Scalable Platforms for Graph Analytics and Collaborative Data Science

Amol Deshpande

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

These slides at: http://go.umd.edu/w.pdf

Joint work with many students and collaborators

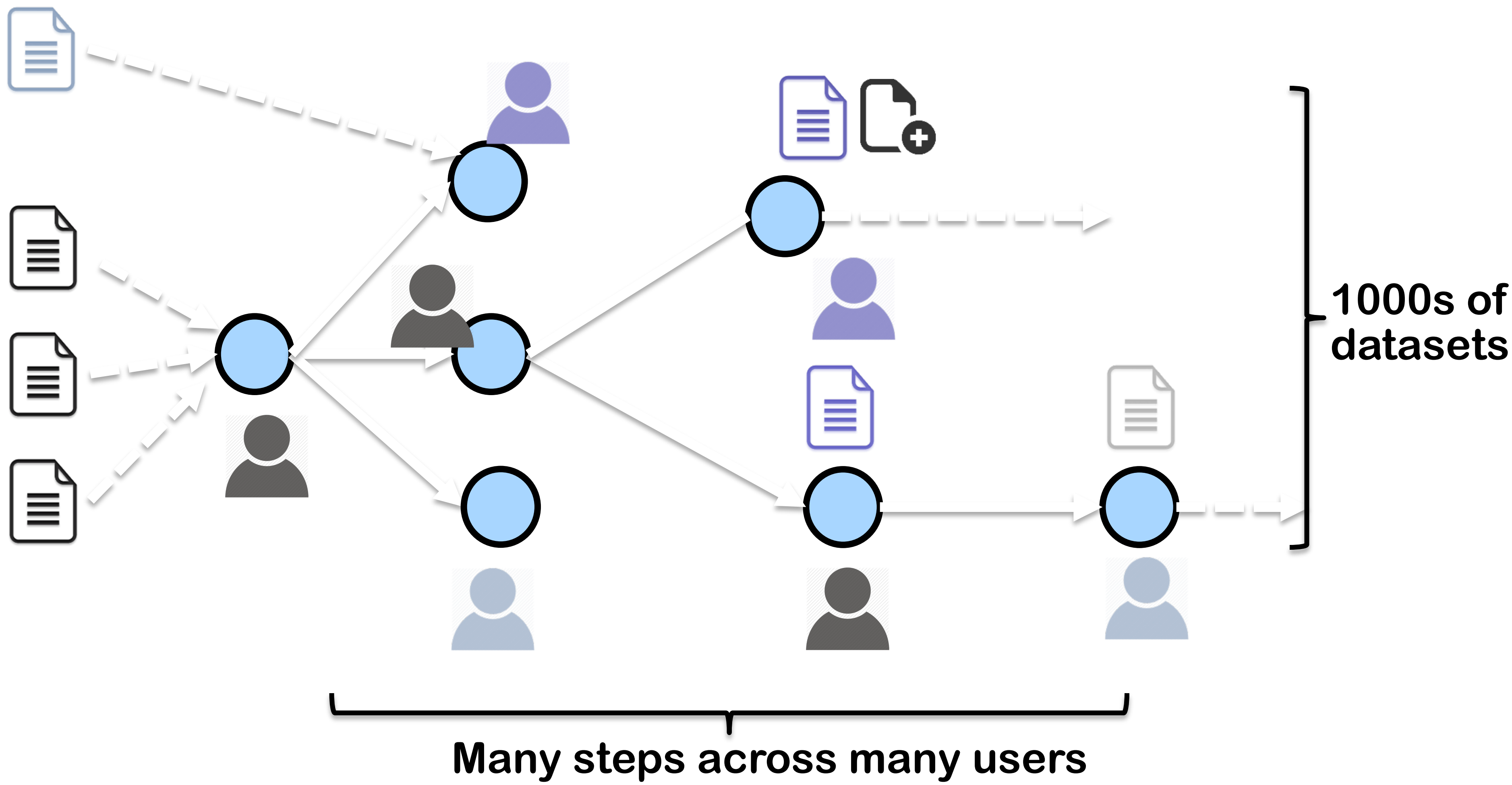
Outline

- DataHub: A platform for collaborative data science
- GraphGen: Graph Analytics on Relational Databases
  - Motivation
  - System Overview
  - Condensed Representations for Large Graphs
  - Experiments

These slides at: http://go.umd.edu/w.pdf

Collaborative Data Science

- Widespread use of "data science" in many many domains



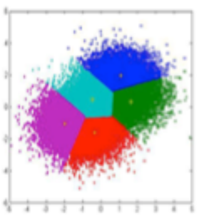
A typical data analysis workflow

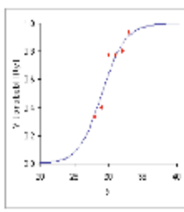
Collaborative Data Science

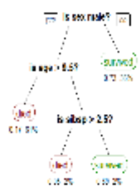
- Widespread use of "data science" in many many domains
- Increasingly the "pain point" is managing the *process,* especially during collaborative analysis
  - Many private copies of the datasets ➔ Massive redundancy
  - No easy way to keep track of dependencies between datasets
  - Manual intervention needed for resolving conflicts
  - No efficient organization or management of datasets
  - No easy way to do "provenance", i.e., find reasons for an action
  - No way to analyze/compare/query versions of a dataset
- Ad hoc data management systems (e.g., Dropbox) used
  - Much of the data is unstructured so typically can't use DBs
  - Scientists/researchers/analysts are pretty much on their own

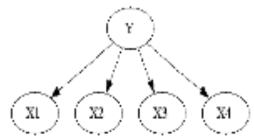
Model Lifecycle Management

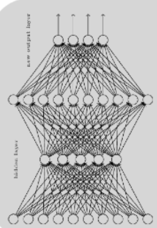
- "Models" are an integral part of data science
- Traditional simple models → today's complex BIG models

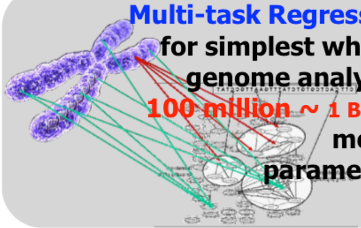

K-
means


Logistic
regression


Decision
trees


Naive Bayes


Google Brain
Deep Learning
for images:
1~10 Billion
model parameters

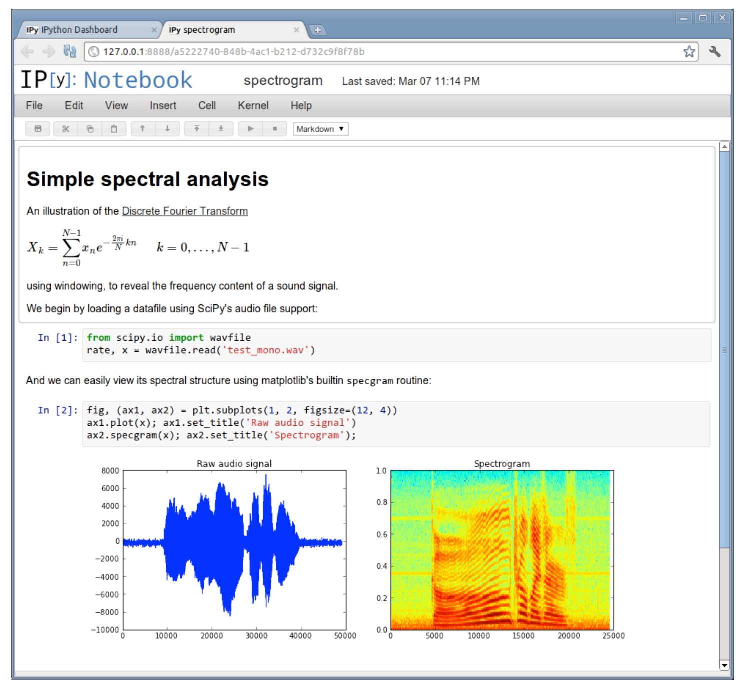

Multi-task Regression
for simplest whole-genome analysis:
100 million ~ 1 Billion
model parameters


Topic Models
for news article analysis:
Up to 1 Trillion
model parameters

Collaborative filtering
for Video recommendation:
1~10 Billion
model parameters

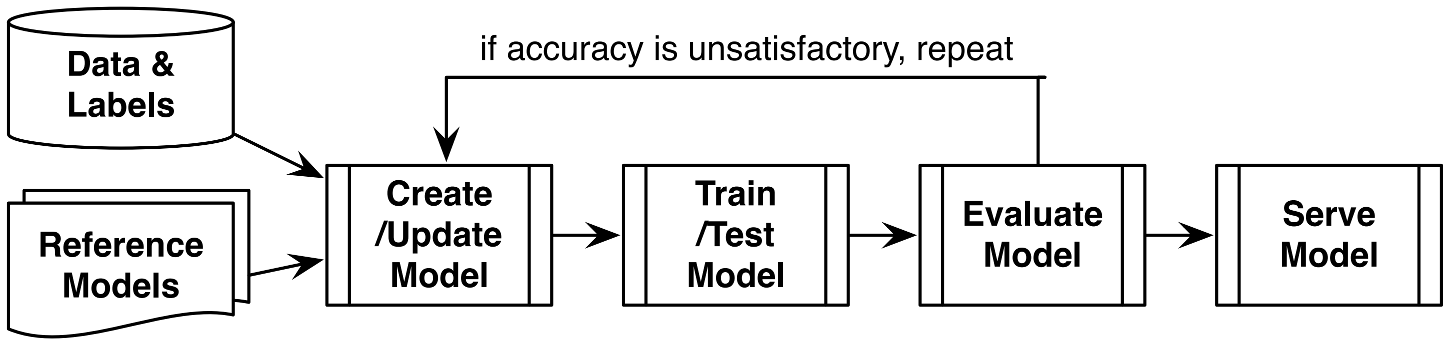
NETFLIX

KDD 15 © Eric Xing @ CMU, 2015 10



Often packaged together with results

Challenges



- What parameter did we use to get the precision?
- How do I know which data corresponds to which model?
  - e.g., IPython notebooks don't usually keep the "data"
- How to compare different "pipelines", identify bugs
- Issues during deployment
  - Monitor model performance, detect problems or anomalies, etc.
- Focus of most current work on scalability, training, etc.
  - Critical history is transient and not captured

DataHub: A Collaborative Data Science Platform

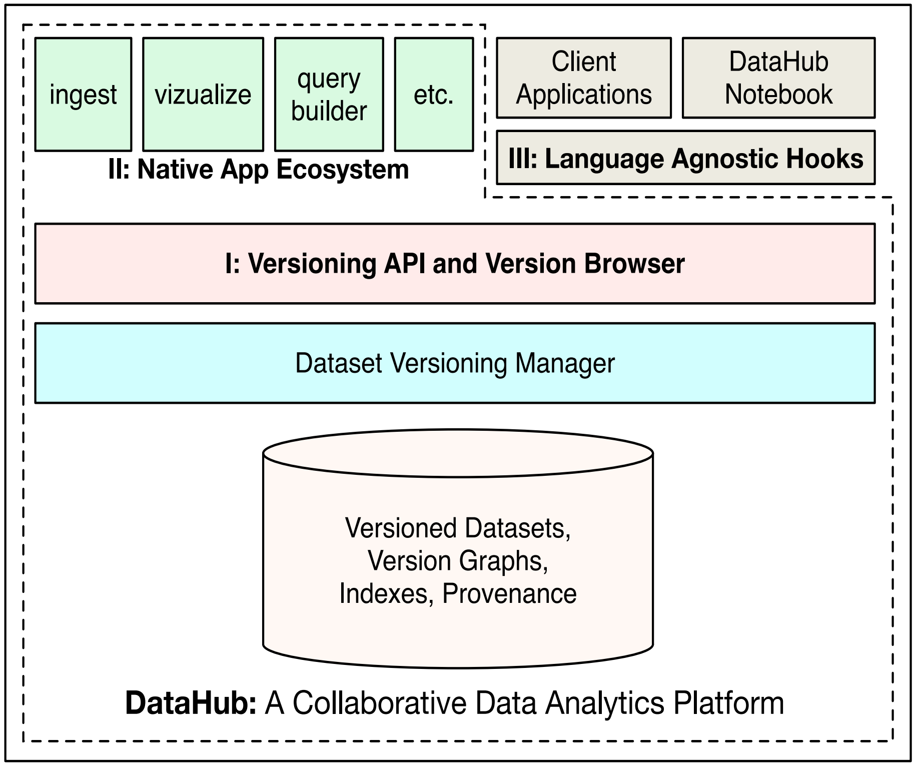
Joint work with:
Sam Madden (MIT)
Aditya Parameswaran (UIUC)

- a dataset management system – import, search, query, analyze a large number of (public) datasets

- a dataset version control system – branch, update, merge, transform large structured or unstructured datasets

- a provenance database system – capture provenance & other metadata, and support analysis/introspection

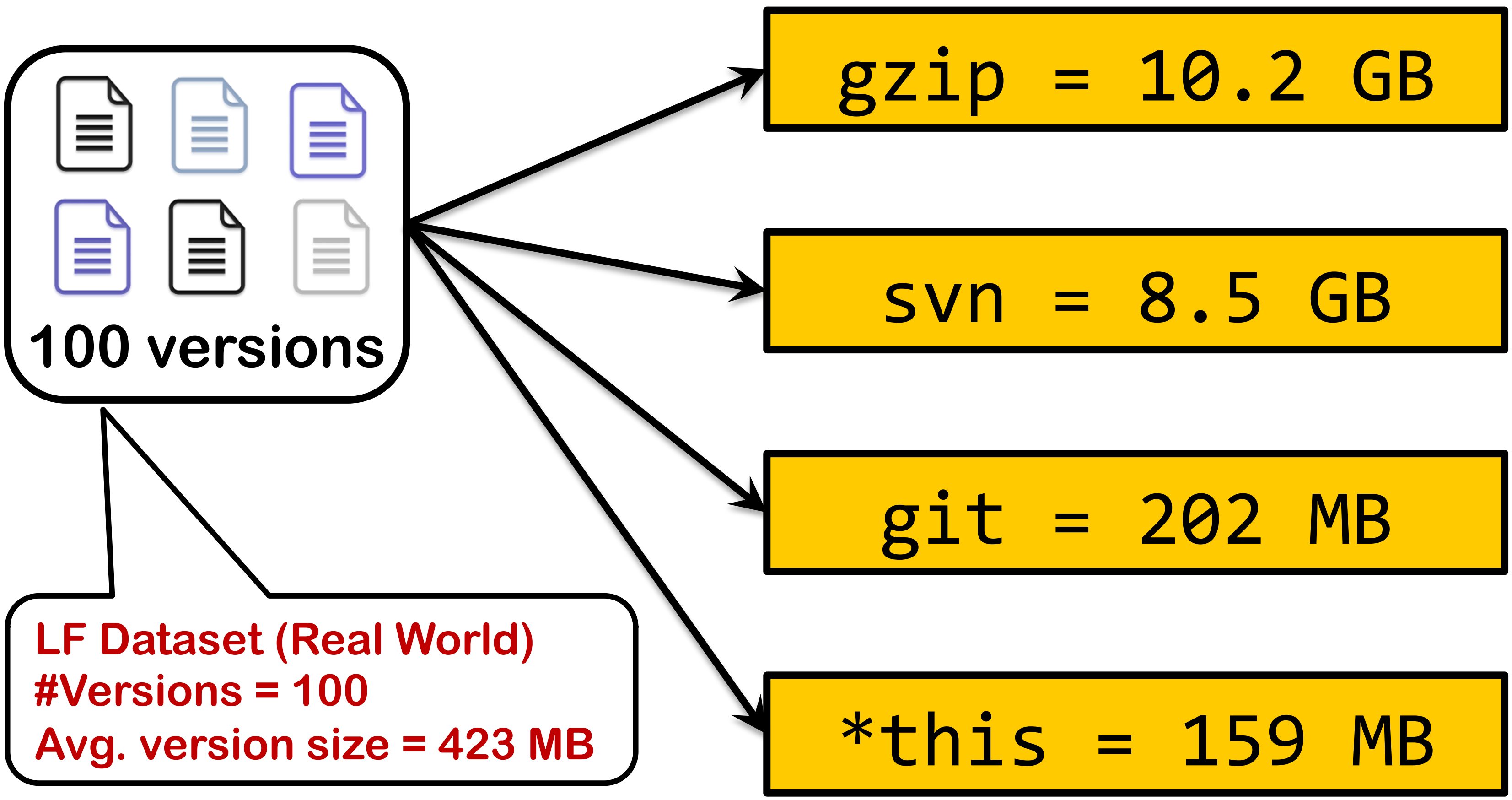
- an app ecosystem and hooks for external applications (Matlab, R, iPython Notebook, etc)



DataHub Architecture

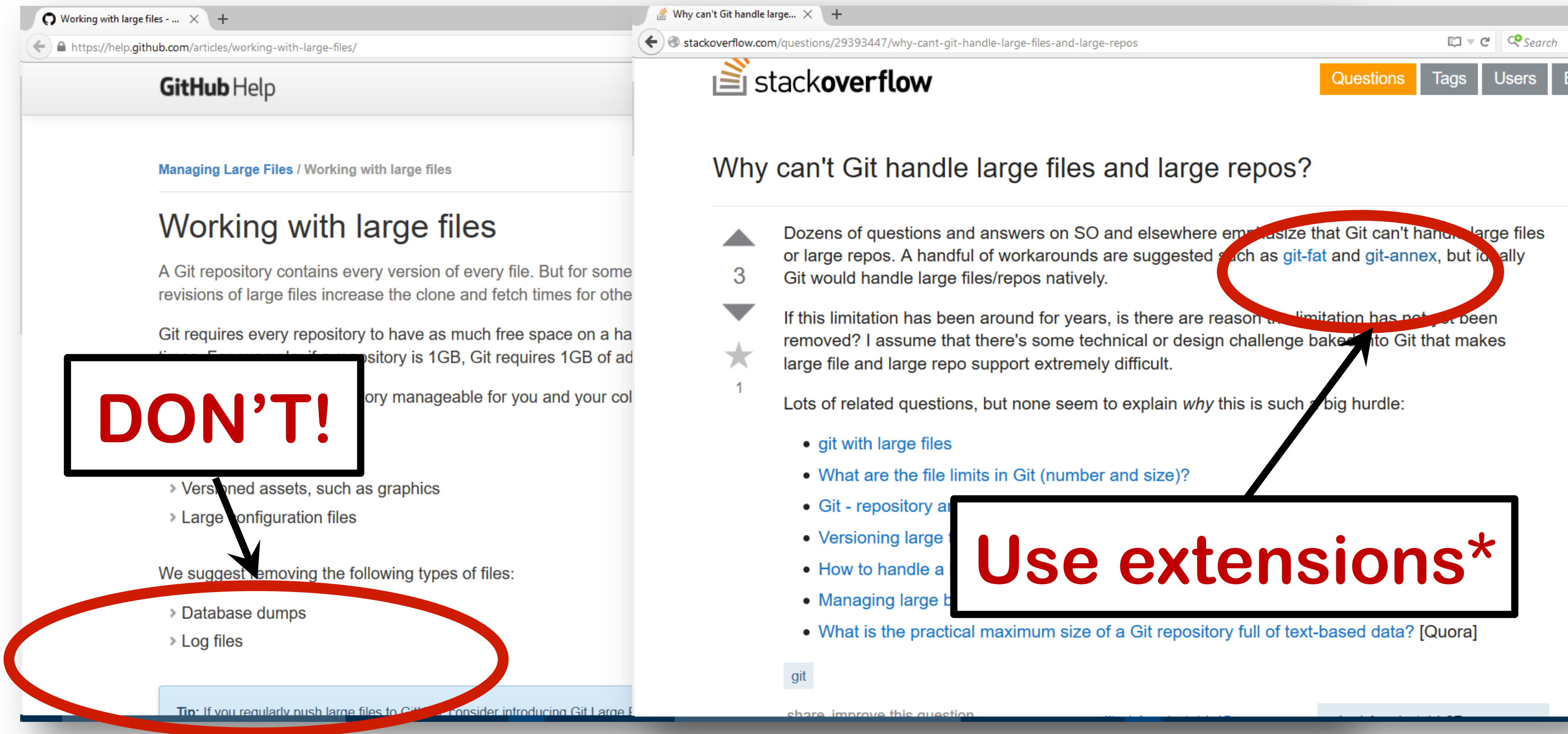
Can we use Version Control Systems (e.g., Git)?

- ✗ **No, because they typically use** **fairly simple algorithms** **and are optimized to work for code-like data**



Can we use Version Control Systems (e.g., Git)?

- ✗ No, because they typically use fairly simple algorithms and are optimized to work for code-like data
- ✗ Git ends up using large amounts of RAM for large files

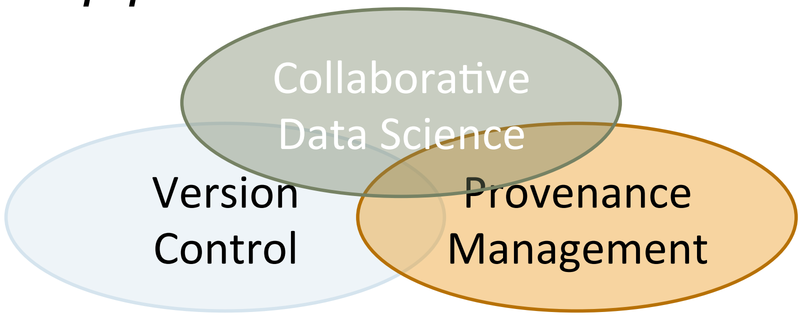


Can we use Version Control Systems (e.g., Git)?

- ✘ **No support for capturing rich metadata about the datasets and/or provenance information**

- ✘ **Primitive querying and retrieval functionalities**

- ✘ **No way to specify queries like:**
  - *identify all predecessor versions of version A that differ from it by a large number of records*
  - *rank a set of versions according to a scoring function*
  - *find the version where the result of an aggregate query is above a threshold*
  - *explain why the results of two similar pipelines are different*
  - *identify the source of an error*



Other Related Work

- Temporal databases are restricted to managing a linear chain of versions of relational data
- Recent work in scientific databases
  - Optimized for array-like data
  - Also largely a linear chain of versions
- "Deduplication" strategies in storage systems
  - Chunk files into blocks and store unique blocks
  - Works well if changes are localized
  - Focus primarily on archival storage minimization, ignore recreation costs
- Metadata/Provenance management systems
  - Much work, but insufficient adoption as yet

Summary of Ongoing Work

- Exploit overlap to reduce storage [VLDB'15,VLDB'16,*,*]
  - … while keeping retrieval costs low
  - … for different types of data (unstructured files, relational data, documents, and large NN models)
- System for managing and querying versioning and provenance information [TaPP'15, *]
  - … along with mechanisms to easily capture provenance
  - Prototype command-line-based provenance ingestion system, built on top of "git" and "Neo4j"
- A vertical for lifecycle management of deep learning models [*]

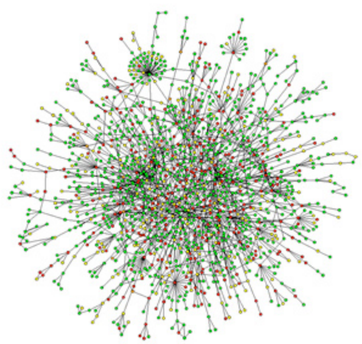
Outline

- DataHub: A platform for collaborative data science
- GraphGen: Graph Analytics on Relational Databases
  - Motivation
  - System Overview
  - Condensed Representations for Large Graphs
  - Experiments

These slides at: http://go.umd.edu/w.pdf

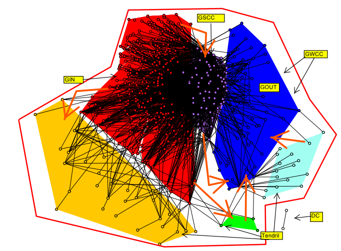
Graph Data

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



A protein-protein interaction network

Social networks



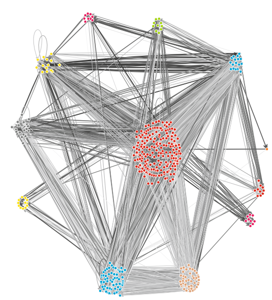
Federal funds networks

Knowledge Graph

World Wide Web

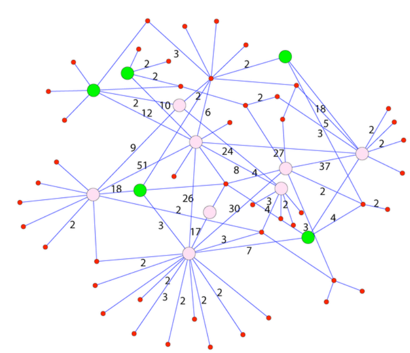
Citation networks

Communication networks



Financial transaction networks

Disease transmission networks



Stock Trading Networks

Wide Variety in Graph Queries/Analytics

Different types of "queries"

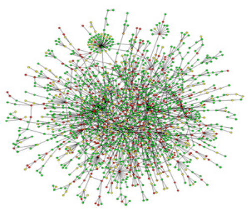
Subgraph pattern matching; Reachability; Shortest path; Keyword search; Historical or Temporal queries…

Continuous "queries" and Real-time analytics

Online prediction; Monitoring; Anomaly/Event detection

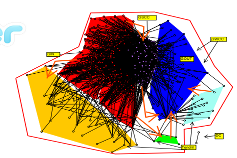
Batch analysis tasks

Centrality analysis; Community detection; Network evolution; Network measurements; Graph cleaning/inference



A protein-protein interaction network

Social networks



Federal funds networks

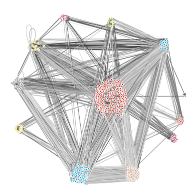
Knowledge Graph

World Wide Web

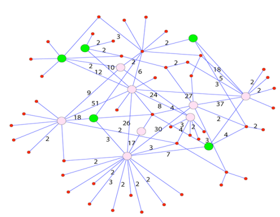
Citation networks

Communication networks

Disease transmission networks



Financial transaction networks



Stock Trading Networks

Machine learning tasks

Many algorithms can be seen as message passing in specially constructed graphs

Graph Data Management: State of the Art

- Graph analytics/network science tasks *too* varied
- Hard to build general systems like RDBs/Hadoop/Spark
  - What is a good abstraction to provide?
    - MapReduce? Vertex-centric frameworks? BSP?
    - Popular graph languages (SPARQL, Cypher) equivalent to SQL
  - No clear winners or widely used systems
  - Application developers largely doing their own thing
- Fragmented research topic with little consensus
  - Specialized graph databases (Neo4j), RDF Databases
  - Distributed batch systems (GraphX, Giraph), HPC Single-memory Engines (Ligra, GreenMarl, X-Stream)
  - Many specialized indexes, prototypes…

What we are doing

- Goal: A complete, function-rich system with unified declarative abstractions for graph queries and analytics
  - Declarative cleaning of noisy and imperfect graphs through link prediction and entity resolution [GDM'11, SIGMOD Demo'13]
  - Real-time continuous queries and anomaly detection over dynamic graphs [SIGMOD'12, ESNAM'14, SIGMOD'14, DEB'16]
  - Historical graph data management and temporal analytics [ICDE'13, SIGMOD Demo'13,EDBT'16]
  - Subgraph pattern matching and counting [ICDE'12, ICDE'14]
  - GraphGen: graph analytics over relational data [VLDB Demo'15, SIGMOD'17]
  - NScale: a distributed analysis framework [VLDB Demo'14, VLDBJ'15,NDA'16]

1. Where's the Data?

- Graph data management systems expect and manage *graph-structured* data, i.e., lists of nodes and edges
- Most data sits in RDBMSs and (increasingly) NoSQL stores
- Graphs must be **extracted** by identifying and connecting entities across the database

1. Example: TPC-H

LineItem

order_key	*part_key*
o1	*p1*
o1	*p2*
o2	*p1*
o2	*p3*
o3	*p1*
o3	*p2*
o3	*p2*

Customer

c_key	*name*

Orders

order_key	*customer_key*
o1	*c1*
o2	*c2*
o3	*c3*

⋈ On **order_key**

Which customer bought which product?

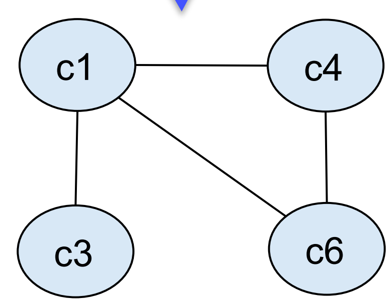
Orders ⋈ LineItem

c_key	*p_key*
c1	**p1**
c1	**p2**
c3	**p2**
c4	**p1**
c6	**p1**

⋈ On **p_key**

Which customers bought the same item?

cust1	*cust2*
c1	**c4**
c1	**c6**
c1	**c3**
c4	**c6**



1. Example: TPC-H

LineItem

order_key	part_key
o1	p1
o1	p2
o2	p1
o2	p3
o3	p1
o3	p2
o3	p2

Customer

c_key	name

Orders

order_key	customer_key
o1	c1
o2	c2
o3	c3

⋈ On **order_key**

Orders ⋈ LineItem

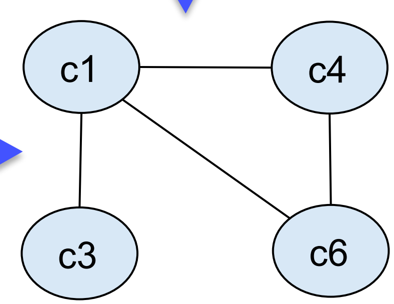
c_key	p_key
c1	**p1**
c1	**p2**
c3	**p2**
c4	**p1**
c6	**p1**

⋈ On **p_key**

Which customers bought the same item?

cust1	cust2
c1	**c4**
c1	**c6**
c1	**c3**
c4	**c6**

Edge weights based on geographical distance



1. Where's the Data?

- Graph data management systems expect and manage *graph-structured* data, i.e., lists of nodes and edges
- Most data sits in RDBMSs and (increasingly) NoSQL stores
- Graphs must be **extracted** by identifying and connecting entities across the database
- Must be done repeatedly as the underlying data changes
  - Tedious and time-consuming
- Also desirable to avoid having to use another data management system

1. Where's the Data?

- Efficiency challenge: Extracted graphs can often be orders-of-magnitude larger than original database
  - Homogeneous graphs (over the same set of entities) invariably require at least one self-join on a non-key
  - DBLP Dataset: 8.6M author-publication table → 43M edges in the co-authorship graph
    - Connecting authors with papers at the same conference = 1.8 B edges
- Even if the final graph is small, database query optimizers unable to optimize these queries well
  - High selectivity errors

1. Where's the Data?

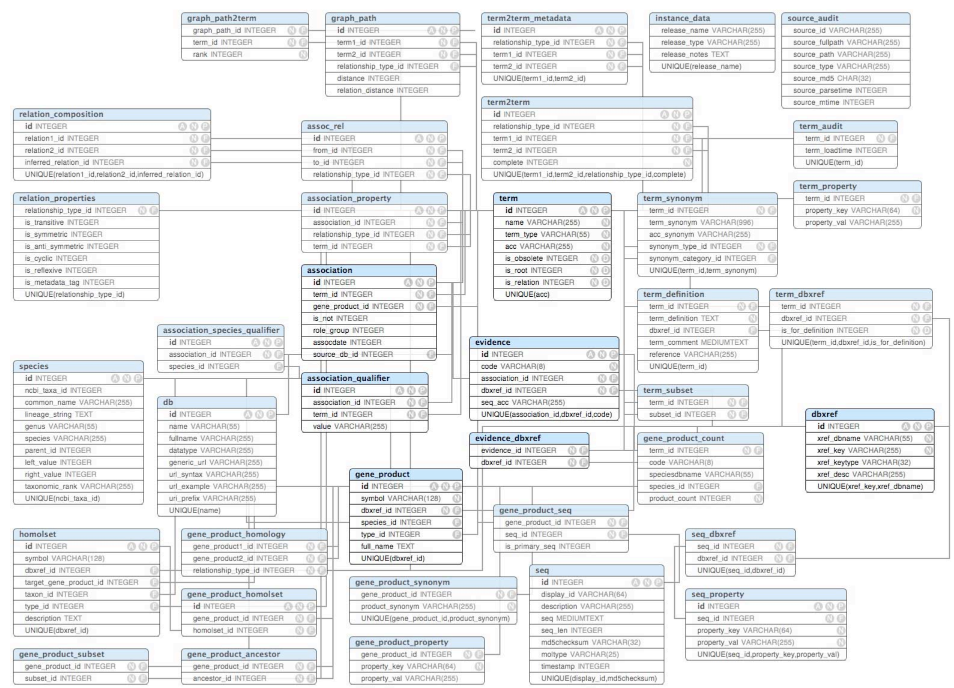
- Efficiency challenge: Extracted graphs can often be orders-of-magnitude larger than original database
  - Homogeneous graphs (over the same set of entities) invariably require at least one self-join on a non-key
  - DBLP Dataset: 8.6M author-publication table → 43M edges in the co-authorship graph
- Ev ur

Graph	Representation	Edges	Extraction Latency (s)
DBLP	Condensed	17,147,302	105.552
	Full Graph	86,190,578	> 1200.000
IMDB	Condensed	8,437,792	108.647
	Full Graph	33,066,098	687.223
TPCH	Condensed	52,850	15.520
	Full Graph	99,990,000	> 1200.000
UNIV	Condensed	60,000	0.033
	Full Graph	3,592,176	82.042

2. Which "Graphs" to Analyze?

- Entities can be connected in a variety of different ways
  - Add an edge if customers bought the same item, *or* at least 5 same items, *or* bought items on the same day in the same store
  - Create a part-supplier bipartite graph by connecting suppliers who apply a part in sufficient quantity

Identifying interesting connections itself a difficult question



2. Which "Graphs" to Analyze?

- Entities can be connected in a variety of different ways
  - Add an edge if customers bought the same item, *or* at least 5 same items, *or* bought items on the same day in the same store
  - Create a part-supplier bipartite graph by connecting suppliers who apply a part in sufficient quantity
- Often need to simultaneously analyze multiple graphs
  - Compare a graph on products today vs yesterday
  - Plot how supplier centrality (e.g., PageRank) evolved over time
- Must exploit overlap, and reduce redundant computation

3. Graph Programming Frameworks

- "Vertex-centric framework" the most popular today
  - GraphLab, Apache Giraph, GraphX, X-Stream, Grail, Vertexica, …
  - Most of the research, especially in databases, focuses on it
- "Think like a vertex" paradigm
  - User provides a **compute()** function that operates on a vertex
  - Executed in parallel on all vertices in an iterative fashion
  - Exchange information at a barrier through message passing

3. Graph Programming Frameworks

- Limitations of the vertex-centric frameworks
  - Works well for some applications
    - Pagerank, Connected Components, Some ML algorithms, …
  - However, the framework is very restrictive
    - Simple tasks like counting neighborhood stats infeasible
  - Fundamentally: Not easy to decompose analysis tasks into vertex-level, independent local computations

- Alternatives?
  - Galois, Ligra, GreenMarl: Low-level APIs, and hard to parallelize
  - Some others (e.g., Socialite) restrictive for different reasons

3. Example: Local Clustering Coefficient

Measures *density* around a node

Compute() at Node n:

Need to count the no. of edges between neighbors

But does not have access to that information

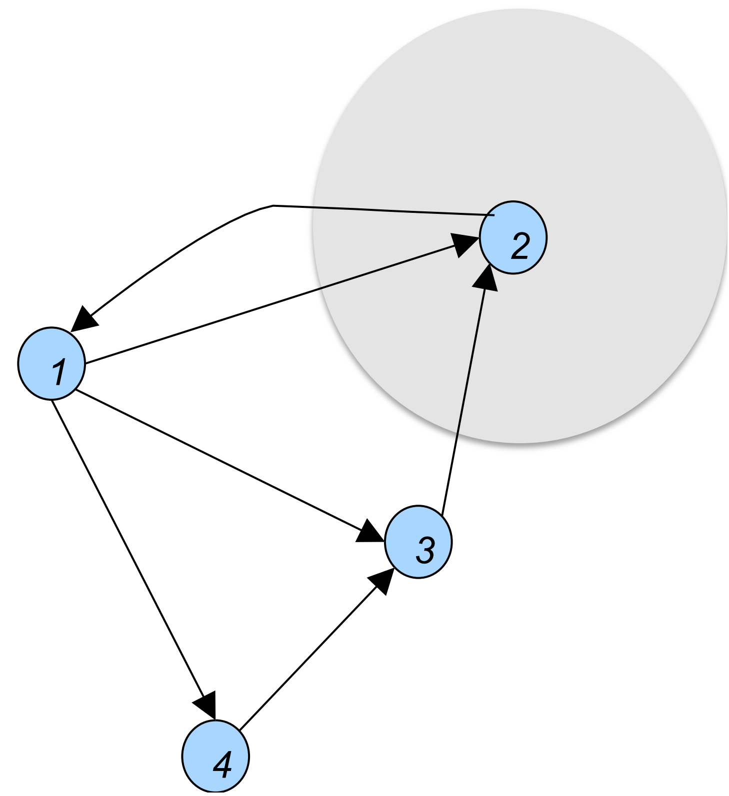
Option 1: Each node transmits its list of neighbors to its neighbors

Huge memory consumption

Option 2: Allow access to neighbors' state

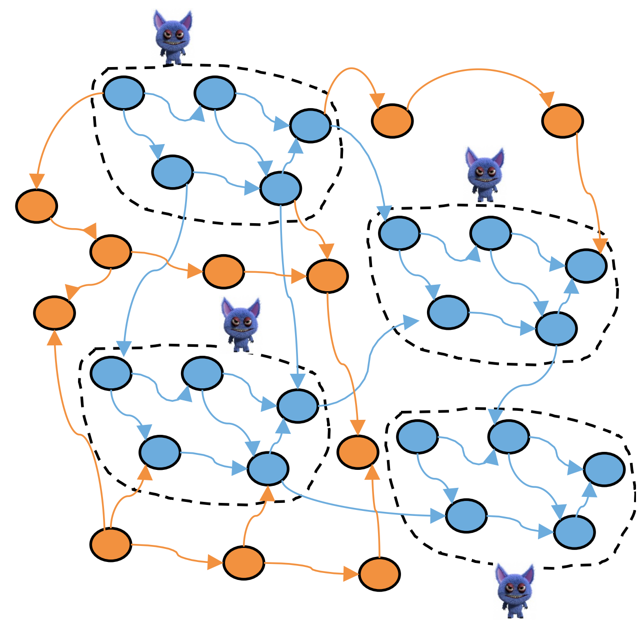
Neighbors may not be local

What about computations that require 2-hop information?



3. Aside: NScale Distributed Framework

- An end-to-end, subgraph-centric distributed graph analytics framework (built on Spark)
- Users/application programs specify:
  - Neighborhoods or subgraphs of interest
  - A kernel compute() to operate on those subgraphs
- Framework:
  - Extracts the relevant subgraphs from underlying data and loads in memory
  - Execution engine: Executes user computation on materialized subgraphs



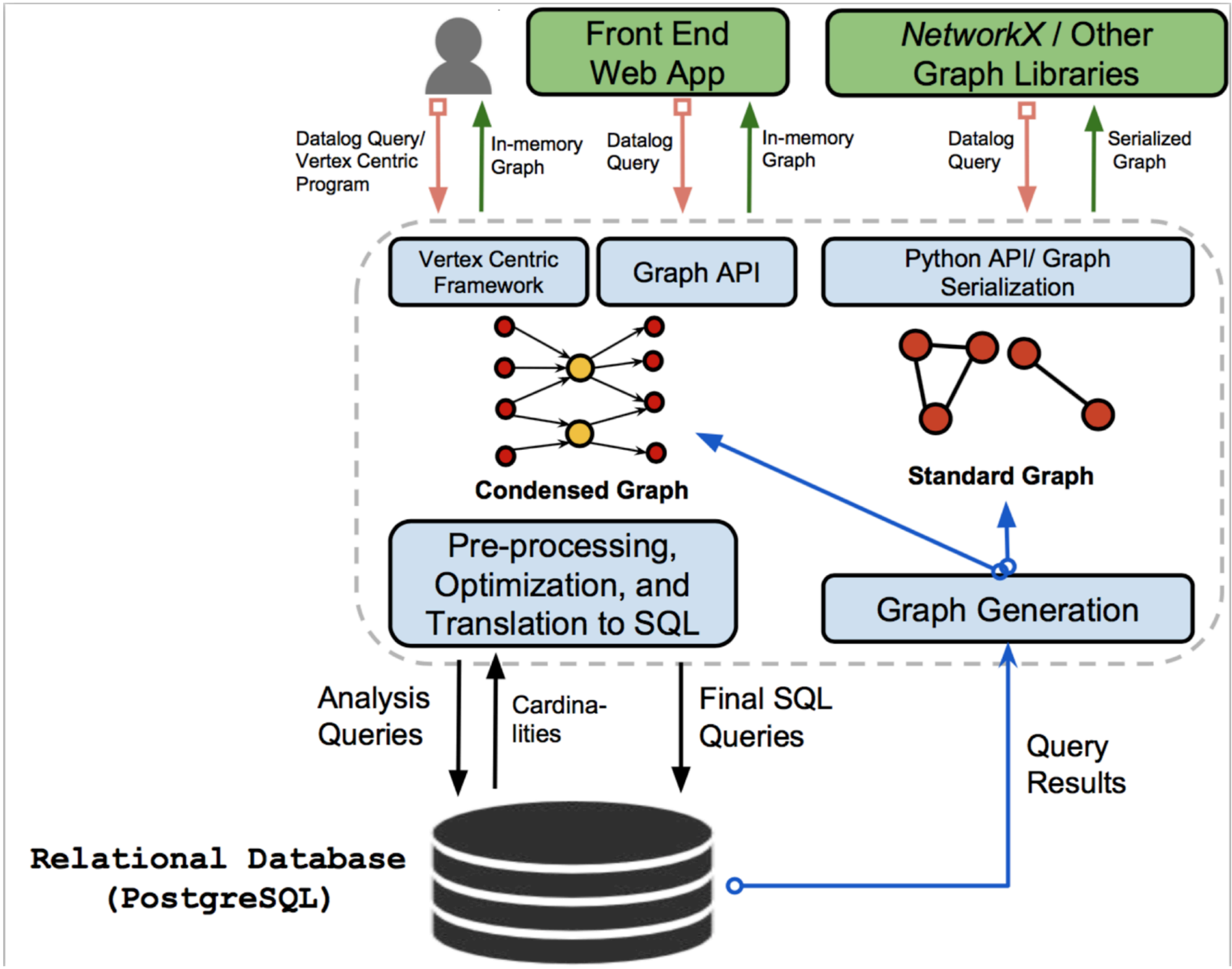
	Local Clustering Coefficient							
Dataset	**NScale**		**Giraph**		**GraphLab**		**GraphX**	
	CE (Node-Secs)	Cluster Mem (GB)	CE (Node-Secs)	Cluster Mem (GB)	CE (Node-Secs)	Cluster Mem (GB)	CE (Node-Secs)	Cluster Mem (GB)
WikiTalk	726	24.16	DNC	OOM	1125	37.22	1860	32.00
LiveJournal	1800	50.00	DNC	OOM	5500	128.62	4515	84.00
Orkut	2000	62.00	DNC	OOM	DNC	OOM	20175	125.00

Outline

- DataHub: A platform for collaborative data science
- GraphGen: Graph Analytics on Relational Databases
  - Motivation
  - System Overview
  - Condensed Representations for Large Graphs
  - Experiments

These slides at: http://go.umd.edu/w.pdf

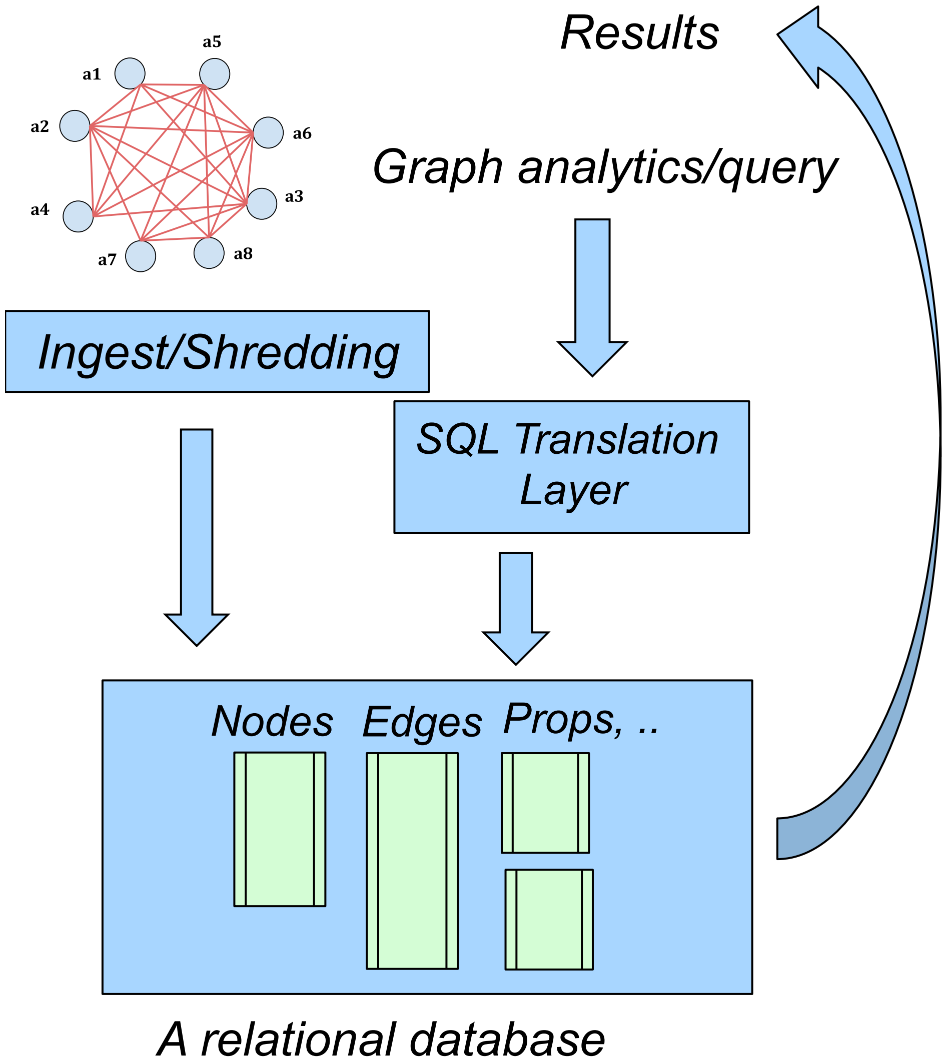
GraphGen Architecture

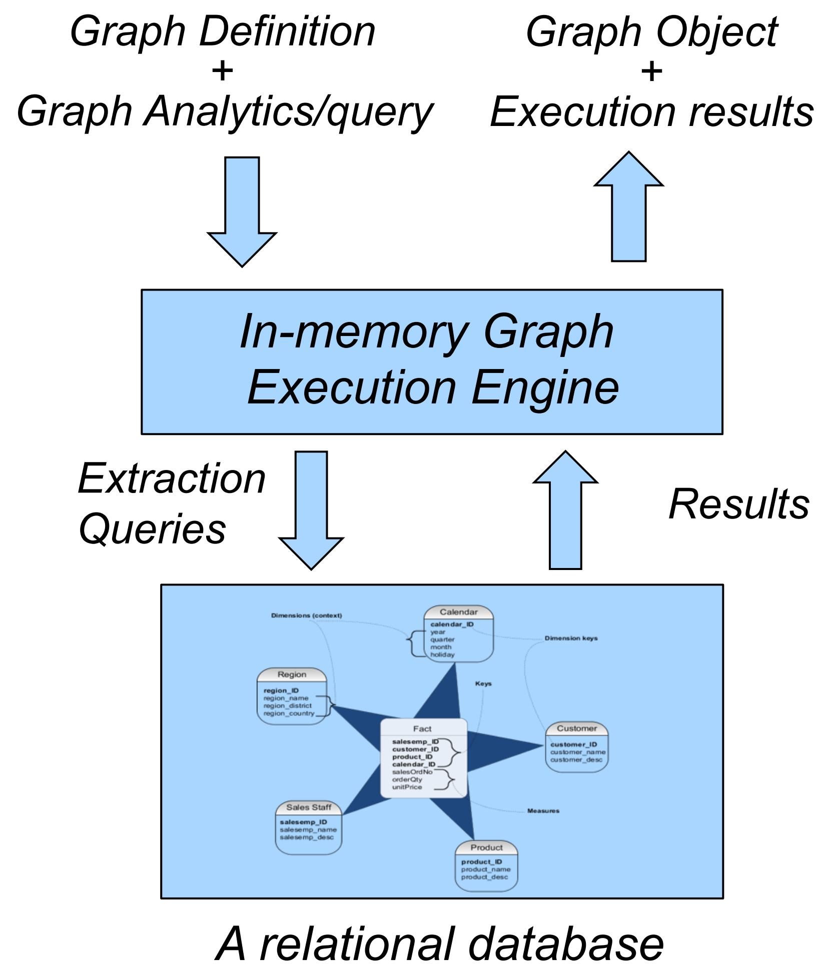


Vertexica/GRAIL/SQLGraph vs GraphGen

Fundamentally different goals

We aim to push computation into RDBMS if possible, but expressive programming framework is a higher priority





GraphGen Graph Extraction DSL

- Based on non-recursive *Datalog*
  - Extended with Aggregation and Looping constructs
- User needs to specify:
  - How the **nodes and edges** are defined
  - Both effectively "views" over the relational data
- Allows for *homogeneous* and *heterogeneous* graphs

1. *Construct customer-customer graph if they bought the same product (TPC-H)*

```
Nodes(ID, Name) :- Customer(ID, Name).
Edges(ID1, ID2) :-
Orders(o_key1, ID1), LineItem(o_key1, part_key),
Orders(o_key2, ID2), LineItem(o_key2, part_key).
```

GraphGen Graph Extraction DSL

2. Construct one **neighborhood** graph for each author (DBLP)

```
For Author(X, _).
        Nodes(ID, Name) :- Author(ID, Name), ID = X.
        Nodes(ID, Name) :- AuthorPub(X,P), AuthorPub(ID,P),
                           Author(ID, Name).
        Edges(ID1, ID2) :- Nodes(ID1, _), Nodes(ID2, _),
                               AuthorPub(ID1, P), AuthorPub(ID2, P).
```

3. A Simple Bipartite Graph over Parts and Suppliers

```
Nodes(ID, Name, Label = "P") :- Part(p_key, Name)
Nodes(ID, Name, Label = "S") :- Supplier(s_key, Name)
Edges(ID1, ID2) :-  PartSupp(ID1, ID2)
```

Additional constructs for aggregates and node or edge “properties” and “weights”

GraphGen in Java: Vertex-Centric API

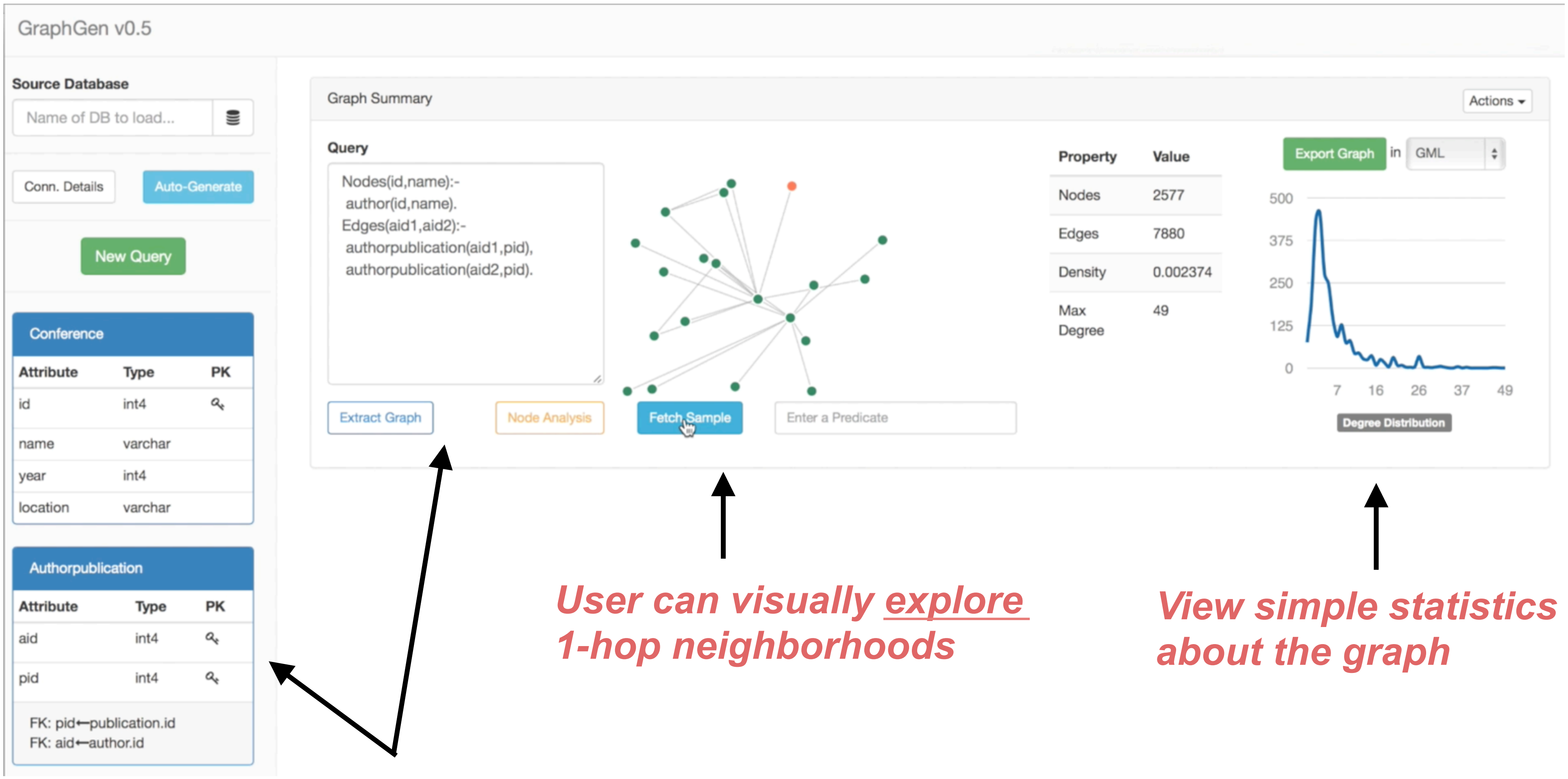
```java
// Establish Connection to Database
GraphGenerator ggen = new GraphGenerator("host", "port",
 "dbName","username", "password");

// Define and evaluate a single graph extraction query
String datalog_query = "...";
Graph g = ggen.generateGraph(datalog_query).get(0);
```

2. Can directly manipulate the graph using a simple API:
- *getVertices(): returns an iterator over all vertices*
- *getNeighbors(v): returns an iterator over v's neighbors*
- *existsEdge(v, u), addEdge(v, u), deleteEdge(v, u), addVertex(v), deleteVertex(v)*

3. Working on supporting a more general neighborhood-centric API from NScale
- Allows parallelism and other optimizations

GraphGen Graph Exploration Frontend



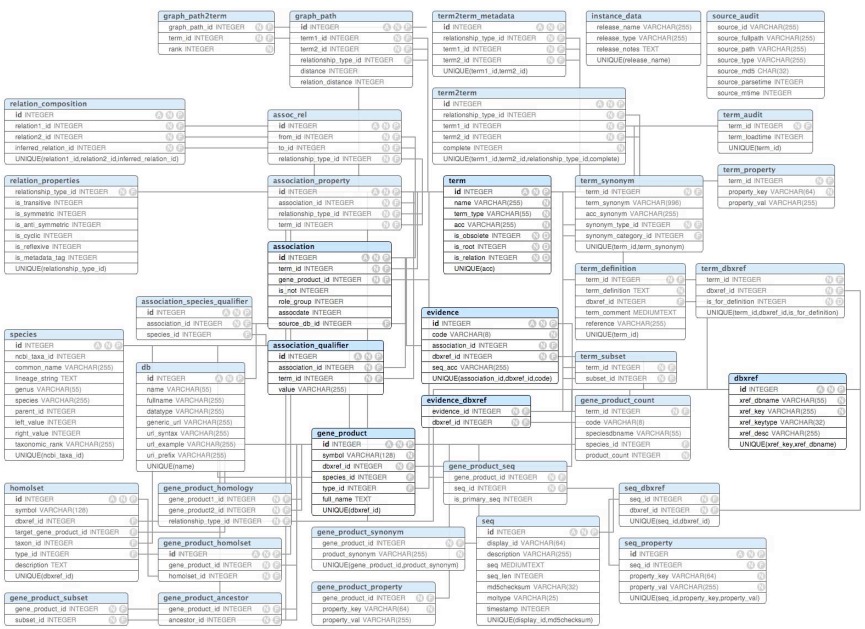
User can visually explore 1-hop neighborhoods

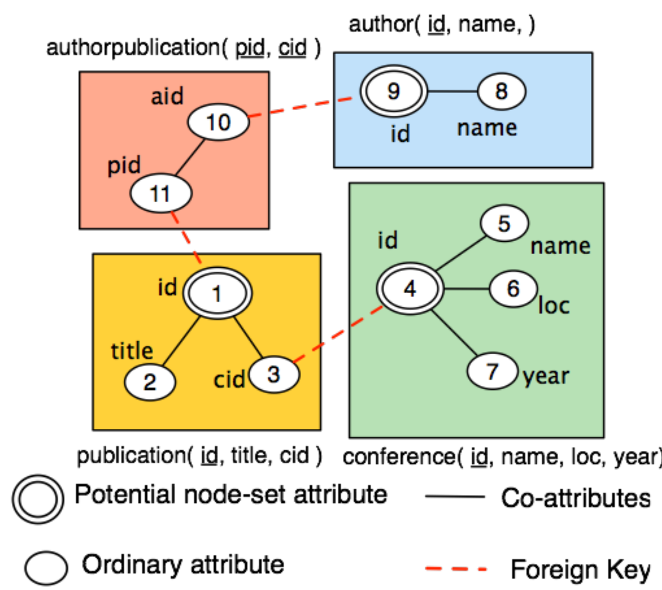
View simple statistics about the graph

User explores schema and specifies graphs to be extracted

GraphGen Enumeration Framework

- Complex relational schemas contain many tables/constraints
  - Hard to identify interesting graphs through just inspection
- Idea: Inspect the database schema, and propose a set of possible graphs by enumerating **paths** or **loops** in the schema graph
  - User provides feedback to drive and fine-tune





Schema Graph for DBLP dataset

Outline

- DataHub: A platform for collaborative data science
- GraphGen: Graph Analytics on Relational Databases
  - Motivation
  - System Overview
  - Condensed Representations for Large Graphs
  - Experiments

These slides at: http://go.umd.edu/w.pdf

Key Challenge

- The extracted graph may be much larger even than the input dataset
  - Expensive to extract: intermediate/final results too large
  - Query optimizers not able to optimize well
  - Possibly infeasible to hold in memory
- Instead: we extract a condensed representation
  - At most the size of the base tables – usually much smaller
  - All Graph APIs supported on top of this representation
  - Need to handle *duplication*

Condensed Representation

Author

a_id	*name*
a1	name1
a2	name2
a3	name3
a4	name4
a5	name5
a6	name6
a7	name7
a8	name8

Publication

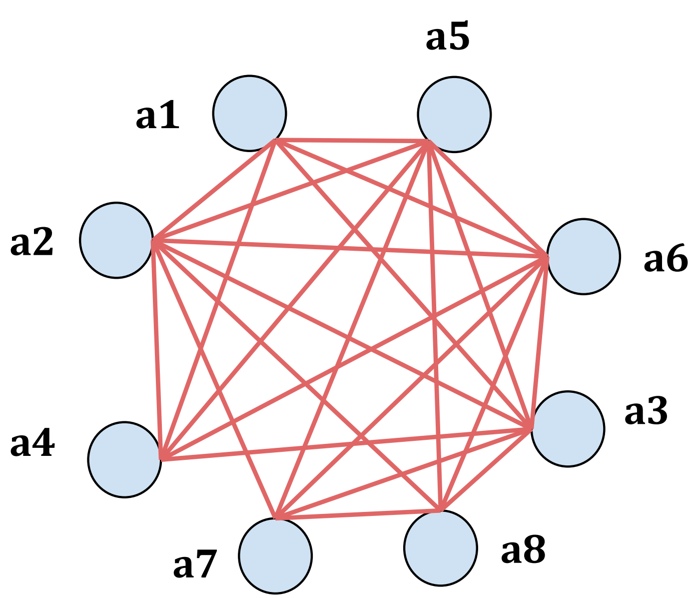
p_id	*title*
p1	title1
p2	title2
p3	title3

AuthorPub

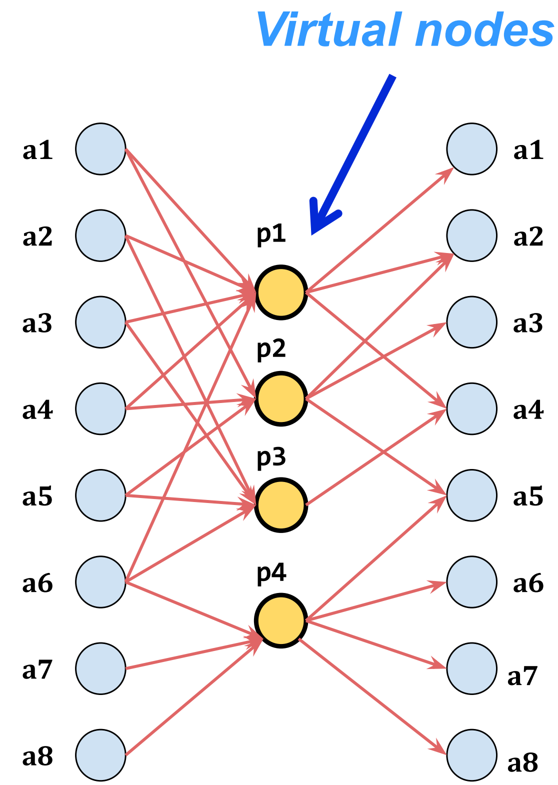
a_id	*p_id*
a1	**p1**
a2	**p1**
a3	**p1**
a4	**p1**
a6	**p1**
a1	**p2**
a4	**p2**
a5	**p2**
a2	**p3**
a3	**p3**
a5	**p3**
a6	**p3**
a7	**p3**
a8	**p3**

```
Nodes(ID, Name):-Author(ID, Name).
Edges(ID1, ID2):-AuthorPub(ID1,
PubID), AuthorPub(ID2, PubID).
```

Query to construct a co-authors graph



Expanded Graph

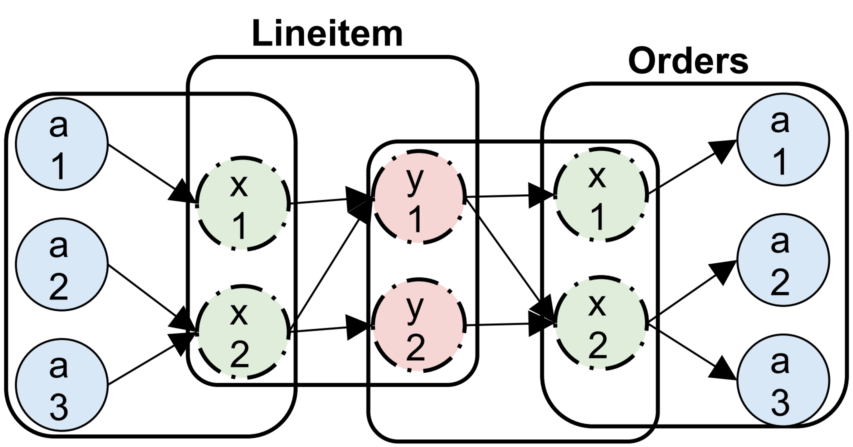


Condensed Graph

Constructing Condensed Graphs

```
Nodes(ID, Name) :- Customer(ID, Name).
Edges(ID1, ID2) :- Orders(order_key1, ID1),LineItem(
     order_key1, part_key), Orders(order_key2, ID2),
     LineItem(order_key2,part_key).
```

1. *Query statistics tables to identify less selective joins*
2. *Break up the overall query to avoid those joins and load intermediate results*
3. *Create a multi-layered representation with "real" and "virtual" nodes (roughly one layer per postponed join)*



Edge from a_i to a_j
==
There is a directed path from a_i to a_j

vertices:315,130 edges:599,902 VS vertices: 15,000 edges: 99,990,000

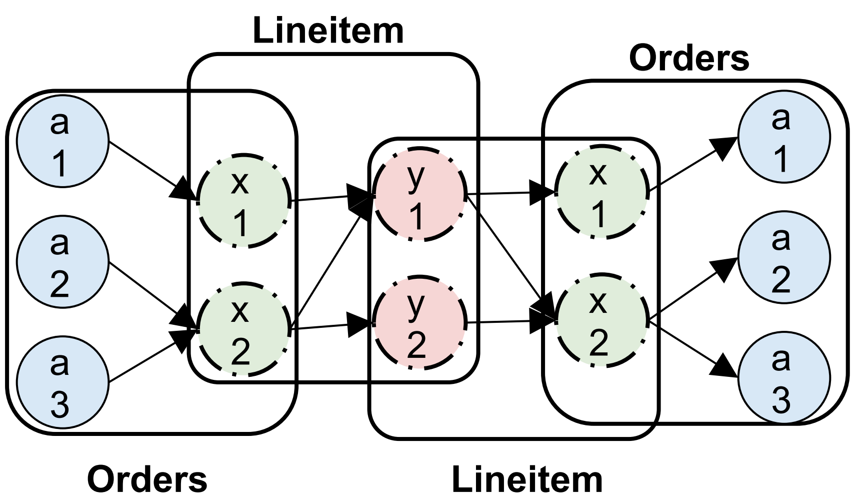
Constructing Condensed Graphs

```
Nodes(ID, Name) :- Customer(ID, Name).
Edges(ID1, ID2) :- Orders(order_key1, ID1),LineItem(
      order_key1, part_key), Orders(order_key2, ID2),
      LineItem(order_key2,part_key).
```

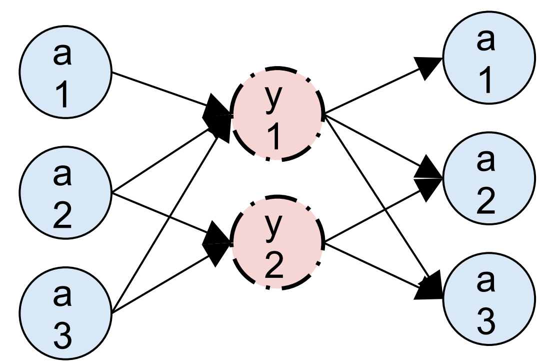
1. Query statistics tables to identify less selective joins

2. Break up the overall query to avoid those joins and load intermediate results

3. Create a multi-layered representation with "real" and "virtual" nodes (roughly one layer per postponed join)

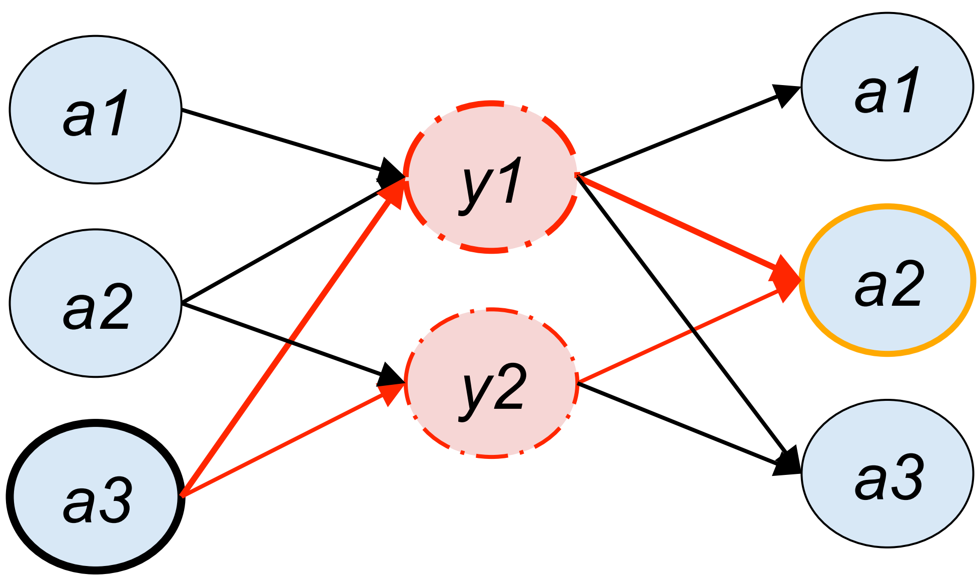


4. Expand low-degree virtual nodes



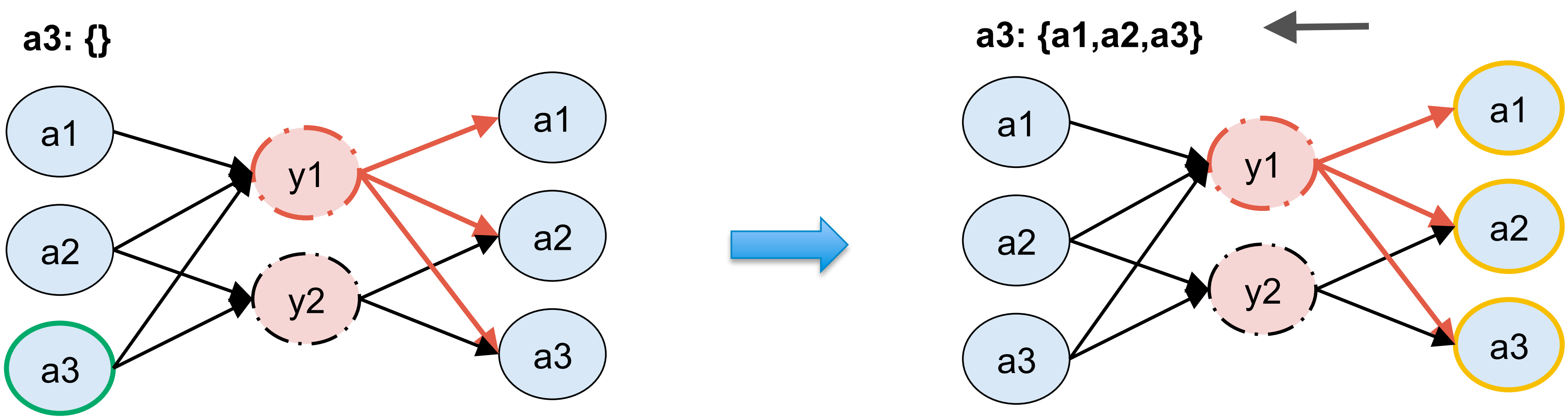
Duplication

- There are duplicate paths between pairs of nodes
- Most graph algorithms cannot handle those
  - Some (e.g., connected components) are tolerant
- Developed several techniques to handle such duplication
  - Different pre-processing, memory, and computation trade-offs



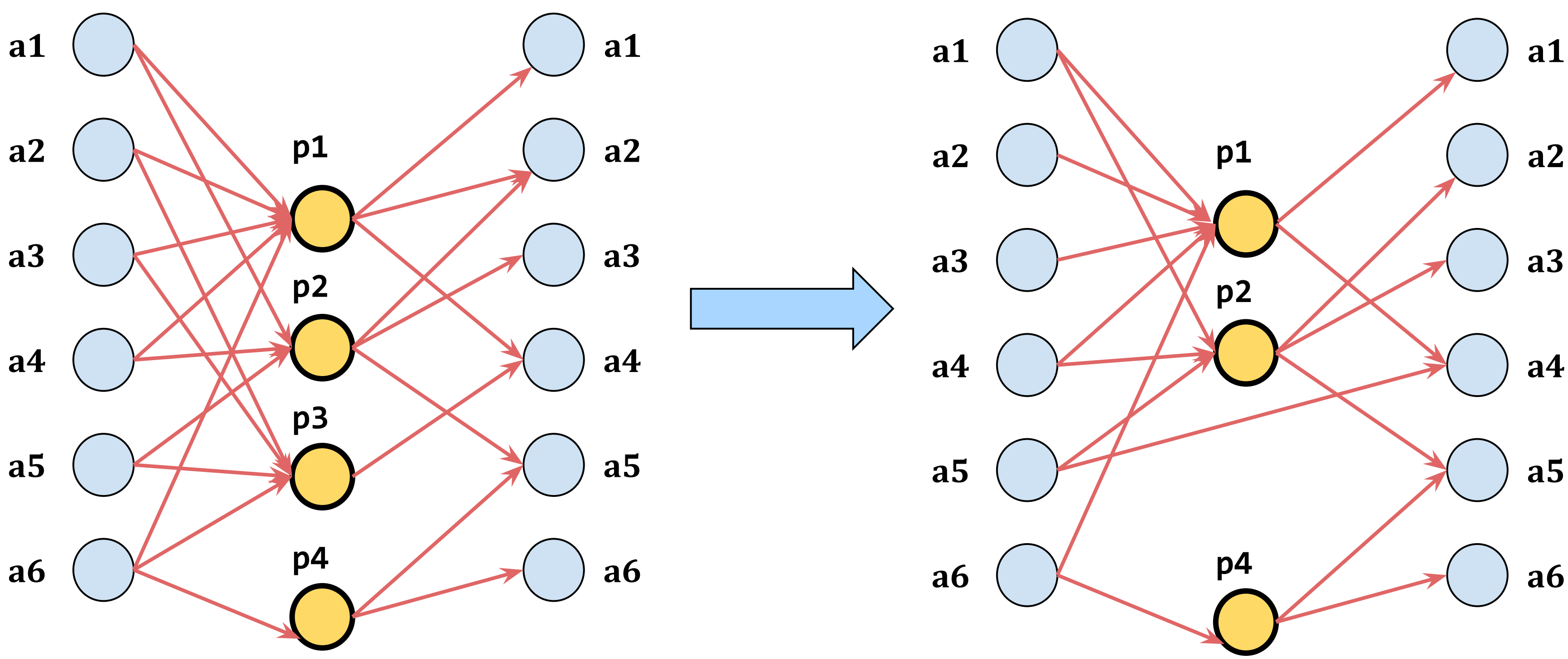
1. CDUP: On-the-fly De-duplication

- Keep the graph in condensed representation
- For every getNeighbors():
  - Do a DFS from the node to find neighbors
  - Cache the neighbor-list (if memory available)
- Overall most memory-efficient
- No pre-processing overhead, but high execution overhead
- Good for graph algorithms that touch a small fraction of the graph



2. DEDUP-1: De-duplicate the graph

- Pre-process the graph to remove duplication, but keep in condensed form
  - i.e., guarantee that there is only one path from a node to each neighbor
- Specialized iterators that return the neighbors one-by-one



2. DEDUP-1: De-duplicate the graph

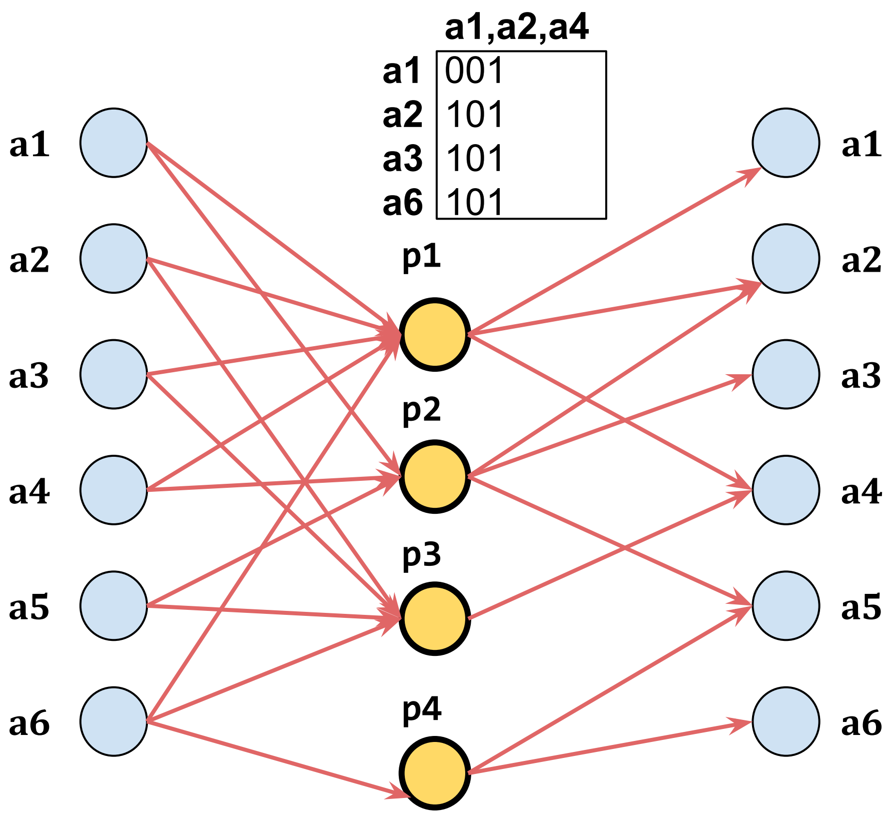
- Pre-process the graph to remove duplication, but keep in condensed form
  - i.e., guarantee that there is only one path from a node to each neighbor
- Specialized iterators that return the neighbors one-by-one
- Most "portable" representation (outside of expanded)
  - Easy to modify other graph libraries to support this
- Pre-processing: is it possible to do this optimally?
  - No: same complexity as compressing a graph by finding cliques or bicliques
  - Prior work in graph compression literature
  - Those algorithms not useful – take expanded graph as input
- We proposed and evaluated 5 algorithms for doing this efficiently
  - Greedily resolve the duplication real node at a time, or virtual node at a time
  - Adaptation of a frequent pattern mining algorithm

3. DEDUP-2: Undirected Virtual Edges

- Allow "undirected" edges between virtual nodes
  - More complicated semantics than "directed" edges
- Can have tremendous benefits for dense graphs
- Limited applicability
- Difficult to work with, and guarantee correctness

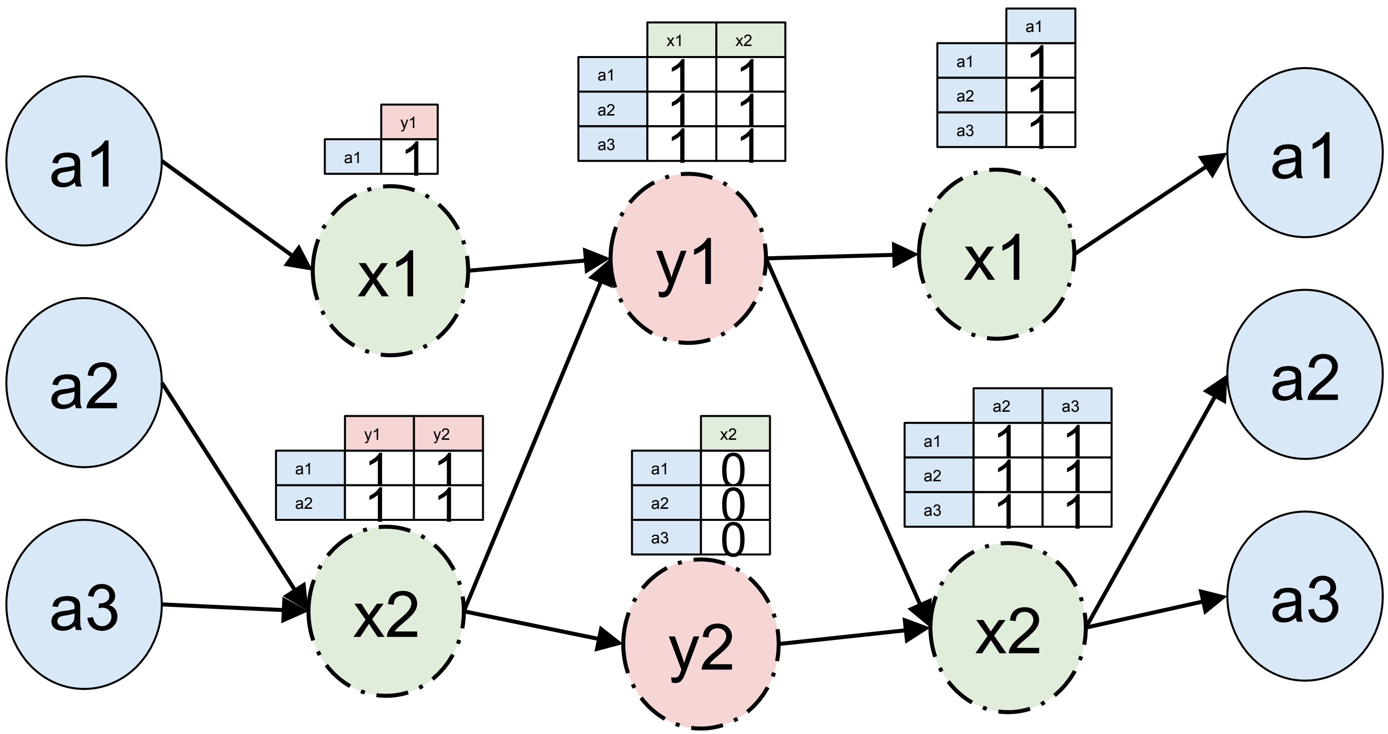
4. Deduplication using Bitmaps

- Use "bitmaps" at virtual nodes to remove duplicate paths
  - Iterators use this bitmaps to return neighbors without duplication
- Typically uses less memory than DEDUP-1
  - At the expense of higher iteration overhead and lower portability



4. Deduplication using Bitmaps

- Use "bitmaps" at virtual nodes to remove duplicate paths
  - Iterators use this bitmaps to return neighbors without duplication
- Typically uses less memory than DEDUP-1
  - At the expense of higher iteration overhead and lower portability
- Works with multi-layered representations too



4. Deduplication using Bitmaps

- Use "bitmaps" at virtual nodes to remove duplicate paths
  - Iterators use this bitmaps to return neighbors without duplication
- Typically uses less memory than DEDUP-1
  - At the expense of higher iteration overhead and lower portability
- Works with multi-layered representations too
  - Some tricks required to keep memory footprint low

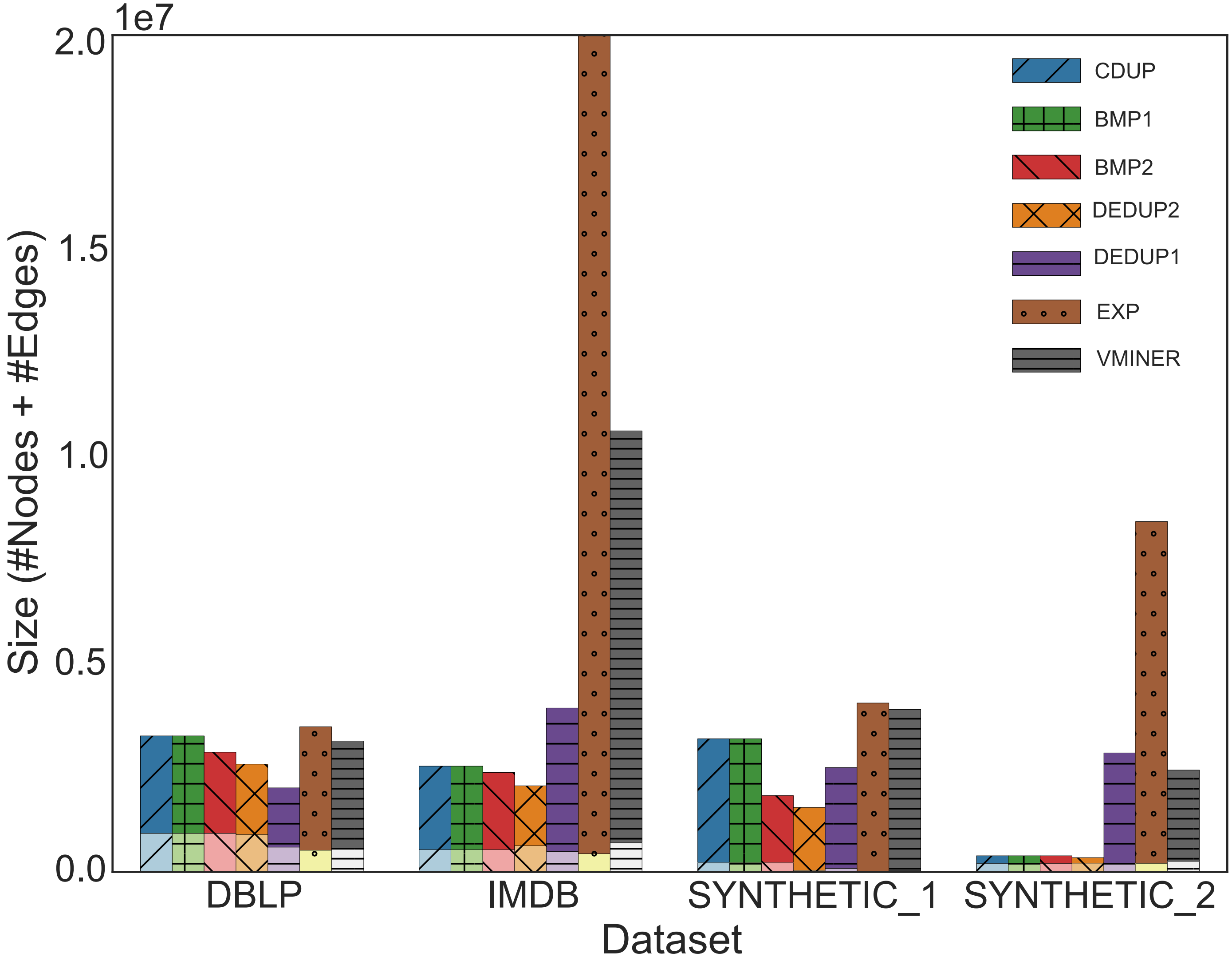
- Preprocessing step to set the bitmaps
  - Turns out to be NP-Hard to do optimally, even for single-layer graphs
  - Non-trivial to parallelize to exploit multiple cores

Outline

- DataHub: A platform for collaborative data science
- GraphGen: Graph Analytics on Relational Databases
  - Motivation
  - System Overview
  - Condensed Representations for Large Graphs
  - Experiments

These slides at: http://go.umd.edu/w.pdf

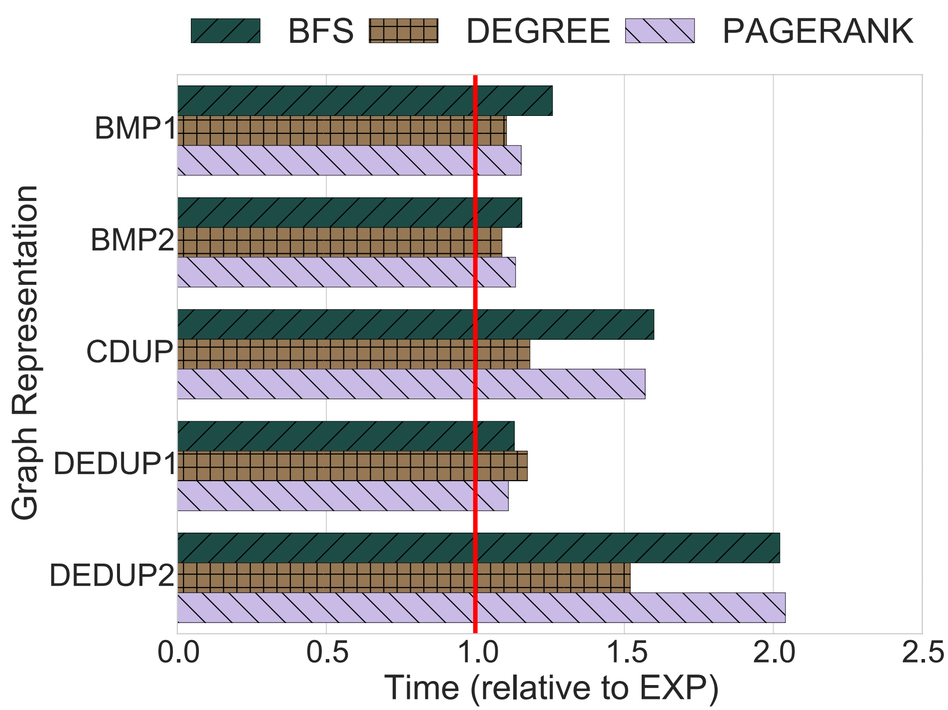
In-memory Graph Sizes



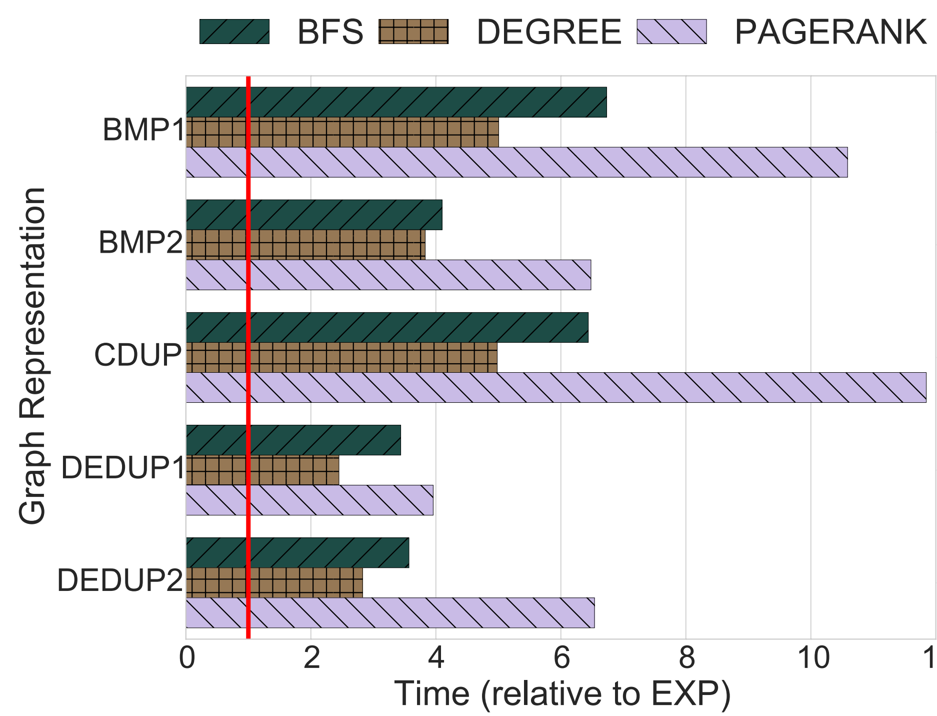
VMiner: Graph compression using Bi-Cliques

Start to see significant differences even with small datasets
Both DEDUP1 or Bitmaps-based approaches work well

Impact on Performance



IMDB Subset



A Synthetic Graph

Generally acceptable performance hit, with DEDUP1 doing the best (at a significantly higher preprocessing cost)

Large Datasets

Memory Footprint (GB)

	CDUP	BMP-DEDUP	EXP
Syn-1	1.421	2.737	>64
Syn-2	1.613	2.258	19.798
Syn-3	1.276	1.493	1.2
Syn-4	9.9	13.042	>64
TPC-H	.023	.049	7.398

Time to run Breadth First Search (seconds)

	CDUP	BMP-DEDUP	EXP
Syn-1	382	284	DNF
Syn-2	129	111	85
Syn-3	0.01	0.02	0.01
Syn-4	1.3	0.12	DNF
TPC-H	86	8.5	16

GraphGen: Summary

- Need to support graph analytics on RDBMSs in situ
- GraphGen provides a declarative DSL and a suite of optimizations for achieving this
- Many computational challenges that we are just beginning to explore
- Working on extending the DSL to support specifying partial graph computations
  - Can push more computation into the RDBMS
- Starting to look at doing this in place on an in-memory database

Thanks !!

More at: http://www.cs.umd.edu/~amol

Questions ?

These slides at: http://go.umd.edu/w.pdf