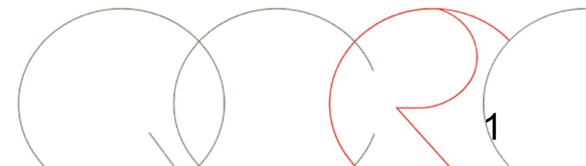
Arabesque.io

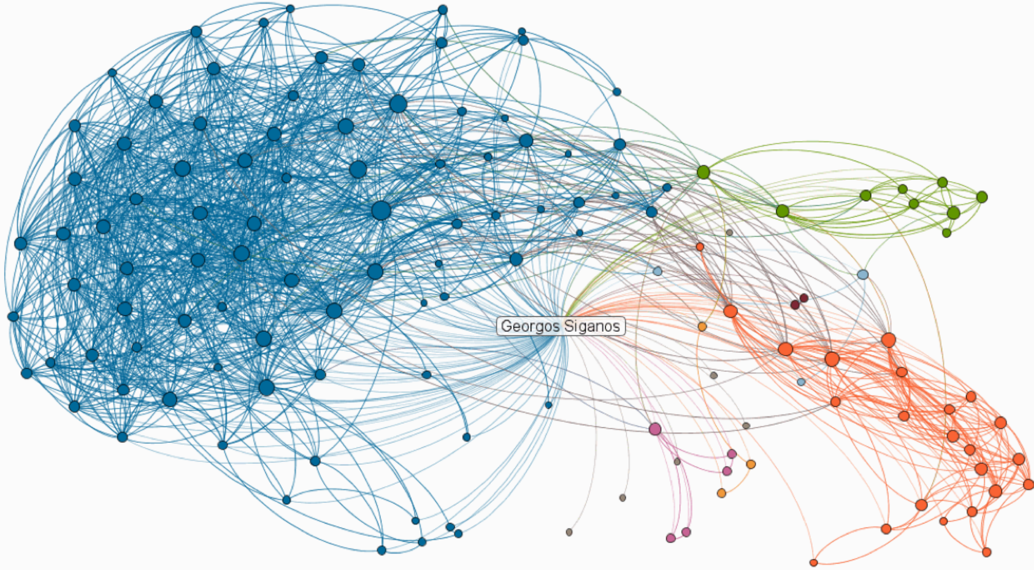
A system for distributed graph mining

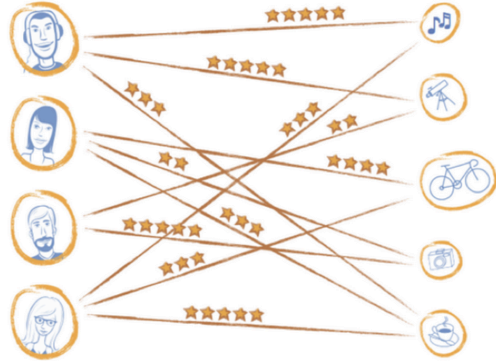
Carlos Teixeira, Alexandre Fonseca, Marco Serafini,
Georgos Siganos, Mohammed Zaki, Ashraf Aboulnaga

جامعة حمد بن خليفة
HAMAD BIN KHALIFA UNIVERSITY



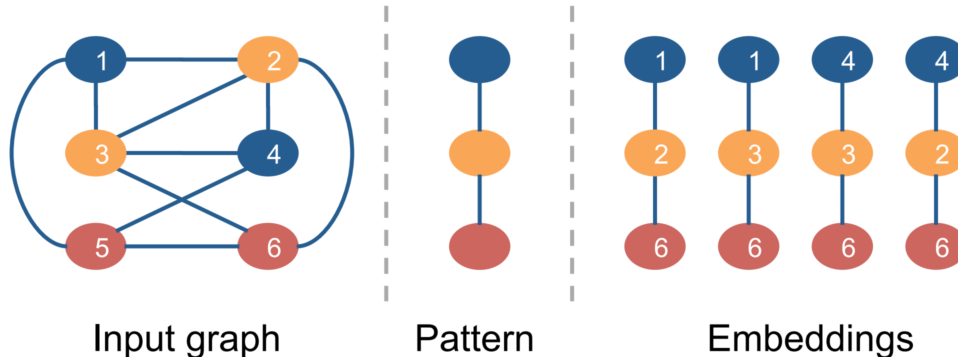
Graphs are ubiquitous



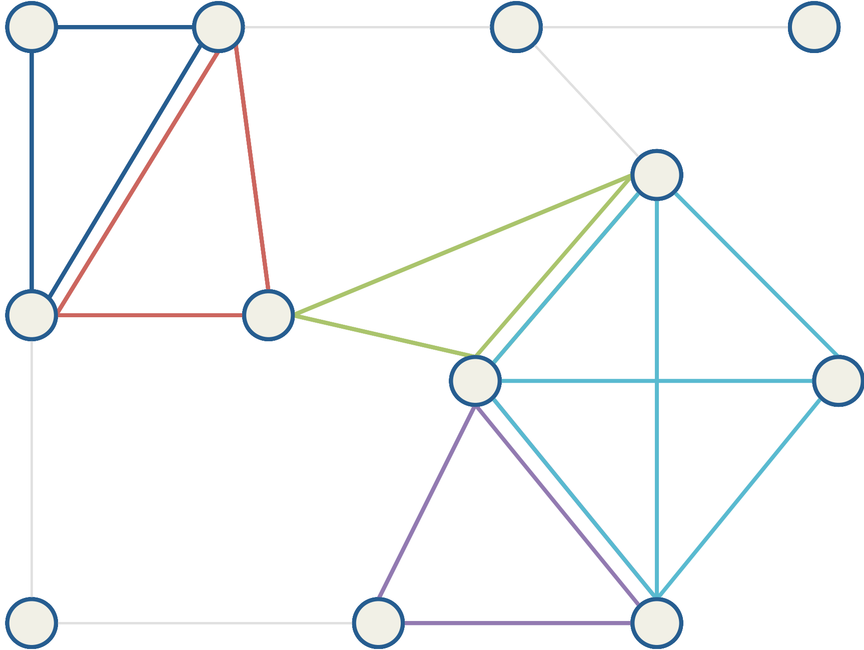


Graph Mining - Concepts

- Label
  - Distinguishable property of a vertex (e.g. color).
- Pattern - “Meta” sub-graph.
  - Captures subgraph structure and labelling
- Embedding - Instance of a pattern.
  - Actual vertices and edges



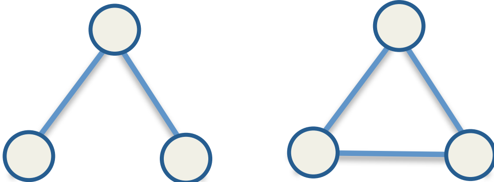
Graph Mining: Cliques



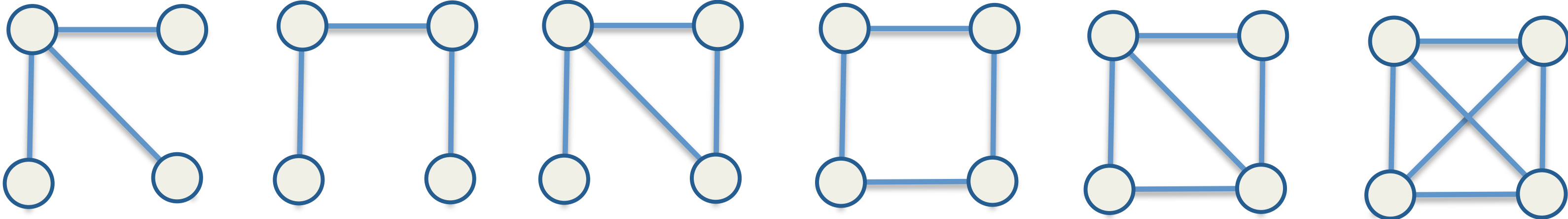
Property: Fully connected subgraphs

Graph Mining: Motifs

Motifs Size = 3



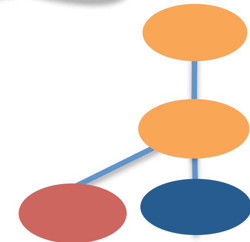
Motifs Size = 4



Graph Mining: FSM

- Frequent Subgraph mining in a single large graph.

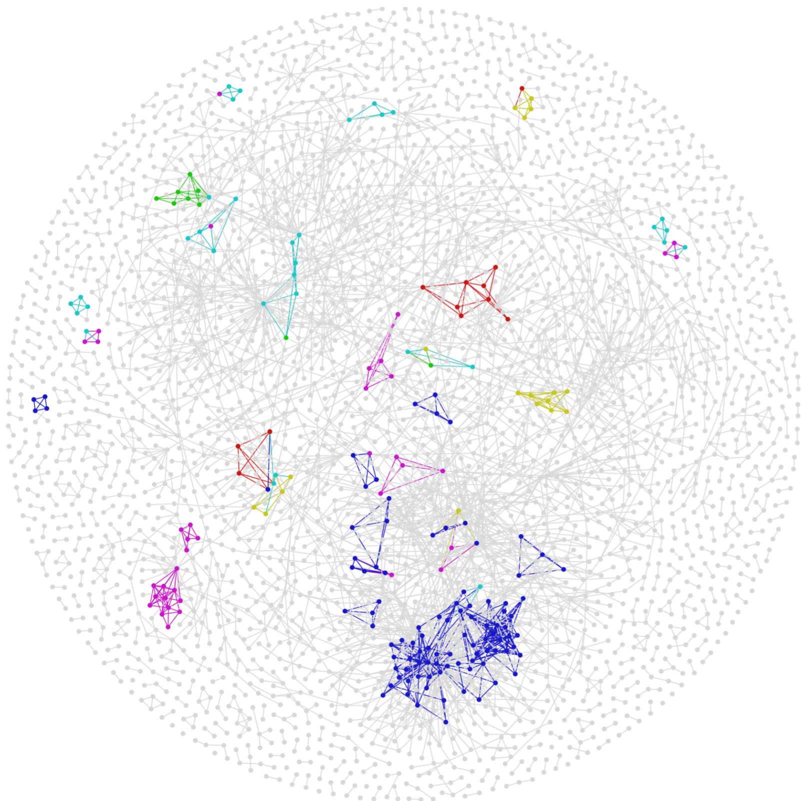
- Find subgraphs that have a minimum embedding count

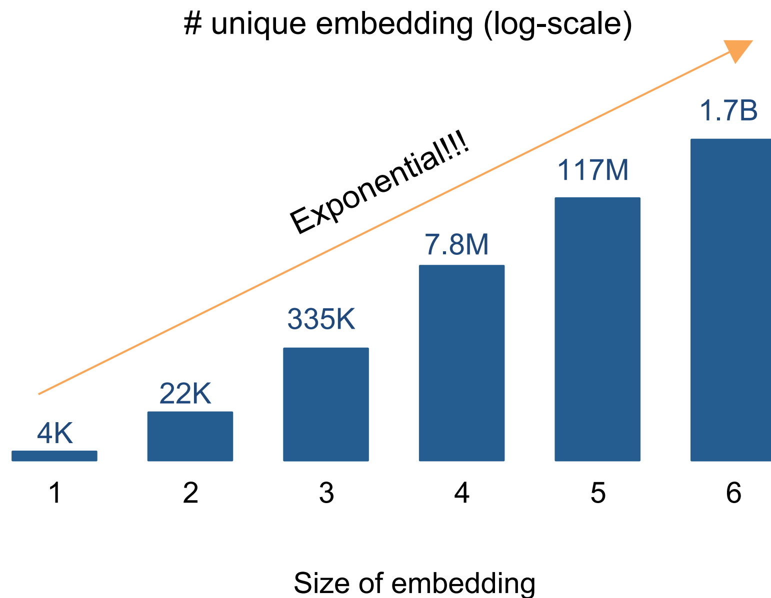


Applications

- Web:
    - Community detection, link spam detection
- Semantic data:
    - Attributed patterns in RDF
- Biology:
    - Characterize protein-protein or gene interaction

Challenges





Exponential number of embeddings

Challenges

- No standard way to solve theses problems.
- No way to distribute the processing easily.
- Way too complicated for programmers (Many …isms)
    - Detect and identify repeated subgraphs – Automorphisms
    - Aggregate to Pattern – Isomorphism

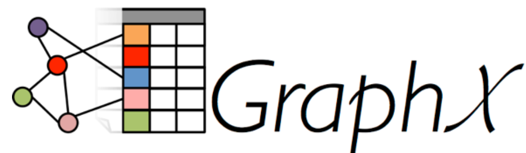
- Above all not all problems are tractable. No cluster grows exponentially.

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

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

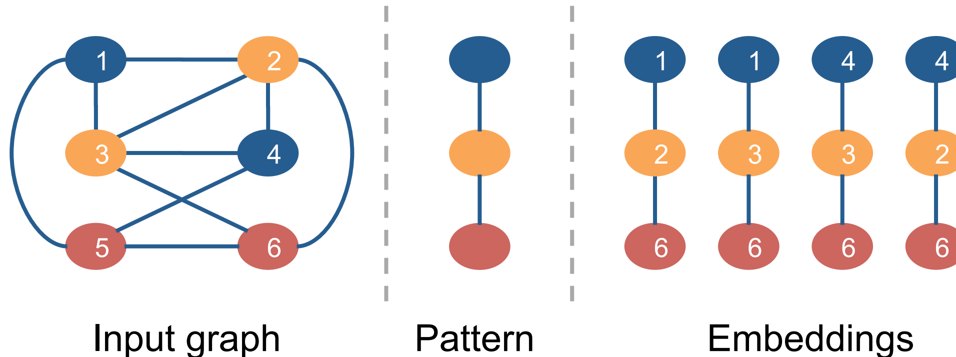
- Contributions:
  - Simple & Generic API
  - High performance
  - Distributed & Scalable by design

Arabesque

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

Graph Mining - Concepts

- Label
  - Distinguishable property of a vertex (e.g. color).
- Pattern - “Meta” sub-graph.
  - Captures subgraph structure and labelling
- Embedding - Instance of a pattern.
  - Actual vertices and edges



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.

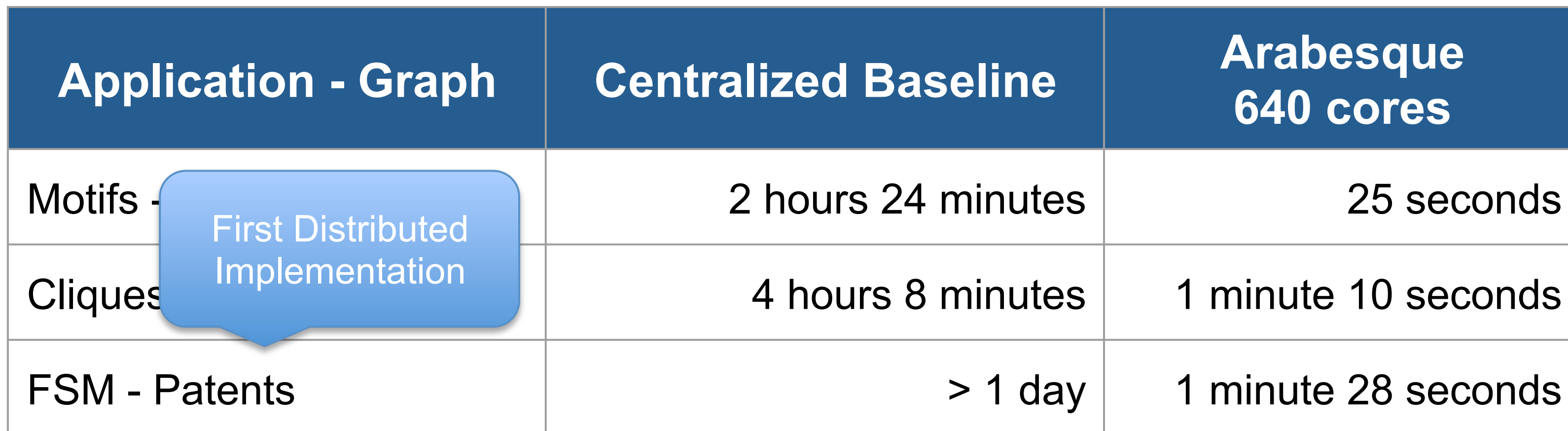
Arabesque: An Efficient System

- 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	~~385s~~ 77s
FSM - CiteSeer (S=300)	4.8s	5s

Arabesque: A Scalable System

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



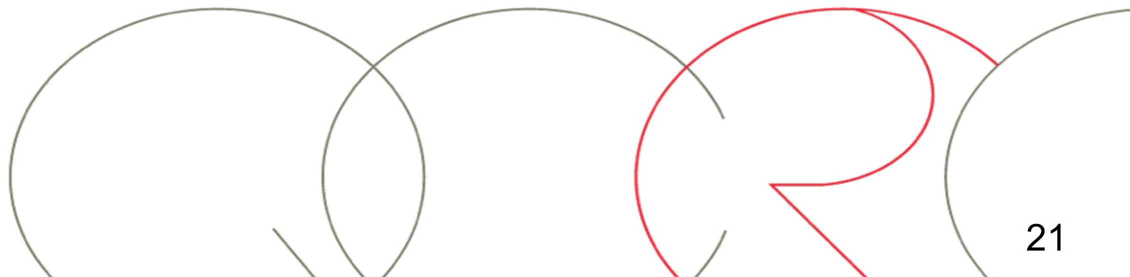
Application - Graph	Centralized Baseline	Arabesque 640 cores
Motifs -	2 hours 24 minutes	25 seconds
Cliques	4 hours 8 minutes	1 minute 10 seconds
FSM - Patents	> 1 day	1 minute 28 seconds

How: Arabesque Optimizations

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

Outline

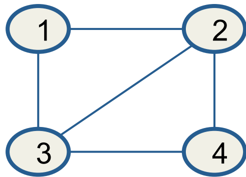
- Graph mining exploration & Arabesque fundamentals
- System Architecture & Optimizations
- Evaluation of System
- How to Run & Code



Graph mining exploration & Arabesque fundamentals

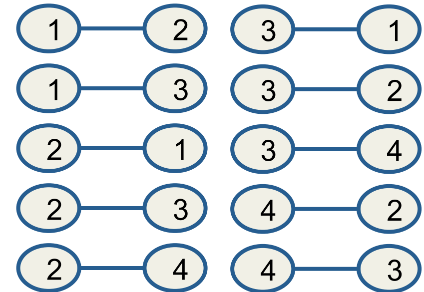
Graph Mining - Exploration

- Iterative expansion
  - Subgraph size n → Subgraph size n + 1
  - Connect to neighbours, one vertex at a time.



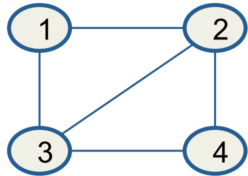
Input graph

Depth 1

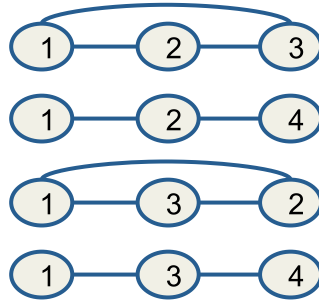


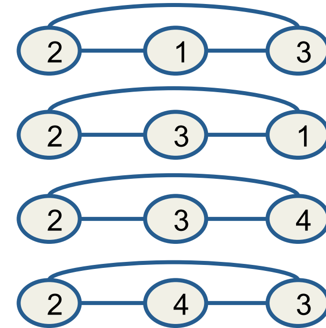
Depth 2

Graph Mining - Exploration

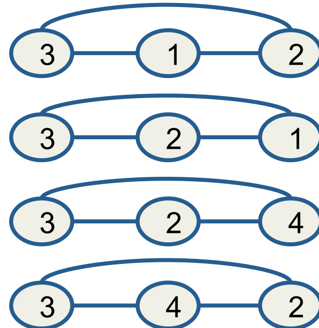


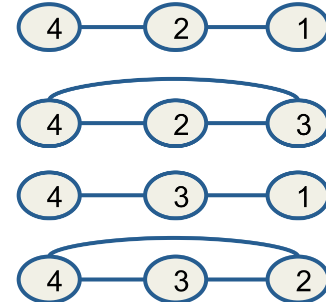
Input graph





Depth 3





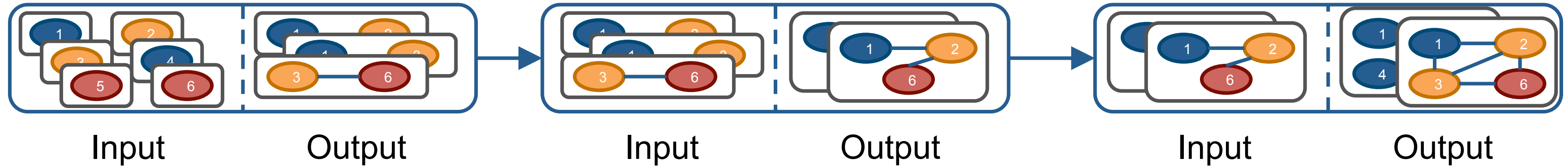
Arabesque: Fundamentals

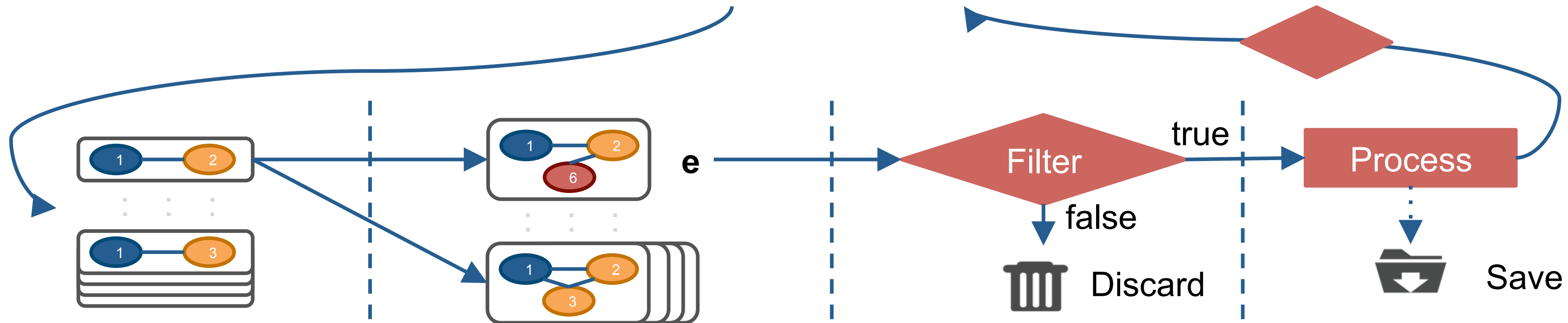
- Embeddings as 1st class citizens:
  - **Think Like an Embedding** model

Arabesque responsibilities		User responsibilities
Graph Exploration	Aggregation (Isomorphism)	Filter
Load Balancing	Automorphism Detection	Process

Model - Think Like an Embedding

Exploration step 1 | Exploration step 2 | Exploration step 3





1. Start from a set of **initial embeddings**

2. **Candidates**: Expand by 1 vertex/ edge

3. **Filter** uninteresting candidates

4. Produce **outputs**

User-defined functions

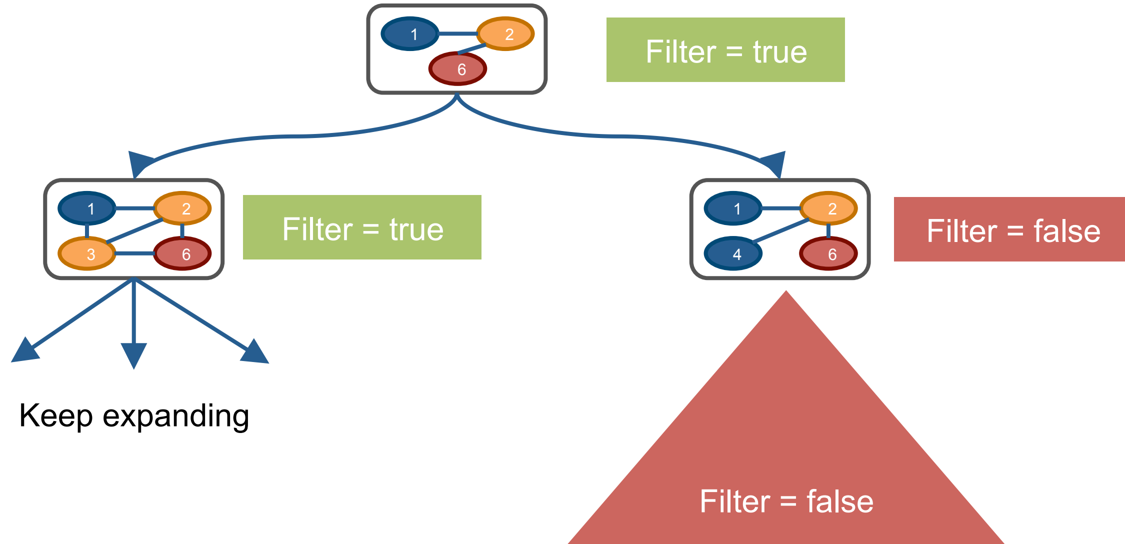
API Example: Clique finding

```
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: <u>Anti-monotonicity</u>



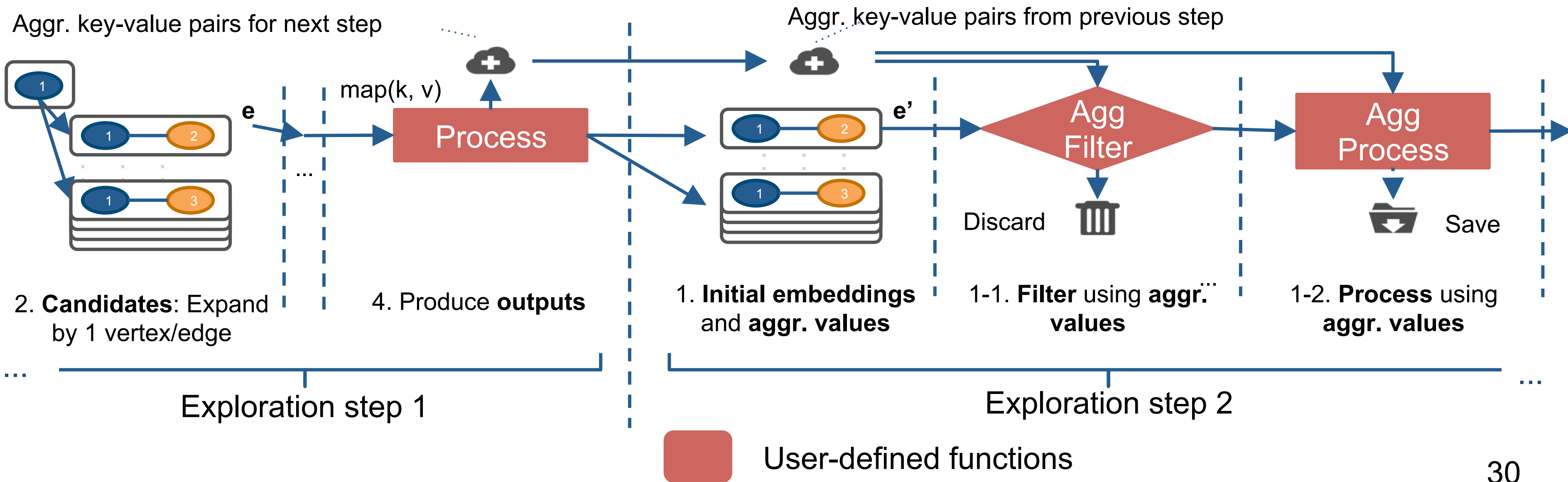
We can **prune** and be sure that we won't ignore desired embeddings

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
  - E.g.: Frequent subgraph mining
    - Frequency requires looking at all candidates



Arabesque API

- Main App-defined functions:

```
boolean filter(E embedding);
void    process(E embedding);
boolean shouldExpand(E newEmbedding);        // Terminate early if max depth defined
boolean aggregationFilter(E Embedding);      // Ignore embedding
boolean aggregationFilter(Pattern pattern);  // Ignore pattern (ex. not frequent)
void    aggregationProcess(E embedding);
void    handleNoExpansions(E embedding);
```

- Performance improvements:

```
void filter(E existingEmbedding, IntCollection extensionPoints); // prune extensions
boolean filter(E existingEmbedding, int newWord);                // Canonicality check
```

- Functions Provided by Arabesque:

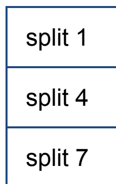
```
void output(String outputString);
void map(String name, K key, V value);
AggregationStorage<K, V> readAggregation(String name);
```

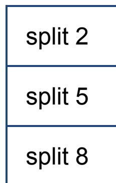
System Architecture & Optimizations

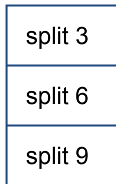
Arabesque Architecture

Previous step

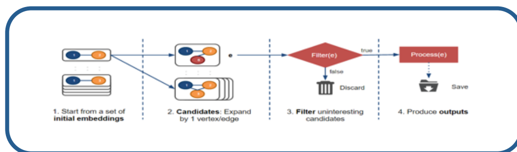
Input
Embeddings size n



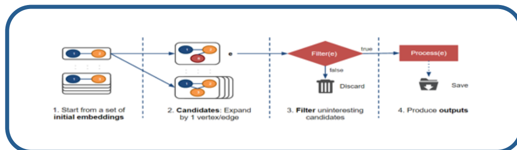




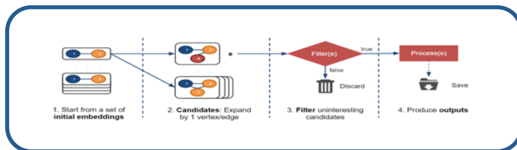
Worker 1



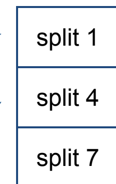
Worker 2

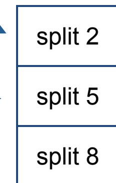


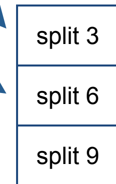
Worker 3



Output
Embeddings size n + 1



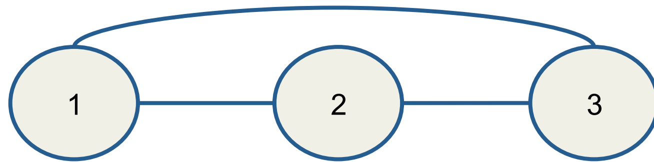




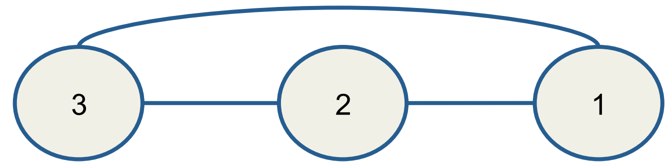
Next step

Avoiding redundant work

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



==

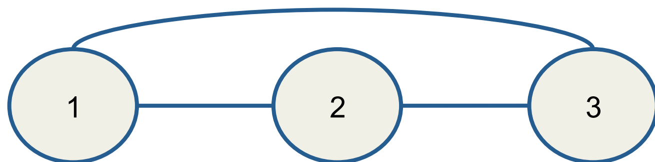


Worker 1

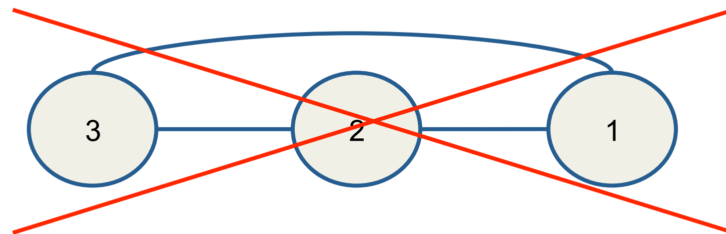
Worker 2

Avoiding redundant work

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



==



Worker 1

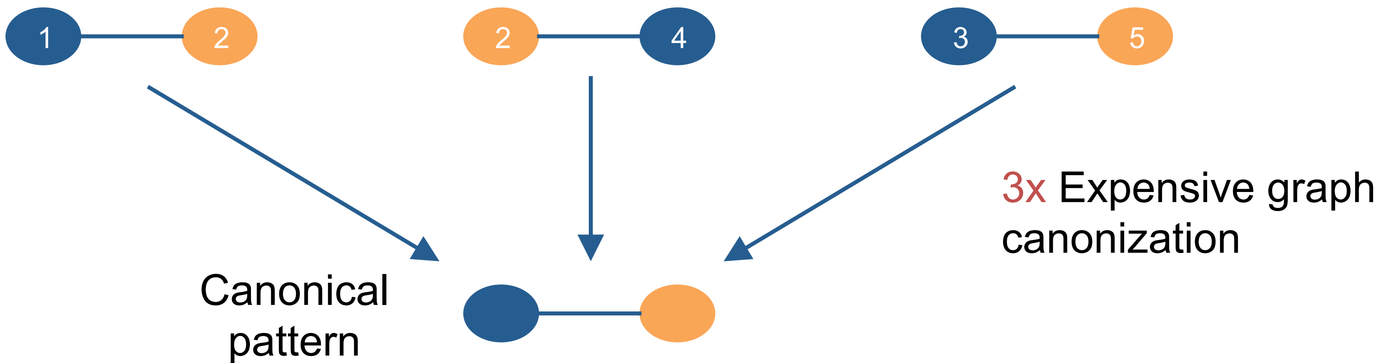
isCanonical(e) → true

Worker 2

isCanonical(e) → false

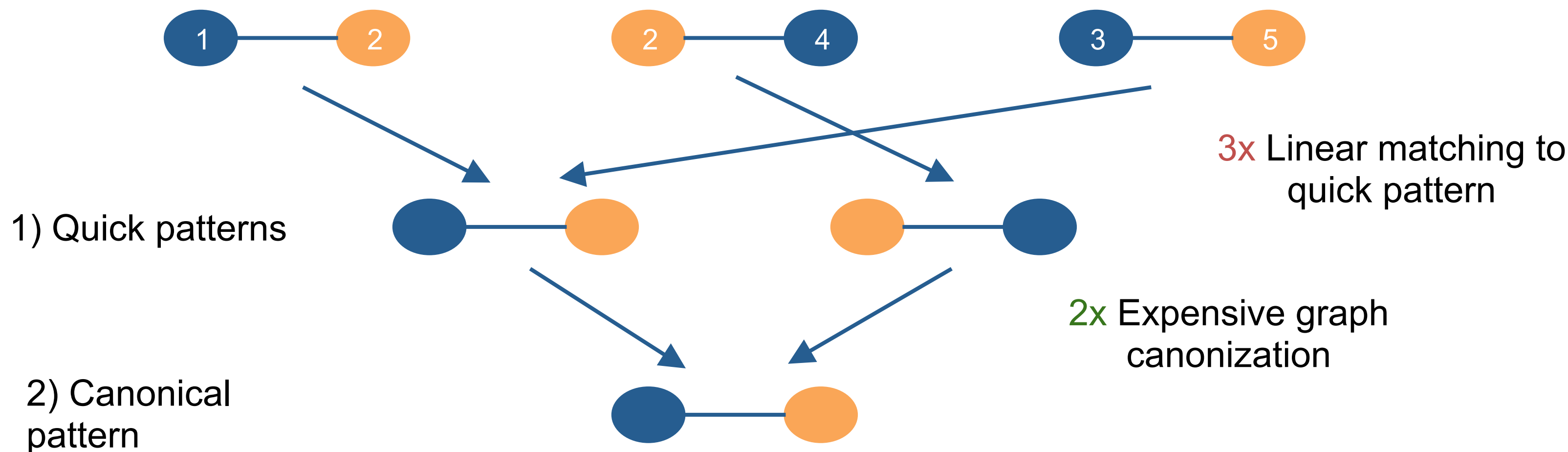
Efficient Pattern Aggregation

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



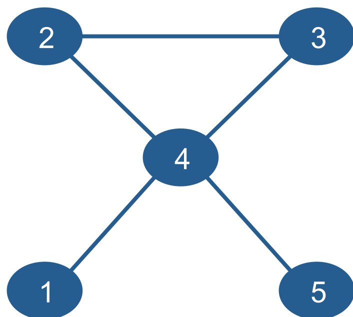
Efficient Pattern Aggregation

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



Handling Exponential growth

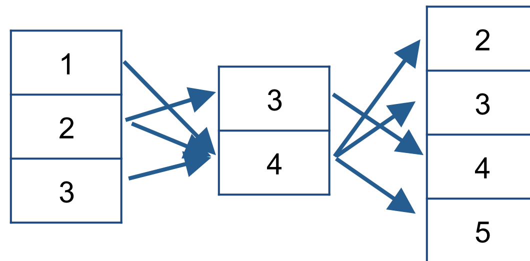
- **Goal:** handle trillions+ different embeddings?
- **Solution:** **Overapproximating DAGs (ODAGs)**
  - Compress into less restrictive superset
  - Deal with spurious embeddings



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

Performance

Evaluation - Setup

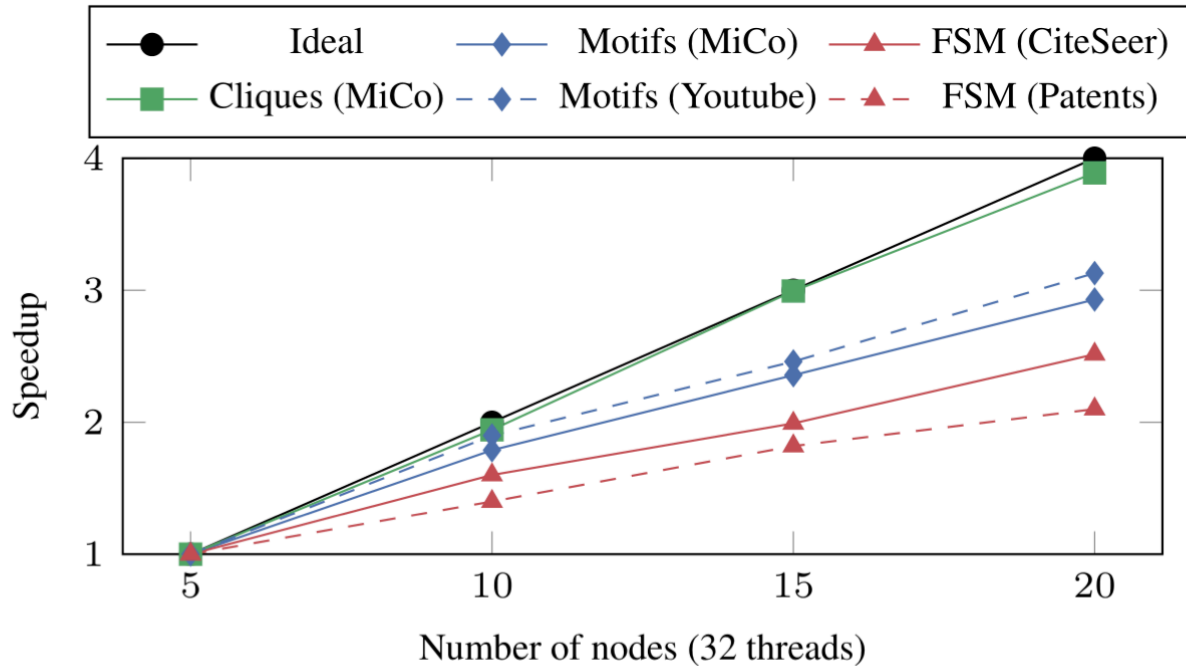
- 20 servers: 32 threads @ 2.67 GHz, 256GB RAM
- 10 Gbps network

- 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

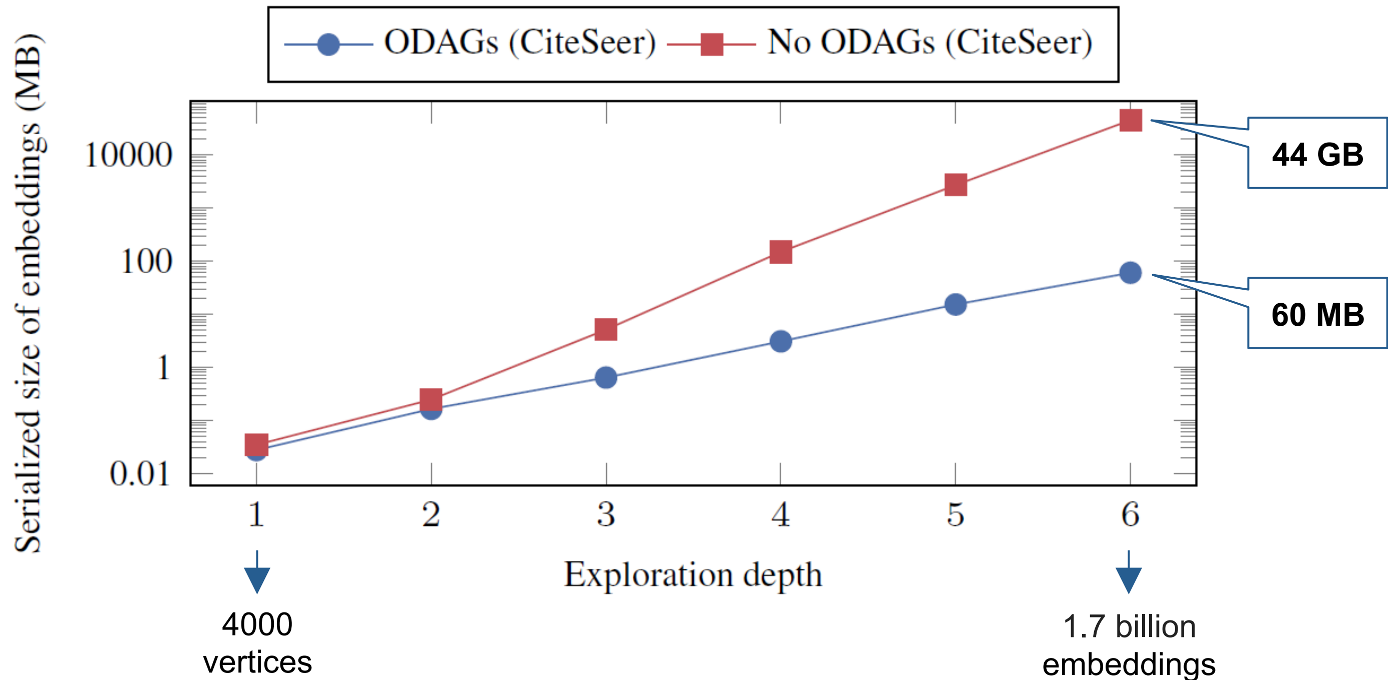
Evaluation - Scalability



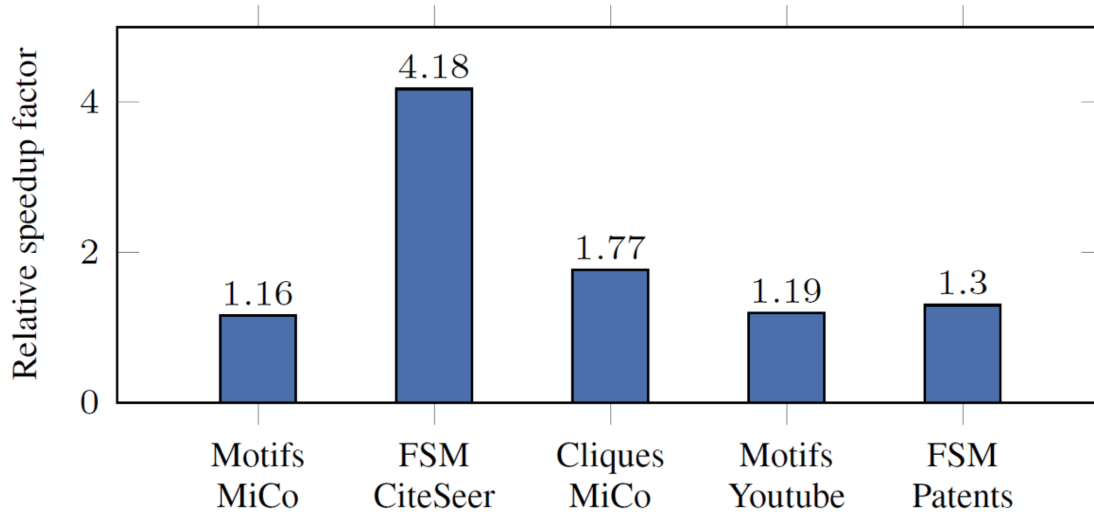
Evaluation - Scalability

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



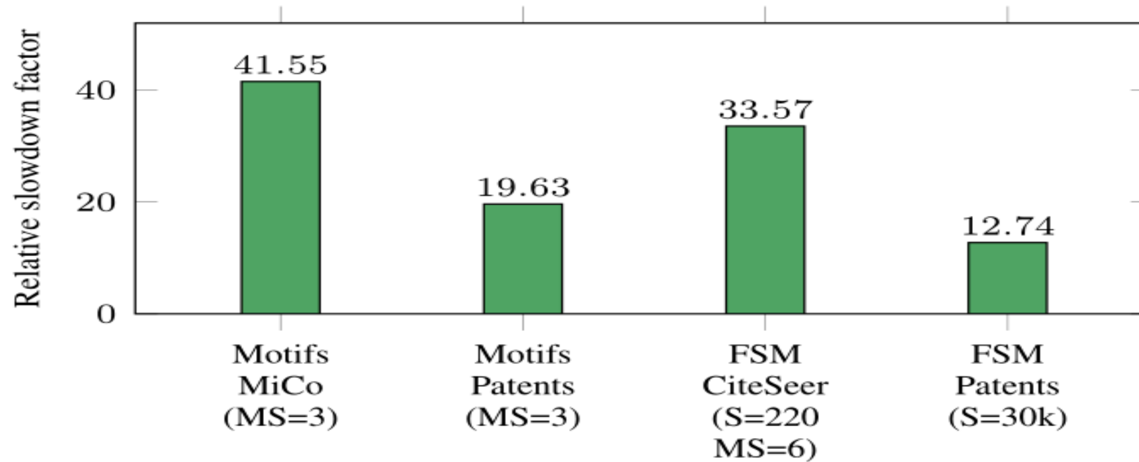
Evaluation - Speedup w ODAGs



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



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
```

Cluster.yaml

```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
```

Fsm.yaml

```yaml
computation: io.arabesque.examples.fsm.FSMComputation
master_computation: io.arabesque.examples.fsm.FSMMasterComputation

input_graph_path: citeseer.graph
output_path: FSM_Output

#communication_strategy: embeddings

# Custom parameters
arabesque.fsm.support: 300
#arabesque.fsm.maxsize: 7
# Split all aggregations in 10 parts for parallel aggregation
# (use only with heavy aggregations)
# arabesque.aggregators.default_splits: 10
```

Cliques.yaml

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

#communication_strategy: embeddings

optimizations:
     - io.arabesque.optimization.CliqueOptimization

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

http://arabesque.io

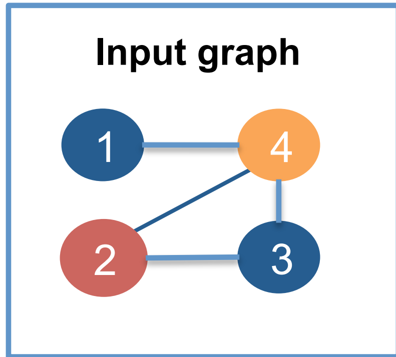
https://github.com/Qatar-Computing-Research-Institute/Arabesque

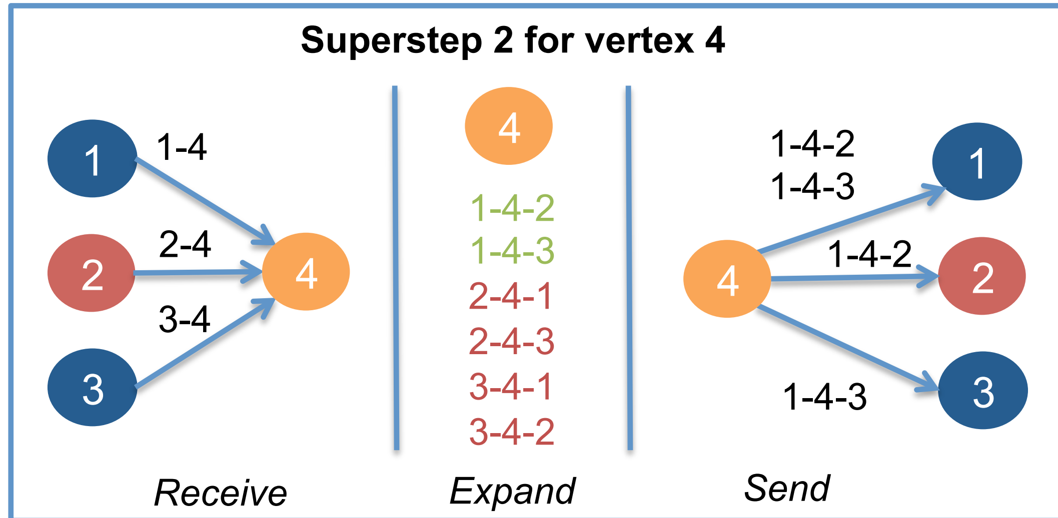
Conclusion

- Graph mining is complex
- Existing approaches not ideal
- Arabesque - facilitate distributed graph mining algorithms
  - General & Simple API
  - Efficient & Scalable
- Just the beginning!!!

Graph Exploration with TLV

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





Evaluation - TLP & TLV

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