# SEMANTIC MANAGEMENT PROJECT

# Graph Analytics

**Team DataImporta**

Niccolò Morabito

Víctor Diví i Cuesta

*June 2022*

# Context

DataImporta is an international commerce data analytics company. As such, the data we are dealing with consists of transactions (imports/exports) between countries.
In particular, for each transaction we have the following information available:

- Country of origin
- Country of destination
- Transaction date
- Product category
- Price
- Quantity

At the moment, the origin/destination entities of the transactions are countries, but in the future, this can change to also include the company that is buying/selling goods. This change wouldn't affect the analysis proposed in this report, since it would just increase the granularity of the data.

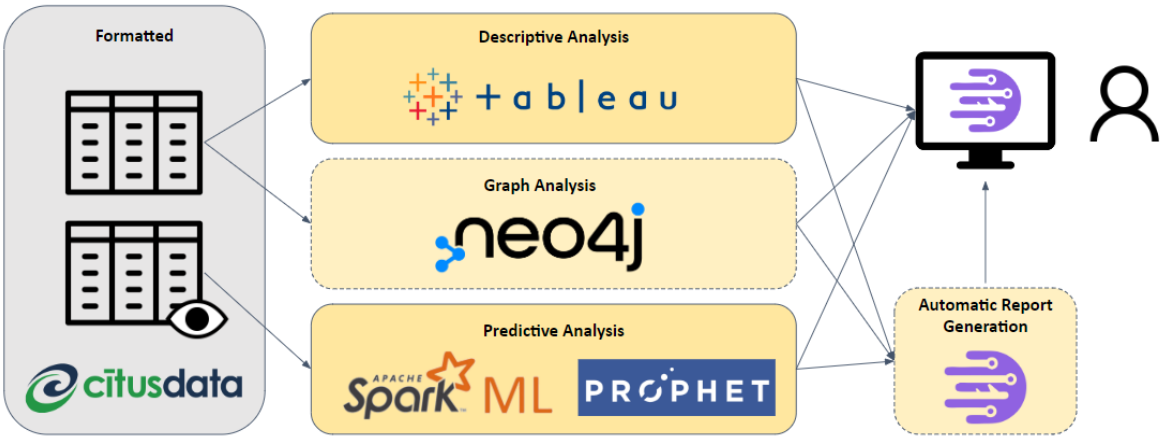


Figure 1. Exploitation phase of data pipeline (analytics)

We include graph-based analytics into our suite of analyses (see Figure 1 above) performed over the data (such as traditional analysis and time-series forecasting) to gain insights into the data that would be otherwise really difficult to obtain (with traditional analysis). In particular, we want to gain insights into two particular aspects:

- **Find countries that have a high volume of both exports and imports of certain products.**
  If a country imports and exports a high volume of the product, it is possible that it is acting just as an intermediary between the producers (high export and low import countries) and the consumers (low export and high import countries). Therefore,

detecting which countries are just middlemen would help give our customers better advice when looking for potential clients by skipping the middleman when possible and selling the same products with a greater margin.

- **Detect anomalous transaction patterns that encompass several countries.** In particular, we want to detect loops of increasing selling prices, which would mean that countries are first selling a product, just to buy that product again at a higher price. While these insights might not be very useful for export companies, they may be helpful to aid decision-making for international commerce policies in governments (e.g. Indian ban on wheat exports to avoid the inner market rising to unaffordable prices[1]) .

Note that for all the analyses, we are focusing on a single product category, so the graphs created are per category.

To carry out these analyses, the data (initially stored as a relational table) is converted into a **property graph**. We are using property graphs because it allows us to easily model the data with a simple schema (countries as nodes, transactions as edges). Since our edges have properties (they basically hold the vast majority of information), having as simple a representation would have not been possible with a knowledge graph. Besides, since we are not interested in interoperability of data sources (since data at this stage has already been integrated from multiple sources), and we have no semantics to exploit, there are not really many points in favour of knowledge graphs.

As per the concrete technology, we are using Neo4j  (and Cypher) and Spark. We are using Neo4j for its performance, reliability and support (it is ranked 1st in the db-engines graph database ranking[2]), although the most relevant aspect is the familiarity of the members of the project with it. We are also using Spark for the loop detection due to the limitations in the Pregel API provided by Neo4j (it only allows edges to have a single decimal weight).

Since we need a lot of data (from several countries) for these graph analytics to make sense (and actually give results) and we only have data from Brazil and Peru, we have generated fake data that we use to feed the graph and the algorithms. This data has the exact same structure as the one in the formatted zone of our data pipeline (see Figure 1 above), so the only integration it needs is to actually read the data from the database instead of from the CSV file (in fact, for the centrality analysis, we are using both fake data and real data from Brazil).

All the code of the project can be found in the repository[3].

---

[1] India wheat export ban: Why it matters to the world - BBC News (https://www.bbc.com/news/world-asia-india-61590756)

[2] https://db-engines.com/en/ranking/graph+dbms

[3] https://github.com/NiccoloMorabito/SDM

# Centrality Analysis

Finding the countries with a high volume of transactions basically means finding the *important* nodes in a graph, that is why the first aspect of our graph analysis can be modelled as a centrality problem. In particular, in our case, the importance of a node (country) is related to the number of incoming and outgoing edges that represent the number of import and export transactions respectively.

To perform this analysis, tabular data is transformed into a graph following the structure shown in Figure 2 below. In particular, nodes have only their code, and each edge has a price and a quantity. Temporality (i.e. transaction date) is not taken here into account because the data used for the graph is already time-bound.

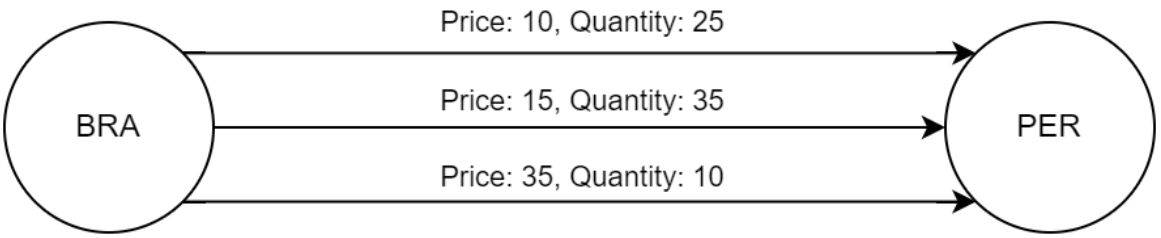


Figure 2. Graph schema

The **Degree Centrality algorithm**[4] counts the number of incoming and outgoing relationships from a node to find popular nodes within a graph. In our case, the degree of a node represents how influential the corresponding country is in the international trade market. It is implemented in the Neo4j Graph Data Science (GDS) library, and no particular projection is required to run the algorithm since it counts the number of transaction edges of each country:

```
CALL gds.degree.stream('countries')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS country, score AS
economic_influence
ORDER BY economic_influence DESC, country DESC
```

The countries are ordered by their economic influence so that it is evident which countries most of the transactions pass through.

[4] https://neo4j.com/docs/graph-data-science/current/algorithms/degree-centrality/

It is also possible to use a variant of the algorithm that measures the sum of positive weights of incoming and outgoing relationships (weighted Degree Centrality algorithm). Weighting the importance of a node according to the price of each transaction makes the final results more reliable if what we want to study is the influence in the international trade market.

```
CALL gds.degree.stream(
   'countries',
   { relationshipWeightProperty: 'price' }
)
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS country, score AS
centralityWeightedOnPrice
ORDER BY centralityWeightedOnPrice DESC, country DESC
```

Another possible property of the relationship that could be used is the quantity, but only in case the analysis is limited to a particular category or, better, to a particular product; otherwise, in case of aggregated data, the quantity would not be really relevant for our clients because a country could be considered more important than a country exporting a small quantity of a very expensive product because it exports a big quantity of a very cheap product. The top results of this analysis, on the other hand, represent the countries where most of the market money is concentrated.

Of course, all the resulting insights must be interpreted in an absolute way, because the transactions are neither weighted according to the population nor the surface of a particular country (that could deeply affect the production capacity) because the objective of the analysis is to find the most influential countries (and, in the future, companies) in the market.

Finally, other centrality analyses that could be carried out in the future, according to the clients' needs, are related to whether a country/company is transitively connected to other important countries/companies (to study how much they are integrated/isolated in the market) or whether it sits on the shortest path of lots of pairs of nodes (in order to see how often it plays a middleman role).

# Increasing Loop Detection

As mentioned previously, we want to detect abnormal patterns in international commerce. In particular, we want to find whether a country is exporting some goods at a low price and then

importing them at a higher price. To do that, we try to find loops in the transactions graph where each edge in the loop has a higher price than the previous.
Since the actual volume of products exported from one country to another is split into the multiple transactions there may be, we first need to aggregate the data in some way.
First, we convert the prices from absolute prices to price/unit.
Then, we add to the quantity of each edge the sum of the quantities of all the edges with a higher price. This “aggregation” transforms the semantic of each edge from “country *a* exported to country *b* *x* amount at price *p*” to “country *a* exported to country *b* *x* amount at price *p* or higher”. Figure 3 below shows an example of the transformation.

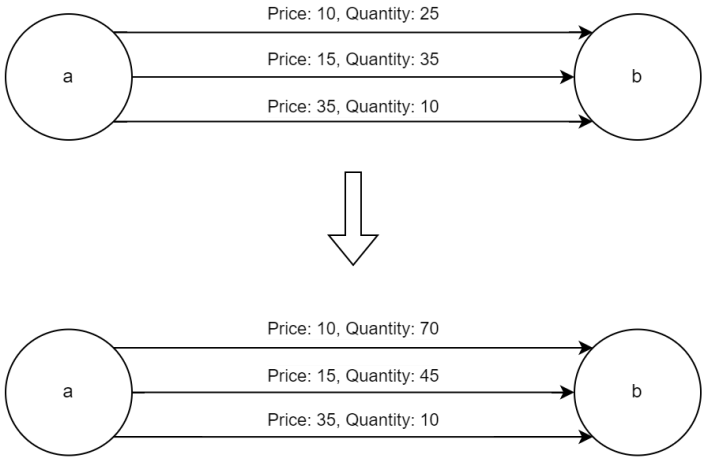


Figure 3. Edge aggregation

This way, lower-price edges “include” higher-price edges, so the paths being traversed don’t have to split when there are several edges with a higher price. This aggregation has been performed in a python script outside of Spark. In future work, when this whole spark graph analysis process is connected to the source database instead of using fake data, this transformation should be moved to spark.
The actual loop detecting algorithm has been implemented as a Pregel with the following parts

**Node information**

- Id
- Current paths: Array containing the new paths received in a superstep that are potentially valid. A path is a sequence of triples, each containing source node, price and quantity (basically source node and selected edge).
- Loops: Array containing the loops found that started from this node

**Edge information**

- Price: price per unit
- Quantity: amount in the transaction

**Message structure**

Array of paths. Although each message sent will be one single path, they are sent in an array to be able to merge several messages incoming to the same node since all paths will have to be processed.

**Initial message**

Dummy message with all values to 0.

**apply (VProg)**

Given a node, for each message (path) received:

- Check if it's a loop by checking the node id of the second element in the path (the first element will be 0 due to the initial message). If it is, save it in the loops array.
- If it's not a loop, check if the node has been visited earlier in the path (we want to avoid inner loops because if we end up finding an increasing cycle with an inner loop, the same cycle removing that loop will also be an increasing cycle). If it has been visited before, ignore the path.
- If it's not a fool loop and it doesn't contain an inner loop, save it in the current paths to use for sending new messages.

**send (sendMsg)**

Given an edge triplet, for every current path in the source node:

- If the edge price is lower than the last edge's in the path, go to the next path.
- Otherwise, add the source node and edge to the path and send it to the destination node.

**merge**

Given two messages (arrays of paths), return a single array that is the concatenation of both messages (no paths are discarded).

An important note is that the current implementation looks for loops starting at every node, and

Note that, even though the algorithm works, it has some aspects that could be improved, and are left as future work. For example, a node will send a message to another node through all the valid edges, while with only one message it would be enough. Although this doesn't yield invalid results (only duplicated ones), this can penalize performance and could be solved by filtering them in the merge phase.

Figure 4 below shows an extract of the output of the execution of the algorithm

```
OMN (0.41, 7225) → SRB (0.48, 8849) → LUX (1.26, 7501) → MMR (1.47, 3191) → SLB (13.63, 714) → MEX (17.41, 207) → OMN
GMB (0.11, 8389) → PRK (0.4, 5990) → LKA (0.9, 6955) → TON (0.91, 9758) → GHA (1.6, 4867) → FG (2.1, 3764) → GMB
VEN (0.06, 4080) → RE (0.27, 9421) → BRN (0.92, 5546) → GBR (0.94, 7625) → MCO (1.01, 6076) → GHA (1.49, 6333) → VEN
VEN (0.06, 4080) → RE (0.27, 9421) → BRN (0.92, 5546) → GBR (1.16, 8515) → SYC (1.67, 5579) → RWA (5.72, 1213) → PLW (6.48, 950) → VEN
FIN (0.03, 3997) → LUX (0.77, 7889) → WSM (0.79, 7426) → BFA (1.01, 545) → CYP (23.55, 416) → FIN
FIN (0.03, 3997) → LUX (1.23, 4503) → BRN (1.61, 5043) → CIV (1.79, 5461) → KAZ (3.35, 1596) → SMR (3.49, 2041) → FIN
ETH (0.67, 8723) → MCO (1.84, 4332) → ETH
ESP (0.34, 2865) → BGR (0.62, 6460) → BIH (0.75, 8358) → PAN (0.84, 928) → LVA (4.04, 1962) → ESP
ESP (0.12, 8231) → JPN (0.59, 9331) → BOL (0.6, 8135) → SLB (1.05, 8546) → HUN (17.88, 443) → ESP
CMR (0.31, 6807) → OMN (1.12, 5264) → CMR
CMR (0.2, 9793) → COD (0.25, 5913) → CRI (0.68, 5603) → SEN (0.97, 7750) → DOM (0.98, 6935) → LSO (2.66, 2046) → NLD (3.02, 812) → CMR
```

Figure 4. Algorithm output

Note that although the algorithm yields a lot of results, a lot of them will probably not be actually abnormal patterns. For example, an output of the same execution as the shown above is

```
MDG (0.29, 8352) → TKM (0.93, 9123) → MDG
```

Figure 5. Example of a potentially false positive

In this case, this loop is reported, but we can see that Madagascar (MDG) is exporting less quantity to Turkmenistan (TKM) than the other way round, so it's most likely a false positive. This is bound to happen in a lot of cases, so this first algorithm should actually be part of a more complex analytical pipeline that helps filter out those false positives. In fact, it would be a good starting point, since although it yields a lot of false positives, it's unlikely that it would yield a true positive (very few false negatives).