Enabling Declarative Graph Analytics over Large, Noisy Information Networks

Amol Deshpande

Department of Computer Science and UMIACS
University of Maryland at College Park

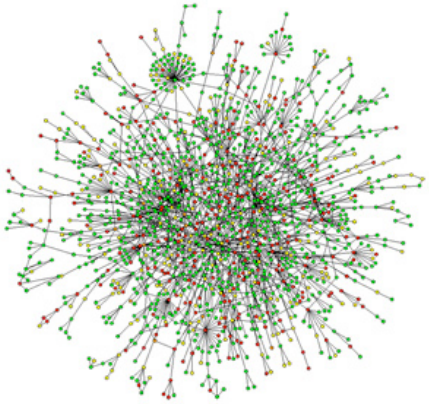
Joint work with: Prof. Lise Getoor, Walaa Moustafa, Udayan Khurana, Jayanta Mondal, Abdul Quamar, Hui Miao

Outline

- Motivation and Background
- Declarative Graph Cleaning
- Historical Graph Data Management
- Continuous Queries over Distributed Graphs
- Conclusions

Motivation

- Increasing interest in querying and reasoning about the *underlying graph structure* in a variety of disciplines



A protein-protein interaction network

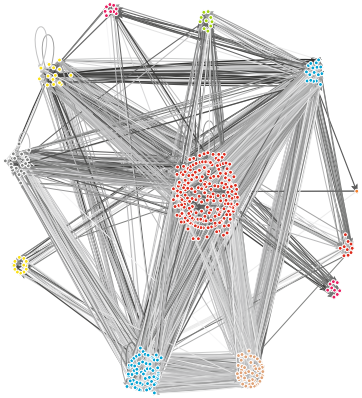
Citation networks

Communication networks

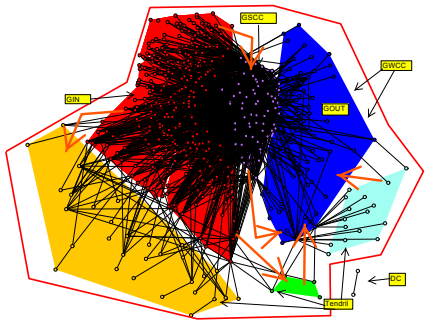
Disease transmission networks

facebook twitter LinkedIn

Social networks



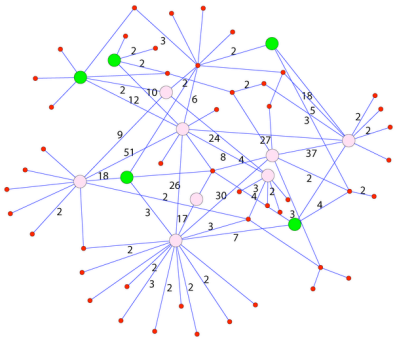
Financial transaction networks



Federal funds networks

Knowledge Graph

World Wide Web



Stock Trading Networks

Motivation

- Underlying data hasn't necessarily changed that much
  - … aside from larger data volumes and easier availability
  - … but increasing realization of the importance of reasoning about the graph structure to extract actionable insights
- Intense amount of work already on:
  - … understanding properties of information networks
  - … community detection, models of evolution, visualizations
  - … executing different types of graph structure-focused queries
  - … cleaning noisy observational data
  - … and so on
- Lack of established data management tools
  - Most of the work done outside of general-purpose data management systems

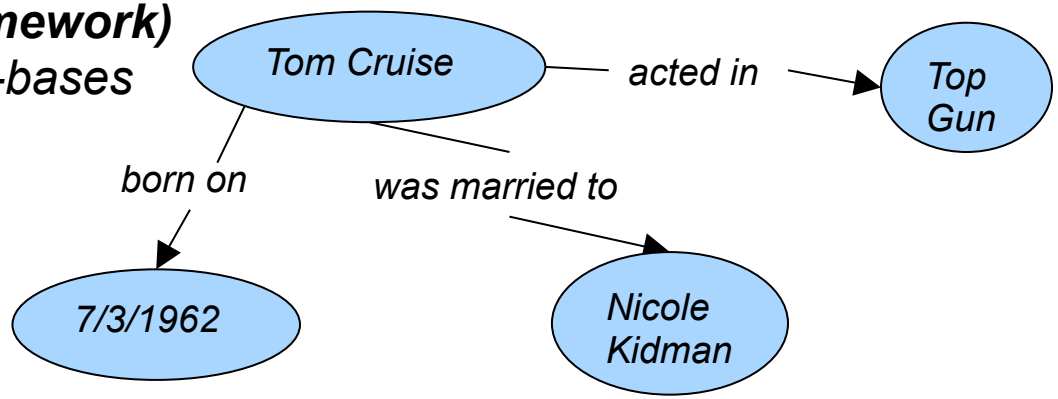
Background: Popular Graph Data Models

RDF (Resource Description Framework)

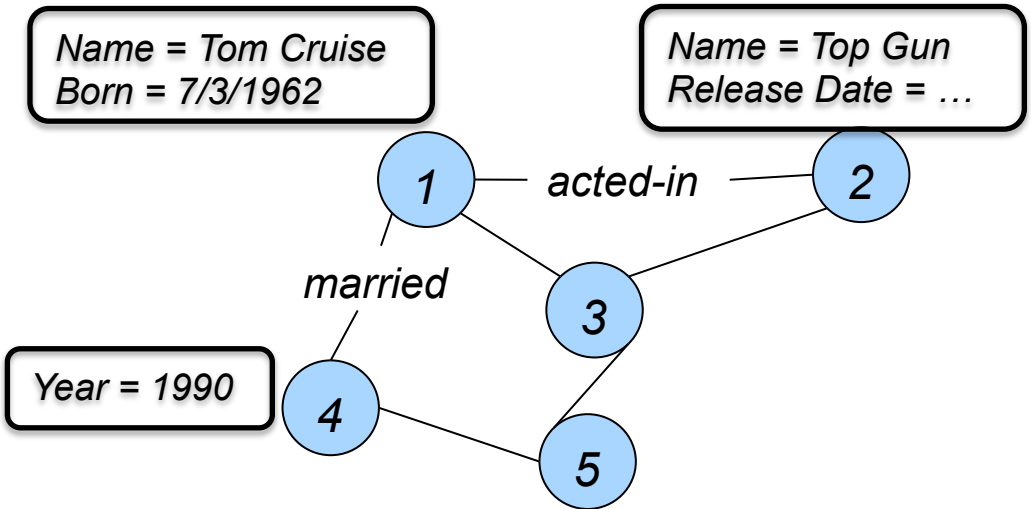
Commonly used for knowledge-bases

Each edge captures:

<subject, predicate, object>

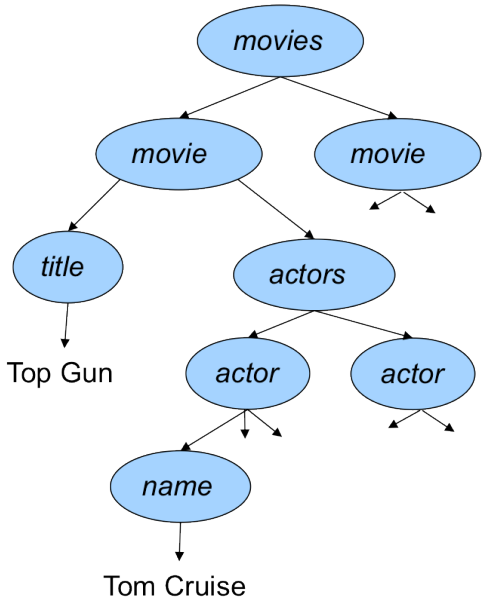


Property graph model:
commonly used by open-source software



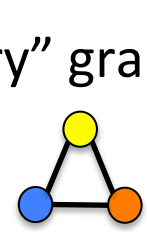
***XML**: Semi-structured data model*

In essence: a directed, labeled "tree"

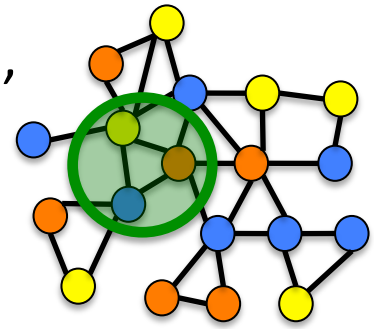


Graph Queries vs Analysis Tasks

- Queries permit focused exploration of the data
  - Result typically a small portion of the graph (often just a node)
  - Examples:
    - **Subgraph pattern matching**: Given a "query" graph, find where it occurs in a given "data" graph



Query Graph

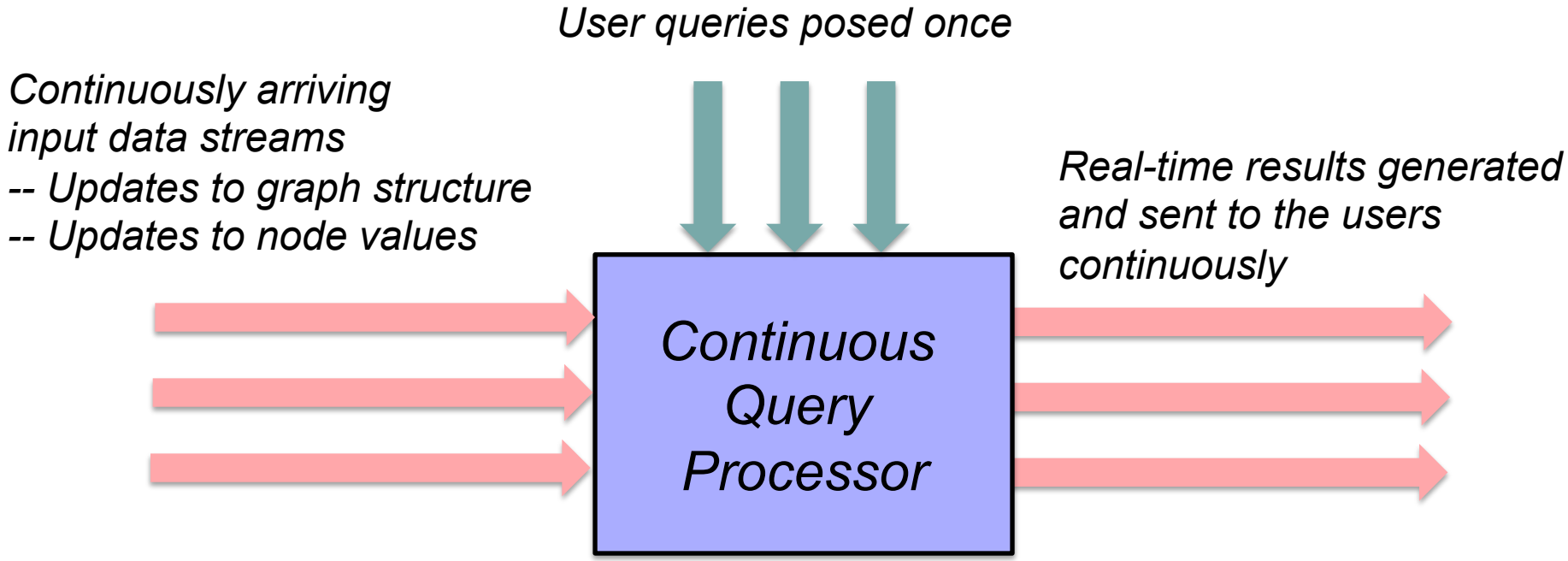


Data Graph

    - **Reachability; Shortest path;**
    - **Keyword search:** Find smallest subgraph that contains all the given keywords
  - **Historical or Temporal queries** over a historical trace of the network over a period of time
    - *"Find most important nodes in a communication network in 2002?"*

Graph Queries vs Analysis Tasks

- Continuous queries
  - *Tell me when a topic is suddenly "trending" in my friend circle*
  - *Alert me if the communication activity around a node changes drastically (anomaly detection)*
  - *Monitor constraints on the data being generated by the nodes (constraint monitoring)*



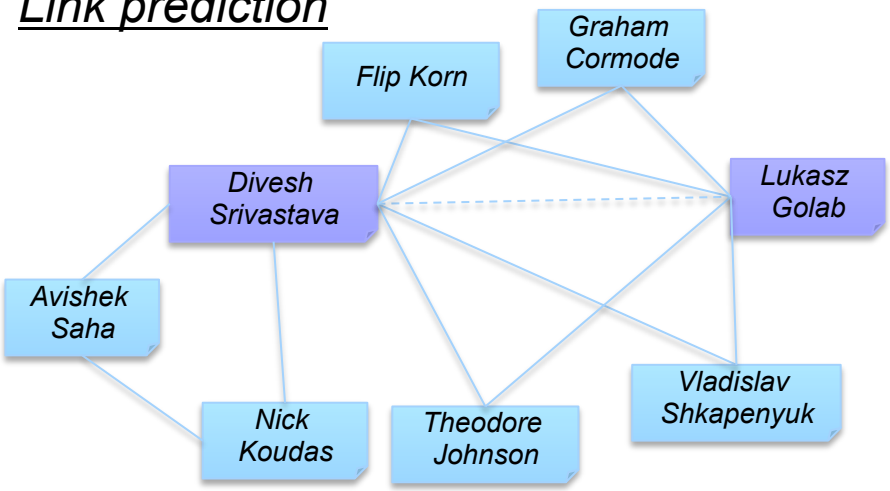
Graph Queries vs Analysis Tasks

- Analysis tasks typically require processing the entire graph
  - **Centrality analysis**: Find the most central nodes in a network
    - Many different notions of centrality…
  - **Community detection**: Partition the vertices into (potentially overlapping) groups with dense interaction patterns
  - **Network evolution:** Build models for network formation and evolution over time
  - **Network measurements:** Measuring statistical properties of the graph or local neighborhoods in the graphs
  - **Inferring historical traces:** Complete historical data unlikely to be available – how to fill in the gaps?
  - **Graph cleaning/inference:** Removing noise and uncertainty in the observed network data

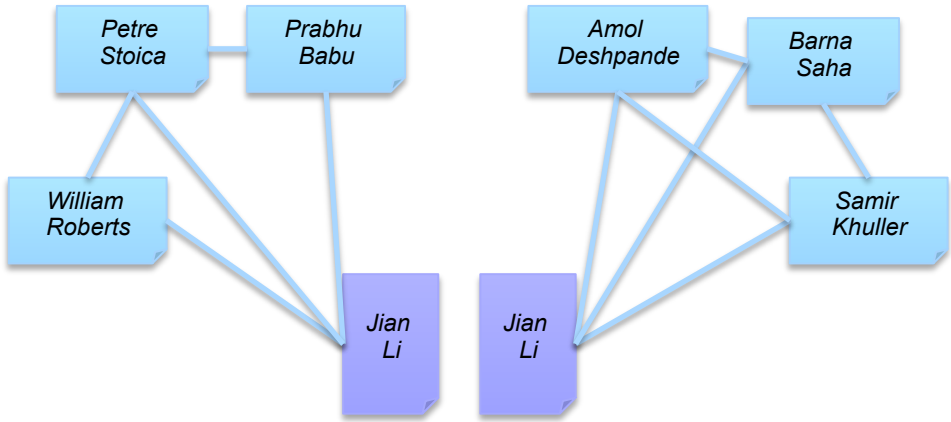
Graph Queries vs Analysis Tasks

- Analysis tasks:
  - **Graph cleaning/inference:** Removing noise and uncertainty in the observed data through –
    - Attribute Prediction: *predict values of missing attributes*
    - Link Prediction: *infer missing links*
    - Entity Resolution: *decide if two nodes refer to the same entity*
  - Inference techniques typically utilize the graph structure

Link prediction



Entity resolution



Data Management: State of the Art

- Most data probably in flat files or *relational databases*
  - Some types of queries can be converted into SQL queries
    - E.g., SPARQL queries over RDF data
  - Otherwise most of the querying and analysis functionality implemented on top
    - Much research on building *specialized indexes* for specific types of queries (e.g., pattern matching, keyword search, reachability, …)
- Emergence of specialized graph databases in recent years
  - Neo4j, InfiniteGraph, DEX, AllegroGraph, HyperGraphDB, …
  - Key disadvantages:
    - Fairly rudimentary declarative interfaces -- most applications need to be written using programmatic interfaces
    - Or using provided toolkits/libraries

Data Management: State of the Art

- Several batch analysis frameworks proposed for analyzing graph data in recent years
  - Analogous to Map-Reduce/Hadoop
    - Map-Reduce not suitable for most graph analysis tasks
    - Work in recent years on designing Map-Reduce programs for specific tasks
  - Pregel, Giraph, GraphLab, GRACE
    - *Vertex-centric:* Programs written from the point of view of a vertex
    - Most based on message passing between nodes
  - Vertex-centric frameworks somewhat limited and inefficient
    - Unclear how to do many complex graph analysis tasks
    - Not widely used yet

Key Data Management Challenges

- Lack of declarative query languages and expressive programming frameworks for processing graph-structured data
- Inherent noise and uncertainty in the raw observation data
  - → Support for *graph cleaning* must be integrated into the system
  - → Need to reason about *uncertainty* during query execution
- Very large volumes of heterogeneous data over time
  - → Distributed/parallel storage and query processing needed
  - → Graph partitioning notoriously hard to do effectively
  - → Historical traces need to be stored in a compressed fashion
- Highly dynamic and rapidly changing data as well as workloads
  - → Need aggressive pre-computation to enable low-latency query execution

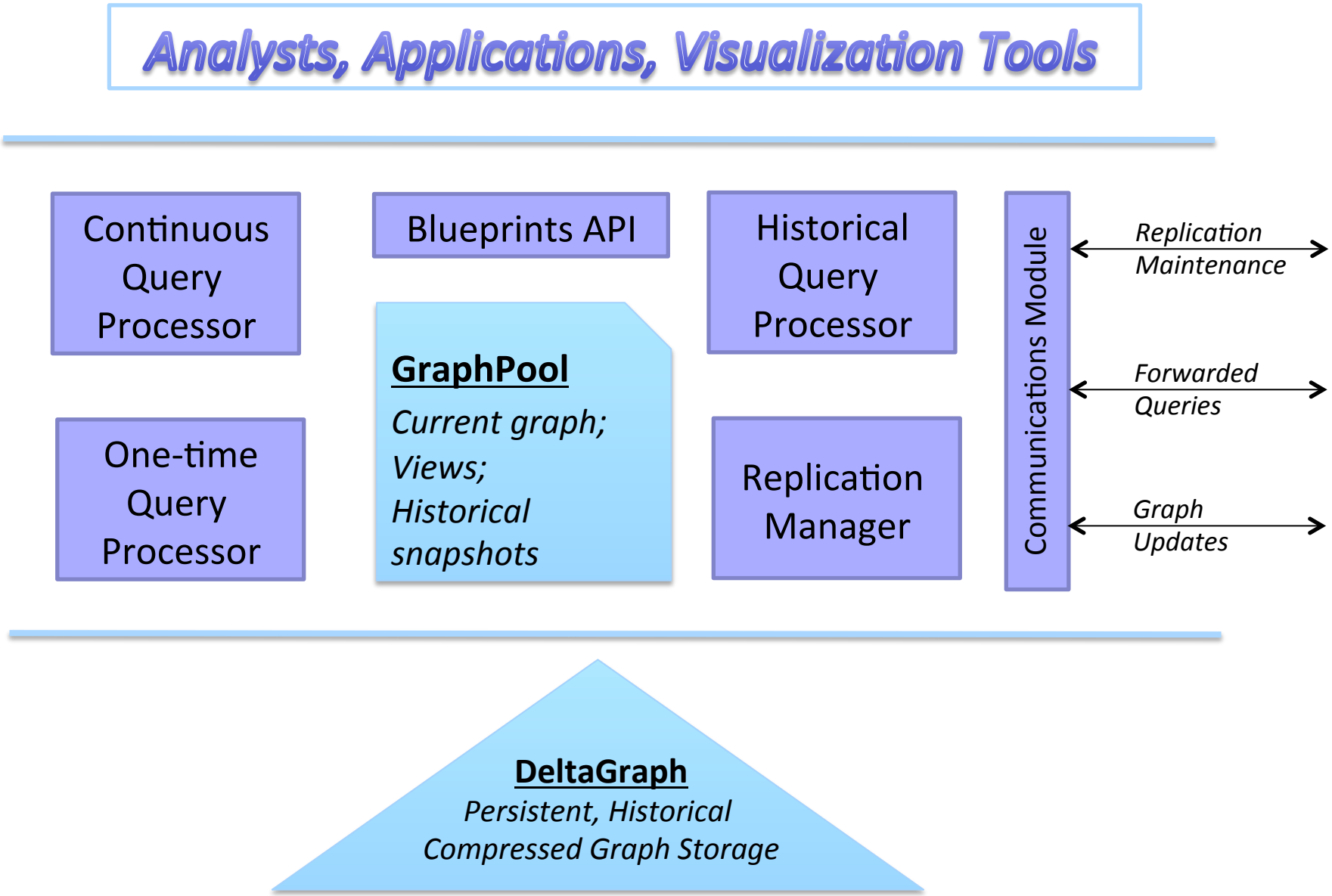
What we are doing

- Address the data management challenges in enabling a variety of queries and analytics

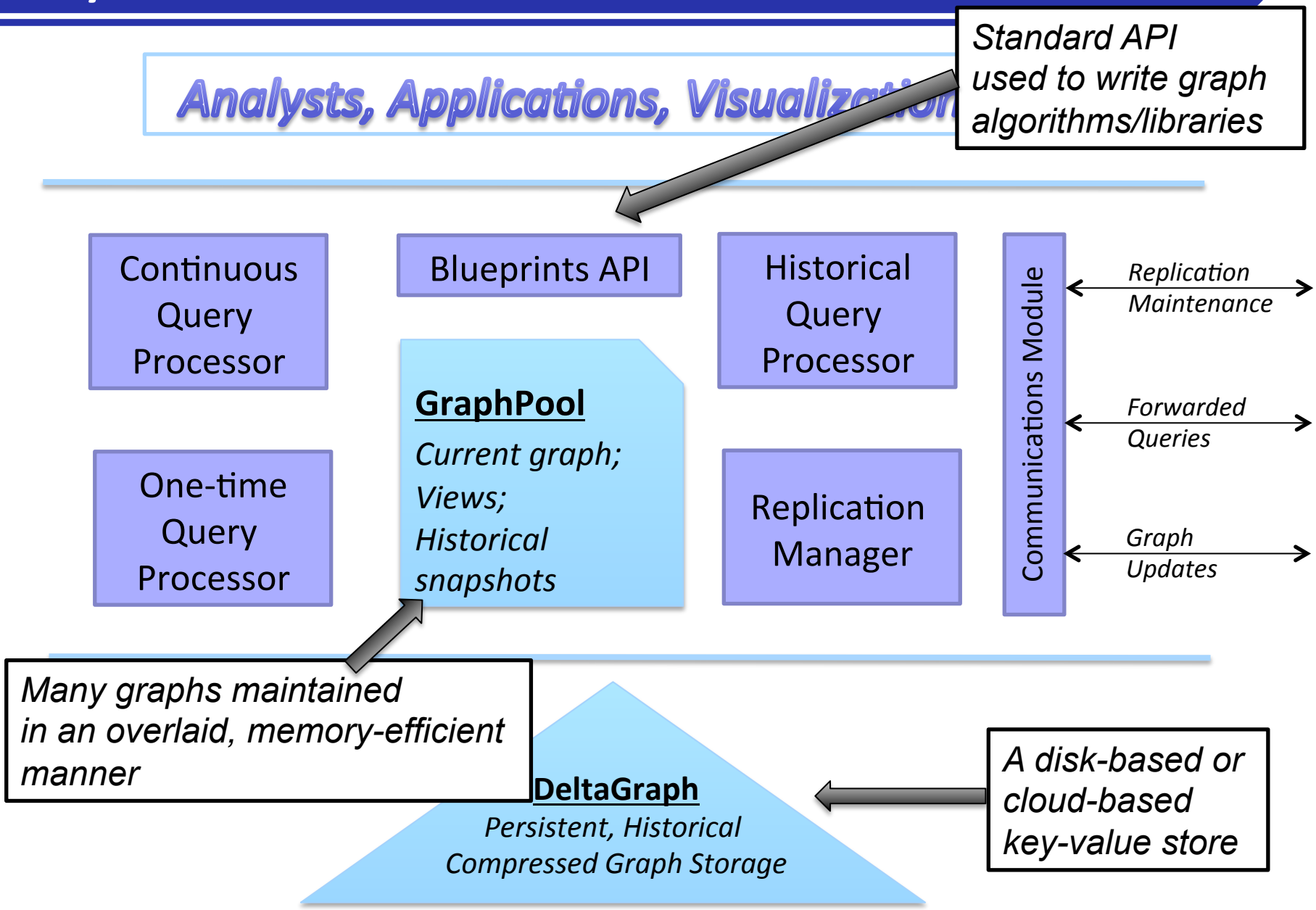
- Aim to support three declarative user-level abstractions for specifying queries or tasks
  - A declarative Datalog-based query language for specifying queries (including historical and continuous)
  - A high-level Datalog-based framework for graph cleaning tasks
  - An expressive programming framework for domain-specific queries or analysis tasks
    - Analogous to MapReduce

- Handle very large volumes of data (including historical traces) through developing distributed and cloud computing techniques

System Architecture



System Architecture



What we are doing

- Work so far:
  - NScale: An end-to-end distributed programming framework for writing graph analytics tasks
  - Declarative graph cleaning [GDM'11, SIGMOD Demo'13]
  - Real-time continuous query processing
    - Aggressive replication to manage very large dynamic graphs efficiently in the cloud, and to execute continuous queries over them [SIGMOD'12]
    - New techniques for sharing [under submission]
  - Historical graph management
    - Efficient single-point or multi-point snapshot retrieval over very large historical graph traces [ICDE'13, , SIGMOD Demo'13]
  - Ego-centric pattern census [ICDE'12]
  - Subgraph pattern matching over uncertain graphs [under submission]

Outline

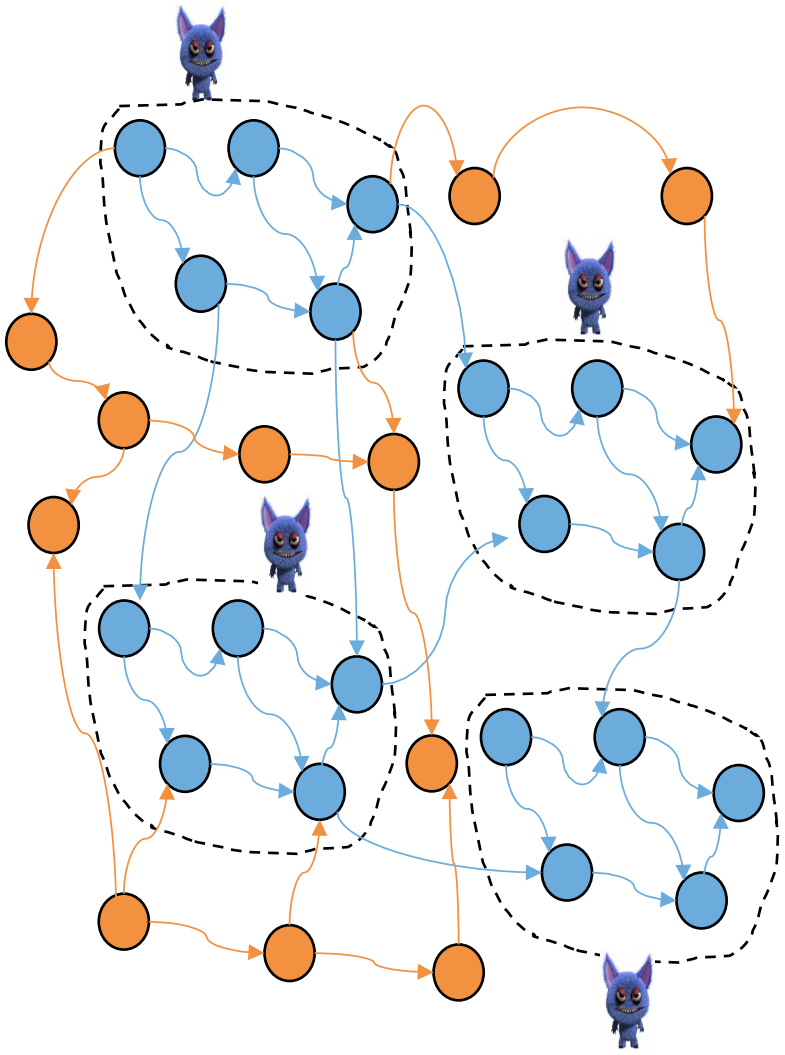
- Overview
- NScale Distributed Programming Framework
- Declarative Graph Cleaning
- Historical Graph Data Management
- Continuous Queries over Distributed Graphs
- Conclusions

Graph Programming Frameworks

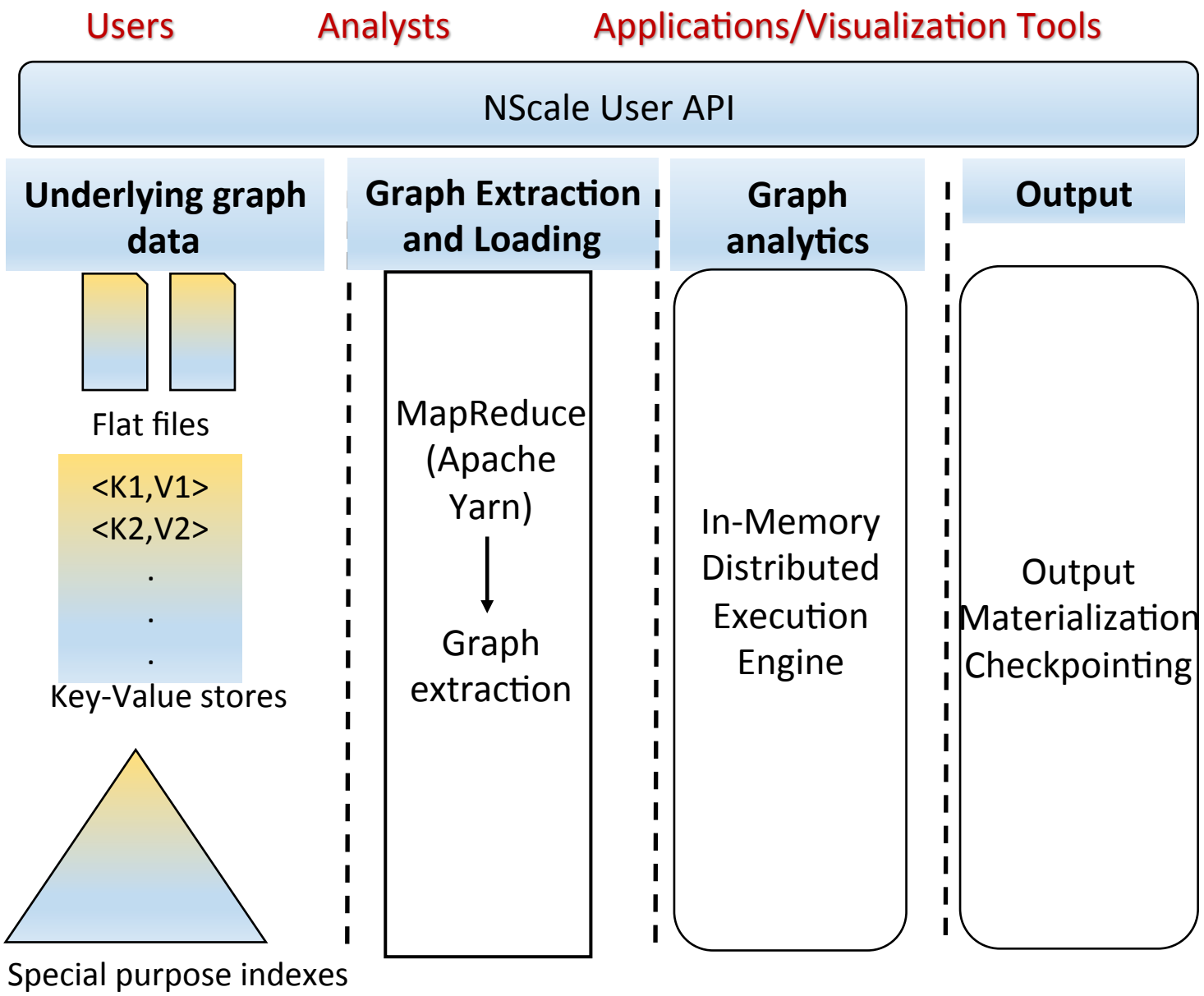
- MapReduce-based (e.g., Gbase, Pegasus, Hadapt)
  - Use MR as the underlying distributed processing framework
  - Disadvantages:
    - Not intuitive to program graph analysis tasks using MR
    - Each "traversal" effectively requires a new MapReduce phase: Inefficient
- Vertex-centric iterative programming frameworks
  - Synchronous (Pregel, Giraph), Asynchronous (GraphLab, GRACE)..
  - No inherent support for applications that require analytics on the neighborhoods of a subset of nodes
  - Not sufficient or natural for many query analysis tasks (Ego network analysis)
  - May be inefficient for analytics that require traversing beyond 1-hop neighbors

NScale Programming Framework

- An end-to-end distributed graph programming framework
- Users/application programs specify:
  - Neighborhoods or subgraphs of interest
  - A kernel computation to operate upon those subgraphs
- Framework:
  - Extracts the relevant subgraphs from underlying data and loads in memory
  - Execution engine: Executes user computation on materialized subgraphs
  - Communication: Shared state/ message passing



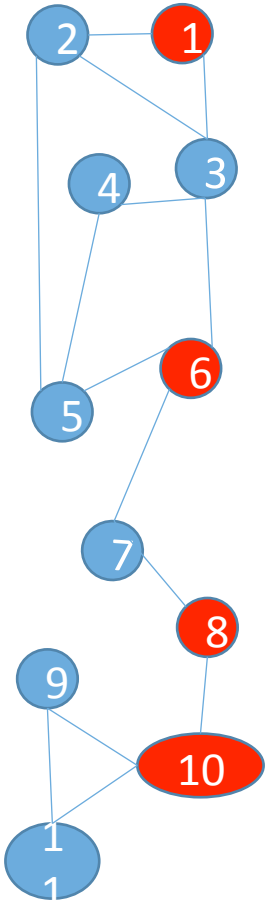
NScale Programming Framework



Example: Local Clustering Coefficient

Nscale User API (Datalog, BluePrints): Query: Compute LCC for nodes where node.color=red

Underlying graph data on HDFS

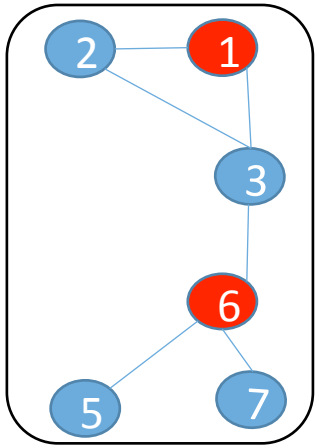


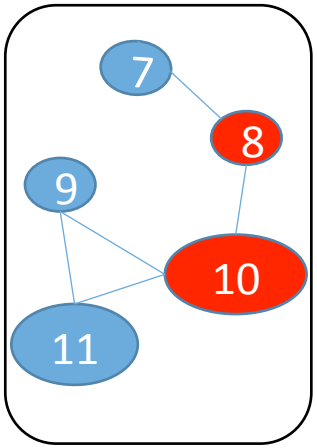
Graph Extraction and Loading

MapReduce (Apache Yarn)

Graph extraction

Subgraphs in Distributed Memory





Graph analytics

Distributed Execution Engine

Distributed Execution Engine

Output

Output Materialization Checkpointing

NScale: Summary

- User writes programs at the abstraction of a graph
  - More intuitive for graph analytics
  - Captures mechanics of common graph analysis/cleaning tasks
  - Complex analytics:
    - Union or intersection of neighborhoods (Link prediction, Entity resolution)
    - Induced subgraph of a hashtag (Influence analysis on hashtag ego networks)
- Scalability: Only relevant portions of the graph data loaded into memory
  - User can specify subgraphs of interest, and select nodes or edges based on properties
    - E.g. Edges with recent communication
- Generalization: Flexibility in subgraph definition
  - Handle vertex-centric programs
    - Subgraph: vertex and associated edges
  - Global programs
    - Subgraph is the entire graph

Outline

- Overview
- NScale Distributed Programming Framework
- Declarative Graph Cleaning
- Historical Graph Data Management
- Continuous Queries over Distributed Graphs
- Conclusions

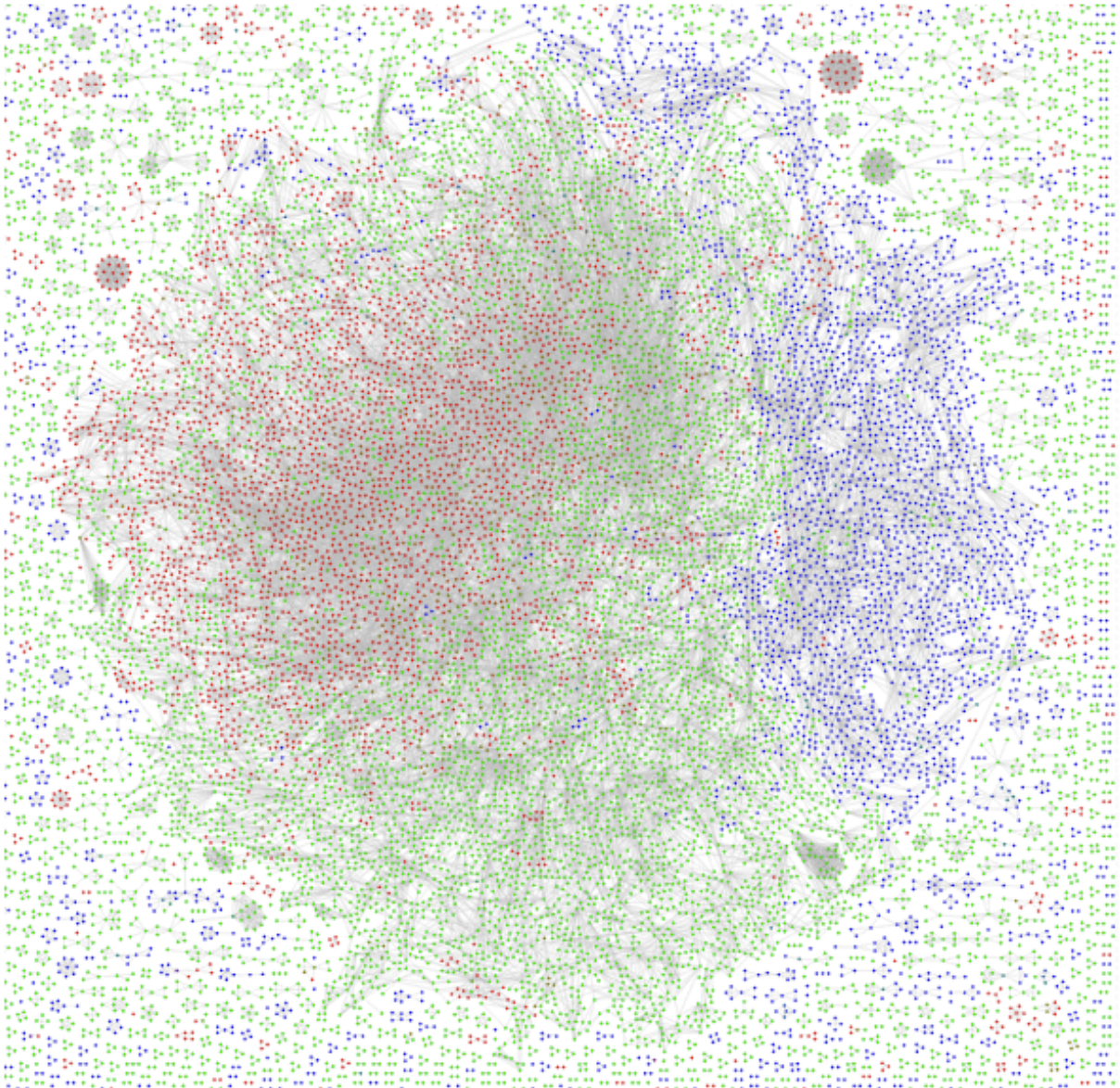
Motivation

- The *observed, automatically-extracted information networks* are often noisy and incomplete
- Need to extract the underlying *true information network* through:
  - Attribute Prediction: *to predict values of missing attributes*
  - Link Prediction: *to infer missing links*
  - Entity Resolution: *to decide if two references refer to the same entity*
- Typically iterative and interleaved application of the techniques
  - Use results of one to improve the accuracy of other operations
- Numerous techniques developed for the tasks in isolation
  - No support from data management systems
  - Hard to easily construct and compare new techniques, especially for joint inteference

Declarative Noisy Network A

localhost/declarative_network_analysis/demo.html

Declarative Noisy Network Analysis



Dataset

DBLP Dataset

Datalog Program

```
DOMAIN Bin(#X,#Y) :- Edge(X,Z,'Co-Aut
   IntersectionCount(#X,#Y,Count<Z>))
   Similarity(#X,#Y,S):-Node(X, Accou
   Features-LP(#X,#Y,F1,F2):-Intersec
}
ITERATE(10) {
  INSERT Edge(X,Y,'Co-Author'):-Featu
  predict-LP(F1,F2)=true,
  confidence-LP(F1,F2) IN TOP 1%
```

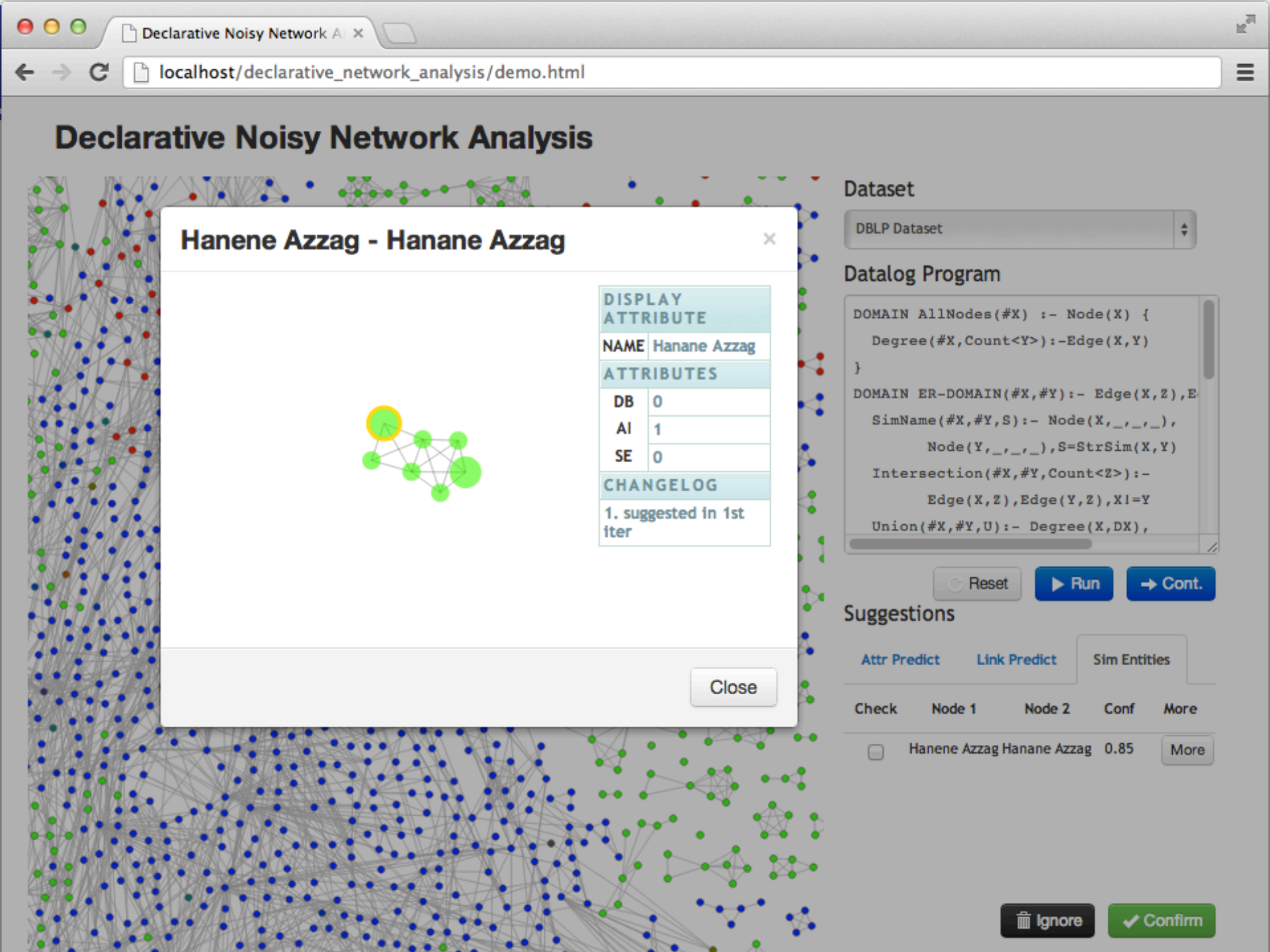
Reset | Run | Cont.

Suggestions

Attr Predict | Link Predict | Sim Entities

Check	From	To	Edge	Conf	More

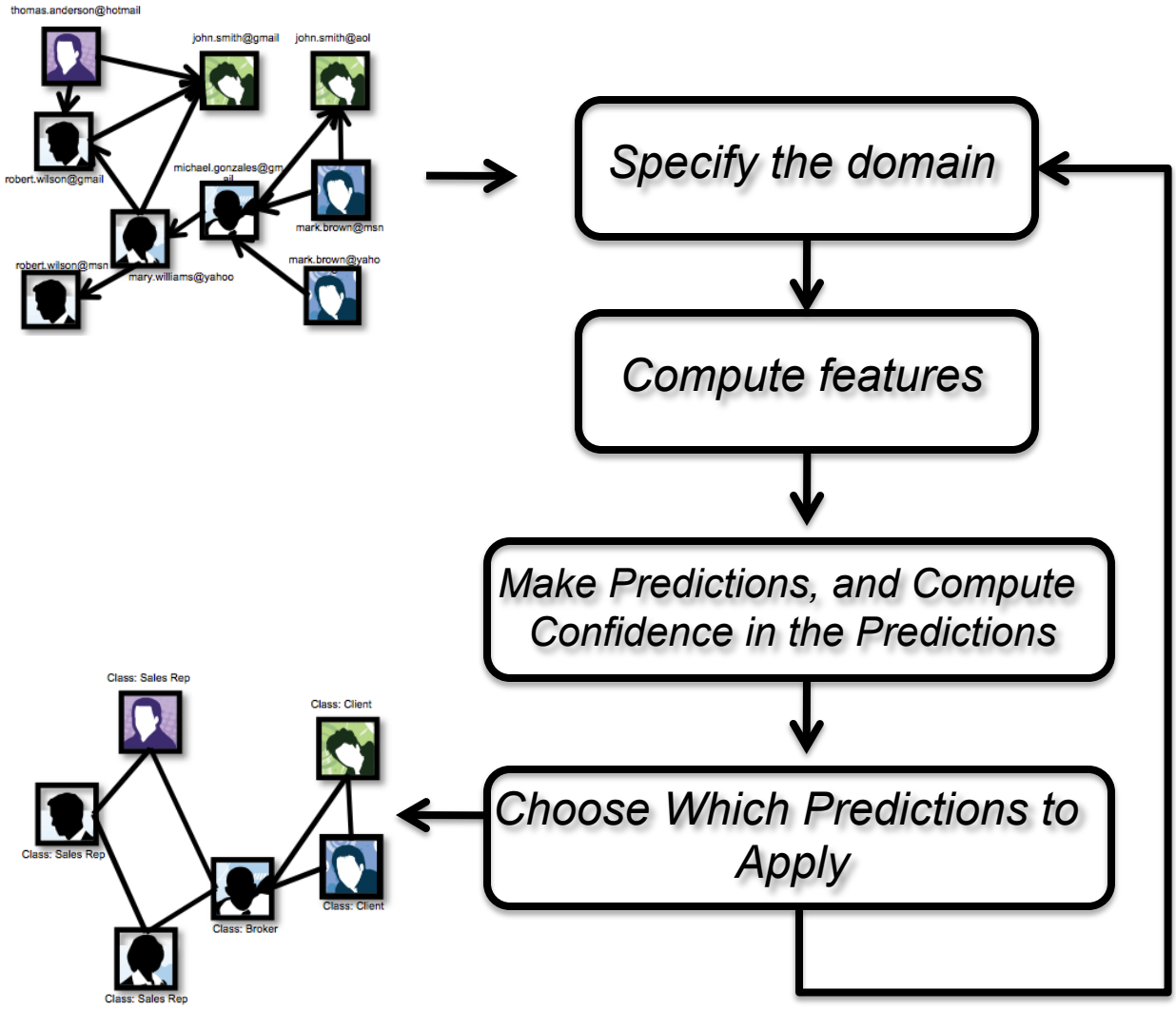
Ignore | Confirm


Declarative Noisy Network A
localhost/declarative_network_analysis/demo.html
Declarative Noisy Network Analysis
Hanene Azzag - Hanane Azzag
DISPLAY ATTRIBUTE
NAME Hanane Azzag
ATTRIBUTES
DB 0
AI 1
SE 0
CHANGELOG
1. suggested in 1st iter
Close
Dataset
DBLP Dataset
Datalog Program
DOMAIN AllNodes(#X) :- Node(X) {
Degree(#X,Count<Y>):-Edge(X,Y)
}
DOMAIN ER-DOMAIN(#X,#Y):- Edge(X,Z),E
SimName(#X,#Y,S):- Node(X,_,_,_),
Node(Y,_,_,_),S=StrSim(X,Y)
Intersection(#X,#Y,Count<Z>):-
Edge(X,Z),Edge(Y,Z),X!=Y
Union(#X,#Y,U):- Degree(X,DX),
Reset
Run
Cont.
Suggestions
Attr Predict
Link Predict
Sim Entities
Check
Node 1
Node 2
Conf
More
Hanene Azzag Hanane Azzag 0.85
More
Ignore
Confirm

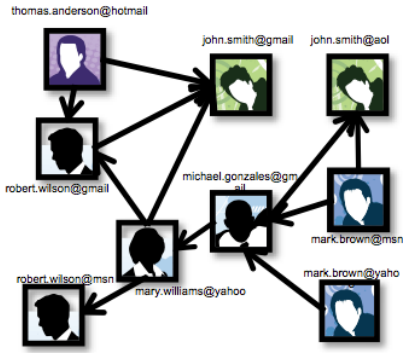
Overview of the Approach

- Declarative specification of the cleaning task
  - Datalog-based language for specifying --
    - Prediction features (including local and relational features)
    - The details of how to accomplish the cleaning task
    - Arbitrary interleaving or pipelining of different tasks
- A mix of declarative constructs and user-defined functions to specify complex prediction functions
- Optimize the execution through caching, incremental evaluation, pre-computed data structures …

Proposed Framework



Proposed Framework

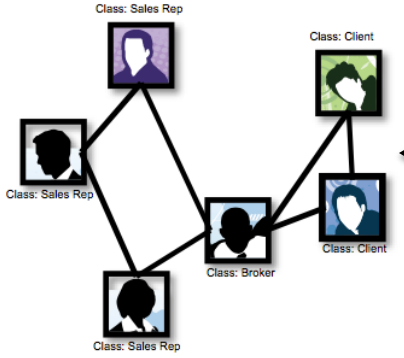


Specify the domain

Compute features

Make Predictions, and Compute Confidence in the Predictions

Choose Which Predictions to Apply

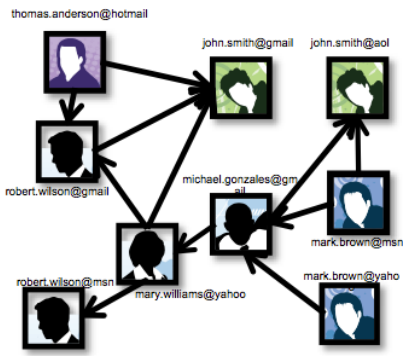


*For **attribute prediction**, the domain is a subset of the graph nodes.*

*For **link prediction** and **entity resolution**, the domain is a subset of pairs of nodes.*

***Local:** word frequency, income, etc.*
***Relational:** degree, clustering coeff., no. of neighbors with each attribute value, common neighbors between pairs of nodes, etc.*

Proposed Framework

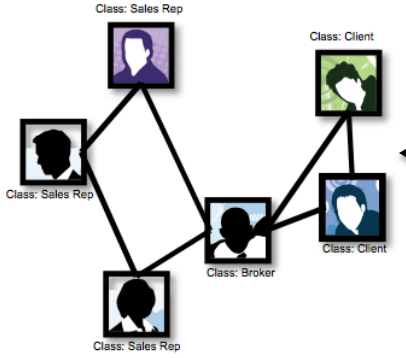


Specify the domain

Compute features

Make Predictions, and Compute Confidence in the Predictions

Choose Which Predictions to Apply



***Attribute prediction**: the missing attribute*

***Link prediction**: add link or not?*

***Entity resolution**: merge two nodes or not?*

After predictions are made, the graph changes:
***Attribute prediction** changes local attributes.*
***Link prediction** changes the graph links.*
***Entity resolution** changes both local attributes and graph links.*

Some Details

- Declarative framework based on Datalog
  - A declarative logic programming language (subset of Prolog)
  - Cleaner and more compact syntax than SQL
  - Not considered practical in past, but resurgence in recent years
    - Declarative networking, data integration, cloud computing, …
    - Several recent workshops on Datalog
- We use Datalog to express:
  - Domains
  - Local and relational features
- Extend Datalog with operational semantics to express:
  - Predictions (in the form of updates)
  - Iteration

Specifying Features

Degree:

Degree(X, COUNT<Y>) :-Edge(X, Y)

Number of Neighbors with attribute 'A'

NumNeighbors(X, COUNT<Y>) :− Edge(X, Y), Node(Y, Att='A')

Clustering Coefficient

NeighborCluster(X, COUNT<Y,Z>) :− Edge(X,Y), Edge(X,Z), Edge(Y,Z)

ClusteringCoeff(X, C) :− NeighborCluster(X,N), Degree(X,D), C=2*N/(D*(D-1))

Jaccard Coefficient

IntersectionCount(X, Y, COUNT<Z>) :− Edge(X, Z), Edge(Y, Z)

UnionCount(X, Y, D) :− Degree(X,D1), Degree(Y,D2), D=D1+D2-D3,
IntersectionCount(X, Y, D3)

Jaccard(X, Y, J) :− IntersectionCount(X, Y, N), UnionCount(X, Y, D), J=N/D

Update Operation

- Action to be taken itself specified declaratively
- Enables specifying, e.g., different ways to *merge* in case of entity resolution (i.e., how to *canonicalize)*

```
DEFINE Merge(X, Y)
{
    INSERT Edge(X, Z) :- Edge(Y, Z)
    DELETE Edge(Y, Z)
    UPDATE Node(X, A=ANew) :- Node(X,A=AX), Node(Y,A=AY),
                              ANew=(AX+AY)/2
    UPDATE Node(X, B=BNew) :- Node(X,B=BX), Node(X,B=BX),
                              BNew=max(BX,BY)
    DELETE Node(Y)
}
Merge(X, Y) :- Features (X, Y, F1,…,Fn), predict-ER(F1,…,Fn) = true,
                                         confidence-ER(F1,…,Fn) > 0.95
```

Example

- Real-world PubMed graph
  - Set of publications from the medical domain, their abstracts, and citations
- 50,634 publications, 115,323 citation edges
- Task: Attribute prediction
  - Predict if the paper is categorized as Cognition, Learning, Perception or Thinking
- Choose top 10% predictions after each iteration, for 10 iterations

```
DOMAIN Uncommitted(X):-Node(X,Committed='no')
{
  ThinkingNeighbors(X,Count<Y>):- Edge(X,Y), Node(Y,Label='Thinking')
  PerceptionNeighbors(X,Count<Y>):- Edge(X,Y), Node(Y,Label='Perception')
  CognitionNeighbors(X,Count<Y>):- Edge(X,Y), Node(Y,Label='Cognition')
  LearningNeighbors(X,Count<Y>):- Edge(X,Y), Node(Y,Label='Learning')
  Features-AP(X,A,B,C,D,Abstract):- ThinkingNeighbors(X,A), PerceptionNeighbors(X,B),
                          CognitionNeighbors(X,C), LearningNeighbors(X,D),Node(X,Abstract, _,_)
}
ITERATE(10)
{
  UPDATE Node(X,_,P,'yes'):- Features-AP(X,A,B,C,D,Text), P = predict-AP(X,A,B,C,D,Text),
                          confidence-AP(X,A,B,C,D,Text) IN TOP 10%
}
```

Prototype Implementation

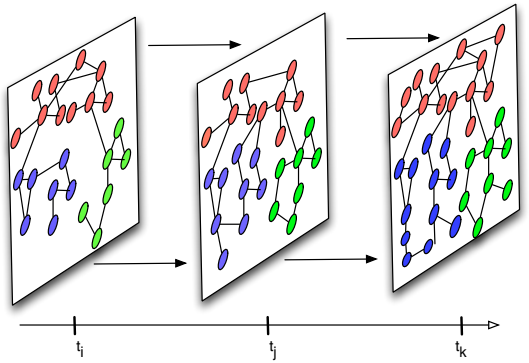
- Using a simple RDBMS built on top of Java Berkeley DB
  - Predicates in the program correspond to materialized tables
  - Datalog rules converted into SQL
- Incremental maintenance:
  - Every *set of changes* done by AP, LP, or ER logged into two *change tables ΔNodes and ΔEdges*
  - Aggregate maintenance is performed by aggregating the change table then refreshing the old table
- Proved hard to scale
  - Incremental evaluation much faster than recompute, but SQL-based evaluation was inherently a bottleneck
  - Hard to do complex features like *centrality measures*
  - In the process of changing the backend to use a new distributed graph processing framework

Outline

- Overview
- NScale Distributed Programming Framework
- Declarative Graph Cleaning
- Historical Graph Data Management
- Continuous Queries over Distributed Graphs
- Conclusions

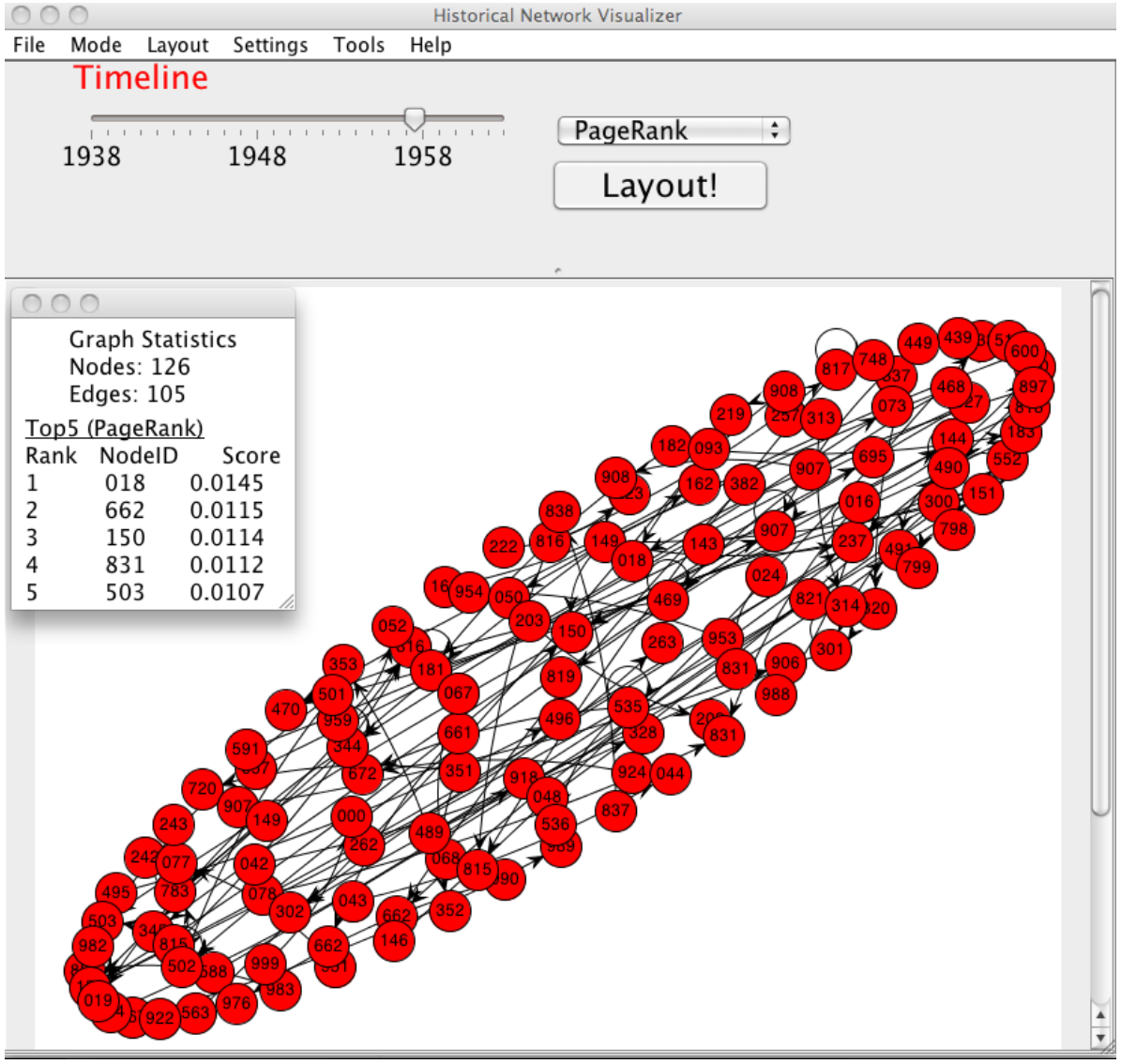
Historical Graph Data Management

- Increasing interest in temporal analysis of information networks to:
  - Understand evolutionary trends (e.g., how communities evolve)
  - Perform comparative analysis and identify major changes
  - Develop models of evolution or information diffusion
  - Visualizations over time
  - For better predictions in the future

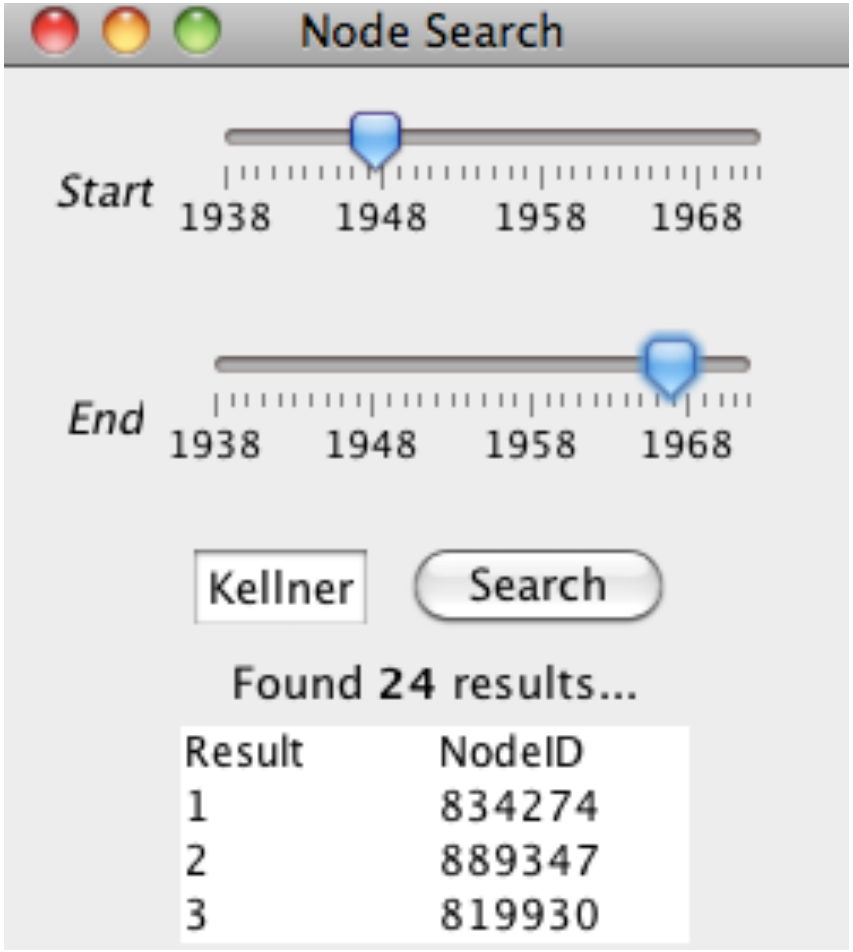


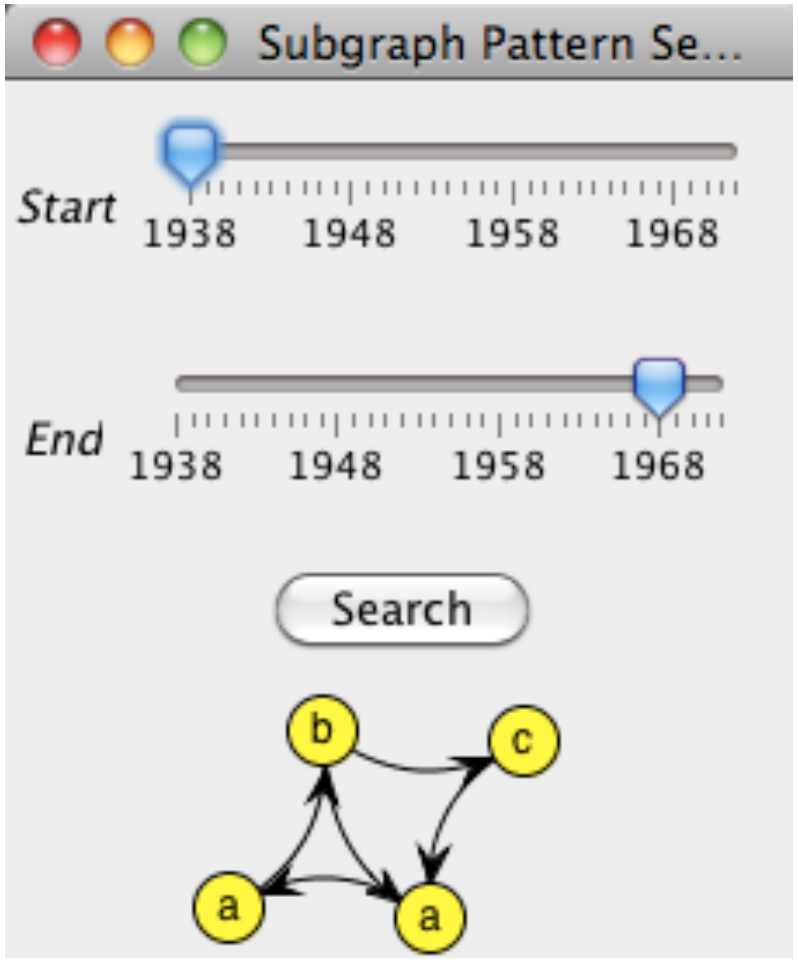
- Focused exploration and querying
  - *"Who had the highest PageRank in a citation network in 1960?"*
  - *"Identify nodes most similar to* X *as of one year ago"*
  - *"Identify the days when the network diameter (over some transient edges like messages) is smallest"*
  - *"Find a temporal subgraph pattern in a graph"*

Hinge: A System for Temporal Exploration

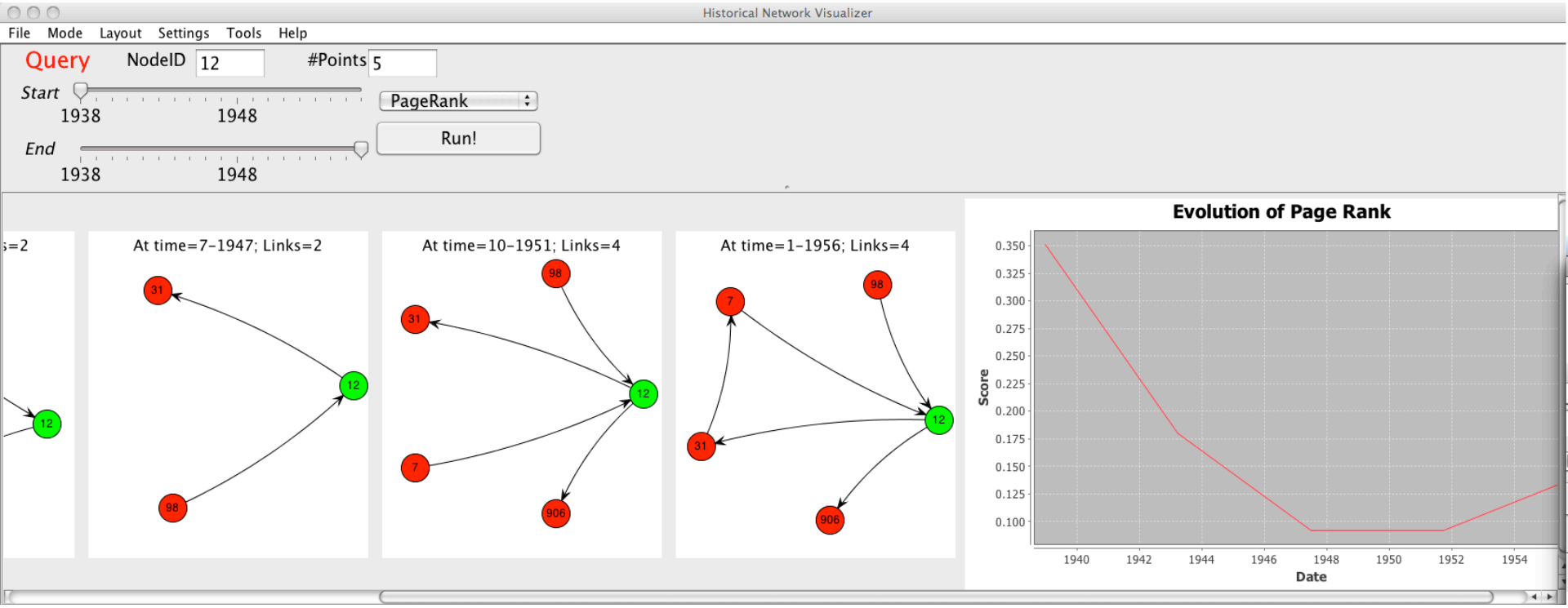


Hinge: A System for Temporal Exploration





Hinge: A System for Temporal Exploration



Snapshot Retrieval Queries

- Focus of the work so far: snapshot retrieval queries
  - Given one *timepoint* or a set of *timepoints* in the past, retrieve the corresponding *snapshots* of the network in memory
  - Queries may specify only a subset of the columns to be fetched
  - Some more complex types of queries can be specified
- Given the ad hoc nature of much of the analysis, one of the most important query types
- Key challenges:
  - Needs to be very fast to support interactive analysis
  - Should support analyzing 100's or more snapshots simultaneously
  - Support for distributed retrieval and distributed analysis (e.g., using Pregel)

Prior Work

- Temporal relational databases
  - Vast body of work on models, query languages, and systems
  - Distinction between *transaction-time* and *valid-time* temporal databases
  - Snapshot retrieval queries also called *valid timeslice* queries

- Options for executing snapshot queries
  - External Interval Trees [Arge and Vitter, 1996], External Segment Trees [Blakenagal and Guting, 1994], Snapshot index [Slazberg et al., 1999], …

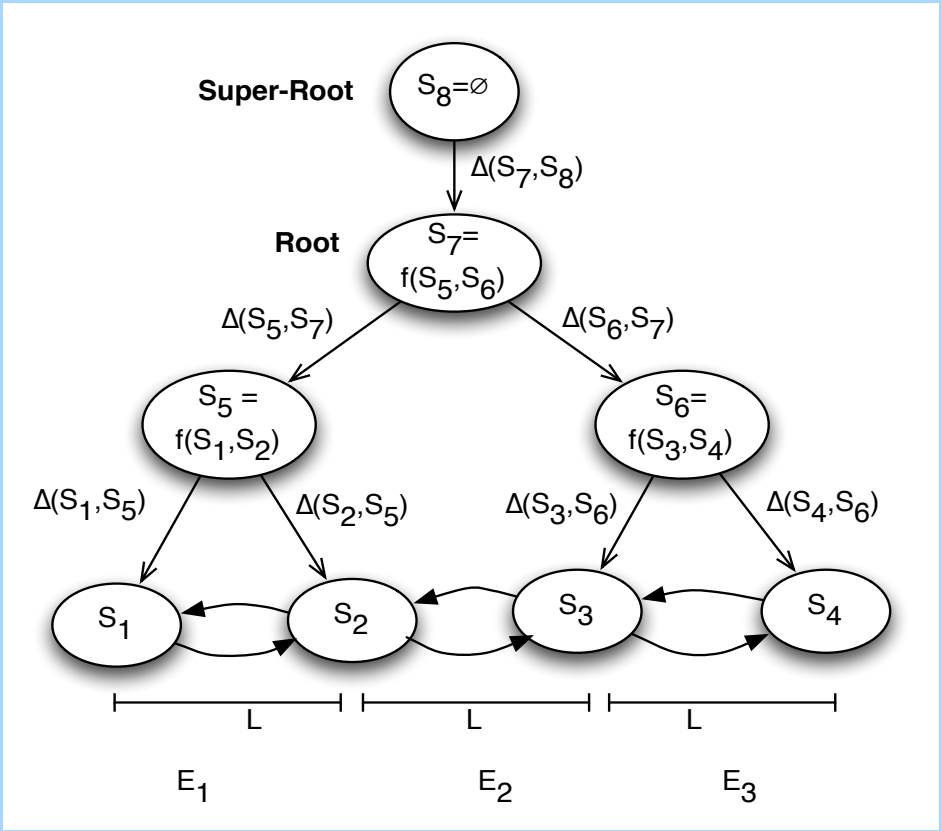
- Key limitations
  - Not flexible or tunable; not easily parallelizable; no support for multi-point queries; intended mainly for disks

System Overview

Currently supports a programmatic API to access the historical graphs

GraphPool: Store many graphs in memory in an

DeltaGraph: Hierarchical index structure with (logical) snapshots at the leaves



Manager(. . .);

ructure along with node names as of

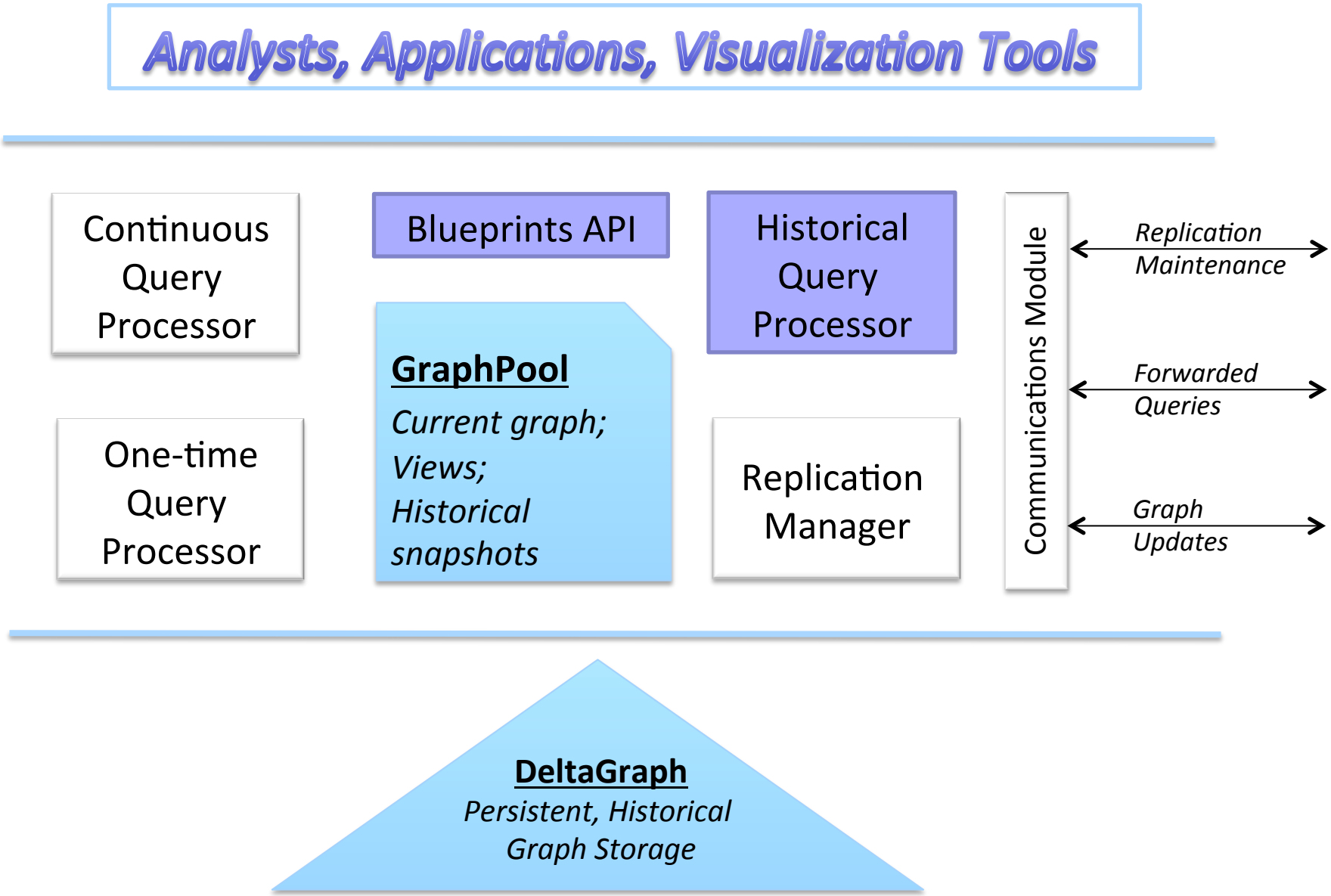
n("1/2/1985", "+node:name");

Nodes();
nodes.get(0).getNeighbors();
des.get(0), neighborList.get(0));

ructure alone on Jan 2, 1986 and Jan

stGraphs(listOfDates, "");

Overview


Analysts, Applications, Visualization Tools
Continuous Query Processor
Blueprints API
Historical Query Processor
Communications Module
Replication Maintenance
Forwarded Queries
Graph Updates
One-time Query Processor
GraphPool
Current graph; Views; Historical snapshots
Replication Manager
DeltaGraph
Persistent, Historical Graph Storage

Overview

Analysts,

Continuous Query Processor

One-time Query Processor

Currently supports a programmatic API to access the historical graphs

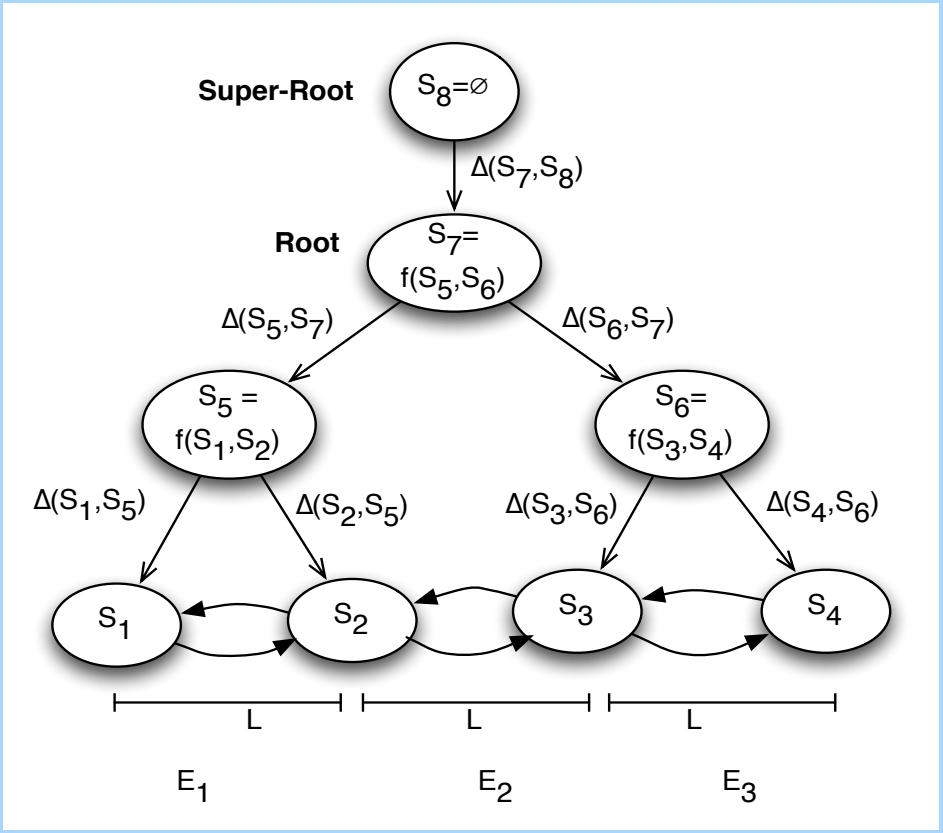
```
/* Loading the index */
GraphManager gm = new GraphManager(. . . );
gm.loadDeltaGraphIndex(. . . );
. . .
/* Retrieve the historical graph structure along with node names as of Jan 2, 1985 */
HistGraph h1 = gm.GetHistGraph("1/2/1985", "+node:name");
. . .
/* Traversing the graph*/
List<HistNode> nodes = h1.getNodes();
List<HistNode> neighborList = nodes.get(0).getNeighbors();
HistEdge ed = h1.getEdgeObj(nodes.get(0), neighborList.get(0));
. . .
/* Retrieve the historical graph structure alone on Jan 2, 1986 and Jan 2, 1987 */
listOfDates.add("1/2/1986");
listOfDates.add("1/2/1987");
List<HistGraph> h1 = gm.getHistGraphs(listOfDates, "");
. . .
```

Persistent, Historical Graph Storage

Overview

Analysts, Applications, Visualization Tools

DeltaGraph: Hierarchical index structure with (logical) snapshots at the leaves



storical
Query
ocessor

lication
anager

Communications Module

Replication Maintenance

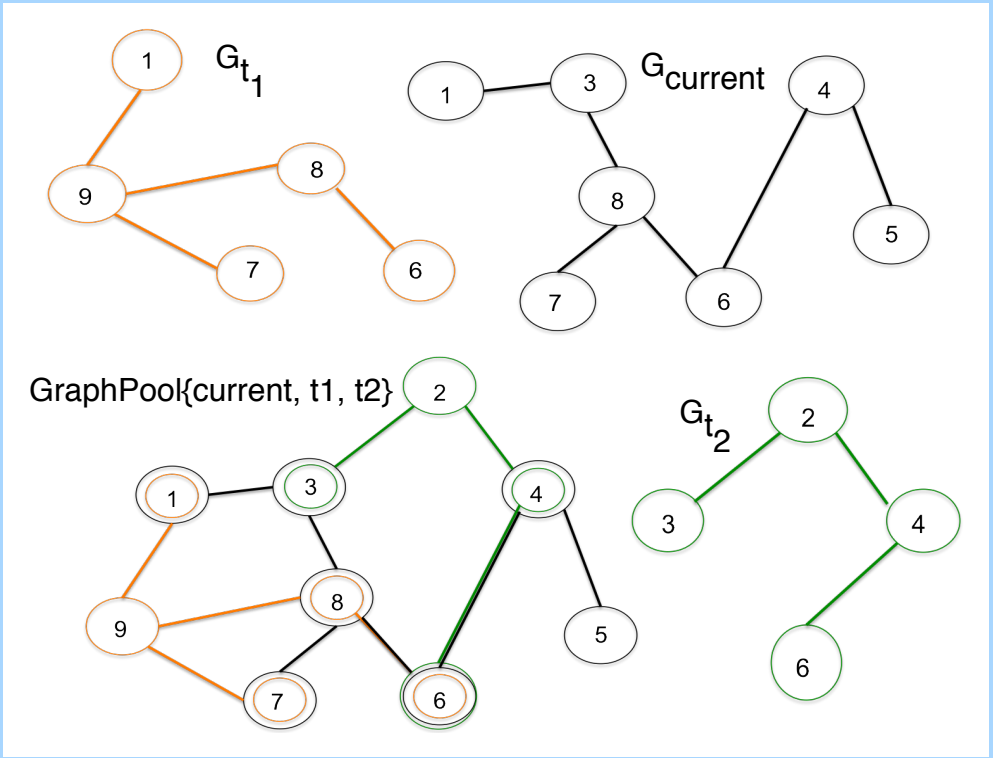
Forwarded Queries

Graph Updates

Overview

Analysts, Applications, Visualization Tools

GraphPool: Store many graphs in memory in an overlaid fashion



rical
ery
essor

ation
ager

Communications Module

Replication Maintenance

Forwarded Queries

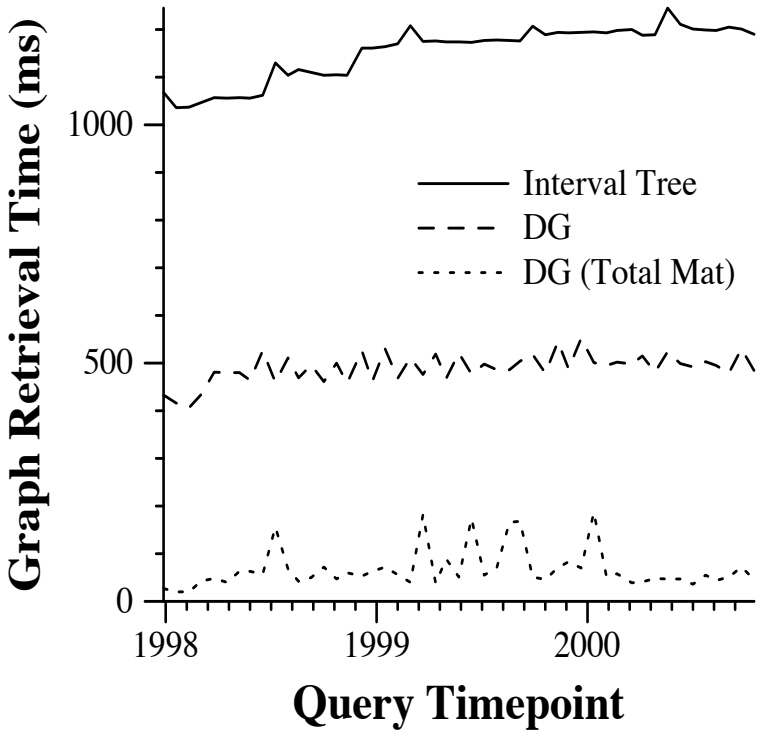
Graph Updates

Summary

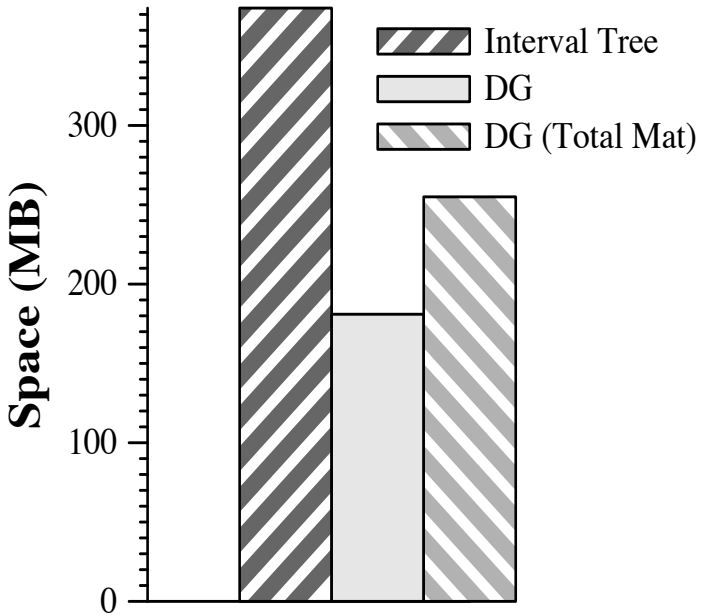
- *Edge deltas* stored in a key-value store
  - Currently uses Kyoto Cabinet disk-based key-value store
  - Parallelized by running a separate instance on each machine
- Snapshot retrieval arbitrarily parallelizable
  - Can load the snapshot(s) in parallel on any number of machines
    - Supports a simplified Pregel-like abstraction on top
- Highly tunable
  - Can control the access times, latencies, storage requirements by appropriate choice of parameter values
  - Supports pre-fetching to reduce online query latencies
- Extensible
  - APIs to extend the basic structure to support *subgraph pattern matching, reachability etc.*

Empirical Results

- DeltaGraph vs In-Memory Interval Tree



(a) Performance: Dataset 2a



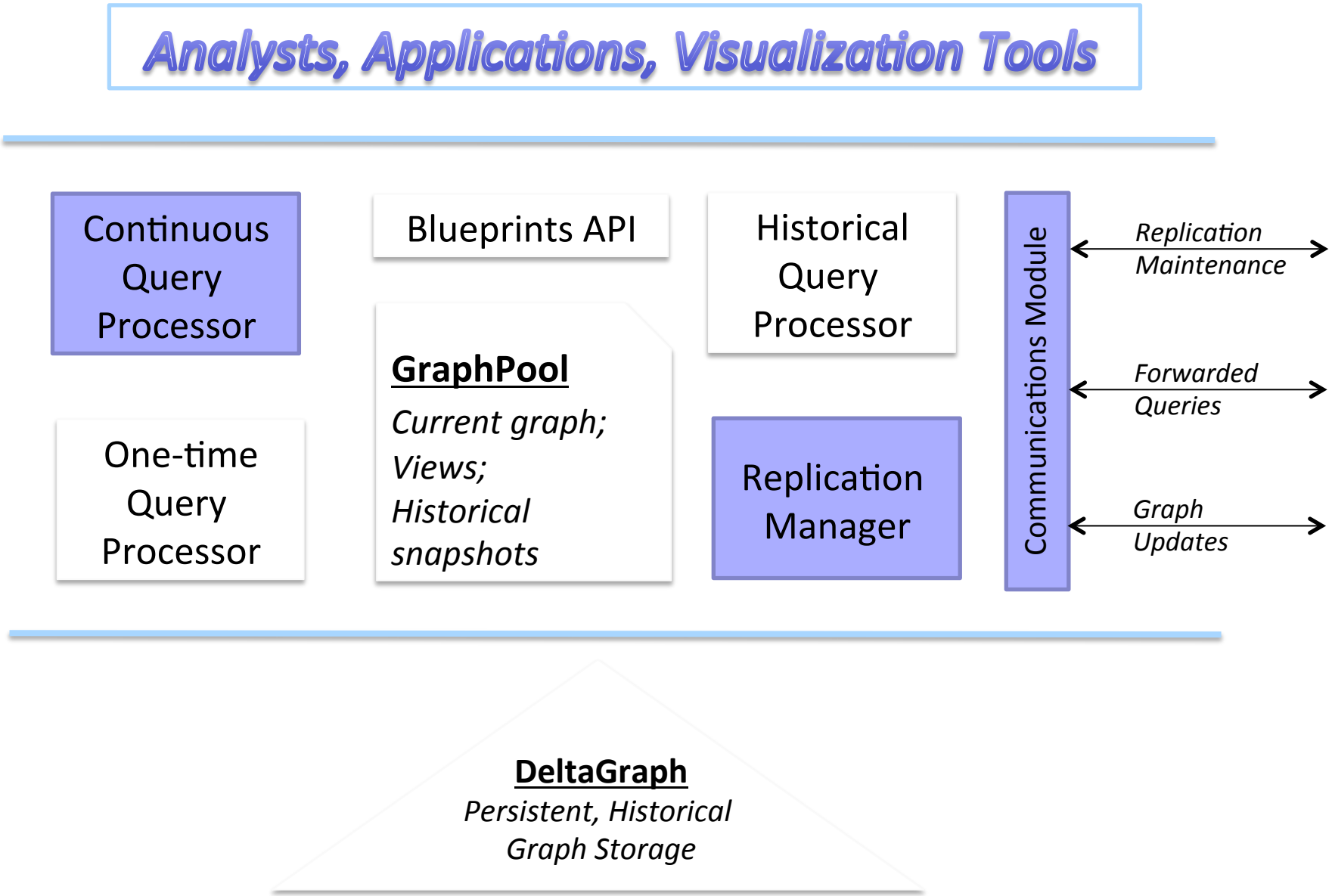
(b) Memory: Dataset 2a

Dataset 2a: 500,000 nodes+edges, 500,000 events

Outline

- Overview
- NScale Distributed Programming Framework
- Declarative Graph Cleaning
- Historical Graph Data Management
- Continuous Queries over Distributed Graphs
- Conclusions

System Architecture



Real-time Graph Queries and Analytics

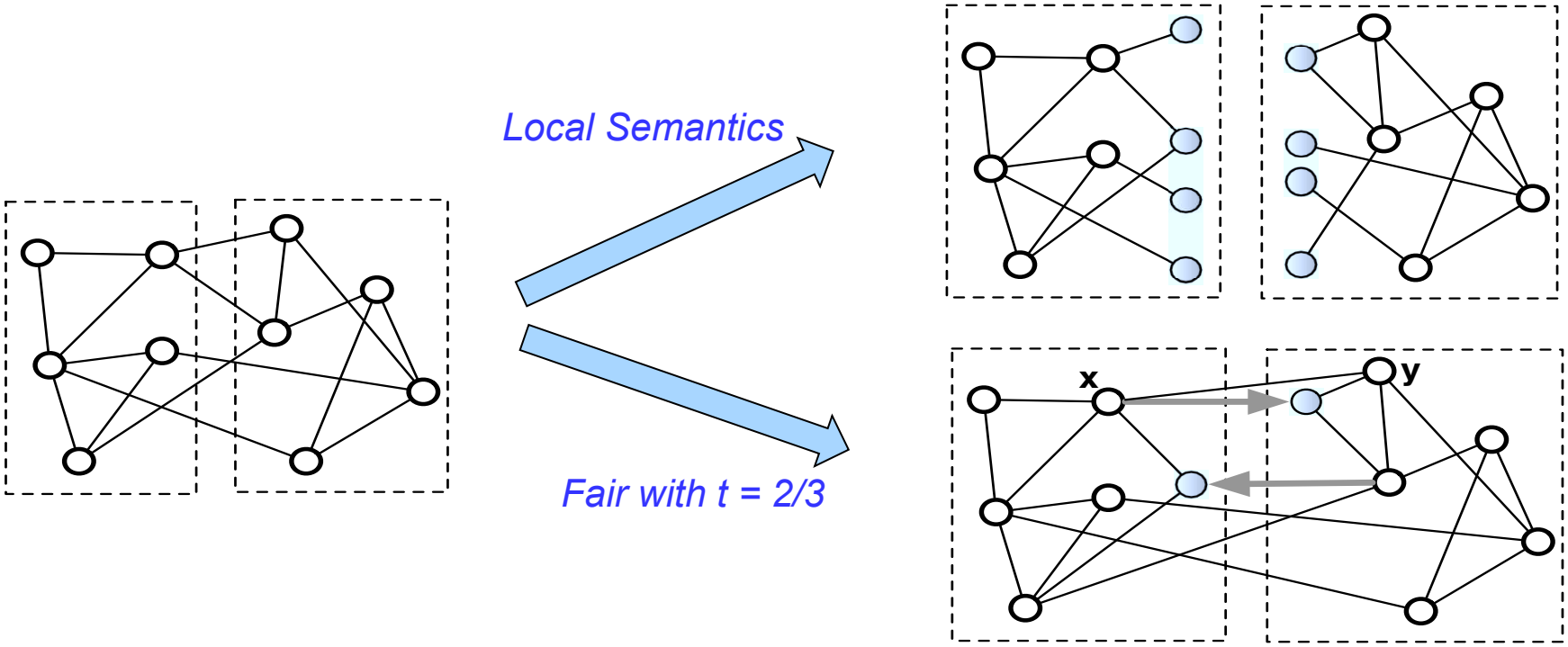
- Increasing need for executing queries and analysis tasks in real-time on “data streams”
  - Ranging from simple “monitor updates in the neighborhood” to complex “trend discovery” or “anomaly detection” queries
- Very low latencies desired
  - Trade-offs between push/pre-computation vs pull/on-demand
  - Sharing and adaptive execution necessary
- Parallel/distributed solutions needed to handle the scale
  - Random graph partitioning typically results in large edge cuts
  - Distributed traversals to answer queries leading to high latencies and high network communication
  - Sophisticated partitioning techniques often do not work either

Example: Fetch Neighbors' Updates

- Dominant type of queries in many scenarios (e.g., social networks)
  - *How to execute if the graph is partitioned across many machines?*
  - A node's neighbors may be on a different machine
- Prior approaches
  - On-demand → High latencies because of network communication
  - Local semantics [Pujol et al., SIGCOMM'11]
    - For every node, all neighbors replicated locally
    - High, often unnecessary network communication overhead
- Our approach [SIGMOD'12]
  - How to choose what to replicate? – A new "fairness" criterion
  - Push vs Pull? – Fine-grained access pattern monitoring
  - Decentralized decision making

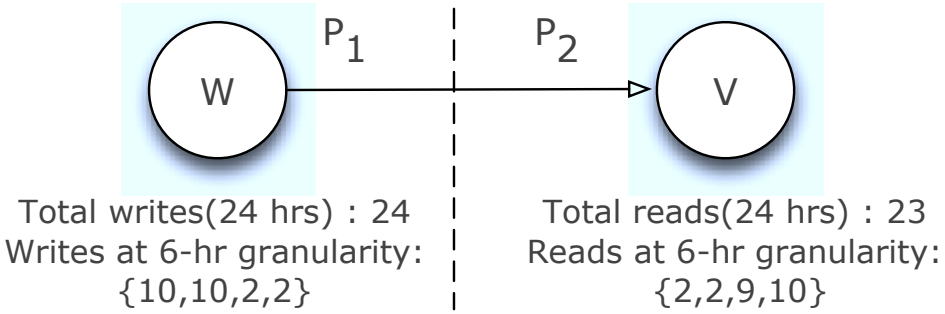
Our Approach

- Key idea 1
  - Use a “fairness” criterion to decide what to replicate
    - For every node, at least *t* fraction of nodes should be present locally
  - Can make some progress for all queries
  - Guaranteeing fairness NP-Hard



Our Approach

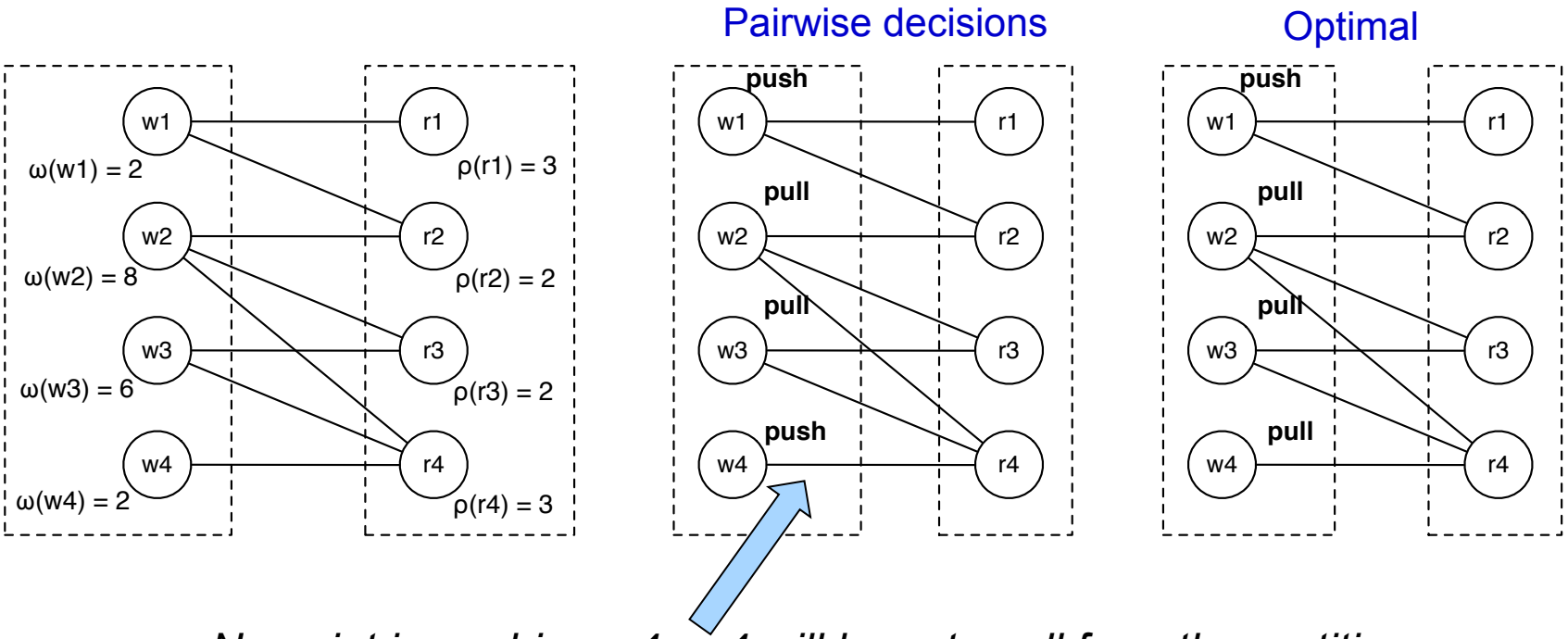
- Key idea 2
  - Exploit patterns in the update/query access frequencies



  - Use *pull* replication in the first 12 hours, *push* in the next 12
  - Significant benefits from adaptively changing the replication decision
  - Such patterns observed in human-centric networks like social networks

Our Approach

- Key idea 3
  - Make replication decisions for all nodes in a pair of partitions together
    - Prior work had suggested doing this for each *(writer, reader)* pair separately
    - Works in the publish-subscribe domain, but not here
  - Can be reduced to *maximum density sub-hypergraph* problem



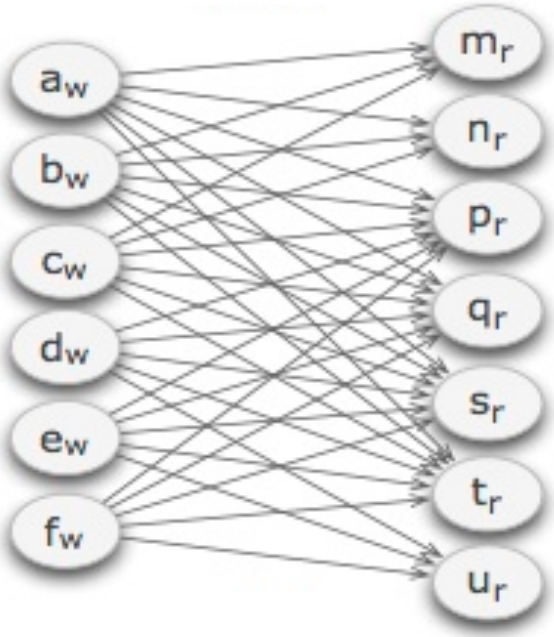
No point in pushing w4 – r4 will have to pull from the partition anyway

Example: Ego-centric Aggregates

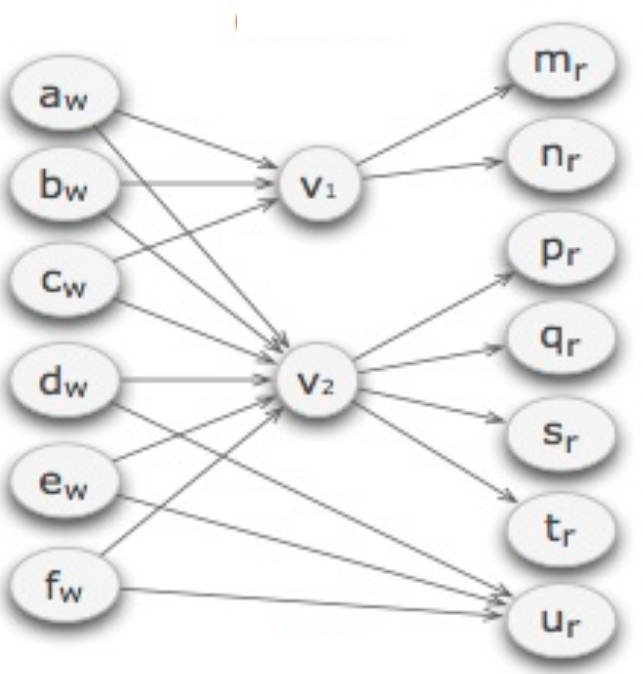
- Continuously evaluate an aggregate in the local neighborhoods of all nodes of a graph
  - For example, to do "*ego-centric trend analysis in social networks", or "detecting nodes with anomalous communication activity*"
  - Challenging even if data all on a single machine
- Prior approaches
  - On-demand → High latencies because of computational cost
  - Continuously maintain all the query results (pre-computation):
    - Potentially wasted computation
    - Too many queries to be executed
- Our approach [ongoing work]
  - Access-pattern based on-demand vs pre-computation decisions
  - Aggressive sharing across different queries

Our Approach

- Key idea 4
  - Exploit commonalities across queries to share partial computation
  - Use *graph compression*-like techniques to minimize the computation



Original dataflow graph for aggregate computation – each edge denotes a potential computation



Computation cost can be reduced by identifying "bi-cliques"

Conclusions and Ongoing Work

- Graph data management becoming increasingly important
- Many challenges in dealing with the scale, the noise, and the variety of analytical tasks
- Presented:
  - A declarative framework for cleaning noisy graphs
  - A system for managing historical data and snapshot retrieval
  - Techniques for managing and querying highly dynamic graphs
- Ongoing work on improving and extending this preliminary work
  - Developing a unified query language based on Datalog
  - Replication and pre-computation for continuous queries
  - Efficiently supporting distributed graph analytics
  - Developing effective graph compression techniques
  - New graph partitioning techniques