Selected Code Extracts

0.1 Creating Nodes

```
query_string = '''
LOAD CSV WITH HEADERS FROM "https://raw.githubusercontent.com/
    MissMaya/CASA0004_DISSERTATION/main/full_cleaned_data/
    full_individuals_cleaned.csv" AS row

CALL {
  WITH row
  MERGE (p:Person
        {id: toInteger(row.id),
        full_name: row.full_name,
        first_name: row.first_name,
        last_name: row.last_name
        })
        ON CREATE SET
        p.year_from = CASE WHEN row.year_from = "no_date" THEN "
    no_date"
                        ELSE toInteger(row.year_from) END,
        p.year_to = CASE WHEN row.year_to = "no_date" THEN "no_date"
                        ELSE toInteger(row.year_to) END
} IN TRANSACTIONS
'''

conn.query(query_string);
```

Source Code 1: In this code sample, a Person node is created and takes several properties, added as key:value pairs on the node. Neo4j imports all data as strings so type conversions must be applied with operations such as `toInteger`. String placeholder values for null values cannot be converted and are dealt with in type conversion through the use of a `COALESCE` command.

```
query_string = '''
LOAD CSV WITH HEADERS FROM "https://raw.githubusercontent.com/
    MissMaya/CASA0004_DISSERTATION/main/full_cleaned_data/
    full_trades_guilds_cleaned.csv" AS row

CALL {
  WITH row
  FOREACH
  (ignoreMe IN CASE WHEN row.guild_memberships is not null THEN [1]
    ELSE [] END |
     MERGE (g:Guild {name: row.guild_memberships})
  )
} IN TRANSACTIONS
'''

conn.query(query_string);
```

Source Code 2: Where placeholders have not been supplied to deal with null values, a `FOREACH` clause is used to skip over records that do not have a value in a field.

0.2 Creating Relationships

```
query_string = '''
LOAD CSV WITH HEADERS FROM "https://raw.githubusercontent.com/
    MissMaya/CASA0004_DISSERTATION/main/full_cleaned_data/
    full_relationships_cleaned.csv" AS row

  MATCH (p1:Person {id: toInteger(row.source)}), (p2:Person {id:
    toInteger(row.target)})
  WITH p1, p2, row
  CALL apoc.merge.relationship(p1, row.relationship, {}, {}, p2)
    YIELD rel
  RETURN rel
'''

conn.query(query_string);
```

Source Code 3: Cypher code using an APOC procedure to create multiple types of relationship, in this case between Person nodes.

```
query_string = '''
query_string = '''
LOAD CSV WITH HEADERS FROM "https://raw.githubusercontent.com/
    MissMaya/CASA0004_DISSERTATION/main/full_cleaned_data/
    full_trades_guilds_cleaned.csv" AS row
with row
MATCH (p:Person {id: toInteger(row.id)})
MATCH (g:Guild {name: row.guild_memberships})
MERGE (p)-[m:MEMBER_OF]->(g)
'''

conn.query(query_string)
```

Source Code 4: Cypher code for creating a single type of relationship, in this case the `MEMBER_OF` relationship between Person and Guild nodes.

0.3 Creating a Cypher Projection

```
query_string = '''
CALL gds.graph.project.cypher(
     "undirected_graph",
    'MATCH (p:Person) RETURN id(p) AS id',
    'MATCH (s:Person)-[r]-(t:Person) RETURN id(s) AS source, id(t) AS
     target'
)
'''
conn.query(query_string)
```

Source Code 5: Cypher code for creating a named, undirected graph as a Cypher projection

0.4 Weakly Connected Components

```
query_string = '''
CALL gds.wcc.write(
    "undirected_graph",
    {
        writeProperty: "componentId"
    }
)
'''
conn.query(query_string)
```

Source Code 6: Cypher code for running the weakly connected components algorithm

0.5 Running the Louvain Algorithm

```
query_string = '''
CALL gds.louvain.write("undirected_graph", {writeProperty: "louv"})
YIELD communityCount, modularity, modularities
'''
conn.query(query_string)
```

Source Code 7: Cypher code for running the Louvain algorithm

0.6 Tabulating Results

```
query_string = '''
MATCH (p:Person)
OPTIONAL MATCH (p:Person)-[:HAD_GEO_LOCATION]->(st:Street)
OPTIONAL MATCH (p:Person)-[:WAS_ACTIVE_IN]->(l:Location)
OPTIONAL MATCH (p:Person)-[:MEMBER_OF]->(g:Guild)
OPTIONAL MATCH (p:Person)-[r:PRIMARY_ROLE]->(primary)
OPTIONAL MATCH (p:Person)-[o:OTHER_TRADE]->(other)
OPTIONAL MATCH (p:Person)-[m:MADE]->(maker)
OPTIONAL MATCH (p:Person)-[s:SOLD]->(seller)
RETURN p.louv as community,
       count(DISTINCT p) as community_size,
       count(DISTINCT st.coord_string) as number_of_streets,
       count(DISTINCT l.location) as number_of_towns,
       count(DISTINCT g.name) as number_of_guilds,
       count(DISTINCT primary.description) as number_of_primary_roles
   ,
       count(DISTINCT other.description) as number_of_other_trades,
       count(DISTINCT maker.product) as number_of_products_made,
       count(DISTINCT seller.product) as number_of_products_sold
ORDER BY community_size DESC
'''

conn.query(query_string)

results = pd.DataFrame([dict(_) for _ in conn.query(query_string)])
```

```
25
26 results.head(10)
```

Source Code 8: Cypher and Python code for generating a table of the 10 largest communities by number of maker, adding in counts of low and high level locations, guild memberships and products made and sold

0.7 Visualising Queries

```
1 '''
2 MATCH (p:Person)-[:WAS_ACTIVE_IN]->(l:Location) with p.louv as
     community, collect(l) as locations, collect(p) as members
3 WHERE community = 2609
4 UNWIND members as member
5 RETURN member,locations
6 '''
```

Source Code 9: Cypher code used to visualise the largest community by high-level location

```
1 '''
2 MATCH (p:Person)-[:HAD_ADDRESS]->(a:Address)
3 MATCH (a)-[:HAD_GEOLOCATION]->(geo)
4 with p.louv as community, collect(a) as addresses, collect(geo) as
     geolocations, collect(p) as members
5 WHERE community = 2609
6 UNWIND members as member
7 RETURN member, addresses, geolocations
8 '''
```

Source Code 10: Cypher code used to visualise the largest community by low-level location

```
1 '''
2 MATCH (p:Person)
3 OPTIONAL MATCH (p:Person)-[:MEMBER_OF]->(g:Guild)
4 with p.louv as community, collect(g) as guilds, collect(p) as members
5 WHERE community = 2609
6 CALL apoc.create.vNodes(['Community'], [{name:community}]) YIELD node
      as comm
7 UNWIND members as member
8 RETURN member, guilds
9 '''
```

Source Code 11: Cypher code used to visualise guild memberships in the largest community

```
1 '''
2 MATCH (p:Person)
3 OPTIONAL MATCH (p:Person)-[:MADE]->(m:Maker_Of)
```

```
OPTIONAL MATCH (p:Person)-[:SOLD]->(s:Seller_Of)
with p.louv as community, collect(m) as products_made, collect(s) as
    products_sold, collect(p) as members
WHERE community = 2609
UNWIND members as member
RETURN member, products_made, products_sold
'''
```

Source Code 12: Cypher code used to visualise products made and sold by the largest community