Graph Databases

Prad Nelluru, Bharat Naik, Evan Liu, Bon Koo

Why are graphs important?

- Modeling chemical and biological data
- Social networks
- The web
- Hierarchical data

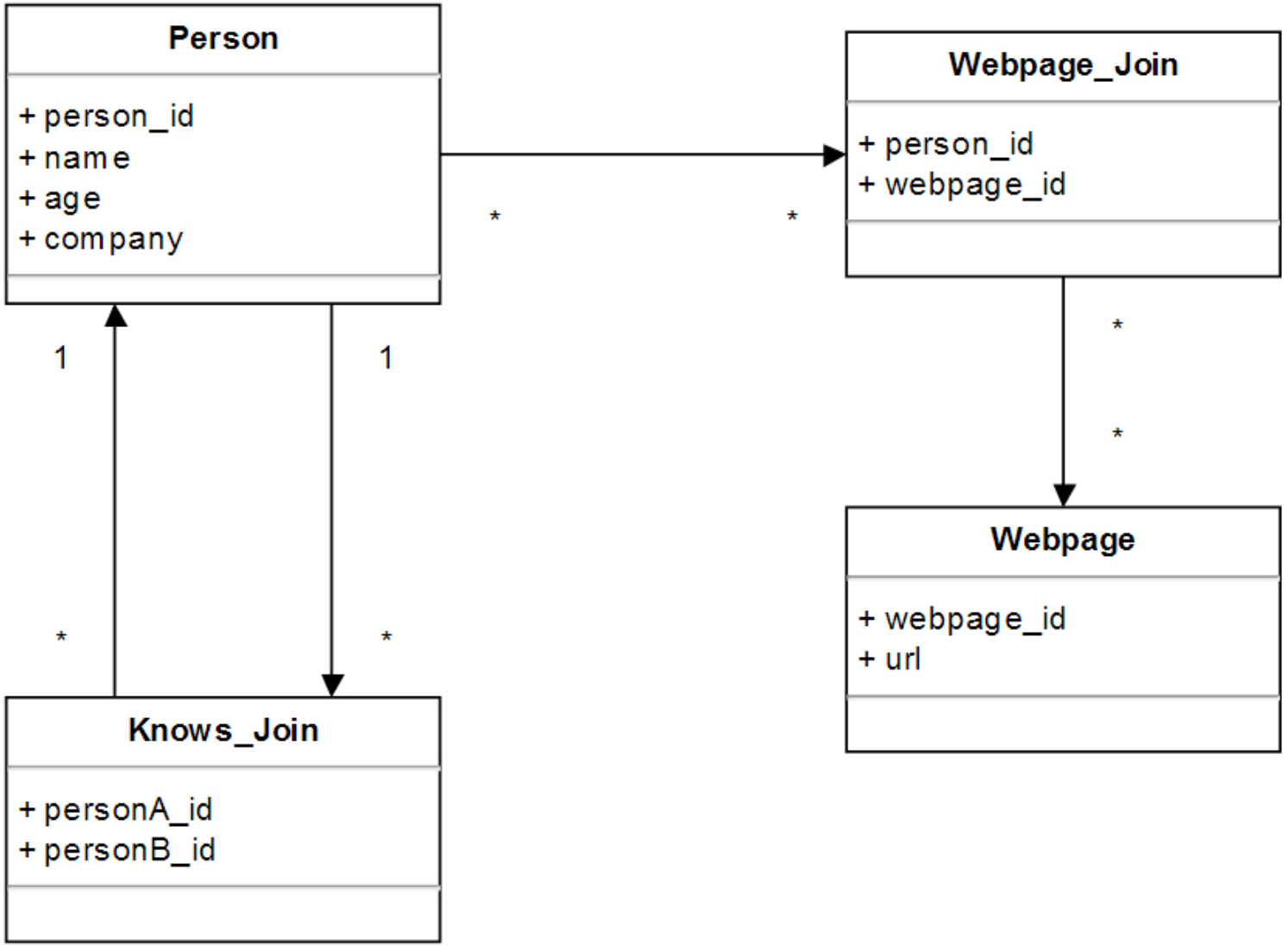
What is a graph database?

- A database built around a graph structure
- Nodes are indexed for fast initial lookup
- Property Graph
  - Each node/edge is uniquely identified
  - Each node has a set of incoming and outgoing edges
  - Each node/edge has a collection of properties
  - Each edge has a label that defines the relationship between its two nodes

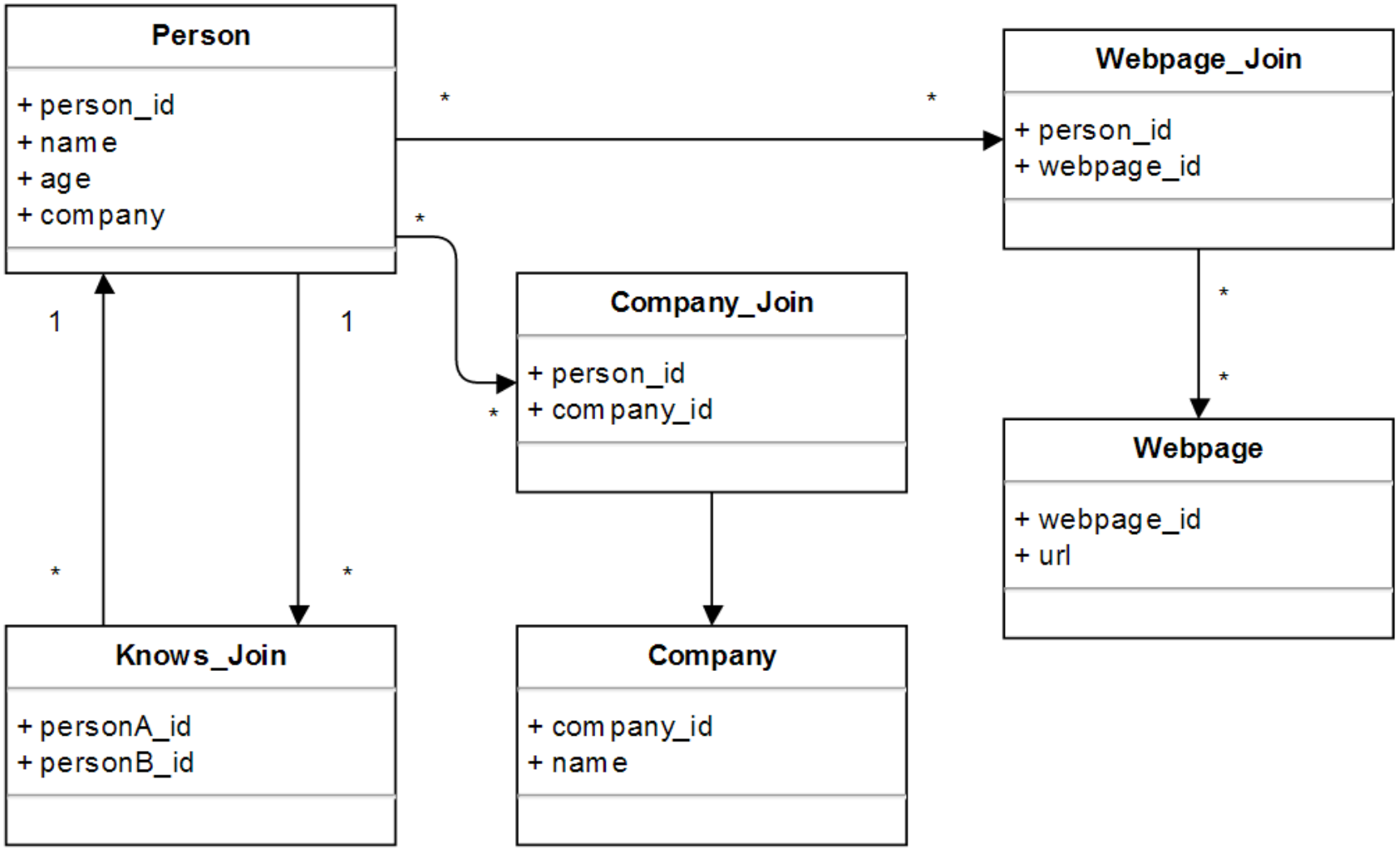
Comparing with Relational DB

- Pros:
  - Schema Flexibility
  - More intuitive querying
  - Avoids “join bombs”
  - Local hops are not a function of total nodes
- Cons:
  - Not always advantageous
  - Query languages are not unified

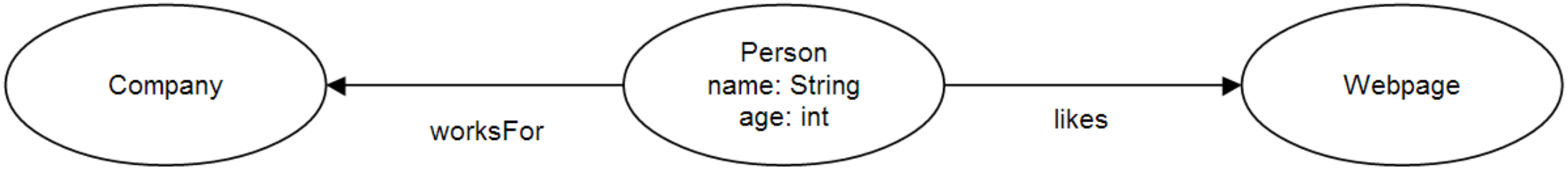
Relational Database Example



Relational Database Example



Graph Database Example

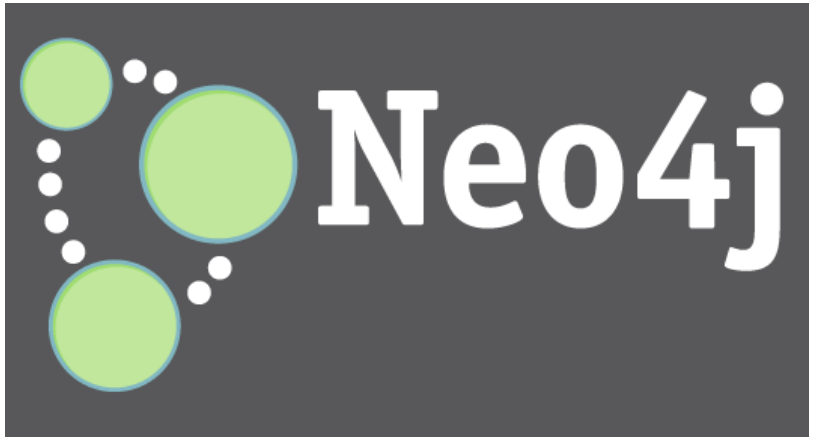


Query Processing in Graph Databases

Graph queries

- List nodes/edges that have this property
- List matching subgraphs
- Can these two nodes reach each other?
- How many hops does it take for two nodes to connect?

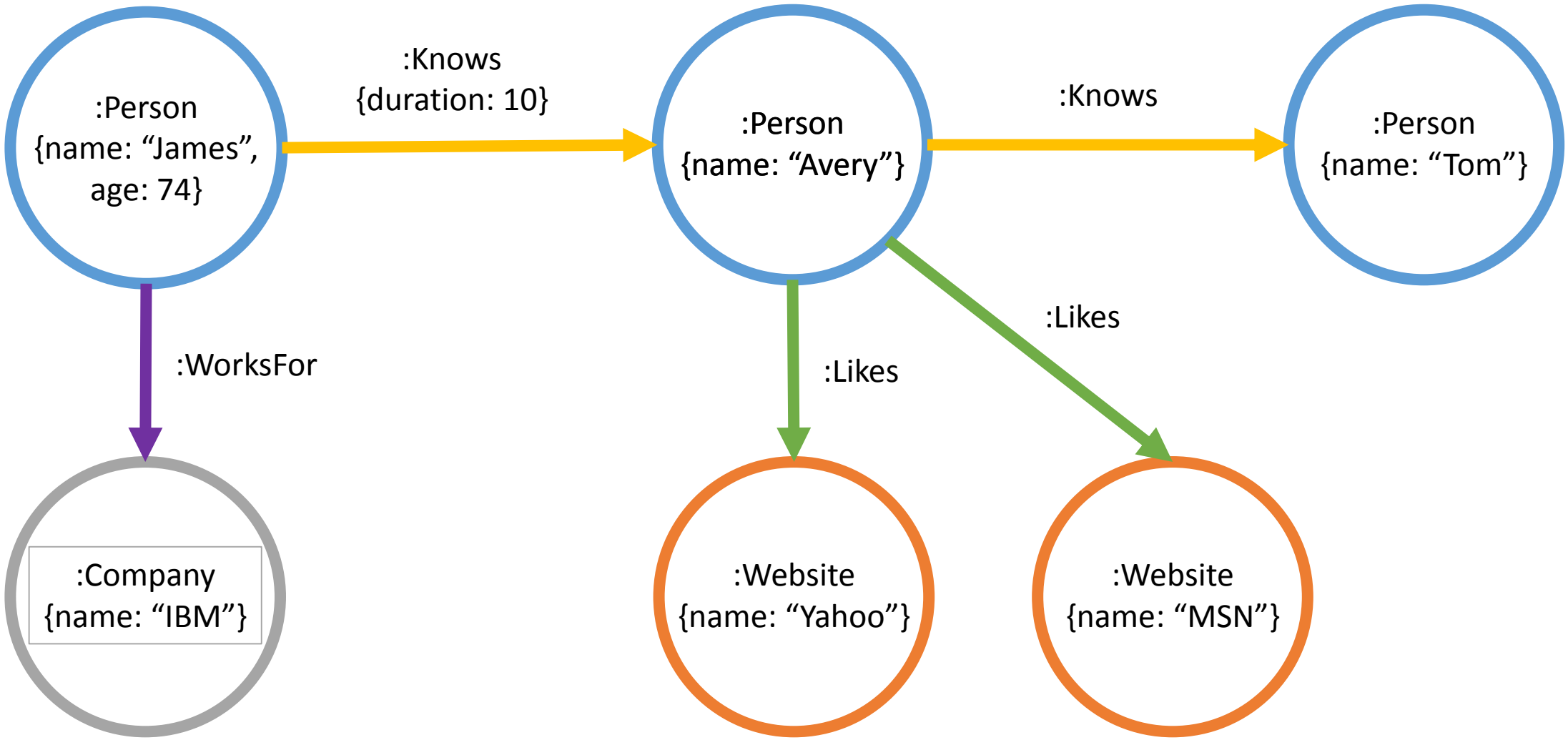
Neo4j



- Most popular commercial graph database
- Transactional queries
- Optimized for single-machine use-cases
- Cypher query language

Neo4j graph model

- Little schema enforcement
  - No special support for a fixed schema
- A graph has nodes and edges
  - Multiple edges between nodes possible
- Nodes and edges can have properties (key-value pairs)
  - Node property: {name: “James”}
  - Edge property: {duration: 10}
- Nodes and edges can have labels
  - Node label: Person
  - Edge label: Knows
- Additional constraints (only schema support)
  - Person id must be unique

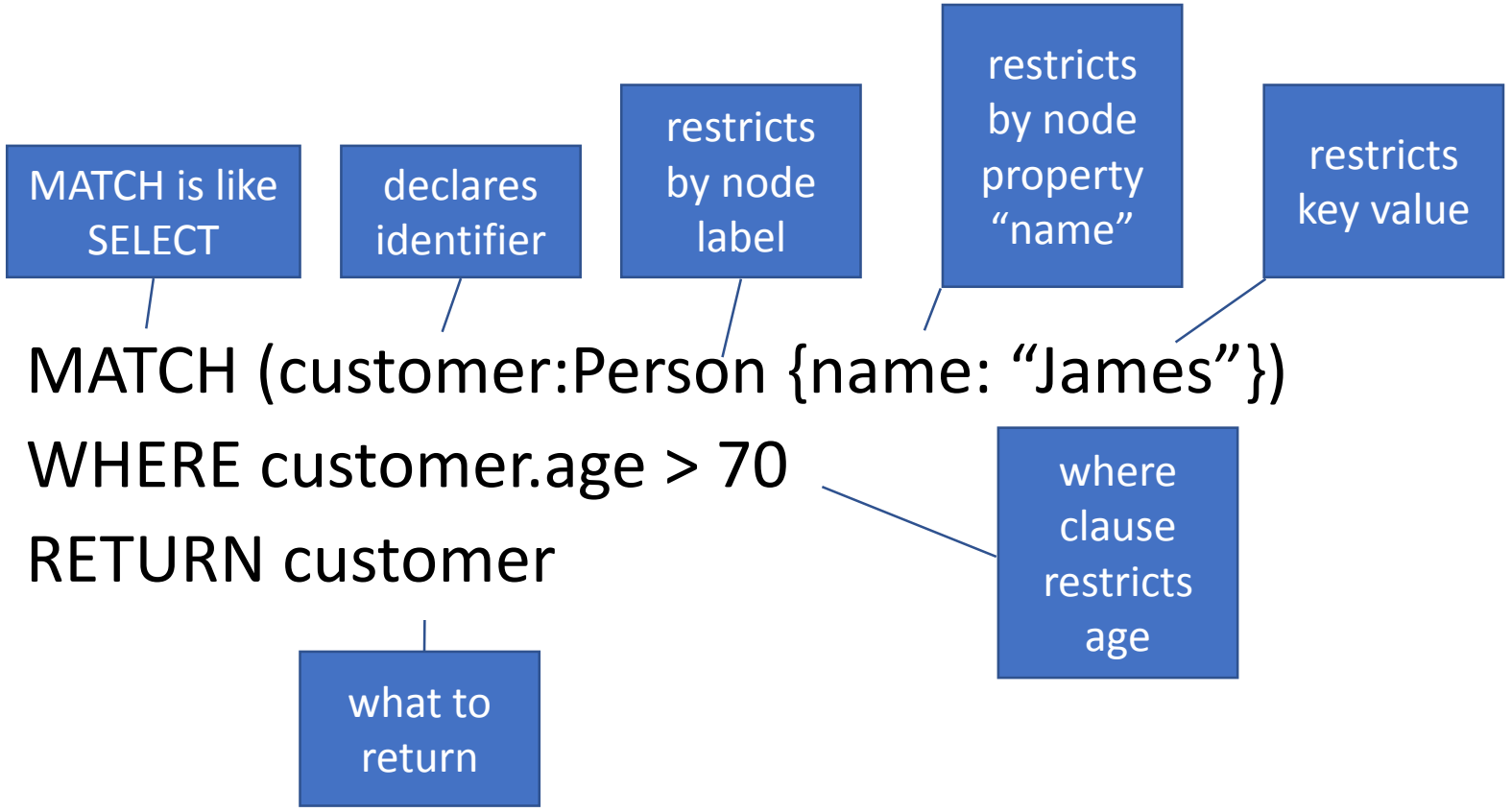

:Person
{name: “James”,
age: 74}
:Knows
{duration: 10}
:Person
{name: “Avery”}
:Knows
:Person
{name: “Tom”}
:WorksFor
:Likes
:Likes
:Company
{name: “IBM”}
:Website
{name: “Yahoo”}
:Website
{name: “MSN”}

Cypher: Neo4j's query language

- Declarative query language based on SQL

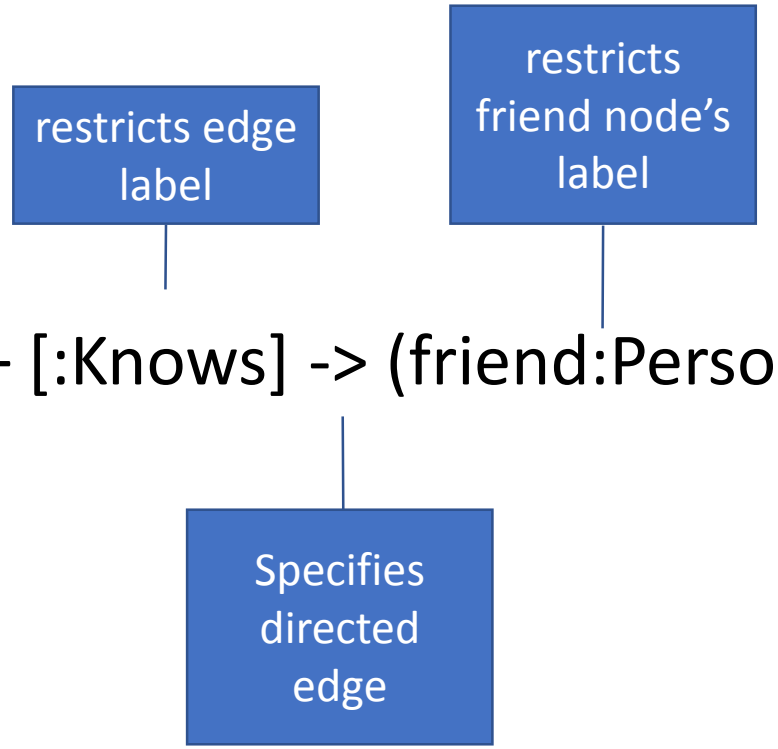
- Matches patterns in graph to
  - retrieve, add, update, delete data
- Manages indexes/constraints

Match older James



Give me people named James older than 70.

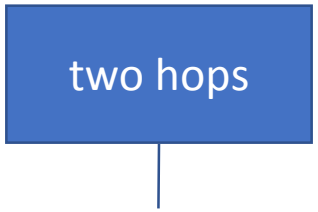
Match James's friends




```
MATCH (customer:Person {name: "James"}) – [:Knows] -> (friend:Person)
RETURN friend
```

Give me James's friends.

Match friends of James's friends

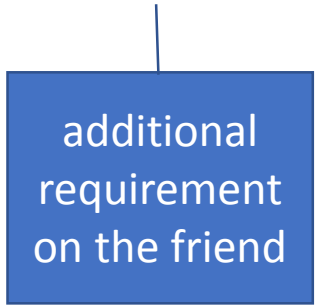



```
MATCH (customer:Person {name: "James"}) – [:Knows * 2] -> (fof:Person)
RETURN fof
```

Give me friends of James's friends.

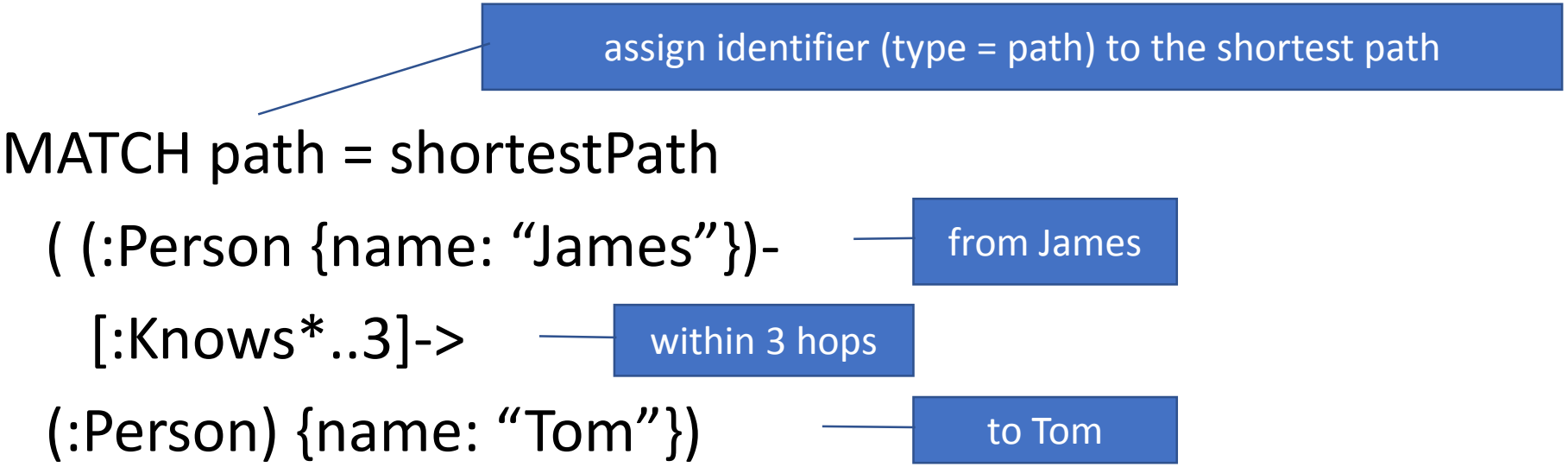
Match James's friends who like Yahoo

```
MATCH (customer:Person {name: "James"}) – [:Knows] -> (friend:Person)
        –[:Likes]->(:Website {name: "Yahoo"})
RETURN friend
```



Give me James's friends who like Yahoo.

Find the shortest path + it's length



Give me the shortest path between Avery and James within 3 hops, and the length of that path.

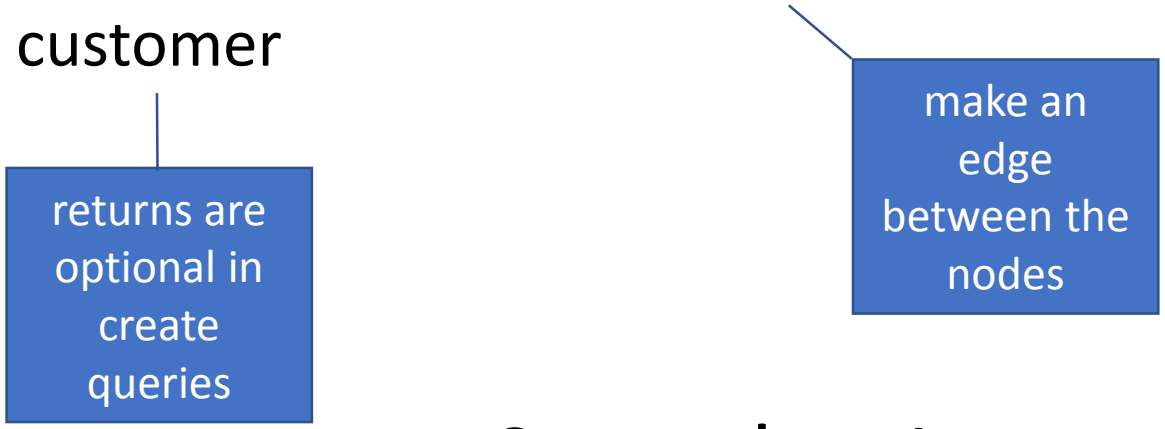
Create a new customer

```
CREATE (:Person {name: “Mike”})
```

Create a Person named “Mike.”

Add a new relationship


```
MATCH (customer:Person {name: “James”}), (site:Website {name: “Yahoo”})
CREATE (customer)-[:Likes]-> (site)
RETURN customer
```



Store that James likes Yahoo.

Answering queries

- Parse query
- Determine starting nodes for traversals
- For each starting node, try to match relationships and/or neighboring nodes with recursive backtracking
- If a match occurs, return

Query optimization

- Neo4j runs Cypher queries as specified
- Not much of an optimizer
- Queries directly translated to action plans

- For the following examples:
  - Assume nodes don’t have labels
  - No indexes are present

Global scan

```
START p = node(*)
MATCH (p {name: “James”})-[:Knows]->(friend)
RETURN friend
```

150ms w/ 30k nodes, 120k rels

Starting from all nodes, find all nodes that know a node named “James.”

*(Numbers from Wes Freeman and Mark Needham: Link)

Introduce a label

Label your starting points

```
MATCH (p {name: “James”})
SET p:Person
```

Find a node named “James” and add a Person label.

Label scan

```
MATCH (p:Person {name: “James”})-[:Knows]->(friend)
  RETURN friend
```

80ms w/ 30k nodes, 120k rels

Starting from all **Person** nodes, find all nodes that know a node named “James.”

Creating indexes

```
CREATE INDEX ON :Person(name)
```

Index lookup

```
MATCH (p:Person {name: “James”})-[:Knows]->(friend)
  RETURN friend
```

6ms w/ 30k nodes, 120k rels

Optimization lessons

- Use labels
- Use indexes on labels
- Order of predicates matter
- Avoid cross products

- Future: query optimizer will do this for you

Graph DB v/s Relational DB for Social Networks

Benchmarking database systems for social network applications

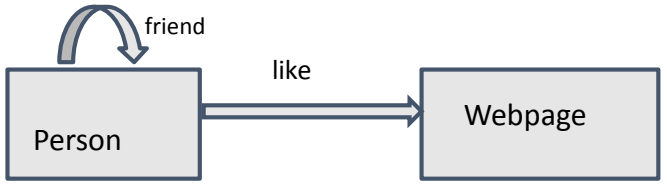
Angles et al in GRADES, 2013

Motivation

- No standard performance benchmarks for social network datasets
- Studies have shown that social network use case is richest in variety
- Authors propose simple microbenchmark - contains very common operations in such networks :
  - Getting friends of friend
  - Looking for similar like pages
  - Shortest path between persons
- Queries are basic primitives, more complex graph-oriented queries can be constructed:
  - Page Rank
  - Recommender Systems

Setup

- Neo4j (v1.8.2 Community) v/s PostgreSQL (v9.1)
- Data model has 2 entities:
  - Person (pid, name, [age], [location])
  - Webpage (wpid, URL, [creation time])



- Graph generation:
  - Stream edge data generation based on R-MAT model
  - Social data generated synthetically to mimic Facebook

Queries evaluated

- Get all persons having name N (Select)
- Get the webpage that person P likes (Adjacency)
- Get the webpages liked by the friends of a given person P (Reachability)
- Get shortest path between two people (Reachability)
- Get the common web pages that two people like (Pattern Matching)
- Get the number of friends of a person P (Summarization)

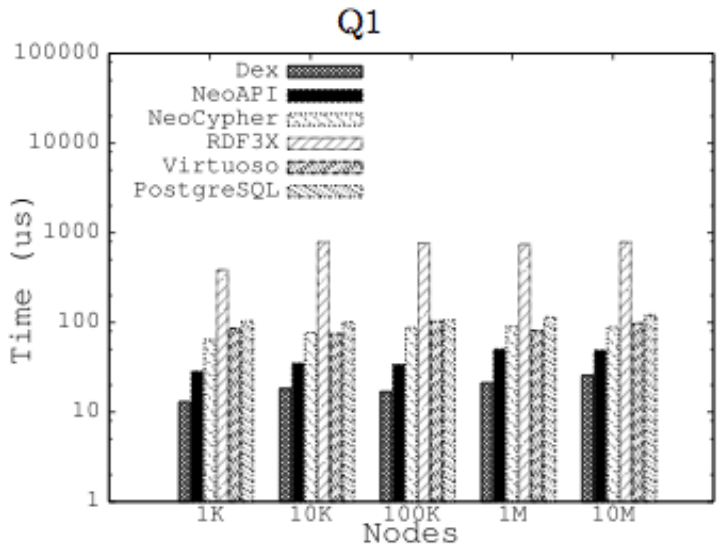
Performance Metrics

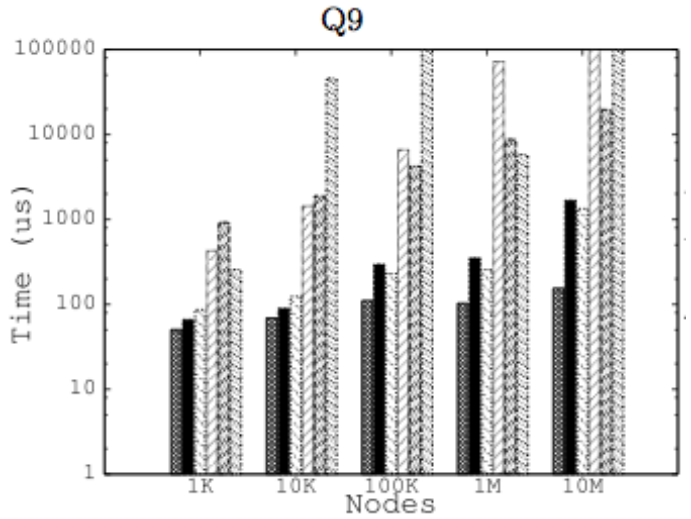
- Data Loading time
  - Time required to load data from source file
  - Build index structures
- Query Execution time
  - Central Performance metric
  - Time spent to execute a single query (averaged over several instances)

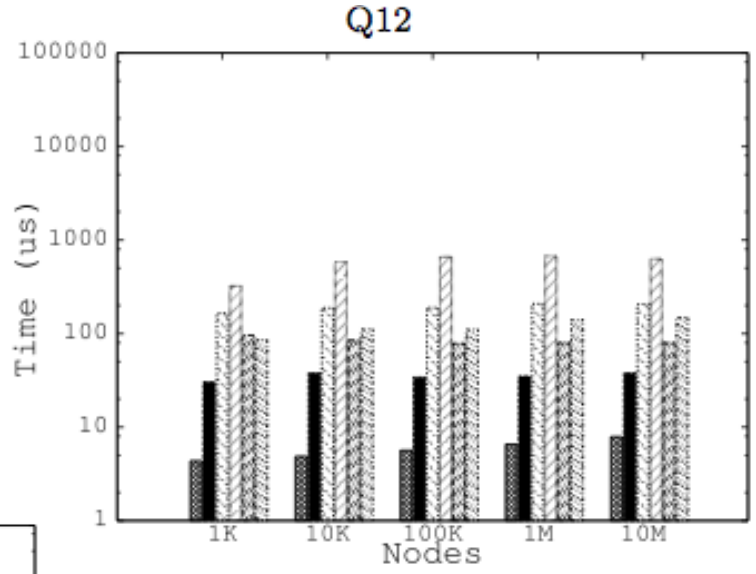
Experimental setup

- Benchmark implemented on Java 1.6
- Indexes were created for primary keys and attributes according to the query requirements
- 1K nodes to 10M nodes
- Focussed only on the queries presented before
- Ran 10K query instances for each query, 3 consecutive runs
- System characteristics:
  - Intel Xeon E5530 CPU at 2.4 GHz
  - 32 GB of Registered ECC DDR3 memory at 1066 MHz
  - 1Tb hard drive with ext3
  - OS: Linux Debian with 2.6.32-5-amd64 kernel

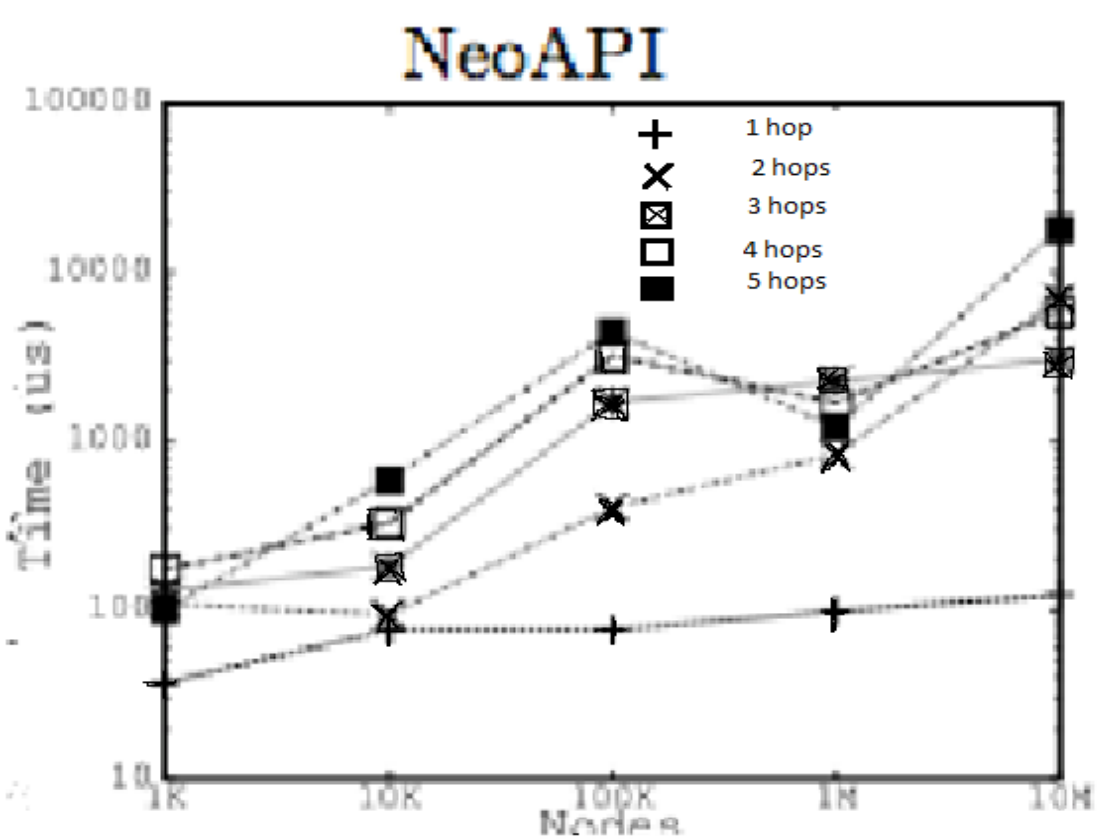
Performance Comparison

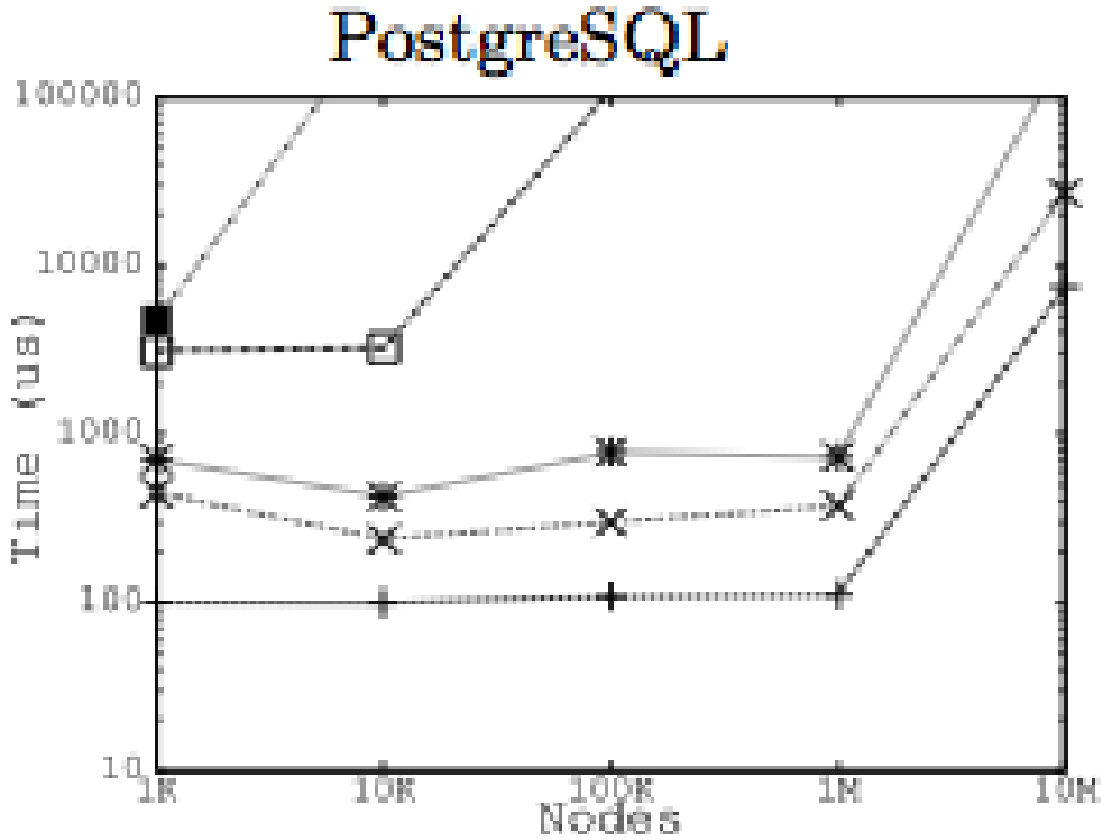






Shortest Path Query





Experimental Results

- Broadly speaking, better performance results on Graph DBs
- Neo4j executed queries in the shortest time with good scalability
- Reachability queries stressed the database systems the most
- For other queries, both systems scaled well independent of graph size
- Neo4j performs best for the shortest path query:
  - Exploit graph-oriented structures
  - Good implementation of BFS
- PostgreSQL:
  - less specialized for path-traversal oriented graph queries
  - Very sensitive to length of paths, scales well till 3 hops (recursive joins)

Conclusions

- Query set mimics real-life queries well : selection, adjacency, reachability, pattern matching and summarization
- Reachability queries were most stressful
- Relational DBs weren’t able to complete query for hops > 4
- Overall, Graph DBs benefit from the benchmark for all query types
- Future work:
  - Stress concurrent user sessions that social networks have to process
  - Composite of several smaller queries that will interact to form a user session
  - Include update operations as part of workload

Distributed Graph Database Systems

Distributed Frameworks

- Existing distributed frameworks are ill-suited for graph operations
- Hadoop (based on MapReduce), for instance, cannot divide graph operation to sub-graph operations to be distributed across workers
- A current step of a graph operation is largely depended on results of its previous steps

Pregel System

- “Pregel: A System for Large-Scale Graph Processing”
- Scalability through Pregel instances
- Fault tolerance via Master/Worker model
- ***Input:*** Graph
- ***Output:*** a set of vertices, an aggregated value, or a subgraph

Algorithm

- [Initialization] Graph is divided into n partitions, which are distributed to n workers, or Pregel instance
- Queries to workers mapping
- All vertices are initially “active”
- Apply superstep, a routine defined by Pregel, repeatedly until all vertices become “halt”

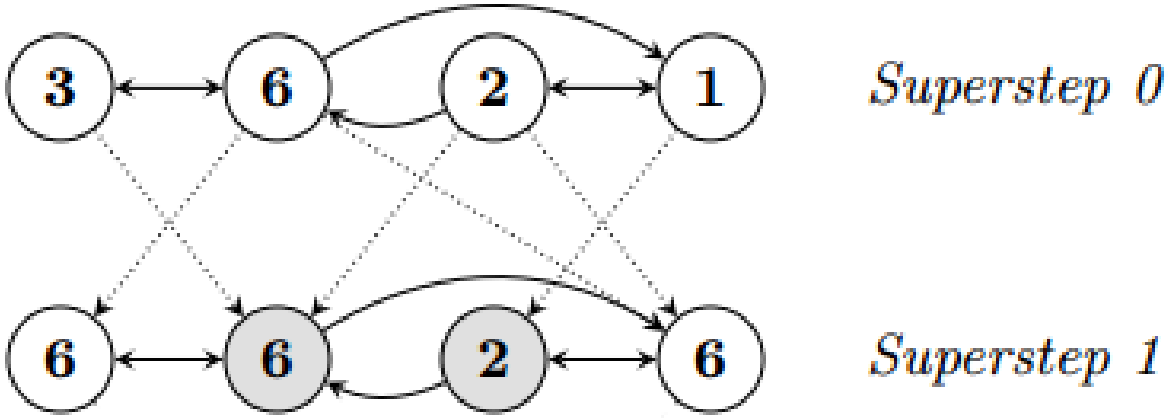
Superstep (for single worker)

- Save its current states of vertices (fault tolerance)
- Apply user-defined graph operation to all “active” vertices in parallel
- Send messages to neighboring vertices
- A vertex may vote to halt if it believes to have no more work to do
- A vertex may go from “halt” to “active” when it receives incoming message(s)

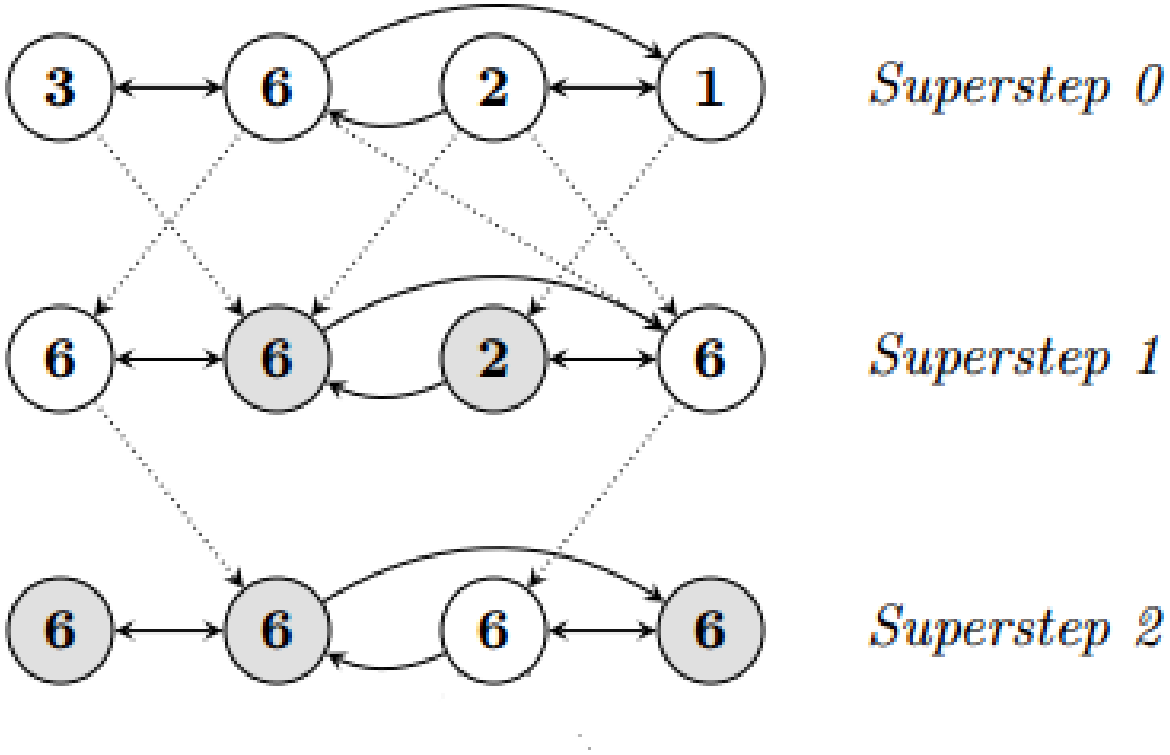
Max Value Propagation Example

Superstep 0

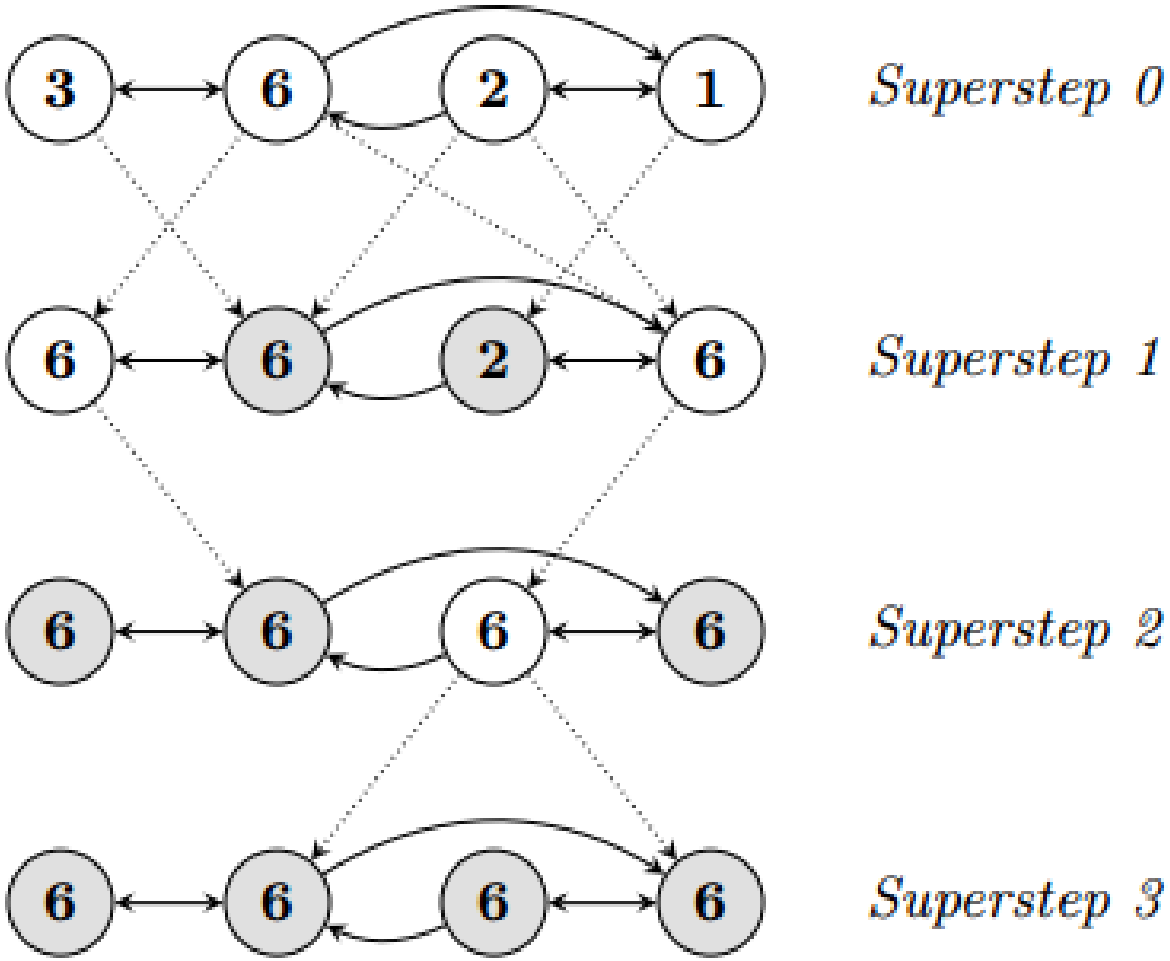
Max Value Propagation Example



Max Value Propagation Example



Max Value Propagation Example



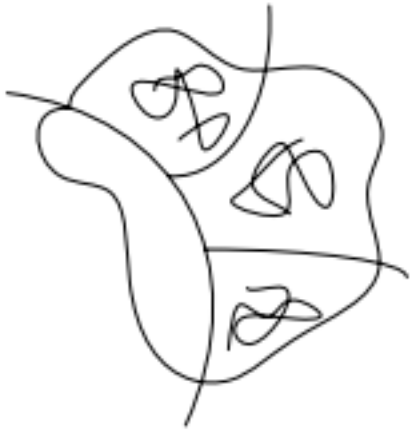
Partitioning Graph

- Pregel's partitioning scheme
  - Divide vertices based on their hash values (mod n) to distribute them to n workers (randomly distributed)
  - Increase inter-machine communications
- "Sedge: Self Evolving Distributed Graph Management Environment"
  - Pregel-based graph partition management system

Two-Level Partition Management

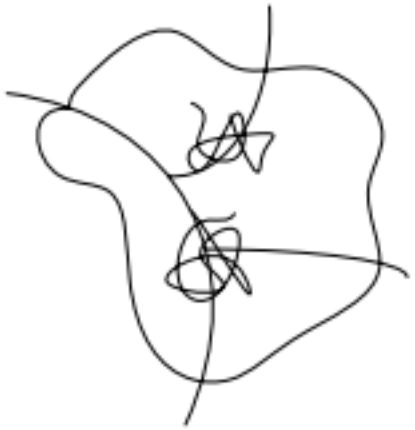
- Primary Partition
  - Each vertex must belong to exactly one primary partition
  - When repartitioned, all cross-partition edges become internal edges
- Secondary Partition
  - Copy of primary partition
  - On-demand partitioning (allocate only when needed)

Partition Replication



- Internal hotspot
  - A frequently accessed part of a graph that is located only within one partition
  - Cause uneven workloads among workers
- Copy internal hotspot(s) to secondary partition(s) of slack workers
- Distributed workloads to increase parallelism

Dynamic Partitioning



- Cross-Partition hotspot
  - A hotspot that spans on 2+ partitions
  - Cause uneven workloads as well as increased inter-machine communications
- Group parts of a cross-partition hotspot together, and allocate the new partition to slack workers
- Increased parallelism and reduced inter-machine communication overhead

Questions?

Backup slides ahead….

Time-varying Social Networks in a Graph Database

- Focus on time varying social graphs
  - Nodes represent individuals
  - Edges represent interactions between them
  - Graph structure and attributes of nodes/edges change over time
- Data gathered from mobile devices and wearable sensors
- Queries involve:
  - Topology of the social network
  - Temporal information on the presence and interactions of nodes
- Challenges:
  - Data curation
  - Cleaning
  - Linking
  - Post-processing
  - Data analysis

Setup

- Social Networks from Wearable Sensors
- Measured by the SocioPatterns collaboration
  - Built a high-resolution atlas of human contact in a variety of indoor social environments
  - Participants wear badges with active RFID devices
  - Spatial range for proximity can be tuned from several meters down to face-to-face proximity
- Data provides detailed sequence of contacts with beginning and ending times
- Represented as time-varying proximity networks, temporal frame of 20s
- For each frame, proximity graph is built with individuals as nodes and interactions as edges

Technical Challenges

- Modeling time-varying networks
  - Can’t use adjacency matrices/lists
    - Face scalability issues with large datasets
    - Provide constrained semantics for querying and exposing data
    - Do not allow flexibility for rich queries involving temporal information
- Storage & Retrieval of Large Networks
  - Topology of real-world graphs is heterogeneous (structural and temporal)
  - Power law distribution i.e. small fraction of nodes have high connectivity
  - Graph problems are data-driven, poor spatial memory locality
  - Runtime dominated by memory access

Data Model

- FRAME nodes
- RUN nodes
- RUN_FRAME relations, RUN_FRAM_FIRST, RUN_FRAME_LAST
- FRAME_NEXT relation
- ACTOR nodes
- FRAME_ACTOR relations
- INTERACTION nodes
- FRAME_INTERACTION relations
- INTERACTION_ACTOR relations

Temporal Indexing

- Suitably index the time-stamped sequence of FRAME nodes
- Attach to the FRAME nodes temporal attributes that can be indexed
- Build tree that explicitly represents the temporal hierarchy of the dataset
- Nodes of tree are TIMELINE nodes
- Top-level node, entry-point for the temporal index, is reachable from the RUN node through a HAS_TIMELINE relation
- Nodes at each level have NEXT_LEVEL relations to nodes at the level below
- At each level, time attributes are associated with the NEXT_LEVEL relation
- Nodes at the bottom level correspond to the finest scale of temporal aggregation

Test Queries

- Get all time frames of run “HT2009”, recorded between 9:00-13:00 of July 1st, 2009, ordered by timestamp
- Get the names of all persons present in a given frame
- Get the weighted proximity graph during a given frame, filtering out the weak contacts
- Get a list of all persons, and for each person, get the number of frames in which they were present
- Get the names of all persons that were present in more than 1000 frames, ranked by time of presence
- List all distinct days on which an actor was present
- Return people in proximity of a given user, sorted alphabetically
- Find common neighbours of any two users
- Compute degree of all persons in contact graph

Results

- Insert table here with median, 5% and 95% quantile times

Conclusion

- Performance:
  - Performed well for exploratory data analysis, querying, data mining
  - Combination of chosen data model and Neo4j proved optimal
  - Issues were encountered while processing densely connected nodes
- Takeaways
  - Graph DBs work well when queries need to be run concurrently with data ingestion
  - Expensive / frequent operations can be pre-computed
  - For densely connected nodes, additional indexing structures are needed to improve performance

Why Neo4j?

- Rich graph data model support
- Reliable storage of large graphs
- Efficient execution of complex queries on large heterogeneous graphs
- Support for property graph data model
- Persistent, transactional storage of very large graphs
- Support for deep graph analytics via efficient many-hop traversals
- Support for Cypher