EXTENSIBLE OPERATORS FOR COMPARISON QUERIES
OF UNCERTAIN GRAPHS

A Thesis
submitted to the Faculty of the
Graduate School of Arts and Sciences
of Georgetown University
in partial fulfillment of the requirements for the
degree of
Master of Science
in Computer Science

By

Denis Dimitrov, Bachelor of Science

Washington, DC
August 19, 2011
Extensible Operators for Comparison Queries of Uncertain Graphs

Denis Dimitrov, Bachelor of Science

Thesis Advisor: Dr. Lisa Singh

Abstract

Observational scientists study animal societies in their natural settings to understand their social relationships and behavior. Such social network data can be captured as a graph, where nodes represent observed animals and edges represent common sightings. A researcher observing a particular animal may be uncertain about its identification or behavior. This uncertainty can be expressed as existence probabilities associated with nodes and/or edges, and attribute value probabilities. Comparing these uncertain graphs is necessary to answer questions about changes in animal sociality over time, similarities of local animal communities to the general animal population, differences among animal subgroups across locations, and observation bias across researchers, to name a few. A variety of tools, databases, algorithms, and research consider analysis of probabilistic data. Similar, but mostly disjoint work has been done on graph analysis. To the best of our knowledge, there are no existing tools or research results that combine both in a general way for the purpose of uncertain graph comparison. The thesis addresses this gap by proposing a collection of operators for comparing uncertain graphs and a supporting extensible system architecture for using these operators in queries. We also present a prototype implementation of the system architecture that includes a basic query language in an existing visual graph mining tool and demonstrate the utility of the proposed operators and architecture using data from the Shark Bay Dolphin Research Project. The main contribution of
this work is the development of a system architecture that facilitates creating new
operators as compositions of existing ones or as new, independent ones. As demon-
strated, this architecture could serve as the basis for developing simple query lan-
guages that enable scientists and others to write their own ad-hoc uncertain graph
comparison queries without extensive programming knowledge.

INDEX WORDS: Uncertainty, Graphs, Uncertain graph comparison,
              Graph query language, Comparison operators,
              System architecture, Thesis (academic)
# Table of Contents

## Chapter

1. Introduction ................................................................. 1  
   1.1 Motivation ............................................................. 2  
   1.2 Proposed solution .................................................... 6  
   1.3 Thesis Contributions ................................................ 7  
   1.4 Thesis Organization ................................................ 8  

2. Probabilistic Formulation ............................................... 9  
   2.1 Uncertainty in graph elements ..................................... 9  
   2.2 Notation and definitions .......................................... 10  
   2.3 Additional Assumptions .......................................... 14  

3. Proposed Graph Operators and Operations ............................ 16  
   3.1 Running example .................................................... 16  
   3.2 Query Semantics ................................................... 16  
   3.3 Operation and Operator Definitions ................................ 19  
   3.4 Query Language Operations ...................................... 19  
   3.5 Uncertain graph comparison operators ............................ 28  
   3.6 Algorithms for computing complex operators .................... 41  

4. System Architecture and Implementation ............................. 52  
   4.1 Design goals ....................................................... 53  
   4.2 Query Engine Overview .......................................... 55  
   4.3 Overview of query processing services .......................... 57  
   4.4 Query compilation ................................................ 58  
   4.5 Compiled Query Execution ....................................... 63  
   4.6 Other modules ..................................................... 69  
   4.7 Developing and Composing Operators .............................. 70  
   4.8 Uncertain graph data model and data types ...................... 73  

5. Real World Composition Construction Utility ........................ 76  
   5.1 Simple queries (Uncertain node filter) ............................ 79  
   5.2 Uncertain subgraph construction .................................. 81  
   5.3 Group By (Groupings) ............................................. 84  
   5.4 Union, intersection, difference, bidirectional_difference ...... 85  
   5.5 Comparison ......................................................... 88
List of Figures

1.1 Comparing observational uncertain graphs. ........................................ 3
1.2 Comparing derived uncertain graphs. ............................................. 5
2.1 Dolphin domain example uncertain graph ........................................ 10
3.1 Example graphs for graph operators and operations. ......................... 17
3.2 Example subgraphs (nodes) for uncertain attribute operators. .......... 32
3.3 Similarity between two ego-networks. ........................................... 35
3.4 Example ego-networks for similarity operators. ............................. 45
3.5 Classification of ego-network similarity. ...................................... 46
4.1 Query processing workflow. ......................................................... 55
4.2 Query Engine. ............................................................................ 56
4.3 High-level query tree. ................................................................. 59
4.4 Logical Query data model. ......................................................... 60
4.5 Query tree specific to the simple select query. ............................... 61
4.6 Layered architecture and the compiler module. ............................... 63
4.7 Simple implementation of an expression tree. .................................. 64
4.8 Proposed generic compiled query tree. .......................................... 65
4.9 Query Executor. ........................................................................ 66
4.10 Sequence diagram of query preparation. ........................................ 71
4.11 Sequence diagram of query execution. .......................................... 71
4.12 Uncertain graph data model and essential data types. .................... 74
4.13 Uncertain graph data model and essential data types. .................... 75
5.1 Editable XML query representation. ............................................. 77
5.2 Multiple results displayed as tables in separate tabs. ...................... 77
5.3 Displaying a graph returned as query result. .................................. 78
5.4 Ids and names of dolphins, whose most likely location includes ‘RCB’.
5.5 Primary location of dolphin ‘OCEANIA’. ....................................... 80
5.6 Subgraph of elements with bin(0.6) = true. .................................... 81
5.7 Subgraph of edges with bin(0.75) = false and their incident vertices. . 83
5.8 Ego-networks of dolphins ‘JOY’ and ‘UPP’. ................................... 83
5.9 Total count of nodes (dolphins). .................................................. 84
5.10 Count of dolphins by location. ................................................... 85
5.11 Common friends of ‘JOY’ and ‘PUCK’. ....................................... 87
5.12 Clockwise from upper left: union, bidirectional difference, intersection,
and difference of ego-networks of dolphin ‘JOY’ between years 2010 and 2009. 88
5.13 Differences in maximum certainty of the location attribute. . . . . . 90
5.14 Count of dolphin observations by sex. . . . . . . . . . . . . . . . . . . . . 94
Chapter 1

Introduction

Graphs can be used to model data in many different domains, such as social networks, biological protein networks, email correspondence networks, and disease transmission networks. Even though we generally think of nodes and edges in a graph as being certain, this is not always the case. We refer to a graph that has probabilities associated with nodes, edges, or attributes as an uncertain graph. While querying a single uncertain graph is important for understanding which elements have higher confidence, analysts and researchers are also interested in comparing two uncertain graphs to understand where these graphs are similar or different. Manual inspection or custom programs are currently used to compare these uncertain graphs. Instead, this thesis proposes developing a set of comparison operators that can be combined by analysts to highlight the differences, similarities, and/or rankings among elements and attributes of a single uncertain graph or between a pair of uncertain graphs. To the best of our knowledge, there are no existing tools or research results that combine both graph analysis and uncertainty in a general way. Therefore, this thesis proposes (1) a collection of operators for comparing uncertain graphs; (2) a supporting extensible system architecture for using these operators in queries; and (3) a prototype implementation of the system architecture in an existing visual graph mining tool. The main contribution of this work is the development of a system architecture that facilitates creating new operators as compositions of existing ones or as new, independent ones. This architecture could serve as the basis for developing simple query
languages that enable scientists and others to write their own ad-hoc uncertain graph comparison queries without extensive programming knowledge.

1.1 Motivation

In this section we motivate comparison of uncertain graphs using two different scenarios, observational scientific data sets and machine learning outputs.

1.1.1 Observational Certainty

Observational scientists study animal societies in their natural settings to understand their social relationships and behavior. Such social network data can be captured as a graph, where nodes represent observed animals and edges represent common sightings. A researcher observing a particular animal may be uncertain about its identification or behavior. This uncertainty can be expressed as existence probabilities associated with nodes and/or edges, and attribute value probabilities. Comparing these uncertain graphs is necessary to answer questions about changes in animal sociality over time, similarities of local animal communities to the general animal population, differences among animal subgroups across locations, and observation bias across researchers, to name a few.

Figure 1.1 shows an example that compares dolphin observation graphs. Figure 1.1 (a) is the base uncertain graph containing vertices that represent dolphins and edges that represent common dolphin observations made by scientists during 2009 and 2010. For this example, each node has attributes dolphin id and sex. Sex is an uncertain attribute. Each edge has one certain attribute, observation year. Possible uncertain attributes, not shown in the figure, include location and behavior. By creating two subgraphs, for observations in 2009 and 2010 respectively, scientists then
Figure 1.1: Comparing observational uncertain graphs.
can use the proposed operators to answer questions of interest, such as: "Which new relationships are established in 2010?", "What are the changes in dolphin C’s ego-network?", and "What is the subgraph of dolphins, who have a sex value determined with low certainty?".

1.1.2 Derived Model Certainty

Uncertainty may also be introduced during data analysis. Some machine learning algorithms, especially those operating on graphs, generate as output uncertain graphs that can be used as the basis for prediction, generalization, and statistical analysis.

Going back to our dolphin observation network example, let’s suppose that we are interested in building a model that predicts the sex of each node/dolphin. This task is referred to as a node labeling task. Suppose further that we build one model by assigning sex attribute labels to nodes based on other attributes of the node. Doing so generates a graph containing probability distributions over the unobserved labels (the different sexes) of the nodes (dolphins) in the graph. Let’s also suppose that we build another model by assigning sex attribute labels to nodes based on the sex attribute labels of the node’s neighbors. Comparing such output uncertain graphs to each other or to a ground-truth graph allows the researcher to either analyze the performance of different machine learning algorithms or to experiment with a single algorithm under different assumptions. Additionally, it is useful for examining and understanding the graph dataset, for instance, by highlighting areas of data where algorithms disagree in their predictions or areas resulting in low confidence predictions.

Figure 1.2 shows an example of the node labeling task. The subgraph in the top left shows the ground truth graph. Suppose we are interested in predicting the sex attribute label for node A, as shown on the top right of figure 1.2. The bottom two subgraphs show predictions from two different machine learning algorithms. For this
Figure 1.2: Comparing derived uncertain graphs.
small example, visual inspection is straightforward. However, for larger graphs, having operators that compare the most probable values, the confidence in predictions, and the accuracy are important for analysis.

1.2 Proposed solution

To help with these different comparison tasks, we propose developing a set of basic operators required to implement queries for our motivating examples. These operators can be used to find common or distinct subgraphs and estimate similarity between graphs, nodes, edges, and their attributes, allowing for comparison at the attribute, element, ego-network, and graph levels. These basic operators can be used to write custom, ad hoc queries in source code. Of course, our target is to provide a mechanism for users to create more complex and meaningful queries by combining and composing operators on their own without writing source code. We therefore design a rudimentary SQL-like language for forming queries. This language provides context to the queries and manipulates their results by supporting capabilities for selection, projection, filtering, joining, aggregation, and sorting. The semantics of this proposed language is therefore a combination between relational database and uncertain graph semantics, part of which is novel and part of which is necessary for the language to be applicable.

We design a supporting extensible system architecture of a query engine, which handles execution of these operators in queries. As a partial step toward full-scale development, we provide a prototype implementation that can be used as a platform for testing and demonstrating proof-of-concept. The implemented execution environment is based on an intermediate language, which is an XML representation of the proposed query language. The novelty of our design is the focus on extensibility and
modularity. Our framework will support integration of new operators without affecting the existing ones and without requiring significant changes to current ones. It also supports composition of complex operators from basic operators that are provided in our framework. To show the utility and composition ability of our operators, we have integrated our query engine with Invenio, a visual analytic tool for graph mining. Doing so allows us to take advantage of some of the visualization capabilities of the tool for displaying some query results. To demonstrate the utility of our approach, we used a real world data set - a dolphin social network obtained through scientific observation in Shark Bay, Australia.

1.3 Thesis Contributions

Our specific contributions to the field are summarized below:

- We develop a basic set of composable, comparative operators for uncertain graphs, where the previous literature focuses on single graph operators, on specific graph algorithms in the presence of uncertainty, or on operators for multiple certain graphs.

- We propose a basic SQL-like language, which incorporates all of the suggested uncertain graph comparison operators, while taking advantage of most of the SQL capabilities.

- We develop a system framework for uncertain operators that differs from other system implementations. Our architecture is extensible, allowing for easy integration of new operations.

- We implement our query framework in Invenio and demonstrate its utility on a real world data set.
1.4 Thesis Organization

The remainder of this work is organized as follows. In Chapter 2 we present background definitions, underlying assumptions, and notation that will be used throughout the subsequent chapters. Chapter 3 describes the semantic framework in which the queries can be formulated by composing the proposed operations and operators. Chapter 4 discusses the system architecture and the prototype implementation of our query engine. Chapter 5 follows with a demonstration of several categories of specific queries that are shown to be applicable in the analysis and comparison of uncertain graphs. In Chapter 6, relevant literature is reviewed. Finally, conclusions and future directions are presented in Chapter 7.
Chapter 2

Probabilistic Formulation

In this chapter we present background definitions, underlying assumptions, and notation that will be used throughout the subsequent chapters.

2.1 Uncertainty in graph elements

The central object of interest are uncertain graphs, sometimes also referred to as ”probabilistic graphs” [16].¹ They represent a generalization of exact graphs, incorporating uncertainty about vertex / edge existence and attribute values. In the simplest case, each edge is associated with a probability that quantifies our confidence that the edge exists. This model is often used in practical applications, such as expressing interaction in social networks or connectivity in communication and transport networks in the presence of uncertainty. It is a subject of some research efforts directed at extending classical graph and graph mining algorithms to uncertain graphs [15], [16], [24]. In a more general data model, which we adopt, uncertainties can be associated with both vertices and edges [23]. In this case, the existence probability of an edge is assumed to be conditional upon the existence of its endpoints. Because it is not uncommon for an exact graph to have attributes associated with their vertices and edges, we further extend the above model by introducing uncertain attributes. Such attributes contain probabilities associated with each possible value from their

¹We prefer not to use this term to avoid confusion with random graphs.
domain, expressing the likelihood that the attribute takes on this particular value. It is worth noting that an uncertain graph \( G \) semantically represents a probability distribution over all implicated graphs of \( G \), where each implicated graph of \( G \) is an exact graph whose vertex set and edge set are subsets of those of \( G \), respectively [24]. This concept is similar to the possible world semantics of probabilistic databases. To broaden the application of our model, we make no assumptions about the nature of probability values assigned to the graphs that we need to compare. In other words, the analyst can decide if the probabilities are marginal or posterior.

2.2 Notation and definitions

In this section, we define our data model, informally presented in the previous section. As a running example throughout this chapter we will use the uncertain graph of dolphin social interactions presented in figure 2.1. This example has two nodes, one edge, and attributes associated with each. We will describe each of these in more detail throughout this chapter.

![Figure 2.1: Dolphin domain example uncertain graph](image)

**Graph** In its simplest form, a graph \( G = (V, E) \) is a non-empty finite set \( V = \{v_1, \ldots, v_m\} \) of vertices together with a set \( E = \{e_1, \ldots, e_n\} \) of edges, where each edge
e_y is a pair of vertices, \( e_y \in V \times V \), where \( V \times V = \{(v_i, v_j) | v_i \in V, v_j \in V\} \).

In our example, vertices would represent dolphins and edges capture a relationship between a pair of dolphins. Vertices are uniquely identified by dolphin ids and edges are uniquely identified by observation ids. For this work, we are not concerned about whether the edges are directed or undirected, so we focus on the undirected case. We refer to both edges and vertices as *graph elements*.

**Attributes** Additional information about graph elements can be represented as attribute values that are attached to these elements. In our example graph, *size* is an example of an attribute. The set of all attributes is defined as \( A = \{A_1, A_2, \ldots, A_s\} \).

Associated with each attribute \( A_j \) is domain \( D_j \), which may be constrained, having a known set of values, or unconstrained. For vertices, we will denote the set of attributes as \( A^V = \{A_1, \ldots, A_p\} \) and for edges we will denote the set of attributes as \( A^E = \{A_1, \ldots, A_q\} \). In these definitions, an assumption is made that the attributes are consistent across vertices and across edges respectively, i.e. all vertices have the same schema and so do all edges. In the future, this assumption may be relaxed.

Some vertex attributes in our example include *dolphin id*, *name*, *birthYear* and *size*. The corresponding domains for these attributes would be: \( D_1 \) and \( D_2 \) equal the domain of all strings, \( D_3 \) is the domain of positive integers, and \( D_4 \) is one of the strings ‘very small’, ‘small’, ‘medium’, ‘big’, or ‘very big’. The set of edge attributes includes *observ_date*, representing the date on which the corresponding pair of dolphins was observed together.

For a particular instance of graph \( G \), each vertex \( v_i \) in \( V \) has a single value for each attribute in \( A^V \). In other words, given an attribute \( A_j \in A^V \) and a vertex \( v_i \in V \), we associate a value \( p_k \in \{D_j\} \) with the pair \( (v_i, A_j) \) and denote it using the notation \( a(v_i, A_j) = p_k \). As shorthand, we use \( a_{ij} \). Similarly, given an attribute \( A_j \in A^E \) and an edge \( e_i \in E \), we associate a value \( p_k \in \{D_j\} \) with the pair \( (e_i, A_j) \).
and denote it using the notation $a(e_i, A_j) = p_k$. As shorthand, we again use $a_{ij}$.

Depending on the context, $a_{ij}$ can refer to an attribute value of either vertex or edge $i$. In cases where there is ambiguity, we may associate additional subscript with $a_{ij}$ to distinguish between vertex and edge attributes. We allow for some attributes to contain null - i.e. $p_k \in \{D_j\} \cup \{null\}$, designating an unknown attribute value.

For example, the vertex with $dolphin\_id$ value ‘B’ in our graph has attribute values: name = ‘Jasper’, birthYear = 2008, size = ‘big’.

We define the set of attributes for a particular vertex $v_i$ as $A(v_i) = \{A_j : A_j \in A^V \text{ and } a(v_i,A_j) \neq null\}$. For example, $birthYear$ does not belong to the attribute set of vertex with $dolphin\_id$ value ‘A’. Similarly, the set of attributes for a particular edge $e_i$ is defined as $A(e_i) = \{A_j : A_j \in A^E \text{ and } a(e_i,A_j) \neq null\}$.

Based on above, we define an attribute graph $GA$ as a generalization of a regular graph $G$: $GA = (V, E, A^V, A^E)$.

Some operators require graph elements to be uniquely identified. Therefore, in our definition of attribute graph $GA$ we assume that one of the attributes is designated to serve as such unique identifier, further referred to as id: $\exists A_j \in A^V, \text{ s.t. } \forall i,k \in [1,m], i \neq k \rightarrow a_{ij} \neq a_{kj}$ and $\exists A_j \in A^E, \text{ s.t. } \forall i,k \in [1,n], i \neq k \rightarrow a_{ij} \neq a_{kj}$. In the running example, the id is represented by attribute $dolphin\_id$ for vertices and $observation\_id$ for edges.

Uncertain Attributes In this work, $PA_j$ represents an uncertain attribute, whose value is a set of pairs of each possible attribute value and a probability associated with each possible value. In our example graph, an uncertain attribute $sex$ with value domain $\{male, female\}$ could be used to reflect the researcher's uncertainty about the sex of the observed dolphin. For a specific vertex, the set of its value pairs could be $\{(male,0.8), (female,0.2)\}$.
More precisely, if \( PA_j \) is an uncertain attribute, value domain \( VD_j \) is the constrained (discrete) domain of possible values associated with attribute \( PA_j \), equal to \{male, female\} in our example. The value domain is considered ordered and we use the notation \( a^t_j \) to designate the \( t \)-th member of \( VD_j \), where \( t \in [1, |VD_j|] \). Continuing with the previous example, \( a^1_j = male, a^2_j = female \).

\( PD_j \) is the domain of uncertain attribute \( PA_j \): \( PD_j = \{((a^t_j, y^t_j = f(a^t_j)) : \forall a^t_j \in VD_j\} : \) for all probability distribution functions \( f(x) \) over the value domain \( VD_j \). In other words, the domain \( PD_j \) is the infinite set of all permissible values for uncertain attribute \( PA_j \), where each of these values corresponds to a different possible probability distribution function and thus in itself represents a set of pairs of each possible value from the value domain \( VD_j \) and the corresponding pdf output. In our example, \( PD_j = \{(male, 0.1), (female, 0.9)\}, \{(male, 0.2), (female, 0.8)\}, ...\}.

By analogy to certain attributes, let the set of all uncertain attributes be defined as \( PA = \{PA_1, PA_2, \ldots PA_o\} \). Let the set of all uncertain attributes for vertices and edges be denoted as \( PA^V = \{PA_1, PA_2, \ldots PA_r\} \) and \( PA^E = \{PA_1, PA_2, \ldots PA_t\} \), respectively. Certain and uncertain attributes are considered separate, i.e. attribute sets \( A \) and \( PA \), \( A^V \) and \( PA^V \), \( A^E \) and \( PA^E \), respectively, are disjoint.

The rest of the definitions related to uncertain attributes follow the definitions of their certain counterparts. Given an uncertain attribute \( PA_j \in PA^V \) and a vertex \( v_i \in V \), we associate a value \( p_k \in \{PD_j\} \) with the pair \((v_i, PA_j)\) and denote it using the notation \( pa(v_i, PA_j) = p_k \). As shorthand, we use \( pa_{ij} \). Similarly, given an attribute \( PA_j \in PA^E \) and an edge \( e_i \in E \), we associate a value \( p_k \in \{PD_j\} \) with the pair \((e_i, PA_j)\) and denote it using the notation \( pa(e_i, PA_j) = p_k \). As shorthand, we again use \( pa_{ij} \). We allow for some uncertain attributes to contain null - i.e. \( p_k \in \{PD_j\} \cup \{null\} \), designating unknown attribute value.
For example, the vertex with \textit{dolphin\_id} value ‘A’ in our graph has uncertain attribute value \textit{sex} = \{(\textit{male}, 0.8), (\textit{female}, 0.2)\}. As mentioned earlier, \(a^t_j\) is used to refer to members of value domain \(VD_j\). For consistency, we define the shorthand \(p^t_{ij}\) to refer to the corresponding probability \(f(a^t_j)\), associated with value \(a^t_j\) for vertex \(v_i\). In the previous example, \(p^1_{ij} = 0.8\) and \(p^2_{ij} = 0.2\).

We define the set of uncertain attributes for a particular vertex \(v_i\) as \(PA(v_i) = \{PA_j : PA_j \in PA^V and pa(v_i, PA_j) \neq null\}\). Similarly, the set of uncertain attributes for a particular edge \(e_i\) is defined as \(PA(e_i) = \{PA_j : PA_j \in PA^E and pa(e_i, PA_j) \neq null\}\).

\textbf{Uncertain Graph} We define an uncertain graph \(GU\) as a further generalization of an attribute graph \(GA\): \(GU = (V, E, A_V, A_E, PA^V, PA^E)\).

Uncertain attributes allow the data model to express semantic uncertainty in the graph. There may also be uncertainty about existence of graph elements. For example, an edge representing a sighting of two dolphins can be assigned a probability for its existence if the researcher is not certain about the identity of the two observed dolphins. To express such structural uncertainty, we designate one of the attributes in \(A^V\) and \(A^E\) to store our confidence about existence of the corresponding parent element: \(\exists A_j : a_{ij} \in [0,1], \forall i \in [1,m]\) and \(\exists A_j : a_{ij} \in [0,1], \forall i \in [1,n]\). Henceforth, we refer to this attribute as ‘conf’ or ‘confidence’. As mentioned earlier, the edge confidence is interpreted as conditional upon existence of its vertex endpoints.

2.3 Additional Assumptions

Graph isomorphism is a large separate area of research. In comparing two graphs \(g_1\) and \(g_2\), we use the simplifying assumption that the following partial mapping exists between their elements:
- Each vertex \( v \) from \( g_1 \) is either not mapped, or mapped to exactly 1 vertex \( u \) from \( g_2 \). The mapping from \( g_2 \) to \( g_1 \) is the inverse: a vertex \( u \) from \( g_2 \) is mapped to a vertex \( v \) in \( g_1 \) if and only if \( g_1.v \) is mapped to \( g_2.u \). Therefore, vertex mapping consists of a bijective mapping function for those vertices that are mapped, plus a set of unmapped vertices in each of the graphs \( g_1 \) and \( g_2 \).

- Edge mapping is equivalent to vertex mapping, with the added constraint that edges \( g_1.e \) and \( g_2.f \) can be mapped to each other only if both of their endpoint vertices are also mapped. Note that the condition does not force edges between mapped vertices to be mapped as well; it also allows freedom of edge mapping in case of multiple edges.

We refer to two graphs with this property as aligned graphs. Different alignment schemas may be possible. Unless stated otherwise, alignment is assumed to be based on element id: elements from graph \( g_1 \) are mapped to elements with the same unique id in graph \( g_2 \); they are unmapped if there is no corresponding element with the same id. If such a mapping results in violating the edge mapping property, the graphs are considered to be not well-formed.

This simplifying assumption is introduced because of the computational complexity of isomorphic subgraph matching, which is outside the scope of this thesis. In many cases, such as in the motivating examples shown earlier, the graphs under comparison are naturally aligned and therefore, we believe that the assumption should not be considered as an oversimplification. In the future, it may be relaxed: it is certainly possible to incorporate additional operators, which do not rely on graph alignment.
Chapter 3

Proposed Graph Operators and Operations

Having defined our uncertain graph data model in the previous chapter, in this chapter we describe a number of operators and operations over this model. Some of them are novel, and others are included for completeness or for the purpose of providing necessary capabilities to the users.

3.1 Running example

In the sections that follow, we use as a running example two aligned graphs, $g_1$ in figure 3.1a and $g_2$ in figure 3.1b.

3.2 Query Semantics

An operator over one or more graphs, elements, or attributes is intended to calculate, find, or derive a specific piece of information from these entities. It is obvious that a single operator in isolation has limited usability. For more expressive power, it is necessary to offer capabilities for combining and nesting operators, filtering, sorting, joining and grouping results, to name a few, besides the minimum capabilities such as invoking an operator with different arguments. Clearly, accomplishing this task requires introducing a query language in addition to the basic set of graph comparison operators. Because such capabilities in relational databases are provided by SQL, a mature and proven language, we chose to use it as a basis for developing a simple
prototype language geared toward graph comparison. The language itself is not a focus of this work, and we therefore do not claim that our language is the best for the purpose or even complete in any sense; however, we believe it is sufficient in expressing a wide range of uncertain graph comparison queries using our set of operators.

Our approach was to take advantage of the SQL semantics by handling graphs, graph elements, and attributes using relations. For example, the simple SQL-like query below retrieves all nodes from graph $g_1$, applies the $conf()$ operator to each node, and stores the resulting value in a table that is returned to the user. Given a graph element, the $conf()$ operator returns the value of the element’s ‘conf’ attribute, which specifies the existence probability for the element.

```
select conf()
from g1 type NODE
```
An example of a less trivial query is selecting nodes with confidence above a certain threshold, ordered by highest confidence:

```sql
select id, conf()
from g1 type NODE
where conf() > 0.5
order by conf() desc
```

Our query language supports the major SQL operations, such as SELECT, FROM, JOIN, WHERE, GROUP BY, HAVING, and ORDER BY, introducing modifications and extensions to accommodate the specifics of graph comparison. The proposed query language is described in detail in the following section.

Given the similarity between our query language and SQL, we would like to address the possibility of including the proposed graph comparison operators in the context of traditional SQL. This is certainly not impossible; however, there are several disadvantages that led us to creating a separate prototype query language engine. On a syntactic and semantic level, a dedicated query engine allows the flexibility for any modifications that best suit the specifics of uncertain graph comparison. On an implementation level, extending an existing SQL query engine effectively means using a relational database as storage for uncertain graph data. Our goal was to develop an approach that is not restricted to a particular storage. Data can be stored in relational tables, as edge lists, etc. This flexibility is important because the data storage impacts the overall performance of different graph queries and may be optimized for certain types of queries. For example, storing vertices and edges in relational tables is inefficient for path queries. If path queries are central to one’s application, then an alternative storage scheme will improve performance.
3.3 Operation and Operator Definitions

This section gives a brief definition for each operation and operator. Some of them require a more detailed description of their algorithm, which is given in the following section.

We make a distinction between operations and operators based on their role in the query language. Operations are the more general, usually top-level query language constructs, such as SELECT, WHERE, etc. Many of them are borrowed from SQL and thus are usually not specific to uncertain graph comparison. Instead, they provide the semantic and syntactic framework, within which the more specific graph comparison operators can be incorporated. These operators typically perform a particular, isolated calculation or algorithm on graphs, graph elements, or attributes. We describe the query language operations first, following with a description of each of our graph comparison operators.

Both on conceptual and implementation level, a query can be represented as an abstract syntax tree or a query tree. The nodes of the query tree in most cases correspond to the operations and operators in the query, although it is not necessarily a one-to-one correspondence. The query tree is executed by the query engine one node at a time, and the results are passed between parent and child nodes. More detail about the query tree and the system architecture are presented in the next chapter.

3.4 Query Language Operations

Each of the operations described in this section represents a top-level clause in a query. As a rule, they take as part of their input one or more relations that may contain graphs, graph elements, or attributes, and produce such a relation as output.
From a logical point of view, operations are linked to each other in the query tree; therefore, those relations are passed between the operations.

There are certain restrictions on presence and ordering of the operations in a query and query execution, similar to those found in SQL. For example, a query needs to include a FROM and SELECT clause at a minimum.

The operations that are part of this initial query language are the following:

- SQL-like operations: SELECT, FROM, WHERE, ORDER BY, GROUP BY, HAVING, JOIN.

- Extensions: MERGE BY, SPLIT BY.

3.4.1 SQL-like operations

\[ \text{FROM(G} \text{raph } g, \text{FromType } f\text{Type}) \]

Given a graph \( g \), the operation returns a relation with a single column that stores \( g \), its elements, or its attributes, depending on the value of \( f\text{Type} \):

\( \{g\} \), if \( f\text{Type} = \text{"GRAPH"} \);

\( \{g.V\} \), if \( f\text{Type} = \text{"NODE"} \);

\( \{g.E\} \), if \( f\text{Type} = \text{"EDGE"} \);

\( \{g.V \cup g.E\} \), if \( f\text{Type} = \text{"ELEMENT"} \);

\( \{g.A^V \cup g.A^E \cup g.PA^V \cup g.PA^E\} \), if \( f\text{Type} = \text{"ATTRIBUTE"} \);

The ‘.’ in the above notation designates that the corresponding set, such as \( V, E \), etc., is a member of graph \( g \). This notation is also used in the definitions that follow.

For example, consider the following statement fragment:

from g1 type NODE
Its output is a relation containing all of the nodes of graph $g_1$, i.e. A, B, C, D, and E (their order is undetermined):

```
A
B
C
D
E
```

It should be noted that when $fType = 'ATTRIBUTE'$, the operation arranges all attributes in a single column, rather than creating a column for each attribute. The reason for that decision is consistency with the other types of output, produced under the remaining values of $fType$.

The graph $g$ in this operation replaces the source relation for the SQL FROM clause. However, this operations allows only one graph as input, unlike the SQL FROM operation that accepts multiple relations. As will be shown later, in order to work with multiple graphs, two graphs are passed as inputs to a JOIN operation.

**SELECT(Relation $r$, Selection $s$)**

The SELECT operation performs projection specified by the Selection parameter $s$. It contains a non-empty list of columns and expressions in any combination. Given the source relation $r$, the operation keeps the listed columns, stores the results of evaluating the expression on each tuple in new columns, and removes the remaining columns. Alternatively, $s$ can contain a single ‘select all’ value, represented as “*” in SQL, in which case the relation remains unaltered. Note: in aggregated relations, * applies only to the aggregated columns, i.e. those appearing in the GROUP BY operation.
select id, conf(), name
from g1 type NODE

would produce the relation (order of tuples is not guaranteed):

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.9</td>
<td>Jasper</td>
</tr>
<tr>
<td>B</td>
<td>0.8</td>
<td>Smudge</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>Galina</td>
</tr>
<tr>
<td>D</td>
<td>0.4</td>
<td>Sami</td>
</tr>
<tr>
<td>E</td>
<td>0.7</td>
<td>Ely</td>
</tr>
</tbody>
</table>

WHERE(Relation r, Expression f)

The WHERE operation filters the relation r by applying the expression f to each tuple and removing those tuples, for which the expression does not evaluate to TRUE.

The result of the example query:

select id, conf()
from g1 type NODE
where conf() > 0.7

would produce the relation (order of tuples is undetermined):

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.9</td>
</tr>
<tr>
<td>B</td>
<td>0.8</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
</tbody>
</table>

ORDER BY(Relation r, Selection s)

The ORDER BY operation sorts the tuples in relation r by values in selection s, which is a non-empty list of references to r’s columns and arbitrary complex expressions of non-aggregate operators, in any combination. The sort is performed based on the first
element of $s$. In case of a tie between two or more tuples, they are sorted based on the second element of $s$, and so on. The operation also offers the capability for sorting in descending order.

The result of the example query:

```
select id, conf()
from g1 type NODE
where conf() > 0.7
order by conf()
```

would produce the ordered relation:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0.8</td>
</tr>
<tr>
<td>A</td>
<td>0.9</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
</tbody>
</table>

GROUP BY(Relation $r$, Expression $e_1 \{,\ Expression\ e_2\...\})$

The `GROUP BY` operation returns a relation $r$ that contains one tuple for each distinct combination of values $val_1$, $val_2$, ... of expressions $e_1$, $e_2$... Each of these tuples contains the set of tuples from relation $r$, for which the expressions $e_1$, $e_2$,... evaluate to the corresponding combination of values $val_1$, $val_2$, .... The nested tuples are only ”visible” to subsequent aggregate operators, such as $\text{count()}$, $\text{min()}$, or $\text{avg()}$.

Expressions $e_1$, $e_2$,... can simply refer to columns of relation $r$, or they can represent arbitrarily complex expressions of non-aggregate operators. The first expression $e_1$ is mandatory, and the rest are optional.

The result of the example query:

```
select size, count()
from g1 type NODE
group by size
```

23
would produce the relation (order of tuples is undetermined):

<table>
<thead>
<tr>
<th>size</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>small</td>
<td>2</td>
</tr>
<tr>
<td>medium</td>
<td>1</td>
</tr>
<tr>
<td>big</td>
<td>1</td>
</tr>
<tr>
<td>very big</td>
<td>1</td>
</tr>
</tbody>
</table>

The intermediate result after executing the GROUP BY operation, but before the SELECT operation would be:

<table>
<thead>
<tr>
<th>size</th>
<th>tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>small</td>
<td>B, C</td>
</tr>
<tr>
<td>medium</td>
<td>D</td>
</tr>
<tr>
<td>big</td>
<td>A</td>
</tr>
<tr>
<td>very big</td>
<td>E</td>
</tr>
</tbody>
</table>

HAVING(AggregateRelation r, AggregateExpression f)

The HAVING operation is the aggregate equivalent of WHERE, which only applies a non-aggregate expression to non-aggregated columns. HAVING, on the other hand, filters the aggregate relation r by applying the aggregate expression f to each outer tuple and removing those tuples, for which the expression does not evaluate to TRUE. It is therefore implied, that this operation can only be used in a query in conjunction with GROUP BY.

Executing the query:

```sql
select size, count()
from g1 type NODE
group by size
having count() > 1
```

would produce the following relation:
JOIN(Relation $r_1$, Relation $r_2$, Expression $e$)

The $JOIN$ operation performs an inner join on relations $r_1$ and $r_2$ using expression $e$, or a cross-join, if $e$ is omitted. A cross-join is a Cartesian product: it produces a result relation $r$ with columns from both $r_1$ and $r_2$, and tuples combining each tuple in $r_1$ with each tuple in $r_2$. An inner join can be defined as the outcome of first taking the cross-join and then returning the tuples, for which expression $e$ evaluates to TRUE. Because computing the Cartesian product is very inefficient, the inner join implementation can be optimized to use other approaches.

The following query fragment illustrates the inner join semantics by example:

```sql
(SELECT g1Node
FROM g1 type NODE as g1Node)
JOIN
(SELECT g2Node
FROM g2 type NODE as g2Node)
on g1Node.id = g2Node.id
```

Given our example graphs, it produces the following relation as a result, where the first and second columns contain nodes from $g1$ and $g2$, respectively, matched by id:

```
A
B
C
D
E
```

Cross-joins are especially useful when one of the source relations contains a single tuple that needs to be combined with every tuple of the second relation. For example,
the union of two graphs can be obtained by first cross-joining the relations containing
each graph:

```
select union(g1, g2)
from (  
  (select *
   from g1 type GRAPH as g1)
  join
  (select *
   from g2 type GRAPH as g2)
)
```

### 3.4.2 Extension operations

**SPLIT BY(Relation \( r \), Column \( c \))**

The *SPLIT BY* operation is used to "flatten out" a relation, when column \( c \) contains
a set of values rather than a single value. In the returned relation, each tuple \( t \) in \( r \) is replaced by a number of tuples, one for every value in the set of values from \( t \) in
column \( c \). The new tuple contains the corresponding single value in \( c \) and retains the
values in the remaining columns. If \( t.c \) contains an empty set or null, then only one
tuple with \( c=\text{null} \) is derived from \( t \).

Let’s take the following source relation \( r \) as an example:

<table>
<thead>
<tr>
<th>A</th>
<th>RCB, WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>PN, WB, HB</td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>

Splitting by second column produces the relation:
A set of values is the result of some common operators, such as \textit{mpv}. Another useful example is the \textit{adjacentVertices} operator that returns the set of adjacent vertices of a given vertex. The \textit{SPLIT BY} operation helps to normalize relations containing the results of such operators.

\textit{SPLIT BY} currently operates on a single column. Splitting by multiple columns can be achieved by nesting subqueries.

\textbf{MERGE BY(}Relation \textit{r}, Boolean \textit{distinct}, \{Expression \textit{e}1, Expression \textit{e}2\ldots\})

Similar to \textit{GROUP BY}, the \textit{MERGE BY} operation returns a relation that contains one tuple for each distinct combination of values \textit{val}1, \textit{val}2, \ldots of expressions \textit{e}1, \textit{e}2\ldots Unlike \textit{GROUP BY}, which creates nested tuples, \textit{MERGE BY} retains the remaining columns as a collection of values from the respective columns of those original tuples from relation \textit{r}, for which the expressions \textit{e}1, \textit{e}2\ldots evaluate to the corresponding combination of values \textit{val}1, \textit{val}2, \ldots. Therefore, all columns are “visible” to subsequent non-aggregate operators.

The query fragment:

\texttt{from g1 type NODE merge by size}
would produce the following intermediate result:

<table>
<thead>
<tr>
<th>medium</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>big</td>
<td>A</td>
</tr>
<tr>
<td>very big</td>
<td>E</td>
</tr>
</tbody>
</table>

Expressions $e_1$, $e_2$, ... can simply refer to columns of relation $r$, or they can represent arbitrarily complex expressions of non-aggregate operators. Unlike the GROUP BY operation, SPLIT BY does not require at least one expression $e_1$. Omitting all expressions, specified as MERGE ALL, is equivalent to merging relation $r$ into a single tuple, where each column contains a collection of values from the corresponding column for all tuples of $r$.

An additional feature is the distinct modifier. When distinct is enabled, the operation discards duplicates for each column that is turned into a collection.

In effect, this operation is semantically opposite to the SPLIT BY operation.

3.5 Uncertain Graph Comparison Operators

Unlike the more general query operations, graph comparison operators typically perform a particular, isolated calculation or algorithm on graphs, graph elements, or attributes. They cannot exist independently; instead, they are part of an operation or an expression that belongs to the operation. Thus, they are represented as child nodes in the query tree. Their input and output is usually not a relation but is specific to the calculation that is performed by the operator.

Below we give an outline of each operator. They are further subdivided based on their target. We distinguish operators that apply to attributes, graph elements, ego-networks, and graphs.
3.5.1 Graph Element operators

Graph element operators require a graph element as input. Let us assume that \( el, el_1, el_2 \) are graph elements of the same type, i.e. \( el, el_1, el_2 \in V \) or \( el, el_1, el_2 \in E \). It is also possible that \( el_1 \) and \( el_2 \) belong to different graphs.

Confidence operator

The \( conf \) operators returns the element’s confidence. More precisely, \( conf(el) = a(el, conf) \), i.e. the value of attribute \( conf \) for element \( el \). If \( el \) is the node with \( id = B \) from graph \( g_1 \), \( conf(el) = 0.8 \).

Bin operator

The general bin operator, \( bin \), assigns each element a true or false bin, corresponding to a high or low confidence of existence. The threshold \( T \) can be user specified or assume the default value of 0.5:

\[
bin(el) = \text{false if } conf(el) < T, \text{ else true}.
\]

If \( el \) is the node with \( id = B \) from graph \( g_1 \), \( bin(el) = true \).

Comparative Bin operator

The comparative bin operator, \( comp \_ bin \) returns a value of "high", "opposite", or "low", depending on the relationship between the output of the bin() operator applied to each of the two operands. \( comp \_ bin(el_1, el_2) = 'high' \) if \( bin(el_1) = true \) and \( bin(el_2) = true \). It returns 'low' if \( bin(el_1) = false \) and \( bin(el_2) = false \). Otherwise, it returns the value 'opposite'. If \( el_1 \) and \( el_2 \) refer to nodes with \( id = B \) from graphs \( g_1 \) and \( g_2 \) respectively, \( comp \_ bin(el_1, el_2) = 'opposite' \).
**Magnitude difference operator**

The magnitude difference operator `magnitude_diff` returns the difference between confidences for the two elements: `magnitude_diff(el_1, el_2) = conf(el_1) - conf(el_2)`. If `el_1` and `el_2` refer to nodes with `id = B` from graphs `g_1` and `g_2` respectively, `magnitude_diff(el_1, el_2) = 0.4`.

**Difference significance operator**

The difference significance operator, `diff_significance`, returns whether the absolute value of magnitude difference is above a threshold `T`, which can be user-specified or a default value. `diff_significance(el_1, el_2) = true` if `|magnitude_diff(el_1, el_2)| ≥ T`, otherwise it is `false`. If `el_1` and `el_2` refer to nodes with `id = B` from graphs `g_1` and `g_2` respectively, `diff_significance(el_1, el_2, 0.3) = true`.

**Value certainty score operator**

The value certainty score operator, `value_certainty_score` returns the average of `max_value_certainty()` of all uncertain attributes belonging to the element: `value_certainty_score(el) = \sum_{PA_j \in PA(el)} \frac{\text{max}_\text{value_certainty}(pa_{el, PA_j})}{|PA_{el}|}`. Please refer to uncertain attribute operators for definition of `max_value_certainty()`.

**Similarity operator**

The similarity operator, `sim(el_1, el_2)` returns the degree of similarity between two elements of the same type (vertices or edges), typically in the range [0, 1]. Different metrics can be used. These are described in more detail in section 3.6.
3.5.2 Uncertain attribute operators

Uncertain attribute operators answer queries about values and probabilities associated with these attributes. For example, the user may be interested in a count of female dolphins in a graph:

```sql
select count()
from g1 type NODE
where 'female' in mpv(gender)
```

Or the user may want to extract nodes with a dominant probability for the location attribute:

```sql
select *
from g1 type NODE
where peak_to_next_dist(location) > 0.3
```

These operators require an instance of one or more uncertain attributes as input. Let us assume for the definitions and examples below that \( u_a \) is an uncertain attribute value for some vertex \( v_i \), i.e. \( u_a = pa(v_i, PA_j) \). For binary operators, let \( u_{a1} \) and \( u_{a2} \) be uncertain attribute values for the same attribute, i.e. \( u_{a1} = pa(v_i, PA_j) \) and \( u_{a2} = pa(v_l, PA_j) \). Usually \( u_{a1} \) and \( u_{a2} \) are associated with two different vertices \( v_i \) and \( v_l \), possibly from different graphs. The operators behave identically when one or both of the uncertain attributes belong to an edge instead of a vertex; therefore, these cases are not reviewed separately. Although unlikely, it is also possible that the user may want to compare two attribute values belonging to different attributes: \( u_{a1} = pa(v_i, PA_j) \) and \( u_{a2} = pa(v_l, PA_m) \). The proposed binary operators would still apply, provided that the two attributes have the same domain, i.e. \( PD_j = PD_m \) (which also requires that \( VD_j = VD_m \)). Figure 3.2 contains example nodes and attributes that will be used for the uncertain attribute operators presented in this section.
Most Probable Value Operator

The most probable value, \( mpv \) and least probable value, \( lpv \) operators return a set of value(s) with the highest probability or lowest probability, respectively:

1. \( mpv(ua) = mpv(pa(v_i, PA_j)) = \{ a_j^t | p_{ij}^t = \max, \forall t \} \)

2. \( lpv(ua) = lpv(pa(v_i, PA_j)) = \{ a_j^t | p_{ij}^t = \min, \forall t \} \)

If \( ua \) is the attribute ”location” for the node with \( id = A \) from figure 3.2a, then \( mpv(ua) = \{ 'RCB' \} \), and \( lpv(ua) = \{ 'PN', 'HB' \} \).

To account for the possibility of a tie between probabilities of more than one value, the operator returns a set. Some additional operators are provided to facilitate the usage of this operator. For example, \( first \) returns the first element of the set, and \( in \) can be used to test for set membership.
Value certainty operators

The value certainty operator, \textit{value\_certainty} returns the probability that the attribute has a specific value from the domain: \(\text{value\_certainty}(ua, val) = p_{ij}^t | (a_j^t = val)\). If \(ua\) is attribute "location" for the node with \(id = A\) from figure 3.2a, then \(\text{value\_certainty}(ua, 'RCB') = 0.6\).

Similarly, maximum value certainty, \textit{max\_value\_certainty}, minimum value certainty, \textit{min\_value\_certainty}, average value certainty, \textit{avg\_value\_certainty}, and median value certainty, \textit{median\_value\_certainty} return the maximum, minimum, arithmetic mean, and median probability, respectively, among all probabilities associated with all of the attribute values:

1. \(\text{max\_value\_certainty}(ua) = \max(p_{ij}^t), \forall t\)
2. \(\text{min\_value\_certainty}(ua) = \min(p_{ij}^t), \forall t\)
3. \(\text{avg\_value\_certainty}(ua) = \text{arithmetic mean}(p_{ij}^t), \forall t\)
4. \(\text{median\_value\_certainty}(ua) = \text{median}(p_{ij}^t), \forall t\)

If \(ua\) is attribute "location" for the node with \(id = A\) from figure 3.2a, then \(\text{max\_value\_certainty}(ua) = 0.6, \text{min\_value\_certainty}(ua) = 0.0, \text{avg\_value\_certainty}(ua) = 0.2, \) and \(\text{median\_value\_certainty}(ua) = 0.1\). Notice that under the current assumption that the probability values for an attribute adds up to 1, the result of \(\text{avg\_value\_certainty}(ua)\) always equals \(\frac{1}{|V_D|}\). The operator will provide more meaningful results when the assumption is relaxed during future extensions.

The value certainty deviation, \textit{value\_certainty\_dev} returns the standard deviation of probabilities associated with all values of the attribute: \(\text{value\_certainty\_dev}(ua) = \sigma(p_{ij}^t), \forall t.\)
Finally, the value certainty range, \textit{value\_certainty\_range} returns the difference between maximum probability and minimum probability among all associated with the attribute values:

\[
\text{value\_certainty\_range}(ua) = \max\text{-value\_certainty}(ua) - \min\text{-value\_certainty}(ua).
\]

If \(ua\) is attribute "location" for the node with id=A from figure 3.2a, then \(\text{value\_certainty\_range}(ua) = 0.6\).

**Peak to average and next distance operators**

The peak to average distance operator, \textit{peak\_to\_avg\_dist} returns the difference between the peak probability, \(\max\text{-value\_certainty}(ua)\), and the average probability among the probabilities associated with the remaining attribute values, other than \(mpv(ua)\):

\[
\text{peak\_to\_avg\_dist}(ua) = \max\text{-value\_certainty}(ua) - \text{arithmetic\_mean}\left(\{p_{ij}, \forall t, s.t. p_{ij} \neq \max\text{-value\_certainty}(ua)\}\right).
\]

If \(ua\) is attribute "location" for the node with id = A from figure 3.2a, then \(\text{peak\_to\_avg\_dist}(ua) = 0.5\).

A similar operator, peak to next distance, \textit{peak\_to\_next\_dist} returns the difference between the peak probability, \(\max\text{-value\_certainty}(ua)\) and the next-highest probability of the attribute:

\[
\text{peak\_to\_next\_dist}(ua) = \max\text{-value\_certainty}(ua) - \max(\{p_{ij}, \forall t, s.t. p_{ij} \neq \max\text{-value\_certainty}(ua)\}).
\]

If \(ua\) is attribute "location" for the node with id = A from figure 3.2a, then \(\text{peak\_to\_next\_dist}(ua) = 0.3\).

**Similarity operator**

The similarity operator, \textit{sim}(\(ua_1, ua_2\)) returns the degree of similarity between two uncertain attributes of the same type, typically in the range [0, 1]. The similarity is generally measured between the two sets of their respective attribute values and probabilities. There are a number of algorithms for calculating similarity. These are described in more detail in section 3.6.
3.5.3 Ego Network operators

Ego-networks, or ego-nets, for our purposes are defined as subgraphs consisting of the starting node $v_i$ (center), all of its adjacent nodes, and all edges between them and the center $v_i$. Edges between any pair of nodes not including the center $v_i$ are ignored. Two simple ego-networks are shown in figure 3.3.

**ego.net**

Given a vertex $v_i$, this operator returns the set of vertices and edges that are part of $v_i$’s ego-network, including $v_i$ itself. For more flexibility, it does not return a graph consisting of these elements. Such graph can be easily constructed using the `to_graph()` operator. Alternatively, the set of these elements can be broken up into individual tuples using the `SPLIT BY` operation.

**Ego-net similarity**

The ego-network similarity operator, $ego\_sim(v_1, v_2)$, returns the degree of similarity between two ego-networks defined by their center vertices $v_1$ and $v_2$, respectively.
similarity value is typically in the range \([0, 1]\). The operator uses different similarity measures and algorithms depending on user-specified constraints and on ego-network containment within the same or different graphs. These cases are described in detail in section 3.6.

### 3.5.4 Graph operators

Operators in this section apply to or produce as a result one or more graphs as a whole. For the examples that follow, we assume that \(g\), \(g_1\), and \(g_2\) are uncertain graphs, such that \(g_1\) and \(g_2\) are aligned.

**INTERSECT**(\(g_1, g_2\))

The *INTERSECT* operator creates a new graph \(g\), which contains elements corresponding to those elements, which are common to both \(g_1\) and \(g_2\). In other words, for each pair of vertices \((v_1, v_2)\) or edges \((e_1, e_2)\), mapped by id in \(g_1\) and \(g_2\), graph \(g\) contains a vertex \(v\) or edge \(e\), respectively. The element has the same id as the corresponding pair. It also contains a set of attributes that is a union of attributes from the corresponding pair, with their names prefixed with the source graph id to avoid name collisions.

**UNION**(\(g_1, g_2\))

The *UNION* operator creates a new graph \(g\), which contains elements corresponding to those elements, which are present in \(g_1\) or \(g_2\). In other words, graph \(g\) contains a vertex \(v\) or edge \(e\), respectively, for each pair of vertices \((v_1, v_2)\) or edges \((e_1, e_2)\) mapped by id in \(g_1\) and \(g_2\). The unmapped elements are represented in \(g\) as well. Each element in \(g\) has the same id as the corresponding pair or unmapped element. It also contains a set of attributes that is a union of attributes from the corresponding
pair or equivalent to the set of attributes of the unmapped element, with attribute	names prefixed with the source graph id to avoid name collisions.

\textbf{DIFFERENCE}(g1, g2)

The \textit{DIFFERENCE} operator creates a new graph $g$, which contains elements corre-
responding to those elements, which are present only in $g1$ and not in $g2$. In other
words, graph $g$ contains a vertex $v$ or edge $e$, respectively, for each unmapped vertex
or edge in $g1$, and has the same id. For edges from $g1$ to be represented in $g$, there
is an additional restriction that their endpoints must be also represented, to avoid
orphan edges. Each element in $g$ contains a set of attributes that is equivalent to
the set of attributes of the corresponding unmapped element from $g1$, with attribute
names prefixed with the source graph id.

\textbf{BIDIRECTIONAL DIFFERENCE}(g1, g2)

The \textit{BIDIRECTIONAL DIFFERENCE} operator creates a new graph $g$, which con-
tains elements corresponding to those elements, which are present only in $g1$ or only
in $g2$, but not in both. In other words, graph $g$ contains a vertex $v$ or edge $e$, respec-
tively, for each unmapped vertex or edge in $g1$ or $g2$, and has the same id. For edges
from either $g1$ or $g2$ to be represented in $g$, there is an additional restriction that
their endpoints must be also represented, to avoid orphan edges. Each element in $g$
contains a set of attributes that is equivalent to the set of attributes of the corre-
sponding unmapped element from $g1$ or $g2$, with attribute names prefixed with the
source graph id.

\textit{BIDIRECTIONAL DIFFERENCE} is in effect equivalent to a difference of
union and intersection of graphs $g1$ and $g2$: \textit{bidirectional_difference}(g1, g2) =
difference(union(g1, g2), intersection(g1, g2))
We also propose an extension to the \textit{INTERSECTION, UNION, DIFFERENCE, BIDIRECTIONAL DIFFERENCE} operators, that takes into account the confidence of elements in g1 and g2. Under that modification, elements whose confidence is below a specific threshold are considered non-existent.

\textbf{GRAPH RECONSTRUCTION OPERATOR}

Sometimes an operator returns a set of vertices and edges. It may be necessary for a different operator to have a graph as input as opposed to a set of vertices and edges. To accommodate this, we introduce a graph reconstruction operator, \textit{to\_graph}. This operator recreates a graph from a set of vertices and edges. The new graph includes all given vertices; those edges, for which both endpoints belong to the given set of vertices; and attributes associated with these vertices and edges.

This operator can be used in a query deriving a subgraph of a given graph based on specified conditions. For example, the following query returns a subgraph of high-confidence elements from the original graph:

\begin{verbatim}
select to_graph(e)
from g1 type ELEMENT as e
where bin(e)
merge by e
\end{verbatim}

While this operator is not as sophisticated as other techniques, such as pattern matching [12], it does provide the possibility of subgraph filtering based on a flexible set of conditions. It also demonstrates how operators can be combined in the context of our framework, both in semantic and implementation terms. It should be noted that the elements in the returned graph are copies of the original elements, since an element can belong only to one graph.
TO_ELEMENTS

This operator, as opposed to the to_graph operator, breaks down a given graph into a set of vertices and edges. It serves as a bridge between operators that produce a graph as output and the functionality provided by element-level operators.

3.5.5 OTHER OPERATORS

In addition to operators related to uncertain graph comparison, the proposed query language supports general operators, most of which are commonly present in many other languages, including SQL.

AGGREGATE OPERATORS

Aggregate operators include: count, average (avg), minimum (min), maximum (max), and sum. These operators apply to a set of values, typically in a SELECT operation that retrieves the set of values from a column in a relation, possibly a non-aggregated column in an aggregate table. Aggregated columns are those appearing in the GROUP BY operation. Applying these operators to a non-aggregated column in an aggregate table precludes using non-aggregate operators in the same SELECT statement, other than those referring exclusively to aggregated columns. It also precludes using other aggregate operators on the aggregated columns. Applying these operators to an aggregated column precludes using non-aggregate operators altogether.

Aggregate operators are also used to specify the filtering condition in the HAVING operation, where they are applied to aggregated columns.

Aggregate operators ignore NULL values.
**Auxiliary element operators**

The following operators are not listed among element-level operators, because they are not novel or specific to uncertain graphs; however, they are indispensable in navigating and examining the graph structure. They include:

- **isVertex, isEdge**: test if the given element is a vertex or edge, respectively

- **degree**: returns the degree of a vertex, i.e. the number of its incident edges

- **incidentElements**: returns the set of incident edges, given a vertex, and the set of incident vertices, given an edge

- **isIncident**: tests whether a vertex and an edge are incident to each other

- **adjacentVertices**: returns the set of vertices, adjacent to the given vertex

- **isAdjacent**: tests whether a pair of vertices are adjacent to each other

- **adjacentIds**: returns the set of ids of vertices, adjacent to the given vertex

- **isAdjacentById**: tests whether a vertex is adjacent to a vertex with a given id

**Comparison operators**

We include the standard comparison operators: <, >, ≤, ≥, ==, ≠. All of them operate as expected.

**Logical operators**

Logical operators currently supported are AND, OR, and NOT.
Mathematical operators

Mathematical operators currently supported are +, −, ×, ÷, and abs_value (absolute value).

Set operators

Some operators, such as adjacentVertices, return a collection of values rather than a single value. Therefore, it is necessary to provide operators for set manipulation. The first operator returns the first element from a collection. This operator is useful in extracting single values (at the cost of loss of accuracy) from a column containing a set of values for each row, such as one containing results of the mpv operator:

```sql
select first(mpv(location))
from g1 type NODE
```

The in operator tests for membership of an element in a collection. The collection can be either returned by an operator such as mpv, or specified in the query using literals:

```sql
select *
from g1 type NODE
where name in ('Jasper', 'Smudge')
```

The size operator returns the number of elements in the collection, while isEmpty tests whether the collection size is 0.

3.6 Algorithms for computing complex operators

Among the more complex operators are those for computing similarity between two entities at different granularities: uncertain attribute, element, and ego-network. In
In this section, we describe the algorithm we use to compute similarity in these different cases.

3.6.1 Uncertain attribute similarity

We continue to operate under the assumption that $ua_1$ and $ua_2$, the two uncertain attributes under evaluation, have the same definition. In other words, they share the same domain of attribute values, and the order of values within their value-probability pairs is the same (see 3.5.2).

In the proposed set of uncertain attribute similarity operators, similarity is classified as either structural or semantic. The former identifies the similarity between the general shapes of the two distributions, ignoring the attribute values and their arrangement relative to each other. For example, attribute (a, 0.8), (b, 0.1), (c, 0.1) should be considered structurally equivalent to attribute (a, 0.1), (b, 0.1), (c, 0.8), as both have a dominant value (peak) of 0.8 and two low-probability values, 0.1 and 0.1. On the other end of the spectrum is an uncertain attribute with a flat distribution of probabilities. The semantic similarity, on the other hand, compares probabilities between the same attribute values.

**Structural similarity**

We support two structural similarity measures, entropy and absolute distance ratio.

The entropy ratio compares the distribution spread for the specified uncertain attribute. To determine the entropy ratio, we first calculate Shannon’s entropy values for $ua_1$ and $ua_2$. Shannon’s entropy values for uncertain attributes is calculated as follows: $H(ua) = -\sum_{i=1}^{[VP]} p_{ij}^t * \log(p_{ij}^t)$. Here $p_{ij}^t$ represents the probability mass of the uncertain attribute values. The entropy ratio is then defined as: $\frac{H(ua_1)}{H(ua_2)}$. 

42
The absolute distance ratio compares the magnitude of the distance between the different uncertain attribute values for an uncertain attribute. We first calculate the absolute distance for \( ua_1 \) and \( ua_2 \): \[ AD(ua) = \sum_{t=2}^{\|VD_j\|} |p_{ij}^t - p_{ij}^{t-1}|. \] Again, \( p_{ij}^t \) represents the probability mass of the uncertain attribute values and the attribute probabilities are indexed starting from 1 to the size of the value domain (designated as \( |VD_j| \)), where \( |VD_j| \) is at least two. To reflect structural similarity, for absolute distance calculation, probability sets in both attributes must be first sorted. Finally, both of these structural similarity measures produce a value that is not necessarily between 0 and 1.

**Semantic similarity**

For calculating semantic similarity, an instance of uncertain attribute can be perceived as a histogram, or a mapping between an integer vector to a set of non-negative reals. The integer vector is the index to the list of all possible attribute values, and their corresponding probabilities represent the non-negative reals describing the distribution of those values. We further refer to each possible attribute value as a ‘bin’ in the histogram, conceptually containing the associated probability. From the definition of uncertain attributes it follows that the sum of probabilities in all bins for a particular uncertain attribute in the histogram equals 1.

The above analogy allows us to use a number of measures that have been proposed for histogram similarity. They generally fall into two categories [17]. The *bin-by-bin* similarity compares contents of only corresponding bins, or in our case, probabilities for the same attribute values in two attribute instances. *Cross-bin* measures, on the other hand, compare non-corresponding bins. This is possible only if the *ground distance* between pairs of non-corresponding attribute values is known. In this work, we focus on the following bin-by-bin similarity measures:
- Default: $sim(ua_1, ua_2) = 1 - \frac{\sum_{t=1}^{V_D}|p_{ij}^t - p_{lj}^t|}{2}$.

- Minkowski-Form Distance:

$$sim(ua_1, ua_2, r) = \left(\sum_{t=1}^{V_D}|p_{ij}^t - p_{lj}^t|^r\right)^{\frac{1}{r}}.$$  

The most common usages of this measure are with $r = 1$ and $r = 2$.

- Histogram intersection:

$$sim(ua_1, ua_2) = 1 - \frac{\sum_{t=1}^{V_D}\min(p_{ij}^t, p_{lj}^t)}{\sum_{t=1}^{V_D}p_{ij}^t}.$$ 

- K-L divergence:

$$sim(ua_1, ua_2) = \sum_{t=1}^{V_D}(p_{ij}^t \times \log\left(\frac{p_{ij}^t}{p_{lj}^t}\right)).$$

### 3.6.2 Element similarity

Currently, there is only 1 measure for element similarity. It assumes that the two elements under evaluation have the same set of uncertain attributes and calculates the average similarity between each pair of matched attributes. The similarity measure between attribute pairs can be selected by the user. When an attribute is present in only one of the elements, the similarity for this attribute is considered to be 0.

### 3.6.3 Ego-network similarity

For measuring similarity between two ego-nets, their elements are mapped as follows: the two center nodes are mapped to each other, each of the non-center nodes from the first subgraph is mapped to 0 or 1 non-center nodes from the second subgraph, and vice versa. Every edge $e$ is mapped to the edge $e'$ in the other subgraph, such that the endpoints of $e'$ are mapped to the endpoints of $e$, if such a mapping exists. For ego-network similarity, we assume that multiple edges between a pair of vertices are not allowed.
We propose several different cases for this operator, each with different semantics and implementation. Based on the desired evaluation criteria, similarity can be structural, semantic or both. Structural similarity only takes into account the existence or confidence of existence of vertices and edges in each mapped pair between the two ego-nets, while ignoring attributes and their values. Semantic similarity, on the other hand, ignores confidence and derives the similarity score by only using similarity measures between the individual nodes and edges in the mapped pairs. Structural-semantic similarity is a combination of both.

Depending on existence and on alignment between the two ego-nets, similarity can be aligned and unaligned. In the aligned case, the mapping is determined by the alignment scheme. If no alignment scheme is chosen (not aligned case), the ego-nets’ elements are mapped in a way that maximizes similarity.

The default alignment scheme is different, depending on whether the ego-nets belong to the same graph or two different graphs. In the latter case, the ego-nets
are aligned using element ids in the same way as two uncertain graphs. If the two ego-nets are a part of the same graph, then only the nodes that they have in common are considered aligned. For obvious reasons, the one exception is the center nodes, which are always mapped to each other.

The different cases of ego-network similarity, including additional cases described below, are outlined in figure 3.5.

In the sections that follow, we outline calculations for different cases of ego-network similarity. The proposed similarity measures are by no means exhaustive - additional extensions may be introduced, depending on data and user needs. Let’s first begin
by introducing some notation. Let \( o_1 \) and \( o_2 \) be the center vertices of the two ego-networks, \( eg_1 \) and \( eg_2 \), respectively. Let \( |eg_1| \) and \( |eg_2| \) be the size of the corresponding ego-net, measured as number of non-center vertices, i.e. the degree of the center vertex. In the aligned case, let a mapping \( rm \) be expressed as \( (e, v, e', v') \). In this notation, \( e \) and \( v \) are an edge and non-center vertex from \( eg_1 \), while \( e' \) and \( v' \) are the corresponding mapped edge and non-center vertex from \( eg_2 \). Let the alignment set \( RM \) be the set of all mappings between vertices and edges from \( eg_1 \) and \( eg_2 \), excluding the center vertex. By definition, \( RM \) does not include unmapped edge-vertex pairs. To represent this case, we define a pseudo-mapping \( pm \) in the same way as the regular mapping \( rm \), except that one side of the pseudo-mapping is always null: \( pm = (e, v, e', v') \), \((e = null, v = null) \) or \((e' = null, v' = null) \). Let the set \( PM \) contain all pseudo-mappings \( pm \) between \( eg_1 \) and \( eg_2 \). Then, the complete alignment set \( M \) is defined as \( RM \cup PM \). \(|RM|\), \(|PM|\), and \(|M|\) designate the sizes of the corresponding sets. In a similar fashion, we define a pair of mapped uncertain attributes as \( mp = (ua, ua') \), where \( ua = pa(v, PA_j) \) and \( ua' = (v', PA_j) \) share the same definition (domain and order of values). The set \( MP \) contains all pairs of mapped attributes between a pair of mapped vertices. Note that one side of \( mp \in MP \) can be \( null \).

**Ego-network Structural similarity**

Structural similarity is further subdivided into topological, probabilistic-topological, and comparison count. The user can select the similarity measure that is most applicable to the comparison.

Topological similarity compares the structure of the two ego-nets based on the existence of their elements, but not on confidence values associated with existence. It is calculated as follows:
- In the aligned case: \( \frac{|RM|}{\max(|eg1|, |eg2|)} \)

- In the unaligned case: \( \frac{|eg1|}{|eg2|} \) if \(|eg1| < |eg2|\), else \( \frac{|eg2|}{|eg1|} \)

Probabilistic-topological similarity takes into account the confidence values associated with edges and non-center vertices. It is calculated as follows. In the aligned case the calculation is: 

\[
1 - \sum_{m \in M} \text{abs}_\text{value} \left( \text{conf}(m.e) \times \text{conf}(m.v) - \text{conf}(m.e') \times \text{conf}(m.v') \right)
\]

In the case of pseudo-mappings, the side that is null results in a confidence product of 0. In the unaligned case, the same formula is used, but the complete alignment set \( M \) is created differently - in a manner similar to the merge-sort algorithm. For both ego-nets \( eg1 \) and \( eg2 \), we calculate the product of confidences for each non-center vertex and associated edge. These are sorted descending for both ego-nets separately. Each mapping \( rm \) is formed by taking the edge-vertex pair with highest values from each list and removing them from the list. When one of the lists is empty, pseudo-mappings \( pm \) are formed using the remaining elements of the other list. This algorithm, as most of our algorithms used in the unaligned case, is an approximation.

Comparison count is simply a count of aligned non-center nodes between the two ego-networks. It is useful when the analyst is interested in an absolute similarity measure, related to the size of the ego-networks, rather than in a ratio between 0 and 1 that is returned by the topological and probabilistic-topological similarity. In the aligned case, the value is \(|RM|\). In the unaligned case, the value is: \( \min(|eg1|, |eg2|) \).

**Semantic similarity**

Semantic similarity takes into account uncertain attribute values and probabilities, rather than confidence of existence of mapped vertices and edges. In the aligned case, similarity is measured by aggregating similarities between pairs of attributes with the same name and definition, belonging to each pair of aligned vertices. In the
unaligned case, alignment is chosen in an attempt to maximize the total similarity of all attribute pairs between aligned vertices - usually by greedy heuristics. Currently, semantic similarity between edges is not supported.

Depending on the number of attributes under consideration, the measure can be either single- or multiple-attribute. In both of those cases, similarity between a pair of uncertain attributes can be estimated using different measures. We propose two of them: mpv and distribution similarity. The former calculates the similarity between a pair of attributes to be 1 if there is either a full or a partial match between their sets of mpv values, and 0 otherwise. The latter represents the user’s choice of one of the uncertain attribute similarity measures, described in a previous section.

In the aligned case, attribute similarity is always calculated as:

\[ \sum_{rm \in RM} \frac{\text{similarity measure}(rm.v, rm.v')}{|RM|}, \text{ where } \text{similarity measure}(rm.v, rm.v') \text{ varies based on one of the single attribute or multiple attribute cases.} \]

**Case 1 - Single attribute.** Let \( PA_j \) be the selected uncertain attribute, and \( ua = pa(v, PA_j) \), \( ua' = pa(v', PA_j) \). Then one of the following measures can be used for similarity:

- **MPV.** The user can select between a partial or a full match between the attribute’s sets of mpv values: \( \text{similarity measure}(rm.v, rm.v') = 1, \text{if } \text{mpv}(ua) \cap \text{mpv}(ua') \neq \emptyset, \text{else} 0, \text{or } \text{similarity measure}(rm.v, rm.v') = 1, \text{if } \text{mpv}(ua) \equiv \text{mpv}(ua'), \text{else} 0. \) The result is also 0 if \( ua = null, ua' = null \), or both.

- **Distribution** In this case, the distribution of certainty values for an attribute is used: \( \text{similarity measure}(rm.v, rm.v') = sim(ua, ua') \), where sim is any user-selected semantic uncertain attribute similarity measure.

**Case 2 - Multiple attributes.** More options and complexity exist when considering multiple attributes. One of the following measures can be used for similarity:
- MPV. Between each pair of mapped uncertain attributes, we use the same similarity measure as in the case of single attribute: \( smpv(ua, ua') = 1 \), if \( mpv(ua) \cap mpv(ua') \neq \emptyset \), else 0, or \( smpv(ua, ua') = 1 \), if \( mpv(ua) \equiv mpv(ua') \), else 0. Because we deal with multiple mapped attributes \( mp \in MP \), the total similarity measure for the set \( MP \) can be derived in different ways from the similarity measure \( smpv \) between each \( mp \) pair.

  - ‘AND’ - the result is 1 if \( smpv(mp.ua, mp.ua') = 1 \), \( \forall mp \in MP \), else 0.

  - ‘OR’ - the result is the average of pairwise attribute similarity for all mapped attribute pairs: \( \frac{\sum_{mp \in MP} smpv(mp.ua, mp.ua')}{|MP|} \).

- Distribution. Between each pair of mapped uncertain attributes, we use the same similarity measure as in the case of single attribute: \( sdistr(ua, ua') = sim(ua, ua') \), where sim is any user-selected semantic uncertain attribute similarity measure. Because the values returned by \( sdistr \) are not restricted to either 1 or 0, we do not apply the ‘AND’ case in dealing with multiple mapped attributes \( mp \in MP \). The result is derived by averaging the pairwise attribute similarity for all mapped attribute pairs: \( \frac{\sum_{mp \in MP} sdistr(mp.ua, mp.ua')}{|MP|} \).

In the unaligned case, we restrict the similarity measure to a single attribute for considerations of computational complexity. Even in the case of a single attribute, the brute force approach for finding the alignment that would maximize similarity is highly inefficient in some cases. In those cases, we propose using a greedy heuristics, similar to the merge-sort algorithm, that reduces running time but does not guarantee optimality.

As in the aligned case, the user has a choice of two similarity measures, MPV and distribution based similarity.
**MPV:** When the ego-nets are not aligned, we impose the additional assumption that the selected uncertain attribute must have exactly one most probable value (mpv) for all vertices ($|\text{mpv}(pa(v, PA_j))| = 1, \forall v \in eg1$ and $|\text{mpv}(pa(v', PA_j))| = 1, \forall v' \in eg2$). The algorithm pairs vertices $eg1.v$ and $eg2.v'$ so that $\text{first}(\text{mpv}(pa(v, PA_j))) = \text{first}(\text{mpv}(pa(v', PA_j)))$. A set $par$ of all such pairs, where each vertex can appear at most once, in effect represents one possible alignment by attribute $PA_j$. There may be multiple ways of creating $par$ based on the same attribute. However, all of these sets have the same size, so the similarity is calculated as: $\frac{|par|}{\min(|eg1|, |eg2|)}$. We use as denominator the minimum of sizes of $eg1$ and $eg2$, because using $|par|$ by analogy with the aligned case of single attribute would always produce a value of 1.

**Distribution:** Similar to the aligned case, the user can select any semantic similarity measure for calculating similarity between a pair of uncertain attributes. The algorithm uses this measure to calculate similarity between the specified attribute for all pairs of non-center vertices from $eg1$ and $eg2$, i.e. $\text{sim}(pa(v, PA_j), pa(v', PA_j)), \forall v \in eg1, v' \in eg2$. Each mapping $rm$ is formed by taking the pair $(eg1.v, eg2.v')$ with highest similarity value from the sorted list and then removing from the list of all pairs containing $eg1.v$ or $eg2.v'$. After constructing the alignment set $RM$ in this fashion, the formula for the corresponding aligned case is applied. This is clearly an algorithm that makes a trade-off between accuracy and performance. It leaves room for improvement during next iterations of our uncertain operator implementation.
We have developed a prototype implementation of a query engine that supports the creation of queries using the operations and operators presented in the proposed query language. In this chapter we present the high level system architecture and an overview of the implementation, emphasizing details related to operator extensibility and composition.

There are many alternatives for implementing the proposed query language. Some of them involve extending existing implementations, including relational databases. Advantages of extending existing technologies include: building upon standard, mature technologies; reusing existing frameworks, services and operators; and taking advantage of implemented optimizations.

Our approach of implementing a custom, independent query engine was selected based on several considerations:

- Our focus is on operator / operation extensibility and composition. It is particularly important, because we anticipate the need to extend the proposed query language with new operators and capabilities. The freedom to develop an execution environment best suited for operator re-use and for seamless integration of future operators outweighed the benefits offered by extending an existing implementation.
Most existing implementations are optimized for a particular data model and query language, which do not always map efficiently to our uncertain graph data model and comparison language. For example, relational databases are very efficient in processing SQL queries on relational data; however, there are many examples of graph queries that cannot be executed efficiently if the graphs are stored in relational tables. Although query optimization is mostly outside of scope of this work, the capability of introducing optimizations specific to our data model and query language in the future was an important factor.

One of our goals was to create an implementation that is not restricted to a particular platform or underlying data storage.

In the following sections, we outline our design goals and present different views of our system. We start by describing an architecture that binds several query processing components into a single customizable system. Then we show the components involved in that process using an example query. The steps involved in integrating new operators are demonstrated, before we conclude with a discussion on operator composition and re-use.

4.1 Design goals

While we have previously mentioned the major factors driving our decisions in designing the query language implementation, here we summarize our highest priority design goals:

- Extensibility. We believe that the proposed query language provides a powerful analysis tool for comparing uncertain graphs. At the same time, we are aware that our set of operations and operators is by no means complete. It is, therefore, important to create an implementation that is open to extensions at all
levels. At the lowest level, the query language implementation must allow easy integration of any additional operators and operations. When the new operators or operations introduce concepts that do not fit in the existing implementation, it is desirable to minimize the required changes to the query processing framework. We refer to this as mid-level extensibility. We faced this challenge in this work when we began incorporating the first aggregate operators. At a high level, the system must provide room for new system capabilities, such as plugging in another data storage implementation.

- Operator composition. Within the proposed set of operators, we came across several cases, in which operators re-use the functionality of other existing operators. The best examples are perhaps the vertex and ego-network similarity operators, which build on top of different attribute similarity operators. Because we anticipate that being a common situation, the system should provide such functionality to the programmer, who creates new operators. The user can also compose operators implicitly within the limits of the query language by creating expressions or within the limits of the pre-programmed sub-operator selections, such as choosing an underlying attribute similarity measure.

- Room for optimization. Optimization in this context refers to the capability of creating and selecting more efficient query execution plans, rather than optimization of each separate operator. In the current iteration, which is more a proof-of-concept than a mature product, performance is not among our main concerns. However, the design should lend itself reasonably well to future optimization changes.

We attempt to accomplish these goals by using some well known object-oriented design techniques. Extensibility calls for abstraction, modularity and encapsulation.
We isolate the common functionality for ease of modification and re-use. We define interfaces and leave hooks for plugging in different implementations in places, which may require different behavior at some point. The specific details are described in the sections that follow.

We realize that versatility and flexibility come as a trade-off against performance. Because efficiency is lower on our priority list, we err on the side of generality, with the understanding that in the future releases some of it may be sacrificed in favor of optimization.

4.2 Query Engine Overview

In this section, we define the query engine as the platform on which we built the query language implementation. Our goal was to create a very small but generic platform, suitable for implementing other query languages as well. In other sections we may refer to the query engine in the broader sense, implying both the platform and our specific query language implementation.
Processing a query in any language normally involves several steps, such as parsing, validation, optimization, execution, etc. (see figure 4.1). These tasks to a large degree can be performed independently of each other. Therefore, we implement each of them as a module. The query engine then becomes nothing more than an environment for deployment and management of several services representing the respective modules. The high-level architecture view of the query engine is presented in figure 4.2. This diagram, as well as most diagrams that follow, is a conceptual view. It is not representative of the actual implementation.

The set of included services is not pre-defined. Any services that make sense for a particular query language can be deployed. The engine offers two important capabilities: service configuration and service lookup. The former allows the same
implementation to be deployed under different configurations, which can be useful, for example, in deploying the same parser with different grammars for languages co-existing in the same query engine. The latter allows flexible and dynamic linking of services. A possible, although not implemented example is a replacement of the underlying data storage implementation, which would be transparent to all other services that need to access it.

This architecture is loosely based on the concept of a service-oriented architecture. It shares some of its advantages, such as modularity, re-use, and flexibility in defining the query processing workflow. We believe that our query engine architecture accomplishes the goal of high-level extensibility.

4.3 Overview of query processing services

The query engine provides a platform for deployment of services, not necessarily related to query processing. The specific set of services that handle user queries represents the next layer in our architecture. The workflow implementing the proposed query language consists of several steps, represented by the Compiler, Validator, Optimizer, Executor, Facade, Connector, Data Store services.

As first step in the process, Compiler translates a query into an internal representation suitable for optimization and execution. Next, Optimizer generates and evaluates several alternative execution plans, choosing the best one. Executor is the key module where the operations and operators that make up the query are executed. Validator can be invoked at different stages to ensure compliance with the pre-defined rules. Facade and Connector provide the interface for interaction between external systems and the query processing workflow. Data Store serves to retrieve the data requested in the query.
Each of these components was also designed with focus on extensibility. The query engine incorporates a default implementation of each component, which offers the most general functionality and defers any specific or alternative behavior to lower level modules. Those are abstracted using simple interfaces, allowing easy customization by plugging in the appropriate implementation. For example, the default Executor uses a post-order traversal in executing the query, which can be substituted for an arbitrary traversal generated by the Optimizer. Of course, the whole default implementation of each service can be swapped in the query engine.

The two essential steps are query compilation and execution, discussed in more detail in the subsequent sections.

4.4 Query compilation

Let’s take the following simple query as a running example. Assume that graph $g_1$ captures a dolphin social network, where vertices and edges represent dolphins and their common sightings, respectively and $sex\_code$ is an uncertain attribute for each dolphin. The query selects the male dolphins from graph $g_1$, by selecting those dolphins whose set of most likely values for attribute $sex\_code$ contains ‘male’.

\[
\begin{align*}
\text{select} & \quad * \\
\text{from} & \quad g_1 \text{ type NODE as } n \\
\text{where} & \quad \text{‘male’ in mpv(n.sex\_code)}
\end{align*}
\]

The query consists of 3 operations: $select$, $from$, and $where$. Each of them references parameters, which specify the behavior of the operation. For example, the $from$ operation is supplied with the parameters ‘$g_1$’, ‘NODE’ and ‘$n$’ designating the source graph (‘$g_1$’), the type of data retrieved from the source graph (‘fromType’=‘NODE’), and the alias for further referencing the retrieved vertices (‘$n$’). Also notice that
a parameter is not limited to simple constants. It can be an expression consisting of several operators. This is demonstrated in the where operation of our example. Finally, there is an implicit data flow between the operations, in the form of passing the relation created with the from operation to where and then select operations.

To capture this data, we represent the query using a query tree, similar to the abstract syntax tree (AST) [1]. A general, conceptual view of a query tree for this type of query is presented in figure 4.3. Operations are connected to each other in a way that indicates the data flow between them, assuming a bottom-up evaluation. Each operation is parameterized according to its definition. Some of these parameters consist of one or more expressions, which will be executed within the parent operation.

We use a generic data model for query trees (figure 4.4), under which a query is composed of operations, constants and variables. The model does not distinguish between operations and operators.

Figure 4.5 shows the exact query tree for the example query. The nodes, visualized as yellow rectangles, represent operators; they are distinguished from operations by
defining the root of the operator expression tree as a constant within the scope of the parent operation, rather than as a direct child node.

Because such a query tree captures the semantics of the query, we refer to it as the logical query. In our design, we maintain a separate structure for a compiled query, which is executed by the query executor module. Mapping a logical query to the compiled query is the task of the compiler module. Decoupling the two structures from each other provides a clean separation between the specification of the user request and the internal operations taking place during query execution, such as optimizations. An additional benefit is that the queries can be precompiled and parameterized at a later time using the logical variable component of the logical query. This separation does not enforce pre-compilation. If interpretation is more desirable, compilation can be done at run-time.

In the default implementation, compiled query consists of CompiledOperations, CompiledConstants and CompiledVariables. These building blocks make up a structure very similar to its logical query counterpart, except that its elements are augmented with any additional data necessary for query optimization and execution.
Figure 4.5: Query tree specific to the simple select query.

This can include data for use by both the CompiledOperation and its environment. An example of the former is the operation’s configuration that usually affects its logic, such as the default threshold for the `bin` operator. The operation’s descriptor, including its parameter definitions and whether the operation is aggregate, serves as an example for the second case, in which the executor, optimizer and other services can query and analyze this data when deciding how to handle the query.

Therefore, the default compilation is straightforward. It performs post-order traversal of operations of the logical query tree, considering logical constants and variables as leaf nodes. At each node, the compiler:

1. Generates the compiled operation corresponding to the logical operation, or delegates processing to a compiler module, if such a module has been assigned to this operation.
2. If no module has been assigned to compile the operation, or the module signals that it does not handle the operation’s parameters, the constant parameters are processed by default as follows:

- If a constant parameter contains an expression, then expression compiler is invoked to generate a Compiled Query for the expression. The role of the expression compiler can be performed by the same or another compiler service, depending on configuration.

- Else, the constant is preserved by creating the corresponding Compiled-Constant.

3. Unless the variable parameters are handled by a compiler module, by default they are pre-processed by creating bindings that will be supplied with the parameter’s value before query execution. Expressions are not allowed to be represented as variables.

As evident from these steps and shown on figure 4.6, the compiler can be configured in several ways. The mapping between logical and compiled operations is given with the operation registry, which performs the role of compiled operation lookup table. For more customization, compiler modules can be assigned to operations that require special handling. This way, the default compiler can be configured for our proposed query languages, and at the same time it is generic enough to be used with other query languages conforming to the same logical structure. Adding new operations to the compiler in most cases requires minimum effort, involving only registering them with the operation registry.

The compiler does not perform validation directly. It can delegate this task to the validator module when processing each node of the query tree. However, for our implementation, this was not necessary. Instead, after the compiled query tree is
completely generated, validation is performed on the whole tree as a separate step in the query execution workflow.

4.5 Compiled Query Execution

The second essential step is compiled query execution. We will return later to the intermediate steps of validation and optimization that take place between query compilation and execution.

A common, simple implementation of expression trees uses operators that incorporate the tree structure and evaluation order in addition to operator logic. For example, the tree is composed of operator nodes that each contain references to their child operators (figure 4.7). When the root operator is executed, first it executes its child operators, and uses the results to perform its own logic. The child operators act in the same way. Therefore, the expression tree is evaluated in a bottom-up, post-
order fashion. The order of execution between the child operators is determined by the parent operator.

Such implementation is straightforward, but does not allow enough flexibility in evaluating the query tree, and therefore, optimizations are non-trivial. For example, execution of a join operation using an index\(^1\) requires first retrieving the relation without the index.

Another issue with the simple implementation is operator overloading. If the query tree node already incorporates the operator logic, it is problematic to dynamically substitute that logic at query execution time, depending on the type of operator’s arguments.

Out of these considerations, our implementation decouples the tree structure, the operator / operation logic, and the execution strategy (figure 4.8):

- The compiled query tree represents a generic tree structure that can accommodate any type of elements. It will not be affected by possible changes in operator implementation.

\(^1\)Our implementation does not support user-defined indices at this time
- The compiled operator is independent of the query tree structure and evaluation order: it is the task of the executor to evaluate the child elements and supply the results as arguments to the operator. The child elements can be evaluated in any order that satisfies the operator’s constraints. For instance, if one of the subtrees evaluates to null, for many operators the result is null, which makes it unnecessary to evaluate the remaining branches. In fact, we attempted to separate our operators from the Executor as well. The Executor uses the OperationInvoker interface to execute a particular operator, with the goal of minimizing the changes required to make the Executor compatible with a different operator implementation. Also, in many cases it is possible to use a CompiledOperation outside of our framework, by calling it directly, unless it relies on composition from other operations that are not part of the query tree.

- Finally, the execution strategy is abstracted as well. Currently, we evaluate the compiled query tree using post-order traversal. However, it is possible to use

---

2 Assuming no side effects, as is the case of our operators
alternative execution plans that can be created by the Optimizer. This design opens up the possibility for pipelining, if the operators support it.

The high-level design of the default Executor is shown in figure 4.9. It goes through the following steps in executing a compiled query that consists of CompiledOperations, CompiledConstants, and CompiledVariables:

Use the execution strategy to traverse the compiled query elements, currently equivalent to post-order traversal.
1. If the element represents a CompiledConstant or a CompiledVariable, retrieve its value either directly or from the set of variable values passed to the Executor for the specific execution instance of the compiled query.

2. If the node represents a CompiledOperation:

   (a) Dynamically resolve the required operation based on the CompiledOperation descriptor and parameters. External resolver is used, which in our implementation is combined with operation invoker from the next step - serving as a facade to operation handling. Dynamic resolving allows operator overloading for different run-time parameters.

   (b) Call all validators configured for the specific CompiledOperation. By default, all operations include ParameterValidator that verifies that the actual parameters comply with the expected parameters, but any operation-specific validation can be configured as well.

   (c) Use OperationInvoker to execute the operation, passing the parameters and the results of the child operations in the Compiled Tree.

   We use this simple and generic DefaultExecutor to handle the operations and operators of our proposed query languages. Although neither the CompiledQuery nor the Executor distinguish between operations and operators, such distinction is accommodated by including the operator expressions as constant arguments to the top-level operations. During traversal of the query tree, the expression is retrieved from the constant, but not executed by the Executor. It is up to the top-level operation to invoke the expression through an Executor service. We use the same executor, but it is possible to configure another Executor service to handle operator expressions.
Both the semantics and implementation of our query language specify that operations pass relations to the parent operation in the query tree. At the same time, operator expressions apply to individual tuples of these relations. In our example query, the clause

\[
\text{where } 'male' \text{ in mpv(n.sex_code)}
\]

consists of the \textit{where} operation that executes the \textit{‘male’ in mpv(n.sex_code)} operator expression on each tuple of the relation, and removing tuples that do not satisfy the condition. This sequence of steps is part of the \textit{where} operation implementation.

Executing the expression on a specific tuple takes advantage of another capability of our framework, Context. Each operator during execution has access to the Context, from which it can retrieve its configuration and any other data previously bound to the Context. Therefore, the parent operation passes tuples one at a time by binding them to a pre-defined Context variable. The tuples are retrieved from the context by the \textit{path} operator, which can also access the specified column of the tuple.

The Context mechanism is more general and has other applications. For example, composed operators, such as ego-network or node similarity, take advantage of this mechanism by retrieving some of their configuration parameters from the Context. Because the user does not directly specify the nested operators, it is not possible to pass their configuration as regular parameters; instead, the configuration is bound to the enclosing operator and passed down to the nested operators.

As implementation note, aggregate operator expressions are executed using a modified version of the DefaultExecutor. After validating that each branch of the query tree contains exactly one aggregate operator, the subtrees of these operators are executed once for each tuple. The obtained set of results is passed to the aggregate
operator, which returns a single value. From there, the parent operators in the tree are evaluated in the same way as by non-aggregate executor.

We believe that the developed basic Executor module provides sufficient flexibility to implementing additional operators and operations. Unless they require some unsupported capability, integrating a new operator or operation involves simply adding it to the lookup registry, which is used by the current OperationInvoker to resolve the CompiledOperation to its implementation. Attaching custom validators in addition to the default validator allows creating validators for a broader class of operators.

4.6 Other modules

The query processing workflow includes several other services:

- Validator is invoked at different stages of query processing to ensure compliance with pre-defined rules. It can validate a single operation or the complete query tree. The validator is usually invoked twice, for compile time and runtime validation. The former detects some early issues with the query, but the majority of the work is done during query execution, on the actual data that is being processed. As mentioned earlier, the validator allows to register custom validators for a set of operations or operators.

- Optimizer is responsible for generating and evaluating several alternative execution plans before choosing the best one. This can be accomplished by reordering and substituting operations in the compiled query tree, or by creating a traversal plan. Because optimization is not part of this work, this module serves as a placeholder for future development efforts.

- Facade and Connector provide the interface for interaction between external systems and the query processing workflow. They both present a high-level interface
for compiling and executing queries, hiding the details of the services behind them. Facade is developed for systems that integrate our query engine, and therefore, communicate with it directly within the same Java virtual machine. Connector is intended for application that issue remote queries through different protocols, but at this point it offers a single, local protocol.

- DataStore has the task of retrieving uncertain graphs from the underlying storage. For this iteration, we use in-memory storage or file storage.

The interaction between those components, which work together on processing user queries, is shown on sequence diagrams 4.10 and 4.11. The first diagram shows the steps that make up the request to compile a query. The steps for executing a compiled query are shown in the second diagram.

4.7 Developing and Composing Operators

In this section we discuss the process of incorporating new operators and composing existing ones. While we use the term "operator", operations use the same mechanism.

Developing an operator involves implementing a simple interface with two methods. The first method allows the Executor to set the operator’s input parameters. Then, the second method is called, in which the operator performs its calculations over these supplied parameters and returns the result.

In the simple case, no other code is required. Adding the operator to the configuration of the OperationRegistry is sufficient to incorporate it into the query language, as the registry is used for both compilation and execution.

For basic validation, or in case the operator is overloaded, we also need to provide an operator descriptor, which specifies the expected parameter types and whether the parameters are required. The operator descriptor is automatically used to validate the
Figure 4.10: Sequence diagram of query preparation.

Figure 4.11: Sequence diagram of query execution.
actual parameters against the expected parameters before invocation. In the future, the descriptor will be extended with additional data that will be utilized during query optimization.

If fine-tuning is necessary, custom modules can be registered for the specific operator at different stages of query processing. The operator can be compiled using a custom compiler module. For additional validation, the operator can be linked to pre-defined or new validators. New validation rules may be defined for the whole query tree as well, to ensure that the operator is not used outside of its intended context. Finally, at the execution step, the operator can be configured with pre- and post-invocation filters, which can process the input parameters or modify the operator output. Our approach combines simple setup in the basic case with flexibility in deploying complex operators.

Another important mechanism is operator composition. Our framework supports operator re-use by allowing the operator developer to invoke other, existing operators. This functionality is used, for example, in the element level similarity operator, which returns the average attribute similarity between all pairs of uncertain attributes of the two elements. Behind the scenes, the same invocation framework is used; however, the invocation details are hidden from the calling operator. Instead, it uses a simple invocation interface, whose implementation instance is obtained from the Context in runtime.

In addition, the operator composition mechanism decouples the calling operator from the implementation of the nested operator. Instead of the outer operator having to create an instance of a particular CompiledOperation, it only references the desired nested operator by name. In deployment time, that name is bound to the operator that will be used as the nested operator. Not only operator instantiation is transparent, but also the inner operator can be substituted during deployment without any code
changes. The outer operator can take advantage of all the filters configured for the nested operator, or bypass them and invoke the operator directly.

While the above mechanism is sufficient for composing operators statically, it does not provide enough flexibility when the nested operator is not known until run-time. For those cases, an operator can define an arbitrary logical query, compile, and execute it using the same framework. To avoid redundant multiple compilation, the operator can parameterize the compiled query before each execution. On the downside, the query definition, compilation, and execution are not transparent to the outer operator any more.

4.8 Uncertain graph data model and data types

After reviewing query representation and processing, we use the UML diagram in figure 4.12 to recapitulate the uncertain graph data model, over which the queries are defined and executed. The diagram in figure 4.13 includes the major data types, other than the uncertain graph data model, that are an integral part of the proposed query language.
Figure 4.12: Uncertain graph data model and essential data types.
Figure 4.13: Uncertain graph data model and essential data types.
In this chapter, we describe a set of sample queries that can be formulated using our proposed operators and query language. These queries fall into several categories of uncertain graph analysis and comparison, which, in our opinion, are likely to be of interest to the users. The various queries that we have identified within these groups demonstrate the capabilities of the specific operators as well as their composition by the user for the purpose of obtaining the desired results.

We integrated our prototype query engine with Invenio, a visual analytic tool for graph mining [19]. Because our implementation at this point does not include a parser, the base queries are pre-programmed and included as menu items, grouped into different sub-menus. Selecting an item brings up an editable XML representation of the query, which allows the user to modify the pre-programmed query as desired. Although it is possible to enter a new, arbitrary query, typically the user would override some of the query parameters, such as the input graph(s) or the name of the dolphin that needs to be selected, as shown in figure 5.1.

The user can execute the query several times, with different parameters. The latest version of the XML query is preserved between query invocations, and the results for each invocation are displayed in tabular form, keeping the previous results in their separate tabs (figure 5.2). For those queries, whose returned results include graphs, the user can select any cell containing a graph and display it using the visualization capabilities of Invenio (figure 5.3).
Figure 5.1: Editable XML query representation.

Figure 5.2: Multiple results displayed as tables in separate tabs.
Figure 5.3: Displaying a graph returned as query result.

We demonstrate the sample queries using a real world dataset, collected through scientific observations by members of the Shark Bay Dolphin Research Project over the course of more than 20 years [14]. It contains demographic data about approximately 800 dolphins, represented as graph vertices with certain attributes such as \((id, \text{conf}, \text{dolphin\_name, birth\_date})\) and uncertain attributes that include \((\text{sex\_code, location, mortality\_status\_code})\). Survey data about social interactions between these dolphins is captured as approximately 29000 edges\(^1\) with attributes \((id, \text{conf})\).

In the queries presented in the sections that follow, the two uncertain source graphs are referred to as \(g1\) and \(g2\). For each query, we briefly outline the semantics

\(^1\)For some queries, we use different variations of this dataset, such as output from node labeling algorithms.
and the intended question of interest, followed by results returned after executing some of the more interesting queries.

5.1 Simple queries (Uncertain node filter)

One of the basic functions that the user may need is restricting the set of graph elements based on some condition.

The following example query selects the names of those dolphins, whose most likely location includes ‘RCB’. The results are shown in figure 5.4.

```
select dolphin_name
from g1 type NODE
where 'RCB' in mpv(location)
```

The second simple query provides the reverse functionality: instead of retrieving all dolphins who satisfy a certain filtering condition, it returns data about a particular
dolphin of interest. More specifically, it gives the user the primary location of the dolphin named ‘OCEANIA’ (see figure 5.5):

\[
\text{select mpv(location)} \\
\text{from g1 type NODE} \\
\text{where dolphin_name = ‘OCEANIA’}
\]

Certainly, it is possible to use more complex expressions than the ones that we chose as illustration. In addition, using graph structural operators such as \text{isAdjacent} allows formulating queries about dolphin relationships. For example, the following query returns true if a pair of specific dolphins were observed together:

\[
\text{select n1.dolphin_name, n2.dolphin_name, isAdjacent(n1, n2)} \\
\text{from (} \\
\text{(select n1} \\
\text{from g1 type NODE as n1} \\
\text{where n1.dolphin_name = ‘JOY’)} \\
\text{) join} \\
\text{(select n2} \\
\text{from g1 type NODE as n2} \\
\text{where n2.dolphin_name = ‘JOYSFRIEND’)} \\
\text{)}
\]

Executing this query does confirm that JOY and JOYSFRIEND have been observed together.
5.2 Uncertain subgraph construction

The result of a simple node filtering query is a relation of nodes, rather than a graph. The $to\_graph$ operator can be used to create a graph from a given set of graph elements. In combination, these constructs provide the capability to select a subgraph of a given graph. Because the $to\_graph$ operator takes as input a set of elements and not a relation, we need to use the $merge$ operation to convert between them.

The query below, given the graph $g1$, selects a subgraph that consists only of dolphins and observations with high confidence of existence (0.6). The selection of such elements is accomplished by the $bin$ operator, $merge$ is used to convert the resulting relation into a set, and $to\_graph$ re-constructs a graph from that set:

```sql
select to_graph(e)
from g1 type ELEMENT as e
where bin(e, 0.6)
merge all
```

Figure 5.6 shows that the resulting subgraph is dense, indicating that the majority of the original graph’s elements have high confidence of existence. Continuing the
analysis, a researcher may use the following query to isolate and visualize the edges with confidence below or equal to 0.75:

```sql
select to_graph(g) as graph
from
  (select g from
   (select to_elements(to_graph(e)) as g
    from g1 type ELEMENT as e
    where or(isVertex(e), and(isEdge(e), not(bin(e), 0.75)))
    merge all)
  )
split by g
where or(isEdge(g), and(isVertex(g), not(isEmpty(incidentElements(g)))
merge all
```

The relative complexity of this query in comparison with the previous one is due to the requirement of not only selecting the low confidence edges, but also their incident vertices, which are necessary for visualizing those edges in a graph. Thus, the innermost WHERE operation retains the desired edges and all vertices, while the outer WHERE operation filters out the vertices that do not have any low confidence edges associated with them. The resulting graph is presented in figure 5.7. The query also demonstrates using the MERGE BY, SPLIT BY operations in combination with the to_graph, to_elements operators to accomplish conversion and transition between graphs and their elements.

To visualize the ego-network of a specific dolphin, the two operators (to_graph and ego_net) can be used in combination:

```sql
select to_graph(ego_net(n))
from g1 type NODE as n
where n.dolphin_name = ?
```
Figure 5.7: Subgraph of edges with $\text{bin}(0.75) = \text{false}$ and their incident vertices.

Figure 5.8: Ego-networks of dolphins ‘JOY’ and ‘UPP’.
By running the same query twice, for dolphins ‘JOY’ and ‘UPP’, the user can observe the difference in sizes of their respective ego-networks (figure 5.8).

5.3 **Group By (Groupings)**

The *Group By* operation allows the user to assign each tuple to a specific group based on some of the tuple’s values. Then aggregate information about each group can be queried.

In the simplest case, when the *Group By* is not present, the aggregate information applies to the whole relation. The following query returns the number of vertices in graph g1, which in our example represents the number of dolphins (figure 5.9):

```sql
select count(*)
from g1 type NODE
```

The *Group By* operation is included when the user needs aggregate information for each group. In case when grouping is based on uncertain attributes, the user will likely group by most probable value. One issue with that choice is that the *mpv* operator returns a set of values. Therefore, if the user is interested in finding out how
many times each domain value is included as mpv, the query needs to take advantage of the $SPLIT\ BY$ operation. To query, for each location, the number of dolphins, which have that particular location among the most probable values of their location attribute:

```
select loc, count(*)
from (select n, mpv(n.location) as loc
     from g1 type NODE as n
     split by n.location)
group by loc
```

Note that the sum of counts by location (figure 5.10) will exceed the total number of dolphins (vertices), if mpv(location) returns more than one value for at least one dolphin.

5.4 UNION, INTERSECTION, DIFFERENCE, BIDIRECTIONAL_DIFFERENCE

This group of queries presents the capability of analyzing structural similarities and differences between a pair of aligned graphs. The general query format for retrieving the union of two graphs is the following:
select union(g1, g2) 
from ( 
    (select * 
    from g1 type GRAPH as g1) 
    join 
    (select * 
    from g2 type GRAPH as g2) 
) 

The above query constructs a graph containing dolphins and observations that are present in at least one of the source graphs g1 and g2. We can build upon this query to answer interesting questions, such as identifying common friends of dolphins ‘JOY’ and ‘PUCK’:

select intersection(ego1, ego2) as commonFriends 
from ( 
    (select to_graph( ego_net(n1) ) as ego1 
    from g1 type NODE as n1 
    where n1.dolphin_name = ‘JOY’) 
    join 
    (select to_graph( ego_net(n2) ) as ego2 
    from g1 type NODE as n2 
    where n2.dolphin_name = ‘PUCK’) 
) 

The graph obtained as result of executing this query is visualized in figure 5.11. The only two connected vertices are ‘JOY’ and ‘PUCK’, showing that they belong to each other’s ego-nets, after all other edges were removed during intersection of the two ego-networks.

The query can be slightly modified to compare ego-networks between two graphs. For example, if g1 and g2 are subgraphs of dolphin observations during years 2009 and 2010, respectively, the user can analyze changes in a particular dolphin’s ego-network as follows:
select union(ego1, ego2) as union, intersection(ego1, ego2) as intersect,
difference(ego2, ego1) as diff, bidirectionalDifference(ego2, ego1) as biDiff
from ( 
  (select to_graph( ego_net(n1) ) as ego1
      from g1 type NODE as n1
      where n1.dolphin_name = ‘JOY’)
  join 
  (select to_graph( ego_net(n2) ) as ego2
      from g2 type NODE as n2
      where n2.dolphin_name = ‘JOY’)
)

It is easy to visualize (figure 5.12) that the dolphin has almost as many new friends (difference) as repeat friends, i.e. occurring during both years (intersection). Researchers can then visually explore who these friends are, what gender they are, etc., to gain more insight about dolphin sociality. Note the order of operands supplied to the difference operator (ego2, ego1), which returns the dolphin’s new friends during 2010 vs. 2009. If the order were reversed, the operator would produce the ”old” friends, i.e. those who were seen together with the dolphin in 2009 but not in 2010.
Currently union, intersection, difference, bidirectional difference are implemented as operators that work on a pair of graphs. One of the planned future extensions is implementing them as operations as well, which would provide the same functionality over a set of graph elements, potentially contained in two separate relations.

5.5 Comparison

In this group we include a set of queries that compare the same information between two aligned graphs $g_1$ and $g_2$, for example, output graphs from two node labeling algorithms.

The first query is used to select pairs of aligned edges from $g_1$ and $g_2$, for which $\text{diff\_significance}$ is true. In our example, this represents those observations, for which the two graphs strongly disagree about their confidence of existence.

```sql
select g1Edge, g2Edge
from (  
  (select g1Edge  
    from g1 type EDGE as g1)  
)  
```
join
(select g2Edge
from g2 type Edge as g2Edge)
on g1Edge.id = g2Edge.id
)
where diff_significance(g1Edge, g2Edge)

While the above query applies to edges, attributes can be analyzed in a similar manner. Below we sort pairs of aligned nodes from graphs g1 and g2 by difference in maximum certainty of their location attribute. By examining the top or the bottom of the list, the user can concentrate on those dolphins, the certainty for whose location is either most or least similar between the two graphs.

select g1Node, g2Node,
    max_value_certainty(g1Node.location) as g1LocCertainty,
    max_value_certainty(g2Node.location) as g2LocCertainty,
    abs_value(max_value_certainty(g1Node.location) -
        max_value_certainty(g2Node.location)) as diffCertainty
from (
    (select g1Node
     from g1 type NODE as g1Node)
    join
    (select g2Node
     from g2 type NODE as g2Node)
on g1Node.id = g2Node.id
)
order by abs_value(max_value_certainty(g1Node.location) -
    max_value_certainty(g2Node.location))

Figure 5.13 is a screenshot of the results obtained by executing the query on a pair of graphs, produced by two different node labeling algorithms, which predict each dolphin’s location.
5.6 Similarity

The queries in this group use various attribute, element, and ego-network similarity operators.

A simple example calculates similarity between two particular dolphins in graphs g1 and g2, using default similarity measure, which averages the pairwise similarity between the corresponding uncertain attributes:

```
select g1Node, g2Node, sim(g1Node, g2Node) as nodeSim
from (select g1Node
      from g1 type NODE as g1Node
      where g1Node.dolphin_name = ?)
join
```
For dolphin ‘OCEANIA’, the query returns nodeSim=0.8571428606907526.

As illustrated in the query below, the user can specify the desired similarity measure. The expression \{measure=KLDivergence\} represents a configuration property passed to the sim operator as a context parameter. The query calculates K-L divergence between the location attributes of aligned pairs of dolphins. Unlike the first query, it does not measure element similarity and it is not restricted to a single pair of dolphins.

```
select g1Node, g2Node, sim(g1Node.location, g2Node.location,
   \{measure=KLDivergence\})
from (   (select g1Node
         from g1 type NODE as g1Node)
       join
       (select g2Node
         from g2 type NODE as g2Node)
on g1Node.id = g2Node.id
)
```

To calculate probabilistic-topological similarity between the aligned ego-nets of two particular dolphins in graph g1 and g2, we can use the following query that also includes similarity measure configuration parameters:

```
select g1Node, g2Node, ego-sim(g1Node, g2Node,
   \{type=probabilistic-topological, aligned=true\})
from (   (select g1Node
         from g1 type NODE as g1Node
         where g1Node.id = ?)
       join
       (select g2Node
         from g2 type NODE as g2Node
         where g2Node.dolphin_name = ?)
   )
)
By slightly modifying the query, we can find the unaligned ego-nets from graph g2, most similar to the ego-net of a particular dolphin in graph g1, using semantic similarity measure based on mpv of location attribute:

```
select g1Node, g2Node, ego-sim(g1Node, g2Node,
   {type=semantic, aligned=false, measure=mpv, attr=location})
from (   (select g1Node
   from g1 type NODE as g1Node
   where g1Node.id = ?)
   join
   (select g2Node
   from g2 type NODE as g2Node)
)
order by ego-sim(g1Node, g2Node,
   {type=semantic, aligned=false, measure=mpv, attr=location}) DESC
```

5.7 Dolphin gender analysis

The queries reviewed so far highlight the capabilities and various functionality aspects of the proposed operators and operations. In this section, we present several sample queries that can be used in accomplishing a specific task: understanding the differences in dolphin sociality by gender.

Executing the following query, we discover that male dolphins are more social, as measured by the average number of their friends: 51.2 compared to 32.6 for female dolphins.

```
select gender, average(cntAdj) as avgCount
from (   
```
select n, size(adjacentVertices(n)) as cntAdj
from g1 type NODE as n
)
group by first(mpv(n.sex_code)) as gender

To obtain a breakdown of the number of dolphin common sightings by sex of their two participants:

select gender, genderAdj, count(adj) as a
from
(  
  select *
  from
  (  
    select n, adjacentVertices(n) as adj
    from g1 type node as n
  )
  split by adj
)
group by first(mpv(n.sex_code)) as gender,
  first(mpv(adj.sex_code)) as genderAdj

The results (figure 5.14) reveal that male dolphins are significantly more likely to be seen together, compared to each of the other two possibilities (male-female and female-female). These findings are consistent with the results of the previous query.

5.8 Discussion

With the preceding examples, we have demonstrated the usability of the proposed query language and uncertain graph comparison operators in examining a real world dataset. Applying the chosen pre-programmed queries, analysts can answer questions that include:

- retrieving dolphins who satisfy a certain condition or returning data about one or more particular dolphins.
- constructing and comparing subgraphs, including dolphins’ ego-networks.

- querying aggregate information, such as count of dolphins by location.

- performing structural comparison between a pair of aligned graphs: finding common friends of two specific dolphins or examining changes in a single dolphin’s ego-network.

- analyzing structural and semantic similarity between graph elements, attributes, and ego-networks, either within the same graph, or between a pair of output graphs from two different node labeling algorithms.

- discovering patterns in data, such as the higher percentage of observations of male dolphins.

These queries show that the proposed query language and operators can be successfully used not only for the initial intended application of uncertain graph comparison, but also for more general graph analysis. In the future, implementing a parser will further expand their usefulness well beyond the pre-programmed queries in the demonstrated proof-of-concept implementation, by allowing the users to define their own ad-hoc queries.
In addition to demonstrating the functionality of the proposed uncertain graph operators, executing the queries within the context of Invenio shows that our framework can be integrated with an existing visual graph analysis tool.
E. F. Codd in [8] defines a data model as a combination of 3 components:

1. a collection of data structure types;

2. a collection of operators or inferencing rules, which can be applied to any valid instances of these data types to retrieve or derive data from any parts of those structures in any combinations desired;

3. a collection of general integrity rules.

Using this definition, our research problem can be expressed as creating a data model, where:

1. the data structures are graphs, enhanced with uncertainty information;

2. the collection of operators that have some generality for retrieving and deriving data, but focus on the specific task of comparing such graphs;

3. the integrity rules are secondary. Instead of allowing user-defined constraints, we define constraints only for the purposes of scoping the research work and of providing correct input to the comparison operators.

Recall that our emphasis is on extensible, flexible system design. Therefore, the solution spans different areas of systems and database research. Since the basic underlying data structures of our model are graphs and our operators focus on graph manipulation, one obvious area of related research is graph databases. Graph databases
cover the problem of storing, retrieving, querying, and manipulating data in graph format. In most cases these databases define a language and operators that allow users to express their queries. They also implement various optimizations for the most resource-intensive graph operations, such as graph search and pattern matching [18], [12].

6.1 Graph Database Implementations

Three mature implementations of graph databases or query languages include Gremlin [2], Neo4J [3], and OrientDB [4]. These databases, to a varying degree, provide syntactic and semantic query constructs for retrieval and traversal of graphs and their elements, which could be used in conjunction with graph comparison operators. Some of them offer useful features often found in traditional databases, such as transaction support, indexing, high availability, etc.

More specifically, Gremlin is a relatively simple but powerful language for graph traversal and manipulation. This is achieved by using scripts, functions, data structures such as lists, sets, maps, and an XPath-like syntax (called GPath) featuring regular expressions, backtracking, etc. Gremlin supports built-in functions such as union, difference, intersection that are applicable to graph comparison. As an added benefit, it includes connectors to several graph formats: TinkerGraph; Neo4j; OrientDB; JUNG, Sesame Sail Quad Store; Rexster RESTful Graph Shell, and other implementations of the graph interfaces provided by Blueprints.

Both OrientDB and Neo4J are high-performance, scalable transactional databases. OrientDB features a subset of SQL as its query language, which is similar to the language developed in this work to express the comparison operators. Neo4J does not support declarative queries and instead offers programmatic navigation between
nodes and relationships via a powerful traverser framework. Both databases have multiple programming language bindings.

However, none of these implementations include extensions for uncertain graphs or operations specific to comparison of those uncertain graphs. While one could extend their query engines to incorporate the uncertain semantics and operators, this option is either non-trivial or introduces new issues. For example, the whole SELECT statement in OrientDB is defined and handled in a single class, making it difficult to add new clauses (such as different join types) or to optimize the statement execution. Because the benefits offered by these databases are not central to our research, we instead concentrate on creating a flexible, extensible query framework from scratch. One possible direction for future work would be taking advantage of the additional features of such graph databases by integrating them as underlying data stores.

6.2 Graph database research

In addition to the fully functional, industry standard graph databases mentioned above, there are a number of graph database models proposed in the research community, with prototype implementations for some of these models. An overview of these models is presented in [6]. Some graph data models, query languages, and algorithms [18], [5], [11], [9], [10], [5] do not incorporate uncertainty. Several other, recent publications on uncertain graphs either focus on specific algorithms [7], [16], or study uncertainty arising from approximate queries rather than from probabilistic data [22]. To the best of our knowledge, no query language has been developed for comparison of uncertain graphs. The operators that we are proposing can serve as a foundation for such a language.
6.3 Probabilistic Databases

Probabilistic databases that we have surveyed typically extend relational databases in terms of data models, query languages and implementations by adding a notion of probability to different aspects of the relational model. Many of these databases support queries based on the concept of Possible World Semantics [13], [21], [20]. While applicable, this concept is different from our focus on graph comparison. Others have probabilistic attributes similar to ours, and some related research defines operators, such as comparison of probabilistic attributes [20], that are applicable in the context of graph comparison. However, their data sets are traditional relational data, whereas our focus is on graphs. Graphs can be modeled as relational data, but many interesting graph queries are not well suited to that representation, such as isomorphic subgraph matching and shortest path queries. So, while graph databases and probabilistic databases each support parts of a solution, neither provides a complete solution.

6.4 Java Scripting Engine API

Java Specification Request (JSR) 223 helps developers integrate Java technology and scripting languages by defining a standard framework and application programming interface (API). Our proposed graph comparison language is not a scripting language; yet some universal design ideas from this API are included in our framework.
CHAPTER 7

CONCLUSIONS AND FUTURE DIRECTIONS

This work represents a first step in solving the problem of comparing graphs with uncertainty, using a flexible set of operators that consider the graph structure, the graph semantics, and the inherent uncertainty in the application domain. We address the problem by proposing a collection of operators for comparing uncertain graphs and a basic, SQL-like query language, allowing the users to compare and analyze these graphs by combining operators. We have designed a supporting system architecture, which places emphasis on operator composition and extensibility. This architecture is well suited for incorporating additional operators in the future, as well as for re-using the existing ones as building blocks for more complex operators. To evaluate the feasibility and usability of our solution, we have developed a prototype implementation of the proposed system architecture. This implementation has been integrated with Invenio and successfully applied to a real-world data set in order to demonstrate how our solution can be useful to scientists in answering some of their research questions.

While we believe it to be a successful attempt, there are many areas that were out of scope of this first iteration. These areas represent interesting and promising directions, on which we intend to concentrate our future research efforts:

- Optimizing computationally intensive operators and the framework as a whole. The former task includes developing more efficient algorithms for specific operators, such as ego-network similarity, beyond the initial optimizations suggested
in this thesis. The latter task entails designing a system-level optimizer, which can create and compare a number of execution plans for the whole query, selecting the plan with the best performance.

- Developing a formal query language and parser. The capability to parse and execute ad-hoc queries that conform to a set of pre-defined syntactic and semantic rules would be a major usability enhancement for the proposed system in real-life application.

- Improving integration with Invenio and other tools for visual analysis. Users can benefit from richer and more interactive visualization of query results than their simple presentation as graphs or tables. Conversely, visual analysis tools such as GPare can increase their flexibility and functionality by delegating the uncertain graph comparison computations to our query engine and using queries in place of the custom algorithms behind the scenes.

- Introducing ranking operators. As demonstrated, it is possible to obtain the desired sequence of results using the \textit{ORDER BY} operation. However, a dedicated ranking operator can extend this functionality and allow for performance optimizations.

- Incorporating continuous uncertain attributes and corresponding operators. Extending the proposed operators to accommodate such data types will expand the application of our query language to datasets that are not restricted to discrete uncertain attributes.
Bibliography


