Enabling Declarative Graph Analytics over Large, Noisy Information Networks

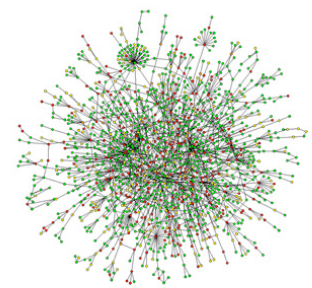
Amol Deshpande

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

Joint work with: Prof. Lise Getoor, Walaa Moustafa, Udayan Khurana, Jayanta Mondal

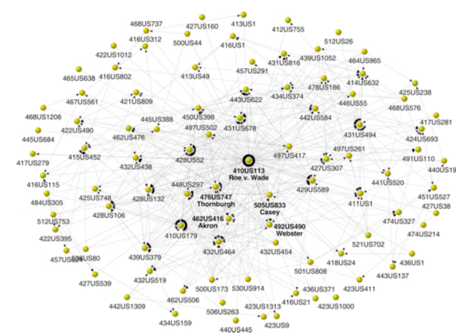
Motivation: Information Networks

- Everywhere and growing in numbers…
  - Social networks, social contact graphs
  - Email networks, financial transaction networks
  - Biological networks, disease transmission networks
  - Citation networks, IP traffic data, Web
  - …



A protein-protein interaction network

- Intense amount of work already on:
  - … understanding properties of these networks
  - … visualizations
  - … developing models of evolution
  - … cleaning inherently noisy observational data
  - … comparative analytics
  - and so on…



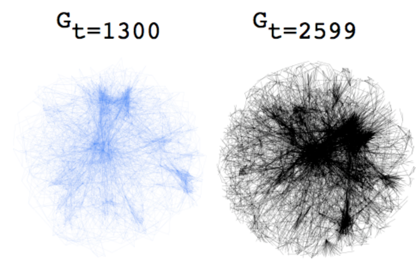
Supreme court citation network

Motivation: Information Networks

- Lack of established data management tools
  - Much of the analysis exploratory, domain specific, and hard to abstract
- Some of the key data management challenges
  - Inherent noise and uncertainty in the raw observation data
    - → Support for *graph cleaning* must be tightly integrated into the system
    - Graph cleaning techniques often domain specific
    - → Uncertainty-aware query evaluation algorithms needed that can handle new types of *identity* uncertainties
  - Very large volumes of heterogeneous data
    - → Distributed/parallel storage and query processing needed
    - Graph partitioning notoriously hard to do effectively
  - Highly dynamic and rapidly changing data as well as workloads
    - → Need to support real-time processing through aggressive replication and pre-computation

Motivation: Information Networks

- Lack of established data management tools
  - Much of the analysis exploratory, domain specific, and hard to abstract
- Some of the key data management challenges
  - Managing historical information
    - → Need to support complex temporal analysis
    - → Must manage large volumes of historical traces and support efficient retrieval of past network snapshots
    - → Need to support different frameworks for *inferring* the trace itself from snapshots
  - Lack of established query languages
    - → Develop new languages !!
    - → … or preferably reuse an old one

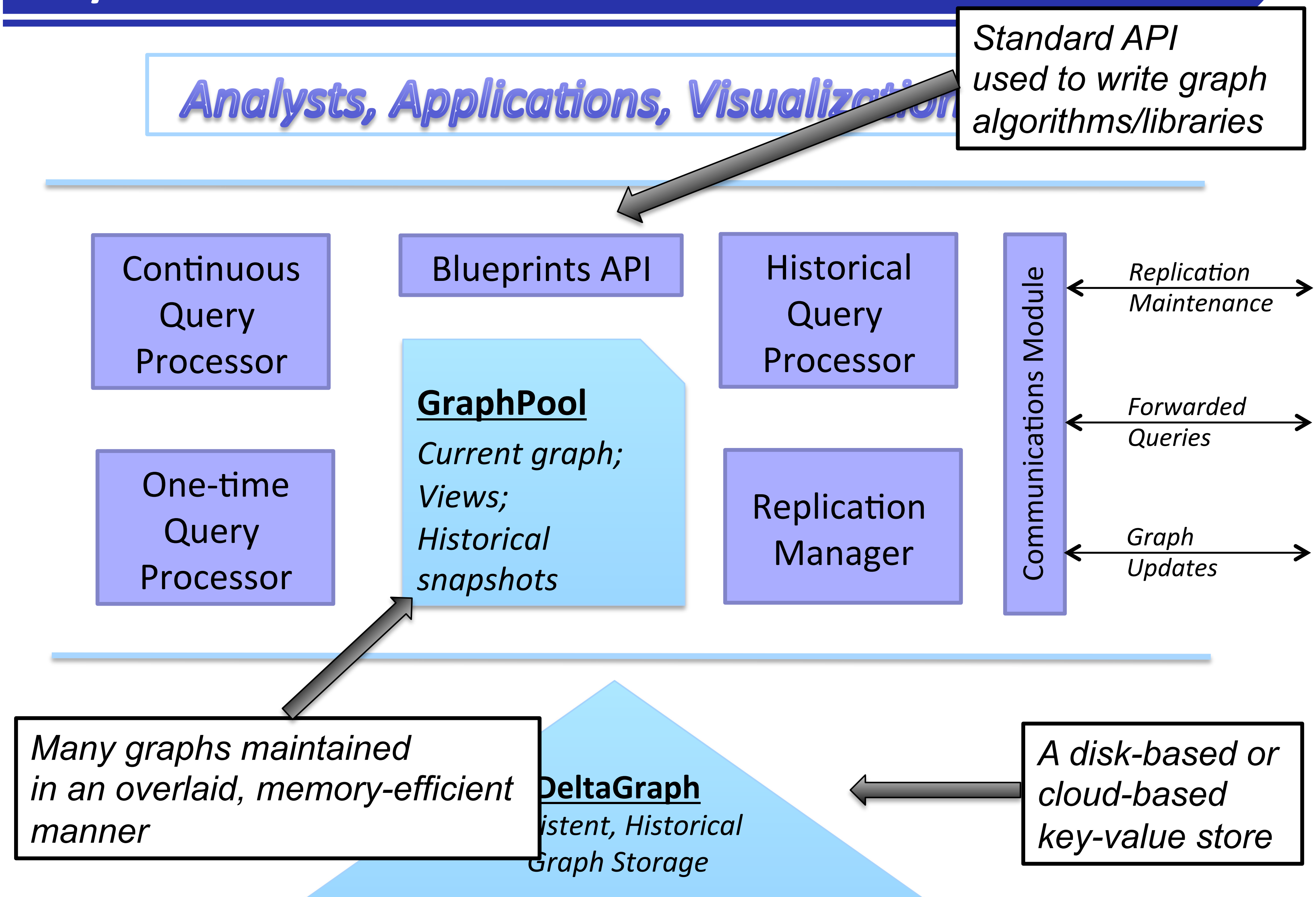


What we are doing

- Goal: build a data management system and frameworks that can manage large dynamically-changing graphs and support a variety of analytics over them
  - Focus on the abstractions and the system, less on specific analysis techniques

- Work so far:
  - Declarative graph cleaning
    - Proposed and built a declarative framework for specifying complex network analysis and cleaning tasks [GDM'11]
  - Real-time continuous query processing
    - Aggressive replication to manage very large dynamic graphs efficiently in a distributed manner, and to execute continuous queries over them [SIGMOD'12]
  - Historical graph management
    - Efficient single-point or multi-point snapshot retrieval over very large historical graph traces [under submission]
  - Ego-centric pattern census [ICDE'12]

System Architecture



Outline

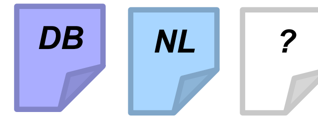
- Overview
- Declarative Graph Cleaning
- Historical Graph Data Management
- Distributed Management of Dynamic Graphs
- Conclusions

Motivation

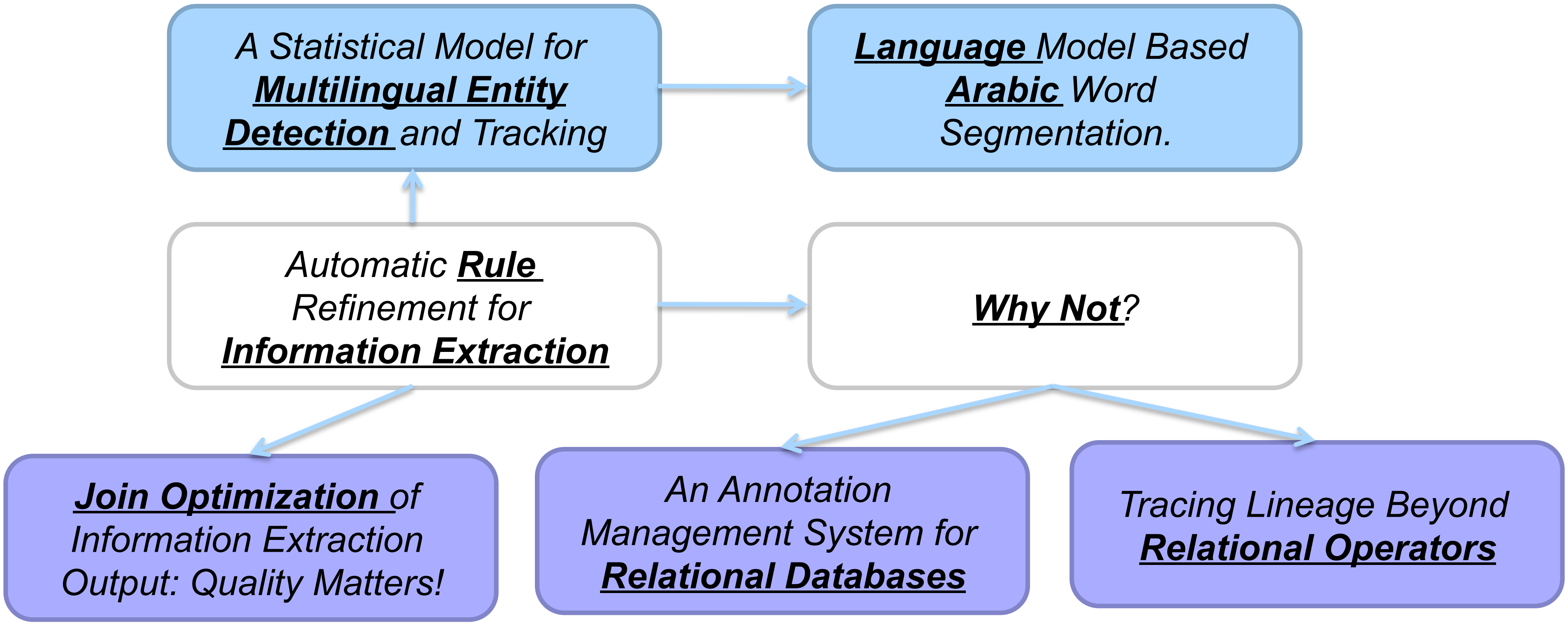
- The *observed information networks* are often noisy and incomplete
  - Missing attributes, missing links
  - Ambiguous references to the same entity
- Need to extract the underlying *true information network* through:
  - Attribute Prediction: *to predict values of missing attributes*
  - Link Prediction: *to infer missing links*
  - Entity Resolution: *to decide if two references refer to the same entity*
- Typically iterative and interleaved application of the techniques
- These prediction tasks can use:
  - Local node information
  - *Relational* information in the neighborhood of the node

Attribute Prediction

Task: *Predict topic of the paper*

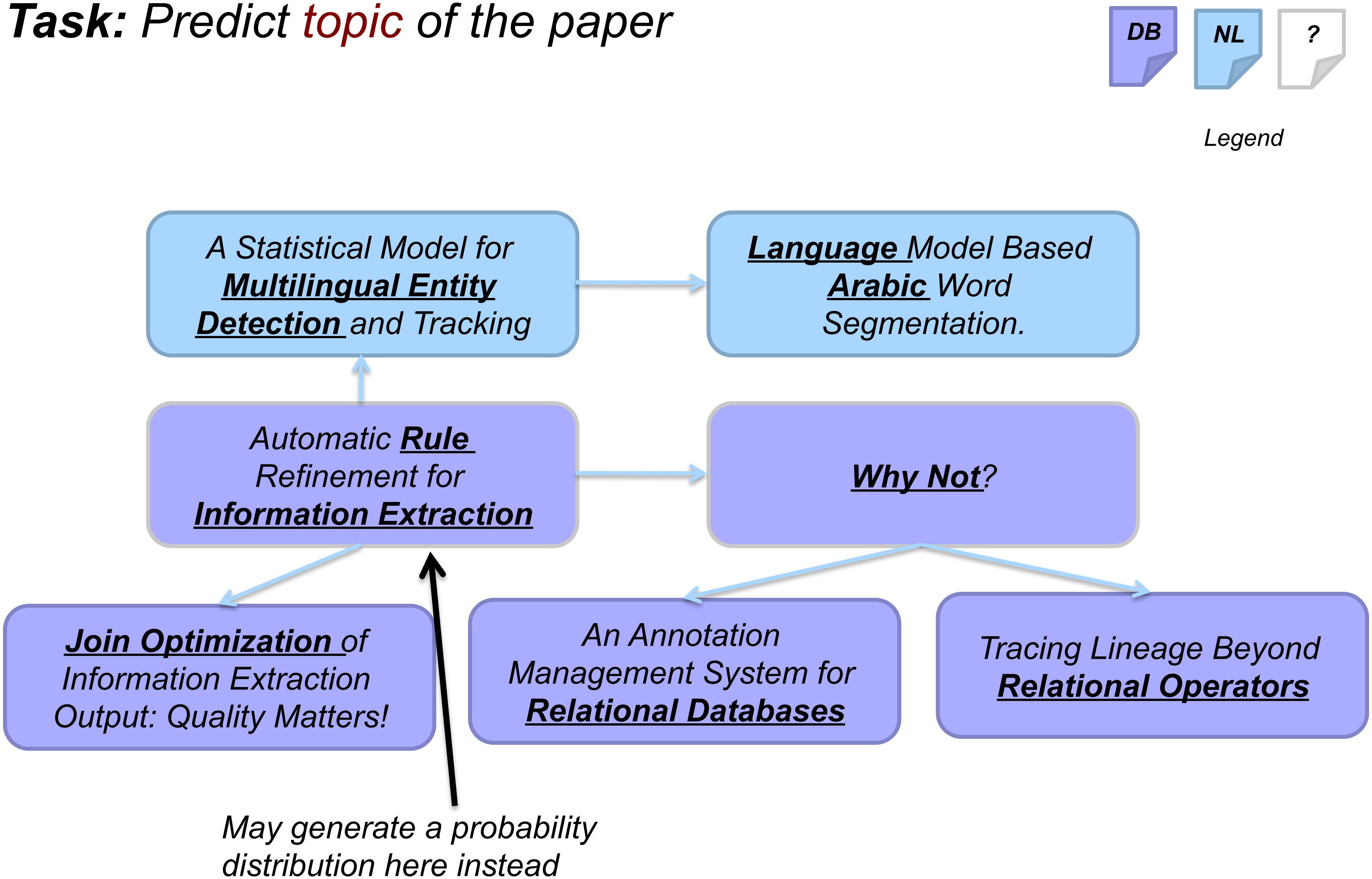


Legend



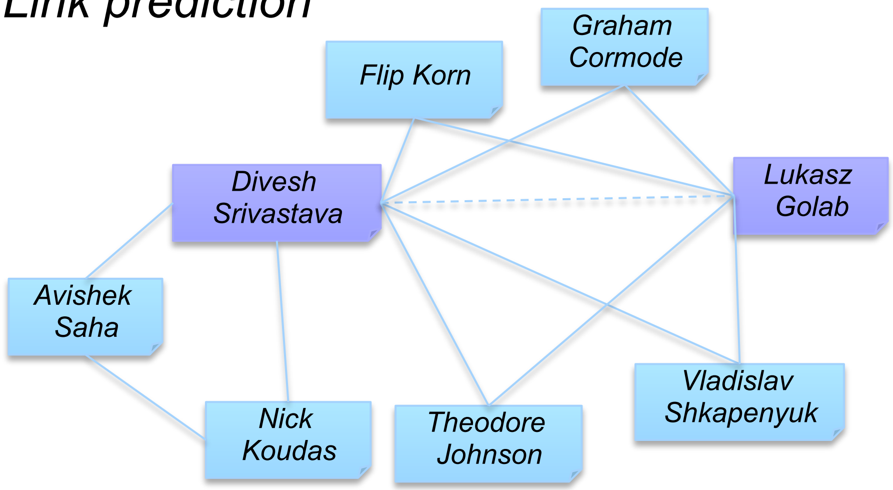
Attribute Prediction

Task: *Predict topic of the paper*

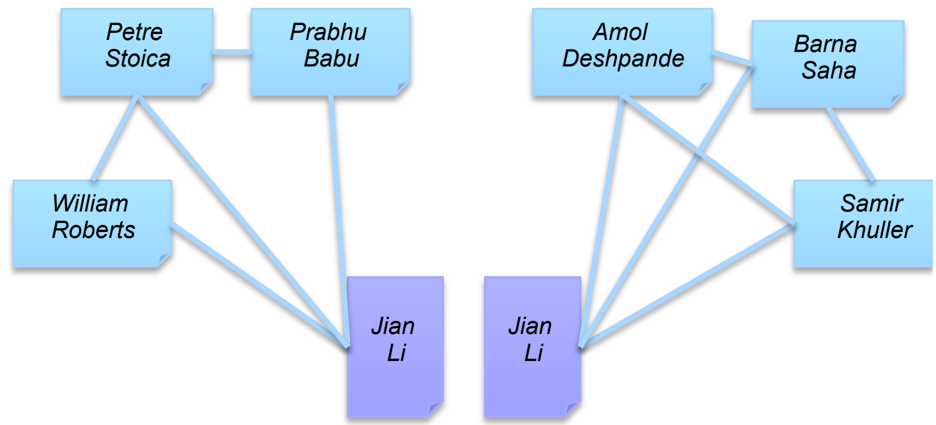


Collective (relational) Inference

Link prediction



Entity resolution



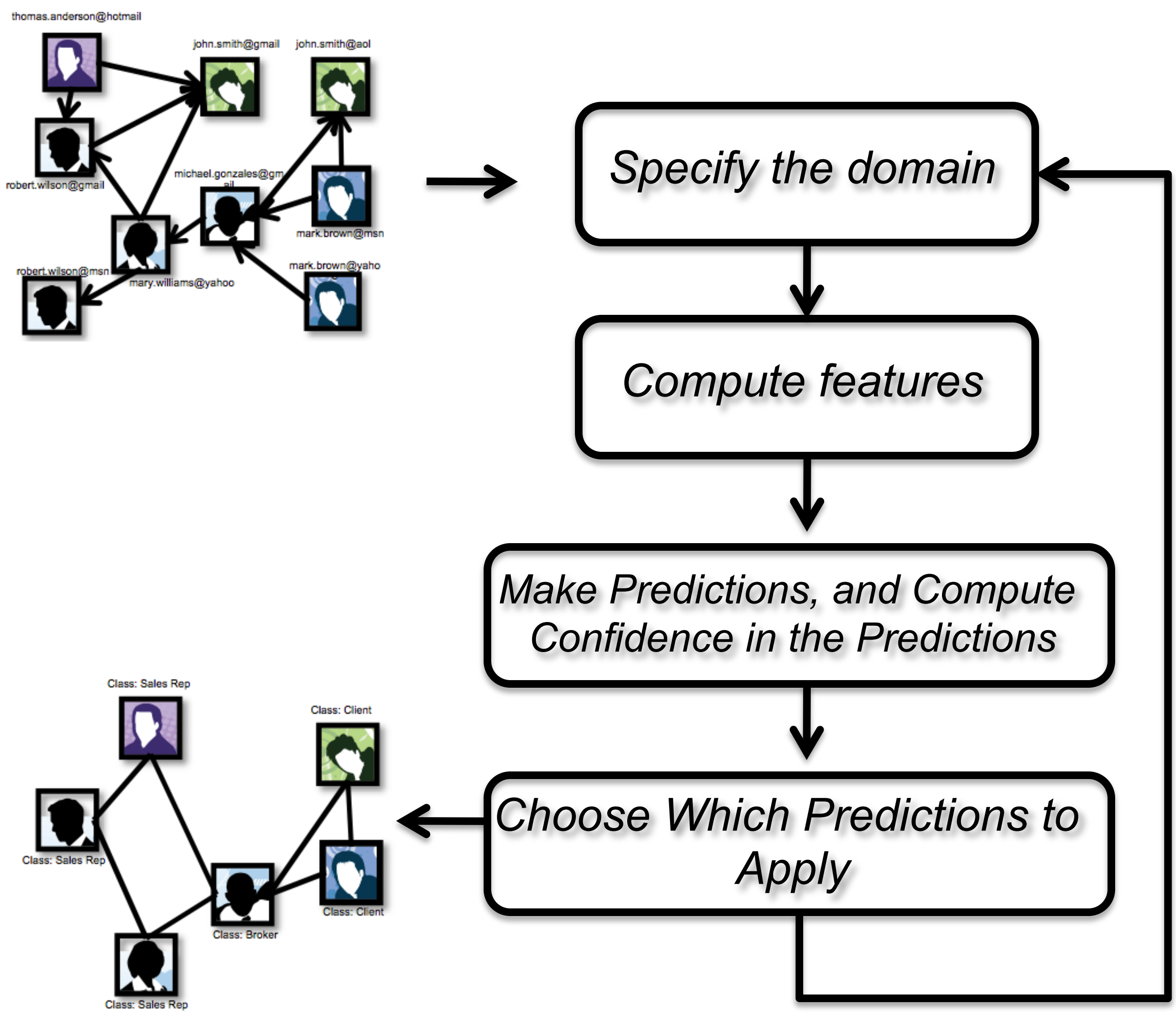
- Many collective techniques have been developed over the years
  - However, no support from data management systems to do this effectively
  - Hard for a network analyst to easily construct and compare new techniques
    - Especially for *joint* inference, i.e., interleaved and pipelined application
  - No re-usability, and much repetition of work

Our Goal

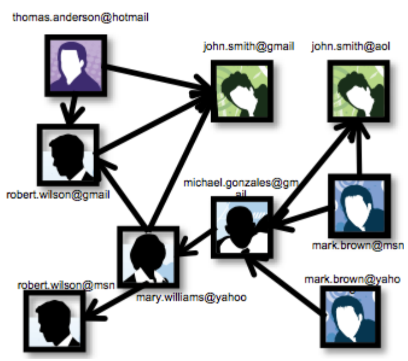
- Motivation: To support declarative network inference
- Desiderata:
  - Declarative specification of the prediction features
    - Local features
    - Relational features
  - (Almost-)declarative specification of tasks
    - Attribute prediction, Link prediction, Entity resolution
  - Support for arbitrary interleaving or pipelining
  - Support for complex prediction functions

Handle all that efficiently

Proposed Framework



Proposed Framework

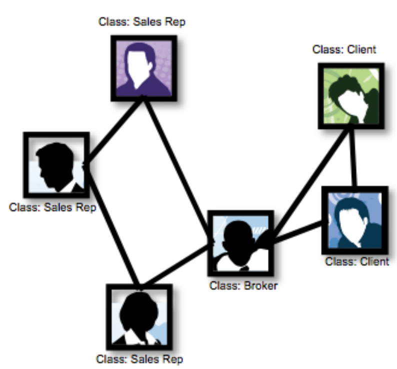


Specify the domain

Compute features

Make Predictions, and Compute Confidence in the Predictions

Choose Which Predictions to Apply

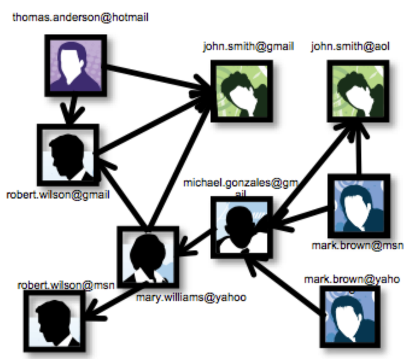


For ***attribute prediction****, the domain is a subset of the graph nodes.*

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

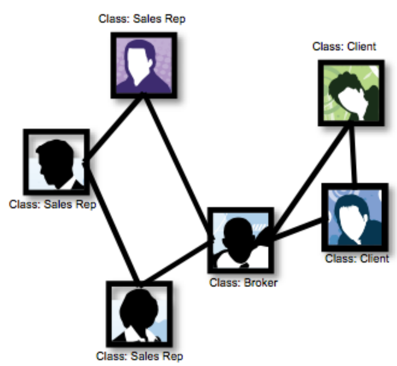


Specify the Domain

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

- Use Datalog to express:
  - Domains
  - Local and relational features
- Extend Datalog with operational semantics (vs. fix-point 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

Specifying Domains

- Domains used to restrict the space of computation for the prediction elements

- Space for this feature is $|V|^2$

  Similarity(X, Y, S) :–Node(X, Att=V1), Node(Y, Att=V1), S=f(V1, V2)

- Using this domain the space becomes $|E|$:

  DOMAIN D(X,Y) :- Edge(X, Y)

- Other DOMAIN predicates:
  - Equality on attribute values
  - Locality sensitive hashing
  - String similarity joins
  - Traverse edges

Prediction and Confidence Functions

- The prediction and confidence functions are user defined functions
- Can be based on *logistic regression*, *Bayes classifier*, or any other classification algorithm
- The confidence is the class membership value
  - In logistic regression, the confidence can be the value of the logistic function
  - In Bayes classifier, the confidence can be the posterior probability value

Update Operation

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

```
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
```

Pipelining

```
DOMAIN ER(X,Y) :- ….
{
    ER1(X,Y,F1) :- …
    ER2(X,Y,F1) :- …
    Features-ER(X,Y,F1,F2) :- …
}
```

```
DOMAIN LP(X,Y) :- ….
{
    LP1(X,Y,F1) :- …
    LP2(X,Y,F1) :- …
    Features-LP(X,Y,F1,F2) :- …
}
```

```
ITERATE(*)
{
    INSERT EDGE(X,Y) :- FT-LP(X,Y,F1,F2), predict-LP(X,Y,F1,F2), confidence-LP(X,Y,F1,F2)
    IN TOP 10%
}
ITERATE(*)
{
    MERGE(X,Y) :- FT-ER(X,Y,F1,F2), predict-ER(X,Y,F1,F2), confidence-ER(X,Y,F1,F2)
    IN TOP 10%
}
```

Interleaving

```
DOMAIN ER(X,Y) :- ….
{
    ER1(X,Y,F1) :- …
    ER2(X,Y,F1) :- …
    Features-ER(X,Y,F1,F2) :- …
}
```

```
DOMAIN LP(X,Y) :- ….
{
    LP1(X,Y,F1) :- …
    LP2(X,Y,F1) :- …
    Features-LP(X,Y,F1,F2) :- …
}
```

```
ITERATE(*)
{
    INSERT EDGE(X,Y) :- FT-LP(X,Y,F1,F2), predict-LP(X,Y,F1,F2), confidence-LP(X,Y,F1,F2
    IN TOP 10%

    MERGE(X,Y) :- FT-ER(X,Y,F1,F2), predict-ER(X,Y,F1,F2), confidence-ER(X,Y,F1,F2)
    IN TOP 10%
}
```

Real-world Experiment

- 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
- Incremental: 28 minutes. Recompute: 42 minutes

```
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

Related Work

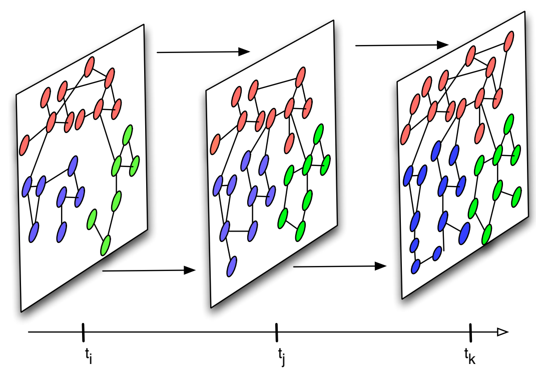
- Dedupalog [Arasu et al., ICDE 2009]: Datalog-based entity resolution
  - User defines hard and soft rules for deduplication
  - System satisfies hard rules and minimizes violations to soft rules when deduplicating references
- Swoosh [Benjelloun et al., VLDBJ 2008]:
  - Generic Entity resolution
    - Match function for pairs of nodes (based on a set of features)
    - Merge function determines which pairs should be merged
- Dyna: Extending Datalog for Modern AI [Eisner and Filardo, 2011]
  - High-level programming language for specifying NLP tasks
  - Many similarities to Datalog

Outline

- Overview
- Declarative Graph Cleaning
- Historical Graph Data Management
- Distributed Management of Dynamic 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"*

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)

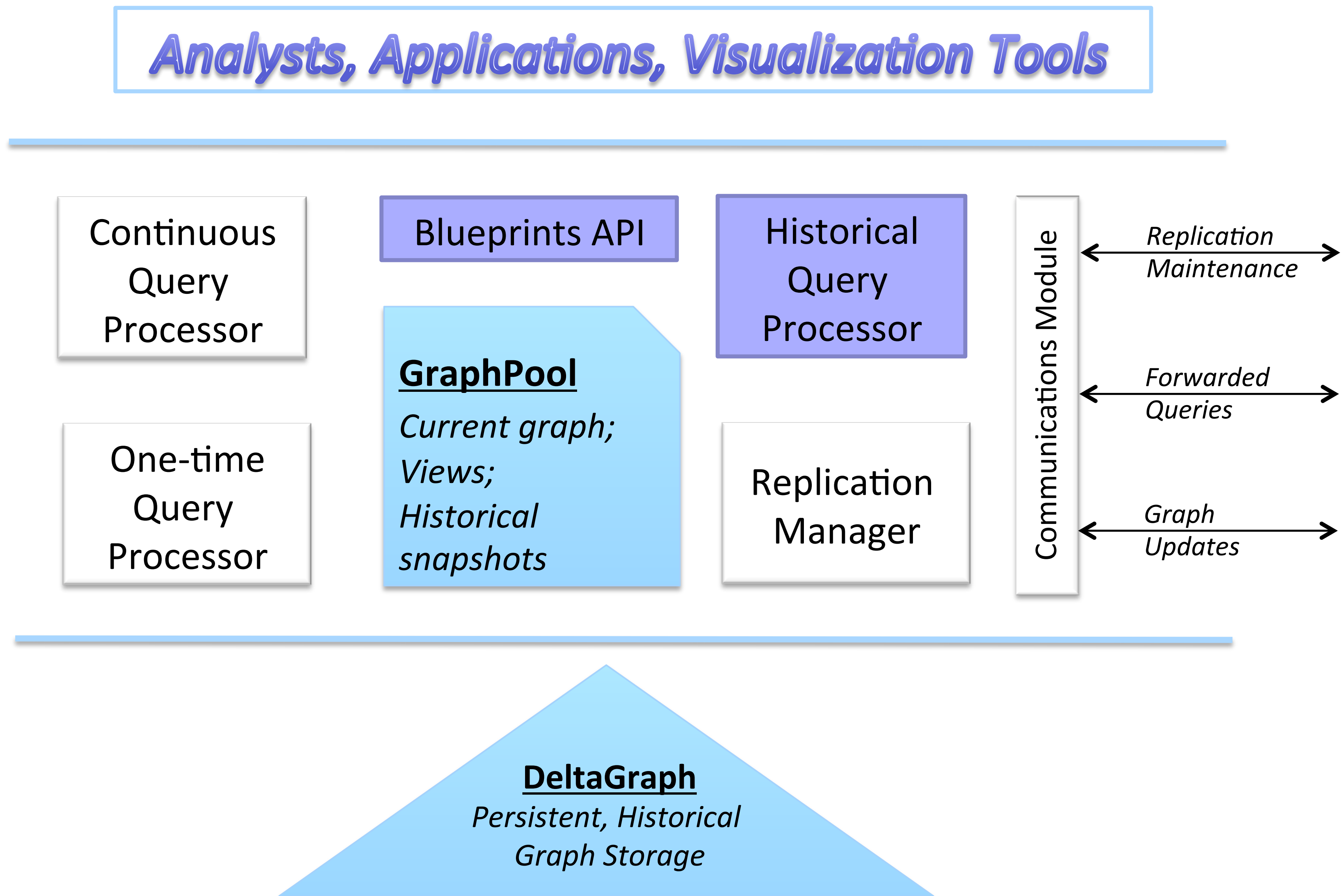
Prior Work

- 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]
    - Optimal storage, optimal (logarithmic) updates for managing interval data
    - Retrieval in the size of the retrieved graph
  - External Segment Trees [Blakenagal and Guting, 1994]
    - Optimal retrieval, but higher storage requirements
  - Snapshot index [Slazberg and Tsotras, 1999]
    - Optimal for *transaction-time* databases
  - Copy + Log
    - Maintain some snapshots explicitly, and keep chains of events between them

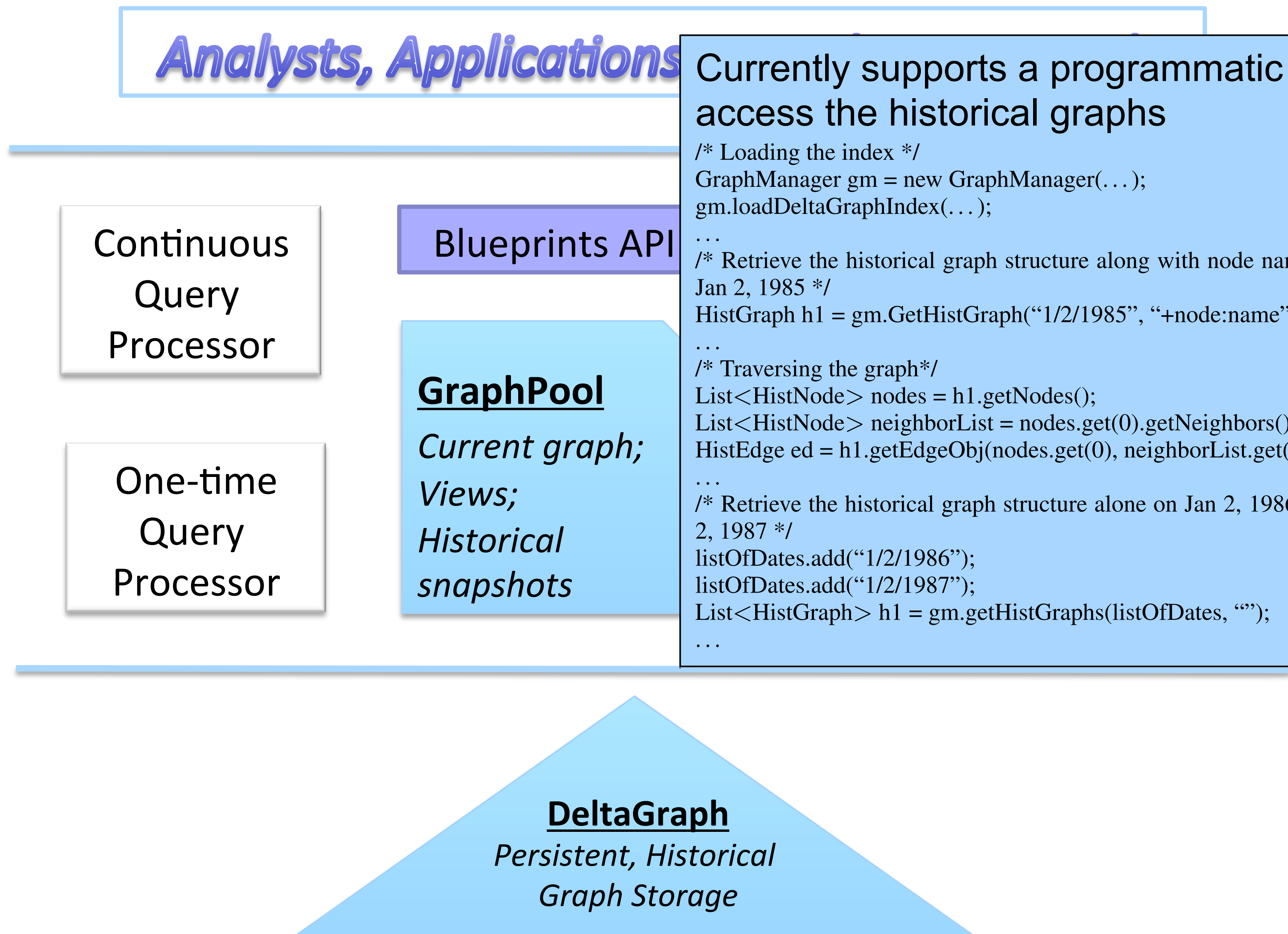
Prior Work: Limitations

- No flexibility or tunability
  - Would like to control the distribution of snapshot retrieval times, at run time
- No support for multi-point queries
- Not easy to support parallel retrieval/processing
- No support for retrieving portions of the network
- Would like to support different storage backends
  - Most prior techniques primarily optimized for disks

System Architecture



System Architecture



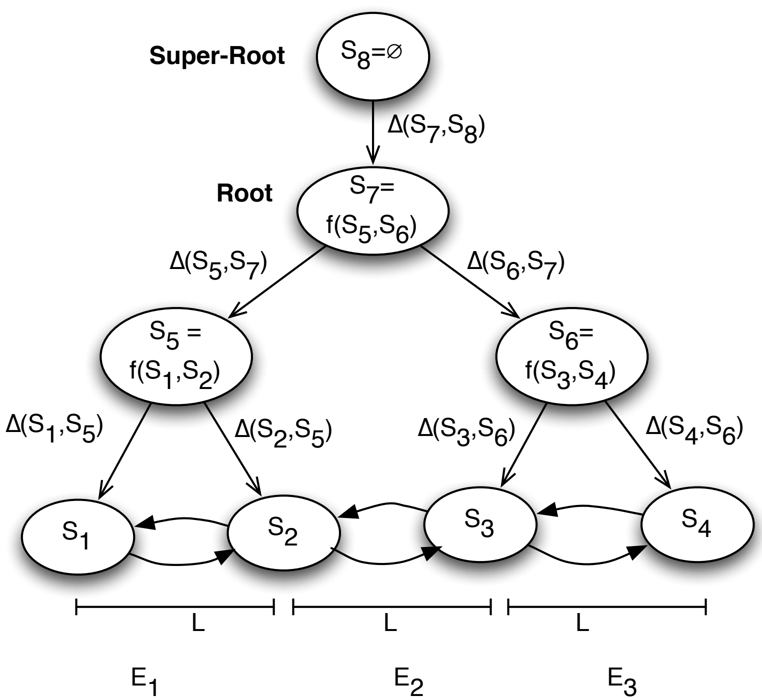
Currently supports a programmatic API to access the historical graphs

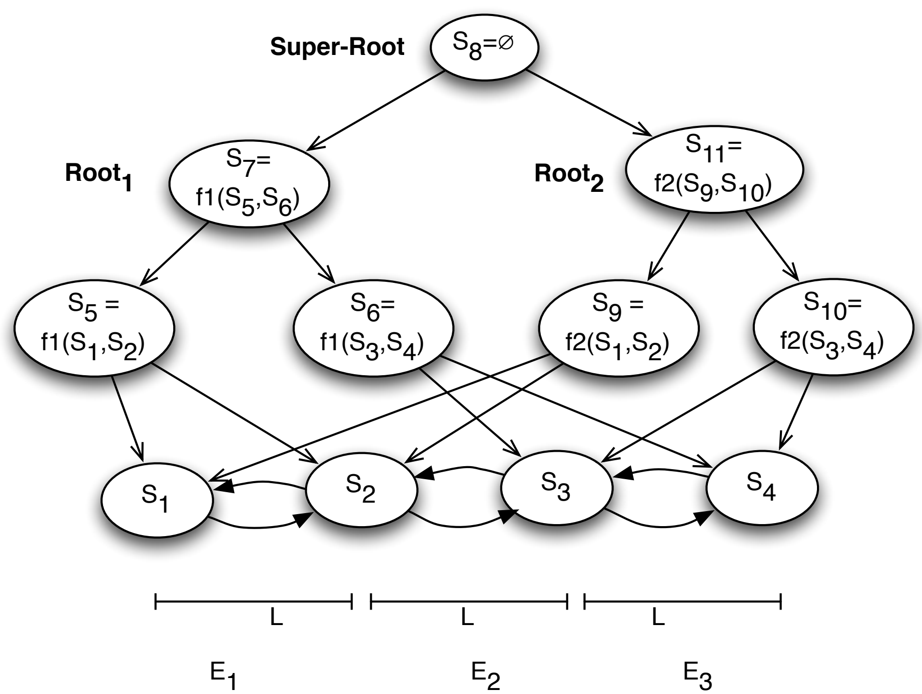
```
/* Loading the index */
GraphManager gm = new GraphManager(. . . );
gm.loadDeltaGraphIndex(. . . );
. . .
/* Retrieve the historical graph structure along with node names as of Jan 2, 1985 */
HistGraph h1 = gm.GetHistGraph(“1/2/1985”, “+node:name”);
. . .
/* Traversing the graph*/
List<HistNode> 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, “”);
. . .
```

DeltaGraph

- Hierarchical index structure with (logical) snapshots at the leaves
- Only the *edge deltas* stored explicitly
- Key parameter: *differential function (f, f1, f2)*
- Can have multiple hierarchies within the same structure

$\Delta(S_i, S_j) = S_j - S_i$



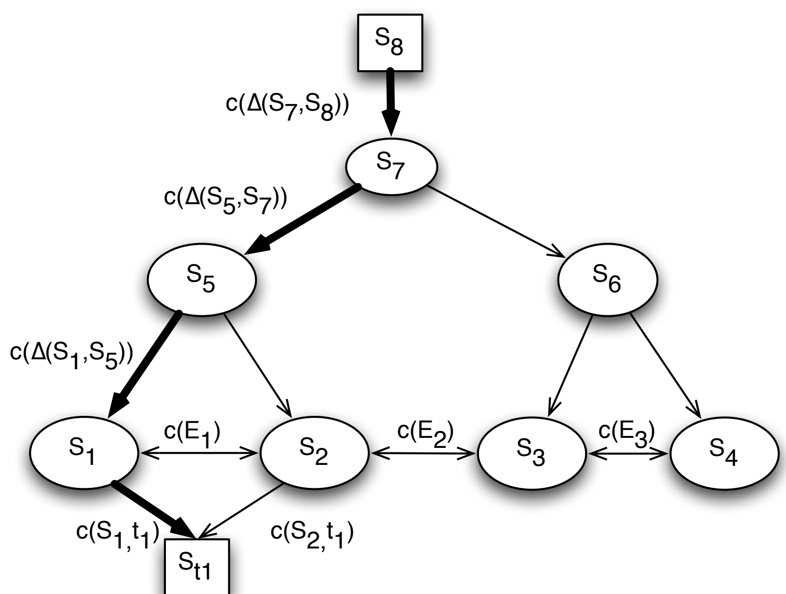


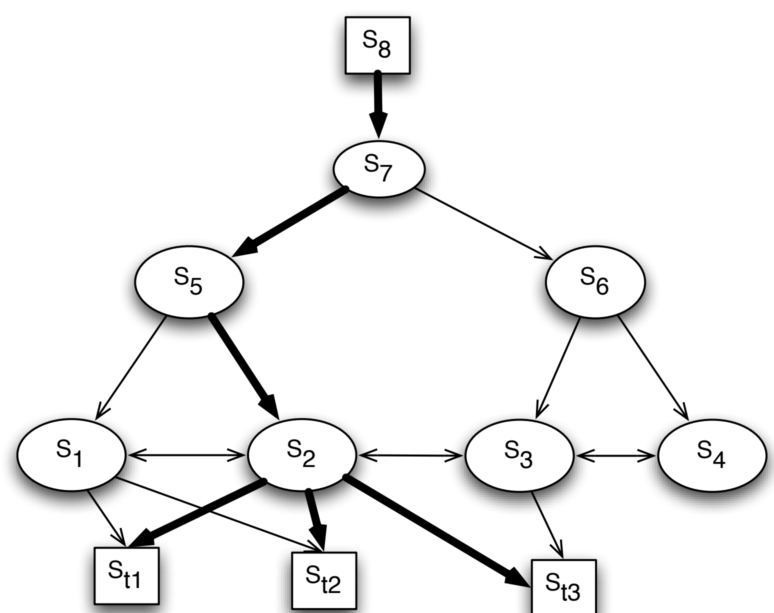
DeltaGraph Storage

- Deltas stored in a *key-value* store
  - Currently using disk-based *Kyoto Cabinet*
- Each edge delta split into multiple smaller deltas
  - Vertically by columns: To retrieve only some attributes
  - Horizontally by nodes: To facilitate distributed processing, and to speed up construction
- The *skeleton* maintained in memory
  - Expected to be small – the deltas are usually large to take advantage of compression and to reduce the number of I/Os
- Memory materialization
  - Basic idea: Explicitly materialize a snapshot in memory
    - “Current graph” treated as materialized (assuming an online system)
  - In the DeltaGraph, add an edge with cost 0 from the root
  - Enables much flexibility in reducing the snapshot retrieval costs

Snapshot Queries

- Single point: Lowest weight Path from Root
  - Edge is associated with several different weights for different attributes
- Multi-point: Lowest weight Steiner Tree from Root
  - Use the standard 2-approximation for this purpose
- Similar techniques for other types of more complex queries involving *time-expressions*





Differential Functions

- Choice of differential function greatly influences the properties
- Many functions of interest

Name	Description
Intersection	$f(a,b,c...) = a \cap b \cap c\ldots$
Union	$f(a,b,c...) = a \cup b \cup c...$
Skewed	$f(a, b) = a + r.(b - a), 0 \le r \le 1$
Right Skewed	$f(a, b) = a \cap b + r.(b - a \cap b), 0 \le r \le 1$
Left Skewed	$f(a, b) = a \cap b + r.(a - a \cap b), 0 \le r \le 1$
Mixed	$f(a,b,c...) = a + r_1.(\delta_{ab} + \delta_{bc} ...) - r_2.(\rho_{ab} + \rho_{bc}...), 0 \le r_2 \le r_1 \le 1$
Balanced	$f(a,b,c...) = a + 0.5(\delta_{ab} + \delta_{bc} ...) - 0.5(\rho_{ab} + \rho_{bc} ...)$
Empty	$f(a,b,c...) = \varnothing$

Analysis

- Model of graph dynamics
  - $G_{|E|}$: Graph after $|E|$ events
  - Assume a constant rate of inserts and deletes
    - Not equivalent to assuming constant rate of change/time
- Summary of results
  - Balanced function balances the retrieval times at the expense of higher storage requirements
  - Space requirements
    - Interval trees: O(|E|)
    - Segment trees: O(|E| log|E|)
    - DeltaGraph: Somewhere between O(|E|) and O(|E| log N)
      - Depending on the differential function, arity, and graph dynamics
      - *N* = Number of leaves

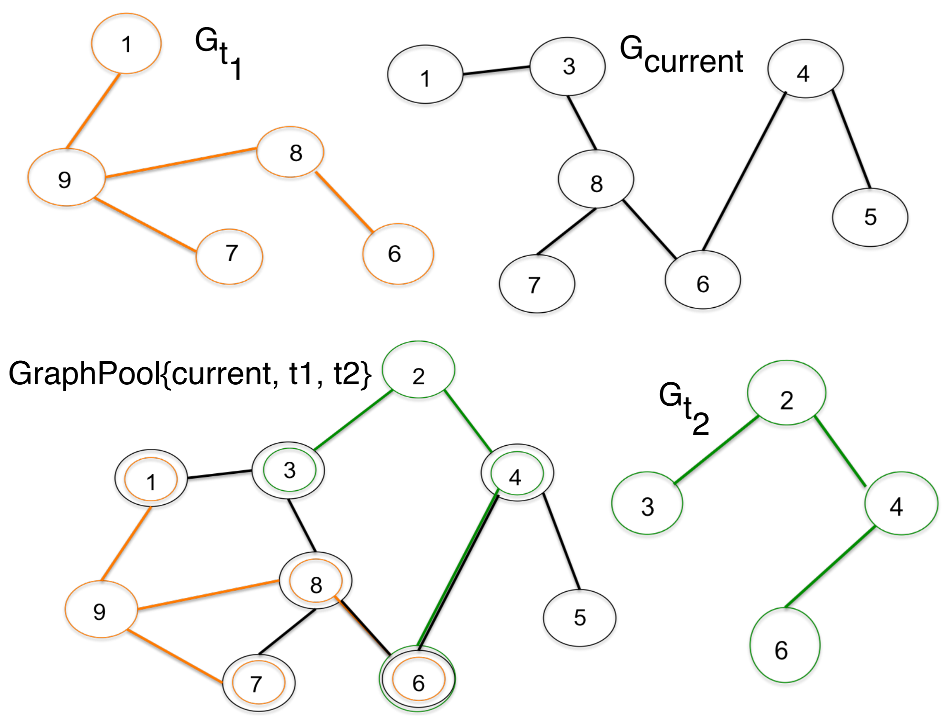
Some More Details

- DeltaGraph Construction
  - Bottom-up: Similar to the construction of a bulkloaded B+-tree
  - Construction parameters:
    - Evetlist size: L, Arity: k
    - Differential Function: f()
    - Partitioning of the nodes
  - Construction algorithm memory intensive
    - Need to do in a partitioned fashion to handle large graphs
    - Details in the paper

- Choosing what to materialize
  - Current approach is to materialize one or two of the top levels
  - Investigating approaches based on *facility location*

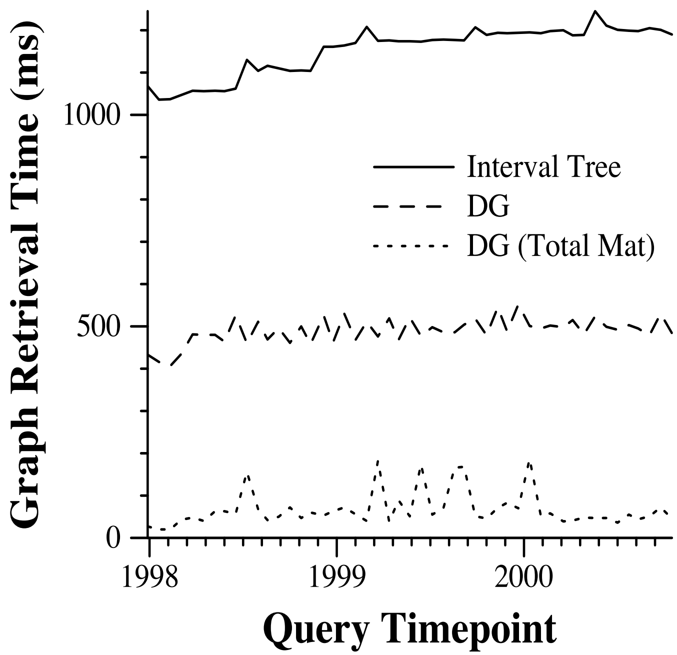
GraphPool

- Goal: Store many graphs in memory in an overlaid fashion
  - To minimize memory consumption
  - To reduce retrieval cost by using bitmaps to encode differences

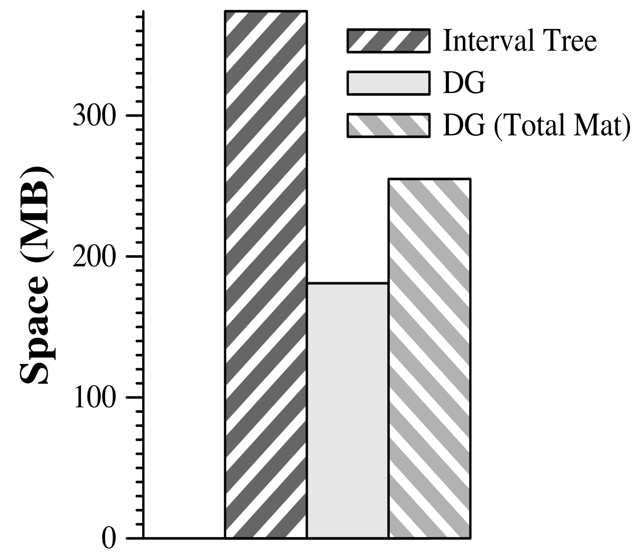


Empirical Results

- DeltaGraph vs In-Memory Interval Tree



(a) Performance: Dataset 2a

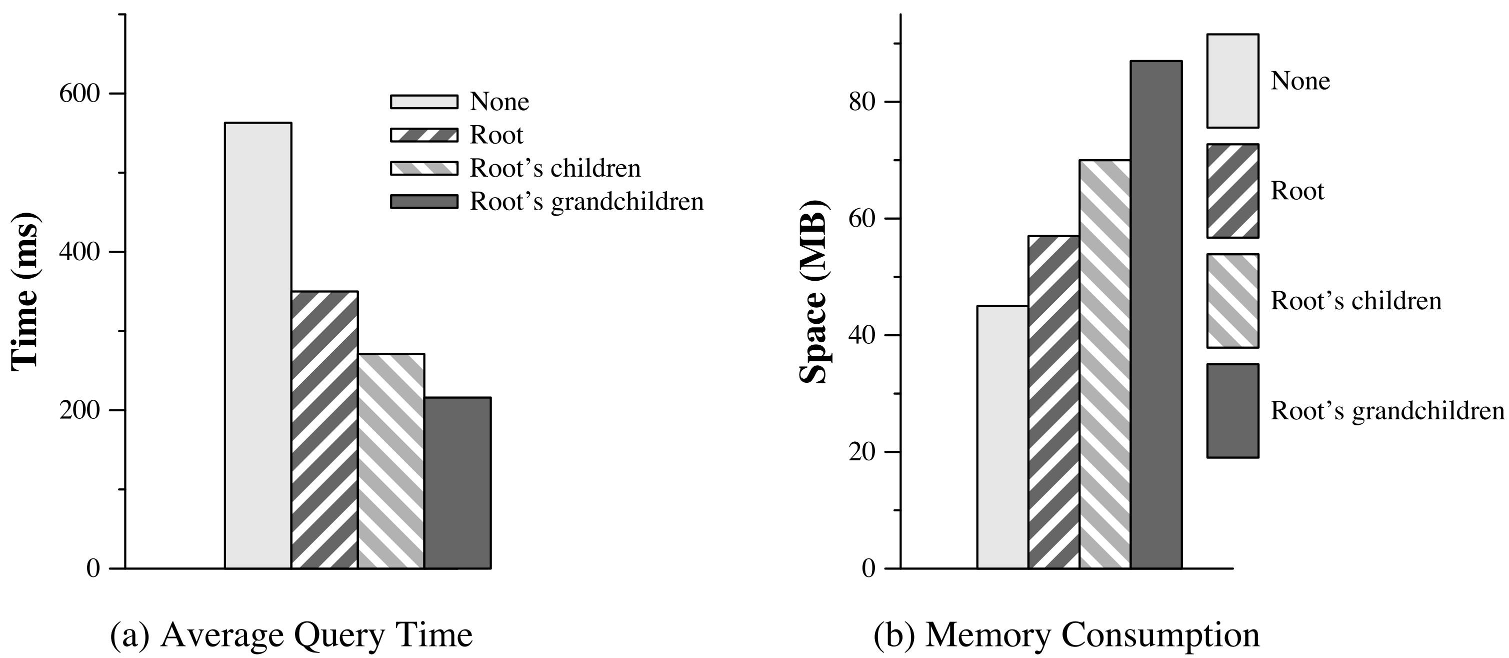


(b) Memory: Dataset 2a

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

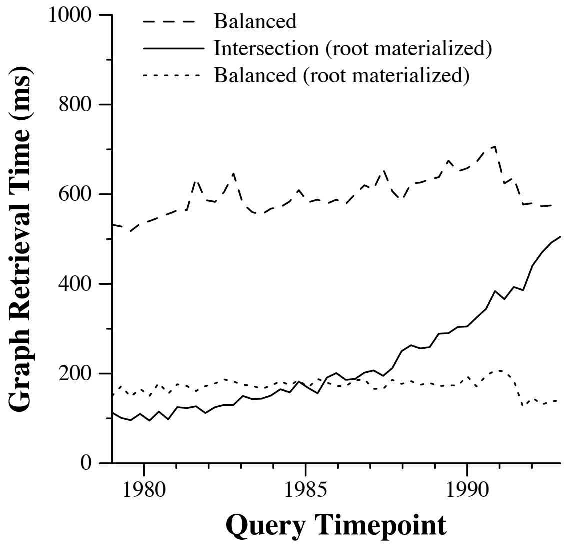
Empirical Results

- Effect of Materialization

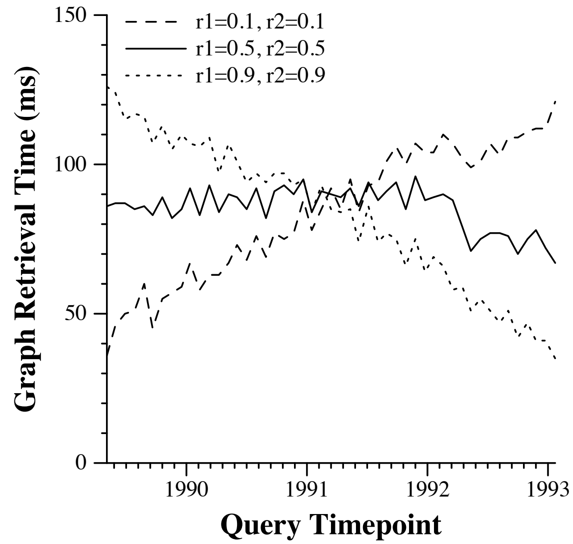


Empirical Results

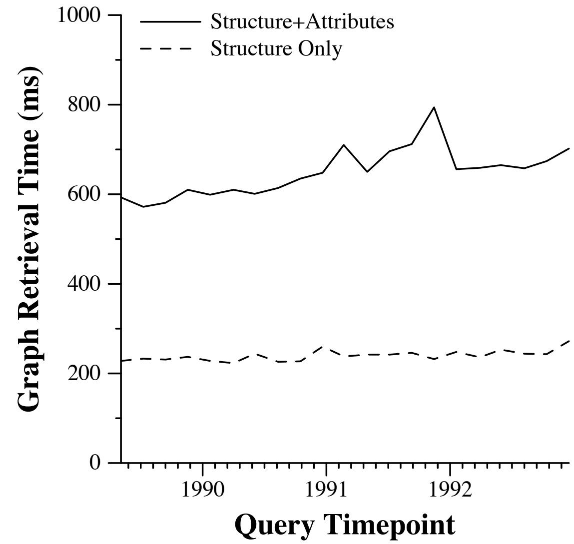
- Differential Functions, Attributes



(a) Int vs Bal (Dataset 1a)



(b) Different mixed functions (Dataset 2c)

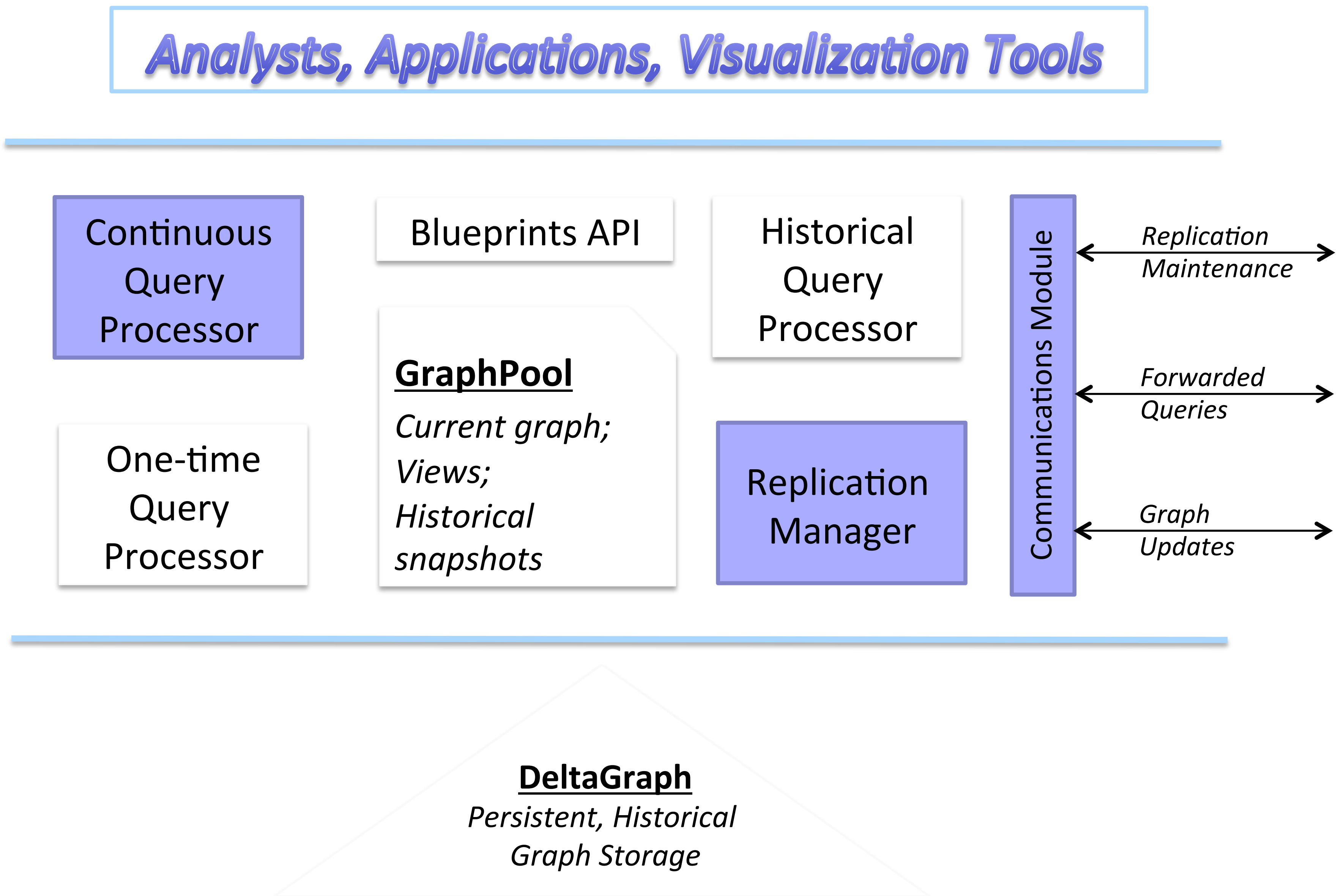


(c) Retrieval with and without attributes (Dataset 2c)

Outline

- Overview
- Declarative Graph Cleaning
- Historical Graph Data Management
- Distributed Management of Dynamic Graphs
- Conclusions

System Architecture



Motivation

- Graph partitioning hard to do effectively
  - Random partitioning typically results in large edge cuts
    - → Distributed traversals to answer queries leading to high latencies
  - Sophisticated partitioning techniques often do not work either
    - Clean, disjoint partitionings often do not exist
    - Hard to scale (although some recent work)
    - Not appropriate for highly dynamic environments

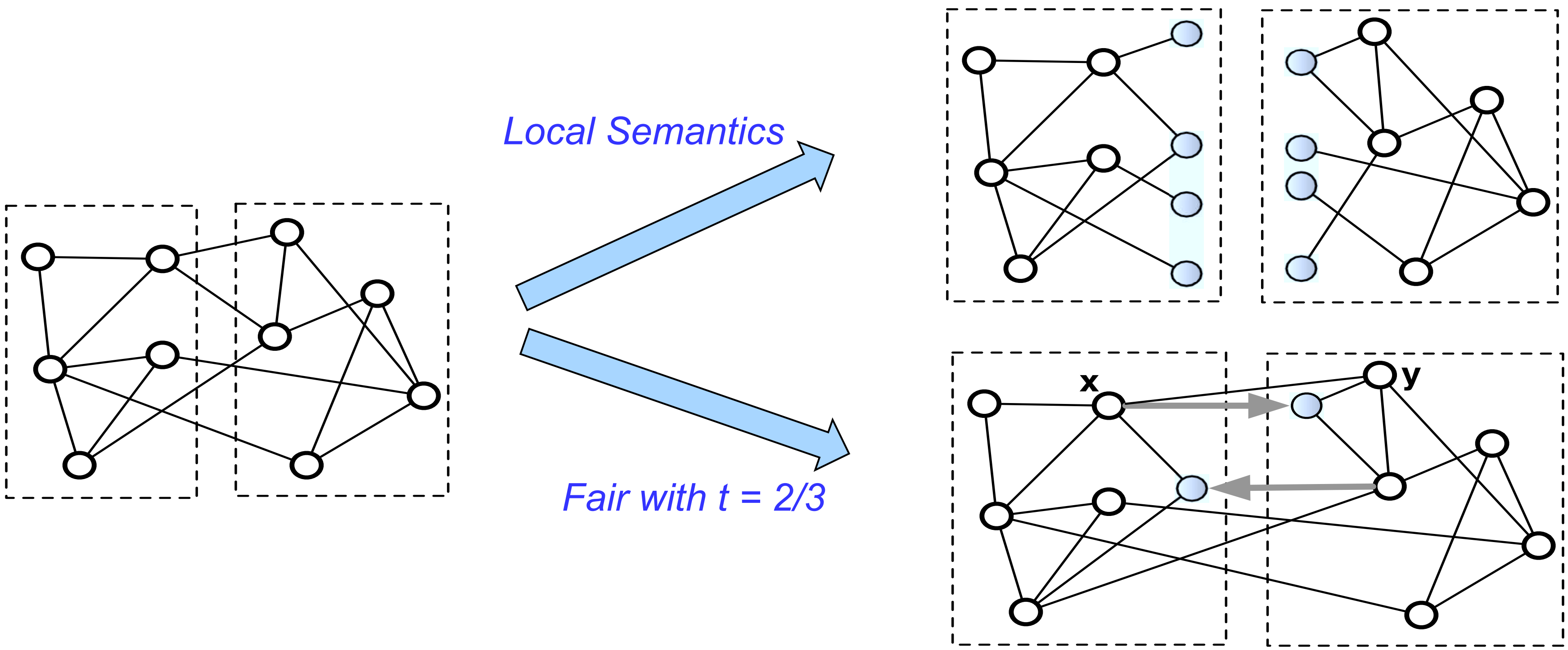
- We employ an aggressive replication approach to reduce latencies
  - How to choose what to replicate? – A new “fairness” criterion
  - Eager or Lazy replication? – Fine-grained access pattern monitoring

Prior Work

- Pujol et al. [SIGCOMM’11]
  - Local semantics: For every node, every neighbor is replicated locally (if not already present)
  - High replication overhead
  - Similar approach proposed by Huang et al. [VLDB’11]
- Adaptive replication [Wolfson et al., TODS’97]
  - Monitor access frequencies
  - Focused on tree communication networks
- Feed delivery [Silberstein et al., SIGMOD’10]
  - Similar problem in a publish-subscribe setting
  - No reciprocal relationship between publishers and subscribers

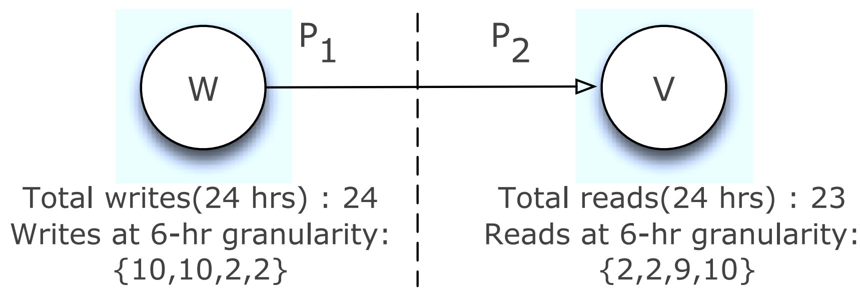
Our Approach

- 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

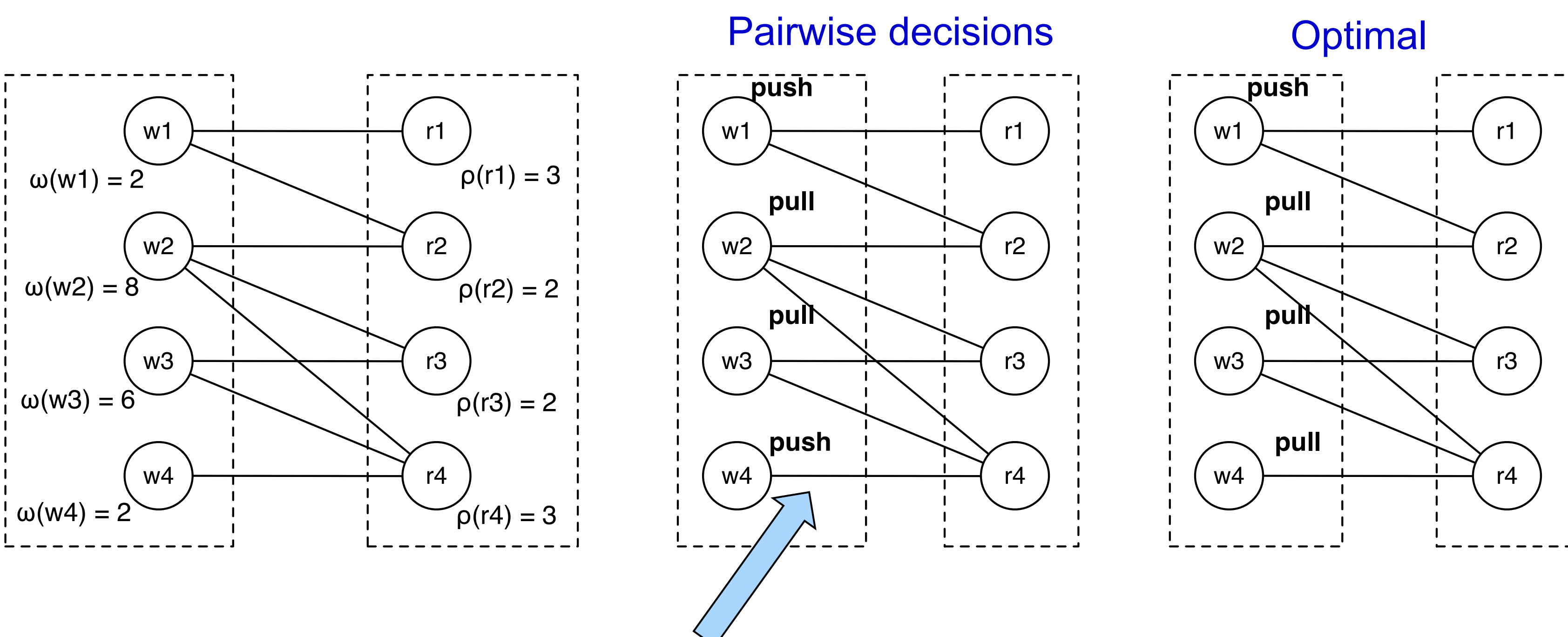
- Key idea 2
  - Exploit patterns in the read/write access frequencies



  - 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

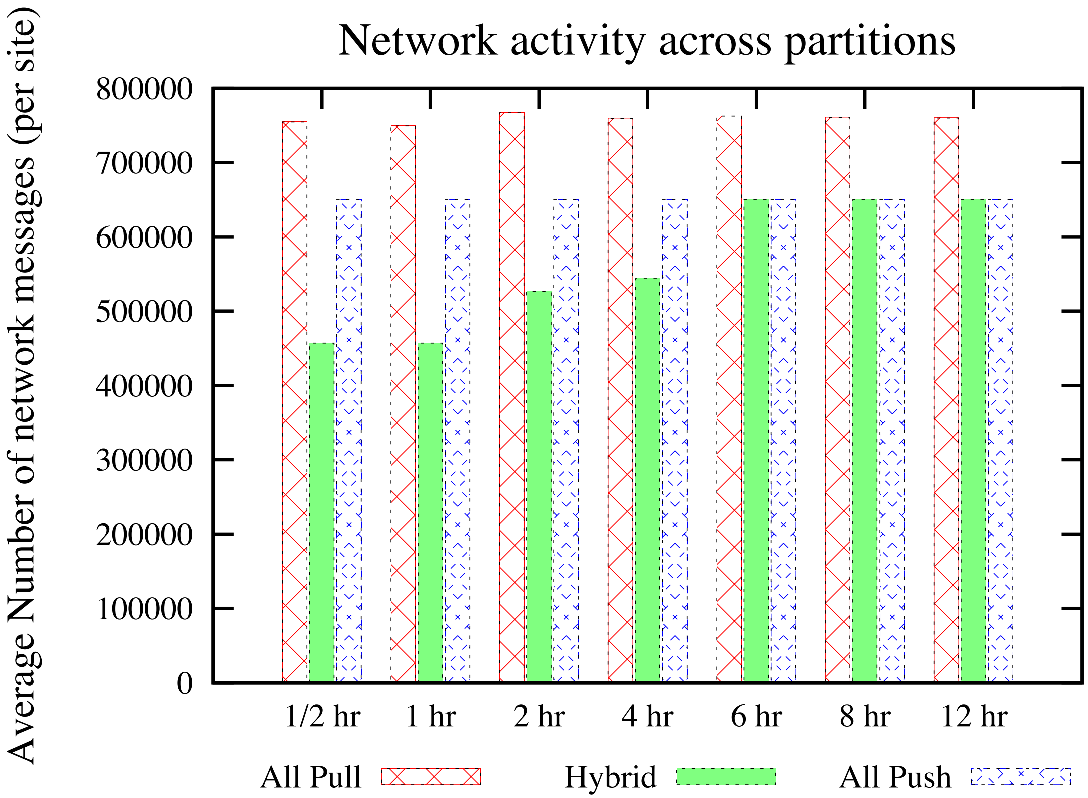


No point in pushing w4 – r4 will have to pull from the partition anyway

Some more details

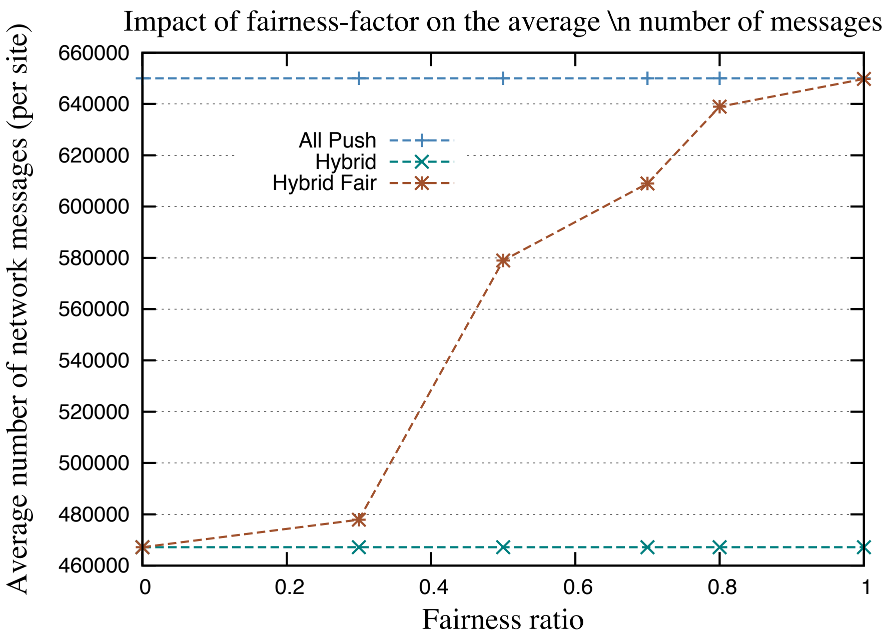
- Hash partitioning
  - The basic partitioning is done using standard hash-based techniques
  - Better load balancing, and much simpler routing logic
- Clustering
  - Infeasible to make replication decisions on a per node basis
  - Instead cluster nodes based on the *read/write frequencies*
  - Significantly reduces the metadata needed to implement replication decisions
- Decentralized algorithms
  - Decisions made/re-evaluated independently at each partition
- Implementation
  - Use CouchDB key-value store for storing the data
  - Leverage upon the replication support built-in

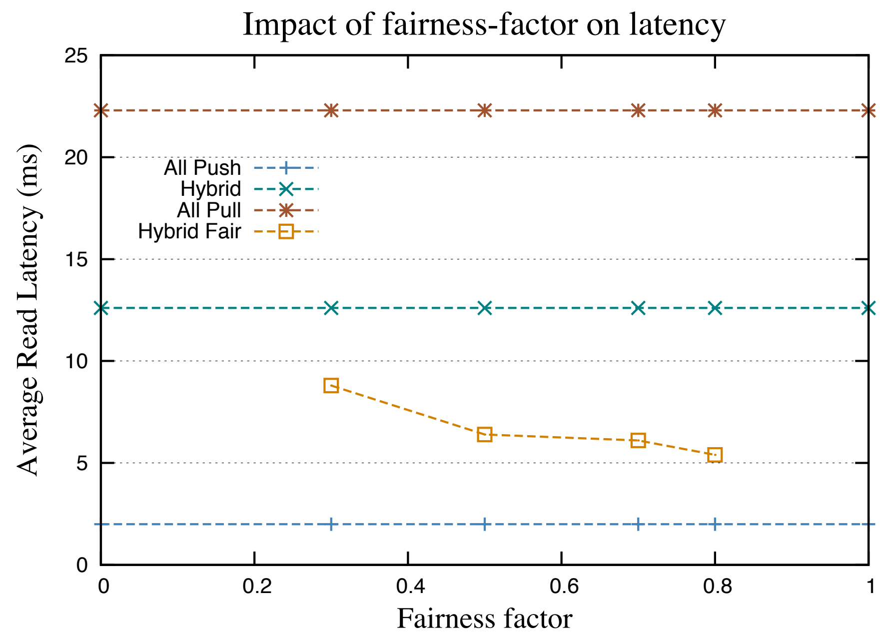
Empirical Results



Fine-grained, adaptive decisions can result in substantial savings in number of messages

Empirical Results





Fairness factor can be used to effectively trade-off latencies and replication cost

Outline

- Overview
- Declarative Graph Cleaning
- Historical Graph Data Management
- Distributed Management of Dynamic Graphs
- Conclusions

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 temporal query languages for graph querying
  - Replication and pre-computation for continuous queries
  - Efficiently supporting distributed graph analytics

Thank you !!