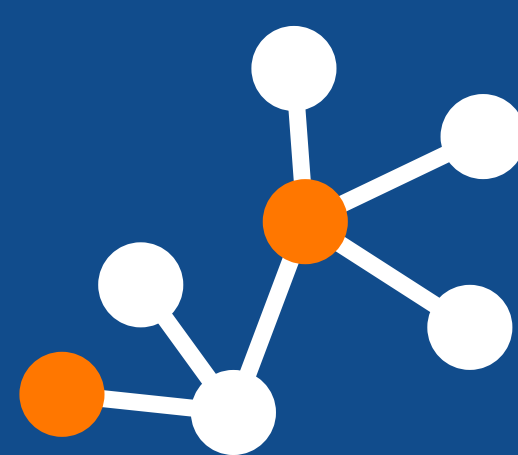


PP30 - Discovering the Power of Graph Data

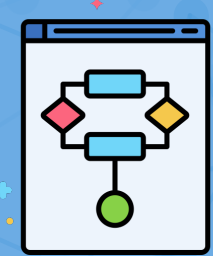


Storing information in a Graph format can provide measurable benefits over conventional data formats (such as Oracle or SAS), made possible by the unique structure of graphs. Looking through the lens of a Graph framework like Neo4j and its query language, Cypher, we can begin to highlight some powerful examples of linked data in action.

Ingesting data or metadata into Graph can be as simple as importing a csv file, a SAS dataset, or even a relational database, using either built-in or open-source libraries such as neointerface and tab2neo. Once imported, one can take advantage of Cypher's pattern matching query structure and parameterization to avoid cumbersome operations and achieve straightforward, yet powerful, filtering.

This poster discusses the potential advantages of Graph data, while still maintaining the functionality of traditional data structures, and aims to overcome the perception that linked data is a challenging concept to grasp.

1. The Graph Perspective A New Point of View



Have you ever wondered if your data holds deeper connections waiting to be unravelled? Graph databases offer possibilities for unique perspectives that go beyond that of traditional relational databases. Unlike the rigid structures of conventional databases, graph databases fully embrace the relationships between data, enabling seamless navigation and querying of complex patterns.

If you've heard of graph databases but have not explored their potential, we hope to offer some insight on the topic. Together we are going to imagine a world where a graphical approach to clinical data has been adopted, explore an implementation and discover some of the exciting possibilities that this world has to offer.

What is a graph?

Graph data is a way of organizing and representing information using **nodes** and **relationships**. Imagine a web of interconnected points where each point represents a specific entity, and the connections between them represent meaningful associations.

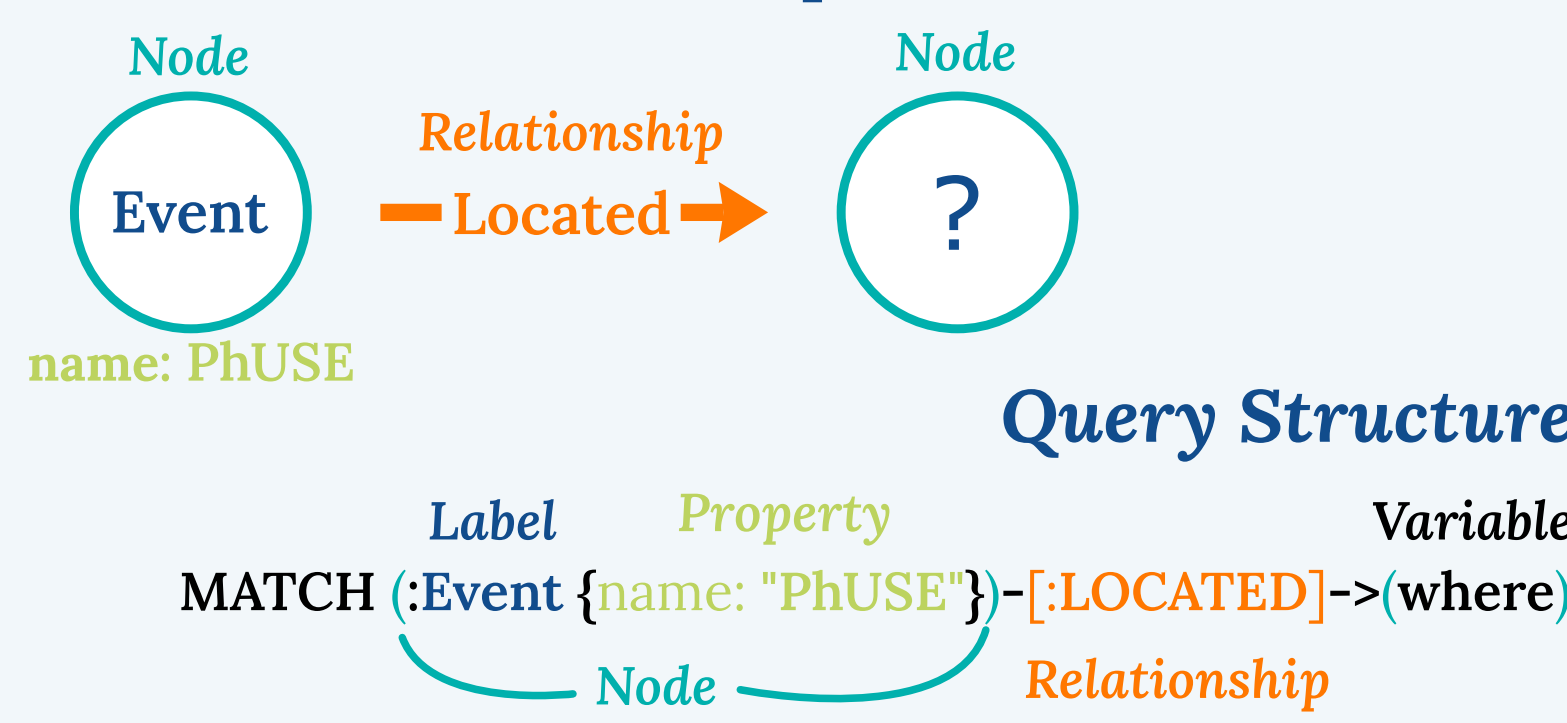
2. Sourcing Data Getting Started



Let's start our graph journey by transforming an example CDISC SDTM v2.0 metadata table, shown below, into a graph data format. For our first step we'll need to reimagine our tabular data into the **Nodes, Labels, Properties** and **Relationships** that make up a graph model. There are several ways to achieve this, including external packages like tab2neo and native CSV imports in Neo4j.

Name	Label	Data Type	Role	Qualifies Variables	Notes	Definition	Variable C-Code
USUBJID	Unique Subject Identifier	Char	Identifier			A sequence of characters used to uniquely identify a subject across all studies for all applications or submissions involving the product.	C69256
AGE	Age	Num	Record Qualifier		May be derived as (RFSTDTC-BRTHDTC), but BRTHDTC may not be available in all cases (due to subject privacy concerns).	A numeric representation of the elapsed time since birth at a specific point in time defined for the trial, used for study data tabulation.	CI70981
AGEU	Age Units	Char	Variable Qualifier	AGE		The unit of time used to express the age, using standardized values.	C50400

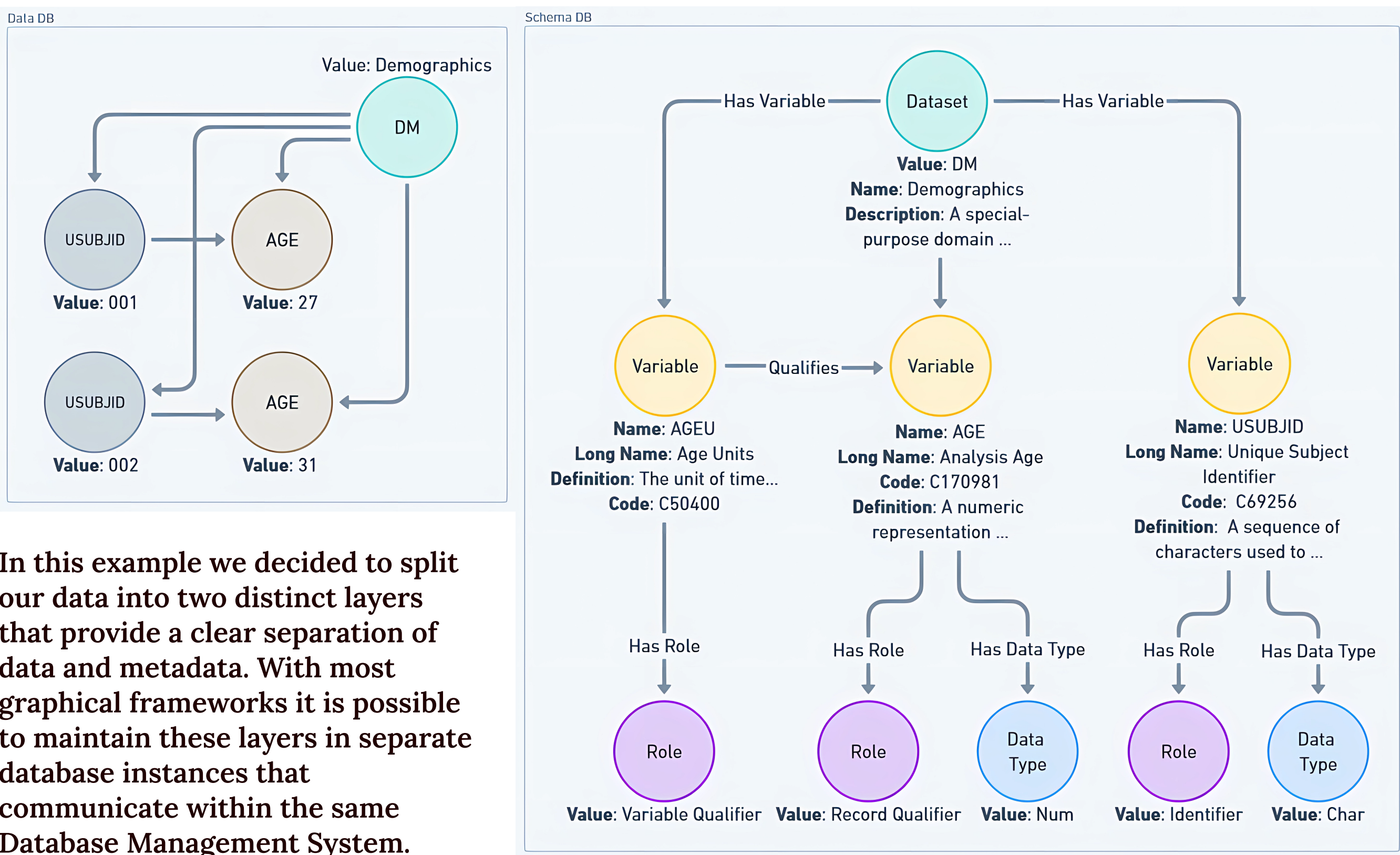
Nodes and Relationships



This reshaping has the immediate benefit of amalgamating our data into a single structure, removing the need for foreign keys and cumbersome table joins.

Similarly, by breaking away from the rigid structure of relational tables into a more flexible graph structure we find a new adaptability to our ever evolving standards. You may even begin to uncover new insights from the reshaping process itself!

A Simple Graph Model



This separation would facilitate the addition of Role-Based Access Control for more granular user permissions while also improving clarity when viewing the data. This separation would facilitate the addition of Role-Based Access Control for more granular user permissions while also improving clarity when viewing the data. In a complete clinical data model an additional layer for controlled terminology could be a logical extension depending on the scale of the data.

The advantages of a graph structure become increasingly evident as you begin to traverse the network and query the data. Moreover, you may already be beginning to observe how the interconnecting nodes (data points) with meaningful relationships facilitate visualisation and comprehension of connections which might otherwise be obscured across multiple tables associated with foreign keys.

3. Data Manipulation Delving Deeper



Now that we have our model, let's start traversing it using Neo4j's declarative query language called Cypher! In Cypher, **Labels** and **Properties** play a vital role. **Labels** like "Variable" or "Dataset" group nodes making it easy to categorise and search data efficiently while **Properties** store key-value information for each **node**.

Lets extend our metadata to also include the Race variable and connect it to our existing **nodes**. Cypher's **MATCH** statement allows us to locate specific patterns already in the database, in this case our "Role", "Dataset" and "Data type" **labels**.

We can then utilise Cypher's **MERGE** statement, which binds existing elements or creates them if they don't already exist.

```
1 MATCH (dataset:Dataset {value: "DM"})
2 MATCH (dtype: `Data type` {value: "Char"})
3 MATCH (role:Role {value: "Record Qualifier"})
4 WITH dataset, dtype, role
5 MERGE (race:Variable {name: "RACE", `Long Name`: "Race",
6   Description: "Race of the subject ..."})
7 MERGE (dataset)-[: `Has Variable` ]->(race)
8 MERGE (race)-[: `Has Data type` ]->(dtype)
9 MERGE (race)-[: `Has Role` ]->(role)
```

Added 1 label, created 1 node, set 3 properties, created 3 relationships, completed after 3 ms.

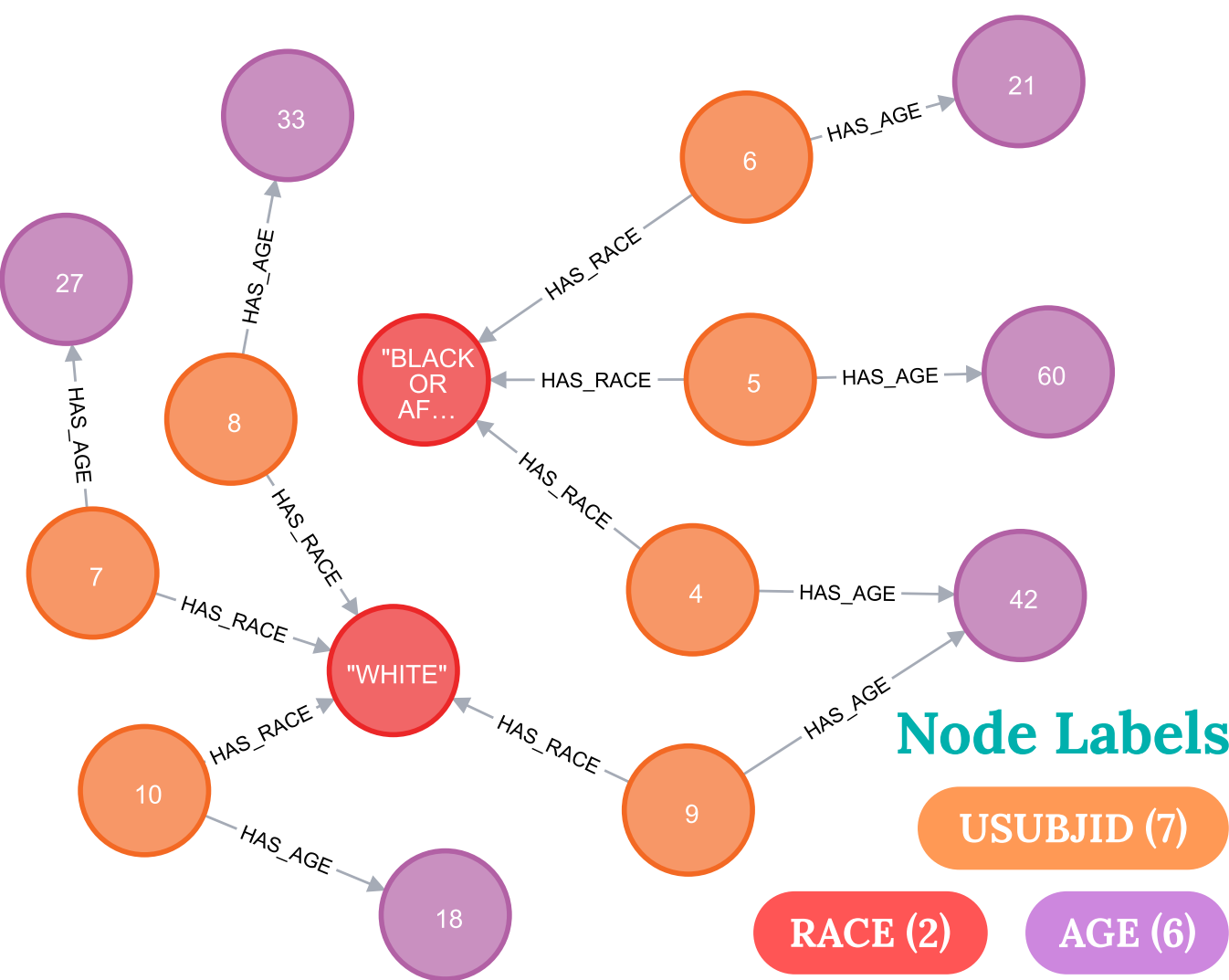
In this example we **MERGE** a new "Variable" **node** called race with several **properties** and then 3 new **relationships** for our newly created **node**. You will likely have noticed that queries are parameterized in Cypher, improving readability and leading to reduced execution times through caching techniques. This opens up additional possibilities for analysts to rapidly explore and interpret our data in real-time, empowering them to quickly grasp the complex interconnections within a large dataset.

Visualisation and Extraction

When egressing data from the graph, we can simply retrieve a bespoke table derived from any part of the pattern we're working with! Here we use a **WHERE** clause to filter our query on the "Char" **property** of our "Data type" **node**. This powerful keyword allows analysts to intuitively refine complex datasets, which when combined with Cypher's simplicity allows them to focus on the data they want to get rather than how to get it.

```
1 MATCH (v:Variable)-[: `Has Data type` ]->(dt: `Data type`)
2 WHERE dt.value = "Char"
3 RETURN v.name as Variable, v.`Long Name` as `Long Name`, v.Code as Code
```

```
1 MATCH (u:USUBJID)
2 MATCH (u)-[: `HAS_RACE` ]->(r:RACE)
3 MATCH (u)-[: `HAS_AGE` ]->(a:AGE)
4 WITH r, collect(u.value) as USUBJIDs, MAX(a.value) as `MAX AGE`, MIN(a.value) as `MIN AGE`, avg(a.value) as `AVG AGE`
5 RETURN r.value as RACE, USUBJIDs, `MAX AGE`, `MIN AGE`, `AVG AGE` ORDER BY r.value
```



The **RETURN** keyword offers us some useful functionality as well. In these examples we **RETURN** multiple **properties**, assign to them aliases and order the final result. Utilising query parameterisation, aggregation functions can also be applied to these **properties** and returned inside our table.

The **WITH** keyword functions similarly to the **RETURN**, but allows the results to be used in further clauses instead of being exported. This allows us to apply more complex transformations and essentially chain queries together with each part feeding into the next.

We can also explore our graph model outside of a tabular format through inbuilt tools like the Neo4j browser or other 3rd party libraries. This makes it easy for analysts to use visualisations to a much greater extent in their work, where narrowing down the context and visualising it quickly can be a powerful tool when working with complex data.

Graph



Relational

Nodes and Edges

Relationships are first-class citizens and can contain properties

Excel at handling complex queries that involve traversing relationship

Highly flexible, allowing for dynamic changes

Model

Relationships

Structure

Performance

Tables with rows and columns

Relies on shared keys to establish relationships between tables through joins

Ridgid structure defined by schema

Well-suited for handling structured data and performing aggregations

4. Interfacing with Graph Connecting the Dots

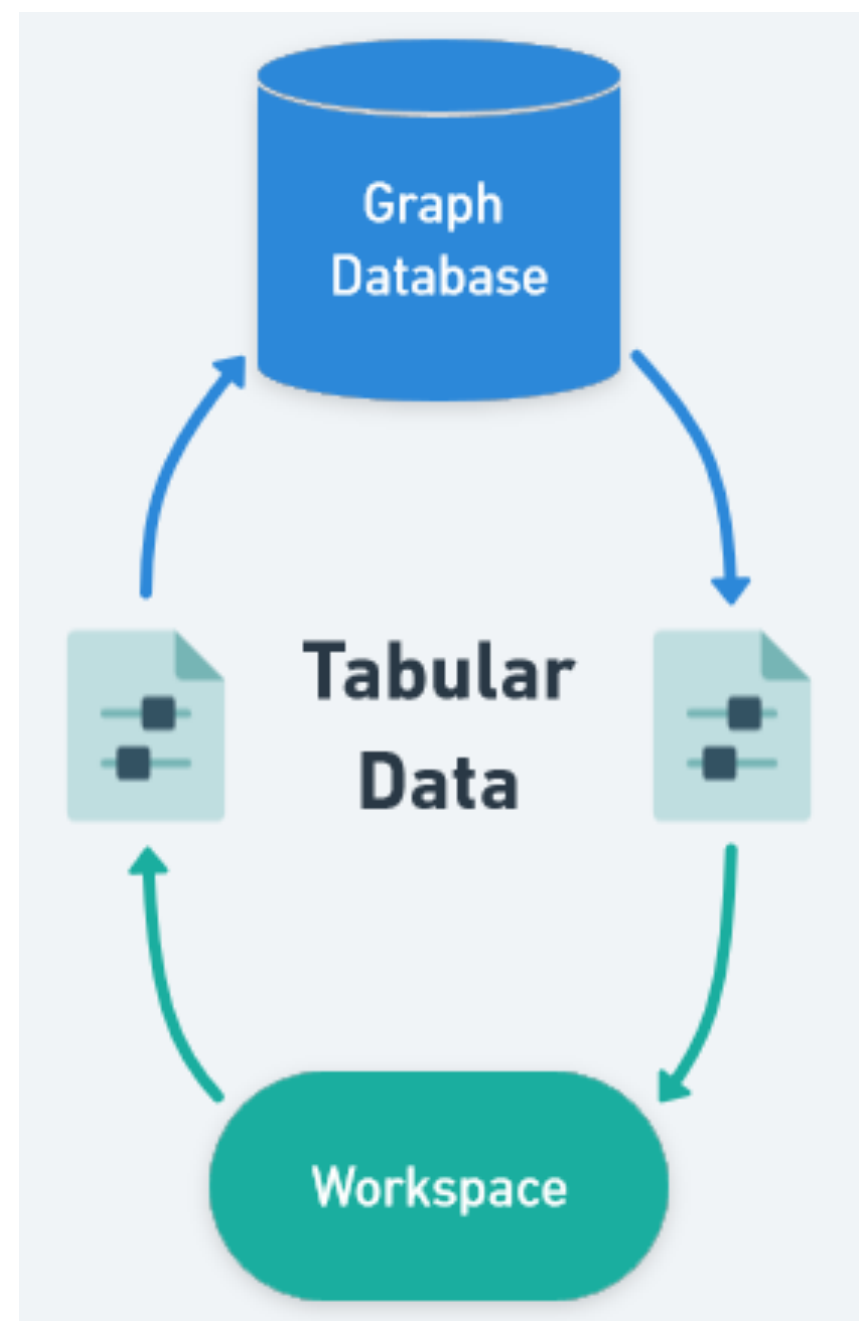


Having explored the possible benefits of using a graph database, let's talk about how it might integrate into existing pipelines. As we've seen, data can be extracted from the graph in a tabular form, allowing analysts to continue manipulating it in their preferred languages. This could facilitate a cyclical process where data flows between graph and tabular as needed.

External Libraries

Most graphical database solutions provide drivers in a variety of languages which facilitate low level interactions with the database. These drivers can be used to develop your own frameworks directly or via 3rd party libraries offering higher level functionality.

For example, the NeoInterface Python library offers general purpose functions that reduces the need for extensive query development. Similarly, the Tab2Neo Python library offers a range of functions for "reshaping" tabular data and loading it back into a graph database. With an ever expanding range of tools and integrations being developed, there are a number of avenues available to explore.



5. Wrapping Up Whats the Catch?



Can you reliably maintain data integrity in a graph?

While the flexibility of graph has its use cases, it does present some unique data integrity challenges. We've primarily worked with Neo4j which does offer a selection of constraints, like uniqueness and or existence, however these are limited compared to their relational database counterparts. It's almost certain that maintaining integrity will require additional effort outside the graph database itself.

What is the biggest challenge faced when switching to graph?

Graph databases are quite different from the relational databases we're used to and there's a learning curve. Familiarising teams with a complex graph data model, a new query language and learning to understand data structured in a graph-oriented way are all significant challenges.

How mature is graph as a technology?

Graph is an established but still evolving technology which means there could be significant updates and integrating those later could be complicated.

Is there room for CDISC's Biomedical Concepts (BCs)?

Yes! CDISC's BCs encompass various biomedical entities, relationships, and their associated attributes, forming a layer of interconnected abstractions. Graph databases, with their inherent ability to model and store relationships as first-class citizens, align perfectly with the nature of BCs. Additionally, the flexibility of graph provides scope for the evolution of BCs moving forward.

It's essential to remember that knowledge graphs are not a silver bullet. However, we believe that leveraging graph databases opens up exciting new avenues for maximising data exploration and understanding particularly in the pharma and clinical research landscape. We've only scratched the surface of what graph has to offer and would recommend exploring the resources below to find out more!

Authors:
Jonathan Deacon
William McDermott

jonathan.deacon@achieveintelligence.com
william.mcdermott@achieveintelligence.com

Biostatistics Neointerface
A Python interface to use the Neo4j driver and simplify its use.

Biostatistics Tab2Neo
Python classes to load & reshape tabular data imported into Neo4j.

neo4j
Case studies from the developers of Neo4j.

Seo-and-web icons created by Flat Icons - Flaticon