Performance Introspection of Graph Databases

Peter Macko
Harvard University
Cambridge, MA

Daniel Margo
Harvard University
Cambridge, MA

Margo Seltzer
Harvard University
Cambridge, MA

Conventional Benchmark

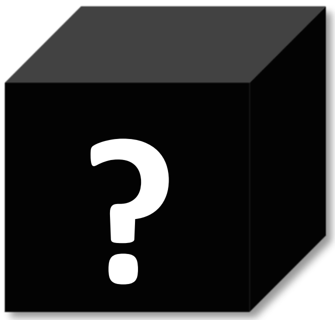
Benchmarking Graph Database X

Dataset with 2 mil. nodes, 10 mil. edges

Unidirectional BFS-based shortest path:

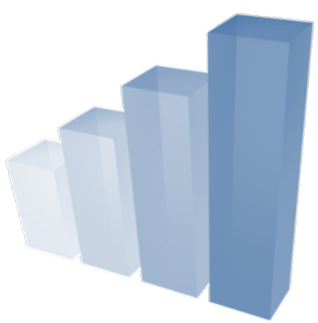
38.3 seconds

Performance Introspection of Graph Databases



- A black-box approach to understanding the strengths and inefficiencies of graph databases.

- A benchmarking methodology that identifies how smaller operations fit together to create bigger operations using quantitative relationships.



- A web-based tool to run the benchmarks and to visualize the results.

Outline

1. Introduction
2. Methodology
3. Implementation
4. Selected Results
5. Conclusion

Methodology

1. Recursively **decompose** a graph application into its primitive graph operations:
   - Get vertex, edge, property
   - Insert/update vertex, edge, property
2. **Measure** each operation.
3. **Model** higher level operations naively in terms of lower-level operations.
4. **Compare** actual and modeled performance to identify strengths/weaknesses of implementation.

Example – Decomposition

- Consider the BFS shortest path:

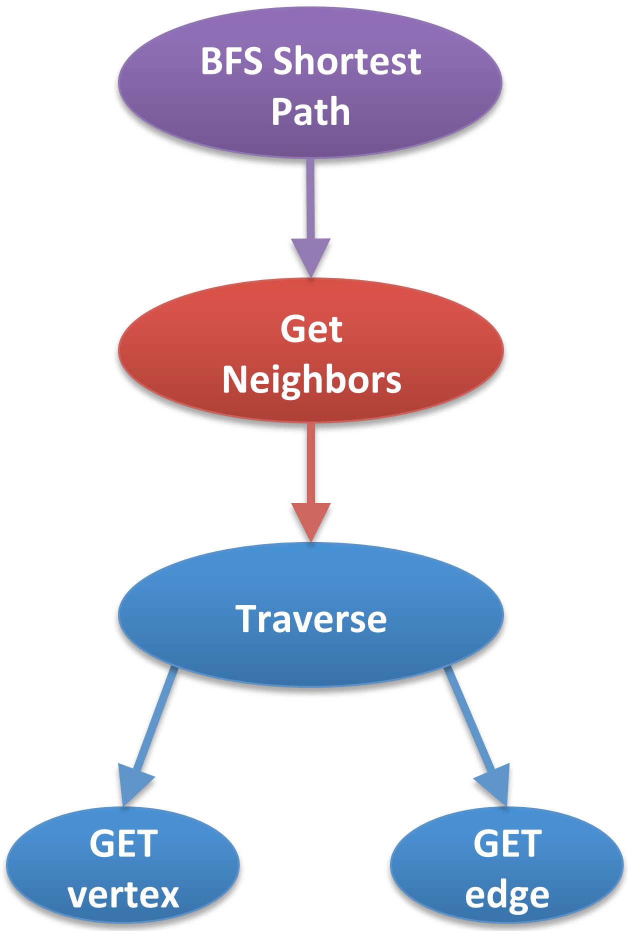
```
Function Shortest-Path(source, target):
  Q ← new Queue { source }
  while Q is not empty:
    v ← dequeue from Q
    if v = target:
      done
    else:
      N ← Get Neighbors of v
      for n ∊ N:
        if n was not yet visited: enqueue n to Q
```

- How long should it take with no optimization?

(Latency of Get Neighbors**) × (**# of visited neighborhoods**)**

Example – Recursive Decomposition

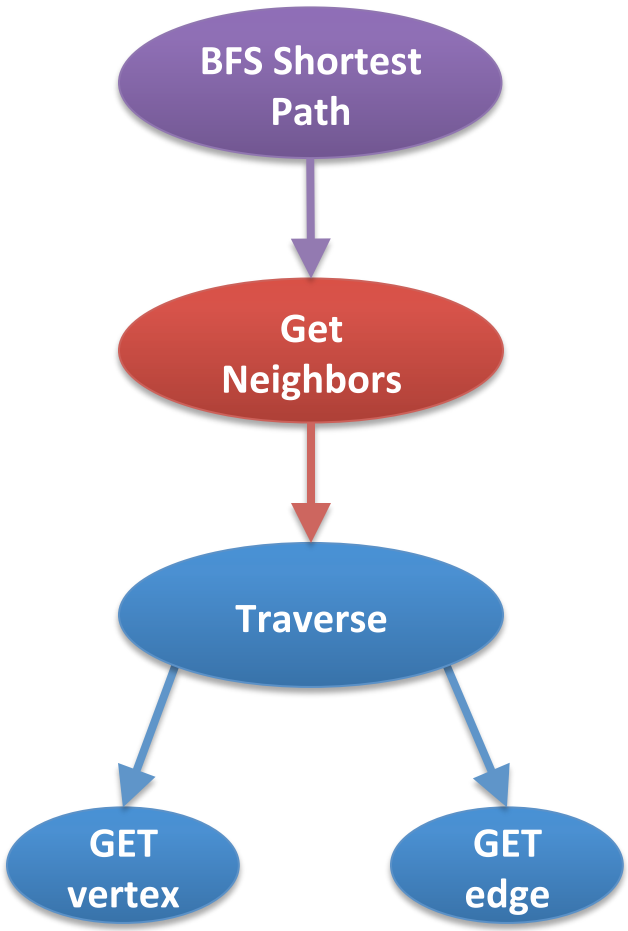
BFS Shortest Path:



- A simple BFS shortest path algorithm decomposes into some number of "Get Neighbors" queries
- A call to "Get Neighbors" traverses on average *n* edges
- A "Traverse" operation gets a single edge from the database and the vertex at the other endpoint

Example – Recursive Decomposition

BFS Shortest Path:



Latency-Model(Shortest Path)

= m × Latency(Get Neighbors)

Latency-Model(Get Neighbors)

= n × Latency(Traverse)

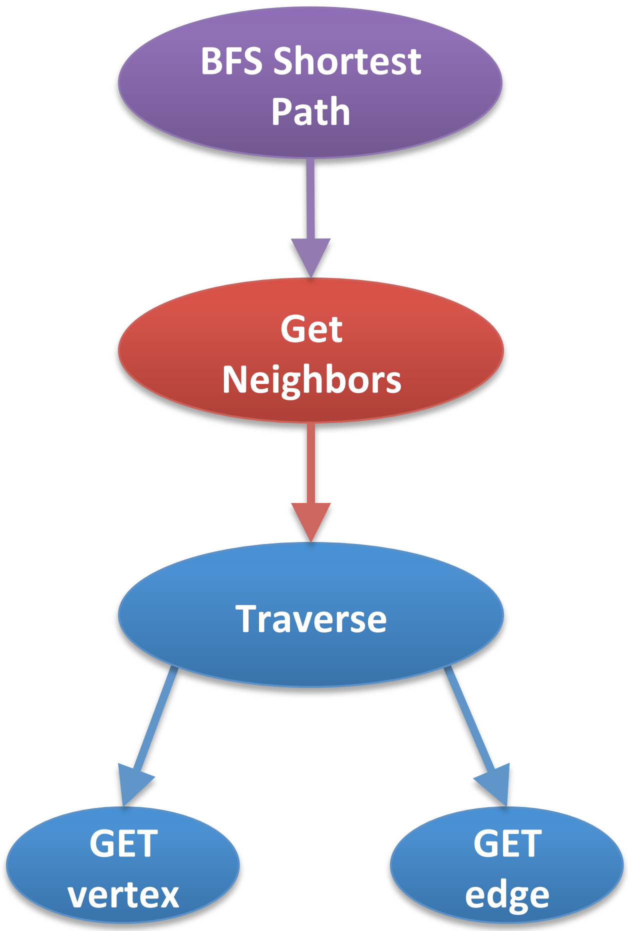
Latency-Model(Traverse)

= Latency(Get Vertex)

+ Latency(Get Edge)

Example – Recursive Decomposition

BFS Shortest Path – Neo4j, 2 mil. node graph:



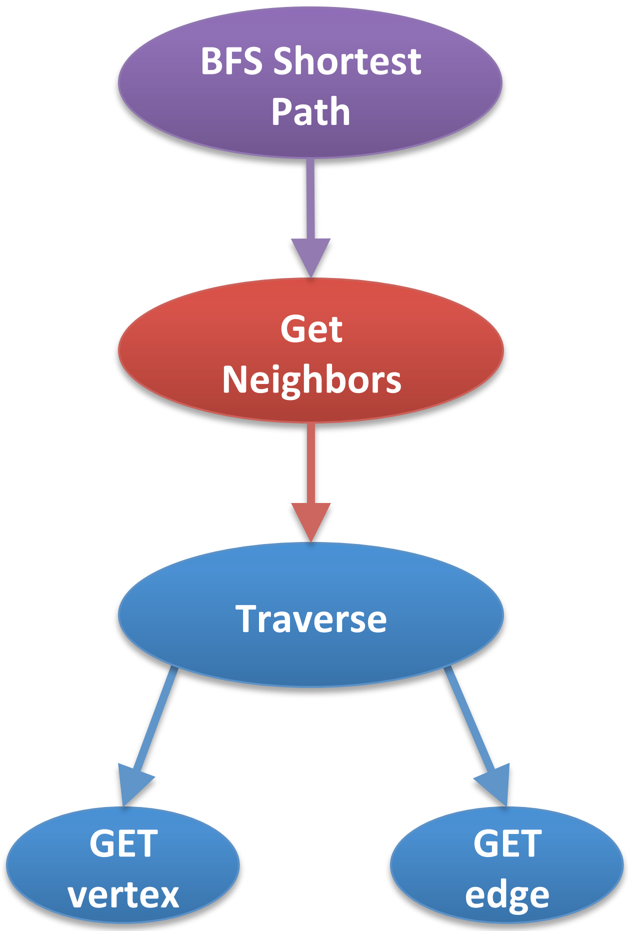
Latency-Model(Shortest Path)
$= m \times$ Latency(Get Neighbors)

Latency-Model(Get Neighbors)
$= n \times$ Latency(Traverse)

Latency-Model(Traverse)
$= 0.5\ \mu s + 3.4\ \mu s$
$= 3.9\ \mu s$

Example – Recursive Decomposition

BFS Shortest Path – Neo4j, 2 mil. node graph:



Latency-Model(Shortest Path)

= m × Latency(Get Neighbors)

Latency-Model(Get Neighbors)

= 10 × 3.9 μs = 39 μs

Actual: 32 μs

OPTIMIZATION DETECTED

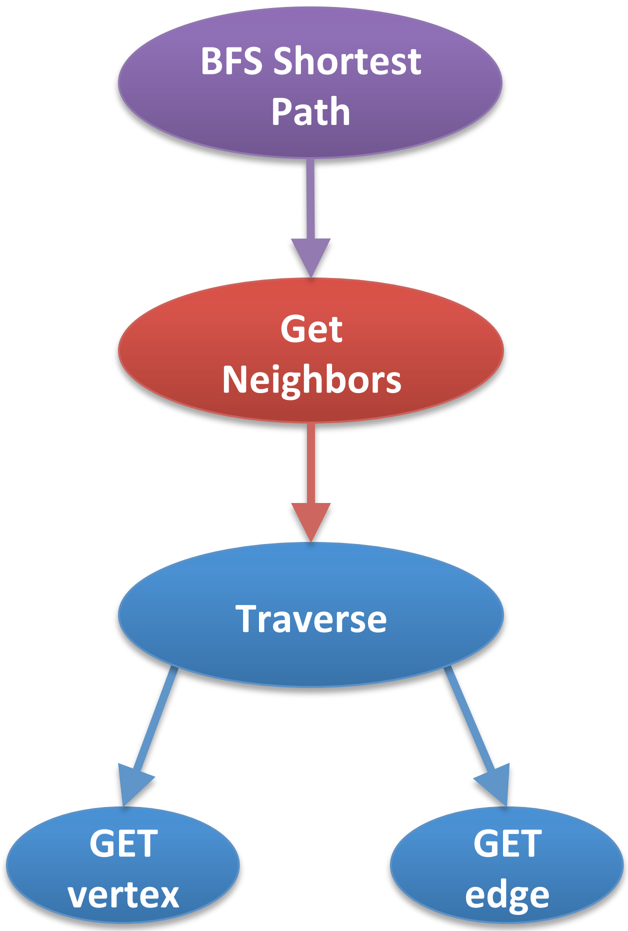
Latency-Model(Traverse)

= 0.5 μs + 3.4 μs

= 3.9 μs

Example – Recursive Decomposition

BFS Shortest Path – Neo4j, 2 mil. node graph:



NO OPTIMIZATION DETECTED

Latency-Model(Shortest Path)

= 523,000 × 32 µs = 35.6 s

Actual: 38.3 s

Latency-Model(Get Neighbors)

= 10 × 3.9 µs = 39 µs

Actual: 32 µs

OPTIMIZATION DETECTED

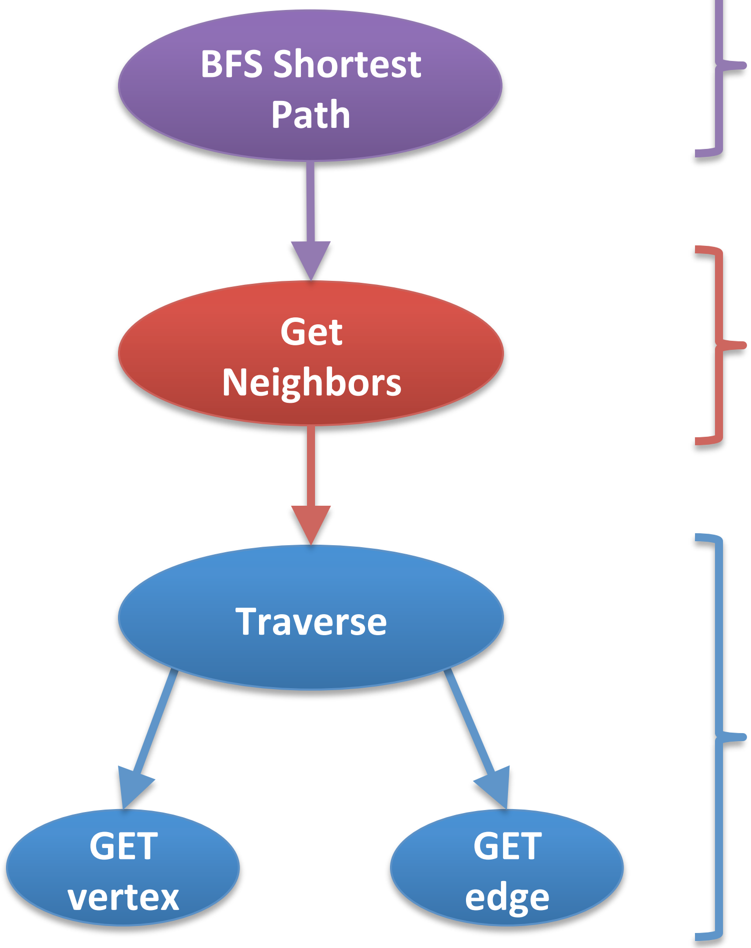
Latency-Model(Traverse)

= 0.5 µs + 3.4 µs

= 3.9 µs

Types of Operations

BFS Shortest Path:



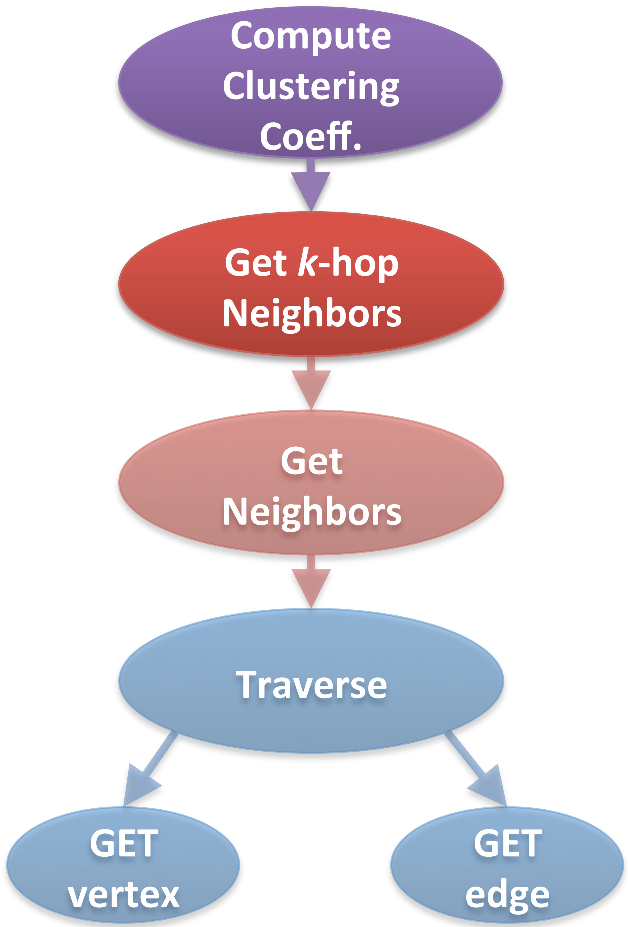
Algorithms: Higher-level operations; often not part of the graph API.

Graph Operations: Common building blocks for higher level operations.

Micro-Operations: Low-level operations that do not further decompose or that cannot be measured directly (and thus must be modeled).

Another Decomposition Example

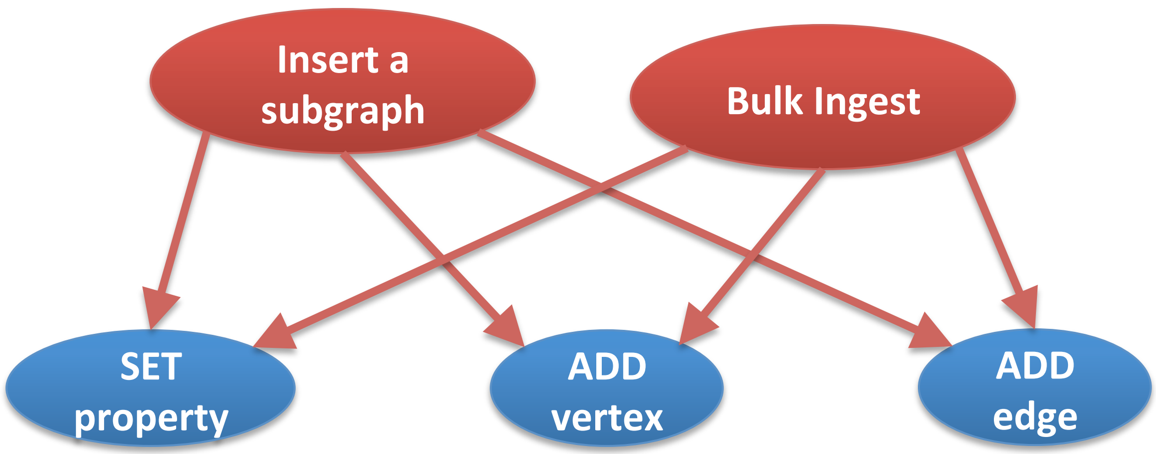
Clustering Coefficients:



- Computing a clustering coefficients (i.e., triangle counting) involves getting k-hop neighborhoods for k = 2
- "Get k-hop neighbors" gets all neighbors that are at most k hops away from a given starting vertex
- (We have already seen "Get Neighbors" before)

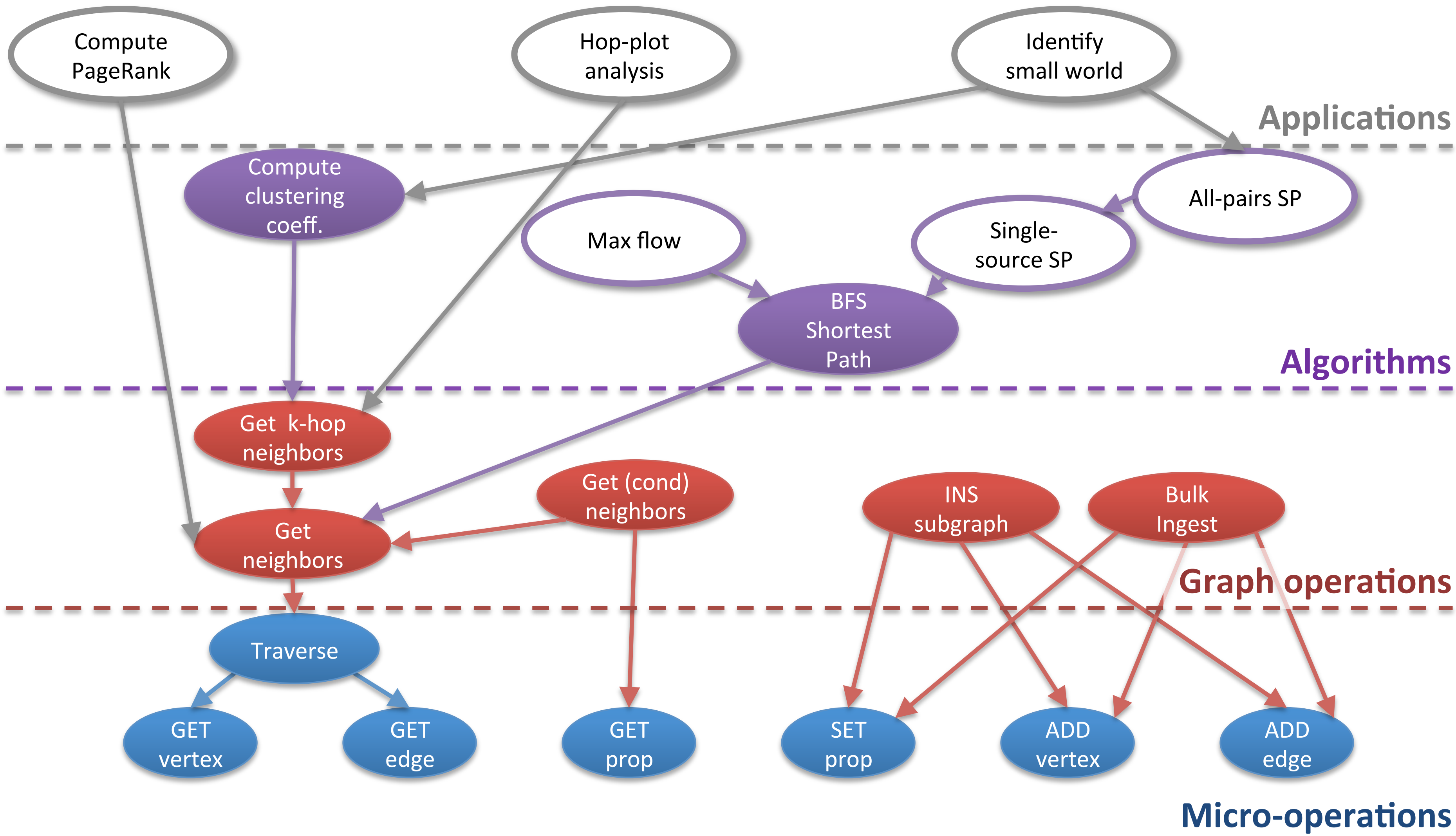
Writes

Ingest:



- Inserting a subgraph into a database is a combination of add vertex, add edge, and set edge or vertex property micro-operations
- Performing one ingest at a time is often inefficient, so databases frequently provide optimized bulk ingest

Operation Decomposition Summary



Outline

1. Introduction
2. Methodology
3. Implementation
4. Selected Results
5. Conclusion

Implementation

- Started with choosing the Blueprints API – a uniform Java API for accessing property graphs (graphs with properties on nodes and edges)

- Benchmark and all tools implemented in Java

Interfacing with Databases

- Blueprints – The benchmark framework and the reference implementation for each operation
- For each graph database:
  - Required: Implement a few methods (150 LOC on average)
  - Optional: Re-implement each operation in the database's native API for improved performance
- Tested with: *dex Neo4j
- During development, also BerkeleyDB and MySQL

Benchmark structure

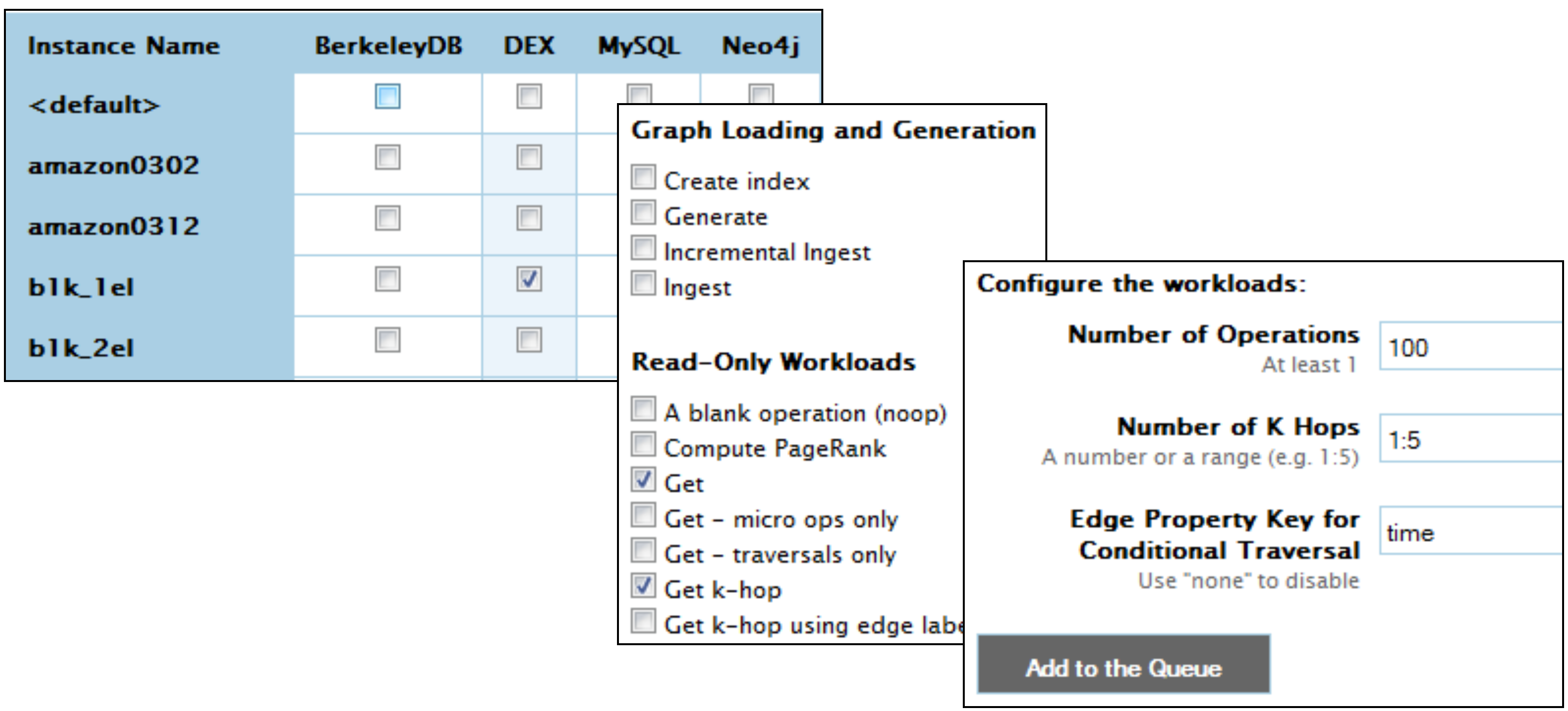
1. Initialize each operation
   – Pick random vertices, edges, and/or property values
   – A vertex can be selected uniformly at random or proportionally to its degree
2. Pollute the caches by a linear scan, to:
   – Warm up the caches, and
   – Ensure that cache contents do not come from initialization
3. Run each operation
   – Report results only for the last 10-25% of executions to make sure we report results from JIT-ed, not interpreted byte-code
   – Collect: time, memory usage, number of accessed vertices and neighborhoods, GC time, etc.

Using the Benchmark

1. Through a command-line:

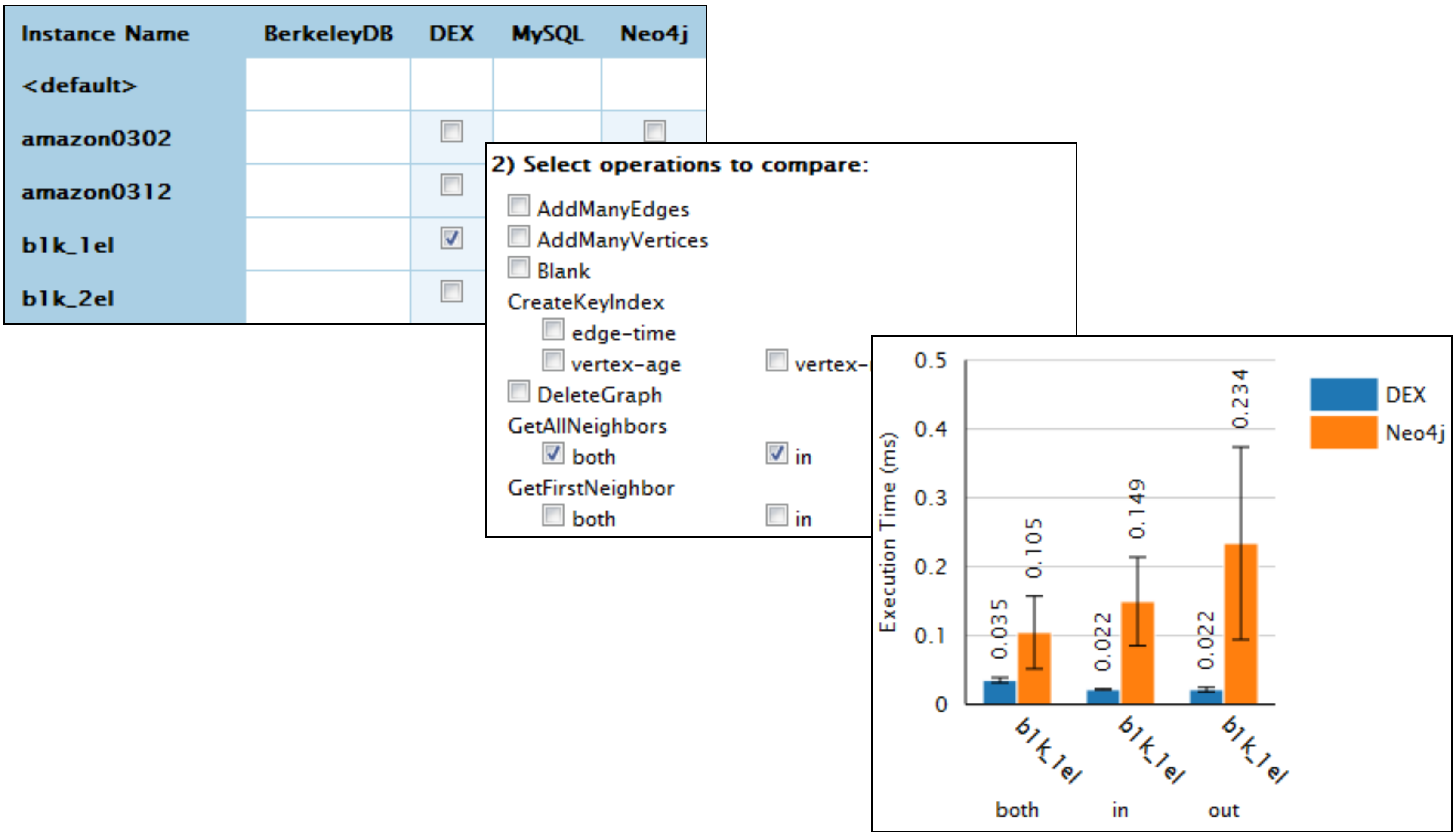
```
graphdb-bench$ ./runBenchmarkSuite.sh --dex -d b1k_1el --get
```

2. Through a web interface:



Viewing the Results

Through a web interface:



Outline

1. Introduction
2. Methodology
3. Implementation
4. Selected Results
5. Conclusion

Experimental Setup: Platform

- Databases:
  - Neo4j 1.8
  - In the paper: DEX 4.6
- Benchmarked on:
  - Intel Core i3, 3 GHz, 4 GB RAM
  - Ubuntu 12.04 LTS
  - 1 GB Cache, 1 GB JVM Heap

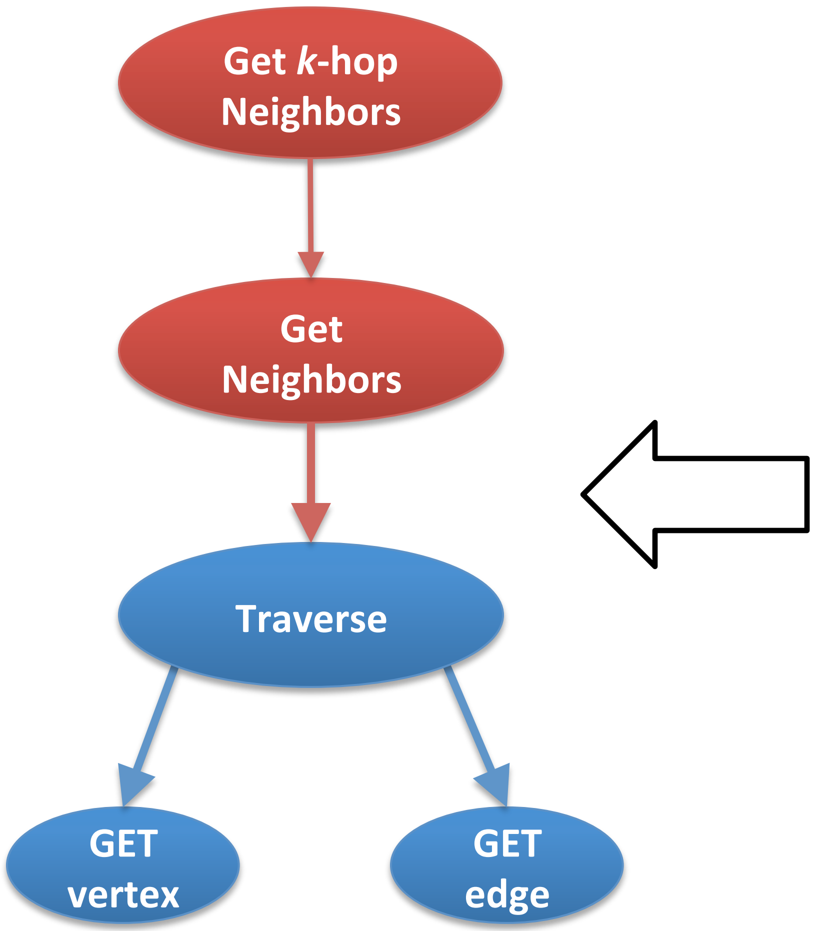
Experimental Setup: Datasets

- Datasets:
  - Barabasi graphs (small world networks), m=5
  - In the paper: Kronecker graphs (natural networks)
  - In the paper: Amazon co-purchasing networks (from SNAP)
- Four different sizes of Barabasi graphs:

# Nodes	Operating Point
1 K	Fits entirely in DB cache (Neo4j: fits entirely in the object cache)
1 mil.	Fits entirely in DB cache
2 mil.	Bigger than DB cache, but fits in memory
10 mil.	Bigger than memory

Experimental Setup: Workload

Get *k*-Hop Neighbors



Evaluate Get Neighbors using modeled Traverse

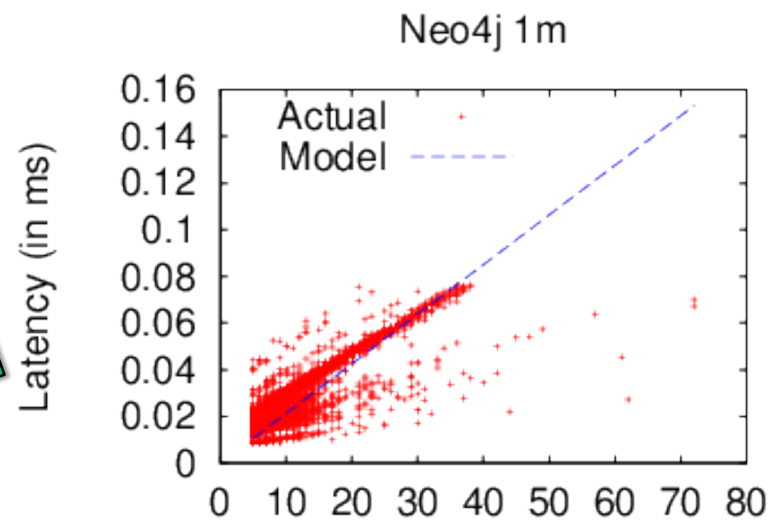
(We cannot evaluate Traverse, since we cannot measure it directly.)

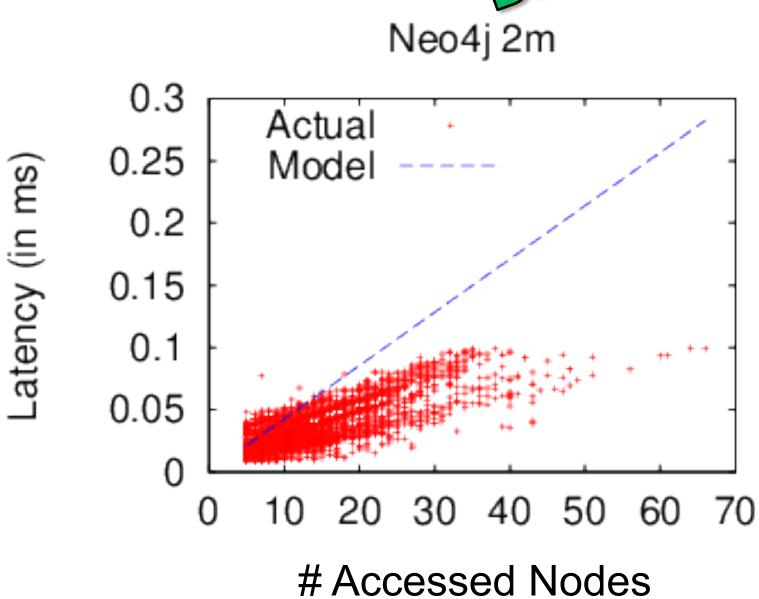
Neo4j: Get Neighbors

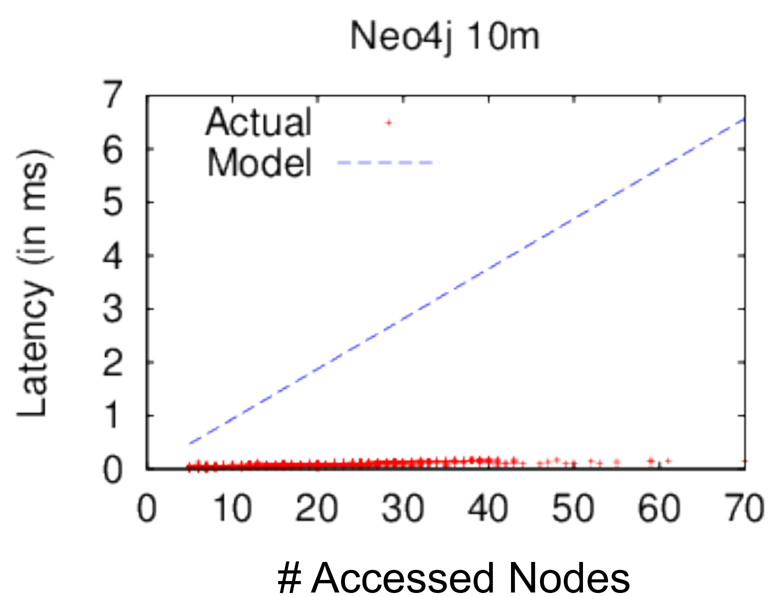
Model:

(# Accessed Vertices)
× **(**Latency(Get Vertex)
+ Latency(Get Edge)**)**

OPTIMIZATION DETECTED

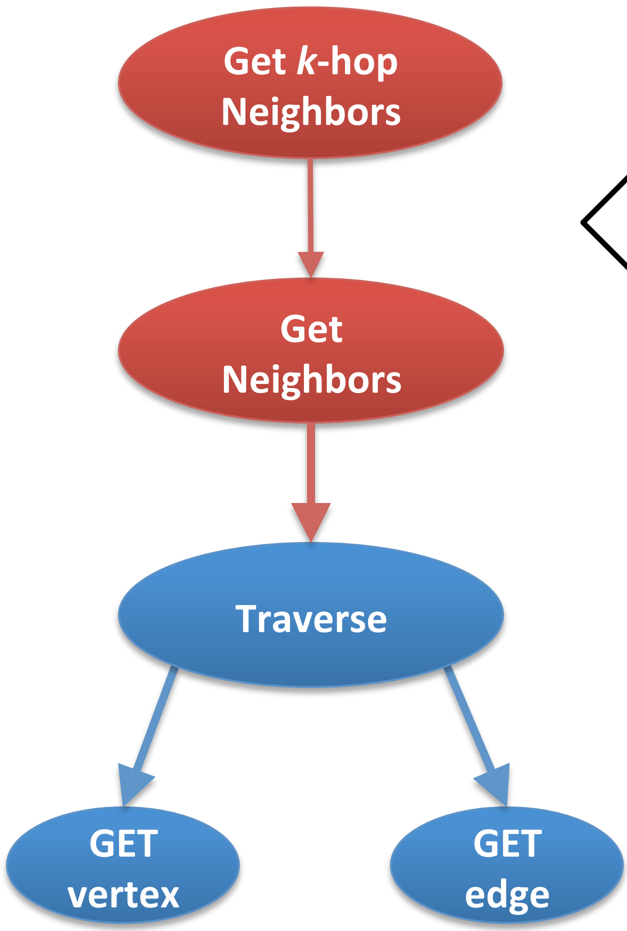






Experimental Setup: Workload

Get *k*-Hop Neighbors



Evaluate Get k-Hop Neighbors using actual Get Neighbors

OPTIMIZATION DETECTED

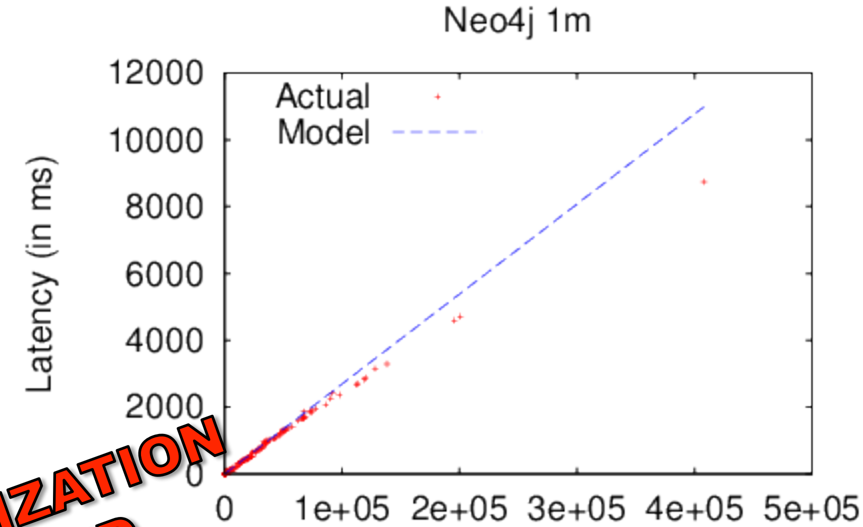
(We cannot evaluate Traverse, since we cannot measure it directly.)

Neo4j: Get *k*-Hop Neighbors

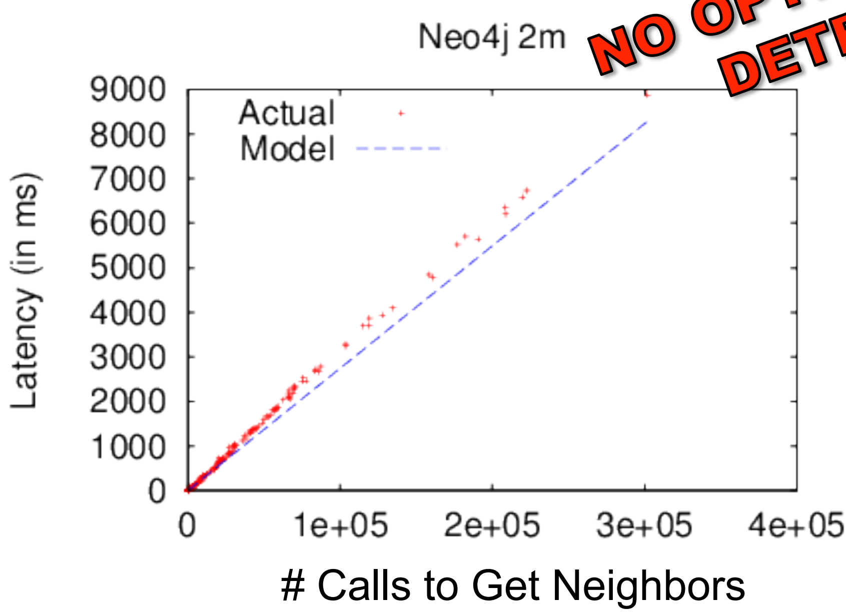
Model:

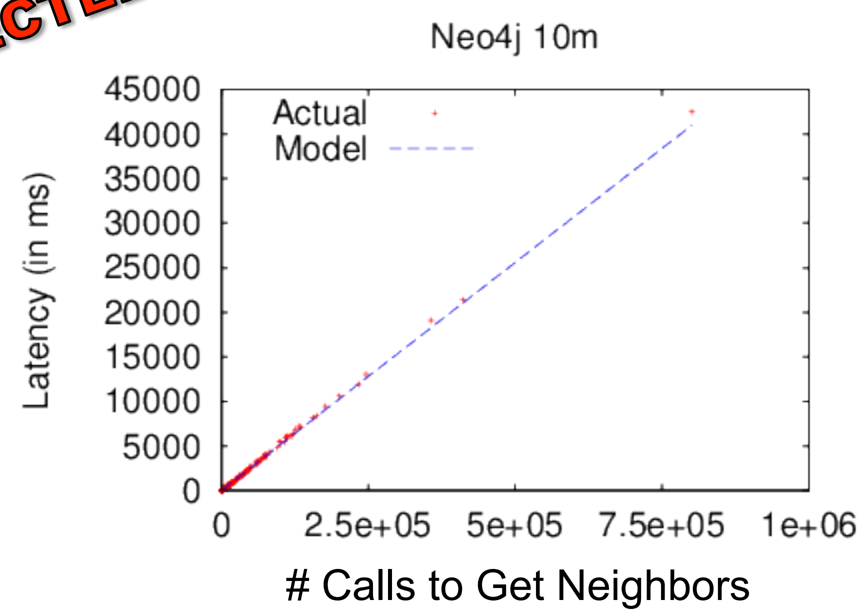
(# Calls to Get Neighbors)
× Latency(Get Neighbors)

Using actual, not modeled latency of Get Neighbors.



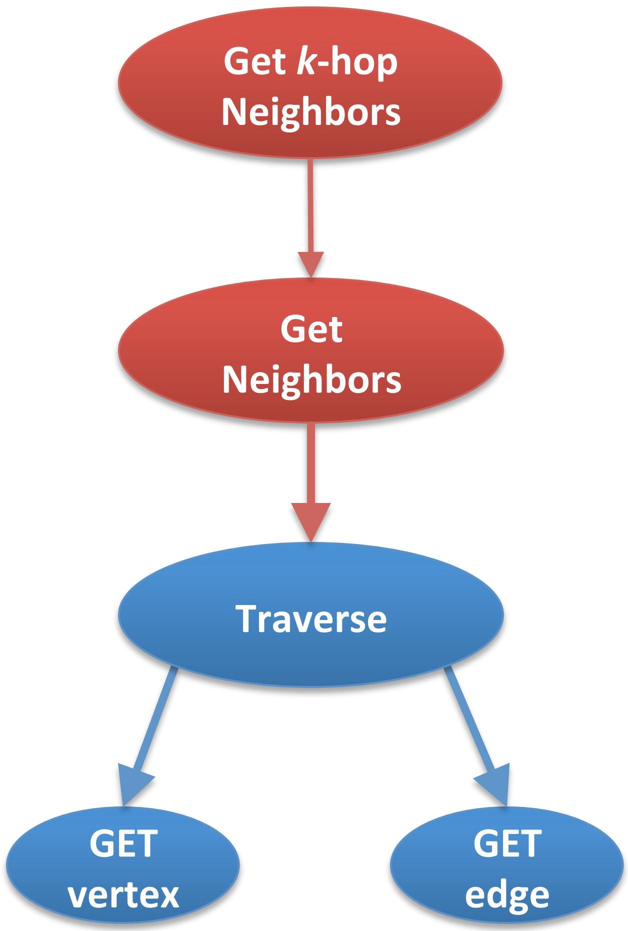
NO OPTIMIZATION DETECTED





Experimental Setup: Workload

Get *k*-Hop Neighbors



NO OPTIMIZATION DETECTED

OPTIMIZATION DETECTED

(We cannot evaluate Traverse, since we cannot measure it directly.)

Selected Results Summary

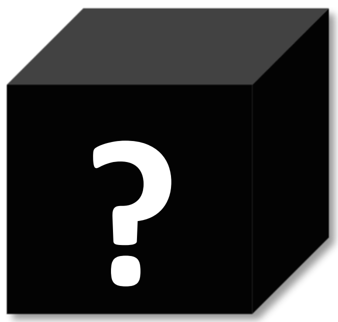
- Neo4j's neighborhood queries
  - Good optimization of individual neighborhood queries when the database does not fit in the cache
  - No optimization of multiple neighborhood queries, even when run in a BFS order

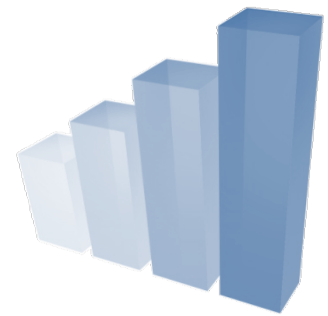
Outline

1. Introduction
2. Methodology
3. Implementation
4. Selected Results
5. Conclusion

Conclusion

Performance Introspection of Graph Databases





A black-box approach to understanding strengths and weaknesses of graph databases by comparing the actual and the modeled performance.

Availability: `code.google.com/p/pig-bench`

Contact: `pmacko at eecs.harvard.edu`

Thanks to: NSF ORACLE *sparsity technologies performance in action neotechnology