Graph Database Fundamentals

Jeremy Deane – jeremy.deane@gmail.com

By 2019 it is projected that data exchanged on the internet will reach two zettabytes per year. That is an astronomical number, but it is dwarfed by the amount of data flowing internally between systems. Harvesting value from these information exchanges is essential in this highly competitive global economy. Arcane Data Warehouses, traditionally used to extract value from this data, are being replaced with Lambda Architectures allowing for both real-time and deep computational analysis. While technologies such as Hadoop and Elasticsearch, Logstash, and Kibana (ELK Stack) support these advanced analytical capabilities, they fall short in one area: connectedness. Graph Databases address this capability gap and so much more.

Graph Theory

Graph Databases are based on Graph Theory, something everyone has been painfully exposed to in high school. Who can forget the traveling salesmen questions such as finding the optimal route via Philadelphia, Columbus, Madison, and Chicago. Unless one planned to be a traveling salesman someday, selling shower curtain rings, these exercises were exasperating. Fortunately, there are a myriad of pertinent modern use cases where graph theory is applicable. Graph Theory is fundamental to Social (Professional) Networks, Supply Chain Logistics, Network and System Operations, and Analytics.

Social networks and retail organizations use graphs to make recommendations, determine personal and group affinity, and most importantly to explicitly or implicitly connect individuals and entities. In fact, with graphs one can measure the importance of a relationship as it strengthens or atrophies over time. Industrial organizations use graphs to design and optimize supply chains, while technology organizations use graphs to map system dependencies, route traffic, and identify bottlenecks. Finally, graphs can be used to rapidly identify irrational connections that might be an indicator of fraudulent activity.

Graphs Theory states that a graph is a representation of a set of *objects* where some pairs of objects are connected by *links*. Objects, a person, place or thing, are vertices (a.k.a. Nodes) while links are edges (a.k.a. Relationships) that connect two objects. An edge can only exist between two vertices. In other words, no "dangling" edges. Furthermore, edges can be directed or undirected. Finally, a *path* represents a way to get from one vertex to another by *traversing* a set of edges.

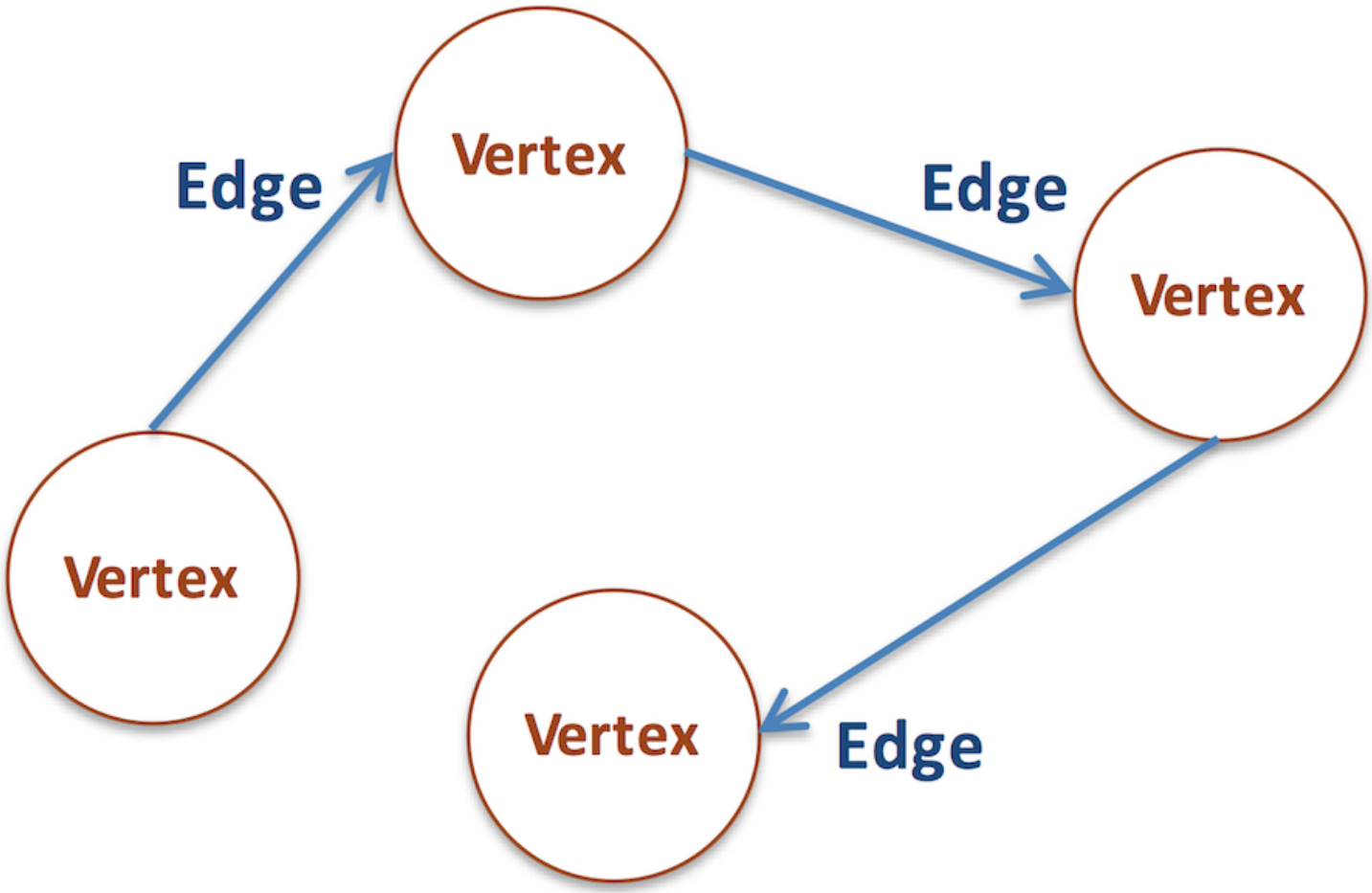

Figure 1. Graph Theory

A specific type of graph is a Property Graph that is made up of nodes and directed relationships with properties or key-value pairs. Neo4j implements an enhanced property graph that supports categorizing nodes with labels. This allows a query to find a starting point in the graph by looking up a node with a specific property or label or both. And because nodes are directly connected via relationships, traversing from one to another is incredibly performant. This capability is referred to as index-free adjacency.

Graph Database Modeling

Modeling graph databases is incredibly white board friendly and does not require participants to have technical background, just a familiarity with the business domain. Nodes are represented with circles and relationships with arrows. These models can be captured in ubiquitous modeling tools such as Microsoft Visio and OmniGraffle. In Figure 2, patients are treated by practitioners who work for an organization and work at a location; organizations maintain those locations.

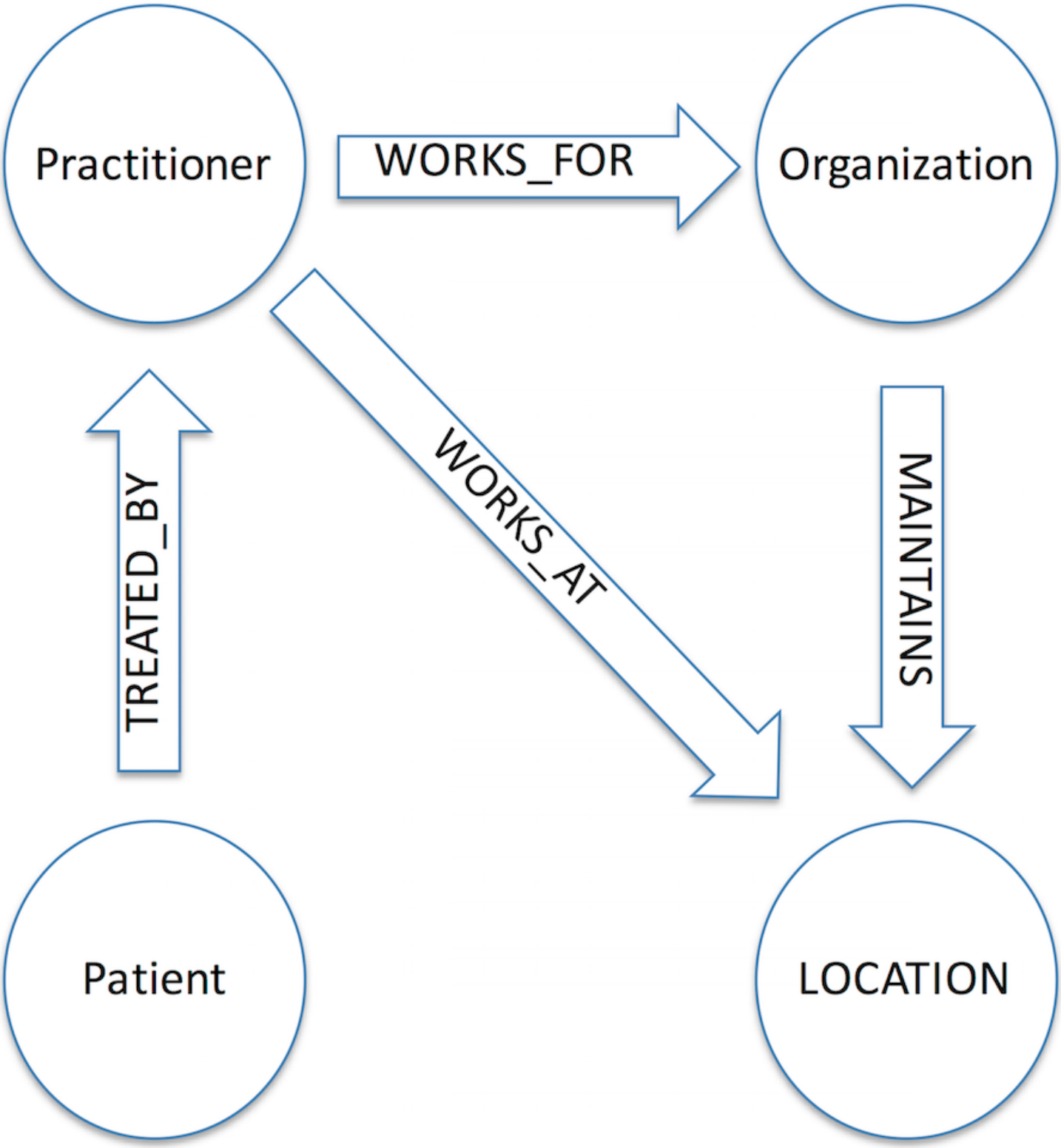

Figure 2. Conceptual Graph Model

While a conceptual view can be quite useful, modeling is often performed with more concrete nodes and relationships. With this bottom up approach, repeatable templates of nodes, relationships and paths emerge forming an implicit schema. This lack of formal structure can be unnerving to those that have spent years working with relational databases where one worries about such minutia as whether a column is functionally dependent on a primary key.

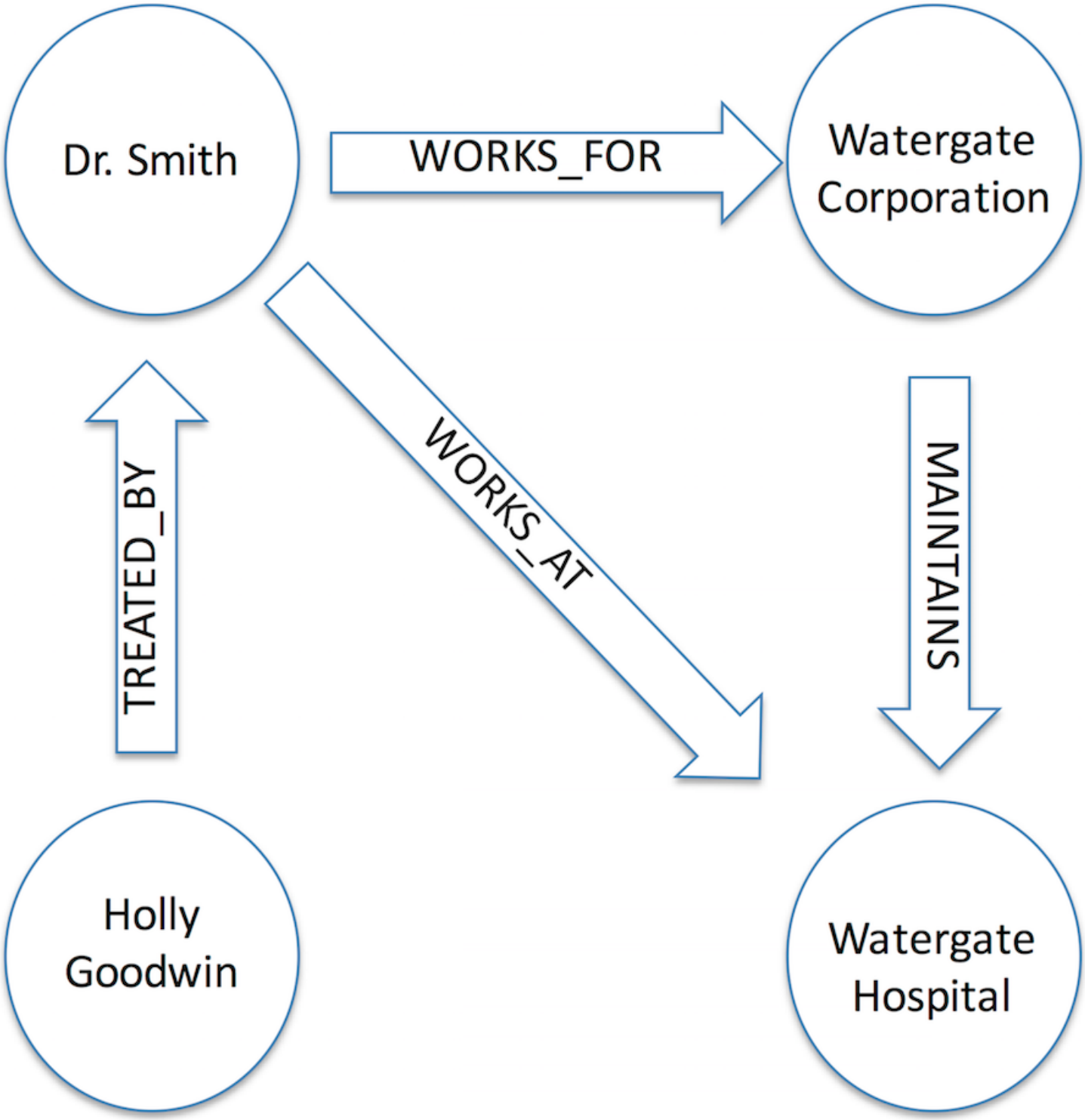

Figure 3. Physical Graph Model

Nodes and relationships are enhanced with properties. A node should be a lightweight representation of an entity with links to the gross details stored in a aggregate or relational database. Similarly, relationships properties should *minimally* enhance the semantic meaning of the relationship, such as a weight or rating, with the facts to support those properties stored in an aggregate or relational database.

Properties are key-value pairs with the key represented as a string and the value represented as a primitive type, such as a string, long or boolean, or represented as an array. A property value cannot be null. Properties can be indexed for all nodes with a given label. In fact, nodes are often categorized with one or more labels. In addition, properties can be constrained for uniqueness and existence.

Generic relationship names such as HAS, IS, and CONTAINS should be avoided. Instead use explicit terms such as OWNS, FOSTERS, or combine terms such as WORKS_FOR. In addition, avoid duplicate relationships such as Brad owns Mustang and Mustang owned by Brad. Even though relationships are directional they can be traversed in either direction. Finally, leverage an intermediate node for n-ary relationships.

Suggested Naming Conventions

Model Concept	Style
Labels	CamelCase
Relationships	SNAKE_CASE_UPPER_CASE
Properties	snake_case_lower_case
Indexes & Constraints	snake_case_lower_case

During the modeling phase it is common to have one person at the white board and another at a computer creating the graph on the fly, using the Read–Eval–Print Loop (REPL) that is part of the Neo4J Web Console, providing constant immediate feedback. This is possible because the graph model maps directly to the primary language used to interact with Neo4j, the Cypher Query Language (CQL). In fact, some developers prefer to start with Cypher, discovering the conceptual model based on what information is required by the ultimate consumers of the graph database.

Graph Database Development

Cypher is the most popular and preferred way to interact with the Neo4j graph database for several reasons. First, the language is written in a very intuitive ascii-art format that maps directly to graph models. Second, the language is somewhat similar to Structured

Query Language (SQL). Third, and where it differs from SQL, is that Cypher is essentially a declarative pattern matching language eliminating the need for tedious imperative node traversals.

Cypher Code Examples

```cypher
//1. Retrieve practitioner Node
MATCH (p:Practitioner) WHERE p.specialty="Neurosurgery" RETURN p.name, p.specialty

//2. Retrieve all Nodes with Label patient and with diabetes
MATCH (m:Patient) WHERE "Diabetes" IN m.conditions RETURN m

//3. Retrieve all Nodes with WORKS_AT Relationship
MATCH (a)-[r:WORKS_AT]->(b) RETURN a,r,b

//4. Find patients who are also practitioners
MATCH (m:Patient), (p:Practitioner) WHERE m.name=p.name RETURN p
```

The fourth Cypher example would be difficult to support in a relational database because of the recursive nature of the query. In fact, with each increase in degrees of relationship separation, relational databases experience an exponential slow-down in query performance. In addition, the number of records contained in a relationship database table also correlates directly to its performance. This is not true for a graph database. The first Cypher example will return in relatively the same amount of time whether there are eight hundred or eight million practitioners.

Cypher is also used to create and update graphs. Cypher statements are submitted individually for creating or updating a specific node or relationship. Alternatively, Cypher statements can be chained together to create a number of related nodes and corresponding relationships. Cypher also supports bulk loads by binding, at execution time, to logical columns stored in a Comma Separated File (CSV) file.

Cypher Code Examples

CYPHER

```
//5. Create Node
CREATE (:Practitioner {name:"Zachary Smith", specialty:"General Medicine"})

//6. Update Node
MATCH (p) WHERE p.name="Zachary Smith" SET p.specialty="Neurosurgery"

//7. Create Nodes and Relationship
CREATE (m:Patient {name:"Jackie Bonk", birth_date:"1978-12-15"})-[r:TREATED_BY]->
(p:Practitioner {name:"Yuri Zhivago", specialty:"Immunology"})
RETURN m, r, p

//8. load caregiver nodes
LOAD CSV WITH HEADERS FROM "file:///YOUR_LOCATION/CaregiverNodes.csv" AS csvLine
CREATE (g:Caregiver {name: csvLine.name, guardian: csvLine.guardian}) RETURN *
```

The eighth Cypher example shows how easy it is to expand and refactor the graph database.

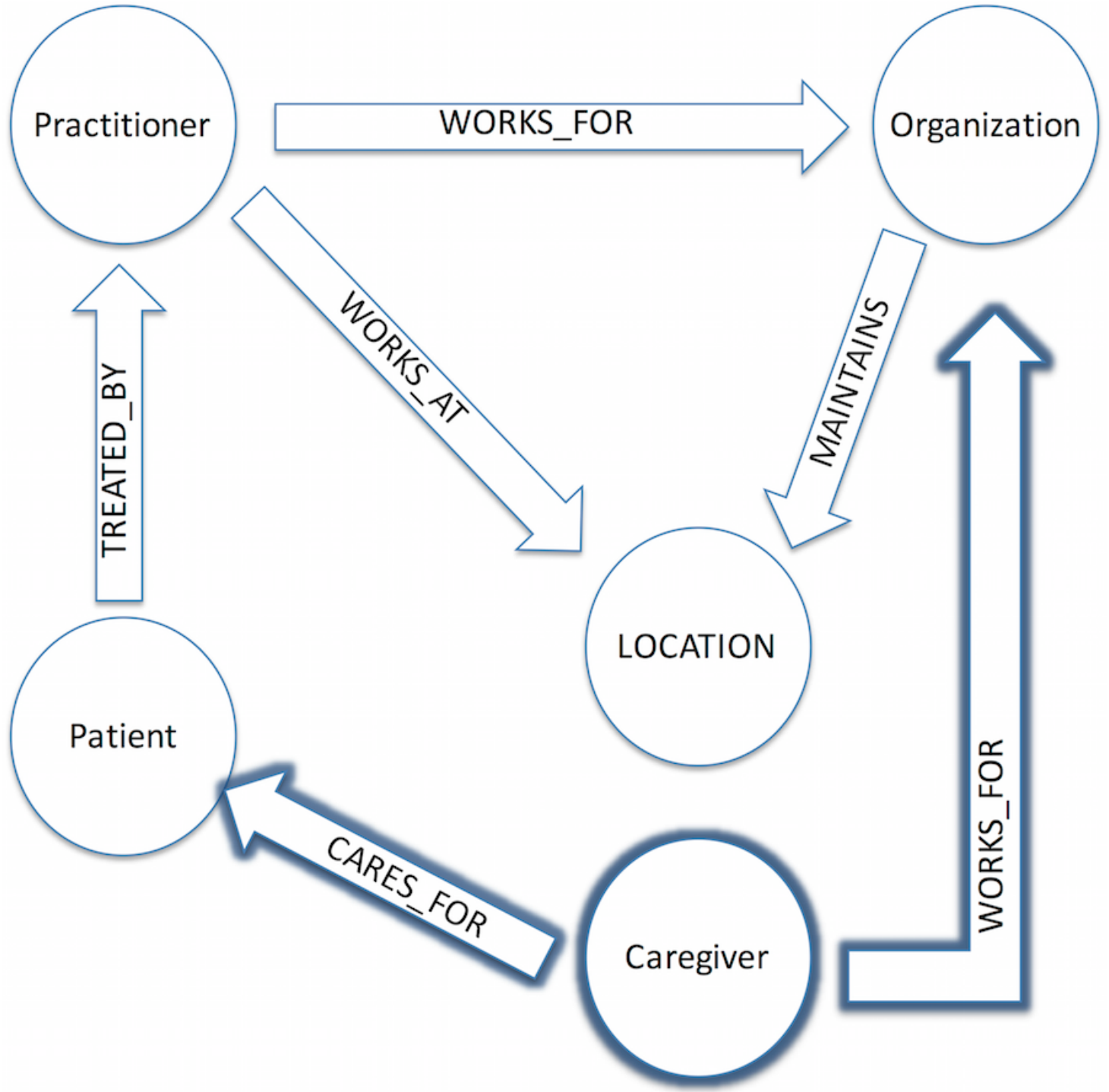

Figure 4. Extended Conceptual Graph Model

The graph model now includes caregivers who care for patients and may work for organizations. Labels offer a powerful mechanism for enhancing this graph with a number of other future topics such as disease management programs or geographic coordinates that would enable calculating the shortest distance from a patient to a practitioner with a specific specialty. Labels also allow consumers to exclude vast parts of the graph and focus only on the parts of the domain that are pertinent to them.

Cypher Code Examples

```
CYPHER
//9. All paths to Lovee Johnson
MATCH paths = (m:Patient)-[*]-(node) WHERE m.name="Lovee Johnson" RETURN paths

//10. Shortest path from Lovee Jonhnson to Florence Nightingale
MATCH (m:Patient {name:"Lovee Johnson"}),
(g:Caregiver {name:"Florence Nightingale"}),
path = shortestPath((m)-[*..10]-(g)) RETURN path

//11. create index on Patient Label
CREATE INDEX ON :Patient(name)

//12. Profile complex query
PROFILE MATCH (m:Patient), (p:Practitioner) WHERE m.name=p.name RETURN p
```

Extracting paths for analysis is a critical activity required to support such activities as intelligent recommendations, efficient route planning, and supply chain optimization. Of course, a starting point is required for a path and an indexed property is often used to find it. Indexes can be configured automatically, for example creating an index on all node properties with key *name*, or they can be created manually using Cypher (example 11).

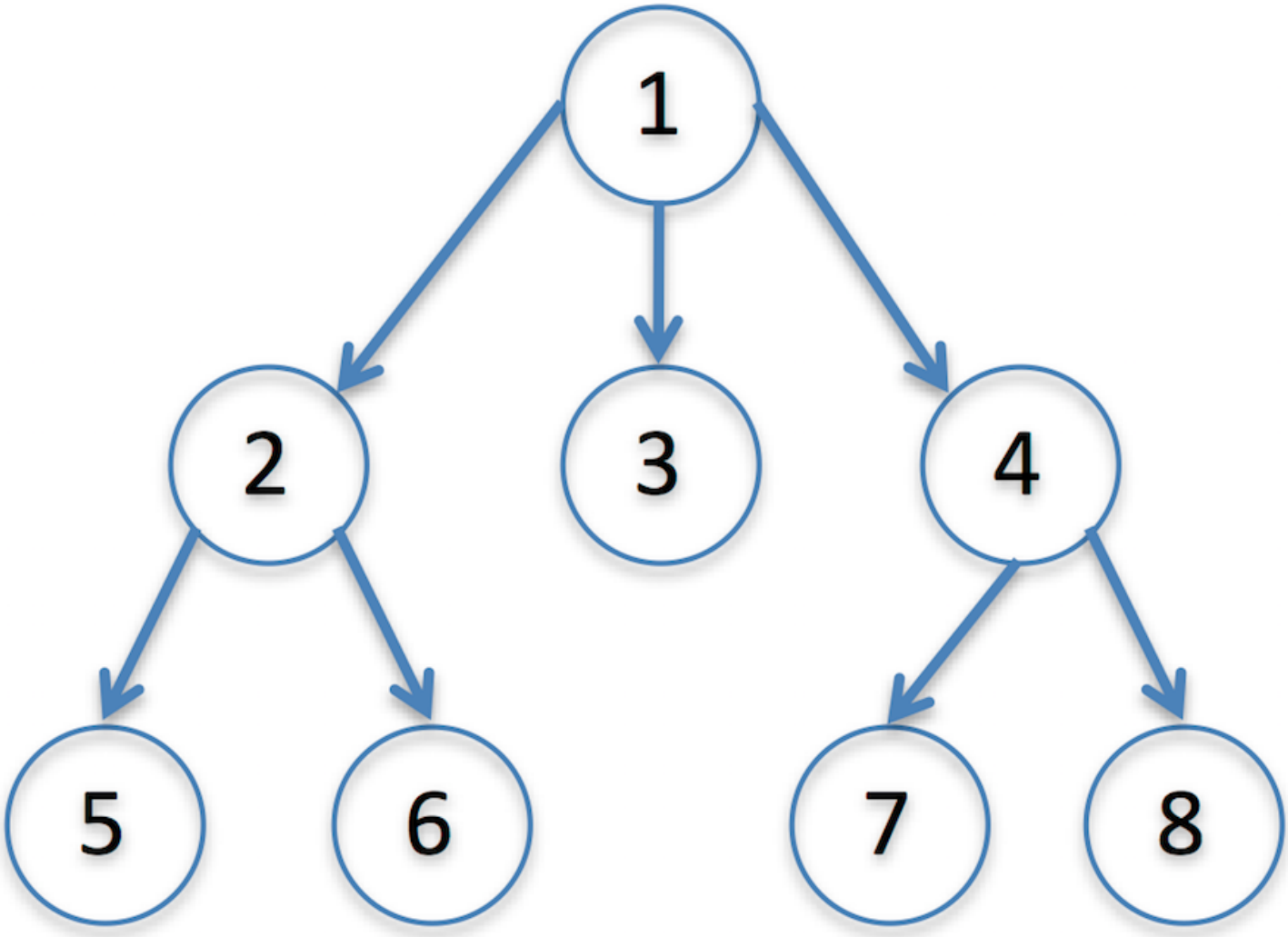

Figure 5. Graph Search

Cypher uses Depth-first search (DFS) by default. Breadth-first search (BFS) is supported by using Neo4j's Traversal API. In Figure 5, the path for DFS is 1,2,5,6,3,4,7,8 and for BFS is 1,2,3,4,5,6,7,8. In general, if one is searching a large number of nodes connecting to one central node (think hub-spoke) then BFS would perform better than DFS. But for most scenarios DFS is more than adequate.

Traversal API Example

```java
//Example Traversal with Breadth-First Search (BFS)
Node doctor = graphDB.findNode("PRACTITIONER", "name", "Zachary Smith");

TraversalDescription doctorsNetwork = graphDB.traversalDescription()
        .breadthFirst()
        .relationships(RelationshipTypes.TREATED_BY)
        .evaluator(new SomeCustomEvaluator())
        .evaluator(Evaluators.atDepth(1))
        .uniqueness(Uniqueness.NODE_GLOBAL);

Traverser traverser = doctorsNetwork.traverse(doctor);
```

A number of powerful graph visualization solutions exist such as Data-Driven Documents (D3.js), Alchemy.js, and Linkurios.js. However, most of these solutions do not process the JSON returned from Neo4j REST API. These solutions require some server or client-side processing to extract the nodes and relationships into arrays. But once the pre-processing is complete one need only configure a few properties and select or apply style sheets to display stunning graph visualizations.

The Cypher examples used in the article are from a Graph Database Workshop (https://github.com/jtdeane/graph) available on GitHub.

Graph Database Architecture

The Neo4j graph database runs in a Java Virtual Machine (JVM) either embedded in another application or as a standalone server. An open source version exists, but it is covered by the GPLv3 license which has a “copy left” provision requiring anyone using Neo4j to open source their code. Established organizations are more likely to purchase the Enterprise Edition which does not have any code sharing requirements. The Enterprise Edition is based on cores but that does not preclude an organization from having more than one instance taking advantage of the same cores; just remember to change the default ports numbers for one of the instances.

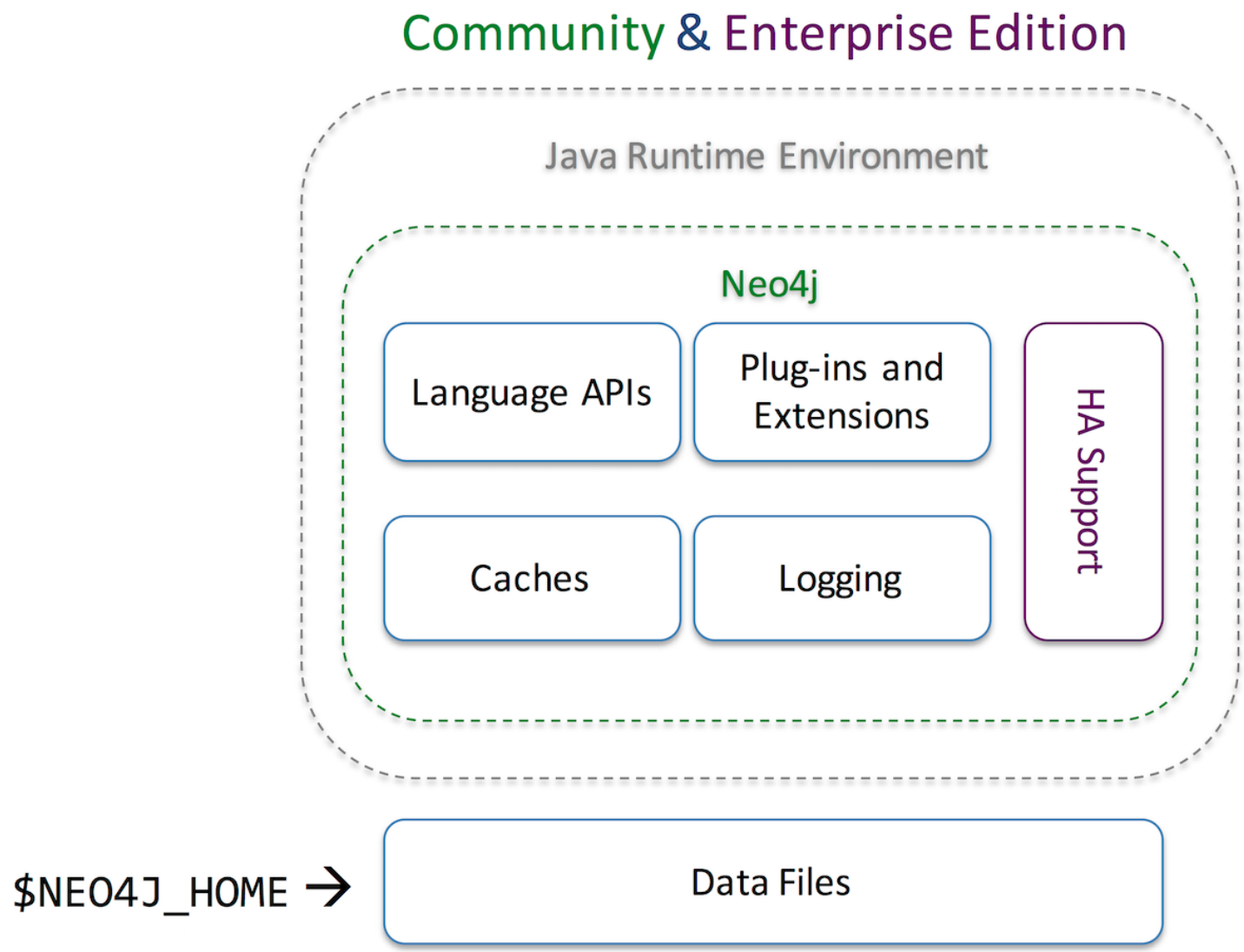

Figure 6. Graph Database Architecture

Neo4j stores its data in a set of standard text files (e.g. nodes, relationships, properties). To support disaster recovery, these files can be periodically backed up, but the process does require downtime. Or, if one has the the enterprise edition, then backups can occur with zero downtime. A solution that encrypts at the O/S block level, like Vormetric or Safenet, can be used to ensure confidentiality of the data at rest.

Neo4j supports several topologies including, standalone (*non-production*), non-clustered instances but with one acting as a cold-standby (*community edition production*), and a highly-available cluster consisting of a master and several slaves (*enterprise edition production*). Regardless of topology, all instances should be monitored, and the logs should be aggregated to a central management platform such as Logstash or Splunk. Neo4j supports secure HTTP (SSL) and access to Neo4j REST API and Web Console can be secured with HTTP Basic; although there are also some open source plugins for LDAP integration.

From an Enterprise Architecture perspective, graph databases are not usually the system of record for entities, which are usually stored in relational or aggregate databases. Graph databases tend to be the system of record for the connections between entities, and in some cases for derived entities. So event sourcing is a common approach for updating graph databases. As transactions occur, events are emitted, and then processed, creating, updating or deleting relationships in the graph database. Of course changes to relationships in the graph database also result in events being emitted back to the enterprise.

Consumers use graph databases to first find information about the connections between entities. Consumers then use the proxy data contained within the graph database results to query a relational or aggregate database to retrieve additional details. For example, if a consumer wanted to create a report on the effectiveness of a medical referral program, the consumer would first look up all specialists referred to by a specific practitioner in a graph database, select a practitioner to specialist relationship, and then issue a subsequent request to an aggregate database to retrieve all the referrals made from that practitioner to the specialist.

Graph Database Summary

Graphs are everywhere from the very large, such as a transportation system, to the very small, yet extremely complicated, such as the human nervous system. Graphs are extremely powerful representations of connected data allowing for the extraction of semantically rich information. This information enables predictions, recommendations, causation analysis, eventing, and many more high value activities.

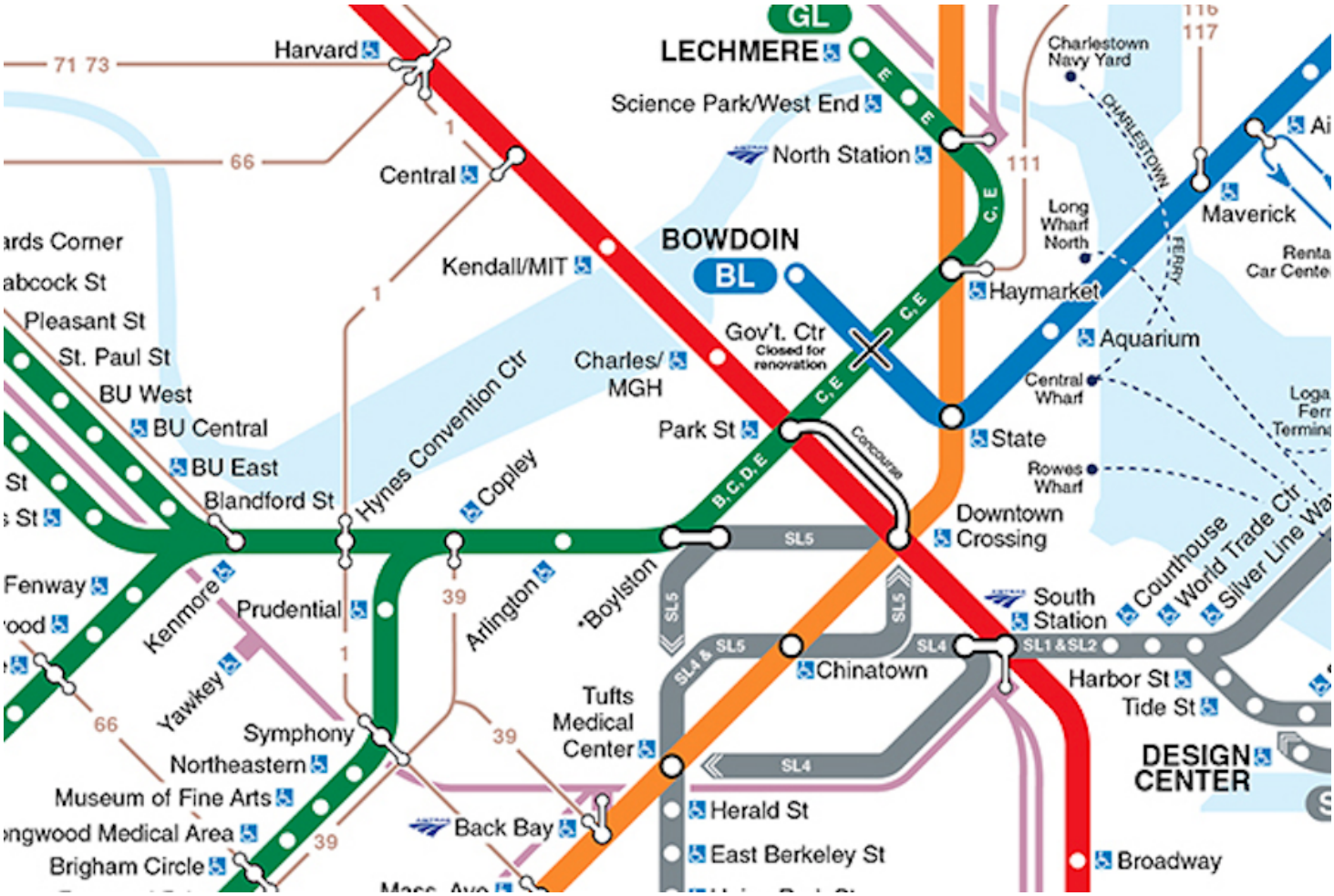

Figure 7. MBTA Graph

The barrier to entry for using graph databases is extremely low, and because graphs do not require a tremendous amount of upfront design, they can be implemented incrementally. Over time graphs are easily refactored to support changing requirements or emergent capabilities. The short ramp-up time is accelerated by the intuitive Cypher language, a wealth of documentation including guides, articles and books, and the fantastic REPL included with the Neo4j Web Console. Finally, it is easy to involve any stakeholder in graph database development because graph modeling is so white board friendly.

About the Author

Jeremy Deane (http://jeremydeane.net) has over 19 years of software engineering experience in leadership positions. His expertise includes Enterprise Application Integration, Web Application Architecture, and Software Process Improvement. In addition, he is an accomplished conference speaker and technical author.

Last updated 2015-12-29 08:25:17 EST