Rokitt Astra platform

Data Sub-setting Use case

Astra platform. Data sub-setting support

Data sub-setting is a procedure of extracting a subset of the database or the other data-source with a respect to the referential integrity of the extracted data.

The results of sub-setting are greatly depended on the requirements and constraints defined by user.

The minimal set of requirements should include:

- The master table(s) or entity(s) that will be used as a starting point for the sub-setting;
- The data sampling schema – random, systematic, convenience, cluster, stratified, or user defined.

The constraints may further refine sub-setting results by exposing limits on selected data and rules that conduct data extraction procedures:

- Size of the sample for the starting set of the entities;
- List of the specific entities that have to be presented in the data subset;
- Attributes that should be included into the resulting dataset and etc.

To ensure a high quality of the resulting dataset the sub-setting procedures should support multivariate sub-settings that will deliver multiple result sets.

The minimal size sub-setting result that meets all requirements and constraints we would consider as an optimal.

JSON presentation of the Metadata Graph

Astra Data discovery builds a Metadata Graph that exceed information that can be collected from metadata of the Databases. In order to use this extended meta-information in data sub-setting, it should be converted into JSON, because this is the data exchange format used by Astra platform.

NPM JSON Graph specification is a very promising proposition that is equally suitable for applying on Front End as well as on Back End of the applications.

Design principles (as they are defined on https://www.npmjs.com/package/json-graph-specification):

- Use meaningful property names that reflect the semantic type of the value.
- Property names should not be excessively long.
- Property names should be plural when value is an array.
- Properties that allow a null value can be omitted.
- Define a JSON graph schema_ for content validation purposes.

This specification is using a very limited and well defined set of constructs:

- Node object;
- Node properties;
- Edge object;
- Edge properties;
- Graph object;
- Graph properties;
- Graphs object (to describe a collection of graphs);
- Graphs properties.

Example of JSON Graph specification

```
{
   "graph": {
      "nodes": [
         {
            "id": "A",
         },
         {
            "id": "B",
         }
      ],
      "edges": [
         {
            "source": "A",
            "target": "B"
         }
      ]
   }
}
```

```
{
 "graphs": [{ "directed": true, "type": "graph type", "label": "graph label",
       "metadata": {"user-defined": "values" },
          "nodes": [
             {"id": "0", "type": "node type", "label": "node label(0)",
                "metadata": {"user-defined": "values"} },
             {"id": "1", "type": "node type", "label": "node label(1)",
              "metadata": {"user-defined": "values"} } ],
          "edges": [ { "source": "0", "relation": "edge relationship", "target": "1", "directed": true,
"label": "edge label",
                "metadata": {"user-defined": "values"} } ]},
       {"directed": true, "type": "graph type", "label": "graph label",
        "metadata": {"user-defined": "values"},
       "nodes": [{"id": "0", "type": "node type", "label": "node label(0)",
                "metadata": {"user-defined": "values"}},
             {"id": "1","type": "node type", "label": "node label(1)",
                "metadata": {"user-defined": "values"} } ],
          "edges": [{"source": "1", "relation": "edge relationship", "target": "0", "directed": true,
"label": "edge label",
                "metadata": {"user-defined": "values"} } ] } ] }
}
```

JSON Graph Spec. Graph(s) object and properties

graph object:

A graph object represents a single conceptual graph.

graph properties:

- A **type** property provides a classification for an object. Its value is defined as a JSON string.
- A **label** property provides a text display for an object. Its value is defined as a JSON string.
- A **directed** property provides the graph mode (e.g. directed or undirected). Its value is JSON true for directed and JSON false for undirected. This property default to JSON true indicating a directed graph.
- A **nodes** property provides the nodes in the graph. Its value is an array of node object_.
- An **edges** property provides the edges in the graph. Its value is an array of edge object_.
- A **metadata** property allows for custom data on an object. Its values is defined as a JSON object.

graphs object:

A graphs object groups zero or more graph object_ into one JSON document.

- The graphs object_ is defined as a JSON array.

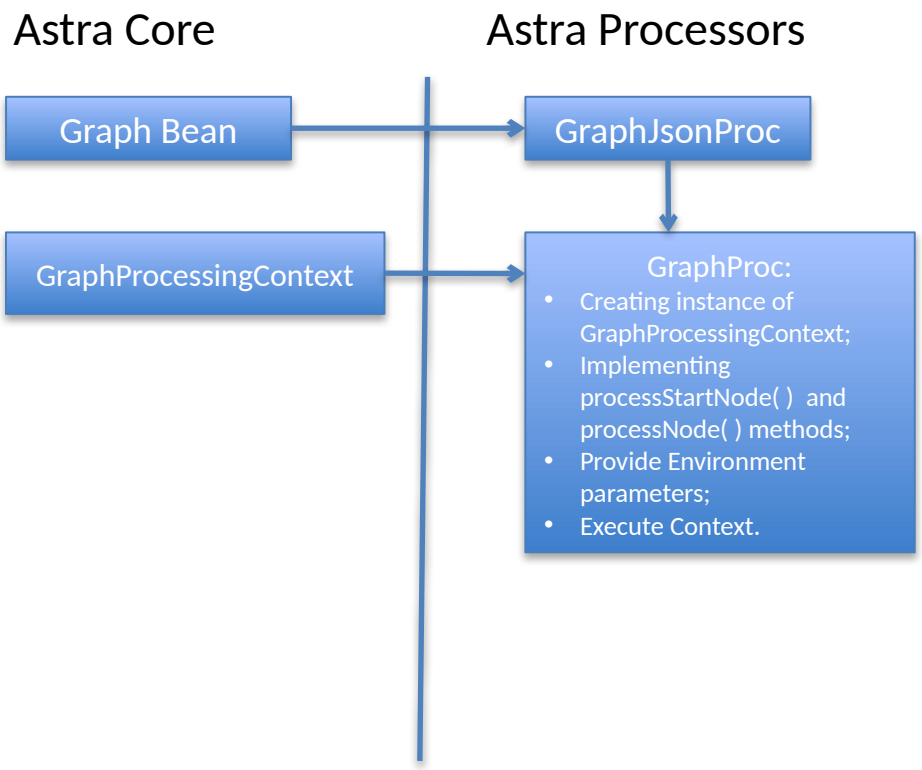
graphs properties:

- A **type** property provides a classification for an object. Its value is defined as a JSON string.
- A **label** property provides a text display for an object. Its value is defined as a JSON string.
- A **metadata** property allows for custom data on an object. Its values is defined as a JSON object.

JSON Spec in the Astra Data Exchange format

```
{
   "info": {
      "userId": "alexmy",
      "appId": "Astra",
      "version": 1,
      "serviceId": "com.rokittech.astra.ml.agent.batch.GraphPipeProcessor"
   },
   "data": "{<JSON Metadata Graph is going here>}",
   "options": {
      “sourcedb”:”<connection object from AstraEnvTool>”,
      “targetdb”:”<connection object from AstraEnvTool>”,
      "command": ”single-db-subsetting"
   }
}
```

Sub-setting Processor Architecture



Architecture of Sub-setting processor is greatly inspired by Apache Flink.
The execution environment in Flink is represented by ExecutionEnvironment. We can specify data processing by configuring this environment.
GraphProcesingContext serves the same purpose and it provides a generic environment to execute a wide class of data processing that can be defined as a graph.
Sub-setting is just a special case of a such graph defined data processing.
The execution environment of GraphProcessingContext is exposed by two methods:

- Map<String, Object> getEnv();
- void setEnv(Map<String, Object> env);

You can use this map to provide parameters like connection string, file location and so on.

To define a specific processing for each node we have to implement two abstract methods:

- T processStartNode(Node node, Processor processor);
- T processNode(Node node, Processor processor).

Astra Sub-setting processor implementation

```
public void run() throws Exception {
    // 1. Implement all abstract methods in GraphContextOld
    GraphProcessingContext<List<Integer>> ctx
            = new GraphProcessingContext<List<Integer>>(data, (Processor) processor) {
        @Override
        public List<Integer> processStartNode(Node node, Processor processor) {
            System.out.println("Start Node: " + node.getLabel());
            List<Integer> list = processNode(node, processor);
            return list;
        }
        @Override
        public List<Integer> processNode(Node node, Processor processor) {
            List<Integer> list = new ArrayList<>();
            for(int i = 0; i < 20; i++){
                list.add((int)(Math. random() * 50 + 1));
            }
            System.out.println("Node: " + node.getLabel());
            return list;
        }
    };
    // 2. Setup execution environment
    String sourceDbUrl = (String) processor.getInput().getOptions().get(GraphVocabulary.SOURCE_DB_URL)
    String targetDbUrl = (String) processor.getInput().getOptions().get(GraphVocabulary.TARGET_DB_URL)

    try {
        this.sourceConn = DriverManager.getConnection(sourceDbUrl);
        this.targetConn = DriverManager.getConnection(targetDbUrl);
    } catch (SQLException ex) {
        Logger.getLogger(GraphProc.class.getName()).log(Level.SEVERE, null, ex);
    }

    ctx.getEnv().put("source", this.sourceConn);
    ctx.getEnv().put("target", this.targetConn);

    // 3. Run graph processing
    ctx.execute();
```

1

2

3

Here is a sample code that demonstrate the use of GraphProcessingContext.

In the section 1 we are creating the instance of GraphProcessingContext – ctx, by implementing two methods processStartNode and processNode.

List<Integer> is a return result produced by those two methods.

The “data” argument in the constructor may be a Graph object or JSON string of the graph that we are going to process.

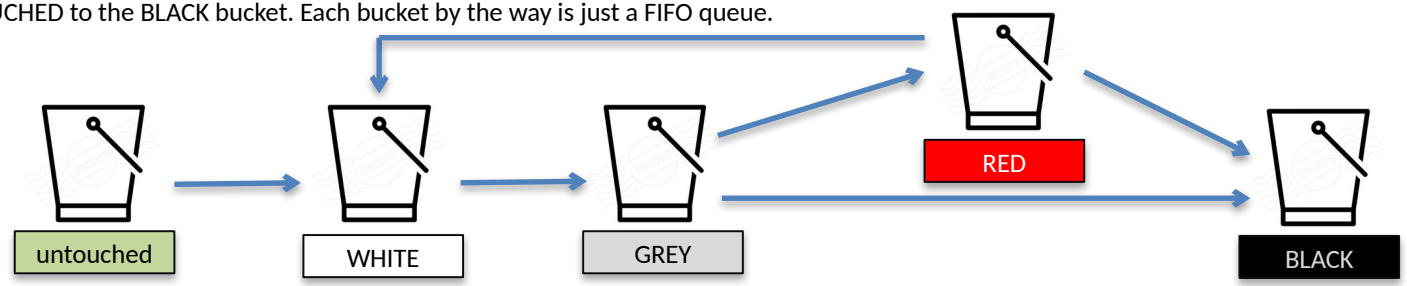
In our case we are using JSON string that is represented Metadata graph of the database. The implementations of those methods are totally dummy.

In the section 2 we are acquiring connection strings for source and target databases and then we are passing connection objects to the ctx.

Finally in the section 3 we are running specified context.

Graph processing algorithm

We are using a non-standard and original algorithm to perform data processing associated with each graph node (at least I did not find an exact analog of the provided algorithm on Internet yet). The main advantage of this algorithm is that it doesn't use a direct traversal over graph nodes, instead it uses buckets to define an execution status for each node. Buckets allow us to implement an asynchronous and a potentially distributed data processing.
To manage processing we are using 5 colored queues (we call them buckets). The goal of each node in the graph is to move from the bucket marked as UNTOUCHED to the BLACK bucket. Each bucket by the way is just a FIFO queue.



0. At the beginning all nodes are in the UNTOUCHED bucket.
1. We are starting from the start node(s). To start processing we are moving start node(s) to the WHITE bucket. Nodes in WHITE bucket are waiting to be processed;
2. We are picking a node from the bottom of the WHITE bucket and moving to the GREY bucket to start an actual processing associated with a given node;
3. By the end of processing node may go to BLACK or RED bucket. If the node has no in-nodes from UNTOUCHED bucket, then it goes to BLACK bucket, otherwise it goes to the RED bucket (in-nodes are the nodes that have a given node as a target in their edges). At the same time we are moving all out-nodes of the current node from UNTOUCHED bucket to the WHITE bucket (out-nodes are nodes that have a current node as a source in their edges);
4. After emptying WHITE bucket we are moving to the RED bucket. (Remember, WHITE bucket may be populated with a new nodes according to the p. 3.):
    - First we are trying to detect all in-nodes that are still in UNTOUCHED bucket for each node in the RED bucket and move them to the WHITE bucket;
    - Then we can move nodes from RED bucket to the BLACK bucket;
    - Finally we are switching to the p. 2.

References

- _JSON graph schema: https://github.com/jsongraph/json-graph-specification/blob/master/json-graph-schema.json;
- _JSON schema: http://json-schema.org;
- _real world examples: https://github.com/jsongraph/json-graph-specification/tree/master/examples;
- _TESTING: https://github.com/jsongraph/json-graph-specification/blob/master/TESTING.rst;
- _RFC 6839: https://tools.ietf.org/html/rfc6839;
- _RFC 6906: https://tools.ietf.org/html/rfc6906;