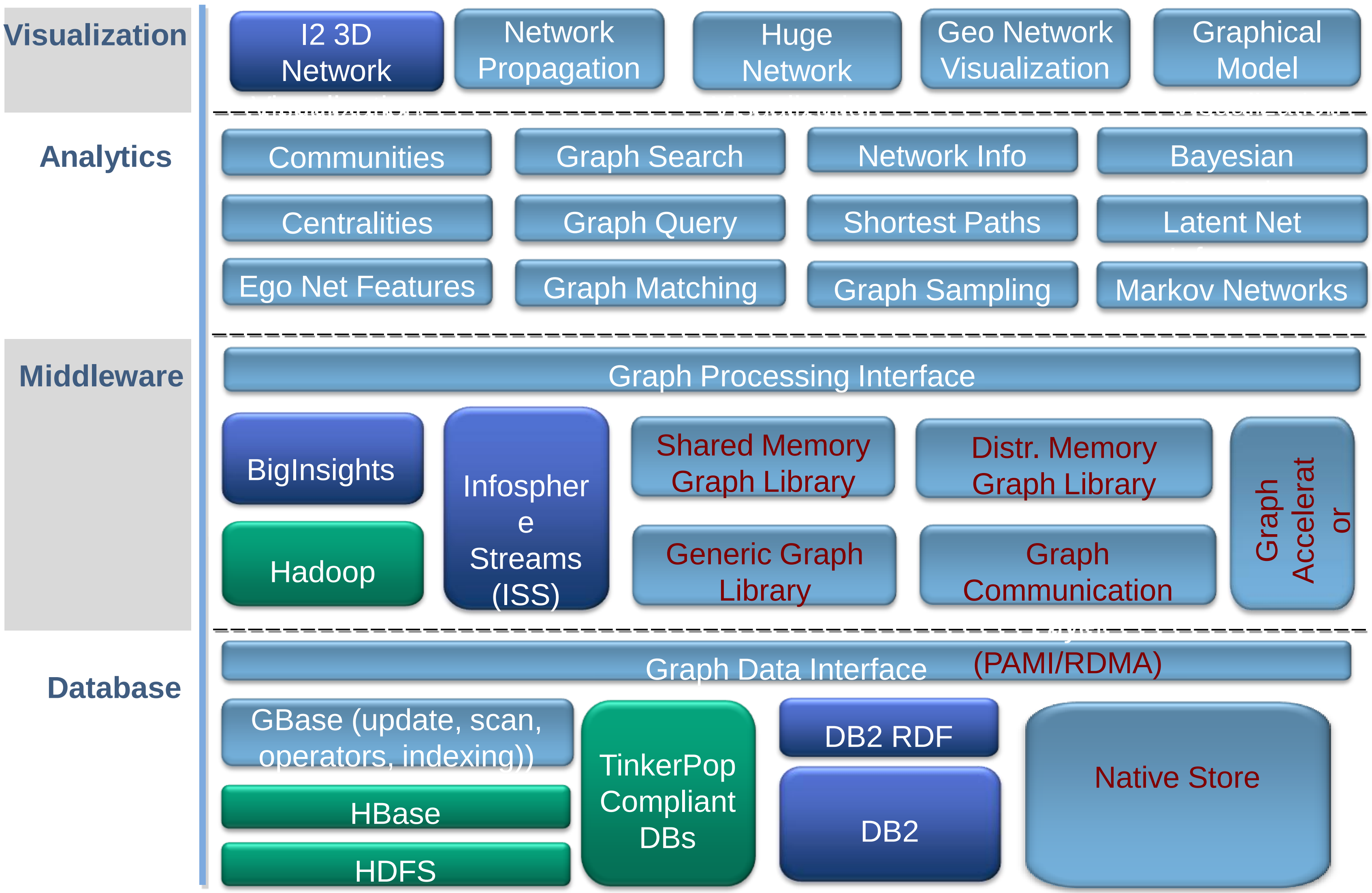
A Highly Efficient Runtime and Graph Library for Large Scale Graph Analytics

Ilie Gabriel Tanase – Research Staff Member, IBM TJ Watson

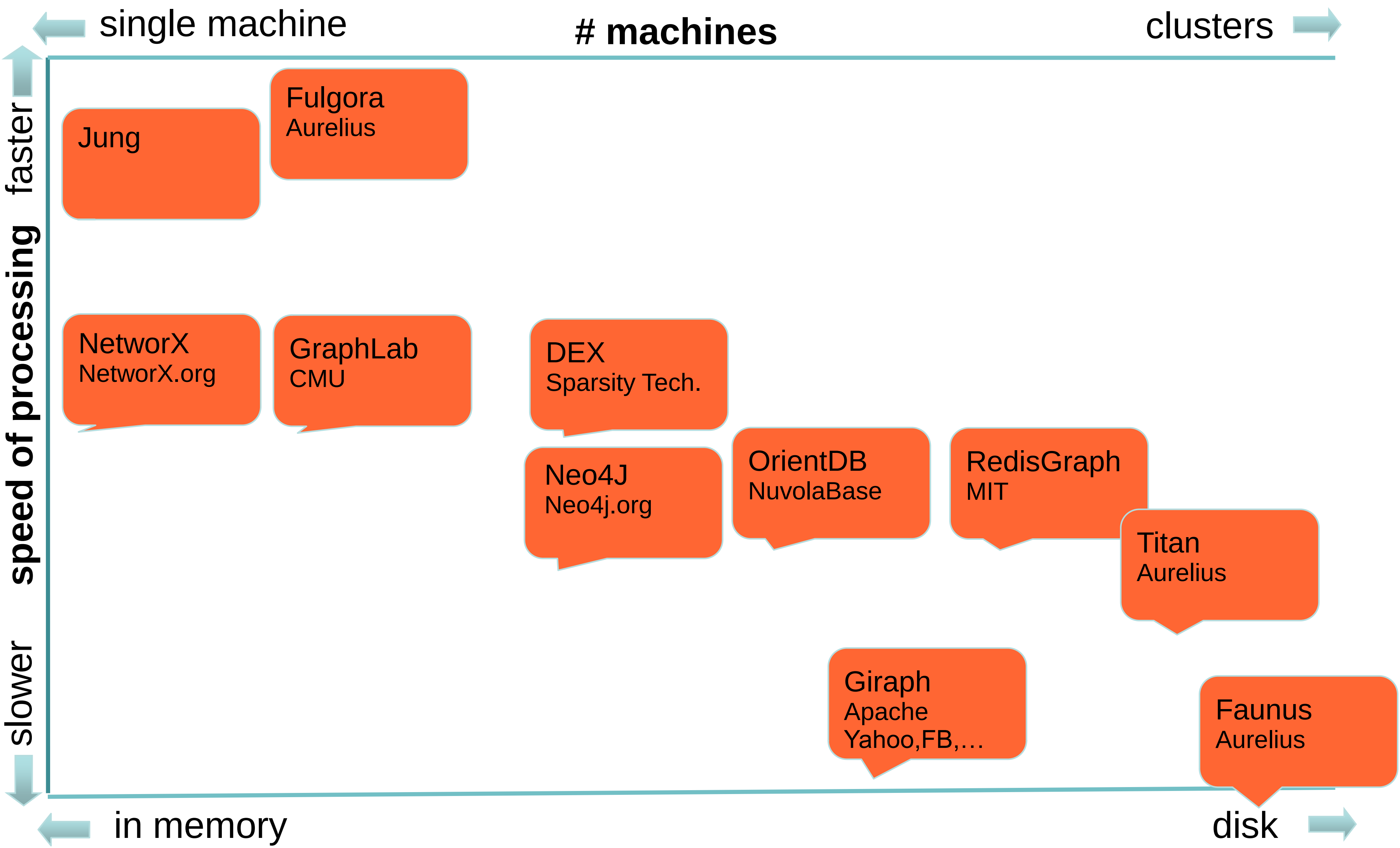
Yinglong Xia, Yanbin Liu, Wei Tan, Jason Crawford,
Ching-Yung Lin – IBM TJ Watson

Lifeng Nai – Georgia Tech

System G v1.0 Architecture



The Spectrum of Open Source Graph Technology



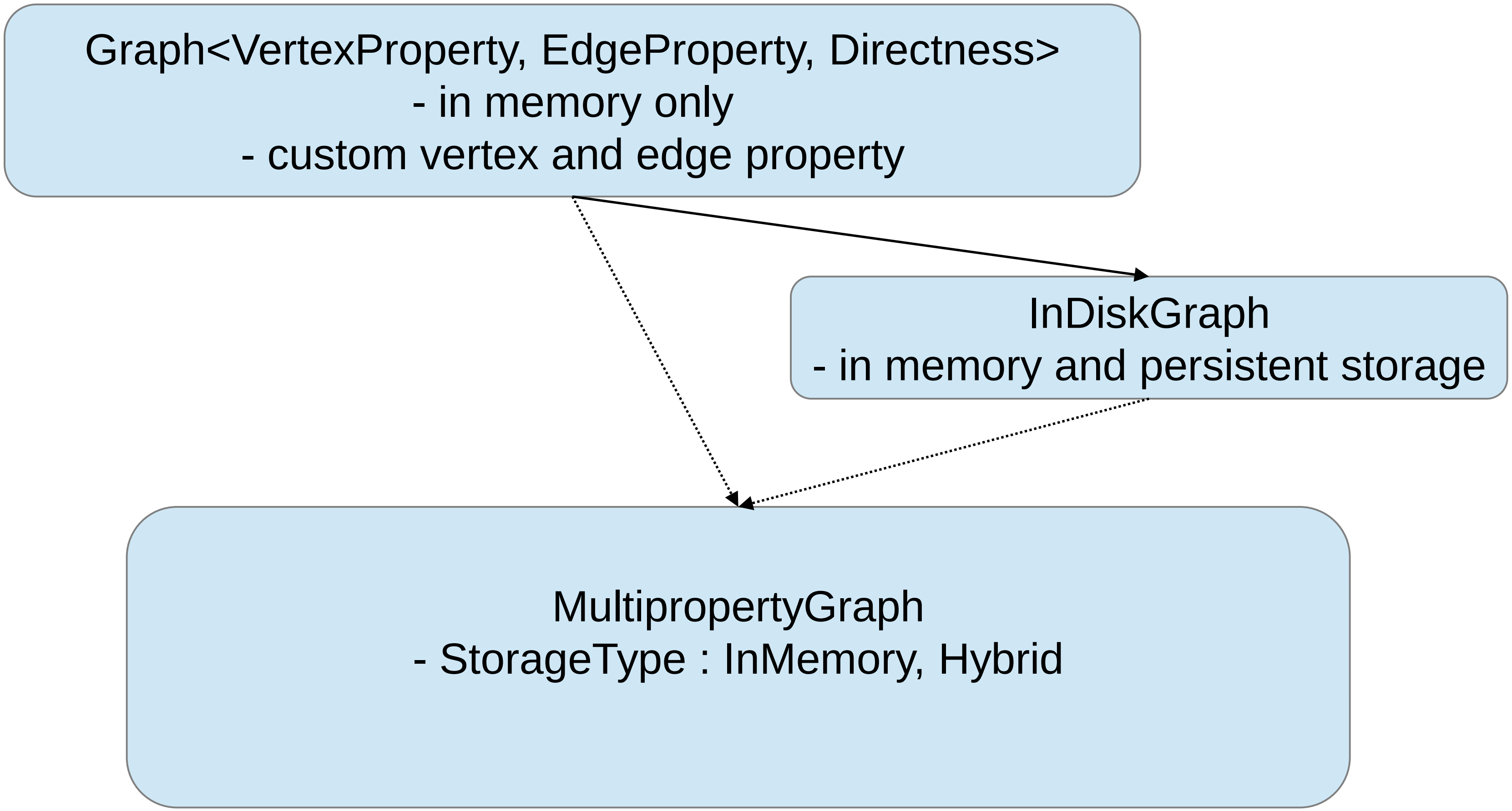
Motivation and Requirements

- Flexible Graph Datastructure
  - In memory only, persistent, both
  - Directed, undirected, directed with predecessors
  - ACID properties
- Run well on IBM machines including X86, Power, Bluegene
  - Large memory, large number of cores
  - Clusters with Infiniband or specialized networks (RDMA)
- Commercial solution

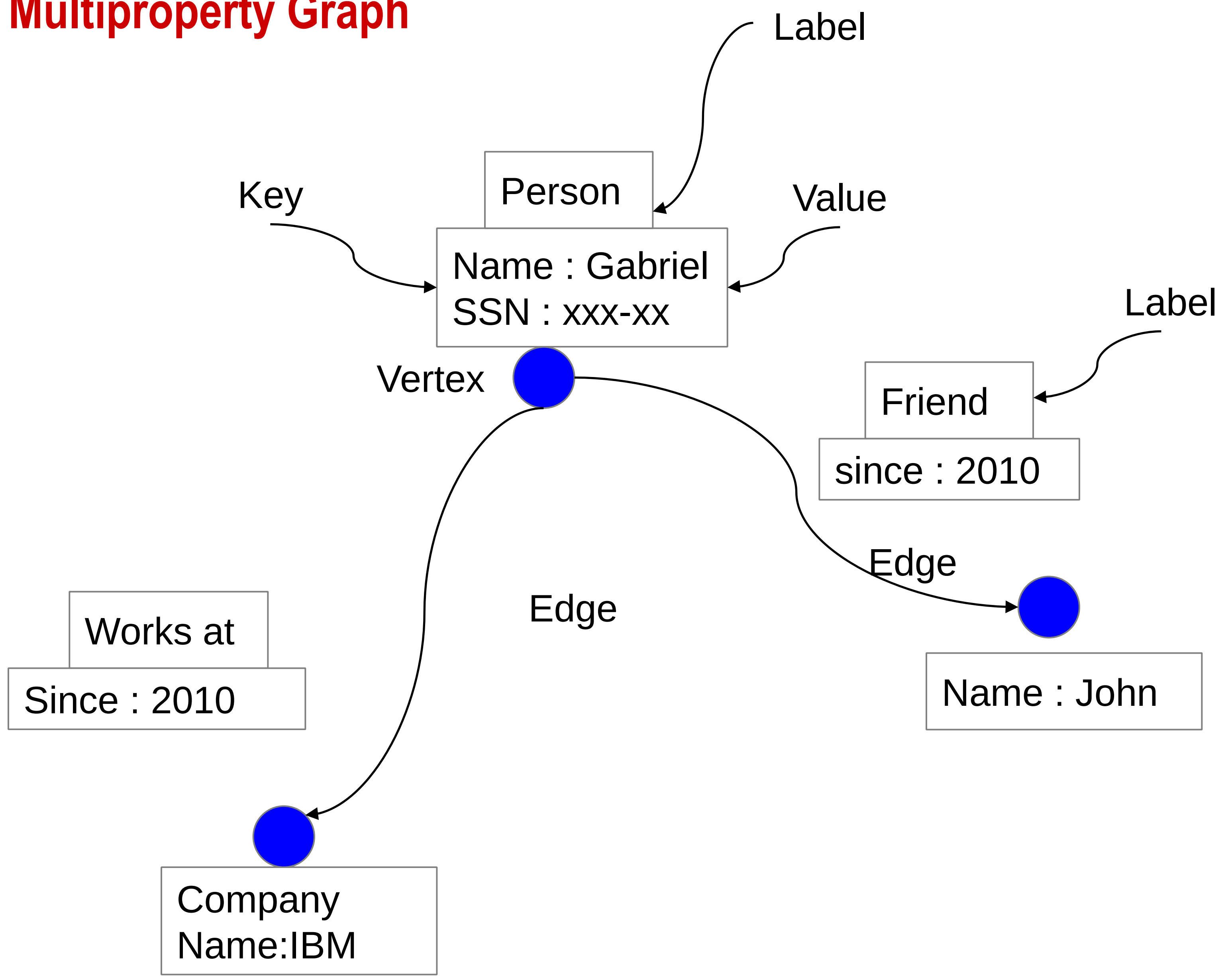
IBM Parallel Programming Library

- C++
  - Object oriented design - inheritance
  - Generic using templates
- Datastructures – Graphs, Hash Tables, Arrays
- Large shared memory
  - Concurrency
- Distributed memory clusters
  - Messaging API based on active messages and RDMA

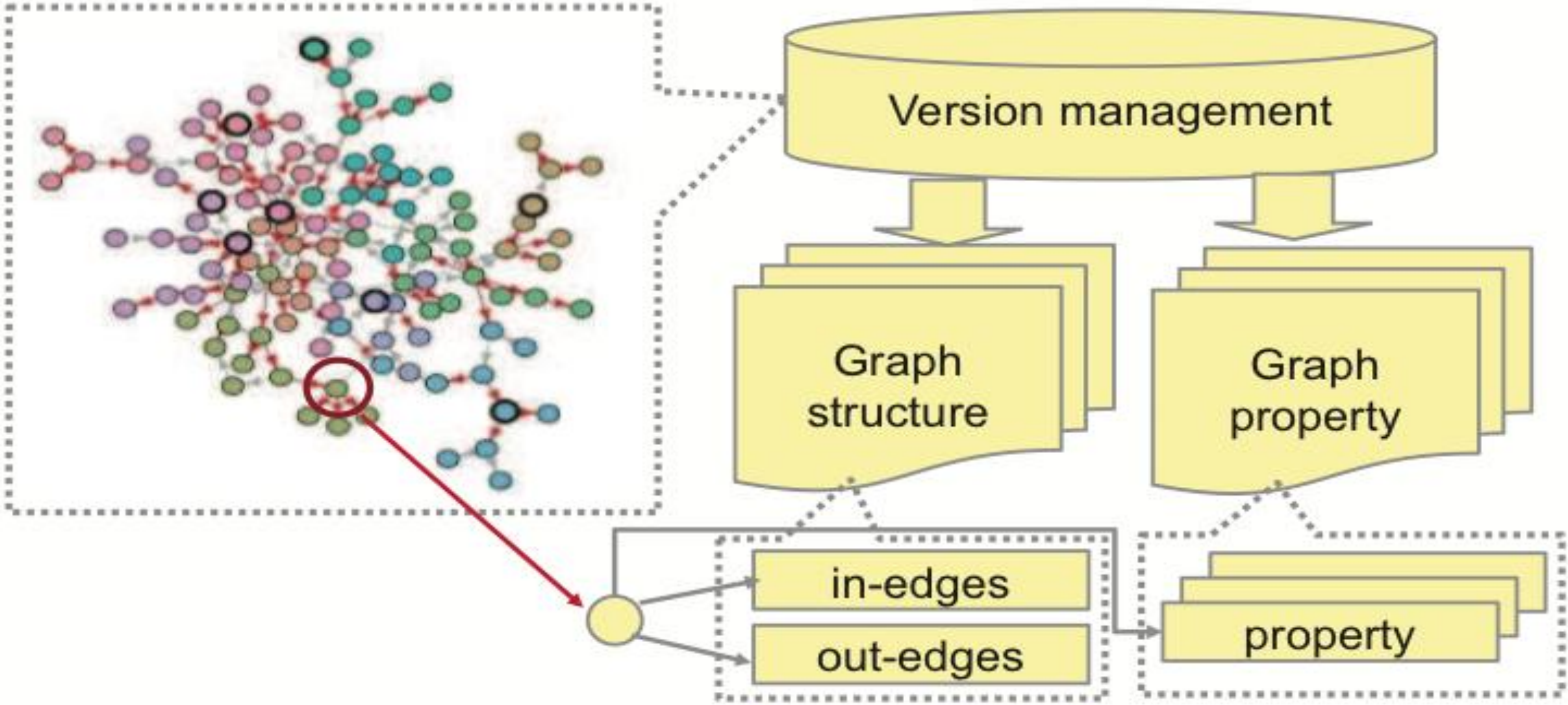
IBM PPL Graph class hierarchy



Multiproperty Graph



Persistent Storage



- Write through policy for now
- Separate structure from properties : benefits computations based on structure only
- Efficient graph loading : on demand
- Versioning

Programming Model/ Runtime

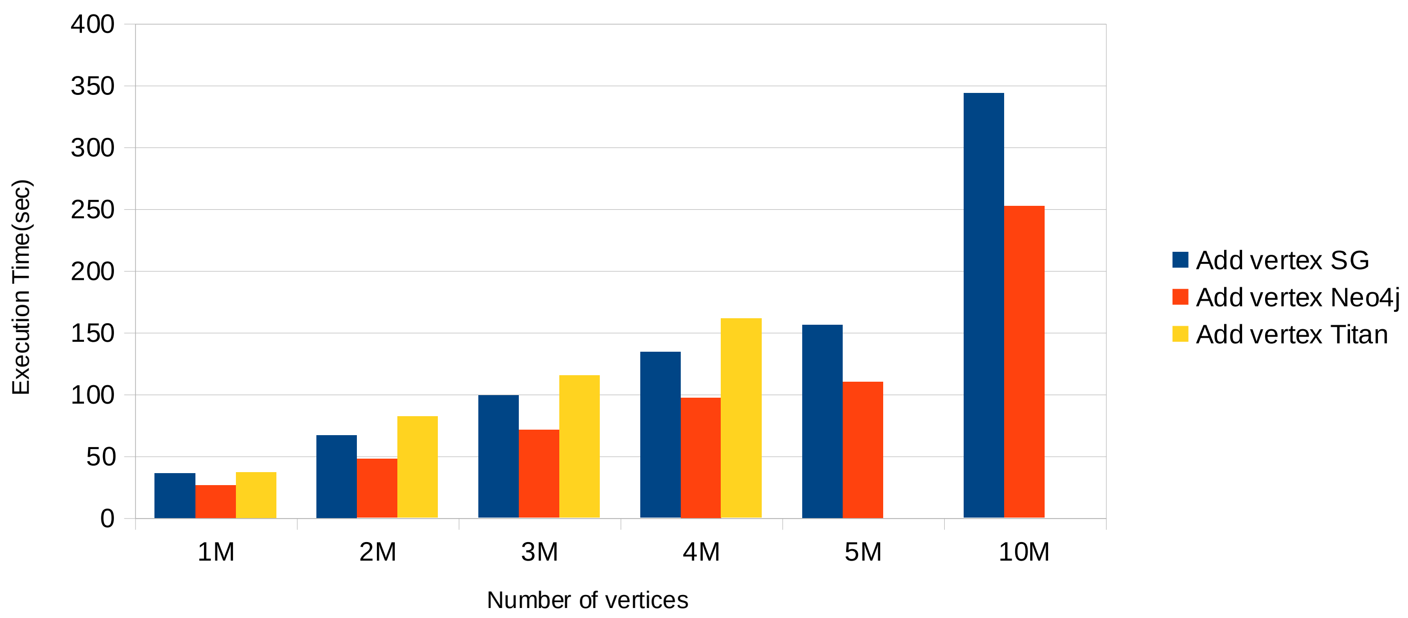
- A graph is a collection of vertices
- Each vertex maintains its in and out edges

- Parallel processing on IBMPPL graph
  - Task based model of parallelism
    - `execute_tasks(wf, num_tasks)`
    - `for_each(graph, wf);`
    - `schedule_task_graph(tg)`
  - Work stealing
  - Two level nested parallelism
  - Within shared memory for now

Performance Add Vertex

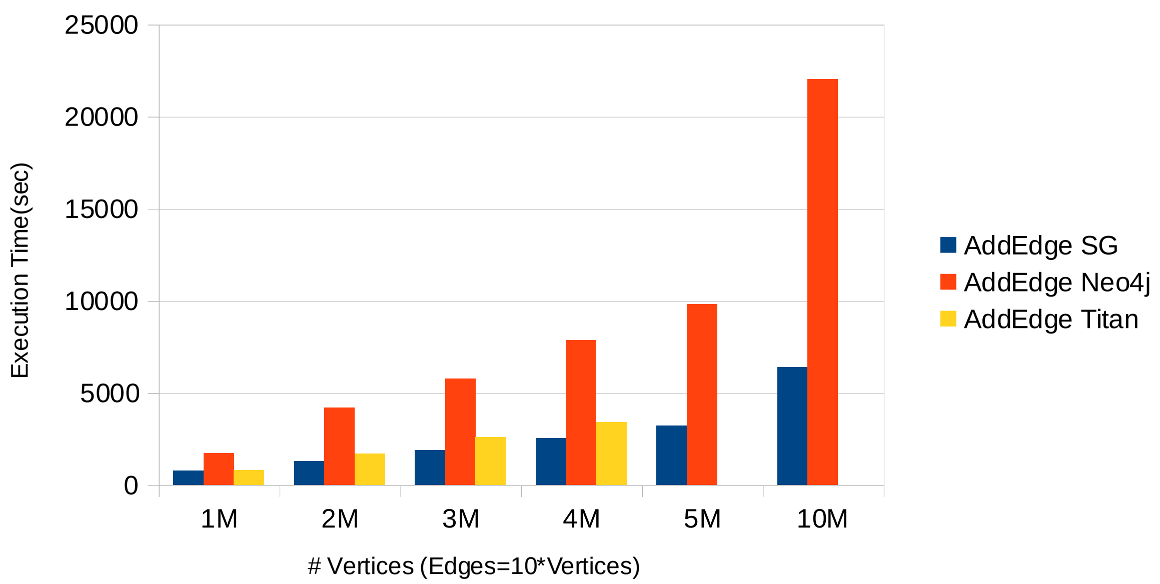
- Add vertices and for each vertex add a property
- Indexed
  - `v=add_vertex(); v.set_property("name", "vertex0");`
- Titan with Berkeley DB backend

Intel Haswell 24 core 2.7GHz and 256 GB, SSD



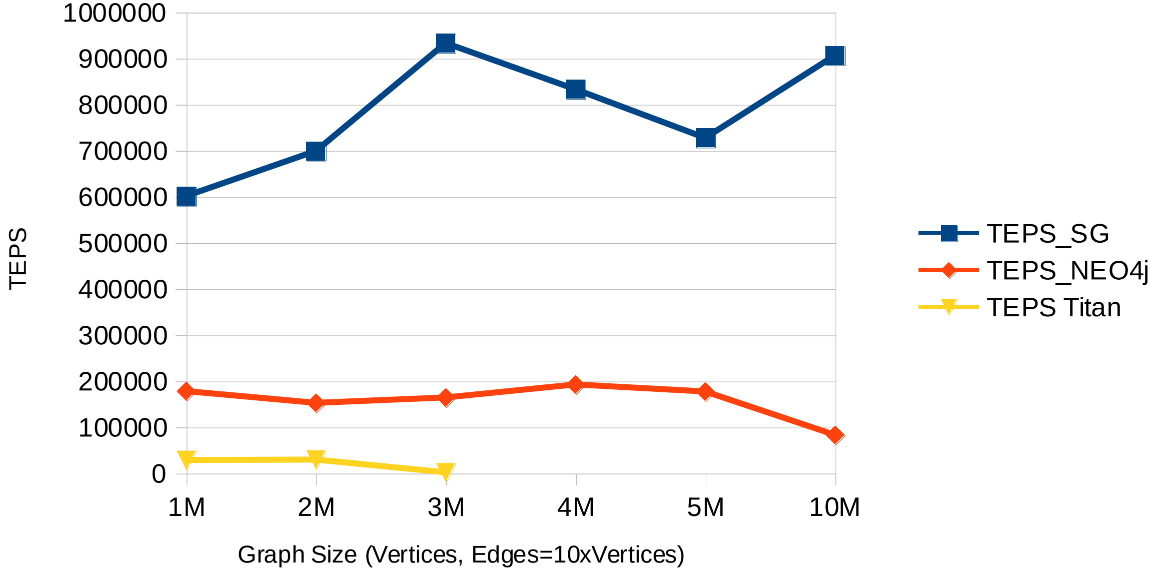
Performance Add Edge

- Add edges randomly
- The source and target are specified as vertex properties
- add_edge("vertex0", "vertex7")
  - Index lookup



Performance - Query

- For a given vertex collect all its neighbors up to depth=3
  - ~1000 edges traversed per query



Query 2 - find the newest 20 posts from your friends

```
MATCH (:Person {id:{person_id}})-[:KNOWS]-(friend:Person)<-[:HAS_CREATOR]-(post:Post)
WHERE post.creationDate<={max_date}
RETURN friend.id AS personId, friend.firstName AS personFirstName, friend.lastName AS personLastName,
    post.id AS postId, post.content AS postContent, post.creationDate AS postDate
ORDER BY postDate DESC
LIMIT 20
```

Query 4 - new topics

Find the top 10 most popular topics/tags (by the number of comments and posts) that your friends have been talking about in the last x hours.

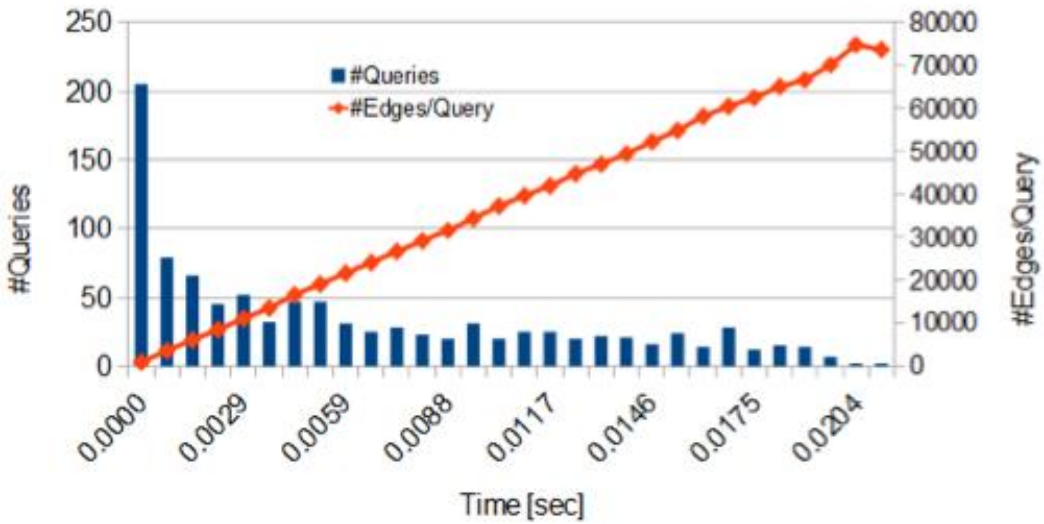
```
MATCH (:Person {id:{person_id}})-[:KNOWS]-(friend:Person)
MATCH (friend)<-[:HAS_CREATOR]-(post:Post)
WHERE post.creationDate>={min_date} AND post.creationDate<={max_date}
MATCH (post)-[HAS_TAG]->(tag:Tag)
WITH DISTINCT tag, collect(tag) AS tags
RETURN tag.name AS tagName, length(tags) AS tagCount
ORDER BY tagCount DESC
LIMIT 10
```

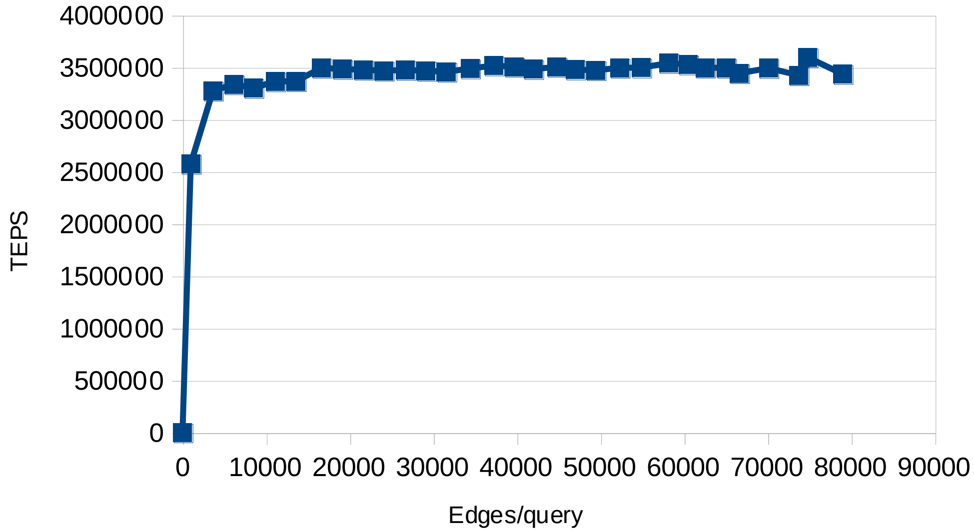
RDF Graph Construction

- Load the .csv files for vertices
- Load the .csv files for edges
- Construct property graph in memory only

Vertices	Size	Properties	Load Time(s)
Person	100,000	8	0.45
Post	54,784,723	7	188.8
Forum	3,676,271	3	10.6
Place	5,130	3	0.01
Tag	12,144	3	0.03
Edges			
Person-knows-person	2,887,797	0	3.21
Person-likes-post	208,241,439	0	311
Post-hasCreator-person	54,784,723	0	54
Post-hasTag-tag	42,797,703	0	34
Forum-contains-post	54,784,723	0	52

Query 2


250
200
150
100
50
0
80000
70000
60000
50000
40000
30000
20000
10000
0
#Queries
#Edges/Query
0.0000
0.0029
0.0059
0.0088
0.0117
0.0146
0.0175
0.0204
Time [sec]


4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
TEPS
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
Edges/query

Impact of Parallelism on Throughput

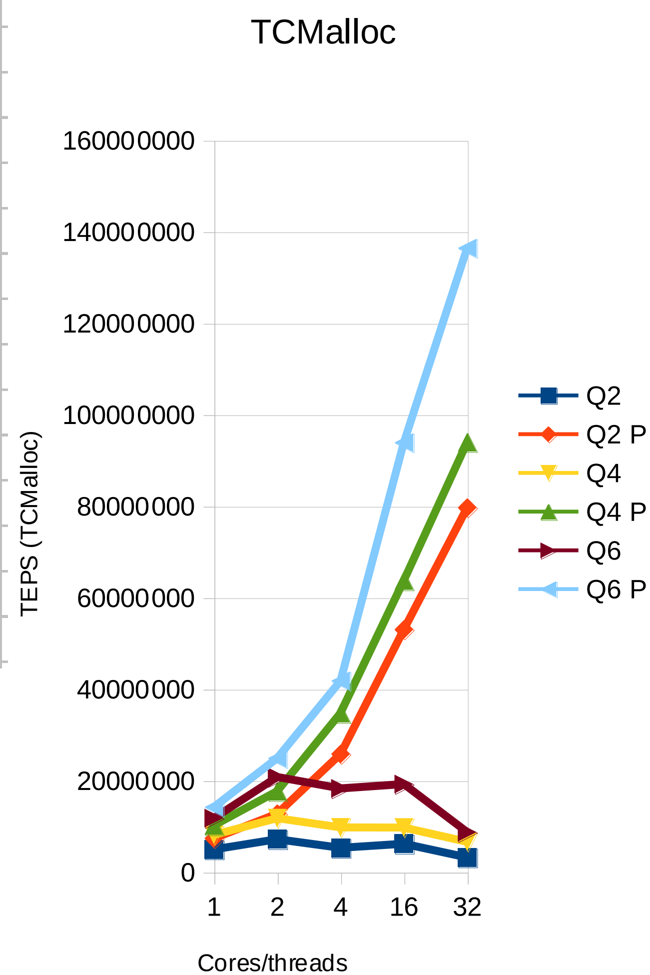
#concurrent component

Queries are processed by a single thread

Queries are assigned to threads evenly

Using Tcmalloc		All results are in TEPS			
	1	2	4	16	32
Q2	4957010	**7182290**	5305690	6189390	3205420
Q2 P	7355770	12747000	25796500	53028100	79668300
Q4	8125290	**11782600**	9761190	9727300	6624530
Q4 P	10200000	17678100	34779900	63752900	93990600
Q6	11792200	**20783200**	18282700	19239900	8730400
Q6 P	14216500	24904200	41805400	93908100	136426000

Using standard malloc	1	2	4	8	16
Q2	**3470870**	3136690	1748780	2826960	3000170
Q2 P	2227440	3844900	8253910	16493000	27355700
Q4	5332460	**6309400**	5382640	5415780	4999060
Q4 P	4122040	10046700	19551400	35303300	56970000
Q6	**7614090**	7599340	6715680	6296860	6467920
Q6 P	5550380	12800100	24178000	41189400	80043500



Conclusion

- Graph databases are gaining in popularity
  - Google, Facebook, Twitter, Paypal, BAML

Feature	System G Native Store	Neo4j	Titan
Back-end	Graph	Graph	Non-graph
Scaling	Yes	Moderate	Yes
Traversal efficiency	Perfect	Good	Poor
Schemaless	Yes	Yes	No
User defined function	Yes	No	Yes
Performance-critical App.	Perfect	Good	Poor
Multi-language APIs	C++, Java, Python, Shell	Java, Cypher	Java, Gremlin