Slide 4, dataflow diagram—the arrows to the RDDs at the top are TF-IDF calculations, i.e., “doc TF-IDF”, “concept TF-IDF”, “sentence TF-IDF”.

Script:

Here is a dataflow diagram of the QA system. Some of this functionality is not yet implemented—in particular, this RDD of concepts has not yet been prepared—but the framework is there. Let me walk you through how it works.

The raw corpus is, as I said, a PDF. There are some manual operations involved in converting it to a text file, then breaking that up into topical pieces. I broke it by chapter, so each chapter is a document. There were various challenges with this first step. Obvious tools like Adobe Acrobat ought to do this conversion well, but resulted in lots of artifacts, like words strung together without spaces in between. It turns out that Microsoft Word does a better job of pdf-to-text. Breaking it up was done manually, but could be automated—that’s a natural language project of its own.

After we have text documents, we process them through three pipelines at the document level (that’s our chapters), and then the sentence level, and then the concept level. Each of these also gets TF-IDF data at that level, so we can make relevance determinations using TF-IDF at the document level and at the sentence level.

Adding manually-constructed concepts, which are a kind of digest of the contents of the document, was an idea I had at the last—that’s this center pipeline. This pipeline is not yet fully implemented, but the documents have been tagged with a digest of relevant nouns on the first line of each document. The idea is to extract concepts—especially multiword noun phrases—in the question, and then match those to the concepts extracted from the document to get a second measure of document relevance. This would be combined with the TF-IDF information to get a better score at the document level. Only when the document first scores highly do we move to the sentence level—this is essential to scale the system in a meaningful way.

One of the interesting aspects of the concept-based scoring is that it captures relevance that may be hidden by assumption. For example, the text I have here covers the Python language. But the word, Python, appears rarely in the text, since it’s obviously a Python book. Some chapters never mention the word, Python—so I have documents that will show zero relevance to the word, Python.

Now, if we have a bunch of books—some Java books, Python books, Scala books—and a user asks, “Does Python have array slices?”, we should match our chapter documents based on the fact that the user wants to know about Python, and this is a Python book. So the Python concept is hand-coded as a kind of tag, and we score on that in addition to relying on TF-IDF.

As for the RDDs themselves, I’m extracting the lemmas, part-of-speech, and named entity classes for all of the tokens, and that’s all. Documents are logically comprised of a sequence of sentences—I’ll show you the object diagram shortly—so that the system can become more sophisticated later as we can do some proximity searching.

One other thing of note here is that we are storing these RDDs as object files, so we can read them in at a later time—so I’ve properly separated all the corpus prep from the actual QA system.

Now for the QA system—let’s start at the bottom with a question. First we lemmatize it, and from this point, forward, we’re just talking about the lemmas. We run the question through the NLP pipeline, which is already initialized and resident in memory, so it’s nearly instantaneous from the user’s perspective. We extract the nouns and verbs from the question, and those become what I call “target terms.” We look up the TF-IDF hashes for those an d save them for later TF-IDF comparison. We also extract the type of the interrogative. Currently my system supports who, what, what is, when, where, and why question types, with varying levels of sophistication. If the question type isn’t one of those, then the answers will just be poorer—more on that in a moment.

The next step is the last bit of functionality that I added—we expand the target terms based on synonym expansion. So, if a user uses the word “function,” we might add the words “method” and “procedure” to the target terms, because they are near synonyms. This technique is somewhat naïve at present, because all added terms are assumed to have the same degree of sameness. A whole semantic graph would be better. The problem with the general ones out there is that they are not specific enough for the domain—so really, a domain-specific one needs to be constructed.

So the expanded query gets scored twice at the document level—once for the whole document text, and once for the concepts—then gets scored at the sentence level for those documents that are most relevant. Then the five best sentences are selected for further processing.

Processing at the sentence level runs like this: We have a default answer, which is to just return the whole sentence—maybe the answer is in there somewhere! If we have a known interrogative type, we try to find it. For example, a “WHY” interrogative means we want to find a clause that begins with “because,” and then return from the word “because” to the end of the sentence—that’s our best shot at an answer. If we can’t find such a clause, we return the whole sentence text.

All five of those answers get returned, which increases our chances of giving the user a useful answer. If we limit it to just the best one, we do get it sometimes, but it’s not very frequent.

<<Class Diagram>>

Here’s a class diagram that shows the implementation classes. Note that some of these—Document, sentence, Token—these are not the Stanford CoreNLP classes of thse names, I made my own classes that get stored in the object files and distributed as RDDs. I did that because the CoreNLP classes maintain a lot of data as strings—for example, the part-of-speech and named entity class tags are strings—so you have all these string equality operations all over your code. I wanted it to be more performant, so I created these enumerated classes here [Named Entity Class, PartOfSpeech] so it would be reduced to fast integer comparisons—this stuff is happening in inner loops, and I wanted it to be performant.

Another thing I did that I felt was a good architectural decision is to extract out the abstract class here [AbstractDocument] which contains the code common to handling Documents and Sentences, so I can do all the TF-IDF scoring without any concern for what kind of thing I have there—the code is pretty clean as a result. You know you’re getting somewhere when the code does more each iteration, and is getting shorter and clearer.

[Discoveries] I would say “synonyms and other concept relations for technical content…” “manually created the concept relations”

“topic discovery was weak, and really needed to be done manually. Preparing the corpus is very manual work, and is a project of its own.”

Add: “Python code was a problem, since our NLP pipeline is trying to interpret it as English. I think the best practice here would be to recognize it and ignore it, because in a textbook, the code doesn’t really need to be queried.”

[Improvements]

Eliminate second bullet point, replace with “Improve the corpus preparation, and extend the corpus!”