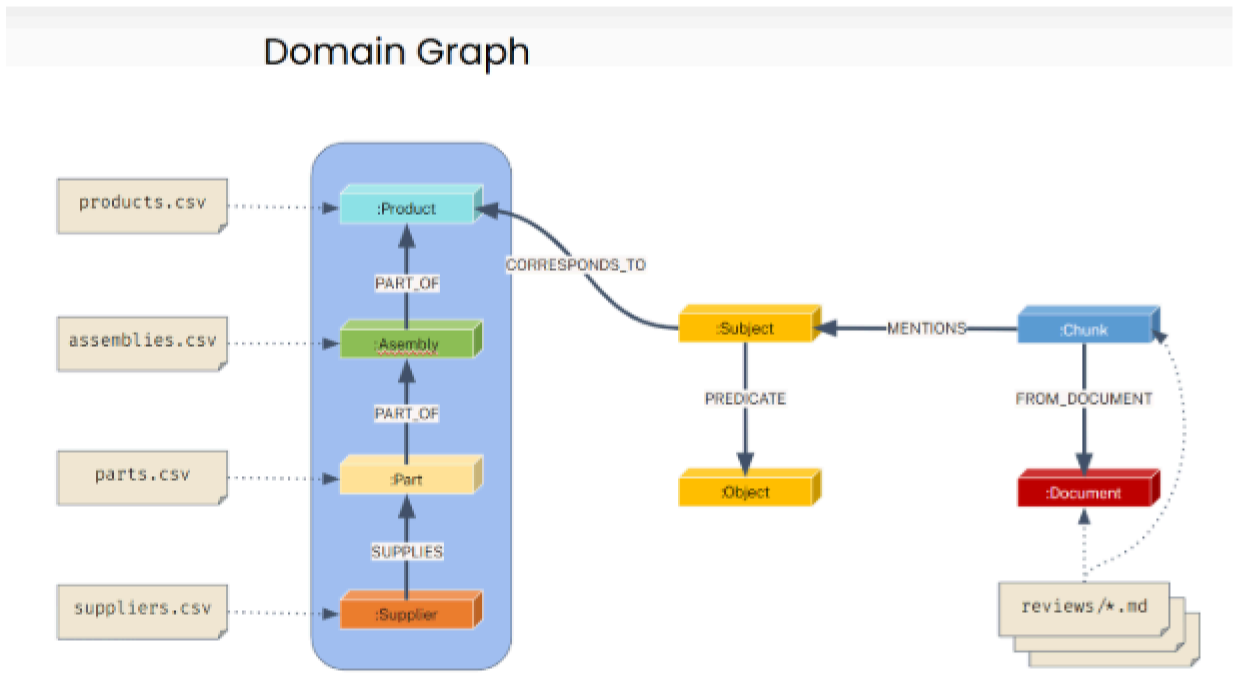

Lesson 8 - Knowledge Graph Construction - Part I

With all the plans in place, it's time to construct the knowledge graph.

For the **domain graph** construction, no agent is required. The construction plan has all the information needed to drive a rule-based import.

Note: This notebook uses Cypher queries to build the domain graph from CSV files. Don't worry if you're unfamiliar with Cypher — focus on understanding the big picture of how the structured data is transformed into a graph structure based on the construction plan.

8.1. Tool

A single tool which will build a knowledge graph using the defined construction rules.

- Input: `approved_construction_plan`
- Output: a domain graph in Neo4j

- Tools: `construct_domain_graph` + helper functions

Workflow

1. The context is initialized with an `approved_construction_plan` and `approved_files`
2. Process all the node construction rules
3. Process all the relationship construction rules

8.2. Setup

The usual import of needed libraries, loading of environment variables, and connection to Neo4j.

```python
# Import necessary libraries

from google.adk.models.lite_llm import LiteLlm # For OpenAI support

# Convenience libraries for working with Neo4j inside of Google ADK
from neo4j_for_adk import graphdb, tool_success, tool_error

from typing import Dict, Any

import warnings
# Ignore all warnings
warnings.filterwarnings("ignore")

import logging
logging.basicConfig(level=logging.CRITICAL)

print("Libraries imported.")
# --- Define Model Constants for easier use ---
MODEL_GPT_4O = "openai/gpt-4o"

llm = LiteLlm(model=MODEL_GPT_4O)

# Test LLM with a direct call
print(llm.llm_client.completion(model=llm.model, messages=[{"role": "user",
"content": "Are you ready?"}], tools=[]))

print("\nOpenAI ready.")
# Check connection to Neo4j by sending a query

neo4j_is_ready = graphdb.send_query("RETURN 'Neo4j is Ready!' as message")

print(neo4j_is_ready)
```

8.3. Tool Definitions (Domain Graph Construction)

The `construct_domain_graph` tool is responsible for constructing the "domain graph" from CSV files, according to the approved construction plan.

Function: create_uniqueness_constraint

This function creates a uniqueness constraint in Neo4j to prevent duplicate nodes with the same label and property value from being created.

```python
def create_uniqueness_constraint(
   label: str,
   unique_property_key: str,
) -> Dict[str, Any]:
   """Creates a uniqueness constraint for a node label and property key.
   A uniqueness constraint ensures that no two nodes with the same label and
property key have the same value.
   This improves the performance and integrity of data import and later queries.

   Args:
       label: The label of the node to create a constraint for.
       unique_property_key: The property key that should have a unique value.

   Returns:
       A dictionary with a status key ('success' or 'error').
       On error, includes an 'error_message' key.
   """
   # Use string formatting since Neo4j doesn't support parameterization of
labels and property keys when creating a constraint
   constraint_name = f"{label}_{unique_property_key}_constraint"
   query = f"""CREATE CONSTRAINT `{constraint_name}` IF NOT EXISTS
   FOR (n:`{label}`)
   REQUIRE n.`{unique_property_key}` IS UNIQUE"""
   results = graphdb.send_query(query)
   return results
```

Function: load_nodes_from_csv

This function performs batch loading of nodes from a CSV file into Neo4j. It uses the `LOAD CSV` command with the `MERGE` operation to create nodes while avoiding duplicates based on the unique column. The Cypher query processes data in batches of 1000 rows for better performance.

Note: The csv files are stored in the `/import` directory of `neo4j` database. When you use the query `LOAD CSV from "file:///" + $source_file`, neo4j checks the `/import` directory by default.

```python
def load_nodes_from_csv(
   source_file: str,
   label: str,
   unique_column_name: str,
   properties: list[str],
) -> Dict[str, Any]:
   """Batch loading of nodes from a CSV file"""

   # load nodes from CSV file by merging on the unique_column_name value
   query = f"""LOAD CSV WITH HEADERS FROM "file:///" + $source_file AS row
   CALL (row) {{
       MERGE (n:$($label) {{ {unique_column_name} : row[$unique_column_name] }})
       FOREACH (k IN $properties | SET n[k] = row[k])
   }} IN TRANSACTIONS OF 1000 ROWS
   """

   results = graphdb.send_query(query, {
       "source_file": source_file,
       "label": label,
       "unique_column_name": unique_column_name,
       "properties": properties
   })
   return results
```

Execute Domain Graph Construction

This cell executes the main construction function using the approved construction plan. It builds the complete knowledge graph by importing all nodes and relationships according to the defined rules.

Function: import_nodes

This function orchestrates the node import process by first creating a uniqueness constraint and then loading nodes from the CSV file. It ensures data integrity by establishing constraints before importing data.

```
def import_nodes(node_construction: dict) -> dict:
    """Import nodes as defined by a node construction rule."""

    # create a uniqueness constraint for the unique_column
    uniqueness_result = create_uniqueness_constraint(
        node_construction["label"],
        node_construction["unique_column_name"]
    )

    if (uniqueness_result["status"] == "error"):
        return uniqueness_result
    # import nodes from csv
    load_nodes_result = load_nodes_from_csv(
        node_construction["source_file"],
        node_construction["label"],
        node_construction["unique_column_name"],
        node_construction["properties"]
    )

    return load_nodes_result
```

Function: import_relationships

This function imports relationships between nodes from a CSV file. It uses a Cypher query that matches existing nodes and creates relationships between them. The query finds pairs of nodes and creates relationships with specified properties between them.

```
def import_relationships(relationship_construction: dict) -> Dict[str, Any]:
    """Import relationships as defined by a relationship construction rule."""

    # load nodes from CSV file by merging on the unique_column_name value
    from_node_column = relationship_construction["from_node_column"]
    to_node_column = relationship_construction["to_node_column"]
    query = f"""LOAD CSV WITH HEADERS FROM "file:///" + $source_file AS row
    CALL (row) {{
        MATCH (from_node:$($from_node_label) {{ {from_node_column} :
row[$from_node_column] }}),
              (to_node:$($to_node_label) {{ {to_node_column} :
row[$to_node_column] }} )
        MERGE (from_node)-[r:$($relationship_type)]->(to_node)
        FOREACH (k IN $properties | SET r[k] = row[k])
    }} IN TRANSACTIONS OF 1000 ROWS
    """

    results = graphdb.send_query(query, {
        "source_file": relationship_construction["source_file"],
        "from_node_label": relationship_construction["from_node_label"],
        "from_node_column": relationship_construction["from_node_column"],
        "to_node_label": relationship_construction["to_node_label"],
        "to_node_column": relationship_construction["to_node_column"],
        "relationship_type": relationship_construction["relationship_type"],
        "properties": relationship_construction["properties"]
    })
    return results
```

Function: construct_domain_graph

This is the main orchestration function that builds the entire domain graph. It processes the construction plan in two phases:

1. **Node Construction**: First imports all nodes to ensure they exist before creating relationships
2. **Relationship Construction**: Then creates relationships between the existing nodes

This two-phase approach prevents relationship creation failures due to missing nodes.

```
def construct_domain_graph(construction_plan: dict) -> Dict[str, Any]:
   """Construct a domain graph according to a construction plan."""
   # first, import nodes
   node_constructions = [value for value in construction_plan.values() if
value['construction_type'] == 'node']
   for node_construction in node_constructions:
       import_nodes(node_construction)

   # second, import relationships
   relationship_constructions = [value for value in construction_plan.values()
if value['construction_type'] == 'relationship']
   for relationship_construction in relationship_constructions:
       import_relationships(relationship_construction)
```

8.4. Run construct_domain_graph()

This cell defines the approved construction plan as a dictionary containing rules for creating nodes and relationships. The plan includes:

- **Node Rules**: Define how to create Assembly, Part, Product, and Supplier nodes from CSV files
- **Relationship Rules**: Define how to create Contains, Is_Part_Of, and Supplied_By relationships

Each rule specifies the source file, labels, unique identifiers, and properties to be imported.

```
# the approved construction plan should look something like this...
approved_construction_plan = {
   "Assembly": {
       "construction_type": "node",
       "source_file": "assemblies.csv",
       "label": "Assembly",
       "unique_column_name": "assembly_id",
       "properties": ["assembly_name", "quantity", "product_id"]
   },
   "Part": {
       "construction_type": "node",
       "source_file": "parts.csv",
       "label": "Part",
       "unique_column_name": "part_id",
       "properties": ["part_name", "quantity", "assembly_id"]
   },
   "Product": {
       "construction_type": "node",
       "source_file": "products.csv",
       "label": "Product",
       "unique_column_name": "product_id",
       "properties": ["product_name", "price", "description"]
   },
   "Supplier": {
       "construction_type": "node",
       "source_file": "suppliers.csv",
       "label": "Supplier",
       "unique_column_name": "supplier_id",
       "properties": ["name", "specialty", "city", "country", "website",
"contact_email"]
   },
   "Contains": {
       "construction_type": "relationship",
       "source_file": "assemblies.csv",
       "relationship_type": "Contains",
       "from_node_label": "Product",
       "from_node_column": "product_id",
       "to_node_label": "Assembly",
       "to_node_column": "assembly_id",
       "properties": ["quantity"]
   },
   "Is_Part_Of": {
       "construction_type": "relationship",
       "source_file": "parts.csv",
       "relationship_type": "Is_Part_Of",
       "from_node_label": "Part",
       "from_node_column": "part_id",
       "to_node_label": "Assembly",
       "to_node_column": "assembly_id",
       "properties": ["quantity"]
   },
   "Supplied_By": {
       "construction_type": "relationship",
       "source_file": "part_supplier_mapping.csv",
       "relationship_type": "Supplied_By",
       "from_node_label": "Part",
       "from_node_column": "part_id",
       "to_node_label": "Supplier",
       "to_node_column": "supplier_id",
       "properties": ["supplier_name", "lead_time_days", "unit_cost",
"minimum_order_quantity", "preferred_supplier"]
```

```
    }
}

construct_domain_graph(approved_construction_plan)
```

8.5 Inspect the Domain Graph

This cell filters the construction plan to extract only the relationship construction rules. This list will be used in the next cell to verify that all relationships were successfully created in the graph.

```
# extract a list of the relationship construction rules
relationship_constructions = [
    value for value in approved_construction_plan.values()
    if value.get("construction_type") == "relationship"
]
relationship_constructions
```

This cell creates and executes a Cypher query to verify that all relationship types from the construction plan were successfully created in the graph.

The query uses several advanced Cypher features:

- `UNWIND`: Iterates through each relationship construction rule
- `CALL (construction) { ... }`: Subquery that executes for each construction rule
- `MATCH (from)-[r:relationship_type]->(to)`: Finds one example of each relationship type
- `LIMIT 1`: Returns only one example per relationship type

This provides a summary view showing one instance of each relationship pattern in the constructed graph.

```
# a fancy cypher query which to show one instance of each construction rule

# turn the list of rules into multiple single rules
unwind_list = "UNWIND $relationship_constructions AS construction"

# match a single path for a given construction.relationship_type
# return only the labels and types from the 3 parts of the path
match_one_path = """
    MATCH (from)-[r:$(construction.relationship_type)]->(to)
    RETURN labels(from) AS fromNode, type(r) AS relationship, labels(to) AS
toNode
    LIMIT 1
"""
match_in_subquery = f"""
CALL (construction) {{
{match_one_path}
}}
"""
```

```
cypher = f"""
{unwind_list}
{match_in_subquery}
RETURN fromNode, relationship, toNode
"""

print(cypher)

print("\n---")

graphdb.send_query(cypher, {
   "relationship_constructions": relationship_constructions
})
```

graph. Join me in this final lesson where we go to find the tools that will execute the construction plan as specified by the workflow definitions. With a graph construction plan and a knowledge extraction plan in place, you are now ready to build a graph. In this lesson, you will dive into the details of the knowledge graph construction tool. If you're feeling generous, you can call this a neuro symbolic agent, a hybrid of language models with rule based systems. Let's take a step back for a moment to look at what these plans will produce. The graph construction plan will load CSV files to produce the domain graph. There is almost a one to one mapping from CSV file to node type, with some extra mappings to create relationships. You could probably ask an LM with a giant context window to perform this task, but this is a very direct mechanical process. You have converged to the code. The work will be performed by a single tool you'll define called the construct domain graph. That tool have a bunch of helper functions that it uses for all the different parts of constructing the graph. So we'll go through each of those tools one step at a time. As always, we'll import some libraries, check that open AI is running, and also make sure neo for J is still there. Perfect. Now, let's start defining the tools themselves. For domain graph construction, we're create a series of tools, each one with a very specific responsibility. You'll start with creating a unique constraint on the database. Now we're moving from thinking about the data files to thinking about preparing the database for the data import that's about to happen. And the first part about preparing the database is to make sure that we have uniqueness constraints. Now the database that ends up meaning that for any nodes that has a particular label, we can say that whenever that label occurs and there's a property on that label, that we want that property to be unique. That correlates exactly what the CSV files that have a unique column ID. That's going to be a unique constraint for each one of those. So this function is going to take care of that for us. We're just going to pass in the label and what the unique property key should be and it'll take care of actually creating this constraint on me for J. Now the query down here, you can see create constraint, the constraint gets passed in, only create it if it doesn't already exist. And it gets created for a particular label. And on that label, we require that this unique property key will be unique. Now, before we were using a lot of query parameters, when we're setting up constraints like this inside new for J, it's not possible to pass in a query parameter for that. So we're going to be doing a little bit of unsafe work here by actually doing string coordination and we know that that's not really a recommended practice. We should have a sanitized function here to to make sure there's nothing happening that's, you know, bad. But for now, this will be just fine. So the next function you want, if you have the ability to actually create uniqueness constraints for nodes, we then want to be able to load nodes from a CSV file. You can define a function for that. So here's a load node from CSV function. It takes in a source file, the label for that source file, which of the columns in that CSV file is unique. and also a list of all the properties that should be created from the source file. Now looking down here at the Cypher that's created, there's a special Cypher syntax for loading from CSV files that are in the import directory. So that's why everything that we're working with are all relative paths. Here, when we're using load CSV and new for J, this file URL that's here is going to end up being relative to new for J's import directory. So we're going to load a CSV file with headers from that directory and then for every row in that file, we're going to call a sub query here. And in the sub query, it's actually going to go ahead and do the merging of those rows. Merge of course is going to do a creation and for seeing if that node already exists. If it doesn't, it will go ahead and create that node based on the unique column that we're passing in. And then this next line here, the for each is a little bit of a unique way of taking advantage of the property list that was passed in. It's really just the names of some properties. And what we end up doing is looping through each of those properties. And for each of those property names, we're going to set the property value on the node that was just created to the row value that was passed in from the CSV file. So in effect, we're going to loop through all the names and go ahead and just iteratively set those values here. Now, by using this inside of a sub query, we then get the option of doing this in batch. So we're going to do this in transaction of a thousand rows. So no matter how big the CSV file is, we're going to do it as a batch of just 1,000 at a time. You can then see where we're actually going to call to new for day, sending in the query. We're going to pass in query and along with a bunch of query parameters that we're going to be used. Now, for each file that you have, you actually want to do both of the functions that we just defined. We want to call both of them. First, call the create uniqueness constraint to make sure that we'll have unique values for each node so the IDs are respected. Then after we've created the uniqueness constraints for each of the files, we're then going to go ahead and actually do the import using the the load nodes from CSP tool that we just saw. So that's what's happening here inside of import nodes. It's just taking the construction rule and then first based on the values in the construction rule, it's going to go ahead and call the create uniqueness constraint tool right here. And then if it doesn't have an error, it will go on to do the import from the nodes from the CSV file. And all it's doing here is just taking from the construction rule, each of the different values that need to be passed into the function. You can do something similar with importing relationships. Now, in relationships, we don't need to apply a uniqueness constraint on those because they don't have any identities themselves. They're going to be connected directly to nodes on either side as the from or the two node, and those should be unique. So the relationship themselves are going to be unique because there's just one of each for every row in the files that we're importing. This is a little particular to the data files that we have here. We want to be a little bit more careful if we had, you know, data files that were set up in a more complicated way. But for the data files that we have in this example, this is an adequate way to actually do the import. As with the node loading, we're going to use load CSV with headers from the import directory and load all those as rows. And for each of those rows, we're going to actually do a little bit more work inside of the sub query. First, we're going to find the existing nodes that were previously loaded. So we're going to find the from node and we're going to find the two node. And after we've found those using the match clause, we'll use the merge clause here to actually create the relationship from the front node to the two node. So this is only going to allow us to create one relationship for any given pair of the front and the two of a particular type of relationship. So if there were two relationships that were, say there are two ABK likes coffee rows, that would only happen once because even if there were two that occurred in the file and merging it here, the merge will look at the nodes, those are already unique on either side, and the relationship already has an ABK likes. So the likes is already there, it's not going to create another relationship. And the same thing with the nodes, we're going to do the property setting on the relationships if there were any properties available, we'll loop through them using this for each construct here, and that will set all the values that are available. And then just go ahead and call the query passing in all the query parameters directly from the construction itself. So all of those functions do most of the real work, and then construct the main graph itself really doesn't have to do a whole lot other than to take in as an argument the full construction plan and then do the right ordering of construct the nodes first, and then now with the nodes already existing, then you can go ahead and create the relationships. With construct main graph already defined, we can go ahead and give it a try. Now, to run this, of course, we're going to need a complete construction plan. So here's the whole construction plan. This will be different from previous lessons depending on what the LM decided to do. When this notebook was created, here's a construction plan that was valid. And you can take a look through that, it's kind of expected. I think the thing to pay most attention to here, that we'll check later, is the relationships that are created. There is a contains relationship that goes from a product contains an assembly, that sounds right. And this is part of goes from a part. A part is a part of an assembly. Okay, that's a little wordy, but that's correct. And the front node here is a part that is supplied by a supplier. Also seems perfectly fine. So that will be the input to our construct domain graph function. So let's give it a run. Now, when that function is done, you won't see any output. If we'd added a print statement or something to let you know that, okay, it's finished doing its work. Instead, what we're going to do here is actually interact with the graph as if we were still doing some querying. And so let's inspect the domain graph with a bit of a fancy cipher statement. What we're going to do is first find all the just relationship construction rules from the construction plan. And we're going to do that with just a list comprehension expression inside of Python. So we're just going to loop through all the construction plan values and whenever the construction has a relationship, it gets put into this list. So here's all the relationship constructions that we saw before, but without the node constructions. And then we're going to do a little bit of cipher magic here. And so I'll walk through this one step at a time. And in this fancy cipher query we're about to run, the goal is basically to do a little sampling with the graph. So we're not going to get the entire graph, just take a look at it. What we want to do is we're expecting that for every relationship construction that was up here, there should at least be one occurrence of that relationship type in the resulting graph. So if we just get a collection of all the different relationship types and just do pattern matching on those, and then see the output of that, we should see that all the things we expected to be created were actually created. And I'm not sure if I said that correctly, but let's actually walk through it and see how this plans out. So here's the first step. We're going to pass in this list of construction rules, and in cipher, you have the ability to take a list of things and unwind it into a collection of rows. So what was a list of rules as a single item would be turned into multiple rows where each element in each row will have a single rule. So unwind basically does that. So the relationship construction list will be turned into a series of rows where each row element will just have a single construction value in it. So then with those, what you want to end up doing is for the construction rule, we want to do a pattern match from some node, and you don't have any qualification for it here. We're going to match from a node through a relationship that we're going to call R. And the relationship type is what we're actually going to use from the construction rule. We're going to pull out the construction rule's relationship type value. And here, of course, the dollar sign we're referring to the query parameter. So we're going to go ahead and pull that out into this relationship type here. And so if we had up here, we see that we've had a supplied by relationship type. That will end up getting substituted in here by the query parameter. So we'll have a from some node that is supplied by some other node. And what we'll do is return just the labels of the from node and the labels of the two node and then the type of the relationship itself. What we should end up with is a little triple of just the relationship labels and the node labels on either side. And we're going to limit to just one because we don't want to see the whole graph. We just want to verify that it exists. We're going to put that inside of a sub query so that we can apply the match there just once to every element that we've unwound earlier. So that's just happening here in this part. We're going to call a sub query and this match one path that we've just defined is going to be put inside there. And then in the fully assembled cipher, we're going to unwind the list, do the matching inside of the sub query, and then from all of that, return just the triples of the from node, relationship and two node, which would be the labels, their types, and the labels on the two node as well. Okay, and you can see that this is going to go ahead and print out the value. So you can see what the cipher looks like before we run it, and then we'll finally run the cipher. So here's the complete cipher statement. And here are the relationships that we have found. We've got from product contains assembly and a part is part of an assembly and a part is supplied by a supplier. Perfect. That matches what we saw earlier in the construction rules. So it looks like this graph has been constructed in a nice way.