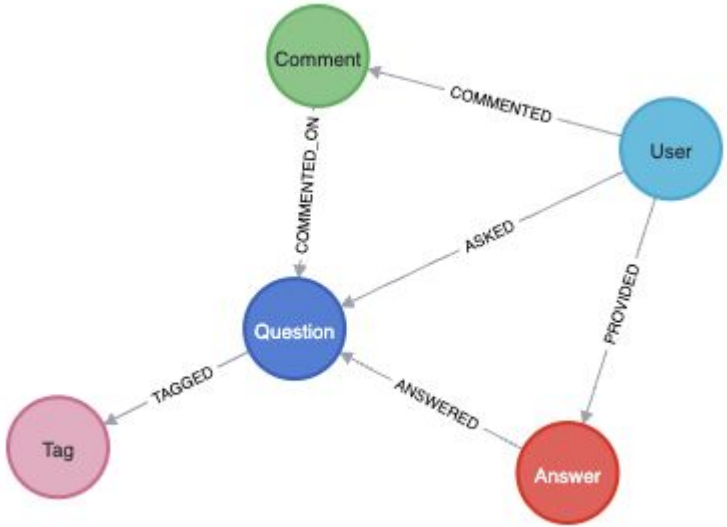
Intro to Neo4j

An expedited adventure

Rahul Singh and Devis Shehu

Graph Databases

- Uses Nodes, Edges, Labels to represent and store data
- Nodes are analogous to entities with the values being stored as Properties of these nodes. Usually have labels to facilitate categorization, filtering and visualization
- Relationships are directional connections between nodes (equivalent to joins) storing indices of the connecting nodes and may also have properties of their own
- Eg: In StackOverflow, entities such as Person, Post, Question are modeled as nodes. The relationship between entities are coded as edges, for example User “asks” a Question, or a Question “has” Comments



Motivation to use Graph Databases

Ideal for highly connected dataset for analytics

Graph databases store Relationships between Nodes as explicit entities, which eases traversing in highly connected models. In a way, the tedious process of performing multiple JOINS on multiple tables is replaced with jumping from one node to another which are directly connected via relationships. This saves time and resources for looking up common values for each join performed.

Intuitivity

Since the relationships are labelled, it is easier for one to follow and interpret how various Relationships work.

Flexibility

Graph databases are NoSQL databases, implying they are flexible in terms of schema. Additionally, CRUD operations can be performed without affecting the rest of the nodes or relationships. This is possible due to the storage mechanism where linked lists are maintained pointing to nodes and relationships, which only need to be altered.

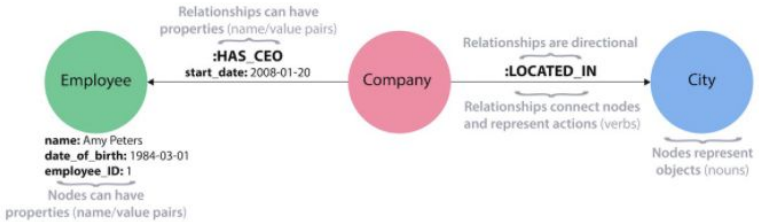
Business Use Cases
While Graph databases are highly specialized tools there are many common use cases including social networks, recommendation engine, enterprise knowledge bases, financial transactions fraud detection, managing IT infrastructure and services, network security forensics.

Graph databases excel for working with highly connected and large datasets in the context of Online Analytical Processing (OLAP) as opposed to transaction data.

Neo4j

Neo4j is one such Graph Database Management System

- Scalable, NoSQL and ACID compliant
- Native Storage and Processing
- Available as standalone desktop app, dockerized environment or serverless offering (AuraDB)
- Cypher is the Querying language used to create, update and retrieve data
- Finds its applications in common use cases including Social Networking, Recommendations, Knowledgebase, Fraud detection, Routing, Network ops



Data Storage and Organization

Intro to Graph DB: Architecture (~2015)

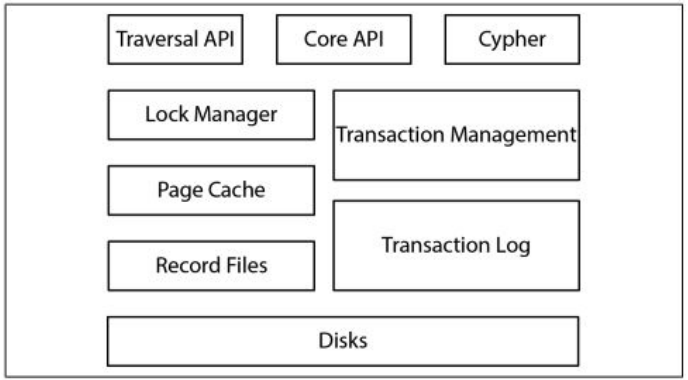
As with other DBMS, Neo4j is made up of a set of components and service that provide all features expected of a modern DB. The DBMS is complex so we touch only the user facing functionality in this slide.

At the highest level, user typically interact with Neo4j using Cypher query language. Neo4j also provides an imperative API for Java developers that allow them to interact with DBs primitives without having to write separate query code and to map DB nodes, relationship and attributes to in memory object. This is in essence analogous to ORM (object relational models) for Relational databases.

Traversal API provides a declarative model for iterating through relationships. API users effectively define filters that restrict how users iterates through relationship.

Core APIs in conjunction with traversal API can be used by developers to create highly tuned code. The disadvantage is the resulting code can be dense and difficult to maintain.

Next slide describes how data is organized on disk.



Transaction Management is handled by the Lock Manager. Transactions are in-memory objects that bundle and apply a lock to a set of updates, node/relationship creation / deletion. If there is an error, the transaction is rolled back. If successful the transaction is written to the transaction log and the associated data are written to disk.

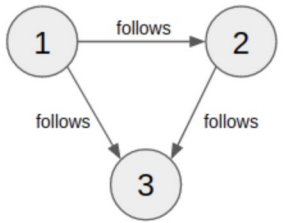
To optimize I/O, implements its own paging mechanism, LRU-K Page replacement algorithm to estimate and predict which parts of the graph database can be brought into and out of memory. Pages are evicted based on Least Frequently Used algorithm. See this paper.

Neo4j Modeling

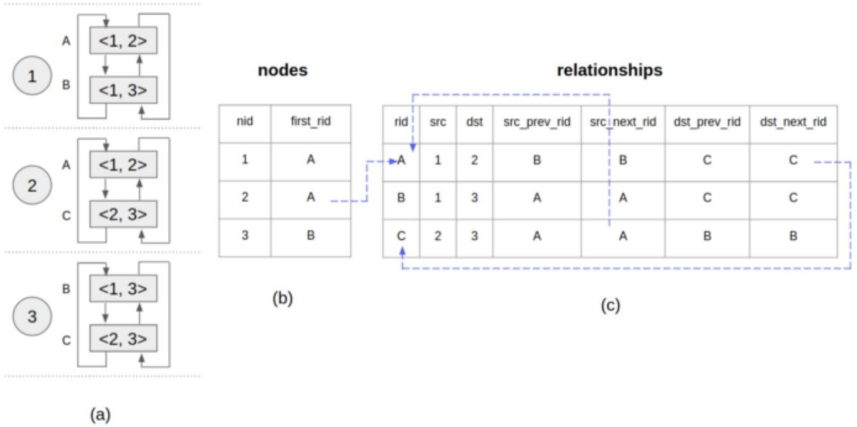
Edges/relationships are encoded as doubly linked lists with traversal from one node to an adjacent node in constant time. Each nodes store a pointer to the first relationship (first_rid) in the relationship table. See diagram to the right showing how a simple graph is structured on disk. Traversing each relationship for a given node pair requires following the respective pointer, previous or next.

Assume we're looking for all the relationships on Node 2

1. Start at Node 2 (dest.) and lookup the first relationship (A).
2. In the relationship table, look up the next relationship (A is dst node) for the dest relationship (C)
3. Follow pointer to get the 2nd (other relationship) - 2,3



Simple model of Twitter followers (left)
and representation in Neo4j (right)



Neo4j Organization & Storage

Data Stores and Files on Disk (Bytes)					
Nodestore.db (15)	Relationshipstore.db (34)	Propertystore.db (41)	Propertystore.db.strings (128)	Propertystore.db.arrays (128)	Indexed property (⅓ * AVG(X)

Node Store Records	
Bytes	Description
0	In used record (reclaimable)
1-4	ID of the first relationship
5-8	ID of the first property
9-14	ID to the storage label
15	Reserved

- **Simplified view** of the implementation as of v3.5
- **Schema-less** data store aspacked **fixed-length** records on disk
- Each record type has its own files/store: node, relationship, labels and properties
- Property record is key/value pair with pointer to the next property.
- Property records are tightly packed with each 32 byte record divided 4, 8 byte records that can hold a key, value or both.
  - Key/type use up 28 bits.
  - Simple types (ints, bools) are stored in the same block
  - Double (float) and 64 bit integers stored in separate blocks.
- If string or array won't fit then are placed in their respective store.
- Property points to the record index
- Nodes and relationships associated with via prev/next linked lists ptrs including labels and properties

As Neo4j relies on an index-free adjacency allowing each node to reference it's adjance node directly. Edges in Neo4j are doubly linked lists referencing both the previous and current node. Graph databases enables queries to perform at a constant rate regardless of size as opposed to O(log n) for B-tree. See next slide for more details on performance.

Relationship Store Records	
Bytes	Description
0	In Use/Other bits
1-4	First Node
5-8	Second Node
9-12	Rel Type
13-16	1st Prev-Relationship
17-20	1st Next-Relationship
21-24	2nd Prev-Relationship
25-28	2nd Next-Relationship
29-32	Next Property ID
33	First in Chain Marker

Performance

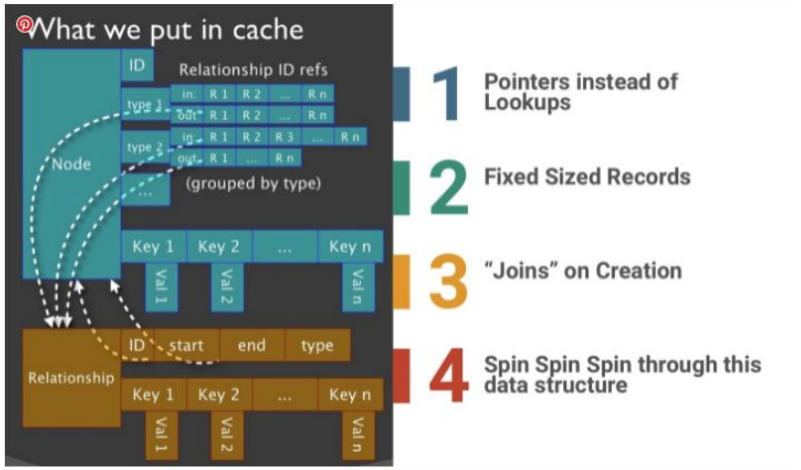
Performance Overview

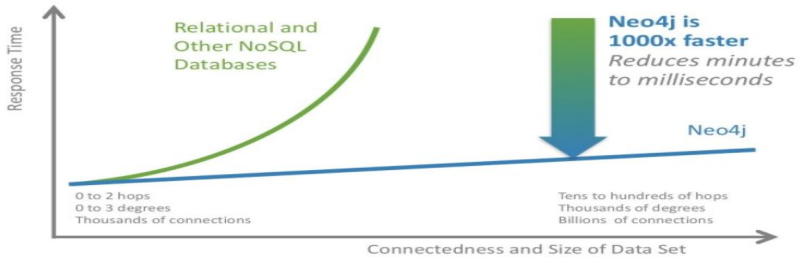
The diagram to the right shows how memory access is optimized in Neo4j especially with joins.

- Index lookups are replaced with pointer lookup
- Fixed-size records allows execution engine to calculate where in memory a record is stored and jump to it
- Data paged in & out of memory/cache. No file-seek or sequential scans to disks except with attributes without indices or when dealing with extremely large graphs

A unique feature of Neo4j is that “joins” are created on the fly as nodes are associated via relationships. For larger datasets, the cost is amortized as we build up the data structure and is proportional to the local subgraph. I.e. not the entire dataset.

Since updates do not typically cascade into larger update across the whole db, as with relational db indices this is a huge performance improvement on relational databases that might be mimicking relationships using 2,3,4 way or self-join. This is much clearer when looking at the relational vs graph db slide below. With relational databases, optimizing these queries requires the use of temp tables, materialized views and table partitioning and many other complex and bespoke (unique to the code) solution. And performance will still not scale proportionally.





Optimization & Scalability

Indices

Typically user queries traverse the nodes and relationship of a graph to gather information. However, there are situation where it's useful to filter nodes on specific attributes: categorization of nodes, additional attribution that can't be coded as a node) Cypher allows for the creation of indices on label and/or properties, or combinations similar to composite keys in relational DBs.

Uniqueness constraints are also supported.

CREATE INDEX on :User(role)

Neo4j supports the following index types:

- B-tree: map property to a node/edge typically for looking specific nodes, relationships via key/value.
- Token lookup: matching a node through label and created by default as of version 4.3. Avoids scanning and filtering on nodes.
- Full-text indexes: indices to speed up searching of text

Hardware Scaling

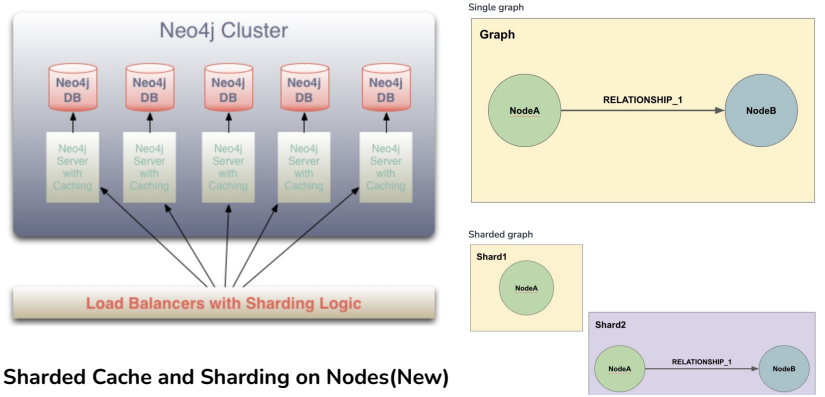
Originally, Neo4j could only scale horizontally via replication, i.e. copies of the database on multiple service. Neo4j cache sharding routes requests to database instance that holds the desired sub-graph in memory. See the 1st diagram on the left.

Conventional sharding can't be used to split up the network since graph relies on physical connection between nodes and maintaining ACID compliance is NP-hard problem. Algorithms for partitioning relations aren't compatible with Neo4j's data structures.

Starting with v4, Neo4j can distribute subgraphs to other servers similar to conventional sharding. Neo4j implements this by duplicating nodes which hold relationships between 2 shards. Queries to the cluster are proxied (relayed) to and from nodes in the cluster via a feature called Fabric.

At a high level Fabric can be thought of a virtual database that appears to the user as a single representation of the entire graph. Queries will hit Fabric first and then get routed to the appropriate instance which needed data. Results are aggregated if data is returned from multiple instances. This is still 1 cluster but with multiple instances.

Fabric can be extended across multiple clusters each with their own instances. But there is still a single coordinator to route query and results which is a bottleneck. There is a significant bottleneck when dealing with large volume of writes which is exacerbated by sharding.



Sharded Cache and Sharding on Nodes(New)

Demo

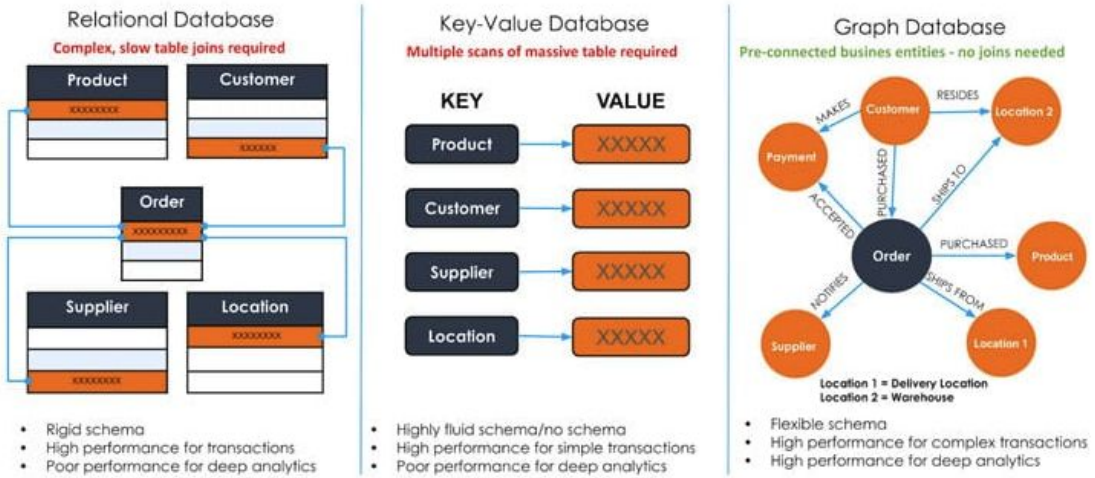
Demo & Walkthrough of Tools

- Demo Stack: Jupyter, Vendor Neo4j driver, Aura DB & Self-Hosted Database
- Part A - Simple Scenarios: Persons & Relationships (Offline Review)
  - Uses the Aura database instead of a self-hosted database
- **Part B - Working with Real World Data: Stackoverflow**
  - Describe and quickly demo available tools: Neo4j Browser & Tutorials, Sandbox, Aura, Cypher and CLI DBMS.
  - Initialize the database by importing JSON data via API
  - Transforming JSON to nodes, relationship and properties
  - Run some insightful queries: filtering by topic/category, top users (most active, most authoritative, highest rated), quiescent topics, relationships and sub-communities (social networks) and deriving categories
  - Embedding visuals in Jupyter notebook (not sure if this is working?)
- Part C - Katacoda Deployment Lab for MSDS students (for Offline Review)
  - First time developing with this software, student feedback is welcome

Additional Materials

Relational, NoSQL and Graph

The diagrams below highlights the different ways that customer orders can be modeled and the advantages/disadvantages of each database model. In a real world use case, it's likely that users and application would utilize a Relational or NoSQL to collect, edit and present the data while graph database would be used for deep analytics such as recommendations. For data collection, and traditional transactional apps, If the schema is mostly static, as with mature ecommerce applications, then a relational DBMS with high performance, consistency and constraints is ideal. On the other hand if the product catalog is changing frequently, with new categories and attribution then the Key/Value store might be best as the model can be modified easily as product changes. It is possible to model Graph data using relational DBMS. This would be a non-native implementation of graph. See slide below for more on native databases and the following link for modeling anti-patterns.



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

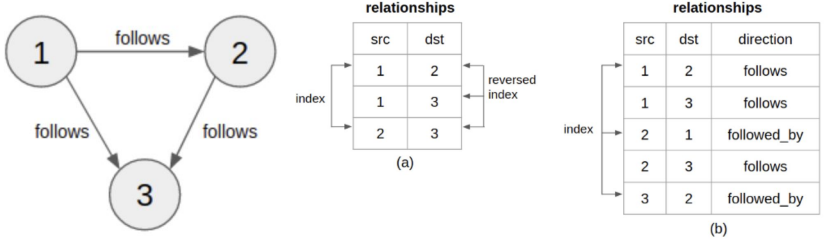
When doing **deep analysis,** for example looking at purchase history and order sequence, customer demographics and income, such as in with product recommendations then the performance characteristics of Neo4j allows for exploration of complex causal relations. As can be seen in the chart above, execution time for traversing relationships increases exponentially and much faster with DBMS.

Native vs Non-native

Neo4j is a native graph database. The underlying storage, query engine and APIs were designed for optimal efficiency and ease of use (Cypher, Core API) for graph queries. A non-native graph database can be built using relational or other data stores (NoSQL). This approach has advantages as users will already be familiar with the relational or NoSQL model. But performance is suboptimal and not scalable. Complex queries (multi-hop) will be much slower and there will be issues, i.e. sharding, partitioning, as the graph grows beyond a few thousand nodes. See below.

Neo4j implementation of double-linked list for traversing nodes and relationships, a sharded cache for scalability and a compact and efficient organization (locality of data/caching) minimizes I/O with queries. With fixed-length data structures efficient lookups via pointer indexing is possible unlike with relational models. See slide on performance.

- On the right a highly simplified model of “Twitter’s” followers
- Relationship: src (leader), dst(follower)
- Variant (a), relationship is stored as index (from leader to follower), reverse index for the other direction
- Logarithmic time on average depending on index implementation. However, reverse index can’t be sparse
- Variant (b) addresses the issue with reverse index but introduce issue with consistency (more rel. to maintain)
- Each relationship encoded twice via direction field
- Indirect relationships (give me all followers) is complex to code and much slower. Index expensive to maintain



Summary of Pros & Cons

Pros

- Ideal for highly connected dataset for analytics; easier and more natural to model relationships

```
MATCH (n:User)->(r:Owns)->(n:Boat)
```

- Relationships are 1st class citizens of the graph model where as in relational database inferred from foreign keys (expensive)
- As in NoSQL DBs, schema is highly extensible and adaptable via labels/properties
- Big performance gains when querying localized subgraphs vs global indexed searches and joins
- Transaction and ACID: means the database is robust and reliable under load and handles errors gracefully
- User friendly query, visualization, management tools

Cons

- Inefficient for high-volume, transactional data
- Cypher not industry standard. No SQL-like universal cross-vendor language.
- Not every problem can be modeled using graphs and graphs are not the most efficient organization structure (geospatial)
- Queries that operate over the whole dataset or a large subset are not as efficient (table scans, searches on predefined categories, aggregation)
- Overkill for simpler use cases such as storing key/values (example: user profile database)
- Inefficient with large objects (BLOBs), text data
- Not as easy to scale to huge datasets as NoSQL or relational (see limitations on sharding)

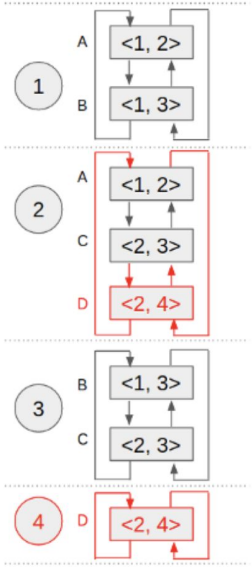
Algorithm for Adding Nodes

Key steps, see diagram on the right

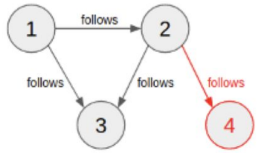
- A new Twitter follower (Node 4) is added
- Associated Nodes updated with pointer to the new relationship (D)
- Node 2 now has 3 relationship which requires updating the doubly linked list for the other relationships (previous and next) - cols 5,6

How does this compare to updating a (sparse) index on relational database?

- Additional overhead new node, new relationship, (2 pointer updates).
- Still a “fixed” amount of work per join: proportional to the number of associated nodes and relationship and not the entire graph



(b)



(a)

nodes

nid	first_rid
1	A
2	A
3	B
4	D

(c)

relationships

rid	src	dst	src_prev_rid	src_next_rid	dst_prev_rid	dst_next_rid
A	1	2	B	B	~~C~~ D	C
B	1	3	A	A	C	C
C	2	3	A	~~A~~ D	B	B
D	2	4	C	A	D	D

(d)

Cypher Queries Cheat Sheet

Creating Nodes, Relationships

CREATE (aNode) : create a single unnamed node
CREATE (aNode), (bNode) : create multiple nodes
CREATE (aNode:aLabel {aKey: aValue}) : create the node, label it and assign one or more properties.

CREATE (node1:label1)-[:RelationshipType]->(node2:label2) : create a pair of nodes and a relationship of the given type.

CREATE (node1)-[label:relationshipType {properties}]->(node): create a relationship with type and properties.

Updates & Merging

MATCH (node:label{properties}) SET node.property = value : create a new node property or update/delete the existing property

MATCH (node {properties}) SET node:label : set label to a existing node

MERGE (node:label {properties}) - it searches for the specified node with given properties and label, if doesn't exists new node will be created

Querying

MATCH (n) RETURN n : return all nodes
MATCH (n:aLabel) RETURN n : returns nodes with label A

MATCH (node:label)<-[:Relationship]-(n) RETURN n : returns node from the given relationship

MATCH (node) WHERE node.property = "value" RETURN node match nodes where the property has a given value

Deletion

MATCH (n) DETACH DELETE n - this will delete all nodes and associated relationships from the db

References

Graph DB & Neo4j Concepts

- Introduction to Graph Database
- Modeling and Querying Graphs with Neo4j
- Native Graph-Database Storage
- Neo4j performance and data structure on disk
- Graph Databases for Beginners: Native vs. Non-Native Graph Technology
- Welcome to the Dark Side: Neo4j Worst Practices (& How to Avoid Them)

Application, Platform and Performance

- Neo4j Python API examples and getting with Cypher and quick intro to common commands
- Integrate Neo4j with Jupyter
- Neo4J Sandbox: free time-limited environment with built in dataset, browser, tutorials and walkthrough
- Aura: a serverless, fully-managed, database for development and production w. free tier
- APOC: plugin that extends Neo4j allowing import and export to CSV, JSON and XML

Data

- Stanford Large Network Dataset Collection: research database covering a wide variety of topics including social media, sales and research
- Finding Data for your Graph Adventures with Neo4j: more source
- Neo4J Graph Datasets: graph databases store in native Neo4j format; data dumps can be imported via browser sans plugin
- StackExchange API: API to get developer Q&A; used to construct the data for Part B of the demo

Other Resources

- Katacoda Neo4j Tutorial and associated code
- Neo4j Custom Browser & Tutorial (future/advanced)
- Comparing graph database

Alternatives to Neo4j

- Amazon Neptune: managed cloud hosted-database from AWS, used at Uber's Self Driving Group and many other organizations
- TigerGraph: costly but popular with financial services and Data Scientist due to mature, efficient implementation