



# THE UNIVERSITY *of* EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.

This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

# Functional Inferences Over Heterogeneous Data

*Kwabena Amoako Nuamah*



Doctor of Philosophy

Centre for Intelligent Systems and their Applications

School of Informatics

University of Edinburgh

2018



# Abstract

Inference enables an agent to create new knowledge from old or discover implicit relationships between concepts in a knowledge base (KB), provided that appropriate techniques are employed to deal with ambiguous, incomplete and sometimes erroneous data.

The ever-increasing volumes of KBs on the web, available for use by automated systems, present an opportunity to leverage the available knowledge in order to improve the inference process in automated query answering systems. This thesis focuses on the *FRANK* (Functional Reasoning for Acquiring Novel Knowledge) framework that responds to queries where no suitable answer is readily contained in any available data source, using a variety of inference operations.

Most question answering and information retrieval systems assume that answers to queries are stored in some form in the KB, thereby limiting the range of answers they can find. We take an approach motivated by rich forms of inference using techniques, such as regression, for prediction. For instance, FRANK can answer “*what country in Europe will have the largest population in 2021?*” by decomposing Europe geo-spatially, using regression on country population for past years and selecting the country with the largest predicted value. Our technique, which we refer to as *Rich Inference*, combines heuristics, logic and statistical methods to infer novel answers to queries. It also determines what facts are needed for inference, searches for them, and then integrates the diverse facts and their formalisms into a local query-specific inference tree.

Our primary contribution in this thesis is the inference algorithm on which FRANK works. This includes (1) the process of recursively decomposing queries in way that allows variables in the query to be instantiated by facts in KBs; (2) the use of aggregate functions to perform arithmetic and statistical operations (e.g. prediction) to infer new values from child nodes; and (3) the estimation and propagation of uncertainty values into the returned answer based on errors introduced by noise in the KBs or errors introduced by aggregate functions.

We also discuss many of the core concepts and modules that constitute FRANK. We explain the internal “alist” representation of FRANK that gives it the required flexibility to tackle different kinds of problems with minimal changes to its internal representation. We discuss the grammar for a simple query language that allows users to express queries in a formal way, such that we avoid the complexities of natural

language queries, a problem that falls outside the scope of this thesis. We evaluate the framework with datasets from open sources.

# Acknowledgements

I am very grateful to my primary supervisor, Alan Bundy. He offered me exceptional guidance and insights into doing research in Informatics. I always looked forward to our regular meetings for his feedback on drafts of this thesis. Beyond his dedication and commitment as a supervisor, Alan has been a mentor and a friend, always ready to help. He has proved to be the best supervisor one can have for a PhD.

I would also like to thank my secondary supervisor Christopher Lucas for making time to meet to discuss my work, for reviewing drafts of this thesis and for providing me with useful feedback and pointers to related work, relevant tools and software libraries.

I am also very thankful to my examiners, Vaishak Belle and Frank van Harmelen for making time to read this work. Their insightful comments have led to a much-improved manuscript and given me new perspectives on my work.

The Centre for Intelligent Systems and their Applications (CISA), together with the Mary and Armeane Choksi Postgraduate Scholarship from the School of Informatics provided funding for this PhD, for which I am very thankful.

Special thanks to members of CISA and the DReaM research groups for offering very useful perspectives and feedback each time I shared progress on my work (both formal and informal) from the very early stages in my PhD to the final reviews. I am also grateful to these research groups for their encouragement and their funding of my attendance to academic workshops and conferences within the UK and at other venues in Europe.

My eternal gratitude goes to my parents, Sampson and Mercy Nuamah, who have had great faith in my ability to complete this work towards the award of a PhD. Words cannot express my appreciation for their immense influence, unwavering support and prayers in all achievements of my life so far. Special thanks, also, to my wonderful sisters Nana Akua and Ama for their ever-present support and encouragement.

Finally, a massive thank you to my amazing wife Rose for being extremely supportive and showing such great faith in me. Her motivation and encouragement have been invaluable throughout this work. Thanks Rose, for your love and companionship. Thanks, also, for taking such good care of our lovely daughter, Rena, who has been the perfect gift to us in the final months of my PhD, bringing smiles to my face every day.

Above all, I am immensely grateful to God for giving me the grace and strength to complete this work.

# Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

*(Kwabena Amoako Nuamah)*

# Table of Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Question Answering . . . . .	5
1.2	Motivation . . . . .	6
1.2.1	Limitations of Current Question Answering Techniques . . . . .	6
1.2.2	Beyond Fact and Answer Retrieval . . . . .	6
1.3	The Uncertainty Problem . . . . .	7
1.4	Hypothesis . . . . .	7
1.5	Document Outline . . . . .	8
1.6	Summary . . . . .	9
<b>2</b>	<b>Literature Survey</b>	<b>11</b>
2.1	Introduction . . . . .	11
2.2	Question Answering Using Logic . . . . .	11
2.3	QA Using Semantic Data and Reasoning . . . . .	12
2.4	QA Based on Natural Language Processing . . . . .	15
2.5	Neural Network Techniques . . . . .	17
2.6	Ensemble-styled Inference Methods . . . . .	19
2.7	Geospatial and temporal reasoning in QA . . . . .	20
2.8	Uncertainty in QA . . . . .	21
2.8.1	Handling Uncertainty in Semantic Web and IR Systems . . . . .	21
2.8.2	Probabilistic Logic . . . . .	22
2.8.3	Approximate Query Processing . . . . .	23
2.8.4	Probabilistic Databases . . . . .	24
2.9	QA in Personal Assistants . . . . .	24
2.10	Summary . . . . .	25



<b>3</b>	<b>Background</b>	<b>27</b>
3.1	Introduction . . . . .	27
3.2	Knowledge Representation . . . . .	27
3.3	Semantic Web . . . . .	30
3.3.1	RDF and SPARQL . . . . .	30
3.3.2	Linked Data . . . . .	31
3.4	Search Algorithms . . . . .	32
3.4.1	Graphs and Trees . . . . .	32
3.4.2	Graph Search . . . . .	32
3.4.3	Heuristics and Admissibility . . . . .	34
3.5	String Matching . . . . .	34
3.6	Question Answering . . . . .	35
3.6.1	Federated Search . . . . .	37
3.7	Uncertainty: Probability and Probabilistic Databases . . . . .	38
3.8	Summary . . . . .	38
<b>4</b>	<b>The FRANK Framework</b>	<b>39</b>
4.1	Introduction . . . . .	39
4.2	Representation . . . . .	40
4.2.1	Triples . . . . .	40
4.2.2	Extending Triples to alists . . . . .	41
4.2.3	Importance of alists in FRANK . . . . .	44
4.3	Search Tree . . . . .	44
4.3.1	Definitions . . . . .	44
4.3.2	Formalising FRANK . . . . .	47
4.3.3	Dual Role of Alists . . . . .	51
4.3.4	Node States . . . . .	53
4.3.5	AND-OR Branching . . . . .	54
4.3.6	Alist and RDF Reification . . . . .	55
4.3.7	Going beyond RDF, SPARQL and SWRL . . . . .	56
4.4	Queries . . . . .	58
4.4.1	Query Grammar . . . . .	59
4.4.2	Query Types and Examples . . . . .	60
4.4.3	Roles of Variables in FRANK . . . . .	63
4.4.4	Variable Types . . . . .	65

4.4.5	Queries as Root Nodes in FRANK Tree . . . . .	65
4.5	Comparing FRANK and SPARQL Queries . . . . .	66
4.5.1	Similarities between FRANK and SPARQL Queries . . . . .	66
4.5.2	Differences Between FRANK and SPARQL Queries . . . . .	67
4.6	Decomposition Rules and Aggregate Functions . . . . .	68
4.6.1	Decomposition Rules . . . . .	68
4.6.2	Aggregate Functions . . . . .	69
4.7	The FRANK Algorithm . . . . .	69
4.7.1	Query Internalization . . . . .	69
4.7.2	Initializing the FRANK Tree . . . . .	70
4.7.3	Search and Grounding . . . . .	71
4.7.4	Expanding the FRANK Tree . . . . .	73
4.7.5	Complexity of the FRANK algorithm . . . . .	74
4.7.6	Uncertainty . . . . .	76
4.7.7	Upward Propagation of Facts and Uncertainty . . . . .	76
4.8	Summary . . . . .	77
<b>5</b>	<b>Inference Operations</b>	<b>81</b>
5.1	Introduction . . . . .	81
5.2	What is an Inference Operation? . . . . .	82
5.2.1	Definition . . . . .	82
5.2.2	Duality of Inference Operations . . . . .	82
5.2.3	Notation . . . . .	83
5.3	Alist Decomposition Inference Operations . . . . .	84
5.3.1	Relevance of Heuristics . . . . .	84
5.3.2	Heuristics for FRANK . . . . .	85
5.3.3	Decomposing an Alist . . . . .	86
5.3.4	Lookup Decompositions . . . . .	86
5.3.5	Alist Normalization . . . . .	90
5.3.6	Temporal Decomposition . . . . .	91
5.3.7	Geospatial Decompositions . . . . .	93
5.4	Execution and Propagation . . . . .	95
5.5	Aggregate Functions . . . . .	96
5.5.1	MAX . . . . .	96
5.5.2	MIN . . . . .	97

5.5.3	COUNT . . . . .	97
5.5.4	SUM . . . . .	98
5.5.5	AVERAGE . . . . .	98
5.5.6	MEDIAN . . . . .	99
5.5.7	DIFFERENCE . . . . .	99
5.5.8	PRODUCT . . . . .	100
5.5.9	GT . . . . .	100
5.5.10	LT . . . . .	101
5.5.11	EQ . . . . .	101
5.6	Function-Generation . . . . .	102
5.6.1	REGRESS . . . . .	102
5.6.2	GP (Gaussian Process Regression) . . . . .	103
5.6.3	INTEGRAL . . . . .	105
5.6.4	DERIV . . . . .	106
5.6.5	APPLY . . . . .	106
5.7	Non-arithmetic Aggregates . . . . .	106
5.7.1	VALUE . . . . .	106
5.7.2	VALUES . . . . .	107
5.7.3	COMP . . . . .	107
5.8	Inference Costs . . . . .	108
5.8.1	Cost Factors . . . . .	109
5.8.2	Heuristics . . . . .	109
5.8.3	Optimality of FRANK Heuristics . . . . .	111
5.9	Summary . . . . .	112
<b>6</b>	<b>Uncertainty in FRANK</b> . . . . .	<b>115</b>
6.1	Introduction . . . . .	115
6.2	Related Methods for Handling Uncertainty . . . . .	117
6.3	Factors Impacting Answer Accuracy . . . . .	119
6.3.1	Source Data Errors . . . . .	119
6.3.2	Query Ambiguity . . . . .	119
6.3.3	Model Misspecification . . . . .	120
6.3.4	Approximation Errors . . . . .	120
6.3.5	FRANK Error Sources . . . . .	121
6.4	What is Uncertainty . . . . .	121

6.4.1	Motivation . . . . .	121
6.4.2	The Gaussian . . . . .	123
6.4.3	Bayesian Inference: Priors, Likelihoods and Posteriors . . . . .	124
6.5	Uncertainty in FRANK . . . . .	125
6.5.1	Coefficient of Variation . . . . .	125
6.5.2	Approximating Priors from <i>cov</i> . . . . .	128
6.5.3	Assumptions . . . . .	128
6.6	KB/Fact Uncertainty . . . . .	129
6.6.1	FRANK Algorithm . . . . .	129
6.6.2	Choosing a Bayesian Method for FRANK . . . . .	130
6.6.3	Calculating and Propagating KB Uncertainty . . . . .	130
6.6.4	Worked Example . . . . .	134
6.6.5	Working with uncertainty in FRANK . . . . .	136
6.7	Inference Operation Uncertainty . . . . .	136
6.7.1	Exact Aggregate Functions . . . . .	136
6.7.2	Approximating Aggregate Functions . . . . .	137
6.7.3	Methodology . . . . .	137
6.8	Propagating Uncertainty . . . . .	138
6.8.1	Uncertainty from child nodes . . . . .	138
6.8.2	Combining uncertainty of aggregate functions and their child nodes . . . . .	139
6.8.3	Communicating Uncertainty to User . . . . .	140
6.9	Complexity of <i>cov</i> calculation . . . . .	140
6.10	Summary . . . . .	141
<b>7</b>	<b>Implementation</b>	<b>143</b>
7.1	Introduction . . . . .	143
7.2	Requirements . . . . .	144
7.3	Architecture . . . . .	145
7.4	Class Wrappers and Dynamic Loading . . . . .	149
7.5	Caching . . . . .	150
7.6	Extending FRANK . . . . .	151
7.6.1	Example of FRANK KB Extension . . . . .	153
7.6.2	Extending FRANK with inference Over Web Services . . . . .	154
7.7	Other FRANK Components . . . . .	156

7.7.1	Query Grammar with Antlr . . . . .	156
7.7.2	Parallelizing FRANK . . . . .	156
7.7.3	Visualising the FRANK Tree . . . . .	157
7.7.4	External Libraries for Aggregates . . . . .	159
7.8	Summary . . . . .	159
<b>8</b>	<b>Evaluation</b>	<b>161</b>
8.1	Evaluation Experiments . . . . .	161
8.2	Experiment 1 - Inferring missing facts (HYP-1) . . . . .	162
8.3	Experiment 2 - Prediction (HYP-2) . . . . .	165
8.4	Experiment 3 - Inferred Functions and <i>cov</i> (HYP-3) . . . . .	167
8.5	Experiment 4: Architecture and Extensibility (HYP-4) . . . . .	168
8.5.1	Experiment 4a - Impact of caching and parallelization . . . . .	168
8.5.2	Experiment 4b - Extensibility of FRANK . . . . .	170
8.6	Experiment 5: Extending FRANK to the Energy Domain (HYP-4) . . . . .	173
8.6.1	Objectives . . . . .	173
8.6.2	Implementation . . . . .	173
8.6.3	Evaluation . . . . .	175
8.6.4	Results and Discussion . . . . .	175
8.7	Discussion . . . . .	176
8.8	Summary . . . . .	178
<b>9</b>	<b>Conclusion</b>	<b>181</b>
9.1	Introduction . . . . .	181
9.2	Contributions . . . . .	181
9.2.1	Representation and Automatic Curation . . . . .	182
9.2.2	Inference Operations . . . . .	183
9.2.3	Dealing with Uncertainty . . . . .	184
9.2.4	Extensibility in FRANK . . . . .	185
9.3	Directions for Future Work . . . . .	186
9.4	Final Remarks . . . . .	188
<b>A</b>	<b>Relevant Probability Theory</b>	<b>189</b>
A.1	Independence and Conditional Independence . . . . .	189
A.2	Bayesian Statistics . . . . .	190

A.3	Gaussian Distribution . . . . .	191
A.4	Combining Variances of Gaussians . . . . .	191
<b>B</b>	<b>FRANK Query Grammar</b>	<b>193</b>
<b>C</b>	<b>Code for Human Expert Knowledge Source</b>	<b>195</b>
<b>D</b>	<b>Evaluation Queries</b>	<b>199</b>
	<b>Bibliography</b>	<b>205</b>



# List of Figures

4.1	Triple for representing facts . . . . .	41
4.2	FRANK alist example . . . . .	42
4.3	FRANK Illustration . . . . .	47
4.4	Alist Semantics . . . . .	52
4.5	Internalization: Converting a FRANK query to an inference node. . .	70
4.6	Downward Decomposition . . . . .	73
4.7	Upward Propagation . . . . .	77
5.1	Algorithm for finding properties from KBs. . . . .	89
5.2	Decomposition: Normalize . . . . .	91
5.3	Gaussian Process Regression Example . . . . .	104
6.1	Priors over KB sources and properties. . . . .	132
6.2	Combining Uncertainty . . . . .	139
7.1	FRANK user interface. . . . .	145
7.2	FRANK Packages. . . . .	147
7.3	FRANK Packages. . . . .	148
7.4	Class hierarchy diagram for KBs. . . . .	148
7.5	An example of a cached value in the FRANK cache. . . . .	152
7.6	An example of a cached function in the FRANK cache. . . . .	152
7.7	FRANK tree visualization. . . . .	158
8.1	<i>cov</i> and estimation error plot. . . . .	164
8.2	Prediction <i>cov</i> and estimation error plot . . . . .	167
8.3	Regression curves for different polynomial degrees. . . . .	168
8.4	Inferred function model errors and <i>covs</i> . . . . .	168
8.5	Impact of function caching on FRANK performance. . . . .	169
8.6	An example of HEKS in use. . . . .	170



8.7	A FRANK tree that uses both GP and REGRESS. . . . .	171
8.8	Extending and chaining multiple instance of FRANK . . . . .	172
A.1	Uncertainty propagation . . . . .	192

# List of Tables

4.1	FRANK Aggregates . . . . .	68
6.1	Worked example . . . . .	135
8.1	Evaluation results by query types, showing the percentage of queries answered successfully. . . . .	164
8.2	Queries answered correctly within specified error margins. . . . .	166
8.3	Inferred answers for the true Ghana population value of $2.6962563e7$ in 2014 for multiple years' data held out. . . . .	166



# Glossary

Below are terms used throughout this thesis and their definitions:

**FRANK** Functional Reasoning for Acquiring Novel Knowledge. Based on an algorithm that recursively decomposes alists until variables in the alist are instantiated by values in a knowledge base.

**Association list (alist)** A data structure created from a list of attribute-value pairs used to label nodes in a FRANK tree.

**Simple alist** An alist whose attributes contain only literals (terms in plain text of string, real, boolean or date-time types) or variables.

**Compound alist** An alist that has at least one attribute that contains a complex expressions such as subquery or a propositional statement.

**Alist Normalization** The process of converting a compound alist to a simple one.

**Attributes** The elements of a FRANK alist including *subject*, *object*, *property*, *time*, *cov*, etc. We sometimes use the following abbreviations

for these attribute names: *subject=subj*, *object=obj*, *property=prop*, *aggregate=ag*, *aggregate variable= agvar*.

**FRANK Tree** A tree whose nodes are labelled by alists, where each child node is derived from a decomposition of its parent node, and values of parent nodes are instantiated by an aggregation and propagation of values from child nodes.

**Downward Decomposition** The process of expanding the inference tree by a map operation that decomposes nodes using decomposition rules.

**Upward Propagation** A reduce operation that recursively propagates instantiated variables (including uncertainty variables) in alists from the leaves of the inference tree to its root.

**Decomposition Rule** A rule that determine how alists in the FRANK tree are decomposed in order to expand the frontier of the search space towards the grounding of alists in KBs.

**Aggregate Function (Aggregate)** Arithmetic or statistical function that combines specified values in the alists of child nodes during upward propagation.

**cov** Coefficient of Variation: A measure of uncertainty in FRANK calculated by normalising the standard deviation of a real-valued answer by its mean.

**RDF** Resource Description Framework: A standard model for data interchange on the Web. RDF has features that facilitate data merging even if the underlying schemas differ, and it specifically supports the evolution of schemas

over time without requiring all the data consumers to be changed (<https://www.w3.org/RDF/>).

**JSON** JavaScript Object Notation: This is a lightweight data-interchange format commonly used in web-based systems.

§ String type

ℝ Real type

ℬ Boolean type

$\tau_f$  Arbitrary FRANK alist type

$\tau$  Arbitrary type



# Chapter 1

## Introduction

### 1.1 Question Answering

Question answering (QA) has remained an interesting challenge since the very early years of modern computer systems. Researchers have always sought ways to make the computer perform interesting feats such as answering questions by searching through knowledge bases or by using inference techniques to deduce, induce or simply calculate answers from the facts available.

QA has become a lot more challenging with the rapid growth of the Internet that is creating more data than can be searched manually. This has led to new and innovative techniques for retrieving facts and answering questions of different kinds. This also means that the mechanism for representing and storing knowledge, reasoning and finding answers have also evolved significantly over recent years.

Our goal in this thesis is to address a few of the existing challenges in this field. We are, especially, interested in QA systems that are capable of finding answers even when the facts needed to answer them are not available or are incomplete. We are also interested in returning to the user not only an answer to the query, but a measure of uncertainty about the answer given the data and data sources and inference operations used to arrive at the answer. This goes beyond the widely used but arbitrary scoring and answer ranking methods that, in most cases, are difficult to interpret in terms of the correctness or accuracy of the answer value that is returned by the QA system. We explore these ideas on real-valued data attributes and on questions that return quantitative answers, not the predominantly qualitative answers that traditional QA systems return from natural language text. The scope of our work excludes the natural language processing aspects of question analysis and focuses on the automatic curation



and inference problems.

## 1.2 Motivation

### 1.2.1 Limitations of Current Question Answering Techniques

When given a question, humans possess the ability to choose from a set of possible strategies one, or a combination of strategies, that best solve the question. This ability allows us to answer questions even when the answer is not pre-stored in our memory or knowledge base. In contrast, question answering (QA) systems, although designed to use inference of varying kinds, tend to focus on the task of efficiently retrieving answer facts that are pre-stored in a knowledge base. Most perform minimal or no inference operations on real-valued facts even when these existing facts easily answer the question.

Consider the question “*What will be the UK population in 2021*”. A QA system will typically attempt (unsuccessfully) to find the pre-stored fact *population(UK, 2021, p)*. It is most likely that it will not find this, and so will give up and return no answer. Humans, on the other hand, are able to answer this kind of question, by indirectly inferring answers not already stored in the KB, from other readily available information. In the example above, we could look up the population values for past census years, and then estimate the population in 2021 using regression. We could also find the population growth rate from Wikipedia<sup>1</sup> and use that to predict the population in 2021. In so doing, we use a combination of heuristics, logic and probabilistic techniques to infer answers. We call this as *rich inference* and refer to our QA system as the Functional Reasoning for Acquiring Novel Knowledge (FRANK) (Nuamah et al. (2016)).

### 1.2.2 Beyond Fact and Answer Retrieval

Whereas NLP techniques are mostly focused on retrieving facts from free text or large text datasets, there are widely untapped non-textual dataset that hold a lot of potential for QA. These include statistical datasets mostly found in relational databases, graph databases and other structured/semi-structured forms. We see the potential not only for retrieving individual facts from these knowledge sources as answers, but for combining these individual facts in a consistent manner to answer more complex queries whose

---

<sup>1</sup><https://www.wikipedia.org>

answer may not even exist as a fact in any of the KBs.

## 1.3 The Uncertainty Problem

Although QA systems are heavily dependent on web knowledge bases, most fail to address the uncertainty that comes with such sources. Existing systems avoid the uncertainty problem and rather focus on scoring candidate answers. Not only does this create difficulty in interpreting the scoring or ranking in terms of the actual answer that a user sees, but it completely sidesteps the challenge of providing a meaningful way of informing the user about any error margins in the answer returned.

## 1.4 Hypothesis

Our claim is that:

*the types of questions that can be asked and variety of answers found by a query answering system is significantly improved when we automatically curate data using a uniform representation, use rich and extensible forms of inference to infer novel knowledge from structured data available on the Internet, and acknowledge and account for uncertainty in answers using a framework that is easy to extend to other domains and question types.*

This improvement can be achieved by (1) finding and aggregating facts from different knowledge bases, (2) devising a generic representation of data from different sources, (3) discovering and caching new facts that are not already stored in any of the original knowledge sources, and (4) creating and reasoning with functions as first class objects of the representation. The rich inference that supports this project includes rules for decomposing questions and logical, arithmetic and statistical functions for aggregating data into answers to questions, not limited by the formalisms of the source data.

We use the term “rich” to emphasize the fact that the FRANK relies on inference methods that go beyond first-order logic. We incorporate higher-order inference techniques, where reasoning about functions expands the range of answers that can be sought. For instance, we can use regression to first construct functions then apply them to make predictions. Our view of QA complements existing NLP-driven approaches by inferring non-trivial answers from readily available facts in KBs by applying such rich forms of inference.

To evaluate our hypothesis effectively, we identify and deal with the following components within it:

- HYP-1: A uniform internal representation of data facilitates the automatic curation of data from diverse sources and supports a flexible inference algorithm.
- HYP-2: Rich forms of inference lead to an extension of the types of questions possible.
- HYP-3: Acknowledging and dealing with uncertainty due to heterogeneous sources of data and inference techniques is useful for placing the answers inferred in context.
- HYP-4: An extensible algorithm and architecture allows the QA framework to be easily extended to new data sources and inference operations.

We emphasize that the contribution of FRANK is not limited to the decomposition rules and aggregate functions discussed in this thesis, and to the accuracy level to which it can predict unobserved data values. FRANK's fundamental contribution includes the overall framework that allows other inference operations to be integrated into the same QA system without the need to modify the core inference algorithm.

## 1.5 Document Outline

This thesis is organised as follows.

In *chapter 2* we provide a brief survey of relevant research work in the fields of question and query answering. We look at the very first types of query answering systems built using deductive reasoning as found in from automated theory proving techniques. We also consider QA systems that leverage the syntactic structures in natural language and those that generate database queries from the parse of the natural language question. Finally, we highlight QA systems based on ideas from neural networks and deep learning techniques.

In *chapter 3* we give the requisite background to this work by explaining some of the underlying concepts that are necessary to understand the work presented in this thesis.

In *chapter 4* we look at the algorithm and the internal representation of our QA system, the Functional Reasoning for Acquiring Novel Knowledge (FRANK). This chapter explores our hypothesis about dynamic curation by considering a generic representation of data in the inference system that is capable of representing data retrieved

from different source with diverse representations (*HYP-1*). We also describe a simple grammar for composing a variety of queries and explain the FRANK algorithm.

*Chapter 5* looks at the diverse kinds of inference that FRANK performs. The chapter also highlights the modularity of FRANK and how these different forms of inference enable our QA system to answer novel kinds of queries (e.g. prediction from statistical data) that traditional QA systems are unable to answer over structured data. We define the decomposition rules and aggregation functions that are currently implemented in FRANK. This chapter addresses the *HYP-2* aspect of our hypothesis.

Next, in *chapter 6*, we establish the need to address uncertainty in QA. We discuss the common source of uncertainty and then explain our technique for incorporating uncertainty in FRANK. Chapter 6 deals with hypothesis component *HYP-3*.

*Chapter 7* deals with our implementation of FRANK. We discuss the modular architecture of FRANK that makes it easy to extend the framework with new knowledge bases and aggregate functions without having to modify the core FRANK algorithm. We show how this gives FRANK's modules the desired plug-and-play capability by discussing two examples of such extensions to FRANK. We also consider other aspects of implementation such as caching and concurrency. This chapter explores hypothesis component *HYP-4*.

In *chapter 8*, we evaluate a prototype our FRANK system that is built using features discussed in the earlier chapters. We perform experiments to test features of FRANK that align with our hypothesis in this thesis. We test assumptions about FRANK's uncertainty calculation and check how the uncertainty measure corresponds with the actual error in the answers calculated. We discuss our results and compare with relevant QA systems.

Finally, we wrap up our work in *chapter 9* and provide some ideas for future directions for this work.

## 1.6 Summary

In this introductory chapter, we explained our motivation for tackling questions answering that goes beyond simply retrieving facts to, for instance, predicting answers that are not pre-stored in the KB. We established the hypothesis that this thesis explores. We also provided an outline of the rest of this document, describing briefly how each chapter addresses the different aspects of our hypothesis.



# Chapter 2

## Literature Survey

### 2.1 Introduction

Question answering has been of interest to researchers throughout the development of the computer and the evolution of data representation and storage methods.

This chapter discusses some existing techniques for question answering. These include logical reasoning methods, techniques that leverage search engines, the use of web knowledge bases that are semantically linked, natural language parsing methods, and statistical and neural network techniques.

### 2.2 Question Answering Using Logic

Before the rise of web-scale retrieval of information, question answering systems based on search and different kinds of logical reasoning were developed. The Baseball program (Green Jr et al. (1961)) was one of the early QA systems developed. The system used a pair list of attribute-value pairs (called a specification list) to represent natural language queries and data about baseball games. Questions were read in through punch cards, converted to a specification list and then matched against the data about baseball games. In (Cooper (1964)), Cooper developed a Fact Retrieval system (a term he invented to distinguish it from Document Retrieval, both of which were considered as Information Retrieval) that used a deductive logical inference system. Acknowledging the difficulty of working with natural language logically due to its undecidability, Cooper created a sublanguage of the English language for a small domain in chemistry and developed a ‘system of logical inference’ to make logical deductions from the stored knowledge (English sentence in his sublanguage) to answer user queries. In

a similar vein, other QA systems based on logical reasoning and techniques in automated theorem proving were explored. QA3 (Green and Raphael (1968)) considered queries as existential logical statement that had variables. The inference system used a theorem proving approach to find substitutions for the variables from the knowledge base of logical statement. DEDUCOM (DEDUctive COMmunicator) (Slagle (1965)) used deduction to find its answers. The system, built in LISP, was first “told” a set of facts about a domain (human fingers, hands and arms), and then asked questions about the information it has been given. DEDUCOM used a depth-first search procedure for deduction to find answers to questions.

## 2.3 QA Using Semantic Data and Reasoning

Significant research has gone into question-answering systems using facts available in web data sources. Information retrieval has been explored in different ways using techniques that include NLP and formal logic. Data sources from which answers are sought also range from logically represented facts, to natural language text on the Internet. The Semantic Web Berners-Lee et al. (2001) offers practical approaches to representing shared knowledge across multiple domains on the Internet. Government agencies are also responding to the initiative for open data by publishing their datasets using Semantic Web technologies. However, most information retrieval and query answering systems are still unable to effectively use these KBs to find answers that require inference beyond the retrieval of facts.

Search engines have provided a means to search large web resources to find information that matches user queries. However, the needs of some users go beyond the return of a list of search results which could possibly answer the query. These users want to get specific facts extracted from knowledge bases using queries expressed in natural language. These problems have been explored extensively in question-answering systems. We discuss a few of them below.

There are also systems that take advantage of publicly available Linking Open Data (LOD) sources to provide solutions that cover a wide range of domains, such as one that uses DBPedia (Bizer et al. (2009b)) as a knowledge base to learn semantic query suggestions (Meij et al. (2009)). Semantic search engines such as eRDF (Guéret et al. (2009)) and SearchWebDB (Wang et al. (2008)) are keyword-based semantic search tools that allow users to search through semantic datasets. Other systems such as *deepdive* (Niu et al. (2012)) deal with knowledge base construction from diverse web

sources. These tools provide resources for QA systems to use for answering queries.

Advances in Semantic Web technology have led to new kinds of data integrated directly into the web. Knowledge representations in RDF (Beckett and McBride (2004)), OWL (McGuinness et al. (2004)) and other structured formats have facilitated the formal representation of knowledge for use on the web. These sources of knowledge have offered the opportunity to explore question-answering systems which take full advantage of well-structured knowledge representations. PowerAqua (Lopez et al. (2012)) and ANGIE (Preda et al. (2010)) have created question-answering system which use RDF/OWL ontologies as background information for answer generation.

ANGIE (Active Knowledge for Interactive Exploration) takes advantage of the many open community-available RDF datasets to answer questions. It gathers data from multiple sources to enrich an RDF knowledge base. The core component of ANGIE is its query translation module. It takes as input a user's query and translates it into a sequence of functions compositions. Based on these decompositions, it then sends SPARQL queries and web calls to the RDF-3X processor. The RDF-3X processor combines triples from local knowledge base and triples from its web calls result set. User queries are answered using the enriched RDF knowledge base via web services.

PowerAqua, however, has a more extensive NLP technique for query analysis and composition of its responses. PowerAqua uses NLP to create query triples which are used to find matching triples from its local storage of ontologies. PowerAqua uses a pipeline architecture with four main components: The linguistic component, element mapping component, triple mapping component and the merging and ranking component. Queries are first processed in the linguistic component using NLP. PowerAqua uses a query triple representation (subject, predicate, object), which is similar to RDF, to formalise the relationship between the query terms. The element mapping component discovers those resources that may contain answers to the query triple. PowerAqua's data resource is made up of RDF triples from various ontologies. These are aggregated into its Semantic Storage Platform which uses plugins to different storage tools such as Virtuoso, Watson Semantic Web Gateway and Sesame. These storage platforms are used to store the datasets such that it gives PowerAqua a unified access to the knowledge resources. In order to identify the relevant resources to answer a query, PowerAqua's ontology discovery component extracts a rough set of triples which may contain the information needed. It then uses its Semantic Validation Component to disambiguate between the various interpretations of the same query term, exploiting the background knowledge provided by terms in WordNet. The Triple Similarity Ser-



vice (TSS) in the Triple Mapping Component explores the ontological relationships of the candidate entity mappings and finds the ontology triples that best match the query triples. To obtain the final response, PowerAqua merges the equivalent entities in the remaining triples and applies a ranking criteria based on confidence of the mapping algorithm, the disambiguation algorithm and the merging algorithm. Although PowerAqua uses its semantic storage platform to provide access to data resources, this approach is not practical in a dynamic data environment. For example, in domains where new data is rapidly being added, stored data will have to be updated to provide accurate answers to questions. Also, storing data locally, as a general approach, is impractical because useful new (and updated) datasets are being added to the LOD at a rapid pace.

Bundy et.al in (Bundy et al. (2013)) explored the use of the Semantic Web in solving guesstimation problems. Guesstimation does not seek find exact answer to quantitative problems, but to find approximations using a combination of intuition, facts and reasoning. They developed the GORT (Guesstimation with Ontologies and Reasoning Techniques) which uses the Single Significant Digit Calculus (SINGSIGDIG) to formalize guesstimation problems and to reason to infer new approximate answers from old ones. GORT uses a set of proof methods which reason over the SINGSIGDIG Calculus. These include the Count, Total Size, Law of Averages, Distance, Energy, Rate of Change, Aggregation over Parts, Generalization and Geometry methods. It also obtains data for reasoning from the SINDICE Semantic Web Search Engine (Tummarello et al. (2007)).

Other semantic search engines such as Swoogle (Ding et al. (2004)), eRDF and SearchWebDB are capable of processing queries to retrieve matching RDF triples. Swoogle, in particular, crawls Semantic Web documents on the web, extracts metadata and computes relationships between the documents. Swoogle also uses a ranking algorithm, Ontology Rank, inspired by the Page Rank algorithm to rank Semantic Web documents. These semantic search engines provide access to a vast amount of semantic data resources.

Ensuring that answers of the correct types are returned to the user is an important requirement for QA systems, especially those that work in open domains. In (Schlobach et al. (2004)) show how with type checking using ontologies, it is possible to ensure that the types of entities correctly match the type required by the answer. This is not easily achieved in NLP techniques without background knowledge or a linguistic resource such as WordNet that specify the types of entities or the classes they

belong to. In RDF knowledge bases such as DBpedia, resources are usually assigned types (or classes)

Other query answering systems provide a natural language interface to knowledge bases. These include question answering over linked data (Yahya et al. (2013)). Examples of this approach are seen the query-answering over linked data (QALD) challenges (Unger et al. (2014)). A variety of techniques are used in this by systems tackling this challenge include the following:

- ISOFT (Xu et al. (2014)) uses a template-based approach for transforming natural language questions into SPARQL queries that are executed by a SPARQL endpoint.
- Intui3 (Dima (2014)) uses syntactic analysis and chunking and semantic cues in the question and a target triple store to generate the SPARQL query.
- In (Zou et al. (2014)), a graph-driven approach is used to match subgraphs in the RDF knowledge base to a graph generated from the dependency parse of the natural language question.

Recently, there has been a push toward QALD challenges that tackle queries that require linked data that have a statistical nature. That is, datasets that do not just textual facts but contain numerical data. In (Höffner and Lehmann (2014)), SPARQL queries are generated to answer questions that perform mathematical function such as sum, average, max on data using the functions defined in SPARQL. Here also, the process starts by extracting the syntactic structure using a natural language parser, and then filling in place holders in SPARQL query templates to generate the queries that are sent to the RDF endpoint for execution.

## 2.4 QA Based on Natural Language Processing

Katz in (Katz et al. (2005); Katz and Katz (1997)) worked on a question-answering system, START (SynTactic Analysis using Reversible Transformations), with which a user can retrieve the information stored in the knowledge base by querying it in the English language. The author introduced the idea of Natural Language Annotations (Katz and Katz (1997)) to bridge the gap between full text natural language question-answering and sentence-level text analysis. These annotations are easier to analyse by

computers, and therefore vital to creating the representations needed for START to answer questions. START parses the natural language annotations and stores the parsed structures with pointers back to the original information segment. To answer questions, Katz et al. explained that START compares a user's query to the annotations stored in the knowledge base. If a match is found between ternary expressions derived from the annotations and those derived from the query, the segment corresponding to the annotations is returned to the user as the answer. START uses a wide set of knowledge bases to find answers to questions. These include web sources such as the CIA's The World Factbook<sup>1</sup>. In order to unify the use of these heterogeneous data sources, Katz et al. created Omnibase (Katz et al. (2001, 2002)). Omnibase provides a structured query interface to heterogeneous data on the web, by creating wrappers (or scripts) which query different databases. It translates user queries into its relational model (*object-property-value*), and tries to find matching concepts from its data sources. It is mostly concerned with finding factoids from data sources, and returning these factoids as answers.

The growth of the Internet and the wide variety of text data available from different sources led to approaches which relied less on NLP-intensive techniques. For instance, the AskMSR question answering system (Banko et al. (2002)) rather than doing NLP intensive tasks, focused on the large text resources on the web as a gigantic data repository. The authors posited that the larger the dataset from which they could draw answers, the greater the chance that they can find an answer that holds a simple easily discoverable relationship to the query string. AskMSR used the existing NLP techniques to parse text, and took advantage of the redundancy of information on the web to improve the efficacy of the QA process. It achieved this by sending queries to search engines, collecting the summaries returned, mining N-grams and composing answers from the best-matching N-grams.

Other techniques anticipate answers to questions in text, annotates these text items and indexes them such that they are easy to access when a question is sent to the QA system. This is known as *predictive annotation* and is used in (Prager et al. (2008)) together with other traditional natural language analysis of questions. Other such as (Li and Roth (2006)) classify questions semantically and use these classifications as a guide for searching for answers or verifying answers found.

Query answering, being one of the fundamental challenges of intelligent systems, led several research projects to try to adapt recent advances in the field to QA. In (Bao

---

<sup>1</sup><https://www.cia.gov/library/publications/the-world-factbook>

et al. (2014)), the QA problem is looked at from the perspective of machine translation. In this project, the emphasis was on unifying the two main steps that the authors considered as necessary for answering natural language queries: first, the translation of the natural language text to their meaning representation and then the retrieval of facts that match the meaning representation. In this project, machine translation techniques are employed (CYK parsing (Chappelier et al. (1998))) to parse each input question, and answers of the span covered by each CYK cell are considered the translations of that cell. The authors also asserted that unlike MT, which uses offline-generated translation tables to translate source phrases into target translations, a semantic parsing-based question translation method is used to translate each span into its answers on-the-fly, based on question patterns and relation expressions.

Similarly, Liang, Jordan and Klein in their dependency-based compositional semantics (DCS) formalism (Liang et al. (2013)) address QA from natural language text by representing the question's semantics as a logical form and computing the answer using data in a structured database. The goal of their approach was to learn the semantic parser from question-answer pairs, considering the logical form as a latent variable that is learned. For this to work, several lexical triggers (part-of-speech, *geo* (e.g. city, country, state, etc.), jobs triggers (e.g. salary, title, company)) are used to make the problem tractable when mapping the natural language questions to the candidate DCS trees that the parser generated. Aggregate relations specified in the DCS trees are then used to calculate answers from the data retrieved from the structured data. This technique works well for closed domains with a well structured database. The authors acknowledge that there is still a gap between their QA system, which they claim can be considered more as a natural language interface to a database, and the ultimate aim of an open-domain QA system.

## 2.5 Neural Network Techniques

Advances in neural networks and deep learning has also led to the application of these techniques in query answering.

The neural programmer (Neelakantan et al. (2015)) a neural network augmented with a small set of basic arithmetic and logic operations that can be trained end-to-end using backpropagation. Neural Programmer can call these augmented operations over several steps, thereby inducing compositional programs that are more complex than the built-in operations. The model learns from a weak supervision signal which

is the result of execution of the correct program, hence it does not require expensive annotation of the correct program itself. The decisions of what operations to call, and what data segments to apply them to are inferred by the Neural Programmer. Such decisions, during training, are done in a differentiable fashion so that the entire network can be trained jointly by gradient descent. The neural programmer uses structured (tabular) data and currently supports two types of outputs: (a) a scalar output, and (b) a list of items selected from the table. The neural programmer is the built-in operations, which have access to the outputs of the model at every time step before the current time step. This enables the model to build compositional programs.

In (Dong et al. (2015)) and (Xu et al. (2016)), the authors perform factoid retrieval by using multi-column convolutional neural networks (Krizhevsky et al. (2012)) over the Freebase (Bollacker et al. (2008)) knowledge to understand different aspects of questions including the answer path, the answer context and answer type. Question-answer pairs are used to train the neural network model to rank candidate answers by learning low-dimensional embeddings of entities and relations in Freebase. This technique avoided the need to use semantic parsers to learn to understand the natural language queries, or the use of pre-defined lexical triggers when annotated logical forms are extracted from the natural language queries.

In order to make a QA system generalizable to new domains and question types, it is important for the QA system to be compositional in its design. In (Andreas et al. (2016)) and (Andreas et al. (2015)), attentional neural module networks are composed to answer queries. Authors construct and learn neural module networks, which compose collections of jointly-trained neural modules into deep networks for question answering. Their technique first extracts the linguistic substructures (a dependency tree) in a question by parsing the question with the Stanford Parser (Klein and Manning (2003)). These structures are then used to dynamically instantiate the modular networks after filtering out the essential features of the tree. Reinforcement learning (Sutton and Barto (1998)) is employed to learn the network-assembly parameters using only (world, question, answer) triples as supervision. Neural modules include *find*, *lookup*, *relate*, *and*, *describe* and *exits*. The module at the root yields the final value of the computation.

Despite the recent advances in machine learning using neural network, there are still limitations when trying to work with very small sparse data. For instance, techniques in QA that utilize neural networks usually require large data to train and generalize from in order to answer queries. Others use pre-trained networks for the

domains of interest and generalize from that.

In QA systems that exhibit compositionality, one obvious feature that is discussed is program induction. Program induction essentially involves searching over the space of programs. This technique is used to find answers to queries by composing complex programs, capable of solving non-trivial queries, from a library of simple ones that perform simple well-defined functions. If the number of programs that are in the search space is high, an exhaustive search over all possible programs quickly becomes an intractable challenge. In (Liang et al. (2013)), authors use words from the natural language text as triggers to guide the search.

## 2.6 Ensemble-styled Inference Methods

Some QA systems use a combination of inference techniques to solve the QA problem. Notable among these is IBM's Watson.

Watson is an open-domain question-answering system built and optimized for the United States' television quiz show Jeopardy! In February 2011, Watson competed against two previous overall human winners and won the game. In preparation for the contest, Watson's performance was compared to previous Jeopardy! winners, using the set of questions used in the previous competitions. Watson performed better than all of them. The objective of the game is to correctly answer a series of open-domain questions using clues. Contestants provide answers in an average of three seconds. In order to achieve this feat, IBM Research developed DeepQA (Ferrucci et al. (2010)), an extensive software architecture for building its question-answering system. DeepQA is essentially a processing pipeline architecture that applies several algorithms that analyse evidence along different dimensions such as classification, time, geography, popularity, passage support, source reliability and semantic relatedness. The key processes in this pipeline were Question and Topic Analysis, Question Decomposition, Hypothesis Generation, Hypothesis and Evidence Scoring, Synthesis, Final Confidence Merging and Ranking, and finally the Answer and Confidence.

Watson's DeepQA architecture, as with any question-answering system, requires knowledge bases from which to find and synthesize its answers. The challenge in open-domain question-answering is that the vast amount of information in the open domain is unstructured and noisy. Acquiring this data in a structured formal representation is a computationally expensive task. In (Ferrucci et al. (2010)), authors showed that shallow syntactic knowledge and its implied semantics can easily be acquired and

used in their DeepQA architecture. To this extent, IBM Research primarily used structured data, knowledge bases and ontologies to support its evidence analysis in DeepQA (Ferrucci et al. (2010)). Some were automatically extracted from unstructured text documents and others were off-the-shelf knowledge resources such as, Freebase (Bollacker et al. (2008)), YAGO (Suchanek et al. (2007)) and WordNet (Miller (1995)). In the game show Jeopardy!, since Watson had to function independently of the Internet (and also to optimize the speed at which answers were generated), all the knowledge bases were extracted and stored offline. Watson extensively uses Natural Language Processing (NLP) techniques for question analysis and answer synthesis.

Another such system that adopts an ensemble-style approach to QA is Wolfram|Alpha, a computational knowledge engine for computing answers and providing knowledge. It works by using its vast store of expert-level knowledge and algorithms to automatically answer questions, do analysis, and generate reports. Wolfram|Alpha<sup>2</sup> uses built-in knowledge curated by human experts to compute on the fly a specific answer and analysis for every query. Although Wolfram|Alpha has question-answering capabilities, it is limited to using facts from carefully curated knowledge by humans.

Additionally, Chu-Carroll et al. (2003) uses a multi-strategy and multi-source approach to question answering. It is based on the process combining the results from different answering agents searching for answers in multiple corpora. Others such as (Tunstall-Pedoe (2010)) complement the core knowledge base with facts from other sources for factoid retrieval in open domain QA.

## 2.7 Geospatial and temporal reasoning in QA

In (Hitzler and van Harmelen (2010)), Hitzler and van Harmelen explain that geospatial reasoning and other forms of geometrical computation or approximate reasoning are not well supported by the Semantic Web. So despite all the data that exists about locations and places, reasoning over these in a consistent way is difficult without the use of storage-engine (triple store) specific implementations of geospatial reasoning.

The implementation of the Regional Connection Calculus (RCC-8) (Randell et al. (1992)) standard in Semantic Web engines makes it possible to reason over spatial data for which the right kind of data satisfying the RCC-8 standard has been stored.

For instance, PelletSpatial, a hybrid RCC-8 and RDF/OWL reasoning and query engine built on top of Pellet (Parsia and Sirin (2004)) allows for querying over con-

---

<sup>2</sup><https://www.wolframalpha.com>

sistent sets of spatial relations and non-spatial semantic RDF relations. Although this works well in Pellet (at least for the prototype that is evaluated), the same cannot be said about many of the triple stores used for knowledge bases, given that the RCC-8 standard is not a core part of the SPARQL standard. However there have been recent efforts in the Semantic Web community, led by the OCG (Open Geospatial Consortium) to make GeoSparql (Battle and Kolas (2011)) the de-facto standard for geospatial reasoning in RDF triple stores. RDF databases such as Virtuoso (Erling and Mikhailov (2009)), Apache Jena (The Apache Software Foundation (2017)) and GraphDB<sup>3</sup> have implemented a large proportion of this standard in their respective systems. However, SPARQL queries that use GeoSparql functions cannot be ported from one database to another since the GeoSparql functions having different names are placed in different namespaces on these systems. Other triple stores such as Blazegraph<sup>4</sup> that have geospatial capabilities implement their own proprietary representation of geospatial information and functions for inference. This diversity of representation and reasoning over geospatial data compounds the problem of using the same SPARQL query across multiple knowledge bases.

Allen's temporal algebra (Allen (1983)) is commonly used in reasoning temporally in QA systems. For example, (UzZaman et al. (2012)) presents a temporal QA system that performs temporal reasoning about any document annotated in TimeML (Pustejovsky et al. (2003)).

## 2.8 Uncertainty in QA

### 2.8.1 Handling Uncertainty in Semantic Web and IR Systems

Although knowledge bases on the web are commonly use for open-domain query answering, there is a cost that comes with it: noise due to errors, ambiguities, and missing data. However, several QA systems that work with these web data sources often proceed on the assumption that these data sources are equally accurate and focus their efforts on retrieval and ranking of answers. However, work in Defacto (Lehmann et al. (2012)) and in (Wienand and Paulheim (2014)) highlight the fact that errors do exist in these knowledge bases and failure to acknowledge them could lead to inaccurate or completely erroneous answers. The need to clean datasets on the linked open data

---

<sup>3</sup><https://ontotext.com/products/graphdb/>

<sup>4</sup><https://www.blazegraph.com/>



cloud in projects such as the LOD Laundromat (Beek et al. (2014)) emphasizes the need to properly deal with erroneous data on the web when using these data sources in automated systems.

In (Ko et al. (2010)), evidence from other text sources (gazetters, search engines) are used to rank and merge answers. This project also identified a limitation of techniques that only translate user queries directly into a database query for retrieval. The limitation was that such techniques did not scale well to open/multi-domain and multi-knowledge base QA. Given that for most questions, the exact answers are not retrieval from the database, there is the need to combine data from multiple sources. In their work, the authors used multiple answering agents to extract candidate answers. They developed a unified probabilistic framework to combine multiple evidence to address challenges in ranking and answer merging for factoid questions and questions that return lists.

### 2.8.2 Probabilistic Logic

Probabilistic logics (Nilsson (1986)) extend classical logics with the ability to reason about uncertain data. In classical logic, a sentence can be either *true* or *false* in each possible world (model or assignments to variables in the sentence). Probabilistic logic provides a semantic generalization of logic in which the truth values of sentences are probability values between 0 and 1.

Probabilistic logic focuses on such uncertain reasoning by a process of probabilistic entailment that formally calculates bounds on the probability of a sentence given a base set of sentences that constitutes its belief about the possible worlds. Also, given a set of sentences and their associated probabilities, probabilistic logic can be used to make new assertions about the world with some posterior probability assigned to these assertions.

Although probabilistic logic attaches uncertainty to logical formulae, it is not applicable in our work where we are interested in attaching uncertainty to the values and assignments in a formula.

In a related work (Fritz and McIlraith (2009)) that focuses on plan robustness in continuous domains, probability densities are treated meta-linguistically. In this work, the robustness of a sequential plan is determined by its probability to execute successfully despite uncertainty in the execution environment in continuous domains. Such uncertainty includes the stochastic effects from actions and unpredictable changes to

state variables. This work uses probability distributions, such as the Gaussian, to reason about uncertainty. The core planner is based on the situation calculus (Levesque et al. (1998)), but the probabilistic reasoning tasks are performed outside the situation calculus. Although the context and usage of uncertainty in this planning work is different from our definition in FRANK, we see the meta-linguistic treatment of probability as relevant. This related work also points to the fact that one may not need the full power of the probability calculus to deal with uncertainty.

### 2.8.3 Approximate Query Processing

Very large databases are also commonplace now, given the amount of data generated from social media channels, customer data, sensor networks and IoT (Internet of Things) systems, etc. In some cases, there are limits (imposed by factors such as time, device memory capacity, network bandwidth, etc.) on how much data a client can fetch from a databases in order to answer a query. In such situations, the client may resort to obtaining approximate answers to queries. Work in approximate query processing (AQP) has focused on such constraints in relational database systems. Rather than exhaustively searching the database to compute answer to queries, AQP systems find approximate answers to queries using a subset of the dataset at a fraction of the cost of computing the answer the traditional way.

Chaudhuri et.al. in (Chaudhuri et al. (2017)) surveyed a number of techniques used in AQP over the past few years. These include dynamic sampling (Babcock et al. (2003)) and sampling with error estimation (Agarwal et al. (2014)) where only a subset of the relations in the database are sampled and used, and more recently, resource-bounded approximation (Cao and Fan (2017)) where a bound is placed on the fraction of the data that can be accessed throughout the entire process of answering the query.

Database factorization (Olteanu and Schleich (2016)) leverages the well-defined structure of tuples in databases, such as relations in normal forms, to decompose queries. This avoids redundancy in the representation of query results and also speeds up query execution. The factorization converts flat joins to a factorized nested join structure resulting in faster execution of the query. Olteanu et.al. note that in factorization, cartesian products capture the independence in the data, unions capture alternative values for an attribute, and references capture caching. Further, techniques such as in-database factorized learning (Ngo et al. (2017)) can be used to incorporate learning from the data to answer queries using regression. This technique moves analytical

computation into the database engine to avoid time wasted on data import/export at the interface between database systems and statistical packages.

#### 2.8.4 Probabilistic Databases

Probabilistic databases, such as Trio (Widom (2004)), seeks to extend the ideas of traditional databases to include data that are inexact in nature with a data values that have been assigned probabilities. Trio is also concerned with information on how the data items stored or answers to queries items came into existence (i.e. lineage or provenance). The Trio data model has three components: data, accuracy and lineage. Data is based on the standard relational model for databases. Accuracy deals with the uncertainty that exists in the data. This includes data values missing from records, probability values assigned to data or probability distributions over the possible data values in a record (such as the use of a Gaussian distribution in GADT (Gaussian Abstract data type) (Faradjian et al. (2002)) for measurements in the physical world). Using this data model, Trio can find answers to queries within some specific confidence bounds or return the confidence values of the data it returns as answers to queries.

Similarly, in (Suciu et al. (2011)) and (Dalvi and Suciu (2007)), work is focused on creating a probabilistic DB that evaluates queries efficiently. In such probabilistic DBs, each tuple is considered to have a probability of belonging to the DB. These systems are aimed at combining structurally rich SQL with uncertain predicates for which uncertain matches can be found, and the ranking of such uncertain results.

### 2.9 QA in Personal Assistants

Recent work in search engines using web data have gone beyond natural language text to use speech recognition tools to translate human speech to text. This gives a more natural user experience on mobile systems and other personal assistant devices. This is a natural extension of search for many companies which are already in the web search and IR domains. Google developed *Google Now* (Google (2012)) to respond intelligently to user questions using both text and speech analysis. Similarly, Microsoft has created *Cortana* (Microsoft (2014)), Apple has created *Siri* (Apple Inc. (2011)) and Amazon has *Alexa* (Amazon (2016)). These are among just a few of the systems in the personal digital assistant application domain which apply voice-based natural language question-answering on personal devices. These systems, are however, limited

in functionality.

## **2.10 Summary**

In this chapter, we reviewed some of the question answering techniques that have been developed. We observed that these techniques are driven not just by the kinds of questions that users want to answer, but by the advances in knowledge representation, inference methods and natural language parsing techniques. Our work in subsequent chapters builds on some of these ideas.



# Chapter 3

## Background

### 3.1 Introduction

In this chapter, we provide a brief introduction to background concepts that are relevant to our work. These include knowledge representation, the Semantic Web, search algorithms, question answering and probability theory.

### 3.2 Knowledge Representation

Knowledge is made up of facts, rules and propositions about the world. In order to perform automated forms of reasoning, knowledge must be represented in a way that allows computation and inference to be possible. Knowledge representation uses formal expressions to represent such facts, rules and propositions to create knowledge bases that can be used to reason about the world. Knowledge can be represented in numerous forms including databases, graphs, frames, networks and formal logics such as propositional logic (PL), description logics (DL), first-order logic (FOL) and higher-order logic (HOL).

Early knowledge representation systems such as KL-ONE (Brachman and Schmolze (1988)) and other frame-based systems provided a mechanism for conceptualising information about *things*. They do this by providing a *language for expressing an explicit set of beliefs for a rational agent* about the world and from which reasoning can be done.

Propositional logics are concerned with representing knowledge as propositions which are either true or false. Rules of inference and axioms are used to derive new propositions. First-order logic is made up of terms, propositions and formula. Terms

describe objects in the domain. A term is either a constant, a variable or an n-ary function applied to n terms. A proposition in FOL has a truth value of either ‘true’ or ‘false’, which describe facts about the domain. Propositions are n-ary predicates applied to n terms. For example  $Costs(car, \$10000, 2010)$  is a ternary relation indicating that ‘a car costs \$10000 in the year 2010’. A formulae in FOL is a proposition, or combines propositions using logical connectives or quantifiers. Logical connectives are conjunction, disjunction, negation and implication. A quantifier can be either a universal quantifier or an existential quantifier. Higher-order logics allow variables over functions (classes), predicates, functions of functions, etc. So, whereas FOL quantifies only variables that range over individuals (instances of classes), second order logic additionally quantifies over functions. Third order logic further quantifies over functions of functions, etc.

Description logics are decidable sub-logics of first-order logic and therefore have formal semantics that specifies the exact meaning of the represented knowledge. A DL knowledge base consists of two two components: a TBox and an ABox (McGuinness et al. (2004)). The terminology that generally describes concepts and their properties are defined in the TBox. This is referred to as intensional knowledge. The ABox is made up of assertions that are specific to individuals in the domain of interest. This is referred to as extensional knowledge. The formal semantics make it easy for humans to create and share representation of domains with minimal ambiguity. Since the semantics are formal, computations can be performed automatically over the representation to make inferences from the axioms that are explicitly stated. These features give DL an advantage over other knowledge representations which do not have formal semantics with strong mathematical grounding. It is also worth mentioning that there are several fragments of Description logics for different purposes with different complexities.

Logical reasoning is typically by deductive inference, which involves determining the entailments of a knowledge base given any sentence. Two important properties of this form of reasoning are soundness and completeness. Reasoning is logically sound if any sentence that is provable from the knowledge base also satisfies all interpretations (assignments of meaning to symbols of the logical expressions) of the knowledge base. Reasoning is logically complete if every sentence that is entailed in the knowledge base can be proved. Unlike description logics, which always terminate during reasoning, First-order logic is not guaranteed to terminate. The reasoning in DL is sound and complete and these properties can be achieved by restricting expressiveness, allowing

only allow unary and binary predicates.

Modelling with description logics, propositional logic, first-order logic, higher order logic or other forms of representations is usually a trade-off between expressivity of the language used and the complexity of the reasoning that can be performed on the representation. As a result, several variants of description logics have been developed to fit specific usage needs. One such usage of description logics is the modelling of domains for the Semantic Web.

The Web Ontology Language (OWL), which comes in many variants depending on the DL used, is one of the common knowledge representations used for expressing facts about the world on the web. OWL plays a significant role in the linked data domain for communicating and sharing ontologies. It uses several lightweight variants of Description Logics to ensure that reasoning is tractable. One feature unique to OWL with regards to other Description Logics is how it handles incomplete information. Every axiom that is stated in an ontology captures some information about the domain being modelled. However, not all axioms are stated explicitly in the ontology, leading to incompleteness in the representation. In Description Logics, rather than making assumptions about the particular interpretation of an ontology, the semantics considers all possible states of the world where the axioms in the ontology are satisfied. This ensures the Open World Assumption where an ontology is essentially an axiom that holds in all interpretations that satisfy the ontology (Krötzsch et al. (2012)). This means that as more axioms are added to an ontology, the number of possible interpretations of the ontology is reduced, since interpretation is constrained by the axioms stated. This results in the monotonic property of semantics of DL as well as PL, FOL and HOL, that is, additional axioms in an ontology always leads to additional consequences. This is in contrast with the Closed World Assumption where the interpretation is closed by assuming that every fact not explicitly stated to be true is actually false. This can lead to non-monotonicity or inconsistency if a previously absent fact is added. The standardization of OWL by W3C, to facilitate the description and linking objects on the Semantic Web, has encouraged the publishing of ontologies on the web. Many organizations that publish large volumes of information have begun making extra efforts to build more formal representations of the information. Organizations such as the BBC (<http://www.bbc.com>), UK Government (<http://data.gov.uk>), Wikipedia, and several others have made tremendous efforts to offer their data in open formats. These formats include OWL, Resource Description Framework (RDF) and Comma-Separated Values (CSV). This is as a result of the advancement of techniques and availability of tools to



help in the creation of ontologies from their data. Some of the datasets created such as that from data.gov.uk are manually curated with the help of ontology creation tools. There are other projects which have automatically created ontologies from text files, database and web pages KnowItAll (Etzioni et al. (2004)), YAGO (Yet Another Great Ontology) (Suchanek et al. (2007)) and NELL (Carlson et al. (2010)) are example of projects which automatically extracted ontologies from text data.

## 3.3 Semantic Web

### 3.3.1 RDF and SPARQL

The web has grown enormously and become a repository of information covering several areas of knowledge. It has liberalized the publishing of content such that anyone, anywhere, connected to the Internet can make content available for other users to consume. However the style and representation of the published content are primarily aimed at humans, not machines or computer agents. Agents capable of using the traditional website are search engines which use the text and HTML (Hypertext Markup Language) content of the pages as well as the links between pages. The lack of a formal, structured representation of the data and its meaning in the web pages makes it difficult to harness the true potential of the web. This means that although there are vast amounts of information available on the web, it is barely usable by machines for learning and reasoning tasks. This led Berners-Lee et al in (Berners-Lee et al. (1998)) to propose the Semantic Web. The aim of the Semantic Web is to allow much more advanced knowledge management systems (Antoniou and Van Harmelen (2004)). Knowledge is organized in conceptual spaces according to meaning, automated tools support inconsistency checking and knowledge extraction is automated. Also, keyword-based search is replaced by query answering and query answering over several web documents is supported.

Semantic web technology aims at using metadata in web pages to capture data about the information in the web document, and in essence give meaning to the document. Antoniou and Van Harmelen in (Antoniou and Van Harmelen (2004)) also claimed that ontologies, within the context of the web, will provide a shared understanding of the world. The Semantic Web has at its core knowledge representation in RDF (Resource Description Framework). Formally, RDF is a binary predicate of the form  $p(a,b)$ , where  $a$  and  $b$  are related by the predicate  $p$ . This goes beyond the rep-

representations in HTML and XML (Extensible Markup Language). RDF's fundamental concepts are resources, properties and statements. A resource is a "thing" that can be talked about. These include people, places, objects and predicates among several others. They are assigned Universal Resource Identifiers (URI) which are unique across the web. Properties (predicates) describe relationships between subject resources and object resources. Statements are assertions defined by triples made up of a subject resource, a property and an object resource. In order to retrieve facts stored in the RDF representations, query languages such as SPARQL (SPARQL Protocol and RDF Query Language) (Prud'hommeaux et al. (2008)) are used to query the knowledge base(s).

### 3.3.2 Linked Data

Linked Data, like the Semantic Web, focuses making the Web more usable by machines. However, it addresses that challenge differently. Linked Data, in its simplest definition, is using the Web to create typed links between data from different sources (Bizer et al. (2009a)). The objective is to have data on the web published in such a way that it is machine-readable and connected to other relevant data items elsewhere on the web. This linking of data attempts to get better meaning from the information that is published. At the core of linked data is the use of URIs and RDF. Rather than linking the published documents directly using traditional URLs, Linked Data uses RDF in the document's metadata to make statements about objects in the document and how they are linked to other objects in other web documents. The use of URIs makes it possible for these different objects to have unique identities such that the likelihood of ambiguity is diminished. The Linking Open Data (LOD) Project<sup>1</sup> is one of the common applications of Linked Data. It has seen several organizations which publish a lot of web content using Linked Data principles to create links between data in their web documents. This linking is not only between data in documents, but also with other ontologies which define schemas, vocabularies and hierarchies about concepts. This interlinking allows a web document to be laced with meaningful information beyond the text that humans can read off the page. It makes web documents friendlier to machines which can view the web of data as a large dataset for computation. The Linking Open Data cloud has been growing rapidly for the past few years. As at 2011, the LOD cloud had about 295 datasets. Currently (as of August 2017), it contains over

---

<sup>1</sup><https://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData>

1014 datasets . Linked Data resources such as DBpedia, GeoNames Ontology and The Music Ontology (Raimond et al. (2007)) have been used in question answering systems such as IBM Watson and PowerAqua (Lopez et al. (2012)).

## 3.4 Search Algorithms

We discuss graphs and trees in this section paraphrasing definitions by Sedgewick and Wayne in (Sedgewick and Wayne (2011)).

### 3.4.1 Graphs and Trees

A *graph* is defined as a set of *vertices (nodes)* and a collection of *edges* between them. A *path* in a graph is a sequence of vertices connected by edges, with a simple path being one where no vertices are repeated. When a graph has at least one path whose first and last vertices are the same, the graph is said to be *cyclic*. An *acyclic* graph is, as its name suggests, one with no cycles. A graph is connected if there is a path from every vertex to every other vertex in the graph. A directed graph is one whose edges have a direction from one vertex to another. An undirected graph doesn't have any notion of direction of the edges.

A tree is an acyclic connected graph. Formally, a tree is a graph  $G$  with vertices  $V$  that satisfy the following properties:

- $G$  has  $V - 1$  edges and no cycles.
- $G$  has  $V - 1$  edges and is connected.
- $G$  is connected, but removing any edge disconnects it.
- $G$  is acyclic, but adding an edge creates a cycle.
- Exactly one simple path connects each pair of vertices in  $G$ .

### 3.4.2 Graph Search

Graphs are data structures for storing data, and strategies (algorithms) exist to search for items stored in the graph. Search strategies are evaluated using different criteria to determine their performance. These criteria include completeness, optimality, time complexity and space complexity, described as follows in (Russell and Norvig (2010)):

- completeness: the guarantee that the algorithm will find a solution if one exists;
- optimality: that the strategy used in the algorithm is the optimal one;
- time complexity: how long it takes to find a solution; and
- space complexity: how much memory is needed to perform the search.

The branching factor (number of successors or child nodes from a given vertex), the depth (length of path from the root node) and the maximum length of any path in the search space are often used to determine the complexity of a search strategy.

Search strategies are categorized into two groups: (1) uninformed search and (2) informed (heuristic) search. Uninformed search strategies explore the search space blindly using no additional information beyond what is observed in the graph and in the definition of the search problem. Informed search strategies, on the other hand, use heuristics to guide the exploration of the graph.

Commonly used uninformed search strategies include the following:

- Breadth-first search: Beginning from the root node, all nodes at a given depth in the graph are explored before any nodes at the next level are explored. This strategy is complete when the branching factor is finite.
- Depth-first search: For a given node, this strategy explores, for each child node, nodes to the deepest level before backing up to the next deepest node that has not yet been explored. This strategy is incomplete when there is an infinite path in the graph.
- Depth-limited search: This is a depth-first search strategy but with a predetermined depth limit such that nodes at this depth limit are considered as having no child nodes. This strategy is meant to solve the incompleteness problem that occurs due to an infinite path in the graph. However, the strategy is incomplete if the search item occurs below the depth limit.
- Iterative deepening depth-first search: This strategy iteratively runs the depth-limited search, gradually increasing the depth limit in each iteration. It combines the benefits of depth-first search and breadth-first search and is useful when the search space is large and depth of the item being searched is not known.

Commonly used informed search strategies include:

- Best-first search: The node in the graph to be explored next is based on a cost function, such that the node with the lowest cost is explored before a node with a higher cost. In some cases, the cost includes a heuristic cost component (the estimated cost from a node to the goal node) that is obtained from prior knowledge about the search space
- Greedy best-first search: A best-first search strategy whose cost function consists only of the heuristic function.
- A\* search: A best-first search strategy whose cost function consists of the cost from the root node to reach the current node being explored from the root, and the cost to get to the goal node from the current node.

### 3.4.3 Heuristics and Admissibility

Informed search strategies such as A\* have two conditions for optimality of their heuristic cost component: (1) admissibility and (2) consistency.

As defined in (Russell and Norvig (2010)), an admissible heuristic is one that never overestimates the cost to reach the goal node.

Consistency, is defined as the monotonicity requirement of the heuristic. That is, a heuristic  $h(n)$  is consistent if for every node  $n$  and every successor  $n'$  generated by some operation  $a$ , the estimated cost of reaching the goal node from  $n'$  is no greater than the step cost of getting to  $n'$  plus the estimate cost of reaching the goal from  $n'$ . That is,  $h(n) \leq c(n, a, n') + h(n')$ .

## 3.5 String Matching

String matching is a basic technique for determining how syntactically similar two strings are. It is commonly used for retrieving data from KBs by matching search terms to terms in the KB. A commonly used metric is the *edit distance*. The edit distance between two strings  $A$  and  $B$  is the minimum number of edit operations (insertion, deletion and substitution) required to transform  $A$  to  $B$ . This is also called the Levenshtein distance (Levenshtein (1966)).

Another string similarity metric is the Jaro (and Jaro-Winkler) similarity (Jaro (1989, 1995)). The Jaro metric considers the length of the strings being matched, the number of matching characters (not counting transpositions) and the number of trans-

positions (number of matching characters but with different sequence order) (usually divided by 2). The Jaro-Winkler similarity (Winkler (1999)) extends the Jaro metric by considering, in addition, the length of any common prefix at the start of the strings up to a maximum of four characters so as give better scores to strings that match from the beginning.

Term document inverse document frequency (TD-IDF) (Salton and Buckley (1988)) is also used in information retrieval with the primary benefit that it weights rare terms higher than commonly occurring terms by considering the frequency of terms not just in a single record, but across all records in all the available KBs.

String matching algorithms have been compared in work related to information retrieval and record linkage. These include empirical comparisons of algorithms such as Jaro-Winkler similarity, Levenshtein distance and TD-IDF in (Cohen et al. (2003)), algorithms for approximate string matching (Ukkonen (1985)). Work in string matching in information integration (Bilenko et al. (2003)) have also evaluated various use cases and performance of several string matching algorithms on different datasets. Our choice of a string matching algorithm in our work is influenced by comparisons these and we discuss this in section 5.3.4.

## 3.6 Question Answering

With our ability to store data comes the need to retrieve the data at will. Given that in very large knowledge bases there is usually a lot more data than can be searched sequentially, there needs to be clever ways of finding the right information fast. Structured databases have query languages for defining the appropriate search criteria in order to retrieve the stored data from the database. Many relational databases often use the SQL (Structured Query Language) whereas graph databases use graph query languages such as SPARQL, Cypher<sup>2</sup> and Gremlin<sup>3</sup> (Holzschuher and Peinl (2013)).

Beyond structured data, there is also the need to find data from natural language text and documents, particularly from the web. Information retrieval primarily deals with such problems. In (Christopher et al. (2008)), information retrieval is defined as finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers). Structured data is data represented formally with a schema in the tables with rows and

---

<sup>2</sup><https://neo4j.com/developer/cypher-query-language/>

<sup>3</sup><https://github.com/tinkerpop/gremlin/wiki>

columns, or in a graph such as subject-predicate-object triples. Unstructured data is usually without a schema or a uniform pre-defined structure. For example plain text from a text document or a web page. However, even for these unstructured data, some structure can be seen in its organizations (for example the heading, chapters, sections, etc.). In web pages, the Hypertext Markup Language (HTML) used for representing the web page could offer some structure to the content of the page (title, body, footer, etc.). These are usually referred to as semi-structured data. Unlike database search that deals with retrieval from structured data, information retrieval is concerned with retrieval from unstructured or semi-structured data.

Information retrieval uses algorithms and techniques in information theory, probability and machine learning to find the documents (or some content within the document) that best matches the user's query. This is the predominant feature of search engines that try to return the best matching web documents for a given user's query.

However, in some use cases, returning a list of ranked documents for a user's query is not sufficient for users. Users sometimes prefer to have a specific answer to their query rather than a list of web links from a search engine that could answer their query. This has driven the desire to have question-answering systems that go beyond retrieving web documents to retrieving facts that answer the user's query.

Question answering is usually considered in two forms: (1) closed-domain QA and (2) open-domain QA. Closed-domain QA deals with finding answers to queries within a restricted domain of interest. Questions relate to this restricted domain and knowledge base(s) used to find answers are also limited to the specified domain. This form of QA simplifies the QA task somewhat since the QA system knows a priori what kinds of data it needs to answer the queries it will receive. Open domain QA works in an unrestricted space of domains, queries and data. That is, there are several possible worlds, with potentially diverse representations, to consider during the answering of a query. As a result, before finding the best matching data to answer the query, an open QA system needs to identify the appropriate domain(s) knowledge bases from which to find the data that answer the query. Since the domain is unrestricted, the QA system usually accesses different KBs depending on the query being answered. In some cases, for a single query, multiple knowledge bases are accessed. Federated queries (section 3.6.1) are useful in cases where the knowledge bases being accessed have the same representation and query language.

### 3.6.1 Federated Search

Searching for information in a knowledge base usually works by executing a search query on the specific instance of the database that contains the required data. However, with the advances in computing technology and cheaper data storage costs, more and more data is available to store and use in search. As a result, the data needed to answer a search query could be distributed across multiple databases (either locally or remotely). For this reason, the search query needs to integrate information from more than one data collection instance in order to successfully return the required results. This is the underlying motivation for the federated search system.

The primary tasks in a federated search system are (1) representing multiple independent collections of data, (2) the search of a group of independent data collections, and (3) the effective merging of the results they return for queries (Shokouhi et al. (2011); Nguyen et al. (2012)). A user makes a single query request which is distributed to the search engines, databases or other query engines participating in the federation. The federated search then aggregates the results that are received from the search engines for presentation to the user. This is often a technique to integrate disparate information resources on the web.

Some modern database systems support federated search across multiple database instances. In particular, the World Wide Web Consortium (W3C)'s<sup>4</sup> RDF specification for federated queries documents the syntax and semantics of the SPARQL 1.1 Federated Query extension for executing queries distributed over different SPARQL endpoints using the *SERVICE*<sup>5</sup> keyword. This allows for combining graph patterns in a single query that can be evaluated over multiple RDF endpoints (Buil-Aranda et al. (2013)).

Federated search queries across databases of the same type is relatively easy since the problem of representation is relatively easy to deal with across the database instances. Comparatively, when the databases have different data representation formalisms and different query languages, federated querying becomes a harder problem requiring additional work to coordinate the integration of data from these diverse systems as discussed in (Shokouhi et al. (2011)) and (Nguyen et al. (2012)).

Finally, a problem that comes with distributed data is that of availability, particularly when the database is managed independently and outside the control of the user. This is a common problem in the linked data community where many SPARQL end-

---

<sup>4</sup>[www.w3.org](http://www.w3.org)

<sup>5</sup><https://www.w3.org/TR/sparql11-federated-query/>



points are hosted by organisations and individuals and a lot of compute resource is spent to execute queries on server. Architectures such as linked data fragments (Verborgh et al. (2014)) provide a data publishing paradigm that allows efficient offloading of query execution from servers to clients through a lightweight partitioning strategy. It enables servers to maintain availability rates as high as any regular server, allowing querying to scale reliably to much larger numbers of clients.

### 3.7 Uncertainty: Probability and Probabilistic Databases

As discussed in section 2.8.4, probabilistic DBs incorporate the concept of uncertainty into traditional relational database. For instance, in (Dalvi and Suciu (2007)), this is done by allowing uncertain predicates using semantics based on probability theory. Probabilistic reasoning deals with reasoning about uncertainty over possible worlds (also called the *sample space*).

Within the context of noisy web data and inference mechanisms that return approximate answers, other forms of uncertainty arise. As well as attribute and tuple uncertainty in the literature above, we also consider knowledge source uncertainty and inference operation uncertainty. In particular, having prior knowledge about the accuracy of facts in a KB should influence the overall uncertainty of answer when the answer is derived from multiple knowledge sources. Bayesian reasoning enables the incorporation the concept of ‘prior beliefs’ in estimating the posterior probability of an even after some observations are made. These techniques are key to our calculation of answer uncertainty in our work.

Relevant concepts and formulas of probability theory can be found in appendix??.

### 3.8 Summary

In this background chapter, we briefly introduced some of key concepts that are used in this thesis. These include knowledge representation, RDF and SPARQL, Linked Data, graphs and search algorithms, query answering, probability theory and the Gaussian distribution.

# Chapter 4

## The FRANK Framework

### 4.1 Introduction

The focus of this project is the inference mechanism that allows us to build on existing techniques for query answering, particularly over web data. This chapter delves into the Functional Reasoning for Acquiring Novel Knowledge (FRANK) framework, our inference mechanism, and discusses how it works.

We explore the first aspect of our hypothesis:

**HYP-1:** *A uniform internal representation of data facilitates the automatic curation of data from diverse sources and supports a flexible inference algorithm.*

The objective of FRANK is to provide a system for predicting answers to queries when the required data that answers the query is not stored in the available knowledge bases. FRANK uses a graph-based algorithm that recursively decomposes queries, eventually grounding out in either stored facts or previously cached answers. A core requirement to achieve this objective is the internal representation of the inference system. We discuss how we extend existing RDF triples representation to accommodate the requirements of FRANK. Another key component is how input queries are expressed formally, since FRANK does not (as yet) take natural language queries. We discuss the formal grammar used to define the various query types, examples of these queries and how they are processed and the process of internalizing the queries to initiate the construction of the inference tree (FRANK tree).

An important contribution of this project focuses on the compositional application of aggregate functions (aggregates) in FRANK as well as the decomposition rules that determine the relevance of aggregates at different stages in the inference process.

These decomposition rules, when applied, result in the decomposition of nodes in such a way that they grow the FRANK tree when lookups in knowledge bases fail to instantiate the unknowns (variables) in nodes. The rules guide inference at each level of the inference tree, and also provide the means for combining nodes during the upward propagation of facts, when they are found in KBs. The decomposition rules (governed by heuristics) and aggregate functions discussed in this chapter are sufficient to understand how they fit into the overall inference algorithm. An in-depth look at them, however, is found in Chapter 5.

We also explain how the inference algorithm works by recursively spawning new nodes from leaf nodes and how it up-propagates through the nodes to the root of the tree to give an answer to the user query. In summary, we are interested in how we can use higher-order inference in query answering and we show how we can achieve this using data from web KBs.

## 4.2 Representation

In this section, we describe how we extend the RDF triple representation into a generic key-value pair data structure that is more flexible for use in FRANK.

### 4.2.1 Triples

Web knowledge bases have traditionally represented data as triples. For instance in relational databases, the value in each cell of a relation is the intersection of a record identifier and a column identifier. Similarly, in Linked Data KBs, data is represented using triples of the form  $\langle \textit{subject}, \textit{predicate}, \textit{object} \rangle$ . In Description Logics (DL), the basic representation of knowledge about entities is the *concept*, and in first order logic (FOL), this is denoted by a unary predicate. To describe properties of an entity, DL uses *roles*, and in FOL, these are represented by binary predicates. For instance, given two entities,  $x$  and  $y$ , and a role  $r$ , the FOL statement  $r(x,y)$  commonly refers to the fact the entity  $x$  has some property  $r$  whose value is denoted by  $y$ . Graphically, this can also be represented as two nodes, one each for  $x$  and  $y$ , and a directed edge,  $r$ , between them (see Fig 4.1). This forms a natural representation of facts about entities and is a widely used representation in knowledge bases.

Facts retrieved from the external KBs used by FRANK are primarily based on RDF (Beckett and McBride (2004)) and are queried using the SPARQL query language or

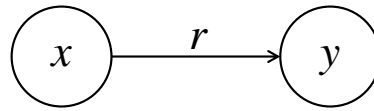


Figure 4.1: Triple for representing facts

---

$x$  has a property  $r$  with value (object)  $y$ .

---

specific web APIs provided by the KBs. For datasets such as tables or document-based databases represented using JSON (Javascript Object Notation)<sup>1</sup>, we normalize the representation by extracting individual facts into the form  $\langle \textit{subject}, \textit{predicate}, \textit{object} \rangle$ .

### 4.2.2 Extending Triples to alists

In FRANK, we require not only information about facts, but also meta-information such as uncertainty in the accuracy of the fact (variances in real-valued answers), temporal information about when that fact holds true and the unit of measurement of any values. Although some of this object-level information (e.g. time and units) about facts may exist as triples in the database, the representation of facts as triples limits the inclusion of the information that is useful for reasoning. That is, we are interested in a representation of facts that tells us the extent to which we can trust a triple, and when in time that fact holds true. It is important to note that although other logics such as temporal logics (Pnueli (1977)) and fuzzy logics (Bellman and Zadeh (1977)) deal with the truth value of facts with respect to time and the probability of truth of a fact, respectively, the semantics of time and uncertainty are different from ours. Our use of time and uncertainty are explained later in this chapter in sections 5.3.6 and 6.6.3 respectively.

In FRANK, we use *association lists* (*alist*) to represent data from KBs. Formally, an alist is a data structure containing a set of attribute-value pairs that describe features of an entity (section 4.3.2).

An important reason for our using a flexible representation of data in FRANK is the variety of ways in which data is represented in web knowledge bases. For instance, graph databases (including Linked-data KBs) represent data as triples (RDF for Linked-data). Traditional relational databases management systems (RDBMS) use tables (relations). Recent NoSQL databases such as MongoDB<sup>2</sup> and CouchDB (Anderson et al. (2010)) use a document-based representation and are used in large-scale

---

<sup>1</sup><http://www.json.org>

<sup>2</sup><https://www.mongodb.com>

```

{
  subject: uk,
  property: population,
  object: 63182000,
  time: 2011,
  aggregate: VALUE,
  cov: 0.035,
  ...
}

```

Figure 4.2: FRANK alist example

This alist represents the *population* of *UK* at *time* 2011 as 63182000. The uncertainty about this fact is given by the *Coefficient of Variation (cov)* of 0.035. The *cov* shows what fraction of the *object* value accounts for the estimated deviation from the true answer. In this example, the error margin is  $\pm 2211370$ . The operator, *VALUE*, is an identity operation that returns the value of the child alist. .

---

web data where the schema of the data is not always known or varies. The flexibility of FRANK’s alist representation means that data from KBs with these different forms of representations can easily be used during inference.

We obtain our alists data structure by extending our representation of facts such that we can dynamically add object-level and meta-level information to the triples. We augment the *subject(subj)*, *predicate(pred)*, *object(obj)* triple found in RDF KBs with an *alist* representation that contains an unbounded number of additional attributes such as time, uncertainty, units of quantities, and other features, as required. An alist is a list of attribute-value pairs<sup>3</sup>. Attribute names include, but are not limited to, *subject*, *property*, *object*, *time* and *cov* (coefficient of variation, a measure of uncertainty in FRANK). In FRANK, we use the more general attribute name ‘*property*’ instead of ‘*predicate*’, which is usually only used for properties in statements whose values are either “true” or “false”. During inference, another attribute that is used is the aggregates attribute that is applied as leaf values are propagated back up the proof tree. We use the “aggregate” key to represent this. For example, the alist shown in Figure 4.2 represents the population of the UK in 2011 and the confidence FRANK has in this fact.

---

<sup>3</sup>This is also referred to as a *dictionary* in some programming languages

Additionally, the alist representation allows the FRANK to specify aggregates in nodes in the FRANK tree. This becomes useful during upward propagation and is discussed in section 4.7.

It is important to note that although alists share some resemblance with ‘frames’ as used in (Minsky (1974)) and (Lehmann (1992)), it differs somewhat in terms of the semantics of the representation and (or) how it is used during inference. Minsky describes an alist as a data structure for representing a stereotyped situation, with several types of information being attached to the frame. Lehmann, similarly, defined an alist as a named data object with a flexible collection of named slots and some slots can be typed. This is similar to our alist representation in FRANK. Whereas Minsky and Lehmann’s definition of frames (and frame systems) represent data structures for representing knowledge about domains, with variable fields in frames, we use alists for curating data in the tree during inference without being constrained by the source KB’s data representation. That is, we consider alists as the representation of units of knowledge used only in the FRANK tree, and not in the source KB’s domain knowledge representation.

FRANK’s alist attributes are not strongly typed for all fields. The only exceptions are the *cov*, a real value, and the *time* attribute that is of *date-time* type based on the ISO date-time standard<sup>4</sup>. We expect arbitrary data values to be stored in other attributes of an alist and so we infer the types during runtime. We interpret time as the period defined by the date-time component that is specific in the alist. For example in figure 4.2, we interpret 2011 as all date-time values with a year component of 2011. When searching knowledge bases, FRANK compares the date-time value in the alist to the appropriate date-time component in the KB’s facts’s date values. The subject and property attributes usually have values of string types. Object attributes have arbitrary types and are determined by the type of the value retrieved from KBs.

Our use of alists is, therefore, similar to the ‘records’ representation in functional programming languages such as Haskell (Hudak et al. (1992)) or ‘maps’ in object-oriented programming languages such as Java (Arnold et al. (2000)) and ‘dictionaries’ in Python (Van Rossum et al. (2007)). The difference between the alist data structure and these is FRANK’s use of field-specific types.

---

<sup>4</sup><https://www.iso.org/iso-8601-date-and-time-format.html>

### 4.2.3 Importance of alists in FRANK

Alist play two important roles in FRANK that makes FRANK different from other inference frameworks. These are (1) automatic curation and (2) enrichment.

Automatic curation helps to put data from different sources into a common framework during runtime to allow FRANK to uniformly process them. This is vital as FRANK performs inference using data from heterogeneous KBs having a variety of knowledge representations.

Beyond its use as a data structure for labelling nodes and for internally representing facts and aggregates in the FRANK tree, alists give FRANK a lot of flexibility to perform different kinds of inference. The length of an alist is not bound. This means that although there are default attributes such as *subject*, *property*, *object*, *time*, *cov* and *aggregate* that are expected in each alist, an alist can be enriched dynamically by adding new attributes to the alist at runtime to allow specific aggregates to store auxiliary variables (see section 4.4.4).

## 4.3 Search Tree

### 4.3.1 Definitions

To facilitate the explanation of the inference process, we first provide brief definitions of the main concepts in FRANK. In subsequent sections, we delve deeper into these concepts.

**Definition 1. Alist:** *A list of attribute-value pairs for representing a query or facts from a KB in FRANK.*

The alist has as a minimum the following attributes: subject, predicate, object and *cov* (coefficient of variation). An alist can however have an arbitrary number of additional attributes. Attributes of an alist can be either literals, variables or logical expressions. We define literals in terms of *plain literals* in RDF (Beckett and McBride (2004)) and not its definition in logical representation.

Variables indicate the attributes of the alist to be instantiated at the leaves by being looked up in KBs, and elsewhere by propagation. They represent the unknowns that the inference framework attempts to find. All variables in a FRANK node are bound and existentially quantified. We assume that there exists an object in a KB, or some object that can be inferred from a KB that we can instantiate the variable with. Variables are

prefixed with either the \$ or ? symbols. Variables whose values are returned from a node are prefixed with the ? symbol. We refer to these as *projection variables* (section 4.4.3). For instance,

```
{ag:MAX, subject:uk, property:population, object:?y, time:2020, agvar:?y}
```

shows that the value of the variable  $y$ , is unknown and must be inferred and returned. The aggregate *MAX* is applied to the aggregate variable (*agvar*)  $?y$ . An answer to a query is found when all projection variables (i.e. variables prefixed with  $?$ ) are instantiated to ground terms in a KB. The \$ sign is used as a prefix for query variables that unknowns in the query and must be instantiated to data in KBs. Variables are instantiated by looking up values in KBs or by aggregating values from other nodes. Aggregate, query and projection variables are discussed in section 4.4.3.

**Definition 2. Simple alist:** *An alist whose attributes contain only literals (terms in plain text of string, real, boolean or date-time types) or variables. Such a alists is said to be in FRANK Normal Form.*

The following is an example of a simple alist:

```
{ag:VALUE, subject:Edinburgh, property:gdp, object:?y, time:2001, agvar:?y}
```

**Definition 3. Compound alist:** *An alist that has at least one attribute that contains a complex expression such as a subquery or a logical expression.*

An example of a compound alist is:

```
{ag:VALUE, subject:capitalOf(Scotland,$x), property:gdp, object:?y,
time:2001, agvar:?y}
```

where `capitalOf(Scotland,$x)` is the logical formulae with free variables. A compound alist is not in FRANK Normal Form.

**Definition 4. FRANK Tree:** *A tree whose nodes are labelled by alists, where each child node is derived from a decomposition of its parent node, and values of parent nodes are instantiated by an aggregation and propagation of values from child nodes.*

The FRANK tree is a search tree that uses AND-OR branching to explore the search space during inference.

FRANK performs inference on a tree in two directions:

1. downward decomposition: a map operation that transforms parent nodes in the FRANK tree by decomposing them into child nodes using appropriate rules.



This begins with the process of internalisation (section 4.7.1) when a query string is converted into a FRANK node and normalized (i.e., compound alists are normalized to simple alists). Aggregate functions of decomposed alists are also changed to specify the aggregate that would later combine the values of child nodes into the values of the parent nodes.

2. upward propagation: a reduce operation that uses aggregate function to combine variables in alists and propagate values from the leaf nodes back up the FRANK tree to the root. This also includes the propagation of uncertainty.

**Definition 5. FRANK Node:** *Each node in the FRANK tree is labelled by an alist.*

Nodes are connected to their descendants by rules after decomposition.

**Definition 6. Decomposition Rule:** *A rule that defines a transformation on an alist from which child nodes in the FRANK tree are derived.*

This happens during the downward decomposition of nodes in the FRANK tree. Selection of rules to be used at a given node of the FRANK tree is based on heuristics. Decomposition rules label the arcs of the FRANK tree and define the aggregate functions of the child alists they create.

A decomposition rule also determines the aggregate that the decomposed alist should use to combine the values that each of the child nodes will eventually return into a single value for the parent node. In some cases, a rule could lead to two different aggregates for combining the child nodes. For instance, in a query that requires the average value of a specified property, a rule would specify the *AVG* aggregate. In another query that requires the *total* value of a property between two dates, a rule could specify the *SUM* aggregate or the *INTEGRAL* aggregate, each of which will take a different strategy to solving the query.

**Definition 7. Aggregate Functions (aggregates):** *A function (usually arithmetic or statistical) that aggregates values from a set of alists.*

Aggregate functions are applied during the upward propagation of alists in the FRANK tree. The aggregate function of an alist is specified in its *aggregate* (shortened as *ag*) attribute. An aggregate can be explicitly defined in the query or can be specified by a rule during decomposition. The variable on which the aggregate is performed is stored in the *agvar* attribute of the alist. For instance the alist:

Which country will have the largest population in Africa in 2021?

`MAX($y, COMP(<?x, $y>, population(?x, $y, 2021) : Country(?x) & location(?x, Africa)))`

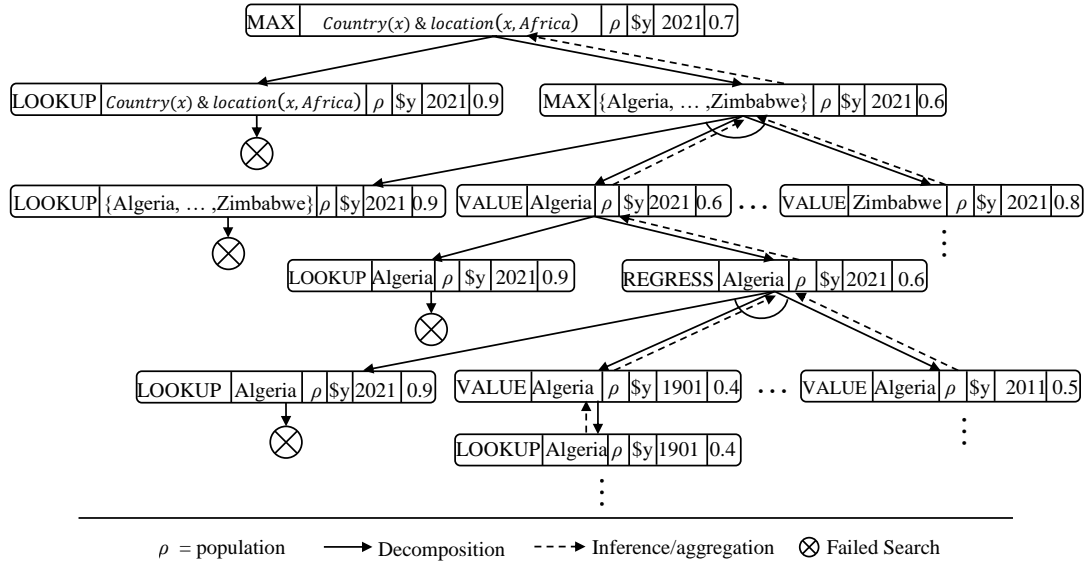


Figure 4.3: FRANK Illustration

Initial decompositions (showing AND-OR search tree). Alists are compactly denoted by  $\{ag, subj, pred, obj, time, cov\}$ .

```
{ag:MAX, agvar:?y, subj:country($x) & locatedIn($x,Europe), property:gdp,
  obj:?y, time:2001}
```

requires that the maximum gdp in 2001 of countries in Europe is found from the child alists, which are expected to be the gdp values of countries in Europe.

Aggregate functions and decomposition rules are discussed in more detail in chapter 5.

### 4.3.2 Formalising FRANK

We give formal semantics to FRANK that correctly captures the behaviour of its data model and query language.

#### 4.3.2.1 Data Model

FRANK performs inference using a set of attribute-value pairs called alists. Attributes include operations, variables on which operations should be performed, subject, object, time, uncertainty and others as determined by the inference operations. Towards capturing this, we define the following three sets:

- the countable infinite set  $\mathcal{A}$  of names, that will serve as names of attributes including (subject, object, time, *cov*, etc.).

We denote attribute names with uppercase letters and their values with lowercase letters. Required attributes in an alist include:

*H* *inference operation*: an identifier of an inference operation that maps to an implemented aggregation function. Possible values of *H* are shown in table 4.1.

*V* *operation variable*: an array of variables in the alist to which *h*, the value of *H*, is applied. This variable must exist as the value of one of the other attributes in the alist.

*S* *subject*: subject element of the subject-property-object triple used in databases and natural language processing. It is the entity that the alist describes.

*P* *property*: the relation (property) element from the subject-relation-object triple in the alist.

*O* *object*: the object from the subject-relation-object triple in the alist.

*T* *time*: a date/time value that specifies the time at which the subject-relation-object triple is true.

*C* *cov*: a coefficient of variation value that assigns a measure of uncertainty to the accuracy of a real-valued fact specified by the subject, predicate, object and time attributes. This can be interpreted as an error bar on the real-valued answer.

- the countable infinite set  $\mathcal{V}$  of names, that will serve as values of attributes including data values, function terms and logical formulae. For the above listed attributes, we denote their respective values as *h, v, s, p, o, t* and *c*.
- the countable infinite set  $\mathcal{X}$  of names that will serve as existentially quantified variables that can be assigned to attributes. Variable are prefixed with ? or \$ and denote an unknown value for an attribute that has to be looked up or inferred. *Projection variables*, prefixed with ?, indicate those that are projected (returned) by an operation. Variable projection is the process of returning the values of relevant attributes to a parent nodes in the inference tree. *Auxiliary variables*, prefixed with \$, are also unknowns to be looked up or inferred but are not projected. See section 4.4.3 for details about the different kinds of variable

in FRANK. Variables can be used as the values of the *subject*, *object*, *time*, *cov* or other attributes. The *property* attribute can also be assigned a variable, although we do not do this in this project. Without loss of generality, we represent variables in the remainder of this section as projection variables.

We now define the data model for FRANK. Formally, an alist is a set of attribute-value pairs (tuples) that describe features of an entity as well as operations to be performed during inference. We construct and use alists as follows.

Let  $\{\langle \mathcal{A}_1, a_1 \rangle, \dots, \langle \mathcal{A}_n, a_n \rangle\}_f$  be an alist,  $f$ , where  $\mathcal{A}_i$  is an attribute and  $a_i$  is the value assigned to the attribute.

$f[\mathcal{A}_i]$  refers to the value  $a$  of the attribute  $\mathcal{A}_i$  in  $f$ .

$f[b/?x]$  denotes the substitution of all occurrences of projection variables  $?x$  in  $f$  with the value or expression  $b$ . Similarly,  $f[b/\$y]$  denotes the substitution of all occurrences of non-projection variables  $\$x$  in  $f$  with the value or expression  $b$ .

$f[\mathcal{A}_i] := a'$  denotes the assignment of the value  $a'$  to the attribute  $\mathcal{A}_i$  in  $f$ . Value assignments are performed with arithmetic operations discussed in section 5.5.

FRANK does not maintain a persistent store of data as found in a relational database or an RDF triple store. Rather, FRANK automatically queries a variety of KBs and curates just the data needed to answer the queries at the leaves of its inference tree. FRANK interfaces with KBs via wrappers created for the different classes of KBs including relational DBs, RDF/SPARQL endpoints and JSON/REST APIs. Query templates are created for each KB wrapper to map alists to KB queries.

Let  $\omega_k$  be a wrapper for KB  $k$  and let  $\tau_\omega$  be a query template in wrapper  $\omega$ . We denote a query to look up facts from a KB or local cache using  $\tau_\omega$  as  $\ell(\tau_\omega(f))$ . Variables at the leaves of the inference tree, except the *cov*, are therefore instantiated by the substitution  $f[\ell(\tau_\omega(f))/?x]$ .

#### 4.3.2.2 FRANK Decomposition and Aggregation Operations

FRANK executes a query by building and evaluating an inference tree using decomposition rules and aggregation operations. Given an alist  $f$  and a decomposition rule operation  $\Delta$ , child alists,  $g_i$ , are derived from  $f$  using expression:

$$f \xrightarrow{\Delta} \langle g_1, \dots, g_n \rangle$$

where  $\xrightarrow{\Delta}$  denotes the transformation of alist  $\mathbf{f}$  under operation  $\Delta$ , and  $\langle \dots \rangle$  is an order-preserving tuple.

Every alist also specifies how the variables in child alists are aggregated and propagated to the parent alist. To instantiate and propagate the operation variable  $v$  from  $\mathbf{g}$  to  $\mathbf{f}$ , we perform the substitution:

$$\mathbf{f}[h(\mathbf{g}_1[v], \dots, \mathbf{g}_n[v])/v]$$

where  $v = \mathbf{f}[V]$ ,  $h = \mathbf{f}[H]$  and  $h$  is applied to the  $v$  attributes of the child alists  $\langle \mathbf{g}_1, \dots, \mathbf{g}_n \rangle$ . This process happens recursively from the leaves of the FRANK tree to the root alist. The value of  $cov$  is also calculated during this step. This is explained in chapter 6.

It is important to note that, unlike inference mechanisms of SPARQL and other relational DBs that are in first-order and so have variables that range only over first-order terms in the KB, alists can have variables that range over functions. This allows functions (e.g. regression functions for prediction) to be assigned to attributes of alists such they can be evaluated on specific values further up in the inference tree. A formalisation of aggregation functions available in RIF can be found in section 5.5.

### 4.3.2.3 Scopes and Bindings

Each alist defines a scope for variables, and scopes are nested according to the structure of the FRANK tree. During decomposition, variables that are not decomposed are passed on to child alists with their respective attribute names. However, the variables in each alist in the tree has a local scope.

Thus, suppose that  $\mathbf{f}$  is an alist in the inference tree and  $\mathbf{f}[\mathcal{A}_i] = ?x$ . The attribute-value pair  $\langle \mathcal{A}_i, ?x \rangle$  can be passed on to its children  $\mathbf{g}_1$  and  $\mathbf{g}_2$ , but the value of  $?x$  in  $\mathbf{f}$  is not necessarily equivalent to the value of  $?x$  in  $\mathbf{g}_1$  or  $\mathbf{g}_2$ . The value of the operation attribute,  $h$ , in a parent alist determine how variables projected from child alists are aggregated and assigned to attributes or bound to its variables. This is done as part of the alist decomposition process to ensure that the appropriate values from the child alists are used for the right arguments in the parent's operation.

### 4.3.2.4 Query Language

We define the syntax of the query language as follows.

$$\begin{aligned}
Q &:= \mathcal{H}(x | \langle x_1, \dots, x_n \rangle, Q | \alpha) \\
\alpha &:= p(s|x, o|x[,t|x]) [ : \theta ] \\
\theta &:= p(s|x [,o|x]) (\& p(s|x [,o|x]))^* \\
\mathcal{H} &:= \text{VALUE} | \text{VALUES} | \text{SUM} | \text{DIFFERENCE} | \text{PRODUCT} \\
&\quad | \text{AVERAGE} | \text{MEDIAN} | \text{MAX} | \text{COUNT} | \text{MIN} | \text{GT} | \text{LT} | \text{EQ} \\
&\quad | \text{REGRESS} | \text{GP} | \text{INTEGRAL} | \text{DERIV} | \text{APPLY} | \text{COMP} \\
x &\in \mathcal{X} \text{ (the set of variable names)}
\end{aligned}$$

$p, s, o, t \in \mathcal{V}$  (the set of attribute values as defined in the data model (section 4.3.2.1))

#### 4.3.2.5 Semantics

The query language described above is meant to facilitate the composition of queries by users. It is a compromise between having users create queries using somewhat non-intuitive sets of attribute-value tuples to define alists, and having them use natural language that FRANK is not equipped for. The query language directly maps query components to alist attributes and so we discuss the semantics of the alist rather than the query language.

We distinguish between *object-level* and *meta-level* components in FRANK. Object-level components, including values of attributes are obtained from the *world*,  $\mathcal{W}$ , a set of KBs. Meta-level components including attribute names, function identifiers and variable names are considered part of FRANK's *environment*,  $\eta$ .

We denote the semantics of an alist  $f$  by  $\llbracket f \rrbracket$ . This is a function that takes a world,  $\mathcal{W}$ , and an environment,  $\eta$ , as inputs and produces the output  $\llbracket f \rrbracket_{\mathcal{W}, \eta}$  which are facts obtained by executing  $f$  on  $\mathcal{W}$  using the environment  $\eta$ .

We define the semantics of components of an alist and operations using the semantic function  $\llbracket \cdot \rrbracket$  in figure 4.4.

#### 4.3.3 Dual Role of Alists

From our definition of the attributes in section 4.3.2, one can observe that an alist is made up of *object-level* attributes and *meta-level* attributes. Object-level attributes define the domain data that is captured in the alist (thus, data about properties of entities or real-world concepts). Object-level attributes include *subject*, *property* and

$$\begin{aligned}
\llbracket \mathbf{v} \rrbracket_{\mathcal{W}, \eta} &= \begin{cases} \mathbf{v} \in \mathcal{W}, & \text{for object-level values} \\ \mathbf{v} \in \eta, & \text{for meta-level values} \end{cases} \\
\llbracket \mathcal{A}_i \rrbracket_{\eta} &= \mathcal{A}_i \in \eta \\
\llbracket \langle \mathcal{A}_i, \mathbf{v} \rangle \rrbracket_{\mathcal{W}, \eta} &= \langle \llbracket \mathcal{A}_i \rrbracket_{\eta}, \llbracket \mathbf{v} \rrbracket_{\mathcal{W}, \eta} \rangle \\
\llbracket \mathbf{f} \rrbracket_{\mathcal{W}, \eta} &\equiv \llbracket \{ \langle \mathcal{A}_1, \mathbf{v}_1 \rangle, \dots, \langle \mathcal{A}_n, \mathbf{v}_n \rangle \}_{\mathbf{f}} \rrbracket_{\mathcal{W}, \eta} \\
\llbracket \{ \langle \mathcal{A}_1, \mathbf{v}_1 \rangle, \dots, \langle \mathcal{A}_n, \mathbf{v}_n \rangle \}_{\mathbf{f}} \rrbracket_{\mathcal{W}, \eta} &= \mathbf{f} \{ \llbracket \langle \mathcal{A}_1, \mathbf{v}_1 \rangle \rrbracket_{\mathcal{W}, \eta}, \dots, \llbracket \langle \mathcal{A}_n, \mathbf{v}_n \rangle \rrbracket_{\mathcal{W}, \eta} \} \\
&\quad \text{for } n > 0. \\
\llbracket \langle \mathbf{f}_1, \dots, \mathbf{f}_n \rangle \rrbracket_{\mathcal{W}, \eta} &= \langle \llbracket \mathbf{f}_1 \rrbracket_{\mathcal{W}, \eta}, \dots, \llbracket \mathbf{f}_n \rrbracket_{\mathcal{W}, \eta} \rangle, \quad n > 0
\end{aligned}$$

During up-propagation,

$$\begin{aligned}
\llbracket \mathbf{f} \{ \langle H, h \rangle, \langle V, \mathbf{v} \rangle \}, \dots \rrbracket_{\mathcal{W}, \eta} &= h(\llbracket \mathbf{g}_1[\mathbf{v}] \rrbracket_{\mathcal{W}, \eta}, \dots, \llbracket \mathbf{g}_n[\mathbf{v}] \rrbracket_{\mathcal{W}, \eta}) \\
&\quad \text{where } \text{children}(\mathbf{f}) = \langle \mathbf{g}_1, \dots, \mathbf{g}_n \rangle, \\
&\quad h \in \eta \text{ and } n > 0
\end{aligned}$$

Figure 4.4: Alist Semantics

*object*. These are comparable to the data triples found in other relational DBs. We also use *time* as an object-level attribute. However, meta-level attributes, which are domain-independent, allow us to (1) make statements about the values of the object-level attributes; (2) express operations to perform on the alist during inference (e.g. the operation and operation variable attributes). For instance, the *cov* denotes uncertainty about the fact in the alist while the inference operation attribute, *h*, specifies the function for aggregating of child alists.

These two categories of attributes highlight the dual role of alists in FRANK:

1. as a data structure for dynamically curating data from diverse KBs into a common format during inference;
2. as a functional structure for performing operations on alists.

Given that values of attributes could also be functions means functions such as a regression curves could be assigned to an object-level attribute in an alist. Similarly, given

that an alist can be interpreted as a function with the aggregate attribute as the function label pointing to an implementation of the function in an external library to FRANK, we can assign values of projection variables from one alists to values of attributes in another alist. This feature gives FRANK its functional and nested query capabilities, and is also a significant difference between FRANK's alist and SPARQL/RDF.

#### 4.3.4 Node States

During inference, it is necessary to keep track of the states of nodes as they are spawned through decompositions and as they are processed and propagated. Similar to common graph search algorithms such as Breadth-First Search, Depth-First Search and Depth-Limited Search, it is important to know which nodes in the graph have been explored and which ones remain at the frontier of the search space. In FRANK, first the graph expands dynamically during inference as different rules decompose nodes and create new child nodes. Then we also have to propagate variables in alists upwards towards the root once we get to the leaves of the graph and nodes get grounded. We therefore need to maintain states of nodes to facilitate this exploration and up-propagation process.

A node in the FRANK tree can be in one of 3 states:

1. **Unexplored:** New nodes that are added to the FRANK tree start off in the unexplored state. The 'unexplored' state marks a node that is available for processing. The policy that governs decomposition rules selection determines which of the nodes is processed first. Currently, we use a policy that is based on the estimated cost of processing the node, where the node with the cheapest cost is selected first. Inference cost is discussed in section 5.8.
2. **Grounded:** This state is used for nodes whose projection (?) variables are all instantiated to actual data values after a successful search. This is applicable to leaf nodes, where search queries generated from their alists match triples in a knowledge base. It also applies to non-leaf nodes whose variables get bound to values reduced and propagated from their child nodes. Propagation involves applying aggregates to alists such that only the evaluated value is passed up to the parent alist and not all values from all child nodes. This is different from resolution inference where such application does not take place. In unification, instantiations are passed up to parents, but the parent expressions are not evaluated before they



are propagated further. Additionally, the *grounded* state determines whether a node, together with its siblings, can propagate their values to their parent node.

3. **Failed:** This state marks a node for which grounding failed. For instance, when no data in the KB satisfies the lookup query generated for a node. It could also happen when there is an insufficient number of grounded nodes required for aggregation in the parent node. This state marks a node as a candidate for decomposition if it occurs as a leaf node. If constraints on the FRANK tree such as “depth limit” are not violated and a decomposition rule is capable of transforming this node, the node gets decomposed and the newly spawned nodes are connected as its child node.

### 4.3.5 AND-OR Branching

The FRANK tree has two kinds of branching: AND branching and OR branching. During inference, new nodes are spawned by the downward decomposition as needed such that an answer is found for the internalized query in the root node. Starting with the root node, a decomposition rule is selected in order to grow the FRANK tree in such a way that an answer can be found even when direct lookups in KBs fails. For any node, one or more aggregate functions listed in table 4.1 may be applicable. As a result, the selection of a rule is an OR choice that leads to different approaches for the node being decomposed.

Next, for each rule selected, FRANK decomposes the node to obtain child nodes. For instance, if a node is decomposed using the temporal rule, a date/time attribute of the alist is generated across many years in order to perform regression. Each of these child nodes must be solved before the parent node gets solved. These child nodes make up the AND branches. That is, AND branches are required to be solved for inference to proceed. The only exception to this requirement is the *lookup* rule that generates similar or related words for attributes in an alist. In this scenario, OR branching is used to find a match to any of the new child nodes.

AND and OR branches both affect the workings of the framework. OR branches are traditionally known to cause headaches when appropriate heuristics are not used to guide the search. The design of these heuristics determine how well the search tree avoids branches that do not lead to a solution to the task, thereby minimizing the size of the FRANK tree. This includes pruning unwanted branches, and ordering the branches to explore based on cost heuristic that leads to a best-first search exploration of the

tree.

However, in FRANK, AND branches can also be a source of significant headaches. For example, a strategy that attempts to solve for the whole by first solving its constituent parts and then aggregating could end up spawning several tens or sometimes hundreds of new nodes, all of which *must* be solved. Each of these new child nodes could potentially spawn OR branches in subsequent decompositions. This makes AND branches in the FRANK tree just as problematic as the OR branches, leading to a combinatorial explosion of the FRANK tree in the absence of appropriate search heuristics.

### 4.3.6 Alist and RDF Reification

RDF reification (Manola et al. (2004)) is the use of RDF statements to describe other RDF statements and to record information about when statements were made, who made them, or other similar information (this is sometimes referred to as “provenance” information). Consider the alist  $\{S = s, P = p, O = o, P_1 = a_1, \dots, P_n = a_n\}_f$ . We can express this as a reified RDF statement using the following RDF triples:

```
< q, type, rdf:Statement >
< q, rdf:subject, s >
< q, rdf:predicate, p >
< q, rdf:object, o >
< q, p1, a1 >
...
< q, pn, an >
```

where  $q$  is the URI that refers to the fact  $\langle s, p, o \rangle$ , and the  $p_i$ s are predicates representing the attribute URIs for the values  $a_i$  respectively.

Although, theoretically, any alist can be described in RDF using RDF reification, there are practical reasons why such reification is not best suited for FRANK.

Reasons include the following:

1. Most RDF KBs do not include reified RDF statements and so FRANK cannot base most queries on RDF reification.
2. FRANK is not built as a data storage engine like an RDF triple store. FRANK's alists only curate facts from KBs that are relevant for calculating answers and is not obliged to store these facts permanently. As a result, the concerns of RDF and reified RDF are different from the concerns of alists.

3. Most importantly, FRANK uses KBs with formalisms other than RDF, and so cannot be limited to representations that are specific to RDF. Since FRANK's mathematical and statistical operations go beyond that offered by SPARQL, and FRANK allows functions to be inferred and assigned to variables, RDF/SPARQL will require significant extensions to perform such non-RDF operations. Such would have required more time and resources than we have for this project, so, we will explore such extensions in future work.

Instead of the reified representation of the RDF KBs, FRANK uses triples in a compositional manner to query KBs at runtime. That is, finding the appropriate property (or predicate URI in the case of RDF KBs) in the KB (if one exists) that matches the alist property for which a data value is needed. This allows FRANK to automatically create simple queries using KB-specific templates with which it can look up data to instantiate variables. It is important to note that these KBs could differ in their representation and formalisms. Also, by 'simple queries', we mean queries that do not include any form of nesting and do not include any mathematical operation. We do not have to automatically construct 'complex' SPARQL queries since FRANK breaks down alists to simple forms from which SPARQL queries that look up properties of subjects can be created. Additionally, the automatic formulation of simple queries used in FRANK makes it possible to perform federated queries on different KBs having different knowledge representations. This is explained in section 7.4 and demonstrated in experiment 4b (section 8.5.2).

#### 4.3.7 Going beyond RDF, SPARQL and SWRL

FRANK is not built to be a replacement for RDF, SPARQL and SWRL (Semantic Web Rule Language) (Horrocks et al. (2004)). For instance, during inference, FRANK automatically composes SPARQL queries using predefined query templates to search KBs that use the RDF representation. However, we do not limit FRANK to RDF knowledge sources only. We also do not limit FRANK to the kinds of operations available in SPARQL. For instance, in FRANK we want to be able to infer functions and assign them to variables, or perform operations on the functions.

So, although the *RDF*, *SPARQL* and *SWRL* (Semantic Web Rule Language) (Horrocks et al. (2004)) have advanced knowledge representation and reasoning on the web, we are unable to successfully tackle our objectives for FRANK using only these tools. For the purposes of our discussion in this paragraph, we summarize FRANK's goals

in the three points below:

1. Use data from KBs with diverse representations
2. to answer questions that require the formation and application of non-trivial mathematical and statistical functions in a compositional manner to infer novel data (including regression functions for prediction)
3. by automatically constructing an inference tree from the recursive decomposition of the appropriate features in the query.

For each of these points, we explain our reasons for choosing options outside the RDF/SPARQL/SWRL family to solve our challenges.

1. *RDF*: Although RDF provides a suitable formalism for representing data in the Semantic Web, several rich sources of data (e.g. the World Bank) provide their data in other formalisms that are popular on the web (e.g. JSON) via their APIs. Our aim of using data from a variety of sources without being limited by their heterogeneous representation cannot be achieved by restricting FRANK to RDF knowledge sources only.

FRANK accommodates RDF and non-RDF datasets by automatically curating the data into a common form as part of the inference process by using the *alist* representation. For instance, relational databases and RDF use data triples representation. However, a single triple is not sufficient to represent facts about a country's population at a given date (using the World Bank's data as an example). Multiple triples are needed to capture the temporal and geospatial dimensions of such facts. Although reified RDF (Manola et al. (2004)) can provide a mechanism to do this, in FRANK, we prefer a more compact representation that allows us to specify information about these dimensions for every fact during curation. For this reason, we prefer the *alist* representation over triples and we, therefore, do not translate non-RDF data into RDF. Section 4.3.6 discusses other reason for preferring *alist* over reified *rdf*.

2. *SPARQL*: SPARQL has just a handful of mathematical operations: Count, Sum, Min, Max, Avg. There have been many projects, particularly in data analytics such as (Dolby et al. (2016)), that extend these operations for specific domains (e.g. medical domain). However, these techniques still ship the data off

to external tools to deal with the operations. Most importantly, unlike FRANK, SPARQL does not infer functions nor perform inference over functions. Our decision not to use SPARQL in this project as the inference mechanism for FRANK is that significant work would be required to extend SPARQL to perform higher order operations using non-RDF formalisms and functionality.

3. *SWRL*: Answering a FRANK query requires the decomposition of the root (query) alist. This means that an operation for the child alists also has to be selected based on the kind of decomposition performed. SWRL (Semantic Web Rule Language) (Horrocks et al. (2004)) is limited since its rules are based on a combination of OWL DL and OWL Lite sub-languages (McGuinness et al. (2004)) with unary/binary Datalog RuleML (Boley et al. (2010)) that are at most first-order logic.

However, FRANK uses rules and operations that are second-order. Since functions (e.g. regression functions) can be formed and assigned to values of attributes in alists, any rules about alists need to include clauses about functions. This cannot be expressed in SWRL since SWRL's formalism cannot express higher-order operations such as linear or polynomial regression.

Finally, to use SWRL rules, the underlying data must be in RDF with clearly defined domain ontologies with classes and subsumption relations present. Unfortunately, as noted in the first point above, FRANK also uses KBs (e.g. the World Bank Dataset) that are not represented in RDF and that do not provide explicit ontologies about entities and types in their datasets. As a result, we cannot universally apply SWRL rules to non RDF-data, even if rules in such rule representation language were a good fit for our framework.

Given these reasons, it makes sense to consider a more general solution to QA outside the RDF/SPARQL/SWRL family, while leveraging SPARQL to search KBs that use the RDF representation.

## 4.4 Queries

**Definition:** A *FRANK* query is an alist with uninstantiated projection variables.

A query is expressed as a composition of aggregate functions and contains both (1) functional expressions for defining aggregates to be performed on alists, and (2) logical expressions for describing entities and relations about which the query is posed.

Although a FRANK tree performs inference using alists, FRANK queries provide a means for users to pose questions in a form that FRANK can parse and process. A query is internalized and converted to alists (see section 4.7.1) when FRANK begins execution. This defines the first few levels in the FRANK tree that are expanded further by decomposition rules. An example of a FRANK query is

```
VALUE(?y,urban_population(Uk,?y,2011))
```

It asks the question “*What was the urban population of UK in 2011?*” We express statements in the logical components in the form

$$property(subject, object[, time])$$

for two reasons:

1. correspond with the  $\langle subject, property, object \rangle$  representation found in triples. The optional *time* attribute allows us to specify temporal filters when searching KBs.
2. to fit better with the overall functional representation of the framework as well as the compositional representation of queries.

Multiple logical statements can be used in a query and are combined to form either conjunctive statements or disjunctive ones. FRANK does not include negation in its queries. Negation increases the complexity of reasoning with the open-world assumption when using web KBs. We leave this for future work.

#### 4.4.1 Query Grammar

We use a context-free grammar in Extended Backus-Naur Form (EBNF) (Wirth (1996)) that, together with the type signatures of aggregate functions, defines well-formed queries. EBNF is a syntactic meta-language that offers a notation for defining the syntax of a language by a number of rules. Each rule names part of the language (called a non-terminal symbol of the language) and then defines its possible forms.

In FRANK, we represent queries with a formal grammar for two reasons:

1. FRANK does not take input queries in natural language. It is not the primary focus of this research but might be added in future work that extends this project. In the absence of natural language, we use a structured query representation that satisfies our requirement that it is (a) easy for users to pose correct queries that

are in the form that FRANK accepts, and (b) easy for FRANK to parse and process the query.

2. It allows us to focus on queries that follow a formal syntactic pattern that enables FRANK to verify the validity of the input query without the complexity of processing natural language queries, which would require non-trivial natural language processing (NLP) techniques.

Although existing query languages such as Structured Query Language (SQL) and SPARQL are popularly used, we choose to define this simple query language for the primary reason that the underlying mechanism for processing the queries in FRANK is significantly different from these other languages. This is highlighted in how the FRANK algorithm works in section 4.7.

FRANK queries take the form:

```
func_expr :: METHOD_NAME(
            (var|<var,var>),
            (logic_expr|func_expr)[,(logic_expr|func_expr)])
```

where *var* are variables, *func\_expr* are functional expressions and *logic\_expr* represents logical expressions. The full query grammar in EBNF is shown in appendix B.

The different types of FRANK queries are described in section 4.4.2. In this thesis, we use the convention where operations are written in uppercase and logical constants begin with either a lower or upper case.

Properties are not pre-defined prior to their use in FRANK. The framework finds property names that match properties in the KBs from which the corresponding subjects (or objects) are retrieved. The matching process uses string functions to split words in a property, leverages language resources such as WordNet (Miller (1995)) to find synonyms and applies edit distance measures to find matches to predicates in a KB. Our algorithm for this matching process is shown in section 5.3.4.

#### 4.4.2 Query Types and Examples

One of our objectives in this work is to show that we can extend the kinds of queries that can be asked, as well as the kinds of answer that can be inferred in an automated query answering systems. In FRANK, we group the queries into four types: (1) Fact retrieval, (2) Aggregation, (3) Nested and (4) Prediction. These are discussed below.

**Type 1 - Fact Retrieval Queries:** These are queries similar to the traditional kinds where the primary task is one of information retrieval. FRANK must find statements in KBs that match the logical expression in the Query. Queries of this type are often posed with the *VALUE* aggregate.

**Example:** *What was the urban population of UK in 2011?*

**Formal:**

```
VALUE(?y,urban_population(Uk,?y,2011))
```

This query simply requires FRANK to find facts about the urban population of the UK and retrieve only that for the year 2011. The main assumption here is that the year ‘2011’ is in the past (given that the current year is 2017) and that the fact exists in the KB. It is, however, important to note that FRANK is not limited by these assumptions, and makes an attempt to infer an answer even when the year is in the future or the fact for 2011 does not exist in the KB. This is one of the key differences between FRANK and other QA inference techniques.

**Type 2 - Aggregation Queries:** These are queries that require a combination (usually arithmetic) of separately retrieved facts to infer an answer. Aggregation queries use aggregate functions such as *SUM*, *MAX*, *MIN*, *AGV*, etc.

**Example:** *Which country had the lowest female unemployment in South America in 2011?*

**Formal:**

```
MIN($y,
  COMP(<?x,$y>, female_unemployment(?x,$y,2011):
    Country(?x) & location(?x, South_America)))
```

Listing 4.1: Aggregation query example

In the query above, the *MIN* aggregate (section 5.5.2) is applied to the list of  $\langle \textit{country}, \textit{population} \rangle$  pairs returned by the *COMP* (set comprehension) aggregate. *COMP* (section 5.7.3) creates these pairs by performing a set comprehension operation over using the *country* and *location* properties as filters over the available *female\_unemployment* datasets. The *MIN* aggregate is technically a ‘*min – by*’ over the population variable  $\$y$ , and the variable returned by the *MIN* aggregate is the *country* variable denoted by  $?x$ .



**Type 3 -Nested Queries:** These are queries that include explicit sub-queries. The recursive form of the FRANK grammar supports nesting aggregate functions to compose such queries. Nested queries require the evaluation of one or more sub-queries before an answer can be inferred for the full query. Queries that compare values are typical examples.

**Example:** *Was the rural population of the country with the largest arable land in Africa greater than the urban population of the country with the smallest arable land in Africa in 2003?*

**Formal:**

```
GT(?b,
  VALUE(?b,rural_population(
    MAX($d,
      COMP(<?c,$d>,arable_land(?c,$d,2003):
        Country(?c) & location(?c,Africa))),?b,2003)),
  VALUE(?b,urban_population(
    MIN($h,
      COMP(<?g,$h>,arable_land(?g,$h,2003):
        Country(?g) & location(?g,Africa))),?b,2003)))
```

Listing 4.2: Nested query example

The *greater-than* aggregate, *GT*, processes each of the two sub-queries defined in the *VALUE* inference aggregate before it can determine if the first is greater than the other. The first *VALUE* also contains a sub-query based on the *MAX* aggregate whereas the second *VALUE* has a different sub-query based on the *MIN* aggregate. The nesting capabilities also highlights the compositional nature of the FRANK operations. That is, complex queries can be composed and answered by applying aggregate functions to other aggregate functions.

Query 4.2 also shows the problems that users could encounter when composing complex queries in FRANK. It is unrealistic to expect ordinary users to formulate such queries easily. A solution to this would be to create a natural language query interface that is based query templates with placeholders that users can fill in. This is outside the scope of this project, but would be a worthwhile extension to FRANK in a future project.

**Type 4 -Prediction Queries:** These are queries that combine features of query types 1,2 and 3 but include the additional feature that some values to be retrieved are dates in the future, or in the past and the KB does not have facts that can be retrieved explicitly

to answer the queries or its subqueries.

**Example:** *What was the GDP in 2010 of the country predicted to have the largest total population in Europe in 2018?*

**Formal:**

```
VALUE(?y,gdp(
  MAX($b,
    COMP(<?a,$b>,population(?a,$b,2018):
      Country(?a) & location(?a,Europe))),?y,2010))
```

Listing 4.3: Prediction query example

The above query is based on the assumption that facts for the year 2018 are non-existent in the KB, although that for the year 2010 may exist. The goal is to predict the facts for 2018 to solve the *MAX* sub-query (using other existing data in the KB), and then use this predicted value in the rest of the inference process.

### 4.4.3 Roles of Variables in FRANK

Variables in FRANK are instantiated and propagated the same way. However, variables in FRANK play four main roles.

1. **Query Variables:** These variables represent unknowns in a query that are instantiated with actual values from a KB. These are defined by users in the query. For instance in the query

```
VALUE($y,urban_population(uk,$y,2010))
```

the \$y represents a variable for the *object* of the alist that is unknown. In this case, \$y represents the value of the urban population of the UK in 2010. FRANK's objective is to find an instantiation for \$y such that other conditions in the query (such as *time*) are satisfied. Query variables are created by prefixing a variable name with the \$ symbol.

2. **Projection Variables:** These are variables that work in a similar way as query variables, but perform an additional function. Projection variables determine which variable, from among other variables in a query, should be returned (projected) as the answer to the query. These variables are prefixed with the ? symbol. Consider the query below that seeks to determine the country with the lowest female unemployment value in South America.

```

MIN($y,
  COMP(<?x,$y>,female_unemployment(?x,$y,2011):
    & Country(?x) & location(?x, South_America)))

```

Listing 4.4: Query with projection variable

It has two variables  $x$  and  $y$  which have to be instantiated to answer the query. However, the primary objective of this query is not to tell the user what the lowest female unemployment value is, but rather the country that has this lowest value. For this reason, although both  $x$  and  $y$  are query variables,  $x$  is the variable to be projected and is the variable that must be instantiated, regardless of the status of other variables in the query. This is similar to the goal of the projection operator in Relational Calculus (Codd (1971)), implemented in the *SELECT* statement in SQL and SPARQL.

3. **Aggregate Variables:** These are query variables that determine which alist attribute an aggregate function should be applied to. For example in query 4.4 above, the *MIN* aggregate is applied to the variable  $y$ . However, *COMP* is applied to both  $x$  and  $y$  to create a pair of the two variables. Operation variables are specified in the *agvar* attribute in an alist.
4. **Auxiliary Variables:** These are additional variables that are created at runtime by rules during alist decomposition. They are also prefixed with the \$ symbol. The goal of a decomposition rule is to modify an alist such that the new alist provides a different approach to get the same (or similar) answer as the alist that failed to do so. Some of the rules generate aggregate functions from which the required answer can be inferred. In order to do so, these need to generate new variables for the parameters that the function will take. This is also another reason for not restricting alists to specific lengths; that is, it gives freedom to the framework to explore different rules by dynamically adding new variables as needed by aggregate functions. For instance, given the alist

```
{ag:VALUE, subj:uk, prag:population, obj:?x, time:2019},
```

temporal decomposition could lead to the creation of the child alist

```
{ag:REGRESS, subj:uk, prag:population, obj:?x, time:2019, fn1:$fn1}
```

where the variable  $\$fn1$  is dynamically created for the new key  $fn$  that will take the function created by the REGRESS (regression) aggregate as its value.

Although query, auxiliary and aggregate variables are expressed with the same prefix \$, their roles are recognized by their use in the FRANK query and alists. Query variables are those explicitly expressed in the user query, auxiliary variables are those not found in the initial user query, and aggregate variables are identified by the value of the “ag” attribute in the alist.

#### 4.4.4 Variable Types

FRANK uses a very basic type inference mechanism. It uses four types: (1) *real*, (2) *string*, (3) *boolean* and (4) *datetime*. However, variables are not assigned types by users during the definition of a query. These types are inferred by the aggregates at runtime. For instance, the *VALUES* aggregate returns a list of objects of the above types. The data type of a variable is determined by the type of the value in the alist attribute that is returned by the child nodes. With the exception of the time and uncertainty attributes of the alist, all other attributes of the alist are, by default, of *string* type. When possible, we convert variables to real types if mathematical aggregates are to be applied. The *time* attribute has *datetime* type and *uncertainty* is a *real*.

The aggregate to be applied at a node in the FRANK tree also determines the type of the alist attribute needed for inference. For instance, if the *AVG* aggregate is to be applied at the node and the aggregate variable required from its child node is ?y, then all ?y variables of the child nodes are expected to be a real numbers. During aggregation, the values of y are cast to real before the arithmetic computation. If any of the variables are not real numbers, then depending on the policy for upward propagation, that variable is either simply omitted, or the entire aggregation at the node fails.

The primary reason for this approach to handling types is to accommodate the heterogeneity of KBs and their representation. FRANK has no guarantees that the results of a query that is run on different KBs to return values of the same type.

#### 4.4.5 Queries as Root Nodes in FRANK Tree

To answer a query, FRANK internalizes the query string by creating an alist and labelling the node with the alist as shown in Figure 4.5. This node forms the root of the

FRANK tree. In the case where the query contains nested subqueries, the subqueries are embedded as either the *subject* or *object* depending on which variable needs to be found. During internalization, nodes that contain subqueries in an attribute of their alist will be decomposed to simplify them. Only when all alists are in FRANK Normal Form does FRANK begin the process of instantiating variables and inferring answers. Internalization allows not only the root alist of the FRANK tree to be specified, but also the first few levels of the FRANK tree.

## 4.5 Comparing FRANK and SPARQL Queries

Although FRANK and SPARQL queries share some conceptual similarities in terms of the pattern matching required to instantiate variables in queries, there are significant differences.

### 4.5.1 Similarities between FRANK and SPARQL Queries

1. When querying RDF KBs, FRANK alists, at the leaves of the tree, reduce to SPARQL queries. That is, FRANK uses templates to compose SPARQL queries that retrieve values from the KB.
2. Some FRANK queries are logically equivalent to SPARQL queries. Below is an example of a question with logically equivalent queries for FRANK and SPARQL: *Which country in Africa has the largest population?.* FRANK query:

```
MAX(?x,
  COMP(<?x,$y>, population(?x,$y):
    Country($y) & location($y,Africa))
)
```

An equivalent well-formed SPARQL query that returns a valid result is:

```
SELECT DISTINCT ?x
WHERE {
  ?xVar a <http://dbpedia.org/class/yago/WikicatAfricanCountries> .
  ?xVar rdfs:label ?x .
  ?xvar <http://dbpedia.org/ontology/populationTotal> ?y .
}
ORDER BY DESC (?y) LIMIT 1
```

## 4.5.2 Differences Between FRANK and SPARQL Queries

1. Although for RDF KBs, FRANK queries are logically equivalent to SPARQL ones, in cases where FRANK queries include operations such as REGRESS (Polynomial Regression) or GP (Gaussian Process Regression), no well-formed SPARQL queries can be composed from the operations currently available in SPARQL that will successfully return results from the RDF KB. Although we could extend SPARQL to include such mathematical and statistical operations, vanilla SPARQL was not built to perform such higher-order operations. Even if we used the *SERVICE*<sup>5</sup> keyword to invoke the federated query processor in SPARQL, SPARQL is still not equipped to perform the automatic query decompositions (e.g. temporal and geospatial decomposition) as FRANK does.

Whereas native SPARQL is built for RDF KBs and so only retrieves data from RDF KBs (even for federated search queries (section 3.6.1)), FRANK can query data from KBs with a variety of knowledge representations provided the KB wrappers (sections 7.4 and 7.6 ) for the KBs of interest are provided. These include RDF, relational DBs, Javascript Object Notation (JSON), REST APIs, etc.

2. SPARQL has been created as a query engine for retrieving stored facts and performing logical inferences on the stored statements. FRANK, on the other hand, is meant to be used in an inference engine that uses other query engines to look up data from KBs. These include SPARQL, SQL, REST<sup>6</sup> APIs (Fielding and Taylor (2000, 2002)), etc. to retrieve facts from KBs to be used in the FRANK tree to infer data that is not possible to infer using SPARQL or SQL alone.
3. FRANK combines its string matching with lookup engines such as SPARQL to retrieve data from RDF KBs. Given that we cannot rely on Reified RDF, it is not a simple task to compose a single query that can answer the user's question in one go. Supplementary queries will have to be created to find additional data, aggregate and compose the appropriate answer. In FRANK, we determine these supplementary queries through the automatic decomposition of alists. Since FRANK systematically queries KBs to instantiate all child alists, the required queries are run on the KBs, effectively resulting in a compositional

---

<sup>5</sup><https://www.w3.org/TR/sparql11-federated-query/>

<sup>6</sup>Representational State Transfer

Table 4.1: FRANK Aggregates

Operation	Description
VALUE	Default aggregate. Identity function on the value of a node
SUM	Add values of nodes of numeric type
AVG	Mean value of child nodes
MEDIAN	Median value of child nodes
MAX	Maximum value of child nodes
MIN	Minimum value of child nodes
COMP	Obtain a set by list comprehension
GT	'Greater Than' function to compare two nodes
LT	'Less Than' function to compare two nodes
EQ	Check if the value of two nodes are equal
REGRESS	Regression function from child nodes
SOLVE	Solve a function by passing in parameters
INTEGRAL	Find the integral of a function
DERIV	Find the derivative of a function

query. SPARQL on the other hand does not perform such decompositions at run-time and, therefore, depends on the full, explicit query being composed to find the exact answer required.

## 4.6 Decomposition Rules and Aggregate Functions

Decomposition rules and aggregate functions are the crucial components that distinguish FRANK from other QA systems. They determine how the FRANK tree is expanded and reduced back to the root in order to answer a query.

### 4.6.1 Decomposition Rules

Decomposition rules determine how an alist in the FRANK tree is decomposed to create child alists. This is an important part of the inference mechanism in FRANK. For instance, when a node contains an entity that is geospatial in nature (e.g. the name of a continent) in its alist, if that node fails after a KB search, one intuitive strategy that a human expert can take is to look at the constituent parts of the geospatial entity,

solve the task for the sub-components and then combine these to infer a solution for original entity. In this example, the strategy will be to solve the task for each *country* in the *continent* and then combine them to solve the original query about the continent.

In FRANK, we achieve this with a set of rules that perform a variety of decompositions. Strategies used in FRANK include: the *temporal* rule, to decompose nodes by date/time features; *geospatial* rule, to decompose nodes by location; and the *lookup* rule, to create child nodes with synonyms of the attributes of the parent, to increase the chances of finding facts in KB. These are discussed in greater detail in chapter 5.

### 4.6.2 Aggregate Functions

Aggregate functions (aggregates) specify the operations to be performed on a FRANK node when combining its children. An aggregate function, first extracts relevant values from the alists of its child nodes to use as inputs and then applies the function associated with the operation. The aggregate function then substitutes the inferred value into the respective alist attributes and returns the complete alist. Aggregates used in FRANK are listed in Table 4.1. Aggregate functions are also used to specify arithmetic and statistical operation in queries.

## 4.7 The FRANK Algorithm

FRANK finds answers to queries by recursively decomposing and aggregating alists until projection variables in the query alist are instantiated in order to return a value to the user. This means that alists are decomposed until variables (query, projection and/or auxiliary variables) in a leaf node are instantiated to facts in a KB. An attempt is then made to up-propagate the instantiated variables up the FRANK tree. The FRANK tree propagates the instantiated variables in alists upwards through the FRANK tree, aggregating values and uncertainty values through nodes to the root of the tree. If this process terminates successfully, then the alist in the root node will contain the answer to the query. The pseudocode of the algorithms are listed in Algorithms 1 to 4. The key aspects of this algorithm are discussed below.

### 4.7.1 Query Internalization

*Internalization* is the process of creating an alist from the query string. The framework currently takes as inputs queries composed using the FRANK grammar discussed in



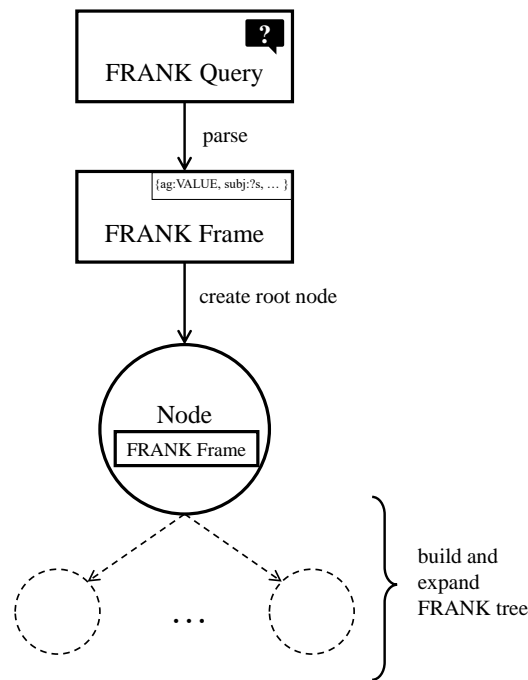


Figure 4.5: Internalization: Converting a FRANK query to an inference node.

Section 4.4.1. Although the preferred mode of expressing queries is in natural language text, we leave that as further work that can be undertaken to make FRANK more user-friendly. We focus, instead, on expressing queries in a formalism that is sufficient to express the range of queries we expect, and is also easy to internalize as alists.

We obtain an alist from the query by parsing the query string using the FRANK grammar. We focus on the outermost aggregate function and identify its *agvar*, property, subject, object, and time variables. These attributes could be variables, literals or compound expressions (logical expression or subquery).

## 4.7.2 Initializing the FRANK Tree

The core FRANK algorithm runs on the FRANK tree. To build the tree, we create the root node of the tree from the alist obtained in the internalization step. The FRANK node adds on additional query-specific housekeeping records to manage the traversal of the FRANK tree. These include node identifiers and pointers to parent nodes.

For nodes that contain compound alists, we extend the FRANK tree further by normalizing the nodes recursively. That is, for each given alist, we pick an attribute with a compound alist and simplify it with the *NORMALIZE* decomposition rule. This rule

---

**Algorithm 1** FRANK

---

```

1: procedure FRANKINIT(query, KB)
2:   rootNode ← INTERNALIZE(query)
3:   answerNode ← FRANKEXECUTE(rootNode, KBs)
4:   return answerNode
5:
6: procedure FRANKEXECUTE(node, KBs)
7:   facts ← {}
8:   facts ← LOOKUPCACHE(node)
9:   if facts is empty then
10:    facts ← LOOKUPKB(node, KBs)
11:    kbPriorCov ← UPDATECOV(facts)
12:    localCache ← UPDATECACHE(facts, KBs)
13:   if facts is empty and depthLimit not exceeded then
14:    rankedCandidateRules ← GETSTRATEGY(node)
15:    childNodes ← DECOMPOSE(node, rankedCandidateRules)
16:    for each child in childNodes do
17:      FRANKEXECUTE(child, KBs)
18:   else
19:     if hasParent(node) is true then
20:       UPPROPAGATE(parent(node), children(parent(node)))
21:       return parent(node)
22:     else
23:       return node

```

---

simplifies an alist by evaluating logical statements and returning new alists with the results of the evaluation. *NORMALIZE* also reduces nested alists in a query to individual alists with a corresponding hierarchical structure (see Chapter 5). This continues recursively until all nodes are in FRANK Normal Form.

Other activities performed during this phase include the setting of global properties of the inference system such as the tree depth limit. The depth limit specifies the maximum depth allowed for any node in the tree beyond which that node should not be decomposed further.

### 4.7.3 Search and Grounding

FRANK only attempts to ground simple alists of leaf nodes in the FRANK tree. Grounding is performed by searching knowledge bases for relevant data. FRANK uses two kinds of knowledge bases: (1) a local data cache, and (2) remote knowledge bases. The local cache is used to store data that has recently been retrieved from remote knowledge bases and used to instantiate a variable in an alist. This makes it faster to access data that could to be reused elsewhere in the FRANK tree by avoiding the

---

**Algorithm 2** Strategy Selection (SS)

---

```

1: procedure GETSTRATEGY(node)
2:   if ISFRANKNORMALFORM(node) is false then
3:     return RULE.normalize
4:   else
5:     ruleWithCost  $\leftarrow$  MAP(COSTFN, {rule.set})
6:     orderedRules  $\leftarrow$  ORDERBY(ruleWithCost, cost, ASCENDING)
7:     return orderedRules

```

---



---

**Algorithm 3** Downward Decomposition (DD)

---

```

1: procedure DECOMPOSE(node, candidateRules)
2:   leafNodes  $\leftarrow$  {}
3:   for each rule in candidateRules do
4:     strategyNode  $\leftarrow$  COPY(node)
5:     ADDORBRANCH(node, rule)
6:     andNodes  $\leftarrow$  EXPAND(strategyNode)
7:     ADDANDBRANCH(strategyNodes, andNodes)
8:     leafNodes.append(andNodes)
9:   return leafNodes

```

---



---

**Algorithm 4** Upward Propagation (UP)

---

```

1: procedure UPPROPAGATE(parentNode, childNodes)
2:   parentNode.variables  $\leftarrow$  REDUCE(parentNode.aggregate, childNodes)
3:   varianceFunction  $\leftarrow$  COMBINEGAUSSIANVARIANCE(parentNode.aggregate)
4:   parentNode.cov  $\leftarrow$  UPDATECOV(varianceFunction, parentNode, childNodes)
5:   return parentNode

```

---

communication overhead of requesting the same data again from its source KB.

The search processes (LOOKUPCACHE() and LOOKUPKB() in Algorithm 1) attempt to find facts that match the subject, predicate, object [and/or time] of the alists being processed, and instantiating the query, projection and/or auxiliary variables in the alist to the corresponding attribute in the triple found in the knowledge base. The decomposition for this is the LOOKUP rule and is discussed in further detail in Chapter 5. This rule handles searches for data triples that are synonymous with the query alist of the node being grounded. That is, the rule generates queries using synonyms of the string literals in the alist.

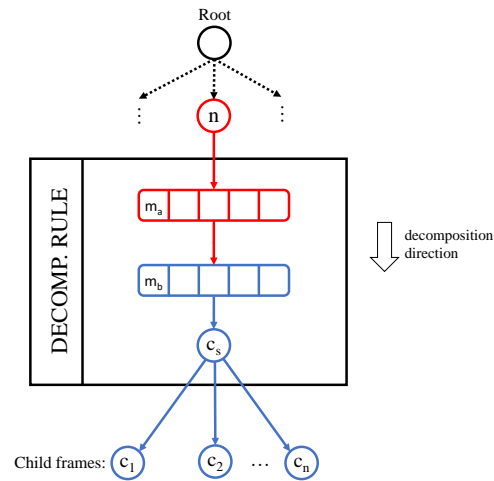


Figure 4.6: Downward Decomposition

First part of decomposition transforms the alist of the node to be decomposed,  $n$ , to  $c_s$  by selecting the appropriate aggregate function  $m_b$  to replace  $m_a$ , given the rule,  $s$ . Second part of decomposition generates the child nodes,  $c_i$  based on  $s$ .

#### 4.7.4 Expanding the FRANK Tree

When FRANK processes simple queries for which triples in the KB match the search query, the FRANK tree generated from the parse of the query is usually sufficient to answer the query. However, this is rarely the case for many user queries as these queries are often so complex that simple triples in the KB alone are insufficient to answer the query. In such instances, the FRANK nodes created from the query have to be decomposed (Figure 4.6) in such a way that the new leaf nodes can be grounded. This process can be recursed several times in FRANK until an answer is found, or until a pre-defined tree depth limit is reached on all branches.

DECOMPOSE() in line 15 of Algorithm 1, with its definition in Algorithm 3, shows how this decomposition happens. Candidate rules create OR branches from the current FRANK node. These new OR branches then create new AND branches for the decomposition that results from each candidate rule.

Consider, for instance this query: “*What will be the total population of Europe in 2023?*”. No data in any of the knowledge bases matches the underlying triple to answer this query. In this case, FRANK identified 2023 as a future year (given the system date obtained from server on which FRANK is executed), and hence recognizes this

as a prediction task. To proceed with an attempt to find an answer, FRANK can select the temporal decomposition rule, for instance, and create a new node that specifies a *FUNCTION* aggregate from which the answer can be retrieved. The *FUNCTION* aggregate can choose from several options such as Linear Regression, Polynomial Regression or a Gaussian Process. Assuming it selects a Linear Regression, it would create new child nodes with each child node having similar attributes in its alist but a different year in the past.

The decomposition of each node is based on the selected rule (`GETSTRATEGY()` in Algorithms 1 and 2). The rule annotates the node with a new aggregate function and links the new node to the node being processed. This is the downward direction of inference. The aggregate function indicates how the child nodes of the decomposed node will be combined when the variables in the child nodes are grounded. This happens in the upward direction of inference.

The recursive nature of the FRANK algorithm (line 17 of Algorithm 1) means that each leaf node is decomposable, either to simplify a node with a complex alist, or to apply a decomposition rule given the observed features in the node's alist. Regardless of the kind of decomposition that happens, the FRANK tree gets extended.

The expansion of the FRANK tree is usually a more expensive process than the upward propagation of nodes. This is because the expansion of the tree itself requires some form of inference and search from knowledge bases such as commonsense KBs and geographic KBs. For instance in the query example above, it may also be necessary to decompose the query using the geospatial rule. This requires identifying that Europe is a place (continent) that can be partitioned into sub-components (i.e. countries). We can then go ahead and search for the countries in Europe and create sub-nodes in the geospatial branch from the node under evaluation.

#### 4.7.5 Complexity of the FRANK algorithm

A precise specification of the complexity of the FRANK algorithm is difficult to determine in this work, given the very large, open-ended and constantly changing space of facts for instantiating variables in alists. Further, the nature of the decompositions, also open-ended (since new decomposition rules could be added), could rapidly increase the growth of the inference tree. In any case, we try to highlight the factors that influence the complexity of the inference mechanisms.

FRANK is at least *NP-Hard* for queries requiring very minimal decomposition

without inferring functions since it reduces to a *FO* Fagin (1974) problem with the additional task of aggregating the instantiated variables rather than returning just the instantiations. *FO* is a complexity class of structures associated with formulae of first-order logic with variables that range over elements of the structure. The *structure* refers to a set built from elements of a domain with specified functions and relations between the elements. Since an alist encodes a function over a  $\langle \text{subject}, \text{property}, \text{object}, \text{time} \rangle$  relation, with one of subject, object, time or any other object-level attributes being a variable at the leaf of the FRANK tree, the task of instantiating variables in alists using data from KBs is equivalent to that of first order queries (e.g. in relational algebra). Generally, though, FRANK tends to be worse than *FO* given that FRANK performs second-order operations when it infers and up-propagates functions during inference for prediction.

The execution time of RIFEXECUTE (in algorithm 1) has exponential growth in the number of alist variables, decomposition rules and child nodes resulting from the recursive decomposition of the query node. GETSTRATEGY (in algorithm 2) is linear in the number of strategies(rules) since it returns applicable rules with their estimated costs used as heuristic for prioritising rules for alist decomposition.

DECOMPOSE (in algorithm 3) (using the lookup, temporal and geospatial decomposition rules) results in an exponential growth in the number of nodes if applied multiple nested times. For non-trivial decomposition rules, DECOMPOSE is dominated by the complexity of the decomposition operation. For example, geospatial decomposition (in section 5.3.7) is an inference operation that finds the components of an entity and is therefore exponential in nature when applied multiple time. Using the Geonames ontology (Vatant and Wick (2012)) (containing over 10,000,000 geographical names), a fact about the ‘*world*’ can be inferred by decomposing the *world* into continents, country, multi-level administrative regions, cities, etc. Following each of these geospatial decompositions, other rules can be applied on each new node resulting in hundreds of new nodes in the tree. That is, the decomposition rules, in their own right, reduce to first-order problems to instantiate the appropriate variables in the alist using facts from KBs to generate new child nodes.

The complexity of UPPROPAGATE (in algorithm 4) is dependent on the complexities of the aggregate function in REDUCE and the UPDATECOV operation. Aggregates for comparing alist values (e.g. GT, LT) are performed in  $O(1)$  time, others such as SUM, AVG and COUNT are  $O(n)$  and more complex aggregates such as Gaussian Process regression are  $O(n^3)$  (Quiñonero-Candela and Rasmussen (2005)), where  $n$  is

the number of child nodes being aggregated. UPDATECOV runs in polynomial time of the number of knowledge bases and the number of data values retrieved from each KB. This is discussed further in section 6.9. Generally, UPPROPAGATE has complexity that could be worse than FO since arithmetic functions can be generated from the aggregation of the values of child alists and passed up to their parents. This ability of FRANK's alists to return not just values, but functions, results in higher-order operations.

### 4.7.6 Uncertainty

Our approach to inferring answers to queries encounters two main forms of uncertainty. First, uncertainty arises due to lack of trust of a knowledge source given variations in data observed there compared to other sources of the same data. Secondly, uncertainty is introduced when aggregate functions that perform complex functions over alists of inference nodes are applied. Our representation of uncertainty in FRANK is the Gaussian distribution,  $\mathcal{N}(\mu, \sigma^2)$ , where  $\mu$  is the mean value observed or inferred, and  $\sigma^2$  is the variance around that mean. This is based on our assumption, given the real-valued data we work with, that the true value of a fact has Gaussian noise and we discuss this in section 6.4. We primarily focus on the variance and normalize it with the corresponding mean value as a Coefficient of Variation (*cov*). That is,

$$cov = \frac{\sigma}{\mu}$$

Our approach to uncertainty currently focuses on queries with real-valued answers. A more detailed discussion of the types of uncertainty and how FRANK deals with them is documented in Chapter 6.

### 4.7.7 Upward Propagation of Facts and Uncertainty

Node decomposition during inference expands the frontier of the FRANK tree. However, when a leaf node of the tree gets grounded, the process of upward propagation starts when the node's siblings are grounded as well. To obtain an answer from the grounded facts, FRANK propagates the alists of nodes upwards to their parent node (Figure 4.7). This is repeated recursively all the way to the root of the FRANK tree in lines 19 to 22 of Algorithm 1. The decomposition rule that specifies the decomposition policy also inserts the aggregate function into the *aggregate* attribute of the node's alist. During upward propagation, the child nodes from the leaf node are combined according to the function specified by the aggregate function in the parent node.

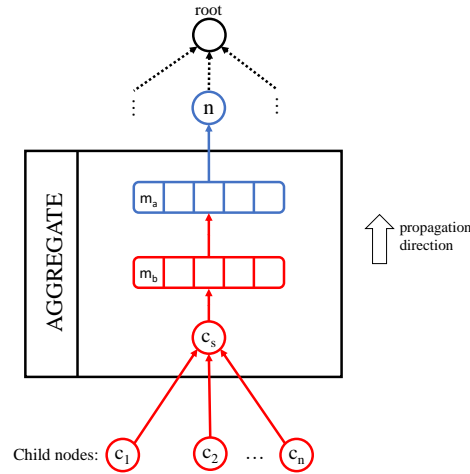


Figure 4.7: Upward Propagation

Instantiated variables in alists of  $c_1 \dots c_n$  are combined using functions specified by aggregate  $m_b$  in their parent node,  $c_s$ .  $C_s$  then updates alist of  $n$  with the newly instantiated variables. Propagation continues until the root node is reached.

For instance, a node with aggregate *AVG* will find the average value of the aggregate variable of all its child nodes.

FRANK also determines the uncertainty in the inferred answer by propagating the uncertainty from the leaf nodes to the root. To capture the uncertainty in the final answer, FRANK propagates these variances through the FRANK tree by combining them using the corresponding aggregates of the aggregate function. For instance, if  $c_1, c_2, \dots, c_n$  have real-valued *agvars*, have variances  $\sigma_1^2, \sigma_2^2, \dots, \sigma_n^2$  (obtained from their *covs*) respectively, and the aggregate  $o_b$  is *SUM*, then

$$\sigma_{c_1+c_2+\dots+c_n}^2 = \sigma_1^2 + \sigma_2^2 + \dots + \sigma_n^2 \quad (4.1)$$

where we assume conditional independence of the  $c_i$ 's. This process of calculating and propagating uncertainty is discussed extensively in Chapter 6.

## 4.8 Summary

This chapter described the Functional Reasoning for Acquiring Novel Knowledge (FRANK) and how it works, the core contribution resulting from this project. The



objective of FRANK is to provide a system for inferring answers to queries when the required data that answers the query is not stored in the available knowledge bases. In FRANK, we use alists to represent facts in the FRANK tree. Internally, an alist is a list of key-value pairs similar to dictionaries in other KBs such as Java and Python. Alists in FRANK are an extension to the triple representation with the goal of making it easy to add new attributes to a node in the FRANK tree. That is, alists go beyond the  $\langle \text{subject, predicate, object} \rangle$  triple representation used in relational databases and RDF-based KBs in the Linked Data and the Semantic Web.

FRANK works by using decomposition rules to determine if and how a node can be decomposed such that the new alists can be grounded in facts in a KB. Heuristics determine the selection of rules for a given node. Features in the alist of a node such as the types of entities, temporal and geo-spatial properties determine the kinds of decompositions that can be used. Decompositions result in new nodes being generated. The new nodes are connected to the decomposed node as its descendants.

We also described how queries are formed using a query grammar for FRANK. The grammar allows queries to specify functional components that describe the kinds of aggregates that the users wants performed to find an answer. The functional components are labelled by the same aggregate functions that are used in the FRANK tree. Queries also contain logical components that allow entities to be described in the query. Multiple propositions can be used to describe an entity and these can be combined conjunctively or as a disjunction. Negations are not used in FRANK due to the difficulty in reasoning with negation under the open world assumption. These are, however, tasks that could be looked at in future work. Together, the functional and logical components are used to generate a variety of queries. The grammar also offers the nesting of queries such that the aggregate functions in the functional parts are composed to create sub-queries

This chapter also defined the types of variables that are used when composing queries as well as in the FRANK tree. Variables in FRANK are used in four ways: (1) query variables that are specified in the user's query as items that need to be bound to actual data in KBs to answer the query; (2) projection variables that determine which of the unknowns should be returned to the user as the answer to the query; (3) aggregate variable that determine which attribute(s) of the alist an aggregate function should be applied to; and (4) auxiliary variables that are dynamically created by FRANK to accommodate the needs of specific aggregate functions.

We also explained the inference algorithm and its primary components. Inference

begins with the internalization of the user query, a process that converts a query into an alist. A FRANK tree is constructed by creating a graph node from the alist. To answer the query, the objective of the algorithm is to instantiate the query variables. For each node, a search query is run against the KBs to find items that match the unknowns using other literals in the alist as conditions for selecting the right values from the KBs. If search fails, a decomposition rule is automatically selected and it determines how the node should be decomposed to find an alternative way to infer the answer. This process expands the FRANK tree by creating child nodes that also need to be processed. When leaf nodes are successfully grounded, the process of upward propagation begins. The values of variables instantiated in the leaf node are propagated upward to their parent node. The parent node uses its aggregate function to reduce the values of variables from its children and also grounds any variables that it can instantiate given the data it received from its child nodes. This process continues recursively until the root node is reached, at which point an answer can be returned to the user. We also briefly discussed how uncertainty affects the returned answer and how FRANK calculates and propagates it in the FRANK tree.



# Chapter 5

## Inference Operations

### 5.1 Introduction

FRANK is built on the idea of solving complex problems by composing the required solutions to sub-problems from several smaller operations (programs or algorithms) recursively. We assume that for any answer to a query, there is a finite sequence of operations that can be applied to data from knowledge bases to obtain the required answer. This chapter focuses on these operations and how they work.

We look at the second aspect of our hypothesis:

**HYP-2:** *Rich forms of inference lead to an extension of the types of questions possible.*

We take a look at some of the inference operations that constitute *rich inference* in this work from which we are able to answer novel types of questions.

Inference operations are central to the successful working of FRANK. In FRANK, inference operations determine how the tree is explored to find answers to queries. That is, inference operations perform a dual role. First, they provide rules that determine how alists in the FRANK tree are decomposed in order to expand the frontier of the search space towards the grounding of alists in KBs. We call these operations *decomposition rules*. Secondly, the operations are executed on data using the appropriate arithmetic or statistical algorithms, based on inference operations and other attributes in the FRANK alists, to aggregate the relevant instantiated variable in alists and then propagate them up the FRANK tree to the root. We call these operations *aggregate functions* (or simply *aggregates*).

In the sections that follow, we look at the different kinds of rules used during the decomposition of alists. We also describe the corresponding operations that label the

alists for use during the upward propagation phase of inference when these operations are executed on alists. We also discuss how FRANK estimates the costs of these inference methods and how these costs are used as heuristics in the best-first search traversal of the FRANK tree.

## 5.2 What is an Inference Operation?

### 5.2.1 Definition

An inference operation is a process that (1) determines how a FRANK alist should be decomposed based on a variable of interest, and (2) specifies how the child alists of the decomposed alist should be combined in order to instantiate variables in the parent alist. These terminologies are similar to the map-reduce operations found in batch data processing architectures Dean and Ghemawat (2008), but very different in their use in FRANK.

Operations in FRANK are glued together by the alist data structure in nodes of the FRANK tree. That is, operations take alists as inputs and return alists. Operations, therefore, transform values of alists attributes and propagate them to their children or parent depending on the type of operation being performed.

### 5.2.2 Duality of Inference Operations

Inference operations perform two related processes during the key phases of inference in FRANK. First, during the downward decomposition of alists, we need to guide the search through the FRANK tree in order to manage the combinatorial explosion problems in the search tree. This map phase of inference tends to be expensive since we are inferring classes or data types of named entities and how they can be decomposed given facts from KBs. The aim of this first part is to ground variables in alists to match data from KBs. We refer to the operations here as *decomposition rules*.

The second part of inference is the upward propagation of variables in alists once these variables have been instantiated. This involves reducing the values of variables in child alists and passing them on up to their parent alists. During this phase, we tend to use inference over actual data, often real-valued data, such that we propagate aggregated values to parent alists. Here, we resort to arithmetic or statistical techniques that calculate values from raw facts or from functions created from observed data. We call these operations, *aggregate functions* (or *aggregates*).

Additionally, the uncertainty values of alists are propagated up the tree to the root. The objective is to assign an uncertainty value to the answer in the root alist of the FRANK tree that reflects the uncertainty in the data retrieved from the sources as well as the errors introduced by the aggregate functions applied at various level in tree during upward propagation. The subject of uncertainty and its propagation is discussed in chapter 6.

Although these two phases perform seemingly separate inference tasks, they are two sides of the same coin. That is, the inference operations applied during decomposition determine the kind(s) of arithmetic or statistical aggregation that can be performed on the child nodes during the upward propagation phase.

### 5.2.3 Notation

To simplify the definitions of decomposition rules and upward propagation aggregates, we use the following notation and assumptions.

- $f[A] = a$  represents the assignment of the value  $a$  to the attribute  $A$  of alist  $f$ .
- For an alist  $f$  with aggregate,  $h$ , and aggregate variable,  $v$ , we compactly represent the alist as  $f_{h(v)}$  to emphasize the function application aspect of an alist.
- We assume that a non-leaf alist  $f$  has child alists  $g_i$ , where  $i \in \{1, 2, \dots, n\}$  are indices of the child alists.
- We express a unary function in the form  $\lambda x.function(x)$ . These represent either:
  - mathematical operations on a variable  $x$  expressed as  $\lambda x.operation(x)$ , or
  - logical expressions with an unknown variable,  $x$ , defined as  $\lambda x.property(S, x, T)$ . The unknown variable,  $x$ , is commonly the value of the *object* for *property* of the subject  $S$  at an optional time  $T$ . When time is optional, the expression returns true for when the property has an object  $x$  regardless of the time specified in the KB, or in the absence of a time altogether.
- For brevity, we express the unary function  $\lambda x.function(x)$  simply as  $z$ .
- We use the notation  $\Delta_d(f) \mapsto \{g_i[a_1, \dots, a_j] \mid i, j > 0\}$  to indicate that a decomposition rule  $\Delta_d$  applied to a parent alist  $f$  will yield a new set of child alists  $g_i$  with attributes  $a_j$ .

- We only highlight the attributes and their new assignments when they are modified by a decomposition rule since all other attributes of the alist remain the same. For example  $g[time = 2011, ag = VALUE]$  means that the time attribute of alist  $g_j$  has been modified to 2011, and the aggregate attribute has also been changed to VALUE.

## 5.3 Alist Decomposition Inference Operations

The first phase of inference in FRANK is the downward decomposition of the root alist to ground its query variable on data in a KB. The decomposition task, therefore, requires some guidance to prune out irrelevant rules so as to avoid the unnecessary explosion of the FRANK search tree and to prioritize the rules that remain.

### 5.3.1 Relevance of Heuristics

In QA, the search spaces tend to get very large. Heuristics are, hence, vital to making the problem tractable. For instance, consider a query to predict the country with the highest cereal production of Africa in 2022. Suppose we use three strategies to (1) lookup the answer (considering synonyms of the various search terms (cereal production, location names, etc)), (2) search for answers for each country and then aggregate for Africa, or (3) search for values for previous years and extrapolate into 2022. Given that (a) there are officially 54 countries in Africa (according to the United Nations<sup>1</sup>), (b) we limit the temporal branching to 20 (since we need past years' production values to predict), and (c) we assume about 3 synonyms for country names, we have about 10000 nodes at the leaves of the search tree at a depth of 4. This increases exponentially as the nodes are decomposed further. Our approach to QA uses reasoning techniques that adopt heuristics that choose between the decomposition rules that can solve sub-problems of the main problem, and aggregate the sub-solutions to give an answer to the user.

Our first use of heuristics is to make the search space manageable by pruning inappropriate rules during decomposition. FRANK use external KBs to provide the required guidance during decompositions. Although these KBs (e.g. ConceptNet (Liu and Singh (2004))) tend to be noisy because they are crowdsourced, they provide useful facts for FRANK to determine types (or classes) of entities that influence the kind of decomposition it performs, particularly for values of the subject attribute in an alist.

---

<sup>1</sup><http://www.un.org/en/member-states/>

Our second use of heuristics is to prioritise the decomposition rules and the appropriate functions to answer the query. Attributes in the FRANK alist determine if, for instance, a temporal and (or) geospatial decomposition of the question is more appropriate. This creates a sequence of strategies that can be used to find a solution. Multiple strategies can be pursued in the OR branches of the FRANK search tree, and their final answers and uncertainties combined.

### 5.3.2 Heuristics for FRANK

Heuristics play a key role in FRANK. Consider the query:

```
VALUE(?y, urban_population(uk, ?y, 2022))
```

that translates to the natural language query “*What will be the urban population of the UK in 2022.*” First, given that the temporal period of the answer required is in the future “2022”, the data required will not exist in a KB (unless it has been previously been estimated and stored in a KB.) A possible strategy towards answering the query will be to retrieve urban population data for the UK for past years and then extrapolate to estimate the required data for 2022.

Alternatively, we could determine from DBPedia or Conceptnet that the UK is a geo-political entity that can be partitioned into four other entities: England, Scotland, Wales and Northern Ireland. We could try to solve the same query for these ‘sub-components’ of the UK and then combine them to solve for the whole. Although both strategies will work, the kind of data observed from available KBs will determine which one will yield an answer first. For instance, if facts exist for the urban population of the UK for previous years and none exist for the country in the UK, then the temporal decomposition will result in fewer decompositions before variables are instantiated, compared to the geospatial decomposition. FRANK takes a simplistic view of these strategies when prioritising them, for instance, the presence or absence of values for certain attributes in the alist. In reality, there are several factors that could determine which decomposition rule is more appropriate. For instance, in the above example, knowing the scope of census data collection in the UK is required. Such a determination is outside the current abilities of FRANK. In terms of relevance, heuristics allows FRANK to focus on just these two strategies and ignore others. Heuristics also ranks these two such that the cheaper one is explored before the other.

In FRANK, we have implemented the above strategies as the *temporal* and *geospatial* rules respectively. We use external knowledge bases to infer types(classes) of en-



tities, to determine if the entity has sub-components and if it can be partitioned. We discuss these in the next sections.

### 5.3.3 Decomposing an Alist

Alist decomposition is the process of creating new child alists from an alist,  $f$ , by transforming attributes values and variables in  $f$ . A decomposition is triggered whenever a LOOKUP fails to return any results after searching the KBs available to FRANK. The objective of decomposing an alist is to increase the chances of inferring the required data when the KBs are searched.

Four decomposition rules determine how we extend and explore the FRANK tree. These are (1) lookup decomposition (2) alist simplification (3) temporal decomposition and (4) geospatial decomposition.

### 5.3.4 Lookup Decompositions

Lookup decomposition is the base case rule for finding matching facts from knowledge bases. This is required to deal with situations where the text labels of subjects, objects or predicates in the query does not match the exact text labels of any constants in searched KBs. One of key the problems that arises in open-domain query answering using web data is that the same concept can be expressed in multiple ways. The most common is the use of related words or phrases (e.g. *UnitedKingdomOfGreatBritain*, *united\_kingdom*, *UK*) to represent the same concept in different KBs. It is also worth noting the variations in casing (e.g. *snake\_casing* with underscores and *CamelCasing*) to separate multi-worded terms.

For instance, given the alist:

```
VALUE($y,population(Gold_Coast,$y,1930))
```

a search though the KBs would probably not find any matches since the subject used in modern KBs would be *Ghana* instead of *Gold\_Coast*, its name before independence. Similarly, the property *population* may find matches in some KBs, but not all. In the World Bank dataset, the population property that we require is stored as “*total\_population*”. The matching process uses a combination of string functions to split terms separated by hyphens or underscores, or joined in *camelCase* form; language resources such as WordNet (Miller (1995)) to find word synonyms; and edit distance measures to find matches to predicates available in a KB.

Since, in our work, the inference problem dominates the string matching problem, we focus, primarily, on the task of inference in FRANK. However, to enable access to data from KBs, we use basic syntactic matching (primarily, exact string matching) of the terms in the alist to the KB being looked up. In our experiments, this has proven sufficient to find the data from the KBs provided the property attribute of the query alist closely matches properties in the KB. It is worth noting, that for some KBs used in this work, exact string matches for properties results in failure since property terms tend to be noisy. Most terms are appended with text to further describe the property or specify the units of the quantities recorded. For example the term *internet coverage* matches *Coverage: Internet*, where as *fertility rate* matches *Fertility rate, total (births per woman)* Also, given that FRANK does not support natural language queries and so property names in queries are separated by underscores instead of spaces, exact string matching fails for the majority of queries. This is, somewhat, restrictive and so a more sophisticated entity matcher that uses state of the art techniques is required in the future.

Experiments discussed in (Cohen et al. (2003); Ukkonen (1985); Bilenko et al. (2003)) show that the Jaro-Winkler (Winkler (1999)) and Levenshtein (Levenshtein (1966)) similarity measures have proved useful in syntactic string matching. These experiments focus on matching strings for linking records in KBs or databases, a task that is similar to our goal for matching terms in FRANK to terms in the KB. Both are readily usable in FRANK, given several off-the-shelf implementations available. We use the implementations in the Apache Commons Text library<sup>2</sup> in our work. We tested both algorithms by matching 50 search terms to over 15,000 properties in the World Bank dataset<sup>3</sup> (WB) and Wikidata<sup>4</sup> (WD). We observed that the Jaro-Winkler and Levenshtein algorithms performed similarly and returned the same results from the knowledge bases in most cases. However, in two of the test examples, we found that the Jaro-Winkler returned a better example because of its preference for strings that match from the beginning.

Given the search term '*renewable\_energy*', the Jaro-Winkler found the best match to be '*Renewable energy consumption (TJ)*' in the WB dataset. However, the Levenshtein distance determined the best match to be '*RISE Renewable Energy - Planning*', also in the WB dataset. Similarly, for the search term '*fertility\_rate*', Jaro-Winkler

---

<sup>2</sup><https://commons.apache.org/proper/commons-text/userguide.html>

<sup>3</sup><http://api.worldbank.org>

<sup>4</sup><https://query.wikidata.org/sparql>

returned ‘*Fertility rate, total (births per woman)*’ while Levenshtein returned “*Wanted fertility rate (births per woman)*”, both from the WB dataset. In both examples, the results from the Jaro-Winkler distance measure were better matches than those from the Levenshtein distance measure.

We observed that since the Jaro-Winkler algorithm favoured strings that matched from the beginning, it usually returned matches that were syntactically closer to the search terms. It is also more consistent with the format in which properties in queries and in KBs are expressed. That is, the most essential terms in a property’s label are observed to be usually at the beginning of the label. This is a desired feature for string matching in FRANK and so we prefer the Jaro-Winkler distance to the Levenshtein distance. However, since both string matching algorithms, generally, return similar results from our KBs and are both implemented in the Apache Commons library, this means that the two similarity measures and others from other libraries are easily interchangeable in FRANK. Our use of the Jaro-Winkler distance for matching a property in a query to properties in the knowledge bases is shown in figure 5.1.

Also, comparing exact string matching with these matchers, we only found exact matches for only 5 out of the 50 search terms tested.

In essence, the alist that would match facts in the KB could be one of the following:

```
{
  VALUE($y,population(Gold_Coast,$y,1930)) ,
  VALUE($y,population(Ghana,$y,1930)) ,
  VALUE($y,total_population(Gold_Coast,$y,1930)) ,
  VALUE($y,total_population(Ghana,$y,1930))
  ...
}
```

In particular, when we search the World Bank dataset, we find a match for the alist

```
VALUE($y,total_population(Ghana,$y,1930))
```

Formally, given an alist

```
f = {ag:VALUE, subj: X, prop: p, obj: o, time: t, ...}
```

where;

- subject,  $X$  has synonymous terms  $\{s_1, \dots, s_m\}$ ,
- property,  $p$  has synonymous terms  $\{p_1, \dots, p_n\}$ , and
- object,  $o$  has synonymous terms  $\{o_1, \dots, o_r\}$ ,

**Algorithm 6** Find Property

---

```

1: procedure FINDPROPERTY(searchTerm, KBs)
2:   searchTerm ← searchTerm.replace(UNDERSCORE, SPACE)
3:   searchTerm ← searchTerm.insert(SPACE)
4:           between a lowercase letter and an uppercase letter
5:   normalizedSearchTerm ← lowercase(searchTerm)
6:   kbPropertyList ← {}
7:   matchedProperty ← {}
8:   for kb in KBs do
9:     uniqueProperties ← Get list of unique properties from kb
10:    /*SELECT DISTINCT... statement for SPARQL endpoints*/
11:    kbPropertyList.append((kb, uniqueProperties)) /*stored in cache for subsequent use*/
12:    if uniqueProperties.contains(normalizedSearchTerm) then
13:      matchedProperty.append((kb, normalizedSearchTerm))
14:  if matchedProperty is empty then
15:    synTerms ← GetWordnetSynonyms(normSearchTerm)
16:    minDistance, jwDist ← MAX_INTEGER
17:    for term in {normalizedSearchTerm} ∪ synTerms do
18:      for (kb,propertyList) in kbPropertyList do
19:        for p in propertyList do
20:          jwDist ← JarowinklerDistance(term, p)
21:          if minDistance > jwDist then
22:            matchedProperty ← (kb, p)
23:  return matchedProperty

```

---

Figure 5.1: Algorithm for finding properties from KBs.

then, the lookup decomposition  $\Delta_{lookup}$  of the alist  $f$  is given by

$$\Delta_{lookup}(f[\text{subj} = s, \text{prop} = p, \text{obj} = o]) \mapsto \{g[\text{subj} = s_i, \text{prop} = p_j, \text{obj} = o_k] \mid i \in \{1, \dots, m\}, j \in \{1, \dots, n\}, k \in \{1, \dots, r\}\}$$

where WordNet Miller (1995) and a commonsense KB ConceptNet<sup>5</sup> are used for finding synonyms or similar concepts of property names.

This technique assumes a number of naming conventions commonly used in KBs. In the future, this technique can be improved using toolkits for natural language processing and other machine learning techniques better suited for learning these syntactic structures from training examples.

Finally, to avoid repeating searches at leaf nodes of similar terms, we cache the related terms locally, and simply reuse them whenever the LOOKUP decomposition rule is to be used. We explain the process of caching in chapter 7.

---

<sup>5</sup><http://conceptnet.io>

### 5.3.5 Alist Normalization

The normalization rule is based on the notion of simple, compound alists and FRANK Normal Form. An alist must be in FRANK normal form for variables in the alist at the leaf of the FRANK tree to be instantiated using data from knowledge bases. That is, compound alists must be unpacked into a combination of simpler ones before FRANK attempts lookups from KBs.

As defined in 4.3.1, a compound alist is an alist that has at least one attribute that contains a nested expression such as a subquery or propositional statements. Consider the query:

```
VALUE(?y, gdp(capitalOf(Scotland,$x),?y,2001))
```

Following the query internalization process (4.7.1) of FRANK, we obtain the alist:

```
{ag:VALUE, subject:capitalOf(Scotland,$x), predicate:gdp, object:?y,
  time:2001}
```

A lookup operation from this alist will fail since there is no atomic term for subject with which to search. We must, first, solve the inner the propositional statement, *capitalOf(Scotland,\$x)*, to identify \$x. Similarly, for the query:

```
GT(?b,
  VALUE(?b,rural_population(
    MAX($d,
      COMP(<?c,$d>,arable_land(?c,$d,2003):
        Country(?c) & location(?c,Africa))),?b,2003)),
  VALUE(?b,urban_population(
    MIN($h,
      COMP(<?g,$h>,arable_land(?g,$h,2003):
        Country(?g) & location(?g,Africa))),?b,2003))
)
```

Listing 5.1: Compound Query example

we need to solve the inner VALUE aggregate (as well as their nested operations) before we can solve the GT aggregate.

That is, for an alist where at least one of the attributes is a compound expression (i.e. a propositional statement or a subquery), decomposition by alist simplification,  $\Delta_{normalize}$ , unpacks compound expressions in the alist such that child alists are created that contain only simple, atomic terms.

Alist normalization plays a key role during initialization of a FRANK tree for queries that contain compound expressions. The normalization process will always run

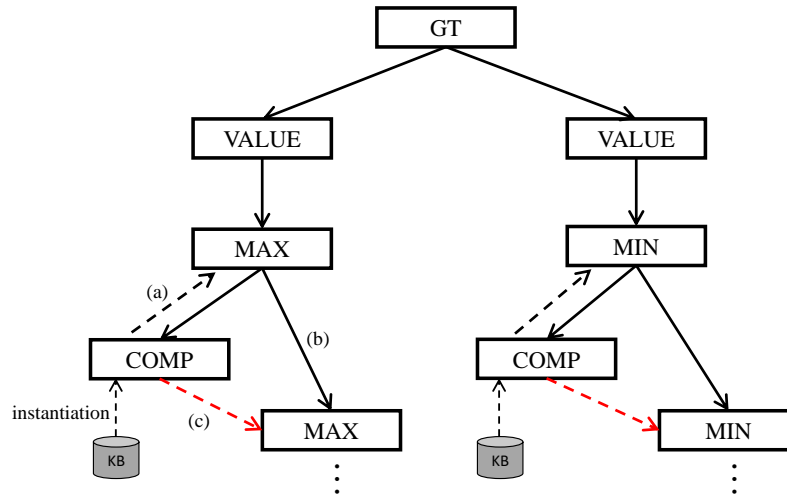


Figure 5.2: Decomposition: Normalize

An illustration of the process of normalizing the nested (compound) query in equation 5.1. The COMP aggregate (set comprehension) is different from other alists because it indirectly, via (c), propagates instantiated variables to its siblings. The instantiated variables (in this case  $?c$  from the sentence  $Country(?c) \ \& \ location(?c, Africa)$ ), are propagated to the parent node, MAX, via (a). MAX then spawns (b) a new child node with a copy of itself and replaces the appropriate variable that contain a subquery with the list of value received from COMP. MAX, now containing a list of values in the variable rather than a subquery, goes on to spawn new child nodes for each of the values in the list. Further exploration of the FRANK tree continues from these new nodes.

before other kinds of decompositions and therefore results in the creation of the first levels of the FRANK tree. The alist resulting from 5.1 will be decomposed one layer at a time, first moving the *VALUE* aggregate into new child nodes of GT aggregate's alist. The *VALUE* nodes are further decomposed to get *MIN* and *MAX* aggregates respectively, and finally *COMPs* (section 5.7.3) are extracted to process the propositional expressions. The *COMP* aggregate performs set comprehension to return the list of pairs  $\langle x_1, x_2 \rangle$  on the property *arable\_land* conditioned on the logical statements. The resulting tree (showing only the aggregates' names) is illustrated in figure 5.2.

### 5.3.6 Temporal Decomposition

Temporal decomposition is based on the intuition that if the query requests data for a specified date and that data point is not available in the KBs, then we can take ad-

vantage of the data observed for different dates and draw inferences from it for the date originally required in the query. In this project, we limit temporal decomposition to the ‘year’ level of granularity for data/time values. Extending this to day, month, decade, century, etc. is possible by adding the requisite date/time functions in FRANK that correspond to the date/time function supported by KBs through SPARQL or APIs through which we access data in the KB. Since FRANK dynamically loads inference rules at runtime we can extend rules easily. The architecture and implementation of FRANK is explained in chapter 7.

Suppose a query requires the value of  $x$  for the year 2012, but then the KB only has values for dates from 2000 to 2011 and from 2013 to 2017. Then, whereas traditional information retrieval will return no answer for this query, in FRANK, we infer the value of  $x$  from the available data points. That is, we infer the required value by decomposing the alist for which the lookup failed and then creating child alists for other dates close to the required date. This is the temporal decomposition of the alist.

Formally, suppose an alist

```
f = {ag:VALUE, subj: someSubject, prop: somePredicate, obj: ?xt,
     time: t, ...}
```

represents a query that requires the real-valued answer  $x_t$  at time index  $t$ , but then values available in the KBs are

$$\{x_{t-n}, x_{t-n-1}, \dots, x_{t-1}, x_{t+1}, \dots, x_{t+m}\}$$

at times

$$t-n, t-n-1, \dots, t-1, t+1, \dots, t+m$$

respectively, where  $m > 0$  and  $n > 0$ , and the time indices specify an ordering of the date/time values, but do not indicate temporal consecutiveness of the date/time values. Then, the temporal decomposition  $\Delta_{temporal}$  of  $f$  is given by:

$$\begin{aligned} \Delta_{temporal}(f[time = t]) &\mapsto s[time = t, ag = \alpha] \\ &\mapsto \{g[time = i] \mid i \in \{t-n, t-n-1, \dots, t-1, t+1, \dots, t+m\}\} \end{aligned}$$

where each  $g$  is a new alist with the same attribute values except:

$g[time] = i$  and  $g[aggregate] = VALUE$ ,

since new alists by default have the *VALUE* aggregate.

In FRANK, alist  $f$  is decomposed into a new child alist,  $s$ , that keeps all attributes of  $f$  except ‘operation’ and ‘time’. The operation in  $s$  is changed to reflect the aggregate function that will be used to combine the values of  $x$  from all the child nodes.

The choice of the temporal decomposition rule constitutes the OR branch, where as the requirement to find solve  $g$  for each of the selected dates forms the AND branch. We refer to  $s$  as the strategy node and it constitutes the OR branching in the FRANK tree. The aggregate  $\alpha$  in alist  $s$  is used as a label in the alist that determines which algorithm is used to combine the relevant fields from the child nodes that represent facts from multiple times.

### 5.3.7 Geospatial Decompositions

Our use of geospatial decomposition in FRANK is based on the ideas of merology and merotopology (Cohn and Hazarika (2001)) that create a spatial logic for regions and connections. This type of decomposition is applied to attributes in alists that represent geo-spatial concepts. We identify geospatial concepts by checking their existence in the GeoNames<sup>6</sup> knowledgebase. We also check the class types of types from Dbpedia, ConceptNet by looking for the predicates *isA* or *type* that have values *location*, *place*, *country*, *city*, *town*, *continent*, etc. The existence of such an entity in an alist gives FRANK the options to use the part-hood heuristic to find an alternative solution to the query if the direct lookup of the alist fails to match any knowledge base facts. The part-hood rule is based on the idea that an entity being partitioned into its sub-parts such that a problem can decomposed into smaller problems for the sub-parts which could be easier to solve and then aggregated to solve for the whole entity. If a geographical entity such as a continent can be partitioned into the countries that it is composed of, then we can solve a problem for its countries first, and then aggregate to get an answer for the continent.

Consider the query:

```
VALUE($y,wind_energy_consumption(Europe,$y,2012))
```

that seeks to find the wind energy consumption of Europe in 2012. The root alist created is:

```
{ag:VALUE, agvar:?y, subj:Europe, prop:wind_energy_consumption, obj:?y,
  time:2012 }
```

A direct lookup of this alist from knowledge bases will most likely yield no match since the required answer is an aggregation of facts that are more readily available for individual countries within Europe. We arrive at this intuition because a knowledge base informs us that Europe *is a* continent and a continent is made up of countries

---

<sup>6</sup><http://geonames.org>



at its next level of partitioning by administrative regions. Therefore we can partition Europe into its constituent parts, solve the query for these parts, and then aggregate to solve for the “whole”. Multiple geospatial decompositions may be required to reach the appropriate level of entity decomposition for which facts can be found.

Formally, given an alist

$$f = \{\text{ag:VALUE, subj: } X, \text{ prop: } p, \text{ obj: } ?o, \text{ time: } t, \dots\}$$

where:

1. the *isA* predicate, that determines the types of entities, holds as follows:

$$isA(X, location) \vee isA(X, place) \in true \quad (5.1)$$

2. we can find  $\{x_1, \dots, x_n\}$  such that the following conditions for a partition are satisfied:

$$partOf(x_i, X) \in true \quad (5.2)$$

$$X = x_1 \cup \dots \cup x_n \quad (5.3)$$

The geospatial decomposition  $\Delta_{geospatial}$  of the alist  $f$  is given by

$$\Delta_{geospatial}(f[subj = X]) \mapsto \{g[subj = s] \mid s \in \{x_1, \dots, x_n\}\}$$

where each  $g[subj = x_i]$  is a new alist with a subject that is geospatially part of (or located in) the subject of alist  $f$ . We obtain the parthood relationships from web knowledge bases such as Dbpedia<sup>7</sup>, Wikidata<sup>8</sup> and Conceptnet<sup>9</sup>. However, satisfying the condition in equation 5.3 is not practical in many cases for all types of entities. For instance, most knowledge bases will specify countries that are part of a continent, but do not state if these countries form a partition of the continent. Different knowledge bases also have variations in the partitions that we infer from the entities and *partOf* relations that they express. In most cases, they offer no such ‘partition’ guarantee in the *partOf* relations that they express. We, however assume that partitions can be formed from the *partOf* relations and absorb the uncertainty due to missing data as part of our calculations of uncertainty in FRANK in chapter 6. Usually only minor parts are omitted whose omission will be within the error bars of the answer.

<sup>7</sup><http://wiki.dbpedia.org/>

<sup>8</sup><https://www.wikidata.org>

<sup>9</sup><http://conceptnet.io>

Finally, because this type of decomposition is used very often, we have added an ontology to our local KB that defines the geospatial hierarchy between entity classes that we can use easily rather than searching for these relations in a KB. The ontology includes relations such as *hasPart(Continent, Country)*, *hasPart(Country, AdministrativeRegion)*, *hasPart(AdministrativeRegion, City)* and *hasPart(AdministrativeRegion, Town)*. These are based on the class hierarchy GeoNames ontology (Vatant and Wick (2012)).

We can generalise geospatial decomposition to a part/whole decomposition by excluding the location/place constraint in the logical statement 5.1. This will however require additional steps to determine the type of the entity and how it is composed to know the dimension(s) along which to decompose it. We leave this as future work for FRANK.

## 5.4 Execution and Propagation

Execution of FRANK includes, as a core process, the application of aggregates to alists. In this section, we discuss the execution of alist aggregate functions and how they are propagated. This means that variables are not always propagated unchanged, but are fed into inference calculations at parent nodes to determine what is passed back up. This differs significantly from the backward propagation of assigned variables in proof trees in automated reasoning (AR) systems where the objective is to determine the truth value assignment to variables. In AR systems, terms are instantiated and combined (by unification) but not evaluated, except in higher order unification where beta-reduction can be applied.

Several of the aggregates in FRANK are arithmetic functions because they cover many of the aggregation functions commonly found in queries. It is also important to note that although FRANK can answer qualitative queries like “*What is the capital city of Ghana*”, our primary focus is on quantitative queries that require numeric answers, and nested queries that have both qualitative and quantitative sub-queries. This means that we need to equip FRANK with the basic arithmetic functions necessary to compute answers from data available in alists. In addition to arithmetic functions that aggregate values, we also include aggregates that generate functions. These functions give FRANK the ability to perform prediction and also reuse inferred functions later. We discuss these aggregates in section 5.5.

FRANK propagates attributes in alists from the leaves to the root as part of the

query-answering process. Propagation of variables in a lists is determined by the aggregates that label a lists during decomposition.

## 5.5 Aggregate Functions

In this section, we briefly describe the aggregates that are available in FRANK. An aggregate works by applying the appropriate function to a specified attribute in the a list or the child a lists. *Aggregate variables* identify the attribute(s) that an aggregate should be performed on.

We now go on to discuss the aggregates used in FRANK. For each aggregate, we provide its type signature and then describe the underlying mathematical function(s) that it performs.

### 5.5.1 MAX

$MAX : [\mathbb{S} \times (\text{Set } \tau_f \mapsto \mathbb{R})] \mapsto \tau_f$

Given a variable name as a string  $\mathbb{S}$ , and a function of a set of objects of a list type  $\tau_f$  onto real numbers  $\mathbb{R}$ , MAX returns the  $\tau_f$  with the maximum value returned by the functions.

In the context of a FRANK query, we express the MAX aggregate as:

$$MAX(\mathfrak{v}, \{z_1, \dots, z_n\}) \quad (5.4)$$

where the  $z_i$ s represent nested a lists. When MAX is used in a FRANK query, each  $z_i$  represents a subquery. These follow the notation outlined in section 5.2.3. Internally, in the FRANK tree, the MAX aggregate finds the a list with the maximum value for a specified attribute. Mathematically, we can express this using an *argmax* function. An *argmax* is the maximum value given an argument that represents a feature or dimension of the set of objects. MAX aggregate is expressed as the *argmax* of the a list on the aggregate variable,  $\mathfrak{v}$ .

$$f_{MAX(\mathfrak{v})}[\mathfrak{v}] = \text{argmax}_{\mathfrak{v}}(\{g_1[\mathfrak{v}], \dots, g_n[\mathfrak{v}]\}) \quad (5.5)$$

where  $g_i$ s are a lists having variable  $\mathfrak{v}$  as an attribute and the  $g_i$ s are children of  $f$  in the FRANK tree.

### 5.5.2 MIN

$MIN : [\mathbb{S} \times (\text{Set } \tau_f \mapsto \mathbb{R})] \mapsto \tau_f$

Given a variable name of type  $\mathbb{S}$  and a function of a set of objects of alist type  $\tau_f$  onto real numbers  $\mathbb{R}$ , MIN returns the alist with minimum value returned by the functions.

In the context of a FRANK query, we express the MIN aggregate as:

$$MIN(\mathfrak{v}, \{z_1, \dots, z_n\}) \quad (5.6)$$

Internally, in the FRANK tree, the MIN aggregate finds the alist with the minimum value for a specified attribute. Mathematically, we can express this as the *argmin* of the alist on the aggregate variable,  $\mathfrak{v}$ .

$$f_{MIN(\mathfrak{v})}[\mathfrak{v}] = \text{argmin}_{\mathfrak{v}}(\{g_1[\mathfrak{v}], \dots, g_n[\mathfrak{v}]\}) \quad (5.7)$$

where  $g_i$ s are alists having variable  $\mathfrak{v}$  as an attribute and the  $g_i$ s are children of  $f$  in the FRANK tree.

### 5.5.3 COUNT

$COUNT : [\mathbb{S} \times (\text{Set } \tau_f \mapsto \tau)] \mapsto \mathbb{N}$

Given a variable name of type  $\mathbb{S}$  and a function of a set of objects of alist type  $\tau_f$  onto arbitrary objects of type  $\tau$ , COUNT returns the cardinality of the set of non-null values returned by the functions.

In the context of a FRANK query, we express the COUNT aggregate as:

$$COUNT(\mathfrak{v}, \{z_1, \dots, z_n\}) \quad (5.8)$$

In the FRANK tree, COUNT finds the number of child alists with non-empty values for  $\mathfrak{v}$ , the aggregate variable.

$$\begin{aligned} f_{COUNT(\mathfrak{v})}[\mathfrak{v}] &= COUNT(\{g_1[\mathfrak{v}], \dots, g_n[\mathfrak{v}]\}) \\ &= |\{g_1[\mathfrak{v}], \dots, g_n[\mathfrak{v}]\}| \end{aligned} \quad (5.9)$$

where  $g_i[\mathfrak{v}]$  is not empty.

where  $g_i$ s are alists having variable  $\mathfrak{v}$  as an attribute and the  $g_i$ s are children of  $f$  in the FRANK tree.

### 5.5.4 SUM

$SUM : [\mathbb{S} \times (\text{Set } \tau_f \mapsto \mathbb{R})] \mapsto \mathbb{R}$

Given a variable name of type  $\mathbb{S}$  and a function of a set of objects of alist type  $\tau_f$  onto real numbers  $\mathbb{R}$ , SUM adds the values returned by the functions.

In the context of a FRANK query, we express the SUM aggregate as:

$$SUM(\mathfrak{v}, \{z_1, \dots, z_n\}) \quad (5.10)$$

In the FRANK tree, SUM finds the arithmetic sum of the attribute  $\mathfrak{v}$  of the child alists of  $f$ . This sum is assigned to the variable that corresponds to the aggregate variable,  $\mathfrak{v}$ , of  $f$ .

$$f_{SUM(\mathfrak{v})}[\mathfrak{v}] = \sum_{i=1}^n g_i[\mathfrak{v}] \quad (5.11)$$

where  $g_i$ s are alists having variable  $\mathfrak{v}$  as an attribute and the  $g_i$ s are children of  $f$  in the FRANK tree.

### 5.5.5 AVERAGE

$AVG : [\mathbb{S} \times (\text{Set } \tau_f \mapsto \mathbb{R})] \mapsto \tau_f$

Given a variable name and a function of a set of objects of alist type  $\tau_f$  onto real numbers, AVG returns an object of type  $\tau_f$  with the mean value returned by the functions.

In the context of a FRANK query, we express the AVG aggregate as:

$$AVG(\mathfrak{v}, \{z_1, \dots, z_n\}) \quad (5.12)$$

Internally, AVG finds the average value of  $\mathfrak{v}$  of the child alists of  $f$ . This mean value is assigned to the variable that corresponds to the aggregate variable,  $\mathfrak{v}$ , of  $f$ .

$$\begin{aligned} f_{AVG(\mathfrak{v})}[\mathfrak{v}] &= \text{mean}(\{g_1[\mathfrak{v}], \dots, g_n[\mathfrak{v}]\}) \\ &= \frac{\sum_{i=1}^n g_i[\mathfrak{v}]}{n} \\ &= \frac{f_{SUM(\mathfrak{v})}[\mathfrak{v}]}{f_{COUNT(\mathfrak{v})}[\mathfrak{v}]} \end{aligned} \quad (5.13)$$

where  $g_i$ s are alists having variable  $\mathfrak{v}$  as an attribute and the  $g_i$ s are children of  $f$  in the FRANK tree.

### 5.5.6 MEDIAN

$MEDIAN : [\mathbb{S} \times (\text{Set } \tau_f \mapsto \mathbb{R})] \mapsto \mathbb{R}$

Given a variable name and a function of a set of objects of alist type  $\tau_f$  onto real numbers, MEDIAN returns the statistical median value returned by the functions.

In the context of a FRANK query, we express the MEDIAN aggregate as:

$$MEDIAN(\mathfrak{v}, \{z_1, \dots, z_n\}) \quad (5.14)$$

Internally, MEDIAN finds the statistical median of the  $\mathfrak{v}$  variables in the child nodes of alist  $f$ .

$$\begin{aligned} f_{MEDIAN(\mathfrak{v})}[\mathfrak{v}] &= MEDIAN(\{g_1[\mathfrak{v}], \dots, g_n[\mathfrak{v}]\}) \\ &= \begin{cases} g_{(n+1)/2}[\mathfrak{v}], & \text{if } n \text{ is odd} \\ \frac{1}{2}(g_{n/2}[\mathfrak{v}] + g_{1+n/2}[\mathfrak{v}]), & \text{if } n \text{ is even.} \end{cases} \end{aligned} \quad (5.15)$$

where  $g_i$ s are alists having variable  $\mathfrak{v}$  as an attribute and the  $g_i$ s are children of  $f$  in the FRANK tree.

### 5.5.7 DIFFERENCE

$DIFF : [\mathbb{S} \times (\tau_f \mapsto \mathbb{R}) \times (\tau_f \mapsto \mathbb{R})] \mapsto \mathbb{R}$

Given a variable name and two unary functions of objects of alist type  $\tau_f$  onto real numbers, DIFF subtracts the value returned by the first function from the value returned by the second function.

In the context of a FRANK query, we express the DIFFERENCE aggregate as:

$$DIFF(\mathfrak{v}, z_1, z_2) \quad (5.16)$$

In the FRANK tree, DIFFERENCE finds the arithmetic difference between  $\mathfrak{v}$  values of two child alists. This difference is assigned to the variable that corresponds to the aggregate variable,  $\mathfrak{v}$ , of  $f$ . The difference aggregate can only take two arguments and therefore uses only the first two child alists if there are more than two. Further, the right child is subtracted from the left child when the aggregate is executed.

$$f_{DIFF(\mathfrak{v})}[\mathfrak{v}] = g_1[\mathfrak{v}] - g_2[\mathfrak{v}] \quad (5.17)$$

where  $g_i$ s are alists having variable  $\mathfrak{v}$  as an attribute and the  $g_i$ s are children of  $f$  in the FRANK tree.

### 5.5.8 PRODUCT

$PRODUCT : [\mathbb{S} \times (Set \tau_f \mapsto \mathbb{R})] \mapsto \mathbb{R}$

Given a variable name and a unary function of a set of objects of alist type  $\tau_f$  onto real numbers, PRODUCT multiplies the values returned by the functions.

In the context of a FRANK query, we express the PRODUCT aggregate as:

$$PRODUCT(v, \{z_1, \dots, z_n\}) \quad (5.18)$$

In the FRANK tree, PRODUCT multiplies the attribute  $v$  of the child alists of  $f$ . This product is assigned to the variable that corresponds to the aggregate variable,  $v$ , of  $f$ .

$$f_{PRODUCT(v)}[v] = \prod_{i=1}^n g_i[v] \quad (5.19)$$

where  $g_i$ s are alists having variable  $v$  as an attribute and the  $g_i$ s are children of  $f$  in the FRANK tree.

### 5.5.9 GT

$GT : [\mathbb{S} \times (\tau_f \mapsto \mathbb{R}) \times (\tau_f \mapsto \mathbb{R})] \mapsto \mathbb{B}$

Given a variable name and two unary functions of a set of objects of alist type  $\tau_f$  onto real numbers, GT returns *true* if the specified variable of the first function returns a value that is greater than that of the second function.

In the context of a FRANK query, we express the GT aggregate as:

$$GT(v, z_1, z_2) \quad (5.20)$$

Internally, the GT aggregate determines if the first child of an alist is greater than the second child when compared by variable  $v$ .

$$\begin{aligned} f_{GT(v)}[v] &= GT(g_1[v], g_2[v]) \\ &= \begin{cases} true, & \text{if } g_1[v] > g_2[v] \\ false, & \text{otherwise.} \end{cases} \end{aligned} \quad (5.21)$$

where  $g_i$ s are alists having variable  $v$  as an attribute and the  $g_i$ s are children of  $f$  in the FRANK tree.

### 5.5.10 LT

$LT : [\mathbb{S} \times (\tau_f \mapsto \mathbb{R}) \times (\tau_f \mapsto \mathbb{R})] \mapsto \mathbb{B}$

Given a variable name of string type and two unary functions of a set of objects of alist type  $\tau_f$  onto real numbers, LT returns *true* if the specified variable of the first function returns a value that is less than that of the second function.

In the context of a FRANK query, we express the LT aggregate as:

$$LT(\mathfrak{v}, z_1, z_2) \quad (5.22)$$

Internally, the LT aggregate determines if the first child of an alist is less than the second child when compared by variable  $\mathfrak{v}$ .

$$\begin{aligned} f_{LT(\mathfrak{v})}[\mathfrak{v}] &= LT(g_1[\mathfrak{v}], g_2[\mathfrak{v}]) \\ &= \begin{cases} true, & \text{if } g_1[\mathfrak{v}] < g_2[\mathfrak{v}] \\ false, & \text{otherwise.} \end{cases} \end{aligned} \quad (5.23)$$

where  $g_i$ s are alists having variable  $\mathfrak{v}$  as an attribute and the  $g_i$ s are children of  $f$  in the FRANK tree.

### 5.5.11 EQ

$EQ : [\mathbb{S} \times (\tau_f \mapsto \mathbb{R}) \times (\tau_f \mapsto \mathbb{R})] \mapsto \mathbb{B}$

Given a variable name of string type and two unary functions of a set of objects of alist type  $\tau_f$  onto real numbers, EQ returns *true* if the specified variable of the first function returns a value that is equal to that of the second function.

In the context of a FRANK query, we express the EQ aggregate as:

$$EQ(\mathfrak{v}, z_1, z_2) \quad (5.24)$$

Internally, the EQ aggregate determines if the first child of an alist is equal to the second child when compared by variable  $\mathfrak{v}$ .

$$\begin{aligned} f_{EQ(\mathfrak{v})}[\mathfrak{v}] &= EQ(g_1[\mathfrak{v}], g_2[\mathfrak{v}]) \\ &= \begin{cases} true, & \text{if } g_1[\mathfrak{v}] = g_2[\mathfrak{v}] \\ false, & \text{otherwise.} \end{cases} \end{aligned} \quad (5.25)$$

where  $g_i$ s are alists having variable  $\mathfrak{v}$  as an attribute and the  $g_i$ s are children of  $f$  in the FRANK tree.



## 5.6 Function-Generation

A key feature of FRANK is the inference of new data values from existing ones when answering a query. This allows FRANK to go beyond looking up data and performing arithmetic aggregation. This is one of the key features of FRANK that makes it different from other query answering systems.

Additionally, the inferred functions can be stored and reused in a query with similar alist attributes. We discuss these aggregates below

### 5.6.1 REGRESS

$REGRESS : (Set \tau \times \mathbb{R}) \mapsto (\tau \mapsto \mathbb{R})$

Given a set of pairs of objects of type  $\tau$  and real numbers, REGRESS forms a unary function,  $h$ , from objects of types  $\tau$  and reals such that  $h(x) \simeq y$  for all  $x$  and  $y$  in the input set (within some error bar).

We express the REGRESS aggregate of an alist with aggregate variables  $t$  representing time and  $v$  representing any other numeric variable in the alist respectively as:

$$REGRESS(\{\langle t, v \rangle\}, \lambda xy. function(x, y)) \quad (5.26)$$

where  $\lambda xy. function(x, y)$  is a binary function representing:

- a binary aggregate expressed  $\lambda xy. aggregate(x, y)$ , or
- logical expression with two unknowns expressed as  $\lambda xy. property(s, x, y)$  where the subject of the property is known, but object and time are unknown.

The REGRESS aggregate performs regression analysis on the retrieved data values. This process generates a regression function from which unobserved data from the KB can be predicted. REGRESS is based on the statistical methods for regression and works with a number of assumptions including:

1. the observed sample is representative of the population;
2. regression error is a random variable with mean zero, conditional on the observed variables.

Formally in FRANK, we define REGRESS as follows.

$$\begin{aligned} f_{REGRESS(t,v)}[h] &= REGRESS(\{(g_1[t], g_1[v]), \dots, (g_n[t], g_n[v])\}) \\ &= Y \approx h(X, \beta) \end{aligned} \quad (5.27)$$

where  $h$  is the generated function in alist  $f$ ,

$X = \{g_i[t], i \in (1, \dots, n)\}$  represent values for the independent variable,  $t$ ,

$Y = \{g_i[v], i \in (1, \dots, n)\}$  represent values for the dependent variable,  $v$ ,

and unknown parameters ( $\beta$  and  $\epsilon$ ) are calculated from the observed data values.

Inferred function takes the form:

$$f[h] = \beta_0 + \beta_1 t + \beta_2 t^2 + \dots + \beta_n t^n + \epsilon. \quad (5.28)$$

In FRANK,  $t$  is the value of the *time* and  $v$  is a real-valued property of an entity. Although FRANK has been implemented to generate polynomials up to degree  $n$ , it defaults to linear functions since it does not know a priori what the true model of the data is and automatic model selection is currently not implemented for FRANK. We can however set the degree parameter in the configuration of FRANK to change to higher degree polynomials when the data model is known before running a query.

## 5.6.2 GP (Gaussian Process Regression)

The GP aggregate applies the Gaussian Process regression technique to infer the values of unobserved data points that FRANK requires during inference. A *Gaussian Process* is a collection of random variable, any finite number of which have joint Gaussian distributions. Thus, a Gaussian process is fully specified by its mean function  $m$  and the covariance function  $k$  (Rasmussen (2006)). That is,

$$h \sim \mathcal{GP}(m, k) \quad (5.29)$$

$$\mathcal{GP} : (\text{Set } \tau \times \mathbb{R}) \mapsto (\tau \mapsto \mathbb{R})$$

Given a set of pairs of objects of type  $\tau$  and real numbers, GP forms a binary function,  $h$ , from objects of types  $\tau$  and reals such that  $h(m, k)$  is a Gaussian process distribution with mean  $m$  and covariance  $k$  generated from the input data pairs  $(x, y)$ .

We express the  $\mathcal{GP}$  aggregate of an alist as:

$$f_{\mathcal{GP}} = \mathcal{GP}(\{\langle x, y \rangle\}, \lambda xy. function(x, y)) \quad (5.30)$$

where  $\lambda xy. function(x, y)$  is a binary function representing:

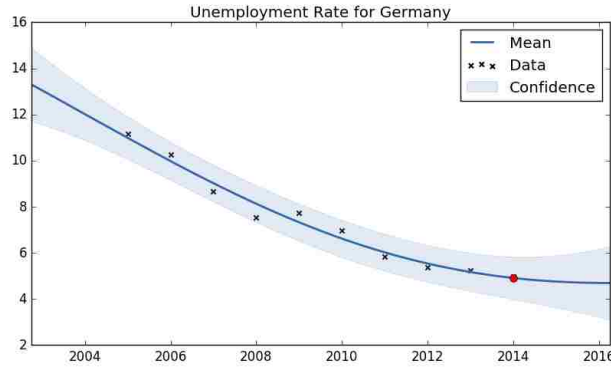


Figure 5.3: Gaussian Process Regression Example

Plot of the unemployment rate for Germany with confidence margin of two standard deviations.

- a binary aggregate expressed  $\lambda xy.aggregate(x, y)$ , or
- logical expression with two unknowns expressed as  $\lambda xy.property(s, x, y)$  where the subject of the property is known, but object and time are unknown.

Formally in FRANK, we define GP as follows.

$$f_{\mathcal{GP}(\nu, \alpha)}[h] = \mathcal{GP}(\{(g_1[\alpha], g_1[\nu]), \dots, (g_n[\alpha], g_n[\nu])\}) \quad (5.31)$$

where  $h$  is a function distributed as a GP with mean,  $m$ , and covariance function,  $k$ .  $\nu$  is the dependent variable and  $\alpha$  is the independent variable of (usually, time) in the alist.

For a given set of observed data,  $D$ , the function  $h$  in alist  $f$  is given by:

$$f[h]|D \sim \mathcal{GP}(m_D, k_D), \quad (5.32)$$

$$\begin{aligned} m_D &= m(x) + \sum (X, x)^\top \Sigma^{-1} (f - m), \\ k_D &= k_D(x, x') = k(x, x') = \sum (X, x)^\top \Sigma^{-1} \sum (X, x') \end{aligned} \quad (5.33)$$

Although REGRESS and GP both perform regression, we use them differently in FRANK. When the query simply requires the prediction of a value from observed data, either one can be used. In the FRANK tree, both strategies are generated. Since the Gaussian Process captures uncertainty directly in the variance of the distribution that

generates the function, it lowers the variance when more values are observed around the point for which we require a prediction. FRANK captures uncertainty in terms of variance of deviation of inferred value from the assumed true value (see chapter 6), so the GP's output easily fits in with the calculation of uncertainty in FRANK. However, when we need to generate a regression function on which other aggregates are applied, we use REGRESS since it directly returns a polynomial function. We consider the REGRESS operation as a cheaper aggregate given that the libraries used for the GP implementation in FRANK takes longer to initialize, where as the REGRESS does not need such initializations.

### 5.6.3 INTEGRAL

$INTEGRAL : (\tau \mapsto \mathbb{R}) \mapsto (\tau \mapsto \mathbb{R})$

Given a unary function that takes objects of type  $\tau$  onto real numbers, INTEGRAL returns the unary function obtained after the symbolic integration of the input function.

$$\begin{aligned} f_{INTEGRAL(v)}[v] &= INTEGRAL(g[v]) \\ &= \int (g[v])(\alpha) d\alpha \end{aligned} \quad (5.34)$$

where  $g[v]$  is a function of  $\alpha$ . The integral is used when we need to calculate the area under the curve to estimate the total of a quantity over a period of time for quantities that are not repeated or reused through different time periods. For example, *population* is repeated since everyone alive in year  $X$  is added to the population value of year  $X + 1$  if they are still alive, and so integrating a function of this property will lead to an erroneous answer. However, a quantity such as *total income* is not repeated and can be integrated. For now, we have no way of automatically determining which properties are integrable and which ones are not. We deal with this by specifying a list of properties in the KBs that can be integrated and those that cannot.

We omit the constant of integration since the definite integral is computed for the symbolic result when the limits of integration are specified in the parent alist. Although this is a potential source of error in cases where the integral function is propagated, it simplifies the calculation to a large extent by not having deal with the integration constant. In the future, we hope to refine this further to have a more robust treatment of the integration constant to ensure the correctness and accuracy of the integrals.

### 5.6.4 DERIV

$INTEGRAL : (\tau \mapsto \mathbb{R}) \mapsto (\tau \mapsto \mathbb{R})$

Given a unary function that takes objects of type  $\tau$  onto real numbers, *DERIV* returns the unary function obtained after the first symbolic derivative of the input function. Since *FRANK* infers continuous polynomial functions, we assume that the input function for the *DERIV* aggregate is continuous and differentiable.

$$\begin{aligned} f_{DERIV(v)}[v] &= DERIV(g[v]) \\ &= \frac{d}{d\alpha}(g[v])(\alpha) \end{aligned} \quad (5.35)$$

where  $g[v]$  is a function of  $\alpha$ .

We compute the symbolic derivative using third-party Java libraries and explain the process in chapter 7.

### 5.6.5 APPLY

$APPLY : [(\tau \times \mathbb{R}) \times \tau] \mapsto \mathbb{R}$

Given a unary function that takes objects of type  $\tau$  onto real numbers and an object of type  $\tau$ , *APPLY* applies the input function to the input object and returns the real-valued output.

$$\begin{aligned} f_{APPLY(v,\alpha)}[v] &= APPLY(g[v], \alpha) \\ &= (g[v])(\alpha), \text{ and } \alpha \in \mathbb{R} \end{aligned} \quad (5.36)$$

where  $g[v]$  is a function and  $\alpha$  is the independent variable that is provided as its argument.

## 5.7 Non-arithmetic Aggregates

### 5.7.1 VALUE

$VALUE : [\mathbb{S} \times (\mathbb{S} \mapsto \tau_1)] \mapsto \tau_1$

Given a variable name of type  $\mathbb{S}$  and a function that take objects of type  $\mathbb{S}$  onto objects of type  $\tau_1$ , *VALUE* returns a single object of type  $\tau_1$  corresponding to the value of

the specified variable returned by the function, or a single object of type  $\tau_1$  that best represents the list of items returned by the function for the input variable name.

In a FRANK query, we express VALUE as:

$$VALUE(\mathfrak{v}, z(x)) \quad (5.37)$$

In the FRANK tree, VALUE is an identity aggregate with respect to attributes in an alist. It returns value from an alist based on the variable name (or the key of the alist) that is specified as the *opvar* (see section 4.4.3) of the alist. When an alist has just one child, the aggregate simply returns the requested alist attribute. If the alist, however, has more than one child, then the values of the child nodes are aggregated using the mean or median function (if the values are real numbers). If they are non-real, the alist with the lowest uncertainty is selected. In cases where nodes have same uncertainty, the modal value (value that appears the most in the child alists) is returned. We define VALUE as:

$$\begin{aligned} f_{VALUE(g,\mathfrak{v})}(\mathfrak{v}) &= VALUE(g, \mathfrak{v}) \\ &= g[\mathfrak{v}] \end{aligned} \quad (5.38)$$

### 5.7.2 VALUES

$VALUES: [\mathbb{S} \times (Set \mathbb{S} \mapsto \tau_1)] \mapsto Set \tau_1$

Given a variable name of string type  $\mathbb{S}$  and a set of functions that takes objects of type  $\mathbb{S}$  onto objects of type  $\tau_1$ , VALUES returns a set of objects of type  $\tau_1$  corresponding to the set of values returned by the function for the specified input variable name.

In a FRANK query, we express VALUES as:

$$VALUES(\mathfrak{v}, \{z_1, \dots, z_n\}) \quad (5.39)$$

In the FRANK tree, VALUES returns a set of values from all the child alists and is defined as:

$$\begin{aligned} f_{VALUES(g,\mathfrak{v})}(\mathfrak{v}) &= VALUES(\{g_1, \dots, g_n\}, \mathfrak{v}) \\ &= \{g_1[\mathfrak{v}], \dots, g_n[\mathfrak{v}]\} \end{aligned} \quad (5.40)$$

### 5.7.3 COMP

$COMP: [Set \tau_1 \times (Set \tau_1 \times \tau_2 \mapsto \mathbb{B})] \mapsto Set \tau_1 \times \tau_2$

Given a set of objects of type  $\tau_1$  and a set of binary relationships between objects of

type  $\tau_1$  and  $\tau_2$ , COMP (set comprehension) returns the set of pair of objects  $(\tau_1, \tau_2)$  such that the binary relationships between the two objects holds. This is usually written as

$$\{y | \exists x \in X. \phi(x, y)\} ,$$

where  $X$  is the set of input set and  $\phi$  the set of input relationships.

We in a FRANK query, we express VALUES as:

$$COMP(v, \{\lambda xy. property(x, y, t)\}) \quad (5.41)$$

In the FRANK tree, COMP is a set comprehension operation that creates a set of data pairs of the form  $\langle x, y \rangle$  when given the set  $x$  and some criteria for their corresponding  $y$  values.

$$\begin{aligned} f_{COMP(v, \alpha)} \cdot v &= COMP(g \cdot v, \alpha) \\ &= search(g \cdot v, \alpha) , \text{ where, } \alpha \text{ is a logical statement} \\ &= set(\langle x, y \rangle) \end{aligned} \quad (5.42)$$

where  $x$  and  $y$  from KBs satisfy the criteria in the logical statement  $\alpha$ .

The COMP aggregate is very different from other aggregates since, it does not propagate it's values to the root node. Its set of values from its child nodes is propagated immediately to its parent which creates a new child node with the instantiated set values that will be decomposed further by NORMALIZE into individual alists. See figure 5.2.

These are not the only inference aggregates that can be implemented in FRANK. We can easily extend FRANK by adding new kinds of aggregate that are relevant for specific domains of the framework's use.

## 5.8 Inference Costs

Inference in FRANK can quickly get very expensive. For instance, a query about *Europe* could get decomposed both temporally and geospatially. Each of these strategies could result in more than 20 new branches, and the same decompositions, if left unchecked, could be reapplied to their descendant nodes if grounding fails for the aggregate variables in alists. Additionally, the cost of inference is also influenced by the type of decomposition required as well as the functions that aggregate alists.

Given these concerns about combinatorial explosion of the search space as well as the complexity of the decompositions and aggregates, we employ *A\* Search*: an

informed search technique that uses heuristics to manage expansion and traversal of the FRANK tree. In the case of FRANK, we use the cost of inference as a search heuristic with objective of minimizing the cost of searching for an answer in the FRANK tree.

### 5.8.1 Cost Factors

In FRANK, two main factors contribute to the cost of getting answers to queries:

1. decomposition cost: cost that is incurred by strategies during the downward decomposition of alists in the FRANK tree; and
2. propagation costs: cost that is incurred during the upward propagation of alists when the requisite variables in alists are instantiated.

During decomposition, inference rules determine how new child nodes are created for an alist that is currently being processed. Some rules are more expensive than others. For instance, the geospatial strategy requires searches through external knowledge bases to find the entities into which a given geo-entity can be partitioned. This usually is a lot more computationally expensive than the temporal decomposition rule because there are more steps required to determine the sub-components of the entity. These include searching knowledge bases for the type of the entity and then finding the components of this entity that satisfy the conditions in equations 5.1 and 5.2. Temporal decomposition on the other hand needs minimal external resources to expand the time attribute in an alist. We desire an expansion of the tree that does not perform decompositions that unnecessarily increase the cost of inference.

Decomposition is, however, just one side of the story. When alist variables have been instantiated at the leaves of the tree, the values assigned to variables have to be propagated back to the root of the tree before FRANK can return an answer. The upward propagation involves the application of aggregates (previously determined by the decomposition strategies). Some of these aggregates are as simple as sums and products, while others like REGRESS are meant to generate functions, and APPLY gets back values by applying the generated functions.

### 5.8.2 Heuristics

We incorporate intuitions from problem solving by human experts to help obtain appropriate heuristics for FRANK. First, we note that the diverse nature of decomposition



rules and aggregate functions means that for a given query, we cannot, with any reasonable level of certainty, state which sequence of rules or aggregates will get us an answer. We can however attempt simpler rules and aggregates first and return to the more costly ones afterwards. This intuition is based on Occam's Razor (Blumer et al. (1987)), i.e. we look for the simplest possible solution to the problem.

We use the A\* search algorithm (Hart et al. (1968)) for searching, expanding and traversing the FRANK tree. We are interested, not only in the cost of getting from the current node to a leaf node state where an alist gets instantiated, but on the cost incurred in getting from the root node to the current node. By considering the cost from the root to the node, we ensure that the entire sequence of rules applied from root to a leaf is the least expensive given all alternatives available (that is, the triangle inequality for the A\* algorithm). We also factor in the estimated cost of the decomposition rule and the aggregates into the cost function  $f$ . We explain our cost function for A\* search heuristic below.

The cost function for  $\mathcal{L}$  for an alist  $f$  in a FRANK tree is calculated as

$$l(f) = g(f) + h(f) + c(f) \quad (5.43)$$

where:

- $g(\cdot)$  is the cost from the root alist to alist  $f$ ,
- $h(\cdot)$  is the estimated cost from node  $f$  to a leaf alist that instantiates the aggregate variable in the alist  $f$ ,
- and  $c(\cdot)$  is the estimated complexity weight of the aggregate to be performed by the alist during up-propagation.

By estimating the cost of expanding alist  $f$  in the tree, we are considering

1. the cost of instantiating variables in the root alist by going through alist  $f$ , and
2. the cost of aggregating the values of child nodes during upward propagation to the root through alist  $f$ .

We use natural numbers to express these costs. For alists that require simple aggregates such as addition or a boolean comparison, a cost  $c(f) = 1$  is incurred. Other function-generating aggregates such as REGRESS will incur  $c(f) = 3$ .

Additionally, we add a cost of 1 for every step that is estimated from the current node to get to an instantiated alist at the leaf for  $h(\cdot)$ . For instance, the *APPLY* aggregate requires that a function is inferred from its child nodes ( $h = 3$ ), and these child nodes in turn need to instantiate variables further down in the tree ( $h = 1$ ). This means  $h(f) = 3 + 1 = 4$ . Simpler aggregates such as *COUNT* require an alist for aggregation ( $h = 1$ ) and then have to instantiate variables in their child nodes ( $h = 1$ ) and hence  $h(f) = 2$ . *VALUE* on the other hand just needs the value of an instantiated variable, and so has  $h(f) = 1$ .

### 5.8.3 Optimality of FRANK Heuristics

We analyse the optimality of the FRANK heuristics using the conditions of *admissibility* and *consistency*. We discuss admissibility and consistency in FRANK using definitions in Russell and Norvig (2010).

For the purposes of the analysis, we will define equation 5.43 as:

$$l(f) = g(f) + z(f)$$

where,  $z(f) = h(f) + c(f)$  (5.44)

From equation 5.44, we split the cost into two parts; costs already incurred (i.e.  $g(f)$ ), and the cost that is estimated to be added if that alist is traversed (i.e.  $z(f)$ ).

A heuristic is admissible if it never overestimates the true cost of a solution via the selected node (Russell and Norvig (2010)). Given that  $z(f)$  is an estimate of the *minimum* number of steps needed to get to a leaf node before upward propagation starts, and an estimate of the complexity of propagating through the node,  $z(f)$  is always less than the true cost through  $f$ . This therefore makes  $l(f)$  admissible as well.

A heuristic is consistent if for every alist  $f$  and every child alist  $f'$  of  $f$  generated by a strategy  $S$ , the estimated cost of reaching the goal from  $n$  is no greater than the step cost,  $d$ , of getting to  $f'$  plus the estimated cost of reaching the goal from  $f'$ . That is,

$$z(f) \leq d(f, S, f') + z(f') \tag{5.45}$$

since  $c(f) > 0$ .

To show that our FRANK heuristics are consistent, we use the two-pronged approach similar to that used by Russell and Norvig (2010). First we show that  $z(f)$  along any path in the FRANK tree is non-decreasing. Using a similar argument to that in Russell and Norvig (2010), we suppose  $f'$  is a successor of  $f$ . Then  $g(f') =$

$g(f) + d(f, S, f')$  for some strategy  $S$ . We then have that

$$\begin{aligned}
 l(f') &= g(f') + z(f') \\
 &= g(f) + d(f, S, f') + z(f') \\
 &\geq g(f) + z(f) = l(f)
 \end{aligned} \tag{5.46}$$

Next, we need to show that whenever a node  $f$  is selected for expansion, the optimal path to that node has been found. In FRANK, unlike in a standard A\* search where there is specific goal node that is being searched for, we have no single node as a goal node other than a leaf node where instantiation first happens. Each alist that is selected for expansion is, therefore, expected to lead to such a leaf node state if one exists.

We could extend the heuristics further by learning the strategy selection process using techniques such as decision trees or neural networks. In this case, we can use the observed sequence of aggregates from several solved examples of queries to learn patterns given features in alists. This is however left as a future work to further enhance the capabilities of FRANK.

## 5.9 Summary

In this chapter, we discussed the decomposition rules and aggregates that FRANK uses to answer queries. Decomposition rules and aggregate functions determine how alists should be decomposed and also specifies how the child nodes should be combined during upward propagation respectively. These are the two key processes in FRANK to answer a query.

FRANK follows a few basic intuitions to tackle the decomposition of alists. First, a complex alist must be simplified prior to an attempt to ground variables. That is, only simple alists are expected at the leaves of the FRANK tree. The NORMALIZE strategy handles this intuition. Next, by making time a key feature of an alist, queries that have time attributes can be answered by decomposing the query along the temporal dimension. This gives us the ability to use techniques such as regression to infer numerical values that would otherwise not be found explicitly in the KBs.

The third intuition is that of thinking in terms of *parthood* relationships. That is, if a query cannot easily be solved for the whole, we can decompose the entity into its constituent parts, solve the problem for these parts for which facts may be readily accessed, and then combine to solve for the whole. We use this heuristic in

the geospatial decomposition strategy where we reason over geographical concepts. Finally, the lookup strategy allows us to improve our chances of finding a match to the query by finding related terms to the subject, property and objects attributes in the query such that they match facts in the KBs even if the original query's terms cannot be matched. Without this, search in DBs will be limited to exact string matches, which will yield less success, particularly, when working with data from different sources published by different authors.

We discussed the various aggregates that combine values in alists during upward propagation. We provided their type signatures when used to express user queries, explained their underlying arithmetic or statistical function and showed how they are used in the FRANK tree. During upward propagation of the decomposed alists, FRANK uses aggregate functions to combine the values of the alists to the root alist, from which an answer can be returned. Additionally, the uncertainty (cov) attributes of child alists are propagated upwards to the root of the FRANK tree (see chapter 6). Aggregates also include the generation of functions that give FRANK the capability of extrapolating from existing facts to predict data values that are not observed in the source KBs.

Finally, we explain the heuristics and algorithm for traversing the FRANK tree. We use the A\* search algorithm, in which the 'best' node is explored first. We define the best node as the lowest estimated cost node to explore. We calculate the cost of a node as the sum of the cost accumulated to get from the root node to the current node, the estimated cost from the current node to the leaf node and the complexity factor of the aggregate function of the current node. We also show the optimality of this heuristic by considering its consistency and admissibility.

In the next chapter, we explore uncertainty in FRANK and how it is addressed.



# Chapter 6

## Uncertainty in FRANK

### 6.1 Introduction

Uncertainty in answers from automated question answering (QA) over web data is increasingly becoming important given the rapid growth of Internet, the vagueness of data and the errors in some web knowledge bases (KBs). Information retrieval and inference aspects of QA have remained interesting challenges since the very early years of modern computer systems. As QA systems incorporate data from web sources into their inference processes, it is now important to acknowledge the uncertainty in the answers inferred and to communicate them effectively to the user.

In QA systems that rely on web KBs, most fail to address the uncertainty that comes with such noisy sources or missing values that have to be inferred. Many more (as discussed in chapter 2) fail to acknowledge the uncertainty in the inference methods used. Such systems avoid the uncertainty problem and rather focus on mere ranking of candidate answers. Not only does this create difficulty in interpreting the ranking in terms of the actual answer that a user sees, but it completely sidesteps the challenge of providing a meaningful way of informing the user about any errors in the answer returned.

The Functional Reasoning for Acquiring Novel Knowledge (FRANK) combines data from knowledge bases using a variety of inference techniques to answer queries. In an ideal world, the facts needed to answer the query, regardless of the source, will be precise and accurate. In addition, the inference operations and calculations performed on the data will be accurate in their model specification, operations will have sufficient data for inference such that approximation errors are avoided. However, this is not the case. FRANK uses data from diverse sources which lie on wide spectrum of accuracy,

ranging from those that can be trusted, to others created from crowd-sourced data that are very noisy, contain errors and (or) inconsistencies, and must be used with caution. FRANK also uses aggregate functions (section 5.5) that provide an approximate answer when no suitable answer is readily available in the KBs, e.g. regression. Such data heterogeneity and answer approximations introduce uncertainty, and as a result, uncertainty plays a significant role in the answers that FRANK finds or infers.

Uncertainty is unavoidable when answering queries over heterogeneous data sources and so providing an uncertain answer is acceptable in such cases as long as the user is informed about the degree of uncertainty.

Here, we work on the third aspect of our hypothesis:

**HYP-3:** *Acknowledging and dealing with uncertainty due to heterogeneous sources of data and inference techniques is necessary for placing the answers inferred in context.*

In this chapter, we focus on two main forms of uncertainty in FRANK:

- Imprecision of KB facts due to errors.
- Inference errors due to model misspecification and approximations errors resulting from the execution of inference operation and the propagation of values that variables are instantiated to in the FRANK tree.

In this chapter, we describe a probabilistic approach that we implement to tackle uncertainty in FRANK. We explain our underlying representation of uncertainty, the technique we use for uncertainty in the inference process and how this uncertainty is propagated through the inference tree. We limit our discussion of uncertainty to real-valued facts retrieved from KBs at the leaves of the FRANK tree, and the propagation of uncertainties through the FRANK tree. Nevertheless, during upward propagation of variables, our handling of uncertainty is applicable to all types of data provided the uncertainty attribute of the alist has been assigned a value. We do not treat uncertainty due to model misspecification in this project. This source of error is evident in our assumption that all data from all sources have an underlying Gaussian distribution. The problem of model selection in automated systems that has drawn active research such as the *automatic statistician* (Lloyd et al. (2014)) and the learning of model structure in a compositional way in (Grosse et al. (2012)) . These are of particular interest to our work on FRANK given the compositional nature of the language of models it uses. We could in future incorporate these techniques in FRANK.

## 6.2 Related Methods for Handling Uncertainty

Question answering over continuous, real-valued facts from web KBs usually involves information that is inherently vague. For instance, consider the sentence:

$$Q : \text{population}(\text{UK}, 2011) = 63M$$

that states that the population of the UK in 2011 is 63 million. Not only is there a vagueness about who counts in the population (given, say, the constant movement of people, or those in the process of being born), but with our current technology it's impossible to be accurate to a single person. This results in  $Q$  having a probability value that is infinitesimally close to 0, even if we change  $63M$  to another value. Therefore, in such as context, a probability value is not much help in conveying the uncertainty in a real-valued answer.

Probabilistic DBs, such as Trio (Widom (2004)), use a relational database model that considers a finite and fully observable dataset with probability values assigned to the records. Unfortunately, we do not have such well-defined KB with uncertainty pre-assigned to facts when answering queries over web data. Even if we did, we will have challenges combining such probabilistic DBs with data from other non-probabilistic KBs such as DBpedia and Wikidata. We will also not be able to convey the error bars in query results from just the probability values of records. We are therefore not able to use the data and query models of such probabilistic DBs.

Probabilistic logic focuses on such uncertain reasoning by a process of probabilistic entailment that formally calculates bounds on the probability of a sentence given a base set of sentences that constitutes its belief about the possible worlds. One can, for instance, estimate the uncertainty in  $Q$  by calculating the probability  $p(Q)$ . However, the probability of the assertion is,  $p(Q) \approx 0$ , given the vagueness of the answer and the fact that we are estimating a real-valued answer. Using a probabilistic logic approach, where we assign a probability to that assertion is, therefore, not the best approach in this domain since, these logics are built for expressing uncertainty about logical formulae, not the specific values within them. They are, therefore, not suited to the kinds of higher order reasoning with real-valued data required in query answering where uncertainty of the values are of interest. As a result, using these logics (and related ones such as Bayesian Logics Programs (Kersting and De Raedt (2002))), we are unable to express the uncertainty of the answer value.



What makes more sense is the probability that the answer lies in some range:

$$\begin{aligned} population(UK, 2011) \in \\ \{n | 63M - 0.32M < n < 64M + 0.32M\} \end{aligned}$$

But for our purposes of providing a meaningful answer to the user, it makes even more sense to invert this and say, for a fixed probability, what range the answer lies in. This is usually understood as the error bar on an answer.

So rather than calculating the probability of assertions in the KB or the inferences made (e.g., in probabilistic logic), we deal with uncertainty by considering the variances in the data observed when inferring answers. To formalise these notions, we need to commit to a statistical model. We assume that the data retrieved from the KB are normally distributed. We assume that the answer is the mean of this distribution and the variance accounts for the uncertainty.

Our use of a measure of variability in data as an estimate of uncertainty allows us to assign a prior variance to a KB's data values even before we see its data, and then update the variance as data is retrieved. This is known as the sequential estimation of the variance and is particularly useful in cases where a dataset is not fully observed and estimations are done online rather than in batch. Given that a knowledge base contains data of different kinds, it is inappropriate to use an unnormalized value as a prior variance attributed to the KB's data since the variance is scaled by the magnitude of the mean. Instead, we use a measure of relative variability known as the *coefficient of variation* (*cov*) calculated as the ratio of the standard deviation to the mean. The *cov* is often expressed as a percentage.

For instance, attaching a prior *cov* of 0.5% to real-valued data records in a KB means that if 63M is retrieved from the KB for the query  $population(UK, 2011)$ , then the error bar associated with this value is  $\pm 0.32M$ . This means that we can store our estimates of errors in the KB's data as a *cov*, and, for a property for which we do not yet have a prior variance, use the *cov* to calculate one once the specific quantity being measured is known. From there on, we can calculate our posterior variances using sequential estimation of the means and variances. We explain the *cov* further in section 6.5.1.

## 6.3 Factors Impacting Answer Accuracy

We first consider some of the factors that have an impact on the accuracy of answers returned by a QA system.

### 6.3.1 Source Data Errors

The use of web data for question answering offers much in terms of the wide domain coverage it brings to QA systems. First, the variety of the information from multiple sources and how they complement each other. For example, how YAGO (Suchanek et al. (2007)) extends DBpedia and how Linked Data (Bizer et al. (2009a)) provides a way to link resources in these datasets, thereby providing some form of grounding of concepts. Second, the sheer number of facts that can be retrieved to aid the QA process. For instance, DBpedia alone has over 500 million facts from a count of the triples in its RDF graph. Such a large set of knowledge sources and facts, however, comes at the cost of the accuracy of some facts. This lack of accuracy arises from human errors during the creation of the KB, or errors in algorithms that automate the elicitation of facts from documents that are converted into KBs (Wienand and Paulheim (2014)).

The sizes of these KBs are also too large for all facts they contain to be verified. As a result, some of these errors cannot easily be identified by a QA system that retrieves them without significant effort to detect them. Other similar source of KB errors arise from ambiguities in names of entities as well as outdated knowledge for which no temporal constraint in the data defines the truth value of the fact. Projects such as the *LOD Laundromat* (Beek et al. (2014)) are working to clean up data minimize these sources of errors.

Additionally, when multiple sources provide conflicting facts, an attempt to combine these facts to answer a query would lead to an error in the answer if no measures are taken to identify the erroneous fact, or to place a lower weight on the wrong fact given some prior knowledge of the credibility of KBs given the accuracy of facts that they contain.

### 6.3.2 Query Ambiguity

In other cases, the features of a query (subject, object, predicate, named entities, etc.) are ambiguous. For instance, a simple query that asks for the population of London, depending on the KB that is queried could return the population of London in the UK,

or London in Canada. Without additional context such as the personal profile of the user issuing the query, or a search log to identify the *London* with the higher frequency, it is nearly impossible to retrieve the correct answer without some guess work. Generally, disambiguation problems that arise in processing natural language text play a major role when processing natural language queries. Since we do not process natural language queries in FRANK, we do not focus on query disambiguation. However, we still have to deal with ambiguity of words (e.g. names of places or properties) in FRANK's formalized queries. Query disambiguation is an active area of research in web search (Mihalkova and Mooney (2009); Bordogna et al. (2009); Koutrika and Ioannidis (2005); Sanderson (1994)) as well as database search (Meng and Chu (1999); Bunescu and Pasca (2006); Cornolti et al. (2014)).

### 6.3.3 Model Misspecification

When finding answers from data, a model of the domain has to be specified as a framework that constrains how the QA system behaves. For instance, when answering queries from text data, the QA system must have the correct models to process such data. When using relational data, the appropriate query model has to be specified in order to avoid erroneous results. Similarly, when performing mathematical operations on numerical data retrieved from knowledge bases, the appropriate data models must be used given the type of data and the kind of inference or calculations being performed. In contrast to QA in well-defined domains where models of the underlying data and operations are properly outlined, QA over web data will often make general assumptions about data and operations which, although would work in several instances, could also lead to errors in the inferred answers. For instance, inferring functions from data usually leads to a model selection dilemma, where finding the appropriate model is non-trivial and in many cases requires an amply-sized data set to achieve. This is another reason why it is important to show the error bars associated with the answers returned to a user such that the answers are not misleading when the wrong models are used.

### 6.3.4 Approximation Errors

The inference operations in a QA system, particularly, ones that perform calculations on data, are prone to approximation errors when the calculations are not as trivial as additions, and multiplications. In the cases where functions are inferred and operations

are performed on these functions, approximate answers are unavoidable. As an example, if a regression function is to be generated from numerical facts retrieved from KBs, the number of data points retrieved will affect the precision of regression curve. If the wrong model is specified for the function (e.g. the wrong degree of the polynomial is chosen), then the approximation errors will be high. Techniques such as Gaussian Process (Williams and Rasmussen (1996)) regression can be used to minimize the impact of approximations by reducing the posterior variance in the regression curve around the observed data point. However, in areas where no data is observed, approximations errors will remain higher.

### **6.3.5 FRANK Error Sources**

The above sources of errors all impact FRANK some way. To a reasonable extent, however, we are able to limit the impact of some of these errors by (1) limiting the scope of knowledge domains to properly-structured datasets, (2) using a well defined grammar to express queries to eliminate some of the query ambiguity problems, and (3) restricting the kinds of models used by the inference operations to appropriate ones given the kinds of queries and data we evaluate in this project. Despite these measures to reduce the impact of errors in the inferred answers, there are still errors that are simply unavoidable. We categorise these as (i) accuracy of facts from KBs given their inherent noise (imprecision or errors in data) and missing data, and (ii) inference errors due to approximations and propagation of instantiated variables up the inference tree. We discuss these in sections 6.6 and 6.7 respectively.

## **6.4 What is Uncertainty**

We consider uncertainty as the extent of deviation of a fact or an inferred answer from its true value. In this section we explain our motivation for our approach to dealing with uncertainty in FRANK. We also provide a brief background to the Gaussian distribution that plays a key role in our work given that it provides a simple, yet effective way of representing this deviation around the data value being sought.

### **6.4.1 Motivation**

Our aim for calculating uncertainty is to estimate the error in an answer computed by FRANK. Rather than looking simply at the error in the individual facts, we consider

the inherent error in a KB as a whole, such that we can infer the accuracy of its facts from the uncertainty associated with the KB.

We assume that each KB has noise that is reflected as a deviation from the true value of the facts they contain. However, we have no distribution over the precision of the KB that we can use to estimate the error in facts retrieved from it. On the web, facts are not unique to KBs since data is duplicated in multiple datasets. For instance, the population data of countries can be found in datasets such as Dbpedia, WorldBank, Wikidata, and several others, some of which share the same source of the data. However, in our work, we have no definitive way of detecting duplicates of facts or facts that share the same publisher. We take a simple view of this and assume that each retrieved fact is independent of the other. Although projects such as DeFacto (Lehmann et al. (2012)) try to determine facts that corroborate each other by tracing the facts to a common source, this approach is not practical for our use in FRANK given that we make minimal use of natural language processing techniques. In any case, our independence assumptions of facts simplifies our ideas on uncertainty in this project.

We take an initial pessimistic view of the precision of KBs by assuming a large error in the data that they contain. As we retrieve more data from the KB, we update our prior error by computing a posterior error given new observations across all the KBs that FRANK retrieved data from. Thus, the posterior error of a KB is conditional on observations from all other KBs that FRANK finds data from.

After several observations, spanning multiple executions of FRANK for different queries, we expect that the posterior error attributed to a KB's fact will approach the actual error that exists given the facts seen so far. We also expect that the KB errors, when incorporated into the FRANK computations, together with the errors that are introduced from FRANK's aggregate functions when answering a query, reflect the error in the computed answer, i.e., the deviation of the calculated answer from the true value. We evaluate this hypothesis in chapter 8.

This approach means focussing not only on the precision of individual facts in a KB, but on the precision on the KB as a whole such that we can infer the error in a retrieved fact, by using assumed for all facts in the KB. The need for this general application of KB error to its constituent data requires us to use the Coefficient of Variation (section 6.5.1) as a normalized form of the error since a KB contains facts of different types and of different magnitudes. For purposes of implementation, we limit the KB uncertainty calculation to real-valued facts.

The sequential nature of updating the KB's prior error is similar to the techniques of sequential estimation using Bayesian methods. We therefore adapt these established techniques to our method. The Gaussian distribution, therefore, plays an important role. We provide a brief background to the Gaussian and related Bayesian inference technique in the next sections.

### 6.4.2 The Gaussian

The Gaussian distribution is a popular statistical distribution used mostly for its simplicity and how effectively it models uncertainty in different domains. Also known as the Normal distribution, its probability density function is defined by two parameter; a mean (and mode)  $\mu$ , and a variance  $\sigma^2$  in the form:

$$\mathcal{N}(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp^{-\frac{1}{2\sigma^2}(x-\mu)^2} \quad (6.1)$$

For a random variable  $X$ , the probability that the  $X$  takes the value  $x$ , written as:

$$p(X = x) = \mathcal{N}(x|\mu, \sigma^2) \quad (6.2)$$

means that the Gaussian distribution is centered at  $\mu$  and has a variance of  $\sigma^2$ . This variance can be inverted to give

$$\lambda = \frac{1}{\sigma^2}$$

in which case we talk about the distribution in terms of its precision. That is, the greater the variance, the lower the precision of the values inferred from the Gaussian. This idea plays a central role in our handling of uncertainty in FRANK as discussed in the next few sections.

Our motivation to use the Gaussian to model uncertainty in FRANK is based on its simplicity, ease of use and accuracy in modelling its diverse domains of application. (Jaynes (2003)) argues that the ubiquitous use of the Gaussian distribution is based on (a) how easy it is to interpret its parameters, (b) its appropriateness for modelling residual errors (noise) since the sums of independent random variables have an approximately Gaussian distribution, and (c) has a simple form with only a few assumptions about its parameter. There is also widespread use of Gaussians in approximate inference techniques such as variational inference methods (we explain why we do not use variational methods in section 6.6.2).

Although we base our estimation of uncertainty on the Gaussian distribution, we make a few assumptions, particularly, the independence assumptions for facts retrieved

from KBs, for this to work properly. Nonetheless, the Gaussian fits nicely with our technique for handling uncertainty in FRANK.

### 6.4.3 Bayesian Inference: Priors, Likelihoods and Posteriors

We paraphrase the explanations by (Tenenbaum (1999); Murphy (2012)) to describe Bayesian inference and sequential estimation of variables in a distribution. Consider a class of objects (also called a concept)  $C$  that a random variable is sampled from. For example,  $C$  is the set of even numbers. Let's also assume a set of observations  $D$ . The probability,  $p(x \in C|D)$ , (also written simply as  $p(x|D)$ ) is called the posterior predictive distribution of the random variable being sampled from  $C$ . In most instance, there are other candidate classes that  $x$  could belong to. These classes form the hypothesis space of concepts  $H$ . For example,  $h = \{\text{even numbers less than 50}\}$ . Let's assume  $H = \{h_1, \dots, h_n\}$ . The probability that the observations  $D$  are from a given hypothesis  $h$  is referred to as the likelihood of  $h$  given the observations  $D$  and is calculated as:

$$p(D|h) = \left[ \frac{1}{|h|} \right]^{|D|} \quad (6.3)$$

In FRANK, given that we are estimating the parameters of the Gaussian distribution for the observed data, we compute the likelihoods of the observations given values of the mean and the variance (see section 6.6.3).

However, before any observations are made, there is usually some *prior* probability  $p(h_i)$  of the hypotheses. A higher probability is naturally assigned to hypotheses that have a higher chance of explaining the data and a lower probability is assigned otherwise.

Based on Bayes' rule, we determine the posterior probability of the hypothesis,  $p(h|D)$ . The posterior is the probability of the hypothesis given the observations  $D$  and is given by the product of the likelihood and the prior, normalized by the joint probability of  $D$  and all  $h \in H$ . That is,

$$p(h|D) \propto p(D|h)p(h) \quad (6.4)$$

$$= \frac{p(D|h)p(h)}{\sum_{h' \in H} p(D,h')} \quad (6.5)$$

With enough data, the posterior has a mode on a single hypothesis known as the *maximum a posteriori* (MAP) estimate  $h^{MAP}$  given by:

$$\begin{aligned}
h^{MAP} &= \operatorname{argmax}_h p(h|D) \\
&= \operatorname{argmax}_h [p(D|h)p(h)] \\
&= \operatorname{argmax}_h [\log p(D|h) + \log p(h)]
\end{aligned} \tag{6.6}$$

As more and more data is observed,  $h^{MAP}$  converges to the maximum likelihood estimate (MLE) since equation 6.6 depends on the number of observations in  $D$  whereas the prior remains constant. As a result, the MLE hypothesis is given by

$$\begin{aligned}
h^{MLE} &= \operatorname{argmax}_h p(D|h) \\
&= \operatorname{argmax}_h \log p(D|h)
\end{aligned} \tag{6.7}$$

This means that as more data is observed, the MAP estimate converges to the MLE, regardless of the prior that is chosen. This result is vital for the sequential estimation of posteriors when data is observed in an online manner.

## 6.5 Uncertainty in FRANK

Uncertainty is central to how FRANK works and infers answers. Unlike some other QA techniques (Yahya et al. (2013), Preda et al. (2010), Lopez et al. (2012) ) where uncertainty is not treated, or is partly treated in only sections of the QA process (Brill et al. (2001), Walter et al. (2012), Tunstall-Pedoe (2010), Höffner and Lehmann (2014)) we incorporate the handling of uncertainty into the FRANK inference process. We assert that there is much more to be gained by acknowledging uncertainty and dealing with it as part of the QA process, rather than making it a limiting factor that prevents the use of noisy data for QA. By addressing this problem we open up the possibility of using a variety of KBs and inference operations, while accounting for noise in data as well as the approximations in calculation in the uncertainty we attribute to the final answer returned to the user.

### 6.5.1 Coefficient of Variation

Uncertainty in a FRANK alist represents the deviation of the inferred alist variable from its true value. We assume the real valued facts retrieved from KBs have an underlying Gaussian distribution hence the assumed true value is the posterior mean of the



retrieved values. We compute uncertainty from the posterior variance and we normalize it with the mean so that the variation between the inferred and actual for different data items in the FRANK tree is not affected by the magnitudes of the data values. We call this normalized standard deviation the *coefficient of variation (cov)* and it is calculated as:

$$cov = \frac{\sigma}{\mu} \quad (6.8)$$

This is a measure of relative variability and is often expressed as a percentage of the mean. For our use in FRANK, this provides four advantages compared with the use of the raw variance or probability values between 0 and 1 used in other inference systems.

(1) Since the *cov* is calculated from the parameters of the Gaussian distribution, it allows us to express the uncertainty in an answer as an error bar based on the standard deviation of the distribution. Whereas most inference systems dealing with uncertainty, based on probabilistic logic, fuzzy logic or related logics, are concerned with the uncertainty of *assertions* in the KB or facts inferred, we are interested in the uncertainty of the *values* of the answers calculated. For the domains we are interested in (socio-economic development), the probabilities of any particular real value will be infinitesimally close to zero and will not be meaningful in describing the uncertainty of the value. Error bars, however, express the variability in the data and give a sense of how far a data value could be from its true value. Error bars are, therefore, a meaningful representation of uncertainty for the datasets we work with, and will be familiar to most users. Also, since the mean exists in the alist (usually as the *object* attribute), we can easily recover the variance from the *cov* during propagation of uncertainty up the tree.

(2) A probability value  $\{p|0 \leq p \leq 1\}$  shows how probable the answer is, but does not indicate by how much the answer could be wrong. Many QA systems including IBM's Watson Ferrucci et al. (2010); Murdock et al. (2012) and Microsoft's AskMSR Brill et al. (2001); Banko et al. (2002) return a probability value or a ranking score attached to their answers. We believe that the semantics of uncertainty in FRANK and its representation offers a better interpretation of the accuracy of an answer than a generic score used to rank the candidate answers. That is, we can attribute meaning to FRANK's *cov*, whereas a ranking score cannot be easily interpreted relative to the answer that a user sees. Also, since the *cov* is expressed as a percentage of the answer (i.e., the mean value), it easily explains how much error there is, potentially, in an

answer. So an answer with a higher *cov* is more uncertain than an answer with a lower *cov* and by turning this into a variance, the user can see an upper and lower bound of the answer.

(3) The normalization of the variance as *cov* allows us to compare and propagate variance in the inference tree regardless of the magnitude of the means of the inferred data values. For instance, if the FRANK tree has alists containing data for *birth rate* and other alists for *population*, the *cov* allows us to uniformly express the uncertainty of these alists in the same FRANK tree, even with data on different scales. This also means that we can compare the error bars of answers to queries in a uniform way irrespective of the magnitude of the answers. The raw variance, however, is scaled by the magnitude of the answer and so does not lend itself to easy comparison of uncertainty of different answers.

(4) A heuristic employed in our estimation of uncertainty in FRANK's answers is the use of previously calculated error bars in data from a KB as initial estimates of variances in a new quantity that we retrieve from a KB for which do not already have a prior variance. Although we use variances in the data for calculating uncertainty, our ability to systematically use these variances as estimates of priors for quantities that we have not previously seen from the KB is limited since variance is scaled by the magnitude of the mean. Our use of the *cov* as a relative variability measure circumvents this limitation. That is, we can use the *cov* assigned to a KB's data values as the prior variance of the data values in a KB when we encounter a property for which we have not previously retrieved data. Section 6.6.3 and the worked example in table 6.1 shows how we do this.

The *cov* attribute in a FRANK alist is dedicated to uncertainty. FRANK first instantiates the *cov* at the leaves of the FRANK tree using the variances (equation 6.25) computed from the knowledge sources (section 6.6). Our goal is to make FRANK update its belief about the uncertainty of facts given what it has previously observed from knowledge bases in a justified way. In the sections that follow, we look at how we calculate uncertainty for real-valued facts. The *cov*, as with other alist variables, is propagated from the leaves to the root of the FRANK tree as discussed in section 6.7.

For real-valued answers, we communicate uncertainty to the user by multiplying *cov* by the answer to obtain the standard deviation from the answer. For example, for a query that returns the answer 27,650,000 with a *cov* of 15%, we interpret its *cov* as  $\pm 15\%$  of the correct answer, and in this example, the uncertainty is the error bar  $\pm 4,147,500$ .

### 6.5.2 Approximating Priors from *cov*

Computations in FRANK span data in different scales and units. Calculating standard deviation using traditional methods means that we have no uniform way of comparing how much error one KB has given different kinds of data retrieved from it. This is especially true when the data are on different scales. However, in order to uniformly handle prior and posterior variances for KBs after data is retrieved for different queries, we need to express the uncertainty in a more generic way, irrespective of the magnitude of the data. This is achieved by the normalization of the standard deviation to give us the *cov* as described in the section above. By doing this we get a sense of the error in the retrieved fact relative to the magnitude of the data that is retrieved. We use this mechanism to store our prior knowledge about uncertainty in a KB. Hence, for data value  $v_\rho$  retrieved about a property,  $\rho$ , from a KB with a *cov* of  $\epsilon$ , we can estimate a prior error  $\sigma$  of

$$\sigma = v_\rho \times \epsilon \quad (6.9)$$

That is, we can recover the value of the prior standard deviation from the *cov* by multiplying the retrieved data by the prior *cov*.

### 6.5.3 Assumptions

In this project, we deal with uncertainty within the context of real-valued facts. However, the techniques developed here can be extended to non-real-valued facts by selecting the appropriate statistical distributions. The main assumption we make in our approach to determining uncertainty is that each numerical fact retrieved from a KB is an observation of the true value with additive Gaussian noise, where the noise depends on the source of the fact. We also assume that the facts are independently and identically distributed (i.e. generated from the same underlying statistical distribution).

That is, we assume that all real-valued observations of a specified property from the KBs are independent and normally distributed with mean,  $\mu$ , and variance,  $\sigma^2$ . Given that the KB's entire data records is not fully observed by FRANK, the normal distribution is a reasonable assumption to make for the data records for each of the real-valued properties in the KB. Although this assumption may not necessarily hold in every case, it allows us to work with existing probabilistic techniques in a reasonable way. Such an assumption, in any case, is not unique to FRANK. For instance, in Trio's data model, individual attribute values for real-valued and ordered domains are either exact values

or approximations. The Gaussian distribution over a range of possible values denoted by a mean/standard-deviation pair (Faradjian et al. (2002)) is used as one such approximation. The uniform distribution across values in a minimum/maximum range is an alternative approximation (Widom (2004)).

In FRANK, we prefer the Gaussian approximation since it is a better model of the natural occurrence of real-valued facts for a given entity's property across multiple KBs. In particular, when we do not know what the true underlying distribution of the data is, the Gaussian approximation is a reasonable assumption to make. We could, in future, use techniques explored in (Ghahramani (2015)) to discover plausible models from data automatically. For now, we can map the parameters of the Gaussian distribution into FRANK alists such that the *cov* encodes the variance and the *subject*, *object* or *time* represent the mean values, depending on which one is the projection or operation variable in the alist.

We also expect that inference operations in FRANK will introduce errors into the inference process, each to a different extent. We also assume that although the use of heuristics in selecting rules for decomposition also leads to potential errors, such errors affect the corresponding inference operations that perform calculations on alists. Although this is a strong assumption, it simplifies our estimation of uncertainty in FRANK as explained in the sections that follow.

## 6.6 KB/Fact Uncertainty

In the sections that follow we show how we incorporate and calculate uncertainty about knowledge bases in FRANK.

### 6.6.1 FRANK Algorithm

Our technique for estimating the uncertainty of answers follows from our FRANK algorithm which works in two parts. First, a tree search algorithm recursively decomposes nodes at the frontier of the inference tree until nodes are grounded in facts in a KB. Next, the retrieved facts are propagated up through the inference tree where aggregate functions are applied and uncertainties from the inputs are propagated. This ends with the root of the inference tree receiving an inferred answer and the corresponding answer uncertainty.

### 6.6.2 Choosing a Bayesian Method for FRANK

Computing the uncertainty in FRANK is one of several processes taking place to find an answer to a query. As such, we want a computationally lightweight technique that does not significantly slow down the assignment of KB data to variable and the upward propagation of instantiated variable. FRANK attempts to estimate the probability distribution of the errors in a KB by comparing its data to data retrieved from other KBs. That is, FRANK attempts to estimate the parameters of the Gaussian distribution on the variance of the dataset in a KB.

There are several other probabilistic techniques for dealing with this problem of estimating the parameters of the underlying distribution, including variational methods (Gibbs and MacKay (2000)) and Laplace approximations (Rue et al. (2009)). However, methods such as the above require more complicated computations beyond the current needs of FRANK, especially with the sequential nature of the variance estimations we are making in FRANK. There is also very little value in working harder to calculate accuracy that would be spurious given the inherent large error bars that exists in the diverse web knowledge sources.

Our approach, on the other hand, based on the Bayesian inference using prior distributions on the parameters and updating them as data is retrieved, works in a straightforward manner, given our intuitions of the kind of uncertainty we are dealing with. The math involved simplifies to additions and multiplications as shown in equations 6.14 and 6.25. We show in our evaluations of FRANK that although this is a computationally simpler method than others mentioned above, we obtain realistic uncertainties when compared to the actual errors between the answer FRANK infers and the gold standard answer found in the KB.

Possibly, in future work, we will take a more detailed look at other techniques for estimating the parameters of a distribution in an online manner as we do in FRANK.

### 6.6.3 Calculating and Propagating KB Uncertainty

We use a Bayesian approach to compute the uncertainty associated with an inferred answer. This complements the design of the FRANK algorithm such the uncertainty is updated as more data is retrieved in a sequential estimation manner with each query that is processed in FRANK. Our calculations below are base on derivations in (Bishop (2006)).

Suppose that for a particular leaf node in the FRANK tree, we require a numeric

fact  $f$ . Let's assume that  $s_1, s_2, \dots, s_n$  are sources of facts, i.e. knowledge bases (KBs). Let  $x_{i,j}$  represent the  $j^{\text{th}}$  fact from source  $s_i$ . We assume that all facts from the KBs are independent, and normally distributed with mean,  $\mu$ , and variance,  $\sigma^2$ . Although this assumption may not necessarily hold in every case, it allows us to work with existing probabilistic techniques in a reasonable way.

We use a random variable  $X^i = \{x_{i,1}, x_{i,2}, \dots, x_{i,j}, \}$  to represent all facts from  $s_i$ , and for each  $x_{i,j}$ ,

$$x_{i,j} \sim \mathcal{N}(\mu_i, \sigma_i^2) \quad (6.10)$$

During inference, FRANK determines the uncertainty associated with each KB and its returned facts. Since FRANK selects the posterior mean value (considering both the prior mean and current observations as in equation 6.12) of the list of facts for inference, we estimate uncertainty based on the variance of the retrieved facts. For notational simplicity in the equations that follow, we refer to the inverse of the variance as *precision* ( $\lambda$ ). That is,  $\lambda = 1/\sigma^2$ . Our technique is based on ideas from the Expectation-Maximization (EM) algorithm (Dempster et al. (1977)) and the Bayesian inference for the Gaussian distribution using sequential estimation. These techniques use prior distributions based on previously seen data, and then update the priors to get posterior distributions as new data is retrieved.

A Bayesian treatment allows us to introduce priors over the parameters in equation (6.10). Our aim is to determine the posterior mean (the assumed true value) and *cov* of a fact given the facts retrieved from KBs. This allows us to take a simple approach to update the parameters sequentially as FRANK retrieves more data in each iteration of the algorithm. An diagram showing how we use prior distributions of sources and properties is show in figure 6.1.

We begin by assuming that the variance,  $\sigma^2$ , of the underlying Gaussian distribution of the true value is known. Our goal in this first step is to infer the mean of this distribution. We obtain the 'assumed' variance from the *cov* as follows:

$$\sigma^2 = \sum_{i=1}^k (cov^i \times \mu_{ML})^2 \quad (6.11)$$

where  $cov^i$  is the *cov* for KB  $s_i$  and  $\mu_{ML}$  is the maximum likelihood mean of the subset of all observations from KB  $s_i$ , where  $i \in \{1, \dots, k\}$ .

Our aim is to find the parameters of the posterior distribution  $p(\mu|X)$  given by:

$$\begin{aligned} p(\mu|X) &\propto p(X|\mu)p(\mu) \\ p(\mu|X) &= \mathcal{N}(\mu|\mu_N, \sigma_N^2) \end{aligned} \quad (6.12)$$

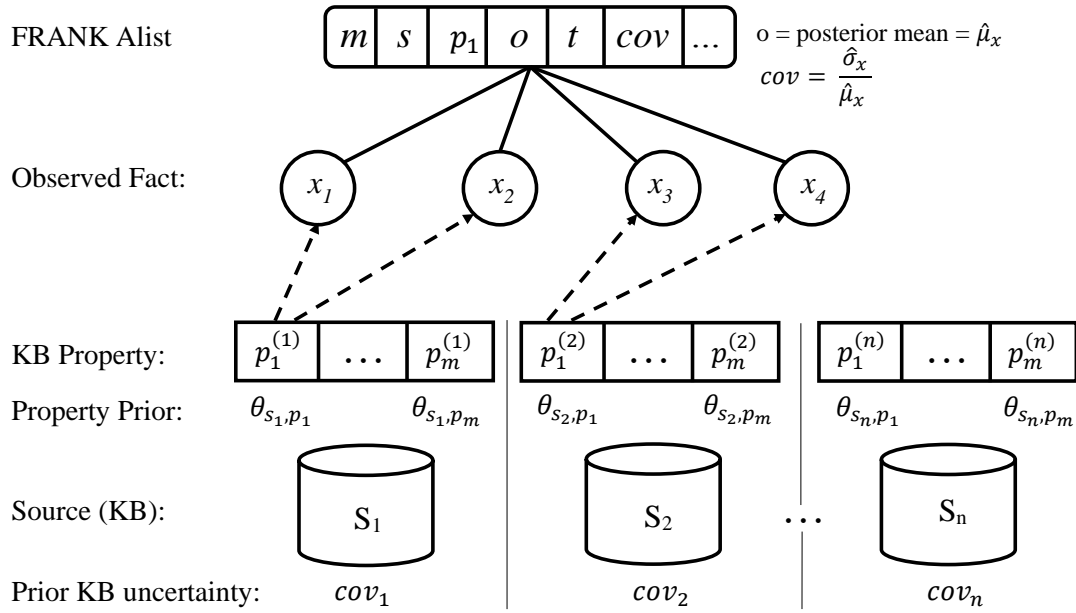


Figure 6.1: Priors over KB sources and properties.

where the prior is given by:

$$p(\mu) = \mathcal{N}(\mu | \mu_0, \sigma_0^2) \quad (6.13)$$

We use an ‘improper’ prior  $\mu_0$  for each KB by taking the maximum likelihood (ML) mean of the observations from that KB. Ordinarily, the prior mean will be obtained from prior knowledge of the specific data that is retrieved. However, in our case, the dataset in a KB is so diverse that it is impossible to find an existing prior mean. We therefore use the ML mean of all the data that is retrieved from a KB as its prior. The parameters  $\mu_N$  and  $\sigma_N^2$  of the posterior distribution of the mean are given by:

$$\mu_N = \frac{\sigma^2}{N\sigma_0^2 + \sigma^2}\mu_0 + \frac{N\sigma_0^2}{N\sigma_0^2 + \sigma^2}\mu_{ML} \quad (6.14)$$

$$\frac{1}{\sigma_N^2} = \frac{1}{\sigma_0^2} + \frac{N}{\sigma^2} \quad (6.15)$$

where  $N = |x_{i,1}, \dots, x_{i,n}|$  is the number of observations from the KBs in the current iteration.

We initialize  $p(\mu)$  (with its hyper-parameters  $\mu_0$  and  $\sigma_0^2$ ) as a prior, and then update the parameters with the posteriors (in equations 6.14 and 6.15) once data is retrieved.

In the second step, we estimate the uncertainty associated with the distribution of the true value. We assume that the mean is known (we use equation 6.14 as the ‘assumed’ mean), so we infer the variance (precision).

Following a similar process as above, our posterior distribution of precision is given by:

$$p(\lambda|X) \propto p(X|\lambda)p(\lambda) \quad (6.16)$$

and the likelihood function with the given mean  $\mu_N$  is expressed as:

$$p(X|\lambda) = \prod_{n=1}^N \mathcal{N}(x_n|\lambda^{-1}) = \frac{\lambda^{N/2}}{(2\pi)^{(1/2)}} \exp\left\{-\frac{\lambda}{2} \sum_{n=1}^N (x_n - \mu_N)^2\right\} \quad (6.17)$$

We use the Gamma distribution (defined below in equation 6.6.3) as the conjugate prior for the likelihood function since the likelihood is a function of the product of a power of  $\lambda$  and the exponent of a linear function of  $\lambda$ .

$$p(\lambda) \sim \text{Gam}(\lambda|a, b) \quad (6.18)$$

where

$$\text{Gam}(\lambda|a, b) = \frac{1}{\Gamma(a)} b^a \lambda^{a-1} \exp(-b\lambda) \quad (6.19)$$

and

$$\Gamma(x) = \int_0^{\infty} u^{x-1} e^{-u} du, \Gamma(1) = 1, \text{ and } \Gamma(x+1) = x! \quad (6.20)$$

We initialize  $\lambda^{(i)}$  for a source  $s_i$  as an uninformative prior and store the parameters  $a^{(i)}$  and  $b^{(i)}$ . For retrieved facts from  $s_i$  we compute  $\lambda^{(i)}$  from equations (6.18) to (6.20) as the prior precision of the source.

Next, we estimate the mean  $\mu$ , the true value of the observation using its prior parameters  $\mu_0$  and  $\sigma_0^2$  as:

$$\mu \sim \mathcal{N}(\mu_0, \sigma_0^2) \quad (6.21)$$

where  $\sigma_0^2$  is the prior variance. Next, we update the source precision using the maximum likelihood mean of all observation from all sources. For source  $s_i$ , we use the stored parameters  $a^{(i)}$  and  $b^{(i)}$  as its prior precision distribution:

$$\lambda_0^{(i)} \sim \text{Gam}(a_0^{(i)}, b_0^{(i)}) \quad (6.22)$$



We obtain the posterior precision as follows by updating the parameters given the new observations as follows:

$$a_N^{(i)} = a_0^{(i)} + N/2 \quad (6.23)$$

$$\begin{aligned} b_N^{(i)} &= b_0^{(i)} + \frac{1}{2} \sum_{j=1}^n (x_j - \mu_N)^2 \\ &= b_0^{(i)} + \frac{1}{2} \sigma_{ML}^2 \end{aligned} \quad (6.24)$$

where  $\sigma_{ML}^2$  is the mean of observations from all sources.

The estimate of the posterior precision is given by the expectation of the precision:

$$\lambda_N^{(i)} = \mathbb{E}[\lambda_N^{(i)}] = \frac{a_N^{(i)}}{b_N^{(i)}} \quad (6.25)$$

Finally, our posterior  $cov_N$  is calculated as:

$$cov_N = \frac{\sqrt{\lambda_N^{-1}}}{\mu_N} \quad (6.26)$$

The posterior precision is propagated back in the inference tree and the posterior  $cov$  is saved and will be used as prior variance in future observations from the KB.

#### 6.6.4 Worked Example

We explain the calculation of uncertainty further with a worked example in Table 6.1. Suppose we have a query that demands the value of the UK's population in 2013. We assume that matching facts are found in two knowledge sources  $S_1$  and  $S_2$ . We assume that FRANK is in an initialized state for these two knowledge sources such that each has a default  $cov$  of 1.0. After searching the knowledge bases, we retrieve the values  $\{63900000, 64100000\}$  from source  $S_1$  and  $\{64000000, 63800000, 64200000, 63500000\}$  from  $S_2$ . In the first part of the algorithm, we assume our variance is known, and therefore estimate our posterior mean. In line 3, we obtain our prior means using the ML means of retrieved data from each source. Using the ML mean  $\mu_{ML}$  for the observations from both  $S_1$  and  $S_2$  in line 4, we recover the prior variances for each source by multiplying the  $cov$  by the  $\mu_{ML}$  and squaring the result (line 5). We apply equation 6.14 to calculate the posterior mean in line 7. This is the mean value given our prior  $cov$  and the data received from the two KBs.

In the next step, we update our  $cov$ . We assume that we know the mean of the distribution and use  $\mu_N$  calculated in line 7 for this purpose. We first initialize the

parameters  $a_0$  and  $b_0$  of the Gamma distribution in lines 8 and 9 and update these parameters using equations 6.23 and 6.24 in lines 11 and 12 respectively. We are now able to calculate the posterior precisions in line 13 using equation 6.25. Finally, we obtain the posterior *covs* in line 14. We store the updated *covs* locally and reuse them whenever we retrieve data from these KBs. The posterior variances (precisions) are passed on to the respective uncertainty attributes of the alist whose variable is initialized by data from the KBs.

#	STEP	KB $S_1$	KB $S_2$
1	<i>cov</i>	1.0	1.0
2	Observations	{63900000, 64100000}	{64000000, 63800000, 64200000, 63500000}
Step 1: Estimating the posterior mean value of the observation, using the <i>cov</i> as the prior variance			
3	$\mu_0 = \mu_{ML}^{(i)}$	64000000	63875000
4	$\mu_{ML}$	63916667	
5	$\sigma_0^2 = (cov \times \mu_{ML})^2$	$4.096 \times 10^{15}$	$4.080 \times 10^{15}$
6	$\sigma^2$	$1 \times 10^{10}$	$6.6875 \times 10^{10}$
7	<b>Posterior Mean,</b> $\mu_N$ (from eqn 6.14)	$6.3917 \times 10^7$	$6.3917 \times 10^7$
Step 2: Estimating the posterior <i>cov</i>			
8	$a_0$	1.0	1.0
9	$b_0$ (set to $\mu_0$ )	64000000	63875000
10	$\sigma_{ML}$	$1.694 \times 10^{10}$	$7.028 \times 10^{10}$
11	$a_N$ (eqn 6.23)	2.0	3.0
12	$b_N$ (eqn 6.24)	$8.536 \times 10^9$	$3.520 \times 10^{10}$
13	$\lambda_N$ (eqn 6.25)	$2.343 \times 10^{-10}$	$8.552 \times 10^{-11}$
14	<b>Posterior <i>cov</i>,</b> $cov_N = \frac{\sqrt{\lambda_N^{-1}}}{\mu_N}$	$1.002 \times 10^{-3}$	$1.695 \times 10^{-3}$

Table 6.1: Worked example

### 6.6.5 Working with uncertainty in FRANK

The core of the FRANK algorithm involves recursively decomposing FRANK nodes in order to ground them in facts in a KB, and then up-propagating the answers the root of the inference tree. Since we are able to estimate the uncertainty each fact that the FRANK node is grounded in, our goal is to up-propagate these uncertainty values through to the root of the inference tree. This helps us estimate the uncertainty in the final answer that is inferred. Our decision to model uncertainty as a Gaussian over each fact allows us to combine the uncertainties from child nodes to their parent in closed form. That is, if the parent node applies an aggregate function over the values of its child nodes, we are able to combine the uncertainties in a likewise fashion. For instance, we can add Gaussians, find differences and products. In situations where we only select one of many child nodes, such as max, min, etc, we simply use the uncertainty associated with the selected node. That is, we are interested in the uncertainty associated with the particular alist of interest.

Additionally, due to our handling of uncertainty with Bayesian techniques, we are able to sequentially estimate the posterior uncertainty in KBs as more and more data are retrieved from them. Although we start off with an uninformative prior that is uniform for all KBs, as we query KBs, we adjust our prior uncertainty for them with the posteriors that we estimate.

## 6.7 Inference Operation Uncertainty

Aggregation functions in FRANK may contribute to uncertainty in the final answer. We categorize aggregate functions in FRANK into two types:

1. Exact aggregates
2. Approximating aggregates

### 6.7.1 Exact Aggregate Functions

These are methods based on functions that perform non-approximating operations on nodes. In addition, these aggregates do not generalize from their child nodes. Examples of such aggregates include MAX, MIN, GT, LT, LOOKUP, VALUES, AVG. Since these aggregates do not approximate their returned values, they do not introduce errors in the final answer inferred and hence, do not contribute any uncertainty to the final

answer. However, these aggregates propagate the uncertainties of their child nodes up the FRANK tree.

### 6.7.2 Approximating Aggregate Functions

These are aggregates that generate functions from their child nodes in the inference tree and infer new values from them. Inference methods of this type include REGRESS (Regression), GP (Gaussian Process), DERIV (Derivative), INTEGRATE (Integration). These aggregates take as inputs FRANK nodes that contain either *values* representing facts, or functions inferred by other methods. Given that these aggregates extrapolate from functions they generate, or simply reuse previously generated functions, they are likely to introduce errors in the answers they return to their parent nodes in the inference tree.

We describe our approach to estimating and propagating uncertainty for inference methods below.

### 6.7.3 Methodology

Our technique is based on the assumption that aggregate functions that generate functions for approximation also indicate the error bars in their functions or values obtained by using the inferred function.

For example, with the REGRESS aggregate function, FRANK performs regression (linear or polynomial) regression using the values from its child nodes in the inference tree. In regression, the Mean Squared Error (MSE) is the sum of squares of the deviation of the dependent variable observations from the fitted function (residuals). When the MSE is divided by  $n - 1$ , where  $n$  is the number of observations, we obtain the unbiased sample variance. This variance conveniently serves the purpose of our estimated uncertainty since it indicates the error bar around the regression function that has been inferred.

Similarly, with our GP aggregate function, we obtain an error bar from the inferred function. A Gaussian Process (GP) (Williams and Rasmussen (1996); Rasmussen (2006)) generates data located through some domain such that any finite subset of the range follows a multivariate Gaussian distribution. The GP gives a distribution over the underlying function that the retrieved data represents. For each independent variable, we can sample the GP for the function. As we generate several samples from this distribution, we observe variances for different independent variable points selected on

the function, which results in different function curves. The point variances gives us the uncertainty that the GP aggregate function contributes in FRANK.

## 6.8 Propagating Uncertainty

### 6.8.1 Uncertainty from child nodes

FRANK propagates the uncertainty from its leaf nodes, through intermediate aggregate function nodes, to the root node of the inference tree to obtain the uncertainty of the answer inferred. In addition to the uncertainty that the aggregate functions introduce, the child nodes of the aggregate functions may also contain uncertainties calculated from further down the inference tree. We therefore have to combine uncertainty values to obtain the uncertainty that is propagated from a given node to its parent node.

Graphically in Figure 6.2, we combine the uncertainties  $\sigma_1^2, \sigma_2^2, \sigma_3^2$  of nodes  $x_1, x_2$  and  $x_3$  respectively with the uncertainty  $\sigma_I^2$  resulting from the aggregate function  $I$  to determine the uncertainty  $\sigma_p$  of node  $p$ . To simplify this propagation, we maintain a Gaussian representation of the uncertainty values in the aggregate function nodes. Each node in the inference tree represents a Gaussian where the mean is the data value of the node and the variance is the uncertainty of the node. It allows us to combine the uncertainties in closed form; that is, we can combine uncertainties in Gaussian form and obtain another uncertainty value which is Gaussian.

For example, suppose aggregate function  $I$  sums up two nodes  $A$  and  $B$ . If  $A$  and  $B$  have variances  $\sigma_a^2$  and  $\sigma_b^2$  respectively, then the variance of combined aggregate function:

$$\sigma_{A+B}^2 = \sigma_A^2 + \sigma_b^2 + 2\sigma_{AB}^2 \quad (6.27)$$

where  $\sigma_{AB}^2$  is the covariance of  $A$  and  $B$ . However, to further simplify our calculation without significantly impacting the correctness of our method, we assume conditional independence<sup>1</sup> of the data values of child nodes of a specified node in the inference tree given the inference strategy applied. This means that

$$\sigma_{AB}^2 = 0 \quad (6.28)$$

---

<sup>1</sup>This may be a strong assumption and could be unrealistic in some cases where the same data is propagated through different inference strategies.

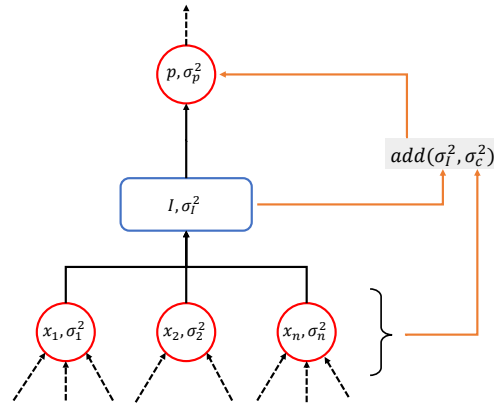


Figure 6.2: Combining Uncertainty

Combining aggregate function uncertainty with the child node uncertainties. Each node  $(\alpha, \sigma_\alpha^2)$  in the illustration shows two attributes from the FRANK alist:  $\alpha$ , an identifier of the node and  $\sigma_\alpha^2$ , its uncertainty .

---

This gives us

$$\sigma_{A+B}^2 = \sigma_A^2 + \sigma_B^2 \quad (6.29)$$

Further examples of these closed-form variance propagations are shown in the Appendix A.

## 6.8.2 Combining uncertainty of aggregate functions and their child nodes

Assume, as illustrated in Figure 6.2, that we have three FRANK nodes  $x_1$ ,  $x_2$  and  $x_3$  with uncertainty values  $\sigma_1^2$ ,  $\sigma_2^2$ , and  $\sigma_3^2$ . Given that their parent node is an aggregate function  $I$  that performs a function that results in a variance of  $\sigma_I^2$  in the inferred value, we propagate the uncertainty to the parent node  $P$  as follows.

First, we obtain the uncertainty that results from the application of the aggregate function to the child nodes to get  $\sigma_I^2$ . Next, we combine the variances of the child nodes by applying a function that corresponds to the one performed by the aggregate function. This gives the variance of the combined child nodes,  $\sigma_c^2$ . The uncertainty propagated is the greater of the two given by adding the two variances:

$$\sigma_p^2 = \text{add}(\sigma_I^2, \sigma_c^2) \quad (6.30)$$

This is performed recursively through the inference tree from the leaves to the

root. The standard deviation at the root is normalized with the data value to obtain a Coefficient of Variation  $cov$  and is calculated as:

$$cov = \frac{\sigma_{root}}{\mu_{root}} \quad (6.31)$$

where  $\mu_{root}$  is the inferred data value in the root node (i.e. the answer) and  $\sigma_{root}^2$  is the uncertainty propagated to the root node. The  $cov$  allows FRANK to express uncertainty in a form that can be compared to answer uncertainties for other queries.

There are other policies, considered for future work, that can be specified for combining uncertainties. For instance we can reduce the uncertainty if different OR branches result in similar answer. That is, when different strategies for solving a query result in the same answer, then we can have higher confidence in the answer.

### 6.8.3 Communicating Uncertainty to User

We translate the  $cov$  to a value relative to the real-valued answer by multiplying  $cov$  by the answer to obtain the standard deviation from the answer. For example, For a query that returns the answer 27,650,000 with a  $cov$  of 0.15, we express the uncertainty as the error bar  $\pm 4,147,500$ .

Although our treatment of uncertainty is focussed on real-valued facts, we do encounter scenarios where a non-real attribute (e.g. the subject of the alist) is projected. For such non-real answers, we simply state the  $cov$  as a fraction and interpret it as the likelihood that the answer is correct given the deviation of the supporting facts in the child nodes from the true answer. We hope to refine this in the future in order to provide a better interpretation of the  $cov$  for all answer types.

## 6.9 Complexity of $cov$ calculation

The  $cov$  is calculated in polynomial time of the number of knowledge bases and the number of data values retrieved from each KB. The significant steps of the sequential Bayesian update process, such as updating the hyperparameters of the priors, calculating the variances, Gamma functions and the combination of the variances are based on mathematical operations that have polynomial time complexity in the size of their inputs. These operations are performed for datasets from each of the knowledge bases.

## 6.10 Summary

FRANK incorporates uncertainty directly into its computation of answers to queries that require real-valued data. The vast set of data available on the web for automated query answering means that there is now a huge opportunity to incorporate data from multiple sources in query answering. However, the variety of sources, the differences in representation, the noise in data and several other factors means that uncertainty in answers is unavoidable. Rather than placing uncertainty simply as a by-product of the inference process, we embed the calculation of uncertainty directly into the inference process such that data values and uncertainty are both propagated through the FRANK tree.

This chapter looked at the two main kinds of uncertainty that we encounter in FRANK: (1) imprecision in the knowledge source data due to noise (2) uncertainty introduced by function approximations in the inference operations in FRANK. Errors in the stored data in the KBs from which we retrieve data means that answers will also be wrong. By using data across multiple data sources, and assigning different weight of trust to these sources, we can comfortably use different KBs given that FRANK can report back to the user on the uncertainty in the answer it returns. Aggregate functions that approximate data values also introduce errors into the FRANK process. In particular, regression aggregate for prediction introduce error bars that need to be propagated along with the data values inferred. Both categories of uncertainty are captured in the uncertainty attribute of the FRANK alist and are propagated as part of the FRANK inference algorithm.

We use of the Gaussian distribution to model uncertainty in the data sources. We also apply the Bayesian inference techniques that allows for sequentially updating the parameters of the mean and variance distributions as data is retrieved. Since we deal with variance for different kinds of data, we use the coefficient of variation, a ratio of the standard deviation to the mean, to store variance of KBs irrespective of the magnitude of the actual data values retrieved. This allows us to sequentially update the *cov* after data values are retrieved from the KB. We make a number of assumptions to make the calculations simple to implement. In particular, we makes assumptions of independence and conditional independence of retrieved data given the sources and the query being solved.





# Chapter 7

## Implementation

### 7.1 Introduction

In this chapter, we first look at the implementation of the ideas discussed in the previous chapters and the software engineering tasks involved. We explain the distributed architecture that we use to achieve the ability to easily extend the capabilities of FRANK with minimal changes to the core inference process.

Our work in this project has been centred on the Functional Reasoning for Acquiring Novel Knowledge (FRANK). In FRANK, we utilize the data on the web while acknowledging the uncertainty that exists in the data and the inference processes involved in processing and propagating this data through alists in the FRANK tree. Also, by incorporating an estimation of the uncertainty of answers directly into the answer computation process, we can give users a better interpretation of the answer. This transforms the query answering process from one of a large black-box process, to one where the inference process is traceable and, in many some ways, provides a reasonable justification for the answer it returns. These objectives and features of FRANK influence our implementation of the framework. For instance, our use of alists in FRANK means that we can represent different kinds of data from any web knowledge based regardless of the source representation. This means that the core FRANK algorithm works so long as we are able to capture the required facts in a FRANK alist. This also means that we can extend FRANK to KBs in other domains, add decomposition rules and aggregate functions without significant modifications (if any at all) to the FRANK algorithm in 4.

This chapter deals with the fourth aspects of our hypothesis:

**HYP-4:** *An extensible algorithm and architecture allows the QA frame-*

*work to be easily extended to new data sources and inference operations.*

In the sections that follow, we explain the key requirements for FRANK and the software architecture choices made. We also discuss the tools, frameworks and third-party libraries used to implement the components of the FRANK.

## 7.2 Requirements

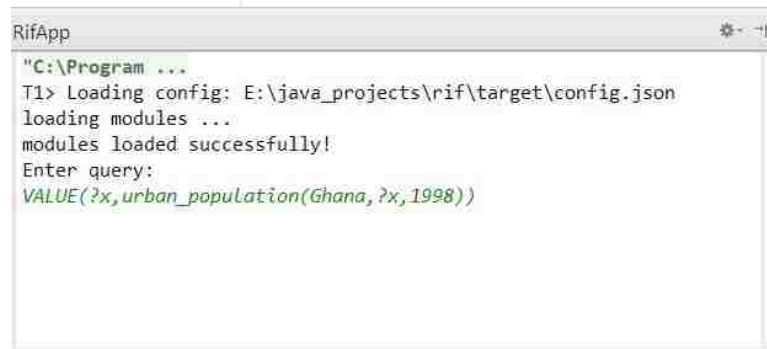
We developed this prototype of FRANK within the time and other resource constraints of an academic research project. That is, we are mostly concerned with the features that enable us to explore:

1. the internal representation of alists and the FRANK tree,
2. the inference algorithm, and
3. an architecture that makes it easy to extend FRANK.

In each of these tasks, we used existing standards or tools as foundations to implement our ideas instead of creating them from scratch. For instance, the FRANK alist representation was implemented as a dictionary (map) data structure, one that can be found in most programming languages. Another requirement was to use a representation and architecture that corresponds with current recommended standards. Although we are not building production-level software, certain design features, such as interfacing with third-party APIs (application programming interfaces), are widely used in our implementation. We implemented FRANK as RIF (rich inference framework).

We chose the Java programming language (Arnold et al. (2000)) for implementing FRANK. This was driven largely by the number of dependencies that we needed to complete a functioning prototype within the time available for this work. Given that we built on existing Semantic Web and linked data techniques and resources, we used the Java language since many of the tools for local triple stores such as Apache Jaena (The Apache Software Foundation (2017)) are built on the Java language.

There are, certainly, more optimal ways to implement FRANK such that it executes queries very fast, given the myriad of programming languages, methodologies and architectures available. However, in this work our emphasis is not on the speed of inference. We acknowledge that the speed of responses plays a crucial role in the usefulness of a QA system. However, we aim to show the relevance of our proposed

A screenshot of a terminal window titled "RifApp". The window shows the following text: "C:\Program ...", "T1> Loading config: E:\java\_projects\rif\target\config.json", "loading modules ...", "modules loaded successfully!", "Enter query:", and "VALUE(?x,urban\_population(Ghana,?x,1998))".

```
RifApp
"C:\Program ...
T1> Loading config: E:\java_projects\rif\target\config.json
loading modules ...
modules loaded successfully!
Enter query:
VALUE(?x,urban_population(Ghana,?x,1998))
```

Figure 7.1: FRANK user interface.

Command line user interface for FRANK with a query.

---

algorithm, the integration of uncertainty and how we satisfy our hypothesis. We only aim for a reasonable speed in query-answer time.

Our primary requirements for implementing FRANK are the following:

1. an architecture for FRANK that is easy to extend with new rules and aggregations functions for new domains.
2. an implementation of the query grammar for FRANK;
3. access to a variety of RDF triple stores and other knowledge bases;
4. a mechanism for caching retrieved data locally;
5. the ability to dynamically load FRANK modules for decomposition and aggregation during runtime;
6. a visualisation of the FRANK graph;
7. multi-threading functionality such that multiple branches of the FRANK tree can be explored concurrently;

We satisfy these requirements in the sections that follow and in our implementation of the relevant features.

## 7.3 Architecture

We chose an architecture that is both easy to implement and easy to extend. We create multiple Java packages that focus on specific aspects of the system. Our overall system architecture for FRANK is shown in figure 7.2. We use the package structure

to separate the various aspects of the FRANK system: decomposition rules, aggregate functions, knowledge-base wrappers, general application housekeeping. We have three key packages for this:

1. *FRANK\_core*: This package contains the core classes for FRANK. These include classes for managing FRANK alists, classes for computing uncertainty and classes caching data, as well as classes for calculating node costs. These classes are used extensively throughout FRANK.
2. *FRANK\_modules*: This package contains sub-packages for key modules in FRANK, some of which are dynamically loaded at runtime. These include packages for:
  - decomposition rules
  - aggregation functions
  - knowledge bases
  - query grammar
3. *FRANK\_app*: This package contains classes and libraries that are relevant to executing the application. It references classes from the *FRANK\_core* and *FRANK\_module* packages. This package provides a user interface for running the application (see figure 7.1). The user interacts with FRANK using the prompt displayed on the screen. That is, it runs in the main application process and spawns new threads for executing the core FRANK algorithm. This module also manages the loading and application of inference operations.

We implemented FRANK in a distributed architecture with respect to the KBs, the decomposition rules and the aggregate functions. That is, we did not have a hard requirement that all these modules be available locally. In fact, it is unrealistic to have all KBs exist locally. In reality, most KBs are located remotely from the FRANK framework, which requires us to provide a uniform way of accessing them. Consequently, we created Java interfaces that define the minimum functionality that we want from each wrapper class for a KB. The minimum functionality needed was:

1. to determine the existence of a property in the KB,
2. to search for property values,
3. to find dates corresponding to property values.

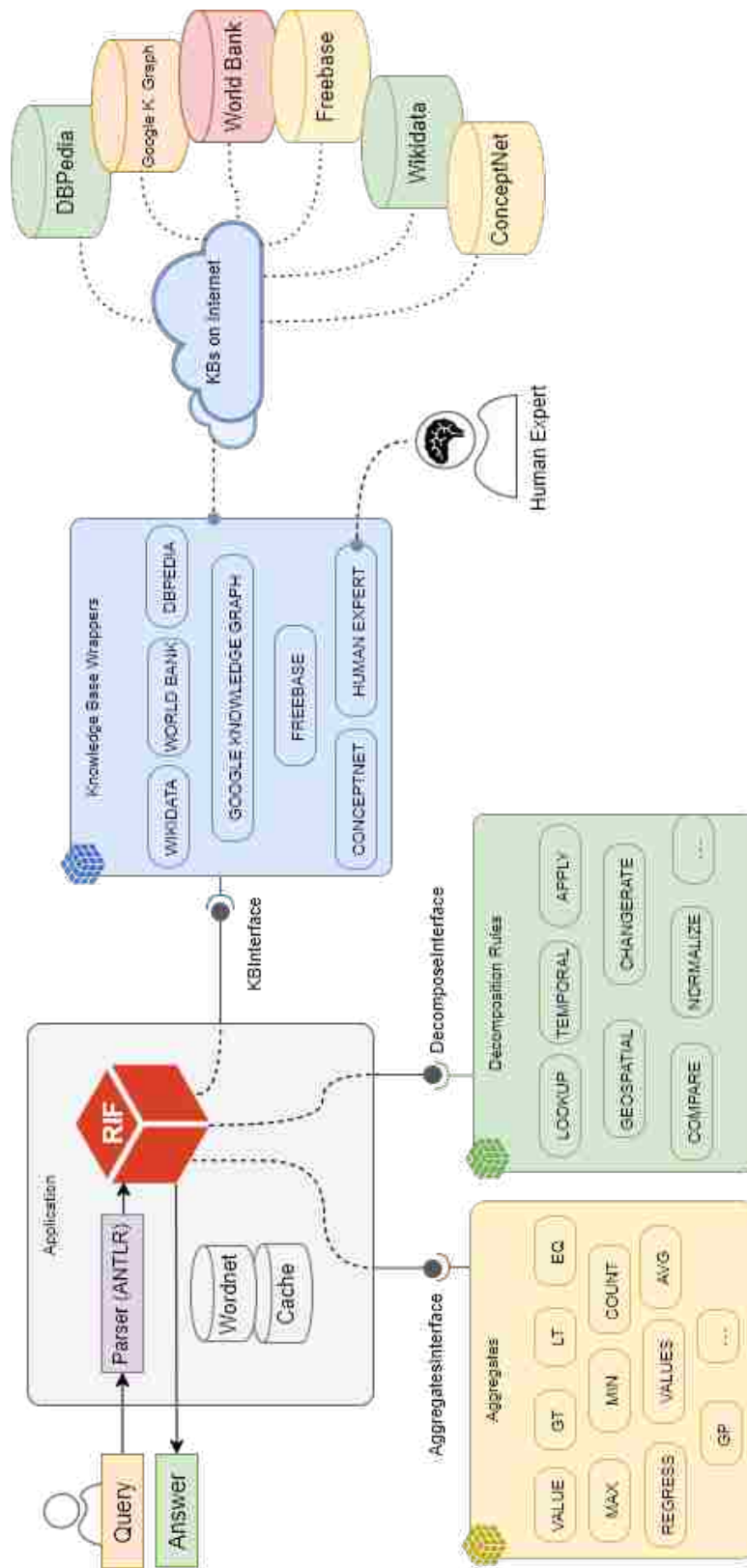


Figure 7.2: FRANK Packages.

Main packages of FRANK (implemented as RIF (rich inference framework) and edges pointing to their dependencies.

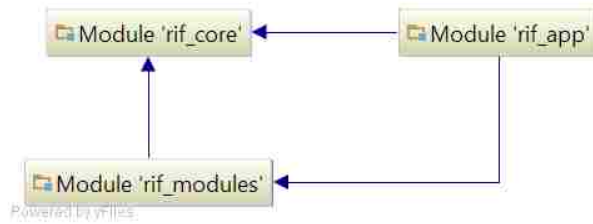


Figure 7.3: FRANK Packages.

Main packages of FRANK and edges pointing to their dependencies.

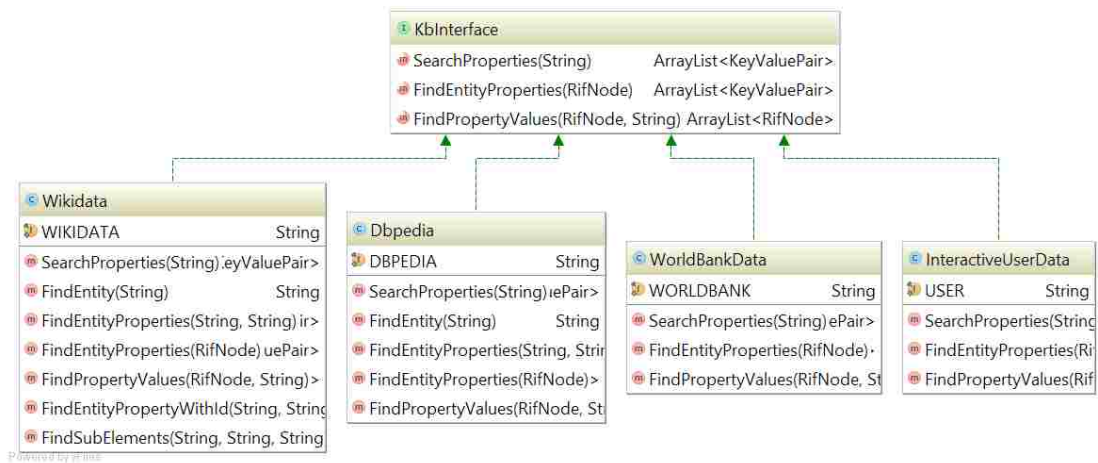


Figure 7.4: Class hierarchy diagram for KBs.

All knowledge bases implement the KbInterface interface.

We use this wrapper architecture for KBs that are accessed via a REST (Representational State Transfer) (Fielding and Taylor (2000, 2002)) API, a SPARQL endpoint and a standard database. We later show in section 7.6.1 how we implement the same interface to provide a human expert interaction to FRANK, which we call the *Human Expert Knowledge Source*.

Similarly, we define the contractual obligations of decomposition rules and aggregate functions by specifying the relevant Java interface methods that it allows. This makes it easy to reference any KB, rule or aggregate function uniformly, regardless of the underlying original form of the module. It also serves as the basis for our ability to dynamically load the relevant modules at runtime, a subject that is discussed in section 7.4.

## 7.4 Class Wrappers and Dynamic Loading

FRANK uses class loaders (Liang and Bracha (1998); Qian et al. (2000)) to dynamically load classes during the execution of FRANK. We use this as a means of making FRANK easy to extend without having to make changes to the FRANK algorithm. Class loaders provide a mechanism for dynamically loading software components in Java. The Java system provides three key features that are directly relevant to our development of FRANK. These include lazy loading, type-safe linkage and user-definable class loading policy (Liang and Bracha (1998)).

1. Lazy loading: This is the process of loading classes on demand. Given that there could be several decomposition rules, aggregation functions and KB wrappers available to choose from, the simplest approach is to simply load all the possible modules when FRANK starts up. This is, however, not efficient and in some cases not practical. One major constraint is the amount of memory needed to load all these modules. This becomes a critical issue when the library of modules for rules, aggregates and KBs grows. There is already a high demand on memory for the FRANK tree during execution, and therefore we want to keep the amount of memory used for idly holding a module to a minimum. By dynamically loading the classes for these modules as they are needed, we make it easy for FRANK to have a large library of modules without having to sacrifice memory.
2. Type-safe linkage: The Java programming language's provision for class loading



does not require additional run-time checks in order to guarantee type safety. Liang and Bracha further state that dynamic class loading does not violate the type safety of the JVM and also that link-time checks are acceptable given that they are performed only once. This is consistent with our use of the loaded classes in FRANK. During runtime, we do not have to perform any specific tasks to ensure that the classes being loaded are type safe. It is therefore efficient to load the classes into memory once at start-up and then reference them during the execution of FRANK.

3. User-defineable class loading policy: Java gives us full control of the dynamic class loading process since class objects are just objects like any other in Java. This gives us a lot of flexibility in determining which classes are to be loaded dynamically, their location (local or remote), and the checks to ensure that they satisfy the basic requirements needed by FRANK: i.e. the implemented functions in FRANK.

The class loader (*FRANKClassLoader*) itself is an ordinary Java object that we define in our *FRANK\_app* package. The classes that are loaded dynamically (e.g. *Temporal* decomposition rule defined in the *com.FRANK.modules.decompose.Temporal* class) are stored in a binary, machine independent standard known as *bytecode* (Yellin and Lindholm (1996)). This can be loaded easily from the stored file, stored in a memory buffer or retrieved from a network stream. This is also a desirable feature of Java that we leverage in FRANK. This means that modules can easily be shared and used at runtime both locally or from remote sources.

## 7.5 Caching

FRANK is an I/O (input/output)-intensive algorithm due to its dependence on facts in web KBs. Web lookups constitute a significant fraction of the evaluation time of a FRANK query as shown in our evaluation of FRANK (section 8.5.1). Due to the time overheads in calling web services, we store a local copy of KBs, such as WordNet, ConceptNet and Geonames, which are used frequently by FRANK and are rarely changed by their authors. However, data resources such as Wikidata, Google Knowledge Graph (Singhal (2012)), and World Bank Data, that are frequently updated, are queried directly using their published APIs. Since these resources return data in JSON (JavaScript Object Notation) format or provide SPARQL endpoints, we built wrappers

in our system for each of these types of data sources. The KB wrappers implement the same Java interface, making it possible to add knowledge sources without modifying the core of FRANK.

However, querying these web knowledge bases significantly increases the time to infer an answer to a question. To deal with this, we cache as much information as we can to speed up subsequent requests. We perform two kinds of caching: fact caching and function caching.

1. **Fact Caching:** This is the common form of caching where FRANK saves all facts retrieved from successful KB lookups at the leaves of the FRANK tree. Additionally, inferred facts that are obtained through intermediate inference processes such as upward propagation, are cached. An example of a cached fact is shown in figure 7.5.
2. **Function Caching:** As its name suggest, this caches functions that are inferred by aggregates in FRANK, particularly, the polynomials from the REGRESS function. One of our key contributions of FRANK is the ability of the framework to generate functions from observed data such that it can answer user queries by extrapolating from the graph or interpolating even when the specific data points were not observed from any KB. In this work, we dealt only with continuous (polynomial) functions and therefore we only need to cache the coefficients of the polynomials. At runtime, FRANK can recreate the functions by simply fetching the coefficients from the cache, and then pass in the arguments needed to infer and new value. For instance, when the regression function is used, we cache the subject and property that generated it, as well as the coefficients of the regression function. If a subsequent query requests data for the same subject and predicate but for a different time, we can reuse this cached function to estimate the value if no explicit facts are found in the cache. Figure 7.6 shows an example of a cached linear function. This form of caching, to the best of our knowledge is novel in the QA domain.

## 7.6 Extending FRANK

We implement two key design decisions that make it easy to extend the framework.

1. Alist representation

```
{
  "_id" : ObjectId("598df0160584e631a019cee3"),
  "subject" : "bulgaria",
  "property" : "birth_rate",
  "object" : "9.2",
  "time" : "2013",
  "cov" : 0.0,
  "source" : "WorldBankData",
  "lastmodified" : "2017-08-11 18:57:42"
}
```

Figure 7.5: An example of a cached value in the FRANK cache.

```
{
  "_id" : ObjectId("598df0110584e631a019ceb2"),
  "subject" : "botswana",
  "property" : "birth_rate",
  "object" : "LIN;111.8982222222222607;-0.04303333333333336234",
  "time" : "2005",
  "cov" : 0.00593965862033461,
  "source" : "inferred_function",
  "lastmodified" : "2017-08-11 18:57:37"
}
```

Figure 7.6: An example of a cached function in the FRANK cache.

## 2. Dynamic class loading

The FRANK alist, implemented as map (dictionary) object being a dictionary, means that we can easily add additional attributes that are relevant to the inference operation that we want to implement in FRANK. The attributes of the alist are propagated from one alist to another in the FRANK tree as the decomposition rules are applied to them. Similarly, the aggregate functions are applied to specific attributes (labelled with the *agvar* (aggregate variable) attribute) in the alist to combine them. For example, if we want to include the conversions of quantity units from one type to another, we can easily include a *unit* attribute in FRANK and then provide decomposition rules that specify how those attributes are set and aggregates that determine how they are up-propagated. This extension of FRANK alists to incorporate units of quantities was part of a final year undergraduate project (Markov (2017)) to extend FRANK in the renewable energy domain. When an attribute is not required by a rule or an aggregate, it simply passes through the operation untouched.

Secondly, the ability to dynamically load modules in FRANK makes it possible to keep the core FRANK algorithm separate from modules that can be extended. We have a configuration file for modules that allows a user to declaratively specify which FRANK modules should be included or excluded when FRANK is executed. This

means that we can readily add new data sources, new decomposition rules and aggregate functions and include them in FRANK module configuration file to be loaded when FRANK starts up. Since extensions to FRANK KBs, rules and aggregates accept alists as specified by the respective Java interfaces, they work seamlessly in FRANK when loaded.

### 7.6.1 Example of FRANK KB Extension

To show our ability of extend FRANK without having to modify the core inference framework, we added a new data source – “the human expert knowledge source (HEKS)”.

Our goal was to add a new feature that allows FRANK to function in a more interactive manner. We intended to treat the human expert as a knowledge source that can be queried for facts. The queries are posted at the command prompt on the screen and the human expert types in the answer. The user is also asked to give the error margin that they will associate with their answer that they have provided.

To add HEKS to FRANK, we created a new Java class named “*InteractiveUserData*” that implements the *KBInterface* class. There are two methods in the interface that the new class must implement:

*SearchProperties(String searchTerm)*  
*FindPropertyValues(FRANKNode FRANKNode, String alistAttribute)*

The *SearchProperties* function would ordinarily search the KB for the existence of a KB. This function is called at the beginning of FRANK to determine if the KB has data about the predicates in the user’s query and returns all the matching property names in the KB. For the *InteractiveUserData* class, we will simply ask the user via the command prompt:

“Do you have knowledge about  $\langle$ searchTerm  $\rangle$ ?”

If the user responds with a *yes*, we will add the human expert to the list of KBs that we can query for data about the property as the FRANK tree is explored.

The *FindPropertyValues* method is implemented by asking the user questions based on the *alistAttribute* parameter that specifies the attribute of the alist for which FRANK is looking up data. If the *alistAttribute* is the *object* attribute, then we ask the user a question composed from the template:

The  $\langle$ property  $\rangle$  of  $\langle$ subject  $\rangle$  [in  $\langle$ time  $\rangle$ ] is ?

where  $\langle x \rangle$  is a place holder in the template to be filled in by FRANK.

If the *alistAttribute* is *time*, we ask the question:

*When  $\langle property \rangle$  of  $\langle subject \rangle$  is/was  $\langle object \rangle$  ?*

If the user is able to provide an answer, FRANK asks one more question to determine the uncertainty. If the answer provided is real-valued, FRANK asks:

*How sure are you of this answer? (Error margin +/-) ?*

For a non-real valued answer, FRANK asks:

*How many other answers do you have in mind?*

We convert this value to a standard deviation and use it as the *cov* of the alist that is propagated from the FRANK node being processed. The full code of this extension to FRANK is listed in appendix C.

## 7.6.2 Extending FRANK with inference Over Web Services

In addition to extending FRANK by adding new knowledge sources, we can also extend FRANK by adding new decomposition rules and aggregates over web services. That is, instead of creating new rules and aggregates (we refer to both as inference operations) by implementing their respective Java interfaces, we can use a more distributed architecture where the operations are external to FRANK and are accessed as web services.

The general architecture of FRANK is modular, having separate components that include the decomposition rules, aggregates and KB wrappers. Given this architecture, we can distribute the inference tasks such that not every inference operation has to be implemented in FRANK. Rather, FRANK can use existing inference operations created by third-parties with a common interface over a web service, preferably a REST-based service. Such an extensibility feature is desirable for a number of reasons, a few of which are discussed below.

1. It is not every decomposition rule or aggregate function that can be implemented directly in the core FRANK algorithm. For a domain where the required data necessary to decompose an alist is not open or publicly available, the ‘owner’ or implementer of the rule provide a REST service that accepts a FRANK alist and returns a set of new alists that constitute it’s decompositions. This is particularly useful in cases where access to relevant data is restricted.

2. Some decomposition rules and aggregates will have very high computational requirements and so are better implemented on a dedicated high-performance compute server, and exposed as web services that are called on-demand.
3. FRANK can leverage other forms of reasoning and inference by considering the idea of ‘rich inference as a service’. This could include, for example, taking advantage of machine learning services such as machine recognition, translation, etc., offered on cloud platforms Amazon AWS<sup>1</sup>, Google Cloud<sup>2</sup>, Microsoft Azure<sup>3</sup>, etc. Several of these libraries are not available as libraries that can simply be integrated into the core FRANK algorithm. Even if it could, the cost of executing it locally outside these cloud platforms is prohibitive (in some cases, impossible) and would far exceed the communication overheads of calling a web service.
4. By keeping the core FRANK algorithm small and light weight, it can be executed locally while calling external web service when certain decomposition rules have to be called to decompose an alist or to aggregate a set of alists.
5. Such a distributed architecture opens up the possibility for others to contribute inference operations that FRANK can call remotely.

Our primary justification and motivation for this concept is similar to the idea of not hosting all open data that FRANK uses in the local machine. Even if it could, the simple fact that the publishers of the data could update the datasets gives FRANK an even harder task of trying to keep its local copy of the data in sync with the publisher’s version. Similarly, it is unwise to assume that a single QA framework can possess all the inference strategies necessary to answer all kinds of queries. However, by distributing it and having a single wrapper class in FRANK for making these remote calls, FRANK can easily have a wide range of inference operations at its disposal.

Additionally, since FRANK moves data around in alists, which can easily be cast as JSON objects, passing alists around over web services, is perfectly within the current web systems architecture.

More importantly, we can achieve a chaining effect with FRANK, where difference instances of FRANK with access to different KBs can pass alists between one another. This is of particular importance in scenarios where the data cannot be made

---

<sup>1</sup><https://aws.amazon.com/>

<sup>2</sup><https://cloud.google.com/>

<sup>3</sup><https://azure.microsoft.com>

public for reasons such as privacy (e.g. in the medical domain). Consider, for example, a query that asks: “How more people in the UK contracted HIV AIDS between 2010 and 2015?” Answering this query may require access to sensitive medical information about patients that the National Health Service(NHS) would not want to release to third-parties in order to safeguard the privacy of patients. However, in a scenario such as this, the answer can be retrieved with a FRANK instance that runs within the medical institution’s domain, exposes a web service that accepts an alist that will serve as the root of the FRANK tree, and returns a FRANK node (or set of nodes) that will constitute the answer to be passed back to the source FRANK instance. This is, in a way, an *Inference-as-a-Service* architecture.

## 7.7 Other FRANK Components

We briefly explain additional components that are implemented as part of the main FRANK package in the following sections.

### 7.7.1 Query Grammar with Antlr

We define a simple grammar for representing FRANK queries. Given that our emphasis in this project is on the inference mechanism for answering queries, and not on the natural language problems related to question answering, we use queries that are formalized in a manner that FRANK can interpret and process. To implement this, we use Antlr (Parr (2013)), a parser generator for reading, processing, executing, or translating structured text or binary files<sup>4</sup>. We specify a grammar from a small vocabulary of terms for FRANK queries in Antlr (in BNF). Antlr generates Java classes for parsing input text using the grammar as input. We import these classes into our FRANK package to create the parser for FRANK queries. The FRANK query internalisation process (section 4.7.1) begins during the parsing of parse tree of the FRANK query in Antlr. The result of this process is an alist that becomes the root alist in the FRANK tree, or the root and the first few branches in the case of nested queries.

### 7.7.2 Parallelizing FRANK

The FRANK algorithm operates by expanding and exploring the FRANK tree. The tree structure lends itself to parallelization allowing FRANK to explore multiple branches

---

<sup>4</sup>[http://www.antlr.org/](http://wwwantlr.org/)

in the tree concurrently. The need for running FRANK in a multi-threaded mode is further highlighted by that fact that FRANK is an I/O intensive algorithm that performs KB lookups at each leaf node. This means that the amount of time spent in FRANK doing non-CPU (central processing unit) tasks increases as the FRANK tree grows. This means that there are times when the CPU is almost idle while waiting for data from a remote KB, significantly slowing down FRANK.

We improve the performance of FRANK by incorporating concurrency in the Java language in our implementation of FRANK. Java offers concurrent programming (Oracle (2015)) features based on *processes* and *threads*. A process<sup>5</sup> is defined as a self-contained executing environment with a complete set of basic run-time resources (e.g. memory space). A thread is defined as a lightweight process, also providing an executing environment, but requiring fewer resources than creating a process. In FRANK, the main application runs in a process spawned by the operating system. This creates the main thread that executes the FRANK algorithm. However, given that each node in the FRANK tree will at some point spend time looking up data from a KB, and will, for that period of time, block the rest of the FRANK algorithm from running, we create a thread pool<sup>6</sup> of worker threads that are capable of calling the FRANK algorithm for each node in the FRANK tree. The number of worker threads in the thread pool can be set in the FRANK configuration file. The queue of tasks that worker threads can be assigned is made up of the nodes at the frontier of the FRANK tree at any given point in time. Each worker thread attempts a lookup, and up-propagates alists if facts variables are instantiated. Otherwise the worker thread decomposed the alist to create new child alists.

We discuss the impact of FRANK parallelization in section 8.5.1.

### 7.7.3 Visualising the FRANK Tree

In addition to FRANK generating a trace of the inference process in text, we also generate a visual tree dynamically as FRANK executes. We implemented this using the GraphStream library (Dutot et al. (2007)). For instance, the visual tree created when FRANK answers the query “*Which country, Ghana or Nigeria, will have the largest population in 2019*” is shown in figure 7.7.

---

<sup>5</sup><http://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html>

<sup>6</sup><https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html>



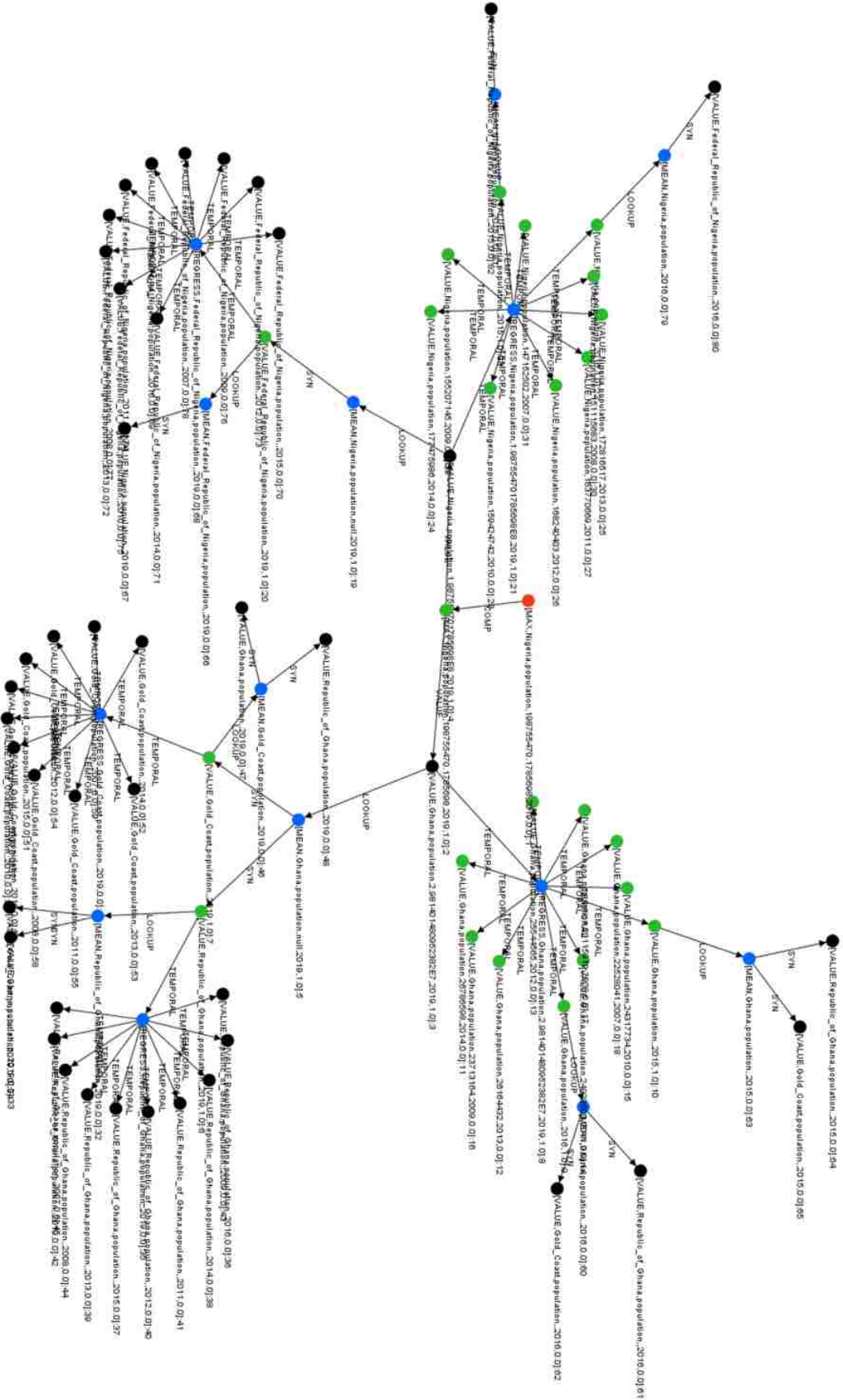


Figure 7.7: FRANK tree visualization.

Generated for the query “Which country, Ghana or Nigeria, will have the largest population in 2019”. The red node is the root of the FRANK tree.

### 7.7.4 External Libraries for Aggregates

Third-party software libraries used for implementing aggregates include:

- The Apache Commons Mathematics Library<sup>7</sup> for general maths functions;
- Symja Library (Java Symbolic Math System)<sup>8</sup> for parsing symbolic expressions for integrals and derivatives;
- GPy<sup>9</sup>(GPy (2012)), a Gaussian Processing framework in Python.

## 7.8 Summary

In this chapter, we discussed our implementation of FRANK. Our design decisions during the development of a prototype of FRANK were driven by the need to implement FRANK in such a way that the core FRANK algorithm is realized in a way that works practically. This meant that we had to implement features such as caching that make it possible to incrementally add to FRANK's local cache of facts from data looked up in remote KBs, from data inferred during up-propagation, as well as from functions inferred by FRANK.

FRANK's architecture is motivated by our desire to make the FRANK algorithm easy to extend with new KBs, new decomposition rules and new aggregate functions. We use of idea of packages and dynamic class loading, a feature of the Java programming language, to separate modules for KBs, rules and aggregates from the core FRANK algorithm. This makes it easy to add to these modules without changing the FRANK algorithm. We achieve this by creating Java interfaces for these modules, thereby defining the methods (and their signatures) required by all classes that implement them. We show how this works by creating a new KB for a human expert and also in a project that extended FRANK to for query answering in the energy domain (Markov (2017)).

We also discussed the need for parallelization in FRANK, given the fact that FRANK spends a lot of time looking up data, a process that is less CPU-intensive compared with other computational tasks in FRANK.

In the next chapter, we test these design choices together with the evaluation of FRANK as a whole.

---

<sup>7</sup><http://commons.apache.org/math>

<sup>8</sup>[https://bitbucket.org/axelclk/symja\\_android\\_library/overview](https://bitbucket.org/axelclk/symja_android_library/overview)

<sup>9</sup><https://github.com/SheffieldML/GPy>



# Chapter 8

## Evaluation

In this chapter, we evaluate our hypothesis (section 1.4, repeated here for convenience), that

*the types of questions that can be asked and variety of answers found by a query answering system is significantly improved when we automatically curate data using a uniform representation, use rich and extensible forms of inference to infer novel knowledge from structured data available on the Internet, and acknowledge and account for uncertainty in answers using a framework that is easy to extend to other domains and question types.*

We tested the ability of FRANK to infer answers when the explicit answers are missing from the knowledge base. We also tested the range and variety of answers by evaluating multiple kinds of queries. Since uncertainty plays a key role in the way FRANK works, we checked to ensure that the uncertainty values that FRANK returned for its answers correlated with the true deviation of the inferred answers from the true one.

Our evaluation is organized by the four aspects of the hypothesis in the sections that follow.

### 8.1 Evaluation Experiments

In each of the experiments, we explain the goal of the experiment and the specific setup used. Generally in this evaluation, we use real-valued data from the World Bank and Wikidata. In a few of the queries that require non-real-valued facts, we use the DBpedia KB. All data sources, as explained in chapter 7 are accessed by FRANK through wrappers implemented from *KBInterface*. Knowledge bases that FRANK uses for processing queries and finding answers are:

- DBPedia (<http://dbpedia.org/sparql>)
- Wikidata (<https://query.wikidata.org/sparql>)
- GeoNames (<http://www.geonames.org/>)
- World Bank Data (<http://api.worldbank.org>)
- ConceptNet (<http://conceptnet5.media.mit.edu/>)
- WordNet (<https://wordnet.princeton.edu/>)

We ran experiments on a computer with the following specifications:

- Processor: Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz, 4 Logical Processor;
- Memory: 8GB;
- Storage: 100GB Solid State Drive.

## 8.2 Experiment 1 - Inferring missing facts (HYP-1)

Our primary hypothesis for this project is the ability to infer answers that are not pre-stored in the knowledge bases available to FRANK. In this experiment, we tested the ability of FRANK to infer answers that were not pre-stored in any KBs by using other facts that are retrieved from the KB. This includes both interpolation and extrapolation from available data. Additionally, we wanted to test FRANK's uncertainty values to verify that they correlated with the deviation of FRANK's calculated answer from the correct answer.

We evaluated FRANK with a variety of queries and compared results to Google Search (GS) and Wolfram|Alpha (WA) outputs. Concretely, we focused on queries that are of interest to public institutions that publish data on the web under the open data initiatives. We based the test queries on real-valued facts available in Wikidata and the World Bank dataset (WBD). We compared results from FRANK to GS and WA because the latter systems already have access to the Wikidata and the WBD. GS has a copy of the WBD in its Public Data Directory ([www.google.co.uk/publicdata](http://www.google.co.uk/publicdata)). WA also has a computational engine that implements most functions used in FRANK methods. WA also uses carefully curated data in its proprietary knowledge base.

Existing test sets for evaluating query answering systems focus on aspects of the inference process that differ from our objective with FRANK. We were interested in

queries that, not only find relevant facts, but also infer non-trivial answers by combining them. We therefore looked at the kinds of questions that are usually asked about demographics and other country development indicators from open data sources such as the World Bank and created a list of test queries to evaluate FRANK. Of particular interest were numerical facts. Our evaluation consisted of forty queries covering four main query types: (1) retrieval of explicit facts, (2) aggregation of facts, (3) nested queries (“sub-queries”), and (4) prediction when answers are not pre-stored in KB. Examples of queries for each query type are shown in appendix D. In test (4), for queries whose answers were known to exist explicitly in our KB, we used the holdout method, where we removed the specific fact which answered the question from the KB, and then checked if the system was capable of inferring the correct answer given the remaining facts in the KB. To test our queries on WA, we used the query format guidelines and examples provided on its website ([www.wolframalpha.com/examples/](http://www.wolframalpha.com/examples/)). Since GS has no specified query format, we tried different natural language formats for each test query and ensured that all relevant keywords were present in the queries. For GS, we considered the top 10 search results returned, while in WA, we looked at the answer it computed. Results for the four query types are shown in Table 8.1.

We configured FRANK to run in evaluation mode (that is to answer multiple queries in batch mode and save outputs to file) and specified the facts to hold out in FRANK’s configuration. We also cleared the cache prior to running each query in the test set to ensure that previously retrieved facts that directly answer the query via a lookup operation were removed.

We generated a set of 100 queries (listed in appendix D) randomly using property terms related to the country indicators in the World Bank dataset. We used 60 of these queries as a training set during the development of FRANK and put the remaining 40 away for these experiments.

### **Results for Experiment 1**

Our results are shown in table 8.1 and figure 8.1.

Overall, FRANK performed better than Google Search (GS) and Wolfram|Alpha(WA) in the four query types tested. FRANK’s use of geospatial, temporal and commonsense facts as well as higher-order functions allowed it to tackle a wider range of queries successfully. FRANK’s main limitation was its word matching mechanism, where it failed to find the appropriate matches from KBs in some cases. This was due to its lack of NLP to handle the useful NL descriptions contained in facts. Hence, FRANK some-

Queries	Google Search(%)	Wolfram Alpha(%)	FRANK(%)
Retrieval	70	80	90
Aggregation Queries	20	70	80
Nested Queries	-	50	80
Prediction	10	20	70
Average %	25	55	80

Table 8.1: Evaluation results by query types, showing the percentage of queries answered successfully.

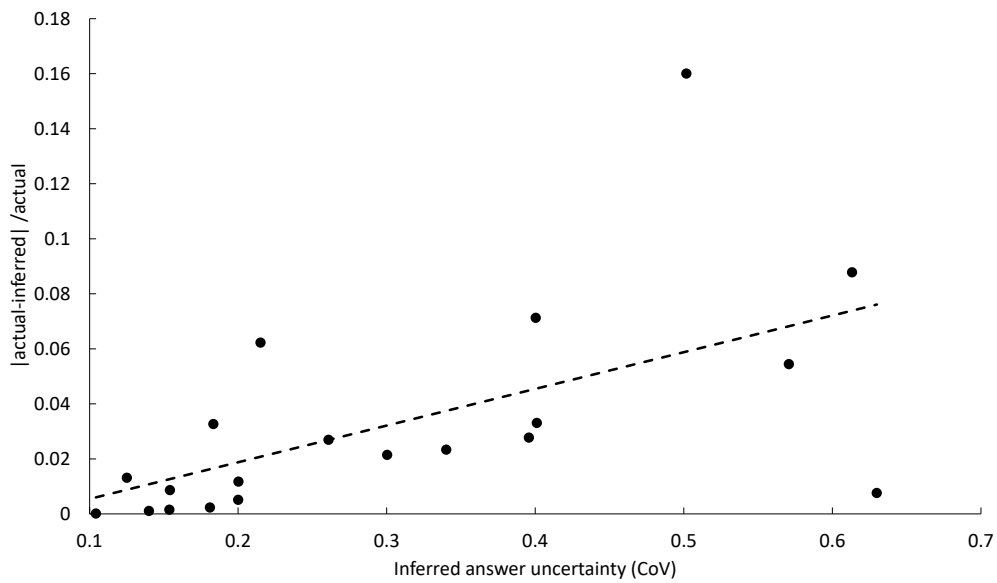


Figure 8.1: *cov* and estimation error plot.

A plot of the *cov* of the inferred answer against the normalized ratio of the deviation to the actual answer shows the positive correlation between the two.

times mixed up data of different units when the same fact was provided in multiple units.

Although GS had access to the KBs containing the required facts, it resorted to searching web documents for answers in most cases. This worked for some queries in type 1 queries, but it could not tackle the majority of the other types of queries. In a few of the commonly used demographic facts such as *population*, *gdp* and *birth rate*, GS used the World Bank Data (WBD) facts to plot charts to show trends. However, GS did not infer answers to prediction queries even when the required facts could not be found in web documents. We could not test nested queries in GS since all nested queries composed failed to return reasonable answers even in the top 10 search results. This could have stemmed from our inability to compose nested queries in a form that made its semantics accessible to GS. WA, although able to answer more queries than GS, only worked for commonly used country and demographics properties. WA also failed to find answers for some queries that had time references, although the queries worked without the time references to find the current value of those properties. For queries in type 4 that required retrieval, prediction, aggregation, and sometimes, comparisons, GS and WA could only answer a very small fraction of the queries.

Finally, our evaluation of the confidence scores estimated by FRANK also showed correlation between the *cov* and the error between the true fact and what was inferred.

### 8.3 Experiment 2 - Prediction (HYP-2)

In this experiment, we tested the predictive capabilities of FRANK. Although we can pose queries that predict values for a future date, it is difficult to evaluate when there is no ground truth to compare to. Particularly, we simulated answering prediction queries by holding out all facts after a given time (year). This approach simulated a QA environment that assumed that the current year was 2005 and then asked questions about 2014. We held out all facts after 2005.

As with experiment 1, we used the World Bank dataset for over 15000 country indicators, as well as KB such as Wikidata, GeoNames and DBpedia. Here also, the task was not just to get the best possible prediction from FRANK, but to calculate a *cov* that corresponds to FRANK's prediction error margin when compared with the ground truth. We used the same set of test queries as in experiment 1, but changed the dates of the required answers to 2014.

We also checked how increasing the number of missing data values for different



Error Margin	10%	20%	30%
% of queries answered correctly	48	62	70

Table 8.2: Queries answered correctly within specified error margins.

Year(s) Held Out	Inferred Answer	<i>cov</i>	Normalized Absolute Error
2014	2.6962104e7	7.6101e-5	0.16996e-4
2013, 2014	2.6956515e7	0.2216e-5	2.24348e-4
2012, 2013, 2014	2.6957935e7	0.1853	1.71669e-4
2011, 2012, 2013, 2014	2.7057470e7	0.4712	3.50761e-3

Table 8.3: Inferred answers for the true Ghana population value of  $2.6962563e7$  in 2014 for multiple years' data held out.

years affected the uncertainty value. For the query

$$VALUE(?x, total\_population(Ghana, ?x, 2014)) ,$$

we assumed the current year of 2014 and held out a number of previous years' data for each run of FRANK. We started with just 2014 held out, then 2014 and 2013, etc., until we had 2014, 2013, 2012, and 2011 all held out. Results of our comparisons of *cov* and the normalized absolute error is shown in table 8.3 and the scatter plot in figure 8.2.

## Results for Experiment 2

The results of this experiment are shown in tables 8.2 and 8.3.

The percentages of queries answered correctly within the 30% error margin around the held out facts remained the same for all query types. However, the *covs* of these queries were generally higher than that of experiment 1 as expected. This shows that, for regression, FRANK provides an adequate measure of uncertainty that correlates with the error margin in cases where it has to predict. In figure 8.2 we observe a positive correlation between the number of missing years' data, the *cov* and the normalized error between the held out value and the inferred answer.

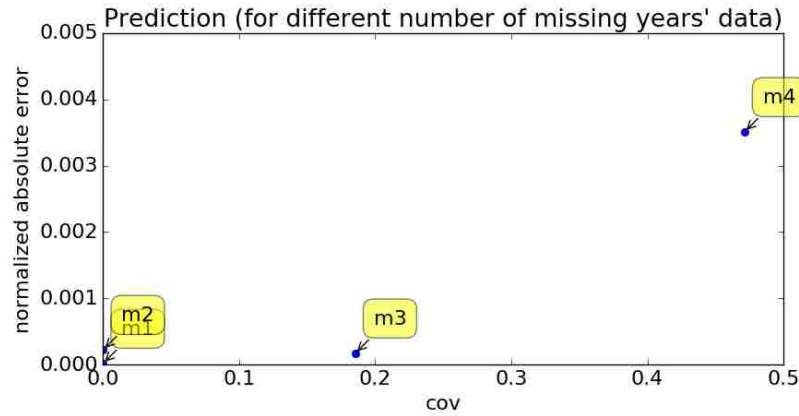


Figure 8.2: Prediction *cov* and estimation error plot

A plot of the *cov* of the inferred answer against the normalized ratio of the deviation to the actual answer for prediction of the query  $VALUE(?x, total\_population(Ghana, ?x, 2014))$ .

Different years are held out:  $m1 = 2014$ ,  $m2 = 2014, 2013$ ,  $m3 = 2014, 2013, 2012$  and  $m4 = 2014, 2013, 2012, 2011$

## 8.4 Experiment 3 - Inferred Functions and cov (HYP-3)

In this experiment, we evaluated the correlation between the different regression models that FRANK can use and the *covs* returned for each run of FRANK with a different model specified. Our objective in this experiment was to check if errors in model specifications are reflected in the *covs* that FRANK returns.

For this test, we used the query ‘ $VALUE(?x, total\_population(Ghana, ?x, 2004))$ ’ and held out the observations for 2004, thereby forcing FRANK to predict. We expected that the poorer the model (degree of the polynomial) is to the actual data observed, the higher the deviation and *cov*. We executed the query four times with FRANK set to polynomial degrees 1 to 4. We also set the temporal decomposition rule’s branching factor to 30, giving FRANK 30 observations from which to generate its functions.

### Results for Experiment 3

The different curves generated by FRANK for each degree setting in FRANK is plotted in figure 8.3. The linear function had the worst fit (the normalized absolute error) compared to the other three. This correlated with the high *cov* (shown in figure 8.4) FRANK returns for this answer. That is, FRANK’s uncertainty calculation for *cov* appropriately captures the error margins based on the fitness of the functions generated

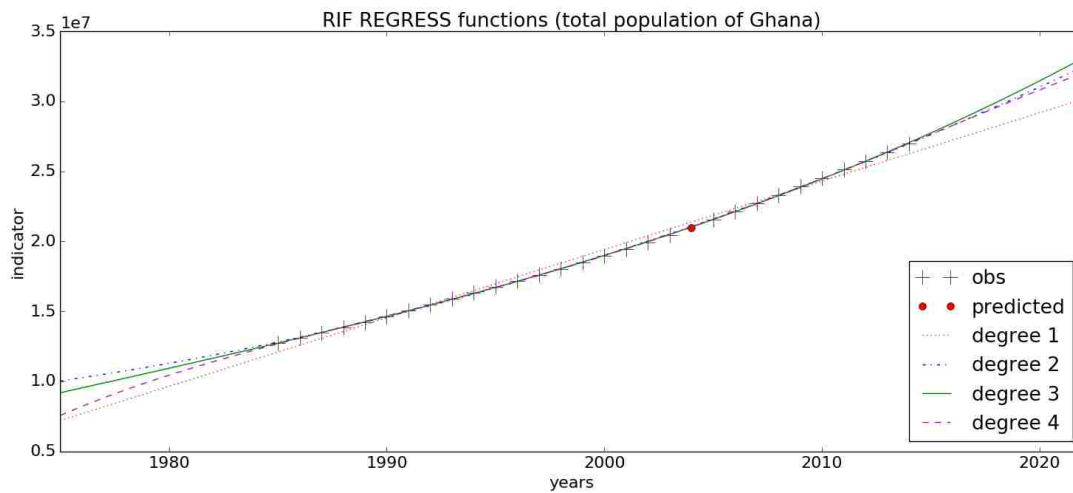


Figure 8.3: Regression curves for different polynomial degrees.

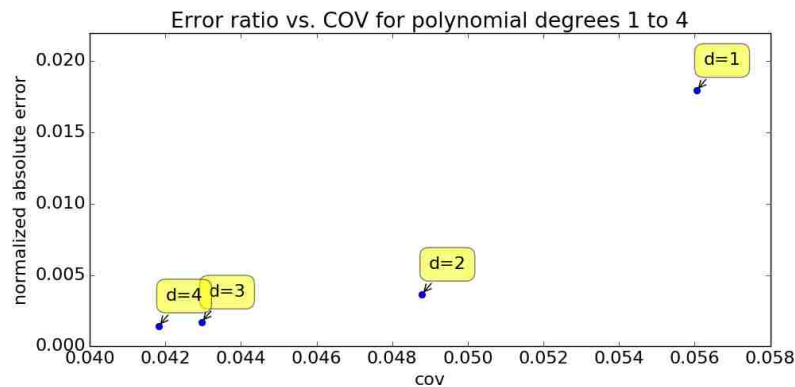


Figure 8.4: Inferred function model errors and covs.

from the data and the predictions from the functions.

## 8.5 Experiment 4: Architecture and Extensibility (HYP-4)

### 8.5.1 Experiment 4a - Impact of caching and parallelization

In this experiment, we evaluated the benefit of caching inferred functions in FRANK. In addition to storing inferred values in its cache, FRANK also saves inferred functions in its cache. We checked how efficiently FRANK uses the cached functions by enabling function caching and measuring the execution times of queries.

We used the test queries from experiment 2 that involve predicting values for fu-

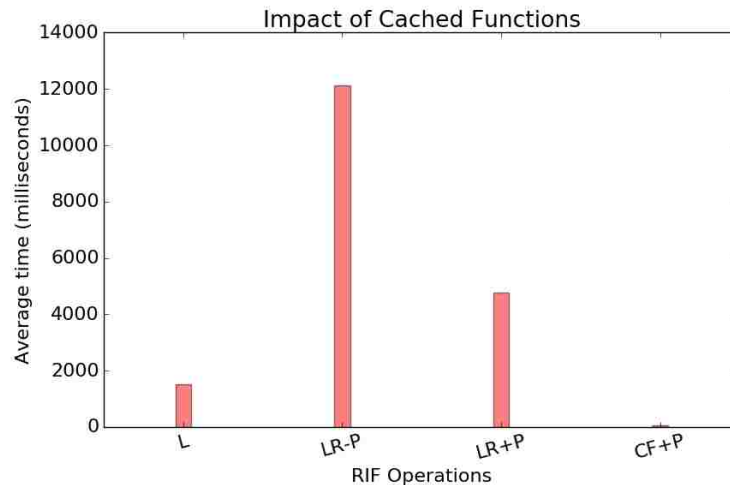


Figure 8.5: Impact of function caching on FRANK performance.

The vertical axis shows the average time in milliseconds to execute the test queries. The horizontal axis specifies the different setups for this experiment: Lookup only (L), lookup with and the use of the REGRESS aggregate without parallel execution of FRANK (LR-P), lookup with REGRESS in a parallel execution of FRANK (LR+P) and an execution of FRANK with function caching enabled (CF+P).

ture years. We ran the queries four times under different settings and recorded the time taken by FRANK to return an answer. In the first setting, using an empty cache, we simply looked up the fact from the remote KB. This served as a baseline for answering the query by looking up the fact directly. Next, we held out the fact that answered the query and forced FRANK to predict the answer using REGRESS. We ran it twice, first without parallelization and then ran it again with parallelization enabled. In both instances, we started with an empty cache. In the final setting, we enabled the use of cached functions inferred from the previous *Lookup + REGRESS* test. We limited the branching factor of the temporal decomposition rule to 10 in order to reduce the number of web service calls to remote KBs. Also, our goal was not to test the accuracy of the prediction, but FRANK’s ability to use cached functions when they exist.

#### Results for Experiment 4a

Results are shown in figure 8.5. FRANK’s lookup of facts to initialize variables at the leaves of the FRANK tree means that it spends a larger amount of time just looking up facts from KBs than aggregating the facts. For the queries in this experiment, FRANK takes just over 12 seconds on average to find the facts to generate the regression function (using the REGRESS aggregate) from which it infers an answer to a query. When

```

T25> |--Subnode: <32 $0 @4> [P=total_population,S=Republic_of_Ghana,T=2013,U=0.00,MV=
T25> |--Subnode: <34 $0 @4> [P=total_population,S=Republic_of_Ghana,T=2012,U=0.00,MV=
T25> |--Subnode: <36 $0 @4> [P=total_population,S=Republic_of_Ghana,T=2011,U=0.00,MV=
T25> |--Subnode: <38 $0 @4> [P=total_population,S=Republic_of_Ghana,T=2010,U=0.00,MV=
T25> |--Subnode: <40 $0 @4> [P=total_population,S=Republic_of_Ghana,T=2009,U=0.00,MV=
T25> |--Subnode: <42 $0 @4> [P=total_population,S=Republic_of_Ghana,T=2008,U=0.00,MV=
T25> |--Subnode: <44 $0 @4> [P=total_population,S=Republic_of_Ghana,T=2007,U=0.00,MV=
T25> |--Subnode: <46 $0 @4> [P=total_population,S=Republic_of_Ghana,T=2006,U=0.00,MV=
T25> |--Subnode: <48 $0 @4> [P=total_population,S=Republic_of_Ghana,T=2005,U=0.00,MV=
T25> Looking up: <4 $1 @2> [P=total_population,S=Gold_Coast,T=2014,U=0.00,MV=?x,?x=L=C
The total_population of Gold_Coast in 2014 ? : 27000000
How sure are you of this answer? (Error margin +/-) : 100000

```

Figure 8.6: An example of HEKS in use.

we parallelize this process, FRANK takes less than half this time (about 5 seconds) to find an answer. There is, however, a significant drop in time, down to an average of 60 milliseconds, when an existing function exists in cache from which an answer can be inferred.

### 8.5.2 Experiment 4b - Extensibility of FRANK

In this experiment, we tested the flexibility of FRANK with regard to the ease of extending the framework to new datasets and aggregate functions. Our main criteria in the experiment was to use the Java interfaces for the FRANK modules: *KBInterface* and *AggregatesInterface* (see section 7.6) to extend the KB and aggregates respectively.

In the first part of this evaluation, we created a new KB that uses human expert knowledge directly by asking the user for help during inference. This new KB module also asks the user for estimates of uncertainty, especially when no answer was available. The primary criterion for success is that the new module should work without having to change the code for the core FRANK algorithm.

In the second part of this experiment, we extended the FRANK aggregates by adding a new one. We implemented the Gaussian Process regression as a Python module deployed as web service. We used the *AggregatesInterface* (section 7.3) for this implementation, and as described above we ‘plug and play’ the new aggregates without any change to the core FRANK algorithm. The only change was to the modules configuration file where we added the new module name to the list of modules that FRANK should load when it starts up.

#### Results for Experiment 4b

We also expanded FRANK’s KBs by adding two new ones. First, an undergraduate final-year project extended FRANK to the energy domain (Markov (2017)). Test queries and the evaluation result showed that we can add new KBs to FRANK without

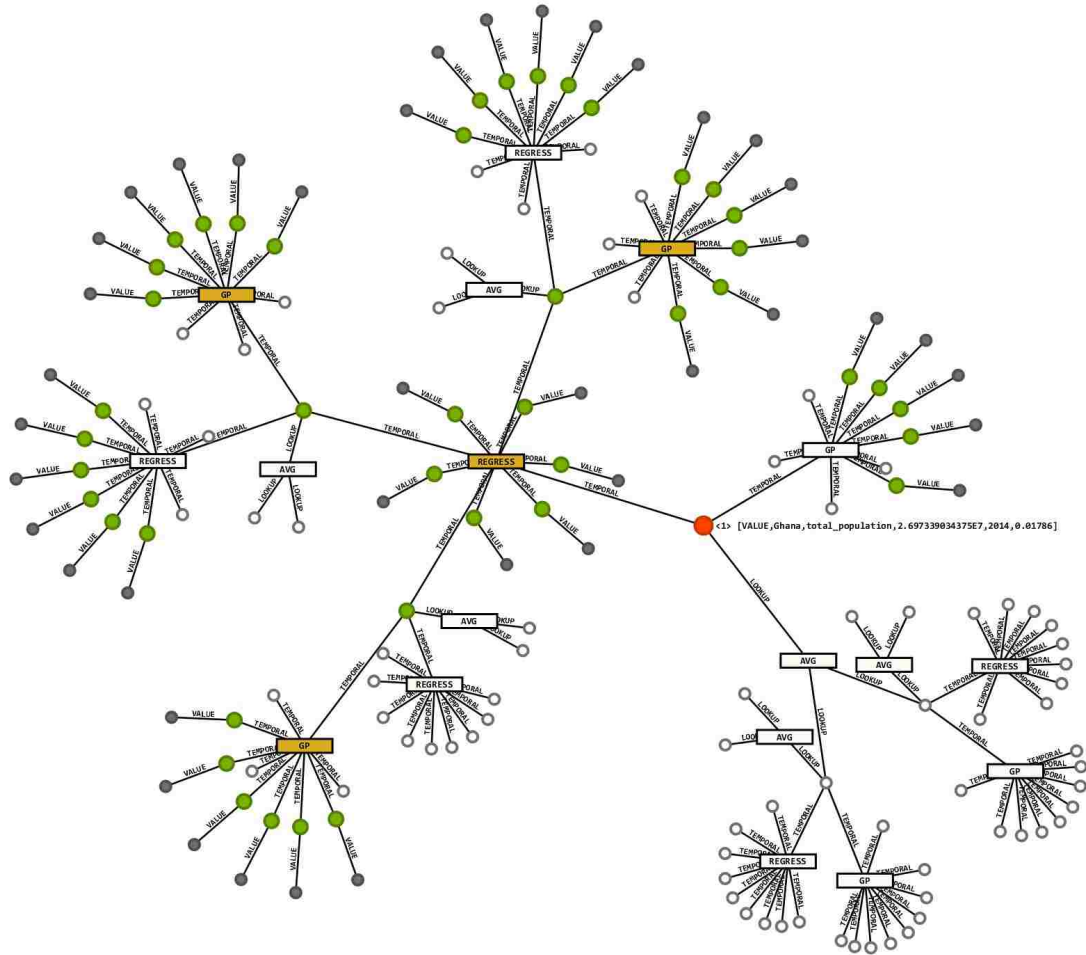


Figure 8.7: A FRANK tree that uses both GP and REGRESS.

Query:  $VALUE(?x,total\_population(Ghana,?x,2014))$  with data for 2011, 2012, 2013 and 2014 held out. The red node is the root of the FRANK tree.

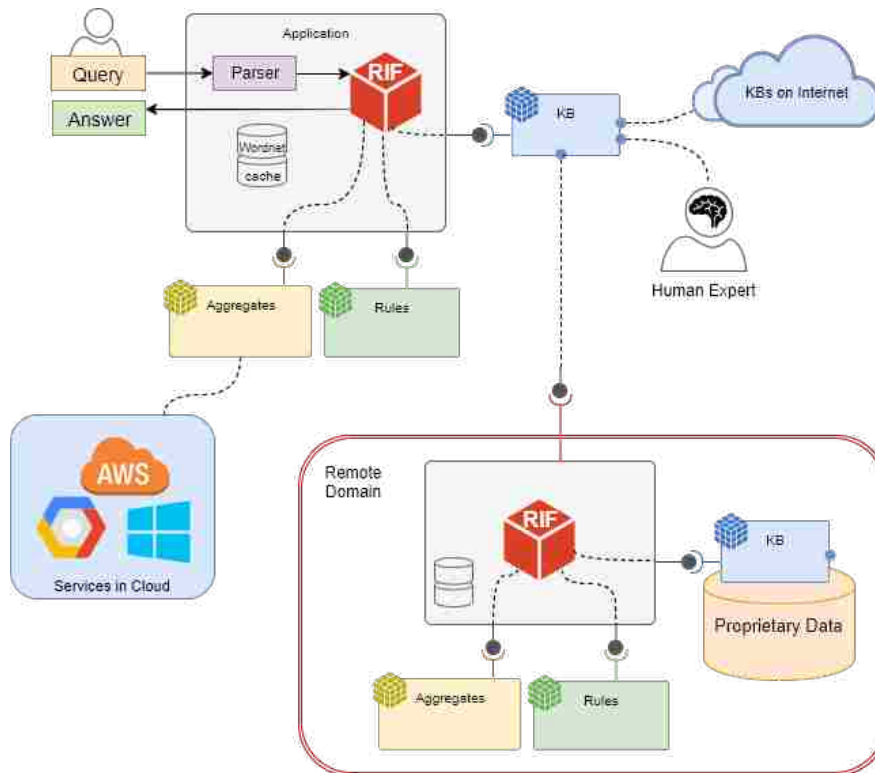


Figure 8.8: Extending and chaining multiple instance of FRANK

changing the core FRANK algorithm. The second KB extension added an ‘interactive’ feature to FRANK. We implemented the new KB wrapper as one that will ask a human, rather than a DB or KB for facts. We call this KB HEKS (human expert knowledge source). A screenshot of an interaction is shown in figure 8.6. HEKS treats the human as any ordinary KB. In order to estimate the posterior uncertainty of facts given via HEKS, FRANK asks for the human’s prior uncertainty about the fact. Our code for this FRANK KB extension is listed in appendix C.

Additionally, we implemented GP (Gaussian Process) regression REST API that takes a list of X-Y data pairs and a list of X values to predict as input parameters. The GP API returns a list of objects each consisting of the X value to be predicted, the predicted Y value and the standard deviation of the estimated Y. Our API uses the GPy<sup>1</sup> (GPy (2012)) Python library. To simulate the ability of FRANK to consume a remotely located inference operation, we implemented FRANK’s GP aggregate function using the *AggregatesInterface* by sending requests (data from the list of FRANK alists to be up-propagated) to the GP API service, and then passing the response back (both the prediction and uncertainty) up into the parent FRANK alist. A FRANK tree integrating

<sup>1</sup><https://github.com/SheffieldML/GPy>

both REGRESS and GP is shown in figure 8.7. Figure 8.8 also illustrates how FRANK can be extended by chaining multiple FRANK instances.

## 8.6 Experiment 5: Extending FRANK to the Energy Domain (HYP-4)

In a experiment conducted by Kalin Markov for his 4th-year honours project (Markov (2017)) supervised by Alan Bundy and Kwabena Nuamah, we considered the extendability of FRANK (an earlier version of FRANK at the time) to queries in the energy domain.

### 8.6.1 Objectives

The objective of this project was to answer guesstimation-styled queries in the energy domain such that it achieved answer accuracy within specified error bars. This required the addition of the appropriate inference operations that together could answer queries such as “*What proportion of the UK will need to be covered by solar panels in order to supply enough energy to meet the entirety of the UK’s energy demand in 1999?*”. In a related work, GORT (Guesstimation with Ontologies Reasoning Techniques) (Abourbih et al. (2010a,b); Bundy et al. (2013)) explored the use of the Semantic Web in solving guesstimation problems by finding approximations of answers to queries by using a combination of intuition, facts and reasoning with methods such as Count, Total Size, Law of Averages, Distance, Energy, Rate of Change, Aggregation over Parts, Generalization and Geometry methods.

### 8.6.2 Implementation

We summarize the steps required to extend FRANK to answer these new kinds of queries in the energy domain.

**1. New knowledge source:** In addition to data from the World Bank’s dataset on country development indicators, Wikidata and DBpedia, we included data from the Enerdata Statistical Year Book (Enerdata (2016)). The source format of this dataset was Microsoft Excel (.xlsx)<sup>2</sup> and so we had the choice to (1) change this to another format (e.g. RDF), or (2) to maintain the source format and instead extend FRANK

---

<sup>2</sup><https://products.office.com/en-gb/excel>



by adding to its KB wrappers. Given our interest in extending FRANK, we chose the later. With the help of the third-party library, Apache POI (Foundation (2016)), we created a new KB wrapper that could look up data from Microsoft Excel files. This wrapper was based on the `KBInterface` discussed in section 7.3 making it possible to curate data in the same manner as we did for the other KBs. We did not modify any of the existing KB wrappers for FRANK nor any other code in FRANK, except for those detailed below.

To lookup property names, we used the same string matching technique described in section 5.3.4. That is, we aimed to find direct string matches in the KB to the query property term, but failing that, found the closest matching string from the set of properties in the KB. Given that the string matching method used in FRANK is domain independent and focuses purely on the syntax of the strings being matched, the technique discussed in section 5.3.4 returned only appropriate matches from the energy domain KBs.

**2. New Attribute in Alist:** For knowledge sources such as Enerdata and the World Bank dataset, the units of the quantities were placed in parenthesis in the property names. We, therefore, added a unit attribute to the alist to store the units of the quantities of retrieved from the KBs. The KB's data on energy was expressed in diverse units. For example, *joules(j)* versus *kilojoules(kj)* or *KWh* versus *MWh* (*kilo- and megawatt hours respectively* ).

**3. New Inference Operation:** Our use of the *unit* attribute led to the addition of *UNIT*, the *Unit Normalization* operation, to FRANK to normalize the units in the child nodes to their base SI(International System of Units) form. All existing decomposition rules (e.g. temporal and geospatial decomposition rules) for FRANK remained the same. However, new inference operations for aggregation were added. This included *DIV*, *MUL*, *CHANGE* for division, multiplication, and change(difference) respectively. The process of adding these new aggregation functions was simple since, given the interface specification for FRANK's aggregation classes, the only task was to implement the specific arithmetic operation to combine the relevant attributes of the child alists as determined by the *operation variable* attribute of the parent alist. Similar to the existing operations, these new operations were not hard-coded into the implementation of the core FRANK algorithm. They were dynamically loaded into FRANK at runtime when needed.

Finally, the query grammar was updated to include the new aggregates that were added, allowing FRANK to accept and successfully parse queries such as:

```

DIV(?k,
  MUL(?a, VALUE(?y, average_size(solar_panel,?y,$z)),
    DIV(?y,
      VALUE(?x, total_energy_consumption(United_Kingdom,?x,1999)),
      VALUE(?x, energy_production_per_year(solar_panel,?x,$z))
    )
  ),
  VALUE(?m, land_area(United_Kingdom,?m,1999))
)

```

This required updating the grammar file of the third-party tool, Antlr<sup>3</sup>, and rebuilding it.

### 8.6.3 Evaluation

This extension of FRANK was evaluated on 5 query types: fact retrieval, aggregation, nested queries, prediction and “extended\_queries” (for guesstimation). Due to the lack of existing query-answer sets for evaluation, 30 queries were randomly generated based on templates for the query types. The answers to these queries were manually calculated and used as the ground truths with which to compare FRANK’s answers.

### 8.6.4 Results and Discussion

This extended version of FRANK successfully answered about 50% of the queries across all query types, getting more answers right on the fact retrieval queries than on the prediction and ‘extended’ queries. For these queries, the primary source of failure was the rapid explosion of the search tree, due to the increase in the number of candidate operations for the energy domain for decomposition. Also, the version of FRANK used required all child nodes to be looked up before up-propagation. This resulted in unnecessary branching even when the majority of the child nodes had been grounded and were capable of approximating a value to be propagated to their parent. This was, particularly, important in cases where a parent alist had to regress over data values in its child alists, but some child alists had missing values for relevant attributes because no data was found in any KB. In such cases, when no other decomposition options were available, FRANK simply aborted.

---

<sup>3</sup><http://wwwantlr.org/>

On the extensibility of FRANK, we observed that almost all the effort was spent in (1) writing the new wrapper for the knowledge base, since none of the existing wrappers for SPARQL and REST APIs worked on the Microsoft Excel data set, and (2) adding new aggregation operations. All the existing FRANK modules for curating facts, and the algorithm for decomposition and that for the upward propagation of facts remained the same and worked in this new domain. Also, our use of alists in the nodes of the tree allowed data from the new knowledge base, as well as the *unit* attribute and its corresponding operation to be used in the QA process without changing the internal representation of alists and the core FRANK algorithm.

Further, our design decision on FRANK's architecture (see section 7.3) to separate the KB wrappers and inference operations (decomposition and aggregate functions) from the core FRANK algorithm allowed a significant proportion of FRANK's components to work out of the box in the new domain, requiring extensions to only those components of FRANK that were specific to the new domain's knowledge representation and inference operations.

## 8.7 Discussion

FRANK tackles four aspects of our hypothesis. First, FRANK is concerned with increasing the range of queries that can be asked by providing a simple grammar capable of composing a wide variety of query types. The process of internalizing the different query types into the same FRANK tree representation means that we can use the same inference mechanism to solve a wide variety of queries. Secondly, FRANK is concerned with extending the kinds of answers that can be returned compared with existing methods for IR and QA. Our evaluation in this chapter shows that FRANK fills in the gaps that existing QA systems fail to address. That is, inferring answers using arithmetic and statistical techniques when answers that would ordinarily be retrieved are not explicitly found in the KB. As noted in experiments the experiments above, QA systems, search-based systems and computation engines do not focus on uncertainty and therefore must return answers that are completely true or false. This creates an unnecessary constraint on the QA system since it cannot always make that determination.

Importantly, our use of alists in FRANK makes it possible to curate data from multiple data sources on the fly irrespective of the source representation. Since FRANK consistently passes alists to and from decomposition rules, aggregate functions and

uses alists to label nodes in the FRANK tree itself, dynamic curation is possible because of the KB-independent representation that we use in alists in the FRANK tree. The flexibility of alists also means that we can internalise different kinds of queries in almost the same ways.

Program induction also plays vital role in FRANK. At its core, FRANK attempts to solve a query to identifying the sequence of functions (programs) in such a way that together, they form a more complex function from which the query can be answered. Program induction, where we try to induce the sequence of programs (decomposition rules and aggregate functions) that find answers to queries using the generic representation above. The difficulty lies in searching through a large space of program compositions to find the right sequence that leads to an answer. Although systems such as that for Dependency-based compositional semantics (DCS) (Liang et al. (2013)) learn logical forms from queries in order to answer queries, this still makes the assumption that all facts necessary to answer the query are always available. Therefore the parse tree generated remains static and facts from KBs are inserted into placeholders in the tree to answer them. FRANK's advantage it that it responds better to variations in the KBs since the inference tree can be expanded at runtime to explore new ways of grounding nodes.

Although FRANK caters for different kinds of queries, there are other QA systems that handle queries in ways similar to FRANK, but with different approaches. For instance, Liang et al in their DCS formalism, make a strong case for grounded language such that automated systems can leverage the structure in the world to guide the learning and interpretation of language. Although the DCS system performs fairly basic arithmetic operations (given their emphasis on learning the semantic parser from question-answer pairs), in domains that are data-intensive, there is a need to interpolate or extrapolate from what is known. Further, it is important not to overstate the accuracy of such computations. Failure to express explicitly the uncertainty in a answer could lead to users putting undue trust in an answer since the automated system returned it. We avoid the problem of understanding the problem compositionally (i.e. learning the semantic parser) and focus primarily on solving the question, given a more formalized version of it expressed in our FRANK vocabulary. This approach, as shown in the experiments earlier in this chapter allows FRANK to answer queries that require prediction, a feature missing in most QA systems including PowerAqua, ANGIE, OQA, etc. For those that have data manually curated to perform prediction (e.g. Wolfram|Alpha), there is no indication of how certain the answer is. This is probably due to the fact

that in such systems, data for commonly asked queries is hand-curated into the KB. So although it performs well on queries related to population data, GDP, etc., it is unable to attempt queries that require properties that are not commonly used.

Other QA systems use techniques that return a score or rank for the answer that it returns. Not only is this score difficult for the user to explain or interpret in terms of QA system's answer, but it does not give any measure of the correctness of the answer. Systems like OQA (Fader et al. (2014)) use some form of probabilistic reasoning, but is difficult to interpret it within the context of the answer that is returned. Similarly, Wolfram|Alphas' computational engine does not indicate the uncertainty in an answer. This has the potential to mislead users when the answer seems readily retrievable but is completely wrong. By implementing uncertainty (i.e. the *cov*) in FRANK, we get to attempt different kinds of answers including approximate ones.

Dealing with open-domain question answering will require a collaborative effort across many domains. Different systems and their KBs could have specialities relating to specific domains. FRANK could leverage these systems as *decomposers* or *aggregators* (in place of rules and aggregates) such that it works in a more distributed way. By having a core framework that uses a generic representation supported by the web, distributing the QA task becomes much easier. This goes beyond federated queries in that not only is the search query aggregating data from different data endpoints, it is distributing the inference processes across different systems that either have the data or the algorithms to transform or expand an alist a step close towards the desired answer.

Our ability to extend FRANK to new KBs and to add new aggregate functions is shown in sections 7.6 and 8.5.2. This gives FRANK the ability to further increase the range of queries it can answer without changes to the core inference system. In addition, we can easily chain multiple instances of FRANK, thereby distributing the entire inference process as shown in figure 8.8. This is a feature of our QA system that we did not find in the other QA systems evaluated.

## 8.8 Summary

In this chapter, we evaluated our hypotheses for FRANK and tested our assumptions about FRANK. We confirmed, through our experiments, that FRANK is capable of inferring answers by interpolation and extrapolation using function-generating aggregates such as REGRESS. In testing the prediction capabilities of FRANK, we were concerned not just with the accuracy of the prediction, but FRANK's ability to state its

uncertainty about an answer. Our evaluation of prediction is therefore coupled with an assessment of the correlation between FRANK's *cov* for an answer and the absolute normalized error between FRANK's answer and the correct answers that were held out.

Although FRANK solves a different kind of QA problem compared with others that do traditionally NLP-intensive QA, we were able to compare FRANK's capabilities and usefulness to that of comparable systems such as Google's search engine (with access to the World Banks dataset) and WA. We showed that there are classes of queries that other QA systems (including computational engines such as Wolfram|Alpha) do not yet address. In addition, we observed that these system do not give any measures of uncertainty and rather rank answers with some other scoring mechanism that is difficult to interpret with respect to the actual answer returned. Some circumvent the need for uncertainty calculations by using carefully curated data to minimise the errors in their answers. FRANK incorporated the estimation of uncertainty directly in the QA algorithm, allowing it to qualify its answers with error margins that users can easily interpret. This adds to the overall value of the QA system.



# Chapter 9

## Conclusion

### 9.1 Introduction

Query answering has caught the attention of researchers for decades. Various techniques have been used to tackle this challenge as research into other areas of computing such as knowledge representation, NLP, data storage, algorithms and computer architectures have advanced. In many cases, the focus has related to finding answers based on data that can directly be retrieved from KBs. However, in many cases, the data needed is not explicitly available in any knowledge base and must be inferred from other data that is accessible. Doing this comes at the cost of accuracy, especially when performing this kind of inference with data from multiple data sources on the internet, often having different data representations, and inference methods that approximate answer and have the potential of introducing errors into answers.

Our novel approach to query answering explored these problems and resulted in the Functional Reasoning for Acquiring Novel Knowledge (FRANK). In the sections that follow, we summarize our major contributions, some challenges and describe possible directions for future work.

### 9.2 Contributions

We discuss our major contributions in this work by considering the four components of our hypothesis (section 1.4) and how we addressed the related challenges during the research and development of FRANK.



### 9.2.1 Representation and Automatic Curation

In order to solve the problem of answering queries automatically over multiple knowledge sources with diverse representations such as RDF, JSON, and data tables, a representation of data in the the QA framework must be capable of supporting the diverse forms in which data is stored (and received) from a KB. The QA system should also be capable of automatically selecting the appropriate inference operations to apply to data that is retrieved in order to successfully answer queries. The core inference algorithm should also be capable of dealing with multiple kinds of queries and data sources.

The first component of our hypothesis (*HYP-1*) explored these challenges:

***HYP-1:** A uniform internal representation of data facilitates the automatic curation of data from diverse sources and supports a flexible inference algorithm.*

Curation involves the dynamic retrieval of relevant data from KBs and alignment of their representations during inference prior to the aggregation of the data. In FRANK, *alists* (section 4.2.2) (a list of attribute-value pairs of data) represent data in the FRANK tree. An alist gives us the opportunity to extend to the triple representation with the goal of making it easy to dynamically add new attributes to a node in the FRANK tree. That is, alists go beyond the ⟨subject, predicate, object⟩ triple representation used in relational databases and RDF-based KBs in the Linked Data and the Semantic Web domains. Also, the alist is the internal representation of data in FRANK, and labels the nodes of the FRANK tree. Inference operation are also applied to alists. The generic nature of alists enables FRANK to automatically curate data that is relevant for inferring answers in the nodes of the FRANK tree.

Another contribution (discussed in chapter 4) is our recursive algorithm for FRANK that utilizes the alist representation. The QA process in FRANK begins with the conversion of query strings to alists (i.e. internalization). A FRANK tree is constructed by creating a root node from the query alist. To answer the query, the objective of the algorithm is to instantiate the query variables by searching KBs. If search for facts fails, decomposition rules are automatically selected to determine how the node should be decomposed to find an alternative way to instantiate the variables. This process expands the FRANK tree frontier. When leaf nodes are successfully grounded, the process of upward propagation begins. The values of variables instantiated in the leaf node are propagated upward to their parent node. The parent node uses its aggregate function to combine the instantiated variables from its children. This process continues

recursively until the root node is reached, at which point an answer (the instantiated projection variable) and its corresponding uncertainty value can be returned to the user.

### 9.2.2 Inference Operations

Another requirement of a query answering system is its ability to tackle different kinds of queries and offer novel answers even when the required facts are distributed across different KBs. It is impossible to build a single monolithic system that caters for all query types using only look-up from sources that are not restricted to the present day's knowledge sources. We tackle this in second component of our hypothesis:

*HYP-2: Rich forms of inference lead to an extension of the types of questions possible.*

We use the term “rich” to emphasize the fact that the FRANK relies on inference methods that go beyond first-order logic. In FRANK, we use inference operations (chapter 5) to explore different strategies that can solve a query. Using an alist to represent a query at the root of the FRANK tree, we attempt different strategies that will lead to the instantiation of the projection variables in the query alist. If the search for data to instantiate variables fails for an alist, inference operations decompose the alists, given the kinds of attributes values available in the alist. The *lookup* decomposition tries to find synonymous alists (alists with synonymous names of entities) to deal with the scenarios where different names are used to refer to the same entity in different KBs. The *normalization* decomposition creates new alists when normalizing compound alist resulting from nested queries. The presence of time attributes lead to the decomposition of the alist across multiple times (in FRANK's case, years) and we refer to this as *temporal* decomposition (section 5.3.6). If the alist has a geographical location, the *geospatial* decomposition (section 5.3.7) is used to partition the location into its sub-locations and then an attempt is made to instantiate the new child alists.

The decomposition rules also specify the aggregates (5.5) that the decomposed alists use when combining the child alists when variable instantiation is successful during the upward propagation phase of the FRANK algorithm. Our use of function-generating aggregates such as regression (REGRESS) and Gaussian Processes (GP) mean that FRANK predicts (both interpolation and extrapolation) from real-valued data available in child alists. FRANK reuse the inferred functions by caching them and re-applying then in a different branch of the FRANK tree or in a different query to skip multiple steps in the inference process. The selection of the inference operations

is based on the A\* search algorithm and we use a node selection policy based on costs assigned to inference operation based on the number of steps estimated to arrive at a leaf node that can be instantiated.

The kinds of inference operations we use in FRANK, and the modularity of their implementation means that novel answers can be found by composing multiple operations that explore different strategies to solve queries. It gives FRANK the ability to infer answers that no other QA system can.

### 9.2.3 Dealing with Uncertainty

Our next contribution is our handling of uncertainty in FRANK. Given that FRANK uses data from multiple sources on the Internet, it was important to acknowledge the inherent uncertainty due to errors in the data or missing data. Also, given our use of aggregates that generate functions for estimation of values in the FRANK tree, there is the need to incorporate these potential sources of uncertainty in the answer that is returned to the user. This is the third component of our hypothesis:

***HYP-3:** Acknowledging and dealing with uncertainty due to heterogeneous sources of data and inference techniques is useful for placing the answers inferred in context.*

Although several QA and IR systems provide scores, mostly based on the algorithms for merging and ranking of candidate answers (e.g. in (Brill et al. (2001))), to measure the correctness of answers returned, this technique is insufficient to capture the varying forms of uncertainty that need to be dealt with.

Chapter 6 of this thesis dealt with FRANK's handling of uncertainty. Our approach to uncertainty began with the assignment of high prior variances on real-valued data from knowledge bases. FRANK then updates these variances as data is retrieved from the KBs when answering a query. The first challenge encountered when automatically dealing with uncertainty for real-valued data is that data values are often on different scales. For instance, the World Bank has over 15,000 country indicators ranging from those in units to others in millions. However, in FRANK, we wanted to maintain a single prior variance on knowledge bases. We assume that the data observed in KBs have Gaussian noise. Our approach stores uncertainty as a coefficient of variation *cov* (section 6.5.1). The *cov* normalizes the standard deviations by the mean values of the retrieved data. We are then able to store uniformly the variance of KBs irrespective of the magnitude of the actual data values observed. This is equivalent to storing the

variance, but proportional variation is a more intuitive measure of uncertainty in an estimate.

We use the Gaussian distribution to model uncertainty in the data sources, which, in combination with Gaussian likelihood functions, lets us use closed-form equations to sequentially update the parameters of the mean and variance distributions as data is observed. This allows us to sequentially update the *cov* after data is observed from the KB. Additionally, we incorporate the uncertainty as an attribute in the FRANK alists and propagate it along with other variables during the upward propagation phase. In this manner, real-valued answers returned by FRANK have an associated uncertainty value expressed as an error margin around the answer. This, while accounting for the uncertainty in the KB data as well as the inference operations, is intended to give the user a more intuitive way of knowing how good the answer is.

#### 9.2.4 Extensibility in FRANK

Our final contribution is an implementation of FRANK in chapter 7. This implementation tested our ideas about the extensibility of FRANK and also allowed us to evaluate HYP-1 to HYP-3. It relates to the fourth component of our hypothesis:

*HYP-4: An extensible algorithm and architecture allows the QA framework to be easily extended to new data sources and inference operations.*

We implemented the recursive FRANK algorithm using the Java programming language. We implemented a grammar using Antlr (Parr (2013)) for FRANK queries (given that full natural language queries are outside the scope of this work). Our use of alists in FRANK to hold other kinds of variables such as uncertainty and automatically created auxiliary variables that are needed during computation of answers, make the QA framework easily extensible. New kinds of queries can be answered using FRANK by simply adding new attribute-value pairs to the alist. We chose an architecture that enabled modularity of the components of FRANK including the decomposition rules, aggregate function and knowledge bases wrappers. Our design choice meant that we were able to add new modules to FRANK's library of components without having to change the core FRANK algorithm. We successfully added new knowledge sources and the human KB (HEKS) (section 7.6.1) to allow users to interactively provide bits of data and error margins to FRANK as the systems processes a query. A separate project by another student extended the framework to answer queries in the energy domain by adding new KBs and inference operations related to the energy domain.

We also successfully implemented the GP (Gaussian Process) aggregate function as a REST API service on a remote server that FRANK sent requests and data to perform prediction in section 8.5.2. This showed that it was possible to extend FRANK with inference operations that are available on third-party services by simply creating wrappers to access those services. The modularity of FRANK's components and the recursive algorithm also enabled FRANK to execute multiple strategies concurrently.

Our evaluation of these hypotheses in chapter 8 showed that FRANK addresses these features as expected. Uncertainty values calculated by FRANK correlated with the errors between the inferred answers and the true answers. We also demonstrated the range of queries that FRANK can answer and the kinds of inferences that it can support which are currently outside the capability of different QA systems we surveyed. We also demonstrated in section 8.5.1 that our ability to cache functions and explore multiple inference strategies concurrently in our FRANK algorithm significantly impacts the speed of returning answers.

### 9.3 Directions for Future Work

Despite our major contributions in this thesis, there are interesting problems outstanding that could be considered in future research. We outline these below.

- **Generalising geospatial decomposition.** We limited our geospatial decomposition module to geospatial entities such as continents, countries and other place names. However, this can be generalized further by removing the constraint that the sub-entities that form the partition of the entity being decomposed must be of type 'place' (statement 5.1). This will then allow FRANK to perform a general part/whole decomposition of entities with sub-parts. However, there will need to be work done to check the types of entities that need to form the partition required.
- **Expanding the temporal decomposition.** Currently, FRANK works with data that has time attributes in years. It will be useful to deal with data and queries that require a finer-grained temporal decomposition such as months or days or coarser-grained ones such as centuries. It will also be useful to implement Allen's temporal intervals (Allen (1983)) in FRANK.
- **Improving the natural language processing capabilities.** In this work our emphasis has been on the inference framework and its algorithm, using formal

queries to test our system. However, we can further improve the usability of FRANK by allowing natural language questions as inputs to FRANK. A starting point for this could be work done in (Liang et al. (2013)) that will allow FRANK to translate natural language queries to logical ones from which FRANK alists can be created. Additionally, FRANK could also search semi-structured data, such as NL text on the web for facts that could help answer a queries. Also, we can improve the *FindProperty* (section 5.3.4) algorithm with NLP techniques to deal with previously unseen naming conventions in properties or predicates defined in KBs using techniques from (Bilenko et al. (2003)) and (Khaitan et al. (2009)).

- **Improving FRANK's user interface.** FRANK would also benefit from a more interactive user interface. This could include a template-based user interface (UI) to help users compose queries of different kinds, a more intuitive rendering of the FRANK's execution log that contains provenance information as well as a summarization of information in the log that serves as an explanation of the answer arrived at. A UI enhancement would also enable a user to visualize the functions FRANK generates for prediction, allowing the user to better understand the answer returned.
- **Uncertainty for non-real-valued data.** Queries we used for this thesis required real-valued answers for which our current handling of uncertainty works. We can, in the future, expand our calculation of uncertainty in a more systematic way for queries that uses qualitative data and returns non-real-valued answers. We can also evaluate the quality of an answer based on the uncertainty that FRANK returns with an answer and the perceived "goodness" a user associates with the answer.
- **Learning to select inference strategies.** Given that FRANK can be equipped with several inference operations, it is possible to improve on the heuristics-based approach for strategy selection to use a learning-based one. That is, FRANK could learn to select the appropriate decomposition rules and aggregate functions as it attempts different strategies that lead to success or failures (i.e. answers with very small or really large error margins). Several machine learning techniques such as reinforcement learning and deep learning can be used. This can also be extended to leverage advances in automatic model selection for real-valued data to improve answer accuracy.

- **More KBs and inference operations.** Generally, we can increase FRANK capabilities by adding more knowledge base wrappers such that FRANK can look up data in knowledge bases beyond those that the current version of FRANK uses. FRANK's range of answer can also be further increased by adding more inference operations that are capable of newer forms of decomposition and aggregation such as differentiation and integration of functions that are inferred. Additionally, the FRANK algorithm can be optimized further and evaluated on its speed and efficiency.
- **Negation in queries.** Dealing with negation in queries when reasoning with open world data can be very costly. Given FRANK's ability to approximate answers, it will be interesting to see how negation can be included in FRANK queries.

## 9.4 Final Remarks

The FRANK Inference Framework (FRANK) provides a novel approach to query answering over heterogeneous web knowledge bases by using inference methods, including functional ones that take advantage of real-valued data that is usually unused for prediction by many QA systems. Compared to existing QA systems that (1) only minimally deal with the real-valued data available in KBs and use algorithms including prediction to find novel answers to queries, and (2) are unable to deal properly with the uncertainty that is inherent in these web KBs, FRANK tackles these challenges to provide a new kinds of answers to queries.

In this thesis, we have explored different aspects of the QA process. We proposed and implemented FRANK for recursively exploring strategies in an inference tree to solve queries, using our new alist representation as a foundation to represent knowledge that is curated automatically. We also implemented our ideas for handling uncertainty using a Bayesian approach. We implemented the framework using a modular design that allows for FRANK to be extended easily and took advantage of facts and function caching. We finally successfully evaluated different aspects of FRANK with a variety of queries.

Overall, our novel contributions can significantly improve current techniques in question answering in terms of the range of answers that can be sought and the uncertainty associated with them.

# Appendix A

## Relevant Probability Theory

In probability theory, a random variable is a variable representing the possible states in a given sample space. A random variable  $X$  takes on different values  $x$  (the *domain* of  $X$ ) from the sample space  $S$ . A probability model associates a measure of uncertainty called the probability  $P(X = x)$ , for every value  $x$  in the sample space of  $S$ . The main axioms of probability theory state that:

- the probability of every item in the sample space has a value between 0 and 1.

That is,

$$0 \leq P(X = x) \leq 1$$

- the sum of all probabilities in the sample space is 1. That is,

$$\sum_{x \in S} P(X = x) = 1$$

An enumeration of all the value of  $P(X = x)$  defines the probability distribution of the random variable  $X$ . When the random variable  $X$  is discrete, the distribution is called a probability mass function. When  $X$  is characterized by a continuous values, the distribution is called a probability density function.

### A.1 Independence and Conditional Independence

In many real-world cases, the probability of  $x$  taking on a specified value is conditioned on another event with a random variable  $Y$ , sometimes called the *evidence*. This is expressed as  $P(X = x|Y = y)$  as is referred to as the conditional probability or the posterior probability.



The conditional probability  $P(X = x|Y = y)$  can be expressed in terms of the prior probability by:

$$P(X = x|Y = y) = \frac{P(X = x \text{ and } Y = y)}{P(Y = y)}$$

where  $P(X = x \text{ and } Y = y)$  is the *joint probability* distribution of  $X$  and  $Y$ , and

$$P(Y = y)$$

is prior probability of  $Y$ . By multiplying both sides of the equation by the prior probability of  $Y$ , we can express the conditional probability as the *product rule* given by

$$P(X = x \text{ and } Y = y) = P(X = x|Y = y)P(Y = y)$$

If the random variables  $X$  and  $Y$  are independent of each other, then the joint probability distribution becomes the product of both  $X$  and  $Y$ . That is,

$$P(X = x \text{ and } Y = y) = P(X = x)P(Y = y)$$

The notion of conditional independence extends the idea of independence but in the presence of a third random variable. That is, the conditional independence of random variable  $X$  and  $Y$  given random variable  $Z$  is

$$P(X = x \text{ and } Y = y|Z = z) = P(X = x|Z = z)P(Y = y|Z = z) \quad (\text{A.1})$$

that states that in given evidence of the occurrence of event  $Z$ , the random variables  $X$  and  $Y$  are independent.

## A.2 Bayesian Statistics

Bayes' rule is one of the fundamental elements of probabilistic reasoning. The rule uses the product rule to express the conditional distributions between two variables in such a way that makes it possible to calculate, particularly when the conditional distribution between the two variables is easier to obtain (or is given) in one direction of the conditioning (e.g.  $P(Y = y|X = x)$ ), but not its reverse ( $P(X = x|Y = y)$ ).

Since the product rule can be written as

$$P(X = x \text{ and } Y = y) = P(X = x|Y = y)P(Y = y)$$

and

$$P(X = x \text{ and } Y = y) = P(Y = y|X = x)P(X = x) .$$

Combining both expressions by equating the right hand expressions gives

$$P(Y = y|X = x) = \frac{P(X = x|Y = y)P(Y = y)}{P(X = x)} \quad (\text{A.2})$$

Equation A.2 is called *Bayes' Rule*.

### A.3 Gaussian Distribution

The Gaussian distribution (also referred to as the normal distribution) is a common statistical distribution for continuous random variables. For a random variable  $X$ , its Gaussian probability density function is defined by two parameters; a mean (and mode)  $\mu$ , and a variance  $\sigma^2$  in the form:

$$\mathcal{N}(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp^{-\frac{1}{2\sigma^2}(x-\mu)^2} \quad (\text{A.3})$$

For a random variable  $X$ , the probability that the  $X$  takes the value  $x$ , written as:

$$P(X = x) = \mathcal{N}(x|\mu, \sigma^2) \quad (\text{A.4})$$

means that the Gaussian distribution is centered at  $\mu$  and has a variance of  $\sigma^2$ .

The Gaussian is commonly used as an approximation of several other distributions, such as the uniform and binomial distributions, as the number of observations approach  $\infty$  (infinity), as proved in (Walker (1969)).

### A.4 Combining Variances of Gaussians

Function	Variance	Standard Deviation
$f = aA$	$\sigma_f^2 = a^2 \sigma_A^2$	$\sigma_f =  a  \sigma_A$
$f = aA + bB$	$\sigma_f^2 = a^2 \sigma_A^2 + b^2 \sigma_B^2 + 2ab \sigma_{AB}$	$\sigma_f = \sqrt{a^2 \sigma_A^2 + b^2 \sigma_B^2 + 2ab \sigma_{AB}}$
$f = aA - bB$	$\sigma_f^2 = a^2 \sigma_A^2 + b^2 \sigma_B^2 - 2ab \sigma_{AB}$	$\sigma_f = \sqrt{a^2 \sigma_A^2 + b^2 \sigma_B^2 - 2ab \sigma_{AB}}$
$f = AB$	$\sigma_f^2 \approx f^2 \left[ \left( \frac{\sigma_A}{A} \right)^2 + \left( \frac{\sigma_B}{B} \right)^2 + 2 \frac{\sigma_{AB}}{AB} \right]$	$\sigma_f \approx  f  \sqrt{\left( \frac{\sigma_A}{A} \right)^2 + \left( \frac{\sigma_B}{B} \right)^2 + 2 \frac{\sigma_{AB}}{AB}}$
$f = \frac{A}{B}$	$\sigma_f^2 \approx f^2 \left[ \left( \frac{\sigma_A}{A} \right)^2 + \left( \frac{\sigma_B}{B} \right)^2 - 2 \frac{\sigma_{AB}}{AB} \right]$	$\sigma_f \approx  f  \sqrt{\left( \frac{\sigma_A}{A} \right)^2 + \left( \frac{\sigma_B}{B} \right)^2 - 2 \frac{\sigma_{AB}}{AB}}$
$f = aA^b$	$\sigma_f^2 \approx (abA^{b-1} \sigma_A)^2 = \left( \frac{fb\sigma_A}{A} \right)^2$	$\sigma_f \approx  abA^{b-1} \sigma_A  = \left  \frac{fb\sigma_A}{A} \right $
$f = a \ln(bA)$	$\sigma_f^2 \approx \left( a \frac{\sigma_A}{A} \right)^2$	$\sigma_f \approx \left  a \frac{\sigma_A}{A} \right $
$f = a \log_{10}(A)$	$\sigma_f^2 \approx \left( a \frac{\sigma_A}{A \ln(10)} \right)^2$	$\sigma_f \approx \left  a \frac{\sigma_A}{A \ln(10)} \right $
$f = ae^{bA}$	$\sigma_f^2 \approx f^2 (b\sigma_A)^2$	$\sigma_f \approx  f(b\sigma_A) $
$f = a^{bA}$	$\sigma_f^2 \approx f^2 (b \ln(a) \sigma_A)^2$	$\sigma_f \approx  f(b \ln(a) \sigma_A) $
$f = a \sin(bA)$	$\sigma_f^2 \approx [ab \cos(bA) \sigma_A]^2$	$\sigma_f \approx  ab \cos(bA) \sigma_A $
$f = a \cos(bA)$	$\sigma_f^2 \approx [ab \sin(bA) \sigma_A]^2$	$\sigma_f \approx  ab \sin(bA) \sigma_A $
$f = A^B$	$\sigma_f^2 \approx f^2 \left[ \left( \frac{B}{A} \sigma_A \right)^2 + (\ln(A) \sigma_B)^2 + 2 \frac{B \ln(A)}{A} \sigma_{AB} \right]$	$\sigma_f \approx  f  \sqrt{\left( \frac{B}{A} \sigma_A \right)^2 + (\ln(A) \sigma_B)^2 + 2 \frac{B \ln(A)}{A} \sigma_{AB}}$

Figure A.1: Uncertainty propagation

(Source: [https://en.wikipedia.org/wiki/Propagation\\_of\\_uncertainty](https://en.wikipedia.org/wiki/Propagation_of_uncertainty))

# Appendix B

## FRANK Query Grammar

```
query ::= method
method ::= unaryOperation | compareOperation | boolOperation |
        calculusOperation
unaryOperation ::= unaryOperationName '(' variable_param ','
                ( predicateExpr ( ':' formula )* | method ) ')'
unaryOperationName ::= 'VALUE' | 'VALUES' | 'COUNT' | 'MIN' | 'MAX'
                    | 'COMP' | 'SUM' | 'AVG' | 'FUNC' | 'REGRESS'
compareOperation ::= compareOperationName '(' variable_param ','
                    unaryOperation ',' unaryOperation ')'
compareOperationName ::= GREATERTHAN | LESSTHAN | EQUALTO
boolOperation ::= boolOperationName '(' variable_param ( ','
                    ( boolOperation | compareOperation ) )* ')'
boolOperationName ::= 'AND' | 'OR' | 'NOT'
calculusOperation
::= ( ( ( 'DERIV' | 'CHANGERATE' ) '(' variable_param ','
        predicateExpr | ( 'INTEGRAL' | 'SUM' ) '('
        variable_param ',' predicateExpr ',' identifier )
        ',' identifier ')' )?
formula ::= disjunction | conjunction
disjunction ::= logicExpr ( '|' logicExpr )*
conjunction ::= logicExpr ( '&' logicExpr )*
logicExpr ::= predicateExpr | '(' formula ')'
predicateExpr ::= pair | triple | quad
pair ::= identifier '(' term ')'
triple ::= identifier '(' term ',' term ')'
quad ::= identifier '(' term ',' term ',' term ')'
term ::= identifier | variable | method | set
variable ::= BOUND_VARIABLE | FREE_VARIABLE
variable_param ::= variable | variable_tuple
```

```
variable_tuple ::= '<' variable ( ',' variable )* '>'
identifier ::= IDENTIFIER
set ::= SET
GREATERTHAN ::= 'GREATERTHAN' | 'GT'
LESSTHAN ::= 'LESSTHAN' | 'LT'
EQUALTO ::= 'EQUALTO' | 'EQ'
BOUND_VARIABLE ::= '$' CHARACTER*
FREE_VARIABLE ::= '?' CHARACTER*
SET ::= '{' ( IDENTIFIER ',' '?' )* '}'
IDENTIFIER ::= [0-9a-zA-Z_#x2D] CHARACTER*
CHARACTER ::= [0-9a-zA-Z_.]
WS ::= [ #x9#xD#xA]+
?
```

# Appendix C

## Code for Human Expert Knowledge Source

```
1 public class InteractiveUserData implements KbInterface {
2     public static final String USER= "user";
3     public ArrayList<RifNode> FindPropertyValues(RifNode rifNode, String
4         nodeAttribute) {
5         if(nodeAttribute.equalsIgnoreCase(RifNode.OBJECT))
6             return FindPropertyObject(rifNode);
7         else if(nodeAttribute.equalsIgnoreCase(RifNode.TIME)) {
8             return FindPropertyTime(rifNode);
9         }
10        else
11            return null;
12    }
13    private ArrayList<RifNode> FindPropertyObject(RifNode rifNode){
14        ArrayList<RifNode> returnVals = new ArrayList<>();
15        try {
16            String message = String.format("The %s of %s", rifNode.getPredicate(),
17                rifNode.getSubject());
18            if(rifNode.hasItem(RifNode.TIME))
19                message += String.format(" in %s", rifNode.getTime());
20            message += " ?";
21            String response = ReadFromConsole(message, false);
22            if (!response.trim().isEmpty() && !response.trim().equalsIgnoreCase("-
23                ")){
24                RifNode n = new RifNode(rifNode.getSubject(),rifNode.getPredicate(),
25                    response);
```

```

23     //ask of error margin
24     message = "How sure are you of this answer? (Error margin +/-) ";
25     String error = ReadFromConsole(message, true);
26     if (!error.trim().isEmpty() && !error.trim().equalsIgnoreCase("-")){
27         if(RifUtils.IsNumeric(error))
28             n.setUncertainty(error);
29     }
30     returnVals.add(n);
31 }
32 }
33 catch(Exception ex){ ex.printStackTrace(); }
34 return returnVals;
35 }
36
37 private ArrayList<RifNode> FindPropertyTime(RifNode rifNode){
38     ArrayList<RifNode> returnVals = new ArrayList<>();
39     try {
40         String message = String.format("When %s of %s is/was %s ?", rifNode.
41             getPredicate(), rifNode.getSubject(), rifNode.getObject());
42         String response = ReadFromConsole(message, false);
43         if (!response.trim().isEmpty() && !response.trim().equalsIgnoreCase("-
44             ")){
45             RifNode n = new RifNode(rifNode.getSubject(),rifNode.getPredicate(),
46                 response);
47             //ask of error margin
48             message = "How sure are you of this answer? (Error margin +/-) ";
49             String error = ReadFromConsole(message, true);
50             if (!error.trim().isEmpty() && !error.trim().equalsIgnoreCase("-")){
51                 if(RifUtils.IsNumeric(error))
52                     n.setUncertainty(error);
53             }
54             returnVals.add(n);
55         }
56     }
57     catch(Exception ex){ ex.printStackTrace(); }
58     return returnVals;
59 }
60
61 private String ReadFromConsole(String inputMessage, Boolean closeStream){
62     BufferedReader br = null;
63     String userInput = "";
64     try {

```

```
62     InputStreamReader inStream = new InputStreamReader(System.in);
63     br = new BufferedReader(inStream);
64     System.out.print(inputMessage + " : ");
65     userInput = br.readLine();
66     if ("-".equals(userInput))
67         System.out.println("\nYour response was: " + userInput);
68 }
69 catch (IOException e) { e.printStackTrace(); }
70 finally {
71     if (br != null && closeStream) {
72         try {
73             br.close();
74         } catch (IOException e) { e.printStackTrace(); }
75     }
76 }
77 return userInput;
78 }
79 }
```





# Appendix D

## Evaluation Queries

```
1 VALUE(?x,population_total(France,?x,2010))
2 VALUE(?x,rural_population(India,?x,1990))
3 VALUE(?x,gdp(Brazil,?x,2008))
4 VALUE(?x,agricultural_land(norway,?x,2001))
5 VALUE(?x,unemployment_total(Germany,?x,2002))
6 VALUE(?x,total_population(UK,?x,2010))
7 VALUE(?x,birth_rate(China,?x,2005))
8 VALUE(?x,urban_population(Ghana,?x,1998))
9 VALUE(?x,fertility_rate(Chile,?x,2010))
10 VALUE(?x,female_unemployment(Japan,?x,2011))
11 SUM(?y,COMP(<$x,?y>,total_population($x,?y,2001):type($x,country) & location
    ($x,Europe)))
12 AVG(?y,COMP(<$x,?y>,gdp($x,?y,2003):type($x,country) & location($x,
    South_America)))
13 MAX($y,COMP(<?x,$y>,agricultural_land(?x,$y,2006):type(?x,country) &
    location(?x,Africa)))
14 MIN($y,COMP(<?x,$y>,gdp_rate(?x,$y,2007):type(?x,country) & location(?x,Asia
    )))
15 MAX($y,COMP(<?x,$y>,unemployment_male(?x,$y,2005):type(?x,country) &
    location(?x,Europe)))
16 SUM(?y,COMP(<$x,?y>,wind_energy_consumption($x,?y,2007):type($x,country) &
    location($x,Europe)))
17 MIN($y,COMP(<?x,$y>,female_unemployment(?x,$y,2003):type(?x,country) &
    location(?x,Asia)))
18 MAX(?y,COMP(<$x,?y>,urban_population($x,?y,2001):type($x,country) & location
    ($x,Africa)))
19 MAX($y,COMP(<?x,$y>,birth_rate(?x,$y,1995):type(?x,country) & location(?x,
    South_America)))
20 MAX($y,COMP(<?x,$y>,cereal_export(?x,$y,2000):type(?x,country) & location(?x
```

```

    ,Africa)))
21 MIN(?y,COMP(<$x,?y>,urban_population($x,?y,2001):is($x,Ghana) | is($x,
    Nigeria)))
22 MIN(?y,COMP(<$x,?y>,agricultural_land($x,?y,2001):is($x,Ghana) | is($x,
    Nigeria)))
23 MAX($b,COMP(<?a,$b>,agricultural_land(?a,$b,2010):is(?a,Ghana) | is(?a,
    Nigeria)))
24 GT(?x, VALUE(?x,fertility_rate(Chile,?x,2012)), VALUE(?x,is(?x,1.2)))
25 GT(?x, VALUE(?x,fertility_rate(Chile,?x,2012)), VALUE(?x,fertility_rate(
    Ghana,?x,2012)))
26 VALUE(?y,gdp(MAX($b,COMP(<?a,$b>,agricultural_land(?a,$b,2010):type(?a,
    country) & location(?a,Africa))),?y,2010))
27 VALUE(?y,urban_population(MAX($b,COMP(<?a,$b>,agricultural_land(?a,$b,2010):
    is(?a,Ghana) | is(?a,Nigeria))),?y,2010))
28 GT(?y, MAX(?y,rural_population(COMP(<$a,?y>,urban_population($a,?y,2003):
    type($a,country)&location($a,Africa)),?y,2003)), MIN(?y,rural_population
    (COMP(<$b,?y>,urban_population($b,?y,2003):type($b,country) & location(
    $b,Africa)),?y,2003)))
29 VALUE(?y,female_unemployment(MAX($b,COMP(<?a,$b>,male_unemployment(?a,$b
    ,2000):type(?a,Country) & location(?a,Europe))),?y,2011))
30 LT(?y, VALUE(?y,gdp(MAX($b,COMP(<?a,$b>,cereal_export(?a,$b,2005):type(?a,
    country) & location(?a,Africa))),?y,2005)), VALUE(?y,gdp(MAX($b,COMP(<?a
    ,$b>,cereal_export(?a,$b,2005):type(?a,country) & location(?a,
    South_America))))
31 GT(?y, VALUE(?y,gdp(MAX($b,COMP(<?a,$b>,arable_land(?a,$b,2010):type(?a,
    country) & location(?a,South_America))),?y,2010)), VALUE(?y,gdp(MAX($b,
    COMP(<?a,$b>,arable_land(?a,$b,2010):type(?a,country) & location(?a,
    Europe))))
32 GT(?y, VALUE(?x,agricultural_land(Africa,?x,2010)), VALUE(?x,
    agricultural_land(South_America,?x,2010)))
33 LT(?x, VALUE(?x,fertility_rate(Japan,?x,2010)), VALUE(?x,fertility_rate(
    Cameroon,?x,2010)))
34 LT(?y, VALUE(?x,female_unemployment(Germany,?x,2011)), AVG(?y,COMP(<$x,?y>,
    female_unemployment($x,?y,2010):type($x,Country)&location($x,Europe))))
35 VALUE(?y,birth_rate(MIN($b,COMP(<?a,$b>,rural_population(?a,$b,2004):type(?a
    ,Country) & location(?a,Asia))),?y,2004))
36 VALUE(?y,urban_population(MAX($b,COMP(<?a,$b>,population_total(?a,$b,2002):
    type(?a,Country) & location(?a,Asia))),?y,2002))
37 VALUE(?x,population_total(Japan,?x,2017))
38 VALUE(?x,birth_rate(China,?x,2022))
39 VALUE(?x,urban_population(Ghana,?x,2019))
40 MAX($y,COMP(<?x,$y>,youth_unemployment_rate(?x,$y,2018):type(?x,country) &

```

```

    location(?x,Africa))
41 VALUE(?y,female_unemployment(MAX($b,COMP(<?a,$b>,male_population(?a,$b,2004)
    :type(?a,country) & location(?a,Europe))),?y,2019))
42 GT(?y, VALUE(?x,birth_rate(Africa,?x,2025)), VALUE(?x,birth_rate(Brazil,?x
    ,2025)))
43 LT(?y, VALUE(?x,youth_unemployment(Africa,?x,2010)), AVG(?y,COMP(<$x,?y>,
    youth_unemployment_rate($x,?y,2021):type($x,country) & location($x,
    Europe)))
44 VALUE(?y,labour_force(MAX($b,COMP(<?a,$b>,rural_population(?a,$b,2004):type
    (?a,Country) & location($a,Asia))),?y,2022))
45 MAX($y,COMP(<?x,$y>,arable_land(?x,$y,2017):type(?x,Country)&location(?x,
    South_America))
46 VALUE(?y,gdp(MAX($b,COMP(<?a,$b>,labor_force(?a,$b,2018):type(?a,Country) &
    location($a,Europe))),?y,2010))
47 LT(?x, VALUE(?x,fertility_rate(Japan,?x,2016)), VALUE(?x,fertility_rate(
    Cameroon,?x,2016))
48 VALUE(?x,urban_population(Ghana,?x,2019))
49 MIN(?y,COMP(<$x,?y>,urban_population($x,?y,2001):is($x,Ghana) | is($x,
    Nigeria))
50 LT(?x, VALUE(?x,fertility_rate(Japan,?x,2010)), VALUE(?x,fertility_rate(
    Cameroon,?x,2010))
51 GT(?x, VALUE(?x,is(?x,1)), VALUE(?x,is(?x,2)))
52 GT(?x, VALUE(?x,fertility_rate(Chile,?x,2012)), VALUE(?x,is(?x,1.2)))
53 MAX($y,COMP(<?x,$y>,arable_land(?x,$y,2017):type(?x,Country)&location(?x,
    South_America))
54 VALUE(?x,total_final_energy_consumption(brazil,?x,2000))
55 VALUE(?x,access_to_electricity(panama,?x,2005))
56 VALUE(?x,urban_population(france,?x,2010))
57 VALUE(?x,hydro_energy_consumption(spain,?x,2007))
58 VALUE(?x,total_population(malta,?x,2011))
59 VALUE(?x,total_final_energy_consumption(mexico,?x,2002))
60 VALUE(?x,renewable_energy_electricity_output(sao_tome_and_principe,?x,2011))
61 VALUE(?x,access_to_electricity(thailand,?x,2007))
62 VALUE(?x,total_final_energy_consumption(hungary,?x,2002))
63 VALUE(?x,access_to_electricity(bolivia,?x,2005))
64 VALUE(?x,urban_population(guinea-bissau,?x,1998))
65 VALUE(?x,urban_population(burundi,?x,1999))
66 VALUE(?x,rural_population(suriname,?x,2008))
67 VALUE(?x,total_population(malta,?x,2007))
68 VALUE(?x,unemployment_total(netherlands,?x,2002))
69 VALUE(?x,total_population(guinea-bissau,?x,2010))
70 VALUE(?x,total_population(indonesia,?x,2001))

```

```

71 VALUE(?x,rural_access_to_electricity(ukraine,?x,2009))
72 VALUE(?x,rural_access_to_electricity(aruba,?x,2001))
73 VALUE(?x,fertility_rate(cambodia,?x,2001))
74 VALUE(?x,hydro_energy_consumption(chad,?x,2011))
75 VALUE(?x,total_electricity_output(new_zealand,?x,2010))
76 VALUE(?x,rural_population(cayman_islands,?x,1999))
77 VALUE(?x,total_electricity_output(sierra_leone,?x,2002))
78 VALUE(?x,total_population(kenya,?x,1998))
79 VALUE(?x,renewable_energy_electricity_output(serbia,?x,2003))
80 VALUE(?x,unemployment_total(romania,?x,2003))
81 VALUE(?x,total_final_energy_consumption(slovenia,?x,2011))
82 VALUE(?x,unemployment_total(french_polynesia,?x,1998))
83 VALUE(?x,total_final_energy_consumption(vietnam,?x,2003))
84 VALUE(?x,rural_population(united_kingdom,?x,2006))
85 VALUE(?x,fertility_rate(equatorial_guinea,?x,2012))
86 VALUE(?x,rural_access_to_electricity(micronesia_fed._sts.,?x,1998))
87 VALUE(?x,renewable_electricity_output(Asia,?x,2005))
88 MIN(?y,COMP(<$x,?y>,rural_population($x,?y,2002):type($x,country) & location
($x,asia)))
89 GT(?y,VALUE(?y,rural_access_to_electricity(MIN($b,COMP(<?a,$b>,surface_area
(?a,$b,2009):type(?a,country) & location(?a,South_America))),?y,2003)),
VALUE(?y,rural_access_to_electricity(MAX($b,COMP(<?a,$b>,surface_area(?a
,$b,2009):type(?a,country) & location(?a,Europe))),?y,2003)))
90 LT(?y,VALUE(?y,fertility_rate(MIN($b,COMP(<?a,$b>,birth_rate(?a,$b,2002):
type(?a,country) & location(?a,South_America))),?y,2001)), VALUE(?y,
fertility_rate(MAX($b,COMP(<?a,$b>,birth_rate(?a,$b,2002):type(?a,
country) & location(?a,Africa))),?y,2001)))
91 LT(?y,VALUE(?y,rural_access_to_electricity(MIN($b,COMP(<?a,$b>,gdp(?a,$b
,2011):type(?a,country) & location(?a,North_America))),?y,2012)), VALUE
(?y,rural_access_to_electricity(MIN($b,COMP(<?a,$b>,gdp(?a,$b,2011):type
(?a,country) & location(?a,North_America))),?y,2012)))
92 MAX(?y,COMP(<$x,?y>,rural_population($x,?y,2003):type($x,country) & location
($x,africa)))
93 MAX(?y,COMP(<$x,?y>,rural_population($x,?y,2008):type($x,country) & location
($x,south_america)))
94 SUM(?y,COMP(<$x,?y>,urban_access_to_electricity($x,?y,2006):type($x,country)
& location($x,asia)))
95 MIN(?y,COMP(<$x,?y>,urban_access_to_electricity($x,?y,2001):type($x,country)
& location($x,africa)))
96 MAX(?y,COMP(<$x,?y>,urban_access_to_electricity($x,?y,2012):type($x,country)
& location($x,south_america)))
97 AVG(?y,COMP(<$x,?y>,rural_population($x,?y,2012):type($x,country) & location

```

```
($x,africa))
98 MIN(?y,COMP(<$x,?y>,rural_population($x,?y,2002):type($x,country) & location
($x,asia))
99 SUM(?y,COMP(<$x,?y>,rural_population($x,?y,2009):type($x,country) & location
($x,north_america))
100 VALUE(?x,rural_access_to_electricity(Africa,?x,2005))
101 VALUE(?x,cereal_production(Europe,?x,2009))
102 VALUE(?x,cereal_production(Asia,?x,1999))
103 VALUE(?x,cereal_production(South_America,?x,1998))
104 VALUE(?x,urban_access_to_electricity(South_America,?x,2011))
105 VALUE(?x,urban_access_to_electricity(Africa,?x,2010))
106 VALUE(?x,urban_access_to_electricity(Asia,?x,2007))
107 VALUE(?x,renewable_electricity_output(Europe,?x,2005))
```



# Bibliography

- Abourbih, J. A., Blaney, L., Bundy, A., and McNeill, F. (2010a). A single-significant-digit calculus for semi-automated guesstimation. In *Automated Reasoning*, pages 354–368. Springer.
- Abourbih, J. A., Bundy, A., and McNeill, F. (2010b). Using linked data for semi-automatic guesstimation. In *AAAI Spring Symposium: Linked Data Meets Artificial Intelligence*.
- Agarwal, S., Milner, H., Kleiner, A., Talwalkar, A., Jordan, M., Madden, S., Mozafari, B., and Stoica, I. (2014). Knowing when you’re wrong: building fast and reliable approximate query processing systems. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 481–492. ACM.
- Allen, J. F. (1983). Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843.
- Amazon (2016). Amazon Alexa. <https://developer.amazon.com/alexa>. Accessed: 2017-08-15.
- Anderson, J. C., Lehnardt, J., and Slater, N. (2010). *CouchDB: the definitive guide*. "O’Reilly Media, Inc."
- Andreas, J., Rohrbach, M., Darrell, T., and Klein, D. (2015). Deep compositional question answering with neural module networks. arxiv preprint. *arXiv preprint arXiv:1511.02799*, 2.
- Andreas, J., Rohrbach, M., Darrell, T., and Klein, D. (2016). Learning to compose neural networks for question answering. *arXiv preprint arXiv:1601.01705*.
- Antoniou, G. and Van Harmelen, F. (2004). *A semantic web primer*. MIT press.



- Apple Inc. (2011). Apple Siri. <http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm>. Accessed: 2017-08-15.
- Arnold, K., Gosling, J., Holmes, D., and Holmes, D. (2000). *The Java programming language*, volume 2. Addison-wesley Reading.
- Babcock, B., Chaudhuri, S., and Das, G. (2003). Dynamic sample selection for approximate query processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 539–550. ACM.
- Banko, M., Brill, E., Dumais, S., Lin, J., and Way, M. (2002). AskMSR: Question answering using the worldwide Web. *Proceedings of 2002 AAAI Spring Symposium on Mining Answers*, (March):1–2.
- Bao, J., Duan, N., Zhou, M., and Zhao, T. (2014). Knowledge-based question answering as machine translation. *Cell*, 2(6).
- Battle, R. and Kolas, D. (2011). Geosparql: enabling a geospatial semantic web. *Semantic Web Journal*, 3(4):355–370.
- Beckett, D. and McBride, B. (2004). Rdf/xml syntax specification (revised). *W3C recommendation*, 10.
- Beek, W., Rietveld, L., Bazoobandi, H. R., Wielemaker, J., and Schlobach, S. (2014). Lod laundromat: a uniform way of publishing other people’s dirty data. In *International Semantic Web Conference*, pages 213–228. Springer.
- Bellman, R. E. and Zadeh, L. A. (1977). Local and fuzzy logics. In *Modern uses of multiple-valued logic*, pages 103–165. Springer.
- Berners-Lee, T. et al. (1998). Semantic web road map.
- Berners-Lee, T., Hendler, J., Lassila, O., et al. (2001). The semantic web. *Scientific american*, 284(5):28–37.
- Bilenko, M., Mooney, R., Cohen, W., Ravikumar, P., and Fienberg, S. (2003). Adaptive name matching in information integration. *IEEE Intelligent Systems*, 18(5):16–23.
- Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

- Bizer, C., Heath, T., and Berners-Lee, T. (2009a). Linked data-the story so far. *Semantic Services, Interoperability and Web Applications: Emerging Concepts*, pages 205–227.
- Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., and Hellmann, S. (2009b). DBpedia - A crystallization point for the Web of Data. *Journal of Web Semantics*, 7(3):154–165.
- Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M. K. (1987). Occam's razor. *Information processing letters*, 24(6):377–380.
- Boley, H., Paschke, A., and Shafiq, O. (2010). Ruleml 1.0: the overarching specification of web rules. In *International Workshop on Rules and Rule Markup Languages for the Semantic Web*, pages 162–178. Springer.
- Bollacker, K., Evans, C., Paritosh, P., Sturge, T., and Taylor, J. (2008). Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1247–1250. AcM.
- Bordogna, G., Campi, A., Ronchi, S., and Psaila, G. (2009). Query disambiguation based on novelty and similarity user's feedback. In *Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology-Volume 03*, pages 125–128. IEEE Computer Society.
- Brachman, R. J. and Schmolze, J. G. (1988). An overview of the kl-one knowledge representation system. In *Readings in Artificial Intelligence and Databases*, pages 207–230. Elsevier.
- Brill, E., Lin, J. J., Banko, M., Dumais, S. T., Ng, A. Y., et al. (2001). Data-intensive question answering. In *TREC*, volume 56, page 90.
- Buil-Aranda, C., Arenas, M., Corcho, O., and Polleres, A. (2013). Federating queries in sparql 1.1: Syntax, semantics and evaluation. *Web Semantics: Science, Services and Agents on the World Wide Web*, 18(1):1–17.
- Bundy, A., Sasnauskas, G., and Chan, M. (2013). Solving Guesstimation Problems Using the Semantic Web : Four Lessons from an Application. *Semantic Web*, pages 1–20.

- Bunescu, R. C. and Pasca, M. (2006). Using encyclopedic knowledge for named entity disambiguation. In *Eacl*, volume 6, pages 9–16.
- Cao, Y. and Fan, W. (2017). Data driven approximation with bounded resources. *Proceedings of the VLDB Endowment*, 10(9):973–984.
- Carlson, A., Betteridge, J., Kisiel, B., Settles, B., Hruschka Jr, E. R., and Mitchell, T. M. (2010). Toward an architecture for never-ending language learning. In *AAAI*, volume 5, page 3.
- Chappelier, J.-C., Rajman, M., et al. (1998). A generalized cyk algorithm for parsing stochastic cfg. *TAPD*, 98(133-137):5.
- Chaudhuri, S., Ding, B., and Kandula, S. (2017). Approximate query processing: No silver bullet. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 511–519. ACM.
- Christopher, D. M., Prabhakar, R., and Hinrich, S. (2008). *Introduction to Information Retrieval*. Cambridge University Press.
- Chu-Carroll, J., Czuba, K., Prager, J., and Ittycheriah, A. (2003). In question answering, two heads are better than one. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 24–31. Association for Computational Linguistics.
- Codd, E. F. (1971). A data base sublanguage founded on the relational calculus. In *Proceedings of the 1971 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*, pages 35–68. ACM.
- Cohen, W., Ravikumar, P., and Fienberg, S. (2003). A comparison of string metrics for matching names and records. In *Kdd workshop on data cleaning and object consolidation*, volume 3, pages 73–78.
- Cohn, A. G. and Hazarika, S. M. (2001). Qualitative spatial representation and reasoning: An overview. *Fundamenta informaticae*, 46(1-2):1–29.
- Cooper, W. S. (1964). Fact retrieval and deductive question-answering information retrieval systems. *Journal of the ACM (JACM)*, 11(2):117–137.

- Cornolti, M., Ferragina, P., Ciaramita, M., Schütze, H., and Rüd, S. (2014). The smaph system for query entity recognition and disambiguation. In *Proceedings of the first international workshop on Entity recognition & disambiguation*, pages 25–30. ACM.
- Dalvi, N. and Suciú, D. (2007). Efficient query evaluation on probabilistic databases. *The VLDB Journal—The International Journal on Very Large Data Bases*, 16(4):523–544.
- Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.
- Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society. Series B (methodological)*, pages 1–38.
- Dima, C. (2014). Answering natural language questions with intuit3. In *CLEF (Working Notes)*, pages 1201–1211.
- Ding, L., Finin, T., Joshi, A., Pan, R., Cost, R. S., Peng, Y., Reddivari, P., Doshi, V., and Sachs, J. (2004). Swoogle: a search and metadata engine for the semantic web. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 652–659. ACM.
- Dolby, J., Fokoue, A., Muro, M. R., Srinivas, K., and Sun, W. (2016). Extending sparql for data analytic tasks. In *International Semantic Web Conference*, pages 437–452. Springer.
- Dong, L., Wei, F., Zhou, M., and Xu, K. (2015). Question answering over freebase with multi-column convolutional neural networks. In *ACL (1)*, pages 260–269.
- Dutot, A., Guinand, F., Olivier, D., and Pigné, Y. (2007). Graphstream: A tool for bridging the gap between complex systems and dynamic graphs. In *Emergent Properties in Natural and Artificial Complex Systems. Satellite Conference within the 4th European Conference on Complex Systems (ECCS'2007)*.
- Enerdata (2016). Global Energy Statistical Yearbook. <http://yearbook.enerdata.net/>. [Online; accessed July-2016].

- Erling, O. and Mikhailov, I. (2009). Rdf support in the virtuoso dbms. In *Networked Knowledge-Networked Media*, pages 7–24. Springer.
- Etzioni, O., Cafarella, M., Downey, D., Kok, S., Popescu, A.-M., Shaked, T., Soderland, S., Weld, D. S., and Yates, A. (2004). Web-scale information extraction in knowitall:(preliminary results). In *Proceedings of the 13th international conference on World Wide Web*, pages 100–110. ACM.
- Fader, A., Zettlemoyer, L., and Etzioni, O. (2014). Open question answering over curated and extracted knowledge bases. *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '14*, pages 1156–1165.
- Fagin, R. (1974). Generalized first-order spectra and polynomial-time recognizable sets. *Complexity of computation*.
- Faradjian, A., Gehrke, J., and Bonnett, P. (2002). Gadt: A probability space adt for representing and querying the physical world. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 201–211. IEEE.
- Ferrucci, D., Brown, E., Chu-Carroll, J., Fan, J., Gondek, D., Kalyanpur, A. A., Lally, A., Murdock, J. W., Nyberg, E., Prager, J., et al. (2010). Building watson: An overview of the deepqa project. *AI magazine*, 31(3):59–79.
- Fielding, R. T. and Taylor, R. N. (2000). *Architectural styles and the design of network-based software architectures*. University of California, Irvine Doctoral dissertation.
- Fielding, R. T. and Taylor, R. N. (2002). Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150.
- Foundation, T. A. S. (2016). Apache POI - the Java API for Microsoft Documents. <https://poi.apache.org/>. [Online; accessed July-2016].
- Fritz, C. and McIlraith, S. A. (2009). Computing robust plans in continuous domains. In *ICAPS*.
- Ghahramani, Z. (2015). Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553):452.
- Gibbs, M. N. and MacKay, D. J. (2000). Variational gaussian process classifiers. *IEEE Transactions on Neural Networks*, 11(6):1458–1464.

- Google (2012). Google Now. <https://assistant.google.com/>. Accessed: 2017-08-15.
- GPy (since 2012). GPy: A gaussian process framework in python. <http://github.com/SheffieldML/GPy>.
- Green, C. C. and Raphael, B. (1968). The use of theorem-proving techniques in question-answering systems. In *Proceedings of the 1968 23rd ACM national conference*, pages 169–181. ACM.
- Green Jr, B. F., Wolf, A. K., Chomsky, C., and Laughery, K. (1961). Baseball: an automatic question-answerer. In *Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference*, pages 219–224. ACM.
- Grosse, R., Salakhutdinov, R. R., Freeman, W. T., and Tenenbaum, J. B. (2012). Exploiting compositionality to explore a large space of model structures. *arXiv preprint arXiv:1210.4856*.
- Guéret, C., Groth, P., and Schlobach, S. (2009). erdf: Live discovery for the web of data. *Billion Triple Challenge at ISWC*.
- Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107.
- Hitzler, P. and van Harmelen, F. (2010). A reasonable semantic web. *Semantic Web*, 1(1, 2):39–44.
- Höffner, K. and Lehmann, J. (2014). Towards question answering on statistical linked data. In *Proceedings of the 10th International Conference on Semantic Systems*, pages 61–64. ACM.
- Holzschuher, F. and Peinl, R. (2013). Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, pages 195–204. ACM.
- Horrocks, I., Patel-Schneider, P. F., Boley, H., Tabet, S., Grosz, B., Dean, M., et al. (2004). Swrl: A semantic web rule language combining owl and ruleml. *W3C Member submission*, 21:79.

- Hudak, P., Peyton Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzmán, M. M., Hammond, K., Hughes, J., Johnsson, T., et al. (1992). Report on the programming language haskell: a non-strict, purely functional language version 1.2. *ACM SigPlan notices*, 27(5):1–164.
- Jaro, M. A. (1989). Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. *Journal of the American Statistical Association*, 84(406):414–420.
- Jaro, M. A. (1995). Probabilistic linkage of large public health data files. *Statistics in medicine*, 14(5-7):491–498.
- Jaynes, E. T. (2003). *Probability theory: The logic of science*. Cambridge university press.
- Katz, B., Borchardt, G., and Felshin, S. (2005). Syntactic and semantic decomposition strategies for question answering from multiple resources. In *Proceedings of the AAAI 2005 Workshop on Inference for Textual Question Answering*.
- Katz, B., Felshin, S., Yuret, D., Ibrahim, A., Lin, J., Marton, G., Jerome McFarland, A., and Temelkuran, B. (2002). Omnibase: Uniform access to heterogeneous data for question answering. *Natural Language Processing and Information Systems*, pages 230–234.
- Katz, B. and Katz, B. (1997). Annotating the World Wide Web Using Natural Language. *Proceedings of the 5th RIAO Conference on Computer Assisted Information Searching on the Internet (RIAO '97)*.
- Katz, B., Yuret, D., and Felshin, S. (2001). Omnibase: A universal data source interface. In *MIT Artificial Intelligence Laboratory Abstracts*.
- Kersting, K. and De Raedt, L. (2002). Basic principles of learning bayesian logic programs. In *Institute for Computer Science, University of Freiburg*. Citeseer.
- Khaitan, S., Das, A., Gain, S., and Sampath, A. (2009). Data-driven compound splitting method for english compounds in domain names. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 207–214. ACM.

- Klein, D. and Manning, C. D. (2003). Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, pages 423–430. Association for Computational Linguistics.
- Ko, J., Si, L., and Nyberg, E. (2010). Combining evidence with a probabilistic framework for answer ranking and answer merging in question answering. *Information Processing & Management*, 46(5):541–554.
- Koutrika, G. and Ioannidis, Y. (2005). A unified user profile framework for query disambiguation and personalization. In *Proceedings of workshop on new technologies for personalized information access*, pages 44–53.
- Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.
- Krötzsch, M., Simancik, F., and Horrocks, I. (2012). A description logic primer. *arXiv preprint arXiv:1201.4089*.
- Lehmann, F. (1992). Semantic networks. *Computers & Mathematics with Applications*, 23(2-5):1–50.
- Lehmann, J., Gerber, D., Morsey, M., and Ngonga Ngomo, A.-C. (2012). Defacto-deep fact validation. *The Semantic Web-ISWC 2012*, pages 312–327.
- Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710.
- Levesque, H., Pirri, F., and Reiter, R. (1998). Foundations for the situation calculus.
- Li, X. and Roth, D. (2006). Learning question classifiers: the role of semantic information. *Natural Language Engineering*, 12(3):229–249.
- Liang, P., Jordan, M. I., and Klein, D. (2013). Learning dependency-based compositional semantics. *Computational Linguistics*, 39(2):389–446.
- Liang, S. and Bracha, G. (1998). Dynamic class loading in the java virtual machine. *Acm sigplan notices*, 33(10):36–44.
- Liu, H. and Singh, P. (2004). ConceptNet - a practical commonsense reasoning toolkit. *BT technology journal*, 22(4):211–226.



- Lloyd, J. R., Duvenaud, D. K., Grosse, R. B., Tenenbaum, J. B., and Ghahramani, Z. (2014). Automatic construction and natural-language description of nonparametric regression models. In *AAAI*, pages 1242–1250.
- Lopez, V., Fernández, M., Motta, E., and Stieler, N. (2012). PowerAqua: Supporting users in querying and exploring the Semantic Web. *Semantic Web*, 3(3):249–265.
- Manola, F., Miller, E., McBride, B., et al. (2004). Rdf primer. *W3C recommendation*, 10(1-107):6.
- Markov, K. (2017). Extending the Rich Inference Framework in the Energy Domain. 4th Year Project Report Computer Science, School of Informatics, University of Edinburgh.
- McGuinness, D. L., Van Harmelen, F., and Others (2004). OWL web ontology language overview. *W3C recommendation*, 10(10):2004.
- Meij, E., Bron, M., Hollink, L., Huurnink, B., and De Rijke, M. (2009). Learning semantic query suggestions. *The Semantic Web-ISWC 2009*, pages 424–440.
- Meng, F. and Chu, W. W. (1999). Database query formation from natural language using semantic modeling and statistical keyword meaning disambiguation. *Computer Science Department. University of California*.
- Microsoft (2014). Microsoft Cortana. <https://www.microsoft.com/en-us/windows/cortana>. Accessed: 2017-08-15.
- Mihalkova, L. and Mooney, R. (2009). Learning to disambiguate search queries from short sessions. *Machine Learning and Knowledge Discovery in Databases*, pages 111–127.
- Miller, G. A. (1995). Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41.
- Minsky, M. (1974). A framework for representing knowledge.
- Murdock, J. W., Fan, J., Lally, A., Shima, H., and Boguraev, B. (2012). Textual evidence gathering and analysis. *IBM Journal of Research and Development*, 56(3.4):8–1.
- Murphy, K. P. (2012). *Machine learning: a probabilistic perspective*. MIT press.

- Neelakantan, A., Le, Q. V., and Sutskever, I. (2015). Neural programmer: Inducing latent programs with gradient descent. *arXiv preprint arXiv:1511.04834*.
- Ngo, H. Q., Nguyen, X., Olteanu, D., and Schleich, M. (2017). In-database factorized learning.
- Nguyen, D., Demeester, T., Trieschnigg, D., and Hiemstra, D. (2012). Federated search in the wild: the combined power of over a hundred search engines. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 1874–1878. ACM.
- Nilsson, N. J. (1986). Probabilistic logic. *Artificial intelligence*, 28(1):71–87.
- Niu, F., Zhang, C., Ré, C., and Shavlik, J. W. (2012). Deepdive: Web-scale knowledge-base construction using statistical learning and inference. *VLDS*, 12:25–28.
- Nuamah, K., Bundy, A., and Lucas, C. (2016). Functional inferences over heterogeneous data. In *International Conference on Web Reasoning and Rule Systems*, pages 159–166. Springer.
- Olteanu, D. and Schleich, M. (2016). Factorized databases. *ACM SIGMOD Record*, 45(2):5–16.
- Oracle (2015). Oracle Java Documentation - Concurrency. <http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>. [Online; accessed 14-August-2017].
- Parr, T. (2013). *The definitive ANTLR 4 reference*. Pragmatic Bookshelf.
- Parsia, B. and Sirin, E. (2004). Pellet: An owl dl reasoner. In *Third international semantic web conference-poster*, volume 18, page 13.
- Pnueli, A. (1977). The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE.
- Prager, J., Chu-Carroll, J., Brown, E. W., and Czuba, K. (2008). Question answering by predictive annotation. In *Advances in Open Domain Question Answering*, pages 307–347. Springer.
- Preda, N., Kasneci, G., Suchanek, F. M., Neumann, T., Yuan, W., and Weikum, G. (2010). Active knowledge: dynamically enriching rdf knowledge bases by web services. pages 399–410.

- Prud'hommeaux, E., Seaborne, A., and Others (2008). SPARQL query language for RDF. *W3C recommendation*, 15.
- Pustejovsky, J., Castano, J. M., Ingria, R., Sauri, R., Gaizauskas, R. J., Setzer, A., Katz, G., and Radev, D. R. (2003). Timeml: Robust specification of event and temporal expressions in text. *New directions in question answering*, 3:28–34.
- Qian, Z., Goldberg, A., and Coglio, A. (2000). A formal specification of java class loading. *ACM SIGPLAN Notices*, 35(10):325–336.
- Quiñonero-Candela, J. and Rasmussen, C. E. (2005). A unifying view of sparse approximate gaussian process regression. *Journal of Machine Learning Research*, 6(Dec):1939–1959.
- Raimond, Y., Abdallah, S. A., Sandler, M. B., and Giasson, F. (2007). The music ontology. In *ISMIR*, volume 422. Vienna, Austria.
- Randell, D. A., Cui, Z., and Cohn, A. G. (1992). A spatial logic based on regions and connection. *KR*, 92:165–176.
- Rasmussen, C. E. (2006). Gaussian processes for machine learning.
- Rue, H., Martino, S., and Chopin, N. (2009). Approximate bayesian inference for latent gaussian models by using integrated nested laplace approximations. *Journal of the royal statistical society: Series b (statistical methodology)*, 71(2):319–392.
- Russell, S. J. and Norvig, P. (2010). Artificial intelligence (a modern approach).
- Salton, G. and Buckley, C. (1988). Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523.
- Sanderson, M. (1994). Word sense disambiguation and information retrieval. In *Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 142–151. Springer-Verlag New York, Inc.
- Schlobach, S., Olsthoorn, M., and Rijke, M. d. (2004). Type checking in open-domain question answering. In *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 398–402. IOS Press.
- Sedgewick, R. and Wayne, K. (2011). *Algorithms*. Addison-Wesley Professional.

- Shokouhi, M., Si, L., et al. (2011). Federated search. *Foundations and Trends® in Information Retrieval*, 5(1):1–102.
- Singhal, A. (2012). Introducing the knowledge graph: things, not strings. *Official Google Blog*, May.
- Slagle, J. R. (1965). Experiments with a deductive question-answering program. *Communications of the ACM*, 8(12):792–798.
- Suchanek, F. M., Kasneci, G., and Weikum, G. (2007). Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, pages 697–706. ACM.
- Suciu, D., Olteanu, D., Ré, C., and Koch, C. (2011). Probabilistic databases. *Synthesis Lectures on Data Management*, 3(2):1–180.
- Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.
- Tenenbaum, J. B. (1999). *A Bayesian framework for concept learning*. PhD thesis, Massachusetts Institute of Technology.
- The Apache Software Foundation (2017). Apache jena. *Available online: jena.apache.org/(accessed on 23 August 2017)*.
- Tummarello, G., Delbru, R., and Oren, E. (2007). Sindice. com: Weaving the open linked data. In *The Semantic Web*, pages 552–565. Springer.
- Tunstall-Pedoe, W. (2010). True knowledge: Open-domain question answering using structured knowledge and inference. *AI Magazine*, 31(3):80–92.
- Ukkonen, E. (1985). Algorithms for approximate string matching. *Information and control*, 64(1-3):100–118.
- Unger, C., Forascu, C., Lopez, V., Ngomo, A.-C. N., Cabrio, E., Cimiano, P., and Walter, S. (2014). Question answering over linked data (qald-4). In *Working Notes for CLEF 2014 Conference*.
- UzZaman, N., Llorens, H., and Allen, J. (2012). Evaluating temporal information understanding with temporal question answering. In *Semantic Computing (ICSC), 2012 IEEE Sixth International Conference on*, pages 79–82. IEEE.

- Van Rossum, G. et al. (2007). Python programming language. In *USENIX Annual Technical Conference*, volume 41, page 36.
- Vatant, B. and Wick, M. (2012). Geonames ontology.
- Verborgh, R., Vander Sande, M., Colpaert, P., Coppens, S., Mannens, E., and Van de Walle, R. (2014). Web-scale querying through linked data fragments. In *LDOW*.
- Walker, A. (1969). On the asymptotic behaviour of posterior distributions. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 80–88.
- Walter, S., Unger, C., Cimiano, P., and Bär, D. (2012). Evaluation of a layered approach to question answering over linked data. *The Semantic Web–ISWC 2012*, pages 362–374.
- Wang, H., Tran, T., Haase, P., Penin, T., Liu, Q., Fu, L., and Yu, Y. (2008). Search-webdb: Searching the billion triples. In *Billion Triple Challenge at the International Semantic Web Conference*.
- Widom, J. (2004). Trio: A system for integrated management of data, accuracy, and lineage. Technical report, Stanford InfoLab.
- Wienand, D. and Paulheim, H. (2014). Detecting incorrect numerical data in dbpedia. In *European Semantic Web Conference*, pages 504–518. Springer.
- Williams, C. K. and Rasmussen, C. E. (1996). Gaussian processes for regression. *Advances in neural information processing systems*, pages 514–520.
- Winkler, W. E. (1999). The state of record linkage and current research problems. In *Statistical Research Division, US Census Bureau*. Citeseer.
- Wirth, N. (1996). Extended backus-naur form (ebnf). *ISO/IEC*, 14977:2996.
- Xu, K., Reddy, S., Feng, Y., Huang, S., and Zhao, D. (2016). Question answering on freebase via relation extraction and textual evidence. *arXiv preprint arXiv:1603.00957*.
- Xu, K., Zhang, S., Feng, Y., and Zhao, D. (2014). Answering natural language questions via phrasal semantic parsing. In *Natural Language Processing and Chinese Computing*, pages 333–344. Springer.

- Yahya, M., Berberich, K., Elbassuoni, S., and Weikum, G. (2013). Robust question answering over the web of linked data. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, pages 1107–1116. ACM.
- Yellin, F. and Lindholm, T. (1996). The java virtual machine specification. *Addison-Wesley*.
- Zou, L., Huang, R., Wang, H., Yu, J. X., He, W., and Zhao, D. (2014). Natural language question answering over rdf: a graph data driven approach. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 313–324. ACM.