RaMa-Scene

Release 0.3-beta

Jul 09, 2020

Contents

1 Introduction 3
- 1.1 Implementation overview 3
- 1.2 Consumers 4
- 1.3 Query Management 5
- 1.4 Tasks 5
- 1.5 Models 5
- 1.6 Views 5
- 1.7 Analyze 5
- 1.8 Modelling 5

2 API descriptors 7
- 2.1 Default calculations 7
- 2.2 Modelling calculations 11

3 Calculation overview 13
- 3.1 Pre-calculating the matrices 13
- 3.2 The four calculation routes 13
- 3.3 Route 1 14
- 3.4 Route 2 15
- 3.5 Route 3 16
- 3.6 Route 4 17

4 Modelling overview 19
- 4.1 General description 19
- 4.2 Settings 19
- 4.3 Processing the settings 20

5 Frontend 23
- 5.1 Package Manager 23
- 5.2 Structure 23

6 Deployment 29
- 6.1 Install Redis [message broker] for Django Channels websocket support 29
- 6.2 Install Django dependencies & prepare SQLlite 30
- 6.3 Management commands and prepare static resources 30
- 6.4 Install and setup nginx [HTTP and Reverse Proxy Server] 31
- 6.5 Celery details and setup 31

6.6 Testing the application . . . 32
6.7 Daemonizing . . . 32
6.8 Management of database results . . . 32

7 Python initialise scripts **33**
7.1 Management commands . . . 33
7.2 Creating EXIOBASE numpy objects . . . 34
7.3 Building mapping coordinates files for the application . . . 34
7.4 Creating custom geojson and topojson files . . . 34

8 Testing **37**
8.1 Unit testing . . . 37
8.2 Integration test . . . 37

9 Performance **39**
9.1 Specs of tested server . . . 39
9.2 Load test setup . . . 39

10 ramascene **45**
10.1 ramascene package . . . 45

Python Module Index **57**

Index **59**

Welcome to RaMa-Scene's docs. This documentation is split into several parts:

1. Introduction. An overview of the application inner workings.
2. API descriptors. The front-end, back-end payloads.
3. Calculations overview. Overview of calculations performed.
4. Modelling. Overview of modelling procedures.
5. Frontend. Frontend description.
6. Deployment. Details for deploying the application.
7. Python initialise scripts. Independent scripts used for constructing files used by the application.
8. Testing. Unittests and integration tests.
9. Performance. Performance of the application and load test results.
10. Modules. Technical details.

CHAPTER 1

Introduction

RaMa-Scene is a Django+React web-application that allows for analyzing Environmentally Extended Input-Output (EEIO) tables from EXIOBASE v3.3.

Several on-the-fly calculations are performed to generate EEIO results. The on-the-fly calculations are made possible by loading in EXIOBASE raw data into memory and employing background processing using Celery.

1.1 Implementation overview

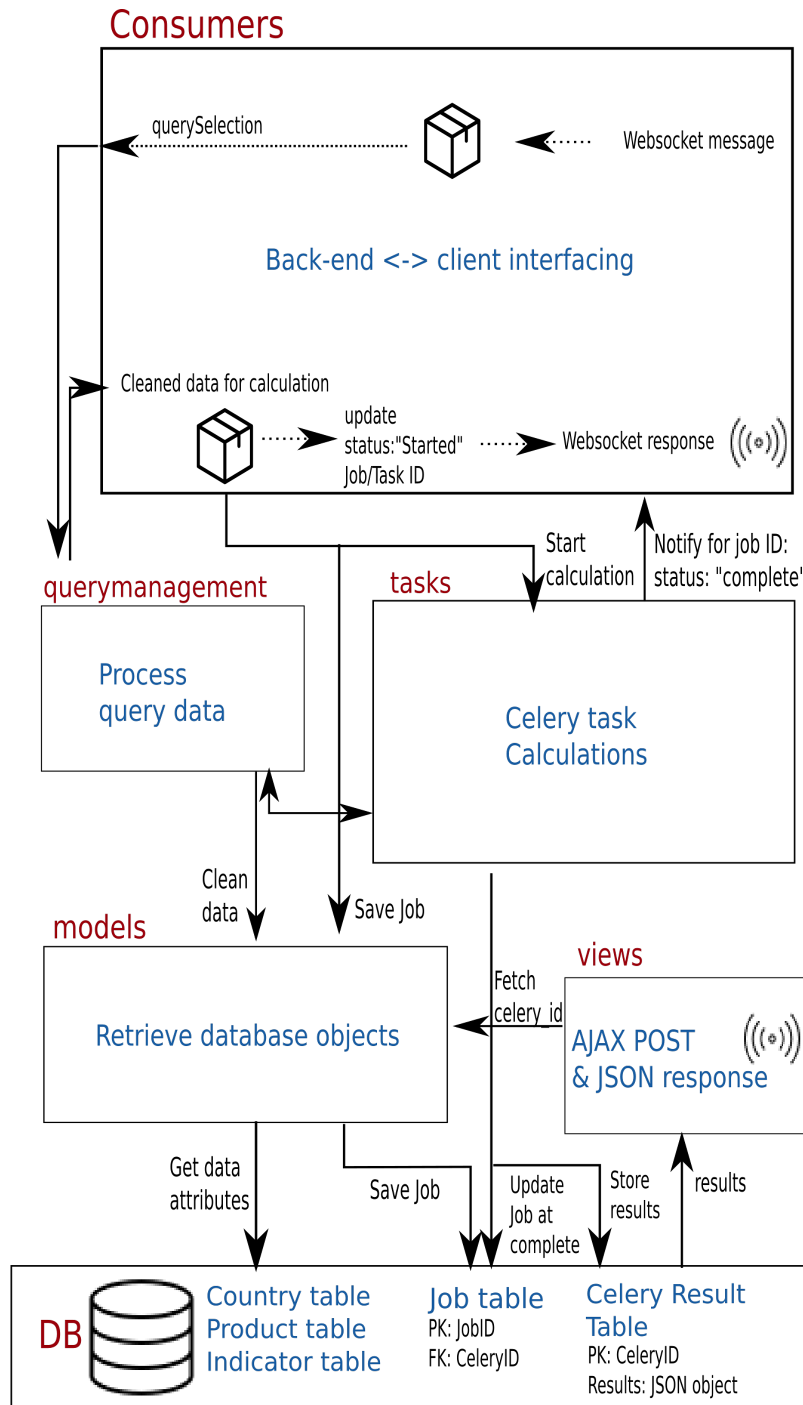
The back-end can receive two main types of requests from the front-end. A websocket-based request and an Ajax-based request.

Websockets are used for notifying the user and sending queries to the back-end, while Ajax is used to retrieve final results from the back-end.

Terminology:

- "Tasks" are used for front-end notifications of a given query and a "task" is a calculation process in Celery.
- "Jobs" are the database objects used by the back-end to remember which calculation (task) is started by which user.

  See figure below for an overview of the back-end processes.



1.2 Consumers

The module consumers is a Django Channels implementation that handles websockets. Meaning that the consumers module is one of the core communication methods between front-end and back-end. This module also invokes query management to process the queries.

1.3 Query Management

Any query received from the front-end needs to be processed in order to perform calculations and generate result data in a proper format using the query management module.

1.4 Tasks

This module is the heart of the EEIO calculations. The Tasks module implements the Celery background processing implementation that allows to process multiple calculations at the same time, but also calculations that take a long time.

1.5 Models

Models allows to insert and fetch database objects. It contains mapping coordinates for front-end and back-end communication as well as the calculation status and results per user.

1.6 Views

The views module handles the AJAX POST and JSON result response.

1.7 Analyze

For performing the actual IO calculations the analyze module is used.

1.8 Modelling

This module is used for implementation of modelling features in the application

CHAPTER 2

API descriptors

The React front-end has access to mapping coordinates reflecting product categories, countries and indicators. See project root static_assets:

1. final_countryTree_exiovisuals.csv
2. final_productTree_exiovisuals.csv
3. mod_indicators.csv

These mapping coordinates are not only used to render tree selectables, but also to transmit the global id's of the product categories, countries and indicators over the websocket channel. In turn the back-end handles these messages to perform calculations and store results. For example all countries and all products in the world represent the global id [1]. The indicator [1] represents product output. For further reference see the mapping CSV files.

API routing:

- API URL Websockets: `<domain-ip>/ramascene/`
- API URL AJAX: `<domain-ip>/ajaxhandling/`
- Interface format: JSON

2.1 Default calculations

The following queries denote the communication between front-end and back-end for performing default calculations.

Interface descriptors [websocket message to back-end]:

Stage	Instances relation	Variable name, dataType, example
Calculation type	Default calculation	**var name: action** ***JSON key: action, JSON value: String*** ex.: “action”:”default”
Dimension	Production, Consumption	**var name: querySelection** ***JSON key: dimType, JSON value: String*** ex.: “dimType”:”Production”
Visualization	TreeMap, GeoMap	**var name: querySelection** ***JSON key: vizType, JSON value: String*** ex.: “vizType”:”TreeMap”
Filter	Product	**var name: querySelection** ***JSON key: nodesSec, JSON value: array*** ex.: “nodesSec”:”[3,4,7]”
Filter	Country	**var name: querySelection** ***JSON key: nodesReg, JSON value: array*** ex.: “nodesReg”:”[1]”
Filter	Indicator	**var name: querySelection** ***JSON key: extn, JSON value: array*** ex.: “extn”:”[2]”
Year	Default reference year	**var name: querySelection** ***JSON key: year, JSON value: array*** ex.: “year”:”[2011]”
All	→ **to back-end** [WS send]	**var name: querySelection & action** ***JSON : querySelection, JSON: action*** ex.:see table below

→ **to back-end complete payload example:**

{
“action”:
“default”,
“querySelection”:{
“dimType”:”Production”,

“vizType”:”TreeMap”,
“nodesSec”:[3,4,7],
“nodesReg”:[1],
“extn”:[2],
“year”:[2011],
}
}

Interface descriptors [websocket messages from back-end]:

Stage	Instances relation	Variable name, dataType, example
Action request status	**from Back-end** → [WS response]	**var name: action** ***JSON key: action, JSON value: string*** ex.: {“action”:”started”}
Job status	**from Back-end** → [WS response]	**var name: job_status** ***JSON key: job_status,JSON value: string*** ex.: {“job_status”:”started”}
Job status	**from Back-end** → [WS response]	**var name: job_id** ***JSON key: job_id,JSON value: int*** ex.: {“job_id”:”176”}
Job name	**from Back-end** → [WS response]	**var name: job_name** ***JSON key: job_name,JSON value: JSON*** ex.: full querySelection as names

→ **from back-end complete response example:**

{
“job_id”:176,
“action”:”check status”,
“job_status”:”completed”,
“job_name”:{
‘nodesReg’: [‘Total’],
‘vizType’: ‘TreeMap’,
‘nodesSec’: [‘Fishing’, ‘Mining and quarrying’, ‘Construction’],
‘dimType’: ‘Production’,
‘extn’: [‘Value Added: Total’],
‘year’: [2011]

}
}

If the websocket message job_status is set to "completed", the front-end can perform a POST request for results via Ajax containing the job_id named as 'TaskID'. For example in the above websocket response we see that job_id is 176, the Ajax POST request is 'TaskID:176'.

Interface descriptors [AJAX response]:

Stage	Instances relation	Variable name, dataType, example
Retrieve calculation	**from Back-end** → [AJAX response]	**var name: unit** ***JSON key: name, JSON value: string*** ex.: {"Value Added":"[M.EUR]"}
All	**from Back-end** → [AJAX response]	**var name: job_id** ***JSON key: job_id, JSON value: int*** ex.: {"job_id":"176"}
All	**from Back-end** → [AJAX response]	**var name: rawResultData** ***JSON key: name, JSON value: array*** ex.: {"Total":"[1256.67]"}
All	**from Back-end** → [AJAX response]	**var name: job_name** ***JSON key: job_name, JSON value: JSON*** ex.: full querySelection as names

→ **from back-end complete response example:**

{
"job_id":176,
"unit": {"Value Added: Total":"M.EUR"},
"job_name": {
"nodesReg": ["Total"],
"nodesSec": ["Fishing", "Mining and quarrying","Construction"],
"dimType": "Production",
"extn": ["Value Added: Total"],
"year": [2011],
"vizType": "GeoMap"
},
"rawResultData":{
Fishing":75172.94699626492, "Mining and quarrying":2151937.135835223,
"Construction":3148250.604361363

    }
}

An important aspect is that in the current version the back-end expects the websocket message to contain a single value for indicator and year. Additionally if the query selection contains "GeoMap" the "nodesReg" descriptor can be an array of multiple elements denoting multiple countries, while the "nodesSec" descriptor can only have a single indicator. On the other hand if the query selection contains "TreeMap" the "nodesSec" descriptor can be an array of multiple elements denoting multiple products, while the "nodesReg" descriptor can only have a single indicator.

2.2 Modelling calculations

The following table denotes the communication between front-end and back-end for modelling calculations. Modelling is applied on existing default query selections.

Stage	Instances relation	Variable name, dataType, example
Product of interest	Model details : Product	**var name: model_details** ***JSON key: product, JSON value: array*** ex.: "product":[1]
Manufacturing region	Model details : Product origin region	**var name: model_details** ***JSON key: originReg,JSON value: array*** ex.: "originReg":[3]
Model type of calculation	Model details : Model type	**var name: model_details** ***JSON key: consumedBy, JSON value: array*** ex.: "consumedBy":[4]"
Region consuming	Model details : Region of consumption	**var name: model_details** ***JSON key: consumedReg, JSON value: array*** ex.: "consumedReg":[5]
Technological change	Model details : Technical change	**var name: model_details** ***JSON key: techChange, JSON value: array*** ex.: "techChange":[5]

The technological change is a single value denoting a percentage. See below for a full query example:

→ **from back-end complete response example:**

{
"action": "model",
"querySelection": {
    "dimType": "Production",

“vizType”: “TreeMap”,
“nodesSec”: [1],
“nodesReg”: [4,5],
“ext”: [8],
“year”: [2011]
},
“model_details”: [
{
“product”: [1],
“originReg”: [3],
“consumedBy”: [4],
“consumedReg”: [5],
“techChange”: [-15]
},
{
“product”: [6],
“originReg”: [5],
“consumedBy”: [7],
“consumedReg”: [18],
“techChange”: [20]
}
]
}

Multiple model selections can be added, however a user can only specify a single-selection per “product”, “originReg”, “consumedBy”, “consumedReg” in the array for this version of the application. The websocket response contains the added model details specified by name.

CHAPTER 3

Calculation overview

The following calculation explanations references the ramascene.analyze module code.

3.1 Pre-calculating the matrices

To reduce calculation times and to reduce load on the server the Leontief inverse or total requirements matrix (**L**) is precalculated according:

$$\boldsymbol{L} = (\boldsymbol{I} - \boldsymbol{A})^{-1}$$

When storing the Leontief inverse as an intermediate result, calculations on the server are reduced to simple matrix – matrix multiplication and the computational intense task of solving a system of linear equations or making a full matrix inverse can be avoided.

The matrices that are stored for further calculation are **L** , **Y** and **B**. They are stored as binary objects in the form of numpy arrays.

3.2 The four calculation routes

If environmental impacts related to final consumption are analysed it is possible to compare the environmental impacts from different points of view. We can compare the impact of consumed products between countries or between different products. This is what is called the consumption based view. Or given a certain final consumption of products we can calculate where the emissions are taking place and compare these between producing sector or countries. This is what we call the production point of view. All in all we distinguish between four different ways comparisons can be made. They are shown in Figure 1.

In each of the four calculation routes, the principal calculation that in each route is done is:

$$\boldsymbol{M} = \boldsymbol{B}\,\boldsymbol{L}\,\boldsymbol{Y}$$

The way the resulting calculations are presented are however different each time. In route 1 the calculated indicators results are presented per final consumed product. In route 2 the calculated indicator results are presented per consuming

country. In route 3 the results are presented per producing country and in route 4 they are presented per produced product. For each route the details are described below.

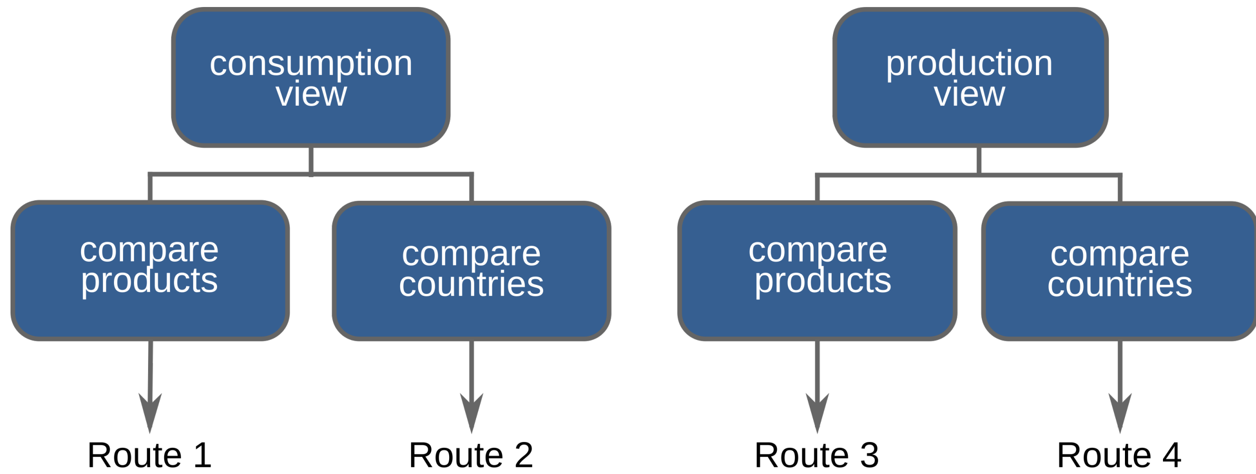


Figure 1: Consumption and production view and the corresponding calculation routes.

3.3 Route 1

In this calculation route the user can compare the environmental impacts associated with the final consumption of different products in a set of selected countries. Environmental impacts generated by production in all countries and by the production of all products are taken into account.

The calculation route has been designed for general application. It can calculate the environmental impacts of different products given a specific selected country selling final product or specific country where the emission takes place or specific sector where the emission takes place. However the actual implementation of route 1 takes into account that country selling product are all countries, the country where the emission takes place are all countries and the emission at all sectors are taken into account.

The calculation starts by creating the final demand vector that contains the selected products for the selected countries.

The symbol **Y** represents the multi-regional final demand matrix, that can be subdivided into sub-vectors that contain final demand for domestically produced products and final demand for imported products. In the final demand table there is no further subdivision into final demand by households, changes in stocks etc. Assume that there are three countries:

$$\boldsymbol{Y} = \begin{pmatrix} \boldsymbol{y}_{11} & \boldsymbol{y}_{12} & \boldsymbol{y}_{13} \\ \boldsymbol{y}_{21} & \boldsymbol{y}_{22} & \boldsymbol{y}_{23} \\ \boldsymbol{y}_{31} & \boldsymbol{y}_{32} & \boldsymbol{y}_{33} \end{pmatrix}$$

And the consuming country 1 and 3 have been selected then Y first becomes:

$$\boldsymbol{Y_s} = \begin{pmatrix} \boldsymbol{y}_{11} & 0 & \boldsymbol{y}_{13} \\ \boldsymbol{y}_{21} & 0 & \boldsymbol{y}_{23} \\ \boldsymbol{y}_{31} & 0 & \boldsymbol{y}_{33} \end{pmatrix}$$

where the subscript **s** stands for selected elements. Subsequently the total final demand for each product is calculated:

$$\boldsymbol{y}_s = \boldsymbol{Y}_s \boldsymbol{i}$$

Where **i** is a column vector of ones of appropriate length.

It is important to note that the user selects a product without specifying the origin of a product. For instance, if a user selects wheat as a product of interest, in the final demand vector wheat from every origin is selected i.e. wheat from Austria, wheat from Belgium etc.

For each of the final consumed products selected the output from each sector (**X**) needed to produce that product is calculated as:

$$\boldsymbol{X} = \boldsymbol{L}\,\widehat{\boldsymbol{y}}_s$$

At this point it is possible to make a sub-selection from **X** to select only the output in countries and sectors that are of interest to the user. For instance if we assume a three country case **X** can be expressed as:

$$\boldsymbol{X} = \begin{pmatrix} \boldsymbol{X}_{11} & \boldsymbol{X}_{12} & \boldsymbol{X}_{13} \\ \boldsymbol{X}_{21} & \boldsymbol{X}_{22} & \boldsymbol{X}_{23} \\ \boldsymbol{X}_{31} & \boldsymbol{X}_{32} & \boldsymbol{X}_{33} \end{pmatrix}$$

If we're only interested in the activities taking place in country 1 as a result of the selected final consumption of products then the subselection is:

$$\boldsymbol{X}_s = \begin{pmatrix} \boldsymbol{X}_{11} & \boldsymbol{X}_{12} & \boldsymbol{X}_{13} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

The next step is calculating the selected indicator:

$$\boldsymbol{m} = \boldsymbol{b}\,\boldsymbol{X}_s$$

where **m** is a vector of the impacts associated with each of the selected final consumed products given a certain selected region and sector where the emission takes place.

There is one last step to make. The user does not specify the geographical origin of a product. For instance if the user selects "rice" the actual product selected are "rice from Italy", "rice from Taiwan". The **m** vector with a length of 49 countries times 200 products needs to be aggregated into 200 product groups:

$$\boldsymbol{m}_{aggr} = \boldsymbol{m}\,\boldsymbol{G}$$

where **G** is an appropriate aggregation matrix.

In practice for the consumption based view the sector and region where the emission takes place is always set to all sectors and all regions. However the code allows to make sub selections.

3.4 Route 2

In route 2 the environmental impacts of final consumption is compared between countries for a selected set of products. Again environmental impacts generated by production in all countries and by the production of all products are taken into account.

The calculation route has been designed for general application. It can calculate the environmental impacts of different products given a specific selected country selling final product or specific country where the emission takes place or specific sector where the emission takes place. However the actual implementation of route 2 takes into account that country selling product are all countries, the country where the emission takes place are all countries and the emission at all sectors are taken into account.

The calculation starts by creating the final demand vector that contains the selected products for the selected countries.

The symbol **Y** represents the multi-regional final demand matrix, that can be subdivided into subvectors that contain total final demand for domestically produced products and total final demand for imported products. Assume that there are three countries:

$$\boldsymbol{Y} = \begin{pmatrix} \boldsymbol{y}_{11} & \boldsymbol{y}_{12} & \boldsymbol{y}_{13} \\ \boldsymbol{y}_{21} & \boldsymbol{y}_{22} & \boldsymbol{y}_{23} \\ \boldsymbol{y}_{31} & \boldsymbol{y}_{32} & \boldsymbol{y}_{33} \end{pmatrix}$$

And the consuming country 1 and 3 have been selected then Y first becomes:

$$\boldsymbol{Y_s} = \begin{pmatrix} \boldsymbol{y}_{11} & 0 & \boldsymbol{y}_{13} \\ \boldsymbol{y}_{21} & 0 & \boldsymbol{y}_{23} \\ \boldsymbol{y}_{31} & 0 & \boldsymbol{y}_{33} \end{pmatrix}$$

where the subscript **s** stands for selected elements.

It is important to note that the user selects a product without specifying the origin of a product. For instance, if a user selects wheat as a product of interest, in the final demand vector wheat from every origin is selected i.e. wheat from Austria, wheat from Belgium etc.

For each of the selected countries, the output from each sector (**X**) needed to produce that final demand for a country is calculated as:

$$\boldsymbol{X} = \boldsymbol{L}\,\boldsymbol{Y_s}$$

At this point it is possible to make a sub-selection from **X** to select only the output in countries and sectors that are of interest to the user. For instance if we assume a three country case **X** can be expressed as:

$$\boldsymbol{X} = \begin{pmatrix} \boldsymbol{X}_{11} & \boldsymbol{X}_{12} & \boldsymbol{X}_{13} \\ \boldsymbol{X}_{21} & \boldsymbol{X}_{22} & \boldsymbol{X}_{23} \\ \boldsymbol{X}_{31} & \boldsymbol{X}_{32} & \boldsymbol{X}_{33} \end{pmatrix}$$

If we're only interested in the activities taking place in country 1 as a result of the selected final consumption of products then the subselection is:

$$\boldsymbol{X_s} = \begin{pmatrix} \boldsymbol{X}_{11} & \boldsymbol{X}_{12} & \boldsymbol{X}_{13} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

The next step is calculating the selected indicator:

$$\boldsymbol{m} = \boldsymbol{b}\,\boldsymbol{X_s}$$

where **m** is a vector of the impacts associated with each of the countries selected final demand.

3.5 Route 3

Using this calculation route the user can compare the emissions taking place in different countries given a certain selected final demand. For instance it is possible to see in which countries emissions take place as a result of final consumption in the USA.

The calculation route has been designed for general application. It can calculate the environmental impacts of different products given a specific selected country selling final product or specific country where the emission takes place or specific sector where the emission takes place. However the actual implementation of route 3 takes into account that country selling product are all countries, the country where the emission takes place are all countries and the emission at all sectors are taken into account.

The calculation starts by creating the final demand vector that contains the selected products for the selected countries.

The **Y** symbol represents the multi-regional final demand matrix, that can be subdivided into subvectors that contain total final demand for domestically produced products and total final demand for imported products. Assume that there are three countries:

$$\boldsymbol{Y} = \begin{pmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ y_{31} & y_{32} & y_{33} \end{pmatrix}$$

And the consuming country 1 and 3 have been selected then Y first becomes:

$$\boldsymbol{Y_s} = \begin{pmatrix} y_{11} & 0 & y_{13} \\ y_{21} & 0 & y_{23} \\ y_{31} & 0 & y_{33} \end{pmatrix}$$

where the subscript **s** stands for selected elements.

It is important to note that the user selects a product without specifying the origin of a product. For instance, if a user selects wheat as a product of interest, in the final demand vector wheat from every origin is selected i.e. wheat from Austria, wheat from Belgium etc.

Following the selection of final consumed products in a selected number of countries the final demand matrix is summed to get total final demand for each product:

$$\boldsymbol{y_s} = \boldsymbol{Y_s i}$$

The output needed to satisfy this final demand is subsequently calculated according:

$$\boldsymbol{x} = \boldsymbol{L y_s}$$

The vector **x** contains all possible product outputs and has a length of 49 countries times 200 products. The emissions or impact indicators are calculated by multiplying selected emission coefficients or selected indicator coefficients with the diagonalised output vector:

$$\boldsymbol{m} = \boldsymbol{b\hat{x}}$$

The vector **m** is subsequently aggregated into emission or indicators per country

$$\boldsymbol{m_{aggr}} = \boldsymbol{m\,G}$$

where **G** is an appropriate aggregation matrix.

3.6 Route 4

Using this calculation route the user can compare the emissions associated with different product outputs given a certain selected final demand. For instance it is possible to see in which product output has the highest emissions as a result of final consumption in the USA. This calculation route starts in the same way as calculation Route 3 but the aggregation step at the end differs from route 1.

The calculation route has been designed for general application. It can calculate the environmental impacts of different products given a specific selected country selling final product or specific country where the emission takes place or specific sector where the emission takes place. However the actual implementation of route 4 takes into account that country selling product are all countries, the country where the emission takes place are all countries and the emission at all sectors are taken into account.

The calculation starts by creating the final demand vector that contains the selected products for the selected countries.

The symbol **Y** represents the multi-regional final demand matrix, that can be subdivided into subvectors that contain total final demand for domestically produced products and total final demand for imported products. Assume that there are three countries:

$$\boldsymbol{Y} = \begin{pmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ y_{31} & y_{32} & y_{33} \end{pmatrix}$$

And the consuming country 1 and 3 have been selected then Y first becomes:

$$\boldsymbol{Y_s} = \begin{pmatrix} y_{11} & 0 & y_{13} \\ y_{21} & 0 & y_{23} \\ y_{31} & 0 & y_{33} \end{pmatrix}$$

where the subscript **s** stands for selected elements.

It is important to note that the user selects a product without specifying the origin of a product. For instance, if a user selects wheat as a product of interest, in the final demand vector wheat from every origin is selected i.e. wheat from Austria, wheat from Belgium etc.

Following the selection of final consumed products in a selected number of countries the final demand matrix is summed to get total final demand for each product:

$$\boldsymbol{y_s} = \boldsymbol{Y_s i}$$

The output needed to satisfy this final demand is subsequently calculated according:

$$\boldsymbol{x} = \boldsymbol{L y_s}$$

The vector **x** contains all possible product outputs and has a length of 49 countries times 200 products. The emissions or impact indicators are calculated by multiplying selected emission coefficients or selected indicator coefficients with the diagonalised output vector:

$$\boldsymbol{m} = \boldsymbol{b\hat{x}}$$

The vector **m** is subsequently aggregated into emission or indicators per country

$$\boldsymbol{m_{aggr}} = \boldsymbol{m\,G}$$

where **G** is an appropriate aggregation matrix.

CHAPTER 4

Modelling overview

The following explanations references the ramascene.modelling module code.

4.1 General description

In order to create scenarios using EEIO data, operations need to be performed on the source data to obtain their counterfactual. Counter-factual scenarios are IO tables representing a structure of the economy different from the baseline structure. They are constructed by adjusting particular elements in the matrices composing the IO system.

In order to create counter-factual IO tables, the RaMa-SCENE platform takes as a reference A, the technical coefficient matrix, and Y, the final demand matrix. These two matrices are a representation of the relationship of input products to outputs (A) and the demand of products by final consumers (Y).

Changes in relationships in the way products are manufactured are therefore implemented by modifying the A matrix, while variations in the way products are consumed by final consumers are applied on final demand.

Once the A and Y matrix are modified through ramascene.modelling module, they are processed by ramascene.analyze module so to output results.

4.2 Settings

Through the interface, under “Scenario Modelling”, users are able to specify the coordinates of the IO elements that they wish to modify and the magnitude of the change for these intersected values.

The coordinates are defined by row-wise and column-wise items:

Row-wise:

- Product: the supply of a product;
- Originating from: the region that is supplying the product.

Column-wise:

- Consumed by: who is consuming the product, namely final consumers (Y) or a manufacturing activity (S);
- Consumed where: the region in which the consumption of the product is conducted.

The magnitude of the change by which the intersected values are modified is specified under the "Technical Change Coefficient". Here both negative and positive relative values can be specified. A negative relative value defines the reduction that need to be calculated relative to the baseline data while a positive one defines an increase.

Once a set of coordinates and a technical change coefficient are specified, users can add the change. This can be done iteratively until one is satisfied with the scenario settings. A wrong addition can also be removed through the "Remove last" button.

One the user is satisfied with the settings, they can be saved and applied by pressing the M icon on one of the previous analyses available under the "Analysis queue" menu.

4.3 Processing the settings

Through the ramascene.modelling module, each added change is processed iteratively. If any change was specified to be applied in "Y: final consumption" the final demand matrix Y is used as a reference, otherwise the software references the technical coefficient matrix (A).

The objective of these operations is to obtain a counter-factual final demand matrix $\mathbf{Y}^{alt}$ and Leontief inverse matrix $\mathbf{L}^{alt}$.

The following equation exemplifies how changes in any given matrix are applied:

$$M_{ij} \neq M_{ij}; if\ ij \in \Omega$$

Where $\mathbf{M}_{ij}$ is a value of specific coordinates in the matrix of reference M. If the coordinates ij are part of a set of pre-determined coordinates Ω, then $\mathbf{M}_{ij}$ is different from its baseline.

The selected value is then modified by following this equation:

$$M_{ij}^{alt} = M_{ij}(1 - k)$$

Where $\mathbf{M}_{ij}^{alt}$ is the modified value intersected through the coordinates and k is the technical change coefficient by which $\mathbf{M}_{ij}$ is modified.

This process is easily applied to final demand in the following way:

$$Y_{ij} \neq Y_{ij}; if\ ij \in \Omega$$

$$Y_{ij}^{alt} = Y_{ij}\,(1-k)$$

However, in order to obtain a counterfactual Leontief inverse matrix **L** ^{alt} , these changes need to be applied to the technical coefficient matrix **A**:

$$A_{ij} \neq A_{ij}; if\ ij \in \Omega$$

$$A_{ij}^{alt} = A_{ij}\,(1-k)$$

From the counter-factual **A** ^{alt} the counterfactual **L** ^{alt} is calculated by using the following equation:

$$\mathbf{L}^{\mathrm{alt}} = (\mathbf{I} - \mathbf{A}^{\mathrm{alt}})^{-1}$$

The total product output is then calculated through the IO equation employing **L** ^{alt} and **Y** ^{alt} :

$$\mathbf{x}^{\mathrm{alt}} = \mathbf{L}^{\mathrm{alt}}\,\mathbf{y}^{\mathrm{alt}}$$

CHAPTER 5

Frontend

The frontend uses the React framework. The source javascript files are stored in the folder assets/js/client. Webpack will build the distributables and places them in /assets/bundles.

5.1 Package Manager

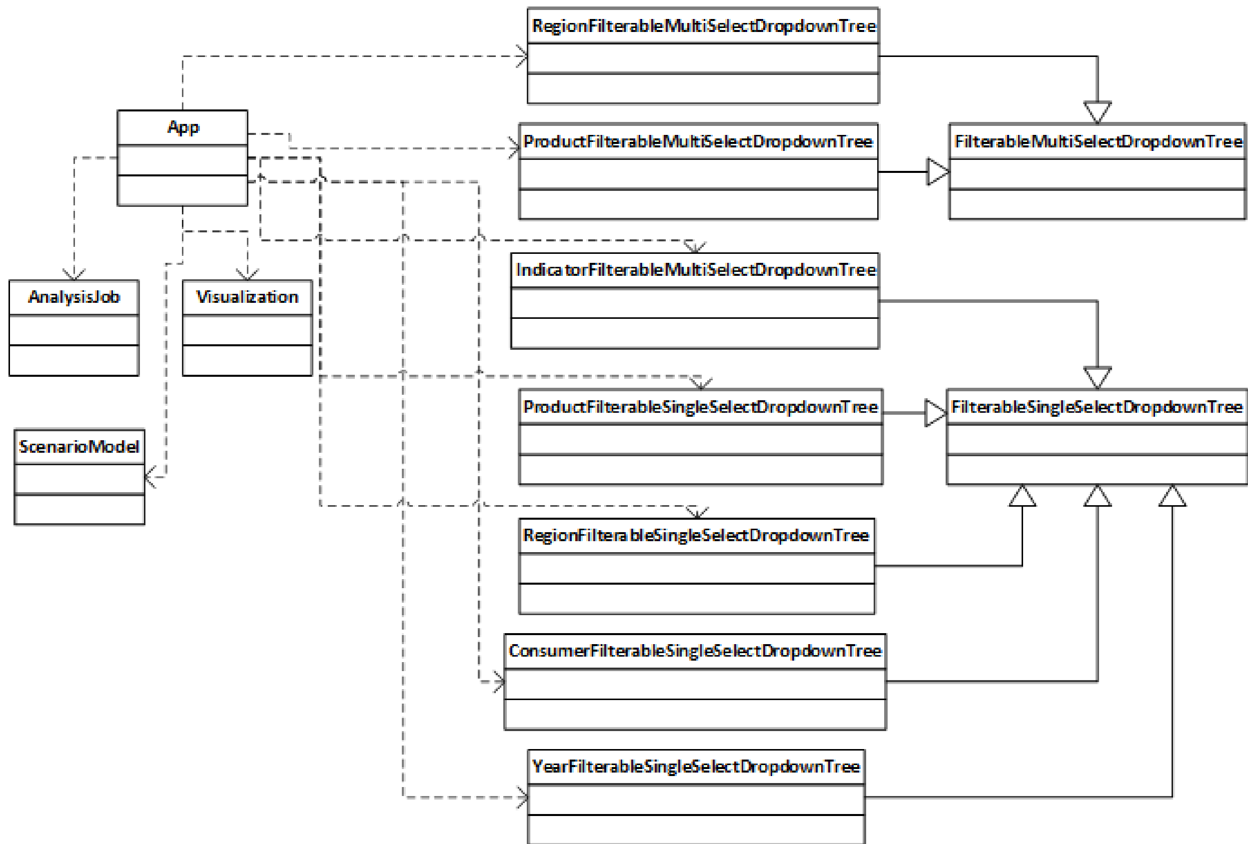
The used javascript package manager is yarn. Yarn uses package.json to create yarn.lock file.

5.2 Structure

5.2.1 The main App component

The main React component is defined in the file ramascene.js : this component defines the main page structure. The main App component uses other sub components :

- **AnalysisJob** : displayed as a job in the Analysis queue panel
- **IndicatorFilterableSingleSelectDropdownTree** : displayed as the Indicator dropdown selectbox
- **ProductFilterableSingle-** and **ProductFilterableMultiSelectDropdownTree** : displayed as the products dropdown selectbox
- **RegionFilterableSingle-** and **RegionFilterableMultiSelectDropdownTree** : displayed as the region dropdown selectbox
- **ScenarioModel** : displayed as the content for the Scenario Modelling panel
- **Visualization** : displayed as the content for the Main and Comparison View panels
- **YearFilterableSingleSelectDropdownTree** : displayed as the year dropdown selectbox



The main App component also defines following constants :

Constant	Value	Description
MAX_JOB_COUNT	10	max. number of jobs permitted in analysis queue
WAIT_INTERVAL	5000	the time a message to the user is displayed when a new job is placed on the analysis job queue

The main App component most important state variables :

- **jobs** : array of Analysis jobs
- **model_details** : array of scenario modelling changes

The main App component functions :

- **handleAnalyse()** : will push a new job on the analysis queue and shows a message for WAIT_INTERVAL time
- **handleModelling()** : will only show a message for WAIT_INTERVAL time
- **handleJobFinished()** : hides the message
- **renderVisualization()** : deselects the currently selected job, selects the new job and renders the Visualization component
- **renderComparisonVisualization()** : almost the same functionality as renderVisualization
- **hideMainView()** : empties the main view panel
- **hideComparisonView()** : empties the comparison view panel
- **deleteJob()** : deletes 1 job from analysis queue

5.2.2 The ScenarioModel component

This component makes it possible to add or remove changes to a new model_details structure and then save it to the model_details. It is also possible to clear the model_details.

The ScenarioModel component uses other sub components :

- **ProductFilterableSingleSelectDropdownTree** : displayed as the products dropdown selectbox
- **ConsumerFilterableSingleSelectDropdownTree** : displayed as the 'consumed by' dropdown selectbox
- **RegionFilterableSingleSelectDropdownTree** : displayed as the 'originating from/where' dropdown selectboxes

The ScenarioModel component defines following constants :

Constant	Value	Description
MAX_CHANGES	5	max. number of changes a scenario can contain

The main functions :

- **handleAddClick()** : pushes a new change to the new model_details structure
- **handleRemoveClick()** : pops the last change from the new model_details structure
- **handleSaveClick()** : saves the new model_details structure
- **handleClearClick()** : clears the model_details structure

5.2.3 The AnalysisJob component

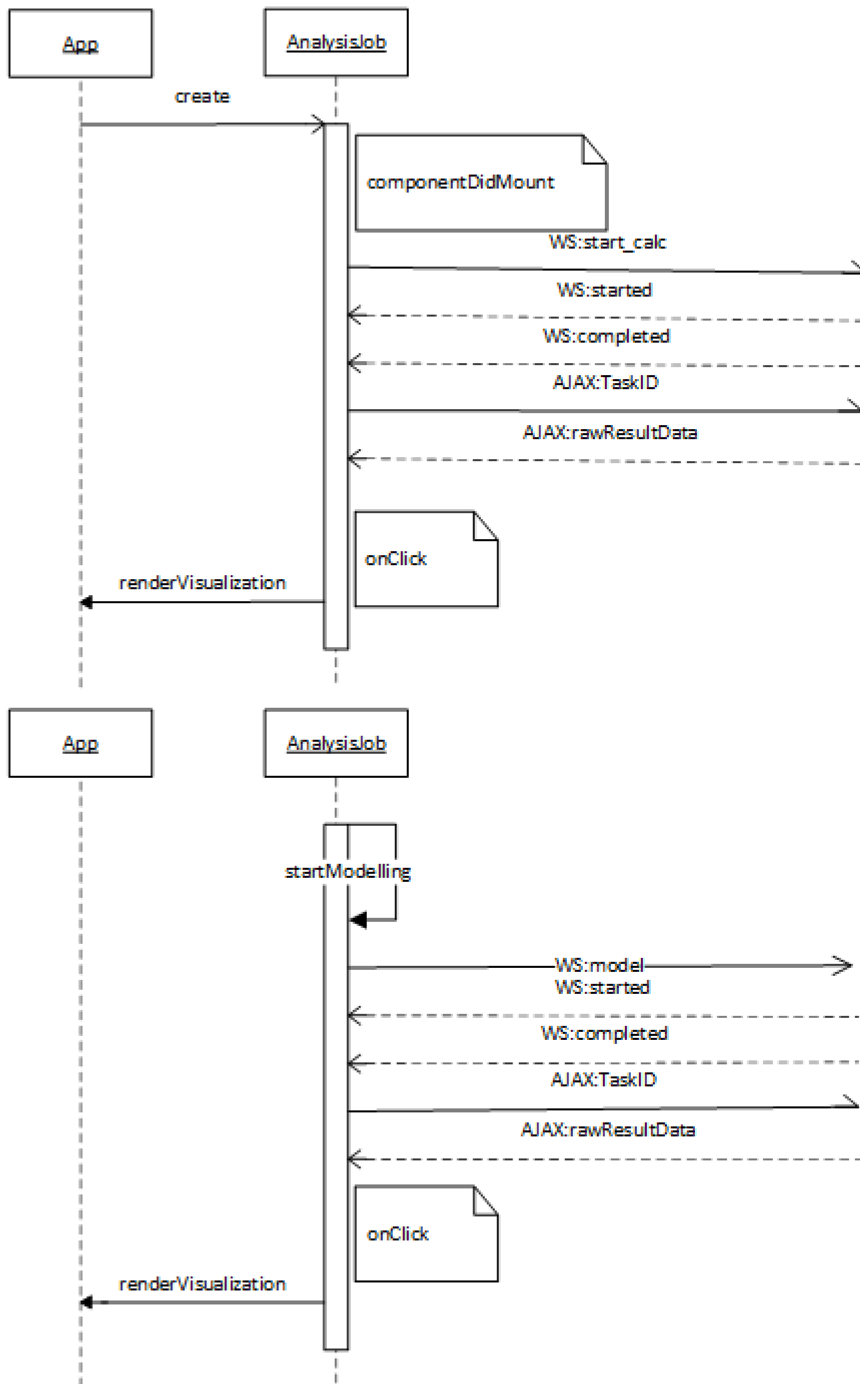
This component does all the Websocket and AJAX traffic.

The AnalysisJob component most important state variables :

- **query** : contains the query

The main AnalysisJob component functions :

- **componentDidMount()** : this is a React lifecycle callback function which is called immediately after the component is inserted into the DOM tree. This function will set up the Websocket connection and sends the analysis query to the RaMa-scene server.
- **handleWebSocketResponse()** : called when a message is received on the websocket. The received data will be parsed as JSON. When job_status == 'started' the job name can be generated. When job_status == 'completed' an AJAX call is executed to retrieve the raw result data. Upon reception of the raw results, some preparations for CSV download are executed.
- **generateCSVdata()** : prepares the CSV data for download
- **startModelling()** : will set up a Websocket connection and sends the model_details to the RaMa-scene server.



5.2.4 The Visualization component

This component will render the result using D3plus package. It uses 3 different topojson files for the geo map : one with a layer over the whole world, one with a layer per continent and one with a layer per individual country/region.

5.2.5 Dropdown Tree select

Because it seems not possible to switch on the fly a multi-select to a single-select box with the rc-tree-select package we're using, two separate base classes are provided :

- **FilterableMultiSelectDropdownTree**
- **FilterableSingleSelectDropdownTree**

The FilterableSingle- and FilterableMultiSelectDropdownTree component most important state variables :

- **data** : contains the tree data
- **value** : contains the current selection
- **callback** : callback to be executed when onChange event occurs. For FilterableMultiSelectDropdownTree it first tries to execute a derived class's handleOnChange function.

The main functions :

- **filterCaseInsensitive()** : this function will return true if the inputValue and the treeNode label are the same text when ignoring the upper- or lowercase.

Most noticeable derived classes of FilterableMultiSelectDropdownTree are **ProductFilterableMultiSelectDropdownTree** and **RegionFilterableMultiSelectDropdownTree**. The ProductFilterableMultiSelectDropdownTree overrides the render() function to add buttons for quickly select all items on one of its three tree levels. The ProductFilterableMultiSelectDropdownTree component also implements the handleOnChange super class function which keeps care that a user doesn't mix selecting items from different tree levels. The RegionFilterableMultiSelectDropdownTree also implements the handleOnChange super class function. It will take care of selecting only total/continent/country items depending on the 'Geographic aggregation level'. If Country aggregation level is chosen and a continent is selected, then it selects all countries of that continent instead. If Continent aggregation level is chosen and countries are selected, then it selects the continents to which these countries belong instead.

CHAPTER 6

Deployment

The web-application deployment process is based on the following documentations and is tested on Ubuntu 16.04 LTS:

1. http://masnun.rocks/2016/11/02/deploying-django-channels-using-daphne/
2. http://channels.readthedocs.io/en/stable/deploying.html
3. https://medium.com/@saurabhpresent/deploying-django-channels-using-supervisor-and-ngnix-2f9a25393eef
4. https://medium.com/@dwernychukjosh/setting-up-nginx-gunicorn-celery-redis-supervisor-and-postgres-with-django-to-run-you
5. https://www.vultr.com/docs/installing-and-configuring-supervisor-on-ubuntu-16-04

It is advised to read these guides. See the next sections for an example to get started quickly.

Please refer to our repository for downloading the raw data, see README.md.

6.1 Install Redis [message broker] for Django Channels websocket support

Install redis: `$ sudo apt-get install redis-server`

Before you create the virtual environment make sure you have python-dev installed via apt-get Create a virtual environment (python3.5 or higher) and install the following:

```
$ pip3 install asgi_redis
$ pip3 install -U channels_redis
```

Test redis:

`$ redis-cli ping`

Return value should be : PONG

Make sure redis is a daemon, see redis.conf.

6.2 Install Django dependencies & prepare SQLlite

In the same virtual env., change directory towards the project root: `$ pip3 install -r requirements.txt`

Make sure you set the following environment variables (see example-prod-env.sh):

- export DJANGO_SETTINGS_MODULE="ramasceneMasterProject.config.<config filename>"
- export DATASETS_VERSION="<ramascene database version available e.g. v3>"
- export DATASETS_DIR="<my/path/to/datasets>"
- export SECRET_KEY="<django secret key>"
- export BROKER_URL="<default is amqp://localhost>"
- export HOST="<ip or domain>"
- export OPENBLAS_NUM_THREADS=<adjust according to how many cores you want to use>

If you are on Linux and using the OPENBLAS library for Numpy. It is advised to set the number of threads Numpy uses. To find which library is used in python:

`>>>np.__config__.show()`

Note: For more information on the OPENBLAS_NUM_THREADS settings see Celery section further down the page.

Prepare SQLlite:

```
$ python3 manage.py makemigrations
$ python3 manage.py migrate
```

Create superuser for administration purposes:

`$ python3 manage.py createsuperuser`

6.3 Management commands and prepare static resources

Populating database classifications:

`$ python3 manage.py populateHierarchies`

Install node.js (node version: 3.10.10 or higher), if not already installed:

`$ sudo apt-get install nodejs`

Prepare static resources:

`$ npm install`

Set webpack conf settings for production:

- Configure webpack.config.js for ajax url and websocket url at webpack.DefinePlugin() to your domain.
- Adjust process environment to "production" at webpack.DefinePlugin().
- Configure Dotenv to point to your environment variables if desired. Alternatively remove dotenv section.

- Make sure that new UglifyJsPlugin() is set.

Built React bundle:

`$ ./node_modules/.bin/webpack --config webpack.config.js`

Django collect static: `$ python3 manage.py collectstatic`

6.4 Install and setup nginx [HTTP and Reverse Proxy Server]

Installing nginx requires apache to be stopped temporarily:

`$ sudo service apache2 stop`

Install nginx:

`$ sudo apt-get install nginx`

Configure nginx, make sure proxy_pass is set to this: `http://0.0.0.0:8001`

See example configuration file `example_nginx`

Check status of nginx: `$ sudo nginx -t`

Allow Nginx to interact with the host machine on the network: `$ sudo ufw allow 'Nginx Full'`

6.5 Celery details and setup

Celery is used to delegate long lasting CPU jobs and heavy memory usage for performing IO calculations on the fly. In this project the message broker rabbitMQ is used. Each user performing a request for calculation is set in the queue and that task is handled when ready by the Celery consumer.

Installing the rabbitMQ broker:

`$ sudo apt-get install -y erlang`

`$ sudo apt-get install rabbitmq-server`

Then enable and start the RabbitMQ service:

`$ systemctl enable rabbitmq-server`

`$ systemctl start rabbitmq-server`

Check the status to make sure everything is running: `$ systemctl status rabbitmq-server`

Celery details:

Each Celery worker spawns a number of child processes and these processes use as much memory as it needs. The first limit to set is the concurrency. It is normally advised to run a single worker per machine and the concurrency value will define how many processes will run in parallel. Concurrency set to 1 follows a first in first out principle for users, if concurrency is increased the server's resources (CPU and MEM) are more extensively used and Celery could handle requests simultaneously. For the RaMa-Scene app one single worker for default calculations and a dedicated worker for modeling final demand is advised, due to the nature of computation extensive modelling. In addition it is recommended to set the concurrency to 1, if increased it is advised to perform load testing.

Setting a Celery MEM limit:

Loading numpy objects over different years can causes severe memory use if Python doesn't release memory after a calculation is finished. The common idea is that Python does garbage collection and frees up memory once finished. However during testing it became apparent that memory wasn't released, refer to https://github.com/celery/celery/

issues/3339. The setting implemented in the Django settings.py is a limit on the number of task handled per child process. If set to 1 a new worker has to be spawned if a tasks is finished, enforcing the release of memory.

Setting a Numpy limit:

Most linux machines use the OPENBLAS library for numpy. OPENBLAS uses all cores available for performing calculations by default. By setting the OPENBLAS_NUM_THREADS it is possible to limit the amount of cores used, leaving resources available on the server.

Note: For more information on Celery refer to the performance page in this documentation and the official celery docs.

6.6 Testing the application

Make sure Daphne is installed and start daphne (in virtualenv):

`$ daphne -b 0.0.0.0 -p 8001 ramasceneMasterProject.asgi:application`

Start the Celery workers in virtual env.:

`$ celery -A ramasceneMasterProject worker -l info --concurrency 1 --queue calc_default -n worker1.%h`

`$ celery -A ramasceneMasterProject worker -l info --concurrency 1 --queue modelling -n worker2.%h` Be careful with load if you raise concurrency. For final production setup remove the parameter -l info.

Test the application to see if everything is running correct in a web-browser.

6.7 Daemonizing

Celery and Daphne need to be deamonized. For example with supervisor. Bare in mind that the environment variables have to be set in the configuration file. See example configuration file `example_supervisord`

If you make changes to the file you have to do:

- sudo supervisorctl reread
- sudo supervisorctl update

If you want to stop or start processes:

- sudo supervisorctl stop <program name e.g. celeryd>
- sudo supervisorctl start <program name e.g. celeryd>

6.8 Management of database results

Cron can be used to clear the database results on a regular basis, see example below:

#at 5 a.m on every sunday `0 5 * * 0`

#delete database contents `. <path to environment>/env.sh && cd /<proj>/ && /<virtual-env>/bin/python /<proj>/manage.py clear_models`

CHAPTER 7

Python initialise scripts

Python initialise scripts provide the following features:

1. **Management commands**
2. **Creating EXIOBASE numpy objects**
3. **Building mapping coordinates files for the application**
4. **Creating custom geojson and topojson files**

Note: the scripts under point 2, 3 and 4 are not directly used by the application at runtime. If you wish to extent or develop features such as using another EEIO dataset it is recommended to investigate how these script work, alternatively these points can be skipped. Deployment management commands are used for populating the database and are needed for deployment.

7.1 Management commands

This application uses a sql-lite database to store mapping coordinates, user queries (Jobs) and user results (Celery results). For mapping coordinates the database needs to be populated before running the application.

The management commands uses CSV files generated by `prepare_csv.py`:

- mod_final_countryTree_exiovisuals.csv
- mod_final_productTree_exiovisuals.csv

The following command is used to populate the database `python manage.py populateHierarchies` populates the database with the mapping files needed.

Aside from database population, it is advised to clean the database of user results whenever needed. This can be done with the following command `python manage.py clear_models`

Note : see project folder ramascene/management/commands

7.2 Creating EXIOBASE numpy objects

Originally EXIOBASE data is structured in tabulated text file format. This application uses the python numpy library to perform calculations and therefore the original EXIOBASE data is converted to numpy objects. Two versions of numpy formatted EXIOBASE data are created:

- v3: an unmodified version. Note that the unmodified version is constructed in such a way that the Rama-Scene calculation procedure including indicators works efficiently as opposed to the default EXIOBASE tabular format.
- v4: a modified version including secondary materials

The original script to create version 3 is located at

python_ini/devScripts/script/create_numpy_objects_v3.py

The script expects the following folder structure:

- script/create_numpy_objects_v3.py -> the actual conversion script
- data/auxiliary -> auxiliary information to determine indicators in the application
- data/clean/<year> -> a folder that contains the years reserved for output (not in the source code, because of the large amount of data)
- data/raw/<year> -> a folder that contains the original txt files per year (not in the source code, because of the large amount of data)

For version 4 please refer to https://bitbucket.org/CML-IE/pysuttoio/src/master/

Note : see project folder python_ini/devScripts/script/create_numpy_objects_v3.py and https://bitbucket.org/CML-IE/pysuttoio/src/master/

7.3 Building mapping coordinates files for the application

EXIOBASE v3.3 has specific classifications that needs to map to user input. For example if Europe is selected by the user as one of the parameters for analyses, the calculation procedure uses indices corresponding to all countries belonging to Europe. In turn these calculation results need to be aggregated back into a single value for Europe. For coordinating user input to calculation procedures we developed the following mapping CSV's based on an older application called ExioVisuals :

- final_countryTree_exiovisuals.csv
- final_productTree_exiovisuals.csv

These files are read in by the script `prepare_csv.py` that in turn makes a slight modification to easily denote aggregated/disaggregated countries or product categories. The output files of `prepare_csv.py` are prefixed with *mod_<filename>* and is used by one of the management commands.

Note : see project folder python_ini/data & python_ini/devScripts.

7.4 Creating custom geojson and topojson files

For visualizations of continental data custom polygons need to be created that reflect the country mapping of EXIOBASE. A script is developed to dissolve countries that belong to a certain continent or "rest of " classification with the library GeoPandas.

Relevant scripts: `geo_dissolve_by_level.py`, `prepare_geomapping_ISO3166_3.py`, `prepare_geomapping_ISO3166_2.py`

1. First the *mod_final_countryTree_exiovisuals.csv* (modified by prepare_csv script above) is further adjusted to contain the 3-letter ISO code available in the DESIRE country list excel.The default modified file only contains 2-letter codes and this cannot be used by the script needed to create polygon files.Changing 2-letter to 3-letter codes is done manually by using the DESIRE country list, the new file is called *mod_final_countryTree_ISO3166_3.csv*.
2. Run prepare_geomapping_ISO3166_3.py with *mod_final_countryTree_ISO3166_3.csv*
3. After step 1 and 2, create a virtual environment and install the requirements.txt in the python_ini/geoMapbuilds to enable the use of GeoPandas.
4. The script geo_dissolve_by_level.py is used to create 3 geojson files, 1) a file for the whole world, 2) a file for continents, 3) a file for countries and rest of regions. This is needed for the visualization library d3plus. Before running the script, please select which of the three files you wish to generate by adjusting the SETTING variable.
5. The generated files are fairly big, in turn it is key to make these small smaller by converting them to topojson and simplify the polygons using MapShaper.org. This is achieved with 1) importing the files into mapshaper, 2) clicking on "simplify", check "prevent shape removal" and finally set the percentage to 5% 3) finally export to topojson with command "id-field=id" "drop-table"

Note : see project folder python_ini/geomapBuilds and python_ini/devScripts

CHAPTER 8

Testing

Various tests are implemented for websocket communication, celery background processing, AJAX, views, and the models. See ramascene/tests/ folder.

8.1 Unit testing

Perform unit tests in the root folder:

```
$ python3 manage.py test -v2
```

8.2 Integration test

A more extensive validation tests is performed with pytest. Several validation files (CSV) are prepared from results computed outside of the web-application. Please refer to ramascene/tests/validation_files for the structure of these files. Each file contains information to generate a query, send a websocket query, receive results back from Celery. These results are in turn matched against the validation files expected results with a given tolerance. Lastly refer to confttest.py to see which scripts are called for performing the test.

Please be aware that at the moment only EXIOBASE v3 can be tested this way

To test over the full life cycle of the back-end you can run the following command in the root folder:

```
$ pytest -vs
```

Make sure to run a celery worker:

```
$ celery -A ramasceneMasterProject worker -l info --concurrency 1 --queue
calc_default -n worker1.%h
```

If the test has succeeded, you'll need to repopulate the database with the following command:

```
$ python3 manage.py populateHierarchies
```

CHAPTER 9

Performance

9.1 Specs of tested server

- Brand: Dell PowerEdge R320
- CPU: Intel Xeon E5-1410 v2 @ 2.80 GHz
- Memory: 8 GiB DIMM DDR3 Synchronous 1600 MHz
- Number of cores: 8

9.2 Load test setup

9.2.1 websocket-tester

A custom websocket-tester is developed to simulate queries to the server. Multiple request are send using a simple loop. For more information on the exact queries see description on testing scenarios further down this page.

9.2.2 Celery flower

For assessing performance of calculations the library Celery Flower is used. Refer to the official docs for more information. To setup Flower on a remote server with Django you have to use the proxy server. For safety setup a htpasswd file in the nginx configuration. See example configuration here for nginx with Celery Flower (adjust to your needs) `example_nginx_flower`

For RabbitMQ the management plugin has to be enabled: `$ sudo rabbitmq-plugins enable rabbitmq_management`

An example to start flower (make sure you are in the project root):

$ flower -A ramasceneMasterProject –port=5555 -Q modelling,calc_default –broker=amqp://guest:guest@localhost:5672// –broker_api=http://guest:guest@localhost:15672/api/ –url_prefix=flower

In turn you can access flower via the web browser with <domain>/flower/.

9.2.3 Settings for Celery

The following settings are in place:

- [Django settings] CELERY_WORKER_MAX_TASKS_PER_CHILD = 1
- [env. variable] OPENBLAS_NUM_THREADS=2 for default calculations
- [env. variable] OPENBLAS_NUM_THREADS=5 for modelling calculations
- [Celery] –concurrency 1 for default calculations
- [Celery] –concurrency 1 for modelling calculations

Each Celery queue has its own dedicated worker (1 worker for default and 1 worker for modelling)

9.2.4 Testing scenarios and simultaneous requests

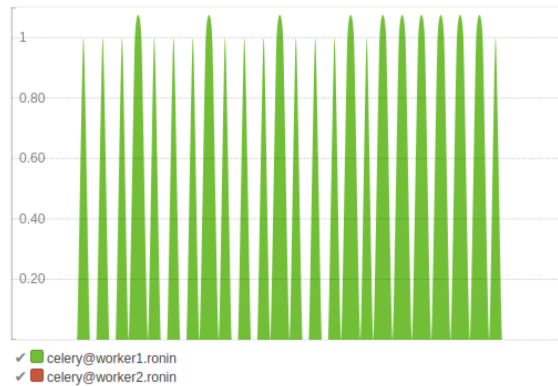
The longest calculation route is the one with the selections TreeMap and Consumption view. For simplicity we use "value added" as the indicator coupled with "total" for product and countries. All scenarios use this selection. For modelling we select "totals" for all categories except "consumed by" which contains the "S: Agriculture, hunting and forestry" aggregate. A single technical change is set to an arbitrary value of 100.

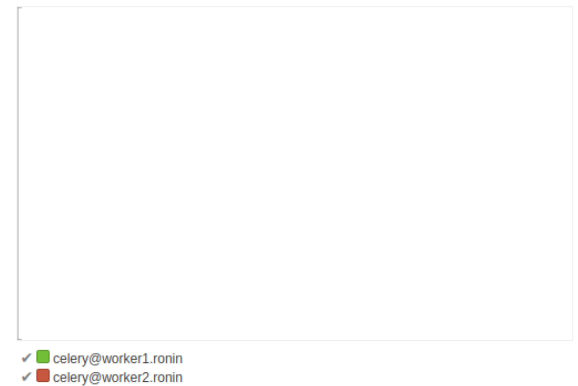
- Scenario A [Analytical]: 30 analytical request whereby 17 requests cover all years. and 13 request use the year 2011.
- Scenario B [Modelling]: 30 modelling request whereby 17 requests cover all years. and 13 request use the year 2011. All requests do heavy calculations covering the modelling of intermediates.
- Scenario C [Analytical + Modelling]: 15 requests over 15 different years for analytical and 15 request over 15 different years for modelling.

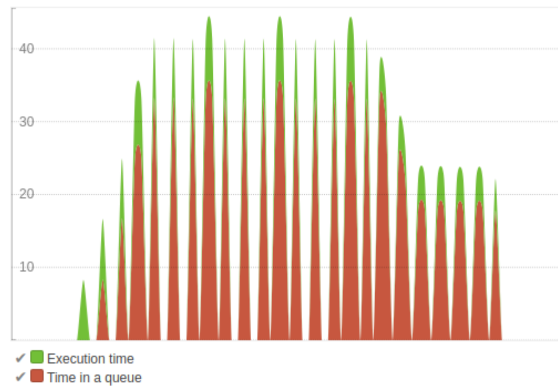
Idle MEM use at point before load test: 572M

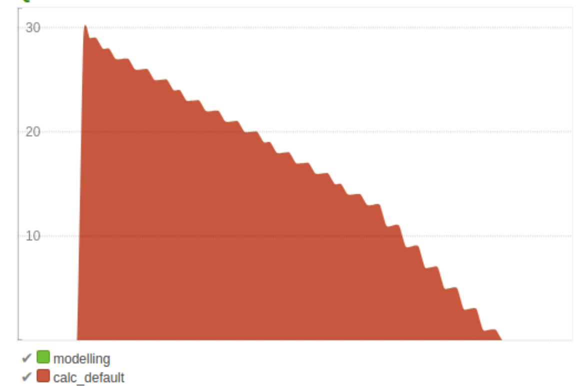
9.2.5 Results scenario A

- Max. time for a given task: 8.3 sec.
- Total time for the last user to finish the task: 4 min. and 58 sec.
- Highest detected MEM load: 2.87G (includes the idle MEM)







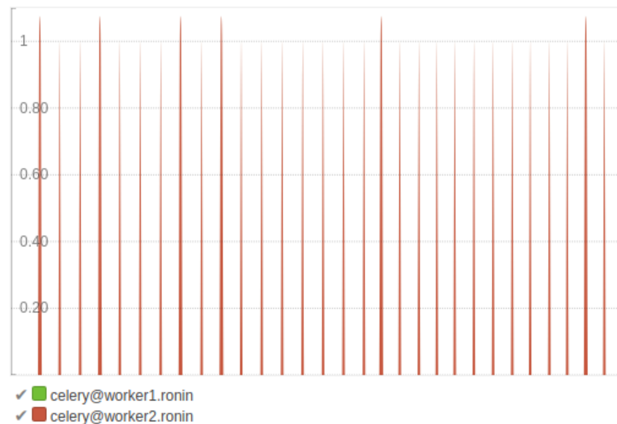


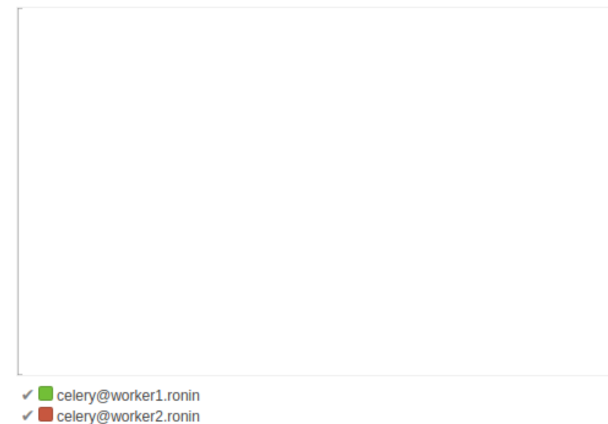
Conclusion:

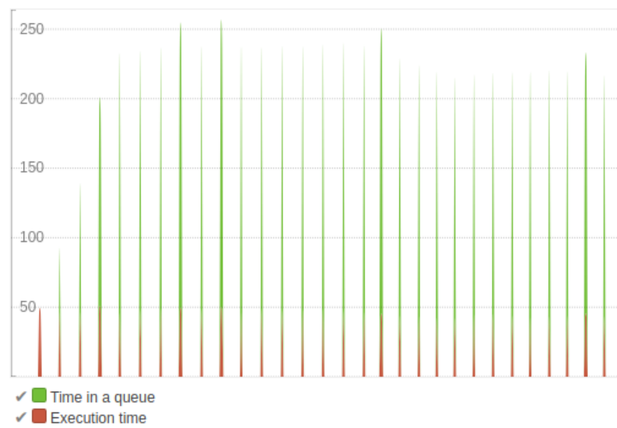
The queued task in the right bottom plot show expected behaviour due to the concurrency set to 1. The time in queue for a given task is relatively long compared to the time take to do calculations. This was expected as the CPU use is limited coupled with no simultaneous requests.

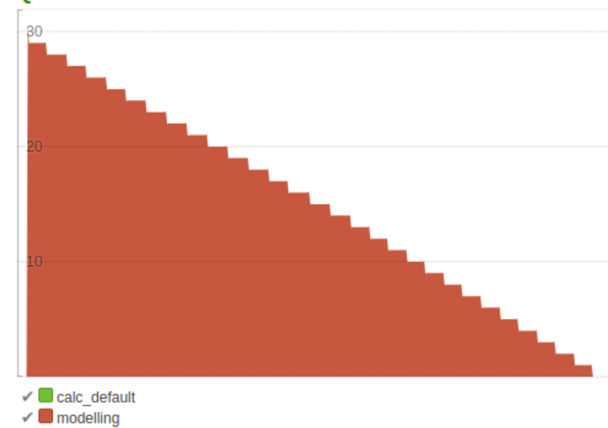
9.2.6 Results scenario B

- Max. time for a given task: 48.97 sec.
- Total time for the last user to finish the task: 22 min. 59 sec.
- Highest detected MEM load: 3.49G (includes the idle MEM)
- Execution time of the first task: 46.09 sec.







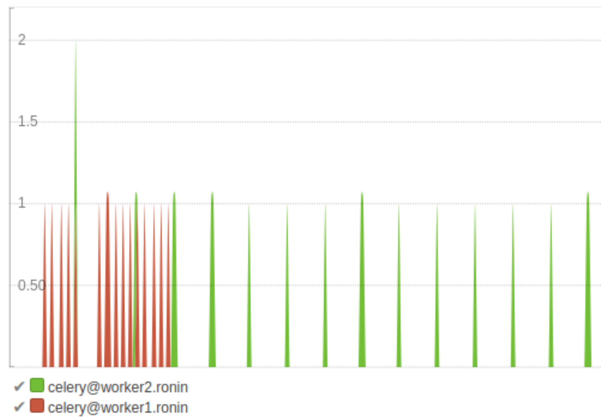


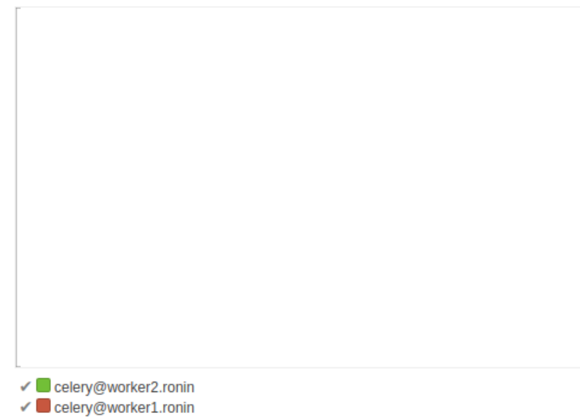
Conclusion:

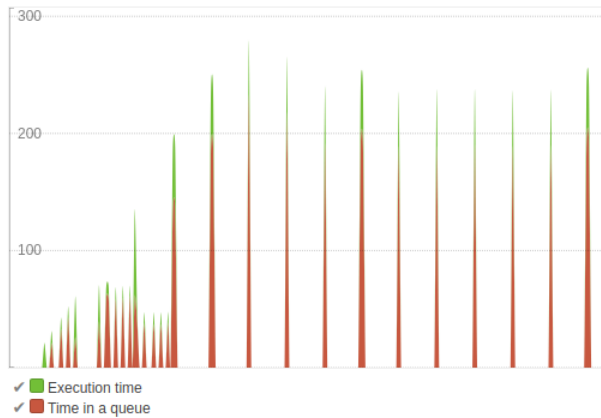
CPU use is less limited for modelling and it can use 5 cores if needed, however that only speeds up execution time. The last user still has to wait considerable time as opposed to the analytical queries. The spikes in the two plots on the left show that there are no concurrent requests handled as set in the settings.

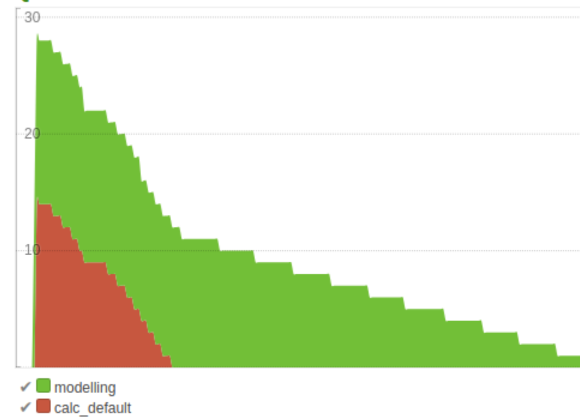
9.2.7 Results scenario C

- Max. time for a given analytical task: 9.48 sec.
- Total time for the last user to finish the task for analytics: 3 min. 28 sec.
- Max. time for a given modelling task: 49.36 sec.
- Total time for the last user to finish the task for modelling: 11 min. 55 sec.
- Highest detected MEM load: 6.44G (includes the idle MEM)









Conclusion:

As shown in the top left and bottom right graph both workers are active. The analytical queue depletes faster than the modelling queue, which is also expected and desired behaviour. The MEM load has increased as both workers use memory.

9.2.8 Final conclusion

Modelling has a significant impact on CPU use, in turn a limit is set on CPU considering the specs of the tested server. This limit results in relatively long waiting time for users doing modelling. To circumvent this either a server with more powerful specs is required or celery can be configured with workers on different machines. In both cases more CPU is required and optimally more memory. If more memory is in place, logically concurrency can be increased however new load tests have to be performed.

CHAPTER 10

ramascene

10.1 ramascene package

10.1.1 Submodules

10.1.2 ramascene.analyze module

class ramascene.analyze.**Analyze**(*product_calc_indices*, *country_calc_indices*, *indicator_calc_indices*, *querySelection*, *idx_units*, *job_name*, *job_id*, *s_country_idx*, *Y_data*, *B_data*, *L_data*)

Bases: `object`

This class contains the method for calculations

route_four()

Perform calculations according to route four.

Returns json result data

Return type json

route_one()

Perform calculations according to route one.

Returns json result data

Return type json

route_three()

Perform calculations according to route three.

Returns json result data

Return type json

route_two()

Perform calculations according to route two.

Returns json result data

Return type json

10.1.3 ramascene.consumers module

class ramascene.consumers.**RamasceneConsumer**(*scope*)

Bases: channels.generic.websocket.JsonWebsocketConsumer

This class represents the Django Channels web socket interface functionality.

celery_message(*event*)
Sends Celery task status.

save_job(*job_name*)
Update and save the job status to started

websocket_connect(*event*)
websocket first connection, accept immediately

websocket_disconnect(*message*)
Websocket disconnect function.

websocket_receive(*event*)
Receives message from front-end.

Tries to parse the message, if successful it will perform pre-processing steps and invokes Celery tasks.

Parameters **event** (dict) – message from front-end

ws_response(*job*)
Sends web socket response that the job is started

10.1.4 ramascene.modelling module

class ramascene.modelling.**Modelling**(*ready_model_details*, *Y_data*, *load_A*, *year*, *model_details*)

Bases: object

This class contains the methods for modeling

apply_model()

model_final_demand(*Y*, *rows*, *columns*, *tech_change*)
It allows for modification of values within final demand for scenario building

model_intermediates(*A*, *rows*, *columns*, *tech_change*)
It allows for modification of values within intermediates for scenario building

unpack(*structure*, *name*)
Unpack deep structure of modelling details (these are arrays of local ids per intervention)

10.1.5 ramascene.models module

class ramascene.models.**Country**(**args*, ***kwargs*)

Bases: django.db.models.base.Model

Country model to store identifiers for the countries and aggregations

exception DoesNotExist
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

code
The country code

global_id
The global id representing the application coordinates as primary id

id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

identifier
an identifier determining if it is a leaf node or aggregate

leaf_children_global
the global id's of the leafs for this continent (if available)

leaf_children_local
the local id's of the leafs of this country (if available)

level
The level of hierarchy this country is in

local_id
The local id, only used if the hierarchy level is the lowest

name
The name of the country

objects = <django.db.models.manager.Manager object>

parent_id
The id representing what parent this country belongs to (by parent global_id)

class `ramascene.models.`**Indicator**(**args*, ***kwargs*)
Bases: `django.db.models.base.Model`

Indicator model to store identifiers for indicators

exception DoesNotExist
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

global_id
The global id representing the application coordinates as primary id

id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

level
The level of hierarchy this indicator is in

local_id
The local id (unusedas there are no direct summing steps performed for the extensions)

name
The name of the indicator

objects = <django.db.models.manager.Manager object>

parent_id
The id representing what parent this indicator belongs to (unused as there are no direct summing steps performed for the extensions)

unit
The unit used for the indicator

class ramascene.models.**Job**(**args*, ***kwargs*)
Bases: `django.db.models.base.Model`

Job model to store Celery jobs

exception DoesNotExist
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

celery_id
The unique identifier for retrieving the results of the job from Celery

completed
The date the Celery job was completed

created
The date the Celery job was created

get_next_by_created(**, field=<django.db.models.fields.DateTimeField: created>, is_next=True, **kwargs*)

get_previous_by_created(**, field=<django.db.models.fields.DateTimeField: created>, is_next=False, **kwargs*)

id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

name
The name of the Celery job

objects = <django.db.models.manager.Manager object>

status
The status of the Celery job

class ramascene.models.**ModellingProduct**(**args*, ***kwargs*)
Bases: `django.db.models.base.Model`

Modelling data-model to store identifiers for the products and aggregations (slight modified version of Product)

exception DoesNotExist
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

code
The product category code

global_id
The global id representing the application coordinates as primary id

id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

identifier
an identifier determining if it is a leaf node or aggregate

leaf_children_global
the global id's of the leafs for this product group (if available)

leaf_children_local
the local id's of the leafs of this product group (if available)

level
The level of hierarchy this product is in

local_id
The local id, only used if the hierarchy level is the lowest

name
The name of the product category

objects = <django.db.models.manager.Manager object>

parent_id
The id representing what parent this product belongs to (by parent global_id)

class `ramascene.models.`**Product**(**args*, ***kwargs*)
Bases: `django.db.models.base.Model`

Product model to store identifiers for the products and aggregations

exception DoesNotExist
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

code
The product category code

global_id
The global id representing the application coordinates as primary id

id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

identifier
an identifier determining if it is a leaf node or aggregate

leaf_children_global
the global id's of the leafs for this product group (if available)

leaf_children_local
the local id's of the leafs of this product group (if available)

level
The level of hierarchy this product is in

local_id
The local id, only used if the hierarchy level is the lowest

name
The name of the product category

objects = <django.db.models.manager.Manager object>

parent_id
The id representing what parent this product belongs to (by parent global_id)

10.1.6 ramascene.productindexmanger module

class ramascene.productindexmanger.**ProductIndexManager**(*c_prd_ids*, *s_cntr_ids*, *p_prd_ids*, *p_cntr_ids*)

Bases: object

The ProductIndexManager provides functions to translate ids of selected countries and products into ids of the combination of selected countries and products and returns ids that cab be used directly to select columns and/or rows from final demand matrices, extension matrices and leontief inverse matrix.

The selected countries and products have to be supplied when initializing the ProductIndexManager. After initialisation no changes to the supplied ids are allowed.

Parameters

- **c_prd_ids** – ndarray 1D array containing integers indicating the ids of selected consumed products
- **s_cntr_ids** – ndarray 1D array containing integers indicating the ids of the selected countries selling final products
- **p_prd_ids** – ndarray 1D array containing integers indicating the ids of the selected produced products
- **p_cntr_ids** – 1D array containing integers indicating the ids of the selected producing countries

get_consumed_product_ids()
Get the ids of the selected consumed products

Based on the ids of the selected consumed products and the ids of the selected countries selling final products the ids of all selected products in the final demand vector are generated. It allows to make a full selection of the selected consumed products from a final demand vector. A full selection means that if the id of the product bread was selected, now the ids of bread from Italy, bread from Belgium etc are generated as long as Italy, Belgium etc are within the ids of countries selling final products. The ids are zero based.

Returns one dimensional numpy array with ids of type int

get_country_count()
Get the number of countries and regions in EXIOBASE

Returns integer object with the number of countries/regions.

get_full_selected_c_product_count()
Get the full number of consuming products selected.

Products in different countries are counted as unique items, i.e. bread from Belgium and bread from Italy consumed in a particular country are considered two items.

Returns integer object with the full count of consumed products selected.

get_full_selected_p_product_count()
Get the full number of produced products selected.

Products in different countries are counted as unique items, i.e. cars produced in France and cars produced in Germany are considered two items.

Returns integer object with the full count of produced products selected.

get_produced_product_ids()
Get the ids of the selected produced products

Based on the ids of the selected produced products and the ids of the selected producing countries the ids of all selected produced products are generated. It allows to make a full selection of the selected produced products from the output vector. A full selection means that if the id of the product car was selected, now the ids of car from Germany, car from France etc are generated as long as Germany, France etc are within the ids of selected producing countries. The ids are zero based.

Returns one dimensional numpy array with ids of type int

get_product_count()
Get the number of products per country in EXIOBASE

Returns integer object with the number of products.

get_selected_c_product_count()
Get the number of consumed products selected.

Returns integer object with the number of consumed products selected.

get_selected_p_country_count()
Get the number of selected producing countries.

Returns integer object with the number of producing countries selected.

get_selected_p_product_count()
Get the number of selected produced products.

Returns integer object with the number of produced products selected.

get_selected_s_country_count()
Get the number of selected countries selling final products.

Returns integer object with the number of contries selling final products selected.

10.1.7 ramascene.querymanagement module

ramascene.querymanagement.**clean_indicators**(*idx_lst*)
Clean data as preprocessing step for calculation.

Clean the selected indicator by converting to integers and applying offset of -1.

Parameters **idx_lst** (*list*) – indicators

Returns indicators(processed)

Return type list

ramascene.querymanagement.**clean_local_leafs**(*a_list*)
Clean data as preprocessing step for calculation.

Clean the country or product data for calculations by splitting and converting to integers.

Parameters **a_list** (*str*) – country or product string of coordinates separated by #

Returns country or product list of coordinates as integers

Return type list

ramascene.querymanagement.**clean_single_leafs**(*leaf*, *OFFSET*)
Clean data as preprocessing step for calculation.

Clean the country or product data for calculations by splitting, applying offset (-1) and converting to integers.

Parameters **leaf** (*str*) – single country or product coordinate (non-processed)

Returns country or product list of coordinates (single element, processed)

Return type list

ramascene.querymanagement.**convert_to_numpy**(*list_obj*)
Clean data as preprocessing step for calculation.

Convert processed country,product, indicator lists to numpy array.

Parameters **product,country,indicator** (*list*) – pre-processed list

Returns numpy arrays of products or countries or indicator coordinates

Return type list

ramascene.querymanagement.**get_aggregations_countries**(*querySelection*, *result_data*)
Sum to construct aggregates results for countries.

Invoked at Celery tasks to sum values that belong to a certain aggregate.

Parameters

- **querySelection** (*dict*) – original querySelection from user
- **result_data** (*dict*) – dictionary of result_data from calculation

Returns dicitonary of result_data, but with aggregations if there are any

Return type dict

ramascene.querymanagement.**get_aggregations_products**(*querySelection*, *result_data*)
Sum to construct aggregates results for products.

Invoked at Celery tasks to sum values that belong to a certain aggregate.

Parameters

- **querySelection** (*dict*) – original querySelection from user
- **result_data** (*dict*) – dictionary of result_data from calculation

Returns dicitonary of result_data, but with aggregations if there are any

Return type dict

ramascene.querymanagement.**get_calc_names_country**(*country_result_data*)
Get name of countries.

Uses the database/model to fetch names, used inside calculation as conversion step

Parameters **country_result_data** (*dict*) – key/value pair product with key as global_id

Returns key/value pair country with key as name corresponding to querySelection global_id

Return type dict

ramascene.querymanagement.**get_calc_names_product**(*prod_result_data*)
Get name of products.

Uses the database/model to fetch names, used inside calculation as conversion step

Parameters **prod_result_data** (*dict*) – key/value pair product with key as global_id

Returns key/value pair product with key as name corresponding to querySelection global_id

Return type dict

ramascene.querymanagement.**get_indicator_units**(*idx_lst*)
Get units of passed-in indicators.

Can be multiple or single units depending on the API implementation version

Parameters **idx_lst** (*list*) – indicators

Returns key/value pair name of indicator and corresponding unit

Return type dict

ramascene.querymanagement.**get_leafs_country**(*country_global_ids*)
Returns the leaf nodes of a given global id

Uses the database/model to fetch leaf nodes.

Parameters **country_global_ids** (*list*) – A list of user selected country global ids

Returns complete list of leaf ids (minus a offset of -1 for calculation purposes)

Return type list

ramascene.querymanagement.**get_leafs_modelled_product**(*product_global_ids*)
Returns the leaf nodes of a given global id

Uses the database/model to fetch leaf nodes.

Parameters **product_global_ids** (*list*) – A list of user selected product global ids

Returns complete list of leaf ids (minus a offset of -1 for calculation purposes)

Return type list

ramascene.querymanagement.**get_leafs_product**(*product_global_ids*)
Returns the leaf nodes of a given global id

Uses the database/model to fetch leaf nodes.

Parameters **product_global_ids** (*list*) – A list of user selected product global ids

Returns complete list of leaf ids (minus a offset of -1 for calculation purposes)

Return type list

ramascene.querymanagement.**get_modelled_names_product**(*prod_ids*)
Get name of products consumed by in modelling

Uses the database/model to fetch names, used for sending selection information to front-end

Parameters **prod_ids** (*list*) – list of products by global id

Returns lists of products

Return type list

`ramascene.querymanagement.`**`get_names_country`**(*country_ids*)
Get name of countries

Uses the database/model to fetch names, used for sending selection information to front-end

Parameters **`country_ids`** (*`list`*) – list of countries by global id

Returns lists of countries

Return type list

`ramascene.querymanagement.`**`get_names_indicator`**(*indicator_ids*)
Get name of indicators

Uses the database/model to fetch names, used for sending selection information to front-end

Parameters **`indicator_ids`** (*`list`*) – list of indicators by global id

Returns lists of indicators as names

Return type list

`ramascene.querymanagement.`**`get_names_product`**(*prod_ids*)
Get name of products

Uses the database/model to fetch names, used for sending selection information to front-end

Parameters **`prod_ids`** (*`list`*) – list of products by global id

Returns lists of products

Return type list

`ramascene.querymanagement.`**`get_numpy_objects`**(*year*, *object_name*)

Retrieve numpy objects per year.

Args: year (int): selected year object_name (str): L, A, B, or Y

Returns: numpy object: numpy object of the given object_name

`ramascene.querymanagement.`**`identify_country`**(*country_id*)
Helper function.

Does database check on countries if the global_id the user selected is an aggregate or not

Parameters **`country_id`** (*`int`*) – global id

Returns identifier e.g. LEAF or AGG or TOTAL

Return type str

`ramascene.querymanagement.`**`identify_modelling_product`**(*prod_id*)
Helper function.

Does database check on products if the global_id the user selected is an aggregate or not

Parameters **`prod_id`** (*`int`*) – global id

Returns identifier e.g. LEAF or AGG or TOTAL

Return type str

`ramascene.querymanagement.`**`identify_product`**(*prod_id*)
Helper function.

Does database check on products if the global_id the user selected is an aggregate or not

Parameters **prod_id** (*int*) – global id

Returns identifier e.g. LEAF or AGG or TOTAL

Return type str

10.1.8 ramascene.tasks module

ramascene.tasks.**async_send**(*channel_name*, *job*)

Send job message to front-end.

uses the channel_name and Job object. Send success or failure status.

Parameters

- **channel_name** (*object*) – websocket channel name
- **job** (*object*) – model object of the job

ramascene.tasks.**default_handler**(*job_name*, *job_id*, *channel_name*, *ready_query_selection*, *query_selection*)

invokes Celery function.

Handler for invoking Celery method.

Parameters

- **job_name** (*str*) – the name of the job
- **job_id** (*int*) – the id of the job
- **channel_name** (*object*) – the websocket channel name
- **ready_query_selection** (*dict*) – the query_selection preprocessed (only needs convertion to numpy array)
- **query_selection** (*dict*) – the original query_selection used for aggregations at later stage

ramascene.tasks.**handle_complete**(*job_id*, *channel_name*, *celery_id*)

Handle a successful Celery Task.

ramascene.tasks.**job_update**(*job_id*)

Update job status to completion.

Update the job status by reference job id. :param job_id: job id :type job_id: int

10.1.9 ramascene.views module

ramascene.views.**ajaxHandling**(*request*)

AJAX handler.

Checks if the request is a post. Uses from the request the task/job id to fetch the Celery unique identifier. In turn it retrieves by using the Celery unique identifier the actual results

Parameters **object** – request

Returns JSON response of result calculation

ramascene.views.**home**(*request*)

Home page.

10.1.10 Module contents

`ramascene.`**`activate_foreign_keys`**(*sender*, *connection*, ***kwargs*)

Python Module Index

r

ramascene, 56
ramascene.analyze, 45
ramascene.consumers, 46
ramascene.modelling, 46
ramascene.models, 46
ramascene.productindexmanger, 50
ramascene.querymanagement, 51
ramascene.tasks, 55
ramascene.views, 55

Index

A

activate_foreign_keys() (in module ramascene), 56
ajaxHandling() (in module ramascene.views), 55
Analyze (class in ramascene.analyze), 45
apply_model() (ramascene.modelling.Modelling method), 46
async_send() (in module ramascene.tasks), 55

C

celery_id (ramascene.models.Job attribute), 48
celery_message() (ramascene.consumers.RamasceneConsumer method), 46
clean_indicators() (in module ramascene.querymanagement), 51
clean_local_leafs() (in module ramascene.querymanagement), 51
clean_single_leafs() (in module ramascene.querymanagement), 52
code (ramascene.models.Country attribute), 47
code (ramascene.models.ModellingProduct attribute), 48
code (ramascene.models.Product attribute), 49
completed (ramascene.models.Job attribute), 48
convert_to_numpy() (in module ramascene.querymanagement), 52
Country (class in ramascene.models), 46
Country.DoesNotExist, 46
Country.MultipleObjectsReturned, 47
created (ramascene.models.Job attribute), 48

D

default_handler() (in module ramascene.tasks), 55

G

get_aggregations_countries() (in module ramascene.querymanagement), 52
get_aggregations_products() (in module ramascene.querymanagement), 52
get_calc_names_country() (in module ramascene.querymanagement), 52
get_calc_names_product() (in module ramascene.querymanagement), 52
get_consumed_product_ids() (ramascene.productindexmanger.ProductIndexManager method), 50
get_country_count() (ramascene.productindexmanger.ProductIndexManager method), 50
get_full_selected_c_product_count() (ramascene.productindexmanger.ProductIndexManager method), 50
get_full_selected_p_product_count() (ramascene.productindexmanger.ProductIndexManager method), 50
get_indicator_units() (in module ramascene.querymanagement), 53
get_leafs_country() (in module ramascene.querymanagement), 53
get_leafs_modelled_product() (in module ramascene.querymanagement), 53
get_leafs_product() (in module ramascene.querymanagement), 53
get_modelled_names_product() (in module ramascene.querymanagement), 53
get_names_country() (in module ramascene.querymanagement), 53
get_names_indicator() (in module ramascene.querymanagement), 54
get_names_product() (in module ramascene.querymanagement), 54
get_next_by_created() (ramascene.models.Job method), 48
get_numpy_objects() (in module ramascene.querymanagement), 54
get_previous_by_created() (ramascene.models.Job method), 48
get_produced_product_ids() (ramascene.productindexmanger.ProductIndexManager

method), 51
get_product_count() (ramascene.productindexmanger.ProductIndexManager method), 51
get_selected_c_product_count() (ramascene.productindexmanger.ProductIndexManager method), 51
get_selected_p_country_count() (ramascene.productindexmanger.ProductIndexManager method), 51
get_selected_p_product_count() (ramascene.productindexmanger.ProductIndexManager method), 51
get_selected_s_country_count() (ramascene.productindexmanger.ProductIndexManager method), 51
global_id (ramascene.models.Country attribute), 47
global_id (ramascene.models.Indicator attribute), 47
global_id (ramascene.models.ModellingProduct attribute), 48
global_id (ramascene.models.Product attribute), 49

H

handle_complete() (in module ramascene.tasks), 55
home() (in module ramascene.views), 55

I

id (ramascene.models.Country attribute), 47
id (ramascene.models.Indicator attribute), 47
id (ramascene.models.Job attribute), 48
id (ramascene.models.ModellingProduct attribute), 49
id (ramascene.models.Product attribute), 49
identifier (ramascene.models.Country attribute), 47
identifier (ramascene.models.ModellingProduct attribute), 49
identifier (ramascene.models.Product attribute), 49
identify_country() (in module ramascene.querymanagement), 54
identify_modelling_product() (in module ramascene.querymanagement), 54
identify_product() (in module ramascene.querymanagement), 54
Indicator (class in ramascene.models), 47
Indicator.DoesNotExist, 47
Indicator.MultipleObjectsReturned, 47

J

Job (class in ramascene.models), 48
Job.DoesNotExist, 48
Job.MultipleObjectsReturned, 48
job_update() (in module ramascene.tasks), 55

L

leaf_children_global (ramascene.models.Country attribute), 47
leaf_children_global (ramascene.models.ModellingProduct attribute), 49
leaf_children_global (ramascene.models.Product attribute), 49
leaf_children_local (ramascene.models.Country attribute), 47
leaf_children_local (ramascene.models.ModellingProduct attribute), 49
leaf_children_local (ramascene.models.Product attribute), 49
level (ramascene.models.Country attribute), 47
level (ramascene.models.Indicator attribute), 47
level (ramascene.models.ModellingProduct attribute), 49
level (ramascene.models.Product attribute), 49
local_id (ramascene.models.Country attribute), 47
local_id (ramascene.models.Indicator attribute), 47
local_id (ramascene.models.ModellingProduct attribute), 49
local_id (ramascene.models.Product attribute), 49

M

model_final_demand() (ramascene.modelling.Modelling method), 46
model_intermediates() (ramascene.modelling.Modelling method), 46
Modelling (class in ramascene.modelling), 46
ModellingProduct (class in ramascene.models), 48
ModellingProduct.DoesNotExist, 48
ModellingProduct.MultipleObjectsReturned, 48

N

name (ramascene.models.Country attribute), 47
name (ramascene.models.Indicator attribute), 47
name (ramascene.models.Job attribute), 48
name (ramascene.models.ModellingProduct attribute), 49
name (ramascene.models.Product attribute), 50

O

objects (ramascene.models.Country attribute), 47
objects (ramascene.models.Indicator attribute), 48
objects (ramascene.models.Job attribute), 48
objects (ramascene.models.ModellingProduct attribute), 49
objects (ramascene.models.Product attribute), 50

P

parent_id (ramascene.models.Country attribute), 47
parent_id (ramascene.models.Indicator attribute), 48

parent_id (ramascene.models.ModellingProduct attribute), 49
parent_id (ramascene.models.Product attribute), 50
Product (class in ramascene.models), 49
Product.DoesNotExist, 49
Product.MultipleObjectsReturned, 49
ProductIndexManager (class in ramascene.productindexmanger), 50

R

ramascene (module), 56
ramascene.analyze (module), 45
ramascene.consumers (module), 46
ramascene.modelling (module), 46
ramascene.models (module), 46
ramascene.productindexmanger (module), 50
ramascene.querymanagement (module), 51
ramascene.tasks (module), 55
ramascene.views (module), 55
RamasceneConsumer (class in ramascene.consumers), 46
route_four() (ramascene.analyze.Analyze method), 45
route_one() (ramascene.analyze.Analyze method), 45
route_three() (ramascene.analyze.Analyze method), 45
route_two() (ramascene.analyze.Analyze method), 45

S

save_job() (ramascene.consumers.RamasceneConsumer method), 46
status (ramascene.models.Job attribute), 48

U

unit (ramascene.models.Indicator attribute), 48
unpack() (ramascene.modelling.Modelling method), 46

W

websocket_connect() (ramascene.consumers.RamasceneConsumer method), 46
websocket_disconnect() (ramascene.consumers.RamasceneConsumer method), 46
websocket_receive() (ramascene.consumers.RamasceneConsumer method), 46
ws_response() (ramascene.consumers.RamasceneConsumer method), 46