

In-Graph-Database Implementation of a General, Reusable Graph Schema and a Modular Data Preprocessing Pipeline For Eye-Tracking Data

Dominique Hausler* Jennifer Landes
0009-0004-2381-133X 0009-0003-1914-598X
University of Regensburg
Data Engineering Group
firstname.lastname@ur.de

December 2024

1 Graph Schema Modelling

In this section we present how CSV files containing eye-tracking data can be used to generate a specified schema. This general and re-usable schema is beneficial over the usual handling of CSV files because it enables storing, manipulating and updating the data in one data model. We choose a graph database, as eye-tracking data resembles highly interconnected time series data. This data can be visualized intuitively within Neo4j.

1.1 Modelling Time Series Data

In order to read the eye-tracking data to Neo4j an APOC (Awesome Procedures On Cypher) function is used (Listing 1 Line 1). Here an example is shown for the data gained for a specific task of one test person. Each row of the CSV file is stored in a timestamp labeled node (Line 2), that contains all of the columns as property keys (Line 3).

Listing 1: Read data from CSV file

```
1 LOAD CSV WITH HEADERS FROM "file:///data.csv" AS row
2 CREATE (n:Timestamp)
3 SET n = row
```

Data Cleaning. In a next step, the validity of the data is checked. This is beneficial in the beginning to increase performance because datasets might be considerably large. To avoid high-compute all invalid data is first selected

*Corresponding author.

(Listing 2 Line 3-5) and removed (Line 11). Consequently, it does not cause additional performance for the preprocessing pipeline. A value other than zero indicate that an error occurred during the eye-tracking experiment.

Listing 2: Check validity

```

1  MATCH (n:Timestamp)
2  // get all cases where data is invalid
3  WHERE n.ET_ValidityLeft IS NULL OR n.ET_ValidityRight IS NULL
4        OR n.ET_ValidityLeft <> n.ET_ValidityRight
5        OR n.ET_ValidityLeft = n.ET_ValidityRight AND (n.ET_ValidityLeft = "4.0"
        AND n.ET_ValidityRight = "4.0")
6  WITH collect(n) AS nonValidNodeIdList
7  // get all nodes with unvalid data by id
8  CALL apoc.nodes.get(nonValidNodeIdList)
9  YIELD node
10 // remove invalid nodes
11 DELETE node

```

To retrieve the desired general and re-usable graph schema that is applicable to perform data preprocessing, new labels need to be created (see Listing 3). Our schema needs the `SourceStimuliName`, holding the information about the performed task. As this currently is stored in a property key, we select it and add it as label to the associated node. First, the desired label is selected in Line 1, through the apoc function in Line 3 is used to add new labels to the nodes selected in Line 1. In the end (Line 8) the results are outputted.

Listing 3: Move property value of 'SourceStimuliName' to become node label

```

1  MATCH(n:Timestamp)
2  // dynamically add values of property key 'SourceStimuliName' to 'Timestamp'
   nodes
3  CALL apoc.create.addLabels(n, [n.SourceStimuliName])
4  YIELD node
5  // dynamically remove property key 'SourceStimuliName' from 'Timestamp'
   nodes
6  CALL apoc.create.removeProperties(node, ['SourceStimuliName'])
7  YIELD node as my_nodes
8  RETURN labels(my_nodes)

```

Afterwards, we need to create fixation, saccade and gaze nodes. This is necessary to specify and give an adequate overview over the stored data. In Listing 4 this is demonstrated for fixations, meaning all data connected to fixations is gathered (Line 2-9) and then outsourced by creating new nodes (Line 11) with all related data as property keys. As copies are created, holding the required property keys were created in Line 11, these are renamed to fixation in Line 16. After renaming the labels of the copied nodes, substrings are used to select the desired property keys (by defining a regex pattern) in Line 23. This is done because all related property keys start with the substring `Fixation` (for saccades the substring would be `Saccade` and so on). The code snippet in Listing 4 can be reused for saccades and gazes by changing the substrings (Line 23) as well as the desired label to give to the newly created nodes (Line 16).

Listing 4: Create fixation nodes for each unique FixationStart value

```

1 CALL {
2   // get all nodes with unique values for the property key 'FixationStart'
3   CALL{
4     MATCH (n:Timestamp)
5     WITH DISTINCT n ORDER BY n.'Fixation Start'
6     WITH n.'Fixation Start' AS fixationStart, collect(n)[0] AS
       distinctFixationStartNodes
7     WITH collect(distinctFixationStartNodes) AS
       distinctFixationStartNodeList
8     RETURN distinctFixationStartNodeList
9   }
10  // copy nodes with unique 'FixationStart' value
11  CALL apoc.refactor.cloneNodes(distinctFixationStartNodeList)
12  YIELD input, output
13  WITH input AS distinctFixationStartNodes, collect(output) AS
       fixationNodes, output
14
15  // rename label of copied nodes to 'Fixation'
16  CALL apoc.refactor.rename.label("Timestamp", "Fixation", fixationNodes)
17  YIELD batches, total, timeTaken, committedOperations
18  RETURN batches, total, timeTaken, committedOperations
19 }
20 MATCH (n:Fixation)
21 WITH n, collect(n) as fixationNodes
22 //only retrieve keys without the substring 'Fixation' by using regex
23 WITH n, fixationNodes, [key IN keys(n) WHERE key =~ '^(?!Fixation)[^$]*$'] AS
       fixationKeys
24 // remove all keys found by regex to only keep property keys with
25 CALL apoc.create.removeProperties(fixationNodes, fixationKeys)
26 YIELD node RETURN node

```

Besides creating the schema by adding fixation, saccade and gaze node, they also need to be connected to the associated nodes they retrieved their data from (see Listing 5). To be able to retrieve this information the outsourced property keys, here fixation start, holding the same information as the newly created fixation nodes were not deleted in Listing 4. This enables the creation of associated relationships (Line 23). After selecting and ordering both the timestamp nodes (Line 2-8) and the fixation nodes (Line 10-15) by fixation start, those are unwinded (Line 17-18). In Line 21 identical values fixation start are searched between timestamp and fixation nodes to create relationships connecting them in Line 23. The last part of the code is used to detect all property keys starting with the substring Fixation (Line 25) to delete these property keys from the timestamp nodes (Line 28). Otherwise unnecessary duplicated would occur.

Listing 5: Create relationship between Fixation and Timestamp labeled nodes depending on identical FixationStart property values

```

1 // a list with all timestamp nodes is created and ordered by the 'Fixation
   Start' values
2 CALL {
3   MATCH (n:Timestamp)
4   WITH DISTINCT n ORDER BY n.'Fixation Start'
5   WITH collect(n) AS timestampNodes

```

```

6     RETURN timestampNodes
7 }
8 WITH [x in range(0, size(timestampNodes)-1) | timestampNodes[x]] AS
    timestampNodesOrdered
9 // all fixation nodes are collected and ordered by the 'Fixation Start'
    value
10 CALL {
11     MATCH (m:Fixation)
12     WITH m ORDER BY m.'Fixation Start'
13     WITH collect(m) AS fixationNodes
14     RETURN fixationNodes
15 }
16 WITH timestampNodesOrdered, fixationNodes
17 UNWIND timestampNodesOrdered AS tsNode
18 UNWIND fixationNodes AS fixNode
19 // check unwinded node lists for identical 'Fixation Start' values
20 WITH tsNode, fixNode
21 WHERE tsNode.'Fixation Start' = fixNode.'Fixation Start'
22 // add new relationships
23 MERGE (tsNode)-[:IS_PART_OF]->(fixNode)
24 // select all property keys starting with Fixation from timestamp nodes
25 WITH tsNode, [key IN keys(tsNode) WHERE key STARTS WITH 'Fixation'] AS
    fixationKeys
26 WITH collect(tsNode) AS tsNodeList, fixationKeys
27 // remove all properties starting with Fixation from timestamp nodes
28 CALL apoc.create.removeProperties(tsNodeList, fixationKeys)
29 YIELD node RETURN node

```

2 Data Preprocessing Steps

In this section we present our modular, adaptable, extendable and customizable preprocessing pipeline. This can be done by freely combining the operators that are available here. Moreover, it the pipeline can easily be extended or combinations can be tested.

2.1 Feature Selection

First all irrelevant features, in order to prepare the data for machine learning based analysis or classification, are deleted from the dataset. Due to name conventions, we rename a property key in Listing 6. This enables us to faster delete unnecessary properties while keeping the data about the pupil size.

Listing 6: Rename property keys of right and left pupile

```

1 MATCH (n:Timestamp)
2 WITH collect(n) AS nodes
3 // rename 'ET_PupileRight' to 'PupileRight'
4 CALL apoc.refactor.rename.nodeProperty("ET_PupilRight", "PupilRight", nodes)
5 YIELD batches, total, timeTaken, committedOperations
6 // rename 'ET_PupileLeft' to 'PupileLeft'
7 CALL apoc.refactor.rename.nodeProperty("ET_PupilLeft", "PupilLeft", nodes)
8 YIELD committedOperations AS results
9 RETURN results;

```

To minimize data complexity, focusing only on the most informative features related to gaze behavior, we now delete unnecessary data (Listing 7). All irrelevant, unnecessary features are deleted by searching for substrings (Line 3).

Listing 7: Remove unnecessary features

```

1 // remove all property keys starting with substring 'ET_', 'Event' or '
  Interpolated'
2 MATCH (n)
3 WITH n, [key IN keys(n) WHERE key STARTS WITH 'Event' OR key STARTS WITH '
  ET_' OR key STARTS WITH 'Interpolated'] AS etKeys
4 WITH collect(n) AS nodes, etKeys
5 CALL apoc.create.removeProperties(nodes, etKeys)
6 YIELD node RETURN node

```

2.2 Data Preparation

In this section a variety of data preparation algorithms implemented in-graph-database are available. Either one algorithm can be selected or the results can be compared to each other, depending on one's needs.

2.2.1 Data Cleaning

Data Cleaning is performed right after reading the CSV file to Neo4j (see Listing 2). The substring search presented for the feature selection (Listing 4 Line 23) can be reused to e.g., remove whitespace. This could be done by searching for property keys starting with a whitespace similar to Listing 5 Line 25.

2.2.2 Missing Value Imputation

Handling missing values is important to prepare the data for further analysis. In this paper we present four different methods to handle null values. In the first code fragment interpolation [2] is shown, all null values of **Gaze X** are selected (Listing 8 Line 3). Then the predecessor and successor nodes are searched by their ids in Line 5. Between those two the mean is calculated and set as new value for the former null value (Line 12). By replacing the the property key where null values need to be handled (like **Gaze X** in Listing 8 Line 3) and the selected label **Timestamp** (Line 1) the code can be reused to perform missing value imputation on other properties of differently labeled nodes as well. In this code snippet the data is directly replaced. By using the **SET** command the method used to do so can be stored in a the value **interpolation** in a property key like **missing_value_imputation_method**

Listing 8: Option A1: Interpolation for Gaze X

```

1 MATCH (n:Timestamp)
2 //search if any mandatory property key is empty
3 WHERE n.'Gaze X' IS NULL OR n.'Gaze Y' IS NULL OR n.'Gaze Velocity' IS NULL
  OR n.'Gaze Acceleration' IS NULL
4 //get all ids

```

```

5  WITH ID(n)-1 AS predecessorNode, ID(n) AS nodeId, ID(n)+1 AS successorNode,
    n, collect(n) AS nodes
6  // get gaze x values of following nodes
7  CALL apoc.nodes.get([successorNode])
8  YIELD node AS nextNodeList
9  // get 'Gaze X' values of foregoing nodes
10 CALL apoc.nodes.get([predecessorNode])
11 YIELD node AS predecessorNodeList
12 CALL apoc.create.setProperty(nodes, 'Gaze X', (toFloat(successorNodeList.'
    Gaze X') + toFloat(predecessorNodeList.'Gaze X'))/2)
13 YIELD node RETURN node

```

The second options would be deleting properties containing null values [6]. How this can be accomplished is shown in Listing 9.

Listing 9: Option B: delete values

```

1  MATCH (n:Timestamp)
2  //search if any mandatory property key is empty
3  WHERE n.'Gaze X' IS NULL OR n.'Gaze Y' IS NULL OR n.'Gaze Velocity' IS NULL
    OR n.'Gaze Acceleration' IS NULL
4  DETACH DELETE n

```

The third operator, implemented by us is lofb (last observed carried forward) [7]. As a fourth option the code can also be adjusted to nobc (next observed carried backward). The following code in Listing 10 illustrates lofb for the property key Gaze X of Timestamp labeled nodes. By simply using the successorNode instead of predecessorNode in Line 10 this can be adjusted to nobc. Here the property keys null value is replaced by the newly calculated one (Line 10).

Listing 10: Option C: lofb – last observed value backward – or nobc – next observed carried backward)

```

1  MATCH (n:Timestamp)
2  //search if any mandatory property key is empty
3  WHERE n.'Gaze X' IS NULL OR n.'Gaze Y' IS NULL OR n.'Gaze Velocity' IS NULL
    OR n.'Gaze Acceleration' IS NULL
4  //get all ids
5  WITH ID(n)-1 AS predecessorNode, ID(n) AS nodeId, ID(n)+1 AS successorNode,
    n, collect(n) AS nodes
6  // get gaze x values of foregoing nodes
7  CALL apoc.nodes.get([predecessorNode])
8  YIELD node AS predecessorNodeList
9  // add property 'Gaze X' to selected nodes
10 CALL apoc.create.setProperty(nodes, 'Gaze X', toFloat(predecessorNodeList.'
    Gaze X'))
11 YIELD node RETURN node

```

2.2.3 Outlier Detection

Another important preprocessing step is the detection and handling of outliers to prepare the data for ML-based analysis. We show three operators to do so: Setting a threshold [8, 1], using the z-score [3] and the interquantil range [5]. Listing 11 demonstrates how datatypes can be changed. In this case the fixation duration is changed from a string to a float.

Listing 11: Change datatype (here from string to float)

```
1 MATCH (n:Fixation)
2 SET n.'Fixation Duration' = toFloat(n.'Fixation Duration')
```

After the datatype was changed, we can start with the outlier detection. The first operator uses a threshold, here a score under 60 milliseconds (Listing 12 Line 3), to identify if any fixation nodes were labeled falsely. If that is the case their label is changed to saccade in Line 5.

Listing 12: Threshold based approach for Fixation nodes

```
1 MATCH (n:Fixation)
2 // set a threshold
3 WHERE n.'Fixation Duration' < 60
4 WITH collect(n) AS outlier
5 CALL apoc.refactor.rename.label("Fixation", "Saccade", outlier)
6 YIELD batches, total, timeTaken, committedOperations
7 RETURN batches, total, timeTaken, committedOperations;
```

Another option is the z-score. As there is no predefined function to calculate the z-score (see Listing 13 Line 17), the variance (Line 8) and the standard derivation (Line 9) for the desired property key needs to be calculated beforehand. Each of these interim results could also be added as property keys to later reuse this value by using the SET command (Line 10). In this example only the z-score is stored, if an outlier was found (Line 17). Values out of a range between +3 to -3 are outliers which could be deleted (Listing 9) or handled with locf (Listing 10).

Listing 13: Z-score

```
1 // Calculate average of fixation duration
2 MATCH (n:Fixation)
3 WITH avg(n.'Fixation Duration') AS mean
4
5 // Calculate standard deviation
6 MATCH (n:Fixation)
7 WITH mean, collect(n.'Fixation Duration') AS fixationDuration
8 WITH mean, reduce(s = 0.0, x IN fixationDuration | s + (x - mean) ^ 2) /
   size(fixationDuration) AS variance
9 WITH mean, sqrt(variance) AS stdev
10 //SET n.stdev = stdev
11
12
13 // Calculate z-score for each fixation duration
14 MATCH (n:Fixation)
15 WITH n, mean, stdev
16 // add z-score value to each fixation node
17 SET n.z_score = (n.'Fixation Duration' - mean) / stdev
18 MATCH (n:Fixation)
19 // filter for outliers
20 WHERE n.z_score > 3 OR n.z_score < -3
21 // return outliers or add code to handle outliers here
22 RETURN n AS outlier
```

The last algorithm is the interquantil range (IQR). Just like for the z-score there is no predefined function. Consequently, all four percentiles are calculated

with the predefined function `apoc.agg.percentiles` in Line 2 of Listing 14. After the outlier is detected in Line 8 for example `locf` or `nocb` could be performed (see Listing 10).

Listing 14: Interquartile range

```

1 MATCH (n:Fixation)
2 WITH apoc.agg.percentiles(n.'Fixation Duration', [0.25, 0.5, 0.75, 1.0]) AS
   percentiles
3 WITH percentiles[0] AS Q1, percentiles[2] AS Q3
4 WITH Q1, Q3, Q3 - Q1 AS iqr
5 WITH Q1, 1.5*iqr-Q1 AS lowerBoundary, 1.5*iqr+Q3 AS upperBoundary, iqr
6 MATCH (n:Fixation)
7 WHERE n.'Fixation Duration' < lowerBoundary OR n.'Fixation Duration' >
   upperBoundary
8 // return outliers or handle them by using locf, nocb ...
9 RETURN n AS outlier

```

2.3 Normalization

The last step of the preprocessing pipeline is the normalization of the data. We implemented the min-max [4] as well as the z-score normalization [3] because we mainly encounter floats or integers. The min-max normalization is demonstrated in Listing 15. Here the datatype change is included in the code snippet in Line 3. The normalized value is stored in an additional property (Line 6) together with the used method (Line 7).

Listing 15: Min-max normalization

```

1 MATCH (n:Fixation)
2 WHERE n.'Fixation Duration' IS NOT NULL
3 SET n.'Fixation Duration' = toFloat(n.'Fixation Duration')
4 WITH min(n.'Fixation Duration') AS minValue, max(n.'Fixation Duration') AS
   maxValue
5 MATCH (n:Fixation)
6 SET n.normalized_value = (n.'Fixation Duration' - minValue) / (maxValue -
   minValue)
7 SET normalization_method = 'min-max'

```

The second operator is the z-score normalization in Listing 16. Storing specific results in property keys comes in handy at this point, showing the benefit of this so-called property storage. We can make use of the standard derivation (stdev) from the z-score calculation in Listing 13.

Listing 16: Z-score normalization

```

1 MATCH (n:Fixation)
2 WITH avg(n.'Fixation Duration') AS avg
3 SET n.normalized_value = (n.'Fixation Duration' - avg) / stdev
4 SET normalization_method = 'z-score'

```


References

- [1] Taisir Alhilo and Akeel Al-Sakaa. “Handling Noisy Data in Eye-Tracking Research: Methods and Best Practices”. In: *2023 International Workshop on Biomedical Applications, Technologies and Sensors (BATS)*. 2023, pp. 39–44.
- [2] Mariska E. Kret and Elio E. Sjak-Shie. “Preprocessing pupil size data: Guidelines and code”. In: *Behavior Research Methods* 51.3 (June 2019), pp. 1336–1342.
- [3] Ivan Miguel Pires et al. “Homogeneous Data Normalization and Deep Learning: A Case Study in Human Activity Classification”. In: *Future Internet* 12.11 (Nov. 2020). Number: 11 Publisher: Multidisciplinary Digital Publishing Institute, p. 194.
- [4] Zulfikar Setyo Priyambudi and Yusuf Sulistyo Nugroho. “Which algorithm is better? An implementation of normalization to predict student performance”. In: *AIP Conference Proceedings* 2926.1 (Jan. 2024), p. 020110.
- [5] H. P. Vinutha, B. Poornima, and B. M. Sagar. “Detection of Outliers Using Interquartile Range Technique from Intrusion Dataset”. In: *Information and Decision Sciences*. Ed. by Suresh Chandra Satapathy et al. Singapore: Springer, 2018, pp. 511–518.
- [6] Shichao Zhang, Zhi Jin, and Xiaofeng Zhu. “Missing data imputation by utilizing information within incomplete instances”. In: *Journal of Systems and Software* 84.3 (Mar. 2011), pp. 452–459.
- [7] Yifan Zhang and Peter J. Thorburn. “Handling missing data in near real-time environmental monitoring: A system and a review of selected methods”. In: *Future Generation Computer Systems* 128 (Mar. 2022), pp. 63–72.
- [8] Jun Zhao, Wei Wang, and Chunyang Sheng. “Data Preprocessing Techniques”. In: *Data-Driven Prediction for Industrial Processes and Their Applications*. Ed. by Jun Zhao, Wei Wang, and Chunyang Sheng. Cham: Springer International Publishing, 2018, pp. 13–52.