspots are stragglers in synchronous systems
  - Hotspots involve lots of locking with asynchrony

PowerGraph[3]

- Replicate high degree vertices
- Gather, Apply, Scatter



Reference: [3] Gonzales et al, OSDI 2012

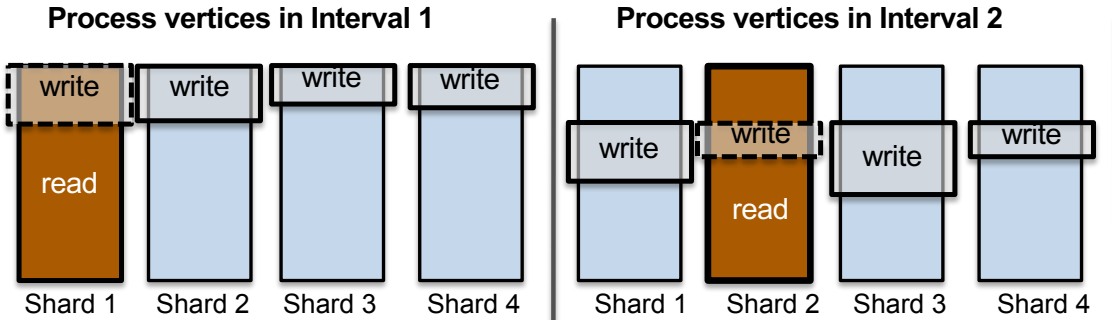
Large Graphs with 1 Server

Out-Of-Core Engines

- Use a single machine
- Instead of distributing graph, store it on disk
- Efficient I/O
  - Sequential read-writes: high bandwidth and low latency
  - Random accesses: very inefficent
  - This holds not only for disks but also SSDs

GraphChi [6]

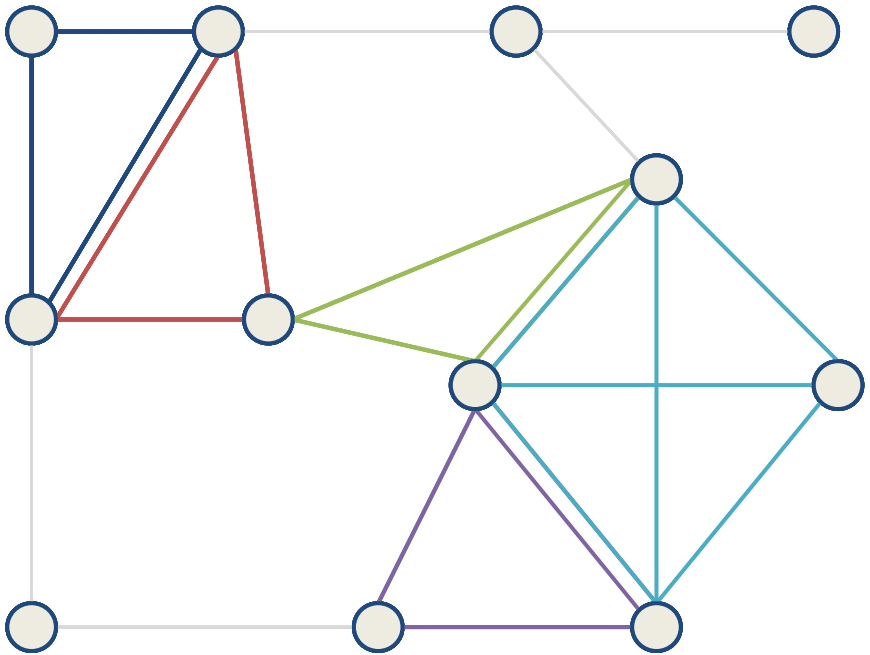
- Parallel Sliding Window
  - Sort vertices by ID and partition them in P **intervals**
  - Each interval is associated with a **shard** of **edges** with **destination** in the interval (sequential **reads**)
  - Sort edges in a shard by **source id** (sequential **writes**)



Reference: [6] Kyrola et al, OSDI 2012

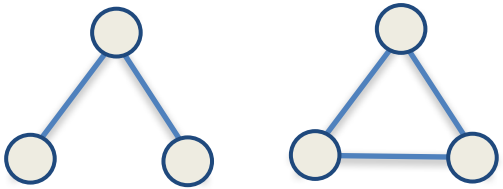
Graph Mining

Graph Mining: Cliques

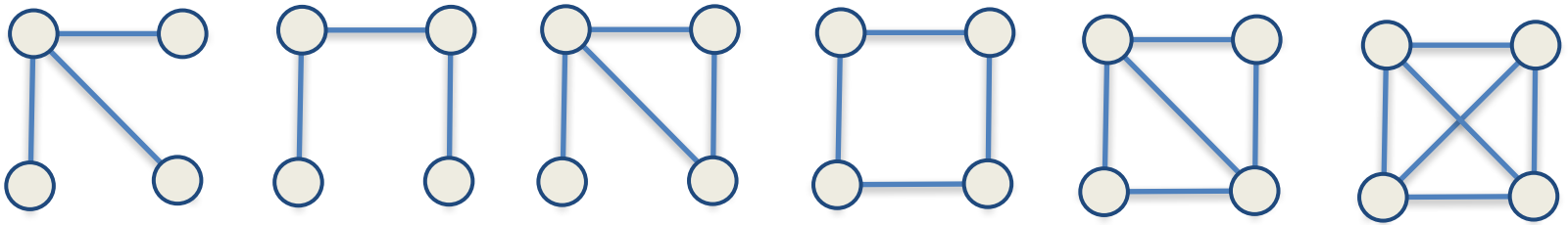


Graph Mining: Motifs

Motifs Size = 3

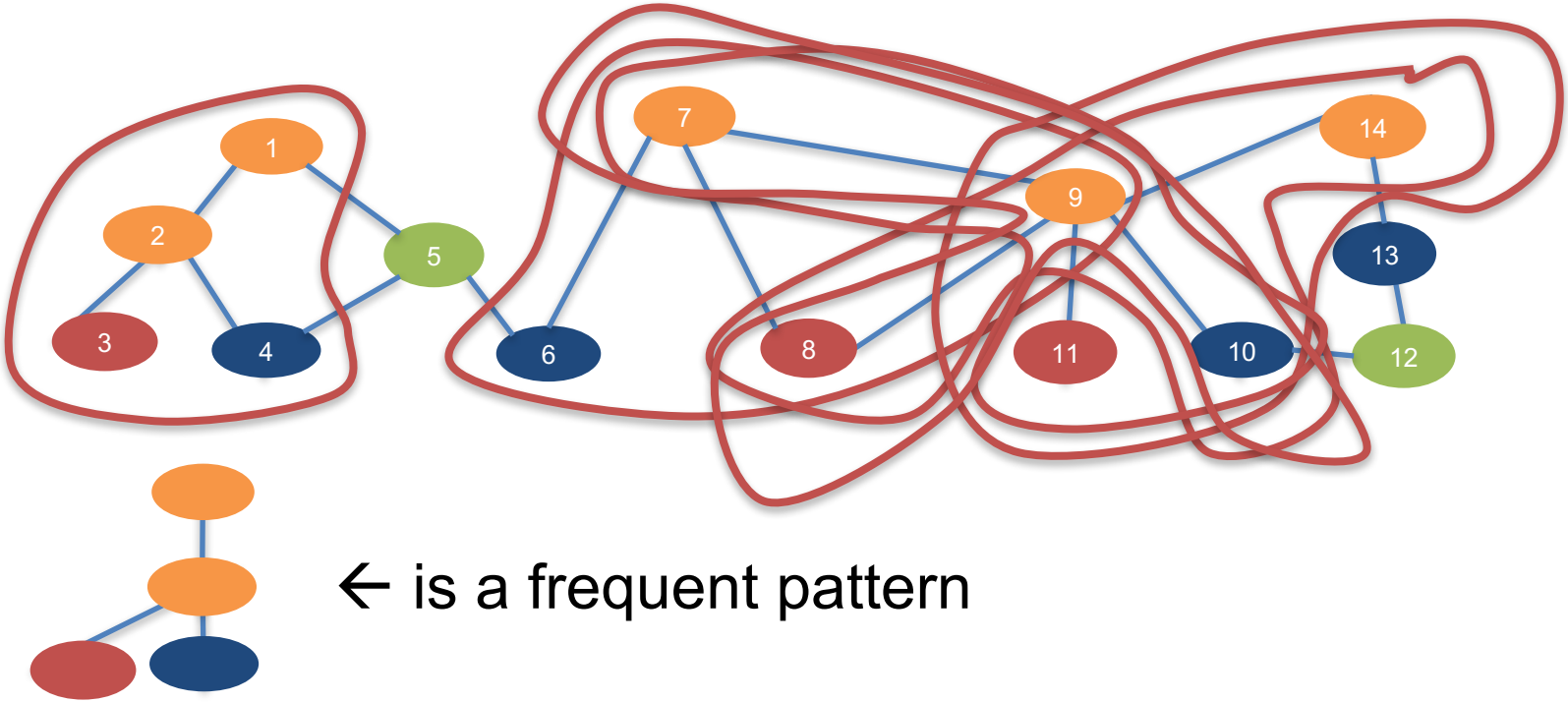


Motifs Size = 4



Graph Mining: FSM

- Frequent Subgraph mining in a single large graph.



Applications: Web and Advertising

- Link spam detection
  - Users, Web pages → Web pages
  - Dense bipartite subgraphs: Spammers

- Identify sub-markets
  - Advertisers / users → queries
  - Dense bipartite subgraphs: Advertisers / users forming sub-markets

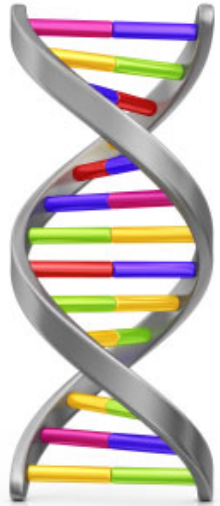
- Attributed edges in RDF datasets
  - Infer missing edges from recurrent patterns

Applications: Social Computing

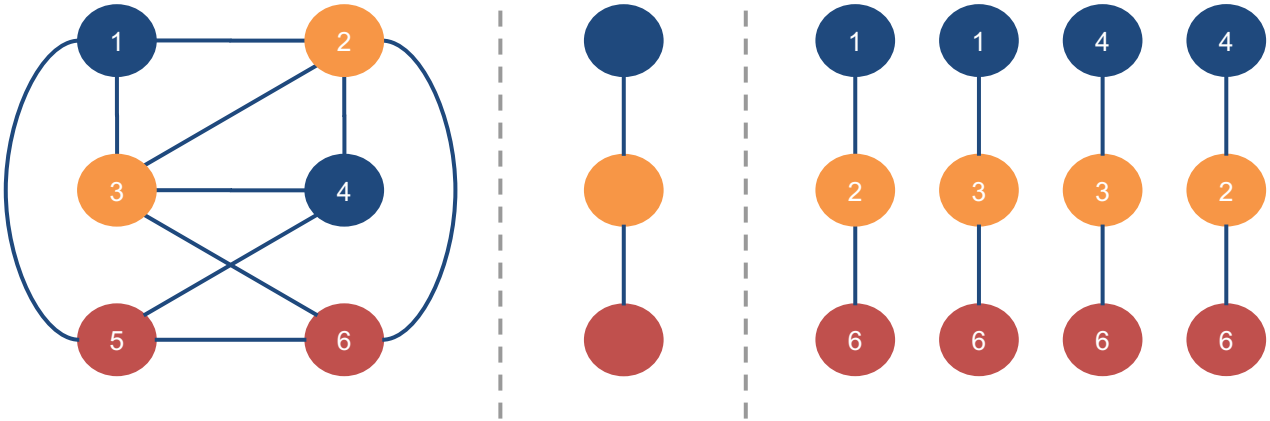
- Friend recommendation
  - Missing edges in quasi-cliques
- Community detection
- Finding stories in social media
  - Vertices: Entities (e.g. politicians)
  - Edges: Co-occurrence in tweets
  - Frequently co-occurring entities are involved in a story

Applications: Biology

- DNA motif detection
- Gene correlation
- Protein-protein interaction



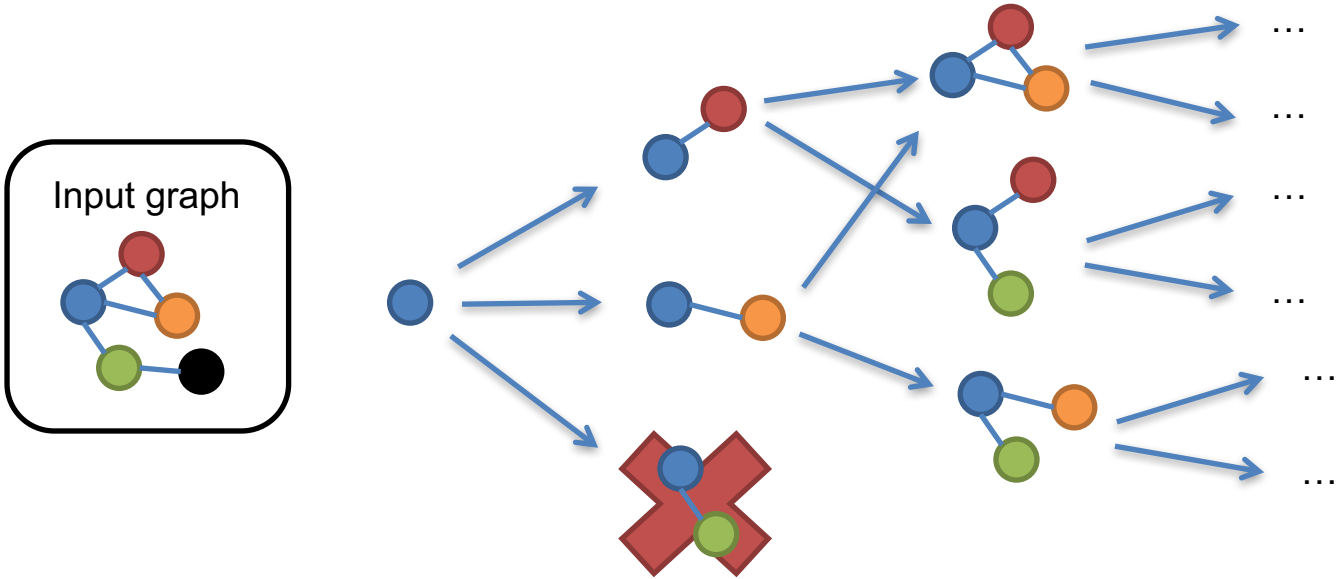
Graph Mining - Concepts



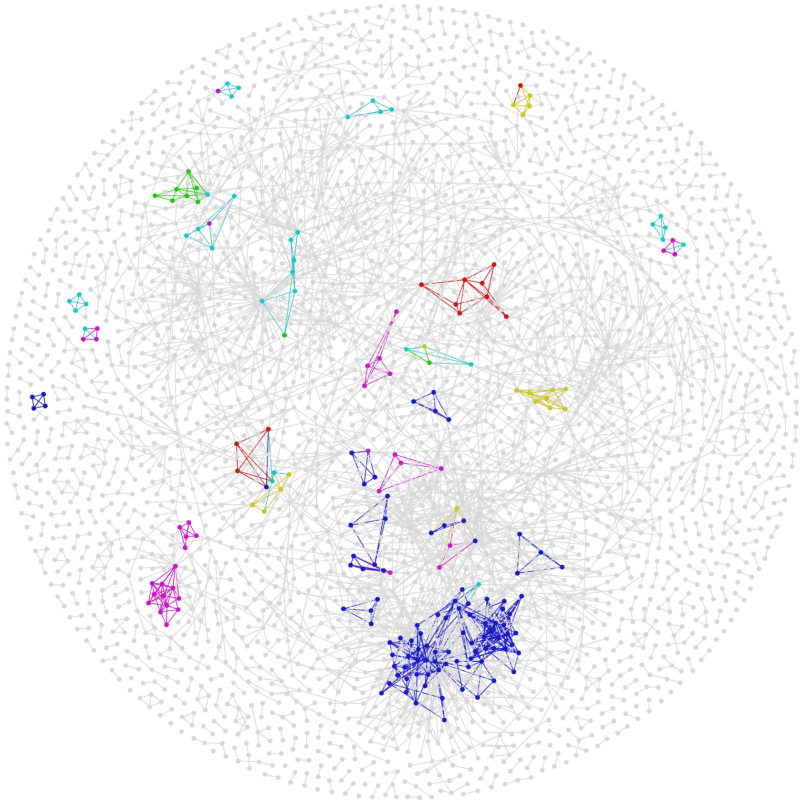
Input graph Pattern Embeddings

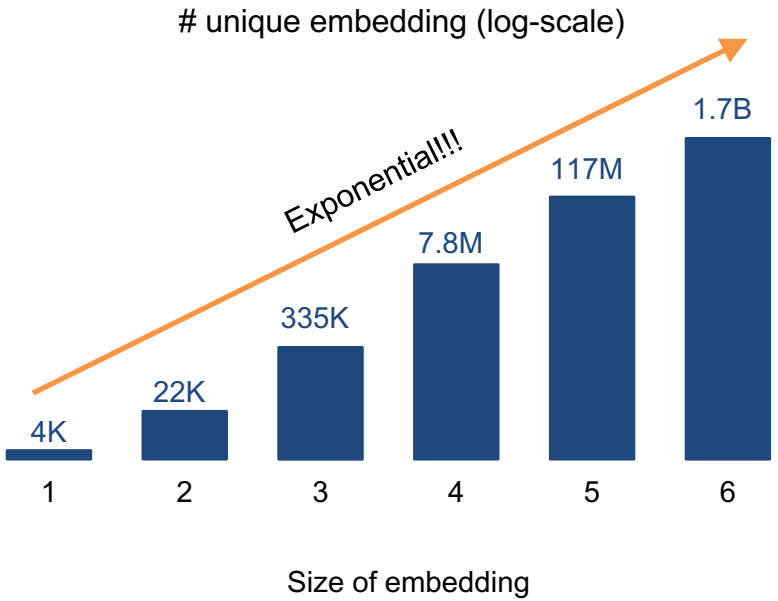
Graph Exploration

- Enumerate (& prune) embeddings
- Aggregate by pattern



Challenges





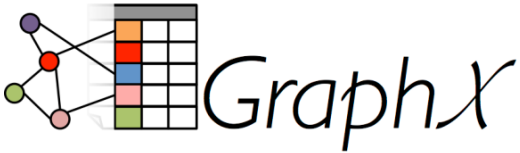
Exponential number of embeddings

State of the Art: Custom Algorithms

	Easy to Code	Efficient Implementation	Transparent Distribution
Custom Algorithms	✗	✓	✗

State of the Art: Think Like a Vertex

	Easy to Code	Efficient Implementation	Transparent Distribution
Custom Algorithms	✗	✓	✗
Think Like a Vertex	✗	✗	✓



Arabesque [7]

- New execution model & system
  - Think Like an Embedding
  - Purpose-built for distributed graph mining
  - Hadoop-based

- Contributions:
  - Simple & Generic API
  - High performance
  - Distributed & Scalable by design

Reference: [7] Teixeira et al, SOSP 2015

Arabesque

	Easy to Code	Efficient Implementation	Transparent Distribution
Custom Algorithms	✗	✓	✗
Think Like a Vertex	✗	✗	✓
Arabesque	✓	✓	✓

API Example: Clique finding

```
1  boolean filter(Embedding e) {
2      return isClique(e);
3  }
4  void process(Embedding e) {
5      output(e);
6  }
7  boolean shouldExpand(Embedding embedding) {
8      return embedding.getNumVertices() < maxsize;
9  }
10 boolean isClique(Embedding e) {
11     return e.getNumEdgesAddedWithExpansion()==e.getNumberOfVertices()-1;
12 }
```

State of the Art
(Mace, centralized)

4,621 LOC

API Example: Motif Counting

```
1  boolean filter(Embedding e) {
2      return true;
3  }
4  void process(Embedding embedding) {
5          output(embedding);
6          map(AGG_MOTIFS, embedding.getPattern(), reusableLongWritableUnit);
7  }
8  boolean shouldExpand(Embedding embedding) {
9      return embedding.getNumVertices() < maxsize;
10 }
```

State of the Art
(GTrieScanner, centralized)

3,145 LOC

API Example: FSM

- Ours was the first distributed implementation
- 280 lines of Java Code
  - … of which 212 compute frequent metric

- Baseline (GRAMI): 5,443 lines of Java code.

Arabesque: An Efficient System

- As efficient as centralized state of the art

Application - Graph	Centralized Baseline	Arabesque 1 thread
Motifs - MiCo (MS=3)	50s	37s
Cliques - MiCo (MS=4)	281s	77s
FSM - CiteSeer (S=300)	4.8s	5s

Arabesque: A Scalable System

- Scalable to thousands of workers
- Hours/days → Minutes

Application - Graph	Centralized Baseline	Arabesque 640 cores
Motifs -	2 hours 24 minutes	25 seconds
Cliques	4 hours 8 minutes	1 minute 10 seconds
FSM - Patents	> 1 day	1 minute 28 seconds

First Distributed Implementation

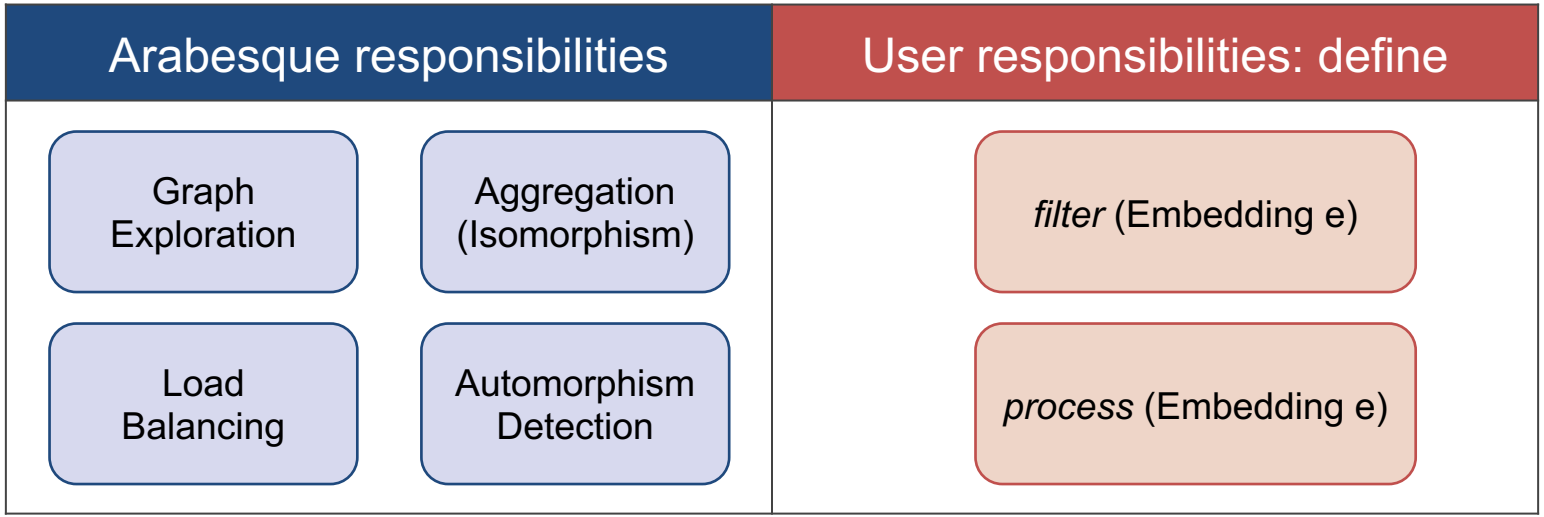
How: Arabesque Optimizations

- Avoid Redundant Work
  - Efficient canonicality checking
- Subgraph Compression
  - Overapproximating Directed Acyclic Graphs (ODAGs)
- Efficient Aggregation
  - 2-level pattern aggregation

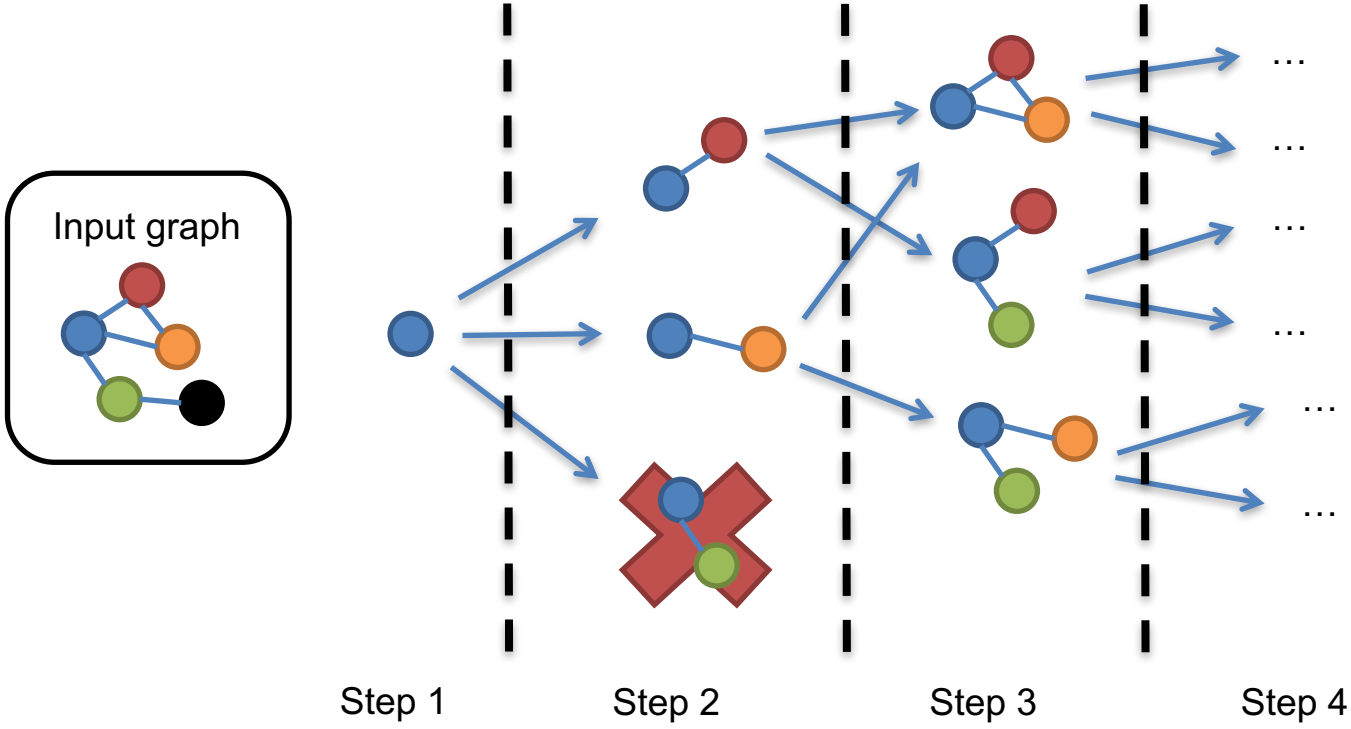
Arabesque fundamentals

Arabesque: Fundamentals

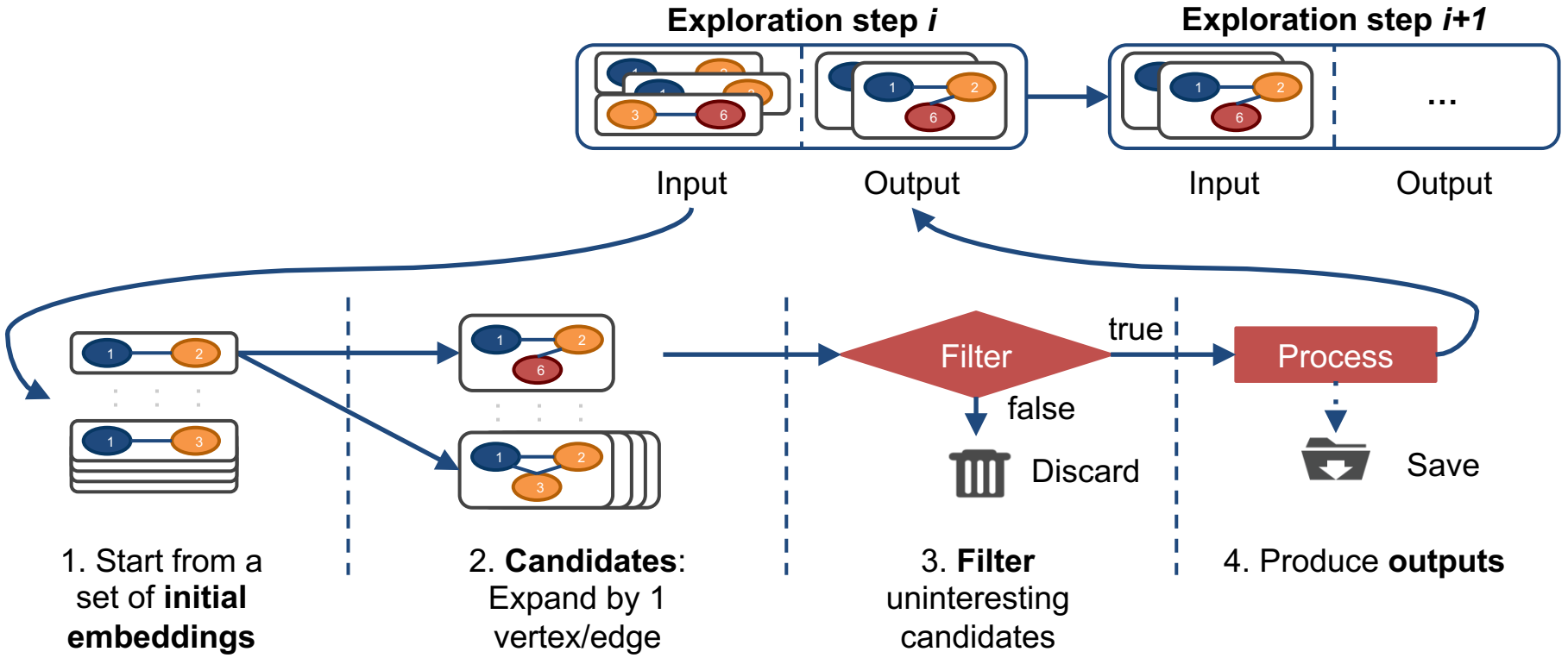
- Embeddings as 1st class citizens:
  - **Think Like an Embedding** model
  - Functional programming



Graph Exploration



Model - Think Like an Embedding



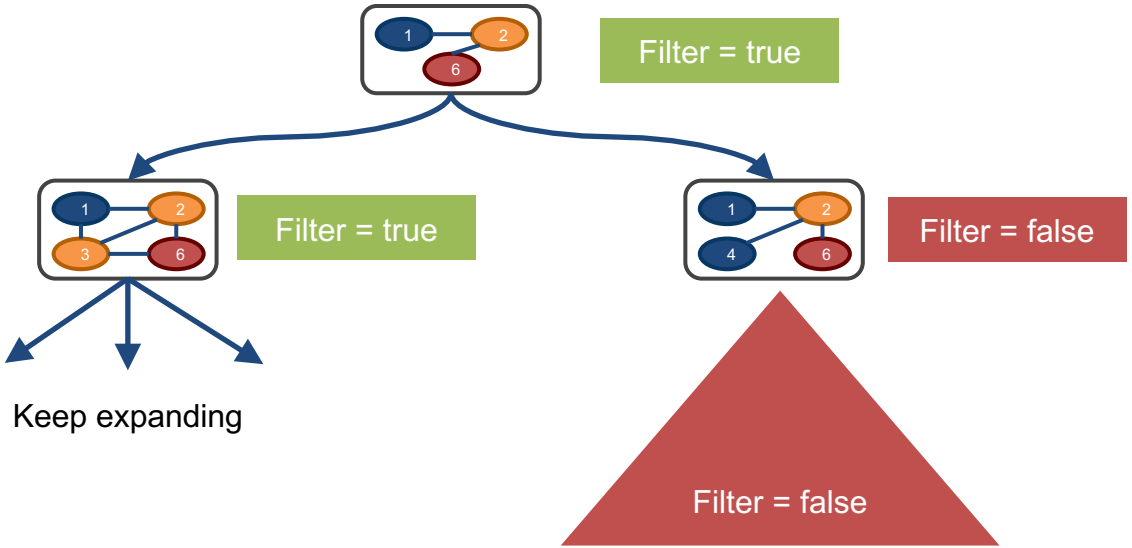
API Example: Clique finding

```
1  boolean filter(Embedding e) {
2      return isClique(e);
3  }
4  void process(Embedding e) {
5      output(e);
6  }
7  boolean shouldExpand(Embedding embedding) {
8      return embedding.getNumVertices() < maxsize;
9  }
10 boolean isClique(Embedding e) {
11     return e.getNumEdgesAddedWithExpansion()==e.getNumberOfVertices()-1;
12 }
```

Guarantee: Completeness

For each e, if filter(e) == true then Process(e) is executed

Requirement: Anti-monotonicity



We can **prune** and be sure that we won't ignore desired embeddings

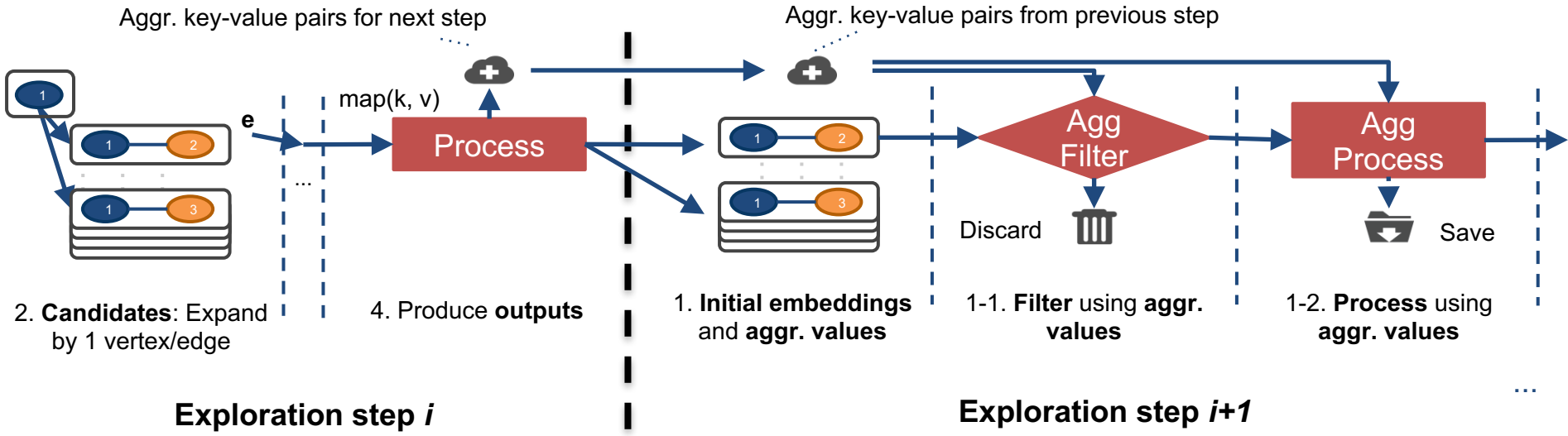
Aggregation during expansion

- Filter might need aggregated values
  - E.g.: Frequent subgraph mining
    - Frequency calculation → look at all candidates

- Aggregation in parallel with exploration step
  - Embeddings filtered as soon as aggregated values are ready.

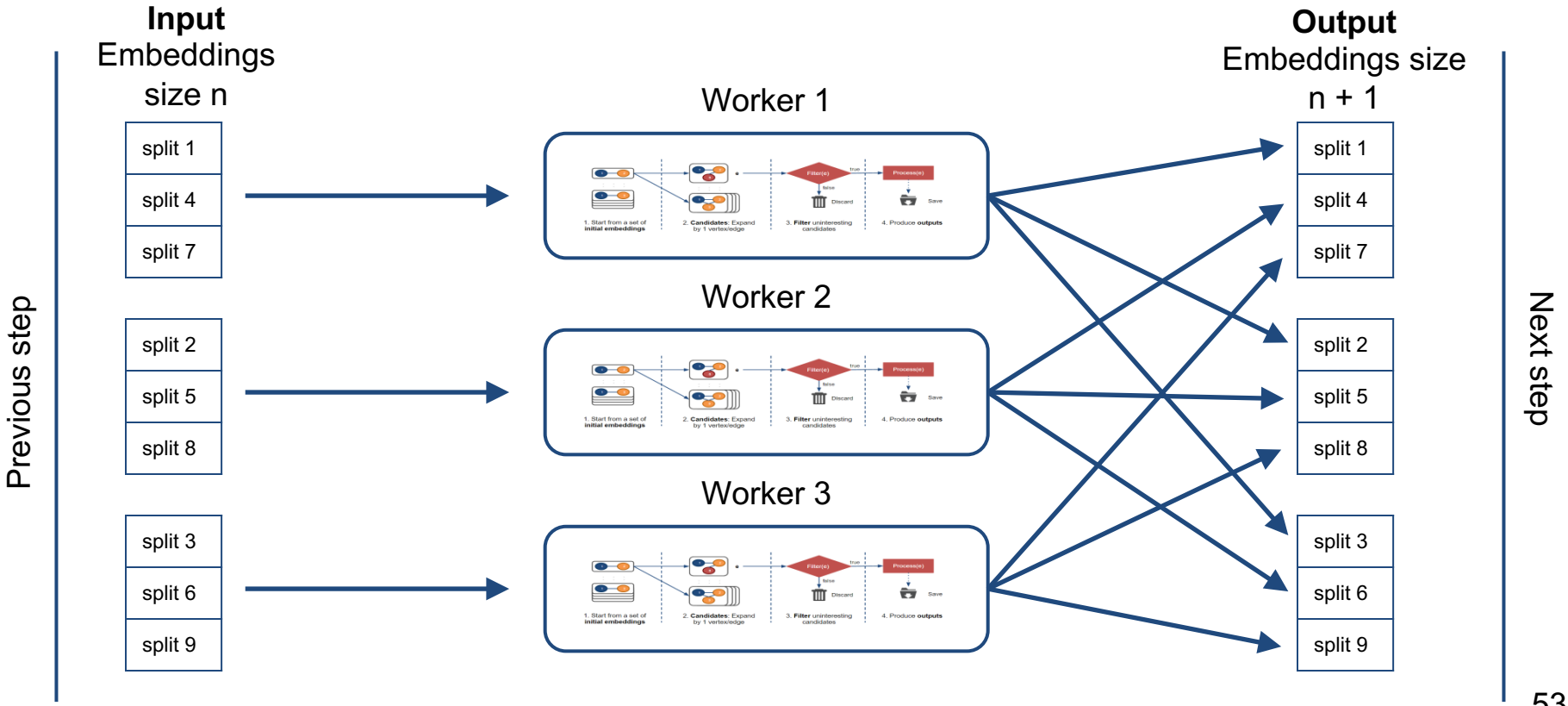
Aggregation during expansion

- Filter function may depend on aggregated data across all embeddings of the same pattern.



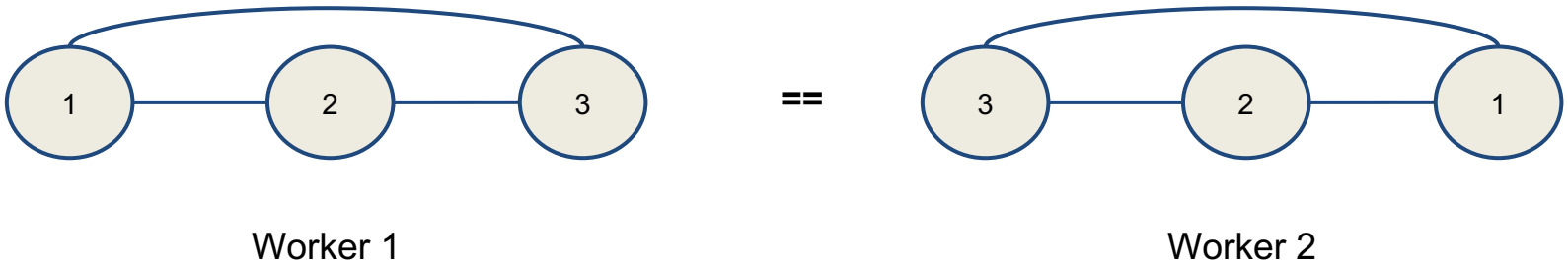
System Architecture & Optimizations

Arabesque Architecture



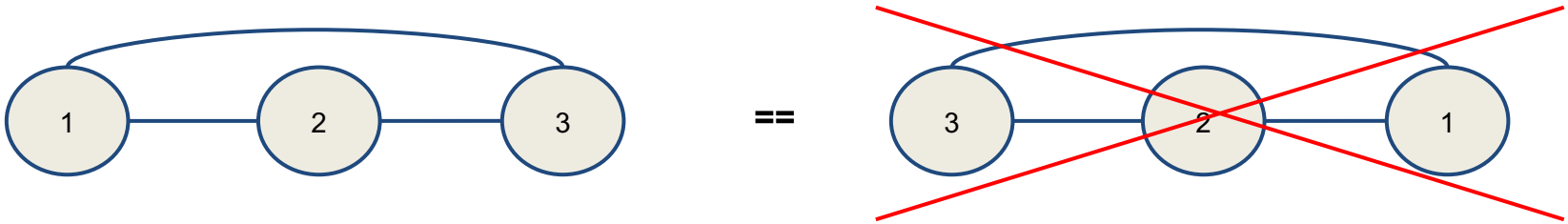
Avoiding redundant work

- **Problem:** Automorphic embeddings
  - Automorphisms == subgraph equivalences
  - Redundant work



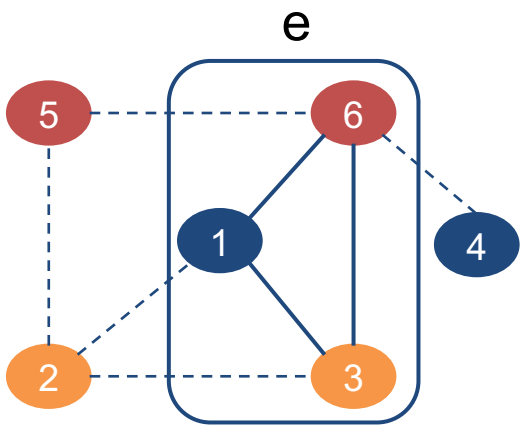
Avoiding redundant work

- **Solution: Decentralized Embedding Canonicality**
  - No coordination
  - Efficient



Embedding Canonicality

- isCanonical(e) *iff* at every step add neighbor with smallest ID



Initial embedding (e)

- 1 - 3 - 6

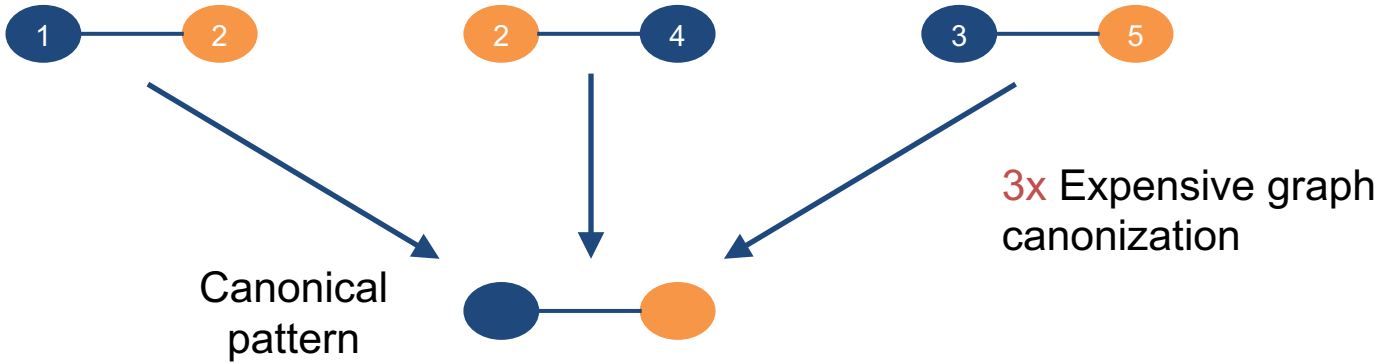
Expansions:

- 1 - 3 - 6 - 5 → canonical
- 1 - 3 - 6 - 4 → canonical

- 1 - 3 - 6 - 2 → not canonical (1 - 2 - 3 - 6)

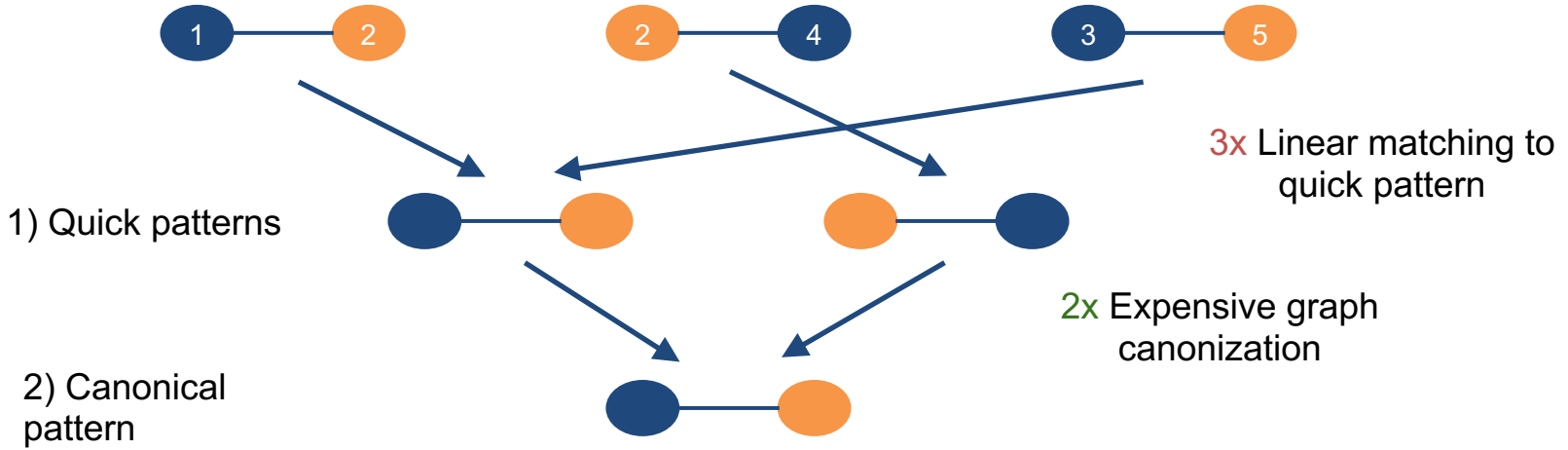
Efficient Pattern Aggregation

- **Goal:** Aggregate automorphic patterns to single key
  - Find canonical pattern
    - No known polynomial solution



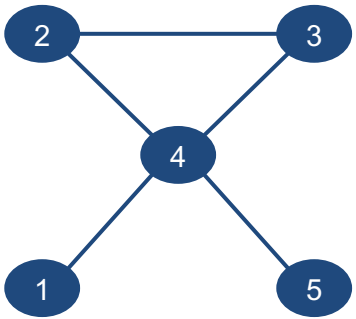
Efficient Pattern Aggregation

- **Solution:** 2-level pattern aggregation
  1. Embeddings → quick patterns
  2. Quick patterns → canonical pattern



Handling Exponential growth

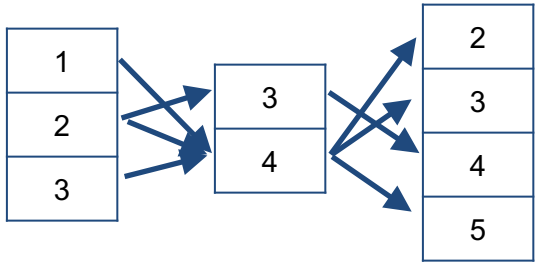
- **Goal:** handle trillions+ different embeddings?
- **Solution:** **Overapproximating DAGs (ODAGs)**
  - Compress into less restrictive superset
  - Deal with spurious embeddings



Input Graph

Canonical Embeddings		
1	4	2
1	4	3
1	4	5
2	3	4
2	4	5
3	4	5

Embedding List



ODAG

Evaluation

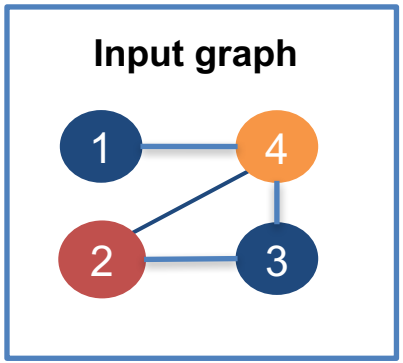
Evaluation - Setup

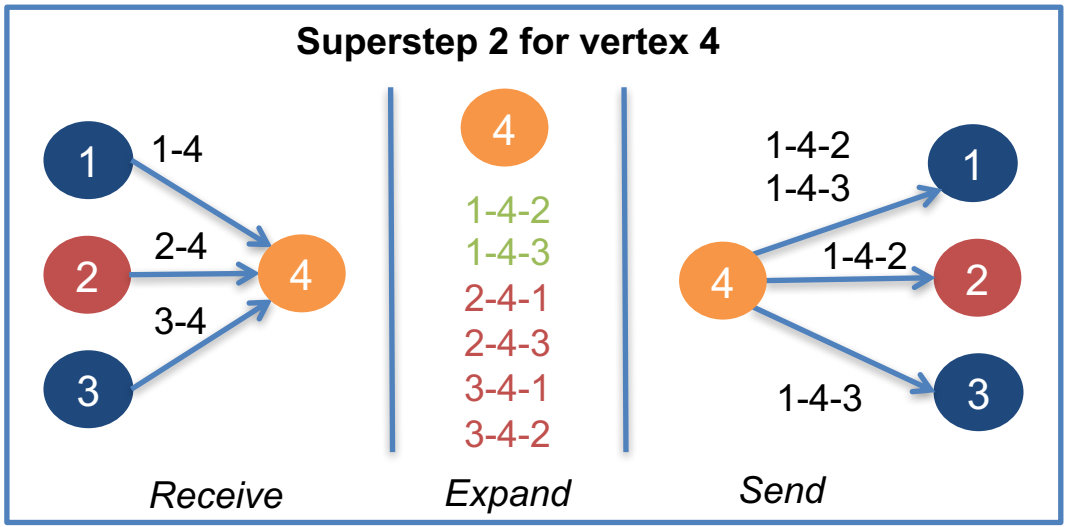
- 20 servers: 32 threads @ 2.67 GHz, 256GB RAM
- 10 Gbps network
- 3 algorithms: Frequent Subgraph Mining, Counting Motifs and Clique Finding
- Input graphs:

	# Vertices	# Edges	# Labels	Avg. Degree
CiteSeer	3,312	4,732	6	3
MiCO	100,000	1,080,298	29	22
Patents	2,745,761	13,965,409	37	10
Youtube	4,589,876	43,968,798	80	19
SN	5,022,893	198,613,776	0	79
Instagram	179,527,876	887,390,802	0	10

Graph Exploration with TLV

1. Receive embeddings
2. Expand by adding neighboring vertices
3. Send *canonical* embeddings to their constituting vertices





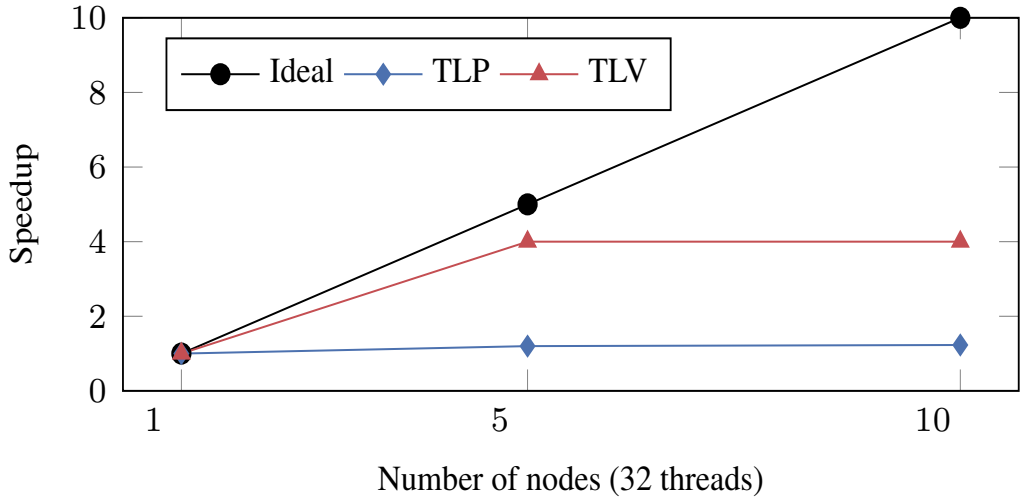
Think Like a Pattern

- Many existing algorithms keep state by pattern
- Advantages
  - Rebuild embeddings from scratch
  - No need to materialize full intermediate state

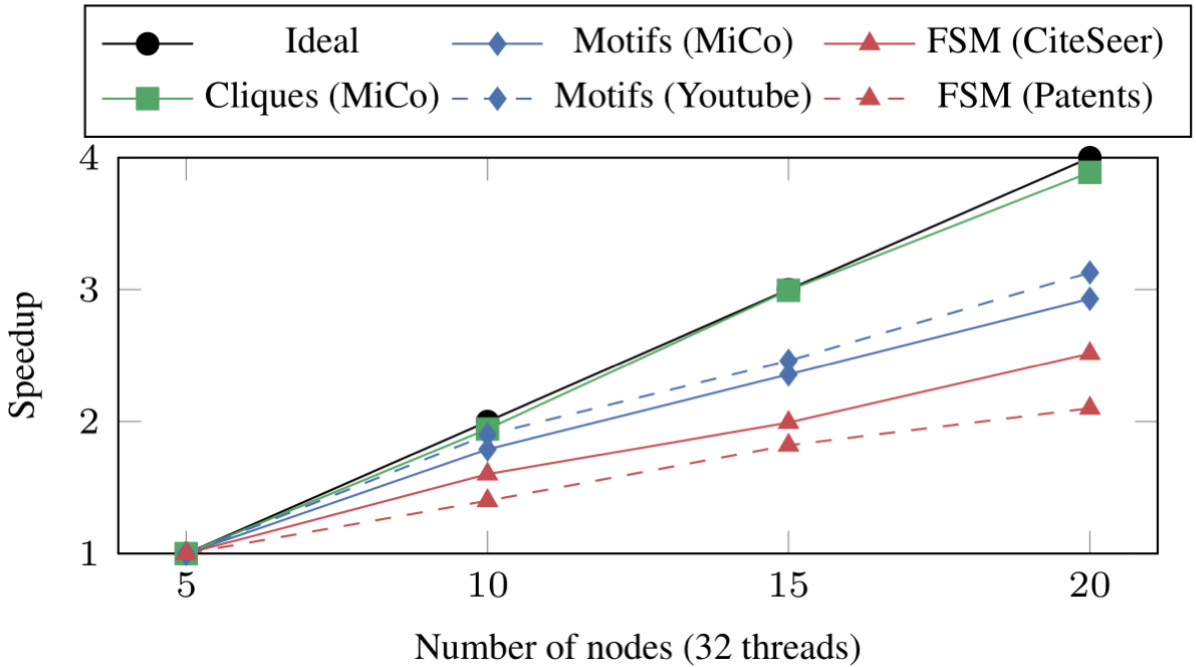
- Idea of TLP:
  - Assign different patterns to different machines
  - Avoid storing materialized embedding

Evaluation - TLP & TLV

- Use case: frequent subgraph mining
- No scalability. Bottlenecks:
  - TLV: Replication of embeddings, hotspots
  - TLP: very few patterns do all the work



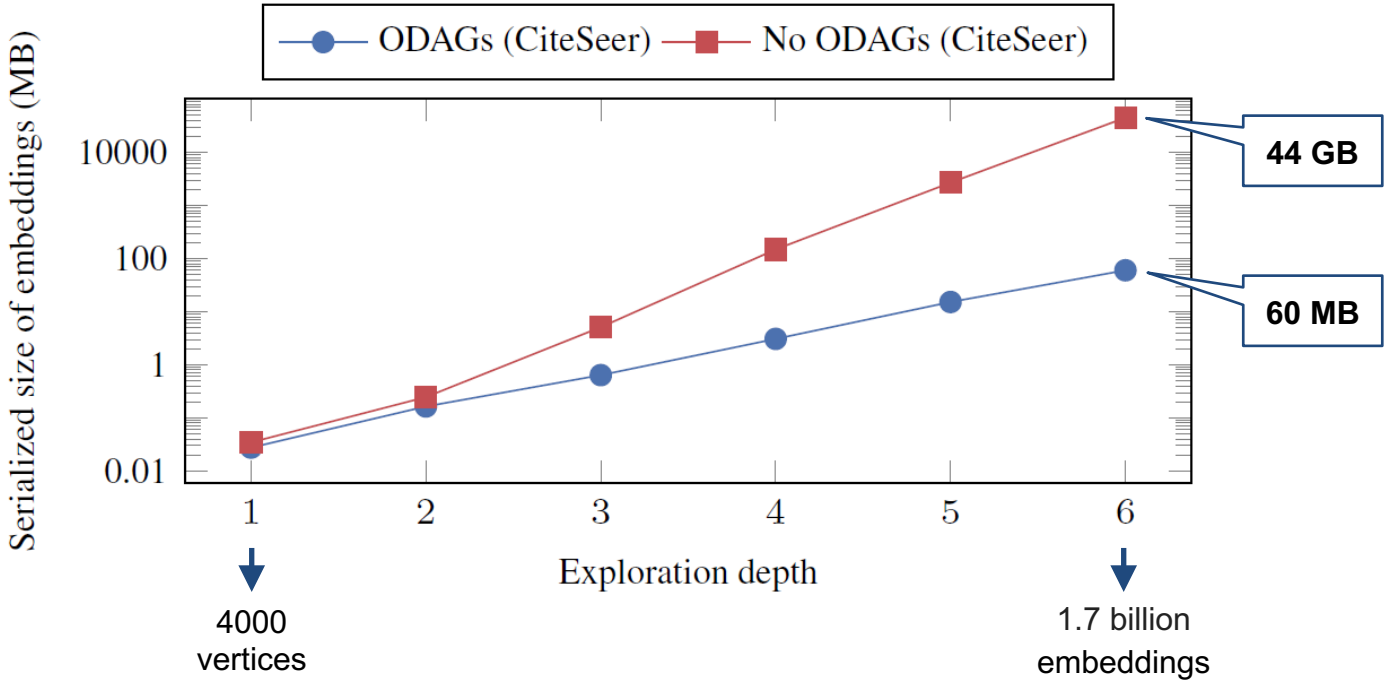
Evaluation – Arabesque Scalability



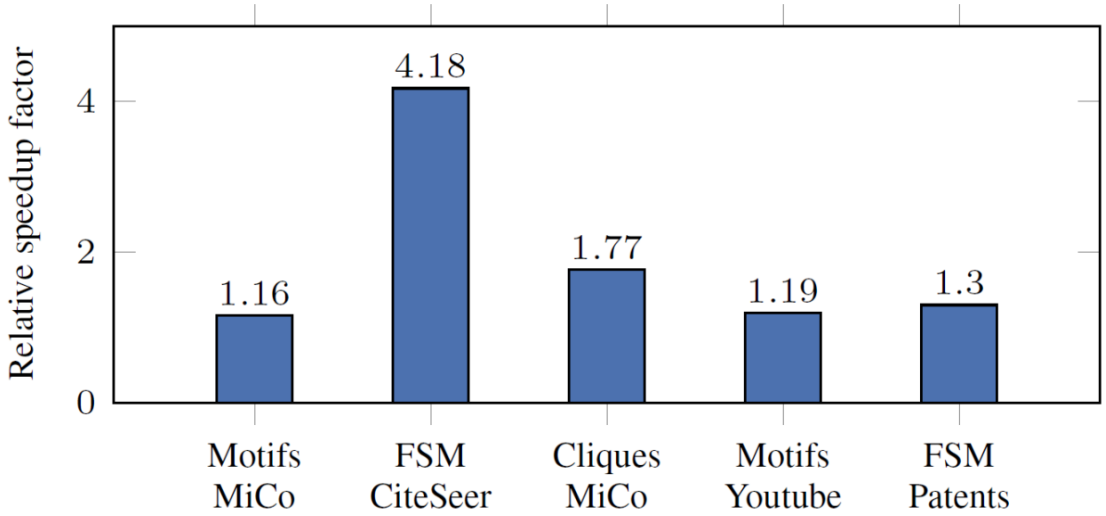
Evaluation – Arabesque Scalability

Application - Graph	Centralized Baseline	Arabesque - Num. Servers (32 threads)				
		1	5	10	15	20
Motifs - MiCo	8,664s	328s	74s	41s	31s	25s
FSM - Citeseer	1,813s	431s	105s	65s	52s	41s
Cliques - MiCo	14,901s	1,185s	272s	140s	91s	70s
Motifs - Youtube	Fail	8,995s	2,218s	1,167s	900s	709s
FSM - Patents	>19h	548s	186s	132s	102s	88s

Evaluation - ODAGs Compression

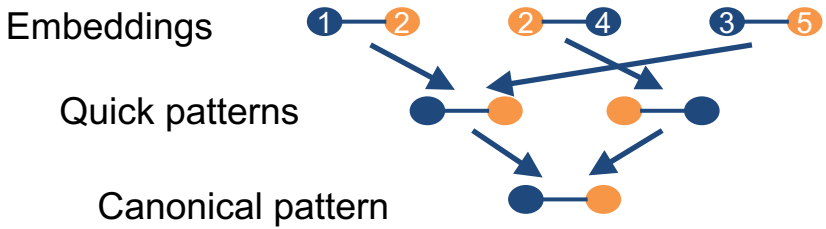


Evaluation - Speedup w ODAGs

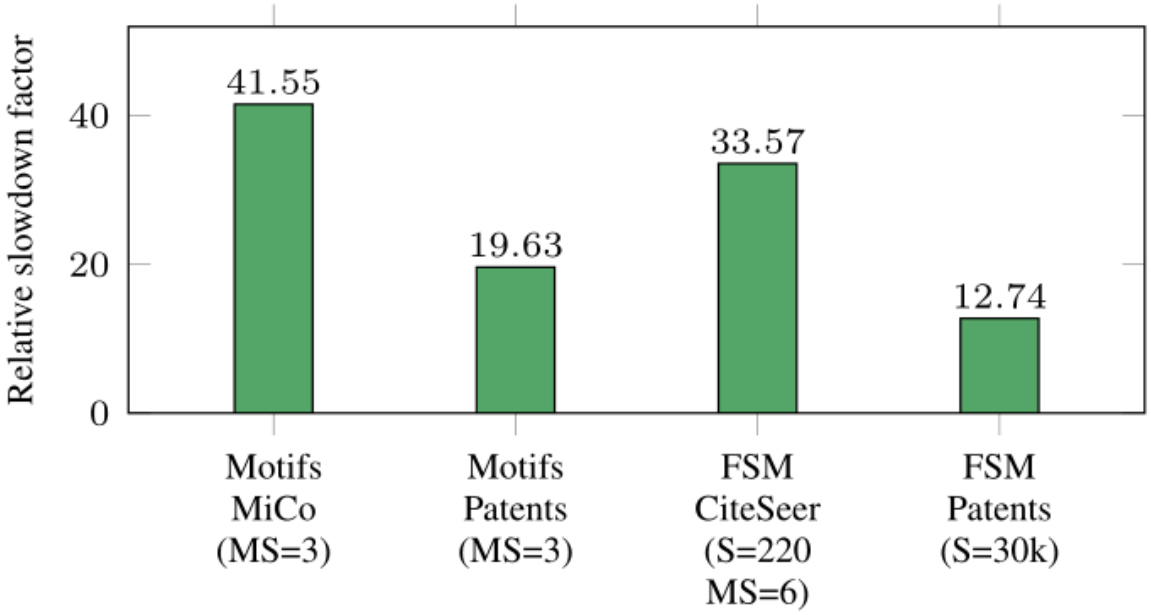


Evaluation - 2-level aggregation

	Motifs MiCo (MS = 4)	Motifs Youtube (MS=4)	FSM CiteSeer (S=220, MS=7)	FSM Patents (S=24k)
Embeddings	10,957,439,024	218,909,854,429	1,680,983,703	1,910,611,704
Quick Patterns	21	21	1433	1800
Canonical Patterns	6	6	97	1348
Reduction Factor	521,782,810x	10,424,278,782x	1,173,052x	1,061,451x

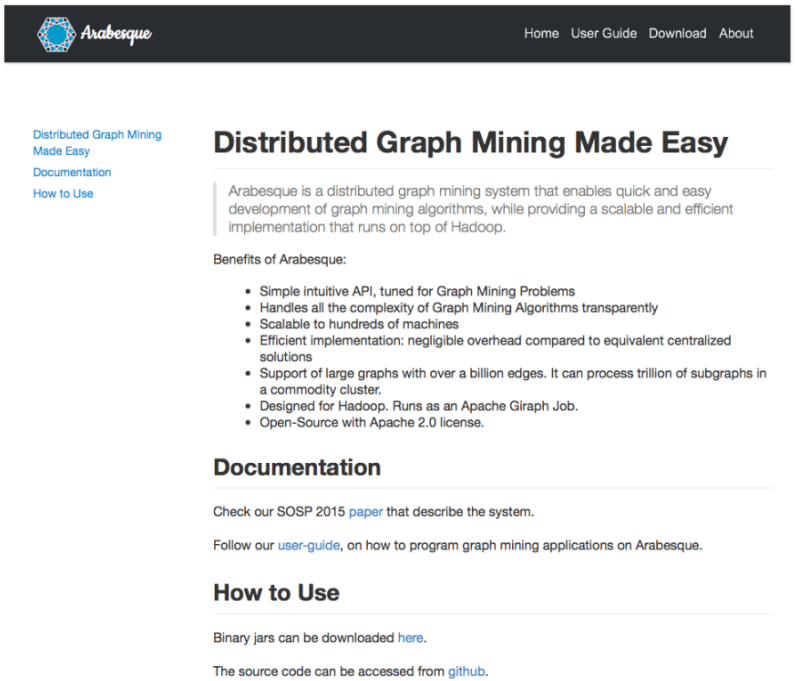


Evaluation - 2-level aggregation



What’s Next?

Download It, Play with It, Hack It



http://arabesque.io

- **Open-source (Apache 2.0)**
- Pre-compiled jar
- User guide

Future Directions

- Full support for graph search + mining
  - TLE simplifies the design of distributed graph search
- Support for real-time graph data
  - Incremental computations
- Multiplatform
  - Spark – ongoing
  - Flink - planned
- Library of graph mining applications

How to Run & Code

Requirements

- Hadoop installation:
  - Runs a map-reduce job (Giraph based)
- To develop:
  - Java 7

Input Graph

- Graphs:
  - labels on vertices
  - labels on edges
  - Multiple edges with labels between two vertices
- Graph should have sequential vertex ids, and it should be ordered

How to Run?

```
./run_arabesque.sh cluster.yaml application.yaml
```

API Example: Clique finding

```
1  boolean filter(Embedding e) {
2      return isClique(e);
3  }
4  void process(Embedding e) {
5      output(e);
6  }
7  boolean shouldExpand(Embedding embedding) {
8      return embedding.getNumVertices() < maxsize;
9  }
10 boolean isClique(Embedding e) {
11     return e.getNumEdgesAddedWithExpansion()==e.getNumberOfVertices()-1;
12 }
```

State of the Art
(Mace, centralized)

4,621 LOC

Cluster.yaml

```
num_workers: 10
num_compute_threads: 16
output_active: yes

# Giraph configuration
#giraph.nettyClientThreads: 32
#giraph.nettyServerThreads: 32
#giraph.nettyClientExecutionThreads: 32
#giraph.channelsPerServer: 4
#giraph.useBigDataIOForMessages: true
#giraph.useNettyPooledAllocator: true
#giraph.useNettyDirectMemory: true
#giraph.nettyRequestEncoderBufferSize: 1048576
```

Cliques.yaml

```
computation: io.arabesque.examples.clique.CliqueComputation
input_graph_path: citeseer-single-label.graph
output_path: Cliques_Output

#communication_strategy: embeddings

# Custom parameters
arabesque.clique.maxsize: 4
```