Columbia University

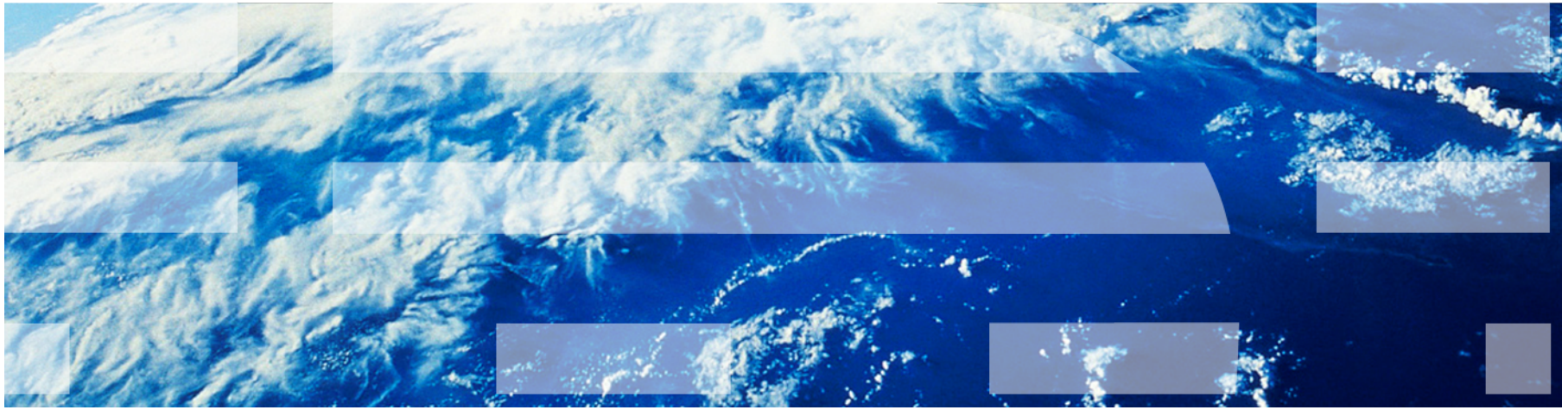
E6893 Big Data Analytics Lecture 8:

Spark Streams and Graph Computing (I)

Ching-Yung Lin, Ph.D.

Adjunct Professor, Dept. of Electrical Engineering and Computer Science

IBM Chief Scientist, Graph Computing



October 29th, 2015

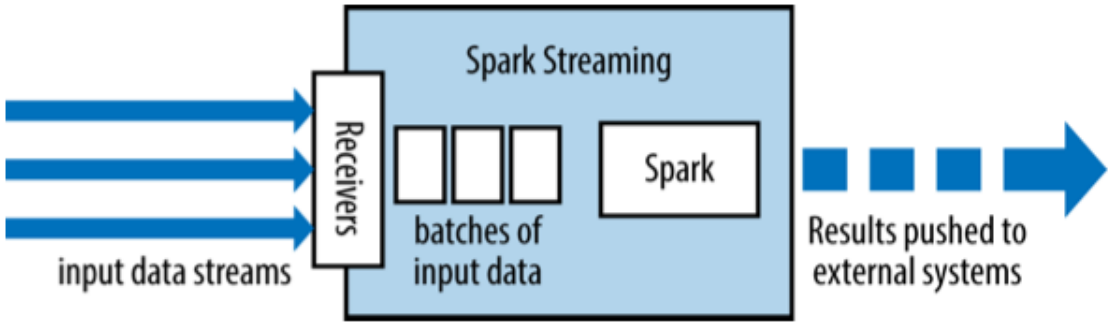
Spark Streaming

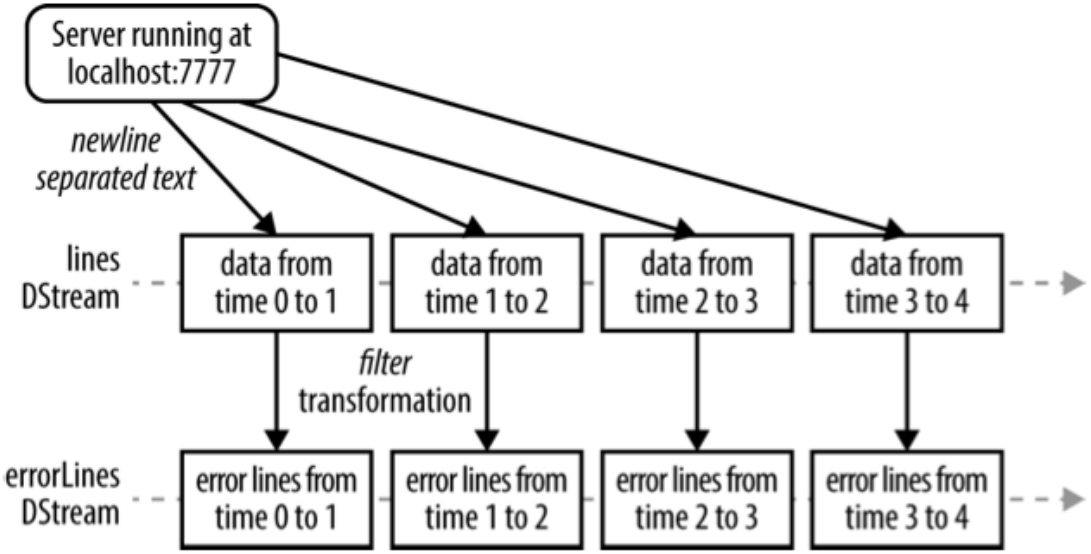
Many applications benefit from acting on data as soon as it arrives. For example, an application might track statistics about page views in real time, train a machine learning model, or automatically detect anomalies. Spark Streaming is Spark's module for such applications. It lets users write streaming applications using a very similar API to batch jobs, and thus reuse a lot of the skills and even code they built for those.

Much like Spark is built on the concept of RDDs, Spark Streaming provides an abstraction called *DStreams*, or *discretized streams*. A DStream is a sequence of data arriving over time. Internally, each DStream is represented as a sequence of RDDs arriving at each time step (hence the name "discretized").

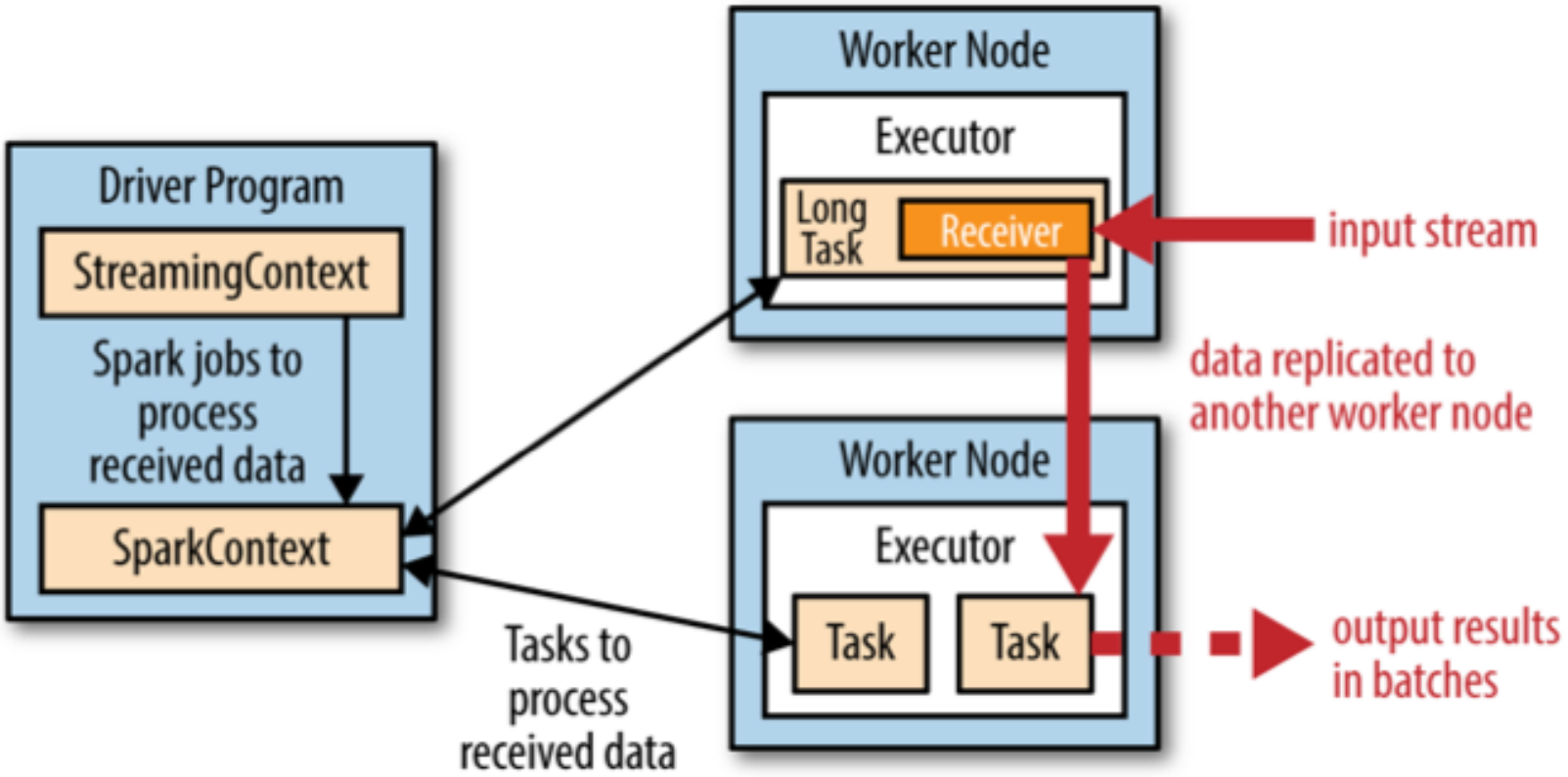
In Spark 1.1, Spark Streaming is available only in Java and Scala.
Spark 1.2 has limited Python support.

Spark Streaming architecture

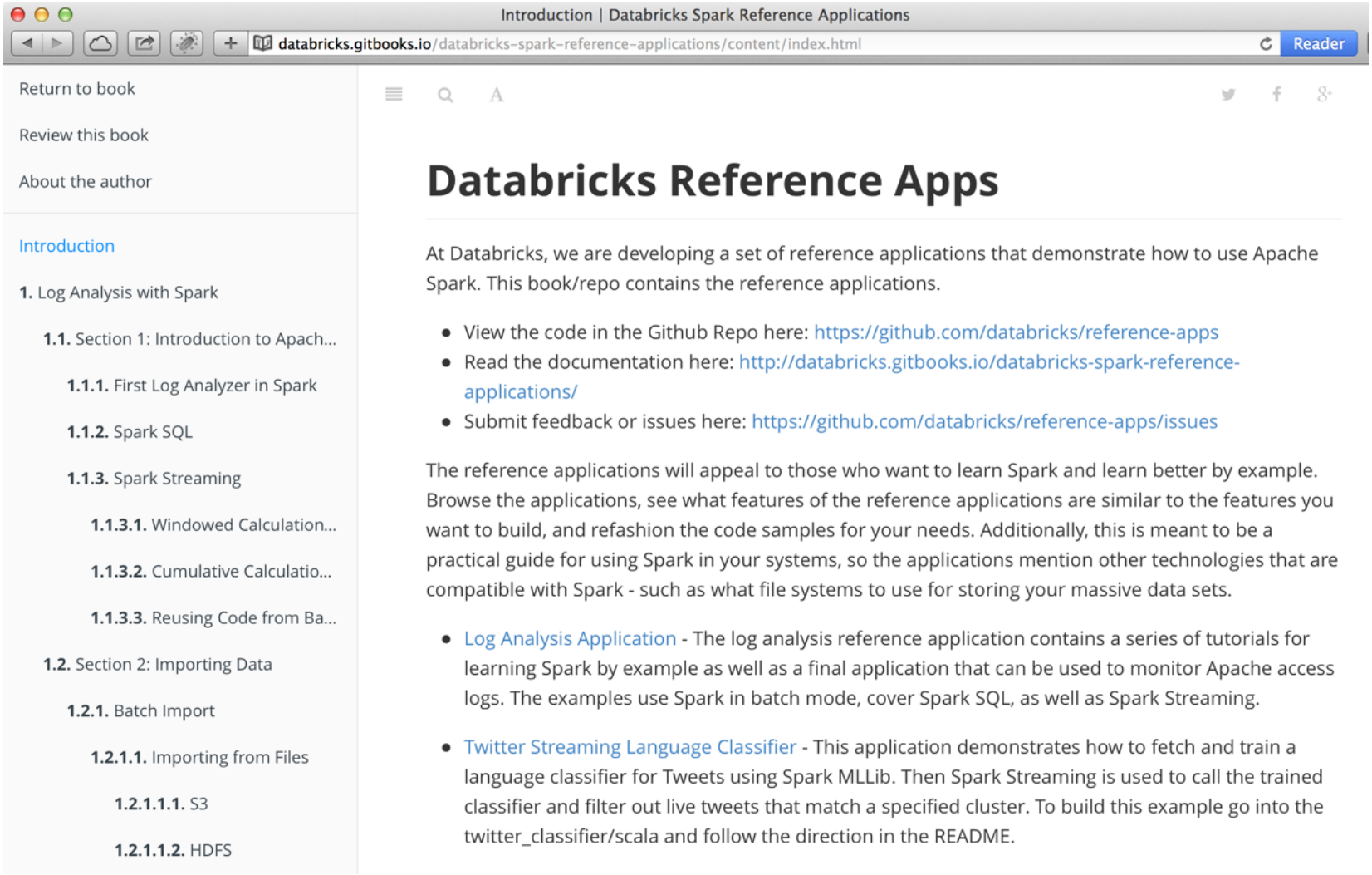




Spark Streaming with Spark's components

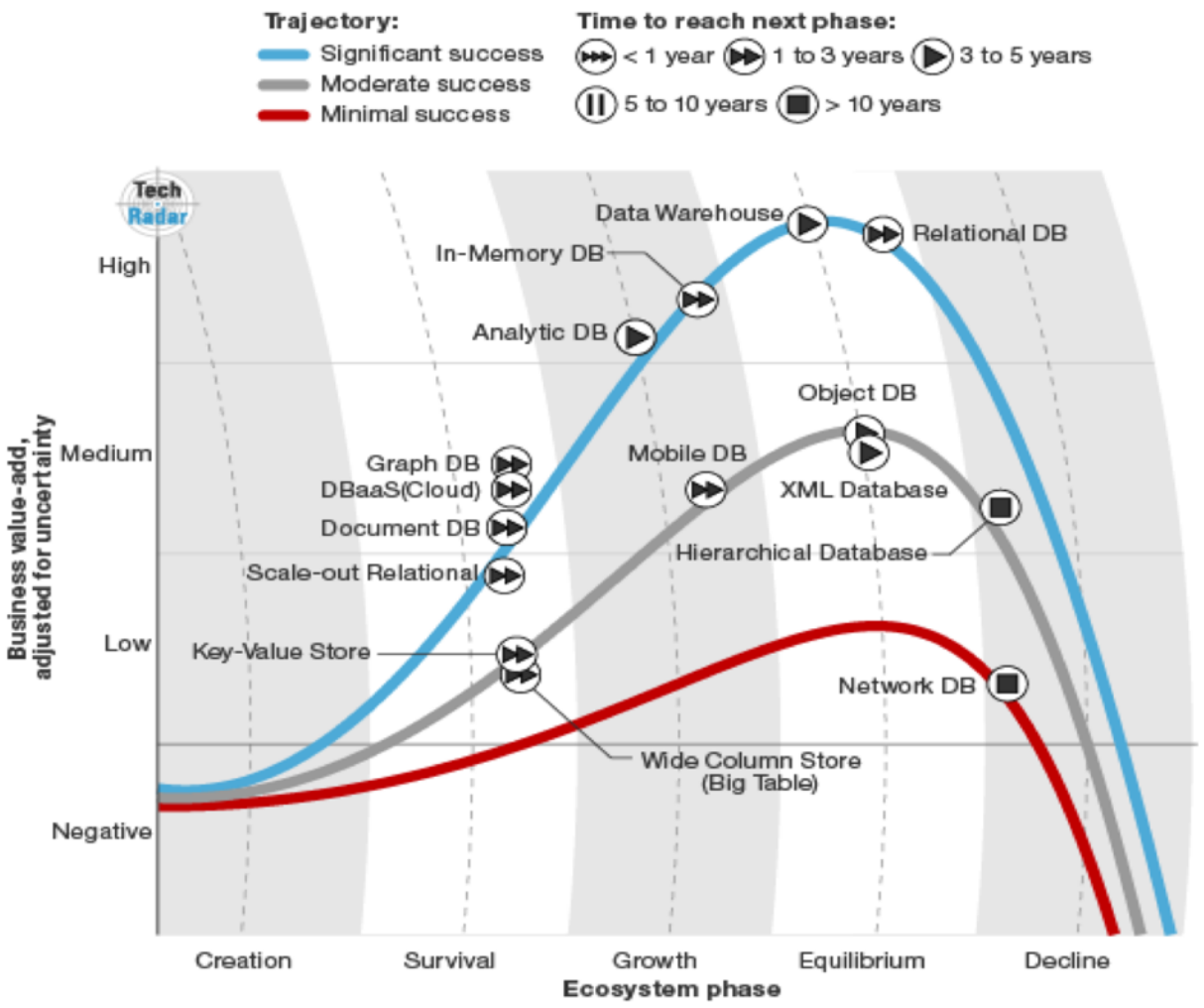


Try these examples



Forrester: Over 25% of enterprise will use Graph DB by 2017

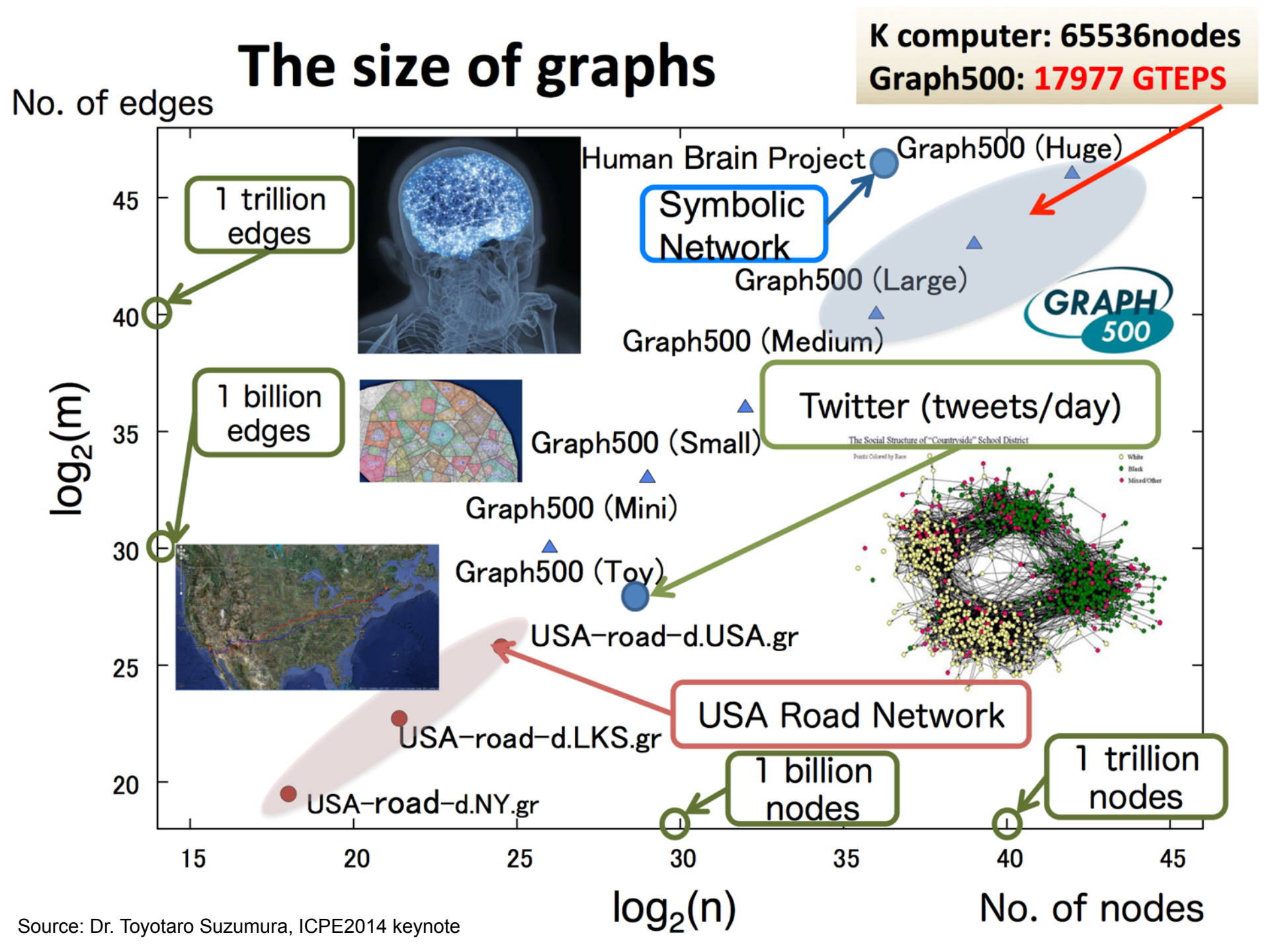
TechRadar: Enterprise DBMS, Q12014



Graph DB is in the significant success trajectory, and with the highest business value in the upcoming DBs.

The size of graphs

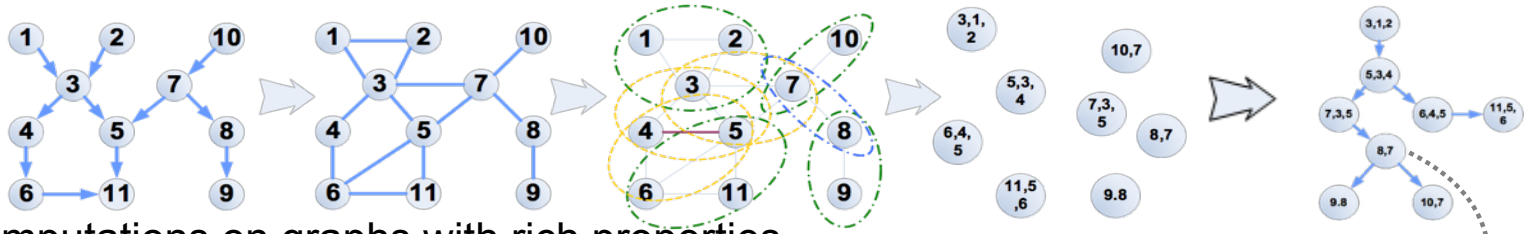
K computer: 65536nodes
Graph500: 17977 GTEPS



Source: Dr. Toyotaro Suzumura, ICPE2014 keynote

Graph Computation

- Step 1: Computations on graph structures / topologies
  - Example → converting Bayesian network into junction tree, graph traversal (BFS/DFS), etc.
  - Characteristics → Poor locality, irregular memory access, limited numeric operations

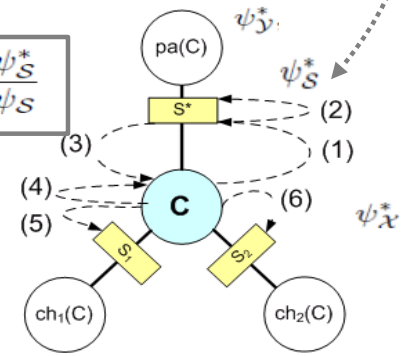


- Step 2: Computations on graphs with rich properties
  - Example → Belief propagation: diffuse information through a graph using statistical models
  - Characteristics
    - Locality and memory access pattern depend on vertex models
    - Typically a lot of numeric operations
    - Hybrid workload

$$\psi^*_{\mathcal{S}} = \sum_{\mathcal{Y}\setminus\mathcal{S}} \psi^*_{\mathcal{Y}}, \quad \psi^*_{\mathcal{X}} = \psi_{\mathcal{X}} \frac{\psi^*_{\mathcal{S}}}{\psi_{\mathcal{S}}}$$

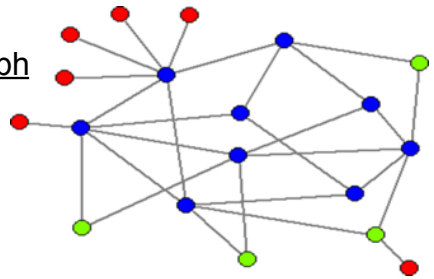
$\psi^*_{\mathcal{X}}$

λ=	0.5	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5	5.0
X=0	0.6065	0.3679	0.2231	0.1353	0.0821	0.0498	0.0302	0.0183	0.0111	0.0067
1	0.9098	0.7358	0.5578	0.4060	0.2873	0.1991	0.1359	0.0916	0.0611	0.0404
2	0.9856	0.9197	0.9197	0.8088	0.6767	0.5438	0.4232	0.3208	0.2381	0.1247
3	0.9982	0.9810	0.9344	0.8571	0.7576	0.6472	0.5366	0.4335	0.3423	0.2650
4	0.9998	0.9963	0.9814	0.9473	0.8912	0.8153	0.7254	0.6288	0.5321	0.4405
5	1.0000	0.9994	0.9994	0.9955	0.9834	0.9161	0.8576	0.7851	0.7029	0.6160
6	1.0000	0.9999	0.9991	0.9955	0.9858	0.9665	0.9347	0.8893	0.8311	0.7622
7	1.0000	1.0000	0.9998	0.9989	0.9958	0.9881	0.9733	0.9489	0.9134	0.8666
8	1.0000	1.0000	1.0000	0.9998	0.9989	0.9962	0.9901	0.9786	0.9597	0.9319
9	1.0000	1.0000	1.0000	1.0000	0.9997	0.9989	0.9967	0.9919	0.9829	0.9682
10	1.0000	1.0000	1.0000	1.0000	0.9999	0.9997	0.9990	0.9972	0.9933	0.9863

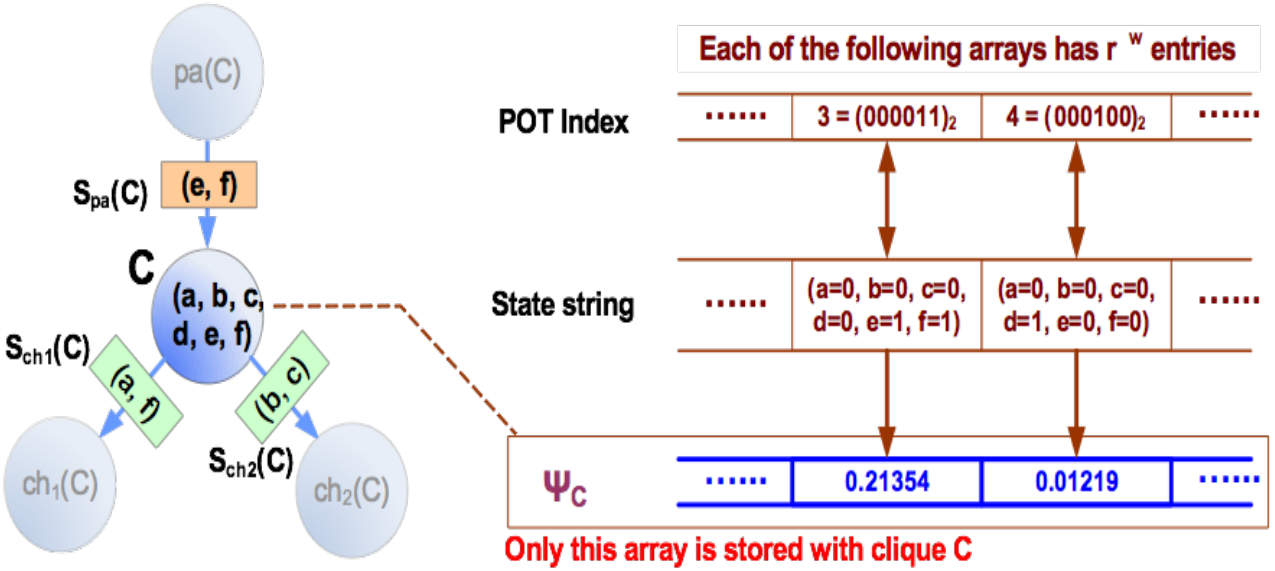


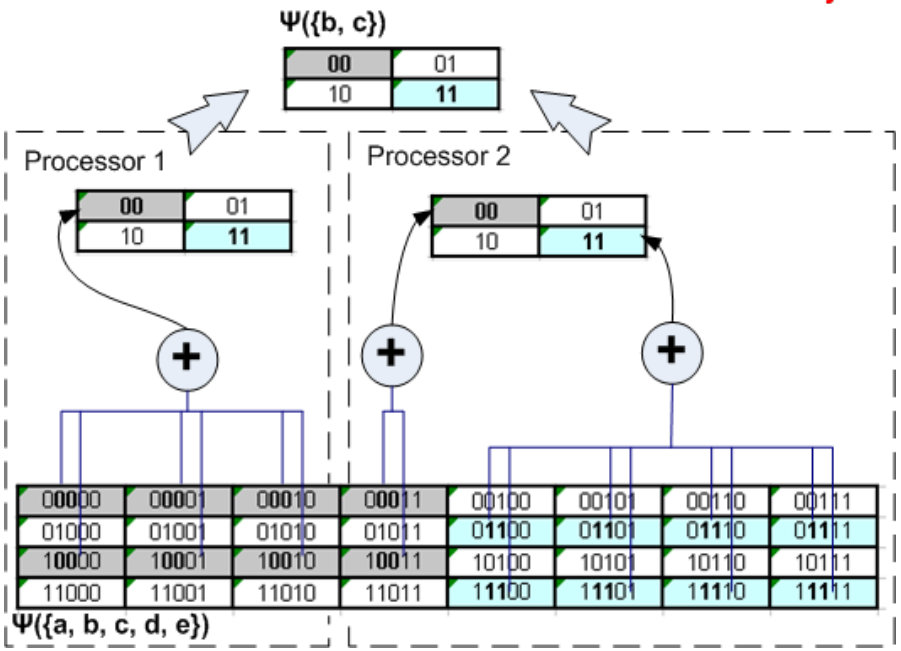
- Step 3: Computations on dynamic graphs
  - Characteristics
    - Poor locality, irregular memory access
    - Operations to update a model (e.g., cluster, sub-graph)
    - Hybrid workload

3-core subgraph



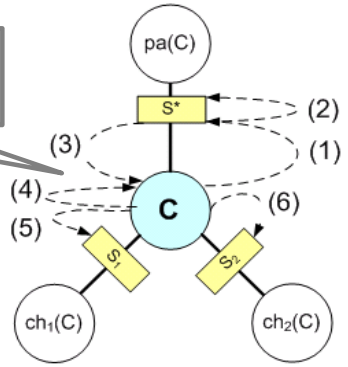
Parallel Inference – Parallelism at Multiple Granularity Levels





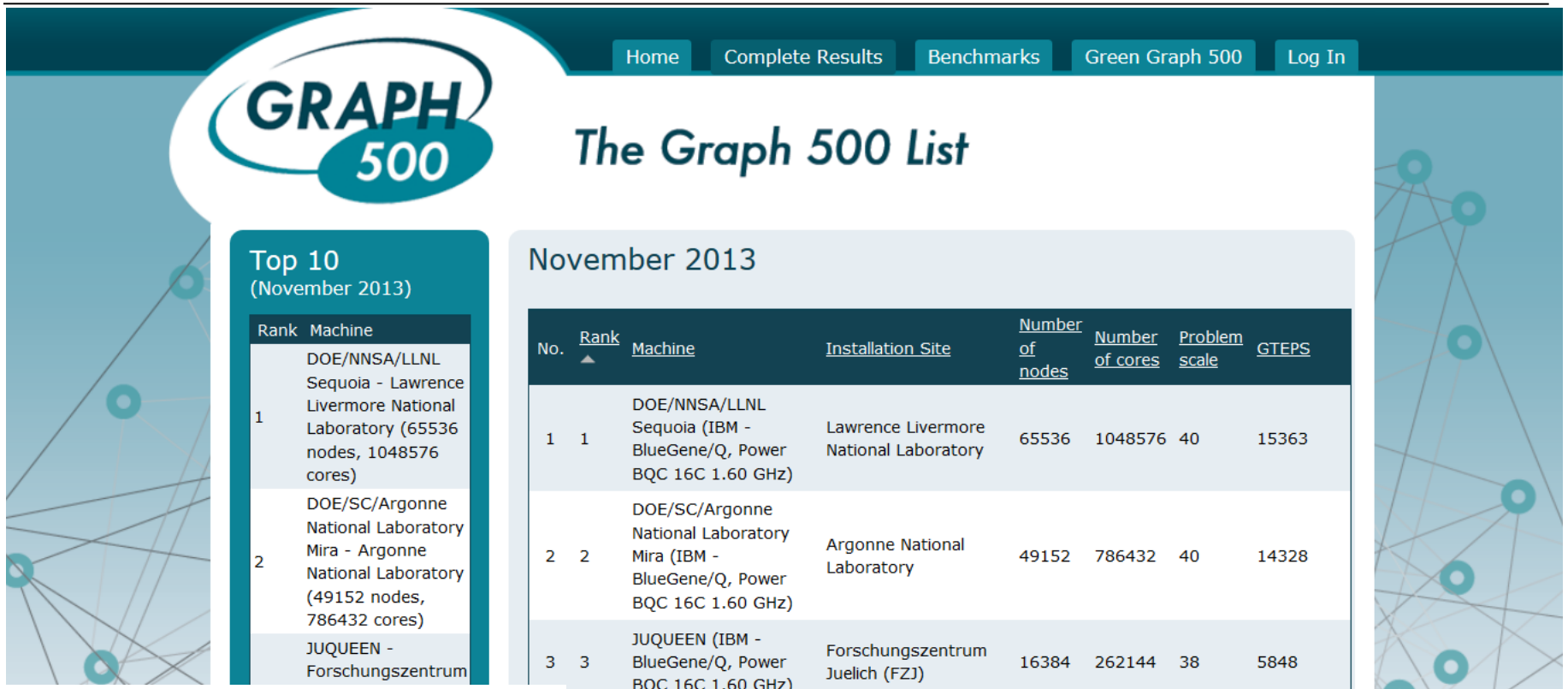
$$\psi^*_{\mathcal{S}} = \sum_{\mathcal{Y} \setminus \mathcal{S}} \psi^*_{\mathcal{Y}}, \quad \psi^*_{\mathcal{X}} = \psi_{\mathcal{X}} \frac{\psi^*_{\mathcal{S}}}{\psi_{\mathcal{S}}}$$

- Absorb
- Marginalize
- Extend
- Multiply/Divide



Marginalization

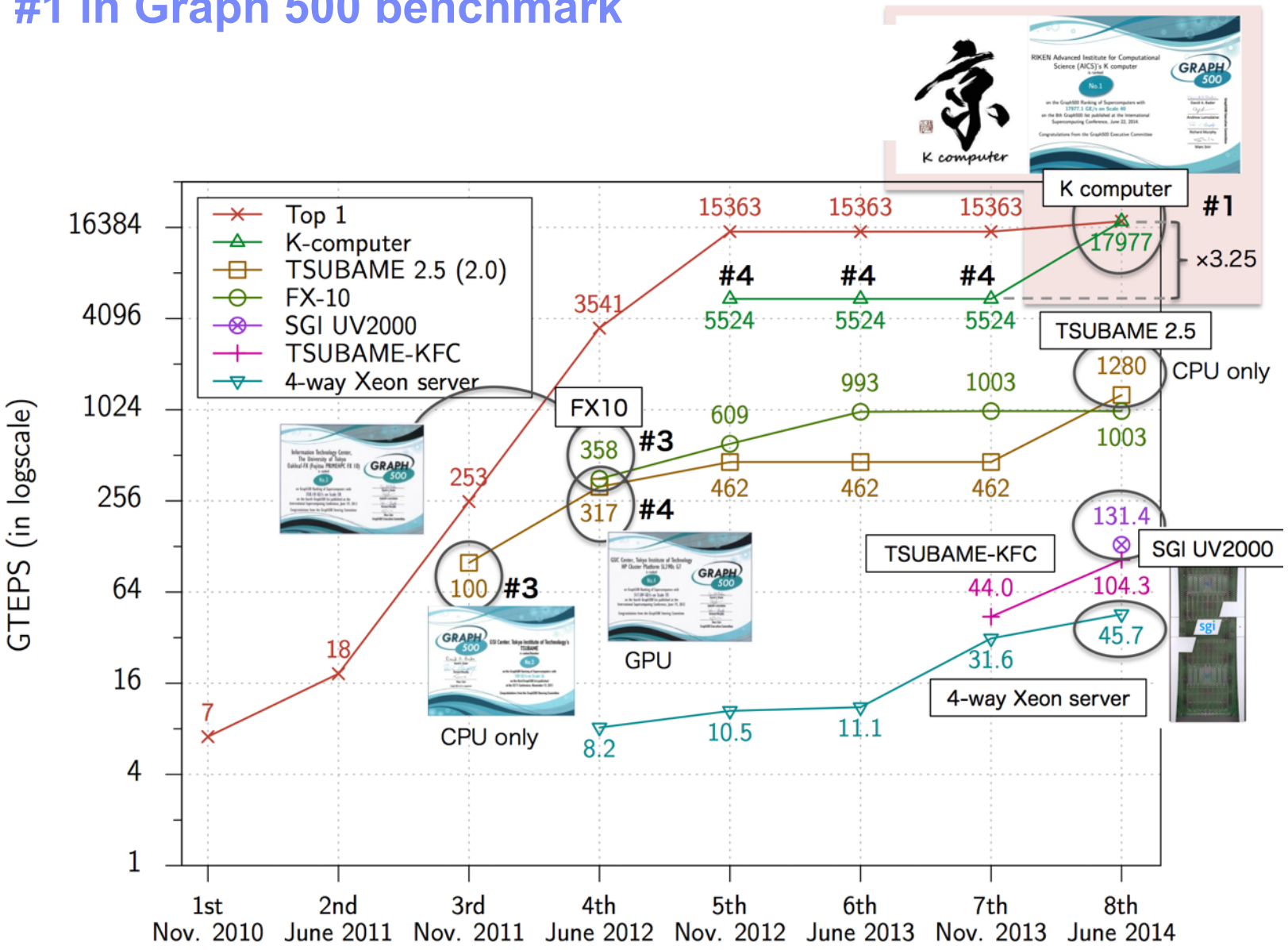
Large-Scale Graph Benchmark — Graph 500



- Graph500 is a new benchmark that ranks supercomputers by executing a large-scale graph search problem.
- The benchmark is ranked by so-called **TEPS (Traversed Edges Per Second)** that measures the number of edges to be traversed per second by searching all the reachable vertices from one arbitrary vertex with each team's **optimized BFS (Breadth-First Search) algorithm**.
- **Problem Size : the total number of vertices, 2^{SCALE}**
  - **Scale** : The logarithm base two of the number of vertices.

IBM BlueGene or P775:
24 out of Top 30, except
#4, #14, #22: Fujitsu (algorithm is from IBM)
#6: Tianhe
#15: Cray
#24: HP

ScaleGraph algorithms (IBM System G's open source) made Top #1 in Graph 500 benchmark



Source: Dr. Toyotaro Suzumura, ICPE2014 keynote

RDF and SPARQL

WHAT DO RDF AND SPARQL BRING TO BIG DATA PROJECTS?

Bob DuCharme
TopQuadrant, Charlottesville, Virginia

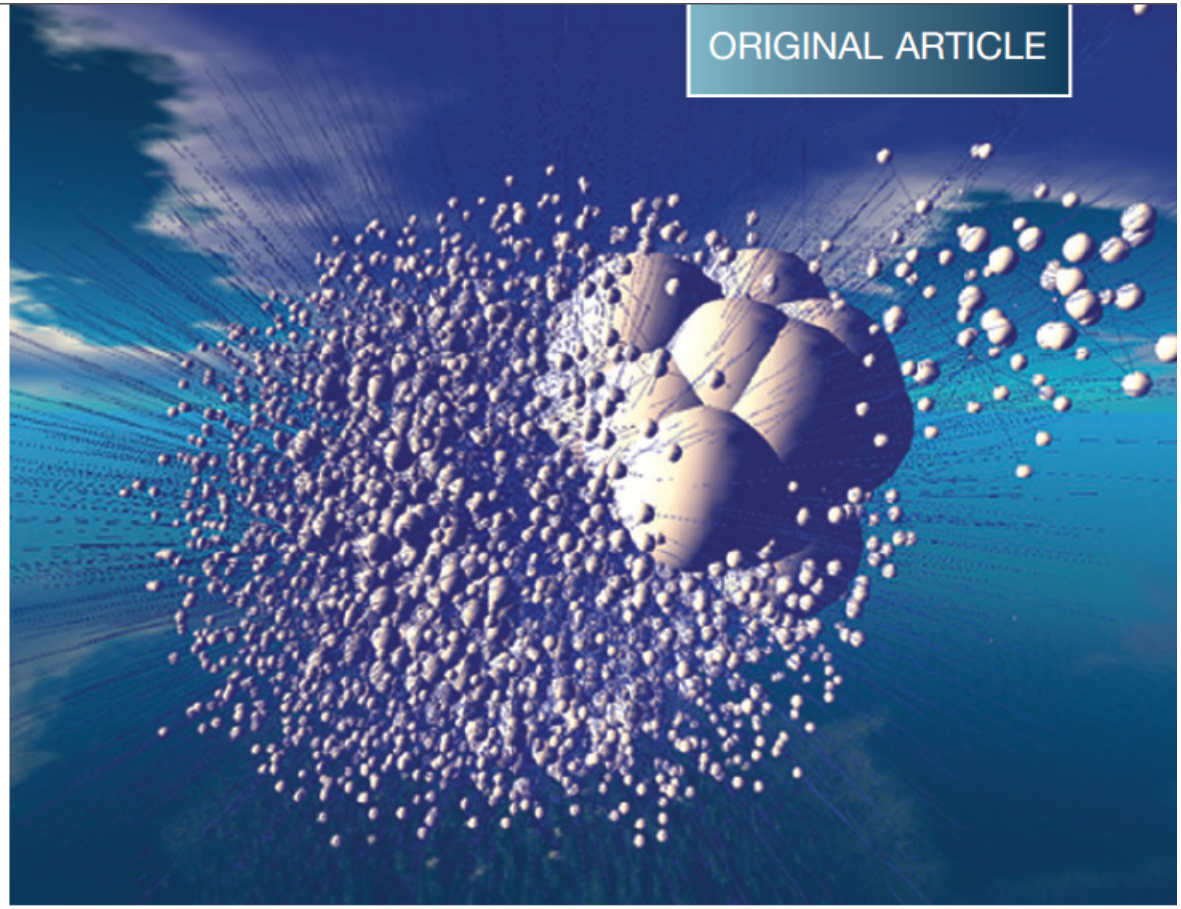


Photo Credit, Erich Bremer: http://www.ebremer.com/nexus/2011-05-15

Resource Description Format (RDF)

- A W3C standard sicne 1999
- Triples
- Example: A company has nince of part p1234 in stock, then a simplified triple rpresenting this might be {p1234 inStock 9}.
- Instance Identifier, Property Name, Property Value.
- In a proper RDF version of this triple, the representation will be more formal. They require uniform resource identifiers (URIs).

```
@prefix fbd: <http://foobarco.net/data/>.
@prefix fbv: <http://foobarco.net/vocab/>.

fbd:p1234 fbv:inStock "9".
fbd:p1234 fbv:supplier "Joe's Part Company".
```

An example complete description

```
@prefix fbd:<http://foobarco.net/data/>.
@prefix fbv:<http://foobarco.net/vocab/>.
fbd:p1234 fbv:inStock "9".
fbd:p1234 fbv:name "Blue reverse flange".
fbd:p1234 fbv:supplier fbd:s9483.
fbd:s9483 fbv:name "Joe's Part Company".
fbd:s9483 fbv:homePage "http://www.joespartco.com".
fbd:s9483 fbv:contactName "Gina Smith".
fbd:s9483 fbv:contactEmail "gina.smith@joespartco.com".
```

Advantages of RDF

- Virtually any RDF software can parse the lines shown above as self-contained, working data file.
  - You can declare properties if you want.
  - The RDF Schema standard lets you declare classes and relationships between properties and classes.
  - The flexibility that the lack of dependence on schemas is the first key to RDF's value.
- Split trips into several lines that won't affect their collective meaning, which makes sharding of data collections easy.
  - Multiple datasets can be combined into a usable whole with simple concatenation.
- For the inventory dataset's property name URIs, sharing of vocabulary makes easy to aggregate.

SPARQL – Query Langauge for RDF

The following SPQRL query asks for all property names and values associated with the fbd:s9483 resource:

```
PREFIX fbd: <http://foobarco.net/data/>

SELECT ?property ?value
WHERE {fbd:s9483 ?property ?value.}
```

The heart of any SPARQL query is the WHERE clause, which specifies the triples to pull out of the dataset. Various options for the rest of the query tell the SPARQL processor what to do with those triples, such as sorting, creating, or deleting triples. The above query's WHERE clause has a single triple pattern, which resembles a triple but may have variables substituted for any or all of the triple's three parts. The triple pattern above says that we're interested in triples that have fbd:s9483 as the subject and—because variables function as wildcards—anything at all in the triple's second and third parts.

The SPAQRL Query Result from the previous example

property	value
<http://foobarco.net/vocab/contactEmail>	"gina.smith@joespartco.com"
<http://foobarco.net/vocab/contactName>	"Gina Smith"
<http://foobarco.net/vocab/homePage>	"http://www.joespartco.com"
<http://foobarco.net/vocab/name>	"Joe's Part Company"

Another SPARQL Example

What is this query for?

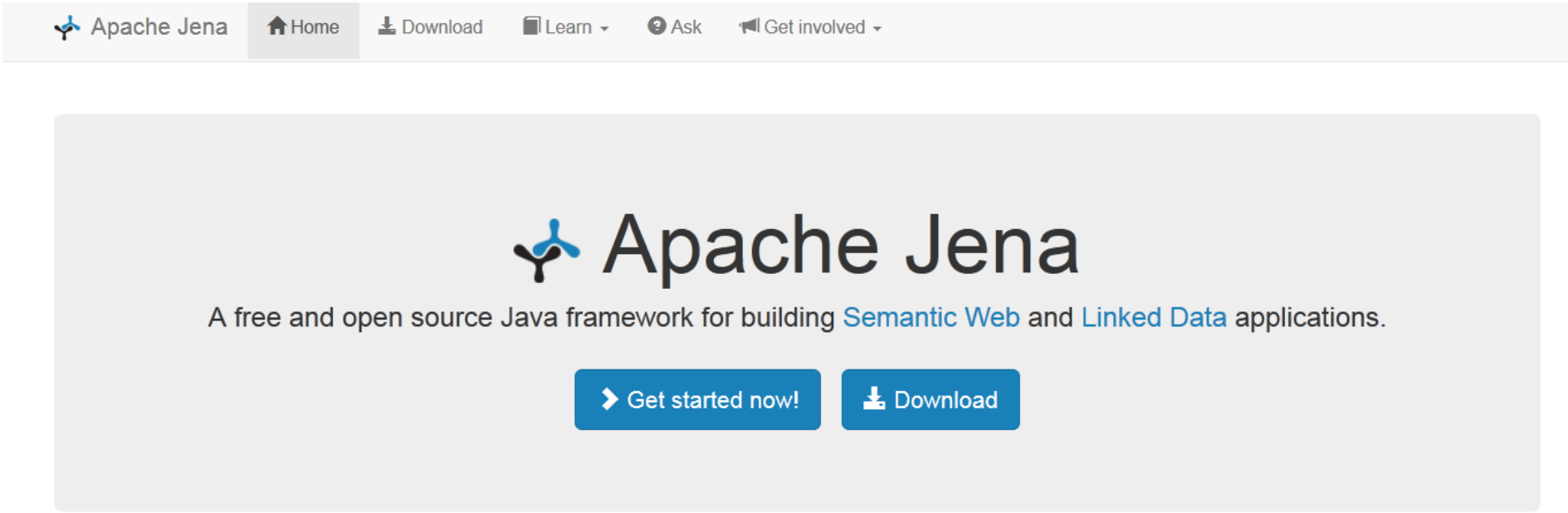
```
PREFIX fbd:<http://foobarco.net/data/>
PREFIX fbv:<http://foobarco.net/vocab/>

SELECT ?flangeContactEmail
WHERE
{
   ?part fbv:name "Blue reverse flange".
   ?part fbv:supplier ?supplier.
   ?supplier fbv:contactEmail ?flangeContactEmail.
}
```

Data

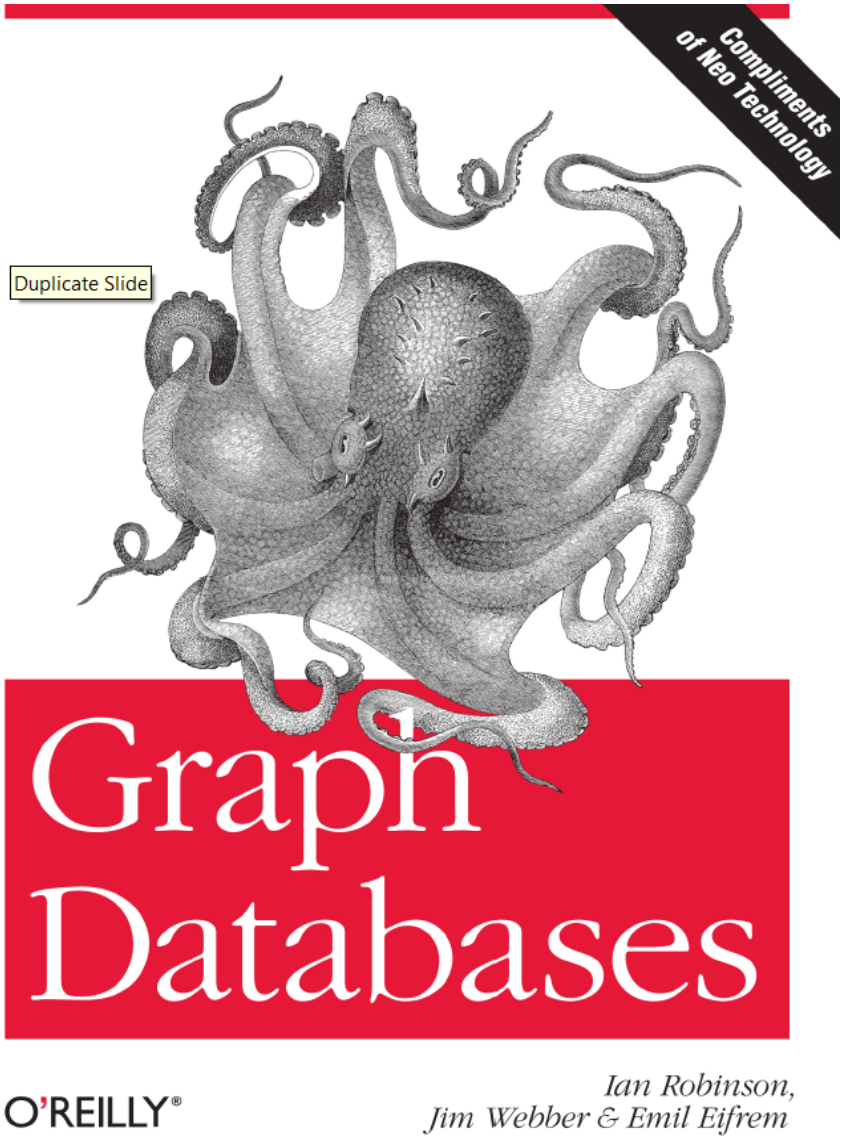
```
@prefix fbd:<http://foobarco.net/data/>.
@prefix fbv:<http://foobarco.net/vocab/>.
fbd:p1234 fbv:inStock "9".
fbd:p1234 fbv:name "Blue reverse flange".
fbd:p1234 fbv:supplier fbd:s9483.
fbd:s9483 fbv:name "Joe's Part Company".
fbd:s9483 fbv:homePage "http://www.joespartco.com".
fbd:s9483 fbv:contactName "Gina Smith".
fbd:s9483 fbv:contactEmail "gina.smith@joespartco.com".
```

Open Source Software – Apache Jena

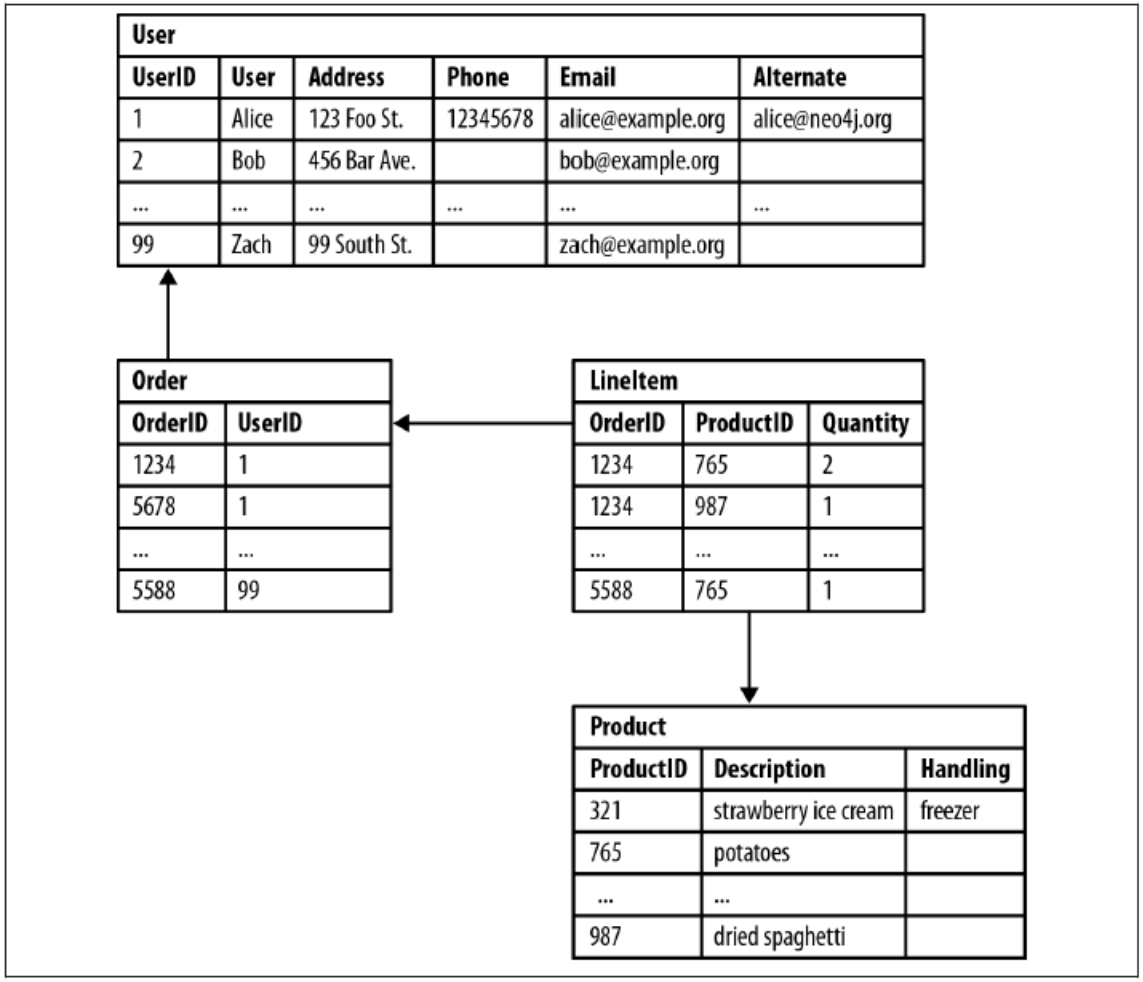


Property Graphs

Reference



A usual example



User

UserID	User	Address	Phone	Email	Alternate
1	Alice	123 Foo St.	12345678	alice@example.org	alice@neo4j.org
2	Bob	456 Bar Ave.		bob@example.org	
...	...	...	...	...	...
99	Zach	99 South St.		zach@example.org	

Order

OrderID	UserID
1234	1
5678	1
...	...
5588	99

LineItem

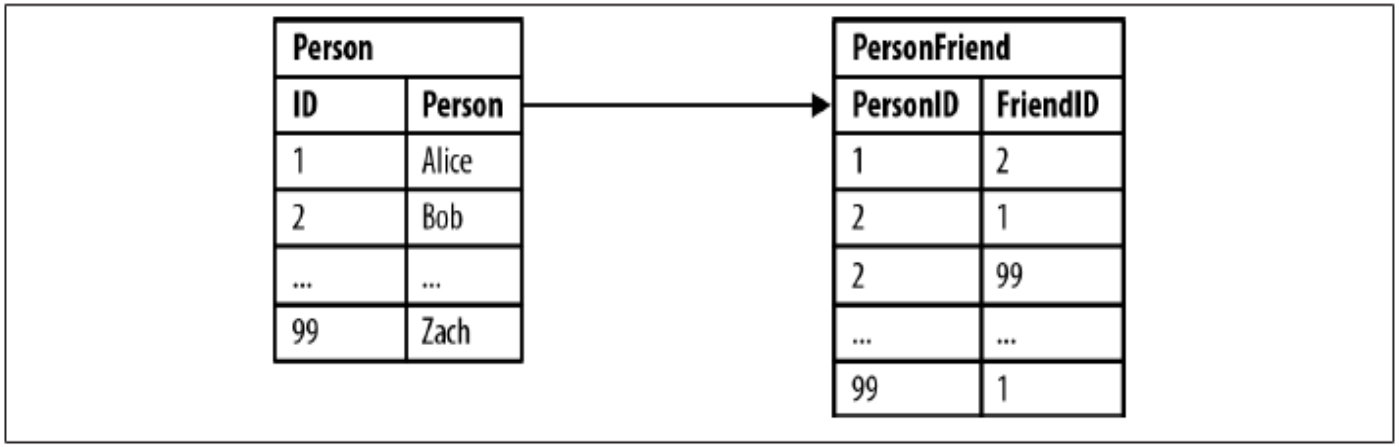
OrderID	ProductID	Quantity
1234	765	2
1234	987	1
...	...	...
5588	765	1

Product

ProductID	Description	Handling
321	strawberry ice cream	freezer
765	potatoes	
...	...	
987	dried spaghetti	

Figure 2-1. Semantic relationships are hidden in a relational database

Query Example – I



Person	
ID	Person
1	Alice
2	Bob
...	...
99	Zach

PersonFriend	
PersonID	FriendID
1	2
2	1
2	99
...	...
99	1

Figure 2-2. Modeling friends and friends-of-friends in a relational database

Asking "who are Bob's friends?" is easy, as shown in Example 2-1.

Example 2-1. Bob's friends

```
SELECT p1.Person
FROM Person p1 JOIN PersonFriend
  ON PersonFriend.FriendID = p1.ID
JOIN Person p2
  ON PersonFriend.PersonID = p2.ID
WHERE p2.Person = 'Bob'
```

Query Examples – II & III

Example 2-2. Who is friends with Bob?

```
SELECT p1.Person
FROM Person p1 JOIN PersonFriend
  ON PersonFriend.PersonID = p1.ID
JOIN Person p2
  ON PersonFriend.FriendID = p2.ID
WHERE p2.Person = 'Bob'
```

Example 2-3. Alice's friends-of-friends

```
SELECT p1.Person AS PERSON, p2.Person AS FRIEND_OF_FRIEND
FROM PersonFriend pf1 JOIN Person p1
  ON pf1.PersonID = p1.ID
JOIN PersonFriend pf2
  ON pf2.PersonID = pf1.FriendID
JOIN Person p2
  ON pf2.FriendID = p2.ID
WHERE p1.Person = 'Alice' AND pf2.FriendID <> p1.ID
```

Computational intensive

Graph Database Example

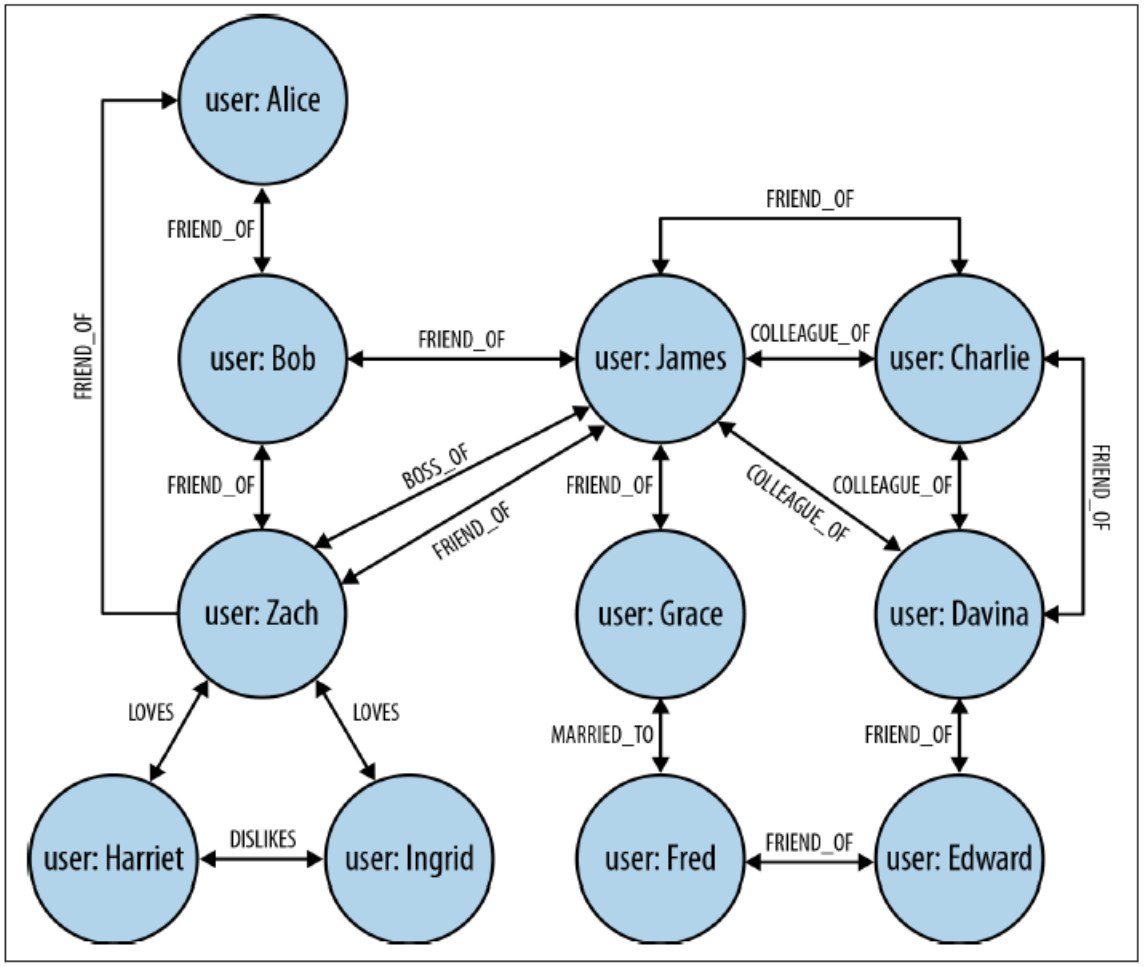


Figure 2-5. Easily modeling friends, colleagues, workers, and (unrequited) lovers in a graph

Execution Time in the example of finding extended friends (by Neo4j)

Partner and Vukotic's experiment seeks to find friends-of-friends in a social network, to a maximum depth of five. Given any two persons chosen at random, is there a path that connects them that is at most five relationships long? For a social network containing 1,000,000 people, each with approximately 50 friends, the results strongly suggest that graph databases are the best choice for connected data, as we see in Table 2-1.

Table 2-1. Finding extended friends in a relational database versus efficient finding in Neo4j

Depth	RDBMS execution time (s)	Neo4j execution time (s)	Records returned
2	0.016	0.01	~2500
3	30.267	0.168	~110,000
4	1543.505	1.359	~600,000
5	Unfinished	2.132	~800,000

Modeling Order History as a Graph

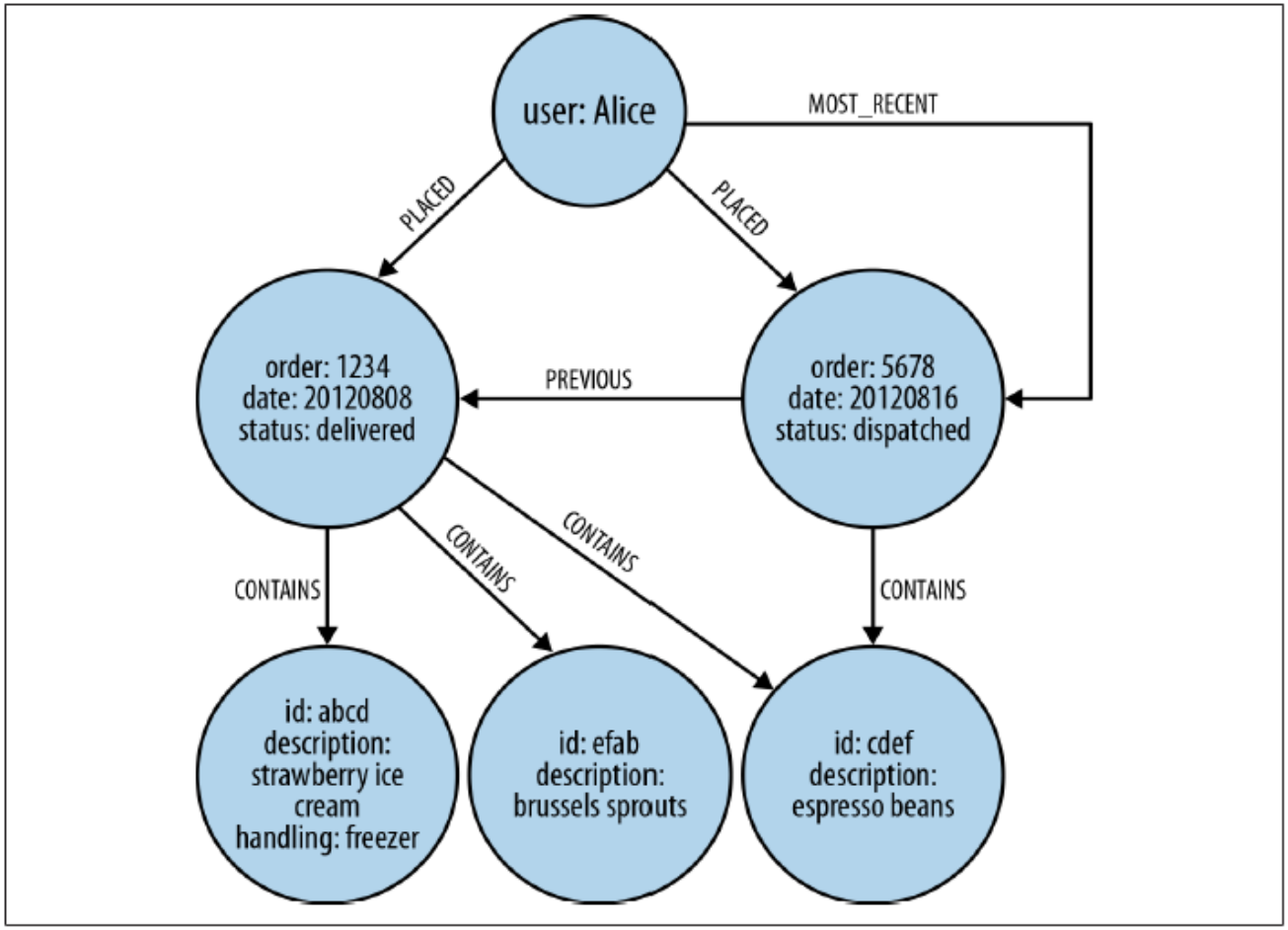


Figure 2-6. Modeling a user's order history in a graph

A query language on Property Graph – Cypher

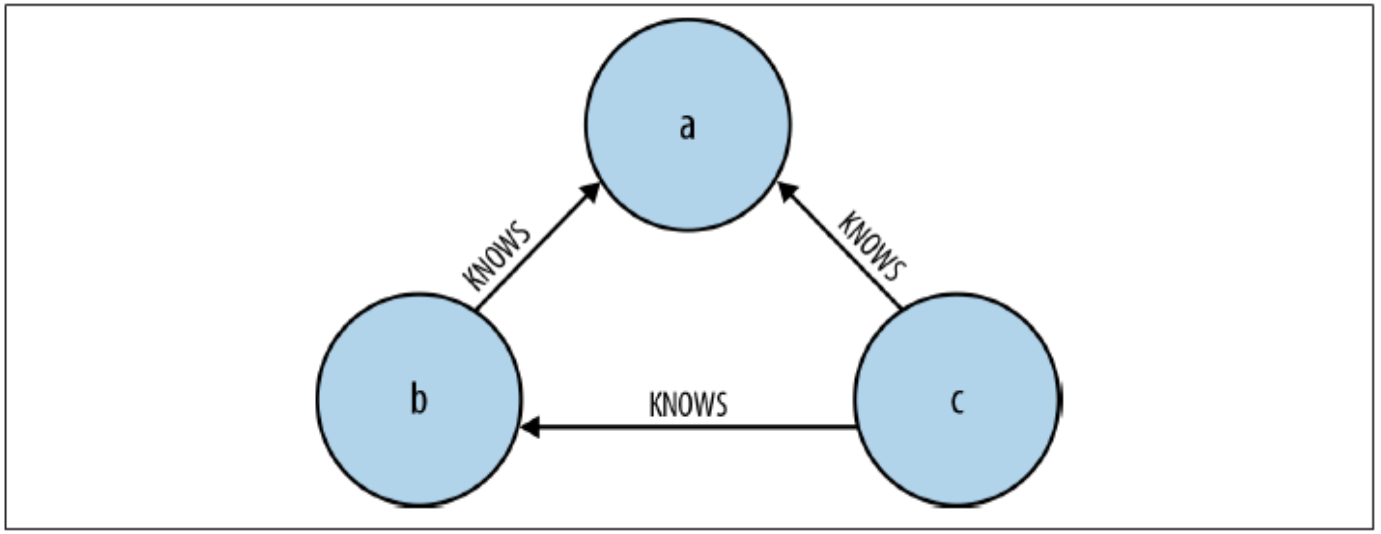


Figure 3-1. A simple graph pattern, expressed using a diagram

This pattern describes three mutual friends. Here's the equivalent ASCII art representation in Cypher:

```
(a)-[:KNOWS]->(b)-[:KNOWS]->(c), (a)-[:KNOWS]->(c)
```

Cypher Example

Like most query languages, Cypher is composed of clauses. The simplest queries consist of a START clause followed by a MATCH and a RETURN clause (we'll describe the other clauses you can use in a Cypher query later in this chapter). Here's an example of a Cypher query that uses these three clauses to find the mutual friends of user named *Michael*:

```
START a=node:user(name='Michael')
MATCH (a)-[:KNOWS]->(b)-[:KNOWS]->(c), (a)-[:KNOWS]->(c)
RETURN b, c
```

Other Cypher Clauses

Columbia University

`WHERE`
: Provides criteria for filtering pattern matching results.

`CREATE` *and* `CREATE UNIQUE`
: Create nodes and relationships.

`DELETE`
: Removes nodes, relationships, and properties.

`SET`
: Sets property values.

`FOREACH`
: Performs an updating action for each element in a list.

`UNION`
: Merges results from two or more queries (introduced in Neo4j 2.0).

`WITH`
: Chains subsequent query parts and forward results from one to the next. Similar to piping commands in Unix.

Property Graph Example – Shakespeare

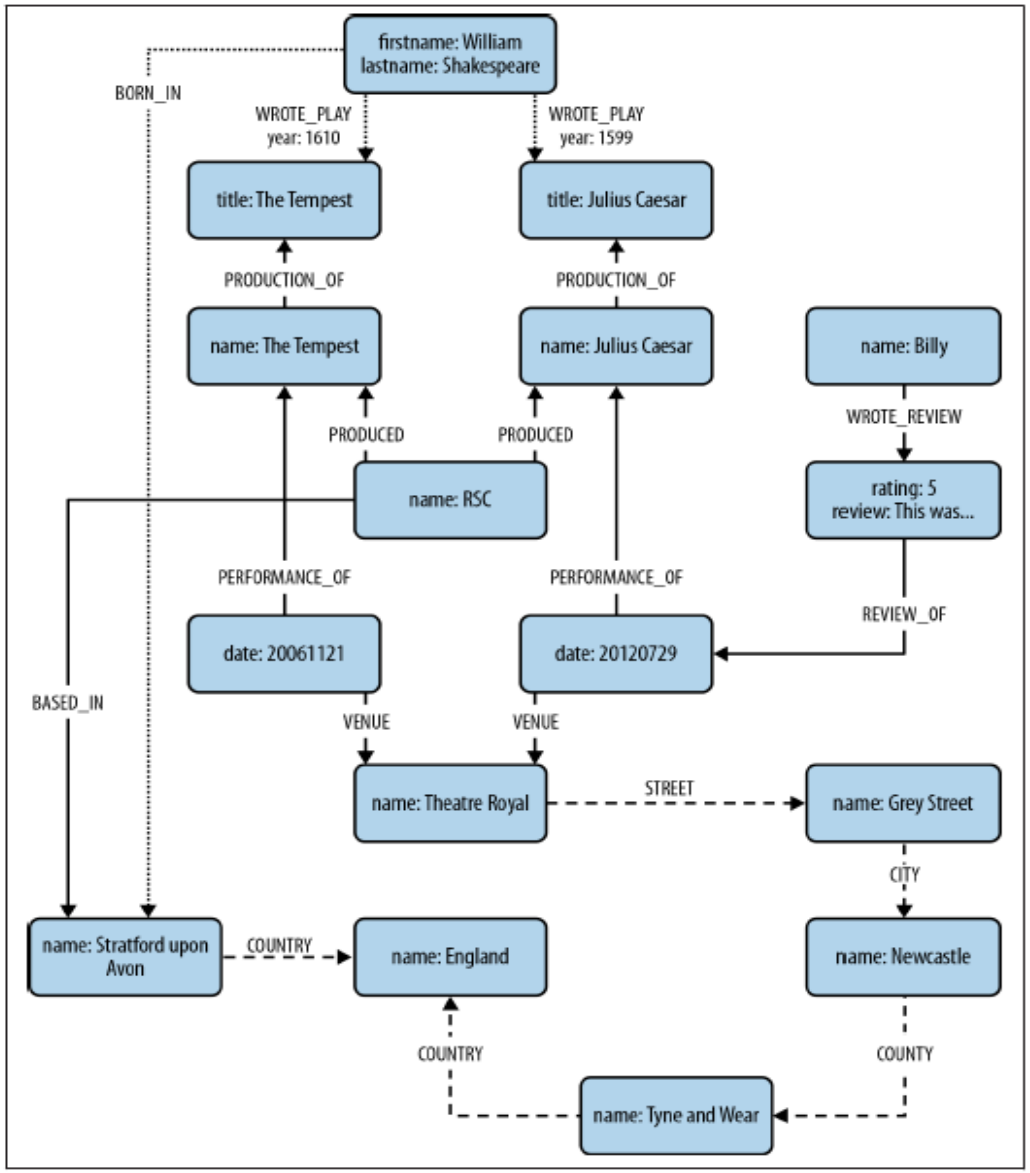


Figure 3-6. Three domains in one graph

Creating the Shakespeare Graph

```
CREATE (shakespeare { firstname: 'William', lastname: 'Shakespeare' }),
       (juliusCaesar { title: 'Julius Caesar' }),
       (shakespeare)-[:WROTE_PLAY { year: 1599 }]->(juliusCaesar),
       (theTempest { title: 'The Tempest' }),
       (shakespeare)-[:WROTE_PLAY { year: 1610}]->(theTempest),
       (rsc { name: 'RSC' }),
       (production1 { name: 'Julius Caesar' }),
       (rsc)-[:PRODUCED]->(production1),
       (production1)-[:PRODUCTION_OF]->(juliusCaesar),
       (performance1 { date: 20120729 }),
       (performance1)-[:PERFORMANCE_OF]->(production1),
       (production2 { name: 'The Tempest' }),
       (rsc)-[:PRODUCED]->(production2),
       (production2)-[:PRODUCTION_OF]->(theTempest),
       (performance2 { date: 20061121 }),
       (performance2)-[:PERFORMANCE_OF]->(production2),
       (performance3 { date: 20120730 }),
       (performance3)-[:PERFORMANCE_OF]->(production1),
       (billy { name: 'Billy' }),
       (review { rating: 5, review: 'This was awesome!' }),
       (billy)-[:WROTE_REVIEW]->(review),
       (review)-[:RATED]->(performance1),
       (theatreRoyal { name: 'Theatre Royal' }),
       (performance1)-[:VENUE]->(theatreRoyal),
       (performance2)-[:VENUE]->(theatreRoyal),
       (performance3)-[:VENUE]->(theatreRoyal),
       (greyStreet { name: 'Grey Street' }),
       (theatreRoyal)-[:STREET]->(greyStreet),
       (newcastle { name: 'Newcastle' }),
       (greyStreet)-[:CITY]->(newcastle),
       (tyneAndWear { name: 'Tyne and Wear' }),
       (newcastle)-[:COUNTY]->(tyneAndWear),
       (england { name: 'England' }),
       (tyneAndWear)-[:COUNTRY]->(england),
       (stratford { name: 'Stratford upon Avon' }),
       (stratford)-[:COUNTRY]->(england),
       (rsc)-[:BASED_IN]->(stratford),
       (shakespeare)-[:BORN_IN]->stratford
```

Query on the Shakespeare Graph

```
START theater=node:venue(name='Theatre Royal'),
      newcastle=node:city(name='Newcastle'),
      bard=node:author(lastname='Shakespeare')
MATCH (newcastle)<-[:STREET|CITY*1..2]-(theater)
      <-[:VENUE]-()-[:PERFORMANCE_OF]->()-[:PRODUCTION_OF]->
      (play)<-[w:WROTE_PLAY]-(bard)
WHERE w.year > 1608
RETURN DISTINCT play.title AS play
```

Adding this `WHERE` clause means that for each successful match, the Cypher execution engine checks that the `WROTE_PLAY` relationship between the Shakespeare node and the matched play has a `year` property with a value greater than 1608. Matches with a `WROTE_PLAY` relationship whose `year` value is greater than 1608 will pass the test; these plays will then be included in the results. Matches that fail the test will not be included in the results. By adding this clause, we ensure that only plays from Shakespeare's late period are returned:

```
+---------------+
| play          |
+---------------+
| "The Tempest" |
+---------------+
1 row
```

Another Query on the Shakespeare Graph

```
START theater=node:venue(name='Theatre Royal'),
      newcastle=node:city(name='Newcastle'),
      bard=node:author(lastname='Shakespeare')
MATCH (newcastle)<-[:STREET|CITY*1..2]-(theater)
      <-[:VENUE]-()-[p:PERFORMANCE_OF]->()-[:PRODUCTION_OF]->
      (play)<-[:WROTE_PLAY]-(bard)
RETURN   play.title AS play, count(p) AS performance_count
ORDER BY performance_count DESC
```

The RETURN clause here counts the number of PERFORMANCE_OF relationships using the identifier p (which is bound to the PERFORMANCE_OF relationships in the MATCH clause) and aliases the result as performance_count. It then orders the results based on performance_count, with the most frequently performed play listed first:

```
+-------------------------------------+
| play            | performance_count |
+-------------------------------------+
| "Julius Caesar" | 2                 |
| "The Tempest"   | 1                 |
+-------------------------------------+
2 rows
```

Building Application Example – Collaborative Filtering

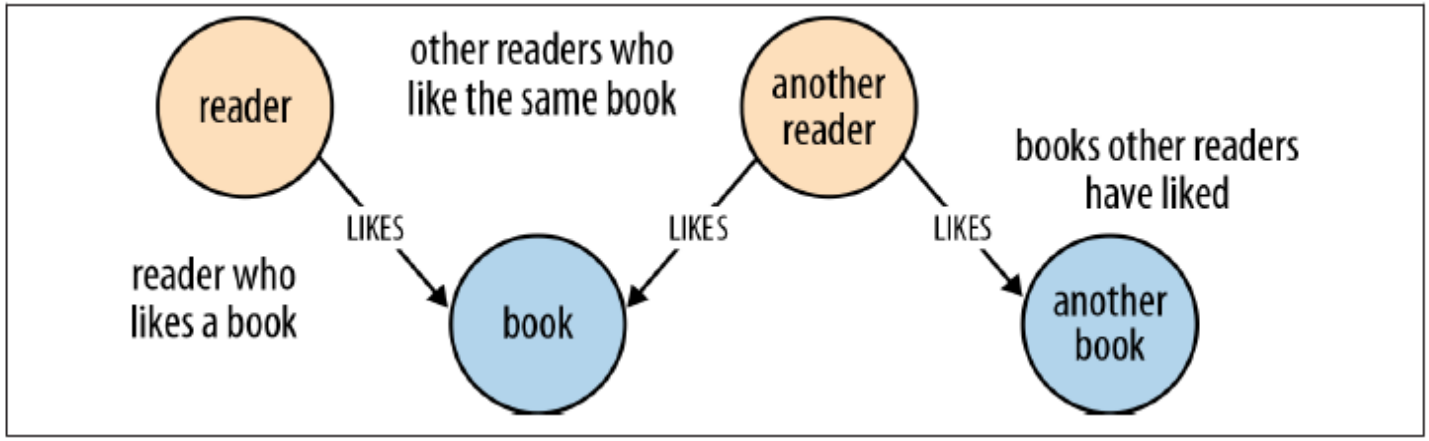


Figure 4-1. Data model for the book reviews user story

Because this data model directly encodes the question presented by the user story, it lends itself to being queried in a way that similarly reflects the structure of the question we want to ask of the data:

```
START reader=node:users(name={readerName})
      book=node:books(isbn={bookISBN})
MATCH reader-[:LIKES]->book<-[:LIKES]-other_readers-[:LIKES]->books
RETURN books.title
```

Chaining on the Query

```
START bard=node:author(lastname='Shakespeare')
MATCH (bard)-[w:WROTE_PLAY]->(play)
WITH play
ORDER BY w.year DESC
RETURN collect(play.title) AS plays
```

Executing this query against our sample graph produces the following result:

```
+---------------------------------+
| plays                           |
+---------------------------------+
| ["The Tempest","Julius Caesar"] |
+---------------------------------+
1 row
```

Example – Email Interaction Graph

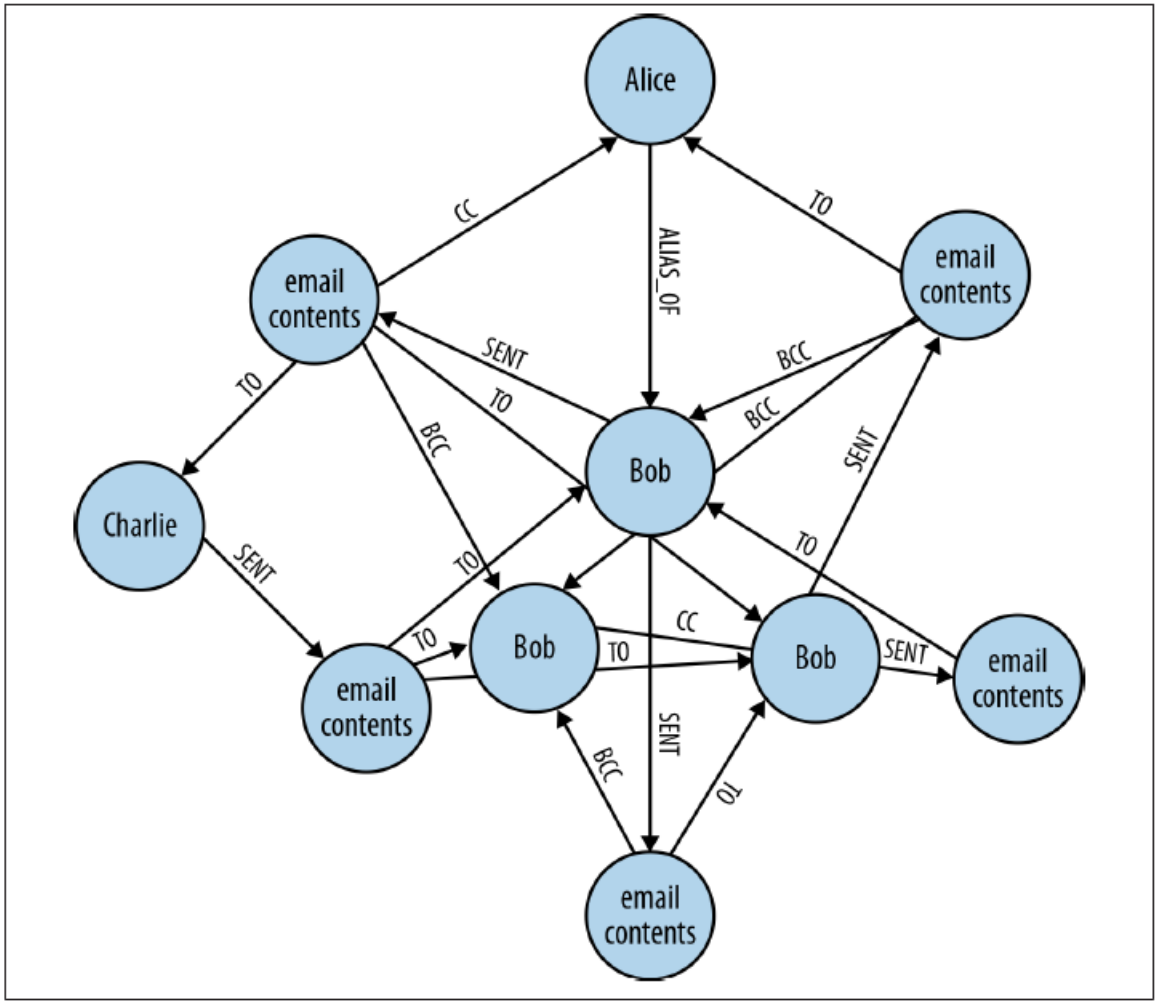


Figure 3-10. A graph of email interactions

```
START bob=node:user(username='Bob')
MATCH (bob)-[:SENT]->(email)-[:CC]->(alias),
      (alias)-[:ALIAS_OF]->(bob)
RETURN email
```

What's this query for?

How to make graph database fast?

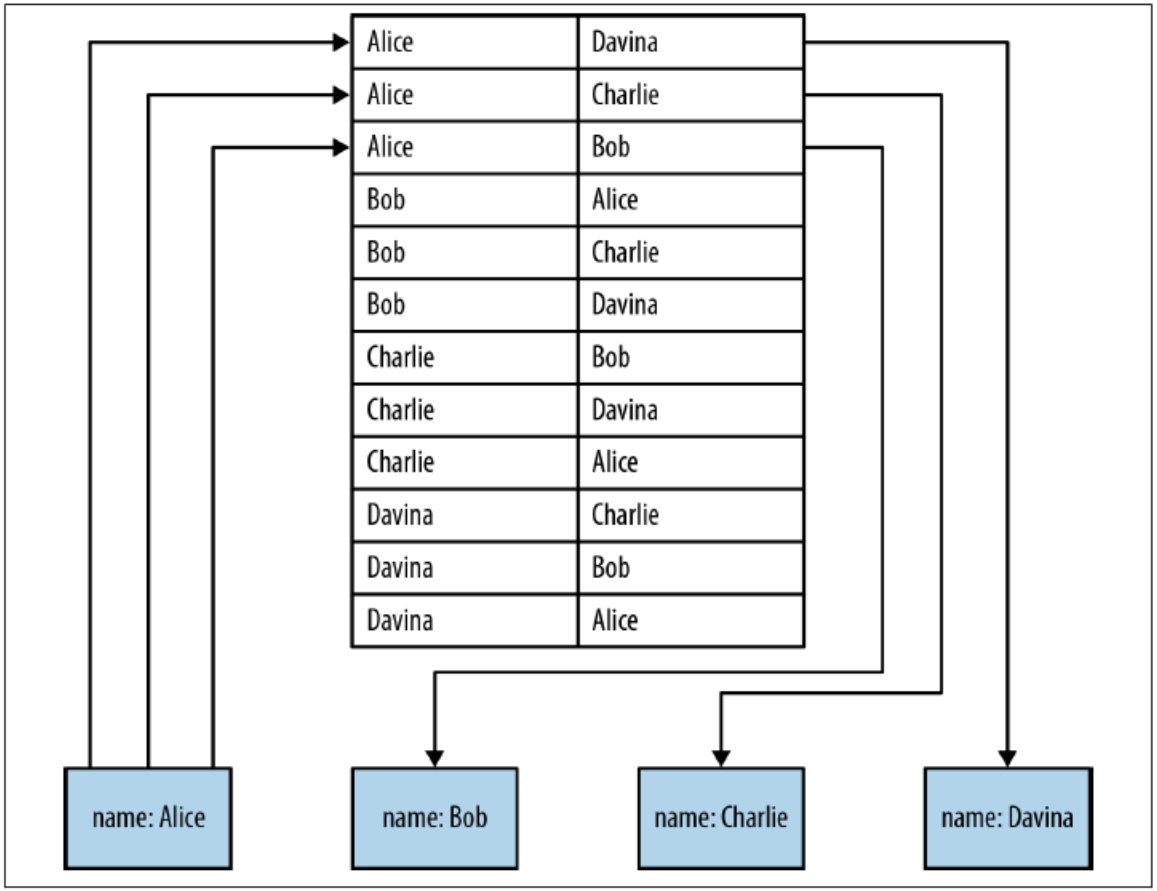
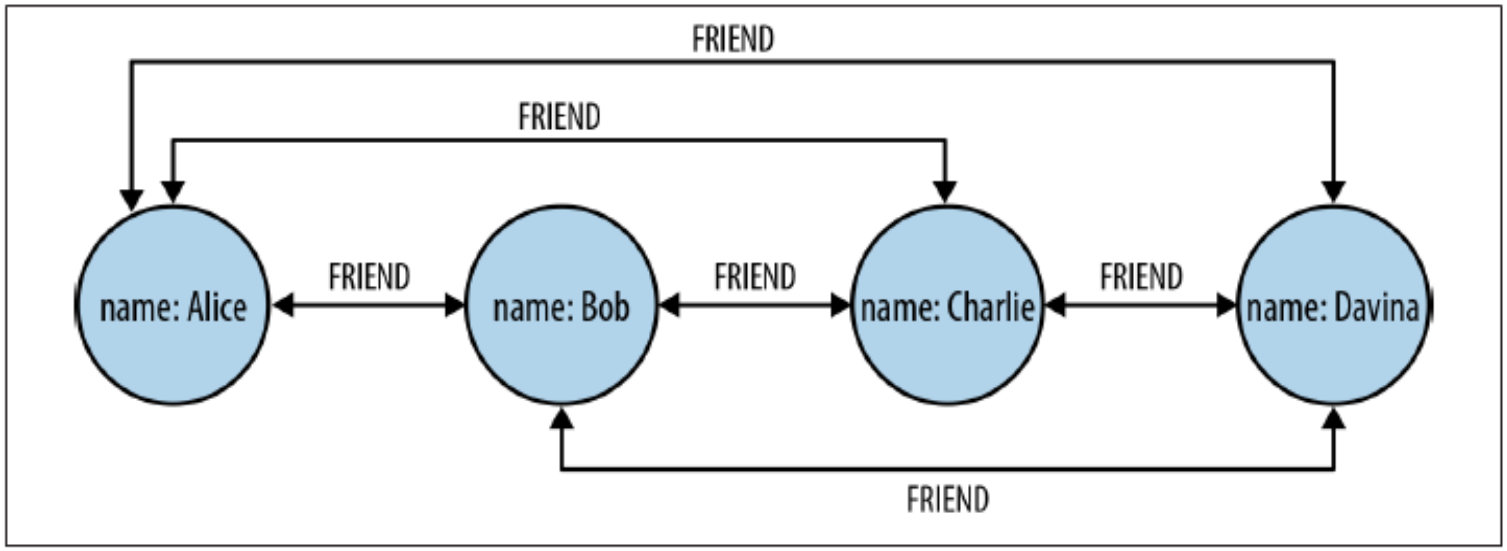


Figure 6-1. Nonnative graph processing engines use indexing to traverse between nodes

Use Relationships, not indexes, for fast traversal



Storage Structure Example

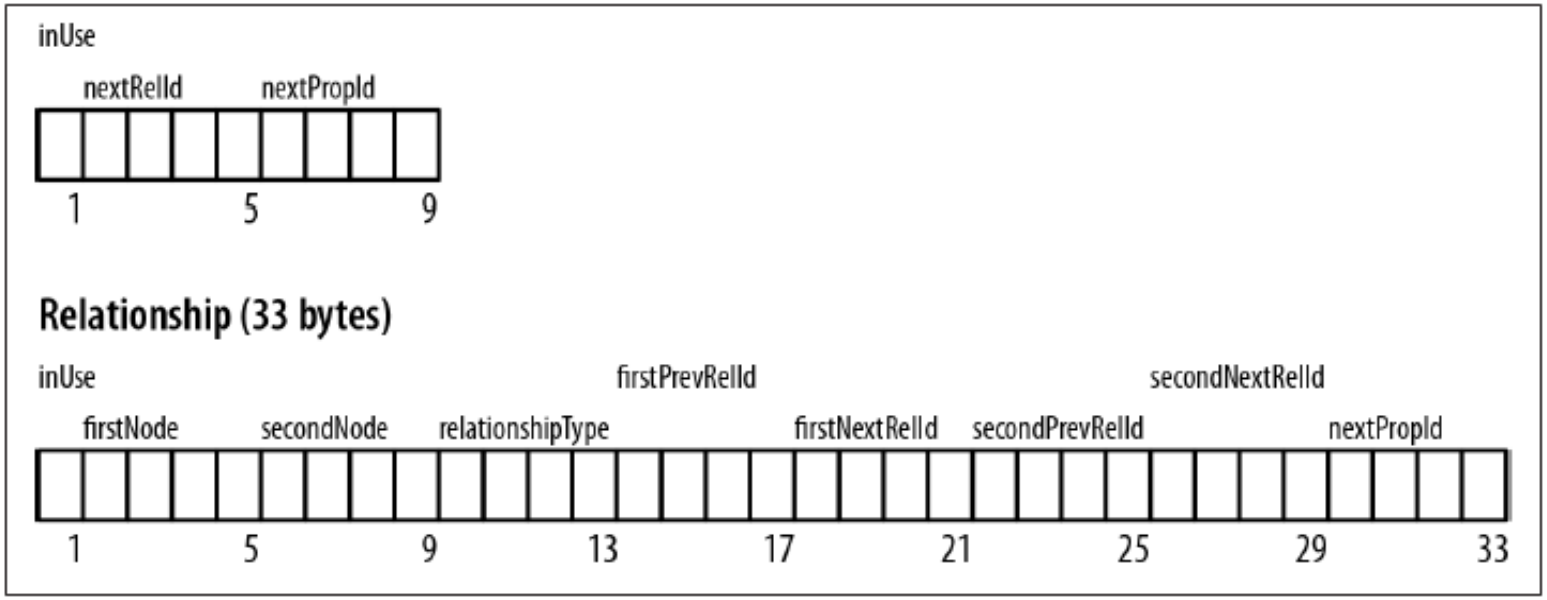


Figure 6-4. Neo4j node and relationship store file record structure

Nodes and Relationships in the Object Cache

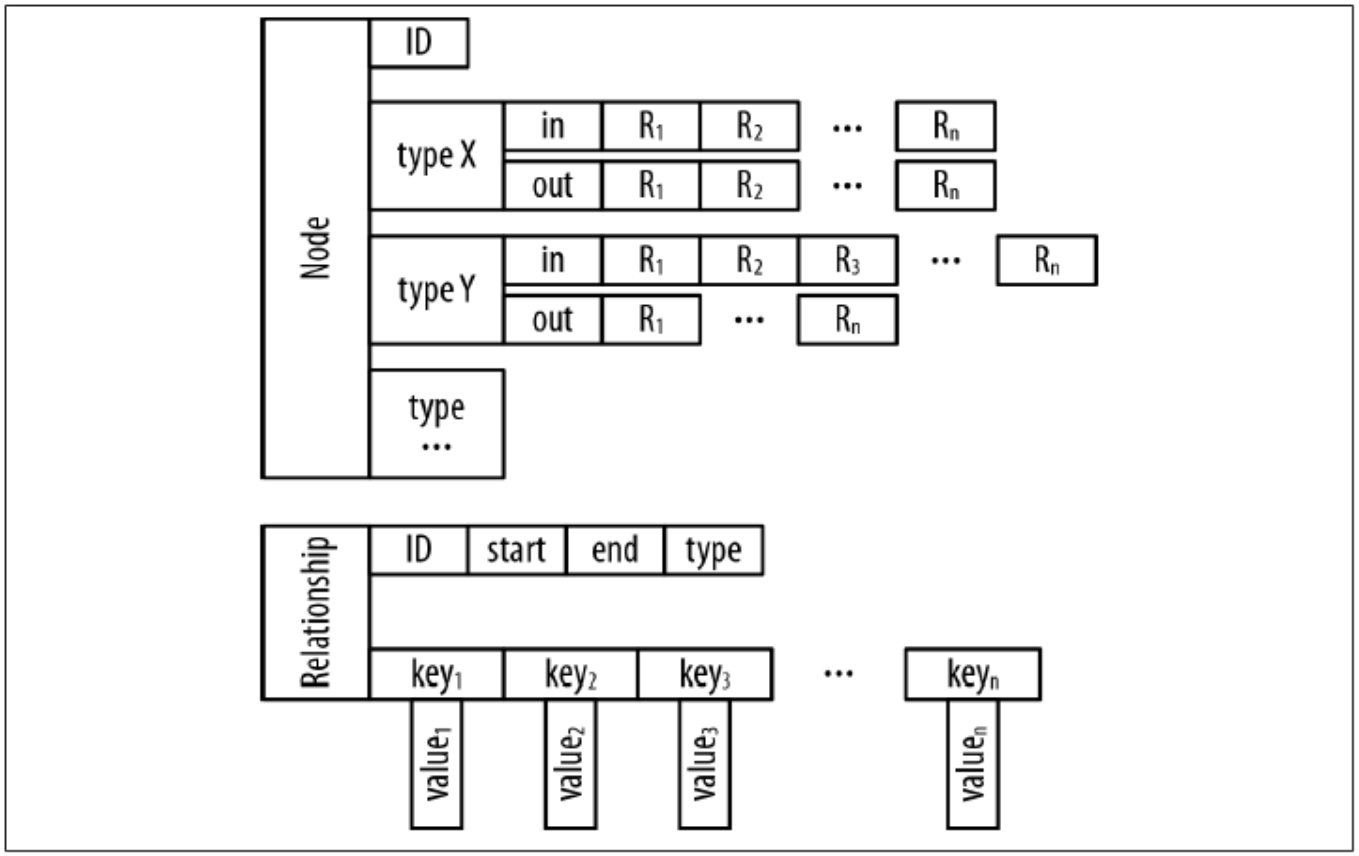
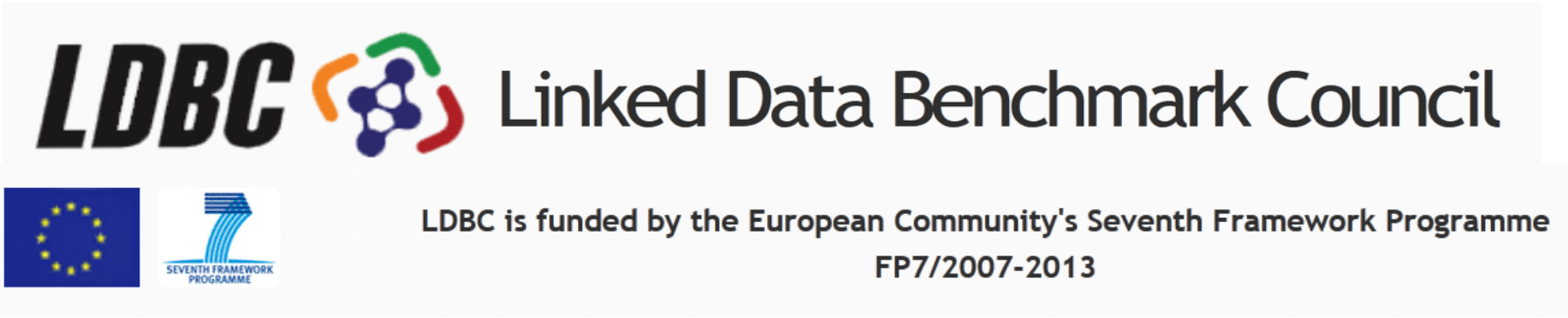
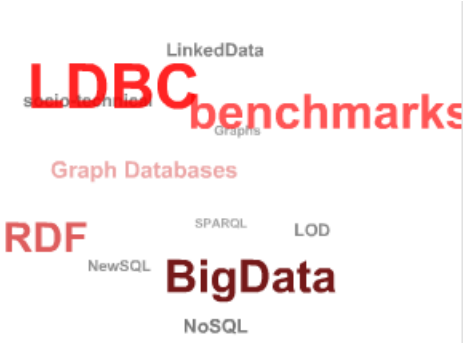


Figure 6-6. Nodes and relationships in the object cache

An Emerging Benchmark



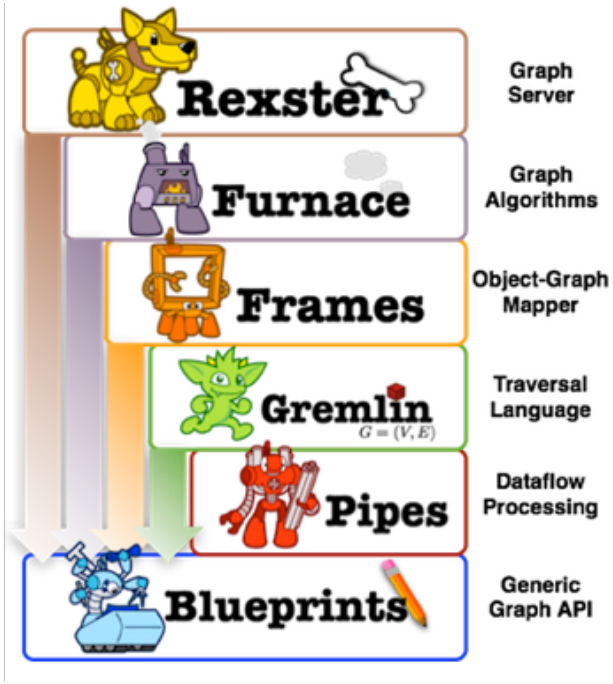


Test Set:
data generator of full social media activity simulation of any number of users

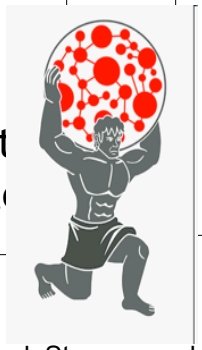
TinkerPop Stack and choices of Open Source Graph DBs

JVM API + Stack for Graph Includes:
- Gremlin traversal language
- - BluePrints JDBC of Graph DB's
- BYO persistence graph store:
- Growing community
- Ex. Titan, Neo4J, InfiniteGraph…

- But…
- It is still work in progress
- - analytics (Furnace) are not yet implemented
- - No support for pushing work to data nodes
- - Big data support is not clear



Titan by Aurelius
Apache2 license
Uses HBase, Berkeley DB, or Cassandra
Includes Rexster, Frames, Gremlin, Blueprint
Recently added: Faunus(M/R). RDF, bulk loa



Neo4J GraphDB
GPLv3 or commercial license
Fast
Graph must fit on each cluster node

A recent experiment

Dataset: 12.2 million edges, 2.2 million vertices

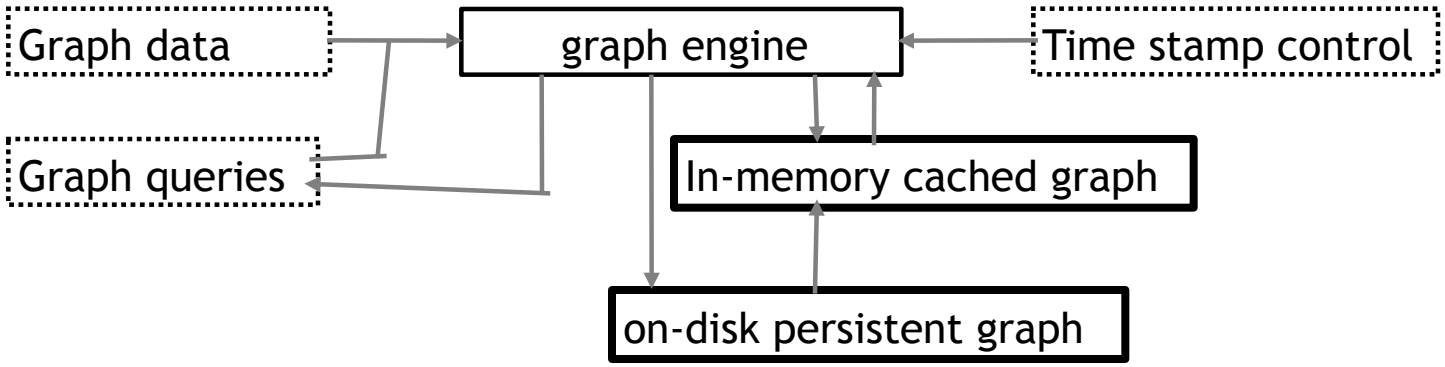
Goal: Find paths in a property graph. One of the vertex property is call TYPE. In this scenario, the user provides either a particular vertex, or a set of particular vertices of the same TYPE (say, "DRUG"). In addition, the user also provides another TYPE (say, "TARGET"). Then, we need find all the paths from the starting vertex to a vertex of TYPE “TARGET”. Therefore, we need to 1) find the paths using graph traversal; 2) keep trace of the paths, so that we can list them after the traversal. Even for the shortest paths, it can be multiple between two nodes, such as: drug->assay->target , drug->MOA->target

		Avg time (100 tests)	
	First test (cold-start)	Requested depth 5 traversal	Requested full depth traversal
IBM System G (NativeStore C++)	39 sec	3.0 sec	4.2 sec
IBM System G (NativeStore JNI)	57 sec	4.0 sec	6.2 sec
Neo4j (Blueprints 2.4)	105 sec	5.9 sec	8.3 sec
Titan (Berkeley DB)	3861 sec	641 sec	794 sec
Titan (HBase)	3046 sec	1597 sec	2682 sec

First full test - full depth 23. All data pulled from disk. Nothing initially cached.
Modes - All tests in default modes of each graph implementation. Titan can only be run in transactional mode. Other implementations do not default to transactional mode.

IBM System G Native Store Overview

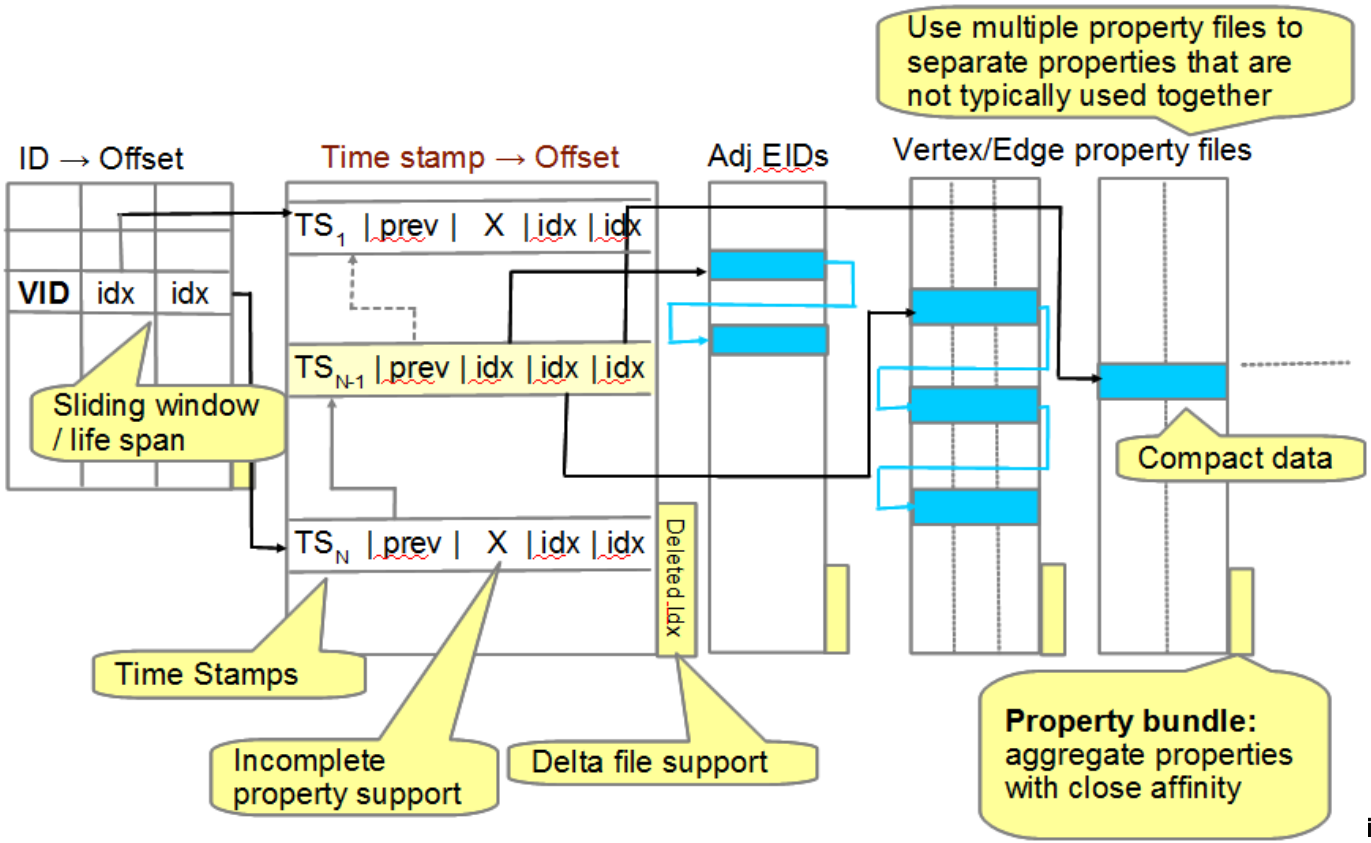
- **System G Native store represents graphs in-memory and on-disk**
  - Organizing graph data for representing a graph that stores both graph structure and vertex properties and edge properties
  - Caching graph data in memory in either batch-mode or on-demand from the on-disk streaming graph data
  - Accepting graph updates and modifying graph structure and/or property data accordingly and incorperating time stamps
    - Add edge, remove vertex, update property, etc.
  - Persisting graph updates along with the time stamps from in-memory graph to on-disk graph
  - Performing graph queries by loading graph structure and/or property data
    - Find neighbors of a vertex, retrieve property of an edge, traverse a graph, etc.



On-Disk Graph Organization

- **Native store organizes graph data for representing a graph with both structure and the vertex properties and edge properties using multiple files in Linux file system**
  - Creating a list called ID → Offset where each element translates a vertex (edge) ID into two offsets, pointing to the earliest and latest data of the vertex/edge, respectively
  - Creating a list called Time_stamp → Offset where each element has a time stamp, an offset to the previous time stamp of the vertex/edge, and a set of indices to the adjacent edge list and properties
  - Create a list of chained block list to store adjacent list and properties

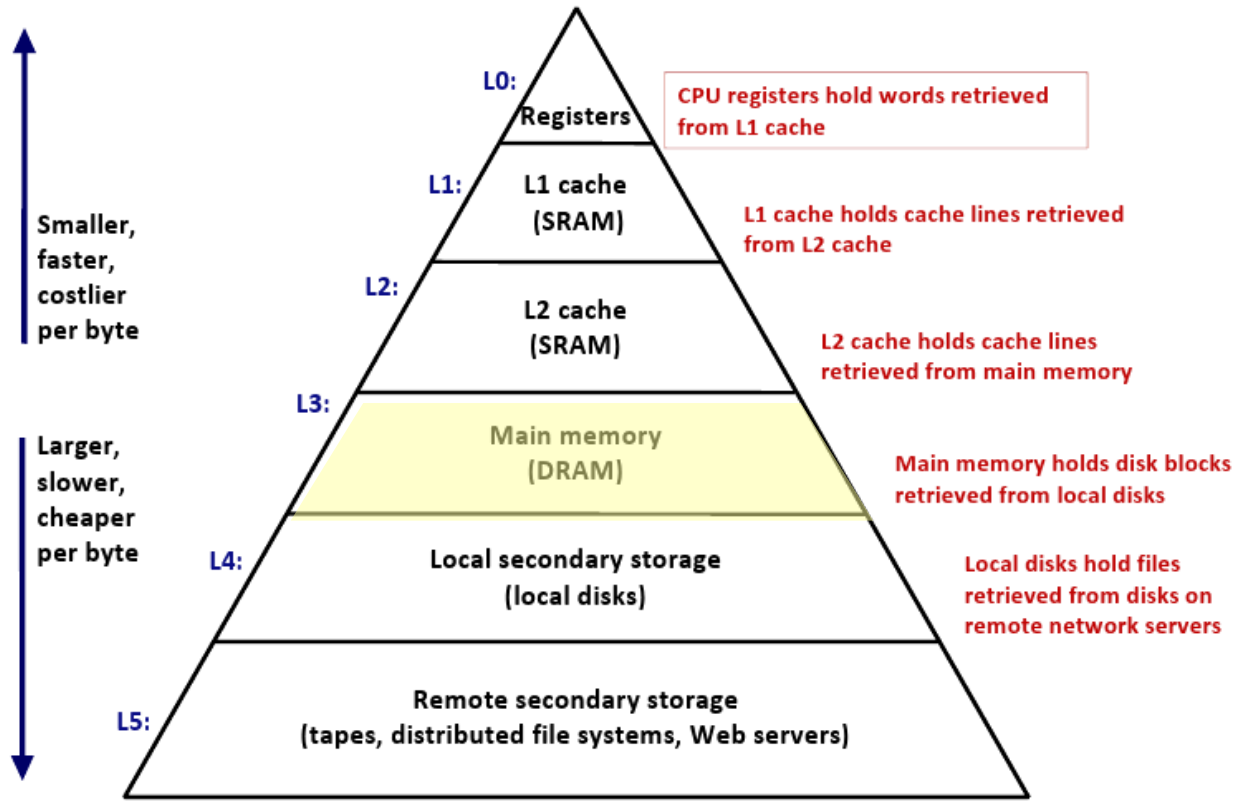
On-disk persistent graph:



Cache-Amenability of In-memory Data Structure

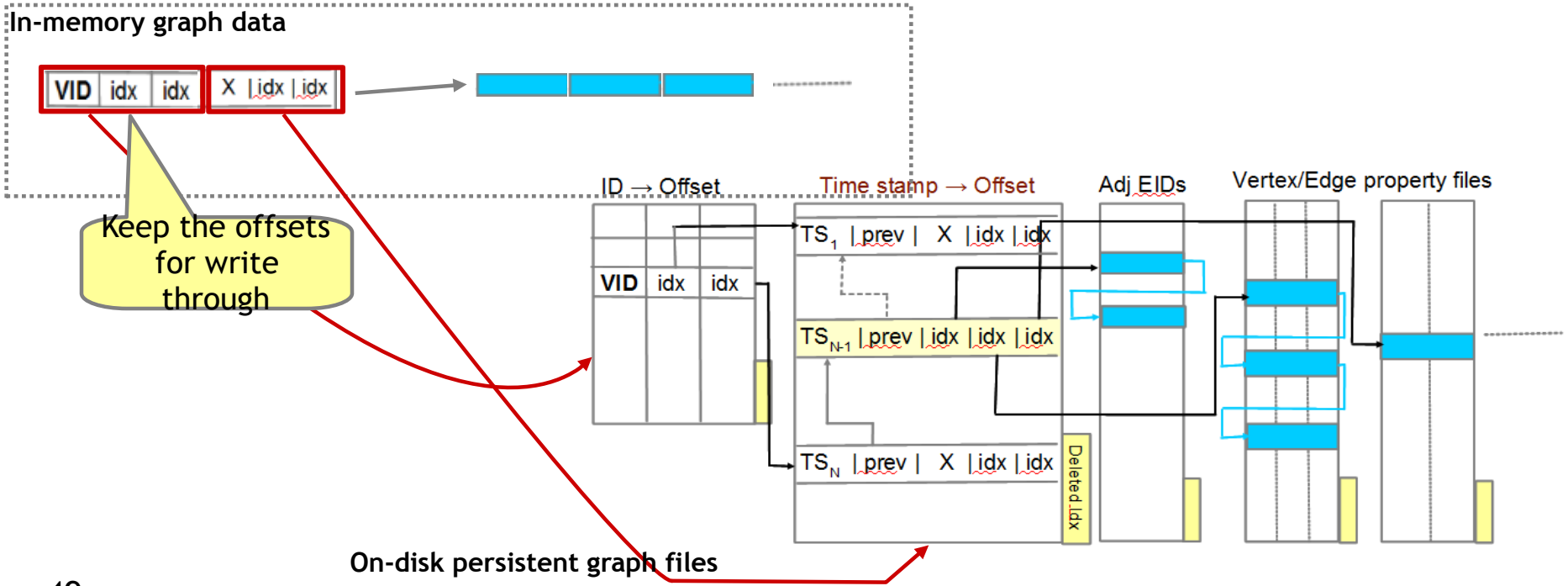
- Graph computations is notorious for highly irregularity in data access patterns, resulting in poor data locality for cache performance
- Data locality still exists in both graph structure and properties

Should we leverage the limited data locality and organize the graph data layout accordingly?



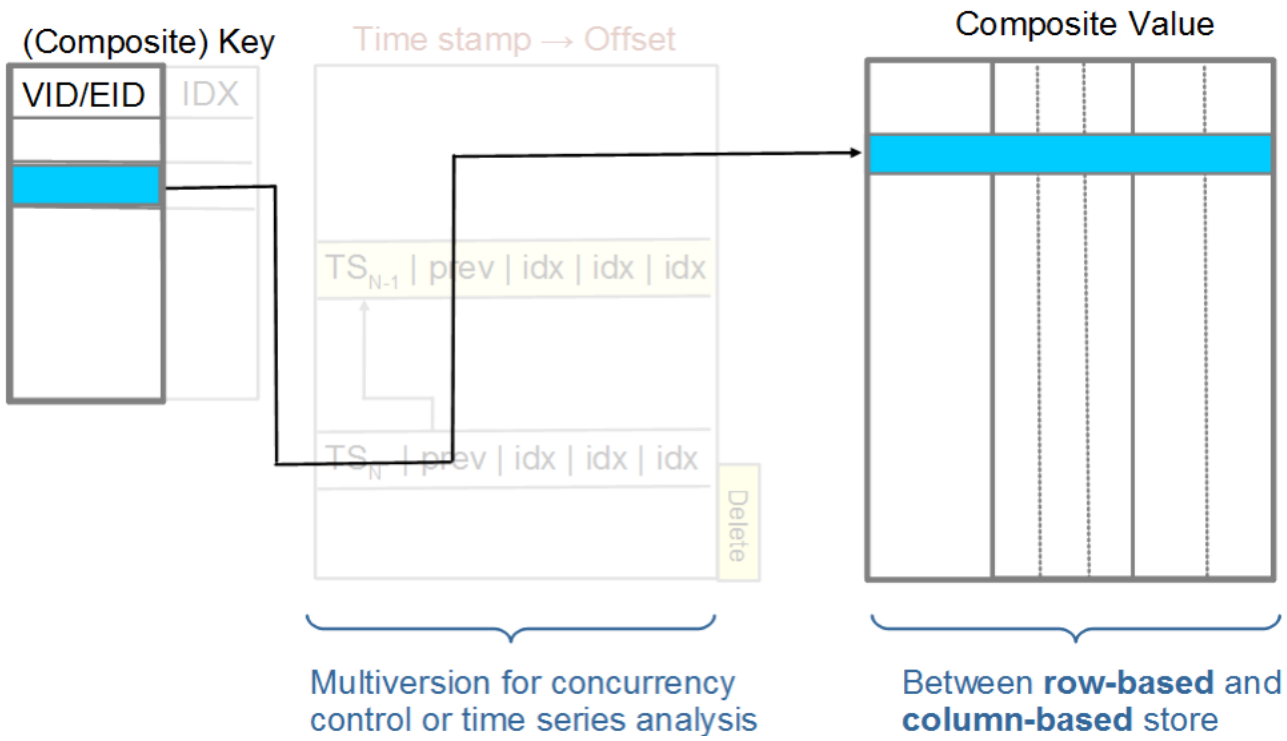
Reducing Disk Accesses by Caching

- **Cache graph data in memory either by:**
  - Caching in batch-mode leads to keep a chunk of the graph data files (lists) in memory for efficient data retrieval
  - Caching on-demand loads elements from on-disk files into memory only when the vertices and/or edges are accessed
  - **Stitching elements** from different data files together according to their VID or EID in-memory element, so as to increase data locality
  - Behaving as a in-memory database when memory is sufficient to host all graph data



Property Bundles — Column Family

- **The properties of a vertex (or an edge) can be stored in a set of property files, each called a *property bundle*, which consists of one or multiple properties that will be read/ write all together and therefore results in the following characteristics:**
  - Organizing properties that are usually used together into a single bundle for reducing disk IO
  - Organizing properties not usually used together into different bundles for reducing disk band width
  - Allowing dynamic all/delete properties to any property bundle

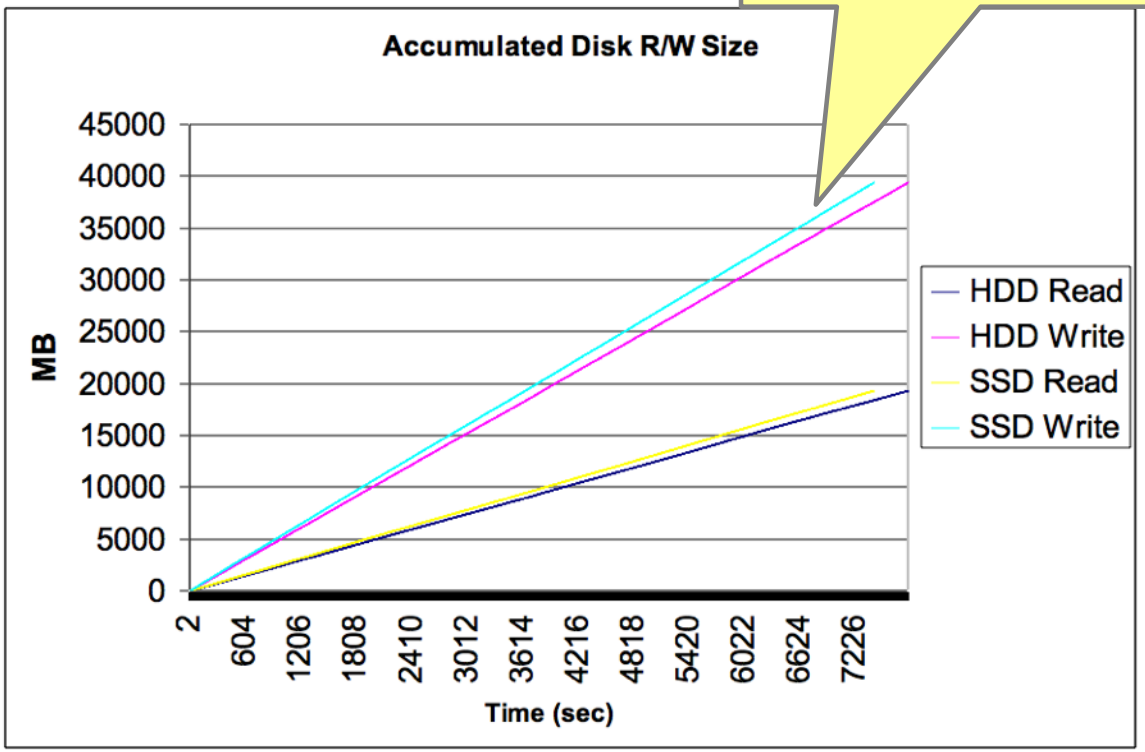


Impact from Storage Hardware

- Convert csv file (adams.csv 20G) to datastore
  - Similar performance: 7432 sec versus 7806 sec
  - CPU intensive
    - Average CPU util.: 97.4 versus 97.2
  - I/O pattern
    - Maximum read rate: 5.0 vs. 5.3
    - Maximum write rate: 97.7 vs. 85.3

Ratio HDD/SDD	TYPE1	TYPE2	TYPE3	TYPE4
	13.79	6.36	19.93	2.44

SSD offers consistently higher performance for both read and write

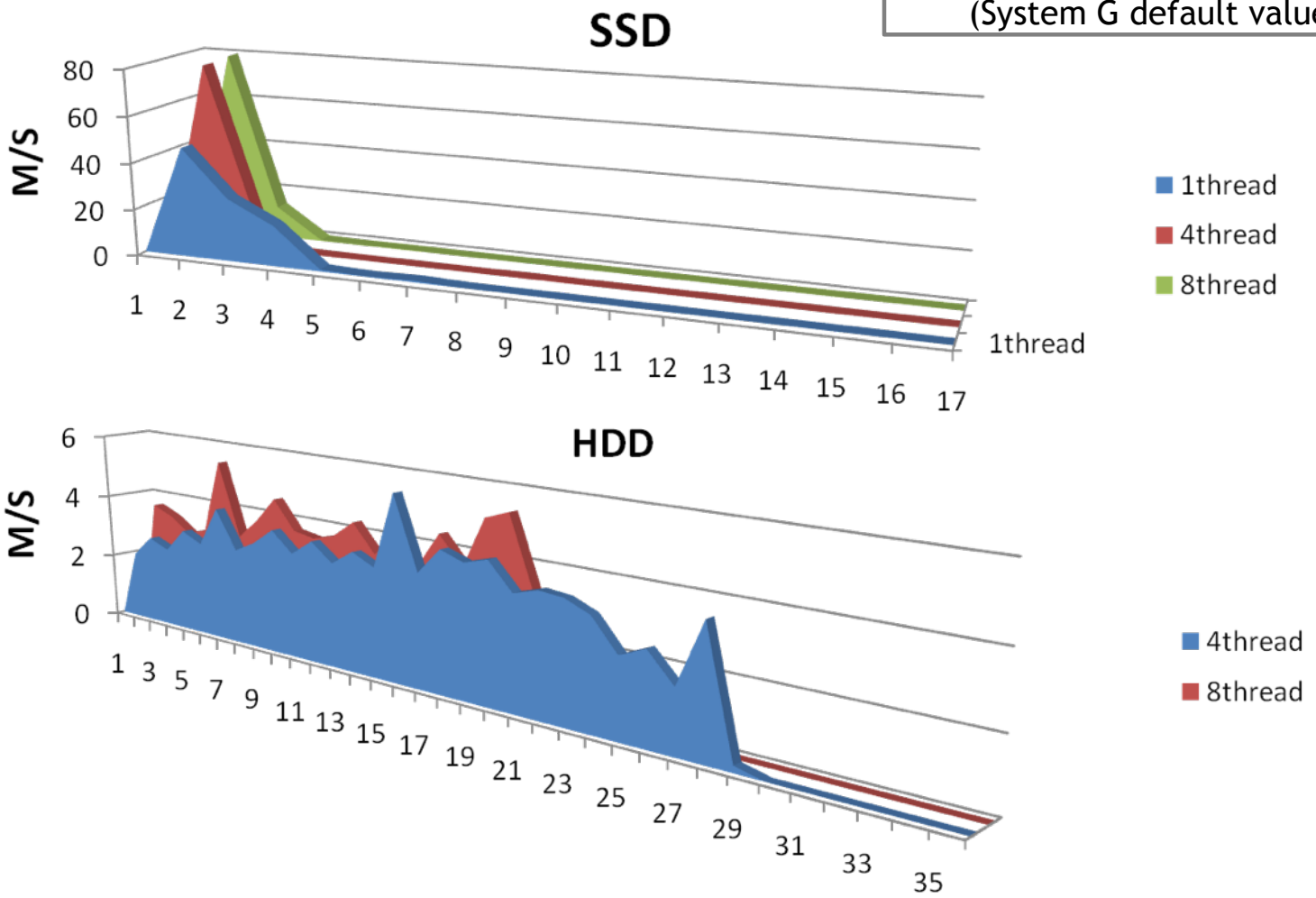


Queries

- Type 1: find the most recent URL and PCID of a user
- Type 2: find all the URLs and PCIDs
- Type 3: find all the most recent properties
- Type 4: find all the historic properti

Impact from Storage Hardware — 2

- Dataset: IBM Knowledge Repository
  - 138614 Nodes, 1617898 Edges
- OS buffer is flushed before test
- Processing 320 queries in parallel
- In memory graph cache size: 4GB (System G default value)



Homework 3

1. Spark:
   1.1: Install and test Spark
   1.2: Use the Wikipedia data you downloaded in Homework 2. Calculate TF-IDF. Do clustering. Discuss the outcome,comparing with what you did from Mahout.

2. Graph Database:
   2.1: Download IBM System G Graph Tools.
   2.2: Download URL links in the Wikipedia data. Create a knowledge graph. Inject it into a graph database. Try some queries to find relevant terms. This can serve as keyword expansion. Show some visualizations of your queries.

Questions?