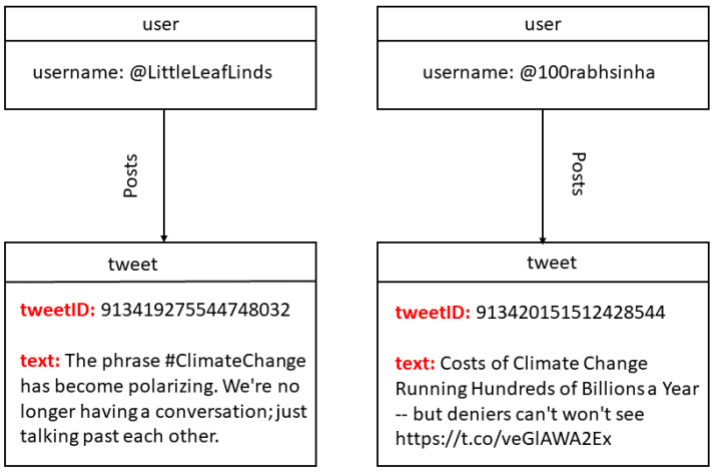
COMP3008 - Big Data Analytics

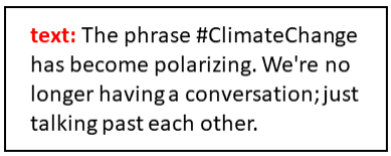
Workshop 2 - CSV Load

Introduction - Graph Databases

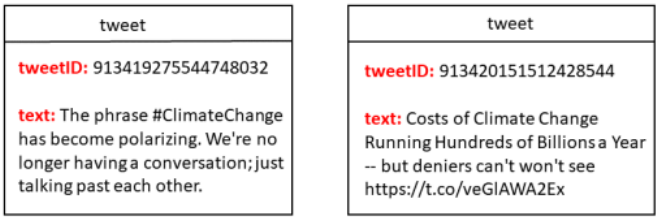


A graph database stores data in a graph, one of the most generic types of data structures. Here is an example of a graph that represents tweets posted in response to comments made by David King, the governments former Chief Scientist, with regards to climate change on 27 September 2017. We will approach this graph step-by-step in the following sections:

A graph records data in *nodes* and *relationships*. Indeed, the fundamental units that form a graph are nodes and relationships. The simplest possible graph is a single node. A node can have zero or more named values referred to as `properties`. Lets start out with a graph with only one node that has a single property named `text`.



Relationships provide directed, relevant connections between two nodes. A relationship always has a *direction*, a *type*, a *start node*, and an *end node*. Like nodes, relationships can have many properties. In most cases, relationships have quantitative properties, such as weights, costs, distances, ratings, time intervals, or strengths. As relationships are stored efficiently in Neo4j, two nodes can share any number or type of relationships without sacrificing performance. Note that although they are directed, relationships can always be navigated regardless of the direction.
There is one core consistent rule in graph databases: **No broken links**. Since a relationship always has a start and an end node, you cannot delete a node without also deleting its associated relationships. You can also always assume that an existing relationship will never point to a non-existing endpoint.
Lets add to the graph in the previous example another node, and one more property on the node (tweetID):

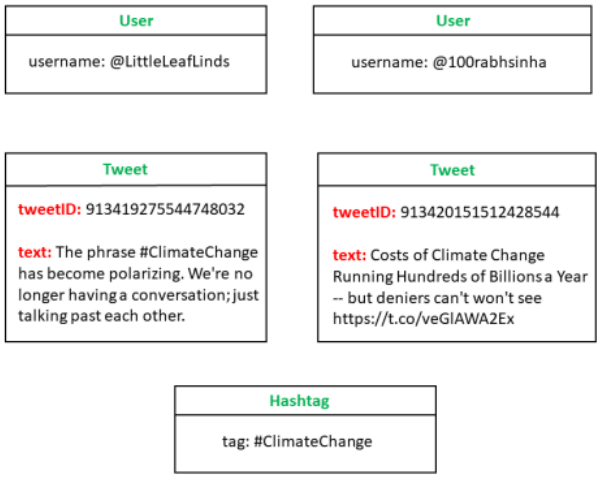


A *label* is a graph construct that is used to group nodes into sets; all nodes labelled with the same label belong to the same set. Many database queries can work with these sets instead of the whole graph, making queries easier to write and more efficient to execute. A node may be labelled with any number of labels, or none, making labels an optional addition to the graph.

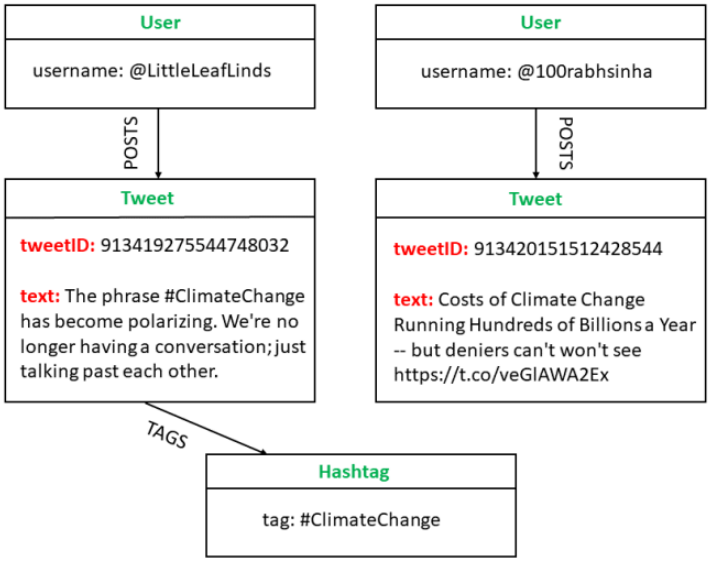
An example would be a label named User, which you can use to label all your nodes representing users. With that in place, you can ask Neo4j to perform operations only on your user nodes, such as finding all users with a given name.

You can also use labels to perform other tasks. For instance, since labels can be added and removed during runtime, they can be used to mark temporary states for your nodes- you might create an Offline label for phones that are offline.

In our example, we will add Tweet, User and Hashtag labels to our graph. Let's add two users-each with a property called username - and one hashtag to the graph. Then, label the nodes using the labels Tweet, User and Hashtag accordingly:



Relationships organise the nodes by connecting them. Relationships organise nodes into arbitrary structures, allowing a graph to resemble a list, a tree, a map, or a compound entity - any of which can be combined into yet more complex, richly interconnected structures. Our example graph will make a lot more sense once we add relationships, such as `POSTS` and `TAGS`:



Exercise 1 - Neo4j: Nodes and Relationships

The CREATE clause is used in Neo4j to create graph elements - nodes and relationships. Creating a single node is done by issuing the following command.

CREATE (n)

While this may be useful in some cases, we will typically want to create a node and associate it with a label. Lets create a node that represents a tweet. We can do so by adding the label Tweet to the node that we will create with the following command.

CREATE (n:Tweet)

When creating a new node with labels, you can add properties at the same time. Lets create a node that represents a tweet and has the `tweetID` and `text` displayed below:

```
tweetID: "913419275544748032"
text: "The phrase #ClimateChange has become polarizing. We're no longer having a conversation; just talking past each other."
```

The command would be as follows,

```
CREATE (t:Tweet { tweetID: "913419275544748032", text: "The phrase #ClimateChange has become polarizing. We're no longer having a conversation; just talking past each other." })
```

Now, lets create a User with the username `@LittleLeafLinds`.

```
CREATE (u:User { username: '@LittleLeafLinds' })
```

To create a relationship between two nodes, we have to specify the two nodes first, and then we have to name the relationship between them, indicating the start node and the end node. In our example, the first node will be the User node, the second node will be the Tweet node, and the relationship will be called POSTS to indicate that the user has posted the corresponding tweet.

```
MATCH (u:User),(t:Tweet)
WHERE u.username = '@LittleLeafLinds' AND t.tweetID = '913419275544748032'
CREATE (u)-[r:POSTS]->(t)
```

Then, we can create a second User, a second Tweet, and the relationship establishing that the second user posted the second tweet.

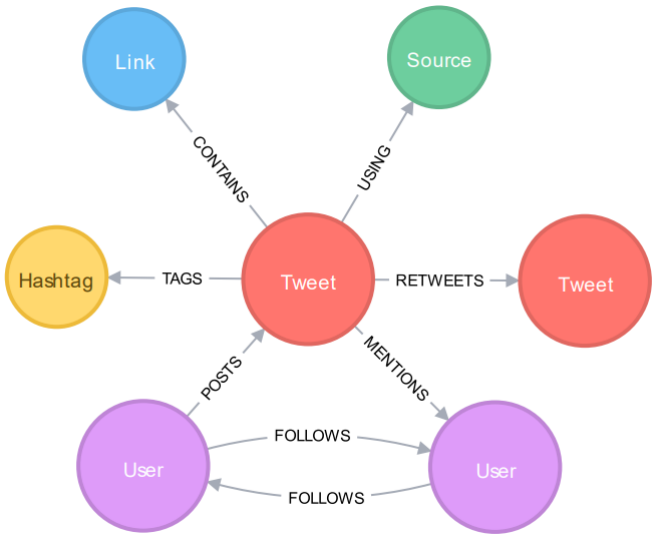
```
CREATE (u:User { username:  "@100rabhsinha" })

CREATE (t:Tweet { tweetID: "913420151512428544", text:  "Costs of Climate Change Running Hundreds
of Billions a Year -- but deniers can't won't see https://t.co/veGlAWA2Ex" })

MATCH (user:User),(tweet:Tweet)
WHERE user.username = "@100rabhsinha" AND tweet.tweetID ="913420151512428544"
CREATE (user)-[r:POSTS]->(tweet)
```

Finally, remove all the nodes and relationships created thus far before starting the following section. This can be done with the command,

```
// Delete all nodes
MATCH (n)
DETACH DELETE n
```

Exercise 2 - Uploading an entire dataset

Consider the following *Property Graph Model*. A property graph model contains connected entities - the nodes - which can hold any number of properties. Nodes can be tagged with labels representing their different roles in your domain.



This section demonstrates how to import CSV data into Neo4j and solutions to potential issues that might arise during this process. We will use Cyphers `LOAD CSV` command to transform the contents of various CSV files into a graph structure. The CSV files are available on the DLE, and you should download them before proceeding with the rest of the practical - look for the *Twitter Dataset* folder.

First, we will create the nodes for the tweets, users and hashtags. It should be observed that importing large amounts of data using `LOAD CSV` with a single query may fail due to memory constraints. This will manifest itself as an `OutOfMemoryError`. However, to prevent this situation, Cypher provides the command `CALL` command. The `CALL` command will process rows until the number of rows reaches a limit. Then the current transaction will be committed and replaced with a newly opened transaction. You can optionally set the limit for the number of rows per commit: `IN TRANSCTIONS OF 500 ROWS`. If no limit is set, a default value will be used.
The following command uploads all the tweets available in the `tweets.csv` file. I have uploaded the tweets dataset both with and without the `CALL` command so you can see how the syntax differs.

```
// Create Tweets
LOAD CSV WITH HEADERS FROM "file:///tweets.csv"
AS csvLine
CREATE (:Tweet {tweetID: csvLine.tweetID, username:  csvLine.username, text:  csvLine.text, date:
csvLine.date});
```

```
// Create Tweets
:auto LOAD CSV WITH HEADERS FROM "file:///tweets.csv"
AS csvLine
CALL(csvLine){
CREATE(:Tweet {tweetID: csvLine.tweetID, username:  csvLine.username, text:  csvLine.text, date:
csvLine.date});
}IN TRANSACTIONS OF 250 ROWS
```

If you are unable to execute the command above, it may be due to a security issue. There is a configuration setting in Neo4j to improve the security of the files being "ingested". The setting is specified in the configuration file `neo4j.conf`. The setting constrains all the files that we want to import to be under the `import` directory. For the purpose of this practical, we will remove or comment this setting to allow the files to be loaded from anywhere in the filesystem.
Once you have uploaded all the tweets available in the `tweets.csv` file, produce your own command to upload all the users - file `users.csv` - and all the hashtags file `hashtags.csv`.

Exercise 3 - Indexing data

Neo4j supports *indexes* on node properties to improve the performance of the application. We can create indexes on properties for all the nodes which have same label name. We can use these indexed columns with the MATCH or WHERE operator to improve the execution of the CQL commands. The main goal is to ensure their quick lookup when creating relationships. The following command should create an index for the `tweetIDs` and another index for the `usernames`.

```
// Next, we'll create indexes on the just-created
// nodes to ensure their quick lookup when
// creating relationships in the next step.
```

```
CREATE INDEX FOR (t:Tweet) ON (t.tweetID);
CREATE INDEX FOR (u:User) ON (u.username);
```

Exercise 4 - Relationships

Now, lets create the relationship that will establish the connections between users and tweets: POSTS. The following command should do the job:

```
MATCH (t:Tweet), (u:User)
WHERE t.username = u.username
MERGE (u)-[:POSTS]->(t)
return u,t;
```

Finally, create the necessary indexes and queries to connect tweets and hashtags via the `TAGS` relationship. You will need the file `tweets_and_hashtags.csv` to establish the relationships between tweets and hashtags.