Knowledge Graphs Demystified Part 2: Practical Session

In this notebook we are going to construct a simple knowledge graph using Python, and run some queries on the graph in Neo4j.

If you would like to run this code yourself, you will need to install the `py2neo` package in Python 3.

To run part 3 onwards, you will need to install Neo4j, which can be downloaded at https://neo4j.com/download/ (https://neo4j.com/download/).

I will be running through the code during part 2 of the master class so there is no need to install anything unless you would also like to try the code out yourself and run some graph queries.

1. Read in the data

Before we can build a graph, we must first read in the example datasets:

- `work_order_file` : A csv file containing a set of work orders.
- `downtime_file` : a csv file containing a set of downtime events.

Here is an example of what the first few rows of each dataset look like:

Maintenance work orders

StartDate	FLOC	ShortText
10/07/2005	1234.1.1	repair cracked hyd tank
14/07/2005	1234.1.2	engine wont start
17/07/2005	1234.1.3	a/c blowing hot air
20/07/2005	1234.1.2	engin u/s

Downtime Events

StartDate	EndDate	FLOC	Description
11/07/2005	12/07/2005	1234.1.2	under repair
15/07/2005	16/07/2005	1234.1.2	engine down
18/07/2005	21/07/2005	1234.1.3	a/c down

We are using the simple `csv` library to read in the data, though this can also be done using `pandas`.

In [63]:
```
from csv import DictReader

work_order_file = "data/sample_work_orders.csv"
downtime_file = "data/sample_downtime_events.csv"

# A simple function to read in a csv file and return a list,
# where each element in the list is a dictionary of {heading : value}
def load_csv(filename):
    data = []
    with open(filename, 'r') as f:
        reader = DictReader(f)
        for row in reader:
            data.append(row)
    return data


work_order_data = load_csv(work_order_file)
downtime_data = load_csv(downtime_file)

for row in work_order_data:
    print(row)
```

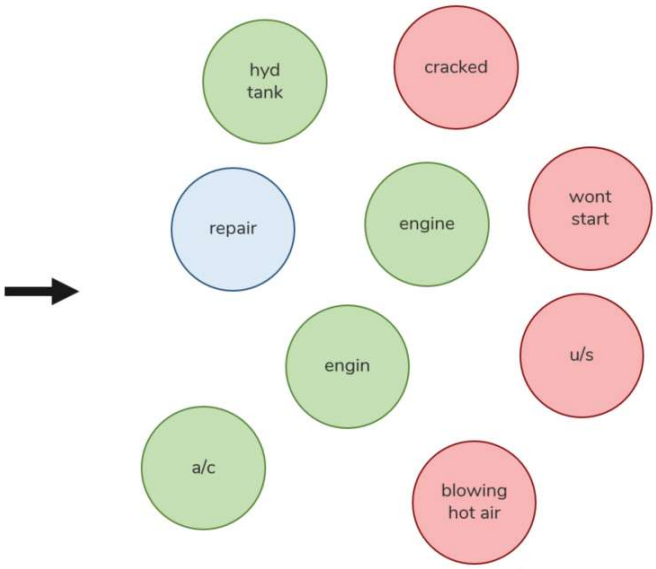
```
OrderedDict([('StartDate', '10/07/2005'), ('FLOC', '1234.1.1'), ('ShortText', 'repair cracked hyd tank')])
OrderedDict([('StartDate', '14/07/2005'), ('FLOC', '1234.1.2'), ('ShortText', 'engine wont start')])
OrderedDict([('StartDate', '17/07/2005'), ('FLOC', '1234.1.3'), ('ShortText', 'a/c blowing hot air')])
OrderedDict([('StartDate', '20/07/2005'), ('FLOC', '1234.1.2'), ('ShortText', 'engin u/s')])
OrderedDict([('StartDate', '21/07/2005'), ('FLOC', '1234.1.2'), ('ShortText', 'fix engine')])
OrderedDict([('StartDate', '22/07/2005'), ('FLOC', '1234.1.4'), ('ShortText', 'pump service')])
OrderedDict([('StartDate', '23/07/2005'), ('FLOC', '1234.1.4'), ('ShortText', 'pump leak')])
OrderedDict([('StartDate', '24/07/2005'), ('FLOC', '1234.1.4'), ('ShortText', 'fix leak on pump')])
OrderedDict([('StartDate', '25/07/2005'), ('FLOC', '1234.1.2'), ('ShortText', 'engine not running')])
OrderedDict([('StartDate', '26/07/2005'), ('FLOC', '1234.1.2'), ('ShortText', 'engine has problems startin
g')])
OrderedDict([('StartDate', '27/07/2005'), ('FLOC', '1234.1.4'), ('ShortText', 'pump fault')])
OrderedDict([('StartDate', '28/07/2005'), ('FLOC', '1234.1.4'), ('ShortText', 'pump leaking')])
OrderedDict([('StartDate', '29/07/2005'), ('FLOC', '1234.1.3'), ('ShortText', 'a/c not working')])
OrderedDict([('StartDate', '30/07/2005'), ('FLOC', '1234.1.3'), ('ShortText', 'a/c broken')])
```

2. Construct nodes from the entities in the short text

Our first task is to extract the entities in the short text descriptions and construct nodes from those entities. This is how we are able to unlock the knowledge captured within the short text and combine it with the structured fields.

Maintenance work orders

StartDate	FLOC	ShortText
10/07/2005	1234.1.1	repair cracked hyd tank
14/07/2005	1234.1.2	engine wont start
17/07/2005	1234.1.3	a/c blowing hot air
20/07/2005	1234.1.2	engin u/s



2.1 Define a Lexicon Tagger class

Extracting the entities in the short text is typically done using a trained Named Entity Recognition model, however for simplicity we will use a Lexicon.

The LexiconTagger class is a simple alternative to a trained named entity recognition model such as an LSTM or Transformer.

This class serves to automatically extract entities from each sentence using a predefined lexicon.

In [64]:
```
# Load the lexicon

import itertools

class LexiconTagger:
    """ A lexicon-based entity tagger.

    Args:
        lexicon_file: The filename of the lexicon.

    """
    def __init__(self, lexicon_file, max_ngram_size = 3):

        lexicon_data = load_csv(lexicon_file)
        self.max_ngram_size = max_ngram_size

        # Convert the loaded csv into a dictionary mapping word(s) to entity class
        self.lexicon = {}
        for row in lexicon_data:
            self.lexicon[row["key"]] = row["value"]



    def get_ngrams(self, sentence):
        """
            Given a sentence, return a list of all combinations of ngrams up to a certain size.

            Args:
                sentence: A list of words, e.g. ["fix", "broken", "pump"].

            Returns:
                ngrams: A list of ngrams containing up to max_ngram_size words.
                        For example, given the input ["fix", "broken", "pump"],
                        return ["fix", "broken", "pump", "fix broken", "broken pump", "fix broken pump"]

        """
        ngrams = []
        for n in range(self.max_ngram_size):
            for c in itertools.combinations(sentence, n + 1):
                ngrams.append(" ".join(c))
        return ngrams

    def extract_entities(self, sentence):
        """
            Given a sentence (a list of words), return a list of (word, entity_class) pairs.

            Args:
                sentence: A list of words.

            Returns:
                ngram_entity_pairs: A list of tuples, where each tuple contains (ngram, entity_class), for
example
                                    ("not working", "observation").

        """
        ngram_entity_pairs = []
        for ngram in self.get_ngrams(sentence):
            if ngram in self.lexicon:
                entity_class = self.lexicon[ngram]
                ngram_entity_pairs.append( (ngram, entity_class))
        return ngram_entity_pairs

    def normalise_ngram(self, ngram):
        """
            Given an ngram, return the equivalent item in the lexicon.

            Args:
                ngram: The ngram to normalise.

            Returns:
                normalised_ngram: The normalised version of the ngram according to the lexicon.
                                  If the ngram is not present in the lexicon, return the ngram itself.
        """
        normalised_ngram = self.lexicon[ngram] if ngram in self.lexicon else ngram
        return normalised_ngram
```

2.2 Extract entities from each work order

Now that we have defined a lexicon tagger, we can process each row in the work order data and extract a set of (ngram, entity_class) pairs for each row.

An "ngram" is a group of one or more words, for example "pump", "hydraulic tank", "blowing hot air".

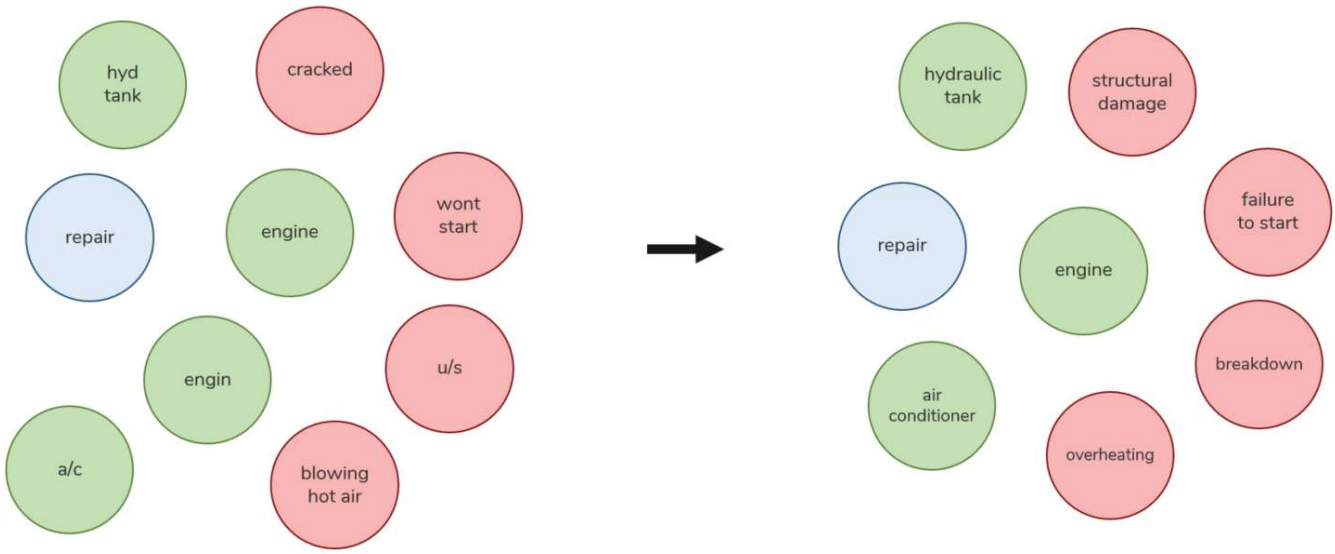
In [65]:
```
lexicon_file = "data/lexicon.csv"
lexicon_tagger = LexiconTagger(lexicon_file)

work_order_entities = []

for row in work_order_data:
    sentence = row["ShortText"].split() # We must 'tokenise' the sentence first, i.e. split into words.
    ngram_entity_pairs = lexicon_tagger.extract_entities(sentence)
    work_order_entities.append(ngram_entity_pairs)

for row in work_order_entities:
    print(row)
```

```
[('repair', 'activity'), ('cracked', 'observation'), ('hyd tank', 'item')]
[('engine', 'item'), ('wont start', 'observation')]
[('a/c', 'item'), ('blowing hot air', 'observation')]
[('engin', 'item'), ('u/s', 'observation')]
[('fix', 'activity'), ('engine', 'item')]
[('pump', 'item'), ('service', 'activity')]
[('pump', 'item'), ('leak', 'observation')]
[('fix', 'activity'), ('leak', 'observation'), ('pump', 'item')]
[('engine', 'item'), ('not running', 'observation')]
[('engine', 'item'), ('problems starting', 'observation')]
[('pump', 'item'), ('fault', 'observation')]
[('pump', 'item'), ('leaking', 'observation')]
[('a/c', 'item'), ('not working', 'observation')]
[('a/c', 'item'), ('broken', 'observation')]
```

2.3 Normalise the entities

The next step is to normalise the ngrams, i.e. convert each ngram into a normalised form. This is important as we would prefer to have a single node for a single concept, e.g. one node for "engine" as opposed to two nodes for "engin" and "engine".

We will once again be using a lexicon for this task, but it would typically be performed by machine learning.



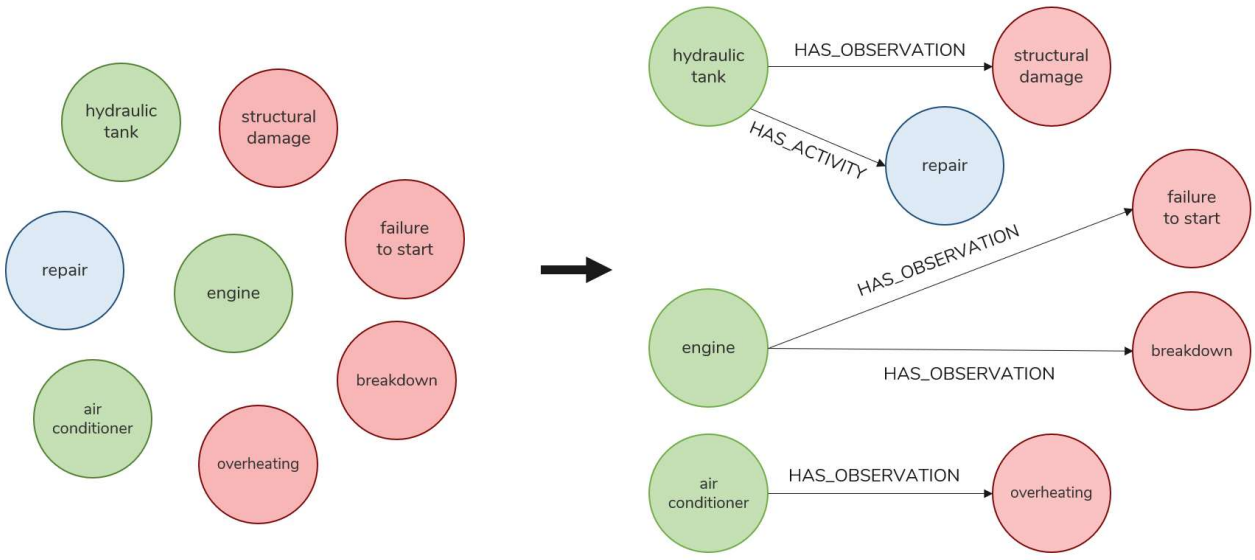
In [66]:
```
lexicon_n_file = "data/lexicon_normalisation.csv"
lexicon_normaliser = LexiconTagger(lexicon_n_file)

normalised_work_order_entities = []

# For every row in work_order_entities, replace each ngram with its normalised counterpart
# as per the normalisation lexicon.
# For example, "engin" will become "engine", "leaking" will become "leak", etc.
for row in work_order_entities:
    normalised_work_order_entities.append([(lexicon_normaliser.normalise_ngram(ngram), entity_class)
                                           for (ngram, entity_class) in row])


for row in normalised_work_order_entities:
    print(row)
```

```
[('repair', 'activity'), ('cracked', 'observation'), ('hydraulic tank', 'item')]
[('engine', 'item'), ('failure to start', 'observation')]
[('air conditioner', 'item'), ('overheating', 'observation')]
[('engine', 'item'), ('breakdown', 'observation')]
[('fix', 'activity'), ('engine', 'item')]
[('pump', 'item'), ('service', 'activity')]
[('pump', 'item'), ('leak', 'observation')]
[('fix', 'activity'), ('leak', 'observation'), ('pump', 'item')]
[('engine', 'item'), ('breakdown', 'observation')]
[('engine', 'item'), ('failure to start', 'observation')]
[('pump', 'item'), ('electrical issue', 'observation')]
[('pump', 'item'), ('leak', 'observation')]
[('air conditioner', 'item'), ('breakdown', 'observation')]
[('air conditioner', 'item'), ('breakdown', 'observation')]
```

2.4 Extract relations between the entities

Now that we have our normalised set of (ngram, entity_class) pairs for each work order, we need to build the relationships between them.

In our graph we are going to link each "item" to every other entity appearing in the work order.



In [67]:
```
triples = []

for row in normalised_work_order_entities:
    for (ngram, entity_class) in row:
        if entity_class != "item": continue

        # If this entity is an item, link it to all other entities in the work order

        for (other_ngram, other_entity_class) in row:
            if ngram == other_ngram: continue # Don't link items to themselves

            relation_type = other_entity_class.upper()
            triples.append(((ngram, entity_class), "HAS_%s" % relation_type, (other_ngram, other_entity_cl
ass)))

for triple in triples:
    print(triple)
```

```
(('hydraulic tank', 'item'), 'HAS_ACTIVITY', ('repair', 'activity'))
(('hydraulic tank', 'item'), 'HAS_OBSERVATION', ('cracked', 'observation'))
(('engine', 'item'), 'HAS_OBSERVATION', ('failure to start', 'observation'))
(('air conditioner', 'item'), 'HAS_OBSERVATION', ('overheating', 'observation'))
(('engine', 'item'), 'HAS_OBSERVATION', ('breakdown', 'observation'))
(('engine', 'item'), 'HAS_ACTIVITY', ('fix', 'activity'))
(('pump', 'item'), 'HAS_ACTIVITY', ('service', 'activity'))
(('pump', 'item'), 'HAS_OBSERVATION', ('leak', 'observation'))
(('pump', 'item'), 'HAS_ACTIVITY', ('fix', 'activity'))
(('pump', 'item'), 'HAS_OBSERVATION', ('leak', 'observation'))
(('engine', 'item'), 'HAS_OBSERVATION', ('breakdown', 'observation'))
(('engine', 'item'), 'HAS_OBSERVATION', ('failure to start', 'observation'))
(('pump', 'item'), 'HAS_OBSERVATION', ('electrical issue', 'observation'))
(('pump', 'item'), 'HAS_OBSERVATION', ('leak', 'observation'))
(('air conditioner', 'item'), 'HAS_OBSERVATION', ('breakdown', 'observation'))
(('air conditioner', 'item'), 'HAS_OBSERVATION', ('breakdown', 'observation'))
```

3. Create the graph

Now that we have our nodes and relations we can go ahead and build the Neo4J graph.

To do this we are going to use py2neo, a Python library for interacting with Neo4J.

There are also a couple of other ways to do this - you can either use Neo4J and run Cypher queries to insert each node and relation, or use the APOC library to import a list of nodes from a CSV file. I find Python to be the simplest way, however.

> Before proceeding, make sure you have created a new graph in Neo4j and that your new Neo4j graph is running.

You can download and install Neo4j from here if you haven't already: https://neo4j.com/download/ (https://neo4j.com/download/). I will be demonstrating the graph during the class so there's no need to have it installed unless you are also interested in trying out some graph queries yourself.

If you need to build your graph again, make sure to run this cell before running subsequent cells.

In [68]:
```
from py2neo import Graph
from py2neo.data import Node, Relationship

GRAPH_PASSWORD = "password" # Set this to the password of your Neo4J graph

graph = Graph(password = GRAPH_PASSWORD)

# We will start by deleting all nodes and edges in the current graph.
# If we don't do this, we will end up with duplicate nodes and edges when running this script again.
graph.delete_all()

tx = graph.begin()

# We will keep a dictionary of nodes that we have created so far.
# This serves two purposes:
#  - prevents duplicate nodes
#  - provides us with a way to create edges between the nodes
created_entity_nodes = {}

# Creates a node for the specified ngram and entity_class.
# If the node has already been created (i.e. it exists in created_nodes), return the node.
# Otherwise, create a new one.
def create_entity_node(ngram, entity_class):
    if ngram in created_entity_nodes:
        node = created_entity_nodes[ngram]
    else:
        node = Node("Entity", entity_class, name=ngram)
        created_entity_nodes[ngram] = node
        tx.create(node)
    return node


# Create a node for each triple in the list of triples.
# Set the class of each node to the entity_class (e.g. "activity", "item" or "observation").
# Create a relationship between the nodes in the triple.
for ((ngram_1, entity_class_1), relation, (ngram_2, entity_class_2)) in triples:

    node_1 = create_entity_node(ngram_1, entity_class_1)
    node_2 = create_entity_node(ngram_2, entity_class_2)


    # Create a relationship between two nodes.
    # This does not check for duplicate relationships unlike create_node,
    # so this code will need to be adjusted on larger datasets.
    relationship = Relationship( node_1, relation, node_2 )
    tx.create(relationship)


tx.commit()
```

3.1 Create nodes for the work orders

In order to query our graph, we need to create nodes for each work order in our dataset as well. We then need to link each Document node to every Entity node appearing in that document.

In [69]:
```
from dateutil.parser import parse as parse_date

# Our work_order_data and normalised_work_order entities allow us to do this quite easily,

tx = graph.begin()

# We will once again keep a mapping of created work order nodes, this time indexed by the row index.
created_work_order_nodes = {}

# Dates are a little awkward in Neo4j - we have to convert it to an integer representation in Python.
# The APOC library has functions to handle this better.
def date_to_int(date):
    parsed_date = parse_date(str(date))
    date = int("%s%s%s" % (parsed_date.year, str(parsed_date.month).zfill(2), str(parsed_date.day).zfill(2
)))
    return date

# The process of creating a work order node is a bit different to creating an entity,
# as we also want to incorporate some of the structured fields onto the node.
def create_structured_node(index, row, node_type, created_nodes):
    if index in created_nodes:
        return created_nodes[index]

    if 'StartDate' in row:
        row['StartDate'] = date_to_int(row['StartDate'])
    if 'EndDate' in row:
        row['EndDate'] = date_to_int(row['EndDate'])

    node = Node(node_type, **row)
    created_nodes[index] = node
    tx.create(node)
    return node

for i, row in enumerate(work_order_data):
    node = create_structured_node(i, row, "WorkOrder", created_work_order_nodes)

tx.commit()
```

3.2 Link the entities to their corresponding work order nodes

In order to properly query our graph, we need to link every entity node to the work order node in which it appears.

This allows us to run queries such as "pumps with electrical issues in the last 3 months".

In [70]:
```
tx = graph.begin()

# We can use the normalised_work_order_entries list to do this.
for i, row in enumerate(normalised_work_order_entities):
    for (ngram, entity_class) in row:

        node_1 = created_entity_nodes[ngram]
        node_2 = created_work_order_nodes[i]

        relationship = Relationship( node_1, "APPEARS_IN", node_2 )
        tx.create(relationship)

tx.commit()
```

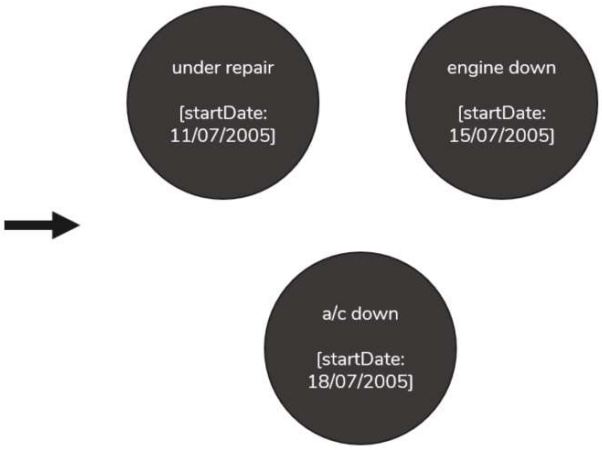
4. Extending the graph to incorporate Downtime events

The next step is to incorporate the downtime events.

For this exercise we are going to link the Downtime events to the first Item node appearing in the work orders with the same FLOC as the downtime event.

Downtime Events

StartDate	EndDate	FLOC	Description
11/07/2005	12/07/2005	1234.1.2	under repair
15/07/2005	16/07/2005	1234.1.2	engine down
18/07/2005	21/07/2005	1234.1.3	a/c down



In [71]:
```
tx = graph.begin()

created_downtime_nodes = {}

# Create a DowntimeEvent node for each row
for i, downtime_row in enumerate(downtime_data):
    node = create_structured_node(i, downtime_row, "DowntimeEvent", created_downtime_nodes)

    # Get all work order nodes with the same FLOC and link the DowntimeEvent to the Items appearing
    # in those work orders
    for j, work_order_row in enumerate(work_order_data):
        if work_order_row["FLOC"] == downtime_row["FLOC"]:

            work_order_entities = normalised_work_order_entities[j]

            for (ngram, entity_class) in work_order_entities:
                if entity_class != "item": continue    # We don't need to link non-items to downtime event
s

                item_node = created_entity_nodes[ngram]
                relationship = Relationship( item_node, "HAS_EVENT", node )
                tx.create(relationship)
                break


tx.commit()
```

5. Querying the graph

Now that the graph has been created, we can query it in Neo4j. This section lists some example queries that we can run on our graph. If you would like to try these yourself you can paste them directly into the Neo4j console.

First, let's try a simple query. Here is a query that searches for **all failure modes observed on engines**:

```
MATCH (e:Entity {name: "engine"})-[r:HAS_OBSERVATION]->(o:observation)
RETURN e, r, o
```

We can also use our graph as a way to quickly search and access work orders for the entities appearing in those work orders. For example, searching for **all work orders containing a leak**:

```
MATCH (d:WorkOrder)<-[a:APPEARS_IN]-(o:observation {name: "leak"})
RETURN d, a, o
```

We could extend this to also show the items on which the leaks were present:

```
MATCH (d:WorkOrder)<-[a:APPEARS_IN]-(o:observation {name: "leak"})<-[r:HAS_OBSERVATION]-(e:Entity)
RETURN d, a, o, r, e
```

Our queries can also incorporate structured data, such as the start dates of the work orders. Here is an example query for **all assets that had leaks from 25 to 28 July**:

```
MATCH (d:WorkOrder)<-[a:APPEARS_IN]-(e:Entity)-[r:HAS_OBSERVATION]->(o:observation {name: "leak"})-[:APPEARS_I
N]->(d)
WHERE d.StartDate >= 20050725
AND d.StartDate <= 20050728
RETURN e, r, o
```

On a larger graph this would also work well with other forms of structured data such as costs. We could query based on specific asset costs, for example.

Now that our work orders and downtime events are in one graph, we can also make queries about downtime events. Here is an example query for the **downtime events associated with assets appearing in work orders from 25 to 28 July (where the downtime events occurred in July)**:

```
MATCH (d:WorkOrder)<-[a:APPEARS_IN]-(e:Entity)-[r:HAS_EVENT]->(x:DowntimeEvent)
WHERE d.StartDate > 20050725
AND d.StartDate < 20050728
AND 20050700 <= x.StartDate <= 20050731
RETURN e, r, x
```

We can of course extend this to specific assets, such as pumps:

```
MATCH (d:WorkOrder)<-[a:APPEARS_IN]-(e:Entity {name: "pump"})-[r:HAS_EVENT]->(x:DowntimeEvent)
WHERE d.StartDate > 20050725
AND d.StartDate < 20050728
AND 20050700 <= x.StartDate <= 20050731
RETURN e, r, x
```

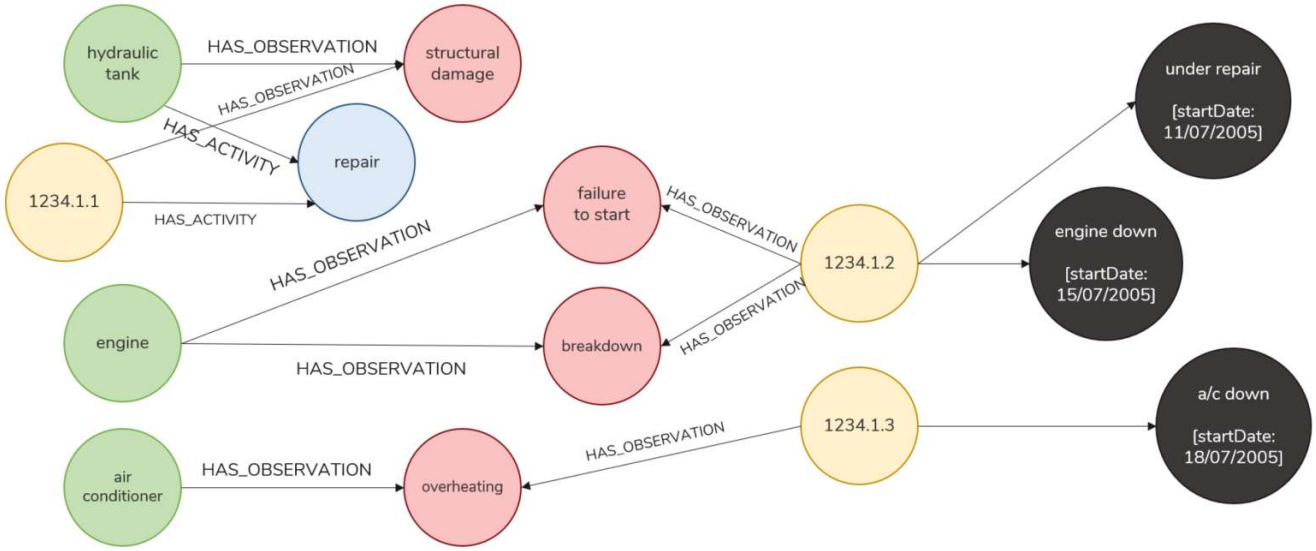
In larger graphs the downtime events could even be further queried based on duration, cost, lost feed, or date ranges.

6. Future improvements

Incorporating FLOCs

Our downtime events are currently linked to Item nodes, but it would make more sense to link them to nodes representing the functional locations.

If you are interested in continuing work on this small graph, the next best step would be to create nodes for the functional location data (`floc_data`) and to link the downtime events to those nodes as opposed to the Item nodes.



Frequencies on edge properties

We could also improve the graph by incorporating frequencies onto the edge properties. For example, if a "leak" occurred on a pump in two different work orders, our link between "pump" and "leak" could have a property called `frequency` with a value of `2` . This would allow us to query, for example, assets that had a particularly high number of leaks.

Constructing a graph from your own work order data

If you have a work order dataset of your own, feel free to download this code and try it out on your dataset.

If you need to extract entities not listed in the lexicon, you will need to update the lexicon file to include your new entities. Alternatively, the LexiconTagger can be substituted for a named entity recognition model.

In [72]:
```
floc_file = "data/sample_flocs.csv"
floc_data = load_csv(floc_file)

# Your code here
```