Query rewriting and visualising for concept evolution

by

Robbert van Erpers Roijaards

A thesis submitted in conformity with the requirements
for the degree of Computing Science

Department of Natural Sciences
University of Utrecht
Project supervisor prof.dr. I. Velegrakis
2nd Examinor dr. A.A.A. Qahtan
Student Number 5739675

Abstract

Concept evolution in data is the process where a certain concept was applied to a type of records that evolves to a new concept that is now applied to the same type of record. There are some existing tools to discover concept evolution and query with this concept evolution in mind. Each approach does have its limitation. The application of these approaches tends to depend on the structure of the data. Concept evolution in a data set can cause problems where historical data is missed when querying because to the concept of the query underwent evolution. For these types of situation rewriting a query based on the concept evolution in the data would mitigate those problems. We introduce a system that creates a concept evolution graph based on the evolution in the data and uses it to rewrite a query to take the evolution into account. Additionally we create visualisations that help understand the evolution in the data and also specific queries.

Contents

Chapter 1

Introduction

When dealing with large amounts of data that is continuously expanded it is possible that the data or the system that creates the data changes over time. These types of changes can lead to concept evolution and can result in problems when querying over this data. It can cause information to be missed due to these changes. An example of this is the case of trying to forecast a trend by using historical data. Due to the evolution in the data some of the historical data can be missed causing the forecast to be made without all the available data. Since customer experience is an important aspect of call centres having a bad service due to a bad forecast is undesired. Nobody wants to wait in a queue all day due to not enough call center operators being available. On the other hand having too many operators is bad from a budgetary standpoint. To avoid either scenario there needs to be a fitting amount of operators available at the call centre to meet the incoming demand. Therefor being able to make a good forecast is vital. Since the evolution in the data can be caused by many different factors it might be hard or impossible for users to know this evolution has taken place. Not knowing the evolution in the data can make it harder for users to effectively do their work.

There are a number of approaches to mitigate the problem of concept evolution. One such approach is that of detecting terminology evolution in text documents by trying to discover semantically similar concepts by association rule mining[10]. While this approach helps with identifying the terminology evolution it does require text documents to mine the association rules. If these semantic links do not exist due to the lack of text documents or simply no links existing for the data set this approach will not work. The use of this approach is however relatively simple as it just extends the answer of a user query without any added effort.

Another approach is the utilisation of temporal information in RDFs[4]. Here an evolution graph is proposed which is build on a temporal RDF. If this graph is build from the concepts in our data set we are able to query this. However, this approach focuses more on the type of relations between the different items in the RDF. Also the querying mechanism for these evolution graphs might be too complicated for everyday use. Ideally, a combination of the two approaches is executed where easy querying is preserved and the ability to query comprehensive graphs is available. Such a system would be able to be used by users without changing their current methods yet still allow for concept evolution to be queried.

Our work aims to create a system that can rewrite a user query to take the evolution of a data set into account. This would allow users to query the data without change, yet still take into account the evolution of the data. Since this might result in oddly looking results for a user. Namely, data that does not seem directly related to the initial query might be returned to explain this the system will also have a visualisation part. This visualisations aims to show the evolution of the concepts in the initial query to better explain the results of the query rewriting. To do this visualisation part the system utilises two different visualisations one to emphasise the evolution of a concept and the other to better visualise the timeline of evolution in the data. To rewrite the query the system utilises a concept evolution graph that the evolution present in the data. Which is traversed in order to rewrite our queries.

Specifically, the contributions we are making in our work is presenting a concept evolution graph that can be traversed to rewrite a query. Based on the evolution of this query found in the graph visualisations are made to better understand the evolution. We define evolution events on which we construct the concept evolution graph (Chapter 4). We introduce a traversal algorithm to correctly traverse through the concept evolution graph based on a concept (Chapter 5.1). The result of this traversal has to be correctly combined in order to take the expanded concepts from a query to a set of rewritten queries (Chapter 5.2). Also based on the expanded concepts from the traversal we create a mechanism to create two different visualisations (Chapter 6.6). A GraphViz visualisation to emphasise the evolution of concepts (Chapter 6.6.1) and an Excel visualisation to emphasise the timeline of the evolution (Chapter 6.6.2). Finally, we perform some experiments to test the performance of the creation of the concept evolution graph and the query rewriting based on the size of the concept evolution graph (Chapter 7).

Chapter 2

Related work

In this chapter we will discuss related work that touches on the topic of this thesis. This ranges from fields where work discussed in this thesis can be applied such as call centres to different approaches to model concept evolution. The goal is to show the applications and work done around dealing with concept evolution.

2.1 Using historical data for call center improvements

Call-centres tend to be the main avenue of communication between companies and their customers. It being a call centres does not imply that there is only voice communication, chat communication is also common.Inbound call centres are very labour intensive since it requires customer support representatives to be present if the call centre wants to avoid large queues. By creating an efficient routing for the call center combined with appropriate staffing this problem can be mitigated. This approach can avoid too large queues or the use of an overabundance of customer support representatives. A common approach is to predict the volume of calls or predict the contents of a call based on historical data to improve the call center´s efficiency and reduce spending.

Analyzing historic data and applying that analysis to incoming calls can improve the routing of a call center[2]. Besides having enough representatives to handle the incoming communication there is also a factor of connecting a customer to the right representative. Customers with certain questions can sometimes only be handled by a representative trained in the area that question is about. Connecting the customer to the wrong representative will take up more time due to the customer having to be transferred to the correct representative. This takes up time from more representatives which can be avoided by having more accurate routing.

The system seen in figure 2.1 describes such a system that tries to efficiently route calls to the correct representatives. The goal of the switch layer is to link incoming calls to the best representatives. For this linking three data sets are used: historical data, customer information and Customer Services Representatives information. These data sets are merged into a single data set and used as a training set for data modelling purposes. The data model is created by an offline training process and used to score incoming calls. The model passes its outcome to the optimiser to create a mapping for routing purposes.

This approach tries to optimise call routing with the help of machine learning. Combining data

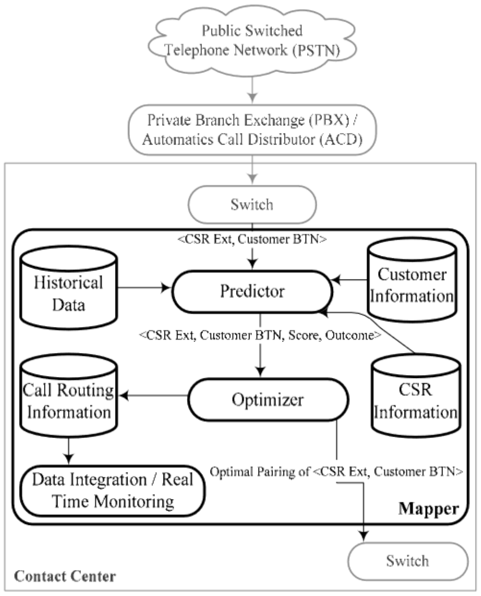

Public Switched Telephone Network (PSTN)
Private Branch Exchange (PBX) / Automatics Call Distributor (ACD)
Switch
<CSR Ext, Customer BTN>
Historical Data
Predictor
Customer Information
<CSR Ext, Customer BTN, Score, Outcome>
Call Routing Information
Optimizer
CSR Information
Data Integration / Real Time Monitoring
Optimal Pairing of <CSR Ext, Customer BTN>
Mapper
Switch
Contact Center

Figure 2.1: Call center routing [2]

sets from customers, representatives and historical data to find the best representative to handle the call. The historic data used here is the raw data of the calls. This system analyses this data combined with the personal data of the customer and representative to use in the prediction of the most optimal customer-representative pairing. This is important if the satisfaction of the call is an important criteria such as in sales.

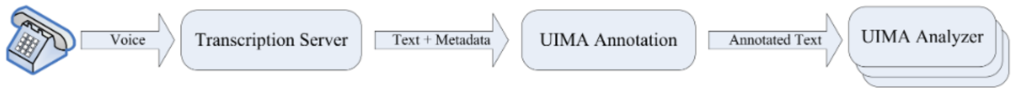


Figure 2.2: Call center classification [15]

Another way to improve call center communication is to help representatives by classifying calls and presenting commonly used solutions [15]. This system as seen in figure 2.2 analyses the data of a call to search and present useful information. Here each call will be transcribed and together with the call metadata such as time-stamps and channel identification (to distinguish between the caller and the operator), is sent to an annotator. This annotator compares the transcribed call in the current corpus, to a general corpus in order to detect significant words in the text. Since detecting sentence endings in transcribed text is difficult a heuristic approach is used to fragment the text. Each time the speaker changes or there is prolonged inactivity a fragment ends. When a call is broken up into fragments each fragment can be annotated with additional knowledge based on the significant words in the fragment. A fragment can also be without any significant words and will be labeled as an insignificant fragment.

When a call is annotated it can be analysed. A representative can search through past calls based on the past call's annotations and find solutions. The system can also present the representative with suggestions. This can be done based on the first few significant fragments of an ongoing call.

Besides detecting significant fragments in a call, insignificant fragments can also be detected and can provide insight for administrators.

There are many areas of a call center that can be improved. This approach focuses on the calls itself by improving the ability of the call center to answer questions by providing relevant information to the representatives. The first work discussed focuses on routing calls in the most efficient ways. One thing both approaches have in common is their reliance of historic data of communication of the call center.

The earlier mentioned issue of staffing call-centres correctly is another call-center problem. This problem can be made more difficult if the call-center deals with multiple types of customers that need different types of operators. Traditionally this problem is an optimization problem, where the objective is to minimise salary costs while meeting the quality of service targets [7].

2.2 Terminology Evolution for Querying

Concept evolution is a problem that can express itself in many ways. A popular case of it is that of search engines trying to find relevant information based on a user query. Directly searching for the words in the query will not always return a satisfying result. There is work done on taking into account the terminology evolution of concepts. It is helpful for our problem that we examine the approaches taken to solve this issue. While the text retrieval is not directly applicable, the overall structure is of interest for how to effectively model a concept evolution algorithm.

A large amount of time stamped documents exists online in all forms, such as news articles, web pages or blog-posts. The terminology within these documents can change over time, which causes a disconnect with the current term and the historical term[10]. This leads to the situation where a user can pose a query about a current concept. This query needs to be translated to a time aware query in order to take the historical concepts into account. These changing concepts can be called Semantically Identical Temporally Altering Concepts (SITAC).

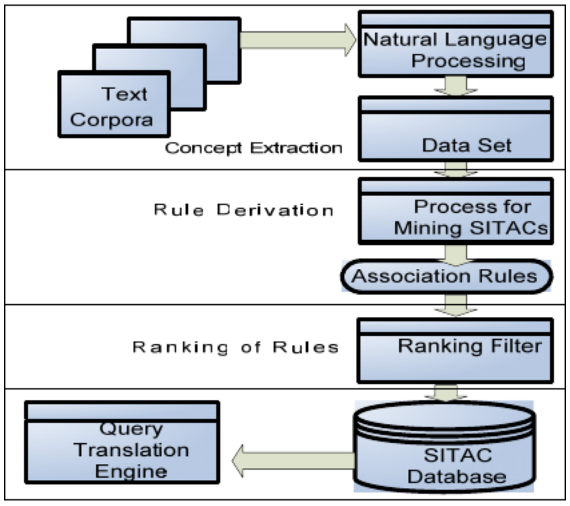


Figure 2.3: SITAC system architecture [10]

To be able to translate a query to a time aware query using these SITAC they first need to be discovered. The process to discover SITACs can be seen in figure 2.3. The initial step is to extract concepts out of text documents with natural language processing. The extracted concepts from the data set upon which the rules are discovered. The association rules that need to be discovered are of the form (C1, T1) $\rightarrow$ (C2, T2), where C1 and C2 are concepts and T1 and T2 are timestamps from the documents in which these concepts appear. These rules can link concepts in two different time periods. To get a link between concepts in a text, there has to be an event, such as a verb, that connects the concepts together. Every time such a rule is discovered a weight is given to the rule. If the rule is already discovered the weight is lowered of the rule is lowered instead. This results in the rules that appear the most are more relevant. These rules with small weights are considered SITACs. These SITACs are then ranked against each other using the *Jaccard* Coefficient to make sure we know what the most relevant SITACs are given a concept. When the SITAC database is built the user queries can be handled as time aware queries. In order to do this every concept of the user query has to be extracted. Search the SITAC database for every concept of the user query, if there are any matches, the query is extended with the SITAC and the time. This results in a query that contains all relevant SITACs and times these are relevant. The query is then executed by searching the data set for all the concepts in the extended query and only accepting matches that also match the time of the SITAC. This will return a set of documents that are likely to be relevant to our initial query.

This approach tried to discover semantically similar concepts by association rule mining with the idea that concepts linked with the same verbs tend to be similar. When these have been discovered any query is then compared against the database to find all semantically similar concepts, which are then used to find relevant documents. This does require terms from different time periods to have a high overlap. The result is that concepts which are deemed semantically similar and linked through SITACs are then stamped as terminology evolution. This means that this approach does not directly find evolution, but sees semantically linked concepts as evolution.

2.3 Concept evolution in RDF graphs

Concepts can evolve over time in many different ways. For example research concepts evolve when we learn more about them. Or the concept of any country goes through many changes during its lifetime even if simply looking at what land area a country occupies. Capturing and querying this wide variety of evolution is not a simple task. An approach of modelling this evolution can be done through a temporal resource description framework (RDF) [8]. An RDF is originally designed as a standard model and language for data interchange [12]. It has come to be used as a more general method to describe graph data. RDF is a directed graph and models a set of resources, basically anything that has a universal resource identifier (URI). A URI can identify the three different parts an RDF is made up of. These so-called triple statements contain a node for a subject, a node for an object and an edge from the subject to the object [9]. This simple model is capable of expressing complex situations. This simple model can be extended to also model the history of the RDF. This multidemnsional RDF (MRDF) has as extension that certain information in the graph is only present under certain circumstances [6]. The information in the RDF can have a condition under which it is present. An example of this is a graph about the work days. Under normal circumstances workdays are monday through friday, but during the summer holiday this is different. This can be expressed as $[\mathit{season} \neq \mathit{summer},\ \mathit{weekday\ not\ in}\ \{\mathit{Saturday,Sunday}\}]$, which means that this definition of workday is only present under the context of the season not being summer.

Extending the RDF can also be done to specifically capture temporal information. To extend an RDF with temporal information in order to create a temporal RDF the temporal information needs to be added to the edges in the model. The temporal information comes from a temporal database. This database holds the triple statement and an interval also called the lifespan of the resource. There are two types of temporal dimension that are considered, valid time and transaction time. Valid time is the time the data is valid in the world, which is the interval stored in the database. Transaction time is the time when the data is stored in the database. The time that is used to extend the RDF is valid time. By extending the edges with an interval $[a, b]$ the RDF can now express when certain relations in the graph are valid, where $a \leq b$.

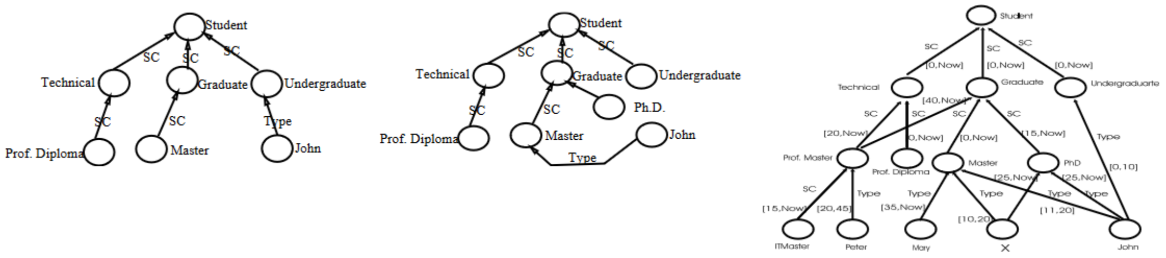


Figure 2.4: Initial RDF graph (left) after some changes (middle) extended to a temporal RDF graph (right) [8]

If we look at our initial RDF in figure 2.4 on the left, we can see a graph that models a current situation of a university. When changes occur the graphs gets altered and information gets lost, as can be seen in the middle figure. Before the change John was an undergraduate and the university did not have a PhD program. After the changes John has become a master student and the university offers a PhD program. This structure only captures the current situation and has no way of showing temporal information. If the RDF gets extended by adding the temporal information to the edge a more complete picture emerges. By looking at the right graph in the figure we can see that John was an undergraduate during time interval [0, 10], a master during [11, 20] and a phd during [25, now]. In essence this extension has created a graph from which other graphs can be inferred. Since each edge now has an interval on which it is valid, a graph can be constructed for different intervals. As such, both the left and middle graph are represented in the right graph, but only under certain time constraints.

Adding temporal information to an RDF allows for more information to be incorporated in the graph, but it does not model the evolution of concepts over time or the evolutionary relationships. However, this model can be extended to also include this information [4]. To model evolution there are two more dimensions introduced besides the temporal dimension, the mereological and the casual. Mereology is the study of parts and the wholes they form [11]. In the model it will be used to capture the parthood relationship between concepts in a way that is carried forward as concepts evolve. This relationship is the special property $partof$, which is reflexive, anti-symmetric and transitive. A resource x is a part of resource y if the concept modelled by the resource x is part of the concept of resource y and $x \neq y$. To model the casual relationships the notion *becomes* is used, to show the interdependency between two resources. The constraint of this relationship for resources x & y is $x_{end} < y_{begin}$. To strengthen this modelling the notion of a *liaison* is used. A liaison is when two concepts are linked through another concept by $partof$ and $becomes$ relations. For example, if a concept is part of another concept ($partof$) and has a casual relationship ($becomes$) with another. See figure 2.5 for more liaison examples.

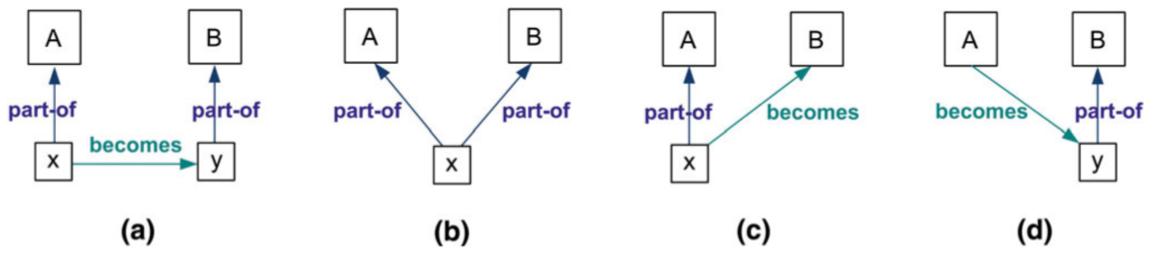


Figure 2.5: Liaison examples [4]

Besides defining these dimensions there are also evolution events that may exist, namely a join $\langle\{c_1 \dots c_n\}, c, t\rangle$, split $\langle c, \{c_1 \dots c_n\}, t\rangle$, merge $\langle c, c', t\rangle$ or detach $\langle c', c, t\rangle$. The join event means that from a part of a concept in $\{c_1 \dots c_n\}$ becomes a part of a concept c. The concept c is created from a part of these concepts at time t. The split event is conceptually the opposite of the join event. A split means that every part of a concept c becomes a part of a concept $\{c_1 \dots c_n\}$ at time t. The merge event is when a concept c ends and becomes part of another concept c' at time t. The detach term is when a new concept c is formed from at least one other concept c' at time t.

With the needed dimension and evolution events defined we can apply this method to a temporal RDF (figure 2.6). In this figure on the left side (a) we can see our original temporal RDF and on the right side is the RDF with evolution. As can be seen in the original RDF East & West Germany are not connected with Germany, while this should be an obvious connection. Applying this method the concept Germany gets linked with East & West Germany through the split event. Later, another link is made when East & West Germany reunite and form Reunified Germany, shown by the join events. The reunification is also unconnected in the original RDF, but in the evolution graph it is connected in by two join events. With these concepts connected we can see that Germany is related to Reunified Germany through a liason (East or West Germany).

The method proposed to query this evolution graph is a navigational query language to traverse temporal and evolution edges in the graph. This language is based on nSPARQL, which is an extension of SPARQL [18]. The nSPARQL query language is designed to navigate through an RDF graph that uses regular expressions as building blocks. The language is expressive enough to both query and navigate through an RDF graph. It uses four different axes: self, next, edge and node. The extension proposed is adding five evolution axes: join, split, merge, detach and becomes. This will extend the traversing capabilities of the query language so it is able to traverse the evolution graph. The grammar of this query language can be seen in figure 2.7.

In this grammar a node is represented by a and I is a timer interval. The semantic function is $\mathcal{E}$ and for a $\mathcal{E}[[exp]]$ the return is a set of tuples of the form $\langle x, y, I\rangle$ such that there is a path from x to y during interval I. An example of this is the expression $\mathcal{E}[[self :: Germany/next :: head/next :: type]]$, which returns the tuple $\langle Germany, Chancellor, [1988, 2005]\rangle$ if applied to graph **b** in figure 2.6. These expressions can be more complicated too. They can take the form of a nested expression $\mathcal{E}$ [[self[next :: head/self :: Gerhard Schröder]]], which would return two tuples ⟨Reunified Germany, Reunified Germany, [1990, 2005]⟩ and ⟨West Germany, West Germany, [1988, 1990]⟩.

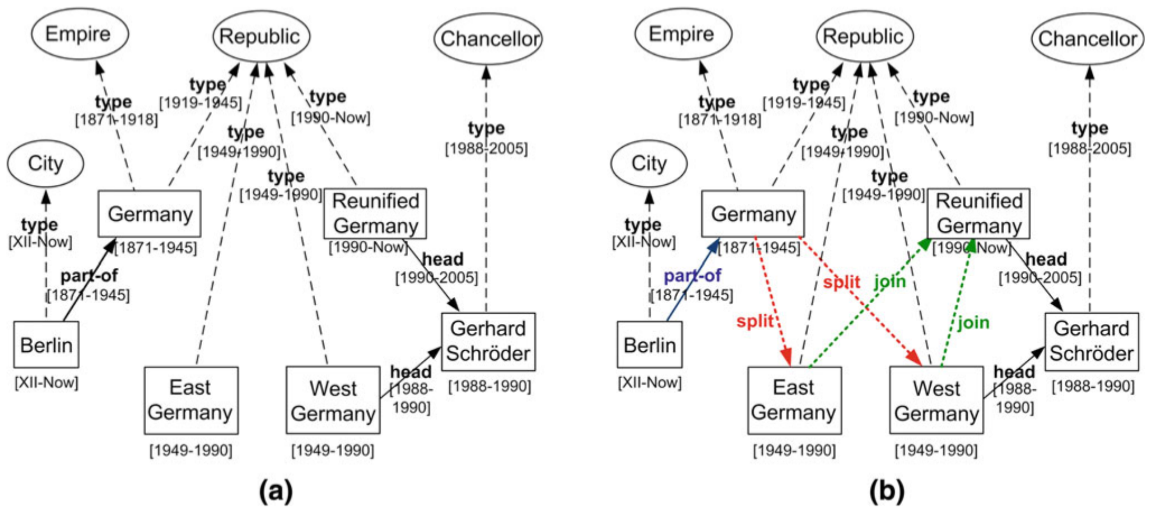


Figure 2.6: A temporal RDF **(a)** and an evolution graph **(b)** [4]

$$\begin{aligned} exp := \mathbf{axis} \mid \mathbf{t-axis} :: a \mid \mathbf{t-axis} :: [exp] \mid \\ exp[I] \mid exp/exp \mid exp|exp \mid exp^* \end{aligned}$$

Figure 2.7: Query grammar [4]

This approach shows that it is possible to add evolution to RDF graphs and also query upon this evolution by extending existing methods. With this approach querying upon the knowledge graph is possible with the addition of uncovering evolution in the graph. It shows a possible architecture for graphs that contain evolution and a method to traverse them through querying.

2.4 Detecting concept evolution

Besides querying an already known data set that contains evolution, there is also the problem of the detection of concept evolution in the first place. Especially in a data stream that can be considered of infinite length it is a difficult problem to detect when a certain concept evolves. It is impracticable to constantly train on all historic data in this situation due to the nature of an infinite data stream.

Data streams are high volume and when a new class type appears in the data it can be considered as concept evolution. This can be in multiple forms such as in network security, where a new type of attack found in the data stream is concept evolution. But this can also be applied to text based data streams such as Twitter. When the conversation on Twitter changes there is a change in text messages in the stream, old topics disappear and new topics emerge. This can also be seen as concept evolution. An approach to handle this is using an ensemble of classification models[13]. These models $M_1 \dots M_L$ are each trained on a class c. So if none of the classifiers recognise a new class c then we can consider it as a novel class. The data stream is split up in k chunks and train a k-NN based classifier on each of them. The clusters are built using semi-supervised K-means clustering. For each cluster a summary, or pseudo point, is saved. This summary contains the centroid, radius and frequencies of data points belonging to each class. The radius is equal to the distance between the centroid and the farthest point in the cluster. These pseudo points are the classification model. Each pseudo point is a hypersphere with center and radius of the pseudo point. The union of the hyperspheres is the decision boundary for the classifier. For example, if there is a test instance x which is inside the decision boundary of any model in the ensemble of classifiers it is classified as an existing class. If this is not the case x will be stored in a buffer and is considered an F-outlier. If there are enough outliers in this buffer a classification is used to detect if there is a novel class. If this is the case the F-outliers are tagged as a novel class instance. This classification measures if the data points are closer to its own class (cohesion) and farther apart from other classes (separation). This is done by computing the q-Neighborhood Silhouette Coefficient: $q-NSC(x) = \frac{D_{c_{out},q}(x)-D_{c_{min},q}(x)}{max(D_{c_{min},q}(x),D_{c_{out},q}(x))}$ where $D_{c_{out},q}(x)$ is the mean distance from F-outlier x to its q nearest F-outlier instances, $D_{c_{min},q}(x)$ is the mean distance from x to its q-nearest existing class instances. This results in a value between $[-1, 1]$. If the value is positive it means that x is closer to the F-outlier instances. This q-NSC value is calculated for each of the classifiers in the ensemble. If there are at least $q'(> q)$ F-outliers that have a positive result for all these classifiers a new class has been found.

A new instance might be detected as an F-outlier since it is outside of the decision boundary of a summary. However, it can be that there is concept drift which caused this instance to drift just outside the boundary. So this new instance is outside but still very close to the boundary. This case can happen frequently with concept drift or just noise. To combat a lot of false alarms a slack space is proposed, this slack space has a threshold that can be adjusted (figure 2.8). If a new instance is a false novel instance, which is an existing class that has been detected as a novel class, then it is defined as an outlier. However, if this instance is a marginal false novel, it must have been close so we adjust our OUTTH value so this instance falls within our slack area. This is only done for marginal cases to avoid having large changes to the slack area.

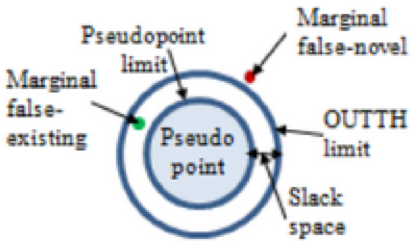


Figure 2.8: A visualisation of the slack space around a summary (or pseudo point) with the OUTH (threshold) limit[13]

The case of multiple new classes is also possible. It is not hard to imagine that for a scenario like Twitter where this can happen constantly. To help detect multiple classes a graph is constructed from the novel instances and the connected components are identified. The number of connected components is the amount of novel classes. The idea behind this is that a class clusters together and thus adheres to the previously mentioned cohesion and separation properties.

This approach proposes a method to detect novel classes in a data stream. Which can be used as a building block for procedures that use these newly detected classes, such as the work previously mentioned that applies evolution to RDFs. With unsupervised detection as proposed here used as a data source to detect evolution without the need for intervention. In the real world outlier cases most likely won't represent themselves neatly. An important aspect to take into account is the time delay with which these cases present themselves. All these cases won't represent themselves all at the same time. Still applying this analysis under time constraints is its own challenge[14].

2.5 Capturing evolving structure of data

There are more ways of adding evolution to an RDF graph or rather dealing with the evolution of the data in the RDF graph. This can happen in a situation where over time more will be known about a subject, which is often the case in the medical field. Old information might be discarded and its place taken by the newly assumed correct information. In cases like this it can be helpful to have information how the data evolved instead of replacing the old data. In order to accomplish this the idea of a snap graph combined with an Evo Graph to handle this is proposed[17]. A snap graph S(V,E) is a directed graph that consists of nodes V, divided into complex and atomic. Atomic nodes are the leaves of the snap graph. An Evo Graph G is a graph that captures all the instances of an evolving snap graph across time. An evo graph has two types of nodes. Data nodes V_D that contain two types, complex V_D^c and atomic V_D^a. The second type are change nodes V_C which represent change events and are also divided in two types, atomic V_C^a and complexV_C^c. Change nodes carry a timestamp of the moment the event instance happened. There are three types of edges. Data edges which depart from all complex data nodes $E_D \subseteq V_D^C \times V_D$. Change edges connect every complex change node to the (complex or atomic) change nodes it encompasses $E_C \subseteq V_C^C \times V_C$. Evolution edges connect each change node with two data nodes, the nodes before and after the change $E_E \subseteq V_D \times V_C \times V_D$. The idea of the evo graph is that it consists of a data graph and a tree of changes of this data graph. So different versions of the data graph are all stored in the evo graph. The different graphs are connected through the evolution edges.

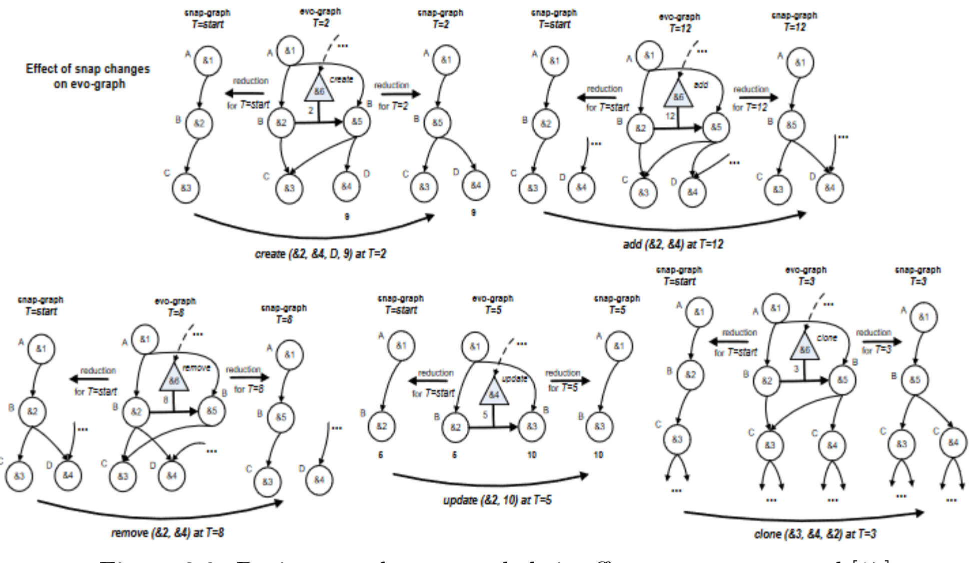


Figure 2.9: Basic snap changes and their effects on an evo graph[17]

There are five basic snap changes possible on the snap graph: create, add, remove, update and clone. The create change $\langle v^p, v, label, value\rangle$ adds a new atomic node v with $label\&value$ and connects it to the parent node v^p. If the parent node is an atomic node it will turn into a complex node since it is no longer a leaf in the graph. The add change $\langle v^p, v\rangle$ adds and edge from v^p to v. Both edges need to exist prior to this change. The remove change $\langle v^p, v\rangle$ removes the edge between v^p and v. If v has no other incoming edges v is also removed and if v^p has no other outgoing edges it becomes an atomic node since it has turned into a leaf node. The update change $\langle v, newValue\rangle$ updates the value of node v to $newValue$. The clone chage $\langle v^p, v^{source}, v^{clone}\rangle$ creates a new data node v^{clone} that has an identical label, value and subtree as v^{source}, the new node is added as a child of v^p so that the nodes v^{source}, v^{clone} become siblings. The effects of these changes can be seen in figure 2.9.

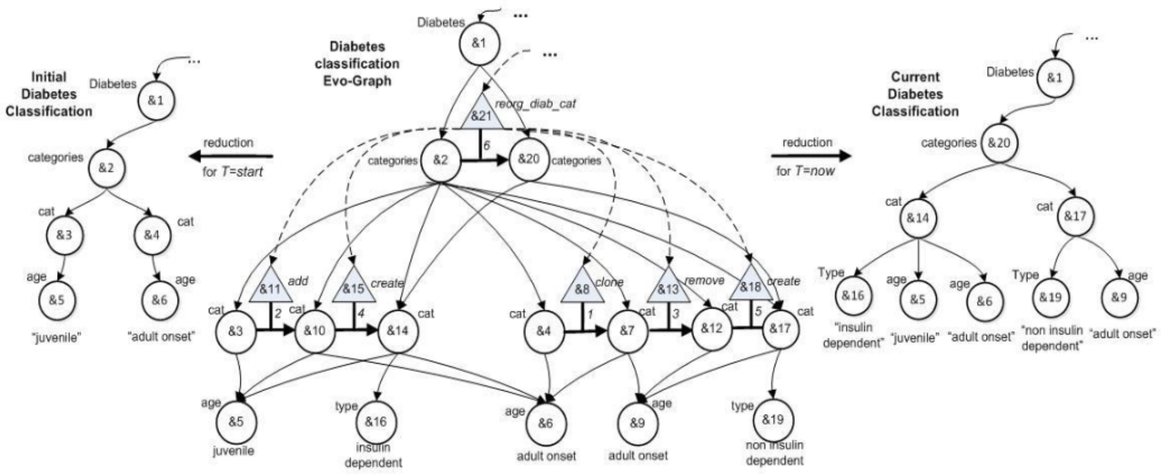


Figure 2.10: Snap graphs of diabetes classification on the left (before) and right (after) with the evo graph in the middle[17]

An example of an evo graph is shown in figure 2.10 in the middle. There are two special nodes, the roots of the graph. The data root at node &1 and the change root at node &21. The figure represents a revision of the diabetes classification. The left graph in the figure is a revision of the data graph in the middle. The revision process is denoted by the *reorg_diab_cat* at node &21. This revision has five basic snap changes at nodes clone at node &8, add at node &11, remove at node &13, create at node &15 and crate at node &18. The timestamp that these changes occurred at are on the change nodes. The graph on the left is the reduction at T=start, which is before any of these changes have occurred. This is a complex changed, which is a collection of basic changes. The change that has taken place here is defined in figure 2.11. The graph on the right is at time T=now which means it is the data graph after all these changes.

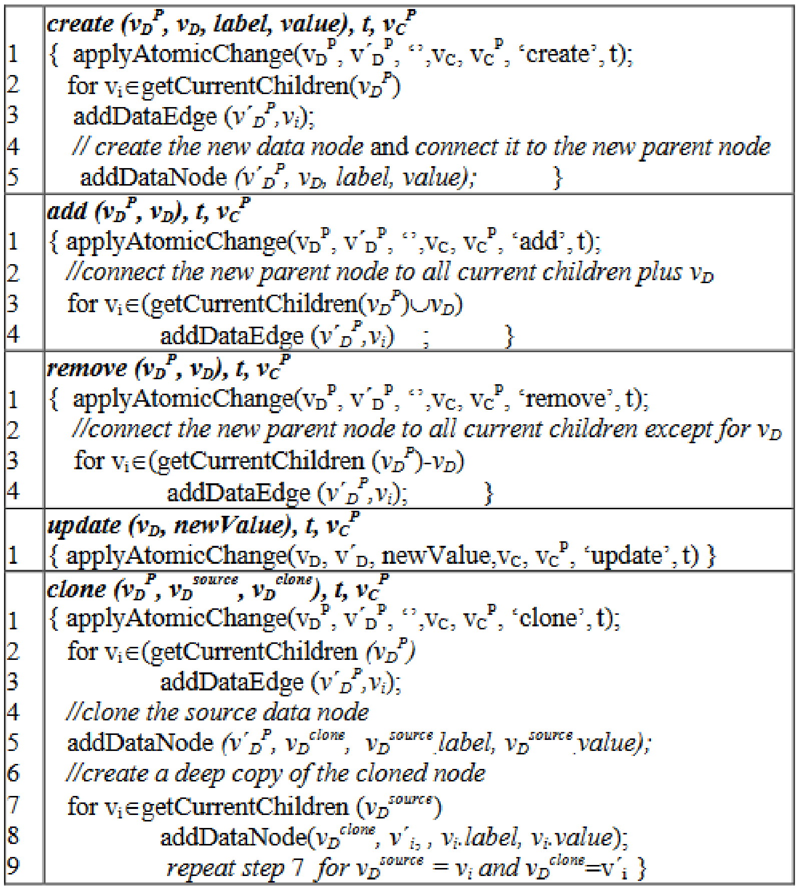
$reorg\_diab\_cat$(&2) {
clone (&4, &6, &9)
add (&3, &6)
remove (&4, &6)
create (&3, &16, “type”, “insulin dependent”)
create (&4, &19, “type”, “non insulin dependent”) }

Figure 2.11: The *reorg_diab_cat* operation expanded for the evo graph in figure 2.10

The changes are applied to the snap graphs, some additional operations need to be added that can be applied to the evo graph itself. There are four operations: addDataNode, addDataEdge, applyAtomicChange and applyComplexChange. Some of these operations are mostly similar to the snap graph changes. The addDataNode $\langle v_D^p, v_D, label, value\rangle$ operation adds a new node v_D to the evo graph as a child of v_D^p. The same logic applies here as to the snap changes where if v_D^p is an atomic node it can change into a complex node. The addDataEdge $\langle v_D^p, v_D\rangle$ creates a new data edge between its two arguments. This operation can also turn atomic nodes into complex nodes. The other two changes evolve nodes as the result of applying a snap change. The applyAtomicChange $\langle v_D^1, v_D^2, value, v_C, v_C^P, label, timestamp\rangle$ first creates a new node v_D^2 that contains the *value* and has the same label as v_D^1. Then a new node v_C is created with the *label* and *timestamp* from the operation which is connected as a child node to v_C^p. The *label* added to this node denotes a snap change. As last step in the operation a new evolution edge is added (v_D^1, v_C, v_D^2) so the nodes are all connected. The applyComplexChange operation $\langle v_D^1, v_D^2, v_C, v_C^P, label, timestamp, \{v_C^1 \dots v_C^n\}\rangle$ first creates a new atomic node v_D^2, which has the same label as v_D^1 but the value of the new node is a default value (empty string). This new node is then added as a child to all parents of v_D^1. Then a new complex change node v_C with the *label* and *timestamp* from the operation is created and added as a child node to v_C^p. The *label* of the operation is the name of a complex change and can be any string. When this is done v_C is connected as a parent of the change nodes $\{v_C^1 \dots v_C^n\}$. As last step an evolution edge is created (v_D^1, v_C, v_D^2) so the created nodes are connected.

With these evo graph operations defined the snap graph operations can also be applied to the evo graph. See table 2.1 how each snap graph operation uses these evo graph operations to apply itself to the evo graph.

This approach shows how to use changes as first class citizens and proposes a structure how to handle this. This way an evo graph can capture how evolution changes the graph structure. A reduction of this evo graph can then be made to retrieve a state of the graph under a certain timestamp.



	create $(v_D^P, v_D, label, value), t, v_C^P$
1	{ applyAtomicChange(v_D^P, $v'_D{}^P$, ‘’,v_C, v_C^P, ‘create’, t);
2	for $v_i \in$ getCurrentChildren(v_D^P)
3	addDataEdge ($v'_D{}^P$,v_i);
4	*// create the new data node* and *connect it to the new parent node*
5	addDataNode *($v'_D{}^P$, v_D, label, value);* }
	add $(v_D^P, v_D), t, v_C^P$
1	{ applyAtomicChange(v_D^P, $v'_D{}^P$, ‘’,v_C, v_C^P, ‘add’, t);
2	*//connect the new parent node to all current children plus* v_D
3	for $v_i \in$(getCurrentChildren(v_D^P)$\cup v_D$)
4	addDataEdge ($v'_D{}^P$,v_i) ; }
	remove $(v_D^P, v_D), t, v_C^P$
1	{ applyAtomicChange(v_D^P, $v'_D{}^P$, ‘’,v_C, v_C^P, ‘remove’, t);
2	*//connect the new parent node to all current children except for* v_D
3	for $v_i \in$(getCurrentChildren (v_D^P)-v_D)
4	addDataEdge ($v'_D{}^P$,v_i); }
	update $(v_D, newValue), t, v_C^P$
1	{ applyAtomicChange(v_D, v'_D, newValue,v_C, v_C^P, ‘update’, t) }
	clone $(v_D^P, v_D^{source}, v_D^{clone}), t, v_C^P$
1	{ applyAtomicChange(v_D^P, $v'_D{}^P$, ‘’,v_C, v_C^P, ‘clone’, t);
2	for $v_i \in$(getCurrentChildren *(v_D^P)*
3	addDataEdge ($v'_D{}^P$,v_i);
4	*//clone the source data node*
5	addDataNode *($v'_D{}^P$, v_D^{clone}, v_D^{source}.label, v_D^{source}.value);*
6	*//create a deep copy of the cloned node*
7	for $v_i \in$getCurrentChildren (v_D^{source})
8	addDataNode(v_D^{clone}, v'_i, , v_i.label, v_i.value);
9	*repeat step 7 for* $v_D^{source} = v_i$ *and* v_D^{clone}=v'_i }

Table 2.1: Steps to apply the snap graph operations to an evo graph using evo graph operations[17]

2.6 Temporal Data Management

Besides querying or detecting evolving data, there can also be improvements sought in the data warehouse design. Many applications that generate a lot of data such as finance whose data is also time dependant benefit from identifying trends by comparing different time states of the data. Analysing these trends can help inform future decisions [20]. If the dimension of time is removed from this data or not taken into account these decisions can be misinformed[3]. However, the proper management of this data is no easy task. Having proper management of this data on its own is difficult due to its potentially evolving nature, but not the sole reason for the difficulties. Since this data can span many years the organisation that generates and uses this data can also evolve. This can be simply because of different ideas or regulations. Which means that the system and the data itself can change over time [16]. These changes need to be handled properly to avoid situations where changes to the system can result in the loss of historical data. A simple change to the data warehouse schema can cause such a loss if the previous structures are deleted. The solution to that is to keep data warehouse versions, so a change to the schema creates a new schema [19].

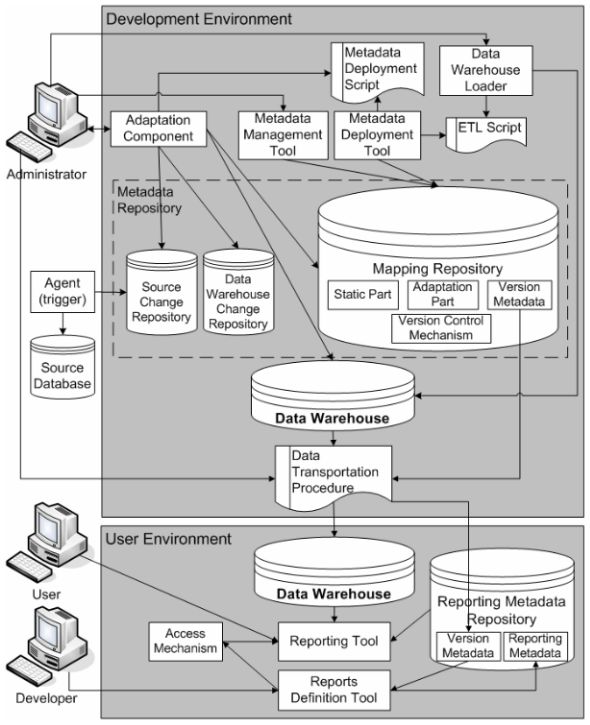


Figure 2.12: Data Warehouse Evolution Framework[19]

A framework to handle this process is shown in figure 2.12. In the development layer the adaption component is the element that processes changes in relations & attributes of the source schemata, identifies potential changes & new versions of the data warehouse, creates a new schema or adapts the current schema based on the settings, creates the needed metadata and adapts data extraction, transformation and loading (ETL) processes. In order to all this the adaption component uses data from the metadata repository. The metadata management tool contains the GUI that is used by the administrator or developer to design the schemas or ETL processes. It contains the static part of the mapping repository, where the metadata of the last data warehouse version and mappings are stored. The adaption part in the mapping repository stores the dependency information of data warehouse elements and source elements which are used for the adaption. The rules for the creation of new data warehouse versions are defined in the version control mechanism. Information about data warehouse versions is stored in the version metadata in the mapping repository. The data warehouse change repository, included in the metadata repository, accumulates the potential changes of a data warehouse schema and version creation options. Agents are added into data sources which can track changes in the source schemata and accumulate them in the source change repository. The metadata deployment tool creates ETL processes based on the metadata from the static part of the mapping repository. These processes get executed by the data warehouse loader. Then the data transportation procedure transfers the data ware house data into the user environment. The version metadata is transferred to the reporting metadata repository. The reporting tool can use the metadata repository to run queries on multiple version of the data.

This shows that it also possible to approach the evolution on a higher level by taking into account that systems around data can change as well as the data.

Chapter 3

Motivating example

Consider a large bank such as Rabobank with a large customer service department. An important task of this department is to handle questions from customers, which is mainly done through a call centre. Since the range of questions customers might have is broad for a big bank, quickly categorising calls to decrease wait times is important. Every call that is received is tagged with a tag which describes the subject. A table can be constructed from these tagged calls. However, the bank is not a static entity and will go through many changes. So the classification of these calls will also change. A subject Rabobank is interested in is a prediction of the expected traffic of a specific tag. A manager of the Rabobank wants to know what the prediction of the tag *Illegal Activities* looks like for the coming month April. This is particularly interesting since in March the decision was made to make changes to the tagging system and create the tag *Illegal Activities.* In order to make a forecast about the volume of the tag *Illegal Activities* a bank employee requests the available data from that tag from their database in table 3.1. By doing so the employee receives all the data that contains the tag *Illegal Activities.* These calls are only from the month March onwards since the tag was created in that month. However, this can potentially be a limited view that does not reflect the real situation. The tag *Illegal Activities* can be related to other tags. This can happen when changes are made to the tagging system and the traffic of certain tags are grouped into a new tag to improve traffic flow. Thus by only receiving data from the month March might give the employee just a small part of what could be a large history. A lot of valuable information can be lost which could have helped the employee make a much more precise prediction.

Forecasting more than only a month is of importance to the manager. They also request a report of the volume of traffic for the tag *Money Laundering* to review its performance. When the employee requests the data for this tag they notice something strange. The traffic suddenly dropped in March, which most likely won't be caused by a disapperance of Money laundering happening anymore. The traffic of this tag was moved to another tag via a change to the tagging system. A better representation of the performance of the tag *Money Laundering* would be by an overview where its traffic went after the changes to get a better picture of the situation. But now the employee can only work with the limited data.

For an organisation as a big bank many changes happen to their systems for a variety of reasons. An employee will not always be aware of these changes and has to work with the limited information that is available to them. These frequent changes can result in a loss of data.

ID	**Date**	**Tag**	**Technical Result**	**Result_Reason**
777000123	20-12-2020	Illegal Actions	Completed	AnsweredByAgent
777000136	25-12-2020	Illegal Actions	Diverted	Unspecified
777000149	30-12-2020	Insurance	Transferred	Unspecified
777000152	4-1-2021	Fraud	Completed	AnsweredByAgent
777000203	8-1-2021	Insurance	Completed	AnsweredByAgent
777000222	25-1-2021	Fraud	Transferred	RoutedTo
777000256	4-2-2021	Mortgage	Transferred	RoutedTo
777000297	10-2-2021	Money Laundering	Completed	ReceivedConsult
777000304	24-2-2021	Phishing	Completed	AnsweredByAgent
777000365	5-3-2021	Illegal Activities	Completed	AnsweredByAgent
777000444	10-3-2021	Criminal Activities	Completed	AnsweredByAgent

Table 3.1: Simplified call transaction database

Chapter 4

Problem Statement

We assume the existence of an infinite set of concepts $\mathcal{C}$, a time domain $\mathcal{T}$ with the special values INF and -INF for which any other time is less or greater, respectively. We also assume an infinite set of attribute names $\mathcal{N}$, an infinite set of values $\mathcal{V}$ and an infinite set of identifiers $\mathcal{O}$. A time period denoted as $[t_1, t_2]$, is a pair t_1,$t_2$$\in$$\mathcal{T}$ with $t_1$$\leq$$t_2$. The set of all the time periods is the set $\mathcal{P}$=$\mathcal{T}$$\times$$\mathcal{T}$.

An attribute is a pair $\langle n, v\rangle$, where $n$$\in$$\mathcal{N}$ and $v$$\in$$\mathcal{V}$. The n is referred to as the attribute name and the v as the attribute value. The set of all the possible attributes is the set $\mathcal{A}$=$\mathcal{N}$$\times$$\mathcal{V}$.

A record is a tuple $\langle id, A, t\rangle$, where $id$$\in$$\mathcal{O}$ is a unique record id, $t$$\in$$\mathcal{T}$ a timestamp, and $A$$\subset$$\mathcal{A}$. A database is a finite collection of tuples. A *query q* is a tuple $\langle c_1 \ldots c_n, [t_1, t_2]\rangle$ where $c_1 \ldots c_n$$\in$$\mathcal{A}$ are the conditions of our query. The answer to the query q is the set of all the records that satisfy the conditions and have a timestamp within the time period, i.e., Ans(q) = {⟨id, ⟨n', v'⟩, t ⟩ — ∀⟨n, v⟩ ∈q: ∃ n=n' ∧ v=v' ∧ t_1 ≤ t ≤ t_2}.

Example 1 *Using a query q* : ⟨⟨*tag, Criminal Activities*⟩, $[1/12/2020 - 1/12/2021]$⟩ *on the database in table 3.1. The expected return of running this query on the database is the record:* ⟨*id=777000444,* ⟨*tag,Criminal Activities*⟩*, [10/3/2021]*⟩*.*

Concepts are not static, they can change through events. These events are used to model evolution. There are three distinct types of events that exist, a creation, an end and a mutation.

Creation is a tuple ⟨t, c⟩, where t is the timestamp and c is the concept.

Ending is a tuple ⟨t, c⟩, where t is the timestamp and c is the concept.

Mutation is a tuple ⟨t, o, d, w$_o$, w$_d$⟩, where t is the timestamp of the event, o is the originating concept, d is the destination concept, w_o is the weight of the origin and w_d is the weight of the destination. The weight of the origin w_o is the ratio of records that were classified as o that are now classified as d. The weight of the destination w_d is the ratio of records that are classified as d which were previously classified as o. Conceptually a mutation event is a situation in which parts of one or more concepts become part of one or more other concepts.

A **transformation** is a set of events that all happen at the same time, but for components only.

Intuitively a creation event correspond with the moment a concept becomes active and the ending event corresponds with the moment that concept stops being active. Mutation events are the links between concepts.

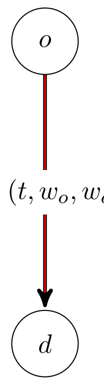


Figure 4.1: Creation, Mutation and Ending

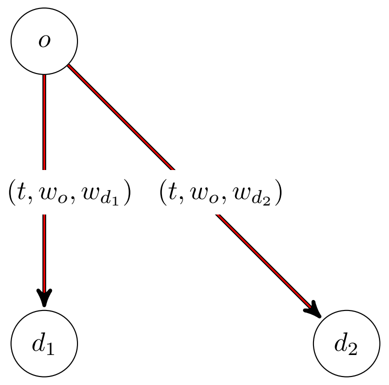


Figure 4.2: Transformation Event

We define a notion of evolution $\langle C, E\rangle$, where C is a set of concepts and E is a set of events. The evolution is well defined if for each $c \in C$ there is one and only one creation event $e_{begin} \in E$ and ending event $e_{end} \in E$. And for each mutation event $e_{mutation}\langle t, o, d, w_o, w_d\rangle \in E$, where $o_{t_{begin}} \leq t \leq o_{t_{end}} \wedge d_{t_{begin}} \leq t \leq d_{t_{end}}$. If the evolution is well defined it can easily be represented as a graph. To model the evolution we introduce a concept evolution graph. A concept evolution graph $G = (V, E)$ is a directed graph. Nodes V are concepts and have two attributes t_1 & t_2, where t_1 the timestamp of the corresponding creation and t_2 the timestamp of the corresponding ending. Edges are mutations, so each edge has a mutation tuple. The situation in table 3.1 is shown in graph 4.3.

If the evolution is well defined a concept $c \in C$ can be valid for any time t if $e_{begin} \leq t \leq e_{end}$.

Example 2 *An example of a concept graph illustrating the situation seen in table 3.1 can be seen in figure 4.3. The values on the edges correspond to the values of the mutations. The names in the nodes are the names of the concepts.*

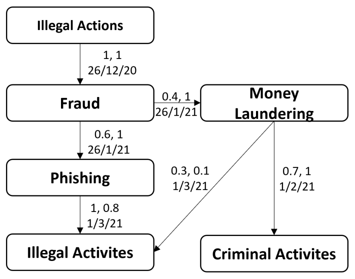


Figure 4.3: Concept Graph Example

ID	**Date**	**Tag**	**Weight**
777000123	20-12-2020	Illegal Actions	0.28
777000136	25-12-2020	Illegal Actions	0.28
777000152	4-1-2021	Fraud	0.28
777000222	25-1-2021	Fraud	0.28
777000297	10-2-2021	Money Laundering	0.7
777000444	10-3-2021	Criminal Activities	1

Table 4.1: Extended Query Result

If we have a query with a concept attribute pair $n = v$ for a period $[a, b]$. If within this period v has evolved to v' or has evolved from v' we want to replace the query with $n = v'$. We replace the period $[a, b]$ with the period $[a', b']$, in which is inside $[a, b]$ and v is v'. When v evolves into v' it does not directly match the v so we want to include a likely hood factor (or weight) w to the set of conditions. So we would like to take the original query $n = v, w[a - b]$ and replace it with $n = v', w'[a' - b']$, where w is the weight or likely hood factor of v in relation to the original query and w' the likely hood factor of v' in relation the original query. Each of these new statements will be valid for a period of time, which is the time when the concept and the query is valid. The query is valid within the time period $[a, b]$. In other words we want to consider every query from a set of conditions. For every evolution of v to v' of our original query we want to have this condition added to our set of queries. Doing this will turn the original query into an evolution aware query by creating a union of queries. A more intuitive example can be seen in example 3.

Example 3 *Our original query in example 1 is* $q : \langle\langle$*tag, Criminal Activities*$\rangle$, $[1/12/2020–1/12/2021]\rangle$*. In this case v is Criminal Activities and the time period* $[a, b]$ *is* $[1/12/2020 - 1/12/2021]$*. If we have the situation where our v evolves into v′ we want to replace the query. If we look at the concept evolution graph in figure 4.3 we can see that our v has indeed evolved. So we replace our original query with* $n = v', w'[a' - b']$*. By looking at the graph we can see that v′ is Money Laundering and the w′ is 0.7. The time period* $[a', b']$ *is that where both the query and the concept are valid, which means that the time period is* $[1/12/2020 - 1/2/2021]$*. So the new query is tag =Criminal Activities, 0.7* $[1/12/2020 - 1/2/2021]$*. However, we can continue this process as we can see that Criminal Activities has also evolved from another concept. The difference in result this will cause is that when our original query q would be applied to the data in table 3.1 there would only be one record. However, by using the concept evolution graph in figure 4.3 and rewriting our query the result should be that of in table 4.1. Which is a much larger result than our original query.*

Chapter 5

Solution

Losing historic data due to the inability to properly access and understand historical data is not an ideal situation. A query that takes concept evolution into account will help combat this problem. The system we propose takes our original query and rewrites it to an evolution aware query with the tools introduced in the previous chapter.

5.1 Concept Graph Traversing

The first step in rewriting a query is to figure out how the concepts in the query evolved. The evolution of these concepts can be found in the concept evolution graph that was introduced in the previous chapter. So in order to figure how a certain concept evolves we need to traverse the concept evolution graph. With this information we can rewrite a query.

To rewrite a query q to take evolution into account it needs to be rewritten to an evolution aware query $e : \langle\langle c_1, w_1\rangle \dots \langle c_n, w_n\rangle,\ [t_1, t_2]\rangle$, where every concept is now a concept-weight tuple indicating the probability that a record with a matching concept belongs to our original query. We use the graph to find the evolution of the concepts in our query. For every concept found in the graph we create an evolution aware query. This results in our original query creating multiple evolution aware queries, since a single concept can have multiple related concepts. On top of this there is the possibility that there are multiple concepts in our original query. In this case there will also be a combination between all related versions of the original concepts. This leads to a single query creating multiple new queries in order to properly take the evolution into account.

The procedure to rewrite the query to an evolution aware query works as follows. The concept evolution graph G is used to find all related concepts by traversing. We need to traverse the graph in both directions, forwards and backwards, since there is a difference in weights used for each traversal. This procedure is also algorithmically explained in algorithm 1. In particular we start with traversing the concept evolution graph for every concept $c \in q$, where q is the original query. So for a concept we start backward and forward traversal, functions for these traversal start on line 1 (Visit_predecessors) & 12 (Visit_successors) respectively. The traversal method traverses the graph in a depth first approach. Each of these traversals keeps a list of visited nodes L, which is later used to construct the evolution aware queries.

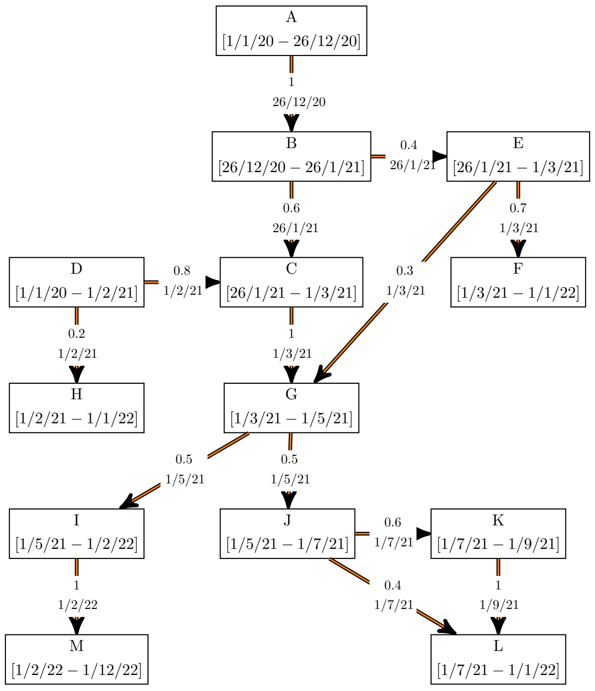


Figure 5.1: Extended Graph Example

Then based on the traversal type we look at all incoming edges for backward traversal or all outgoing edges for forward traversal (line 3 & 14). For each of these edges we check if the timestamp associated with this event or edge is within the time range of our current step of the traversal. The checks for these timestamps are on line 4 & 15. The time range of the traversal starts with the time range of the query and is updated with every new traversal step. This means that the upper bound for backwards and lower bound for forwards traversal are updated with the timestamp of the event for every new step. This is to make sure that no situation can happen where a concept would be regarded as relevant if the timestamp of the event is later or earlier than the time range we reached the node during the traversal. If the timestamp does not fall within the time range we do not take it into account as it does not fall within the scope of the query.

For each of the paths that the algorithm traverses, a weight is kept. This weight can be different for every path since the weight is in relation to the origin of the traversal. When traversing through the graph whenever we reach a new node $x\langle t_c, t_e\rangle$ from our old node c through an edge $e\langle t, w_o, w_d\rangle$ a tuple $\langle x, w_{new}, [t_1, t_2]\rangle$ is added to L, where w_{new} is the current weight of our path adjusted for this node and the time range is when the concept x. The algorithm adds these tuples to our list of expanded concepts on line 2 for backward traversal and 13 for forward traversal. Since this new node is only handled if the timestamp of the edge connecting the our old node with the new node is within our time range we know it is at least valid based on our origin. The time range that is added to L is different as opposed to the time range used for the continued traversal. The time range used for the traversal is in relation to our initial query, the time range that is added to the tuple in L should be in relation of our query and the current node. For backwards traversal this time range should be $[max(t_c, t_1), min(t, t_2)]$, which is the maximum of the time of creation of the concept or the lower bound of the query till the lowest of the event time and the upper bound of the query time range. We take the minimum from those two times since a time range can only be valid if the upper bound is within the query time range and concept lifetime time range. This is the other way around for forward traversal which means that the time range is $[max(t, t_1), min(t_e, t_2)]$. Which is the time range from the maximum of the event time and the lower bound of the query till the minimum of the moment the concept ceases to exist or the upper bound of the query. See example 4 for a visual explanation of this process.

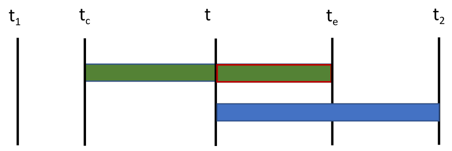


Figure 5.2: Time range visualisation

Example 4 *Take the situation in figure 5.2 where we have two concepts, green and blue, whose lifetime time ranges are visualised as bars. In a situation of backwards traversal for a query that has a time range $[t_1 - t_2]$ and we reach the green concept from the blue concept at time t. To get the correct time range for the green concept in relation to our query is important. If the time range would be that of the query itself it would encompass the entire lifetime of the green concept. However, we only reached the green concept halfway through its lifetime at time t. Any time above time t should not be taken into account, this part is marked with a red outline. This is the upper bound of the time range of the green concept in our query. The lower bound of the time range is going to be the maximum of the lower bound of the query and the creation of the concept, in this case this will be t_c. Thus the time range of this concept is $[t_c - t]$.*

The time range for the traversal also changes each step as was mentioned before. For backwards traversal we adjust the time range downwards since we are looking in the past. This means that our original time range $[t_1, t_2]$ gets adjusted to $[t_1, t]$. The t_1 timestamp will stay the same throughout our traversal and is the lower bound of our query. For forwards traversal we adjust the time range upwards since we are looking at the future from where our traversal started. This means that our original time range $[t_1, t_2]$ gets adjusted to $[t, t_2]$. In forward traversal the upper bound of the query t_2 stays the same during the traversal.

Besides handling the time ranges we also need to update the weight of the path when a new node is reached. When this node is reached the weight of the path has to be adjusted based on our last step. This is done by multiplying the current weight of our path with the weight of the edge that was used to reach the new node. So the new weight is $w_{new} = w * w_e$, where w is the current weight of the path and w_e is the weight on the edge. There are two weights on the edge, namely $w_o \& w_d$. For backwards traversal the weight of the origin w_o is used, this weight is the ratio of items that were once classified as x that are now classified as our new concept c. For forward traversal the weight of the destination w_d is used. The weight of the destination is the ratio of items in our new concept c that were previously classified as x.

The traversal terminates when there are no more edges either incoming or outgoing depending on traversal type or the timestamp of the edge is not within the traversal time range. When the traversal stops the visited list containing all visited concepts with time ranges and weights is returned. However, this list should be edited before it can be used. It is possible for a single node to be visited by multiple paths. When this happens it also creates multiple entries in our visited list. Since this visited list is directly used to create our evolution aware queries it is important that this situation is handled to avoid having duplicate queries. This issue is avoided by making sure there are no duplicate cases in our result, this procedure can be seen on line 23 as the function *Sum_Duplicates*. However, a case is only a duplicate if their concept and time range are equal. So for every tuple $\langle x_1, w_{new_1}, [t_{1_1}, t_{2_1}]\rangle$ in our list L we check it against all other tuples in the list. For every tuple $\langle x_2, w_{new_2}, [t_{1_2}, t_{2_2}]\rangle$ found that satisfies $x_1 = x_2 \wedge t_{1_1} = t_{1_2} \wedge t_{2_1} = t_{2_2}$ we remove the tuple from L and replace the weight in our original tuple $w_{new_1} = w_{new_1} + w_{new_2}$. There is one special case that has to be taken into account, namely the origin of the query. So for all cases where the concepts and time ranges are equal the weights are summed. An example of backwards traversal can be found in example 5.

Backwards traversal

Attributes	**Weight**	**Time Range**
C	1	[26/1/20-1/3/21]
D	0.8	[1/1/20-1/2/21]
B	0.6	[26/12/20-26/1/20]
A	0.6	[1/1/20-26/12/20]
E	0.3	[26/1/21-1/3/21]
B	0.12	[26/12/20-26/1/21]
A	0.12	[1/1/20-26/12/20]
G	1	[26/1/21-1/5/21]

Backwards traversal after combining

Attributes	**Weight**	**Time Range**
A	0.72	[1/1/20-26/12/20]
B	0.72	[26/12/20-26/1/20]
C	1	[26/1/20-1/3/21]
D	0.8	[1/1/20-1/2/21]
E	0.3	[26/1/21-1/3/21]
G	1	[26/1/21-1/5/21]

Forwards traversal

Attributes	**Weight**	**Time Range**
G	1	[26/1/21-1/5/21]
I	0.5	[1/5/21-1/1/22]
J	0.5	[1/5/21-1/7/21]
K	0.3	[1/7/21-1/9/21]
L	0.3	[1/9/21-1/1/22]
L	0.2	[1/7/21-1/1/22]

Forwards traversal after combining

Attributes	**Weight**	**Time Range**
G	1	[26/1/21-1/5/21]
I	0.5	[1/5/21-1/1/22]
J	0.5	[1/5/21-1/7/21]
K	0.3	[1/7/21-1/9/21]
L	0.3	[1/9/21-1/1/22]
L	0.2	[1/7/21-1/1/22]

Table 5.1: Expansion of the query $(c, G)[1/1/20-1/1/22]$ with backwards traversal top and forwards traversal bottom (left) and after removing duplicates with backwards traversal top and forwards traversal bottom (right)

Example 5 *If we have a query* $(c, G)[1/1/20 - 1/1/22]$ *and the graph in figure 5.1 is our concept evolution graph. Performing the graph traversal algorithm on this graph according to the query will start both traversals from the node G. There will be two results, each for one of the traversals, as can be seen in table 5.1 on the left side. The upper table is our backwards traversal and the lower table is the forwards traversal. One thing that stands out in both tables is that certain concepts appear more than once, namely A, B and L. The output of the following step, the removal of duplicates, is shown in the tables on the right. Here we can see the concepts A & B now only appear once since their time ranges were exactly the same and their weights are now summed. The concept L however still appears twice since their time ranges are not the same.*

Algorithm 1: Traverse_graph

```
Input: Concept graph $G$, Concept $q$, TimeRange $[t_1, t_2]$
Result: Predecessors & Successors $\langle P, S \rangle$
Function Visit_predecessors(Concept graph $G$, Concept node $c\langle t_c, t_e \rangle$, List of Expanded Concepts $L$, Weight $w$, TimeRange $[t_1, t_2]$):
    $L \leftarrow L \cup \langle c, w, [t_c, min(t_2, t_e)] \rangle$
    foreach $e\langle t,\ w_o,\ w_d \rangle \in IncomingEdges(G, c)$ do
        if $t_1 \leq t \leq t_2$ then
            $x \leftarrow AdjacentVertex(G, c, e)$
            $w_{new} \leftarrow w * w_o$
            $L \leftarrow L \cup Visit\_predecessors(G, x, L, w_{new}, [t_1, t])$
        end
    end
    return $L$

Function Visit_successors(Concept graph $G$, Concept node $c\langle t_c, t_e \rangle$, List of Expanded Concepts $L$, Weight $w$, TimeRange $[t_1, t_2]$):
    $L \leftarrow L \cup \langle c, w, [max(t_1, t_c), t_e] \rangle$
    foreach $e\langle t,\ w_o,\ w_d \rangle \in OutgoingEdges(G, c)$ do
        if $t_1 \leq t \leq t_2$ then
            $x \leftarrow AdjacentVertex(G, c, e)$
            $w_{new} \leftarrow w * w_d$
            $L \leftarrow L \cup Visit\_successors(G, x, L, w_{new}, [t, t_2])$
        end
    end
    return $L$

Function Sum_Duplicates(List of Expanded Concepts $L$):
    $R \leftarrow \{\}$
    $X \leftarrow \{\}$
    foreach $\langle c_1, w_1, [t_{1_1}, t_{2_1}] \rangle \in L$ do
        if $\langle c_1, [t_{1_1}, t_{2_1}] \rangle \in X$ then
            foreach $\langle c_2, w_2, [t_{1_2}, t_{2_2}] \rangle \in R$ do
                if $c_1 = c_2 \wedge t_{1_1} = t_{1_2} \wedge t_{2_1} = t_{2_2}$ then
                    $w_2 \leftarrow w_2 + w_1$
                end
            end
        else
            $R \leftarrow R \cup \langle c_1, w_1, [t_{1_1}, t_{2_1}] \rangle$
            $X \leftarrow X \cup \langle c_1, [t_{1_1}, t_{2_1}] \rangle$
        end
    end
    return $R$
$P \leftarrow$Sum_Duplicates(Visit_predecessors($G, q, \{\}, 1, [t_1, t_2]$))
$S \leftarrow$Sum_Duplicates(Visit_successors($G, q, \{\}, 1, [t_1, t_2]$))
return $\langle P, S \rangle$
```

5.2 Query rewriting

When the traversal procedure is completed we do not yet have our complete answer. If there are multiple concepts in our original query we need to combine concepts from the traversal to get a complete answer. So, after the traversal procedure of the graph we will have two sets of tuples ⟨c, w, $[t_1, t_2]$⟩ for each concept in our query q, one set for forwards traversal T_f and one set for backwards traversal T_b. This procedure is also algorithmically explained in algorithm 2. In particular, to rewrite our original query to an evolution aware query we need to combine each of these concepts with the other concepts in the set. This is done by continuously taking the Cartesian product of the tuples of different concepts, as can be seen on line 15 & 26. For each of the sets of concepts in our query $\langle c_1 \dots c_n, [t_1, t_2]\rangle$ we have a list of tuples with related concepts. We start with the list of concept c_1 (line 11 & 12), if there are any other concepts in the query we iteratively build up the list of results. Combining the list for forwards and backwards traversal is done differently, see figure 5.3. When taking the Cartesian product of two sets we combine the tuples within the lists. When combining the tuples the concepts and weights of two tuples can be combined without further work. However, the time ranges of the two tuples need to be correctly handled, which is the point that differs between the two traversal methods. For backwards traversal we take the maximum of the two lower bounds and minimum of the upper bound as can be seen on line 18. For forwards traversal we take the minimum of the lower bounds and the maximum of the two upper bounds as can be seen on line 29. Doing this step can create a time range that is not valid due to $t_1 > t_2$. In order to discard these combinations we check if $t_1 < t_2$ as can be seen on line 19 & 30.

$$T \times T$$
$$\langle c_1, w_1,\ [t_{1_1}, t_{2_1}]\rangle \times \langle c_2, w_2,\ [t_{1_2}, t_{2_2}]\rangle$$
For T$_b$:
$$\langle c_1, w_1\langle c_2, w_2\rangle,\ [max(t_{1_1}, t_{2_1}), min(t_{1_2}, t_{2_2})]\rangle$$
For T$_f$:
$$\langle c_1, w_1\langle c_2, w_2\rangle,\ [min(t_{1_1}, t_{2_1}), max(t_{1_2}, t_{2_2})]\rangle$$

Figure 5.3: Query rewriting procedure

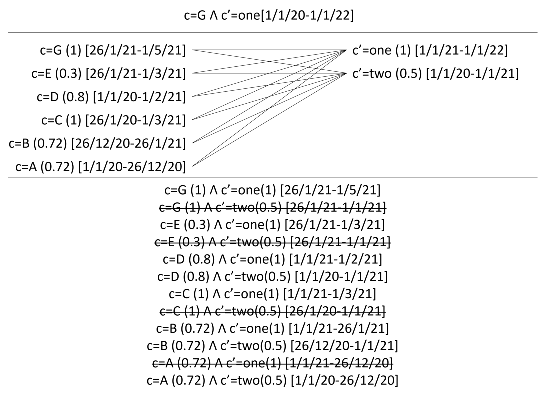

c=G ∧ c’=one[1/1/20-1/1/22]
c=G (1) [26/1/21-1/5/21]
c=E (0.3) [26/1/21-1/3/21]
c=D (0.8) [1/1/20-1/2/21]
c=C (1) [26/1/20-1/3/21]
c=B (0.72) [26/12/20-26/1/21]
c=A (0.72) [1/1/20-26/12/20]
c’=one (1) [1/1/21-1/1/22]
c’=two (0.5) [1/1/20-1/1/21]
c=G (1) ∧ c’=one(1) [26/1/21-1/5/21]
~~c=G (1) ∧ c’=two(0.5) [26/1/21-1/1/21]~~
c=E (0.3) ∧ c’=one(1) [26/1/21-1/3/21]
~~c=E (0.3) ∧ c’=two(0.5) [26/1/21-1/1/21]~~
c=D (0.8) ∧ c’=one(1) [1/1/21-1/2/21]
c=D (0.8) ∧ c’=two(0.5) [1/1/20-1/1/21]
c=C (1) ∧ c’=one(1) [1/1/21-1/3/21]
~~c=C (1) ∧ c’=two(0.5) [26/1/20-1/1/21]~~
c=B (0.72) ∧ c’=one(1) [1/1/21-26/1/21]
c=B (0.72) ∧ c’=two(0.5) [26/12/20-1/1/21]
~~c=A (0.72) ∧ c’=one(1) [1/1/21-26/12/20]~~
c=A (0.72) ∧ c’=two(0.5) [1/1/20-26/12/20]

Figure 5.4: Example of the combination for tuples found through backwards traversal of the query $c = G \wedge c' = one[1/1/20 - 1/1/22]$

This procedure is done for every combination created by taking the Cartesian product. Doing this yields a new list of combinations with adjusted time ranges such as can be seen in figure 5.4. During this process impossible combinations can be created as mentioned before. These impossible combinations have been crossed out in the figure. The new list that is created this way is used for the next combination procedure. This will cause every valid combination to be created since only valid combinations are kept. The end result of the combination is every valid combination between concepts and their related concepts. This combination includes a weight for every concept in the resulting query indicating the ratio records with this concept belong to the original concept in our query. Included as well is the time range where that specific query is valid. When all combinations are made for both backwards and forward traversal tuple sets we combine both sets of combinations. This combined set is the set of our rewritten evolution aware queries. A thing to note here is if there is only one concept in the original query there is no combination between concepts needed as there is only one set in T_b & T_f and they can be combined immediately.

Both T_b & T_f contain one identical tuple, that of the origin of the traversal. This tuple is needed in both sets in order to create all possible combinations in both directions. However, this will cause there to be duplicates in the final set of queries. So as a final step after combining is done, is removing duplicates from the final result to avoid having these duplicate queries.

Algorithm 2: Concept algorithm

```
Input: Concept graph $G(V,E)$, Query $Q$, TimeRange $[t_b,t_e]$
Result: Rewritten queries $R$

$P \leftarrow \{\}$
$S \leftarrow \{\}$
$Concepts \leftarrow \{\}$
foreach $\langle c,a\rangle \in Q$ do
    $P[c], S[c] \leftarrow$ Traverse_graph$(G, a, [t_b, t_e])$
    $Concepts \leftarrow Concepts \cup c$
end

$x \leftarrow pop(Concepts)$
$K \leftarrow P[x]$
$Q \leftarrow S[x]$
foreach $c \in Concepts$ do
    $C \leftarrow \{\}$
    $K \leftarrow K \times P[c]$
    foreach $\langle\langle a_1,[t_{1_1},t_{2_1}]\rangle,\langle a_2,[t_{1_2},t_{2_2}]\rangle\rangle \in K$ do
        $B \leftarrow a_1 \cup a_2$
        $T\langle t_1,t_2\rangle \leftarrow [max(t_{1_1},t_{1_2}), min(t_{2_1},t_{2_2})]$
        if $t_1 < t_2$ then
            $C \leftarrow C \cup \langle B,T\rangle$
        end
    end
    $K \leftarrow C$

    $D \leftarrow \{\}$
    $Q \leftarrow Q \times S[c]$
    foreach $\langle\langle a_1,[t_{1_1},t_{2_1}]\rangle,\langle a_2,[t_{1_2},t_{2_2}]\rangle\rangle \in Q$ do
        $B \leftarrow a_1 \cup a_2$
        $T\langle t_1,t_2\rangle \leftarrow [min(t_{1_1},t_{1_2}), max(t_{2_1},t_{2_2})]$
        if $t_1 < t_2$ then
            $D \leftarrow D \cup \langle B,T\rangle$
        end
    end
    $Q \leftarrow D$
end
$R \leftarrow$Remove_Duplicates$(K \cup Q)$
return $R$
```

Chapter 6

Implementation

Besides the traversal and the query rewriting the system there have been more aspects that will be discussed in this chapter. The structure of the system, the data structures and the supporting functions will be discussed.

6.1 Architecture of the system

The system architecture can be seen in figure 6.1. The system contains two main modules that process the query, which is our user input. An input data set in the form of a JSON file is used to construct a concept evolution graph. This graph is used by the query rewriting module to rewrite queries. The graph is also used by the visualisation to create visualisations of the complete or partial graph. The query rewriting model will take take the query and rewrite it to a set of rewritten queries. A partial visualisation can also be created based on the query after being rewritten.

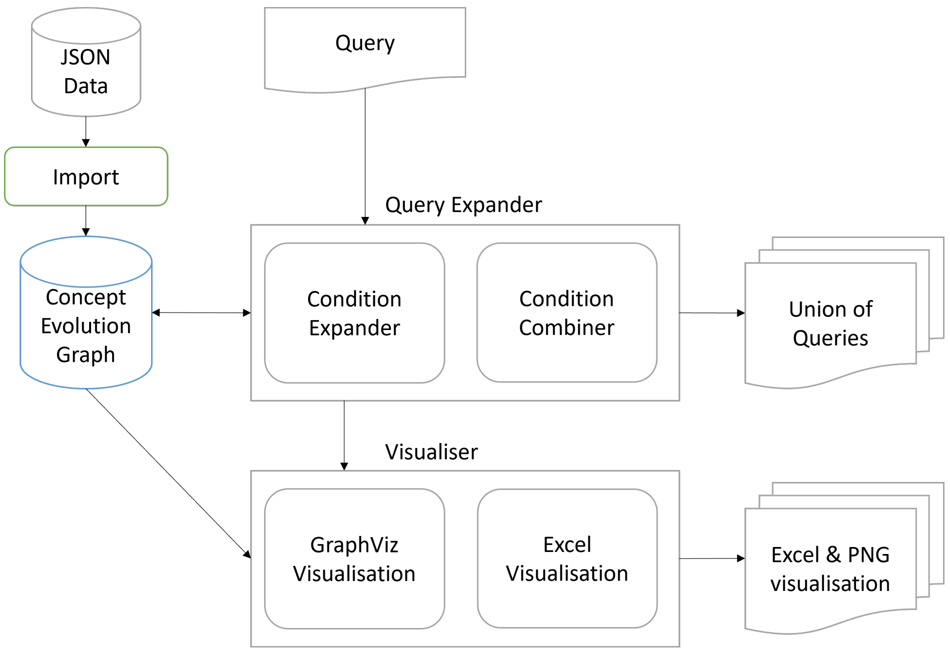


Figure 6.1: System architecture

6.2 File Format

The concept evolution graph is constructed from a JSON file. Every entry in the JSON file is an event, the entry contains 3 fields ⟨Event, Timestamp, Value⟩. Event contains the type of the event, so Creation, End, Mutation or Transformation. Timestamp corresponds to the timestamp t of an event. Value contains all values of an event, excluding the timestamp.

```
{ID:
      {
       ”Event”: string,
       ”TimeStamp”: string,
       ”Value”: [
             ”Origin”: string,
             ”Destination”: string or null,
             ”Weight_Origin”: float or null,
             ”Weight_Destination”: float or null]
      }
}
```

Figure 6.2: JSON file format

6.3 Import Function

Importing the JSON file is straightforward. Every entry is read and added to our data. The only step that is done here besides the importing is converting the date strings to datetime objects. Besides the data in each entry the index is also saved. The index of the entry will function as the ID of the event. When the file has been imported a data set where each entry is of the form ⟨Event, Timestamp, Values, ID⟩. This data set can be used to construct the concept evolution graph.

6.4 Query Handling

Since we apply an abstraction to the data in the graph by converting all concept names to IDs, we also need to apply this procedure to the input query. This is mostly straightforward as we have a concept Look up table (LUT) where we can find the ID of any concept name if present in the graph. However, in the case of duplicate concept names the situation gets somewhat more complicated. Since there is no certain way to know based on a query which of the instances of a certain concept is targeted by a query we take all of them into account. This means that if there are multiple concepts with the same name in the graph the system will run an input query for each of them. So for every ID linked to the concept name a query is run.

6.4.1 Building the Graph

Using the imported data we can construct the graph. Constructing the graph is not as straightforward as it seems as we apply an abstraction here. To make it possible for having two or more concepts with the same name, we will give each concept an ID. When there are multiple concepts of the same ID, we will give them a different ID for every time range they are active. This is easily done as the import function makes sure that every event has an ID associated with it. These IDs will all be unique and in order for there to be two concepts with the same name it requires two different begin events. There is one important restriction to this, the active time of these similar concepts cannot overlap. Which means that given concepts a & b, where $a_{begin} < b_{begin}$, the following statement must be satisfied $a_{begin} < a_{end} < b_{begin} < b_{end}$. As our algorithm does not care if the nodes in the graph are names or numbers we can simply make this swap in the input without having to complicate the rest of the system. The output however should be managed correctly. For all the different output the IDs need to be switched out for the original names, for obvious readability and usability reasons. In order to accomplish this we will keep a lookup table that links IDs back to their original names and vice versa to link names to their IDs.

With this abstraction in mind we can convert the data that has been imported to a graph format. But first there are some basic restrictions that apply to all events of a concept to ensure that the correct flow of time is preserved. The events of a concept a have to satisfy $a_{begin} < a_{end}$, as well as for each mutation event that contains concept a with timestamp t has to satisfy $a_{begin} < t < a_{end}$. To make sure we will be able to handle all data in the correct order we will sort the imported data based on the timestamp so we can start with the first event.

Our system is implemented using Python and to model the graph we use the python library NetworkX[1]. This library allows us to iteravely add nodes & edges to a graph and change the information on them at any time. The first step of adding data to this graph is done by looping through the data and handling begin and ending events. When we encounter a begin event we have to make the earlier discussed abstraction so we will add a node with the event ID. As data we add the timestamp of the begin event to the node. Here we add the link between this ID and the concept name to our lookup tables. We will only handle this begin event if there is no other begin event with the same name active at the timestamp of this begin event. Otherwise we are dealing with incorrect data and will stop the graph creation. When we find an ending entry we find the corresponding begin node through our lookup tables. The information from the lookup tables allows us to add the ending timestamp to the node in the graph. Here it is important to check if the timestamp of our begin event is before our ending event timestamp. One important thing to note here is that not all concepts will have an ending event. If a concept is still active at the time of creation there will not be an ending event. These concepts will not have an ending time. However, since we have handled every begin and ending event at this point we always assume that if a concept does not have an ending event the ending timestamp is that of *now*. The *now* timestamp constitutes the timestamp at the moment of evaluation. So for every part of the system will handle a missing ending timestamp as a *now* timestamp.

After doing this procedure we will have a graph with all the nodes in our data. The reason we will first construct all the nodes before adding any edges to the graph is that we want to make sure that every mutation event satisfies its conditions as mentioned earlier. Each mutation event in our data has an origin and a destination. We first have to translate these names to IDs. To do this we will lookup the ID of the concept that was active during the timestamp of the mutation. Since there is only one possible concept with a specific name active at the same time this lookup will always have one or no results. If there are no results we are again dealing with faulty data and stop the graph creation process. If the lookup process is successful for both our origin and destination we will add an edge in the graph linking the corresponding nodes. The edge will contain the values of the mutation event. We will also add the mutation ID to edge, which allows for better identification during the creation of the visualisation and output functions.

When each mutation event is handled successfully the graph is created. To keep only relevant information in the graph we will also keep our concept & ID lookup tables. These lookup tables allow us to change IDs to the names of concept at the last step of any output of the system.

6.4.2 Structures in memory

Edge: Each edge contains an *origin*, a *destination*, an *origin weight*, a *destination weight*, a *mutation ID* and a *timestamp*. The origin and destinations are *concept IDs*. Timestamps are datetime objects, weights are a number in the range of [0,1] and IDs are unique identifiers.

Node: Each node contains a *begin timestamp* and an *ending timestamp*.

Graph: The graph contains *nodes* and *edges*.

Query: A query is a tuple with three items, a list of *concept attribute pairs*, a *begin timestamp* and an *ending timestamp*. The *concept attribute pairs* are strings and the *timestamps* are date time objects.

Concept LUT: The node lookup table is a list of key value pairs, where the keys are *concept names* and the values are tuples of *begin timestamp*, *ending timestamp*, *concept ID*.

ID LUT: The ID lookup table is a list of key value pairs where the keys are a *concept ID* and the value is the *concept name*.

6.5 Export function

Our system also includes an export function for the graph. This function will convert the current graph into JSON file according to the structure mentioned earlier. The networkX library allows us to loop through all our nodes and edges. For each node in our graph we convert the data contained in the node to potentially two entries. The first entry is that of a begin event which will be the type of our entry, where we use the begin time of the node as the timestamp and the name of the concept as the value. We can get the name of the concept based on the ID of the node and the lookup table. If there is an ending timestamp in the node we will also add an ending entry. Which is the same as the begin entry, but has a different type and uses the ending timestamp as timestamp. For each edge in the graph we add a mutation entry to our output, where the timestamp is the timestamp of the edge. The value of the entry is that of the origin, destination, weight of the origin and weight of the destination. Here the names of the origin and destination are converted from their ID format to their names using the lookup table. After both these procedures are done the graph is converted to a JSON file and can be exported.

6.6 Visualisation

Understanding evolution is important, the understanding of which can be made easier by visualising what is going on. The phrase "A picture says more than a thousand words" exists for a reason. There are two different visualisations we include. A graphical GraphViz representation that can clearly show the graphical representation comparable to the graph figures shown in the previous chapters. The other visualisation focuses on the showing the time flow of the concepts and is encapsulated in an Excel graph. Taking this dual approach allows for visualisations to show that the evolution of the data and the time flow in a format that supports these categories better than it would have been in a singular format.

6.6.1 Graphviz visualisation

The structure of our graph is captured by the networkX library, which has an export function to Graphviz [5]. There are two different Graphviz exports possible for our system. The complete graph with every available node and the partial graph specific for our query. The partial graph contains every concept that is related to the concepts in our original query. These concepts are compared to a copy of the complete graph, any node in this graph is compared the list of related concepts. If the node does not appear in the related concepts list it is removed from the graph. The result of this process is a graph that only includes nodes that are in our related concepts list. Edges are automatically removed from the graph if either one of the nodes it is connected to is removed. This will ensure only the mutation events that are relevant to our list of related concepts remain in the partial graph. An example of a partial graph can be seen in figure 6.3. When this graph or the complete graph is created the graphs can be written to image files as output and shown.

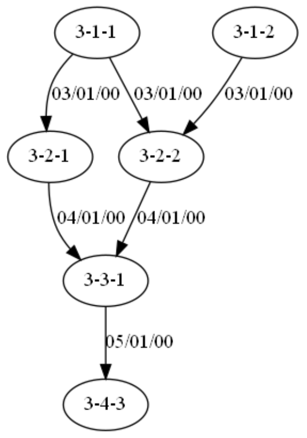


Figure 6.3: GraphViz visualisation with mutation event dates on edges and concept names in nodes

6.6.2 Excel Visualisation

The Excel visualisation is different from the Graphviz visualisation as it does not show the layout of the graph like the traditional visualisations. The Excel graph visualisation instead has as main goal to better illustrate the time flow of events of our data set. The structure of the Excel graph emphasises how the evolution takes place set out against the time and mainly uses background colours of a cell to illustrate this. To achieve this every column constitutes to a specific day. Since it is possible to have multiple events happen at the same day multiple columns can be attributed to the same day. Every other row is used by a single concept. To create this in the visualisation we write to an Excel worksheet with the $worksheet.write(row, col, text, style)$ function, where row is the row index, *col* is the column index, string is the *text* written to that cell and *style* is the style on the cell (background colour, border, font colour etc.).

The concept row corresponds with a node in the graph. Each concept row has three different parts to its structure: the begin header, the end header and the life time bridge. The begin header corresponds to a creation event, so the timestamp of the begin header is that of the creation event. The begin header is placed on the row of the concept and the column of the timestamp from the corresponding creation event. The begin header will also contain the name of the concept and the time range as text. The ending header corresponds to the timestamp of the ending event of the concept. The lifetime bridge is each column in between the begin and end header on the same row as the two headers.

With nodes added to the visualisation there are only edges left that need to be added. Edges are represented as mutation columns, which follow a similar structure as a concept row: origin header, destination header and evolution bridge. Since a mutation columns represent edges from the graph the origin and destination structures are based on the origin and destination from the corresponding edge. An origin header is placed on a column that corresponds with the timestamp of that can also be retrieved from the edge and the row corresponding to the row of the origin concept of the edge. The destination header is placed on the row corresponding with the row of the destination concept from the corresponding edge and placed in the same column as the origin header. The evolution bridge is every row in between the origin and destination header on the same column as the two headers.

At the top of the visualisation is a date header of three rows each showing a different granularity in the time. The first row shows the year, the second row shows the month and the third row shows the number of the day in the month. The header will only show a specific year, month or date only once. So in the case of multiple events on the same day creating multiple columns the header will not repeat itself.

The goal of this visualisation is to show the events in the graph related to the flow of time. Which puts an emphasis on the drawing of the visualisation and especially the order, the algorithm to create the Excel visualisation is found in algorithm 3. Some preparation work has to be done to properly create the visualisation. Since the visualisation is based on the graph, we will first read out all the information in the graph. For all nodes we extract the begin and ending timestamp as well as the name of the concept of the node, this process is done on line 2 through 9 in the algorithm. We add both of these times bundled with the concept name separately to a list of our events and tag each of them with the proper event, which is either begin or ending. As an extra value to both the begin and ending tuples we will add the node id. This extra value is extended for the begin value where the ending time of the node is also added. This is done so we can add the time span of the concept.

For each edge we extract the origin, ID, destination and the timestamp. We will add a new entry to the list with the timestamp to the list, tagged with the mutation tag. As extra value we will add the destination, origin and the ID of the edge. This process is on lines 10 through 12 in the algorithm. Doing this will create a collection of the data in the graph. The next step is to sort this data based on the timestamp of each entry. This will turn the list into a chronological list of our events.

With the preparation work out of the way we can start the first step of drawing the visualisation.

We start by looping through the data. When doing this, entries in our list can have the same date, if this is the case we will only draw a background to our header. If the date is different than our previous date we will update the header accordingly. This means that if we have reached a new year, month or day of the month the header is updated accordingly (line 17). During the looping of our data we will keep two indexes, the row index and the column index (line 14). The header will only draw itself based on the column index but its rows are set to 1,2&3, based on the granularity. Which also means that our first non header structure we draw starts at row 5, we will leave a gap of 1 row in between each row for readability and apply the same to the header and the first row. Every time we encounter a different date we increment the column index by 1, which can be seen in lines 18 through 21. On line 24 we write in our Excel worksheet on the correct row & column combination. We also write the concept name and time span of the concept in this cell. The last argument of every worksheet write operation are static style objects, which determine things such as the background colour and border style of the cell. The column index will also be incremented if we encounter a mutation entry. We do this to separate all mutation columns for better visibility.

In our first loop through the data whenever we encounter an entry tagged with as begin, we will draw the begin header on the location of the current row and column indices. In addition to this we will add the name and time range to this cell, since we added both the begin and ending time to our begin entries. After drawing the begin header we will increment our row index by two. We will keep track of what row corresponds to this concept so we are later able to draw on this row when we encounter events associated with this concept. This process can be found in lines 22 through 27. When we encounter an entry tagged as ending, we will draw the ending header. Since we saved the row where the begin header of this ending entry was drawn at we can draw the ending header at the current column index and the saved row index. Besides drawing the end header we will also keep a list on what column the ending header is drawn so we can later add in the lifetime bridge without having to loop through the entire data collection again (lines 28-31). Whenever we encounter a mutation event during our first loop we note the current column index and save it in a list so we can create the mutation columns later. In addition to this we will also draw a mutation header under the date header. This will signify that on a certain date a mutation has taken place to contrast these columns to the columns that only contain begin of end headers. In addition the header can also contain the ID of the mutation. This ID corresponds to the ID of the mutation from the input. (lines 32-35).

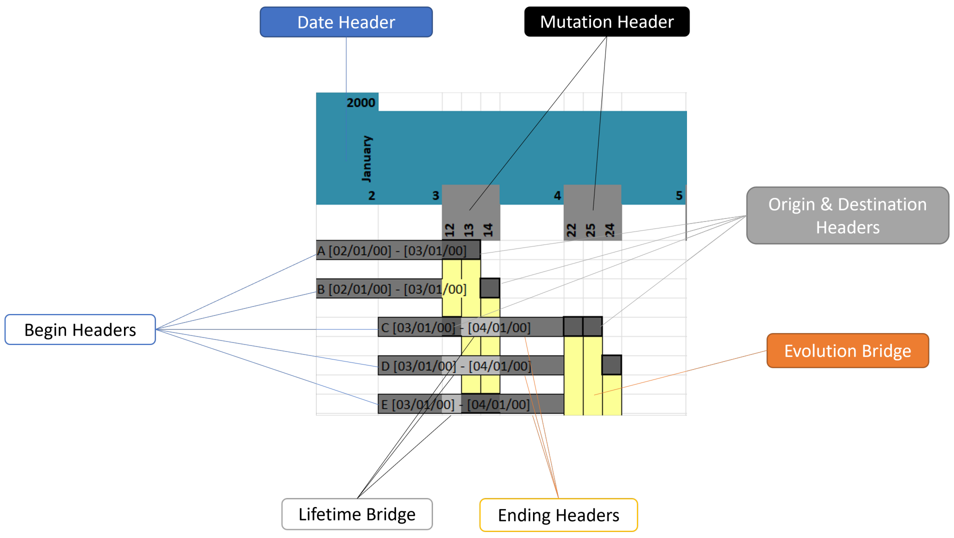

Date Header
Mutation Header
2000
January
2
3
4
5
12
13
14
22
25
24
Origin & Destination Headers
A [02/01/00] - [03/01/00]
B [02/01/00] - [03/01/00]
C [03/01/00] - [04/01/00]
D [03/01/00] - [04/01/00]
E [03/01/00] - [04/01/00]
Begin Headers
Evolution Bridge
Lifetime Bridge
Ending Headers

Figure 6.4: A partial Excel visualisation with each structure and element pointed out.

After our first loop we have drawn the begin & end headers and know what row belongs to each concept. With this knowledge we can draw the structures for the mutation events. Each of these entries contain an origin, destination and timestamp. Since we know the row of each concept and have earlier identified the column of each mutation it is easy to draw the headers on the right spot. The origin header is drawn at the row of the origin concept and the column of the mutation. The destination header is drawn at the row of the destination concept and the column of the mutation. Each row at the column of the mutation in between these headers is then drawn as an evolution bridge. This procedure is found on lines 37 through 45.

The last step is to draw the lifetime bridges of our concepts (lines 46-50). Since we have earlier determined the begin column, the ending column and the row of our concepts it is simple to draw this last structure. So in between every begin and end header we will draw a life time bridge. This is drawn last to ensure it is on after the evolution bridges to make sure the lifetime bridges of the concepts are the most clear. We will also set the width of the column as small as possible without removing the possibilities to read numbers to improve the presentation of the graph.

Executing this entire process will create the Excel visualisation. An example of a partial Excel visualisation can be seen in figure 6.4. In this figure each structure is pointed out for easy identification. There you can see the begin headers and how they have the concept name as well as their time range for when they are valid. The start of the date header in this picture starts at the second of January in the year 2000 and goes until the fifth of January, but as can be seen here only the days are noted here as they are the only date variables that change. Worked into the header are the mutation headers, with the mutation ID in them. This layout allows for easy identification where the mutation take place in the timeline. The layout not only allows for easy identification when the mutation takes place but also between which concepts. If we examine the mutation with ID *12*, we can see that on the third of January a mutation has taken place. This mutation connects concept A with concept C by looking at the Origin & Destination headers and the connecting evolution bridge.

Algorithm 3: Excel visualisation algorithm

```
Input: Concept graph $G(V, E)$, ID Lut $I$, Excel worksheet $worksheet$
Result: Excel worksheet $worksheet$
1  $event\_list, concept\_rows, mutation\_columns, concept\_start\_column, concept\_end\_column \leftarrow \{\}$
2  foreach $node\langle id, t_b, t_e\rangle \in V$ do
3      $event\_list \leftarrow event\_list \cup \langle ''Begin'', t_b\langle t_e, id\rangle\rangle$
4      if $t_e$ then
5          $event\_list \leftarrow event\_list \cup \langle ''Ending'', t_e\langle id\rangle\rangle$
6      else
7          $event\_list \leftarrow event\_list \cup \langle ''Ending'', t_{now}\langle id\rangle\rangle$
8      end
9  end
10 foreach $edge\langle t, o, d\rangle \in e$ do
11     $event\_list \leftarrow event\_list \cup \langle ''Mutation'', t\langle o, d, id\rangle\rangle$
12 end
13 $event\_list \leftarrow sort(event\_list)$
14 $row\_index \leftarrow 5$; $column\_index \leftarrow 0$
15 $current\_date \leftarrow date.min()$
16 foreach $\langle k, t\langle v\rangle\rangle \in event\_list$ do
17     $Update\_header()$
18     if $current\_date \neq t$ or $k = ''Mutation''$ then
19         $column\_index \leftarrow column\_index + 1$
20         $current\_date \leftarrow t$
21     end
22     if $k = ''Begin''$ then
23         $row\_index \leftarrow row\_index + 2$
24         $worksheet.write(row\_index, column\_index, I[v_{id}][t, v_{t_e}], begin\_header\_background)$
25         $concept\_rows \leftarrow \cup concept\_rows \cup \langle v_{id}, row\_index\rangle$
26         $concept\_start\_column \leftarrow concept\_start\_column \cup \langle v_{id}, column\_index\rangle$
27     end
28     if $k = ''Ending''$ then
29         $concept\_end\_column \leftarrow concept\_end\_column \cup \langle v_{id}, column\_index\rangle$
30         $worksheet.write(concept\_rows[v_{id}], column\_index, '''', ending\_header\_background)$
31     end
32     if $k = ''Mutation''$ then
33         $mutation\_columns \leftarrow mutation\_columns \cup \langle v_{id}, column\_index\rangle$
34         $Update\_Mutation\_Header(v_{id})$
35     end
36 end
37 foreach $\langle k, \langle v\rangle\rangle \in event\_list$ do
38     if $k = ''Mutation''$ then
39         $worksheet.write(concept\_rows[v_o], mutation\_columns[v_{id}], '''', origin\_header\_background)$

40         $worksheet.write(concept\_rows[v_d], mutation\_columns[v_{id}], '''', destination\_header\_background)$

41         for $i = concept\_rows[v_o], i < concept\_rows[v_d], i++$ do
42             $worksheet.write(i, mutation\_columns[v_{id}], '''', evolution\_bridge\_background)$
43         end
44     end
45 end
46 foreach $\langle id, row\rangle \in concept\_rows$ do
47     for $i = concept\_start\_column[id], i < concept\_end\_column[id], i++$ do
48         $worksheet.write(row, i, '''', lifetime\_bridge\_background)$
49     end
50 end
51 return $worksheet$
```

Chapter 7

Experiments

In order to test the performance of the system we identify two different parts of the system. The first part we will test is the import functionality, which covers the reading and importing of the JSON file and the creation of the concept evolution graph from the imported data. The second part is the query answering time and how it relates to answer size and nodes visited in the graph. The experiments are executed using an Intel Core i7-5500U CPU @ 2.40GHz (4 CPUs) and 8192MB of RAM.

7.1 Import performance

To test the performance of the import function we compare the time it takes to import different sizes of graphs. In order to do this we randomly create graphs of certain sizes, where the size of the graph is measured by the amount of nodes in the graph. We will randomly connect nodes in the graph so that the amount of edges will always be roughly one and a half times the amount of nodes in the graph. We measure the performance of the import function by measuring the time it takes to import the file and parse the data so our graph creation algorithm can be used. We will start by using a graph size of 100 nodes and scale up the size by a factor 10. By generating each graph size 10 times and taking the average of the time elapsed we get the result shown in figure 7.1. In this figure we can see that increasing the graph size by a factor 10 leads to the same increase in import function time. Even large graph of a million nodes are able to be processed in a reasonable time of slightly over 1 minute.

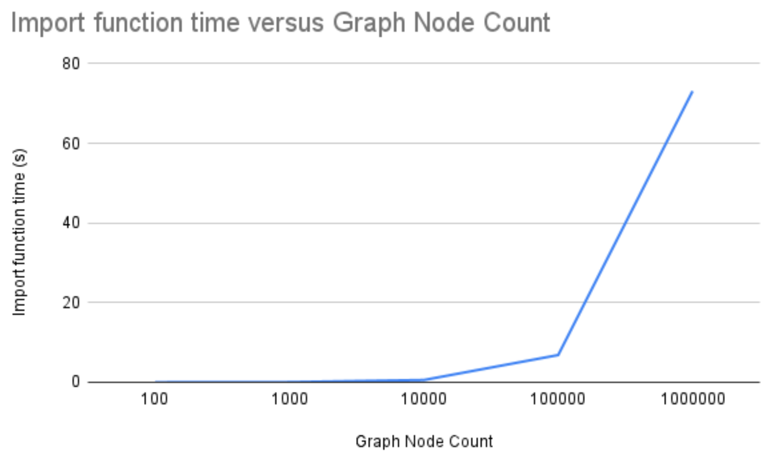


Figure 7.1: The time it takes to execute the import function for certain graph sizes ($n = 10$)

Doing the same process as for the import function for the graph creation results in the data shown in figure 7.2. Here the same relation holds that the factor of the increase in graph size results in the same factor of time increase. This means that the entire import functionality, which consists of the JSON import and the graph creation, will scale with the same factor as increase of the graph size.

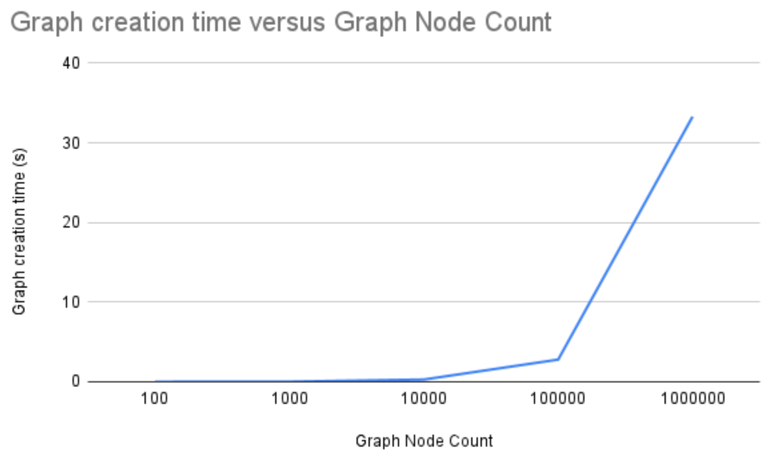


Figure 7.2: The time it takes to create the graph from imported data for certain graph sizes ($n = 10$)

7.2 Query performance

To measure the query performance we measure the time it takes to execute a query and set it out against the size of the result or the amount of nodes visited. We do this so we can measure which of the two, the result size or the amount of nodes visited, is important for the query performance. These experiments are run on a randomly created graph with a million nodes, that contains a large amount of sub graphs. Each layer of a sub graph has a random number of nodes which can randomly connect with a node from the previous layer of the sub graph. To simulate random queries we will randomly pick a node from the graph and use it as our concept to query. The time range of the query will spawn the entire graph to allow for the maximum result of the query. Measuring the result size, which is the amount of rewritten queries, our random query creates is shown in figure 7.3. Here we can see that the result size does not directly correlate with the query execution time. While bigger result are likely to take longer, it is not a rule. So a expecting a large result does not necessarily mean that the execution time of the query will also be large.

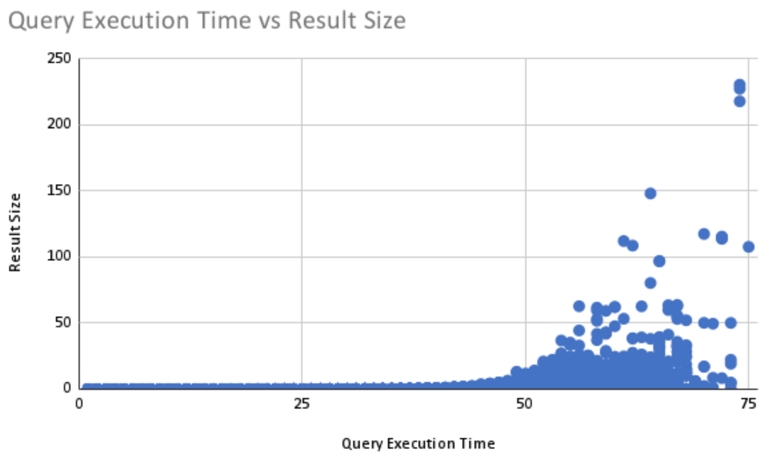


Figure 7.3: Amount of rewritten queries created from a random query set out against the execution time of the query rewriting procedure ($n = 100000$)

Since the result size of a query does not seem to directly correlate to the execution time of a query we look at the amount of nodes visited. The amount of nodes that are visited are the nodes visited during the traversal of the concept evolution graph. These are not unique nodes, as nodes can be visited multiple times by different paths. Counting the amount of nodes visited for a random query set out against the time it takes to rewrite the query is shown in figure 7.4. Here we do see a direct relation between query execution time and the amount of nodes visited. A striking thing from this figure is that randomly generating graphs can lead to a very large amount of nodes visited for certain queries. A graph does not need to have a large amount of nodes to cause the graph traversal to have a high count of nodes visited. A more important factor is the amount of edges between the nodes in the graph. If there are many edges between nodes in a graph there are also potentially a large amount of paths available from nodes in the graph. From this we can conclude that the execution time of a query is better defined by the amount of edges in a graph, as this is the main cause of a high amount nodes visited during the traversal.

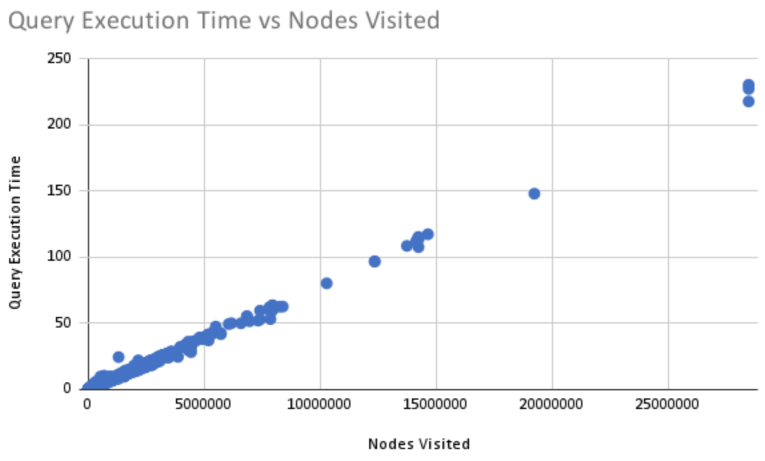


Figure 7.4: Amount of nodes visited during traversal for a random query set out against the execution time of the query rewriting procedure ($n = 100000$)

Chapter 8

Conclusion

In this work we have shown an approach that allows all data in a data set to still be of use when evolution causes concepts to change in a data set. We have created a system that allows users to query the complete data set without having knowledge of this evolution. The system also visually shows the evolution of a data set and the evolution of a query in two different formats. One format, the GraphViz format, to clearly show how concepts evolve. The other format, the Excel format, to clearly show when concepts evolve and how that relates to the rest of the data. The system has the combination of querying while taking evolution into account and visualising the evolution in order to allow users to not only query the data set with evolution but also understand the evolution. With this approach we hope that users will be able to more easily understand the results of their queries without having prior knowledge of a data set and the changes that have occurred to the data set.

8.1 Future work

A clear next step for future work is the automatic detection of the evolution within a data set and the creation of the JSON file as described in this work. This would allow for an entirely automatic system that does not require human intervention to detect evolution in data and allow users to query the data. It would be a tool to detect and understand, by querying and visualising, evolution in an unknown data set.

Bibliography

[1] Networkx is a python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. https://networkx.org.

[2] Abbas Raza Ali. Intelligent call routing: Optimizing contact center throughput. In *Proceedings of the Eleventh International Workshop on Multimedia Data Mining*, MDMKDD '11, New York, NY, USA, 2011. Association for Computing Machinery.

[3] Markus Blaschka. Fiesta: A framework for schema evolution in multidimensional databases (abstract). *Datenbank Rundbrief*, 27:65–66, 01 2001.

[4] Siarhei Bykau, John Mylopoulos, Flavio Rizzolo, and Yannis Velegrakis. On modeling and querying concept evolution. *Journal on Data Semantics*, 1, 05 2012.

[5] J. Ellson, E.R. Gansner, E. Koutsofios, S.C. North, and G. Woodhull. Graphviz and dynagraph – static and dynamic graph drawing tools. In M. Junger and P. Mutzel, editors, *Graph Drawing Software*, Mathematics and Visualization, pages 127–148. Springer-Verlag, Berlin/Heidelberg, 2004.

[6] Manolis Gergatsoulis and Pantelis D. Lilis. Multidimensional rdf. In *OTM Conferences*, 2005.

[7] Itay Gurvich, James R. Luedtke, and Tolga Tezcan. Staffing call centers with uncertain demand forecasts: A chance-constrained optimization approach. *Manag. Sci.*, 56:1093–1115, 2010.

[8] Claudio Gutierrez, Carlos Hurtado, and Alejandro Vaisman. Temporal rdf. In Asunción Gómez-Pérez and Jérôme Euzenat, editors, *The Semantic Web: Research and Applications*, pages 93–107, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[9] Claudio Gutierrez, Carlos A. Hurtado, Alberto O. Mendelzon, and Jorge Pérez. Foundations of semantic web databases. *Journal of Computer and System Sciences*, 77(3):520–541, 2011. Database Theory.

[10] Amal Kaluarachchi, Aparna Varde, Srikanta Bedathur, Gerhard Weikum, Jing Peng, and Anna Feldman. Incorporating terminology evolution for query translation in text retrieval with association rules. pages 1789–1792, 10 2010.

[11] C. Keet and Alessandro Artale. Representing and reasoning over a taxonomy of part-whole relations. *Applied Ontology*, 3:91–110, 01 2008.

[12] Ora Lassila, Ralph R. Swick, World Wide, and Web Consortium. Resource description framework (rdf) model and syntax specification, 1998.

[13] Mehedy Masud, Qing Chen, Latifur Khan, Charu Aggarwal, Jing Gao, Jiawei Han, and Bhavani Thuraisingham. Addressing concept-evolution in concept-drifting data streams. pages 929–934, 12 2010.

[14] Mohammad Mehedy Masud, Jing Gao, L. Khan, Jiawei Han, and Bhavani M. Thuraisingham. Classification and novel class detection in concept-drifting data streams under time constraints. *IEEE Transactions on Knowledge and Data Engineering*, 23:859–874, 2011.

[15] Gilad Mishne, David Carmel, Ron Hoory, Alexey Roytman, and Aya Soffer. Automatic analysis of call-center conversations. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*, CIKM '05, page 453–459, New York, NY, USA, 2005. Association for Computing Machinery.

[16] Tadeusz Morzy and Robert Wrembel. On querying versions of multiversion data warehouse. In *Proceedings of the 7th ACM International Workshop on Data Warehousing and OLAP*, DOLAP '04, page 92–101, New York, NY, USA, 2004. Association for Computing Machinery.

[17] George Papastefanatos, Yannis Stavrakas, and Theodora Galani. Capturing the history and change structure of evolving data. 02 2013.

[18] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. nsparql: A navigational language for rdf. *Journal of Web Semantics*, 8(4):255–270, 2010. Semantic Web Challenge 2009 User Interaction in Semantic Web research.

[19] Darja Solodovnikova. Data warehouse evolution framework. volume 256, 01 2007.

[20] Hai Wang, Zeshui Xu, Hamido Fujita, and Shousheng Liu. Towards felicitous decision making: An overview on challenges and trends of big data. *Information Sciences*, 367-368:747–765, 2016.