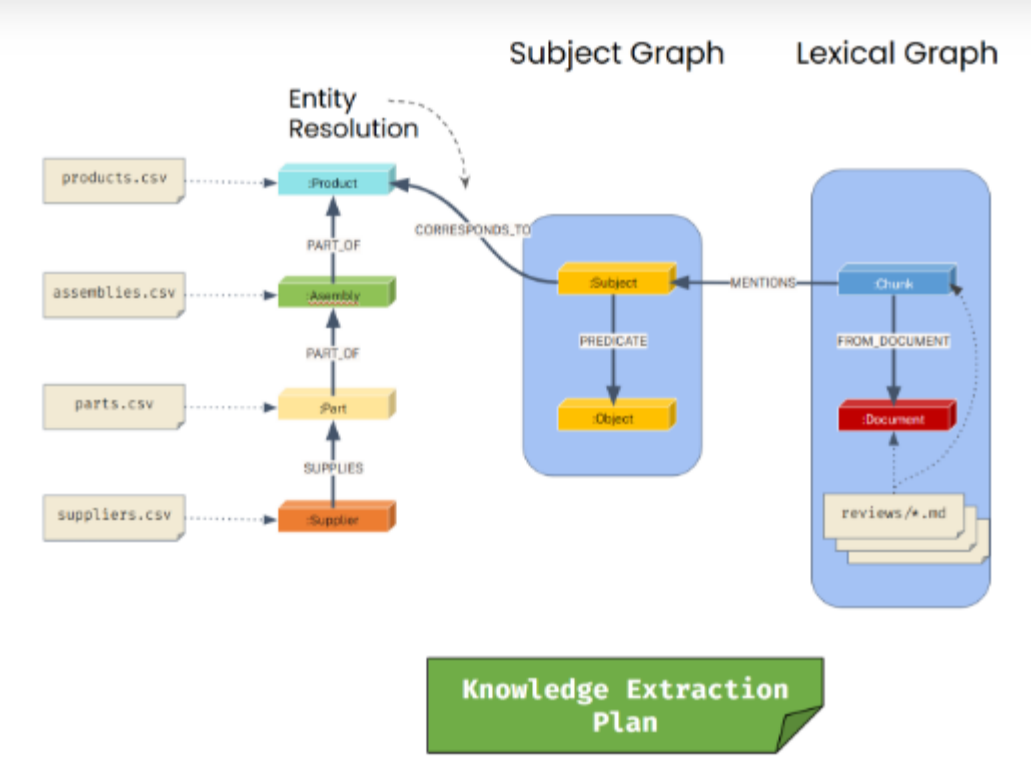

Lesson 8 - Knowledge Graph Construction - Part II

In this lesson, you'll continue with knowledge graph construction. The previous lesson created the domain graph from CSV files according to the construction plan. Now, you will process the markdown files, chunking them up into the lexical graph and the subject graph which will connect to the domain graph for a complete knowledge graph.

You will learn:

- how to use Neo4j's graphrag library to perform the chunking and entity extraction
- techniques for entity resolution

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 unstructured data is transformed into a graph structure based on the extraction plan.

💻 **To access the helper.py, neo4j_for_adk.py and tools.py files :** 1) click on the *"File"* option on the top menu of the notebook and then 2) click on *"Open"*.

8.1 Tools
Two tools, with helper functions:

1. `make_kg_builder` - to chunk markdown and produce the lexical and subject graphs
2. `correlate_subject_and_domain_nodes` - to connect the subject graph to the domain graph
    - Input: `approved_files`, `approved_construction_plan`, `approved_entities`, `approved_fact_types`
    - Output: a completed knowledge graph with domain, lexical and subject graphs

Workflow

1. The context is initialized with an `approved_construction_plan` and `approved_files`
2. For each markdown file, `make_kg_builder` is called to create a construction pipeline
3. For each resulting entity label, `correlate_subject_and_domain_nodes` will connect the subject and domain graphs

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

8.2.1 Common Setup
```
# Import necessary libraries
import os
import re
from pathlib import Path

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

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)
```

8.2.2 Load Part of the Domain Graph with a Helper Function

You're only loading the product nodes from the domain graph, because they're the only nodes that you'll use to connect the domain graph to the lexical graph.

```
from tools import load_product_nodes

load_product_nodes()

# expect to find non-entity nodes with a "Product" label
graphdb.send_query("MATCH (n) WHERE NOT n:`__Entity__` return DISTINCT
labels(n) as nonEntityLabels")
```

8.2.3 Initialize State from Previous Workflow

```
# 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"]
    }
}
```

```
approved_files = [
    "product_reviews/gothenburg_table_reviews.md",
    "product_reviews/helsingborg_dresser_reviews.md",
    "product_reviews/jonkoping_coffee_table_reviews.md",
    "product_reviews/linkoping_bed_reviews.md",
    "product_reviews/malmo_desk_reviews.md",
    "product_reviews/norrkoping_nightstand_reviews.md",
    "product_reviews/orebro_lamp_reviews.md",
    "product_reviews/stockholm_chair_reviews.md",
    "product_reviews/uppsala_sofa_reviews.md",
    "product_reviews/vasteras_bookshelf_reviews.md"
]

# approved entities from the `ner_agent` of Lesson 7
approved_entities = ['Product', 'Issue', 'Feature', 'Location']

# approved fact types from the `relevant_fact_agent` of Lesson 7
approved_fact_types = {'has_issue': {'subject_label': 'Product',
'predicate_label': 'has_issue', 'object_label': 'Issue'},
'includes_feature': {'subject_label': 'Product', 'predicate_label':
'includes_feature', 'object_label': 'Feature'}, 'used_in_location':
{'subject_label': 'Product', 'predicate_label': 'used_in_location',
'object_label': 'Location'}}
```

8.3 Tool Definitions for loading, chunking and entity extraction

The Neo4j GraphRAG library has a convenient `SimpleKGPipeline` which you can use to process chunks and extract entities with relationships.

For the markdown files you will be processing, you'll need to create some helper functions.

8.3.1 SimpleKGPipeline Interface

```
from neo4j_graphrag.experimental.pipeline.kg_builder import
SimpleKGPipeline
# for example, creating a KG pipeline requires these arguments
if False:
    example = SimpleKGPipeline(
        llm=None, # the LLM to use for Entity and Relation extraction
        driver=None,  # a neo4j driver to write results to graph
        embedder=None,  # an Embedder for chunks
```

```
        from_pdf=True,    # sortof True because you will use a custom loader
        pdf_loader=None, # the custom loader for Markdown
        text_splitter=None, # the splitter you defined above
        schema=None, # that you just defined above
        prompt_template=None, # the template used for entity extraction on
each chunk
    )
```

8.3.2 Text-Splitter for Chunking up the Markdown

Define a custom text splitter that uses regex patterns to chunk markdown text. This splitter breaks documents at specified delimiters (like "---") to create meaningful text segments for processing.

```
from neo4j_graphrag.experimental.components.text_splitters.base import
TextSplitter
from neo4j_graphrag.experimental.components.types import TextChunk,
TextChunks

# Define a custom text splitter. Chunking strategy could be
yet-another-agent
class RegexTextSplitter(TextSplitter):
    """Split text using regex matched delimiters."""
    def __init__(self, re: str):
        self.re = re

    async def run(self, text: str) -> TextChunks:
        """Splits a piece of text into chunks.

        Args:
            text (str): The text to be split.

        Returns:
            TextChunks: A list of chunks.
        """
        texts = re.split(self.re, text)
        i = 0
        chunks = [TextChunk(text=str(text), index=i) for (i, text) in
enumerate(texts)]
        return TextChunks(chunks=chunks)
```

8.3.3 Custom Markdown Data Loader

This custom loader adapts the Neo4j GraphRAG PDF loader to work with markdown files. It reads markdown content, extracts the document title from the first H1 header, and wraps it in the expected document format for the pipeline.

```
# custom file data loader

from neo4j_graphrag.experimental.components.pdf_loader import DataLoader
from neo4j_graphrag.experimental.components.types import PdfDocument,
DocumentInfo

class MarkdownDataLoader(DataLoader):
   def extract_title(self,markdown_text):
       # Define a regex pattern to match the first h1 header
       pattern = r'^# (.+)$'

       # Search for the first match in the markdown text
       match = re.search(pattern, markdown_text, re.MULTILINE)

       # Return the matched group if found
       return match.group(1) if match else "Untitled"

   async def run(self, filepath: Path, metadata = {}) -> PdfDocument:
       with open(filepath, "r") as f:
           markdown_text = f.read()
       doc_headline = self.extract_title(markdown_text)
       markdown_info = DocumentInfo(
           path=str(filepath),
           metadata={
               "title": doc_headline,
           }
       )
       return PdfDocument(text=markdown_text, document_info=markdown_info)
```

8.3.4 Set up LLM, Embedder and Neo4j Driver

Initialize the core components needed for the Neo4j GraphRAG pipeline: an OpenAI LLM for entity extraction, an embeddings model for vectorizing text chunks, and the Neo4j database driver for graph storage.

```
from neo4j_graphrag.llm import OpenAILLM
from neo4j_graphrag.embeddings import OpenAIEmbeddings

# create an OpenAI client for use by Neo4j GraphRAG
llm_for_neo4j = OpenAILLM(model_name="gpt-4o",
model_params={"temperature": 0})

# use OpenAI for creating embeddings
embedder = OpenAIEmbeddings(model="text-embedding-3-large")

# use the same driver set up by neo4j_for_adk.py
neo4j_driver = graphdb.get_driver()
```

8.3.5 Entity Schema

Use the approved entity types from the previous workflow as the allowed node types for entity extraction. This constrains the LLM to only extract entities of these specific types.

```
# approved entities list can be used directly
schema_node_types = approved_entities

print("schema_node_types: ", schema_node_types)
```

Transform the approved fact types into relationship types by extracting the predicate labels and converting them to uppercase format for the schema.

```
# the keys from approved fact types dictionary can be used for
relationship types
schema_relationship_types = [key.upper() for key in
approved_fact_types.keys()]

print("schema_relationship_types: ", schema_relationship_types)
```

Create relationship patterns by converting fact types into tuples that specify allowed relationships between specific node types (subject-predicate-object patterns).

```
# rewrite the fact types into a list of tuples
schema_patterns = [
   [ fact['subject_label'], fact['predicate_label'].upper(),
fact['object_label'] ]
   for fact in approved_fact_types.values()
]

print("schema_patterns:", schema_patterns)
```

Assemble the complete entity schema dictionary that will guide the LLM's entity extraction, combining node types, relationship types, and patterns into a single configuration.

```
# the complete entity schema
entity_schema = {
   "node_types": schema_node_types,
   "relationship_types": schema_relationship_types,
   "patterns": schema_patterns,
   "additional_node_types": False, # True would be less strict, allowing
unknown node types
}
```

8.3.6 Contexualized Entity Extraction Prompt

This helper function extracts the first few lines from a file to provide context for entity extraction. This context helps the LLM better understand the document structure and content when processing individual chunks.

```
def file_context(file_path:str, num_lines=5) -> str:
   """Helper function to extract the first few lines of a file

   Args:
       file_path (str): Path to the file
```

```
        num_lines (int, optional): Number of lines to extract. Defaults to
5.

    Returns:
        str: First few lines of the file
    """
    with open(file_path, 'r') as f:
        lines = []
        for _ in range(num_lines):
            line = f.readline()
            if not line:
                break
            lines.append(line)
    return "\n".join(lines)
```

This function creates a contextualized prompt template for entity and relationship extraction. It combines general extraction instructions with file-specific context to improve the accuracy of the LLM's entity recognition on each text chunk.

```
# per-chunk entity extraction prompt, with context
def contextualize_er_extraction_prompt(context:str) -> str:
    """Creates a prompt with pre-amble file content for context during
entity+relationship extraction.
    The context is concatenated into the string, which later will be used
as a template
    for values like {schema} and {text}.
    """
    general_instructions = """
    You are a top-tier algorithm designed for extracting
    information in structured formats to build a knowledge graph.

    Extract the entities (nodes) and specify their type from the following
text.
    Also extract the relationships between these nodes.

    Return result as JSON using the following format:
    {{"nodes": [ {{"id": "0", "label": "Person", "properties": {{"name":
"John"}} }}],
    "relationships": [{{"type": "KNOWS", "start_node_id": "0",
"end_node_id": "1", "properties": {{"since": "2024-08-01"}} }}] }}

    Use only the following node and relationship types (if provided):
    {schema}

    Assign a unique ID (string) to each node, and reuse it to define
relationships.
    Do respect the source and target node types for relationship and
```

```
   the relationship direction.

   Make sure you adhere to the following rules to produce valid JSON
objects:
   - Do not return any additional information other than the JSON in it.
   - Omit any backticks around the JSON - simply output the JSON on its
own.
   - The JSON object must not wrapped into a list - it is its own JSON
object.
   - Property names must be enclosed in double quotes
   """

   context_goes_here = f"""
   Consider the following context to help identify entities and
relationships:
   <context>
   {context}
   </context>"""

   input_goes_here = """
   Input text:

   {text}
   """

   return general_instructions + "\n" + context_goes_here + "\n" +
input_goes_here
```

8.4 Make and Use the Knowledge Graph (KG) builder

8.4.1 Make the Neo4j KG Builder Pipeline

This function creates a customized KG builder pipeline for a specific file by extracting file context and creating a contextualized extraction prompt. It combines all the previously defined components (loader, splitter, schema, LLM) into a complete pipeline.

Process each approved markdown file by creating a KG builder pipeline and running it asynchronously. This extracts entities and relationships from the text chunks and stores them in the Neo4j database as the subject graph.

```
def make_kg_builder(file_path:str) -> SimpleKGPipeline:
   """Builds a KG builder for a given file, which is used to contextualize
the chunking and entity extraction."""
   context = file_context(file_path)
   contextualized_prompt = contextualize_er_extraction_prompt(context)

   return SimpleKGPipeline(
```

```
        llm=llm_for_neo4j, # the LLM to use for Entity and Relation
extraction
        driver=neo4j_driver,  # a neo4j driver to write results to graph
        embedder=embedder,  # an Embedder for chunks
        from_pdf=True,    # sortof True because you will use a custom loader
        pdf_loader=MarkdownDataLoader(), # the custom loader for Markdown
        text_splitter=RegexTextSplitter("---"), # the splitter you defined
above
        schema=entity_schema, # that you just defined above
        prompt_template=contextualized_prompt,
    )
```

🚨 **Different Run Results:** The output generated by LLMs can vary with each execution due to their stochastic nature. Your results might differ from those shown in the video.

```
from helper import get_neo4j_import_dir

neo4j_import_dir = get_neo4j_import_dir() or "."

for file_name in approved_files:
    file_path = os.path.join(neo4j_import_dir, file_name)
    print(f"Processing file: {file_name}")
    kg_builder = make_kg_builder(file_path)
    results = await kg_builder.run_async(file_path=str(file_path))
    print("\tResults:", results.result)
print("All files processed.")
```

8.5 Tool Definition for Entity Resolution

Connect entities in the subject graph to entities in the domain graph.

For each type of entity in the subject graph, you will devise a strategy for correlating with the right node in the domain graph.

For example, you should expect that Products with product names exist in the subject graph, and that these should correlate with products in the domain graph.

To do this, you will:

1. find the unique entity labels in the subject graph
2. find the unique node labels in the domain graph
3. attempt to correlate property keys
4. perform entity resolution by analyzing the similarity of property values

8.5.1 Unique Entity Labels in the Subject Graph

The unique triples of (subject, predicate, object) will give you an idea about what the subject graph looks like.

Query the Neo4j database to find all nodes that have the `__Entity__` label (entities created by the knowledge graph builder) and return their distinct label combinations.

```
# first, take a look at the entity labels
results = graphdb.send_query("""MATCH (n)
   WHERE n:`__Entity__`
   RETURN DISTINCT labels(n) AS entity_labels
   """)

results['query_result']
```

Flatten the label arrays into individual label strings using UNWIND, which transforms the array of labels into separate rows for each label.

```
# unwind those lists of labels
results = graphdb.send_query("""MATCH (n)
   WHERE n:`__Entity__`
   WITH DISTINCT labels(n) AS entity_labels
   UNWIND entity_labels AS entity_label
   RETURN DISTINCT entity_label
   """)

results['query_result']
```

Filter out internal Neo4j labels that start with double underscores ("__") to focus only on the meaningful entity type labels extracted from the text.

```
# filter out labels that start with "__"
results = graphdb.send_query("""MATCH (n)
   WHERE n:`__Entity__`
   WITH DISTINCT labels(n) AS entity_labels
   UNWIND entity_labels AS entity_label
   WITH entity_label
   WHERE NOT entity_label STARTS WITH "__"
   RETURN entity_label
   """)

results['query_result']
```

Combine the previous query steps into a reusable function that returns all unique entity labels from the subject graph, excluding internal Neo4j system labels.

```
# wrap the query into a callable function
def find_unique_entity_labels():
   result = graphdb.send_query("""MATCH (n)
       WHERE n:`__Entity__`
       WITH DISTINCT labels(n) AS entity_labels
       UNWIND entity_labels AS entity_label
       WITH entity_label
       WHERE NOT entity_label STARTS WITH "__"
       RETURN collect(entity_label) as unique_entity_labels
       """)
```

```
    if result['status'] == 'error':
        raise Exception(result['message'])
    return result['query_result'][0]['unique_entity_labels']
```

Test the function to see what entity labels were actually extracted from the processed markdown files into the subject graph.

```
# try out the function
unique_entity_labels = find_unique_entity_labels()

print("Unique entity labels: ", unique_entity_labels)
```

8.5.2 Unique Entity Keys for a Given Label

Create a function to find all unique property keys for entities of a specific label in the subject graph. This helps identify what properties are available for matching with domain graph nodes.

```
def find_unique_entity_keys(entityLabel:str):
    result = graphdb.send_query("""MATCH (n:$($entityLabel))
    WHERE n:`__Entity__`
    WITH DISTINCT keys(n) as entityKeys
    UNWIND entityKeys as entityKey
    RETURN collect(distinct(entityKey)) as unique_entity_keys
    """, {
        "entityLabel": entityLabel
    })
    if result['status'] == 'error':
        raise Exception(result['message'])
    return result['query_result'][0]['unique_entity_keys']

# try out the function to get the unique keys for
# subject nodes labeled as Product
find_unique_entity_keys("Product")
```

8.5.3 Unique Domain keys for a Given Label

Create a function to find unique property keys for nodes of a specific label in the domain graph (nodes without the `__Entity__` label). This enables comparison with subject graph properties for entity resolution.

```
def find_unique_domain_keys(domainLabel:str):
    result = graphdb.send_query("""MATCH (n:$($domainLabel))
    WHERE NOT n:`__Entity__` // exclude entities created by the KG builder,
these should be domain nodes
    WITH DISTINCT keys(n) as domainKeys
    UNWIND domainKeys as domainKey
    RETURN collect(distinct(domainKey)) as unique_domain_keys
    """, {
        "domainLabel": domainLabel
    })
```

```
    if result['status'] == 'error':
        raise Exception(result['message'])
    return result['query_result'][0]['unique_domain_keys']


find_unique_domain_keys("Product")
```

8.5.4 Normalize keys

This is a simple version of "stemming" as done in NLP.

Define a function to normalize property key names by removing label prefixes, converting to lowercase, and standardizing spacing. This helps match similar property keys that may have different naming conventions between subject and domain graphs.

```
def normalize_key(label:str, key:str) -> str:
    """Normalizes a a property key for a given label.

    Keys are normalized by:
    - lowercase the key
    - remove any leading/trailing whitespace
    - remove label prefix from key
    - replace internal whitespace with "_"

    for example:
        - "Product_name" -> "name"
        - "product name" -> "name"
        - "price" -> "price

    Args:
        label (str): The label to normalize keys for
        keys (List[str]): The list of keys to normalize

    Returns:
        List[str]: The normalized list of keys
    """
    lowercase_key = key.lower()
    unprefixed_key = re.sub(f"^{label.lower()}[_ ]*", "", lowercase_key)
    normalized_key = re.sub(" ", "_", unprefixed_key)
    return normalized_key

print(normalize_key("Product", "Product_name"))
print(normalize_key("Product", "Product Name"))
print(normalize_key("Product", "product name"))
print(normalize_key("Product", "price"))
```

8.5.5 Correlate Keys for a Given Label

Use fuzzy string matching to find correlations between entity graph property keys and domain graph property keys. This function compares normalized key names and returns matches above a similarity threshold, helping identify which properties can be used for entity resolution.

```
# use the rapidfuzz library for fuzzy text similarity scoring
from rapidfuzz import fuzz

# for a given label, get pairs of entity and domain keys that correlate
def correlate_entity_and_domain_keys(label: str, entity_keys: list[str],
domain_keys: list[str], similarity: float = 0.9) -> list[tuple[str, str]]:
    correlated_keys = []
    for entity_key in entity_keys:
        for domain_key in domain_keys:
            # only consider exact matches. this could use fuzzy matching
            normalized_entity_key = normalize_key(label, entity_key)
            normalized_domain_key = normalize_key(label, domain_key)
            # rapidfuzz similarity is 0.0 -> 100.0, so divide by 100 for
0.0 -> 1.0
            fuzzy_similarity = (fuzz.ratio(normalized_entity_key,
normalized_domain_key) / 100)
            if (fuzzy_similarity > similarity):
                correlated_keys.append((entity_key, domain_key,
fuzzy_similarity))
    correlated_keys.sort(key=lambda x: x[2], reverse=True)
    return correlated_keys

label = "Product"
entity_keys = find_unique_entity_keys(label)
domain_keys = find_unique_domain_keys(label)

# try correlating with a low-ish threshold
correlated_keys = correlate_entity_and_domain_keys(label, entity_keys,
domain_keys, similarity=0.5)

print(f"{label} correlated keys (entity key, domain key, similarity
score)...")

# show the keys
correlated_keys
```

8.5.6 Value Similarity using Jaro–Winkler Distance

The Jaro–Winkler distance is a string comparison method, emphasizing common prefixes to favor strings that match from the start.

- measures "edit distance" between two strings

- produces values from 0.0 (exact match) to 1.0 (no similarity)
- use `similarity = 1.0 - distance` to get a similarity score

See Jaro-WinklerDistance for details.

Ideally, you would sample a few values that you expect to correlate well, trying different similarity metrics to find one that works well for that particular value pair.

Neo4j provides many text similarity functions. Other options include:

- `apoc.text.hammingDistance`
- `apoc.text.levenshteinSimilarity`
- `apoc.text.sorensenDiceSimilarity`
- `apoc.text.fuzzyMatch`

And for vector similarity:

- `vector.similarity.cosine` to directly calculate cosine similarity
- `db.index.vector.queryNodes` to perform vector similarity search (after first creating a vector index on the domain nodes)

Wrap the entity resolution logic into a reusable function that correlates subject and domain nodes based on property value similarity using Jaro-Winkler distance. This creates the bridge between extracted entities and the existing domain graph.

```
# use the Jaro-Winkler function to calculate distance between product
names
results = graphdb.send_query("""
// MATCH all pairs of subject and domain nodes -- this is an expensive
cartesian product
MATCH (entity:$($entityLabel):`__Entity__`), (domain:$($entityLabel))
WITH entity, domain, apoc.text.jaroWinklerDistance(entity[$entityKey],
domain[$domainKey]) as score
// experiment with different thresholds to see how the results change
WHERE score < 0.4
RETURN entity[$entityKey] AS entityValue, domain[$domainKey] AS
domainValue, score
// experiment with different limits to see more or fewer pairs
LIMIT 3
""", {
   "entityLabel": "Product",
   "entityKey": "name",
   "domainKey": "product_name"
})

results['query_result']
```

Create `CORRESPONDS_TO` relationships between subject graph entities and domain graph nodes with similar property values. Uses MERGE to avoid duplicate relationships and adds timestamps to track when correlations were established.

```
# connect all corresponding nodes with a relationship
results = graphdb.send_query("""
MATCH (entity:$($entityLabel):`__Entity__`),(domain:$($entityLabel))
// use the score as a predicate to filter the pairs. this is better
WHERE apoc.text.jaroWinklerDistance(entity[$entityKey],
domain[$domainKey]) < 0.1
MERGE (entity)-[r:CORRESPONDS_TO]->(domain)
ON CREATE SET r.created_at = datetime()
ON MATCH SET r.updated_at = datetime()
RETURN elementId(entity) as entity_id, r, elementId(domain) as domain_id
""", {
   "entityLabel": "Product",
   "entityKey": "name",
   "domainKey": "product_name"
})

results['query_result']

# run this repeatedly to illustrate that MERGE only happens once
```

Test the Jaro-Winkler distance function by finding all pairs of subject and domain Product nodes where the name properties have similarity scores below a threshold, showing potential matches for entity resolution.

```
# wrap as a function
def correlate_subject_and_domain_nodes(label: str, entity_key: str,
domain_key: str, similarity: float = 0.9) -> dict:
   """Correlate entity and domain nodes based on label, entity key, and
domain key,
   where the corresponding values of the entity and domain properties are
similar

   For example, if you have a label "Person" and an entity key "name", and
a domain key "person_name",
   this function will create a relationship like:
   (:Person:`__Entity__` {name: "John"})-[:CORRELATES_TO]->(:Person
{person_name: "John"})


   Args:
       label (str): The label of the entity and domain nodes.
       entity_key (str): The key of the entity node.
       domain_key (str): The key of the domain node.
```

```
        similarity (float, optional): The similarity threshold for
correlation. Defaults to 0.9.

    Returns:
        dict: A dictionary containing the correlation between the entity
and domain nodes.
    """
    results = graphdb.send_query("""
    MATCH (entity:$($entityLabel):`__Entity__`),(domain:$($entityLabel))
    WHERE apoc.text.jaroWinklerDistance(entity[$entityKey],
domain[$domainKey]) < $distance
    MERGE (entity)-[r:CORRESPONDS_TO]->(domain)
    ON CREATE SET r.created_at = datetime() // MERGE sub-clause when the
relationship is newly created
    ON MATCH SET r.updated_at = datetime()  // MERGE sub-clause when the
relationship already exists
    RETURN $entityLabel as entityLabel, count(r) as relationshipCount
    """, {
        "entityLabel": label,
        "entityKey": entity_key,
        "domainKey": domain_key,
        "distance": (1.0 - similarity)
    })

    if results['status'] == 'error':
        raise Exception(results['message'])

    return results['query_result']


correlate_subject_and_domain_nodes("Product", "name", "product_name")
```

8.5.7 Correlate and connect the subject nodes to the domain nodes

Execute the complete entity resolution workflow by iterating through all extracted entity labels, finding the best property key correlations, and automatically creating connections between the subject graph and domain graph to complete the knowledge graph integration.

```
# do it all:
# - loop over all entity labels
# - correlate the keys
# - correlate (and connect) the nodes
for entity_label in find_unique_entity_labels():
    print(f"Correlating entities labeled {entity_label}...")

    entity_keys = find_unique_entity_keys(entity_label)
    domain_keys = find_unique_domain_keys(entity_label)
```

```
   correlated_keys = correlate_entity_and_domain_keys(entity_label,
entity_keys, domain_keys, similarity=0.8)

   if (len(correlated_keys) > 0):
       top_correlated_keypair = correlated_keys[0]
       print("\tbased on:", top_correlated_keypair)
       correlate_subject_and_domain_nodes(entity_label,
top_correlated_keypair[0], top_correlated_keypair[1])
   else:
       print("\tNo correlation found")
```

In this lesson, you'll continue with the knowledge graph construction. The previous lesson created the domain graph from CSV files according to the construction plan. Now, you will process the markdown files, chunking them up into the lexical graph and extracting entities for the subject graph. You will learn how to use the Neo4j graph library, perform the chunking and entity extraction. You'll also learn about techniques for doing entity resolution. Ultimately, this will not be an agentic part of the workflow. This will be handled entirely by tools. So, we're going to define some tools and also some helper functions that those tools need to actually do their work. The two tools we need are one, actually constructing a knowledge graph builder, or have one builder per file to have it process the files to do the chunking and extraction. The other function that you'll define is correlate subjects and domain nodes. These two tools together will do the graph construction out of the markdown files and then take the resulting extracted entities and correlate those between the subject graph and the existing domain graph that was previously created from the CSV files. That might be a lot to take in right now. It'll make sense as we go through each step. As with the other notebooks, you'll begin by importing the needed libraries. So let's go through doing the common setup, all the libraries that you need. We'll get those to import. Got to make sure that OpenAI is ready to go, and also check that Neo4j is ready. That all looks good. As part of the setup here, we need more than just the previous states that we had in the previous lessons that were created. So now we're about to start the workflow. The previous lesson actually created part of the graph in Neo4j. So because Neo4j will be fresh when we load this notebook, we need to add actually some of the product nodes that were created in the previous lesson. And we have a helper function for that. So this helper function, load product nodes, we can just go ahead and load it and call it. And what we're going to expect is that the only nodes that exist instead of the graph have the label product. Okay, that worked correctly. That's super. Now, you do need some of the initial states for this session. The initial state should be part of what was created from the previous parts of the workflow. And it's the same kinds of things we've loaded before. We need the construction plan and the approved files to actually work with. And also because we've done some of the planning for the entity extraction, we need the approved entities and the approved fact types. So you're going to create each of those initial states. Here's the approved construction plan. Here are the approved files, the approved entities, and the approved fact types. Okay. You can now start to define all of the different functions that we need to actually be creating a pipeline that's going to do all the processing of the markdown files, doing the chunking up and then extracting the entities. Now, the Neo4j graph-drive library has a convenient simple KG pipeline, which you can use to do all the processing of the chunks and the entity extraction. So for all the markdown files you'll be processing, you'll create some helper functions to actually get that set

up correctly. But first, let's take a look at the interface for the simple KG pipeline. Now, this is just a little sample code that of course won't run because it doesn't have valid values here, but just to show the shape of actually what it means to actually create an instance of the simple KG pipeline. It needs to have an LLM, of course, for actually doing some of the entity extraction. It needs a driver for Neo4j to actually write the graph out to. You're going to need an embedder interface. That could be the same LLM or a different one if you want to. And because we're processing markdown files instead of PDFs, this is where we'll say from PDF is false. But it's actually kind of true because we're going to use a custom PDF loader instead of loading PDFs, we're going to load markdown files. So this is just a little quirk about the way the interface works where we're going to end up being saying true here, but we will define a custom loader that we're going to add into this part here for the PDF loader. We're going to have a custom splitter as well because we have some assumptions about what the markdown looks like. And if you've done any kind of chunking before, you know that just doing text splitting itself is an art. There's probably multiple courses on deep learning here that you can take a look at to actually really dig into that topic. We're going to do a simplified version making some assumptions about the data that we have. You can also pass in the schema. And the schema is of course going to be based on what the previous lesson had done for setting up the proposed schema for what kinds of facts and what kinds of entities can be pulled out of the text. And then of course, to put all that stuff together, we're going to have a custom prompt that we pass in that's going to let the LLM know exactly what we're looking for and how to think about doing the extraction. Let's walk through the end-to-end workflow of the Neo4j KG builder to help you understand what these components do. First, each document is loaded. There's built-in support for PDFs, but you'll use a custom markdown loader. Next, the text splitter component will perform chunking. This is the common chunking that happens in many different frameworks. For each chunk, the chunk embedder will calculate a vector embedding. The chunk is then analyzed using an LLM as instructed by the entity and relationship extractor. Now this component will be configured to align with our knowledge extraction plan. Once the graph has been constructed, an optional graph cleaner can be used to clean up the graph. This all happens in memory. The KG writer is the component responsible for saving the in-memory graph into Neo4j itself. Finally, an entity resolver component will merge nodes that are likely the same entity. Okay, so now you can start to define some of these custom functions we're going to need to pass into the pipeline. And the first thing we'll do is set up a custom text splitter. And if you remember the markdown files that we had, the markdown files had a H1 type header at the beginning and then they had these page breaks basically for splitting up the reviews. And so we're going to use a simple regular expression text splitter. So for Neo4j, there's this base class called text splitter. You'll extend that with custom functionality. And so that's what we're going to do here. We're going to extend text splitter. And then in the run function for the text splitter, because it's part of the pipeline, here's where our custom functionality is going to be put in place. We're just going to use a regular expression that's going to split the text based on this regular expression is going to find that's going to be passed into the function itself. So for any kind of regular expression that's passed in, you can use that as the the split points within the text. The final part that it does here is go ahead and just use a list comprehension for taking each of those plain text splits and putting them into an object that the Neo4j graph library expects called text chunk. Text chunk will just have the text which is the string itself, an index for that particular chunk of text, and that's just going to be paired into a single list with these text chunk objects that the list is going to be called text chunks. You'll also be loading the markdown files with a special data loader, and similar to the text splitter, this is going to extend the base class from the Neo4j graph-drive library. And there's a couple of these that are available, but we're going to define a special one here ourselves. So we're going to

import the data loader base class, and also these types that actually need to be used for the upload. We're going to make it look as if we have parsing a PDF document, so we're going to load a PDF document type. In this markdown data loader, the key thing that's really going to be doing here is that, you know, that there's some metadata within the markdown that we actually want to be able to extract and add to the context of the data loading when the chunking is happening. So we've got an extra utility function inside of the class called extract title. It's going to use a very basic regular expression to just kind of assume that the first time it finds a single H1 headline, it's going to use that as the title for the document. And so when the run function is called here and the load is really the thing that's going to load the data from some data source, so it's going to load from the file system, it's going to take markdown text and just read an entire file, extract the title, and then turn all that into this document info here. And it's the document info that really is the interesting part. This is the kind of metadata for the document. That gets combined with the text of the document, which we've loaded here into markdown text, and put into this PDF document class. So the other things that you need to set up the knowledge graph construction pipeline is, of course, you need an LLM. We already have that, but we're going to be using OpenAI again, of course. We're going to be using OpenAI's embedder as well, and then we have to grab the Neo4j driver. So this is using Neo4j's graph-drive library for that. So Neo4j has support, of course, for OpenAI and other models as well. We're going to use OpenAI for both the LLM and also for the embeddings. So we're just going to create instances of those LLMs for Neo4j, the embedder as well, and we're going to grab the Neo4j driver from our handy-dandy singleton that we've been using for this entire series for Neo4j. That singleton lets you grab the internal driver that it uses, so that's what we're doing here. Okay, you've defined some of the utility functions needed for the knowledge graph pipeline and also some of the other components like the LLM and the embedder as well. And now we're going to turn to the context required for actually doing the knowledge graph entity extraction. Now, we're going to start with the entity schema. So the entity schema required by the Neo4j graph-drive package has a couple of different components. One is it wants to know the node types that it should look for inside of the text. And here we're just going to make a copy or just really an alias of the approved entities. We're also going to get the schema relationship types. And for that, we're going to go ahead and extract from the approved fact types that we have. All of those approved fact types actually exist as a dictionary where the key in the dictionary is actually the relationship type itself. So we're just going to extract all the keys that should be the schema relationship types, and you can see them here. You're also going to use the approved fact types, but as what's called schema patterns by the knowledge graph builder pipeline. And it's really just a repackaging of the existing information. So from each of the facts, we're going to turn that into a list that has the subject label, the predicate label, and the object label. And we're going to turn the predicate label over to make it uppercase, which is the convention within Neo4j. The complete schema then required by the knowledge graph builder is here. It's got the node types, relationship types, the patterns about how those fit together. And we're going to set this flag to false saying only use these types that we've just defined. Don't go adding new things. The next part that's important for getting the context for the entity extraction correct is a custom prompt. That's going to be injected with both the text within each of the chunks, but also the schema is going to be added to that. And also we're going to be having a special utility function that's going to be adding in the file context. The pipeline will be processing an individual chunk, and the context from the overall file is added to the prompt so that the LLM knows when it's looking at this chunk, what is the document that it was part of. So for that file context, we're going to have this little helper function here. All this is really doing is going to grab the first couple of lines of text in the file. That's going to include the title of the file, but also a little bit of introduction

information. That'll give enough context for each of the individual chunks. Then you can define the prompt itself. And here we're going to look at the prompt as kind of one big chunk of text. It's going to follow roughly the same format that we've had before. We have this notion of, you know, what is the role of the LLM in its current work, you know, what's the goal that it's actually after. But then a lot of this is setting up both some design hints and also the context. And the context includes both the schema definition and also that little bit of the file that we're going to extract. So as you look through this prompt, you'll see that it's got, you know, instructions about the role and goal. It's got some design instructions here about what it should do. It also has some specifications about what the output format should be like. Here also, for every chunk, the schema is going to be injected. And once all of that's been put in place, then as part of the output, we're going to ask it to do unique IDs for each of the nodes as it's creating those nodes. And also, we're giving it some instructions here about what to do with each of the properties. You can see in the output here that we want the node output to have some IDs, we want to know what the label is that it's picked, and also what are the properties that should be assigned to that entity. Also then, this is where the context from the file, so the document level context is going to be injected. And here because this is the helper function that's going to create the overall prompt, we're going to just actually inject that here as kind of static text. So it's not going to be injected each time the chunk is created. This is going to be part of the static text. And then finally, the chunk data itself will be placed into this template format here. Okay, with all the context that we've set up, with all the different helper functions that the knowledge graph builder needs for doing the pipeline processing, we can go ahead and make an all graph builder and then use it. So because we're going to create a knowledge graph builder for every file, so that we have a specialized prompt based on the file content, here's how we're going to set that up in a helper function. We're going to have a helper function that is going to make the knowledge graph builder. And the output of it will be a simple knowledge graph pipeline. That's the Neo4j graph-drag class. And the first thing we're going to do inside this helper function is get the document level context, and that's going to be using this file context utility that we created. And again, that's going to grab just the first couple of lines of the data file. And then from that, it's going to create the contextualized prompt passing in that context is the only argument. Then we're going to get the full prompt that we want to use. So when we create the simple knowledge graph pipeline, we're going to grab the LLM that was defined, the driver we're using, the embedder. We're going to pretend that we're processing a PDF because we've got a custom PDF loader that's actually going to be loading markdown. We're going to have that splitter that's going to be using a regular expression. Here the regular expression we're using is just going to be looking for dash-dash-dash because that's the markdown page break that if you look at the markdown files we saw earlier, that's how each of the reviews are split up. So we're going to use that as the text splitting. The schema that we assembled as the entity schema, and then finally this contextualized prompt that was just constructed just up here. Putting all that together, we get a complete pipeline that can then do the chunking up and then the entity extraction. Now that we have a helper function for generating a knowledge graph pipeline for a particular file, now we just need to go ahead and loop through all the files in the import directory. And for each of those, we're going to get the full file path, have a little output statement letting us know what's happening, create the knowledge graph builder, and then run the knowledge graph builder. Now, this will take a couple minutes. We have, I think about 10 files that it's actually going to process. Each one of those will take, depending on how much responsiveness we get out of OpenAI, it could take up to maybe a minute to actually do all the natural language processing of the document, doing the extraction, and along the way, we'll be also doing the chunking. Okay, once this loop is finished, you will end up with a complete lexical graph. That's the parts of the graph that's going to contain the chunks connected

to each other and connected to a document node as well. And then also the subject graph that's going to have all the extracted entities and also how they're related to each other. Now that we have all files processed, this has been completed. You now have a lexical graph, a subject graph, and a domain graph. However, the knowledge graph isn't complete because the subject graph and the domain graph are not connected. And if you recall, the domain graph was created from CSV files and we just created the subject graph as extracting data from markdown files. The next step is to connect the entities that we extracted from the markdown files into the subject graph. We want to be able to connect those with the domain graph. So if there are products in the domain graph that refer to the same product as the extracted entities through in the subject graph, we want to connect those. So we're going to define a couple of tools in order to do that. Now, for each type of entity in the subject graph, you're going to devise a strategy for correlating with the right node in the domain graph. For example, you should expect that the products with the product names exist in the subject graph. And that they should correlate with the products in the domain graph. So in order to do this, you will do a couple of things. First, you're going to find all the unique entity labels in the subject graph. Similarly, you'll find all the unique node labels in the domain graph. And then you'll attempt to correlate the property keys between those two different sets of labels. So you can figure out if there's a product with a couple of different properties in the subject graph, how does that correlate with a product with a couple of different properties in the domain graph? So that's the very final step is performing entity resolution by analyzing the similarity of property values. The first step is to find the unique entity labels in the subject graph. Now, let's take a look at the subject graph schema to see what the nodes look like. After the Neo4j graph-drive library has done its work, the resulting nodes that are part of the subject graph will have an extra label identifying them. They're going to be labeled with this underscore-underscore-entity-underscore-underscore label. So you'll see that plus additional labels to be identifying what type of entity it is. So if we run this query, matching any nodes, and we have a predicate looking for where that node has this label, we can then return the distinct sets of labels and we'll call that the entity labels. So let's take a look at that result. And you can see there's a couple of special labels that have been added here. There's the underscore entity one, which also is the underscore KG builder, indicating that the KG builder is the one responsible for creating those nodes. But what we really care about is that we see the product, location, issue, and feature. Now, we do know this because that's what we were expecting from the previous steps. But what's important is some of those steps might have had us trying to find things like location, but those might have failed to actually occur inside of the graph because the LLM failed to actually find them. So what we'll do instead is we're going to query the graph for what actually happened. Then based on the labels that we find inside of the graph, we'll call those the unique entity labels that are part of our subject graph. So let's walk through a little elaboration of this query. The first thing is, if you take that, what you realize is each of these rows is really a list of labels. So for each row, if we go ahead and just use that unwind clause, so here you're saying, here, unwind the entity labels, which is going to be the labels for every node. And we're going to call that individual rows as entity label. We can then return the distinct entity labels. We should see all the same values, but now rather than as a bunch of lists, we'll see a single list with all the values. Okay. So now the list is really just a couple of rows where each row has a single label, the KG builder, the product, the entity, you can see it right there. So let's elaborate a little bit further on this query. We want to filter out these underscore labels. So what we're going to do that by having exactly the same query. We're going to start with the match, have a predicate for just finding things that actually are entities, unwind all of those to their individual, and then with those, we're going to have another predicate that says, okay, now get rid of any of the entities that start with underscore. That should get rid of KG builder and

underscore entity. So that's the list that we're looking for, just product, location, issue, and feature. You can then find a utility function that just wraps that call to Neo4j. If you try out the utility function, you should see of course, just that list. It's exactly what we want. Now, for each of those unique entity labels, we also want to find the unique keys that each of those labels have. So for every one of those, we're going to do a similar thing. We're going to define a utility function that's going to look for all nodes that have a particular label, but instead, once we've found some sample set of those, we're going to find just the unique keys that occur on those. Let's take a look at what that utility function looks like. Here it's going to get an argument, which is what's the entity label that we're looking for. And now in the match, rather than finding all nodes, we're only going to find nodes that have a particular entity label. And here, we're going to have to turn things around a little bit. We actually want to look, of course, only where that label co-occurs with the underscore entity label. Then, rather than finding the distinct set of labels, we're going to find the distinct set of keys on those nodes and return those. And similar to what we did before, we're going to unwind those lists of keys into just a couple of rows of keys. So that's what the unwind is doing here. And then we're going to collect those all back into a list and return that as a single result. So let's define that function and then give it a run, looking just for the unique keys related to product within the subject graph. Okay. And it looks like we didn't really constrain that too much when we were doing the entity extraction. So this time around, the LLM actually found a lot of different keys that it created those entities with. So across the different products, here's all the different kinds of properties that it actually derived from the reviews. It figured out things like the material, different features about, I don't know, the shelf depth apparently appears somewhere. You have to look at the text of the markdown to actually see why the LLM discovered these different properties. Depending on the particular review that was being used for doing the extraction, some of the reviews probably have some of these properties being mentioned, others would not. So this is not going to be consistent across all properties. But some of these things probably are. But particularly, I'm thinking the name is going to co-occur. And you'll see later how we actually try to take this list of keys and correlate them with keys that are available on the domain graph, and that's how we're actually going to sync everything up. The next utility function that we will define is very similar, but now we're going to turn our attention to the domain graph. And on the domain graph, we already kind of know how to find the labels that are part of the domain graph. We know that they're labels where there's not an entity actual label on it. So if we match for a particular label, let's say for a product, and we're going to find the unique domain keys for a product, we're going to go ahead and match for that, and then filter out any of the ones that have an entity because those are part of the subject graph. And then same as we did for the previous function, that we're then going to go ahead and get the unique keys on those domain nodes and then collect those up into a list. And so if we look for the unique property keys on the domain labels, this should be consistent because these were all of course imported from a single CSV file. It's a much smaller list. They're probably correlated exactly with what the CSV file had, the product name, the price, the description, product ID. To help that process along, we're going to define a helper function called normalize key. You can just pass in for a particular label what is the key that you want to normalize here. And what this function is going to do is going to collect it to a lowercase key. It's going to remove extra whitespace that might have been pretty helpful. And if the prefix is actually present there where the prefix is the label itself, go ahead and remove that. That would do things like this product name would end up being just name. Then it could obviously make it very easy to correlate with the name key from the subject graph. So product underscore name would end up just being name. Product space name would be just name. And of course, a property key like price would just be price. I wouldn't get changed at all. So again, the implementation is pretty straightforward, but the

purpose of this is actually just to kind of make the property keys themselves easy to compare. And you can look at some examples just for sanity checking. That all looks good. The next utility function that you'll put together is for correlating keys for a given label. Now this is going to take advantage of some of the utility functions we already had. And let's take a look at the implementation. We're going to import a new library here in Python called RapidFuzz. It's really just used for doing text similarity scoring, and this is a simple edit-based scoring. There's a couple of different ways for actually trying to figure out whether text strings are very similar to each other. RapidFuzz is a pretty good library for that. Inside of the function itself, given a particular label, some keys from the entity node, and some keys from the domain node, we're then going to look at a similarity score and a threshold where like, how well do these keys correlate with each other according to the scoring from RapidFuzz's fuzz function? And then for keys that correlate very highly, we're going to pair those up and say these keys are probably worth comparing the values of. Okay, let's look at this in detail. The correlated keys, so the keys that we want to pair up, start off with being empty. And we're going to iterate through all the keys in the entity keys collection. And kind of classic style, we're going to have a for loop inside of a for loop. We're going to loop through all the entity keys, and then inside of that, we'll loop through all the domain keys. And for each pairing there, we're going to consider how closely they correlate. We're going to go ahead and just normalize each of those keys. And because the RapidFuzz similarity score goes from zero up to 100, we're actually going to flip that around so that it's look more like a similarity score. And we're going to go ahead and calculate the ratio of this normalized domain key and the entity key. That should be the fuzzy similarity. Now, if that fuzzy similarity happens to be greater than the similarity threshold that was passed into the function up here, which is currently set to 0.9 as a default. And so if it surpasses that threshold, we're going to go ahead and add that to the correlated keys as a pair that we think is valid. We'll go ahead and just sort those keys then, just so it's easier to take a look at the ones that are the most highly correlated versus less correlated. And we're going to take the top correlated ones later on to actually use. So after defining the function, we're going to go ahead and this part here is just to be trying it out. So let's define the function for a product. It's going to go ahead and find all the unique entity keys, find all the domain keys, pass it into this function and see the results. So for a product, we can see that name and product name correlate perfectly. Price and price correlate perfectly. Description and description correlate perfectly. But design description, that's pretty low correlation, dimensions and description, low correlation. Those aren't super great. What's the threshold that we passed in? Yeah, we we asked for a similarity of 0.9. You can try different values here to see what kind of different results you get. The key for value of 0.9 is one of those classic kind of high thresholds you want to set. Okay, one more bit of background here, and then we're going to actually put all this stuff together. We've now been able to correlate for a given label in the subject graph and in the domain graph, the keys on those labels, what the good pairings are for actually trying to decide whether two nodes have to be the same node. Now this is all by way of having what's called entity resolution. This is a technique for achieving that. There are many different techniques. This is a pretty good baseline. It works perfectly for this current data set that we have. Probably for different data sets, you're going to want to have multiple techniques. And of course, you can imagine an entire agent built around just trying to figure out for the data that you have, what's the right technique for actually doing the resolution. But for what we're doing today, we're going to make some assumptions. The next function that we actually want to use is actually when we're going through and processing all the data inside of Neo4j with some Cypher calls, Cypher has some support for doing string comparison methods as well. One of the value similarity functions that's available is called the Jaro-Winkler distance, and that is really just a string comparison method similar to you might use

vector similarity where you actually have to calculate a vector embeddings. Instead, you can figure out text distance in lots of different ways. Jaro-Winkler does that by calculating what's called the edit distance between two strings. And it really looks at how many small edits have to be made to take some string A and some string B. How many edits do you have to make to actually make them be the same string? So the values end up being between zero and one, where kind of an inverse way, zero ends up meaning that it's an exact match, and one means that there's no match whatsoever between these two strings. Now, Neo4j's library of text similarity scores include this Jaro-Winkler calculation, but you can also use Hamming distance, the Levenshtein distance, there are Sørensen–Dice similarity, and also fuzzy matching. And of course, if you wanted to, you could also do a cosine similarity using vector embeddings. All those have different, you know, features, and like everything else that we've been doing so far with this part of the knowledge graph construction, you could probably pick and choose which one's better for a given a particular data set. For the data set that we have, this Jaro-Winkler just seems to be just fine. So you can take a look at what Jaro-Winkler would look like in action using just pure Cypher. If you look at the Cypher query that's happening here in the match clause, we're actually going to be matching a pair of both entities and domains for a particular label. Here we're going to be looking for the entity label that's the same on both. And of course, the entity node also has the extra label of underscore entity. And then with the entity and the domain, and then calculating this Jaro-Winkler distance score. We're going to pass it on with a filter where the score has to be less than 0.4. If you remember, the score of zero means it's a perfect match. So we're going to have kind of the opposite way of scoring here. We're looking for low values. So filter on that and then return those values to take a look. So in this example, we're going to be passing in some query parameters. We're going to be looking for the entity label product both on the subject graph and also on the domain graph. On the subject graph, we're going to be looking for the entity key name, and the domain key is going to be product underscore name. Okay, you can see that these are perfect scores. Gothenberg table and Gothenberg table, these all look exactly the same, of course. So you would expect them to have a very low score. You can try this with different values to try to see what it would be like if you maybe had a little bit less strict threshold, like 0.5. You can see that the Gothenberg table and Vaster's bookshelf apparently are not terrible. Gothenberg table and Stockholm chair, I don't know how those are actually very similar text wise, but I guess the distance gives you that. This is why you want to end up with in practice having a really low threshold like 0.1 so that they're as close as possible to being exactly the same thing. So let's elaborate on that query a little bit. This is exactly the same query, but we've added one extra part. Whatever we find as two entities that have a very low threshold, so here we've repackaged things a little bit. Now in the where clause in this pairing of entities and domain nodes, we're going to have the where clause actually just have the predicate on the Jaro-Winkler distance. We're setting the threshold to be 0.1. And for any pairs that pass that correspondence, we're actually going to go ahead and create a relationship from the entity to the domain node. And so we're going to use the fixed type here. It's not going to be parameterized of corresponds to. So the relationship will be this entity node corresponds with this domain node. Now, there's two sub clauses that we're going to use here in case you decide to run this multiple times, this can be very useful. What you do a merge, merge instead of for days have like an upsert, and it actually has two sub clauses based on the behavior of the upsert. So if the merge ends up finding a match for actually what you've described, then there's a sub clause for on match. And then whatever follows that will be executed. Or if you're doing it the first time, it's being created, the clause on create is being called. So after you do the merge, if the very first time you've done it, on create will be called and we're going to set this value created at time with the timestamp. The other one, if you already have it created, you're just

matching again, we're going to say that updated at also gets a timestamp and that will be updated every time you run this again. Okay, and that runs across all the corresponding products. Now it looks pretty good. That's exactly the query that's going to do exactly what you want. So go ahead and wrap it inside a function call passing in the label, entity key and the domain key. And also the similarity score that you want to have as a threshold. The query is pretty much the same here. and you can see that the parameters are being passed in as query parameters from the arguments to the function. And again, we're just going to go ahead and run that. We're trying that out on product, name and product name. These are the two different keys from the entity and also from the domain graph. Great, so that found 10 relationships that made sense between the subject graph and the domain graph. So for completeness, while we know that we can connect the products, we want to do this for all the entities that are available. So to correlate and connect all of the subject nodes to the corresponding domain nodes, we'll just put in a for loop that's going to go through all of the unique entity labels, then try to correlate them from the entity side over to the domain side. So the subject nodes and the domain nodes will be connected. So you can see that product was connected, of course. Location actually doesn't have any correlation. Neither does issue or neither does feature, but that's what we expected. What has happened now as a result of doing all of this work, you finally have a complete domain graph that was constructed out of CSV files. You also have a lexical graph and a subject graph that was created from markdown files. And with this final step, you've connected the entities in the subject graph with the entities in the domain node. Now you have a completely connected knowledge graph. Well done.