**UNIVERSITY OF OSLO**
**Department of Informatics**

# Question Answering using Syntactic Patterns in a Contextual Search Engine

Master thesis

Kim André Sand

**31st July 2006**

# Abstract

Question Answering (QA) systems promise to enhance both usability and accuracy when searching for knowledge. This thesis presents a prototype QA system built to leverage the extraction capabilities of a modern, context-aware search platform; Fast ESP. Questions in plain English are transformed to queries which target specific entities in the text that correspond with the identified answer types. A small set of unified patterns is demonstrated as adequate to classify a wide variety of syntactic constructs. For the purpose of verifying the answers, a semantic lexicon is compiled using an automated procedure. The whole solution is based on pattern matching and presents this as a viable alternative to deeper linguistic methods.

# Acknowledgments

I want to thank my supervisor, Knut Omang, for all help and guidance during my work with this thesis — and for introducing me to the exciting field of search engine technology in the first place.

Aleksander Øhrn and Kathrine Hammervold deserve my gratitude for their many creative suggestions and their invaluable technical assistance.

Cecilie D. Widsteen has my genuine appreciation for engaging me in long and enriching discussions, and for providing valuable feedback.

Finally, Jennifer has earned my very special thanks for meticulously reading through the whole text, and for patiently putting up with my absence when I needed to concentrate the most.

# Contents

# List of Tables

# Chapter 1

# Introduction

*Question Answering (QA)* has become a hot topic of research due to the ever increasing popularity of search engines for information access. These engines are able to quickly locate useful documents among myriads of sources too numerous for humans to administer. But using them reliably usually requires either a guesswork of keyword combinations or specialized and complicated syntax. Further, information on the document level is frequently too verbose for quick knowledge probing sessions. QA promises to alleviate these issues by allowing users to interact with the engines in their own languages and receive concise answers in return.

I present a prototype of a Question Answering system built on top of *Fast ESP* — a modern search platform. My system is able to take questions in plain English and translate them to queries optimized for the underlying engine. Rather than discarding the context and structure inherent in questions, this information is leveraged to better interpret the intent of the users. The effect is more reliable answers.

While there have been many earlier attempts at QA systems, most of these have had to operate with plain keyword searches. I'm in a privileged position because of the underlying search engine's ability to recognize useful items of knowledge, *contextual entities*, in the text. Questions can thus be mapped to queries that are more likely to retrieve answers of the right type.

My specific contributions are:

1. Generic syntactic patterns for question classification. I demonstrate how a few, unified patterns can recognize a multitude of different question types. These patterns can replace both syntactic parsing and part-of-speech tagging. They can also recognize simple semantics based on particular syntactic constructs.

2. Specifier dictionaries that are able to map answer types to general

entities recognized by the engine. While the retrieved answers may not be correct, they can at least be guaranteed to be of the right, general type.

3. Patterns that extract frequent syntactic relationships from text indicative of "kind-of" or hyponym/hypernym relations. These are utilized to build a semantic lexicon that can verify the specific facets of an answer entity in accordance with the identified question focus. The same patterns can also be used to compile new dictionaries of entities and specifiers.

## 1.1 Document structure

- *Chapter 1.* Introduces the topic of Question Answering (QA) and the goals of the thesis. Explains my background and motivation for choosing this particular field of research. Also briefly introduces some basic terminology (mainly linguistic) that is used throughout the text.

- *Chapter 2.* Presents the essential concepts and techniques of the field of Information Retrieval (IR). Explains how a basic search engine works. Demonstrates, through query examples, why these engines are unsuited to provide answers. The chapter ends with a brief presentation of Information Extraction (IE) and how it can improve the answering ability of a search engine. Explains how Fast ESP (Enterprise Search Platform) does exactly this and thereby provides a solid foundation for QA.

- *Chapter 3.* Details the promise, the challenges and the state-of-the-art of Question Answering (QA). Evaluates semi-automated approaches (Q&A) as an alternative to full automation. Further explains the considerations that have to be made when designing a QA system. Outlines TREC, the Text REtrieval Conference, which has become a crucial benchmark for worldwide QA research. Also presents specific systems and how these perform (mostly in light of TREC). The chapter ends with an explanation of the importance of semantics in QA and how this property can be assured.

- *Chapter 4.* Describes the goals and achievements of my QA implementation. Presents the general functionality of my system and the design behind it. Further introduces a selection of my entity extractors and the procedure I devised for building them. Details my question classification patterns and what they are able to capture. Explains how I

map answer types to entities and then transform questions into suitable scope queries. Finally outlines how answers can be extracted and presents my semantic verification of these by using an automatically built lexicon. The chapter ends with a brief list of future enhancements.

- *Chapter 5.* Concludes my work. Briefly summarizes my achievements. Discusses my discoveries and what I've learned.

## 1.2 Motivation

Every day, people have questions that need answers. They might need to know what something means, where they can find something (or someone), how something works, when something happens or who is involved in something. Some of these questions are trivial, while others are crucial to the proper operation of a business or even a government. Since knowledge is one of modern society's most valuable commodities, efficient acquisition of up-to-date intelligence is of the utmost importance. What, then, is the best way to find answers today?

### Looking for answers

The traditional choice of turning to a library or bookstore still applies. Books exist on nearly every topic imaginable. But acquiring a good book and reading it takes effort. This route requires a surplus of time, energy and dedication. It is thus best reserved for occasions when thoroughly researching some topic. Also, the information is only as recent as the publication date of the book. For more recent knowledge, newspapers and magazines are better. But they're not particularly well organized with regard to specific topics. Neither the freshness nor the structure of these traditional sources are their biggest problem, however, but rather their speed. Printed material is simply too slow to retrieve and consume for quick answers.

There is a clear demand for instant access to up-to-date information. Today, this demand has largely been satisfied by digitalization and networking. Many current sources of knowledge exist in digital form for more convenient consumption. Some of these, e.g. books and magazines, are often digitalized versions of their printed counterparts. Many others, e.g. discussion forums, weblogs, news feeds, e-mails, FAQs and entire sites, exist only in digital form. A great share of these new mediums have spawned as a direct consequence of the communication possibilities allowed by networked computers. The result

of having all these available sources is that people frequently turn to computers to find their answers. Specifically, they turn to the world-spanning network of machines; the Internet.

There is no shortage of digitalized information available online. It is the single, most comprehensive source of knowledge ever conceived. Indeed, generous estimates of the size of the publicly accessible World Wide Web has been as high as 11.5 billion pages [17] as of early 2005. For the most part, this estimate does not even include all the information available through the *deep web*. That is, the parts of the web that e.g. requires registration to access, uses other protocols than HTTP (e.g. FTP), is not linked to by any other source, or is simply contained in file formats or knowledge repositories not based on HTML. In 2000, a study concluded that the deep web was at least 400 to 550 times larger than the indexed web [6]. While this particular estimate may no longer hold, the general consensus is that the deep web is magnitudes larger than the commonly recognized surface web. In short, the scope of information available to the general public today is mind-numbingly vast.

### Searching for answers

Navigating through this immense jungle of facts, fiction, opinions and discussions would be painfully impractical if not for the existence of search engines. These engines allow speedy and convenient access to all their sources through uniform interfaces. They help locate documents of interest and separate the relevant from the irrelevant. Of course, not all of these sources are available through any one engine. For instance, web search engines are mostly restricted to the surface web. But many of the knowledge repositories in the deep web are likely to have local search functionality of their own. Regardless of any particular engine, searching represents a paradigm shift in information access. It has largely replaced the conventional route of actively looking for knowledge. Instead, information is presented to the user on demand and allows for the discovery of valuable treasures that would otherwise stay hidden.

For many purposes, search engines have today become the primary tool of the information seeker. But they are far from the ultimate achievement in information access. Particularly, they suffer from the following problems:

- *Too comprehensive.* While search engines do help retrieve documents of potential interest, they still can't access and present any of the answers located in those documents. Users are left to scrutinize the documents themselves. And even if it is has become easier than ever to locate

relevant information, the human mind has not become any more efficient at digesting, sorting and making sense of it. Information on the document level is simply too verbose. Due to time constraints and busy schedules, people are often far better off with quick, to-the-point facts or short summaries rather than comprehensive dissertations with myriads of details. In general, the more accurate the information, the better.

- *Too unreliable.* An abundance of information necessarily also implies an abundance of *poor* information. This is especially true on the Internet, where everyone can publish their thoughts and opinions. A large part of the 11.5 billion pages mentioned above is bound to be spam or just plain noise, e.g. arbitrary, dynamically generated pages [3]. While the quantity of information has increased by orders of magnitude, the quality has arguably dropped proportionally. Search engines do not particularly discriminate their sources nor assess the information content. Finding good, reliable information can thus take a lot of effort, and often several sources have to be consulted to establish some degree of accuracy and truth. The ability to collect answers from multiple sources and have these presented in a meaningful fashion for comfortable evaluation, would prove a considerable boon to the discerning information critic.

- *Too awkward.* With regard to access speed, searching is an improvement over looking. But it is an artificial way for humans to interact. Searching, i.e. querying, is a form of communication that takes place largely on the computer's premises. In that regard, it can even be considered a step backwards from the traditional approach of, for instance, going to a library. In the latter case, it has always been possible to ask a librarian if one does not know where to start looking or how to use the library. This is not an option when searching. Here, users are expected to understand the intricacies of successful query generation. It's far easier to just enter a couple of keywords, but that doesn't give the computer much to work with.

### Asking for answers

The challenge, then, lies in being able to automatically retrieve exact and trustworthy answers from a mountain of information. The answers are out there, but they're often buried deep. The process of uncovering them is tedious, even with the tools available today. As will be demonstrated throughout this thesis, a substantial amount of the knowledge contained within these

sources is fairly easy to recognize once located. This means a computer should be able to aid users far more efficiently than what is being utilized today.

Further, humans communicate best through their own languages. The most natural and productive route to answers is simply to ask someone. Even if users know exactly how to formulate their information needs into explicit questions, there is no readily available tool that can utilize those questions sensibly. In other words, a new paradigm shift is needed: Users should be able to ask for answers rather than have to search for them.

Consequently, some kind of automated tool that could better understand the inquiries of the users and better pinpoint the answers would be tremendously helpful. Sadly, no such tool has yet surfaced that is practically viable. The ideal system would be one that could accept a question in its most natural form. That is, formulated just as if asking another person. It would respond with meaningful answers and rank them based on an examination of all the sources of the system. These information nuggets would be exact, but also able to present their context upon request. This context is required to provide necessary support for the given statements.

Given these specifications, the overall goal of this thesis is hence twofold:

1. To explore how such a system could be created. What technology would be required? How far has research come in this direction?

2. To assemble a prototype of such a system built on already existing technology. In my case, mainly the industry-leading contextual search engine of Fast ESP.

## 1.3 Background

This thesis has been a major undertaking for me. Coming from a general computer science background, I was unfamiliar with most of the knowledge required for satisfying work in language processing. Basically, I've had to learn and comprehend the essential concepts, ideas and techniques from six different fields, none of which I had any particular knowledge in advance:

1. Linguistics

2. Computational Linguistics (CL)

3. Natural Language Processing (NLP)

4. Information Retrieval (IR)

5. Information Extraction (IE)
6. Question Answering (QA)

NLP is also heavy on statistics (probability theory in particular), an area which I unfortunately had too little time to explore well enough to utilize in this work. I've studied through the significant chapters of at least one comprehensive book on each of these subjects. Then there's all the specific papers and articles. Much of this material, namely that which I've used directly in some way, is listed in the bibliography.

Needless to say, researching background material has been a major part of the work that went into this thesis. The research was a necessary foundation for comprehending the aspects of QA, but I did not have the opportunity to utilize as much of it as I would like. Due to time and space restrictions, I've had to focus my presentation and discussion on IR and QA, both of which get a separate chapter in this thesis. IE also gets a short treatment at the end of the IR chapter. But even if it's not explicitly mentioned, all I've learned through my research has influenced the work in one way or the other.

## 1.4 Basic terminology

Here I will briefly explain essential terms that will be used throughout the thesis:

- *Anaphora.* A lexical unit referring back to a previous lexical unit, typically a pronoun. E.g. "them" referring to "cookies" in "I like cookies. Do you like them?".
- *Anaphora resolution.* A technique for identifying which lexical unit a particular anaphora refers back to. Difficult when there are multiple such units.
- *Corpus.* A collection of texts. Frequently used as a basis for language processing and testing.
- *Entity.* Any interesting item in a text worth recognizing. E.g. a person name.
- *Factoid.* An unverified "fact" which is generally accepted as true because of frequent repetition.
- *Grammar.* Collective term for all the rules governing a language, including syntax and semantics.

- *Hypernym.* A word that is more specific than a given word. E.g. "flute" versus "instrument".
- *Hyponym.* A word that is more general than a specific word. E.g. "instrument" versus "flute".
- *Language model.* A probability distribution of words in a sentence. Used to predict the next word in a sentence given the previous words. An example model is n-grams.
- *N-gram.* A sub-sequence of n items from a given sequence. E.g. both "we are" and "are happy" are bigrams of the sentence "we are happy".
- *Noun phrase (NP).* A phrase whose head (main word) is a noun. E.g. "president of the United States of America".
- *Part-of-speech (POS).* A word class such as nouns, verbs, prepositions, adjectives, etc. Every word can be classified into a POS. Some words have multiple POSes. E.g. "play" can be both a verb and a noun depending on the syntax.
- *POS tagging.* A technique for detecting the POSes of all words in a sentence and tagging them accordingly. Detection is based on both the definition and the context of the word. Accuracy is difficult because several words have multiple POSes.
- *Parsing.* A technique for building a syntactic tree (possibly annotated with semantics) representing a sentence. Computationally expensive due to the ambiguity inherent in human languages.
- *Preposition phrase (PP).* A phrase whose head (main word) is a preposition. E.g. "in the jungle of Congo".
- *Semantics.* The meaning of a word, phrase, sentence or a whole text.
- *Semantic lexicon.* A dictionary of words annotated with semantic meaning. Typically contains links between the words to establish their semantic correspondence or semantic classes.
- *Syntax.* The set of rules governing how words can be combined in a language to form sentences.
- *Verb phrase (VP).* A phrase whose head (main word) is a verb. E.g. "developed the Macintosh computer".

- *Word sense.* One of the meanings of a word. E.g. the verb "play" can mean to take part in recreation ("play a game"), to use a musical instrument ("play the piano"), to act in drama ("play a character"), etc.

- *Word sense disambiguation.* A technique for identifying the sense of a word in a sentence based on context.

# Chapter 2

# Information Retrieval

## 2.1 Introduction

These days, when looking for specific information, the normal procedure is to use a search engine. Search engines are the main tangible products of decades of research within the field of *Information Retrieval (IR)*. The main goal of this field is to present information to the user in response to a given query. In their various incarnations, search engines index billions of web pages and also act as portals to the vast amounts of text stored within libraries and other knowledge repositories.

This information is typically unstructured, meaning it hasn't been carefully collected, sorted and organized in advance. Rather, the information is usually the result of human communication exchanges. For instance, in the form of articles, reports, essays, e-mails, weblogs or even whole books. Nearly all of this material is written in *natural languages*, which is the scientific term for human languages to separate them from artificially constructed languages like programming languages. Since people communicate most conveniently through natural languages, the information is rarely tagged or otherwise prepared for machine consumption. This lack of structure means databases and other traditional forms of storage are unsuited tools of access beyond the document level (e.g. a database of manually categorized articles). The information contained within these articles is not easily accessible without some kind of functionality that can operate on plain text.

This is where search engines come in. Unlike databases they are built to work directly on text which has no structure beyond that which is inherent in natural language itself. As such, they don't require any careful preparation of the texts in advance. But computers are, by their very nature, nowhere near as proficient with human languages as humans. Search engines are thus

usually built to leverage information processing speed rather than finesse. This means the current generation of engines has some severe limitations in the language processing department, but they are nevertheless solid foundations for further advanced processing. Their limitations aside, current search engines are nonetheless invaluable tools for the information seeker. Without search tools the task of sifting through all the available information today would become unmanageable.

For answering questions, though, current search engines are quite inadequate. This chapter will explore why. First, I will give a short query example (2.1.1) in a popular search engine to demonstrate my point. Then, I will explain how these engines work (2.2, 2.3, 2.4) to shed some light on the reasons for these limitations. Finally, I will return with more detailed query examples (2.4.3) and discuss inadequacies and areas of potential improvement (2.5). The chapter also has a section in the end that presents the possibilities of Information Extraction in contrast with IR limitations (2.6).

### 2.1.1 A short query example

For the purpose of demonstrating query examples in this chapter, I will use Google [54]. This is one of the most well-known and frequently used search engines today and thus a highly likely tool of choice for someone looking for information. Further, by using this accessible engine all examples are easily verifiable by any interested party. Note that the results may differ slightly as this is a live engine, but the general type of results will be the same.

Let's say we want to know the title of the first episode of the Futurama animated series. The most intuitive formulation of our information need is in a question like *What is the title of the first Futurama episode?*. Example 2.1 shows what happens if trying to input the question directly to Google. These are the top 5 hits from the result set.

(2.1) Can't Get Enough **Futurama**: **Episode** Capsule: 3ACV20 - Godfellas
**Futurama** Capsules are meant as complete guidelines for the **episodes**.
... **Title**: Godfellas **First** aired: 03/17/02 Production Code: 3ACV20 ...
www.got**futurama**.com/Information/Capsules/3ACV20/

Can't Get Enough **Futurama**: **Episode** Capsule: 3ACV21 - Futurestock
**Futurama** Capsules are meant as complete guidelines for the **episodes**.
... **Title**: Futurestock **First** aired: 03/31/02 Production Code: 3ACV21 ...
www.got**futurama**.com/Information/Capsules/3ACV21/

"**Futurama**" (1999)
Doomsville (USA) (working **title**) Runtime: 30 min (72 **episodes**)
Country: USA ... The last **first**-run **episode** of the show aired on 10 August 2003. ...
www.imdb.com/**title**/tt0149460/

List of **Futurama episodes** - Wikipedia, the free encyclopedia
The **first** number represents the production season. ACV is FOX's series code for **Futurama**. The second number is the **episodes** number ...
en.wikipedia.org/wiki/List_of_Futurama_**episode**s

**Futurama** - Wikipedia, the free encyclopedia
The **title** of the **episode** itself is also an obvious reference. ... In the USA (DVD Region 1), the **first** season of **Futurama** was released on DVD on March 25, ...
en.wikipedia.org/wiki/Futurama_(TV_series)

These entires are not very helpful. The answer we're looking for is the simple title *Space Pilot 3000*, but it's nowhere to be seen. In fact, it's not even on the first five result pages. Instead we get all kinds of references to titles, episodes, seasons and releases. Most of these are Futurama related so at least it's a good starting point. But we're left with either starting to dig through the page links or trying to rephrase the query. Understanding the reason behind these rather poor results means understanding how search engines work, which is the topic of this chapter.

### 2.1.2 Search engine overview

All search engines are built around providing rapid access to information. This speed increase is accomplished mainly by centralized indexing, which will be explained shortly (section 2.2.2). The user interacts with the engine through a query interface. Each query is preprocessed (and possibly transformed) by the engine before being submitted for retrieval, thereby assuring better results. The queries (or rather, the keywords comprising them) are then matched against the index. Finally, documents containing matches are ranked according to expected relevancy (to the user's query) and returned to the user as an ordered list of page links. In short, we can say that information retrieval systems are composed of three different parts:

- the document subsystem (section 2.2)
- the retrieval subsystem (section 2.3)

- the user subsystem (section 2.4)

Each of these subsystems will be now be explained in turn.

## 2.2 Organizing documents

### 2.2.1 Crawling

Any search engine must contain information to be useful. This information can come from several sources, typically the Internet (i.e. web sites), intranets or databases (e.g. a library of books, or an encyclopedia). In cases where the sources are innumerable, unpredictable and dynamic — particularly on the web — the information must first be gathered somehow. This is normally accomplished by a *crawler* (a.k.a. "web crawler", "spider", "web spider" or sometimes just "robot"). The crawler follows hyperlinks and visits pages systematically, downloading selected textual content and storing it in a local repository for later use. Whether the information is crawled (e.g. the web) or readily accessible (e.g. a local database) it must be normalized to some common form. A lot of meta-data will be stripped in this process. Optionally, this additional data can be exploited to better process the text.

When constructing a crawler, several efficiency considerations have to be made. One is when to refresh the index, which means recrawling specific sites for updates. This recrawling is important to avoid stale information and erroneous links, but if the crawlers were to constantly poll the same servers for information then these servers would be overloaded and their response time severely increased. Also, by continuously refreshing, the crawlers would never be able to reach new servers, but would be polling the same servers again and again. Some balance consequently has to be reached between information freshness and workload.

A crucial observation here is that certain kinds of pages are updated more often than others, and thus should be prioritized for recrawling. Figuring out this frequency for a given site is thus extremely valuable, but by no means trivial. The task would become much easier if some update statistics from each server was readily accessible, but unfortunately no such kind of data exchange is standardized. Servers do have a special file (i.e. "robots.txt") in which they can specify which pages not to crawl. However, there is currently debate as to whether certain crawlers purposefully disregard this file.

Another consideration is how to traverse the links between pages. The two traditional approaches are breadth first and depth first. In the former case, all pages linked to by a given page are visited first, leading to a wide but shallow coverage. This approach tends to yield a huge number of rapid

requests which can quickly slow down a server. In the latter case, the first link of a page is followed as far as possible before returning recursively, resulting in a narrow but deep coverage. With the advent of algorithms analyzing page connectivity, i.e. how frequently pages link to each other, approaches started to favor traversing the "best" pages first. Others indicators of quality are the pages' general popularity and relevance to the page currently being explored. These algorithms will be discussed in section 2.3.3.

### 2.2.2 Indexing

When a sufficiently large local copy of interesting documents is available the actual process of preparing the information for rapid access can begin. If, when given a query, the search engine were to search sequentially through all the text in this huge collection, retrieval would take a very long time indeed. Instead, one or more indices are built around the words contained in all the documents in the whole collection. An index of this kind basically provides the same functionality as the index in the back of a traditional book. That is, it comprises a list of the important words in the text and directs the reader to the pages where these words occur in a significant, often defining, context.

The computerized version is often called an *inverted file* because the words now refer back to the documents containing them. An inverted file is much more comprehensive than a book index. It typically contains the complete vocabulary (e.g. the set of all words) of the entire collection of documents. Each word in the index has pointers back to every instance of its occurrence. In opposition to book indices an inverted file thus aims for quantity rather than quality (i.e. aiming for all occurrences of a word, not just the "best" ones). Quality of information is instead assured in a later step when *ranking* the hits of a given query, as will be explained later in section 2.3.3. To make it easier to rank the documents, each indexed word is further accompanied by values representing its frequencies of occurrence in every document.

The inverted files of current search engines typically also employ *full inversion*. In short, this entails that words point to the relative position of their occurrence in a document instead of simply to the document as a whole (or to any other logically structured segment). Full inversion in this manner provides a foundation for operations on relative word positions, like phrase and proximity queries, both which will be described under section 2.4.1 shortly.

An index with distinct words is not only much smaller than whole documents, but also much quicker to access. For one thing it can be kept entirely in the main memory of dedicated servers, avoiding costly disk operations. Further, incoming query keywords can be directly mapped to their indexed counterparts. All relevant documents can thus be decided in one lookup.

Finally, the index can be stored in an efficient data structure, such as a B-tree, for speedy access. However, performance degrades as the number of keywords increases. Each keyword has to be looked-up separately resulting in multiple list of documents. These lists then have to be merged and the resulting entries prioritized sensibly.

In addition to pure words an index can also contain *additional meta data*, and some of this data may be searchable through the query interface. Examples of such meta data are document titles, teasers, categorization keywords, summaries or even various page statistics. The big advantage of this kind of indexing is, of course, again the speed of access. Instead of retrieving and parsing the necessary information at query time it can simply be looked up in the index instead. Advanced engines like Fast ESP can even index named entities and phonetic information (i.e. a representation of pronunciation) opening up whole new possibilities within intelligent retrieval.

### 2.2.3 Preprocessing

#### Tokenization

*Tokenization* is the process of properly dividing the input text into token units. Most importantly by identifying the words of a sentence. This might sound like a trivial task; after all, aren't words simply strings of characters separated by whitespace? Well, consider the sentence "Mr. Adams isn't home, sweetheart.". If we match all characters except whitespace, the letters "isn't" will be picked up as one word, even though they actually form two words; "is" and "not". And by the same approach "home," will also be considered a word, including the comma, which will make it distinct from the word "home", without a comma. If, on the other hand, we only consider alphanumeric characters, "home" will be identified properly but "isn't" will now be picked up as the two words "isn" and "t", which is clearly wrong. And what about strings like "C++", "3.5%", "4th of July", "first-class" and "30-year-old"?

Deciding which characters should separate tokens is clearly not just a matter of whitespace. Words often consist of various punctuation characters and delimiters such as periods, commas, semicolons, hyphens and apostrophes. Figuring out which of these to keep isn't obvious, but nonetheless crucial. An IR system needs to be able to properly tokenize the input text since all further processing is based on this step, both on the query side and the document retrieval side. Once the tokens are identified the work can proceed on counting and comparing words, create frequency statistics and index the text. These results provide the foundation for ranking the documents,

comparing them, classifying them and mining them for information.

### Stopwords

Even though all words of a sentence are necessary bearers of information, they're not equally important in an IR sense. Some of them are simply there to form syntactically correct constructs and carry no significant meaning, like "the", "in", "of", "around" etc. The most "describing" words are thus far better at distinguishing one document from another, and consequently more valuable when comparing these. This begs the question of how the value of a word can be measured. As it so happens, the most frequent words are also usually the ones we tend to subconsciously regard as least important. This attitude corresponds nicely with the statistical point of view, wherein the most unique words are considered the most distinctive and thus valuable.

A *stopword* list is a list of words identifying the least significant ones of which we have no real use. A common way to create such a list is to index all the words in a corpus and calculate their frequency counts. The first few hundred highest ranking words will be mere "fillers" and can safely be regarded as stopwords. Amongst the first thousands there will also be many words of questionable value, but these will have to be more carefully considered as their frequencies might only be representative of the indexed information. While not useful as stopwords they might at least be considered as poor distinguishers of documents in that particular index. This information can also be exploited when processing queries, but here anti-phrasing is more likely to be used (see section 2.4.2).

Stopwords were traditionally not indexed by IR systems due to the desire to keep the indices compact. Current systems do index these words, however, because they're needed to support phrase and proximity searches. This is often called doing *full-text indexing*. In the extreme case, a query on the famous phrase "to be or not to be" would be impossible without indexing stopwords as it consists of only stopwords. Stopwords are nevertheless usually disregarded when used as stand-alone keywords in a query, unless the query is somehow interpreted as a phrase.

### Stemming

*Stemming* is the process of reducing a word to its baseform, or grammatical stem, thereby stripping away any morphological suffixes. This leads to fewer word forms being present in the vocabulary, and so a query on one form of a word will also yield matches on occurrences of its other forms. A stemmed index is also considerably more compact than a full one due to the merg-

ing of word forms. Examples of stemming are "cars" reduced to "car" and "beginning" reduced to "begin".

Popular stemmers, like the *Porter stemmer* [34], operate mainly with heuristics and not dictionaries. As such they must be trained on a corpus and tuned to perform with the right balance between too light and too heavy stemming. They will still make errors and often fail on cases like "thieves" becoming "thiev" and "amiably" becoming "amiabl". For pure searching purposes these errors aren't significant, as the user doesn't see the stemmed forms anyway. But when stemming is used for automatic labeling, for instance, the user will be presented many strange and invalid names. These failures become crucial if trying to process the stems linguistically, for example to detect their parts-of-speech (POS), as these invalid forms of the words will not be recognized. Even worse, stemming may result in a new word with a different meaning altogether. For example by altering the POS of the noun "happening" to the stemmed verb "happen". Or worse yet, by altering a word to an entirely unrelated meaning, as in "witness" reduced to "wit".

As natural language is highly irregular, stemming algorithms based on heuristics are error-prone and of limited use. They are also costly performance wise, as the suffix rules often have to be applied iteratively to reach the base. These rules naturally have to be executed at indexing time, but here runtime isn't critical. The problem is that the words from a query can't be matched against the base forms in the index unless stemming is also performed at query time. And any increase in query latency is highly undesirable.

Lastly, recent research [22] has been unable to uncover any huge benefits from applying stemming, and indeed shows it has little impact on retrieval as a whole. Its use generally yields higher recall but in return the precision is severely hurt due to its errors and generalizations. As such, the biggest advantage of stemming is perhaps the significant reduction in index size.

### Lemmatization

A more promising alternative to stemming is that of *lemmatization*. In a sense, these two approaches can be viewed as the reverse of one another. In lemmatization, instead of reducing all forms of a word to its base, the idea is to expand the base to all its inflectional forms (or "lemmas", referring to each distinct form of a word). An example is expanding the adjective "large" to "larger" and "largest". This expansion consequently yields roughly the same increase in recall as stemming, because all forms of the word now maps to each occurrence of each form. Of course, this expansion also increases the size of the index, in direct opposition to stemming.

The real gain lies with the precision. Because lemmatization uses dictionaries for its expansion, this technique is very accurate and doesn't risk corrupting words as may happen with stemming. Also, all forms of a word still exist in the index and can be distinguished when necessary. This increase in recall, while still largely maintaining precision, is an important enhancement to retrieval systems. The runtime performance during indexing is comparable to that of stemming, but as there's no longer any need to reduce query words to a common form there is no similar increase in query latency. If the increased index size is a tolerable condition, then lemmatization can be seen as preferable to stemming for the purposes of document retrieval.

If increasing the index is not desirable, lemmatization can alternatively be implemented on the query side. Selected words in the query can then be expanded to all their forms and these be included in the query as equals (e.g. by boolean OR'ing them). This, of course, increases the query processing latency, but reduces the index processing runtime similarly. As lemmatization can be deployed either on the index or the query time, it is thus more flexible than stemming which requires processing on both sides.

### Language detection

No significant linguistic processing can be performed without first detecting the language of the input text. The obvious reason is that the words themselves are different across languages, thereby demanding separate dictionaries and stopword lists for each language. But equally important is the fact that all languages have distinct rules for how words and sentences are formed. As most of the linguistic research is being performed on the English language the resulting algorithms might be completely inappropriate for other languages.

For instance, in many Asian languages, like Japanese, there are no marked boundaries between words. Tokenization can become quite a challenge when whitespace has no significance. Here, each letter (or morpheme) might be a word, but might also just be part of a word. The situation is context dependent and proper word segmentation requires processing beyond tokenization.

Further, in Germanic languages, like Norwegian, compound nouns tend to be joined together into one word. For instance "livsforsikringsselskap", meaning "life insurance company". These compounds create many variations over the same nouns that might be better to process individually, hence the need to recognize the constituents of the compounds of these languages.

Finally, languages like Finnish and Turkish have much more complex morphology than English. In morphological terms, these languages are classified as *agglutinative*, which simply means that words are formed by joining (or "gluing", hence the expression) *morphemes* (the smallest meaningful units of

a word) together. For instance, in English the verb "walked" consists of the morphemes "walk" and "ed", the last unit denoting the tense of the verb. Turkish is far more complex. Here a word can simultaneously be *inflected* (i.e. modified) for tense, person, case and number. Thus, a single word can function as a sentence by itself and may need to be decomposed to be of any use. These complex rules also make stemming and lemmatization harder.

## 2.3 Retrieving documents

### 2.3.1 Models of comparison

Search engines are valuable not only because of their retrieval speed, but also because of their ability to help us separate relevant from irrelevant information. In doing so they must be able to compare incoming queries with the stored documents to calculate their mutual correspondence of terms. Basically, the better a document corresponds with a query, the more relevant it is believed to be. Several models have been proposed for estimating this resemblance. For reasons of brevity I will mention only the two most influential ones. This section is mostly based on [4].

#### Boolean model

The *Boolean model* is perhaps the easiest model to understand and implement. Its logical foundations, being based on the mathematics of set theory and Boolean algebra, are proven and sound. The model is also naturally suited to handling queries with *Boolean operators*, a traditionally popular form of query logic (see section 2.4.1). For these reasons the model was one of the first conceived and successfully deployed.

In essence, queries using Boolean logic specify which keywords must be present in, or absent from, the documents of interest. Finding relevant documents is then simply a matter of locating documents fulfilling these requirements. Simple, but efficient. Unfortunately, this simplicity is also the major drawback of the model. Since the query terms are considered absolute, the decision criterion is purely binary; either a document is relevant or it is not. There is no degree of relevancy, no such thing as a partial match. This view leads to a model that is too narrow in practice:

- Users are unlikely to know beforehand which exact terms a relevant document will contain or not. Their queries must thus largely be considered approximations, and not as exact blueprints of target terms (which strict Boolean logic tends to enforce).

- All terms are considered equally important. This is hardly the case, as some words are definitely more frequent, and hence less describing, than others. These variations should be accounted for in the model.
- Many users have trouble translating their information needs to Boolean logic. This means that poor results will be obtained from any engine that expects explicit logic from the user.
- The model doesn't support any apparent way to rank documents. All relevant documents are treated equally, requiring the user to browse through all returned documents to find the most relevant ones.

The result is a model that corresponds nicely with logic but is largely mismatched against users' abilities and the particulars of human languages. Enhancements have been proposed, for instance by using fuzzy logic to allow degrees of truth, but today the model has largely been abandoned in favor of the following model.

### Vector model

The desire for a graded measure of relevance led to the *vector model*, which is probably the most trusted and wide-spread model in use today. Here the documents (and queries) are represented as vectors of the terms they consist of. Each term is assigned a weight according to its expected importance. These *term weights* thus not only quantify the vector coordinates, allowing the vectors to be computed upon, but they also account for the different distributions of words in the text.

Let $\vec{d}$ be the vector of term weights for the document $d$ and let $\vec{q}$ be the vector of term weights for the user query $q$. Their similarities can now be estimated by comparing their directions in t-dimensional vector space, where $t$ is the total number of terms in the index.

More precisely, their similarities can be calculated by finding the *cosine of the angle* ($\theta$) between the two vectors, as given by the following equation (2.2).

$$sim(d, q) = \frac{\vec{d} \bullet \vec{q}}{|\vec{d}| \times |\vec{q}|} \tag{2.2}$$

By computing these similarities, documents can now be ranked according to their degrees of relevance to the query. Further, documents which only partially match the query, i.e. that only correspond in some of the terms, can nonetheless be included in the result set. Their ranks might even be high, as

the cosine measure is not only determined by term correspondence, but also by term weights. The exact requirement for inclusion can be controlled by demanding that similarities measure above some set threshold. By accepting keywords as input, and figuring out their importance by document statistics instead of explicit user input, this model hence allows for much simpler usage and elegantly solves the above-mentioned problems of the Boolean model.

The specifics of how the term weights are computed greatly affects the behavior of the model. The most common way to compute the weights is by using some variant of the $tf \cdot idf$ formula (see equation (2.3).

Here, the *term frequency* ($tf$) refers to the frequency of a term within a given document and thus measures how well that term describes the document. It's balanced out by the *inverse document frequency* ($idf$), which measures how frequent the term is in the whole collection of documents. This latter factor is important because terms that are frequent across many documents are poor at distinguishing between those documents.

$$tf \cdot idf = \frac{n_i}{\sum_k n_k} \times \log \frac{D}{d_j} \tag{2.3}$$

As seen by equation 2.3, $tf$ is found by dividing the number of occurrences of a given term ($n_i$) by the number of occurrences of all terms ($\sum_k n_k$). Similarly, $idf$ is often found by dividing the total number of documents ($D$) by the number of documents containing the given term ($d_j$) and taking the logarithm of this number.

### 2.3.2 Measuring performance

To be able to measure the performance of a search engine some kind of metric must be established. In data retrieval systems the interesting properties are usually response time and storage requirements. In IR systems, however, the information retrieved is rarely exact nor complete. At best it's an approximation of what the user is likely to consider most relevant to the query. The most interesting property is thus how accurate the result set is in relation to the query. This accuracy is commonly measured as two separate components: precision and recall.

Consider a collection of documents. Given a query, let $|R|$ be the subset of all documents relevant to this query. Further, let $|A|$ be the subset of documents returned by the system in response to the query (i.e. the answer set). We want $|A|$ to include as many documents as possible from the relevant set, but as few as possible from the rest of the collection (the irrelevant set). The system doesn't actually know which documents are relevant (contained

in $|R|$), though. Therefore, let $|Ra|$ be the set of relevant documents the system did return, i.e. the intersection of $|R|$ and $|A|$.

*Recall* is now the fraction of relevant documents returned, i.e.:

$$Recall = \frac{|Ra|}{|R|} \tag{2.4}$$

*Precision*, on the other hand, is the fraction of documents returned that are relevant, i.e.:

$$Precision = \frac{|Ra|}{|A|} \tag{2.5}$$

In other words, recall measures how many documents from the relevant set we managed to return. Precision measures how many of the documents we *did* return that were relevant, as the answer set is likely to include many documents from the irrelevant set.

Because these two measures are so intertwined, many felt the desire to measure these two aspects using a single metric. The harmonic mean, or *F-measure*, was created for this purpose:

$$F = \frac{2 \times precision \times recall}{(precision + recall)} \tag{2.6}$$

In its most basic form, the balanced F-measure simply weights precision and recall evenly. For many purposes, though, one or the other might be more important, and so the general version can be tuned to suit a particular purpose:

$$F_\alpha = \frac{(1 + \alpha) \times precision \times recall}{((\alpha \times precision) + recall)} \tag{2.7}$$

Here, (equation 2.7) $\alpha$ represents the balance between precision and recall. In the harmonic F-measure above (equation 2.6), $\alpha$ is simply 1 ($F_1$). Other popular versions are $F_{0.5}$ and $F_2$ which weights precision twice as much as recall, and half as much as recall, respectively.

In addition to providing relevant results, web retrieval engines must also be able to retrieve and present documents in response to hundreds of millions of queries each day. Thus, while accuracy is commonly considered the most important property, speed must be regarded a close second.

### 2.3.3 Result ranking

The main purpose of ranking algorithms is to maximize expected relevance in the result set. That is, sort the results in order of decreasing relevance.

Due to the sheer number of web pages indexed by modern search engines, any given query is likely to yield an insurmountable number of hits. Without properly sorting this list the user would likely have to wade through hundreds, if not thousands, of documents just to find something of interest. In addition to precision and recall, ranking completes the trinity that defines how relevance is measured and maintained in IR.

Of course, subjective relevance varies from person to person and can't be established on an individual basis without some form of personalization in the search engine. But current engines do at least try to determine some objective measure of relevance based on a number of methods. These utilize statistics and frequency counts from the index to establish the value of a particular word, or a particular combination of words. But more importantly, they utilize the hypertext link structure between web pages and the corresponding anchor texts in their algorithms. These links can be considered a vast interconnected net which describes how the information flows on the Internet as a whole. Generating some statistics on these links can help identify popular information and current trends. Further, the anchor text is often a useful indicator of the content of the linked web page.

Unfortunately, it's difficult to find solid information on the exact models of ranking used in current search engines as these models are proprietary and considered a major competitive factor. But some of the algorithms have been published and are well-known.

### PageRank

Google's *PageRank* [10] is an example of an algorithm that analyzes hyperlinks between documents to help determine relevance. Google was by no means the only company providing such an algorithm, but due to relevant results, good timing and clever marketing the engine soon became one of the most popular for web information retrieval.

The theory behind PageRank is that the more frequently a page is linked, the more important it is considered to be on the whole. In essence, a link to a page is considered a "vote" for that page. The value of the vote depends directly on the PageRank of the referring page, but it's divided amongst all the outgoing links from that page.

This kind of ranking scheme has proven to correspond well with users' idea of relevance. Due to its objective measures the original PageRank was also considered fairly neutral. Today this algorithm has long since been recognized as too open for manipulation in its basic form. It's worth taking a moment to explain this practice.

Because "votes" are the main commodity in PageRank, many web masters

try to collect these to increase the rankings of their pages. This means constantly modifying their pages to correspond with the changes in the ranking algorithm. A large number of dedicated sites even exist with the sole purpose of generating or exchanging links for profit. This practice has turned into a whole industry; *Search Engine Optimization* (SEO). The methods used are many and varied. Some are approved by the search engine companies and mainly focus on building better sites. The illegitimate exploits, however, largely defeat the neutrality, ranking scheme and hence also value of the algorithms. These common attempts at manipulation must also be considered a major reason for the secrecy surrounding companies' specific ranking algorithms. Today, PageRank is only one of hundreds of factors in Google's ranking scheme.

#### ExpertRank

Ask's ExpertRank (formerly known as the Teoma algorithm) builds on the idea and popularity of PageRank. But instead of just counting the number of incoming links to a page it tries to also assess the quality of the referrers. This is accomplished by clustering sites on their topics and identifying the most authoritative sites on each given topic. The links from these sites are given more weight than regular links from unverified sources. The details of the algorithm are unpublished, but given Ask's history of reliance on human input there is likely considerable manual effort involved in this process.

## 2.4 Interfacing with the user

### 2.4.1 Querying

The user interacts with the search engine through a query interface. This interface is usually centered around an input box where the user can type in keyword queries. Some kind of custom notations are also usually accepted to give the user better control over the keywords. This kind of enhanced functionality is often available explicitly in an "advanced" interface, comprising multiple input mechanisms. These allow users to take advantage of the functionality without having to remember, or even understand, the syntax or the notations. I will now briefly describe the most common notations a modern query interface supports.

### Boolean operators

*Boolean logic* has traditionally been popular in the IR community. Most query interfaces support some form of Boolean operators, although they might not present them as such to avoid confusing the user. For instance, Google allows specifying that a keyword must be present in all documents (e.g. *+mickey mouse*), that a keyword should not be present (e.g. *-minnie mouse*) or that a word may be used in place of another (e.g. *mickey OR minnie mouse*). Other engines use the boolean operators explicitly, i.e. *AND*, *OR* and *NOT*.

The AND operator is usually implicit for all keywords, and thus only useful for special cases, e.g. for words that would normally be ignored, like stand-alone stopwords. The NOT operator is mostly useful for filtering out undesired hits, but it's often better to just add a few additional keywords to point the search in the right direction. The OR operator is largely being surpassed by query expansion techniques (see section 2.4.2). For instance, Google has an operator for expanding the query with semantically related words. For instance, *mickey mouse ~movie*, will also yield results with "film" or "DVD" in place of "movie").

Though powerful in theory, the value of explicit Boolean operators is becoming increasingly questionable as search engines mature. When also considering the potential alienation of the average user, Boolean logic may be increasingly phased out of the interfaces in the future.

### Phrase searches

Many interfaces allow specifying relative positioning when using multiple keywords in a query. The most common variant is the *phrase search*. Phrases are simply keywords enclosed in quotation marks, e.g. *"mickey mouse"*, implying that the keywords must appear in that particular order in documents. Additionally they can't be separated by anything but whitespace. This construct is handy when searching for compound nouns or if the exact wording in a target sentence is known. But it tends to be too limiting for general purposes, as natural language is highly unpredictable and words may appear in other orders and contexts than those specified. These properties make phrase searches best suited for cases of high precision and low recall.

### Proximity searches

A more relaxed variant of phrasing is the *proximity search*. Here it's enough that the keywords are "near" each other. Their order is not important and they might be separated by other words, but they have to occur together.

This kind of search is useful when knowing the words of interest but not their exact co-occurrence. Some interfaces provide this functionality through a *NEAR* keyword. Others, such as Fast ESP, also allow expressing how far apart words may be.

In Google's case, proximity searches are accomplished using a wildcard character. Querying for *"mickey * movie"* for instance finds any phrase enclosed by "mickey" and "movie", like "Mickey Rourke movie". The query *horse **** finds co-occurring words like "horse racing", "horse breeding" etc. Entering more asterisks increases the minimum number of words required as fillers. Some interfaces also allow searching for prefixes and suffixes in this manner. For instance, *for** matches words like "format", "forensics", "foreigner", and so on.

### Custom notations

Modern interfaces usually provide many *custom notations*. For instance to restrict the search to certain domains, languages, file types or dates lending the additional benefit of allowing the engine to perform specific optimizations within these parameters. Similarly, structural searches require that the keywords appear within a certain structural element, for instance in a title or as a hypertext. Additionally, certain commands or constructs often trigger special behavior. Some examples from Google are mathematical operations (e.g. *13% of 133*), definitions (e.g. *define: procrastination*) or currency and/or measurement conversions (e.g. *133 norwegian kroner in british pounds*).

### Custom query languages

Some specialized search engines go one step further and accomodate their own *custom query languages*. These are generally very powerful and perfectly tailored to suit each engine's particular functionality. For instance, the Fast Query Language (FQL) allows addressing much of the advanced functionality of the Fast ESP engine directly through the query interface. This saves implementors the trouble of writing custom code or going through configuration files to set up complex searches. These custom languages are therefore being utilized successfully by developers and implementors as high-level interfaces to specialized engines.

However, the threshold for reaching proficiency within these custom languages can be very high for the average user. As such, they have not catched on for general purpose search engines. These are instead tailored mostly towards pure natural language keywords. The idea is to try to adapt the engine to suit the user instead of the other way around. Thus, instead of requiring

the user to specify his query using an advanced syntax, current engines are increasingly trying to employ smarter techniques to interpret the query.

### 2.4.2 Query processing

Even with detailed knowledge of the retrieval environment, it's a challenge formulating queries that will produce good results. For instance, the user doesn't know which documents the engine has indexed, how it ranks them or the exact terminology used in interesting documents. Neither is it apparent what might be good keywords, or combinations of such, given the word distribution in the corpus as a whole. The user is thus likely to spend much time experimenting with and reformulating queries in order to achieve useful results.

Consequently, initial queries can largely be regarded as first guesses and far from optimal. A modern retrieval system is thus usually designed to aid the user by leveraging its direct access to the index and superior information processing capabilities. Some examples will follow shortly. The query can then be improved in one or more of the following ways:

- expanding the query (adding keywords)
- rewriting the query (replacing, removing or correcting keywords)
- reweighting the keywords in the query (reshuffling by importance)
- presenting additional queries (similar, relevant topics)

The necessary input for these enhancements can be gathered either from the users, from general word usage statistics or from the indexed documents.

#### Relevance feedback

A popular way to gather information from the user is through *relevance feedback*. The idea is that the user, when presented the results of the initial query, can provide feedback on which documents seem most relevant. The engine can then modify the query with important keywords from the selected documents. A benefit of this approach is that the actual query reformulation is automatic and completely hidden from the user. In practice, though, the average user can't be expected to tag multiple documents. A common variant of relevance feedback thus instead displays a "similar documents" link besides each returned hit. Clicking this link triggers a new search using the salient keywords of the document. The search is thus likely to retrieve information on the same topic. Still, any such explicit feedback mechanism makes the user interface more complex.

### Query log analysis

An alternative to direct feedback is to gather usage statistics from the *query log*, which contains all the queries having been submitted to the engine. By comparing the keywords of the incoming query with popular queries in the query log, new queries can be suggested that are most likely related. Of course, without any additional feedback statistics on the value of these various queries, there's no guarantee that popular queries are also good queries. But at least this is a cheap way to discover and present semantically related keywords and phrases. For instance, a search for "apple pie" could present a list of popular searches like "apple pie recipes", "baking apple pie", "apple pie filling", "apple pie crusts" and so on. These suggestions might help the user more accurately specify his intent, or even discover new, intriguing topics. Of course, the presence of undesirable noise and junk in the log will have to be handled somehow.

### Query segmentation

Users often don't bother creating phrases and instead just list up all interesting keywords in sequence. Explicit phrasing usually provides better results, though, by specifying which words are supposed to occur together. For instance in the noun phrase "account manager". Finding a match on this phrase is likely more relevant than simply matching the words "account" and "manager" separately, which can occur in many irrelevant contexts. Phrases also help disambiguate between different meanings. Take for instance "popular motion picture festival". Here the intent is to find a popular "motion picture" festival, not a "picture festival" with a "popular motion". Stating the proper relation between terms could be crucial for the accuracy of the search.

*Query segmentation* is a technique to automatically recognize meaningful phrases and segment the query accordingly. Segments are decided by consulting a dictionary of e.g. common noun phrases and proper nouns. The query log of a search engine can also be mined for frequent collocations. [38] demonstrates an algorithm for accomplishing this by validating candidate segments against frequency of co-occurrence in the log.

### Anti-phrasing

*Anti-phrasing* can be seen as an enhanced form of stopword removal that applies to queries. The intent is the same, i.e. removing words that are unlikely to benefit the query. In this case, though, whole phrases are tried filtered out. For instance, a question like "Where can I find information about

Information Retrieval?" will probably be reduced to the query "information retrieval".

Anti-phrasing can be implemented by compiling a dictionary of common, information-poor phrases. These can be automatically generated by rules that construct variations over known, poor phrases. Candidates for inclusion can also be gathered from the stopword lists or mined in the same way (see section 2.2.3).

### Spell-checking

Some of the keywords in a query are likely spelled wrong; either due to typing errors or the lack of language proficiency on the part of the user. In any case, suggesting alternatives to suspicious words is a useful, non-obtrusive feature employed by many search engines. In its basic form, *spell checking* simply entails matching each word against a dictionary. If a match can't be made, or the keyword is very similar to another frequent word, an alternative is suggested.

Word similarity is often measured by some variation of the *Levenshtein* metric, wherein the cost of modifying one word to become another denotes their similarity. E.g. "rogue" can become "rouge" by replacing "g" with "u" and vice versa, thus these two words differ only by a cost of 2. Changing "horse" to "towel" on the other hand requires 4 operations (i.e. replacing all characters except "o"). These modification operations, i.e. insertions, deletions and replacements, can be weighted differently and the algorithm tuned to favor fixing typical writing mistakes (e.g. making the required operations cost less).

### Synonyms and thesauri

Stemming and lemmatization have already been described in section 2.2.3 as techniques that can be used to map a query word to all forms of its morphology. This has the benefit of improving recall, but at the cost of reducing precision. Recall can be improved even further by also considering synonyms and related words. The drawback is that synonyms tend to have slightly different meanings (e.g. "to speak" vs "to state"). Ambiguity can further result in two unrelated words having a synonym in common (e.g. "state" as in "government" vs "state" as in "condition"). This creates an undesirable link between two different concepts. Sticking to the correct word sense and conceptual meaning is thus vital in maintaining precision. The utilization of a proper semantic dictionary such as WordNet [62] can prove helpful in this effort.

### 2.4.3 Query examples

Let's try a few typical questions in Google to see how it responds and to unravel its abilities as an answer-finding tool.

#### Example 1

Let's return to the question from the introductory example. It's time to explain the rather poor results from example 2.1. First of all, due to filtering mechanisms like stopword removal (2.2.3) and anti-phrasing (2.4.2) the question is basically reduced to a query like *title first futurama episode*. Indeed, the results from this query (example 2.8) differ little from the results from the initial question.

While both of these queries seem very natural to us, Google doesn't know that the object in question is actually an episode title. Nor would it know how to identify one in the text. And it definitely doesn't know that we're looking for the *first* of these. Instead, it just finds pages containing all keywords in near proximity of each other. In many cases this is enough; such pages are likely to contain sentences mentioning the desired target and will hopefully even include one such sentence in the teaser. In this case we're less fortunate.

(2.8) "**Futurama**" (1999)
Doomsville (USA) (working **title**) Runtime: 30 min (72 **episodes**) ... I'll admit that I too was among the people crying foul when **Futurama first** came out, ...
imdb.com/**title**/tt0149460/

Can't Get Enough **Futurama**: **Episode** Capsule: 3ACV09 - The Cyber ... **Futurama** Capsules are meant as complete guidelines for the **episodes**. These documents are produced by ... **Title**: The Cyber House Rules **First** aired: 04/01/01 ...
www.got**futurama**.com/Information/Capsules/3ACV09/

Can't Get Enough **Futurama**: **Episode** Capsule: 4ACV13 - Bend Her **Futurama** Capsules are meant as complete guidelines for the **episodes**. ... **Title**: Bend Her **First** aired: 07/20/03 Production Code: 4ACV13 Written by Mike Rowe ...
www.got**futurama**.com/Information/Capsules/4ACV13/

List of **Futurama episodes** - Wikipedia, the free encyclopedia
The **first** number represents the production season. ACV is FOX's series code for **Futurama**. The second number is the **episodes** number ...

en.wikipedia.org/wiki/List_of_Futurama_**episode**s

**Futurama** - Wikipedia, the free encyclopedia
The **title** of the **episode** itself is also an obvious reference. ... In the USA (DVD Region 1), the **first** season of **Futurama** was released on DVD on March 25, ...
en.wikipedia.org/wiki/Futurama_(TV_series)

The title of the first entry is indeed a movie title, but it refers to the Futurama series as a whole, rather than to a specific episode. The reason this entry comes first is largely because the link to the page contains the word "title". This explicit mention is a strong indication that the page discusses the subject indicated by the keyword. A clever ranking scheme knows to exploit this. Especially since this keyword is the first in the query, thus regarded as the most important one. The high rank is also due to the title containing the word "Futurama". The teaser further contains all the words from the query, but in a quite different context than intended. There is no mention of a first episode at all.

The next two entries do contain titles in the teaser, which are gathered from tables and explicitly marked as such on the form "Title: Bend Her". Because the search engine has no notion of a title, this explicit mention of the word "title" is again the most likely way that a title will indeed be presented. It won't be marked as such though, although the reader will have little trouble identifying it. Again, there is no mention of a first episode, but rather a first airing.

The fourth entry refers to a list of Futurama episodes. Neither the title nor the teaser names any specific episode, but the corresponding web page is very likely to contain the desired episode title. A quick check confirms that the title in question, "Space Pilot 3000", is indeed mentioned in the top of the list of that page. Finding this information requires opening and browsing the page though. There's no sign of a first episode in the teaser here either, but instead it mentions a first number.

The fifth entry is completely off mark and doesn't contain any interesting information at all. It does mention a first season though, which is coincidentally the closest match to a first episode yet. But it's quite apparent from these results that the engine has no notion of the kind of information we're after but rather tries its best to match our keywords.

This example demonstrates that in many cases, if we want the answer, we have to open the page and look for it ourselves. This is as expected though. Google, like search engines in general, was never intended to present exact results. The teaser is only there to give an indication of whether the page is

relevant, and must not be confused with an explicit answering mechanism.

### Example 2

Let's try shuffling the query words a bit to see if this helps retrieval. As seen above, the engine clearly missed the connection between the words "first" and "episode". Thus we might try juxtapositioning them to make our purpose more clear. The query *title futurama first episode* presents us a result set where the desired title is mentioned in the teaser of one of the top 5 hits (example 2.9).

(2.9) **Futurama** - Wikipedia, the free encyclopedia
In the USA (DVD Region 1), the **first** season of Futurama was released on DVD on March 25, ... **Title** sequence extracted from the Space Pilot 3000 **episode**. ...
en.wikipedia.org/wiki/**Futurama**_(animated_series)

Once again this seems to be coincidental as there is no explicit mention of a first episode, nor any mark-up of the target. Instead the title appears in a string referring to an audio extract from an episode which happens to be the first aired. The specific string is "Title sequence extracted from the Space Pilot 3000 episode". The rest of the top 5 hits have nothing to do with titles.

### Example 3

A further refinement entails being even more explicit and providing our own phrasing, as in *title futurama "first episode"* (example 2.10).

(2.10) Can't Get Enough **Futurama**: Episode Capsule: 4ACV12 - The Sting
A huge site dedicated to the show **Futurama**. Includes **Futurama** downloads, **Futurama** ... LT; in the **first episode** space pilot 3000, when professer fransworth ...
www.got**futurama**.com/Information/Capsules/4ACV12/

Again the desired title is mentioned in the top 5, but this time finally in the right context, namely that of a first episode. When the connection between these two words is explicitly defined, a hit is finally returned where all the pieces of the query fall together.

**Example 4**

In the above examples, the word "title" was entered as part of the query. Even though the engine didn't know what a title looks like, this nevertheless increased our likelihood of actually matching a title. Let's try a query where the answer type is not explicit, namely *Who dies in the "Half Blood Prince"?* (example 2.11).

(2.11) Harry Potter and the **Half Blood Prince**

Harry Potter And the **Half-Blood Prince**... Which Characters **Died** In Book 6? In the land of magic, death is a relative phenomenon. While there are characters ...
www.chiff.com/a/harry-potter-hbp-deaths.htm

Harry Potter and the **Half-Blood Prince**
Pottermania has arrived. Fans are finding out the answers to...WHO **DIES**? WHO IS THE **HALF-BLOOD PRINCE**? But now want to know, WHO IS RAB and rumors on book 7.
www.chiff.com/a/harry-potter-hbp.htm

Harry Potter and the **Half Blood Prince** Rumors for Book 6
Who **dies** in the **Half Blood Prince**? Fred? George? New: Harry Potter Book 7 Rumors, Theories and Facts Index New Quiz: Harry Potter and the **Half Blood Prince** ...
parentingteens.about.com/od/ harrypotterrumors/a/harry_rumor6104.htm

Harry Potter Book 6 Rumors and Facts Index for the **Half Blood Prince**
Who **Dies** in Harry Potter and the **Half Blood Prince**? ... So, who **dies**? Maybe the **Half Blood Prince**? Email me and tell me who you think dies in Harry Potter ...
parentingteens.about.com/ od/harrypotter/a/harryrumor_6i.htm

Blogcritics.org: Harry Potter and the **Half Blood Prince** Review
Snape is the **half blood prince**. Dumbledore **dies** and its Snape who kills him ... If someone had told you who the **half blood prince** was or who **dies**, ...
blogcritics.org/archives/2005/07/17/011055.php

Again, this question essentially yields the same results as the query *dies "Half Blood Prince"*. But this time it's more interesting to examine the results from the *question*, because the word "who" turns out to be detrimental to the query.

As can be seen, many hits just repeat the question blindly. The presence of the word "who" doesn't aid the engine towards a person but rather distracts it towards matching actual questions. This is a major problem with entering questions directly in keyword-based query engines. While the presence of a question might indeed indicate the presence of an answer, that answer will likely be hidden somewhere in the resulting page. As we're interested in the answer, not the question, this behavior is not very useful.

Also note that the engine focuses on the verb "die", and does not try to expand upon this verb by using synonyms or a related noun such as "death". This means potentially missing useful constructs using these synonyms. On the other hand, this restriction is a cheap way to assure high precision which is more critical than recall given the amount of data web search engines normally have access to.

The teaser of the fifth and last result correctly presents the information that Dumbledore is the one that dies in the book. Given the previous blank shots, this relation again seems coincidental rather than intentional.

If Google could at least be made to understand that we're looking for the name of a person we would probably get far more relevant results. Unfortunately this level of sophistication is beyond the current crop of search engines. As can be seen, "who" is just treated like any other keyword. The verb "dies" is not recognized as having to do anything with a person either. And adding the noun "person" won't make the engine any wiser. As it is, if we're lucky the name is revealed as part of the teaser, but if not we're again left to browse through the returned documents in search of it ourselves.

Even if the engine were to recognize person names, these results demonstrate a potential trap. Harry Potter, being the star of the series, is mentioned frequently in relation with the Half Blood Prince. If the system were to base its answer primarily on frequency of occurrence, Harry Potter is almost guaranteed to be picked as the answer.

## 2.5 Discussion

Search engines work with queries, not questions. When supplied questions, they will simply reduce them to queries through various filtering steps. This process ignores the context that provides clues as to what the questions are all about. Instead of interpreting this context, the questions are simply converted to vectors of ambiguous keywords. These vectors are then matched, word by word, against potentially useful documents. Behavior of this kind is obviously detrimental if the system is to be able to handle questions. Such filtering techniques may thus have to be disabled if the engine is to be

enhanced to handle questions properly.

By completely ignoring the meaning of the words and how they relate to each other, the ability to identify useful knowledge is sacrificed for speed and language independence. Passages containing answers are only likely to be retrieved if these use the same set of words as the questions. Further, there is no way to specify the type of answer that needs to be located. Even if it were, the engine wouldn't know how to identify it. Unless the answer type (e.g. "title") is mentioned explicitly somewhere, the engine won't know that it's dealing with the right type. This kind of ignorance is bound to miss a lot of perfectly good answers. Also, it will likely return many seemingly good matches that are totally irrelevant from a semantic perspective. In short, the engine doesn't really know what to look for.

But while current search engines may not be very good at answering questions, at least experiments conducted by e.g. Radev et al. [36] demonstrates that they show great potential as components in a QA system. When feeding simple fact-based questions to nine popular web search engines, 75% of the time they all managed to return at least one document, among the top 40, that contained the correct answer. Identifying and extracting that answer is, of course, not trivial. But at least the amount of documents that have to be scanned more thoroughly are reduced to a manageable fraction.

In general, the trend seems to be moving from powerful query languages to smarter engines. This shift has been made possible by steady advances in a number of fields dealing with natural language processing. These advances have again become viable due to cheaper and faster computer hardware. The result is that the focus is increasingly moving from easing the task of the computer to easing the task of the user. Instead of the user having to struggle with complex query languages to make the computer understand the object of desire, the computer is increasingly being taught to identify what the user wants to know. Proper question answering technology is a natural extension to this line of thought. It allows the use of a language that is both powerful and easy to use at the same time, with the ability to be very specific when needed: the human language.

In conclusion, while current Information Retrieval systems are very good at telling users where to look, they can't provide them with any answers. In this aspect they resemble efficient librarians more than knowledgeable experts. Users still have to manually read through the returned documents until they hopefully find some solid answers. This process can be very time consuming and tiring. While IR is inarguably a crucial step in the right direction, it's not enough by itself.

## 2.6 Information Extraction

IR was never intended to provide answers. The original goal was simply to aid users in retrieving the documents that best match their queries. With full-text inversion came the ability to show users where within these texts their query keywords are being mentioned. This ability is rarely utilized to help users navigate the documents, though, but rather to generate teasers. If users are lucky, these short snippets of information contain the answer they look for. But teasers are really only intended to help users determine whether a given document is relevant. This is far from providing answers, and merely a gradual narrowing down towards the part of a document that is most likely to be interesting.

Focusing on the words that constitute a document is sufficient for retrieval, but not for making sense of the text. A related field of research instead tries to aid users in more easily getting access to the actual knowledge contained within a given text. This field is called *Information Extraction (IE)*. While IR concentrates on finding interesting passages of text, IE focuses on the meaning contained within these boundaries. IE is motivated by the desire to collect hard facts from natural language texts and present these to the users. More specifically by discovering *entities*, i.e. items of interest such as persons, companies or locations, and how these relate. For instance, by creating an overview of which persons are employed by companies in which locations. This kind of automation is very beneficial as it saves the users from having to read through all the information themselves. This repetitive process would be immensely time consuming and tiring for a human, but is perfect for a computer. Of course, it all comes down to how adept the tool is at interpreting the text. But even if lacking in the language department, it will nevertheless be useful in providing a rough overview quickly.

IE can even be used to supplement existing tools. Because there are already many other tools available for organizing, mining and discovering important information in large sets of data. For instance, advanced databases, data warehouses, topic maps and semantic networks. But the efficient operation of these tools relies on vast amounts of information already having been collected and given structure somehow. IE is a promising technology for providing this kind of structure automatically by recognizing the very structure inherent in natural language.

### 2.6.1 Pattern matching

IE typically works by scanning through the text looking for patterns that match certain predefined sequences of characters, digits or symbols known to

represent an instance of a given entity. For instance, "22:35" can be recognized as specifying a certain time of day, given the valid hour and minute components separated by a colon. Similarly, "John Adams" can be recognized as a typical person name given a suitable dictionary containing such names. Recognizing these kinds of entities is called Entity Extraction (EE) and is a critical component of IE. Modern regular expressions are often used for this purpose (section 2.6.3).

Recognizing entities is useful in finding instances of a given type, but it's not enough by itself. Let's say, for example, that a user is trying to gather a list of employees at a certain company. Using EE it's possible to find both persons and job titles co-occurring with said company within some specified boundary of text. However, this proximity in no way implies that the entities have anything to do with each other. Specifically, it does not imply that the mentioned person has the mentioned job title and works in the mentioned company. For instance, the person in the sentence *John Adams said that the CEO of XYZZYSoft has no real strategy* is clearly not the CEO of the company. On the other hand, *John Adams was recently appointed CEO of XYZZYSoft* contains highly interesting information. For knowledge gathering purposes it is thus not enough to recognize entities by themselves, but they must also appear in valid syntactic relations.

These relations can be recognized by utilizing the same form of pattern matching as on the entities. In this case, a pattern such as "*PERSON was ... appointed JOBTITLE of COMPANY*". Here the uppercase words represent entities and the dots represent a sequence of arbitrary words that don't invalidate the relation, like e.g. a negation ("not appointed") would. These kinds of patterns are often called *templates*, since they represent predefined templates with empty slots that need to be filled in with actual instances to be complete. These slots can be filled by any entity of the right type. Once all slots are populated, the template is said to constitute an extracted fact or piece of knowledge.

Of course, it would be a lot of work creating patterns manually for all possible valid relations. Instead, IE systems typically employ machine learning components. Riloff [37] describes one such system. AutoSlog-TS generates templates (called "case frames" in this system) automatically by learning from examples. Training is performed by running syntactic analysis on two sets of texts, relevant and irrelevant. This analysis is performed automatically and allows the system to discover valid patterns. The templates are then ranked (basically by dividing relevant matches by total matches) and submitted for manual review. While mostly automatic, the process still requires manual effort in identifying relevant examples and discarding defunct templates. Machine learning components are thus mostly useful for present-

ing possible suggestions, but these will still have to be quality assured by a manual process.

Entities and facts extracted in this manner constitute solid information that can be utilized either directly (for knowledge gathering or text mining) or used to populate a structured knowledge base (e.g. a database system) for later data mining. Even more interesting, however, are the possibilities that open up when enhancing a modern search engine with an IE component. Specifically, the potential for extracting answers in response to queries.

### 2.6.2 Fast ESP

*Fast ESP (Enterprise Search Platform)* [53] is a modern search engine enhanced with IE capabilities. These enhancements are collectively referred to as *Contextual Insight* [18], because they allow users to explore all kinds of contextual information relating to their queries. I will not go into details on how this specific engine functions, as I've already covered the most important aspects of proper IR operation. Nor will I explain here how entity extraction is integrated with the system. This integration will instead be presented when discussing my implemented extractors in section 4.3.

However, it's useful explaining briefly how IE can be utilized for QA purposes by using this specific engine. The promise lies with how the entities are processed. While Fast ESP is able to identify dozens of entities of various kinds, the real benefits come from being able to index these entities as meta-information. Mining and indexing entities means that they can be queried upon just like a keyword. But unlike a keyword, which is basically a meaningless sequence of characters, an entity contains meaning. That is, once an entity is matched, it represents an instance of the semantic type of that specific entity, e.g. a person. It's thus possible to specify a search for a person and only get presented results that contain this kind of entity. Further, the results are not required to mention the word "person", or any equivalent term, as the person names will be recognized by their implicit semantics, not by the explicit mention of type.

Searching for entities is possible in Fast ESP by using a scope search. *Scopes* are predefined portions of text in a document, typically a paragraph, a sentence or an XML tag (e.g. a body-tag in a HTML page). But a scope can also be an entity. It's thus possible to query for a certain instance of an entity, e.g. *person(John Adams)*, or simply require that a person is present in the result, e.g. *scope(person)*. These constructs are components of the *Fast Query Language (FQL)*, and can only be addressed through this custom language. This means FQL queries must be written to utilize scopes, which adds an undesirable element of complexity compared to simple keyword

searches. But more importantly, the engine will not be able to provide an answer to a question unless the user translates the question to FQL using a scope search for the intended answer entity type.

As it is, questions can't be utilized directly in Fast ESP. That is, they can be submitted but will just be handled as queries. The engine contains the basic requirements for returning answers of the right type, but only if an explicit search for that type is deployed. This limits the general usability of the engine for QA purposes. To be able to provide answers, the engine would need to automatically determine the focus of the question and translate this to a proper scope search. If this kind of functionality was realized, it could greatly enhance the value of search engines for the average user. This is the big promise of using Fast ESP for QA and it is the primary target for my implementation.

### 2.6.3 Modern regular expressions

Since pattern matching is frequently based on *regular expressions (regexes)*, it's useful to briefly mention these here. It's important to note that regexes are no longer what they used to be. The definition of a *regular language* is one that can be processed by a finite state machine, also known as an automaton. Modern regexes go quite far beyond this definition, and can even surpass context-free languages. The definition of a regex today thus depends on whether speaking of formal language theory or pattern matching. I use the term in the latter sense.

The theories of automatons and formal languages had been conceived long before computer systems existed that could make any practical use of them. The well-known mathematician Stephen Kleene refined these theories into the formal notation of regular sets. This notation was then popularized in the 1960s by Kenneth Thompson's work on the *QED* and *ed* editors for Unix. As such, regexes have been around for decades and is a mature technology with a solid mathematical foundation. Today, the regex is the prime mechanism behind pattern matching techniques and plays a significant role in most successful information extraction systems.

Traditional regular languages provide limited forms of expression. The resulting patterns are usually combinations of alternations, groupings and quantifications. Modern regex libraries like *PCRE* [59] *(Perl Compatible Regular Expressions)*, go beyond these conventional boundaries. In doing so they support clever constructs that greatly increase their area of applicability. Basically, a modern regex is like a specialized programming language operating on text. It even allows named back references ("variables") and conditionals. These added features make modern regexes very powerful and

versatile. Today they're usable for all kinds of matching and extraction tasks. But the added functionality also means the expressions are strictly not regular anymore. They're still commonly referred to as such, though, for lack of a better term.

Using modern regexes it's thus possible to write quite sophisticated extractors without having to resort to complex and slow syntactic parsers (for natural languages). The results from this "simple" pattern matching can then be used for more advanced processing later on if needed. A typical use is to run "shallow" matching on large amounts of text to pinpoint the interesting areas and then process these areas more thoroughly by using "deeper" techniques. This enhanced functionality comes, of course, at the cost of processing efficiency. By breaking out of traditional boundaries, modern regexes demand complexity beyond that which can be provided by finite state machines alone. Even so, they display both an ease-of-use and an execution speed that is still far ahead of syntactic parsers.

Regular expression support in library form (like in PCRE) is ideal for embedding within a framework tailored to entity extraction. A remarkable proof of this is the Matcher framework of Fast ESP which will be detailed in 4.3.1, followed by examples of effective patterns and the procedure used for conceiving them.

# Chapter 3

# Question Answering

## 3.1 Introduction

Looking for answers by searching for and reading through relevant information is a valid course of action. Both IR and IE provide helpful tools in this effort, and both fields have made much progress. But the simplest and most intuitive route to answers has been largely avoided; that of asking. Of course, success traditionally depends on having someone knowledgeable available for questioning. And the more difficult and important the question, the harder it is to find that someone.

But what if a system could be created that automatically responded to questions? IR already fills the role of an efficient librarian. IE can further assist in pin-pointing the answer. What is needed now is the tool that ties these two fields together: The knowledgeable expert that can directly answer questions — someone who's available all the time and responds immediately and tirelessly. Researching the foundations for such a system has formed a new field of science; *Question Answering (QA)*.

QA has sprung up in recent years as a technology with the potential to provide easier and more intuitive access to information. The goal is to allow users to pose questions in their own natural language (e.g. *During which season do most thunderstorms occur?*) and receive concise, to-the-point answers in return (e.g. *Spring*). Allowing such questions as input could save users lots of time that would otherwise be spent trying to figure out the right combination of search keywords that yields a good result set. And often, further effort must be expended on manually examining the documents in this set.

Currently, there is a large *language mismatch* between users and engines. For successful results, the users have to carefully rethink and reformulate

their questions into queries that suit the engine's retrieval model. Alternatively, if inputting questions directly, the engine will simply strip the questions of useful contextual information. Or even worse, misinterpret some of these structural words as important matches in the resulting documents. Major benefits of a QA system would thus be the ability to accept questions directly, thereby avoiding this error-prone, and often complex, translation step. Accepting questions also helps users focus on their exact information needs, and aids them in expressing these needs more precisely to the system. The system, in return, can deduce more easily what the user wants to know. Intuitively, these factors can help improve both user satisfaction and retrieval accuracy.

This chapter will explore QA in detail, both from an academic and a commercial perspective. I open with some short background on the evolution of the field (3.1.1) and go on to introduce the semi-automated approach (3.2). Then I explain important considerations when designing a QA system (3.3) and how performance can be evaluated (3.4). The latter section also has detailed information on QA evolution in light of the TREC conference. Finally, I present some actual systems (3.5) and some thoughts on semantic answer verification (3.6). I end with a discussion on the limitations and possibilities of QA (3.7).

### 3.1.1 Background

QA has existed in some form or the other since the 1960s. The first incarnations were used as natural language interfaces to *expert systems*. These are rule-based systems able to automatically infer solutions to problems within their domain. At their finest, they're able to produce solutions equal to, or better than, human experts. This power would largely be wasted unless users could comfortably state their problems, hence the interface enhancement.

QA is now increasingly attracting attention from the related fields of IR and IE. The same promise of allowing users to interact with existing systems in a more natural way is again one of the main attractions. In exchange, QA has been made more viable due to the results achieved within these fields. The benefits of merging are thus mutual.

QA can be seen both as an extension to these fields, and as a kind of intersection between them. In QA and IE, unlike in IR, retrieving a whole document is unacceptable. And in QA and IR, unlike in IE, no restrictions are placed on the type and domain of questions. QA can thus combine the openness of IR with the exactness of IE, and could prove to be a useful bridge between them.

While QA has only fairly recently started to receive full attention, part of

the required functionality has long existed within other fields. Database (DB) systems, for instance, already supported returning exact answers, but also demanded strict structuring of both the data (e.g. tables and relations) and the query (e.g. SQL). Given a properly structured (and populated) knowledge database, however, finding an answer is only a matter of formulating the right query. For instance, a question such as *The Hindenburg disaster took place in 1937 in which New Jersey town?* could be formulated as seen in example 3.1.

(3.1) **Question:**
*SELECT location FROM disasters WHERE name = "Hindenburg" AND year = "1937" AND location IN (SELECT * FROM towns WHERE state = "New Jersey")*

**Answer:**
*Lakehurst*

These severe structure requirements, and the high complexity of the query, don't exactly form an ideal base for QA. When IR systems started to mature they soon abolished these restrictions on both data and queries. By allowing unstructured text and keywords as source and input, respectively, a far better foundation for QA had been achieved. The same question example, now targeted at an IR engine, can be seen in example 3.2.

(3.2) **Question:**
*Hindenburg disaster "New Jersey" 1937*

**Answer:**
*Hindenburg Disaster One such event happened to reporter Herb Morrison on May 6th, 1937 in Lakehurst, New Jersey. The mighty German passenger Zeppelin, ...*

Focusing on efficient retrieval meant losing the property of exactness, though. These systems could consequently only return documents, or at best passages of text, which hopefully contained the needed information. But the ability to find exact pieces of information soon resurfaced within the field of IE.

Advances in IE soon made it possible to locate exact entities in the same kind of unstructured text as that provided by IR. By enhancing IR with IE, proper question answering has moved one step closer. Few search engines

actually employ IE directly, though, and instead leave this kind of processing to some other tool. Fast ESP is a notable exception. Here, these two fields have been successfully combined into a powerful symbiosis. But most QA systems have to rely on a plain IR engine and integrate their own IE component as part of the answer extraction process.

A big obstacle is how to access these entities unintrusively through an interface similar to the simple keywords-based query box. Currently, a search for entities can only be specified accurately through structured queries in some custom language. For instance, example 3.3 demonstrates how the question could look in the Fast Query Language (FQL). Due to the merging of IR and IE, the answer can be provided with or without contextual information. This results in a long or a short answer; a passage or an entity.

(3.3) **Question:**
*and("Hindenburg", "disaster", "New Jersey", scope(location), date(1937))*

**Short answer:**
*Lakehurst*

**Long answer:**
*The research examined the disaster of the airship Hindenburg, which occurred at Lakehurst, New Jersey, on May 6, 1937.*

As concluded in the IR chapter (section 2.5), pure keyword-based queries often contain too little information for the engine to be able to recognize the user's intended target of inquiry. On the other hand, custom query languages, as just demonstrated with FQL, are just marginally more user friendly than traditional structured languages like SQL. Current search engines can thus greatly benefit from a query language with the simplicity of IR queries and the power of IE queries.

QA thus provides the last piece of the puzzle by allowing users to place inquiries in their own, unmodified, natural language. By interpreting and translating the user's question to the structured form accepted by the engine, both power of expression and preciseness of intent are maintained. An engine that could properly support a question answering paradigm would thus prove most beneficial to the information seeker. Just to complete the example, 3.4 shows the simplicity allowed by QA.

(3.4) **Question:**
*The Hindenburg disaster took place in 1937 in which New Jersey town?*

**Answer:**
*Lakehurst*

This evolution of technology from databases to QA is summarized in table 3.1.

| Field | Answer | Source | Query |
|---|---|---|---|
| DB | exact | data | structured |
| IR | inexact | text | keywords |
| IR w/IE | exact | text | structured |
| QA | exact | text | question |

Table 3.1: Evolution towards question answering

## 3.2 Q&A: The semi-automated approach

Question answering is essentially an *AI-complete* problem, meaning that it can only be solved in its entirety by creating an "intelligent" agent able to fully comprehend human language and understanding of the world. Creating such an agent has proven mind-numbingly complex and the current state-of-the-art is nowhere near this level of ability. Some companies have thus taken the question answering paradigm quite literally and base their solutions on answers provided by humans. These solutions can either be fully manual or semi-automated.

While not strictly QA systems, these are nonetheless worth mentioning briefly as they provide a realistic alternative to automation. After all, the ability of humans to successfully answer questions should not be underestimated. The legitimacy of automated systems should thus be reviewed in this light. There is not much sense in creating an automated system if a manual one performs adequately.

Still, the QA community seems largely to have avoided this aspect. I've been unable to find solid research on the topic, and the implementations I've seen have all been commercially motivated. I see this as all the more reason to describe how these solutions work. To distinguish them from QA systems, however, they will be referred to as *Q&A services* (Questions & Answers).

There are several variants of these solutions, but they all share the property that humans provide the answers to questions submitted by the users. They differ mostly in the following aspects:

- *Dynamic or static.* A dynamic solution is one where humans answer new questions continuously as they're submitted to the system. A useful knowledge base of Q&A is thus gradually compiled over time as new answers are entered. A static solution, in contrast, is one where the knowledge base consists of precompiled collections of Q&A pairs, such as FAQs. Of course, a collection of FAQs might be updated regularly but will still be static compared to the frequent flow of answers in a dynamic solution.

- *Answer source.* Dynamic solutions mainly have two possibilities. First, the answers could come from information experts which are proficient at locating useful information quickly. These are likely to a number of information gathering tools from the fields of IR and IE at their disposal. Secondly, answers could be exchanged between users which are simply interested in the same topics. Finally, the source of static solutions is the source that generated the Q&A pairs in the first place.

- *Cost of service.* Information experts normally demand a fee for their work while users are likely to exchange answers free of charge. The quality of the answers is naturally tied to this aspect. A FAQ collection or similar is likely freely available, but might be private and charge a fee for access.

- *Openness.* Answers might be provided exclusively to the person asking the question or submitted for public use. The former case primarily makes sense in a commercial setting. If answers are exclusive they can be provided upon charge on an individual basis. This hinders the gradual compilation of an accessible knowledge base for general use. Or rather, it means this knowledge will be kept private by the providers of the service.

So far a system with the properties described can be fully manual in nature, except for a thin layer of functionality driving the service. Automation comes into the picture for those systems that try to match new questions against known Q&A pairs. The purpose is, of course, to increase efficiency by automatically responding to already answered questions. This kind of solution is automated in the sense of retrieving known answers to known questions. But it's manual in the sense of actually locating the answers in the first place. It can thus be called a *semi-automated* system.

The easiest way to accomplish this is simply to make the knowledge base searchable by a standard keyword based search engine. The keywords from a question can then simply be matched against the answers. Basing retrieval on

keyword matching against answers does, of course, bring the inadequacies of these engines as questions answerers to the surface. Just because a question shares terms with an answer doesn't mean that they form a good pair.

There is another possibility, however. Instead of matching questions against known answers they can be matched against known questions. After all, the best way to assure a correct answer is by assuring that the question actually asks for that answer. But matching against questions is challenging in two important ways:

1. Questions are far shorter than answers and thus contain far fewer words. This gives keyword searches very little to work with.

2. Questions having the same semantic meaning can be phrased differently. That is, they might use different words and have different syntactic structures. This means question similarity can't be compared on syntax alone.

These two properties demand that questions are interpreted semantically in some way to account for word brevity and differences in syntax and vocabulary. An algorithm such as [23] is an effective way to accomplish this. Briefly, the idea is to compare questions semantically by examining their answers. The answers from a training set of Q&A pairs are compared by several measures, including cosine similarity (see the vector model in section 2.3.1) and a probability based language model. This last model is used to generate a likely question from an answer, and then to find similar answers by computing the likelihood that these answers would generate the target question.

### 3.2.1 Presentation of solutions

As seen, there are many variants of Q&A services. Some of the most characteristic ones will now be presented in increasing order of automation.

#### Yahoo! Answers

*Yahoo! Answers* [63] is based on leveraging the community built around the company's search and directory services. This service is provided at no cost and is completely user driven. It's also open so that anyone can read the answers.

Basically, users submit their questions to the system which posts them on something resembling a forum. Anyone can answer questions but no question is guaranteed an answer. Submitted answers are appended to their respecting

questions like forum postings. In this manner the service resembles a modern front-end to a traditional user-moderated forum with open discussions. Existing answers are searchable by keyword queries.

Though quality is attempted assured by a rating and reputation system, the lack of moderation means the service is plagued by spam and poorly written answers. In fact, *Ask* (formerly known as *Ask Jeeves*) previously attempted a similar service called *AnswerPoint*. This service has now been disbanded, but allegedly the conclusion was that the free nature of the service provided people with little incentive to answer other people's questions, especially the harder ones ([35]).

### Google Answers

*Google Answers* [55] try to ensure quality by providing a kind of marketplace for questions & answers.

In this solution, customers specify how much they want to pay for an answer and how fast they need it. Then they wait for someone to pick up their "order". The first person answering the question gets rewarded the sum assigned to the order. The better-payed and/or easier answers will naturally be picked up by someone more quickly.

The service is open and the answers are available to, and searchable by, anyone. Not everyone can write answers, though. Potential answering candidates must first prove their abilities by going through a screening process. This weeds out the most unsuitable elements. This restriction, and the payment incentive, are important differences from Yahoo!'s system. Consequently, both the quality and the response time of the answers seem notably improved.

### Info Angels

*Info Angels* [56] is an example of a fully manual service. It's also commercial and closed, which means each answer is only available to the person asking the question. In other words, there is no reusable knowledge base of Q&A open to the users.

The solution is based on employing a large number of full-time, certified information researchers. These can be contacted by any means, for instance by e-mail, phone (voice or SMS) or the web. Each customer will be assigned a personal information researcher for the session, referred to as an *Angel*. These individuals will immediately set to work figuring out the answer to their respective customer's question. They can even be instructed to locate and

purchase items or services matching their customer's specifications. Payment is handled by paying for the time the Angels spend on research.

The success of this approach is difficult to judge due to the closed nature of the service. The major benefit of this solution, however, seems to be the response time. As each user is assigned a personal assistant an answer will be returned fairly quickly. The major drawback is, of course, the expensiveness of full-time employment. And answers are still not instantaneous.

### Ask Jeeves

In addition to the previously mentioned AnswerPoint service, Ask Jeeves initially tried pursuing an advertisement-driven search engine based on questions rather than queries. This service, aptly named Ask Jeeves, was launched in 1996. Thus it was not only the first commercial QA engine for the web, but in fact one of the first web search engines in general.

In their approach, the submitted questions were automatically matched against manually annotated, precompiled question patterns for which editors had already discovered answers. All questions were monitored and those which the system could not find an answer to were marked as such. The editors would then try to find suitable answers on the web. The most popular questions were answered first in this manner. When a question had received an answer it was added to the knowledge base for future automatic lookup. Questions which could not be matched against known ones were transferred to a keyword-based search engine; Teoma.

The semi-automated QA approach proved to be most valuable within limited, specific domains. An example is technical support. Here, customers could efficiently be served useful answers in response to frequent questions without having to wait for a customer representative. This relieved frustration on both sides. When this part of the company was sold to Kanisa (now KNOVA [58]) in 2002, the service focused on Teoma, their keyword-based engine. This is the driving force behind the new Ask [48].

Though the original Ask Jeeves is disbanded, traces of the QA service can still be witnessed in the *Ask for Kids* [49] service. It's probably branded as such due to its severely limited coverage and focus on trivia. The new Ask service also has a limited form of QA, though of a different kind which will be discussed in section 3.5.6.

### FAQ Finder

*FAQ Finder* [11] is a well-known example of a static solution based on precompiled knowledge. When operative it was open and non-commercial in

nature. It was based on two stages:

1. Questions were keyword matched against entries in selected FAQ files. This had the main purpose of narrowing down the set of interesting Q&A pairs.

2. This set was then inspected further by more carefully matching the questions against the selected answers. Similarity was determined both statistically by comparing term-vectors (see the vector model in section 2.3.1) and semantically by consulting WordNet's synonym sets. Thus, the system accounted for differences in vocabulary.

The main purpose of the system was to provide automated navigation through FAQ files, not to answer every thinkable question. As such it could be restricted to a static collection. As neither the collection, nor the system, has been updated since 1996, it has long since been surpassed by more modern variants. These are largely based on the same principles though, differing primarily in their similarity measures.

### 3.2.2 Evaluation of Q&A

The primary attraction of Q&A solutions is the knowledge base of readily available answers. Since these answers have been carefully crafted and paired with questions, no extensive parsing or semantic understanding of the questions have to be performed in order to find suitable answers. Nor is it necessary to prepare or compile the answers. It's enough to find a similar question and retrieve the corresponding answer.

As should be apparent by now, Q&A services have a large number of potential problems:

- *Response time.* If an answer to a given question isn't readily available, the submitter might have to wait a long time before someone answers. This is main drawback of a dynamic Q&A service and defeats the purpose of any QA system; that of the always available expert. In contrast, automated solutions are instantaneous. For simple questions it's probably far more efficient to search for answer using a regular search engine than to wait for a manual response.

- *Limited coverage.* It goes without saying that a system based on manual answers can't hope to cover anywhere near the multitude and variety of an automated solution. This is the main drawback of a static Q&A service, but also a major problem for the dynamic ones. For an answer

to be available someone has necessarily had to go through the trouble of finding or researching it and submitting it. Due to this limitation these services are often supplemented by some kind of keyword-based engine. Questions which can't be mapped to the knowledge base are then passed on to this engine.

- *Poor quality.* This aspect applies mainly to user-driven solutions without quality assurance. Answers from experts, on the other hand, are likely to surpass automated ones. It's fairly easy to assure some level of quality from automation, though, by mining answers from reliable sources. As such, sabotage and spam is less likely than in a free-for-all service where anyone can answer. It's similarly easy to control the knowledge in a static solution, of course.

- *Lack of trust.* As the person asking the questions doesn't know the answer, it's fairly easy to be misguided by a deliberate false answer. For instance, on discussion forums answers are rarely backed up by adequate references or evidence. Unless the reply is given by someone that can be verified as an authority on the subject, it's just a statement of belief by some random person. An automated system, on the other hand, can easily present the context and source of its matches for verification.

- *Lack of incentive.* As concluded by the AnswerPoint team, people have little incentive to answer hard questions. Even when rewarded by payment, as in the case of Google Answers, there's a tendency to prefer answering many easy questions rather than a few tough ones. This tends to pay better in the long run. On the other hand, harder questions are also more difficult to find answers to by automated systems.

## 3.3 Dimensions of automation

In light of the problems with Q&A solutions, most research teams have apparently turned their attention towards full automation. The promise of instant answers from multiple sources is luring. As shall soon be apparent, though, automating question answering is by no means trivial. I mentioned in section 3.2 that question answering can only be realized in its entirety by creating a system able to rationalize over the full extent of human thoughts and languages. Fortunately, the problem doesn't have to be solved exhaustively in order to achieve useful results. Much progress has already been made simply by limiting the scope of the problem to manageable tasks.

Any significantly complex problem has many potential solutions, though, and QA is no different. Thus, a lot of widely different ideas have been explored for QA purposes. To be able to discuss and compare these approaches it's important to identify the most significant *dimensions* through which the systems differ. By "dimensions" I mean the various considerations that has to be made when planning a QA system. The following dimensions will be detailed:

- Domain
- Question type
- Answer type
- Method
- Scale
- Processing

### 3.3.1 Domain

The answering ability of the system is largely dependent on the heterogeneity of the domain in which it's going to be applied. If the domain is limited to a specific set of interesting topics, the system can be perfectly tuned to recognize common answers within those precise topics. The more varied the topics, and hence also the terminology, the more work is required to reach the same answer quality and coverage as in a more homogeneous domain. This problem is especially significant if the system relies heavily on manually developed rules or otherwise hand-crafted knowledge. But as long as it only has to handle one domain, or at least not more than a few, this work is still manageable. Such a specialized system is called *domain dependent*, and is referred to as dealing with *closed domain* topics.

The real challenge starts when the system needs to handle topics from an *open domain*. The system is now *domain independent* and general purpose, rather than specialized. The fact that the domain is "open" means that it's essentially no longer possible to determine every kind of topic it has to handle. Basically, the heterogeneity has exploded and become nonassessable. Relying solely on hand-crafted knowledge is thus no longer an option, no matter the amount of work invested. It it possible, though, to identify the most important questions and topics (e.g. through user feedback, expert opinions or usage statistics) and focus the work on these. To be able to

reach an acceptable level of coverage the system then has to provide some kind of fall-back solution to handle unsupported inquiries.

An example of a successful domain-dependent QA system is the *reading comprehension system*, e.g. Deep Read [21]. These systems are used to measure how well human subjects understand the texts they have just been reading. While useful, these systems only work on specific texts and accept only a limited set of questions. To be genuinely valuable, this answering ability has to be expanded to work on large, full-text collections without any domain restrictions. This is the big promise granted by the merging of QA with IR and IE.

### 3.3.2 Question type

#### Ways to ask a question

What topic a question asks for is linked to the domain the question targets. But while there is an unlimited number of possible topics there are only a few different manners in which to phrase a question. These are bound by the specific ways in which human languages are used to produce inquiries.

A question can be classified by the way it addresses the person for which the question is intended. Table 3.2 shows a few examples.

| **Question** | **Type** |
|---|---|
| Who invented Trivial Pursuit? | Direct |
| Do you know in which city the River Sein is? | Indirect |
| Tell me when Elvis Presley died. | Commanding |
| I want to know who killed J.F.K. | Intention |

Table 3.2: Ways to ask a question

Only a direct question is interesting from the perspective of carrying meaningful information content. The others are just ways of expressing subtleties in human communication and socialization. For a computer these are simply unnecessarily complex ways to get to the point. For instance, *Tell me when Elvis Presley died* could just as easily be phrased as *When did Elvis Presley die?*. Every type of question that is not on a direct form can thus preferably be reduced to this form, e.g. by anti-phrasing, before being processed further.

Note that all of the questions in this table use so-called *wh-words*. That is, they all use one of the *interrogative words* starting with the letters "wh". In modern English these words consist of who, (whom, whose), what, which,

where, when, why and how. A question that is not direct might not necessarily contain one of these words, as in the example *Give me the address of the Opera*. This kind of question is essentially a "what" question, though, as can be seen by the rephrasing *What is the address of the Opera?*.

### Multi-sentence questions

So far, the example questions have been rather straightforward. That is, they've been concise and have contained all necessary information within a sentence. Once questions start spanning multiple sentences they are far more difficult to handle. The topic of the question might no longer be part of the same sentence as the interrogative word, if there is such a word at all. Further, there might even be any number of topics, and these might be intertwined with suggestions, observations or partial solutions. Consider the following question from a technical support forum:

(3.5) Since upgrading to the final 6.3.0 version I am having the following problem: When I browse the music folder and add music to the play list the right side pane turns white and does not show the chosen music until I manually refresh the page. This does not occur when using any of the other browse functions. I have tested this in Firefox 1.5 and Internet Explorer with the same result. It seems to be skin related. It malfunctions with dark, purple, bagpuss, NBMU, and touch, but not with default or default2. Suggestions?

While this is indeed a question it might be notoriously difficult for a computer to identify as such. Compared to the previous examples it is not even phrased as a proper question. The only real indication is the question mark at the end. Automatically finding the focus of this question demands severe language processing and is likely beyond the abilities of current technology. The question is obviously meant for human consumption though. Most of the information is meant to aid in understanding the problem and formulating a proper response. This information is irrelevant for the sole purpose of locating an existing answer to the question. If it were to be targeted at a computer it could thus be greatly simplified, for instance as:

(3.6) Why is the right side pane blank when I browse the music folder in version 6.3.0.?

In short, it might be necessary to put some limitations on the way questions can be phrased when addressed to a computer. While some of the promise of QA is for the users to be able to phrase questions in their own

languages, it's not realistic to expect current technology to understand overly complex constructions. Indeed, another human might even have difficulty understanding what such a question is all about. This kind of restriction is not necessarily a drawback though. By forcing the users to phrase clear questions it might even aid them in understanding the essence of their inquiries.

### 3.3.3 Answer type

Questions can also be classified based on the complexity of the expected answer type. There's an important class distinction between those questions that can be regarded as closed and those which are correspondingly open. A question is referred to as *closed* as long as it's possible to answer by identifying some distinct item in the text as the target of inquiry. Examples of closed questions are seen in table 3.3.

| **Question** | **Answer type** |
|---|---|
| Where do apple snails live? | Location |
| In which city would you find the Louvre? | City |
| How big does a pig get? | Measurement |
| What metal has the highest melting point? | Metal |
| Who was the first person to reach the North Pole? | Person |
| When was the telephone invented? | Date |

Table 3.3: Closed questions

In contrast, an *open* question is one which might require an explanation, a procedure, an opinion or some kind of causal relationship to explain. These kinds of questions often use the interrogatives "how" and "why". Table 3.4 gives examples of open questions.

Some questions are seemingly answerable by a simple "yes" or "no". An example is *Does a shark lay eggs?* from table 3.4. One might thus be led to think that it belongs to the closed class. Answering this kind of question is not an act of finding an actual "yes" or "no" in the text, though, but rather of finding a statement confirming or denying the proposition. The particular nature of this statement is indeterminable This example question is particularly deceptive, as the answer is either yes or no depending on the shark. A good answer is thus one which asserts this fact. Such an answer looks decidedly close to an explanation, though, which only confirms the open nature of the question. Indeed, an explanation might be valuable in supporting any "yes/no" question.

| **Question** | **Answer type** |
|---|---|
| Does a shark lay eggs? | Yes/no |
| Who was Galileo? | Description |
| Will there be a new world war? | Opinion |
| Why did the U.S. invade Iraq? | Opinion |
| How accurate are HIV tests? | Explanation |
| How popular is the iPod Nano? | Sentiment |
| What does CPR stand for? | Definition |

Table 3.4: Open questions

So far the questions have asked for a single answer. Closed-classed questions might also explicitly ask for multiple entities (table 3.5). These are often called *list questions*, as the system is required to return a list rather than a specific entity.

| **Question** | **Answer type** |
|---|---|
| Which presidents were republicans? | Persons |
| List the names of cell phone manufacturers. | Companies |
| What are Canada's two territories? | Locations |
| Name foods high in zinc. | Nourishments |

Table 3.5: List questions

It's understandably easier for a QA system to answer closed-class questions than open ones. Identifying a specific entity is far simpler than some loose description. Most current QA systems consequently focus their attention on the closed class.

Finally, a very important attribute of any system is the ability to assert that an answer does not exist. This is an Achilles heel of many system based on deductive logic. While highly proficient at finding a proof (e.g. an answer) they are rather hopeless at figuring out that there is no such proof. Instead they will just try new chains of inference, possibly indefinitely. Determining whether an answer exists was identified as critical enough to warrant its own measure in the TREC conferences from 2001 and onwards (section 3.4.1). Systems that are based on identifying potential answer candidates in advance (i.e. offline) rather than on-the-fly (i.e. online) have a natural advantage in

this department. If a question does not match one of the candidates, an answer does not exist (e.g. can not be recognized).

### 3.3.4 Automation

There are primarily two ways to go when working with patterns: heuristics or machine learning. Both of these have advantages and disadvantages, and the choice of which to pick largely depends on the domain of use. There are, of course, also many variants in between which utilize automation to various degrees.

#### Heuristics

*Heuristics*, or hand-coded rules, are especially effective for domain-dependent systems as they can be tailored specifically to each domain. Basically, this route implies analyzing questions and answers manually and then writing custom rules to handle the cases identified. Methods based on regular expressions and pattern matching fall naturally under heuristics.

There have been a few attempts, though, at applying machine learning also to entity extraction [64]. The intent has been to automatically discover new patterns that can be used for recognizing entities Machine learning can potentially yield higher recall as it can identify and cover more cases within a limited time frame. This is especially true in situations where there is a huge variation in the material to be processed, or just too much material to deal with in an orderly fashion. But, on the other hand, heuristics tend to be more precise and are perfect for cases of low variation. Humans can come up with much more clever, not to mention creative, schemes than an automated algorithm. As long as it's practical to apply heuristics, both accuracy and general performance is likely to surpass automated methods. For instance, a recent comparison in ontology building [40] uncovered that the additional rules discovered by machine learning had significantly lower precision than the original manual ones.

#### Machine learning

Heuristics are not very flexible, though, and they usually require a substantial of manual labor to construct. Systems that employ them are thus often supplemented with *learning algorithms*. As the name implies, these learn from examples, either automatically (unsupervised) or semi-automatically (supervised). When properly trained they can detect patterns that would

otherwise go unnoticed and can classify information based on parameters not available to a rule set

As an example of a learning algorithm, *Tritus* [1] uses three stages:

1. It classifies the questions based on their similarity to questions in a training set.

2. It uses the answers to those questions in the same set to construct queries likely to match similar patterns.

3. It tweaks the system to use different search engines based on which engine performs best with which type of questions. This tweak is performed by running test questions through each engine and comparing their results with the test answers.

Learning algorithms have another advantage. Using NLP as a basis usually implies language dependence, as both the grammar (the valid constructs of a language) and the lexicon (the words of a language) are unique. By providing training material in the desired target language, a learning algorithm can adapt to that language. As an example, Tritus learns the patterns of answers, and thus can be trained on many languages with only minor modifications.

### 3.3.5 Scale

Answers to questions must be extracted from some corpus of text material. The scale of this corpus has a major influence on the type of processing that will be required (section 3.3.6). It will also influence the applicability of the system, the trustworthiness of the sources and the quality of the answers.

#### Small-scale benefits

Domain-dependent systems are usually of a rather small scale because of their very specific natures. But a small-scale system might just as well be open-domain. There's no direct dependency between domain and scale. For instance, early open-domain systems had no choice but to limit their scale as there was simply not enough text material readily available in digitized form. Even today, with the vast and accessible web, small-scale answering has its uses:

- If one is interested in providing answers from a specific collection of documents, these can be selectively indexed by an IR engine. For

instance, a business might be interested in discovering new facts from its private archives. Or it might want to provide some form of automated support to its employees or customers. This means the corpus should be limited to the corporation's knowledge base. As another example, a university might want to restrict its answering system to sources containing material known to be correct and useful to its students. They can thus avoid the rants, opinions and general noise present on the web. In other words, a small-scale system can provide selective source control for sensible or specialized information.

- Certain kinds of material is better suited for answer mining than others. For factual information, for instance, an encyclopedia is likely to contain more useful information per page than the average web site. Likewise, on any given topic there's bound to be a number of specialized sites that are guaranteed to contain much more relevant knowledge on that topic than the web in general. Identifying and selectively indexing high quality sites can improve overall answer quality. Another useful source is sites or pages providing manually prepared answers, e.g. FAQ-style information. Semi-automated Q&A solutions (section 3.2) have already proven the worth of focusing on such sources.

When the source of answers must be controlled in some way, the scale will naturally have to be limited accordingly. This rules out using general, web-based search engines for document retrieval, because these don't allow detailed source control. Rather, a specific engine must be set up to index only desired content, but such an engine might not be available for all purposes. Finally, a small-scale likely has negative impact on the general coverage of the system but will, on the other hand, provide benefits in the form of better and more reliable answers.

### Large-scale benefits

The vast amounts of information available on the web makes it an attractive resource for answering a variety of questions. The sheer number of pages almost ensures that no matter the question, there is bound to be an answer out there somewhere. Not only that, but the answer is likely to be phrased in several different ways. Some of these will be far easier to identify than others. By focusing on the simplest ones, the effort needed to find good answers can thus be greatly reduced. If simple patterns are adequate for matching these there is no need for further structure and meaning analysis.

An example will make this clear. Let's say the system is given the question "Who is the king of Norway?". This relation can be expressed in numerous

ways, not necessarily using the terms in the question. But by searching through enough documents we're eventually bound to find a sentence containing the exact wording "X is the king of Norway". The difficult task of semantically relating different terms to the same concept can thus be bypassed.

Clarke et al. [13] noticed that the number of accurately extracted answers increased as the corpus grew. However, he also remarked that this advantage is somewhat counterbalanced by the poor quality of the individual documents. Fortunately, redundancy can act as a safeguard against poor quality and erroneous information; the more sources say the same thing, the more likely it is to be true. Similar answers can thus "vote" for each other, and the highest voted candidate can be returned as the most likely answer. But repeating something frequently doesn't necessarily mean its true. Clarke et al. thus also demonstrate how to use information gathered from the web to instead boost answers from a primary, more authoritative source. This source may, for instance, come from a small-scale controlled index as described above.

Banko et al. [5] further suggested that redundancy could be useful even when no obvious answer strings are to be found. Relationships could be discovered between entities if they occurred frequently and likely answers could be guessed at based on the question-type and distance to the nearest likely answer entity. This idea was not investigated further in the mentioned paper, but an example of use is given.

### 3.3.6 Degree of NLP

A major separating factor between the implementations is the degree of NLP involved. Even though an absolute boundary can't be drawn between *NLP heavy* and *NLP light* systems, the designers' intentions can at least be interpreted as either desiring to use NLP as a major component of their approaches or not. The resulting implementations can then be classified into respectively *deep* or *shallow* approaches, referring to the depth at which they process the language.

The decision of which approach to pick seems to be largely based on the amount of data available for processing. The general rule of thumb is that the more data available, the less sophistication required. This rule follows intuitively from the observation that given enough heterogeneous data even simple pattern-based techniques are bound to discover something of interest (section 3.3.5).

This relation between processing power and data storage is not a new discovery. Decades of research within wholly separate areas of computer science

has concluded that processing power can to a large degree compensate for data deficiency and vice versa. In other words, algorithms can estimate and rebuild missing data, and pre-generated data can boost simple algorithms. In this case, thoroughness can compensate for information poverty, and information abundance can compensate for superficial behavior. The methods always have to be adapted to the resources available within the target domain.

The question of whether to choose a deep or a shallow approach is a recurring theme in recent QA literature. Some examples are [13], [15], [7] and [12]. These teams have all agreed in their conclusions: Deep processing is best suited to detailed questions within specific domains while shallow pattern matching is better suited to general open-domain questions. With the huge, ever-growing information repository of the Internet, this topic is now more relevant than ever. Indeed, an increasing number of systems submitted to recent QA conferences (not the least TREC (section 3.4.1)) have based their implementations on information from the Web.

Given the increasing availability of free information, much of the current research trend (i.e. spanning the last five to ten years) seems to be moving steadily from deep semantic processing to clever, shallow pattern matching. In exploring the potency of this trend, as well as sketching out the foundations for my own QA implementation, I've read and compared papers presenting both points of view. Systems from proponents of both schools of thought will thus be presented in section 3.5.

## 3.4 Comparing and evaluating systems

### 3.4.1 TREC: The QA conference

To be able to compare QA systems to each other and establish potential benefits of the various approaches, it's imperative to have a common set of data upon which to test. These kinds of sets exist, and the *Text REtrieval Conference (TREC)* provide some of the most widely used ones. In addition to large test sets, TREC provides a controlled environment for scoring and comparison, as well as a forum for discussion and cooperation.

The conference has proved a major success, with hundreds of participants from both academic, commercial and governmental groups. The performances of submitted systems have improved considerably through the years and much of this open research has directly benefited the community. Several of the innovations have also been commercially viable - some even leading to the establishment of new businesses.

TREC is conducted by the U.S. National Institute of Standards and Technology (NIST) with support from the U.S. Advanced Research and Development Activity (ARDA), and has run yearly since 1992. There are many *tracks* running in parallel, each trying to solve different issues within information retrieval and processing. The QA track started in 1999, and the stated goal was to be able to return exact answers, from open domains, in response to questions phrased in natural language.

### Evolution of the QA track

In a recently released book about the TREC conferences [47], a concise history of all the yearly tracks (up to and including 2003) is given. The evolution of the QA track is interesting as it shows how the systems become progressively more sophisticated as the track grows more demanding. Through this evolution, many useful insights have surfaced that are of interest to my own research. Further, the stated goals of each year's track provides some background for the systems that participated in that track. I will soon be presenting some of these systems (section 3.5). For these reasons it is prudent to provide a short outline of the annual tracks.

### TREC-8

In TREC-8 each participant was given a series of questions and a collection of documents. For each question, the task was to be able to locate an answer in the collection. Each question was guaranteed to contain an explicit answer as all the questions were simply back-formulations from sentences in the corpus. E.g. a question like *Who wrote Hamlet?* would have been generated from an existing sentence like *Shakespeare wrote Hamlet* by a simple rephrasing.

Due to this straightforward correspondence, finding an answer was fairly trivial. The general approach of most systems was consequently to classify questions according to the interrogative word and the expected answer type (e.g. "who" refers to a "person"). Documents were then retrieved (by traditional IR) and shallow parsed for entities of the right type occurring near interrogative words. For instance matching a person name like "Shakespeare" occurring near "wrote Hamlet" in the example above.

Not all questions revealed the entity type of the answer. Questions like *What language is commonly used in Bombay?* or *Name a film in which Jude Law acted* couldn't readily be answered by using this approach, and the systems struggled accordingly. The more robust ones reverted to general entity extraction with varied results.

### TREC-9

TREC-9 used essentially the same document collection but expanded upon it by approximately 50% more material. The biggest difference from before was that questions were now drawn from actual log files instead of simply being back-formulations from the documents. In other words, these were real questions from real users of online encyclopedias and other sources. Answers were still guaranteed to exist in the corpus, but they wouldn't necessarily be phrased similarly to the questions.

Some questions were now also more abstract than mere factoids, e.g. *Who is Mahatma Gandhi?*. And syntactic variations were created to test the robustness of the question parsing components. E.g. *What is the name of the tallest mountain?* might be accompanied by a rewrite such as *What is the highest peak in the world?*.

To be able to handle these variations the systems were refined to classify questions based on factors besides the interrogative word. Many started using machine readable dictionaries like WordNet to account for synonyms and related words and to verify that the extracted answer was of the right entity type.

An unintentional side-effect of these syntactic rewrites were the slight semantic variations that unintentionally followed. E.g. a "peak", from the example above, doesn't necessarily refer to a "mountain", but could be basically be any protruding structure. This proved to be one of the toughest challenges both for the systems and the judges.

### TREC 2001

In 2001, the QA track still used the same corpus as the previous year, but brought two significant additions to the questions:

1. Questions were no longer guaranteed to have an answer in the provided corpus. In these cases, a system was expected to return *NIL* to signify its belief that no answer existed. Systems that always reverted to returning their best guess were thus penalized.

2. A new type of question appeared that required processing the data in new ways; *the list question*. This kind of question was deemed different enough to warrant its own, separate task. Basically, the purpose of the task was to respond to questions having more than one entity as answer. Further, the exact amount of entities to be returned was stated explicitly. An example of such a question is *Name 9 novels written by Isaac Asimov*. To be successful systems had to compile information from

multiple sources and recognize duplicates (even if they were formulated differently) to reach the target number.

By the time of this track, systems could be observed to belong to one of two main categories based on their general approach. Those which had continued improving on the general technique mentioned earlier had now become quite sophisticated, and used increasingly complex linguistic methods. But a second kind of system had also started appearing — using a quite different strategy. Instead of digging deep into the text and trying to reason upon the data, these relied on examining huge amounts of data using simple patterns in the belief that a match would eventually be found. The advantages of each of these two strategies were detailed in section 3.3.6.

### TREC 2002

The track held in 2002 switched to an entirely new corpus; the *AQUAINT Corpus of English News Text*. And once again, two significant changes to the questions were made:

1. Answers were required to be exact rather than portions (e.g. a sentence) of text with the answer contained somewhere within. The idea behind this change was for the systems to provide precise answers that could be used automatically for further processing. Previously it had been enough that a human was able to identify the answer from the returned snippet. For a discussion of the effects of this change, see section 3.4.3.

2. Each answer had to be bundled with a *confidence score*, representing the systems' degree of certainty that the answer is correct. Further, each question and answer pair were to be ranked according to this score (explained in section 3.4.2 below).

### TREC 2003

By 2003, it was decided to combine the two previous question types, factoid and list, into one task. Most participants had previously focused their efforts on factoids, and the intent was to encourage interest in a broader range of questions. Besides, list questions also ask for factoids — only several at the time. Additionally, a new class of questions was introduced: the definition question.

A *definition question*, e.g. *What is acetaminophen?* is more difficult to answer than merely locating some entity. The answer usually requires providing a description of some kind — at the very least a short noun phrase

like *a crystalline compound*. The answer should preferably also explain the purpose of the compound, like in the more complete phrase: *a crystalline compound used in chemical synthesis and in medicine to relieve pain and reduce fevers*.

List questions were also modified so that they did not inquire about any specific number of targets, but instead expected the systems to return all interesting entities matching the question; e.g. in response to a question like *Which novels are written by Isaac Asimov?*. This change made the list class more generally useful, but also more difficult to evaluate because the systems might return different entities to the same question (see section 3.4.2).

### TREC 2004

Since the QA track's introduction it had primarily focused on plain factoid questions. Through several iterations, the best participating systems had matured to the point that they could answer more than 80% of these open-domain questions [43]. As the QA track evolved, it was argued that, while useful for certain tasks, factoid answers was only part of what real users needed. When observed, users tended to ask more knowledge gathering questions than query about specific facts. Because of this tendency, list and definition questions were gradually introduced. But the real departure came in 2004 [45]:

1. Questions were no longer independent and self-contained, but were instead grouped together based on some topic. Only the topic itself contained the subject. The questions merely referred back to the topic using anaphora. For example, given the topic *Insane Clown Posse* one of the questions was *Who are the members of this group?*, and another was *What is their biggest hit?*. These questions are useless without considering the topic.

2. The definition questions were replaced with a new class that was informally dubbed "other". Basically, this "other" meant providing additional, related information that was not explicitly asked for nor specified. The idea was, for a given topic, to present users with valuable knowledge that they would not have thought of asking themselves. The answer extraction process thus required the ability to separate essential descriptions from peripheral ones.

To allow for this new knowledge-centric view, the structure of the test set also had to change. The test set moved from a custom, but basically plain, list of questions to an XML-based hierarchy. Here, the type of question (i.e.

factoid, list or "other") could be specified and a group of questions could be bundled together with the topic into a *question series*.

### TREC 2005

Due to the major modifications of the previous year's track, the participants needed more time to adjust their systems to the new paradigm. It was thus decided to run the 2005 track largely unchanged from the previous one. Still, one noteworthy change was made:

*Events* were introduced to the class of topics. The previous track had only focused on "things" with substance; e.g. persons or vehicles. Questions might now also pertain to some intangible happening. For example, given the topic *Russian submarine Kursk sinks*, one of the questions was *How many crewmen were lost in the disaster?*. This class was added because of the newswire nature of the corpus in which events played a major role. Further, subsequent questions now started inquiring about the answers from previous questions, e.g. *Which countries expressed regret about the loss?*. This reaffirmed the *session* paradigm.

The track also wanted to research the role of document ranking in QA. That is, whether ranking affects the quality of the final answers and if so, in what way. A new task to measure this property was added; the *document ranking task*. Participants in this task were not required to present answers, which opened the task to the IR community.

It was also decided to focus further on the "bigger picture" of knowledge gathering. This resulted in yet another task; the *relationship task*. Here, the systems would be presented a description of the information need rather than a series of questions. For example *The analyst is interested in the Al Qaeda terrorist network. We know Osama Bin Laden is in charge, but what other organizations and people are involved with Al Qaeda?*. This kind of "question" is a mix of topic, known facts and inquiries for specific targets.

### TREC 2006

As the 2006 track is currently under progress and closed to non-participants, the following information has been gathered mostly from recent discussions on the TREC QA mailing list [60].

The main task of 2006 will again be similar to the previous two years. The major change in this track will be the addition of the ciQA task for *complex, interactive Question Answering*. This track is partly a replacement for last year's relationship task and partly a way to formally introduce interactivity

(i.e. user feedback) into the QA process. Questions will consist of a template and a narrative as in:

(3.7) **Template:** What is the relationship between [person: Colombian businessmen] and [organization: paramilitary forces]?

**Narrative:** Specifically, the analyst would like to know of evidence that business interests in Colombia are still funding the AUC paramilitary organization.

The purpose of the *template* is primarily to ease the task of the participants. The performance leaders in the tracks up to and including 2005 were the few systems that were heavy on semantic processing, e.g. PowerAnswer (see 3.5.4). These were naturally more proficient at finding semantically correct answers than the others. In an attempt to level out the playing field, the templates will explicitly state the entity types.

The free-form *narrative* gives the participating systems a chance to deduce additional meaning. They can subsequently provide any valuable knowledge they deem fit. This opportunity will benefit the semantically adept systems and takes over from the relationship task.

Finally, the *interactivity* is optional and entails allowing participants one round of interaction with their systems. That is, they're allowed providing feedback once per question to their respective systems and let the final answers stand. The intent is to explore how this kind of interactivity affects answer quality and user satisfaction.

### The future of the tracks

The major departures introduced since 2004 essentially mean the tracks have departed from the traditional search paradigm of separate and independent queries. Instead they have moved increasingly towards supporting a kind of dialogue with the user, where the contexts from earlier questions maintains their validity in the succeeding ones. This kind of functionality in a multi-user environment requires maintaining some kind of session to keep track of which questions belong to which users. Also, the system must be able to recognize when the topic changes.

Further, the systems are now intended to locate and present helpful knowledge on a given topic. They're consequently becoming more knowledge-gathering rather than fact-specific which requires a different design from previously. This shift is probably a good thing, as the focus is moving from mere system performance to also include user benefits, as requested by participants

(section 3.4.3). But as this is a major departure from traditional QA systems, this view will not be discussed any further in this thesis.

### 3.4.2 Measuring performance

As should be apparent by now, TREC is providing a genuinely useful baseline of progress and achievements within the field of QA as a whole. Another important aspect of the conferences is their establishment of several useful measures for evaluating the performances of the participating systems. These measures are representative of the general consensus for effective comparisons within the field. As such they are beneficial to explain, and again it's useful to follow their evolution from the beginning.

In 1999 (TREC-8), two answers for each question was required. One of 50 bytes and the other of 250 bytes length. The systems thus merely returned an excerpt of the appropriate length from the sentence(s) containing the answer. Pure IR (i.e. passage retrieval) proved quite adequate for 250-byte answers, while IE techniques were required to successfully pin-point a 50-byte answer. The same results were observed the following year, and so the 250-byte answers were retired in 2001.

In 2000 (TREC-9), each extracted answer was required to provide a supporting document in the corpus from which the answer was found. It was no longer enough to stumble upon the right entity by chance. Two scoring schemes were devised; a *lenient* score for unsupported answers, and a *strict* score for supported ones.

In 2001, the main difference was the inclusion of a separate measure for answers correctly marked as NIL (e.g. not found in the provided corpus).

For the first three years (1999-2001), participants were instructed to return an ordered list of five answers to each question. The main scoring metric was the *mean reciprocal rank* or *MRR*. Basically, each answer-list was scored by the reciprocal (or inverse) of the first correct answer. For instance, if the first correct answer was the third one, the answer-list was scored with 1/3 of the maximum, i.e. 0.33 (as the maximum is 1). The score for each system as a whole was then the mean of all these individual reciprocals.

Due to the impreciseness of the expected target of certain questions, multiple correct answers could be returned. For instance, consider the questions *What is the deepest lake?* and *Where is the Eiffel Tower?*. The answers depend on what scale is considered, e.g. a city, a country or the world. This was handled by granting a full score, regardless of scale, as long as the answers were deemed sensible by a judge.

In 2002, the scoring scheme was revised further. Answers now had to be exact and were judged either wrong, unsupported (but correct), inexact (but

correct) or right. Only the answers judged right counted in the final score.

By 2002, systems also had to rank their answers according to their confidence in their correctness (as previously noted in section 3.4.1). This resulting confidence-weighted score rewarded correct answers more the higher the confidence had been set. Specifically, for the total amount of questions $Q$ in the test set with $r_i$ being the number of correct answers for the first $i$ ranks, answers were rewarded according to the following equation:

$$\frac{1}{Q}\sum_{i=1}^{Q}\frac{r_i}{i} \tag{3.8}$$

### Evaluating list answers

In 2002, list answers were evaluated based on their accuracy, that is:

$$\frac{number\ of\ distinct\ entities\ located}{total\ number\ of\ entities\ present} \tag{3.9}$$

This was sufficient as long as each question stated the number of entities present in the collection.

By 2003, list questions were expected to be answered by the maximum number of correct entities found. As this number would vary from system to system a new measure was required. List questions were thus instead evaluated according to their *instance precision* ($IP$) and *instance recall* ($IRe$). This is a measure similar to the balanced F-measure from IR (section 2.3.2). In this case:

$$F = \frac{2 \times IP \times IRe}{(IP + IRe)} \tag{3.10}$$

List answers could now be scored on a balance between how many correct instances they managed to locate, and how many instances were correct of the ones returned.

### Evaluating definition answers

Definition questions proved more difficult to evaluate than factoids, as several answers might be deemed correct. Correct answers might further contain various degrees of information. Because answers were no longer entities they could no longer be required to be exact. In fact, they were allowed to contain any amount of elaboration as long as it had some kind of focus. To be able to evaluate this focus the possible answers to a question were divided into *information nuggets*; each a separate, measurable fact. The judges would

then classify these nuggets as vital or not. Finally, each answer would be checked for which vital nuggets it covered and scored accordingly.

Recall alone soon proved insufficient as a measure because this always prioritized longer answers. The more information returned the higher the score would become. A guaranteed road to success would then be to return the whole document. To reward selectiveness some measure of precision had to devised. It was decided upon a maximum allowance of characters and to reward shorter answers higher than longer ones as long as it covered the same vital facts. Precision would now be measured thusly:

$$Precision = 1 - \frac{length - allowance}{length} \tag{3.11}$$

As only vital nuggets were considered in this computation the non-vital ones were simply regarded as neutral.

The final score was again an F-measure, but this time rewarding recall by 5 times as much as precision, due to the judges preference for recall and the crudeness of the precision estimation. Hence, according to the general F-measure given in section 2.3.2:

$$F(\beta = 5) = \frac{26 * precision * recall}{(25 * precision) + recall} \tag{3.12}$$

### Comparing performance on the web

Many systems competing in TREC have started using the web as a source to leverage the advantages inherent in its massive corpus (section 3.3.5). Comparing these systems would be difficult if relying on the web exclusively to provide the test material. The web is vast and ever changing, creating a tough environment for controlled experiments. To compare their systems, teams have instead turned to predefined data sets like the ones provided by TREC. Of course, as the environments are very different, the results can't be directly translated to performance on the web. The immense corpora of the web offers vastly more resources than the TREC sets, and is much more heterogeneous.

In order to take advantage of the web and still be able to measure their results against a test corpora, some teams have created a workaround. Katz et al. [26] employs a technique they refer to as *answer projection*. In essence this is an extra step that which can be used to combine the wealth of information on the web with the reliable data in smaller sources like TREC or encyclopedias. The general idea is to mine the large and noisy sources for candidate answers, and then verify these answers by searching for them in

smaller but more reliable collections. In this way, trusted sources can be used for verification, either by finding support directly or by issuing new queries and mining for confirming answers, even though the corpus wasn't actually used in retrieving the answer.

As the TREC collection is very different from the web, the scores from these conferences must be viewed in light of that limitation and not as a fair comparison of deep versus shallow approaches.

### 3.4.3 Answer length and the role of context

Despite numerous performance advances within QA, the area of user interface exploration has almost been neglected.

Satisfied with current discoveries, the TREC QA track moved from allowing passage-style answers toward demanding exact answers [44]. The primary reason was to push the technological envelope toward more sophisticated and accurate algorithms. It's also easier to compare the efficiency of systems when they extract and return the exact answers rather than just returning passages containing the answers in some form. While these reasons probably justify the shift, little effort has been made to explore how this trend might affect actual users of these kinds of systems.

Lin et al. [30] tried to shed some light on this issue and explore the role of context in QA. They point out the fact that improvements in performance don't necessarily translate into increased usability, and that ultimately the benefits to the end users are what matters. Additionally, results gathered from IR interface studies can't be directly translated to QA. The interaction in IR mainly consists of browsing results and navigating quickly and comfortably without too much cognitive load. In QA context plays a different role; namely that of providing additional information and lending credibility to the answer.

They also wanted to find out to what degree the trustworthiness of the source affected the users' acceptance of the answers. Initially users believed that source reliability and context would be equally important to their acceptance. After the test, however, the importance of reliability had dropped noticeably while the importance of context had risen comparably, leaving a statistically significant gap between them. Interviews revealed that users didn't really mind where the answers came from, and that they preferred to read some portion of the text to establish the answers' trustworthiness regardless of source reliability.

User-testing was performed with four categories of answers, each with increasing amounts of context: exact (answer only), sentence, paragraph and document. Intriguingly more than half the users (53.33%) preferred a

paragraph sized response. One in five (23.33%) preferred the whole document and likewise (20.00%) for sentences. Almost none (3.33%) were satisfied with just the answer, which is alarming given that this kind of response is preferred in TREC QA. Users explained that paragraphs usually gave them the right amount of information, exact answers were too little and the entire document too much. Sentences weren't thought to provide much benefit over just the answers. While more experimentation is certainly needed, this finding should definitely be taken into consideration when designing a QA interface.

Lin et al. also tested multi-question scenarios, in which users were asked to complete, as quickly as possible, a scenario with closely-related questions on the same topic. Testing revealed that the completion time remained practically equal regardless of the amount of context. On the other hand, the more context users were given, the fewer questions they had to ask to complete the whole scenario. In fact, they needed to ask more than twice as many questions when given exact answers as opposed to being handed the whole document. The team thus concluded that given additional surrounding text, users will read it when they want to learn more. Context helps users validate the answers as well as provide additional feedback on related questions.

## 3.5 Presentation of specific approaches

A selection of QA solutions will now be presented. Each represents a different take on the general issues and achieves individual benefits compared to the others.

### 3.5.1 General architecture

Depending on how a particular system balances the dimensions (section 3.3) to suit its specific purpose or intent, quite distinctive designs will result. A few examples of this variety will be explored in a moment. Individual peculiarities aside, though, in the end they all have to accomplish the same thing: Providing an answer to a natural language question. While the details of how this is achieved differs, the general steps that have to be undertaken are essentially the same for every system. These steps will now be briefly presented.

There are four main components in any QA system which operate in the following sequence:

1. *Question classification.* The topic of the question and the expected answer type is determined.

2. *Query transformation.* The question is rephrased into a query likely to improve the search (or lookup), typically keyword-based.

3. *Document retrieval.* Documents containing potential answers are retrieved for closer inspection. This step is often implemented by interfacing with a standard IR search engine.

4. *Answer extraction.* A suitable answer is attempted located within the retrieved documents.

### 3.5.2 AskMSR

Brill explores how far it's possible to get using only simple pattern matching techniques [7]. He refers to these as *data-driven methods* [9] to emphasize their reliance on large amounts of data rather than on sophisticated heuristics.

He argues that large online text resources, like the World Wide Web, makes the extraction job harder for complex NLP systems trained on limited domains due to increased heterogeneity The performance potential of plain open-domain pattern matchers, on the other hand, is increased in this very same environment.

#### Method

To prove his point he refers to his own QA system *AskMSR* (Ask Microsoft Research) ([8]), where pattern matching against the web is the primary driving factor. AskMSR has many resemblances to the contemporary Mulder system [28], which might be more familiar to some.

In particular he utilizes two main techniques:

1. *Simple query rewrites.* Questions are rewritten to better resemble answers. For instance, given an example question *What imaginary line is halfway between the North and South Poles?*, the system tries to find an answer on the form of e.g. *The equator is halfway between the North and South Poles*.

2. *Answer mining.* This provides a solution when query rewrites don't yield any results. The answer can then be guessed by mining all candidates of the expected data type occurring near the question keywords and choosing the most frequent one. This completely ignores any syntactic relations between the words.

Though the solution is fairly simplistic, this is also the beauty of the approach, as it's very easy to implement. The biggest drawback is that it relies almost exclusively on large amounts of data to function properly.

It's also interesting to note that AskMSR does not apply any form of part-of-speech (POS) tagging, further emphasizing the straightforwardness of the approach. Word classes and morphology are instead decided using plain dictionaries, thereby inducing the traditional risk of ambiguities.

#### Performance

In [15] the applicability and performance of AskMSR is measured on the small corpus of the TREC (section 3.4.1) collection. Not only is this an interesting analysis, as the system is mainly tuned to exploit large amounts of data, but it's also a necessity. Without some common, controlled corpus there is no way to compare the performance of the various contending approaches.

That said, AskMSR appears to fare reasonably well in the small TREC collection, performing slightly below average in the TREC 2001 QA track [42]. This is nevertheless a good achievement given the simple inner workings of the system. The team argues that their system parameters could also be tuned to work better in this kind of environment, but the accuracy of the system will in any case be far below the theoretical estimate given the mismatch between system and environment.

The team notes that their approach seems least applicable in applications that involve proprietary data. In these cases, they regard their redundancy method as inappropriate. Instead, sophisticated analysis is identified as necessary to map user queries to the lexical surface forms that occur in the text collection.

### 3.5.3 Aranea

Search engines provide neat access to vast amounts of unstructured data. QA systems using these engines as a foundation can leverage the massive amounts of redundancy on the web to answer many types of questions, even uncommon ones, and use quantity to somewhat make up for the lack of quality. One system built to exploit the web is *Aranea*.

Aranea [29] is a factoid QA system based on heuristics. It was first submitted to TREC 2002 by a team from the MIT Artificial Intelligence Laboratory. In many ways it is the next-generation iteration of AskMSR, detailed above, and the earlier START system [25]. As a result, the team has consisted of the original creators of both of these systems at various times during its iterations.

The system uses the web as a source. Here, two components run in parallel to find answers:

- The *knowledge mining* component uses a traditional search engine to locate useful information, which is then mined for answers.
- The *knowledge annotation* component simulates a virtual database and queries several knowledge repositories on the web.

The team's opinion is that these two approaches are complementary, as their strengths lie in different types of questions. Each component will now be described in turn.

### Knowledge mining

The knowledge mining component has a serialized data flow through eight modules:

1. Questions are translated to queries. Two types of queries are built: inexact and exact. These differ in that inexact queries are simply bags of words while exact queries are rewritten to match the syntactic patterns of likely answers.
2. The queries are executed using Google as the search engine. Whole summaries are extracted for inexact queries, while for exact queries an answer is only extracted if it fulfills predefined positional constraints. Answers returned by exact queries are also given preference.
3. N-grams (where $1 \leq n \leq 4$) are generated exhaustively from the returned text, and these serve as candidate answers. In other words, all combinations of up to 4 following words are regarded as potential answers, sidestepping the need for complex parsing.
4. The n-grams vote for each other so that frequently occurring answers are given more weight (leveraging redundancy).
5. Heuristic filters are applied to discard poor candidates. For instance, answers containing words from the question are disregarded (as they needlessly repeat the question), and answer types are checked against fixed lists to make sure they're of the right type.
6. Longer answers subsuming shorter ones are given more weight because they contain more detail. This is in line with the general scoring scheme of TREC QA.

7. Candidates are scored using a formula that balances out prior likelihoods of individual keywords.

8. All answers are verified against the web to counter any artifacts generated by any of the modules.

### Knowledge annotation

Given the lack of structure on the web, similar components based on search engines are implemented in several web QA systems. The team from MIT went one step further though. They've developed a technique they call knowledge annotation that takes advantage of the structure that does exist on the web. The foundations of this concept were lain in [24] as early as 1988 and pioneered by the START web QA system [25] in 1997.

Central to their idea is the fact that amongst the hyperlinked, unorganized tangle of the web there are pockets of structured knowledge. These sites provide solid information within their chosen domains of specialization. Examples of such sites are Wikipedia [61], the Internet Movie Database [57], the CIA World Factbook [50], Dictionary.com [51] etc. Each of these sites has its own means of access, thereby often rendering the knowledge contained within inaccessible to search engine crawlers.

Further, this knowledge resource as a whole can't be tapped effectively until its components are uniformly accessible. Fortunately, this problem of integrating heterogeneous sources has already been dealt with successfully within the domain of databases. The concept of a federated database system is ideally suited in this case, and techniques developed within this domain for managing semistructured data can be borrowed advantageously.

Federation has several benefits: Queries can be delegated to specialized and trustworthy sources, thereby assuring answer quality. The information is more up-to-date than that provided by web crawling. Answers to definition questions are formulated concisely and to-the-point. Detailed answers are easier to acquire than with knowledge mining, as the latter approach would have to rely on discourse processing and multi-document summarization.

Federation doesn't happen by itself though. The big challenge is the manual labor required to make it work. As each site provides a custom method of access to its resources, individual wrappers have to be crafted for each specific site. Although seemingly a daunting task, the team mentions promising existing solutions in the form of helpful authoring tools and machine learning techniques that can automate wrapper generation.

Another issue with annotation is that the knowledge available for querying is limited by the domains covered by the sources. However, through analyzing

the test sets of questions used in developing and evaluating existing QA systems, the team discovered an interesting property: many similar questions occurred frequently. This made them well suited for translation to database queries.

To accomplish this translation the team uses schemas that map the semantics of questions into their respective database terms. This is possible due to the existence of semantic parsers with very high degrees of accuracy, e.g. [20]. This discovered pattern of user querying is helpful in that relatively few schemas are needed in order to cover most users' questions. The rest can be answered by the knowledge mining component.

### Answer boosting

The results from both the annotation and the mining components are channeled into an *answer boosting module*. Here answers are processed heuristically based on the type of questions they answer, and boosted (i.e. given more weight) if identified as likely candidates. This module is more extensive than the previous filtering module of the mining component, and recognizes complex entities like geographic locations (e.g. *on the north shore of Lake Ontario*).

### Performance

Aranea has continuously run against the other TREC systems since 2002 and has generally performed slightly above average ([43], [44], [45], [46]).

The system was built to find answers on the web, though, and TREC demands that answers should be located within the test corpus. Aranea thus found correct answers that were nevertheless invalidated resulting in a lower overall score. This demonstrates the potential difficulty in evaluating an open system with a closed data set. Because the team judged these shackles unfair, they also presented their own results with significantly higher scores.

Analysis of the components showed that annotation yielded much higher accuracy, while mining yielded much broader coverage. The manual effort involved in annotation was further regarded small given its good performance. The team is especially proud of their annotation component. While attempts have been made before to unify heterogeneous web sources, these have usually required specifying queries in some formal language like SQL. A solution based on natural language promises to be far more accessible, and the team considers their research in this direction as their unique contributions to the community.

Katz et al. developed an extension [26] to Aranea, in accordance with

TREC 2003, that also tries to handle definition and list questions. This extension works in a similar sequential manner to the knowledge mining component, but will not be explored here.

### 3.5.4 PowerAnswer

Language Computer Corporation (LCC) [31] demonstrate the feasibility of deep processing within open domains. Their methods are based on reasoning over general knowledge of concepts rather than on hand-writing rules.

*PowerAnswer*, their commercial QA system, has been the undisputed leader of factoid answers in the TREC QA track for many years. It has consistently performed far beyond most of the other participants' systems in this area ([43], [44], [45], [46]). To be able to reach this level of accuracy their system utilizes a large arsenal of NLP techniques, including entity extraction, part-of-speech (POS) tagging, deep syntactic parsing, word sense disambiguation and semantic lexicons.

The core of their system though — the very distinguishing factor of their approach that has contributed significantly to their success — is the *automated reasoning component*. This component essentially tries to establish the meaning contained in natural language sentences. By doing so, it can *infer* whether answers corresponding to the concepts of the question. This feat is accomplished by the three following modules:

- The *Logic Form Transformer* transforms the syntactic parse tree into a logical form that can be automatically reasoned on and subsequently used to identify relations between the terms in the sentence. While the English language is too complex to cover all syntactic rules, adequate performance is reached by focusing on the 10% of the rules being used 90% of the time, as identified by the team.

- The *Lexical Chainer* aims to solve the problem of different terms being used to describe the same concept, e.g. "to die" versus "to pass away". If the question uses one term and the answer the other, the lexical link between them is obscured. It can be recovered using a manually annotated and machine readable dictionary like eXtended WordNet (XWN) [52]. Here *chains* of semantically related concepts can be followed that provide the missing links between two different, but conceptually similar, words. By resolving these chains of *synsets* (synonym sets), the correspondence between questions and answers can be established even though they have few terms in common.

- The *Logic Prover* feeds the output from these two previous steps into a proof engine based on inference rules. This engine can deduce that a statement is a semantically correct answer to a given question even though the statement is phrased completely different from the question.

LCC are convinced that deep semantic analysis and logical reasoning are requirements for advanced QA. They plan on building upon that foundation by enhancing their system to cope with more complex questions than mere factoids. They also mention the desire in the future to handle implicit knowledge in this same manner; basically by reasoning out a fact that is not stated explicitly in the text, but follows logically from other statements. This is an area which has been practically untouched since the advent of AI. While that remains to be seen, their approach is unquestionably powerful.

Still, the amount of work that has gone into building PowerAnswer is daunting, and the system is critically reliant upon encoded knowledge of the world. These factors limit the portability and applicability of the system to languages, not to mention topics, carefully prepared for machine consumption. Unless methods are devised to automatically enrich these kinds of knowledge bases, this kind of system is bound to demand continuous human resources to keep up with the evolution of human languages.

### 3.5.5 Beyond factoids

Heuristics have proven very effective for answering factoid questions, and have been used with great success by many systems in the TREC QA tracks. Still, many argue that factoid questions are rarely what users ask, and that these kinds of answers are relatively easy to find anyway using a search engine or an encyclopedia.

Research is thus starting to move beyond factoids [44]. But as the complexity of the answers increase, so does the heuristic requirements. Many more rules have to be crafted to cover all the new possibilities, and the application of heuristics is starting to look a lot less attractive. Still, many systems competing in TREC are building heuristic-based layers on top of their factoid systems to answer tougher questions.

Soricut & Brill [41] tried another solution. Complex questions (e.g. *How does a movie qualify for an Academy Award?*) can't be answered with entities, nor simple definition lookups, but require (often long) explanations. Reformulating the question to resemble answers is thus pretty much a dead end, and indeed the team experienced that this technique often worsened performance rather than increasing it.

Further, the task of detecting question types was identified as overwhelmingly more complex than for factoid questions, and this idea was also abandoned. The team instead decided upon a solution motivated by statistics. The idea was to identify answers by their similarity to answers in a training corpus, given a comparable question.

For this idea to work they had to accumulate massive amounts of question/answer pairs somehow. Like others before them they turned to FAQs for this data. But unlike previous attempts, which were based on FAQ collections, they instead harvested the web directly, searching for pages containing the string "faq" in the URL. By identifying question/answer pairs using simple lexical cues (high recall), and filtering out noise afterward (high precision), they were able to harvest no less than 1 million question/answer pairs. This yielded a corpus orders of magnitude larger than previous attempts, like the 30,000 pairs of Tritus [1].

With the training data collected, they turned to query formulation using two search engines, MSNSearch and Google, as the foundation. Instead of reformulating questions they applied *statistical chunking*. Basically this means dividing the questions into chunks (or phrases) based on phrases learned from answers in the training corpus. The idea is that these phrases will give better results than keywords. A nice side effect is that chunking is *language independent*, given a training set of the desired language, as opposed to question reformulation.

Answers were extracted using two techniques: n-grams and a *noisy channel model*. The latter model assumes that an answer has been "corrupted" by noise into a question, and hence tries to rediscover the answer. In practice, this noise is due to a likely answer being generated and then transformed into a question. The system then estimates the probability of this question given the answer. With this "inverse" probability, Bayes' law can be used to discover the probability of an answer given this question. Actual answers can then be extracted with high probabilities of matching the questions.

When testing the system, the team noted that chunking resulted in significantly better response than plain keywords, as expected. More importantly, about 40% of questions asked returned satisfying answers. As a first attempt at an open-domain, language-independent QA system without question-type restrictions, this is a promising achievement. The downside is the massive amounts of QA pairs that need to be collected and fed to the system during training to increase the chances of question matching.

The team identified several areas in their system that could be improved, most notably the construction of a question typology targeted at FAQs. But it is not clear from the text how this typology might be crafted without simultaneously abandoning language independence, thereby eliminating one

of the major advantages of this approach in the first place.

### 3.5.6 Fact mining

Most big, commercial actors have traditionally focused on manual or semi-automated QA, as shown in section 3.2. Presumably because these solutions have been easier to implement and to determine any direct value from. Things are beginning to change, however. Current commercial search engines are gradually starting to move beyond merely providing increasingly refined IR functionality. They're presently being enhanced with a form of implicit fact lookup. While not strictly QA, this form of automation can answer a limited set of inquiries.

The technique is based on identifying certain limited patterns in queries which indicate that the users want to find some kind of factoid. For instance, the query *population of Japan* can easily be recognized as asking about the population count of a nation. If entered into Google, the resulting first hit is in the form of a short measurement factoid:

(3.13) **Japan Population**: 127,417,244
According to
http://www.cia.gov/cia/publications/factbook/rankorder/2119rank.html

Google is now also able to answer simple definition questions, such as *Who is Abraham Lincoln?*. This query results in the following topmost hit:

(3.14) **Abraham Lincoln** ... sometimes called Abe **Lincoln** and nicknamed Honest Abe, the Rail Splitter, and the Great Emancipator, ...
According to http://en.wikipedia.org/wiki/**Abraham Lincoln**

Note that such queries must be entered at the official site [54], not one of the many internationalized versions, as this feature has yet to be implemented beyond the official one.

Google calls their feature *Google Q&A*. Yahoo!, MSN and Ask have their own competing services and call these *Yahoo! Shortcuts*, *Encarta Answers* and just plain *Ask*. MSN seems to prefer their own favorite encyclopedia (Encarta) for all answers, while Yahoo has a tendency to consult Wikipedia frequently. Ask seems to look up the information in their own knowledge base, but provides links to supporting sites. Google apparently tries to route all questions to the most appropriate source on the web. The encyclopedia-centric approach (e.g. MSN) is probably superior on general coverage, since questions can be answered without having to be tuned to a separate source. But the web-centric approach (e.g. Google) is likely better on specialized

and fresh information. It's also more flexible and trustworthy since it's using multiple sources.

I chose Google's solution for further inspection. Here, the answer type decides which source a fact is extracted from. For instance, the question above about population counts (example 3.13) is mapped to the CIA World Factbook [50]. Answers are identified by the specific structure in which they occur on the target pages. For instance, the population count is listed in a table with the unambiguous syntax *Population: 127,463,611*. Answer extraction is thus only a matter of creating a simple wrapper that targets this specific syntax on that specific page. This approach of hand-coding wrappers to specific sites can be seen as a form of knowledge annotation, as utilized by Aranea (section 3.5.3).

Of course, in all of the above-mentioned services the fact lookup is likely based on previously extracted and compiled knowledge rather than live web pages. Each vendor bases its answers on trusted sources rather than general web pages. These approaches work great for well-established facts, as such inquiries are adequately answered with just one source. In this manner the services function as a kind of automated encyclopedias. This saves users the trouble of seeking out appropriate resources and looking up the knowledge themselves. But the solution severely limits the answering ability for disputed factoids where it's nice to see several sources. The answers are also limited to the concepts recognized by the system and thus wrapped to the format of a corresponding source.

These solutions are effective for many simple facts, but can't really be called QA systems. They're not open-domain, as they're limited to the specific concepts recognized by the implementors. And they're mostly restricted to measurements and definitions. Further, the answers are not discovered automatically. Someone has to create a wrapper for each specific answer type. In this manner, the solutions resemble enhanced Q&A services (section 3.2) more than true QA. They are thus specialized, and fairly limited, fact enhancements to existing engines rather than general purpose QA.

## 3.6 Semantic verification of answers

No matter which techniques a particular QA system employs, the end result is usually an entity of some sort believed to constitute an answer of the right type. For instance, take the answer to a question like "What mineral helps prevent osteoporosis?". Let's say the system has been lucky and found the corresponding string "calcium helps prevent osteoporosis". The word "calcium" is picked as a likely candidate for an answer. But how can the

system know that this is actually a mineral? That is what the question asks for, after all. Likewise, how can the system know that "regular exercise" from the string "regular exercise helps prevent osteoporosis" is not a mineral?

Figuring this out is often called *semantic verification*. When utilized, this is usually implemented as an additional step performed after extraction, with the purpose of validating the semantic class of the extracted entity.

In the early TREC QA tracks, few systems employed this kind of verification. Initially, it was only mentioned in the list of future enhancements of far-sighted participants. The first crude implementations were added mostly as after-thoughts. During later tracks, however, the notion became increasingly popular. Today, in the QA field as a whole, many view it as a crucial step in assuring high precision answers. After all, finding a nice-looking answer is one thing. Verifying that it is of the right type is quite another.

Verification is especially important for shallow systems, which are primarily based on surface patterns or statistical processing. These might locate entities in any number of ways, none of which are likely to rely on syntactic parsing to any significant degree. As such, they often can't base their verification on the syntactic structure of the answer. In some cases, this kind of structure is adequate if the answer explicitly mentions the entity type. For instance, take the sentence "H. P. Lovecraft, the famous author of horror fiction, invented the Cthulhu mythos". The right syntactic rule can easily relate "H. P. Lovecraft" to the defining noun "author", thus answering a question like "Which author invented the Cthulhu mythos?" in one step. Shallow systems, on the other hand, might only identify that the proper noun phrase "H. P Lovecraft" occurs near "Cthulhu mythos". Or even that they frequently appear together in a corpus. In such cases a separate step is needed to verify the entity type.

Determining the semantic class of a word isn't trivial. Words don't carry any semantic information by themselves, and only get their meaning from the observer's learned associations. A machine doesn't have this kind of knowledge. The only apparent way to automate the process is thus to look up each word and verify it. Basically, by validating each candidate against a list of phrases known to constitute instances of the particular semantic class. The key to semantic discovery is thus having access to a thorough semantic lexicon.

### 3.6.1 Semantic lexicons

Of course, machine-readable dictionaries (e.g. WordNet [62]) were created largely for the purpose of providing this kind of semantically tagged information. A lot of QA systems have therefore turned to them for verification

purposes. But by being manually constructed they are rather limited in coverage.

There are major advantages of analyzing large amounts of texts compared to simply looking up relations in a machine-readable dictionary such as WordNet [62]:

- Manual ontologies contain many rare and misleading senses due to a goal of being as thorough as possible. E.g. "to die" is arguably more often used in the sense of ceasing to exist than in the sense of using a die (i.e. dice) as a tool. Manual ontologies treat all senses equally but automation prefers the most frequent ones.
- Manual ontologies lack many domain-specific terms since every domain can't be covered thoroughly. Automation can be tailored to a specific domain.
- Automation allows discovering interesting relations from the same documents that will later be queried upon.
- Automation can cover far more material and more variations. The manual dictionaries are mostly limited to common nouns.
- Automation can discover unique, special or up-to-date concepts highly unlikely to occur in manually prepared dictionaries.
- Automation allows loose language dependence. It's far easier to rewrite and tune the patterns to support a new language than to manually assemble a thorough dictionary with the right relations. Further, the chances of locating a solid and specially prepared dictionary in a given language is slim, as these are not exactly abundant.

It would therefore be far more future-proof, not to mention interesting, to find a way to discover these semantic classes automatically.

There are two general paths to automating the assemblage of a semantic lexicon:

1. The bottom-up approach tries to establish semantic relations directly by looking at the syntactic structure of sentences. Common patterns are searched for that imply trusted relations between semantic classes and its members. As a consequence, each match is essentially treated separately from the others.

2. The top-down approach tries to look at the entire text collection as a whole. It focuses on discovering complete semantic classes rather than separate relations. This is accomplished by comparing the usage patterns and contextual cues of all candidate words. The thoroughness of this process naturally makes it very demanding. It is typically based on clustering or latent semantics. That is, by establish relations between words based on their mutual co-occurrences with other words.

The goals of both of these approaches is similar but the method is entirely different. The main strength of the former approach lies in the accuracy of the patterns. Because of the very specific syntax used to describe these relations in natural language, only one instance has to be found to establish fair certainty of a valid relationship. On the other hand, a large amount text is needed to find enough of these trusted relations. The latter approach, on the other hand, can find some kind of relation with relatively little text. But this relation is nowhere near safe unless verified by a large amount of similar occurrences.

I only had time to properly explore one of the approaches and settled on the bottom-up one. I chose it because it's better suited to integration with the large-scale pattern matching allowed by search engines. The other path required a whole different set of tools than what I had available. For details of a very promising top-down solution, see [32] and [33]. A presentation of a bottom-up solution now follows.

### 3.6.2 Lexico-syntactic patterns

Researchers in computational linguistics have long since discovered [2] that pattern matching can outperform parsing both in accuracy and efficiency for the purpose of finding simple semantic relations. A natural consequence of this discovery was trying to determine what kind of patterns might be useful in extracting candidate phrases for building semantic lexicons.

Hearst [19] is often cited as a pioneer in this bottom-up, pattern-based approach to semantic discovery. She identified the following properties as crucial for any generally useful *lexico-syntactic patterns*:

- They must occur frequently and in many different kinds of text material. Otherwise they won't result in enough instances.
- They must always indicate the correct relation of interest (or at the least; fail to do so very rarely). Erroneous relations lead to erroneous systems.

- They must be recognizable with little or no pre-encoded knowledge. Their very purpose is in discovering this kind of knowledge, not relying on other tools to provide it for them.

By using a method detailed in her paper, she manually identified 6 patterns fulfilling these requirements. My version of the patterns is detailed in my implementation in section 4.7.3. However, she saw no apparent way to automate the pattern discovery process as it depends on a good portion of intuition and semantic understanding.

All of her patterns are based on relations between hypernyms and hyponyms. Essentially, a *hypernym* is a word that is more generic than a given word. A *hyponym*, in contrast, is more specific. For instance, "vehicle" is a hypernym of the hyponym "car". That is, a car is a vehicle, but a vehicle is not necessarily a car. Similarly, "Abraham Lincoln" is a hyponym of the hypernym "president of the United States".

According to her evaluations, her patterns turned out to have high precision but low recall. Other more insecure patterns could be added to increase recall, but she does not recommend gambling on these. Instead she proposes using only the (likely few) patterns proven to be secure and increase the size of the corpus. In my opinion this principle corresponds nicely with large-scale pattern matching (section 3.3.5), and suggests a natural synergy with these kinds of systems.

The value of her approach has been confirmed by many parties, recently including Snow et al. [40]. They originally intended to explore a way to automate hyponym classification based on learning hyponym patterns from WordNet. Here, lexico-syntactic patterns are represented as dependency paths, i.e. syntactic relations between two words. They used Hearst's patterns as a frame of reference. After thorough testing, they basically ended up with the same patterns. A few others were discovered but these were of considerably lower precision. It was confirmed that all of Hearst's patterns were near optimal in a balance of precision and recall. They thus concluded that hand-selected patterns are at least as suited for building lexicons as automatically discovered ones.

### Additional queries

Another pattern-based verification scheme is also worth mentioning. Rather than building a lexicon it is based on verifying answers on the fly. Specifically, [27] tried using a separate query, generated from the question, to find semantic support for the answer. For example, given a question like *What type of currency is used in Australia?* they generated dual queries with one

looking for answers, e.g. *X is used in Australia*, and the other verifying the answer, e.g. *X is a type of currency*. Only returned entities matching both patterns were regarded as verified.

This kind of solution tends to be precise but also has several limitations:

- It has low recall due to its very specific reuse of syntactic structures from the question, rather than the above-mentioned, common, lexico-syntactic style of patterns. Of course, if the latter were used then the query processing time would likely increase with these high-frequency, general matches. Also, the precision would be expected to decline with the introduction of false positives. Only simple syntactic rewrites are affordable as new queries.

- Query-time verification generally limits coverage to the same corpus as the answer. That is, the corpus must not only contain the answer but also some kind of explicit type definition stating that the answer entity is of the right kind. A corpus likely to contain answers might not necessarily be likely to contain definitions, and vice versa.

- Running on-the-fly semantic queries is costly as it requires running a whole new set of queries just to verify the answer. The response time of the system degenerates accordingly.

## 3.7 Discussion

This chapter has presented Question Answering in light of related areas of science, and has explored some of the recent research within this field. While much has been accomplished, QA is still in its early stages.

Current systems are very good at handling relatively straight questions (like factoids), but struggle with more complex ones, like definition and list questions [44]. Some systems (section 3.5.5), try to circumvent language complexity by relying on statistics instead. They have yet to reach performance near factoid levels. For answering capability, nothing currently beats deep processing (section 3.5.4) although pattern matching solutions are maturing and gradually catching up (section 3.5.3).

While factoid answering has its uses, it might also be regarded as somewhat redundant. Keyword-based search engines and encyclopedias handle these types of questions adequately, and the advantages gained by pure factoid answering are thus limited. Still, the major search engines are presently being enhanced with implicit fact lookups (section 3.5.6) because the companies see value in the efficient answers this functionality allows.

The ability to respond with multiple answers on a given question could prove to be a valuable addition to the factoid answering paradigm. Especially if the list was ranked according to confidence in each answer, and support for each answer was easily accessible in the text. Users can quickly locate well-known facts by consulting the appropriate knowledge bases, but it's much more difficult finding reliable statements on current events, disputed factoids or novel topics. There is thus considerable benefit in a system that can gather all answers to any given subject and let the user survey them, even if these answers are mere factoids. This functionality goes beyond what current fact lookup services can provide, since these are tuned to established facts and singular sources.

Further, the value of QA would be much increased if it could handle complex questions that are not readily answerable by lookups. Two notable examples are explanations and opinions. Systems handling these kinds of questions are necessarily significantly more difficult to implement. Good explanations demand solid discourse processing and might even require some kind of dialogue to match the advantages of asking a human expert. Opinion recognition demands some model of human emotions, and the ability to separate factual statements from irony as well as to recognize metaphors, idioms and other subtle manners of speech. However, the implementations don't have to actually understand the answers as long as they can at least identify the type of answer. It's better to leave thorough semantic analysis to the human readers and concentrate on aiding them in that task rather than trying to outperform them.

The power of humans to answer questions, especially complex ones, should not be underestimated. A response will take a lot longer but will also be much more thorough when provided by an expert rather than a machine. Unfortunately, there's little incentive in answering difficult questions on an individual basis unless there's some potential for profit. Many people seem willing to pay for this kind of service, however. Still, there's no reason why humans should be bothered with researching simpler questions when a machine could be tuned to perform at an equal level. Both the semi-automated and the fully automated solutions have their place in the market.

All things considered, QA systems have the potential to provide a clean and intuitive interface to several different sources of information, whether they're located on the web, in the users' own file systems, in databases or even in specialized applications such as topic maps. Currently these technologies, to be effective, require the users to learn both a formal query language and an ontology to get to grips with the complicated interface. Or they must provide several iterations of feedback to disambiguate and narrow down the search. A QA interface could greatly improve the usability of these applications as

well as significantly lower the entry requirements. QA could be the key to providing a uniform point of access to these combined resources, tying them together transparently.

# Chapter 4

# Implementation of a QA prototype

## 4.1 Introduction

When describing my initial motivation for this thesis (section 1.2), I stated that my overall goal was to research the foundations for a QA system and start work on implementing a prototype. The previous chapters have presented and discussed crucial technologies required for a working system. They have also listed important considerations that have to be made when designing the system. It is now time to present my implementation and the features I targetted.

This chapter opens with a brief overview of the design goals for my system (4.1.1). I then introduce my initial analysis of the specific functionality required and the potential benefits of this design (4.2). This section also outlines the system components and how they interact (4.2.2). Next, I present my procedure for building entity extractors and give examples of a few extractors (4.3). This is followed by a thorough analysis of questions and a detailed presentation of the patterns I use to classify these (4.4). I further explain how to transform these questions to queries (4.5) and how to extract answers from the results (4.6). Finally, I present my views on semantic verification (4.7) and the extractor I built for compiling a semantic lexicon (4.7.3). The chapter ends with a short list of potential future enhancements (4.8).

### 4.1.1 Goals and achievements

I wanted to base my solution on the general architecture of a pattern-based QA system because this model is best suited to integrate with a search engine. My implementation would be tailored to the specialized contextual scope

search paradigm of Fast ESP (section 2.6.2). The evolution of the TREC QA tracks were used as a general guideline for features. The system was intended to provide functionality comparable to the solutions of the first conferences. With this foundation, advanced functionality from later conferences could be progressively added in future upgrades.

QA systems based on pattern matching have a fairly solid track record. These systems have already performed admirably (section 3.5), as measured by conferences like TREC. But I had no intention of competing directly against these systems:

1. They were created by experienced teams through several iterations. An initial one-man effort couldn't reasonably be expected to cover the same kind of functionality within the allotted time frame.

2. I would not have time to build a fully functional system. At best, I hoped to be able to present simple answers to a limited set of questions. My priorities lay with researching and presenting my solution properly, not with implementing all the various details required for successfull performance.

Still, by using Fast ESP with its extractors as a basis, much of the necessary ground work had already been completed. I already had access to a full-fledged search engine with several available entity extractors that could be used for answer extraction. Also, by integrating the QA component directly with the engine, I could avoid the need for writing wrappers and intermediary code. This would have been required if using a separate engine through a general API.

These are the specific design goals I set for my system:

- *Pure pattern matching approach.* I did not want to rely on syntactic parsing or part-of-speech (POS) tagging. I wanted a simple grammar that only had to be accurate enough to classify questions.

- *Open-domain questions.* My system should be able to respond to questions on any topic. I saw no benefit in limiting the questions to certain domains. Rather, my intent was to demonstrate the extent of questions that could be covered by simple patterns.

- *Loose language dependence.* I wanted a design based on simple heuristics with patterns that could easily be converted to other languages. I also aimed for automated ways to generate dictionaries by using a process that could be replicated across languages.

- *Semantic verification.* Creating a basis for semantics was a crucial component in my intended system all along. I did not target extensive semantic support, but wanted to show that patterns and dictionaries could provide a basic foundation.

- *Multiple answers.* I wanted a design for providing multiple answers, ranked by confidence in each answer. These should also provide necessary context to help users evaluate and consider the answers.

- *Flexible scale.* The system should be equally adept at handling QA on a large scale (e.g. web) as on a smaller, local scale (e.g. selective indexing). This required balancing precision and recall in the patterns.

As will become clear throughout this chapter, all of the above goals have been successfully met.

## 4.2 Preliminary analysis

To be able to realize my QA prototype I first had to accomplish two preliminary steps:

1. I had to analyze questions and answers to see what was actually processable. Without some general idea of what could be accomplished I would not be able to devise any specific approach.

2. I had to figure out which components I needed and how to interface them with the underlying search engine. This meant accounting for both the requirements of the dimensions I would focus on and the possibilities allowed by Fast ESP.

### 4.2.1 General approach

I started examining the questions from the TREC QA tracks and figuring out which steps were needed for proper answer extraction. I was particularly careful to pick steps that could be made fully machine automated. That is, steps that would not require human intervention but would rather be reproducible by algorithms. Specifically, I targeted steps that corresponded with the processing pipeline of Fast ESP. As it turns out, the general architecture of a QA system is well suited for implementation in Fast ESP. Note that the following overview is very abstract. Each step will be presented and discussed in detail in its respective section later. This is the general approach:

1. Classify the question. That is, determine answer type, word classes and other elements that might be useful for the particular question.

2. Build and run a scope query for the general type of entity the question asks for. In the query, include any elements from the question that might improve the search.

3. Extract entities of the right general type from the result set.

4. Verify that the answer entities appear in a syntactic relation that corresponds with the question. Also verify semantically that the entities are of the right, specific type.

To clarify, consider the sample question *What French ruler was defeated at the battle of Waterloo?*. This question would be processed in the following manner:

1. The answer type is determined as "French ruler". The noun "ruler" is recognized as describing a person. This is reinforced by the verb "defeated" which implies a conscious entity. The verb phrase "was defeated" is further recognized as passive, meaning the person in question will be the syntactic object in the answer – the one receiving the defeat, not dealing it.

2. A scope query for a person is built which includes the verb "defeated" and the nouns "battle" and "Waterloo". The answer type term ("French ruler") is not included as there's no guarantee that the answer entity will be described as such in the answer. The noun "defeat" is derived from the verb and included as an alternative to the verb. The resulting query is *and(battle, Waterloo, or(defeated, defeat), scope(person))*.

3. The extracted answer entities (i.e. person names) are verified to be syntactic objects in accordance with the verb. Negations (e.g. "not defeated") and speculations ("if defeated") are also filtered out. Examples of valid relations are given in example 4.1 below and invalid relations in example 4.2.

4. Finally, the extracted answer entities are tried verified semantically. That is, by finding some link between an entity and the term "French ruler", e.g. from a sentence such as *Napoleon is recognized as the ruler of France* or *Napoleon Bonaparte, the former French ruler*.

(4.1) *French emperor Napoleon was defeated*
*X was defeated*

*Napoleon is defeated at the Battle of Waterloo*
*X is defeated*

*Napoleon Bonaparte, defeated then and even more decisively*
*X, defeated*

*defeated Napoleon at the Battle of Waterloo*
*defeated X*

*Napoleon Bonaparte received a crushing military defeat*
*X received ... defeat*

*Napoleon's final defeat*
*X's ... defeat*

(4.2) *he was defeated again by Wellington*
*defeated ... by X*

*defeat at Waterloo against an allied force under Wellington*
*defeat ... under X*

*Henry V decisively defeated a much larger French army*
*X ... defeated*

*in case Wellington were defeated*
*in case X were defeated*

In order to realize this process in full, the following components are needed:

- Question classification patterns that are able to determine answer type, POS of words, active and passive use of verbs and more.
- Entity extractors that can recognize the most important, general answer types that are needed to reach sufficient open-domain coverage. At the very least persons, dates and locations, as these represent some of the most frequently asked about entities in the TREC QA test set. They're also the only types with their own interrogative words; who, when and where.

- Dictionaries that can map an answer type to an appropriate entity extractor. That is, patterns able to identify that e.g. "ruler" and "emperor" refers to a person and "city" to a location.

- A dictionary of verbs that denote action, e.g. "defeated", "fought", "gave" and "elected". This dictionary can be used to disambiguate nouns like "ruler", but might also be useful in separating conscious from unconscious entities based on the verb.

- A dictionary that can translate verbs like "defeated" to nouns like "defeat" and vice versa. This will be useful in enhancing the query since these two forms are often used interchangeably, e.g. "Elvis died" vs "Elvis' death".

- Relation verification patterns that are able to separate syntactic subjects from syntactic objects. Preferably also able to recognize negation, speculation and more.

- Semantic verification patterns that can establish that an entity belongs to a desired class. Possibly indirectly through synonyms. That is, they should recognize both "is a"-relations like "Napoleon is an emperor" and "kind of"-relations like "an emperor is a kind of ruler".

### 4.2.2 System outline

My QA system interfaces with Fast ESP in the following ways:

- *Question processing.* When a query is submitted to the Fast ESP query interface, this query can be intercepted and modified before being sent to retrieval. This step in the ESP engine is called Query Transformation (QT) and it's the point where my classification patterns interface with the engine. That is, I take the original query (i.e. question) as input, classify the question, generate a scope query for the right entity and resubmit the query. These classification patterns are detailed in section 4.4. A custom ESP module handles question normalization, term expansion and scope query generation.

- *Document retrieval.* Once the scope query is substituted for the original question, document retrieval ensues. The search engine is responsible for finding text that fulfills the requirements of the query and rank it according to a best-fit measure. This is where my implementation stops. Due to time restrictions, I've had to focus on question classification, not answer extraction. I simply take the returned results and

present them in their raw format. The rest of the steps illustrate how my system was intended to proceed from here.

- *Answer processing.* The retrieved results are formatted in XML. For each entry there's an XML tree that contains both the retrieved text and a lot of meta-data tags. These represent all the interesting information (e.g. structure) identified in the text, including entities. The entities are thus readily available for further processing. Basically, any passage containing both an entity of the right type and the keywords from the question would constitute a potential answer. To verify the results before they're presented to the user, two custom steps would be interfaced at the document retrieval point in the ESP engine. A set of patterns would verify the syntactic relation between the candidate answer entity and the verb. These patterns would again be implemented in the Matcher framework. An additional module would verify the semantic class of the candidate answer by verifying it against a semantic lexicon. I have implemented an automated way to build such a lexicon although I will not be able to demonstrate it in answer processing.

- *Result presentation.* The answers would be presented in a user-friendly manner. Specifically, they would be presented as a list of entities ranked by probability of being correct. This probability would be calculated based both on frequency of occurrence in the retrieved documents, and on the semantic correspondence between the extracted answer entity and the answer type identified in the question. This list would, in fact, be an ESP *navigator*. This is a component which allows *drill-down* into the result set. In short, this entails being able to click on an answer entity in the list and see all the contexts in which the entity appears. This provides the necessary support for the entity that the user needs in assessing the credibility of the answer.

### 4.2.3 System dimensions

To explain my system in light of the dimensions identified in section 3.3, these are the main attributes of my solution:

*Domain.* The system was designed to be domain-independent in that it will accept questions on any open topic. This was the only sensible choice given the general-purpose nature of the underlying search engine. Open-domain as it may be, however, it will not provide equally good answers on every topic. Thorough answer processing requires support in the form of entity extractors. Those answer types for which there are readily available

extractors will thus be better covered than those that have to be identified by general noun phrase extraction.

*Question type.* TREC, up until 2004, was used as a guideline for the kind of questions the system would support. Almost all of these questions are direct, while a few are commanding. The questions are further single-sentence and contain all necessary information within each sentence. That is, there is no need for anaphora resolution or access to question history to interpret a given question. Each question is thus independent and self-contained. This is the dominating search paradigm today and the one I wanted to support. I deliberately avoided the question sets from 2004 and beyond because they broke these properties and were tailored more towards dialogue systems.

*Answer type.* The supported answer types are also based on TREC. That is, I primarily targeted factoid, closed-class questions with some support for definition and list questions. My system is also able to classify many advanced question types although there is little support for identifying and extracting the corresponding open-class answers. Being able to recognize what kind of answer is required is a solid step in the right direction though. Additionally, the system is able to determine if a question is unanswerable given the offline entity indexing approach of Fast ESP. More on this advantage in section 4.2.5.

*Automation.* Because my system is built on entity extractors and pattern matching it is purely based on heuristics. Rather than attempting to prematurely implement some learning algorithm, I decided instead to apply the skills I had developed from constructing entity extractors. I knew that there would only be a rather limited set of variations I would need to support and wanted full control over the matching process. This approach allowed me to thoroughly analyze questions and answers and figure out the best, unified patterns.

*Scale.* Interfacing with a local IR engine allows full control over the documents to be indexed. In contrast with systems based on web search engines, my system can be easily tailored to provide answers from any selected source. Because it is domain independent, switching sources is only a matter of initiating a new indexing process. This freedom has allowed me to balance scale (i.e. information richness) versus trust (i.e. source reliability). I can thus favor a high quantity of high quality sources like encyclopedias, specialist sites and FAQs.

*Degree of NLP.* My system focuses on shallow approaches because of their speed, simplicity and flexibility. When I started writing extractors in Fast ESP it became clear to me that a vast number of interesting properties could be discovered merely by clever application of rules. I wanted to follow

this line of thought further, and apply the same kind of pattern matching throughout the QA system. It was thus only natural to build upon the valuable techniques I had learned through the extractors.

### 4.2.4 Advantages of pattern matching

The main advantages pattern matching based on modern regexes (as defined in section 2.6.3) has over syntactic parsing are:

- *Speed.* Regex technology is mature and has lightning-fast implementations. Checking for the existence of characters in a string in accordance with various patterns is intuitively much quicker than building a whole parse tree based on the intricacies of human language.

- *Simplicity.* Regexes allow quite complex rules. But their resource requirements are nowhere near the vast demands of parsers. The latter must have access to syntactically sound rules and near-complete dictionaries. Regex implementations are also broad and standardized and easy to acquire. The syntax, while slightly intimidating at first, is nonetheless easy to learn. And in contrast to parsers, their usage doesn't necessitate any previous experience in linguistics.

- *Scalability.* Regexes scale very well with large amounts of data. Whereas the parse trees grow ever more elaborate with the length and number of sentences, regexes just continue checking character by character. Unless the expressions are exceptionally poorly written, performance shouldn't degrade significantly with an increase in data volume.

- *Adaptability.* Due to their simplicity, both in implementation and use, regexes are easy to adapt to new domains and languages. Because they only skim the surface of texts, adapting them is essentially just a matter of adding rules that match the new, valid terms. A parser, in contrast, is so heavily tied to the specific use of language that changing this use, or worse; the whole language, necessitates a complete reworking of the underlying model.

- *Autonomy.* Again, due to their simplicity, regexes are ideal candidates for machine learning and automation. The syntax is so easy, formal and precise that programs can be written without much effort that automatically generate rules to match patterns often occurring in texts. A noteworthy example of implementation is given in [14].

Particularly, the benefits of using pattern matching in my solution are these:

- Low response time is crucial in a commercial search engine. Patterns are processed quickly.
- Atomic tools tend to be generally more useful for multiple purposes. Patterns are more atomic, interchangeable and reusable than parser rules.
- Fast ESP already has a solid framework for entity extraction of which I'm deeply familiar. Parsing support is new and unfamiliar to me.
- Fast ESP has readily available extractors for many entity types, some of which I've written myself. By using the same framework for processing questions and answers as for extracting entities, the same rules and techniques can be shared.
- Keeping all processing within the same framework reduces the risk of a potential mismatch between the separate steps. The number of different technologies that have to cooperate successfully is also minimized.
- An extractor-based QA layer should be easy to integrate directly with Fast ESP. The engine already provides most of the necessary functionality.

### 4.2.5 Comparison with traditional QA

Due to the tight integration of the QA layer with Fast ESP, a system based on this foundation can potentially provide several benefits over traditional QA:

1. *Retrieval performance.* The index can be consulted to restrict retrieval to sentences known to contain an entity of the right type. The number of sentences that have to be checked for answers can thus be greatly reduced.
2. *Answer extraction performance.* The interesting entities have already been identified in the retrieved sentences (by index lookup) and don't need to be processed at query time.
3. *Extraction accuracy.* Index-time entity extraction allows much more advanced and complete methods to be performed than what can be afforded at query time.

4. *Ability to determine unanswerable questions.* All entities of a given type have already been reliably indexed. Thus, if the index does not contain an entity of the required type, or if it does but the entity doesn't appear in the context of any term from the question, an answer is very unlikely to exist in the document collection.

5. *Discovering additional information.* The same entity (or entities of the same type) can easily be located in other sentences by consulting the index. This can be used for providing additional information on the same subject in response to a query. Or even to suggest new queries.

These assumptions are verified by research such as [16]. Of course, successful extraction depends on there being an extractor able to identify instances of the given entity type.

## 4.3 Building entity extractors

The whole foundation for my QA solution is based on the underlying entity extractors of Fast ESP. These are crucial for classifying questions and extracting correct answers. Because of the importance of these extractors, their inner workings will now be presented before going further in explaining my system. I will start with a general overview of the Matcher framework of Fast ESP which is the environment the extractors are written in and run under. Then I will explain the general procedure I've developed for creating extractors efficiently. And finally I will give a few detailed examples of how some of the extractors I've written work.

### 4.3.1 The Matcher framework of Fast ESP

The entity extractors in Fast ESP are implemented through *matchers*. Basically, a matcher is a component that is able to run a pattern on a given text to find all places where the pattern matches. There are several classes of matchers and these use different techniques and levels of sophistication to recognize different types of patterns. The specifics are not important to understand the extractors. Matchers can be utilized both on general texts (e.g. for indexing), on queries and on retrieved results. These last two cases are what my respective question classification and answer verification components target.

For entity extraction purposes, each matcher provides all the necessary foundations for efficiently implementing specific extractors of the given class. Essentially, extractors are specified through a custom XML configuration.

The patterns themselves are PCRE-style regular expressions enhanced with certain functionality to increase expressiveness and ease the task of the implementor. There are many different tags and parameters in this XML-tree, but the most important are the master, the slave and the transformation patterns:

- The *master pattern* represents the full expression and is the pattern that will be used for matching. Since regexes can quickly become long, complicated and hard to read, parts of the expression can be split up into slave patterns.

- *Slave patterns* are typically shorter patterns, each representing some atomic component of text. They can be included at appropriate positions in the master pattern, and can also include each other. They can be thought of as variables in a programming sense, since their contents are inserted at the point where they are addressed. They're addressed by using a dollar sign (*$*), e.g. (*$some_slave*). A further benefit of slave patterns is that they allow sub patterns to be shared instead of replicated. The resulting patterns are far easier to maintain and modify correctly.

- *Transformation* patterns can be used after a match has been made. These patterns describe ways in which the extracted text should be transformed before being returned. The typical use is to perform some kind of normalization. E.g. to standardize times to a 24-hour ISO format.

The extractors typically recognize entities from text in one of the following ways:

- *Structure* inherent in the entity. E.g. a time such as "22:35", which consists of hours (a number between 0 and 23), minutes (a number between 0 and 59) and a colon separator.

- *Contextual information* around the entity. E.g. "7 o'clock" or "director John Adams". Here, "7" is recognized as a time, and either "John Adams" as a person or "director" as a title, depending on which of the entities is used as context for the other.

- *Dictionaries* of known instances of a given entity type. For instance, a dictionary of male person names would recognize "John Adams" as a person without any context.

- *Partial matches* can be expanded to full matches by utilizing previous matches. E.g. if already having matched "John Adams", the mention of "John" or "Adams" in a following sentence is likely a reference to the already recognized entity. These can thus be expanded to the full name.

### 4.3.2 General procedure for building extractors

The following section will describe a general procedure applicable when building a new extractor, as well as detail the concepts behind a few successful extractors. The basic procedure I followed for writing an extractor is this:

1. Gather material that is representative of the kind of texts that will be targeted by the extractor. Focus on these texts rather than some random sampling or much time will likely be wasted on writing unnecessarily complex (and slow) rules describing patterns that rarely occur in typical material anyway. In other words, tailor the extractor to a specific set of texts. The extractor can always be expanded with new rules later.

2. Collect sentences from these texts containing the entities intended to be matched and store them in a common reference file (e.g. plain text or XML) for easy access. It's important to gather whole sentences as the context around the entities often is a helpful indicator of the entities themselves. Try to collect sentences that specify these entities in all the different kind of ways that are interesting to recognize. Also collect a few sentences that don't include these entities, especially if they describe (similar) entities or constructions that are not desirable to match. These "false" cases are useful to check the extractor against to make sure it isn't too general and won't match too much noise.

3. Go through each sentence in the reference file and identify common patterns describing the entities. Write down these patterns (using natural language or (pseudo) regexes) and categorize them by their common features. This is most likely to end up with a few general patterns with lots of minor variations. Try to spot sub-patterns appearing in more than one pattern, factor them out and reference them from each super-pattern. This kind of hierarchical pattern building will make the rules more tidy and will help recognize similarities between the patterns. These similarities could potentially be used to reduce complex, distinct patterns to one general pattern with several variations.

4. Build the patterns step by step, one at a time, and run them frequently on the reference file to see if they actually match the entities they're supposed to and stand clear of everything else. Most likely the extractor won't match every variation of a pattern and will probably pick up a few false positives. This is a trade-off that must be made between coverage (e.g. precision and recall) and complexity (e.g. clarity, expandability and speed). Focus on the most common patterns first and add increasingly infrequent ones. This process eventually comes to a point where a rare pattern would require significant work to recognize for only some small gain. This is the place to stop.

   If it's difficult identifying the common features that constitute a class of entities, try a completely different approach. There are many ways to describe a pattern and the best ones may not be obvious at first. This will become clear through the examples that will follow shortly. It's also an idea to think differently if ending up with far too complicated or performance heavy rules. Simplicity is the key here, and the simplest pattern that will work is likely to be the superior one.

5. Run the extractor on huge amounts of test data and browse through the results to verify the correctness of the rules. Modify and rerun the extractor accordingly until it reaches the desired level of performance. It's useful to develop some kind of framework first for creating and testing the patterns, as this cycle will need to be repeated a lot.

Now for some examples of the extractors I've written by using this approach. I have constructed many more extractors than will be mentioned here, but I feel these are the most prominent and characteristic ones.

### 4.3.3 My entity extractors

#### Time Extractor

Times are specified in many ways, the arguably most common being the digital 24-hour format, e.g. *20:40*. In this format, the time consists of four digits separated by a colon, or sometimes a dot (e.g. *20.40*). Checking for valid digits and a separator goes a long way, but without further information it's hard to be certain that the item is a time and not, for instance, an interval or a decimal number. Hours and minutes specified this way might also designate length (e.g. the playtime of a song) instead of a moment in time. Therefore, it's necessary to look at some additional context in front of or behind the entity. A valid time followed by a timezone (e.g. *20:40*

*CET* or *20:40 +05:00*) or an AM/PM designator (for the 12-hour format, e.g. *8:40 pm*) is a safe bet. Many prepositions also typically precede times, for instance *at 18:45* or *around 7:30* and can be used as prefix indicators. Some keywords can also be used to help separate AM from PM, like *at 8:30 this evening* where *evening* clearly means PM.

Additional time formats that are easy to detect (due to their keywords) are military hours (e.g. *1500 hours*) and whole hours followed by *o'clock* (e.g. *5 o'clock*). These are generally rare but nevertheless occur frequently within certain domains. An interesting observation is that *o'clock* used in military texts more often refers to a position (e.g. *an enemy at 12 o'clock*) than actual time, demonstrating that discretion is advisable when choosing which rules to apply for an extractor within a certain domain.

When the time-entities are extracted, they will occur in many different forms, as seen above. It can thus be useful to normalize them into a common format, for instance the 24-hour, ISO standardized format. Here the timezone is specified as an offset of UTC (Universal Time Coordinated), making events searchable relative to each other on a global scale. Normalization will make the extracted information far easier to work with in later processing steps, like when gathering statistics or checking for the occurrence of a certain entity.

Sometimes the same context information can apply to several entities, for instance *we'll meet at 8:00 or 8:15 pm*, where both times are clearly PM. For entities where this kind of multiple listing is typical, it's an easy enhancement to also detect typical words or constructs designating more than one item, e.g. *or*, *and*, / etc.

### Ticker Extractor

There is a practically infinite number of possible stock tickers. The stock market changes on a daily basis and new tickers appear while old ones are rendered obsolete or possibly recycled. Each stock exchange has its own format which allows certain combinations of alphanumeric characters. This heterogeneity effectively destroys any hope of detecting common patterns that can safely identify tickers, except by making them too general to be useful (e.g. any combination of alphanumeric characters).

To some extent this generality can be exploited by building a dictionary of tickers and checking any valid alphanumeric string against it. This approach is not without difficulties, though. Firstly, it requires building and maintaining this dictionary of valid tickers from within a vast and dynamic domain. Assuring good and accurate coverage will not be easy. Second, many valid tickers are also plain words, acronyms or abbreviations. Filtering out the

most common ones is an option, but for a certain document no guarantees can be made that a sequence of letters is actually a ticker and not a plain word. There is bound to be many false positives.

Again, like with the Time Extractor, it's necessary to look at contextual information. A useful observation is that tickers in free text commonly occur together with their associated stock exchanges (e.g. *Nasdaq: MSFT*). This allows detecting tickers with high accuracy if first identifying the exchange. Luckily, exchanges prove to be much easier to detect safely than tickers. For one, there are orders of magnitude fewer exchanges than tickers, making maintaining a dictionary manageable. Their existence also stays somewhat constant as new exchanges don't pop up all the time. Finally, they're often specified with names, which are substantially easier to recognize safely than just combinations of alphanumeric characters.

The disadvantage of basing the extractor on exchanges is, of course, that it won't detect tickers specified without an exchange. For some ticker formats, though, there is a remedy. Some exchanges namely extend their tickers with a dot and additional characters to indicate which exchange they are traded on. These tickers can thus be recognized with fair certainty if the extension proves valid. Many tickers also trail certain keywords, like *ticker symbol* or *trading code*, and these words can be used as designators.

Tickers are often specified in listings on financial pages. These listings usually leave out any designators, like an exchange or a keyword, making extraction difficult. But what they lack in context they somewhat make up for with structure. These listings are often defined with strict layout elements, using for instance tables. Tickers contained within these elements can thus potentially be recognized based on the structure of their containing element, but this kind of matching requires a completely different framework than the one specified and will only be mentioned here as a possible future enhancement.

Ultimately it would be best if whole documents could be determined to contain financial information and certain rules only ran if this was the case. This level of detection would allow better control over the performance of an extractor, because potentially unsafe or expensive (but useful) rules would only fire when they're likely to yield good results. Document classification is, of course, another field of research, and will not be explored further here. But it's worth mentioning that classification can be simulated to some extent by conditional extraction. That is, if the rules are split up into separate extractors then the rules of one extractor can be set to fire only when the rules of another extractor have first yielded any results. The rules of the first extractor thus effectively becomes the "classifier" upon which the second extractor is based. For example, by only checking potential tickers against the

ticker dictionary once a stock exchange has already been reliably identified (thus implying that the text is financial in nature).

### Job Title Extractor

Job titles can constitute everything from a simple abbreviation (e.g. *CEO*) to a complex construct spanning multiple words (e.g. *financial director of harvesting and production in mining operations*). Keeping the domain out of the job title (e.g. extracting only *financial director*) simplifies the problem, which can then be approached by compiling dictionaries of common abbreviations and simple (e.g. two-word maximum) titles. Extraction then becomes just a matter of doing lookups in a dictionary. On the other hand, ignoring the domain means losing vital information about the function and extent of the title. Without the domain several distinct titles can also seemingly imply the same function (e.g. *financial director of staff* and *financial director of operations* both being detected as just *financial director*).

Expanding the dictionary to cover titles past two-word constructs would yield an improvement in coverage, but also significantly increase the complexity of dictionary compilation. Supporting every possible *n-word* construct of potential job titles quickly becomes unmanageable as $n$ increases past two. Especially since the additional words refer to arbitrary domains and not a limited set of common title fragments.

A better solution is based on the recognition that job titles consist of a few patterns with words from specific word classes, for instance *executive director of assembly* having the pattern "*adjective noun preposition noun*". Identifying these word classes, and how they're assembled to form valid constructs, grants the option of splitting the dictionary into multiple dictionaries based on word class. Note that nouns can further be distinguished semantically as titles (e.g. *director*) and domains (e.g. *assembly*), and can thus result in two separate dictionaries. This distinction helps writing rules that only match semantically valid job titles and avoids constructs like *executive assembly of directors*, even though the pattern of word classes is the same as before.

The advantages of a class-based dictionary approach are multiple. First, the dictionaries become simpler and easier to work with. One huge dictionary with all sorts of complicated constructs can be replaced by multiple, well-arranged single-word dictionaries. Allowing all syntactically (and optionally semantically) sound combinations of words from these dictionaries, gives better coverage while simultaneously assuring only valid constructs. The speed performance of the extractor also likely increases, as it's cheaper to check one word at the time (and skipping the following ones if the first doesn't match) than handling multiple words in one big chunk. This potential speed

increase depends largely on the implementation, of course.

Dealing with multiple dictionaries has one large drawback; the problem of filtering words into the right dictionaries. If this has to be done manually one can imagine a workload similar, or even greater, to that of quality assuring a single multiple-word dictionary. Each word in a potential job title would have to be manually checked and sorted. Luckily, with the proper dictionary this filtering can be performed mostly automatically. If using comprehensive lists of adjectives and nouns (and preferably also a list of titles, or at least nouns applicable to persons), the task of creating a script that filters each incoming word into the proper dictionary becomes easy.

### Quotation Extractor

Quotations refer to something someone has exclaimed. These utterances might have been communicated verbally or in writing (in any form, traditional or electronic). A first approximation for extracting them is therefore to focus on identifying keywords referring to acts of communication. Examples are *said*, *told*, *wrote*, *claimed*, *protested*, *answered* and so on. If one of these keywords is preceded or followed by words encapsulated in quotation marks, there's a fair degree of certainty that the sentence deals with communication in some form. An example is: (*"I won the election fair and square", he retorted*). Covering all possible keywords referring to utterances is no small feat, however, and there are many subtle and metaphorical ones.

A bigger problem is that it's easy to get fooled by the placements of quotation marks in relation to keywords, as in this example: (*In "My Friend, Robin" he wrote that rats are nice companions*). Here, it's easy for a loosely defined extractor to erroneously interpret the quotation marks as referring to an utterance instead of the title of a book. What is needed is a more accurate way to separate actual utterances from titles, names and other noise. Preferably without resorting to complex syntactic rules.

An important property to realize about quotations is that they are cited literally, word-by-word, exactly as the person in question uttered them. Indirect utterances, rephrasings or retells are not quotations. In an example like *he told the police that he saw a man driving a yellow car*, the information transmitted is accurate but not quoted. The defining property of a quotation is thus the quotation marks and not the act of communication. By leaving out the keywords, the extractor can be greatly simplified. But even by focusing on the quotation marks there's still need for a way to separate quotations from noise. Two observations in particular can help out:

1. An interesting utterance can be assumed to contain more than 3 words.

If it doesn't, there's a good chance it's dealing with something else, like a name or a definition.

2. In the English language most titles are written in title case, meaning that all words except closed-class words are uppercased.

The *closed-class words* is the finite set of words that rarely changes during the evolution of a language, like articles, prepositions and conjunctions. *Open-class words* like nouns, on the other hand, receive additional entries frequently as new terms are coined. Being constant, at least for all practical purposes, means that a dictionary of closed-class words can be compiled easily. By requiring that at least one word in a potential utterance is lowercased, and that this word doesn't belong to a closed class, most titles can be avoided. Armed with these observations I wrote a fast and simple extractor based on quotation marks that avoids the most common mistakes.

**Example code**

As an example of what the code behind an entity extractor looks like, I've included the configuration file for my Time Extractor in appendix A.1. This code may seem extensive for the functionality I described above, but keep in mind that this degree of sophistication is required for solid performance in actual text material. To show why, I've also included the reference file I used to test the extractor with in appendix A.2. This file gives an idea of the large amount of subtle variations an extractor must be able to handle. All of the examples in the reference file have been collected from actual text. The extractor recognizes all the time formats noted in the time tags in the reference file. It also avoids the false examples listed at the end.

In the extractor code (appendix A.1), note that the transformation patterns in the end are not part of the matching process. Rather, they're responsible for normalizing all the extracted time variations to the ISO standardized 24-hour format. Also note that the *$ws* slave simply matches whitespace.

## 4.4 Question classification

Even when pattern matching, there are many possible ways to process questions for classification. The particular approach that is chosen greatly affects the quality of the answers.

- The simplest is to match only enough to determine answer type and just convert the rest of the question to a plain keyword query. This

kind of *surface* matching triggers on certain predefined sequences of words and ignores syntax completely. All entities identified in the text, which are of the right type and occur near the keywords, are returned as answers.

- A more sophisticated solution is the *syntactic* one. This tries to recognize how the constituents of a question relate, to better determine what form the answers will take. Entities in the text must fulfill the same requirements as in the surface approach and must additionally appear in a valid syntactic relation to be considered answers. The actual semantics of the answer is not considered though, nor of the question for that matter.

- Finally, the *semantic* approach. The meaning of a question is deduced and formalized. Potential answers (which again must fulfill the requirements of the surface approach) are similarly processed by deducing and formalizing the meaning. Unless the semantics of the formalized structures of both the question and the answer agree, the answer will not be considered correct. This solution typically relies on a syntactic component, but not necessarily. Semantics can also be inferred from language modelling and statistics.

These approaches are only rough outlines and there are many variations over each type. There is no strict border between them either, and particular elements from each can be combined in various ways to suit a specific purpose. My solution falls somewhere in between the syntactic and the semantic approaches. I want to classify questions in such a way that answers can be verified both syntactically and semantically. But my approach is neither strictly syntactic nor strictly semantic. It uses a variant of syntactic patterns (tailored to my specific needs) to capture certain semantics in the questions. The resulting classification can be used to verify answer types semantically, and answer relations syntactically. The semantics of an answer as a whole can not be deduced nor proven to coincide with the question. This is not my intent either, as this relies far too much on semantic processing.

### 4.4.1 General discussion

Consider the question *The Hindenburg disaster took place in 1937 in which New Jersey town?*. This is one of the more complex questions in the test set. What kind of elements can be identified in this sentence to aid in finding an answer to the question? I've identified the following traits:

- The *target* of the question, i.e. the expected answer type, is a location. This type is too general an answer, though. It further has to be a town and it has to be seated in New Jersey. These semantic requirements should be assured.

- The *focus* of the question is the Hindenburg disaster. This is the most important aspect to include in a search and should be treated accordingly. All the other words from the question are secondary, and might, or might not, be useful in a keyword search. Note that "focus" has no strict, linguistic definition. It's merely what I've considered paramount in a given question type. The focus may change between different question types, but stays the same within the same type. It's thus possible to identify and match upon. Most often it is a noun phrase, but not always. Simply put, the focus of the question is the phrase that becomes the focus of the query.

- The entity type of the focus noun head (i.e. "disaster") is an event. As the goal is to verify the answer, not the question, determining this type has no immediate benefit. But it might be useful in e.g. limiting the search to sentences mentioning a disaster (or a general event) given that such an extractor exists. Or it might be used in assuring that "Hindenburg" in a sentence refers to the disaster and not e.g. the German general for whom the airship was named.

- The question is time dependent. That is, it specifies a particular instance in time (i.e. 1937) to which the answer must refer. In this case it's not crucial, but it would be if there had been more than one such disaster. To see this problem more clearly, a title such as "president" refers to different persons through different periods of time.

- The interrogative word in the question (i.e. "which") is not at the beginning of the sentence. To classify the question, this word, and the associated answer type, must be located within the sentence.

### Premises

As I set out to analyze how I could best classify questions, I had the following premises:

- I wanted to learn what constitutes a question. That is, how questions are formed syntactically, what they have in common, what kind of information can be gathered from them, by what measures they can

be classified and so on. I had already browsed through a few sample questions and noted observable attributes that could be made machine recognizable. In doing so I saw that more could probably be done than what the papers I had read suggested. To be certain I needed to explore the same kind of questions without bias.

- I wanted to see to what extent I could process the questions using pattern matching alone. Practically every system I had seen relied on some form of POS tagging for accuracy, or even full syntactic parsing. Through my work with the extractors I had become convinced that for limited applications such as this (i.e. question processing), a comparative level of performance could be reached with clever patterns. But I also knew that I had a tool at my disposal that the other systems did not; the Matcher framework. I wanted to exploit this opportunity.

Given these premises, I decided that the only way to thoroughly explore the questions was to do it manually. That is, to organize them by some property (or several such) and establish common traits. It would be nice to be able to automate this process somehow, but automation is best applied when already having analyzed and learned which properties can be automated successfully. Premature automation is bound to miss crucial information. Besides, as this was my first analysis of the questions I lacked the luxury of intimate knowledge.

Further, due to my desire to use pattern matching, the choice of processing method naturally fell on heuristics. As explained in section 3.3.4 the main advantages of machine learning are coverage and language-independence, while heuristics win on accuracy and performance. Heuristics are also perfect for cases of low variation such as for a limited set of questions. Besides, maximizing recall in this case is only a matter of generalizing the patterns to cover each variation in question formulation. I demonstrate how when describing my classification patterns in sections 4.4.2 and 4.4.3.

### The question set

The questions used in the TREC QA tracks were used as a basis for analysis. These are clean, varied, generally factoid in nature and freely available. Duplicating the effort of carefully compiling such a list of suitable questions for machine consumption was thus pointless. Besides, since TREC was used as a general outline for my implementation, it was prudent to tailor the system to answer the same kind of questions as the participating systems of these conferences.

In section 4.3.2 I've described the general procedure I found effective when building any new pattern matching component. The same procedure was used as a guideline when building the QA component. But since the TREC questions already represent the type of material that would need to be matched, the main focus was on analyzing questions by looking for common patterns.

As a basis for analysis I collected all the questions from three TREC QA tracks, namely from the years 2001-2003. These contain a total of 1500 questions, which should be more than adequate as a starting point for manual examination. I avoided the earlier two tracks partly because I felt I already had enough questions to examine and partly because succeeding tracks maintained or surpassed the complexity of earlier ones anyway. By skipping them I wouldn't miss out on anything except quantity.

The reason I started at year 2001 was because this is the year the track moved beyond standard factoids and introduced list questions. In 2003 the definition questions were also added, and I wanted to explore these two types. I stopped at 2003 because in the following year the question paradigm changed drastically towards a more dialogue-centric, knowledge-gathering type of system (section 3.4.1). This new paradigm demanded quite a different overall design than what I was targeting, and was thus of no interest to my goal.

### Preparing for analysis

When I had filtered out the questions from the meta-data in these files and merged them into one, it was time to prepare the questions for analysis. In doing so I devised, and utilized, the following procedure:

1. Sort questions by entity type. That is, figure out what general type of answer the question asks for (e.g. person, company, location, date) and group questions by this type. This type is the main property by which questions will be classified.

2. Separate questions with implicit from explicit answer type. That is, split each group from above into two parts based on whether the answer type is implicit in the interrogative word (e.g. "who") or mentioned as part of the question (e.g. "which city"). These two cases must be handled separately.

3. Sort questions by syntactic similarity. That is, move questions with similar phrasing together so that it's easier to see shared features. E.g. *When was Algeria colonized?* and *When was Hiroshima bombed?* belong

naturally together. Ignore the actual meaning of the words. Only the syntax is important. Sort by increasing complexity.

4. Find representatives of each syntactic variation. That is, pick a sentence for each type of phrasing that is characteristic of the type. Be careful to recognize, and represent, also minor variations, as these help assure proper coverage. Gather these hand-picked sentences for further analysis.

5. Find common patterns among the gathered sentences. This is where the real work begins. Identifying shared features is not always easy, and describing these formally is even harder.

The groups of example sentences, and corresponding patterns, that will follow throughout this chapter have all been organized in this manner before analysis. This will become apparent when seeing how they're organized by type, similarity and complexity.

### Describing questions with patterns

I wanted to capture the part-of-speech (POS) of each word because these POSes are useful for different purposes, e.g. for term expansion (section 4.5.2). As I would not be using a POS tagger I had to establish these word classes by other means.

My starting point was a set of dictionaries organized by POS, e.g. verbs, nouns, adjectives. The open-class dictionaries were already available for general use in Fast ESP, but could just as well have been built on need by using a syntactic parser on a large corpus of English texts. The closed-class dictionaries, e.g. prepositions, were extracted by myself from linguistic resources available on the web.

Given such a set of dictionaries, the POS of a word can be determined by consulting these for a match. A given word may well get multiple matches though, as many words have multiple POSes. For instance, "play" is both a verb and a noun. Dictionaries are thus not enough by itself. From my experience with writing extractors I knew that context was invaluable in properly classifying an entity. This is no different for POSes, as this is merely another case of type classification. To take an example, in "play the game", the word "play" can be classified as a verb because it is followed by an article and a noun. This relationship can be formalized into a rule.

Of course, context rules are already heavily used in POS tagging tools. But these tools are meant to be usable on any type of sentence. As such, they have to contain a huge set of rules covering all kinds of ways in which words

can be used in relation to each other. Constructing these rules manually would take tremendous effort, so instead the taggers are based on learning these rules from a carefully tagged training corpus. In my case, however, it's not a matter of properly classifying every kind of sentence. I only have a limited set to work with, namely the questions from the TREC test sets. In other words, I only need rules that cover these limited ways in which questions are asked. As this is manually achievable I chose to write these rules myself. Besides, these rules are not only useful in determining POS but also in describing the syntax of the whole question.

When classifying questions it's not enough to merely look at the interrogative nor the words immediately following it. I thus wondered: Why not utilize the whole sentence structure? Normally, this would be difficult as it requires setting up and employing some form of syntactic parser. In this case, the process can be greatly simplified by integrating both POS tagging and shallow parsing as part of the classification process. In other words, by writing patterns that describe the various syntactic constructs that need to be supported and tag POSes accordingly. Full syntactic parsing at query time would greatly affect runtime performance, but in this case it's only a matter of running a set of quick matching rules.

Using a test set such as the TREC set is a golden opportunity as it represents the various ways in which questions to the system will be phrased. The heuristics of the system can thus be tailored to handle these exact kinds of phrases. This does not restrict the topic of the questions. The questions are still domain independent because the syntax does not influence the semantics. But it does, on the other hand, keep question classification limited to a certain set of phrases. This is not necessarily a bad thing though. As explained in (section 3.3.2), placing some limitations on the syntax might well help the users express their needs more clearly to the system. And it does aid the implementation greatly as it assures that the syntactic variations are deterministic. This creates a situation in which pattern matching thrives. Specifically, the situation is ideal for building a classifier based on syntax.

Note that my question patterns are not fault tolerant. That is, they assume valid grammar. Syntax mistakes will necessitate falling back to a general set of patterns that mostly ignores syntax beyond capturing answer type. Spelling mistakes can be handled by using Levenshtein matchers for extraction. This kind of matcher is merely one of the various classes of matchers mentioned in section 4.3.1. Its speciality is the Levenshtein algorithm explained briefly in section 2.4.2. Shortly put, the matcher can be configured to tolerate simple typing mistakes.

### Explanation of my pattern syntax

I'm using pattern matching as a form of shallow parsing. Because my needs are "shallow" I can be satisfied with a linear representation of syntax. That is, I don't have to parse a sentence as a hierarchy, but can rather make do with a sequence of words. I have no use for a "deep" parse tree as I don't need to understand every aspect of the syntax. I only need to parse enough to classify the question and the POSes of the words within.

In doing so I've created my own set of shallow patterns capturing the various elements of the questions I've found interesting. These patterns are based on PCRE-style regular expressions combined with slave patterns. In the same manner as my previous entity extractors, they run under the Matcher framework of Fast ESP as explained in section 4.3.1. The slaves are based on linguistic concepts of syntax and named accordingly, e.g. an *$np* is a noun phrase. However, they must not be confused with precise renditions of these concepts. They only capture enough of their respective grammatical aspects to be useful for my purposes. They may very well be incorrect in a strict linguistic sense, but are correct in the sense that they capture what they're supposed to. The slaves I use in my patterns are described in table 4.1.

| **Name** | **Explanation** |
|---|---|
| *$adj* | adjective |
| *$np* | noun phrase |
| *$pron* | pronoun |
| *$pp* | preposition phrase |
| *$vp* | verb phrase |
| *$np_cn* | *$np* restricted to common nouns |
| *$np_pn* | *$np* restricted to proper nouns |
| *$np_cnpn* | *$np* on the form *$cn $pn* |
| *$np_prep* | *$np* which is part of a *$pp* |
| *$np_verb* | *$np* which is part of a *$vp* |
| *$np_type* | *$np* indicating answer type; always a *$np_cn* |
| *$np_unit* | *$np* denoting unit of measurement |
| *$be* | tenses of "to be"; i.e. is, are, was, were |
| *$do* | tenses of "to do"; i.e. do, does, did |
| *$have* | tenses of "to have"; i.e. have, has, had |
| *$get* | tenses of "to get"; i.e. get, got, gotten |

Table 4.1: Slave patterns for question classification

The patterns used to describe questions in the following sections all use a

simplified syntax. That is, the actual patterns used in the matching process are longer and more thorough. They need to be able to match actual questions and handle all the syntactic details. But this additional complexity is not required to understand the patterns. On the contrary, including them here will only induce confusion. I've thus kept the example patterns as clean as possible to better explain the underlying concepts. For an example of code, refer to section 4.4.5.

One exclusion that is worth mentioning, however, is that of the slave patterns handling stopwords. These words will not be used for further processing and are simply discarded. As such there will be more words present in the questions than the patterns indicate. These words are handled, of course, but have no value in explaining the interpretation of the questions.

The syntax will become clear with the examples in the following sections, but a brief example here might be useful:

(4.3)

| When | is | the official first day | of summer? |
|---|---|---|---|
| *when* | *$be* | *$np* | *$np_prep+* |

In example 4.3 the phrase "the official first day" is recognized as one single *$np*. The adjectives "official" and "first" are not matched separately with *$adj* because they are not required to distinguish the pattern. As any noun might have one or several adjectives, it makes sense to match these as part of the NP. Articles, e.g. "the", are also captured with the noun. Further, there is no separate *$pp* for "of summer" either. Because only the noun here ("summer") is interesting in a query, the preposition ("of") is captured as part of the *$np_prep* ("of summer") and simply discarded. The noun ("summer") is of course kept. Also note that the trailing plus-sign (+) indicates one or more instances as this is a regex. This means that two *$np_prep* constructs, for instance, might be chained such as in "of summer in Jamaica".

Finally, questions surrounded in parentheses are not from the TREC test set. They are rather ones I've created to demonstrate some specific syntax. They may or may not ask sensible questions:

(4.4)

| (Where | is | the Hubble telescope?) |
|---|---|---|
| *where* | *$be* | *$np* |

### 4.4.2 Implicit answer type

Questions for which the general answer type can be determined from the interrogative word will now be discussed. These have an *implicit answer type*. The benefit of implicitness is, of course, that the questions can be

immediately mapped to a corresponding entity extractor. There are three kind of questions fulfilling this property; "when", "where" and "who". The corresponding entities are roughly time/date, location and person. I say "roughly", because, as will soon become clear, their coverages are more broad than these precise entity types.

### "When" questions

Questions based on the interrogative word "when" all ask for some moment in time. This moment can be designated in several ways. For instance, as hours and minutes (22:45), as month and day (June 16th), as year (1937), as time of day (noon), as a season (spring), or as a combination of several of these (22:45 on June 16th, 1937). Semantically, these variations represent different spans of time, i.e. different levels of accuracy on the time scale. A year is a longer span than a day, for instance. Time might also be specified as an interval, e.g. "22:45 to 23:45" or "June to August". Again, the difference lies in the span. The moment has to be specific, though, i.e. designated with a starting point and an optional stopping point. A length of time, e.g. "3 hours", does not fulfill this requirement and is, indeed, not a moment in time but a unit of measurement. These are handled in section 4.4.3. Examples of characteristic time-based questions from the TREC test set follow:

(4.5)

| | | | |
|---|---|---|---|
| When | is | the summer solstice? | |
| When | was | the first kidney transplant? | |
| *when* | *$be* | *$np* | |
| | | | |
| When | is | hurricane season | in the Caribbean? |
| When | is | the official first day | of summer? |
| *when* | *$be* | *$np* | *$np_prep* |
| | | | |
| When | was | Abraham Lincoln | born? |
| When | was | the first Wal-Mart store | opened? |
| *when* | *$be* | *$np* | *$vp* |

The questions in example 4.5 are many and varied. Still, they can all be collected in a unified pattern because they all ask for the same entity type. Generalizing them into one common pattern is easy since they only use variations over the same basic elements. The unified pattern (example 4.6) will handle all of these cases and more:

(4.6) (When was the first Wal-Mart store opened in Ohio)?
*when* *$be* *$np* *$vp?* *$np_prep**

In addition to these questions there is another common syntactic variation in the TREC set. This variation warrants a separate set of patterns (example 4.7 which can be unified into the pattern in example 4.8:

(4.7) When did Elvis Presley die?
When did the Hindenburg crash?
*when* *$do* *$np* *$vp*

When did Idaho become a state?
When did North Carolina enter the union?
*when* *$do* *$np* *$vp* *$np_verb*

(4.8) (When did Michelangelo paint the ceiling of the Sistine Chapel?)
*when* *$do* *$np* *$vp* *$np_verb?* *$np_prep**

The reason the questions in example 4.7 should be separated from the others is because the auxiliary verb (*$do*) has changed. Specifically, it has gone from a tense of "to be" to a tense of "to do". In contrast with the former, the latter auxiliary verb requires a main verb, so that a *$vp* is no longer optional but mandatory in the pattern. But more importantly, this auxiliary verb changes the form of the main verb from passive, e.g. "was born", to active, e.g. "did enter". A change in this form affects the verification of the syntactic relations in the answer, since the target now becomes the syntactic object rather than the syntactic subject. This can be seen by exploring possible answers to the question *When did North Carolina enter the union?*. Contrast the answer *North Carolina entered the union in 1789* with *The union entered North Carolina in October*. Only the former construct corresponds with the active role of the main verb in "did enter" and is thus a valid answer. The latter speaks of a whole different event.

There is one further variation worth mentioning: The active role of a verb can change back to passive with the introduction of another auxiliary verb. Consider the following example:

(4.9) When did John F. Kennedy get elected as President?
*when* *$do* *$np* *$get* *$vp* *$np_verb?*

Normally, the presence of "did" would imply an active role for the succeeding verb. But since the verb is also precluded by "get" the role remains passive, i.e. Kennedy was elected by someone – he did not elect himself. In other words, this type of question belongs with the passive verb questions in example 4.6. This becomes clear when rewriting the question in the following manner and pairing it with the unified pattern from example 4.6:

(4.10) (When was John F. Kennedy elected President?)
*when* *$be* *$np* *$vp?* *$np_prep**

Finally, the question could also be modified to an active form by one of the following changes:

(4.11) (When did Kennedy elect the President?)
*when* *$do* *$np* *$vp* *$np_verb*

(When was Kennedy electing the President?)
*when* *$be* *$np* *$vp* *$np_verb*

The first case in example 4.11 will get caught by the unified pattern in example 4.8. The second case is tricky, however. It will not be matched by any of the previous patterns. It is close to the examples in 4.6, but that pattern won't handle *transitive verbs*, i.e. verbs with arguments (captured by the sequence $vp $np_verb). And rightly so as transitivity implies activity. This new pattern is unique and will thus unambiguously handle similar questions. The trickster in this case is not the transitivity, however, but the *present participle* form of the verb, "electing". This form also implies activity but may not necessarily have an argument. This can result in a question like the following:

(4.12) (When was Kennedy talking?)
*when* *$be* *$np* *$vp*

This example (4.12) will erroneously be interpreted as passive by my patterns (example 4.6). Handling it correctly is only a matter of creating a separate rule for verbs ending in "-ing", which is the form of the present participle. But as these last questions are already beyond what the TREC set requires, I have not spent any more effort on handling them.

| Pattern | Type |
|---|---|
| *when $be $np $vp? $np_prep** | be |
| *when $do $np $get? $vp $np_verb? $np_prep** | do |

Table 4.2: Unified "when" question patterns

### "Where" questions

In much the same way as a "when" question refers to a point in the dimension of time, a "where" question refers to a point in the dimension of space. As dimensions go, these two interrogative words are thus strikingly complementary. The answer type to this kind of question constitutes a location. Such a location is often geographic and might further be political, e.g. a country or a city, or natural, e.g. a mountain or a river. But just as "when" might refer to any instant in time, or any span thereof, so might "where" refer to any location in three-dimensional space. A location could thus just as well be "behind the closet", "on the finger", "inside the house", "atop the tree", "under the bush", "up in the air" and so on. These kinds of spatial locations can usually be identified by the preposition precluding them. But in the most general case, any noun, tangible or not, that can either belong in space or confine a space of its own, can constitute a location. Some examples of characteristic location-based TREC questions follows:

(4.13)
| | | | |
|---|---|---|---|
| Where | is | the Eiffel Tower? | |
| Where | is | John Wayne airport? | |
| *where* | *$be* | *$np* | |

| | | | |
|---|---|---|---|
| Where | was | the first golf course | in the United States? |
| (Where | is | the city | of Atlantis?) |
| *where* | *$be* | *$np* | *$np_prep* |

| | | | |
|---|---|---|---|
| Where | is | Hitler | buried? |
| Where | are | the British crown jewels | kept? |
| *where* | *$be* | *$np* | *$vp* |

The questions in example 4.13 can all be unified for the same reasons as the "when" questions in example 4.5. Further, the unified patterns will be identical except for the interrogative word:

(4.14) (Where was the decisive battle fought in the American Civil War)?
*where* *$be* *$np* *$vp?* *$np_prep**

This resemblance between the unified patterns of 4.14 and 4.5 suggests that a similar correspondence exists between the patterns of the active forms. And indeed it does, as shown in examples 4.15 and 4.16 (unified).

(4.15) Where did Howard Hughes die?
Where do apple snails live?
*where* *$do* *$np* *$vp*

(Where did Robin Hood defeat Little John)?
(Where does a tree frog lay eggs)?
*where* *$do* *$np* *$vp* *$np_verb*

(4.16) (Where did the Aztecs build the pyramids in South America?
*where* *$do* *$np* *$vp* *$np_verb?* *$np_prep**

This only demonstrates the interchangeable nature of questions about time and space. They can essentially ask about the same events only differing in the dimension in which the answer lies. The location questions in the TREC set do contain a few hitherto unseen variations though. For instance, consider the question:

(4.17) Where is the volcano Mauna Loa located?
*where* *$be* *$np_cnpn* *$vp*

First of all, this example (4.17) shows that the verb is not always useful. In this case it is, in fact, superfluous. Not only is a location already implied by the interrogative word, but many potential answers are unlikely to state explicitly that a location is in fact a such, e.g. *Mauna Loa lies in Hawaii*. Using this verb in a query might thus prove detrimental rather than beneficial. These kinds of obvious verbs might instead be identified and filtered out to keep them from restricting the search. They can be recognized in much the same manner as described in section 4.5.3.

Secondly, note that "Mauna Loa" is defined as a volcano in the question. This is accomplished by the pattern *$cn $pn* and is captured by the corresponding *$np_cnpn* slave. Normally, this kind of information would probably not be very helpful as it concerns the focus, not the target, of the question. However, it can be useful in assuring that a potential answer refers to the

volcano and not any other kind of similarly named entity. There is, for instance, also a solar observatory and a brand of nuts with the same name. Extracting the location of any of these would not be very useful.

The final example (4.18) demonstrates that an *$np_prep* may well follow immediately behind the interrogative word instead of trailing the question:

(4.18)

| Where | on the body | is | a mortarboard | worn? |
|---|---|---|---|---|
| *where* | *$np_prep* | *$be* | *$np* | *$vp* |

The meaning is the same, and likewise the result of processing, but the patterns have to explicitly handle this case. This simply means the unified patterns crafted so far must be updated to support an optional *$np_prep?* after the interrogative. I will not do this here, however, to avoid the clutter.

| **Pattern** | **Type** |
|---|---|
| *where $np_prep? $be $np $vp? $np_prep** | be |
| *where $np_prep? $do $np $get? $vp $np_verb? $np_prep** | do |

Table 4.3: Unified "where" question patterns

### "Who" questions

Questions using the interrogative word "who", or the related forms "whom" or "whose", typically inquire about persons. Such questions might refer to one or several individuals, with one being the arguably most frequent variant. But they might also refer to a collective of individuals as a whole if these fit naturally under a common banner, e.g. a company, an organization or a band. And it doesn't stop there:

Humans have a tendency to lend human attributes to almost any living (or simulated) organism able to perform an action (physical or not). This phenomenon is called *anthropomorphism*. For instance, "who" might just as well refer to a monkey, a dog, a hamster, a robot, a toy figure or a cartoon character. These might be referred to by their given names (e.g. "Fluffy") or their collective names (e.g. "the dog").

Finding the right answer entity depends on interpreting the context properly, which is why it's so important to process the whole question. Since "who" questions all refer to some kind of being, rather than points in a temporal or spatial dimension, the question patterns naturally differ somewhat from the "when" and "where" questions.

Some examples from the TREC set will now be explored. Note that I will refer to the answer type as "person" for simplicity, regardless of whether the answer is an actual person or any other anthropomorphized organism.

(4.19) Who was Galileo?
Who is Duke Ellington?
*who* *$be* *$np_pn*

A "who" question with a lone *$np* (4.19) is more complex to answer than a similarly lacking "when" or "where" question. A thorough answer to who someone is requires defining that person in some manner. This can result in an arbitrarily long description. As such it can be viewed as a definition question rather than a factoid question, and the target type is consequently a definition rather than a person. Put differently, a "who is X" question is essentially the person-centric equivalent to the more general "what is X" and belongs with that class of questions. How to answer these will be discussed in 4.4.4. Note that this property is restricted to proper nouns (hence the *$np_pn*). The situation gets more blurry when the focus is a common noun:

(4.20) (Who is the president?)
(Who is a computer programmer?)
*who* *$be* *$np_cn*

Here (4.20) it is no longer certain whether to answer with person names or definitions. The questions might ask either who fulfills these roles (e.g. "George W. Bush") or what functions the roles embody (e.g. "The president is the governmental head of the republic"). In the latter case it's more accurate to change the question to a "what is X", e.g. *What is a computer programmer?*. Because this option is available to the users, my system will interpret these questions as asking for a specific person. It doesn't take much to disambiguate the answer type, though, as the next example demonstrates:

(4.21) Who was the first female U.S. Representative?
Who is a German philosopher?
*who* *$be* *$adj+* *$np*

What keeps these questions (4.21) from avoiding the same uncertainty in answer type as in example 4.20 is the additional adjective. This adjective makes all the difference in that it changes the focus from a general role to a specific one, thus targeting the person (or persons) fulfilling that role. By keeping an adjective mandatory in the pattern, a person answer is certified. It doesn't have to be an adjective, though. Adding a PP is also enough to ensure a person answer:

(4.22) Who was president in 1913?
Who was the original voice of Mickey Mouse?
*who* *$be* *$np* *$np_prep*

This example (4.22) also illustrates the first use of a time-dependent question. These were mentioned briefly in section 4.4.1. The PP "in 1913" restricts a valid answer to this specific point in time. Extracting a president from any other period is not very useful. Finding a valid answer might be difficult unless this exact year is stated explicitly in the answer. It's not enough that it's included in some time period (e.g. "1910 to 1914") as understanding this kind of implication requires specialized logic. Additionally, there might even be several correct answers to such a question if multiple entities fulfill the requirement. In this case they do, as 1913 was an election year in which the presidency changed hands.

When including a verb in the question an interesting property can be observed:

(4.23) Who was the first person to reach the North Pole?
*who* *$be* *$np_type* *$vp* *$np_verb*

Who was the first American to walk in space?
*who* *$be* *$np_type* *$vp* *$np_prep*

Who was the first African American to win the Nobel Prize in literature?
*who* *$be* *$np_type* *$vp* *$np_verb* *$np_prep*

The semantics of the sentences in example 4.23 depart from what has previously been seen. The focus of the sentence is no longer the NP but rather the VP. I reiterate that this view has no linguistic foundation but is merely my intuition as to which part of the question should be focused on in query transformation.

To see my point, take the question *Who was the first person to reach the North Pole?*. The significant information here is who reached the North Pole, not who was the first person (in general). Of course, both pieces of information are important, but a query for the former will likely yield better results than a query for the latter, hence the focus. The fact that it was a person is even implicit in the "who". And that it was the "first" such

is rather a modifier to the event. This becomes clear when rewriting the question into the equivalent *Who reached the North Pole first?*.

A slightly more involved example can be seen in *Who was the first American to walk in space?*. If rewriting it in the same manner as above it would become *Who walked in space first?*. This clearly misses the integral part that the person in interest is an American. A solution is to rewrite it into a "which" question explicitly specifying the target type, i.e. *Which American walked in space first?*. Questions with explicit types will be handled in section 4.4.3. But note already now that I mark the NP as denoting answer type with the slave pattern *$np_type*. This type will always refer to a person given the "who" nature of the question.

Using a verb as the focus of the question might be more understandable if considering that a verb might easily be converted into a noun, as in the case of "to walk" becoming "the walker". Consequently, the question *Who was the first American to walk in space?* might be rephrased as *Who was the first American walker in space?* or even *Who was the first American space walker?*. These kinds of modifications have benefits that will be explained in section 4.5.2.

The "who" questions in the TREC test set also introduce a new element into the sentences; the pronoun:

(4.24)

| Who | was | the abolitionist | who led | the raid | on Harper's Ferry |
|---|---|---|---|---|---|
| *who* | *$be* | *$np_type* | *$vp* | *$np_verb* | *$np_prep* |

| in 1859? |
|---|
| *$np_prep* |

| Who | is | the actress | known | for her role | in the movie "Gypsy"? |
|---|---|---|---|---|---|
| *who* | *$be* | *$np_type* | *$vp* | *$np_prep* | *$np_prep* |

Both "who" from "who led" and "her" from "her role" are pronouns referring to the person who is the answer to the question. These pronouns might also be exchanged with "that" for "who", i.e. "that led" and "a" for "her", i.e. "a role". Since these pronouns are not utilized anywhere in the system, they're not matched separately but rather identified and discarded as part of the VP or NP, respectively. This is in accordance with how the other stopwords are treated.

All the "who" questions mentioned so far, except the definition questions in example 4.19 may be unified in one single pattern:

(4.25)

| (Who | was | the artist | that changed | his name | to a symbol?) |
|---|---|---|---|---|---|
| *who* | *$be* | *$np_type* | *($vp* | *$np_verb?)?* | *$np_prep** |

As before, this unification (4.25) is possible because the questions all ask for the same entity type and only use variations over the same constructs. If a question contains neither a verb nor a preposition it reverts to the common noun example of 4.20. The common noun here is assured by using *$np_type*, which is neither valid nor useful for proper nouns as these can't determine type.

The active "do" form seen in "when" (4.8) and "where" (4.16) questions is conspicuously missing from the TREC test set. For sake of completion I've included the unified pattern for the "do" form of "who" questions also:

(4.26)
| | | | | | |
|---|---|---|---|---|---|
| (Who | did | Harry Potter | zap?) | | |
| (Who | did | Harry Potter | give | the Wand | of Zap to?) |
| *who* | *$do* | *$np* | *$vp* | *($np_verb?* | *$pp)?* |

The final examples show the most interesting use of a verb in a "who" question:

(4.27)
| | | | |
|---|---|---|---|
| Who | discovered | America? | |
| Who | developed | the Macintosh computer? | |
| *who* | *$vp* | *$np_verb* | |

| | | | |
|---|---|---|---|
| Who | won | Ms. American | in 1989? |
| Who | developed | the vaccination | against polio? |
| *who* | *$vp* | *$np_verb* | *$np_prep* |

| | | |
|---|---|---|
| Who | dies | in the "Half Blood Prince"? |
| Who | lived | in the Neuschwanstein castle? |
| *who* | *$vp* | *$np_prep* |

Because a question of this kind (4.27) refers to a person, someone who's able to perform actions, the main verb (the action) may come directly after the interrogative. In other words, there's no need for an auxiliary verb like "did" to introduce the performer of the action. Further, like in example 4.23, the focus of the question lies with the VP. These verbs may also be modified to nouns in the same manner as before, e.g. *Who discovered America?* becoming *Who is the discoverer of America?*.

These kinds of questions don't have to specify an active role. They may just as well be passive:

(4.28)
| | | | | |
|---|---|---|---|---|
| Who | was | elected | president | of S. Africa in 1994? |
| (Who | was | destroyed | | in the Trojan War?) |
| *who* | *$be* | *$vp* | *$np_verb?* | *$np_prep** |

Finally, the questions from these last examples (4.27 and 4.28) may be unified in a common pattern (4.29):

(4.29) Who invented Trivial Pursuit?
*who* *($be|$get)?* *$vp* *($np_verb $np_prep*|$np_prep+)*

| Pattern | Type |
|---|---|
| *who $be $np_type ($vp $np_verb?)? $np_prep** | be |
| *who $do $np $get? $vp ($np_verb? $pp)? $np_prep** | do |
| *who ($be|$get)? $vp ($np_verb $np_prep*|$np_prep+)* | vp |

Table 4.4: Unified "who" question patterns

### 4.4.3 Explicit answer type

Questions with an *explicit answer type* can't be mapped to a corresponding entity extractor without finding out what general type the question asks for. How to accomplish this is detailed in section 4.5.3. This explicitness has a benefit, though. Whereas the implicit types are more easy to determine general type from, the explicit types are more specific. That is, once the general type is decided the subtype is specified explicitly. Implicit types lack this preciseness and could refer to any subtype within the general type.

#### "How X" questions

Questions starting with "how" are often used to inquire about an explanation or procedure, e.g. *How does a combustion engine work?* or *How did Wellington defeat Napoleon?*. When trailed by an adjective, though, the answer type frequently changes to a measurement, e.g. *How long is the Great Barrier Reef?*. An adjective by itself is no sure indicator, as the answer type might also be a opinion or estimate, e.g. *How dangerous is a killer whale?*. But if the adjective can be determined to specify a measurement, the question can be translated to a corresponding scope query for measurement entities.

There is also a similar type of questions that asks for a quantity instead of a measurement, e.g. *How many Olympic Games were canceled because of World War I?*. Since the quantity indicator here is an adverb, not an adjective, it is easy to recognize as such. Such a question might sometimes appear to ask for a measurement, e.g. *How many liters are there in a gallon?*, but the answer is in fact a quantity as it does not require the unit of measurement. That is, *3.79* is just as good as *3.79 liters*.

A few example of measurement questions will now follow. The new slave pattern *$measure* only matches adjectives that are known to specify measurements.

(4.30)

| | | | | |
|---|---|---|---|---|
| How | close | is | Mercury | to the sun? |
| How | long | is | the Great Barrier Reef? | |
| *how* | *$measure* | *$be* | *$np* | *$np_prep?* |

| | | | | |
|---|---|---|---|---|
| How | far | is | it | from Earth to Mars? |
| (How | long | is | it | between the equinoxes?) |
| *how* | *$measure* | *$be* | *$pron* | *$np_prep+* |

| | | | | |
|---|---|---|---|---|
| How | late | is | Disneyland | open? |
| *how* | *$measure* | *$be* | *$np* | *$vp* |

Note the different meanings of the adjective "long" in example 4.30. It might refer both to a distance and a duration. This ambiguity can to some degree be dissolved by separating the patterns in which they occur, as just exemplified. But in some cases they both use the same patterns. To be absolutely sure, both types of units must be queried for.

Even when a measurement is unambiguously recognized, e.g. length or temperature, these measures can be denoted using different units. For instance, a length can be given in kilometers, feet or even astronomical units (AU). Similarly, a temperature can be given in Celsius, Fahrenheit or Kelvin. When no unit of measurement is specified, all of these are valid. But questions might also ask for a unit explicitly:

(4.31)

| | | | | |
|---|---|---|---|---|
| How | large | in square miles | is | North Carolina? |
| (How | fast | in kph | is | the fastest speedboat?) |
| *how* | *$measure* | *$np_unit* | *$be* | *$np* |

In these cases (4.31), only the specified unit is interesting. Still, it might be useful to also return other units as long as the desired unit is ranked first. This unit may not be located, however. But if others are, a possible feature enhancement is to automatically convert the measure to the desired unit. This kind of specialized behavior is possible since the patterns are sophisticated enough to recognize the unit semantics, and would not be as easy if using only a POS tagger.

So far, the questions have all been of the form "to be". These patterns are similar enough to be unified (table 4.5). But the questions might also use other forms, like "do", "can" and "shall":

(4.32)

| How | big | does | a pig | get? |
|---|---|---|---|---|
| *how* | *$measure* | *$do* | *$np* | *$get* |

| How | hot | does | it | get | in Death Valley? |
|---|---|---|---|---|---|
| *how* | *$measure* | *$do* | *$pron* | *$get* | *$np_prep* |

| How | long | did | Rip Van Winkle | sleep? |
|---|---|---|---|---|
| *how* | *$measure* | *$do* | *$np* | *$vp* |

| How | fast | can | a king cobra | kill | you? |
|---|---|---|---|---|---|
| *how* | *$measure* | *$can* | *$np* | *$vp* | *$pron* |

| How | cold | should | a refrigerator | be? |
|---|---|---|---|---|
| *how* | *$measure* | *$shall* | *$np* | *$be* |

These patterns (4.32) can also be unified (table 4.5). Again, some of the measures are ambiguous. "Long" this time refers to time, which is indicated by the verb. "Big" can refer to both weight and length. In this case, probably weight, but both measures are useful to include since the question is imprecise.

Table 4.5 summarizes the unified measurement patterns. Observe that these are basically equal to the unified "be" and "do" patterns of "when" (table 4.2) and "where" (table 4.3) questions, except that they also include pronouns and auxiliary verbs of the form "can" and "should". These additions are useful for some of the more abstract measurement questions exemplified.

| **Pattern** | **Type** |
|---|---|
| *how $measure $np_unit? $be ($np\|$pron) $vp? $np_prep** | be |
| *how $measure $np_unit? ($do\|$can\|$shall) ($np\|$pron) ($be\|$get\|$vp ($np_verb\|$pron)?) $np_prep?* | do |

Table 4.5: Unified "how" question patterns

### "Which X" and "what X" questions

The interrogatives "which" and "what" are often called *determiners*. This term is highly appropriate in a classification sense, since the succeeding noun effectively determines the answer type. These two interrogatives are used interchangeably in the following example questions and are handled identically by the system. For simplicity, though, I will just use "what" in the patterns presented here regardless of which of the interrogatives is actually used in the questions.

All of the previously mentioned question types have their representatives in which/what questions. Each of these use their own specifier dictionary to recognize valid answer types within their own domain.

Time questions are exemplified in (4.33):

(4.33)

| | | | | | |
|---|---|---|---|---|---|
| What | date | was | Dwight D. Eisenhower | born? | |
| What | year | was | the Mona Lisa | painted? | |
| *what* | *$date* | *$be* | *$np* | *$vp* | |

| | | | | | | |
|---|---|---|---|---|---|---|
| What | date | did | Neil Armstrong | land | | on the moon? |
| What | year | did | Mussolini | seize | power | in Italy? |
| *what* | *$date* | *$do* | *$np* | *$vp* | *$np_verb?* | *$np_prep** |

| | | | |
|---|---|---|---|
| What | is | the date | of Mexico's independence? |
| *what* | *$be* | *$date* | *$np_prep+* |

| | | | |
|---|---|---|---|
| What | was | the last year | that the Chicago Cubs won the World Series? |
| *what* | *$be* | *$date* | *$clause* |

Note the possessive NP "Mexico's independence". This will be normalized to "independence of Mexico" before question classification and will thus be matched by the regular preposition patterns. This normalization is explained in 4.5.1.

The first two patterns of (4.33) are basically identical to the unified "be" and "do" patterns of the implicit "when" questions (table 4.2). This is only natural since the interrogative words serve the same purpose in both the implicit and the explicit case. This explicitness allows for better precision by specifying the subtype of entity, e.g. "what year" versus simply "when". The similar nature of these question types can be seen clearly by rewriting *What year was the Mona Lisa painted?* as *When was the Mona Lisa painted?*. In terms of implementation, the benefit of this similarity is that the patterns

can basically be merged since they differ only in the initial interrogative part of the questions.

The last two patterns of (4.33) introduce a new and important way to formulate questions permitted by the explicit type. When the type is mentioned as a noun (e.g. "the date"), the rest of the phrase can refer back to this noun by using e.g. a preposition phrase or a relative pronoun.

The relative pronoun is the most important as it introduces a new slave pattern; *$clause*. None of the earlier question examples have been complex enough to have need of this pattern. It's very useful, however, if the questions use clauses. Shortly put, a *clause* is a sub phrase that can either stand on its own (independent) or must be part of a whole (dependent). In this case, *that the Chicago Cubs won the World Series* is a statement that has no meaning on its own, since the relative pronoun "that" refers back to "the last year". These clauses can basically form new phrases of complexity equal to the original phrases. A sentence containing such a clause is thus rightly called a *complex sentence*.

With clauses, it's not necessary to create separate sub patterns for each specific pattern and possibly have to replicate these across all the patterns needing them. Instead, I've collected all of these variations in a common *$clause* pattern and simply refer to this wherever appropriate. This pattern is shown in example 4.34 along with a few examples of the kinds of sub phrases it matches (mainly from the TREC set):

(4.34) when he died
where the meteoroid hit the surface
when it comes into contact with a strong acid
that have returned looted Nazi art to their owners or descendants
*$relpron ($pp $np|($np|$pron) $auxverb? $vp $np_verb?) $np_prep?*

This pattern (4.34) introduces two new slave patterns of its own. *$relpron* is the relative pronoun that introduces the sub phrase, represented by the first term in the examples (4.34). *$auxverb* is just an auxiliary verb like "get" or "do" that often introduces the main verb. I see no need to differentiate these auxiliary verbs here.

Location questions are exemplified in (4.35):

(4.35) What U.S. state's motto is "Live free or Die"?
What state is the geographic center
*what* *$location* *$be* *$np*

of the lower 48 states?
*$np_prep**

| | | | | |
|---|---|---|---|---|
| What | continent | is | Argentina | on? |
| What | province | is | Montreal | in? |
| *what* | *$location* | *$be* | *$np* | *$pp* |

| | | | | | |
|---|---|---|---|---|---|
| What | country | did | Ponce de Leon | come | from? |
| What | state | did | the Battle of Bighorn | take place | in? |
| *what* | *$location* | *$do* | *$np* | *$vp* | *$pp?* |

| | | | |
|---|---|---|---|
| What | is | the location | of the Sea of Tranquility? |
| What | is | the capital | of Ethiopia? |
| *what* | *$be* | *$location* | *$np_prep+* |

| | | | | |
|---|---|---|---|---|
| What | Canadian city | has | the largest population? | |
| What | state | has | the least amount | of rain per year? |
| *what* | *$location* | *$have* | *$np* | *$np_prep** |

These example patterns (4.35) can basically also be unified into patterns equal to the "where" questions (table 4.3). The are a few differences, though:

- The new examples show that both "be" and "do" questions can end with a single *$pp* rather than a full *$np_pron*. This is especially important for location-based questions since prepositions typically refer to locations. But it might also be a useful addition to other kinds of questions, e.g. *Who was the Golden Ticket given to?*, and should thus be introduces to all appropriate unified patterns.

- These examples introduce a new auxiliary verb; "have". Syntactically this form is essentially equal to the "be" patterns. Due to the different semantics (having versus being) it is useful separating the patterns though, to allow for separate treatment. This form is just as applicable to other question types, e.g. *Which former Austrian has the largest biceps?*, and should also be added to the unified patterns.

Person questions are exemplified in (4.36):

(4.36) What French ruler was defeated at the battle
(What country leader was awarded the 2000 Nobel Peace Prize?)
*what* *$person* *$be* *$vp* *$np_verb?*
of Waterloo?

*$np_prep**

(What student did Snape zap?)
(What jolly dwarf did the piano fall upon?)
*what* *$person* *$do* *$np* *$vp* *($np_verb?* *$pp)?*

What American composer wrote the music
Which U.S.A. president appeared
*what* *$person* *$vp* *$np_verb?*
for "West Side Story"?
on "Laugh-In"?
*$np_prep**

The person patterns in (4.36) are also similar to the unified "who" questions in table 4.4, with some differences. The focus changes back from the verb, e.g. *Who developed...*, to the noun, e.g. *What developer...*. The "be" form of the questions require more than an *$np* to be complete. The "do" form is glaringly absent from the TREC set, so a few examples have been added. And there is no need for a *$clause* here, since it belongs more naturally in a "who" question, e.g. *What is the president that...* versus *Who is the president that...*.

Measurement questions are exemplified in 4.37:

(4.37) What is the atomic weight of silver?
What is the temperature of the sun's surface?
*what* *$be* *$measure* *$np_prep**

What is the average weight of a Yellow Labrador?
What is the normal blood sugar range for people?
*what* *$be* *$measure* *$np_prep**

What is the distance in miles from the earth to the sun?
*what* *$be* *$measure* *$np_unit* *$np_prep**

| | | |
|---|---|---|
| What | is | the earth's diameter? |
| What | is | the world's population? |
| *what* | *$be* | *$measure $np_prep** |

In contrast with the earlier measurement questions of the "how" kind, these explicitly mention the answer type just like the other "which" questions. It's thus only a matter of recognizing these nouns as measurements using a regular specifier, and verify that the answer belongs to the right type.

The unified patterns for the explicit questions are summarized in table 4.6.

| |
|---|
| **Time patterns** |
| *what $date $be $np $vp? $np_prep** |
| *what $date $do $np $get? $vp $np_verb? $np_prep** |
| *what $be $date ($np_prep+\|$clause)* |
| |
| **Location patterns** |
| *what $location $be $np $vp? ($np_prep*\|$pp)* |
| *what $location $do $np $get? $vp $np_verb? ($np_prep*\|$pp)* |
| *what $location $have $np $np_prep** |
| *what $be $location ($np_prep+\|$clause)* |
| |
| **Person patterns** |
| *what $person $be $vp $np_verb? $np_prep** |
| *what $person $do $np $get? $vp ($np_verb? $pp)? $np_prep** |
| *what $person ($be\|$get)? $vp ($np_verb $np_prep*\|$np_prep+)* |
| |
| **Measurement pattern** |
| *what $be $adv? $measure $np_unit? $np_prep** |

Table 4.6: Unified "what" question patterns

### 4.4.4 Advanced questions

#### List questions

List questions in the TREC set are either explicit and commanding, e.g. *List female astronauts or cosmonauts.*, or implicit, direct and querying about multiple entities, e.g. *What are the colors of the German flag?*. They can thus

be recognized by these two properties, i.e. command (4.39) and/or plural noun(s) in the answer type (4.38).

(4.38)

| | | | |
|---|---|---|---|
| What | are | the colors | of the German flag? |
| What | are | the two houses | of the Legislative branch? |
| *what* | *$be* | *$np_type* | *$np_prep* |

| | | | |
|---|---|---|---|
| What | gasses | are | in the troposphere? |
| *what* | *$np_type* | *$be* | *$np_prep* |

Although a question may ask for a limited number of entities, e.g. "two houses", I see no point in restricting the answers to any predefined amount. The system might as well return a complete list of all matching entities, ranked according to importance, and let the users decide which are interesting; most likely the top entries. Also, even though a question may seemingly ask about a single entity, there may still be multiple valid answer entities. For instance, the answer to *What do you eat at a sushi restaurant?* might be "sushi", "fish", "maki", "rice" or any other suited food. And in *Who developed the Macintosh computer?* both the persons involved and their company are interesting answers; different entity types may all be correct. Not to mention a question like *How big does a pig get?* which depends on the measurement (length or weight) and unit (centimeters or inches, or kilograms or pounds).

The commanding questions introduce a new aspect into the questions; conjunctions:

(4.39)

| | | |
|---|---|---|
| List | female astronauts or cosmonauts. | |
| List | the names | of cell phone manufacturers. |
| *list* | *$np* | *$np_prep?* |

| | | | |
|---|---|---|---|
| List | Hezbollah members | killed or apprehended | by Israeli forces. |
| List | the names of casinos | owned | by Native Americans. |
| *list* | *$np* | *$vp* | *$np_prep* |

An $np handles multiple phrases by recognizing coordinating conjunctions (i.e. "and" and "or") and marking each noun phrase separately, e.g. "astronauts" and "cosmonauts". The same applies to a $vp, e.g. *killed or apprehended* split into the verbs "killed" and "apprehended". Conjunctions are translated directly to booleans in the scope query, meaning "or" creates

a list of alternative keywords. Further, the phrases also handle comma separated listings created by conjunctions, e.g. *dogs, cats, hamsters and mice*, although these are not used in the TREC set.

The biggest problem with conjunctions is figuring out whether the surrounding elements apply to just one or all of the items in the clause. For instance, in example 4.39 are the cosmonauts supposed to be female or only the astronauts? Can the cosmonauts be male? And are the Hezbollah members supposed to have been killed by Israeli forces or does this requirement only apply to the apprehension? Can they have been killed by other means? These kinds of ambiguities follow conjunctions all the time, but humans usually resolve them unconsciously because only one of the alternatives usually makes sense. Encoding this sense into an algorithm is not easy, but a general rule of thumb is:

- All elements are applied to all items in the clause unless they already have elements of their own.

For instance, *female astronauts or cosmonauts* is likely to mean *female (astronauts or cosmonauts)*, while *female astronauts or male cosmonauts* means just that. Likewise, *killed or apprehended by Israeli forces* is likely *(killed or apprehended) by Israeli forces*. For simplicity, however, I capture adjectives as part of the NP as usual, and treat verbs as equal alternatives. Thus, the Hezbollah example is treated as stated while the astronaut example is interpreted as *(female astronauts) and cosmonauts*. This is easier to implement and also more clear due to requiring users to be explicit rather than making assumptions that are bound to have unpredictable exceptions. If a different behavior is required, the users can reformulate their questions or simply split them into multiple questions without using conjunctions.

### Definition questions

Definition questions are typically on the form "who is X" or "what is X". Example 4.40 shows some representative questions from the TREC test set:

(4.40)

| | | |
|---|---|---|
| Who | was | Abraham Lincoln? |
| Who | is | Duke Ellington? |
| *who* | *$be* | *$np* |

| | | |
|---|---|---|
| What | is | Teflon? |
| (What | is | black magic?) |
| *what* | *$be* | *$np* |

One way to answer definition questions is by looking the subject up in a dictionary or encyclopedia. The new fact mining components of Google, Yahoo and MSN do this (section 3.5.6). This approach will provide thorough answers, but the answers will be paragraph sized or even page sized unless the source has been tailored specifically to answer concisely. Such long answers are probably useful for their contextual value (see 3.4.3) but might also be too verbose if just looking for summarized facts. Of course, the text chunk may be trimmed upon retrieval by only extracting the interesting parts from the entry.

A drawback is that these sources tend to be sparse on information beyond common facts. That is, they work nicely for well-established and fairly static concepts, like e.g. abbreviations, laws of nature or important historical moments. But they often have bad coverage beyond these areas, for instance with information that is state-of-the-art, disputed, specialized or simply little known. Discussions on such topics tend to flourish around the open web, though, which is ideal for solutions based on crawling and indexing. Besides, all topics, even "undisputed" facts, are useful to get information on from several sources and angles. An encyclopedia only provides one view; the one which is favored by the author of the treatise. If only one source is used, one might question why the users should go through the QA system at all. They might instead search directly in the encyclopedia. This will probably be just as quick.

Another problem with this solution is that the source of definition answers is entirely separate from the corpus indexed by the engine. Since the corpus doesn't provide support for the answers, this rules out any benefit from selective indexing. The definition component might as well be regarded as an entirely separate auxiliary module. Again, it might be better for users to simply go directly to the source, since the information isn't live anyway.

For these reasons, it's attractive to extract definition answers in the same manner as other answers. The difficulty lies with the open nature of these answers. Since it's no longer a matter of finding entities, but rather sentences of arbitrary length, traditional entity extraction is not applicable. It is no trivial task recognizing what kind of sentences constitute solid information worthy of being definitions, nor is indexing upon these. Proper definition extraction probably requires sophisticated and costly result-time processing. Of course, a quick solution is to simply return all phrases where the focus of the question is the subject. This is likely to produce many irrelevant facts though, such as *Lincoln supported the Second Bank of the United States*, *Lincoln was unsentimental about agriculture* and *Lincoln wished to share Douglas's fame*. These extracts need context to be informative, but even then they're likely not very important in defining the person Abraham Lincoln.

But a compromise can be made between these two approaches; that is, between looking up and mining definitions. Definition questions typically inquire about who or what someone or something is. This "is-a" relationship is the exact premise upon which my semantic lexicon is built (section 4.7.3). In fact, the semantic lexicon can be viewed as a repository of definitions mined from the corpus. Some examples from my lexicon are: *Abraham Lincoln: great United States president*, *Duke Ellington: notable black musician*, *Teflon: low friction plastic* and *black magic: practice of witchcraft*. Of course, this restricts the answers to noun phrases rather than verb phrases. That is, it will miss *Galileo invented the telescope* but catch *Galileo, inventor of the telescope*. In a way, this solution entails using the lexicon in reverse – to look up information rather than verify it. As such, the requirements on precision are much stricter than what my current lexicon can fulfill. But the general idea might be carried further.

### 4.4.5 Classification summary

This classification section demonstrates that it's easy to create a few unified patterns with broad coverage that can efficiently determine answer type and POS at the same time. There are several other interesting aspects that can be detected in the TREC answer set, but these were the ones I had time to present.

The disadvantage of using syntactic patterns is that each type of phrase must be explicitly supported, otherwise the system won't understand the question. This isn't as bad as it may sound, though, as a few general patterns with a fair share of variants can capture a lot of different phrases. Once valid sub phrases have been identified, these can be reused and shared among the patterns. Also, while questions may well be asked in cunning and peculiar ways, there is an argument to be made for restricting users to phrase their questions in a clear and precise manner. This will help both the users and the system to focus.

It's also fairly easy to implement a fall-back strategy for unsupported phrases. A set of shorter patterns can be derived from the existing ones that only match far enough to recognize the answer type. The rest of the question can then simply be converted to a general keyword-based query. Since this implies ignoring the contextual clues inherent in the questions, these clues can't be used when verifying the syntactic relations in the answer. Instead, answer extraction will also have to utilize a fall-back step where entity types are verified purely based on their semantic classes and their proximity to important keywords.

So far, the syntactic structure of the sentence is not used for much except

determining POS. Since the answer type can basically always be determined in the start of the sentence, these extensive patterns may seem rather excessive. It might appear better to just match enough to determine type and then leave the rest to an optimized POS tagger. That way, the patterns wouldn't have to support every kind of syntactic variation. But this structure is, in fact, useful for other purposes than POS tagging. Specifically, the slave patterns I've created can be configured into new patterns that recognize certain semantics in the questions. For instance, the unit conversion suggested in the measurement questions. I'll give more examples of how syntactic structure can aid semantic discovery when creating scope queries in section 4.5.4.

When dealing with implicit answer type I simply translate the interrogative word to the appropriate supported entities. This kind of simple syntactic classification is not exact. For instance, "who" might refer to both a person and an organization. For maximum precision, the system would have to be able to distinguish which of these types is requested when unspecified. It's possible to go to great extents to accomplish this, and some solutions do just that.

On the other hand, one might question whether this level of precision really matters to the end user. For instance, given a question like *Who developed the Macintosh computer?* both the persons and the company behind the product are correct answers. The only difference is the scope of the target, e.g. single individual or multiple. If multiple, they might be listed individually or under some common banner (e.g. that of a company). If only one of these is returned as the answer, it is, of course, important which one is picked. But if multiple answers can be ranked and presented comfortably it's valuable seeing all applicable answers. Fast ESP navigators are perfect for this purpose. The short answers (e.g. entities) can be presented as a list and the user can drill down on the right entity type (e.g. persons vs companies) and further drill down on individual entities to see the long answers that support the particular entity.

If the user does not specify the type precisely, e.g. "which person" rather than "who", chances are that the specific type is not regarded as important. If it is, the user can simply drill-down to the appropriate entity type or even rephrase the question to be more specific. For instance, *Which person developed the Macintosh computer?* or even *Who was the person that developed the Macintosh computer?*. One advantage of QA technology is that it gives users the power to be very specific without requiring a complex syntax. This power also extends to e.g. specifying unit in a measurement questions like *How fast in kph is an African swallow?* and for explicitly creating a definition question like *What is the president* rather than *Who is the president?*. Given this power, the responsibility for assuring precision largely falls to the users.

As long as it's just as easy to create a precise question as an imprecise one, there's no real reason to implement sophisticated logic to try to outguess the users. Poorly formulated questions can be just as confusing to humans as to an algorithm. An imprecise question will give a corresponding imprecise answer, but in the world of ambiguous natural languages this is as it should be.

#### Example code

To demonstrate how I've implemented the question classifiers, I've included the configuration file for my person classifier in appendix A.3. This code is able to recognize all the various person questions I described in section 4.4.2 and 4.4.3. Basically, the code is an implementation of the unified patterns in table 4.4 (implicit questions) and table 4.6 (explicit person questions). Note that this configuration file is rather sparse. This is because most of the work is handled by the slave patterns I described in my pattern syntax in section 4.4.1. These slave patterns are required by all the classifiers, so I've collected them in a common auxiliary file. The code for this file can be viewed in appendix A.5.

## 4.5 Query transformation

### 4.5.1 Normalization

Language is complex and allows lots of variants and quirks. To lessen the burden of having to deal with all these variants upon question classification, it's useful to normalize the questions before passing them along to these matchers. This section will explain how and why this is done.

#### Interrogative sub phrases

Questions can be phrased in different ways. The most common way is by starting with the interrogative word, such as in *What state did the Battle of Bighorn take place in?*. Here, the first part, "what state", indicates the target of the question. But the target may well be precluded by other words. For instance, by a preposition like "on" in *On what continent is Egypt located?*. In fact, the interrogative words may even come last in the question, as seen in the example *The Hindenburg disaster took place in 1937 in which New Jersey town?*.

All of these variations have to be supported. But since the question will not be parsed it's difficult to rely on the syntactic structure of the sentence

to identify the interrogative words. That is, there is no parse tree built that will clearly state where the interrogative sub phrase starts regardless of how the question is phrased. These variations will instead have to be handled by patterns.

Writing separate patterns to match every variant is possible, as each extractor can easily handle multiple patterns. This is not a good solution, though. It's time consuming to write, hard to maintain because modifications must be applied to each pattern, and slow to execute because it entails running each pattern in sequence until one matches.

A better idea is to retrieve the interrogative sub phrase separately, simply by ignoring the other words. This requires matching twice, though; once for the target and once for the subject. The problem here is that the Matcher framework in Fast ESP is built to extract one element at a time. In other words, two separate matchers will have to be set up and executed for each question type. The same kinds of problems as above surface.

Ideally, both the target (e.g. "New Jersey town") and the focus (e.g. "Hindenburg disaster") should be extracted by one and the same pattern. Instead of adapting the extraction to the questions it might be possible to adapt the questions to the extraction. By analyzing the question variations as patterns, instead of as sentences, it becomes apparent that the interrogative sub phrase is just shuffled around in each variation. This crucial realization implies that the questions may be normalized before extraction by reshuffling the sub phrases to create the same general pattern.

For example, *On what continent is Egypt located?* becomes *What continent is Egypt located on?*. And *In the Bible, who was Jacob's mother?* becomes *Who was Jacob's mother in the Bible?*. Of course, this transformation runs the risk of invalidating the syntax, as when *The Hindenburg disaster took place in 1937 in which New Jersey town?* is transformed into *Which New Jersey town the Hindenburg disaster took place in 1937 in?*. This kind of deformation produces questions that will not be matched unless the patterns are modified to be more lenient, i.e. less dependent on syntax. For instance, by not requiring an auxiliary verb (like "did") to appear between two noun phrases. Alternatively, the transformation can be enhanced to produce syntactically valid constructs by implementing heuristics that rewrite general patterns of one form to another form, regardless of question type or topic.

This solution still requires two matchers; one for normalizing the question and one for extracting it. But reshuffling is trivial compared to writing additional patterns and also identical for each type of question. Instead of two custom steps for each and every question type there's now only one additional step that they all have in common. Best of all, the question matching component can now treat each question pattern as a normalized

phrase always starting with the interrogative word.

### Contractions

English uses many contractions to shorten frequent phrases, e.g. "isn't", "he's" and "could've". To avoid having to handle these variants it's useful to expand them before matching, i.e. "is not", "he is" and "could have". This is especially important considering that interrogative words frequently use contractions too, as in "who's the actor" and "what're the names". Also, expansion is a cheap way to guarantee that these words will never be tokenized into meaningless constructs like "isn" and "t" by a later step.

### Possessive markers

The possessive marker (') is used in English to denote possession in nouns, e.g. "the dog's collar". Instead of handling these phrases as special cases when processing nouns, it is more convenient to instead normalize them to a form that is already supported. This kind of transformation is possible by observing that the pronoun "of" can in most cases be used as a replacement without changing the meaning, like in "the collar of the dog". By expanding the noun phrase with a preposition phrase the possessive form can thus be removed. Note that the article "the" is inserted for common nouns. This insertion doesn't have to be grammatical as long as the resulting question pattern is valid.

The questions in the TREC test set use the possessive form in three main ways, three of which are expandable, but for different purposes:

- Consider *What is Shakespeare's nickname?*. This is the regular case in which the possessive occurs. A naive pattern would either regard the words of the NP as separate, i.e. "Shakespeare" and "nickname", or use the NP as an exact phrase, i.e. *"Shakespeare's nickname"*. None of these solutions make it clear that the answer type is a nickname. By instead expanding the question to *What is the nickname of Shakespeare?* it is immediately clear to my patterns that a nickname is the interesting answer type here.

- Another useful case is seen in *What city's newspaper is called "The Enquirer"?*. If expanding this phrase in the same way as above, it would become *newspaper of the city*. This would indicate that the target is a newspaper, which is wrong. The target is a city, but "The Enquirer" refers to the newspaper, hence the formulation. The proper behavior is

inserting the auxiliary verb "has" and removing the previous auxiliary, i.e. *What city has the newspaper called "The Enquirer"?*

- Consider *St. Patrick's Day* and *Harper's Ferry*. In these cases the possessive is part of the name. Proper nouns using possessive form should thus stay unmodified.

- The exception from the above rule is if the NP contains an explicit phrase, such as *Mozart's "Don Giovanni"* or *Rowling's "Harry Potter"*. These cases should be expanded by the preposition "by" instead of "of"; e.g. *"Don Giovanni" by Mozart*.

Note that this kind of expansion has to work recursively. Consider the question *What was J.F.K.'s wife's name?* from the TREC test set. The target here is not the wife but her name. Performing one expansion is inadequate as it produces *What was the wife's name of J.F.K.?*, which has quite a different meaning. Repeating the step doesn't help either, as this simply results in *What was the name of J.F.K. of the wife?*. Iteration is not the proper tool here, but recursion. The phrase "wife's name" first has to be expanded to "name of the wife", then "J.F.K.'s name of the wife" to the final sentence *What was the name of the wife of J.F.K.?*. Of course, this kind of recursion can be simulated by iterating backwards. In this exact case, however, the work is largely wasted. The person (the wife) and the name will likely result in the same answer anyway (a person name). But replace "name" with "hobby" and the functionality is justified.

### Stopwords

As explained throughout section 4.4.1, stopword removal is handled as part of the question classification process. The patterns that match the questions are responsible for discarding any words deemed irrelevant, i.e. stopwords.

I first considered identifying just the answer type and then send the rest of the question unmodified to the search engine. E.g. *Who was the first person to reach the North Pole?* would become the query *and(string("the first to reach the North Pole", mode="AND"), scope(person))*. The idea was to just rely on the stopword removal mechanism of the search engine instead of creating one myself.

The big disadvantage of this approach was that I would have to rely on the search engine to interpret the rest of the question. That is, the question would just be treated as a plain keyword search. In other words, the only real improvement over a combined IR and IE solution would be the ability to automatically recognize the answer type. I wanted to do more. I saw great

potential in utilizing the contextual information in the question to both direct the search and verify the answer. It also looked promising to expand terms based on what was useful for the particular question class, rather than just providing general synonyms or some such. In other words, I quickly decided that it was better to control my own stopword removal than rely on Fast ESP to do so.

In this regard I first contemplated filtering out the stopwords at the start, as part of the normalization step, and then send the filtered query for question classification. Not having to deal with stopwords in the query would make the processing easier, but it would also make it much more difficult both to tag the remaining words properly and to figure out the intent behind the question pattern. Thus I decided on the thorough question classification procedure detailed in 4.4.

An added benefit of fully controlling my own stopwords is that negation can be maintained. For instance, if the word "not" is filtered out from a sentence, the meaning will effectively be inverted. Take for instance *What are the animals that don't have backbones called?*. If applying standard stopword removal the query would end up with something like *what animals have backbones called*, which is very likely to return the exact opposite information from what was intended. Of course, the negation will still have to be upheld when verifying the answer, but recognizing it as part of question classification allows just this.

### 4.5.2 Term expansion

#### Verb/noun conversion

As mentioned when discussing the subject of a "who" question in section 4.4.2 it's quite easy to change the verb in a sentence into a noun. This was the case in "to walk" becoming "the walker".

This kind of transformation doesn't only apply to verbs denoting activity. It also works for passive verbs where, for instance, "to employ" might become both "the employer", for the active part, and "the employee" for the passive. Further, the verb doesn't have to be converted to a person but might in fact become any other object or concept playing the part of a noun. For instance, "to walk" might become "the walk", referring to an event. And "to employ" would similarly become "the employment".

This close verb/noun relationship is not merely a curiosity but a central part of how new words are formed from existing classes when the need arises; e.g. to describe an action as a thing or vice versa. This transformation property can be very useful. Consider the question *When did Elvis Presley*

*die?*. A potential answer might not necessarily use a tense of the verb "die", as in *Elvis died on August 16, 1977*. Instead, the event might be referred to as a noun, as in *...death of Elvis in 1977*.

If this kind of relationship between a verb and a noun could be formally established, it could be used in expanding the query with additional terms. It would probably also be useful in finding potential answers by increasing the amount of valid relations. That is, by looking for syntactic construct using both verbs and nouns. These enhancements could prove quite useful in boosting the recall of the answer extraction component. Note that this is not the same as synonym expansion, as these alternate words are not synonyms but rather different parts-of-speech (POS) of the same words. Synonyms are well-known to sacrifice precision because of subtle nuances and unintended word senses. In this regard, I believe term expansion using verb/noun conversion could potentially be more precise as it's essentially the same words in different forms. I can't confirm this assumption, though, as I've not found any study on the topic. Of course, changing the POS also means that the syntactic relations between the words change. Verbs and nouns can't simply be exchanged for one another in the same positions in a sentence. Rather, different patterns have to be used to verify the relations based on the POS.

One reason I've not seen other systems use this kind of expansion, is probably because the properties of synonyms have been researched and understood while this verb/noun relation has not. Also, synonyms are readily available while this relation is hard to come by. Without a proper dictionary containing the links between verbs and nouns the relation can't be exploited. The question, then, is how to build such a dictionary.

There seems to be no strict rule as to how a verb can form a noun and vice versa. But there does seem to be a limited number of suffixes that are used when going from one form to the other. I thus came up with an automated method based on taking the stem of the word and expanding it with such common suffixes. E.g. the verb "employ" could be expanded to a noun by adding the suffixes "ment", "er" or "ee". Using these rules on "develop" would give "development", "developer" and "developee". While "construct" would give "constructment", "constructer" and "constructee". In the latter case, the proper suffixes would be "ion" and "or". As the amount of such distinct suffixed seems to be limited, they could simply be mined and then used exhaustively on every stem to generate possible nouns. Clearly, many of these rules will yield invalid constructs on any particular stem, but it doesn't really matter. The generated words can simply be verified against a dictionary of nouns and all invalid constructs discarded. Once a custom dictionary of verb-to-noun relations has been built, going from noun to verb is just a matter of reversing the relations.

To explore this proposition further I started looking around for a manually compiled list of verb/noun relations. This kind of information turned out to be scarce. Eventually I found out that WordNet [62] contains quite a few of these relations in addition to the regular synonym sets. The related words are widely scattered throughout several custom-formatted database files, though, and I needed to collect them all in an easily surveyable list. For these purposes WordNet provides several interfaces for programmers but none of them suited my particular need. I instead ended up examining the WordNet file format specifications and writing my own script to extract the exact relations I required. To make this work I had to interpret several numerical and symbolical columns and follow links through three separate files just to end up with a correct relation. Once completed, though, the script found and extracted 4450 verbs having such relations, each with at least one (frequently two or more) corresponding nouns. This was a mere 12% of the 36704 verbs in the verb dictionary I used, but it was a start.

Skimming through the compiled list revealed that basically all verbs used the same kind of suffix rules in becoming nouns. If this list is representative of the verb class as a whole, an automated stem-suffix expansion method such as outlined here should be quite functional. I did not have time to implement this method, though, but instead ended up using the list compiled from WordNet to demonstrate the validity of the expansion idea.

Lastly, I identified two minor complications that have to be addressed upon implementation:

- There are some notable exceptions to the stem-suffix rules where a verb and its corresponding noun do not share a common stem, e.g. “die” vs “death” and “born” vs “birth”. On the other hand, these two are the only ones I spotted when skimming the 4450 verbs from the WordNet list. There are probably others, but if so, they seem so rare that they will likely have only a minor impact anyway.

- The list generated from WordNet only contains verbs in their infinitive tenses. This means all verbs will have to be reduced to this form before looking up the relation. When automatically generating such a list it might be an idea to either use the stems for lookup or to expand the verbs with lemmatization. Only the latter case avoids having to modify the verbs on-the-fly before looking them up.

### Lemmatization and synonyms

Lemmatization and synonym expansion are the most common ways to increase recall in a search. Both of these techniques are supported directly in

Fast ESP and utilizing them is just a matter of interfacing with the right components. Alternatively, I could use the underlying dictionaries directly. While lemmatization is generally regarded safe and beneficial, the problem with synonyms, as always, is precision. But because proper answers are far more demanding to locate than a chunk of related information, synonym expansion is also more attractive in a QA context than in an IR one. The answer verification step (which is non-existing in IR) can be used to ensure precision. The underlying IR engine should thus focus on maximizing recall and result ranking.

These expansion techniques are well-known and established and leave little room for further research. As such they will neither be utilized nor built upon in my system.

### Mining for related terms

Verb/noun correspondence and synonyms are not the only interesting relations that can be discovered between words. Nor is manual effort (e.g. WordNet [62]) the only way to establishing these relations. Any procedure that can be automated is highly interesting. Automation is likely to cover orders of magnitude more words than possible by a team of linguists, at least within a reasonable time frame. It also paves the way for a general method that can be used across languages if tailored to the intricacies of each particular language.

I've learned, through experimenting with extractors, that automation is possible in many regards. For instance, many kinds of word correlation classes can be mined automatically by identifying some representative that would frequently co-occur with an instance of the class. For instance, a person extractor can be used in finding verbs applicable to a person's actions, as in e.g. *I eat...*, *she walks...* and *John said...*. Similarly, the same extractor can also identify verbs representing actions that can be done unto a person, as in e.g. *...showed me*, *...gave her* and *...asked John*. It doesn't stop with verbs, though. To find out what kinds of nouns are edible, for example, phrases such as *...ate the apple* and *...eating the shrimps* can be collected. Or to find out what kind of nouns cause deaths, a quick web search provides e.g. *death by stoning*, *death by hyperpyremia* and even *death by chocolate*.

Of course, there are likely to be many false candidates caused by e.g. a metaphor such as *She ate the words*, or a dubious phrasing such as *death by stereo*. Some level of assurance is possible by counting the frequencies in which these terms occur in the desired relationships. Only candidates above some set treshold are considered safe. Still, to maintain precision it's crucial to at least establish the right POS and sense of a given word. For

this case of general linguistic processing, both full POS tagging and parsing could preferably be utilized. As performance is not an issue when building offline dictionaries, nor recall given the near limitless amount of possible texts available for scrutiny, the procedure guaranteeing the highest precision should be used. Still, absolute precision is not crucial when the resulting dictionaries will merely be used for verification and not for generating any kind of output.

All of these automatically discoverable relations might be interesting in a QA context. The more world-knowledge that can be utilized upon question classification, the better. For instance, the above-mentioned list of edible things might be useful if the target of the question is something edible. Or the list of death causes might be applied when responding to a question of how someone died. Due to time constraints I will not be able to follow this work further, though.

### 4.5.3 Translating answer types to scopes

There's not much use in having a lot of extractors if the answer types identified in the questions can't be mapped to these. Unfortunately, questions will rarely use terms that correspond directly with the supported entities, e.g. "which person..." for person entities. Rather, this mapping from answer type to general entity must go through an ontology that knows which terms apply to which entities. I use dictionaries for this purpose and refer to them as *specifiers*, since they specify the entity type.

These specifier dictionaries are organized by entity type, and ideally there should be one per extractor. The only thorough dictionary I've had time to acquire is the person specifier. It contains 7585 different terms that all apply to persons. These terms were automatically mined by using a procedure that will be described shortly. The rest of the specifiers only contain the most essential terms that were identified manually through question analysis.

Specifiers indicate general types that corresponds with the existing extractors. Used alone, they are unlikely to produce an answer that is precise. They don't have to either, as semantic verification is utilized to further determine subtype. But verification can be costly if it has to be performed on all kinds of phrases that might constitute an answer. Specifiers are thus useful in narrowing down the answer candidates to only the entities that are worth processing. In this regard, specifiers can be seen as a form of preliminary verification for supported types.

Even if the subtype can't be verified, specifiers are still highly valuable as they will at least assure that the answers are of the correct, general type. For instance, the answers to a question such as *Which president was unmar-*

*ried?* will at the very least be persons, which is far more accurate than simply returning all kinds of proper nouns, e.g. companies, product brands or countries.

#### Mining for specifiers

The person specifier dictionary was mined by using the person extractor. Specifically, by extracting common nouns directly preceding person names. This kind of pattern has the interesting property that the common nouns define the succeeding proper nouns. E.g. *president Clinton*, *country musician Willie Nelson* and *megalomaniac Trevor Goodchild*. This property is not exclusive to person, but can be used to identify specifiers for other kinds of entities as well. For instance, locations like *river Nile*, *volcano Mauna Loa* and *city Alexandria*.

Specific patterns can also be generated that match phrases which denote the desired type. For instance, the pattern "*in the $cn of $location*" will catch phrases such as *in the state of New Jersey*, *in the suburbs of Chicago* and *in the wilds of Connemara*. The preposition "in" restricts the common nouns to words that represent potential locations in one way or the other.

If instead starting out with a small dictionary of manually compiled specifiers, it can be enhanced automatically by creating a hyponym extractor of the kind that will be described in section 4.7.3. Basically, by extracting hypernym/hyponym relations that denote membership to a certain class of concepts, e.g. *rulers such as emperors*, *musicians, e.g. songwriters* and *towns and other locations*. This kind of dictionary boosting can, of course, also be performed on the above-mentioned automated specifiers.

### 4.5.4 Building scope queries

The final step in query transformation is to build the actual scope queries. The overall strategy here is to classify the answer type and use this to determine the appropriate entity to use in the scope search, e.g. person, location or measurement. This answer type is recognized primarily by using the interrogative word (for implicit types) or a specifier (for explicit types). For example, the question *What date was Dwight D. Eisenhower born?* becomes the query *and("Dwight D. Eisenhower", or(born, birth), scope(date))*.

#### Precision versus recall

When generating queries the goal is to increase recall by retrieving as many potential answers as possible. To see why, consider creating queries to in-

crease precision instead. The most precise answer is, of course, the one that uses the exact same phrasing and vocabulary as the question. This means the semantic agreement between question and answer will be maximized, as in example 4.41.

(4.41) What French ruler was defeated at the battle of Waterloo?
*The French ruler Napoleon Bonaparte was defeated at the battle of Waterloo.*

But this answer may not exist in the document collection. An answer might instead come in the form of example 4.42.

(4.42) What French ruler was defeated at the battle of Waterloo?
*At Waterloo in Belgium, Napoleon Bonaparte suffers defeat.*

It is thus better to focus on retrieving as many candidates as possible, and leave the task of figuring out whether these answers are correct to the answer extraction phase. This rules out using an exact phrase search for the kind of answer in example 4.41. The query must instead account for possible syntactic and lexical variations. For instance, by avoiding excessive phrasing (e.g. *"the battle of Waterloo"*), by including the noun forms of the verb (e.g. *defeat*) and by not requiring that the answer type (e.g. *French ruler*) appears in the sentence. But the query must also stay within the same frame of semantics. Extensive use of synonyms and lemmatization is especially dangerous as these subtle variations in meaning and tense can quickly sidetrack the search. Generating a lot of irrelevant hits will only increase the processing demands of the answer extraction phase.

### Answer type term inclusion

Some consider it natural to include the answer type term in the generated query, e.g. "color" in the case of *What color is a giraffe's tongue?*. There is no guarantee that an answer will explicitly mention the answer type. For instance, an answer to the above-mentioned question might well be *A giraffe's tongue is blue-black*. Nowhere does it mention that blue and black are colors. If an appropriate extractor exists, in this case capturing color entities, it's better to require that the answer contains a color entity rather than the word "color". The IE component of Fast ESP allows just this. Even if a specific extractor does not exist, a specifier can map the answer type to a general extractor. The specific answer type can then be validated through semantic verification. More on this in section 4.7.

However, the answer type term is useful when there's simply no appropriate extractor available. If neither the entity type nor the answer term is included in the query it can easily become much too general, e.g. the question *What continent is Argentina on?* becoming the query *Argentina*. The top ranked results may not even contain a continent. Including the answer term in these cases provides much needed focus to the search. I thus apply the rule of including the term only if an appropriate extractor does not exist.

### Capturing semantics

As mentioned in section 4.4.5, it would be beneficial to use the classification patterns for something besides determining answer type and POS. This "something" comes in the form of writing specific patterns that recognize the particular semantics in a type of questions. These semantics can be used both to create a better query and to better verify the answer. Some examples of specific patterns that can capture semantics will follow.

Questions that use the preterite tense of the auxiliary verb "to do", i.e. "did", use the present tense in the main verb. For instance, "sink" in *What year did the Titanic sink?*. An answer is more likely to use the preterite tense of the main verb, e.g. "sank" in *The Titanic sank in 1912*. It's thus useful to recognize the tense of the auxiliary verb in patterns and change the tense of the main verb to the preterite tense in these cases. The following pattern can accomplish this and produce the desired scope queries:

(4.43)

| | | | | |
|---|---|---|---|---|
| What | year | did | the Titanic | sink? |
| What | continent | did | Columbus | discover? |
| *what* | *$np_type* | *did* | *$np* | *$vp* |

*and(Titanic, or(sank, sinking), scope(date, type=year))*
*and(Columbus, or(discovered, discoverer, discovery), scope(location, type=continent))*

Observe that I've included a parameter in the scopes (example 4.43), e.g. "type=year" in *scope(date, type=year)*. This parameter is not presently supported by scope searches, but I present it as an example of how the semantics from question classification can be passed on to semantic verification. The general entity that will be queried upon is a date, but the system is also told that the specific type is a year. This type parameter is valuable when verifying the answer type. It might also be used in the search if the extractors were to be expanded with subtypes, as will be discussed in 4.7.2.

When one or more adjectives are used as modifiers to the answer type, these adjectives are also useful to include as types for later verification. In

this manner, all facets of the answer type can be provided for verification (example 4.44):

(4.44) What French ruler was defeated at the battle of Waterloo?
*and(battle, Waterloo, or(defeated, defeat), scope(person, type=ruler, type=French))*

Who was the first African American to win the Nobel Prize in literature?
*and("Nobel Prize", literature, or(won, winner, winning), scope(person, type=African, type=American))*

If an *$np_prep* is used in a "where" question, this phrase functions as a modifier that narrows down the answer type to a certain scale. In other words, the NP of the phrase can be considered a supertype attribute of the scope:

(4.45)

| Where | was | the first golf course | in | the United States? |
|---|---|---|---|---|
| *where* | *$be* | *$np* | *$pp* | *$super* |

| Where | on | the body | is | a mortarboard | worn? |
|---|---|---|---|---|---|
| *where* | *$pp* | *$super* | *$be* | *$np* | *$vp* |

*and("first golf course", scope(location, super="United States"))*
*and(mortarboard, worn, scope(location, super=body))*

Note that "super" here (example 4.45) is different from the earlier "type" parameter. For instance, the location of the golf course should be *a part of* the United States, it should not *be* a United States. Similarly, the bodypart for the mortarboard should be just that, a part of the body, not a body in itself. There is an important semantic distinction between these two parameters.

This supertype parameter is even more interesting for explicit location questions:

(4.46)

| What | U.S. | state's | motto | is | "Live free or Die"? |
|---|---|---|---|---|---|
| *what* | *$super* | *$location* | *$np* | *$be* | *$np* |

| What | New York City | structure | is | also known as | the Twin Towers? |
|---|---|---|---|---|---|
| *what* | *$super* | *$location* | *$be* | *$vp* | *$np* |

*and(motto, "Live free or Die", scope(location, type=state, super="United States"))*
*and("Twin Towers", scope(location, type=structure, super="New York")*

In these examples (4.46), parameters for both subtype and supertype are useful. This is because the questions reveal both the specific answer type and the supertype to which they belong.

For measurement questions specifying unit, the unit warrants its own parameter:

(4.47) How large in square miles is North Carolina?
*how $measure $pp $unit $be $np*

What is the distance in miles from the earth to the sun?
*what $be $measure $pp $unit $np_prep+*

*and("North Carolina", scope(measurement, type=size, unit="square miles"))*
*and(earth, sun, scope(measurement, type=distance, unit=miles))*

If a certain entity type can be recognized as a modifier to the sentence, it can be included as a specific scope in the query. This is the case with, for instance, time modifiers in the time-dependent questions below:

(4.48) Who was president in 1913?
*who $be $person $pp $year*

What city had a world fair in 1900?
*what $location $have $np $pp $year*

*and(scope(person, type=president), date(1913))*
*and("world fair", scope(location, type=city), date(1900))*

Finally, verbs can also trigger scopes if they can be recognized by an appropriate dictionary. This can result in quite powerful queries:

(4.49) What do you eat at a sushi restaurant?
*what $do $pron $vp $np_prep*

What prince said "to be or not to be"?
*what $np $vp $quote*

Who works at Microsoft?
*who* *$vp* *$np_prep*

*and("sushi restaurant", scope(food))*
*and(scope(person), quotation("to be or not to be"))*
*and(scope(person), scope(jobtitle), scope(Microsoft))*

### Heuristics

There are some general heuristics that are useful to employ regardless of what kind of information the question classification revealed. Some of these rules were witnessed above. I summarize the most important rules I've identified here:

- Explicit phrases are maintained. E.g. the question *Who dies in the "Half Blood Prince"?* becomes the query *and(or(dies, death), "Half Blood Prince", scope(person))*. There is no reason to lower precision by breaking up phrases.

- NPs are converted to explicit phrases. E.g. *Who painted the ceiling of the Sistine Chapel?* becomes *and("Sistine Chapel", ceiling, or(painted, painting, painter), scope(location))*. This increases the likelihood of the constituents of an NP being matched correctly, since they won't be split up and possibly matched in unrelated contexts. Note that this phrasing is limited to nouns since no other POS are kept by my NP patterns. This is the reason the above phrase does not become *"ceiling of the Sistine Chapel"*.

- Adjectives are included in the NP. E.g. *How fast is an African swallow?* becomes *and("African swallow", scope(measurement, type=speed))*. By tying the adjective to the NP, higher precision can be assured when extensive syntactic relation verification can't be utilized.

- If there are multiple adjectives, these are included as alternate phrases with the NP. E.g. *How much does the human adult female brain weigh?* becomes *and(or("human brain", "adult brain", "female brain"), human, adult, female, scope(measurement, type=weight))*. This achieves a compromise between recall and precision: One of the phrases is required for precision, while the rest of the adjectives can occur outside the phrase to increase recall.

- Verbs are transformed to nouns, if possible, and used to boost the query. E.g. *Who founded American Red Cross?* becomes *and("American Red Cross", or(founded, founder, foundation, founding))*. This means answers can be identified even if using the noun form of a word instead of the verb.

- Any non-alphanumeric characters, white spaces and stopwords are ignored. This does not require any explicit rule during transformation. Rather, such noise will simply not be captured as part of question classification.

## 4.6 Answer extraction

Due to time restrictions, I had to focus my efforts on question classification. This meant abandoning the effort of designing and implementing a procedure for extracting correct answers. I believe this was the right choice since question classification is a crucial prerequisite for answer extraction. Though abandoned, this section contains a brief description of my intended method for assuring answer correctness.

### 4.6.1 Processing answers

The steps performed up until this point have all focused on the questions. These questions have now been extensively classified and the proper answer types identified. The questions have further been converted to scope queries and expanded with relevant terms likely to improve retrieval. From now on, the rest of the steps focus on the answers:

1. Passages of information relevant to the query must be retrieved for inspection. This is the task of the underlying search engine, and will not be discussed further beyond my general presentation in section 2.3.

2. Potential answer candidates must be extracted from these passages. This means answers of the right, general type as identified in question classification. This step is basically also handled by the underlying engine, since it already knows which entities are present in the returned passages.

3. Finally, the answer candidates must be verified as belonging to the right, specific type. They must also occur in valid relations with the search terms within the retrieved texts. In other words, answer types

must be verified semantically, and answer relations syntactically. Both types of verification rely on the information gathered from question classification.

Since my queries are tailored to maximize recall, the task of maintaining precision falls to verification. Both syntactic relation and semantic type warrant their own verification step. Unfortunately, I did not have time to properly explore relations. Semantic type assurance will, however, be discussed shortly in section 4.7.

### 4.6.2 Handling unsupported entities

The ability to extract entities at index time is crucial for answer accuracy. Without indexed entities, answer mining would have to resort to some form of generic noun phrase (NP) extraction. This kind of NP "entity" can't be reasonably indexed as there are simply too many NPs in any given text. Basically, almost every sentence contains an NP. Most of these are hardly useful and would merely clutter the index. This means answer extraction, without entity indexing, would have to be performed in full when processing the results. This, of course, is a lot more costly than doing a major share of the work at indexing time. Also, the more processing performed, the longer the users have to wait for their answers. For this same reason, index-time entity extractors can afford to be painstakingly thorough, while query-time extractors can not. When combined with an extensive semantic lexicon that has been prebuilt offline, this thoroughness translates to both higher accuracy and quicker response. Consequently, the foundation upon which my QA solution is built has a solid advantage compared to traditional pattern-based QA systems.

However, this indexing advantage only applies to supported entity types. If a question asks about an entity type that is not indexed, this type obviously can't be looked up in the index. This means unsupported entities have to be indentified and extracted from the answer passages. But this does not necessarily imply generic NP extraction. Rather, the semantic lexicon can be used as a guideline for extraction. Since only entities contained in the lexicon can be verified, these known instances of a given type can simply be pattern matched against the answer passages. If no match is found, then there is no verifiable entity in the candidate answer anyway. Of course, this rules out potentially correct candidates that are not contained in the lexicon. Still, it's an option that provides both speed and precision. For better recall, though, my solution likely needs a fast, generic NP extractor that can be run

under result processing. My NP heuristics from the hyponym extractor can be modified for this purpose.

### 4.6.3 Returning multiple answers

In contrast with TREC, which has traditionally required single and exact answers, I aim for returning multiple answers with additional supporting context. Basically, my system is designed to treat every question as a list question. This, in my opinion, is a benefit, not a limitation. It allows users to view multiple aspects of a topic and choose interesting answers intelligently, rather than having some algorithm trying to guess for them. The hard part of finding answers is collecting sensible candidates. As I've just described, the system can easily handle this task. Surveying a list of candidates after retrieval does not detract from usability but rather has the potential to enhance it. If the topmost answer is not correct, one of the following ones may be. Lists give users the chance to recognize those answers and even verify their credibility by their contexts. There is a reason ranked results is the established presentation paradigm of search engines.

As I've explained before, though, QA systems work with answers, not just information. Returning passages of text is thus not precise enough, even if the answer entities are highlighted in the passages. Rather, answers need to be exact; i.e. on the entity level, not on the passage level. These two properties are not mutually exclusive, though. Exact answer can be returned with contextual information without cluttering up the results. Fast ESP has built-in functionality in the user interface for this exact kind of purpose; *navigators*. In light of my system, a navigator can basically be thought of as a list of answer entities. When a user picks one of the entities, the full answers will be presented in a ranked list with the entity instances highlighted. This kind of navigation is called *drilling down* and provides a clean, hierarchical way to access the result set.

The entities in a navigator represent an aggregation over all the matches. They are ranked by probability of being correct. This probability has to be computed somehow, though. There are two factors involved in this probability; answer verification and frequency of occurrence. If only dealing with verified answers (i.e. verified semantic type and syntactic relation), answers won't make it to this list unless having been verified. This leaves ranking up to the frequency measure. If, however, unverified answers are also accepted, the list will contain a mix of verified and unverified answers. The verified ones should thus be boosted using some weight to move them higher up in the list. But this weight should be balanced against the frequency count, since an unverified entity that is mentioned often should be considered a

potentially correct answer. The fact that it can't be verified does not mean that it is not correct, merely that it is not supported in the semantic lexicon.

The frequency measure requires some explanation. It is based on the following hypothesis.

### 4.6.4 Trusting redundancy

Shallow parsing often relies on redundancy as a rough form of verification. The underlying hypothesis is that if enough sources state the same belief it can be regarded as truth by virtue of consensus. In other words, potential answers are ranked based on their frequency of occurrence, or "popularity" in the source material. To carry any weight, this hypothesis necessarily requires a large amount of distinct sources. If not, the source repeating itself the most would simply always end up with the winning view. It must be noted, however, that even though many implemented systems rely on this hypothesis for verification, I've yet to locate any satisfying scientific evidence backing it up.

Still, in my case, the hypothesis is fairly intuitive because the retrieval process has made sure all of the potential answer entities appear in a fairly reliable context. Of course, this property is by no means certain. This is the reason for the verification process in the first place. But frequency can, in fact, be useful in directing this verification attempt.

The users can't be expected to suffer long delays while the results are being processed. If there are lots of answer candidates all of them can't be verified thoroughly. In other words, it is crucial to pick the most likely candidates for this treatment. The frequency count of entity instances provides a nice guideline as to which entities are worth processing further.

## 4.7 Semantic verification

Verification of semantics has been mentioned frequently throughout this thesis. It's now time to explain how I intended this to work. I will first present my general take on semantics before I go on to detail my semantic lexicon.

Consider the question *What continent is Argentina on?*. There is no guarantee that an actual answer will contain the word "continent". An answer might instead be phrased as e.g. *Argentina, the second largest country in South America*. It would be wasteful to miss such a perfectly good answer just because the answer type is not mentioned explicitly. QA systems that rely on immense amounts of material (e.g. the web) do exactly this. That is, they throw away many such opportunities in the quest for the perfect answer

(see section 3.3.5). But smaller-scale or specialized QA systems can't afford to be that picky, especially not when trying to establish trustworthiness by collecting and comparing answers.

This need for explicitness is one of the major shackles of current search engine technology. If the answer isn't stated explicitly (i.e. resembling the question in syntax and vocabulary) it simply won't be located. It will then be up to the user to piece together the answer. This explicitness is enforced by a deficiency in semantics. Specifically, a search for answer terms rather than answer types can be viewed as an attempt to compensate for a lack of semantic knowledge. A true semantic search would be able to locate actual instances of the answer type (e.g. continents) rather than just the terms describing the answer type itself. This kind of search is, in fact, within reach. Entity indexing, such as performed by Fast ESP, provides the necessary foundations. The biggest obstacle on the road to main-stream deployment is the level of detail. Or rather, the lack thereof.

The current entity extractors are simply too general, and hence too imprecise. For instance, they do know about continents, in a sense, but only because these are locations. They don't know which locations are continents or, hence, whether a location is a continent or not. Since all locations are treated equally, the system naturally won't be able to differentiate between them. An exact search for a continent is thus impossible. This generalization is intended to maximize recall, and accomplishes just that. For result navigation it is more than adequate. But for QA purposes the precision is just too low. An open-domain QA system won't be very useful unless it covers a large variety of answer types. Lack of variety will mean a lack of ability to answer a lot of different questions. To be truly useful for QA, the level of detail in a semantic search must thus be increased. This is where semantic verification enters the picture.

### 4.7.1 Approaches

Semantic verification can either be regarded as implicit during entity extraction (i.e. when indexing) or explicit during result processing. The former case means running a search for the exact type of entity that the question targets, e.g. a continent. In the latter case, a general search for a location is ran instead. The extracted locations are then verified against known continents (e.g. by using a semantic lexicon) when the results are processed. These two approaches have different challenges that must be overcome:

- Increasing semantic precision in entities requires indexing entities with much more detail than at present. For instance, in a search for *What*

*Canadian city has the largest population?* it is not sufficient to look up a location in the index, but it also has to be a city, and a Canadian one at that. In short, the extracted entities must be denoted with subtypes as well as a general type. When consulting the index it must be possible to query for a type on all of these levels. This puts a lot of requirements on the design of the entity extractors.

- Semantic result processing moves the problem from the extractors to the semantic lexicon. The same search for a city will now be a search for general locations instead. The correct level of detail must then be verified by comparing the extracted locations against known instances in the lexicon. If a location can be determined to be both a city and Canadian (not necessarily both at once) the semantic type of the answer is correct. This solution, of course, requires access to a thorough semantic lexicon. Such a lexicon may not be available.

Each of these two approaches has its advantages and disadvantages. Both will now be discussed in more detail.

### 4.7.2 Semantic entities

If relying exclusively on the extractors for semantics, the level of detail of the extractors roughly equals the semantic precision of the system. One way to increase precision in such an environment is to divide the existing extractors into new, more specific ones. Another is to simply add more extractors. Basically, for each entity type that is desirable to support, an appropriate extractor must be written that locates instances of that particular type. As a possible guideline, Sekine et al. [39] have identified roughly 150 common entity types that occur frequently in general text material. To reach proper coverage, a general-purpose QA system should aim to support a number of extractors on this scale. Of course, that is a lot of extractors. Building them all will require significant effort, especially if writing all the heuristics manually.

But let's say, for now, that this effort is expended and the result is 150 separate extractors. Initially, these might be implemented on the same level as this is easiest to accomplish. This means there is no longer a location extractor. Instead, there are separate extractors for cities, countries, rivers, mountains, continents and so on. This kind of separation applies to all the other entity types as well, resulting in a large multitude of separate extractors. Such a large mass of extractors is likely to be unwieldy and confusing. A search for a general location, for instance, can no longer be performed

without searching for all the separate subtypes that together constitute a location. Clearly, having high precision extractors is not enough. For practical reasons the system needs to be able to handle entities on both a general and a specific scale. This suggests that the extractors should be organized as a hierarchy. In fact, the entities presented by Sekine et al. are already structured in this manner, and this hierarchy could serve as a guideline.

### Hierarchical entities

One way to implement a hierarchy is to abandon the linear extractor approach and instead create hierarchical extractors. This means that the extractors will be kept as general as possible but that each such extractor divides its rules and dictionaries by subtype. To use the location extractor as an example again, there will now be separate dictionaries (with corresponding rules) for cities, countries, rivers, mountains, continents and so on. When running the extractor, all matches will be returned as locations, but they will also be marked with subtype based on which subset of rules that triggered the match. E.g. a city will be marked both as a city and as a location and can be queried upon as both.

The major benefit is that the heuristics of a general type is shared amongst all the subtypes, since they're essentially the same extractor. In other words, there's no redundant matching since the rules don't have to be duplicated across the extractors. To be usable from a scope search though, there will also have to be a corresponding subdivision of types in the index. But since each extractor specifies how its entities will appear in the index, the hierarchy can simply be mirrored in the index. The problem is that the framework doesn't allow an extractor to return more than one entity type, nor does the index support hierarchical types. The solution thus requires some restructuring and rewriting. Also, the solution is not very flexible with regard to subtype division. For instance, a university is a facility, but is it not also an institution and a location?

### Hierarchical configuration

A more flexible solution is to separate the hierarchy from the extractors. This means reverting back to a linear organization of extractors again. The hierarchy still has to be specified somewhere, but this can be accomplished in a separate configuration file. It's important to realize here that the system doesn't use the extractors directly. That is, it doesn't know about the internal workings of the extractors. Rather, the extractors are presented to the system through a common configuration file; the *pipeline*. Here, each extractor is

made addressable as a separate unit. This linear organization can instead be restructured to become hierarchical. In other words, all entities of the same general type can be configured together under a common supertype to make them addressable as a single unit. A city extractor, for instance, would then only match cities, but since a city is a subtype of location in the pipeline, the match will also be marked as a location.

This organization essentially moves the complexity from the extractors to the configuration. The benefit is that changing or adding a subtype is now only a matter of modifying a configuration line. A university extractor, for instance, can now belong to both the facility type, the institution type and the location type. When a school is matched it will be marked as belonging to all these types. This allows a much higher degree of flexibility in how the subtypes can be utilized. Also, there's basically no limit to how many supertypes a subtype can belong to. Of course, some rewriting is still required to support a hierarchy both in the pipeline and in the index. And there's bound to be redundant matching again, since many extractors of the same subtype have to run the same general rules.

#### The problem with entity semantics

Not every concept is suited to be organized in a strict hierarchy. This was exemplified with the university extractor which could fit under several types. Duplicating entries in the configuration hierarchy works to some degree, but it can get quite messy. Sekine's hierarchy suffers from classic problems of classification. Should a planet really be considered a subtype of location? Is a phone number an address? Is an address a location, and hence a phone number a location? Is a vehicle a product? If so, isn't a monument also a product rather than a facility?

So far, only nouns have been discussed for semantics. It gets a lot worse if also considering adjectives. Take for instance a question about a Norwegian city. Clearly, there can't be an extractor (linear or hierarchical) for this kind of entity as it would mean every country would need it's own city extractor. The concept of Norwegian cities should not be a subtype of cities, but rather a facet of cities. That is, it does not belong in a hierarchical relationship with cities but rather as an aspect that some cities share and other don't. Similarly, political organizations could be regarded a facet of organizations, and French rulers a facet of rulers. Adjectives, in general, translate well to the facet concept. The type parameters to the scopes in section 4.5.4 can be regarded as facets since they describe aspects of the scope. The question is how to implement these facets. They clearly don't belong in a hierarchy, but should rather be attributes to the entities. I believe a semantic lexicon

is the proper placement, as the very purpose of such a lexicon is to describe the semantic aspects of each entry. This will be discussed shortly in section 4.7.3.

To summarize, there's quite a bit of work involved if this approach is to be followed. A lot of extractors have to be written and they must be organized hierarchically to be addressable in a practical manner. Even if all of this work is completed the extractors still only represent a small part of all concepts a user may possibly ask about. This leads to the fact that it's not practically viable to use extractors exclusively for semantic precision. There are simply too many semantic classes imaginable. Ultimately, every concept thinkable can be regarded a separate class, and may or may not be queried upon at some time. Implementing all these extractors is clearly not viable.

### 4.7.3 Semantic lexicons

My solution is to keep the extractors as general as possible, thus maximizing recall, while leaving precision to the semantic lexicon. Some extractors may, of course, be practical to divide into separate types. Sekine's entity hierarchy can again serve as a guideline. Note, for instance, that location is a hierarchy while person is not. This is because locations have distinct subtypes, like country and city, while dynamic person roles, like president and developer, are better suited as facets.

Keeping the extractors general and supplementing with a semantic lexicon means adequate coverage can be assured without using a lot of extractors. New extractors can then be written as needed while the rest of the entities are verified against the lexicon. Since specifiers indicate general entity, low precision can be accepted beyond that. There's no need to support every specific entity directly.

The data structure of the lexicon is a plain text file, but it can be organized in several ways depending on what purpose it is to be used for. Since I will use it primarily for verification, it is organized by entity instance. That is, each line in the file starts with a specific entity followed by a comma separated list of every semantic class this entity belongs to. For instance, *Napoleon* is followed by *important figures, great French soldiers, major historical figures, effective military leaders, famous men, ....* The reverse organization is useful if instead needing to find all instances of a given class, e.g. *historical figures* followed by *Napoleon, Nebuchadnezzar, Nero, Newton, ....*

A lexicon is a rather poor substitute for a dedicated extractor, though. An extractor can use quite sophisticated heuristics and contextual clues tailored to each specific entity type. The lexicon, on the other hand, is built using only a small set of common patterns that describe certain syntactic relations.

These will miss all entities that do not occur in one of these relations. There are more advanced ways to build lexicons, though. These use some form of clustering or latent semantics and will likely recognize far more relations, but I've not had the time to explore these options thoroughly.

The biggest disadvantage of using a semantic lexicon is that the entities contained within can't be looked up in the index. In other words, they can't be used to restrict the scope search. Instead, a pure keyword search has to be ran and general noun phrases have to be matched and verified during result processing. Also, the lexicon is not applicable to entity types that can't be verified by a distinct lookup, such as combinations of digits and symbols.

Once built, the lexicon can be organized by entity type. It can then be seen as a replacement for the dictionary part of an extractor. That is, it can be used to identify already known instances of entities in a text by dictionary lookups. This can be performed both at result processing time, i.e. to find known valid matches in a certain passage of text, and at indexing time, i.e. to scan the whole text for these matches and index them. In the latter case, the lexicon would have to be quality-assured manually and interesting entity types hand-picked to avoid indexing every kind of strange relation that might appear.

The procedure I used to build the semantic lexicon will now be described.

#### Building a semantic lexicon

A method based on accurate lexico-syntactic patterns, such as devised by Hearst (section 3.6.2), is ideal for combining with the huge amounts of information processable by a modern IR engine like Fast ESP. As these kinds of patterns are limited by recall, not precision, they benefit from all the data they can get.

Such patterns are all based on recognizing certain combinations of noun phrases with common, but fairly safe, terms indicative of relations between hyponyms and hypernyms. The problem, in this case, is how to accurately determine a noun phrase since I'm neither using a POS tagger nor a syntactic parser. I wanted to explore what is possible to achieve with powerful pattern matching alone. Without any POS tagging I had to resort to contextual information to try to determine the boundaries of a noun phrase.

I thus, once again, used the procedure I'd conceived for constructing efficient patterns (section 4.3.2) to analyze what constituted a noun phrase.

### Noun phrase extraction

Traditional NP detection methods use some variant of POS tagging. While this achieves high accuracy, it might be considered overkill for the sole purpose of detecting NPs. After all, what is the purpose of classifying every word in a sentence when only the nouns are needed? As long as NP detection is maximized it doesn't matter whether the other POSes are identified correctly or not.

A method based on dictionaries and regular expressions is bound to be much faster (and far simpler to implement) than a full POS tagger. The Matcher framework (section 4.3.1) of Fast ESP is ideal for this task. Through my classification patterns, I've already demonstrated how NPs can be recognized primarily by using dictionaries given proper context.

In this case, the context is provided by the syntactic patterns that constitute a valid relation. I won't be extracting NPs from random parts of a text but only from those syntactic positions where NPs are likely to occur. In other words, I don't need a separate NP extractor when building the lexicon, because NPs can be identified directly through the hyponym extractor.

It's also useful to realize that only NPs identified in this manner are interesting anyway. These are the only ones which have semantic support and can thus be verified. This means that since the semantic lexicon is built from a given corpus in advance, it already contains all the semantic classes that can be verified. Instead of extracting general NPs from a potential answer and verifying it against the lexicon, one might just as well reverse the process and match known instances from the lexicon against the answer. This alleviates the need for general NP extraction in the result processing step also, and integrates it into the verification step. Of course, this kind of matching requires a lexicon organized by semantic class (i.e. "these are emperors") rather than instances (e.g. "Napoleon is an emperor"), since every known instance of a class will have to be matched against the answer.

### Hyponym extraction

For the purposes of building my semantic lexicon, I created a combined hyponym and hypernym extractor based on Hearst's patterns. I unified her 6 patterns into 4 and added a few variations of my own. These 4 patterns are:

1. *$hypernym such as $hyponyms*. E.g. *the bow lute, such as the Bambara ndang*. The trigger is "such as".

2. *such $hypernym as $hyponyms*. E.g. *such authors as Herrick, Goldsmith and Shakespeare*. Again, the trigger is "such as", but the phrase is

interjected by the hypernym.

3. *$hypernym $sub phrase $hyponyms*. E.g. *all common-law countries, including Canada and England* and *most European countries, especially France, England and Spain*. Hearst used separate patterns for these two examples, one triggered by "including" and the other by "especially". But since the syntax is the same, and they only differ in the word that indicates the sub phrase, I unified them into one. I also added a few triggers of my own, namely "*most? notably*" and "*particularly*", because I observed frequent occurrencies of these terms when testing the extractor. I did not have time to evaluate how these additional triggers influence precision and recall values compared to the others though.

4. *$hyponyms (and|or) other $hypernym*. E.g. *wounds, broken bones or other injuries* and *temples, treasuries, and other important civic buildings*. Again, these were originally separate patterns, triggered by "and other" and "or other" respectively. For runtime efficiency I saw no reason to keep them separate.

Note that there's an important difference between mining for general classes (e.g. an emperor is a ruler) and specific instances (e.g. Napoleon is an emperor). The difference can often be recognized by whether the head noun is proper or common, e.g. capitalized or not. When building my lexicon I was interested in both cases and treated these as equal.

For this extraction to work I needed to recognize many different items in the text, including NPs and other POS classes (to find the syntactic boundaries of the NPs). I therefore built an auxiliary extractor to contain various common and useful multi-purpose patterns. This auxiliary file was also used extensively in the question classification extractors.

The main aim of the hyponym extractor is to find NPs that appear in the relations indicated by the patterns above. Recognizing an NP properly without POS tagging is a challenge since many words have multiple senses. Consider the sentence *Marsupials such as kangaroos carry their offspring in a pouch*. Here, the hypernym is "marsupials" and the hyponym "kangaroos". However, the word "carry" can also be a noun, namely "a carry" which refers to the means by which something is carried. An indiscriminate pattern would thus capture "kangaroos carry" as the NP rather than just "kangaroos". This behavior can be avoided by, e.g. recognizing that the possessive pronoun "their" can't follow a noun. This won't help in the case of *Marsupials such as kangaroos carry offspring in a pouch*, though. Here, the NP is in danger of

becoming "kangaroos carry offspring", since all the words are nouns. This is only one of many traps that must be avoided for proper extraction.

The same kind of problem surfaces when matching adjectives. For instance, in *Microsoft purchased sites* it's not easy to know whether "purchased" is an adjective or a verb. In a strict grammatical sense the adjective should be specified as "Microsoft-purchased", but people are rarely strictly grammatical.

It's also difficult knowing how much of an NP should be matched. Consider the sentence *searchable databases for the internal pages of large sites that are dynamically created, such as the knowledge base on the Microsoft site*. The hypernym doesn't have to contain all the words in front of the comma to be useful, it's enough with "searchable databases". On the other hand, if following the same rule on the hyponym, the resulting relation becomes *searchable databases, such as the knowledge base*. Knowing that a knowledge base is a searchable database is certainly useful, but it might be even more useful knowing that this applies to the base on the Microsoft site. It depends on whether mining for general relations or specific instances. Also, capturing the first part of a hypernym NP is detrimental in the case of *the most recent generation of search engines such as Google*. Here, the interesting NP is "search engines" not "the most recent generation". If following this rule for the first example, the hypernym would become "large sites" rather than "knowledge base".

Further, conjunctions and comma-separated lists are challenging, since conjunction (or comma) can indicate both the next element in the list or a new sub phrase entirely. Take for instance *large data producers such as the U.S. Census Bureau, Securities and Exchange Commission, and Patent and Trademark Office*. Here, "Securities" is not a separate element from "Exchange", nor is "Patent" from "Trademark". They must be recognized as parts of their respective NPs. Another example is *languages such as French and cities such as Toulouse*. Here, "French" is separate from "cities". In fact, the latter is a hypernym of a new relation. This situation can be avoided by disallowing the last potential hyponym if it's followed by a word that indicates a new relation. Also, it might be useful recognizing proper noun hyponyms and common noun ones separately.

Finally, the extractor is bound to capture many useless relations such as this example: *useless relations such as this example*. The hypernym becomes "useless relations" and the hyponym "this example". Even if knowing what "this" referred to, it's not very valuable. Another example is *elements on the page, such as lists*. A list can certainly be described as an element, but so can most other things.

I've now mentioned a few of the problems I had to deal with when building the hyponym extractor. Some of these issues were handled satisfactorily while others were not. No matter how much work is put into fine-tuning the heuristics, an automatically created lexicon will never be as solid as a manually compiled one. In the end, though, it doesn't really matter. Most of this noise will likely not be used anyway. It doesn't matter how strange or erroneous some of the relations are as long as no one asks about them. When building the lexicon, it's thus better to maximize useful relations than to minimize useless ones.

### Areas of use

Both the hyponym extractor and the resulting semantic lexicon are useful for many purposes:

- When the general answer type is supported by an extractor, but not the specific type, candidate entities in the results can be verified against the lexicon to also assert the specific type.
- When the general answer type is unsupported, candidate entities must be extracted from the results. The lexicon can then be used to verify general NPs believed to be answers. Or better yet, to help locate general NPs of the right type.
- The lexicon can be consulted both for answering instance-specific questions (e.g. *Which emperor lost at Waterloo?* yielding *Napoleon*) and concept-specific questions (e.g. *Who is Napoleon?* yielding *important military leader*). Noise is far more problematic for the latter case, since it entails presenting this information to the user rather than merely using it for verification. Each instance should thus be marked with a frequency count to indicate reliability. That is, the more often a specific hypernym/hyponym relation occurs, the more reliable it is.
- The lexicon could be used to expand queries instead of just to verify answers. This requires better quality assurance. Each hyponym could be included in a boolean "OR" separated list as candidates for the hypernym.
- The relationships that the hyponym extractor discovers are also useful for expanding the dictionaries of existing extractors, e.g. by adding extracted authors. Further, interesting relations could even provide the foundation for creating an entirely new extractor. For instance, if

the hypernym "toxic substances" turns out to have many associated hyponyms.

- The extractor can likewise be used to populate specifier dictionaries. For instance by discovering various terms that are used to describe locations. The relations between connected terms will eventually become a hierarchy, e.g. a king is a ruler, an emperor is a ruler, a ruler is a person, which means both a king and an emperor are persons. By following the links in such a hierarchy, related terms can be mined from the lexicon.

Due to the limited quality of the automatically generated lexicon, it should not be used to discard unverified entities but rather to boost verified ones. That is, if an entity can't be confirmed to belong to a subtype through the lexicon, that does not mean that the entity is not an instance of that subtype. It merely means the lexicon doesn't contain such a relation. The entity might still be a good answer. But if the entity can in fact be confirmed against the lexicon it should be boosted so that it ranks above all the entities that could not be confirmed in the resulting list of answer entities. After all, if an entity belongs to the correct general type (e.g. a person) the exact subtype may not be critical.

#### Extractor code

The hyponym extractor configuration file is included in appendix A.4. Like with the person classifier, most of the work is handled by the auxiliary patterns in appendix A.5. The reference file used for testing the extractor is too big to be included. For an example, refer instead to the time extractor reference file in appendix A.2.

After testing the hyponym extractor on several smaller texts, I felt confident enough to try it out on a larger scale. I ran it on a local crawl of Wikipedia, which resulted in 101576 hyponyms, each with at least one hypernym. As an example of output, I've included some random chunks from the lexicon in appendix A.6. The output is organized by hyponym.

### 4.7.4 Unit-of-measurement verification

The lexicon I've built can be used to verify known entity instances but is not applicable on measurements. E.g. "Napoleon" can easily be looked up as an emperor, but "165.83 kg" can't be verified against the lexicon as a kilogram measure. In the latter case, however, only the unit of measure ("kg") is interesting to verify, as the number itself is valid simply by being a number.

Verifying the right measure is not particularly difficult. Since there is currently only one measurement extractor (i.e. not separate extractors for e.g. weight and length), it must be run for every measurement question. Answer verification then means accepting only measurements with the right unit. If the specific unit (e.g. kilograms) is given in the question, it simply means checking against all versions of this unit (i.e. "kilograms" and "kg"). These versions can be retrieved from a corresponding hash-type dictionary organized by unit. E.g. where the key "kilograms" contains the values "kilograms" and "kg". It's also possible to accept other appropriate measures and convert these to the given unit, as described in section 4.4.3 on measurement patterns. If the question instead asks for a general measure (e.g. weight), it's again a matter of checking the unit against values corresponding with the key "weight". This same approach is not only applicable to measurements, but can also be used on currencies, quantities and frequencies.

## 4.8 Future enhancements

There were many areas and techniques I would have liked to explore further. This section will present the most important ones:

- *Answer extraction.* This is the main feature I had hoped to utilize properly. Particularly relation verification. This would have allowed me to test and verify my assumptions regarding answer correctness and present the results properly formatted. I briefly tested my assumptions manually on a handful of texts and the results showed promise. There are lots of answers showing the same kind of relational patterns as presented in example 4.1 in section 4.2.1.

- *Fall-back patterns.* My question classification patterns should have a general fall-back step to handle unsupported syntax. This would also allow short-hand questions like *what famous monument in new york* or even *famous monument new york*. These can be mapped to appropriate extractors without requiring the users to supply every stopword in between. Of course, these "questions" will not be as precise, and are likely to result in multiple interpretations. But my suggestion for a ranked list of entity types can nevertheless be used to cleanly present all the resulting entities for each of the possible interpretations of a particular sequence. Since they are interpreted as contracted questions, more meaning can be induced that simply treating them as keyword queries.

- *Semantic lexicon expansion.* My lexicon could be enhanced further by adding CN/PN (common noun/proper noun) patterns to the hyponym

extractor. These patterns extract the common noun in front of a proper noun. E.g. a "president Clinton" statement defines Clinton as a president. Fleischman [16] notes that CN/PN patterns occur roughly 40 times more often than the Hearst patterns, and could thus provide a solid boost to recall. Further, semantics can be mined from questions too. For instance, a question often reveals the entity type of its nouns. E.g. "Who is Galileo" reveals that Galileo is a person. Retrieving lists of questions (e.g. FAQs) and mining these using reliable patterns could yield many valuable relations. This approach can be used to enhance the lexicon with specific topics by selective mining question collections.

- *Anaphora resolution.* It would be most beneficial to recognize a valid answer across multiple sentences. This requires resolving the pronoun back-references. For instance, an answer to *When was John Lennon born?* could thus recognize that "he" in *He was born in 1940* means "Lennon" and not "Cartney" in the previous sentence *Lennon was a friend of Cartney*. This functionality is also useful for questions. For instance, *Who said that on the show?* can't be answered without knowing what "that" and "the show" refers to.

- *Deep NLP methods.* A future Fast ESP release will be integrated with a complete linguistic processing framework. It will then be possible to work directly with grammars rather than patterns. It's also possible to address specific attributes of words and capture precise syntax. This framework provides the foundation for quite sophisticated NLP approaches to QA. A corresponding solution would be most interesting to compare against my pattern approach. Without this framework, though, similar processing requires separate steps that have to be tied together through some custom controlling module. It's thus better to wait until the framework is live.

- *POS tagging.* I would have liked to compare the POS tagging ability of my question classification patterns with an approach based on full POS tagging. Fast ESP provides a good Viterbi tagger and variants of Brill's tagger are freely available. These have to be trained on a massive amount of appropriate tagged text though.

- *General questions.* The system should be able to respond to general questions. That is, provide a compiled summary of the most relevant facts given a topic. Since users tend to ask general questions when they don't know exactly what they are looking for, this could hopefully provide all the information they need in one chunk.

- *Additional information.* Entity indexing allows providing additional information to answer "other" questions, as I suggested in 4.2.5. That is, the system could be enhanced to infer additional questions from the one given and return a coherent answer with all the related information tied together.

I had also hoped to get more of the implementation ready. Even though my system is not complete I believe that my prototype can serve as a solid basis for future work:

- The necessary requirements for a complete QA system have been researched and presented. There is no need to go through this work again. Rather, the more of these requirements are fulfilled and implemented, the closer the system will get to completion. A major part of the system is ready for implementation and experimentation.

- A solid analysis of question classification has been presented with corresponding patterns. These can already cover a wide variety of questions and can easily be expanded further with the included pattern syntax. Especially since I've detailed the procedure I followed when analyzing questions and have presented thorough examples.

- I've explained how answers can be verified satisfyingly and outlined the process. This step requires research before implementation, but can utilize my hyponym extractor to build appropriate lexicons. The question patterns can also be enhanced to capture more semantics, as demonstrated.

- More entity extractors of all kinds can be written using the procedure I devised. My examples serve as a guideline to the thought process required to build them successfully. Every additional extractor will increase the answering ability of the system.

# Chapter 5

# Conclusion

**Achievements.** I have created a Question Answering prototype as a layer on top of a state-of-the-art, contextual search engine. This fusion produces a system that is able to accept questions in natural language and translate them to queries optimized for the engine. Through the presentation of my solution, I have shown how to create a few simple, unified patterns that are able to recognize a wide variety of questions. I have also shown how to write patterns that capture specific semantics. This information can be utilized to better attune the queries to the engine and to find answers more reliably. Further, I've demonstrated how answers can be verified both syntactically and semantically. Specifically, I've shown how to build a semantic lexicon useful for this purpose. The same method can be used to mine semantic links between the entity types of the question and the answers. By using the results from this procedure, answers can be verified to correspond with both a general type of entity and with specific facets indicative of any particular subtype. Answers can thus be ranked according to semantic confidence.

**Limitations.** My solution is based on manually created heuristics. It would have been interesting to compare the performance of these rules with probabilistic, statistical or machine-learning methods. In general, rule-based methods seem to be adequate for simple tasks. They even have the potential to outperform automated ones, e.g. in the case of compiling semantic lexicons. Automated methods, however, seem more appropriate for cases of unpredictable data or overwhelming complexity. Even though I would not have had the time to implement multiple solutions, I could have compared my system against the contenders in the TREC conference.

Unfortunately, I did not have the time to implement all the steps necessary for a complete QA process. There are several reasons for this:

- It took long to focus my work and determine where the effort should be spent. I did not achieve an equal balance between writing and implementing. Nor did I recognize soon enough the extent of work required for the implementation to function.
- I spent much time on researching and comparing the necessary foundations for a complete system rather than picking a specific approach and starting to implement. In the end, I had to concentrate all my efforts on interpreting questions and abandoned answers.
- There were several technical problems that had to be resolved in order to get the system to run as intended. Some of these required fixing issues with the Fast ESP search platform that I did not have control over. This is a disadvantage of basing a solution on an external resource. I did not prioritize creating work-arounds to get the system to run.

Shortly put, I was not able to realize a complete solution, nor did I expect to be able to. As I explained in my design goals (section 4.1.1), I focused on research and design, not on implementation details. Consequently, I can't properly verify my assumptions nor produce tangible results as evidence of the legitimacy of my approach. Still, I feel that I've presented my solution clearly and argued my points logically. I've also shown support for my views through references to other works.

To create a test bed for answer extraction, I indexed a local crawl of Wikipedia. Sadly, I did not get to utilize it for that purpose. It did, nevertheless, prove useful as a basis for extracting hyponyms for the semantic lexicon. While I can not present results in the form of answers, I do present an excerpt from this lexicon to show what kind of semantic knowledge it is able to recognize (appendix A.6).

**Discoveries.** I've learnt that factoid answering isn't too difficult to implement. Once implemented, it provides a solid foundation for answering harder questions. Further, since the number of different question types is limited, these can be successfully categorized. Fairly simple patterns can then be constructed to classify questions and locate answers given an adequate amount of text material.

Though my question classification focus has been on the TREC questions, I recognize that these are not representative of questions in general. Since my system has only been tailored to answer these kinds of questions, it will necessarily require more work to function comparably on other sets of questions. The TREC questions are, however, easily recognizable as common ways to phrase questions. Also, most of the examples I came across

in solutions unrelated to TREC fell naturally into one of the existing categories. Further, my experience with writing patterns makes me confident that additional question sets will display many similarities. This will make them fairly easy to also generalize into unified patterns.

Some questions might not map directly to an extractable entity. This is especially the case for open-class questions. But it's nevertheless useful to analyze the question properly to identify the semantic target. This might help locate constructs that indicate probable answers in the documents. Even if an answer can't be pinpointed, the search can at least be tuned to accompany the type of question asked. For instance, by expanding the query with helpful constructs or by preferring certain sources when ranking the results. Such behavior is still an improvement over pure keyword searching.

Finally, I recognize that not all information inquiries are easy to express as explicit questions. QA is thus not a good solution for every type of information access. I do not intend it as a full replacement for IR and IE either, but rather as a useful addition for many situations. Plain queries still have their uses in general information gathering.

**Benefits.** With new approaches like fact mining services (e.g. Google Q&A), work of the kind I've performed on question classification is as important as ever. The focus, answer type and semantics of questions have to be determined regardless of how, and from where, the answers are extracted. Answers may, for instance, just as well be mined from specific structures on certain pages as from unstructured text. They can even be looked up in appropriate knowledge bases. In this regard, question classification is arguably more important than answer extraction, and I've spent my efforts accordingly.

My implementation is not web-based nor tailored exclusively for information access on the large scale provided by the Internet. I intended my solution to be useful also on smaller collections and for purposes beyond searching the web. This meant crafting patterns that would help assure reliable answers even if those answers did not resemble the questions. It also meant sacrificing some precision to increase the recall. The same line of thought went into building the semantic lexicon. But there it is always beneficial to have as much information available as possible, due to the high-precision, low-recall hyponym patterns. However, the lexicon can easily be supplemented with data independently from the corpus intended for answer mining.

A method based on heuristics is necessarily language dependent. A language processing tool that is not somewhat language dependent is, after all, rather paradoxical. But I've tried to assure that my solution at least has

loose language dependence. It is not heavy on rules, and the unified patterns are few and composed of a small number of atomic parts. Most likely it would thus be fairly easy to create similar heuristics for another language for someone proficient in the particular language.

**Final words.** Coming from computer science, rather than computational linguistics, has proven challenging in many areas of this work. But it has also allowed me to explore solutions from a computer science perspective instead of relying solely on established linguistic methods. This has primarily manifested itself in my pattern matching approach to question classification. Instead of parsing sentences syntactically, I've analyzed them from a pure pattern perspective. This has achieved many of the same benefits while avoiding the associated costs.

There are fields besides QA I could have explored that are more commercially viable. But I wanted to use my master's thesis as an opportunity to pursue a topic that might not be granted time and money in a commercial setting — a topic that might nevertheless prove valuable. And as I have shown, the interest in QA is definitely not merely academic. There have been several attempts at commercialization, some by major players within the field of search. But regardless of the potential for profit, research in QA has value in itself because of the benefits it promises to bring to people's daily quests for answers.

From what I've learned through this work, I sincerely believe that QA done right will enhance the user experience. It can bring efficient acquisition of knowledge to an entirely new level. But a solution has to build upon the strengths inherent in computer processing. It should not try to compete with the human mind in its ability to reason and comprehend. QA is primarily helpful as a tool of convenience, not as a replacement for thought.

# Bibliography


[1] Agichtein, E., Lawrence, S., and Gravano, L. Learning to Find Answers to Questions on the Web. *ACM Transactions on Internet Technology 4*, 2 (May 2004), 129–162.

[2] Ahlswede, T., and Evens, M. Parsing vs. Text Processing in the Analysis of Dictionary Definitions. In *Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics* (State University of New York at Buffalo, Buffalo, New York, USA, June 1988), pp. 217–224.

[3] Baeza-Yates, R., and Castillo, C. Crawling the Infinite Web: Five Levels are Enough. In *Proceedings of the third Workshop on Web Graphs (WAW)* (Rome, Italy, October 2004), Springer LNCS. (To appear).

[4] Baeza-Yates, R. A., and Ribeiro-Neto, B. A. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[5] Banko, M., Brill, E., Dumais, S., and Lin, J. AskMSR: Question Answering Using the Worldwide Web. In *Proceedings of 2002 AAAI Symposium on Mining Answers from Texts and Knowledge Bases* (March 2002).

[6] Bergman, M. K. The Deep Web: Surfacing Hidden Value, July 2000.

[7] Brill, E. *Processing Natural Language without Natural Language Processing*. Springer-Verlag Berlin Heidelberg, 2003.

[8] Brill, E., Dumais, S., and Banko, M. An Analysis of the AskMSR Question-Answering System. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing* (2002).

[9] Brill, E., Lin, J., Banko, M., Dumais, S., and Ng, A. Data-Intensive Question Answering. In *Proceedings of the Tenth Text REtrieval Conference (TREC 2001)* (2001).

[10] Brin, S., and Page, L. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks and ISDN Systems 30*, 1-7 (1998), 107–117.

[11] Burke, R., Hammond, K., Kulyukin, V., Lytinen, S., Tomuro, N., and Schoenberg, S. Question Answering from FAQ Files: Experiences with the FAQ Finder System. *AI Magazine*, 18(2) (1997), 57–66.

[12] Cao, J., Robles-Flores, J. A., Roussinov, D., and Jr., J. F. N. Automated question answering from lecture videos: Nlp vs. pattern matching. In *Proceedings of the 38th Hawaii International Conference on System Sciences* (2005).

[13] Clarke, C. L. A., Cormack, G. V., and Lynam, T. R. Exploiting Redundancy in Question Answering. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2001)* (2001), pp. 358–365.

[14] Clifton, T., and Teahan, W. Bangor at TREC 2004: Question Answering Track. In *Proceedings of the Thirteenth Text REtrieval Conference (TREC 2004)* (2004).

[15] Dumais, S., Banko, M., Brill, E., Lin, J., and Ng, A. Web Question Answering: Is More Always Better? In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2002)* (August 2002).

[16] Fleischman, M., Hovy, E., and Echihabi, A. Offline Strategies for Online Question Answering: Answering Questions Before They Are Asked. In *ACL '03: Proceedings of the 41st Annual Meeting on Association for Computational Linguistics* (2003), Association for Computational Linguistics, pp. 1–7.

[17] Gulli, A., and Signorini, A. The Indexable Web is More than 11.5 Billion Pages. In *WWW 2005* (2005).

[18] Hane, P. J. FAST Has the ESP for Enterprise Search. http://www.infotoday.com/newsbreaks/nb060206-1.shtml, 2006.

[19] Hearst, M. A. Automatic Acquisition of Hyponyms from Large Text Corpora. In *Proceedings of COLING-92* (1992), pp. 539–545.

[20] Hermjakob, U. Parsing and Question Classification for Question Answering, 2001.

[21] Hirschman, L., Light, M., and Breck, E. Deep Read: A Reading Comprehension System. In *Proceedings of the 37th Annual Meeting of the Assoc. for Computational Linguistics* (June 1999).

[22] Hull, D. A. Stemming Algorithms: A Case Study for Detailed Evaluation. *Journal of the American Society of Information Science 47*, 1 (1996), 70–84.

[23] Jeon, J., Croft, W. B., and Lee, J. H. Finding semantically similar questions based on their answers. In *28th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR '05)* (2005), ACM Press, pp. 617–618.

[24] Katz, B. Using english for indexing and retrieving, 1988.

[25] Katz, B. Annotating the World Wide Web using Natural Language. In *In Proceedings of the 5th RIAO Conference on Computer Assisted Information Searching on the Internet (RIAO '97)* (1997).

[26] Katz, B., Lin, J., Loreto, D., Hildebrandt, W., Bilotti, M., Felshin, S., Fernandes, A., Marton, G., and Mora, F. Integrating Web-based and Corpus-based Techniques for Question Answering. In *Proceedings of the Twelfth Text REtrieval Conference (TREC 2003)* (November 2003).

[27] Kosseim, L., Plamondon, L., and Guillemette, L.-J. Answer formulation for question-answering. In *Advances in Artificial Intelligence, 16th Conference of the Canadian Society for Computational Studies of Intelligence, AI 2003, Halifax, Canada, June 11-13* (2003), Y. Xiang and C. draa Brahim, Eds., vol. 2671 of *Lectures Notes in Artificial Intelligence*, Springer-Verlag, pp. 24–34.

[28] Kwok, C. C. T., Etzioni, O., and Weld, D. S. Scaling question answering to the Web. In *World Wide Web* (2001), pp. 150–161.

[29] Lin, J., and Katz, B. Question Answering from the Web Using Knowledge Annotation and Knowledge Mining Techniques. In *Proceedings of the Twelfth Conference on Information and Knowledge Management (CIKM 2003)* (November 2003).

[30] Lin, J., Quan, D., Sinha, V., Bakshi, K., Huynh, D., Katz, B., and Karger, D. R. What Makes a Good Answer? The Role of Context in Question Answering. In *Proceedings of the Ninth IFIP TC13 International Conference on Human-Computer Interaction (INTERACT 2003)* (September 2003).

[31] Moldovan, D., Harabagiu, S., Girju, R., Morarescu, P., Lacatusu, F., Novischi, A., Badulescu, A., and Bolohan, O. LCC tools for Question Answering, 2002.

[32] Pantel, P., and Lin, D. Discovering word senses from text. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining* (New York, NY, USA, 2002), ACM Special Interest Group on Knowledge Discovery in Data, ACM Press, ISBN:1-58113-567-X, pp. 613–619.

[33] Pantel, P., and Ravichandran, D. Automatically Labeling Semantic Classes. In *HLT-NAACL* (2004), pp. 321–328.

[34] Porter, M. F. An algorithm for suffix stripping. *Program 14*, 3 (1980), 130–137.

[35] Price, G. Before Google Answers and Yahoo Answers there was "Answer Point" from Ask Jeeves, December 2005.

[36] Radev, D. R., Libner, K., and Fan, W. Getting Answers to Natural Language Questions on the Web. *Journal of the American Society for Information Science and Technology 53*, 5 (2002), 359364.

[37] Riloff, E. *Information Extraction as a Stepping Stone toward Story Understanding*. MIT Press, 1999, ch. 13.

[38] Risvik, K. M., Mikolajewski, T., and Boros, P. Query Segmentation for Web Search. In *WWW (Posters)* (2003).

[39] Sekine, S., Sudo, K., and Nobata, C. Extended Named Entity Hierarchy. In *Proceedings of the LREC-2002* (2002).

[40] Snow, R., Jurafsky, D., and Ng, A. Y. Learning syntactic patterns for automatic hypernym discovery. In *Advances in Neural Information Processing Systems 17* (2005).

[41] Soricut, R., and Brill, E. Automatic Question Answering: Beyond the Factoid. In *Proceedings of the Human Language Technology and*

*North American Association for Computational Linguistics Conference (HLT/NAACL-2004)* (May 2004).

[42] Voorhees, E. M. Overview of the TREC 2001 Question Answering Track. In *Proceedings of the Tenth Text REtrieval Conference (TREC 2001)* (2001).

[43] Voorhees, E. M. Overview of the TREC 2002 Question Answering Track. In *Proceedings of the Eleventh Text REtrieval Conference (TREC 2002)* (2002).

[44] Voorhees, E. M. Overview of the TREC 2003 Question Answering Track. In *In Proceedings of the Twelfth Text REtrieval Conference (TREC 2003)* (2003).

[45] Voorhees, E. M. Overview of the TREC 2004 Question Answering Track. In *In Proceedings of the Thirteenth Text REtrieval Conference (TREC 2004)* (2004).

[46] Voorhees, E. M. Overview of the TREC 2005 Question Answering Track. In *In Proceedings of the Fourteenth Text REtrieval Conference (TREC 2004)* (2005).

[47] Voorhees, E. M., and Harman, D. K. *TREC: Experiment and Evaluation in Information Retrieval*. Digital Libraries and Electronic Publishing. MIT Press, September 2005.

[48] Web. Ask. http://www.ask.com.

[49] Web. Ask for Kids. http://www.askforkids.com.

[50] Web. CIA World Factbook. http://www.cia.gov/cia/publications/factbook.

[51] Web. Dictionary.com. http://www.dictionary.com.

[52] Web. eXtended WordNet. http://xwn.hlt.utdallas.edu/.

[53] Web. Fast ESP. http://www.fastsearch.com.

[54] Web. Google. http://www.google.com.

[55] Web. Google Answers. http://answers.google.com.

[56] Web. Info Angels. http://www.infoangels.com.

[57] Web. Internet Movie Database. http://www.imdb.com.

[58] Web. KNOVA. http://www.kanisa.com.

[59] Web. Perl Compatible Regular Expressions. http://www.pcre.org.

[60] Web. TREC 2006 tracks. http://trec.nist.gov/tracks.html.

[61] Web. Wikipedia. http://www.wikipedia.org.

[62] Web. WordNet. http://wordnet.princeton.edu/.

[63] Web. Yahoo! Answers. http://answers.yahoo.com.

[64] Zhou, G., and Su, J. Named Entity Recognition using an HMM-based Chunk Tagger. In *ACL '02: Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* (2001), pp. 473–480.

# Appendix A

# Extractor examples

## A.1 Time Extractor configuration

```
<?xml version="1.0"?>

<!-- Copyright (C) 2006 Fast Search &amp;amp; Transfer ASA -->

<configuration>
  <matcher type="pattern" debug="no" verbose="no">
    <pattern flags1="0" flags2="0">

      <!-- Include common and reusable patterns. -->
      <include namespace="" filename="etc/resources/matching/configuration.
          aux.xml"/>

      <!-- Include common UTF-8 symbols. -->
      <include namespace="" filename="etc/resources/matching/configuration.
          symbols.aux.xml"/>

      <!-- Digits followed by one of these signs are not times. -->
      <slave name="sign_prefix">
        <schema><![CDATA[(?<!(\\$|#|\.|:))(?<!($pound|$sect))(?<!($euro))]]>
            </schema>
      </slave>

      <!-- Hours with one or two digits. -->
      <slave name="hours">
        <schema>$sign_prefix((0?\d)|(1\d)|(2[0-3]))</schema>
      </slave>

      <!-- Hours with two digits. -->
      <slave name="hours_2digit">
        <schema>$sign_prefix((0\d)|(1\d)|(2[0-3]))</schema>
      </slave>

      <slave name="minutes">
        <schema>[0-5]\d(?!\d)</schema>
      </slave>

      <slave name="seconds">
        <schema>$minutes</schema>
      </slave>
```

```
<slave name="am_pm">
  <!-- Match AM/PM with or without punctuation. -->
  <schema>(([ap]m)|([ap]\. ?m\.?))(?![a-z])</schema>
</slave>

<!-- Dictionary with timezone abbreviations, e.g. "EST", "PST".-->
<slave name="timezoneabbr">
  <schema>[A-Za-z]{2,4}</schema>
  <acceptor>
    <matcher type = "levenshtein" debug = "no">
      <levenshtein automaton = "resources/dictionaries/matching/
          timezoneabbr.aut"
        phonetics = "no" exact = "no"
        match = "complete" start = "beginning"
        prefix = "0" tolerance = "0">
        <quality type = "raw" penalty = "1" threshold = "0"/>
      </levenshtein>
    </matcher>
  </acceptor>
</slave>

<!-- Dictionary with timezone names, e.g. "Eastern Standard", "
    Mountain".-->
<slave name="timezonename">
  <schema>[A-Za-z ]+</schema>
  <acceptor>
    <matcher type = "levenshtein" debug = "no">
      <levenshtein automaton = "resources/dictionaries/matching/
          timezonename.aut"
        phonetics = "no" exact = "no"
        match = "complete" start = "beginning"
        prefix = "0" tolerance = "0">
        <quality type = "raw" penalty = "1" threshold = "0"/>
      </levenshtein>
    </matcher>
  </acceptor>
</slave>

<slave name="timezone" meta="yes">
  <schema>\b($timezoneabbr|$timezonename( [Tt]ime)?)\b</schema>
</slave>

<!-- A designator is either am/pm or a timezone, and follows the time.
    -->
<slave name="designator_">
  <schema>(($am_pm($ws$timezone)?)|($timezone))</schema>
</slave>

<slave name="designator">
  <!-- Match a designator by itself OR a designator surrounded by
      punctuation. -->
  <schema>($designator_|($left_punctuation$ws?)?$designator_($ws?$
      right_punctuation)?)</schema>
</slave>

<!-- Keywords used to indicate that a time follows next. -->
<slave name="keyword_prefix" visibility="none">
  <!-- The prefix word can be trailed by another word, e.g. "after
      PRECISELY". The trailing word/space is matched here to keep it
      ghosted.-->
  <schema>(?=[abftu])(a(fter|t|round|s of)|b(efore|etween|y)|from|to|
```

```
      until)$ws($word$ws)?</schema>
</slave>

<!-- Times specified with a colon, e.g. "10:30". -->
<slave name="time_colon">
  <schema>(($hours ?: ?$minutes)( ?: ?$seconds)?)</schema>
</slave>

<!-- Times specified with a dot, e.g. "10.30". -->
<slave name="time_dot">
  <schema>($hours ?\. ?$minutes)</schema>
</slave>

<!-- Times specified concatenated, e.g. "1030". -->
<slave name="time_cat">
  <schema>($hours_2digit$minutes)</schema>
</slave>

<!-- Military hours, e.g. "1030 hours". -->
<slave name="time_cat_hours">
  <schema>($time_cat$ws?($ws?$more$ws?$time_cat$ws?)*$ws?hours)</
      schema>
</slave>

<!-- Words directly translatable to times. -->
<slave name="time_names">
  <schema>(12 )?(noon|midnight)</schema>
</slave>

<slave name="weekdayword">
  <schema>(?=[mtwfs])((mon|t(ue|hurs)|wednes|fri|s(atur|un))day)</
      schema>
</slave>

<slave name="dayword">
  <schema>(?=[ty])(to(day|night|morrow)|yesterday)</schema>
</slave>

<slave name="timeword">
  <schema>(?=[maen])(morning|afternoon|evening|night)</schema>
</slave>

<slave name="timeword_postfix">
  <schema>($sub?$ws?($filler|$dayword)$ws?$timeword)</schema>
</slave>

<!-- Characters indicating listings, e.g. "10:30, 12:30 and 13:30 pm".
    -->
<slave name="more">
  <schema>(/|,|($ws)and($ws)|($ws)or($ws)|\+|\x26|($ws)to($ws)|\-)</
      schema>
</slave>

<!-- Filler words between times and their timewords, e.g. "this" in "
    10 this evening".-->
<slave name="filler">
  <schema>(?=[itobln])(in|th(e|is|at)|on|by|last|next)($ws?$word)?</
      schema>
</slave>

<!-- Characters indicating the start of a subsentence.-->
<slave name="sub">
```

```
  <schema>([,\-\(])</schema>
</slave>

<!-- Legal formats of times before a designator. -->
<slave name="designator_timeformat">
  <!-- Matching on $timecat alone yields too many false hits, e.g. "
       2004 UTC" -->
  <schema>($time_colon|$time_dot|$time_cat_hours)</schema>
</slave>

<!-- Legal formats of times after a prefix. -->
<slave name="prefix_timeformat">
  <!-- Matching on $timedot is unsafe here, e.g. "from 6.25 kg" -->
  <schema>($time_colon|$time_cat_hours|$time_names)</schema>
</slave>

<!-- Legal formats of times before a postfix. -->
<slave name="postfix_timeformat">
  <schema>($time_colon|$time_dot|$hours)</schema>
</slave>

<!-- Time formats requiring a designator; e.g. "7:50 AM", "11.45pm", "
     1230 GMT". -->
<slave name="time_designator">
  <schema>($designator_timeformat$ws?($ws?$more$ws?$
       designator_timeformat$ws?)*$designator)</schema>
</slave>

<!-- Time formats requiring a prefix, e.g. "after 8:45" -->
<slave name="time_prefix">
  <schema>($keyword_prefix$prefix_timeformat($ws?$more$ws?$
       prefix_timeformat)*($ws?$designator)?($timeword_postfix)?)</
       schema>
</slave>

<!-- Time formats requiring a postfix, e.g. "6 tomorrow morning" -->
<slave name="time_postfix">
  <schema>($postfix_timeformat($ws?$more$ws?$postfix_timeformat$ws?)
       *($dayword|$ws$filler$ws?($timeword|$weekdayword)))</schema>
</slave>

<!-- Hours require am/pm, not just timezone, to avoid e.g. "2 et.". --
     >
<slave name="hours_designator">
  <schema>($hours(?!\d)$ws?($am_pm($ws$timezone)?))</schema>
</slave>

<!-- Whole hours followed by "o'clock", e.g. "5 o'clock this evening".
     -->
<slave name="time_oclock">
  <schema>($hours$ws?($ws?$more$ws?$hours$ws?)*o'clock($ws$designator)
       ?($timeword_postfix)?)</schema>
</slave>

<!-- All of the time variations collected in one pattern -->
<slave name="time">
  <schema>(?i)\b($time_designator|$hours_designator|$time_prefix|$
       time_oclock|$time_postfix)</schema>
</slave>

<master optimize="yes">
  <schema>$time</schema>
```

```
<!-- Bump to lowercase. -->
<transformation source="^.*$" target="$0" command="lowercase"/>

<!-- Remove any "fluff". -->
<transformation source="[(){}\[\]]" target=""/>

<!-- Remove spaces in times. -->
<transformation source="\s*([\.:])\s*" target="$1"/>

<!-- Convert "more" separators to "; ". -->
<transformation source="\s*(?:/|,| and | or |\+|&| to |\-)\s*"
    target="; "/>

<!-- Put a : between dot times, e.g. 15.30 -> 15:30. -->
<transformation target="$1:$2">
  <source><![CDATA[(?<![:\d])(\d{1,2})\.(\d{2})]]></source>
</transformation>

<!-- Put a : between cat times, e.g. 1530 -> 15:30. -->
<transformation target="$1:$2">
  <source><![CDATA[(?<![:\d])(\d{2})(\d{2})(?: hours)?]]></source>
</transformation>

<!-- Add minutes to hour-only times, e.g. 8 pm -> 8:30 pm. -->
<transformation target="$1:00">
  <source><![CDATA[(?<![:\d])(\d{1,2})(?: o'clock)?(?![:\d])]]></
      source>
</transformation>

<!-- Add a 0 before 1-digit hours, e.g. 8:30 -> 08:30. -->
<transformation target="0$1:$2">
  <source><![CDATA[(?<![:\d])(\d):(\d{2})]]></source>
</transformation>

<!-- Convert "timewords" to AM/PM.-->
<!-- Note: The "more" conversion above might have converted a "sub"
    to a "; ". -->
<transformation source=";? ?(?:[a-z]+ ?){1,2} morning" target="am"/>
<transformation source=";? ?(?:[a-z]+ ?){1,2} (?:afternoon|evening|
    night)" target="pm"/>

<!-- Convert 12 AM to 24-hour format. -->
<transformation target="00$1$2">
  <source><![CDATA[(?<![:\d])(?:12)((?::\d{2}){1,2})(.*?a\.? ?m\.?)
      ]]></source>
</transformation>

<!-- Convert PM times to 24-hour format. -->
<transformation target="13$1$2">
  <source><![CDATA[(?<![:\d])(?:01)((?::\d{2}){1,2})(.*?p\.? ?m\.?)
      ]]></source>
</transformation>
<transformation target="14$1$2">
  <source><![CDATA[(?<![:\d])(?:02)((?::\d{2}){1,2})(.*?p\.? ?m\.?)
      ]]></source>
</transformation>
<transformation target="15$1$2">
  <source><![CDATA[(?<![:\d])(?:03)((?::\d{2}){1,2})(.*?p\.? ?m\.?)
      ]]></source>
</transformation>
<transformation target="16$1$2">
```

```
        <source><![CDATA[(?<![:\d])(?:04)((?::\d{2}){1,2})(.*?p\.? ?m\.?)
            ]]></source>
      </transformation>
      <transformation target="17$1$2">
        <source><![CDATA[(?<![:\d])(?:05)((?::\d{2}){1,2})(.*?p\.? ?m\.?)
            ]]></source>
      </transformation>
      <transformation target="18$1$2">
        <source><![CDATA[(?<![:\d])(?:06)((?::\d{2}){1,2})(.*?p\.? ?m\.?)
            ]]></source>
      </transformation>
      <transformation target="19$1$2">
        <source><![CDATA[(?<![:\d])(?:07)((?::\d{2}){1,2})(.*?p\.? ?m\.?)
            ]]></source>
      </transformation>
      <transformation target="20$1$2">
        <source><![CDATA[(?<![:\d])(?:08)((?::\d{2}){1,2})(.*?p\.? ?m\.?)
            ]]></source>
      </transformation>
      <transformation target="21$1$2">
        <source><![CDATA[(?<![:\d])(?:09)((?::\d{2}){1,2})(.*?p\.? ?m\.?)
            ]]></source>
      </transformation>
      <transformation target="22$1$2">
        <source><![CDATA[(?<![:\d])(?:10)((?::\d{2}){1,2})(.*?p\.? ?m\.?)
            ]]></source>
      </transformation>
      <transformation target="23$1$2">
        <source><![CDATA[(?<![:\d])(?:11)((?::\d{2}){1,2})(.*?p\.? ?m\.?)
            ]]></source>
      </transformation>
      <transformation target="12$1$2">
        <source><![CDATA[(?<![:\d])(?:12)((?::\d{2}){1,2})(.*?p\.? ?m\.?)
            ]]></source>
      </transformation>

      <!-- Remove AM/PM. -->
      <transformation source=" ?[ap]\.? ?m\.?" target=""/>

      <!-- Convert named times to 24-hour format. -->
      <transformation source="midnight" target="00:00"/>
      <transformation source="noon" target="12:00"/>

      <!-- Strip "time" from timezone name. -->
      <transformation source=" time" target=""/>

      <!-- Normalize timezonename to timezoneabbr. -->
      <transformation source="(?:[a-z]{5,} ?)+ ?" transducer="resources/
          dictionaries/matching/timezonenametrans.aut" target="$0"/>

      <!-- Normalize timezoneabbr to UTC format. -->
      <transformation source="[a-z]{2,4}" transducer="resources/
          dictionaries/matching/timezoneabbrtrans.aut" target="$0"/>
    </master>

  </pattern>
 </matcher>
</configuration>
```

## A.2 Time Extractor reference file

```
<?xml version="1.0" encoding="iso-8859-1"?>

<file>
  <sentence>Monday, June 13, 2005 Posted: <time>8:06 AM EDT</time> (<time>
      1206 GMT</time>)</sentence>
  <sentence>Citing the logbook, which covers al-Qahtani's interrogations
      from November 2002 to January 2003, Time reports that daily interviews
       began at <time>4 a.m.</time> and sometimes continued until <time>
      midnight</time>.</sentence>
  <sentence>At around <time>3 a.m.</time>, some men head for a break at the
      1873 Cafe de Paris.</sentence>
  <sentence>Filed at 7/14/2004 <time>12:23:42 PM</time>.</sentence>
  <sentence>Filed at <time>12:40 a.m. ET</time>. Modified at <time>20:00 EST
      </time>.</sentence>
  <sentence>Tuesday <time>6.30-7.30pm</time></sentence>
  <sentence>Wednesday <time>5.30 to 6.30pm</time></sentence>
  <sentence>You get there about <time>3:40, 3:50</time>, and the teachers
      are usually gone.</sentence>
  <sentence>Mr. BELL moved that hereafter the Convention will meet at <time
      >9 A. M.</time>, take a recess from <time>2 o'clock P. M.</time> to <
      time>4 o'clock P. M.</time>, and also from <time>6 P. M.</time> to <
      time>7 P. M. UTC</time>.</sentence>
  <sentence><time>12:50</time> - the gun is wheeled back and prepared for
      loading.</sentence>
  <sentence>A tongue - in - cheek reversal of 'vampyre' subculture with
      young vampires who wear bright clothes, drink wine, and stay up until
      <time>noon</time>.</sentence>
  <sentence>Launched on April 17, 1989, it provides business news
      programming from <time>5 am</time> to <time>7 pm eastern time</time>,
      and talk shows from <time>7 pm</time> until <time>midnight eastern
      time</time>.</sentence>
  <sentence>On July 2, 1937, at <time>midnight GMT</time>, Earhart and
      Noonan took off from Lae.</sentence>
  <sentence>Not for publication until <time>1500 hours (bst)</time> on
      Thursday, 16 may 2002.</sentence>
  <sentence>They moved to attack at precisely <time>1700 hours GMT</time>,
      and were soon defeated (<time>1800 hours GMT</time>).</sentence>
  <sentence>Afternoon tea served between <time>1500 and 1700 hours</time>.</
      sentence>
  <sentence>"It's usually at <time>3 or 4 o'clock in the afternoon</time>.
      Alternatively at <time>3 or 4 in the morning</time>."</sentence>
  <sentence>Beatty turned and fled towards the Grand Fleet and from <time>18
       : 30</time> until nightfall at about <time>20 : 30</time> the two
      huge fleets were heavily engaged.</sentence>
  <sentence>Jellicoe broke contact with the Germans at about <time>16 . 45</
      time> and Hipper turned back to Scheer around <time>18 . 00</time>,
      but didn't meet rendezvous until <time>18 . 30</time>.</sentence>
  <sentence>By <time>19 : 15</time>, Jellicoe had crossed the " T " yet
      again</sentence>
  <sentence>Web page retrieved <time>15 : 30</time> Oct 6 , 2004 ( UTC )</
      sentence>
  <sentence>President Bush will make his address at <time>9 p.m. ET</time> (
      <time>0200 GMT</time> Thursday)</sentence>
  <sentence>Nonetheless Scheer slipped away as sunset (<time>20:24</time>)
      approached.</sentence>
  <sentence>It was precisely <time>7.15 on Saturday night</time>.</sentence>
  <sentence>About 3 km in the outskirts of Kitgum town at around <time>3
      :00pm</time>, <time>3.00pm</time>, the army spokesman in the north
      said.</sentence>
```

```
<sentence>The secretary of the commission drove in at about <time>6. 30 PM
    </time>.</sentence>
<sentence><![CDATA[<b>]]><time>8:50AM</time> Handleman reports Q4 results<
    /sentence>
<sentence>I'll be there at <time>7:01, 6:47 and 6:25</time>.</sentence>
<sentence>As you know, at <time>4:30 yesterday afternoon</time>, Alaska
    Airlines Flight 261 crashed into the Pacific Ocean.</sentence>
<sentence>We received the initial notification of the downed aircraft from
     the park ranger at <time>4:26, yesterday afternoon</time>. At <time>4
    :27</time> we issued an urgent marine broadcast to all boaters in the
    area to come to assist in that position.</sentence>
<sentence>To the best of knowledge as of <time>6:00, this morning</time>,
    we didn't have any reports.</sentence>
<sentence>We didn't find Rebecca until <time>11:45 that morning</time>.</
    sentence>
<sentence>The first one begins at <time>8:45 Eastern this evening</time
    >.</sentence>
<sentence>That means <time>4:44 Paris time</time>.</sentence>
<sentence>It is <time>8:15</time> here on the East Coast, that makes it
    about <time>3:44</time> local.</sentence>
<sentence>So <time>4:00 Eastern</time>, we'll be able to bring that to
    you when it happens.</sentence>
<sentence>The family will be out here at <time>2:00 Mountain time</time>,
    <time>4:00 Eastern</time>.</sentence>
<sentence>Just before <time>6:00 A.M. East Coast</time> time.</sentence>

<!-- False examples -->
<sentence>Secretary of State Colin Powell now says 10 Americans were
    killed in car bomb attacks.</sentence>
<sentence>The week capped at $4.43 in today's dollars.</sentence>
<sentence>A drop from about 7 per cent today.</sentence>
<sentence>We ate 5 sandwiches last night.</sentence>
<sentence>They are expected to sell at least 3.30 kg this evening.</
    sentence>
<sentence>Mr Costello told the ABC's 7.30 Report last night.</sentence>
<sentence>Shares in Great Southern Plantations were changing hands at
    $3.15 on Tuesday afternoon</sentence>
<sentence>We lost 4.12 last night, and 12.20 this evening.</sentence>
<sentence>Customer profitability systems after 3.19 years and customer
    acquisition cost after 3.16 years.</sentence>
<sentence>Some Labor MPs believe Labor may have to wait until 2010 before
    it has a realistic chance of reclaiming power.</sentence>
<sentence>Inspired by the doctrine embraced by the American administration
     after the 2001 terrorist attacks.</sentence>
<sentence>The Nammer had nine stores pylons and could carry up to 6.25
    tonnes</sentence>
<sentence>Which leads to a value of pi equal to 3.16049</sentence>
<sentence>Les articles 8.9.2 et 6.1.6 et propose des modifications.</
    sentence>
<sentence>Where to stay: Drake Hotel (1150 Queen Street West;
    416-531-5042)..</sentence>
<sentence>Most riders either take a BMW R 1150 GS or a Kawasaki KLR 650.</
    sentence>
<sentence>Alternative Talk 1150am is your home for UW Baseball and
    Softball in 2005!</sentence>
<sentence>Elite Video P-1208 has a lamplife of about 2000 hours.</sentence
    >
<sentence>February 20 - 2001 UK foot and mouth crisis begins.</sentence>
<sentence>Joe Montana, Mario Andretti and Morgan Freeman, for example,
    have all zeroed in on the enemy at 12 o'clock high in prop planes with
     co-pilots provided by Air Combat USA, based in Fullerton, Calif.</
    sentence>
```

```
  <sentence>A six-foot-tall bride, resplendent in white silk, rippling
      biceps and five o'clock shadow.</sentence>
  <sentence>Purchasing power parity - $4.8 billion (2002 est.) , $4.2
      billion (1999 est.)</sentence>
  <sentence>The ship was built in 1930 at the (formerly Royal) Naval
      Shipyard of Castellammare di Stabia (Naples).</sentence>
  <sentence>The average household size is 2.94 and the average family size
      is 3.46.</sentence>
  <sentence>Rashid Ramzi - Semifinal, 3:44.60 (did not advance).</sentence>
  <sentence>Mohammed Abdelhak Zakaria - Round 1 , 13 : 42 . 04 ( did not
      advance ).</sentence>
  <sentence>Track listing; #1 "Wheels Of Confusion" - 8:00</sentence>
  <sentence>King Melchizedek of Salem, identified by Rashi as being Shem the
       son of Noah by another name, is the first person in the Torah to be
      called a Kohen (Genesis 14:18).</sentence>
</file>
```

## A.3 Person Classifier configuration

```
<?xml version="1.0"?>

<!-- Copyright (C) 2006 Fast Search &amp;amp; Transfer ASA -->

<configuration>
  <matcher type="pattern" debug="no">
    <pattern flags1="0" flags2="0">

      <!-- Include common NLQ patterns. -->
      <include namespace="" filename="etc/config_data/QRServer/webcluster/
          etc/qrserver/nlq/configuration.nlq.aux.xml"/>

      <!-- Specifier dictionary for persons -->
      <slave name="person_specifier">
        <schema>($word)</schema>
        <acceptor>
          <matcher type = "levenshtein" debug = "no">
            <levenshtein automaton = "resources/dictionaries/matching/nlq/
                person_specifiers.aut"
              phonetics = "no" exact = "no"
              match = "complete" start = "beginning"
              prefix = "0" tolerance = "0">
              <quality type = "raw" penalty = "1" threshold = "0"/>
            </levenshtein>
          </matcher>
        </acceptor>
      </slave>

      <!-- Person specifier -->
      <slave name="person" visibility="surface">
        <meta report="yes" name="person"/>
        <schema>($adjective$ws)?$person_specifier</schema>
      </slave>

      <!-- $do $np $get? $vp ($np_verb? $pp)? $np_prep* -->
      <slave name="person_do_part">
        <schema>$ws$do$ws$np($ws$get)?$ws$vp($ws$np_verb?$ws$pp)?($ws$
            np_prep)*</schema>
      </slave>

      <!-- ($be|$get)? $vp ($np_verb $np_prep*|$np_prep+) -->
      <slave name="person_vp_part">
        <schema>($ws($be|$get))?$ws$vp($ws$np_verb($ws$np_prep)*|($ws$
            np_prep)+)</schema>
      </slave>

      <!-- who $be $np_type ($vp $np_verb?)? $np_prep* -->
      <slave name="who_be">
        <schema>$who$ws$be$ws$np_type($ws$vp$ws$np_verb?)?($ws$np_prep)*</
            schema>
      </slave>

      <!-- who $do $np $get? $vp ($np_verb? $pp)? $np_prep* -->
      <slave name="who_do">
        <schema>$who$person_do_part</schema>
      </slave>

      <!-- who ($be|$get)? $vp ($np_verb $np_prep*|$np_prep+) -->
      <slave name="who_vp">
```

```
      <schema>$who$person_vp_part</schema>
    </slave>

    <!-- what $person $be $vp $np_verb? $np_prep* -->
    <slave name="which_person_be">
      <schema>$which$ws$person$ws$be$ws$vp($ws$np_verb)?($ws$np_prep)*</
          schema>
    </slave>

    <!-- what $person $do $np $get? $vp ($np_verb? $pp)? $np_prep* -->
    <slave name="which_person_do">
      <schema>$which$ws$person$person_do_part</schema>
    </slave>

    <!-- what $person ($be|$get)? $vp ($np_verb $np_prep*|$np_prep+) -->
    <slave name="which_person_vp">
      <schema>$which$ws$person$person_vp_part</schema>
    </slave>

    <slave name="person_question">
      <schema>($who_be|$who_do|$who_vp|$which_person_be|$which_person_do|$
          which_person_vp)</schema>
    </slave>

    <master optimize="yes">
      <schema>$person_question</schema>

      <!-- Expand contractions like "isn't" to "is not" to be able to work
           with full words -->
      <!-- Might specify general rules like -- source="([a-z]+)n't" target
          ="$1 not" -- instead of a dictionary -->
      <transformation source="[a-z]+'[a-z]+|cannot" target="$0" transducer
           = "resources/dictionaries/matching/nlq/contraction_expansion.
          aut" />

      <!-- Remove excessive whitespace -->
      <transformation source="\s+" target=" " />

      <!-- Create a general scope query for a person. The matched terms
          are reported as meta-tags. -->
      <!-- This query will be completed by the nlq2fql module -->
      <transformation source="^.*$" target="xml:sentence:and(*NLQ*, scope(
          person))" />

    </master>
  </pattern>
 </matcher>
</configuration>
```

## A.4 Hyponym Extractor configuration

```
<?xml version="1.0" encoding="utf-8"?>

<!-- Copyright (C) 2006 Fast Search &amp;amp; Transfer ASA -->

<configuration>
  <matcher type="pattern" debug="no">
    <pattern flags1="0" flags2="0">

      <!-- Include common NLQ patterns. -->
      <include namespace="" filename="etc/config_data/QRServer/webcluster/
          etc/qrserver/nlq/configuration.nlq.aux.xml"/>

      <!-- Only report the actual noun(s) of the NP, not the PP, although
          the whole phrase must be matched to be useful in a hypernym/
          hyponym expression. -->
      <slave name="hypernym">
        <schema>$hypernym_np($ws$pp)?</schema>
      </slave>

      <!-- Report the noun(s) of the NP. We are only interested in common (i
          .e. not proper) nouns as hypernyms. -->
      <slave name="hypernym_np">
        <meta report="yes" name="hypernym"/>
        <schema>$np_common</schema>
      </slave>

      <!-- Avoid capturing as a hyponym an NP that is also a hypernym at the
          start of a new phrase. -->
      <!-- E.g. "cities" in "...languages such as French and cities such as
          Toulouse..." -->
      <slave name="hyponym">
        <schema>$np(?!,?$ws\(?such($ws)as|(,$ws|$ws\()$sub_phrase)</schema>
      </slave>

      <!-- A sequence of hyponyms separated by commas and/or conjunctions --
          >
      <slave name="hyponyms" meta="yes">
        <schema>$hyponym($more$hyponym)*</schema>
      </slave>

      <!-- Terms that indicate a subphrase of hyponyms -->
      <!-- Other possibilities: "e.g.", "i.e.", "s.a.", etc. -->
      <slave name="sub_phrase">
        <schema>(including|especially|particularly|(most$ws)?notably)</
            schema>
      </slave>

      <!-- Such $hypernym as $hyponyms -->
      <!-- E.g. "...such authors as Herrick, Goldsmith and Shakespeare..."
          -->
      <slave name="such_hyper_as">
        <schema>[Ss]uch$ws$hypernym($ws)as$ws$hyponyms</schema>
      </slave>

      <!-- $hypernym such as $hyponyms -->
      <!-- E.g. "...the bow lute, such as the Bambara ndang..." -->
      <slave name="hyper_such_as">
        <schema>$hypernym,?$ws\(?such($ws)as$ws$hyponyms</schema>
      </slave>
```

```
      <!-- $hypernym $subphrase $hyponyms -->
      <!-- E.g. "...all common-law countries, including Canada and England
          ..." -->
      <!-- E.g. "...most European countries, especially France, England and
          Spain..." -->
      <slave name="hyper_sub">
        <schema>$hypernym(,$ws|$ws\()$sub_phrase$ws$hyponyms</schema>
      </slave>

      <!-- $hyponyms (and|or) other $hypernym -->
      <!-- E.g. "...wounds, broken bones or other injuries..." -->
      <!-- E.g. "...temples, treasuries, and other important civic buildings
          ..." -->
      <slave name="hypo_other">
        <schema>$hyponyms,?$ws(and|or)($ws)other$ws$hypernym</schema>
      </slave>

      <slave name="hyponym_sentence">
        <schema>($such_hyper_as|$hyper_such_as|$hyper_sub|$hypo_other)</
            schema>
      </slave>

      <master optimize="yes">
        <schema>$hyponym_sentence</schema>

        <!-- Use a preprocessor to quickly filter out sentences not
            containing crucial keywords for the hyponym extractor -->
        <preprocessor radius="150">
          <matcher type="verbatim" debug="no">
            <verbatim match="boundary" start="boundary" swap="yes" lowercase
                ="yes">
              <entries>
                <entry><value>especially</value></entry>
                <entry><value>including</value></entry>
                <entry><value>notably</value></entry>
                <entry><value>other</value></entry>
                <entry><value>particularly</value></entry>
                <entry><value>such</value></entry>
                <entry><value>Such</value></entry>
              </entries>
            </verbatim>
          </matcher>
        </preprocessor>

      </master>

    </pattern>
  </matcher>
</configuration>
```

## A.5 Auxiliary patterns configuration

```
<?xml version="1.0" encoding="utf-8"?>

<!-- Copyright (C) 2004-2006 Fast Search &amp;amp; Transfer ASA -->

<configuration>
  <matcher type="pattern" debug="no">
    <pattern flags1="0" flags2="0">

      <!-- Upper case character -->
      <slave name="upperchar">
        <schema>([A-Z])</schema>
      </slave>

      <!-- Lower case character -->
      <slave name="lowerchar">
        <schema>([a-z])</schema>
      </slave>

      <!-- Upper case or lower case character -->
      <slave name="anychar">
        <schema>([A-Za-z])</schema>
      </slave>

      <slave name="pre">
        <schema><![CDATA[(?<!$anychar)]]></schema>
      </slave>

      <slave name="post">
        <schema>(?!$anychar)</schema>
      </slave>

      <!-- Capitalized word -->
      <!-- E.g. "Microsoft", "AltaVista", "eMentor" -->
      <!-- Also e.g. "J." and "Mr." to allow constructs like: "J. K. Rowling
          ", "Mr. Universe", "Bush Jr." -->
      <slave name="capword">
        <schema>($pre($lowerchar?$upperchar($lowerchar+\.?|$anychar+|\.))$
            post)</schema>
      </slave>

      <!-- Uppercase abbreviations and acronyms -->
      <!-- E.g. "FAQs", "F.A.Q." -->
      <slave name="abbrword">
        <schema>($pre($upperchar{2,}s?|($upperchar\.)+)$post)</schema>
      </slave>

      <!-- Word, capitalized or not -->
      <slave name="word">
        <schema>($pre($upperchar|$lowerchar)$lowerchar+$post)</schema>
      </slave>

      <!-- Whitespace -->
      <slave name="ws">
        <schema>[ \t]+</schema>
      </slave>

      <slave name="more">
        <schema>(,$ws(and$ws|or$ws)?|$ws(and($ws?[&/]$ws?or)?|or)$ws|$ws
            ?[&/]$ws?)</schema>
```

```
    </slave>

    <slave name="article">
      <schema>($pre(an?|the)$ws)</schema>
    </slave>

    <slave name="stopword">
      <schema>$word</schema>
      <acceptor>
        <matcher type = "verbatim" debug = "no">
          <verbatim automaton = "resources/dictionaries/matching/nlq/
              stopwords.aut"
             match = "boundary" start = "boundary" swap = "yes" lowercase
                 = "yes">
          </verbatim>
        </matcher>
      </acceptor>
    </slave>


<!-- Adjectives -->

    <slave name="adjective_dict">
      <schema>$word</schema>
      <acceptor>
        <matcher type = "levenshtein" debug = "no">
          <levenshtein automaton = "resources/dictionaries/matching/nlq/
              adjectives.aut"
            phonetics = "no" exact = "no"
            match = "complete" start = "beginning"
            prefix = "0" tolerance = "0">
            <quality type = "raw" penalty = "1" threshold = "0"/>
          </levenshtein>
        </matcher>
      </acceptor>
    </slave>

    <!-- Ordinal numbers, e.g. "1st, 2nd, 3rd, 4th, ..." -->
    <slave name="ordinals">
      <schema>(\d*1st|\d*2nd|\d*3rd|\d*[04-9]+th)</schema>
    </slave>

    <!-- E.g. "beautiful", "user-assisted", "full-text", "SEC corporate",
        "23rd" -->
    <slave name="adjective">
      <schema>(($capword$ws)?($noun_dict-$adjective_dict|$adjective_dict
          (-$noun_dict)?|$ordinals))</schema>
    </slave>

    <slave name="adj_meta">
      <meta report="yes" name="adjective"/>
      <schema>$adjective</schema>
    </slave>

    <slave name="adjs_meta">
      <schema>($adj_meta(,$ws$adj_meta)*)</schema>
    </slave>


<!-- Prepositions -->

    <slave name="preposition">
```

```
    <schema>$word</schema>
    <acceptor>
      <matcher type = "verbatim" debug = "no">
        <verbatim automaton = "resources/dictionaries/matching/nlq/
            prepositions.aut"
          match = "boundary" start = "boundary" swap = "yes" lowercase
            = "yes">
        </verbatim>
      </matcher>
    </acceptor>
  </slave>

  <slave name="pp_word">
    <schema>$word</schema>
    <rejector>
      <matcher type = "verbatim" debug = "no">
        <verbatim automaton = "resources/dictionaries/matching/nlq/
            pp_word_rejector.aut"
          match = "boundary" start = "boundary" swap = "yes" lowercase
            = "yes">
        </verbatim>
      </matcher>
    </rejector>
  </slave>

  <slave name="pp">
    <schema>($preposition(($ws|[-'])$pp_word[-']?)+(?=,|$ws\())</
        schema>
  </slave>


<!-- Verbs -->

  <slave name="verb_dict">
    <schema>$word</schema>
    <acceptor>
      <matcher type = "verbatim" debug = "no">
        <verbatim automaton = "resources/dictionaries/matching/nlq/verbs
            .aut"
          match = "boundary" start = "boundary" swap = "yes" lowercase
            = "yes">
        </verbatim>
      </matcher>
    </acceptor>
  </slave>

  <slave name="verb">
    <schema>((to$ws)?$verb_dict)</schema>
  </slave>

  <slave name="verb_meta">
    <meta report = "yes" name = "verb"/>
    <schema>$verb</schema>
  </slave>


<!-- Nouns -->

  <slave name="noun_dict">
    <schema>$word</schema>
    <acceptor>
      <matcher type = "levenshtein" debug = "no">
```

```
        <levenshtein automaton = "resources/dictionaries/matching/nlq/
            nouns.aut"
          phonetics = "no" exact = "no"
          match = "complete" start = "beginning"
          prefix = "0" tolerance = "0">
          <quality type = "raw" penalty = "1" threshold = "0"/>
        </levenshtein>
      </matcher>
    </acceptor>
  </slave>

  <slave name="noun">
    <schema>$noun_dict(?!-)</schema>
  </slave>

  <slave name="safe_noun_dict">
    <schema>$word</schema>
    <acceptor>
      <matcher type = "levenshtein" debug = "no">
        <levenshtein automaton = "resources/dictionaries/matching/nlq/
            safe_nouns.aut"
          phonetics = "no" exact = "no"
          match = "complete" start = "beginning"
          prefix = "0" tolerance = "0">
          <quality type = "raw" penalty = "1" threshold = "0"/>
        </levenshtein>
      </matcher>
    </acceptor>
  </slave>

  <slave name="safe_noun">
    <schema>$safe_noun_dict(?!-)</schema>
  </slave>

  <slave name="proper_noun">
    <schema>$capword(($ws|-)$capword)*(\.com|!(?=,))?</schema>
  </slave>

  <!-- Phrases in quotes -->
  <!-- E.g. "Half Blood Prince", "Gone with the Wind" -->
  <slave name="phrase">
    <schema>(".*")</schema>
  </slave>

  <!-- E.g. "1989", "300,000", "256 000", "512.000" -->
  <slave name="numex">
    <schema>$pre\d+([\.,]\d+|([\., ]\d{3})*)?$post</schema>
  </slave>

  <slave name="any_noun">
    <schema>($capword|$noun|$phrase|$numex)</schema>
  </slave>


<!-- NPs capturing long phrases, including PP and limited subphrases -->

  <slave name="np_pre">
    <schema>($article?($adjective(,$ws$adjective)*$ws)?)</schema>
  </slave>

  <!-- The $safe_noun negative look-ahead at the end is there to assure
      we capture the longest possible NP.-->
```

```
    <!-- Otherwise slaves referring to this one might be satisfied with a
         shorter NP, and interpret the rest of the phrase in another
         context. -->
    <slave name="np_part">
      <schema>($np_pre($proper_noun|$noun)($ws$noun)*($ws$abbrword)?(?!$ws
          $safe_noun))</schema>
    </slave>

    <!-- NP including "of", e.g. "president of the United States" -->
    <slave name="np">
      <schema>($np_part($ws(of|on|in)$ws$article?$np_part)?)</schema>
    </slave>

    <slave name="np_common_part">
      <schema>($np_pre($proper_noun$ws)?$noun($ws$noun)*($ws$abbrword)
          ?(?!$ws$safe_noun))</schema>
    </slave>

    <slave name="np_common">
      <schema>($np_common_part($ws(of|on|in)$ws$article?$np_part)?)</
          schema>
    </slave>


<!-- NPs capturing short phrases, e.g. no PP or subphrases -->

    <slave name="np_short_pre">
      <schema>($article?($adjs_meta$ws)?)</schema>
    </slave>

    <slave name="np_short">
      <schema>($any_noun(($ws|-)$any_noun)*)</schema>
    </slave>

    <slave name="np_prep">
      <schema>($preposition($ws$stopword)*$ws$np_short_pre$np_prep_meta)</
          schema>
    </slave>

    <slave name="np_prep_meta">
      <meta report="yes" name="np_prep"/>
      <schema>$np_short</schema>
    </slave>

    <slave name="np_subj">
      <schema>($np_short_pre$np_subj_meta)</schema>
    </slave>

    <slave name="np_subj_meta">
      <meta report="yes" name="np_subj"/>
      <schema>$np_short</schema>
    </slave>

    <slave name="np_verb">
      <schema>($np_short_pre$np_verb_meta)</schema>
    </slave>

    <slave name="np_verb_meta">
      <meta report="yes" name="np_verb"/>
      <schema>$np_short</schema>
    </slave>
```

```
    <slave name="np_type">
      <schema>($np_short_pre$np_type_meta)</schema>
    </slave>

    <slave name="np_type_meta">
      <meta report="yes" name="np_type"/>
      <schema>$np_short_common</schema>
    </slave>

    <slave name="vp">
      <meta report="yes" name="vp"/>
      <schema>$verb</schema>
    </slave>


<!-- Auxiliary verb classes -->

    <!-- Tenses of "to be" -->
    <slave name="be">
      <schema>(is|are|w(as|ere))</schema>
    </slave>

    <!-- Tenses of "to do" -->
    <slave name="do">
      <schema>(do(es|id)?)</schema>
    </slave>

    <!-- Tenses of "to get" -->
    <slave name="get">
      <schema>(g(o|e)t?)</schema>
    </slave>

    <!-- Tenses of "to have" -->
    <slave name="have">
      <schema>(ha(ve|s|d)?)</schema>
    </slave>

    <!-- Any auxiliary verb -->
    <slave name="auxverb">
      <schema>($be|$do|$get|$have)</schema>
    </slave>


<!-- Interrogative words -->

    <slave name="who">
      <schema>([Ww]ho(m|se)?)</schema>
    </slave>

    <slave name="when">
      <schema>([Ww]hen)</schema>
    </slave>

    <slave name="where">
      <schema>([Ww]here)</schema>
    </slave>

    <slave name="what">
      <schema>([Ww]hat)</schema>
    </slave>

    <slave name="which">
```

```
        <schema>([Ww]h(at|ich))</schema>
      </slave>

      <slave name="why">
        <schema>([Ww]hy)</schema>
      </slave>

      <slave name="how">
        <schema>([Hh]ow)</schema>
      </slave>

      <master name="dummy">
        <schema>dummy</schema>
      </master>

    </pattern>
  </matcher>
</configuration>
```

## A.6 Examples of extracted hyponyms

Bonfigli:style of older Perugian painters
Bonfire night:specific times of year
Bong:number
Bonham Road:myriad of former Governors
Boniface:names
Bonifacio:tourist areas
Bonifacio Bembo:miniaturists of Ferrara school
Bonin Islands:genus of tree native
Bonington Hall:halls of residence
Bonn:Global protests, global protests, cities, demonstrations
Bonnaroo:festivals
Bonneville:prairies of west, establishing communities
Bonnie:criminals, nicknames
Bonnie Joan:poems
Bonnie Raitt:artists
Bonnie St:artists
Bonnie Tyler:artists
Bonnyrigg:housing developments in areas
Bono:mythology of Akan people
Bonobo apes:species
Bonus march on Washington:rising
Bonwit Taylor:businesses
Bonzo Dog Doo Dah Band:only source of artists
Bonzo Goes:movies
Boobytrap:labels
Booch:techniques
Boogie Nights:appeared films
Book:commercial book clubs, works, document, parallel
Book Excalibur:comics
Book Talk:groups
Book of Abraham:significant Latter Day Saint documents
Book of Armagh:insular manuscripts
Book of Daniel:eschatological sections of Bible
Book of Durrow:motifs
Book of Enoch:They offer examples
Book of Exodus:modern historians due
Book of Fermoy:century manuscripts
Book of Genesis:established christian doctrines, traditions
Book of Gospels:illuminated codices
Book of Henoch:extra-canonical books of Judaism
Book of Jasher:traditions
Book of Kells:important books, treasures, well-known manuscripts
Book of Mormon:LDS scripture, names in LDS scripture, volumes of scripture
Book of Revelation:prophecies
Book of Wisdom:influenced Alexandrian writers
Booker:integrated groups
Booker T. Washington:exploitation of Congo, Historical figures
Booklets:useful materials
Books:media, materials, items
Martins Creek:locations
Marty Friedman:careers of dozens
Martyn:writers
Martyrdom of Sebastian:subjects
Marvel:CCA sponsors, introduction, script mainstream comics, Large comic book publishers, companies
Marvel Comics:recognizable work on various comic books, places, titles, publications, characters, members of comic industry release, comic-book houses
Marvel Directory:official Marvel publications

```
Marvel Knights:part of Marvel Universe
Marvel Team-Up:Men characters, part of Marvel Universe
Marvel UK:imprints
Marvel Universe:shared universes
Marvel comics:corporate publishers
Marvel vs:number of fighting games
Marvell Technology Group:emerging companies
Marvellous:timeless singles
Marvels:number of different titles
Marvian:Tajiks ancestors
Marvin:depressive characters in literature
Marvin Gaye:hometown stars, famous musicians, concept albums, artists
Marvin Handfield:islands
Marvin Minsky:robot bodies, artificial intelligence, figures in computer
    science
Marwari:Rajasthani languages
Marx:philosophers, German philosophers, Hegelian philosophers, students,
    thinkers
Marx Brothers:comedic stars of day, entertainers, radio performers, various
    MGM films, movies, films
Marxian:Modern variants of Marxist economics
Marxian schools:analysis of political actions
Marxism:Some ideologies, diverse traditions, economic ideology, views,
    theories, certain western schools of economic thought, ways of thinking,
     cases, unrelated fields of applied ethics, foreign ideologies, systems
Marxism-Leninism:various derivatives
Marxist Platform of BSP:Bulgarian groups
Marxist philosophy:questions
Marxist theory:forms of exploitation
flippers:standard pinball paraphernalia
flips:aerials, acrobatic maneuvers
flirtation:means of cooperation
float:basic data types
floaters:visual disturbances
floating:shore bases, macroeconomic reforms of Hawke government, water
    closet improvements, effects, monetary reforms
floating exchange rate:economic reforms
floating kelp:flotsam
floating point:features
floating point arithmetic:advanced features
floating point math support:functionality
floating point numbers:numerous advances
floating point unit:separate component
floating practice:two volume book
floating reserve:fifteen divisions
floating ribs:softer parts of body
floating sunglasses:related gear
floating tree trunks:man-made means
floating-point numbers:attempt
floating-point units of mainframe:computers
floatplanes:Reconnaisance aircraft
floats:floats
flock:cognates, thousands of people
flock of one thousand flamingos:specimens of magnificent bird life
flocks:came
flocks of birds:creatures
flocks of thousand:favoured sites
flogging:prohibited brutality in punishment, acts, forms of impact play
flogging Boudicca:number of atrocities
flood:natural disaster, emergencies, response, catastrophes
flood control:services
flood search routing:deterministic routing scheme
```

flooded basements:water damage
flooded fields:fresh water
flooded fields in winter:wet open lowlands
flooded quarry:local body of water
flooding:sources, conventional methods, behaviour therapies, support views of large local floods
flooding of large areas:environmental effects
flooding of river:successive recurrences of seasonal event
floodplains:found, well-watered areas, variety of open wet areas
floods:loss of human life, farmers, calamities, high visibility, frequent natural diasters, hard times, During natural disasters, natural disasters, cases of emergency situations, living quarters, even natural disasters, Mass emergencies, disasters, natural phenomena
floods in Mississippi:news of local disasters
floor:flat surface, functional surface, back on firm surface, surface, comfortable level
goats:plants, artiodactyl herbivores, domestic mammals, animals, livestock, mammals, Some forms of livestock, described fantastic animals on Moon, going, peoples using pigs, even-toed ungulates, widespread use of livestock, Other grazing species, processes goods, variety of livestock, nutrition of ruminant animals, raise small livestock, wild animals, herbivores, Other ruminant species
goblets:collectable set of memorabilia
goblin folk:creatures
goblins:mischievous demons, fantastical creatures, stories, costs, races, mythical beings, traditional fantasy elements
gobo:accessory
gobos:list
god:supernatural concepts, supernatural being
god complexes:number of notions
god of fish:sea creatures
god of metals:underground items of great value
goddess:one, time
goddesses:beings
godlike being:higher station
gods:time commitments permit one, varieties, archetypes, mythical subjects, supernatural beings, supernatural concepts, attribute personhood, same sort, supernatural agencies
gods of Hinduism:mythology
goggles:forms of eye protection
going:activities, bigger goal, tasks, contemporary events, writing on chalkboard, going, use of GOTO instructions, inquiries
going back:movies
going in thick:danger
going on three hunger strikes:struggle
going swimming:considered simple tasks
goitre:illnesses
royal jubilees:mark special events
royal kitchens of France:lands
royal palaces:English houses today
royal prerogatives:customs
royal princes:vast cortege
royal threadfin:species
royals:nobles, gigantic project of draining England
royalties:terms
royalties on number:benefits
royalty:mockery of various topics, keep
rubber:important source of supplies, make products, synthetic materials, growth in nontraditional primary exports, construction materials, cables, trade in items, industries, tropical plants, harvesting goods, unusual materials, materials, extractive, odd skin composition, cash crops, elastomeric material, elastomers, demand, contraptions, exotic goods,

```
    soft layer makes, process raw materials, specific materials, elastomeric
     materials, products
rubber band:fastener, binding
rubber bands:technologies
rubber bullets:weapons, lethal rounds, natural force
rubber gloves:rubber products
rubber hammers:things
rubber mat:slip surface
rubber plants:plants
rubber-based products:host
rubbing:soft tissues of body, show behaviours, responds
rubella:maternal problems, common childhood diseases
rubidium:new elements
rubies:gems, Artificially produced gemstones, center of gemstone mining
ruby:Several gemstones, precious stones, traces of chromium
web server:Internet device, software program, server model
web server software:back-end database software
web servers:Internet services, applications
web services:server-side applications, new markets
web site content:creative works
web sites:discussions, data, computer-based media, resources
web tool:repertory grid
web traffic:communication channel
webbed hands:various features
weber:distributing standards
webmaster:administrative person
webpage:resource
webpages:establishment of applications
webs:areas
webs in woodpiles:places
website:project, publicity material, Internet server, OR profession
website access:promotional items
website of student newspaper:organizational websites
websites:foreign sources, fiction
webzine:publications
wedding:similar occasion, set, attendees, service, event, prior, Punjabi
    social occasion, joyous occasion
wedding band:functional accessories
wedding bands:wearing, applications in consumer products
wedding celebrations:playing music
wedding dress:part of very formal outfit
wedding feasts:indulgent affairs
wedding of Bouncing Boy:events
wedding of Charles:major royal events
wedding reception:social occasions
wedding receptions:Scout activities, short-term rental, formal events
wedding rings:personal nature
weddings:celebrations, worldly activities, large social gatherings, similar
    services, special events, church, gatherings, dessert of choice,
    functions, outdoor social events, Jewish celebrations, life-cycle events
    , formal occasions, events, days of religious ceremonies, intended,
    public use, large functions, community events, holidays, special
    portraits, women in formal situations, extra special occasions,
    ceremonies, sorts of Beti gatherings, number of social gatherings,
    important ceremonies, special occasions, individuals
```