COLLABORATIVE TEXT MANIPULATION FOR FINE-GRAINED EDIT DATA COLLECTION

Solving some natural language processing problems requires lots of data. For example, Statistical machine translation (SMT) models use large parallel corpora of sentences translated from a source into a target language. These corpora are expensive to procure, especially to solve the machine translation problem in its most general case, from any language to any other language in any domain. So, I created a web application to help get data on where mistakes occur in a way that helps people who don't speak the original language well contribute meaningful edits anyway. The latter part in particular requires the application to maintain linking between target-language words and source-language words (illustrated in Figure 1), and requires the maintenance of connections between words across different versions of text as it is edited. Additionally, fine-grained edit history can be used in novel ways, both in enabling real-time feedback for human editors (e.g., suggestions), and incorporating human feedback into a human-in-the-loop MT mechanism, such as by applying statistical methods for automating the iterative improvement of translation output. My goal is to create an application that is able to generate a large dataset for future research in this area.

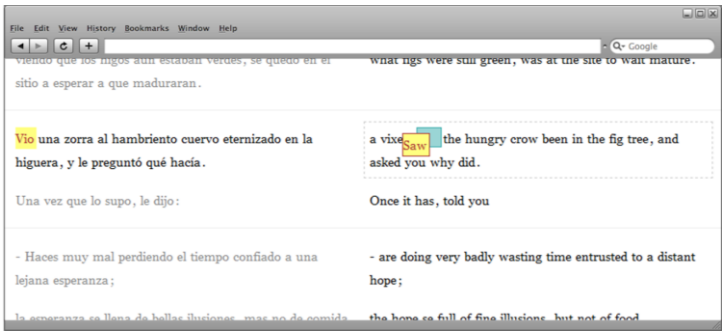

Figure 1. Linking between tokens in different versions of the text is a vital feedback mechanism that allows users to "debug" errors in text

The following at the main challenges:

1. maintain information that allows both providing quick access to the state of text at a given time and recovering the full history of edits done to text
2. maintain enough information to be able to link tokens from any version of the edited text to any other version
3. maintain metadata about edits done by multiple users – not only who enacted an edit, but any other appropriate information (e.g., discussion of an edit)
4. store this information in a way that still allows write operations to be passably fast, over a sizeable corpus of text
5. make the database (and accompanying application) deployable at plausible production scale as a web application for data collection

In the Approach section, I will describe a method for accomplishing 1, 2, and 3, and in the Evaluation section, I will describe how I will test the performance in terms of 4 and 5 against a more precise definition of "passably fast," and "plausible production scale." Outside the scope of the class project is deployment of the web application, the front end for which has been already designed and developed.

APPROACH

I propose the following schema for this problem. First, a representation for describing text documents:

- Document(id, lang_id, added, doctype_id, textnode_id)
  - Doctype_id is a foreign key to DocType.id
  - Textnode_id is a foreign key to TextNode.id, which is explained later
- DocType(id, name)
- Language(id, name, shotcode)
  - *represents a short list of languages*
  - shortcode is the unique 2-character code for languages popularly used (eg, ru, en)
  - name is also unique and is the full name for that language (eg, Russian, English)
- DocAttrType(id, name)
- DocAttr(doc_id, docattrtype_id, value)
  - *represents a value for an attribute*
  - doc_id and docattrtype_id are foreign keys into doc and DocAttrType, respectively
- DocRelType(id, name)
  - *Represents a type of relationship between documents – eg, a copy of it to translate to another language, for example*

- DocRel (id, parent_doc_id, child_doc_id, docreltype_id branched, user)
    - *Represents a hierarchical instantiation of such a relationships – eg, this is a translation of that*
    - Parent_doc_id and child_doc_id are foreign keys to doc.id; docreltype_id is a foreign key to docreltype

For example, the book *Hunting of the Snark* could be represented by the following rows:

- in Document: (1, 1, 2012-05-15 3:00pm, 1)
- in DocType: (1, "book")
- in Language: (1, "en", "English")
- in DocAttrType: (1, "author"); (2, "title"); (3, "source")
- in DocAttr: (1, 1, "Lewis Carroll"); (1, 2, "Hunting of the Snark"); (1, 3, "Project Gutenberg")
- in DocRelType: (1, "translation")
- in DocRel: (1, 2, 2012

The goal here is to support storing documents without a prespecified set of attributes; no complex queries are expected, beyond listing available documents and retrieving the metadata for a document, so this is fine.

Then, we represent the actual text of a document as individual words, referring to a dictionary, arranged in chunks:

- Word(id, content)
    - *contains, kind of, a language-agnostic dictionary. Content is unique here (case-sensitive).*
- Node(id, child_id, word_id, depth, order, edit_from_id, edit_to_id)
    - (id, child_node_id, edit_from_id) is a unique key
    - Child_id is a foreign key to Node, but may be null when depth=1. Same for word_id, which must be nonnull when depth=1, and may have data otherwise (eg, chapter heading, as in Figure 2).
    - edit_from_id and edit_to_id is a foreign key to text_edit
    - edit_to_id is 0, which is a special "no edit" edit, if the node has never been removes / altered
    - depth never changes, and its reasonableness is enforced at the application level
- Edit(id, user_id, enacted)
    - enacted is a timestamp for when an edit was enacted
    - may contain additional information (eg, foreign key to a table of threads)
    - user_id is a foreign key to a table of users; because using an existing plugin for user account management, that set of relations is left out entirely and user_id is used where needed
    - this is referenced by two other relations affected by edits – text_edge and text_order

The main point here is to embed the temporal versioning information within the structure of the data. In this case, each node in the tree is associated with an order, and exists only between the edit that creates it and the edit that removes it. The tree hierarchy helps to localize updates required upon edits that affect the order of nodes.

Consider the very short text represented by the tree in Figure 2. Each node in the tree shown is associated with a tuple in the Node relation, with the specified depth. Nodes at the same depth *d* are ordered by the numeric Node.order attribute, allowing text to stay in order. This tree can be quickly constructed for a given time, and by conditioning upon the time of the creating edit (Node.edit_from_id) and either the nullness of the destroying edit (Node.edit_to_id), meaning that the node was never destroyed, or its time.

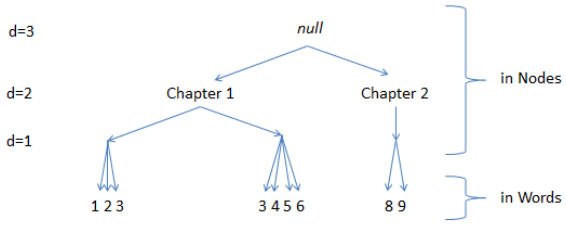

Figure 2. A tree for a hypothetical text with two chapters.
The first reads, "1 2 3. 4 5 6 7." and the second, "8 9."
Let's say those numbers are the id's of the words, from the Words table.

Let us consider several of the tuples shown above, from the second "sentence" in Chapter 1, stored in Node(id, child_id, word_id, depth, order, edit_from_id, edit_to_id):

- (7, *null*, 3, 1, 1, 1, 0)
- (7, *null*, 4, 1, 2, 1, 0)
- (7, *null*, 5, 1, 3, 1, 0)
- (7, *null*, 6, 1, 4, 1, 0)

Where the Edit(id, user_id, enacted) table contains the following corresponding tuples:

- (1, 345, 2012-05-15 9:15AM)
- (0, 0, 0) *this is a special no-op edit, which can have any values for the other two fields and is interpreted differently*

And "345" is a user id associated with the user who uploaded the text in the first place.
In order to get that sentence, we need to execute a simple select with a join, given some time *time* at which to get the state of the txt:

```
SELECT content FROM Node n
        JOIN Edit a ON n.edit_from_id=a.id
        JOIN Edit b ON n.edit_from_id=b.id
        JOIN Word w ON n.word_id=w.id
        WHERE a.enacted<=time AND (b.id=0 OR b.enacted>time)
        ORDER BY n.order DESC;
```

So the interpretation for the time interval is inclusive on the left/from end, and exclusive on the right/to end.
This provides all the words under the depth=1 node. But if we start at the root, which is depth=3, we have to adjust to a query with more joins (but a small, known number, given the known depth):

```
SELECT content FROM node x
        JOIN node y ON x.child_id=y.id
        JOIN node z ON y.child_id=z.id
        JOINE word w ON z.word_id = w.id
        WHERE [x y and z all exist at time]
        ORDER by x.order, y.order, z.order
```

With a few complications to allow fetching chapter titles and so on.
This complication of representing text in a wide, shallow tree-like (tree with ordering on nodes) structure allows localizing update on edit. The edits possible are:

- **Delete** a word: requires only updating the given node with a nontrivial edit_to_id. The adjacent words can still be ordered reasonably, even if a number is missing in the middle; the order attribute in node is not presumed to have consecutive values, only monotonically increasing.
    - Eg: the sentence is "A B C", which have order attribute values 1 2 3. On removal of "B", order attribute values 1 and 3 remain for A and C.
- **Insert** a new word: *may* require updating all the nodes that follow it to have a bigger "order" number, *if* no gap exists. When this occurs, inspired by resizing strategy of many data structures, the order attribute of the right-adjacent node is doubled, rather than incremented by one. So if the right-adjacent value was *k*, it is now *2k*, and *k* is added to every following element:
    - Eg: if we add B back into the above sentence, no changes ot adjacent nodes are necessary, because there is a gap. But, if we add "D" to the front, to get "D A C," we update the order values to "1 2 4"
    - To "update order values" means to effectively delete previous nodes by setting the edit_to_id to a nontrivial edit that is occurring, and to create new nodes with that same edit as the edit_from_id
- **Re-order** a word to another location:
    - This requires updating the order values of all the words in between, with the "update" notion described above.

Another relation is used to maintain linking between words in different verisons:

- Link(id, edit_id, original_node_id, edited_node_id)
    - edit_id id a foreign key to Edit.id
    - original_node_id and edited_node_id are foreign keys to Node.id; at the application level original_node_id must be one that refers to a node that existed in the very first version of this text

This is updated in the fourth, and most common, element in the vocabulary of edit types:

- **Replace** a word with one or more new words:
    - May require the same action as insertion if inserting multiple words
    - Main distinction between a replacement and a deletion followed by insertion is that this also requires an update of the Link relation.
    - Eg, if we edit "A B C" to be "A F C," replacing B with F, there will be a tuple added to Link, from the original "B"-containing node to the new "F"- containing node. Then if we later edit "A F C" to be "A G C," replacing F with G, we add a new tuple to Link, from the original "B"-containing node to the new "G"- containing node.

This way, given any node *node* and time *time*, we can find all the other nodes that are linked to *node* in time *time.* In other words, we will be able to know that F is semantically related to G, and provide this as historical feedback during editing, which is an important interaction in the data collection application. To do this, we can:

```
SELECT y.edited_node_id FROM Link x
    JOIN Link y ON x.original_node_id=y.original_node_id
    WHERE x.edited_node_id=node AND [y.edited_node_id exists at time time]
```

Because we want this to be synchronously collaborative by multiple users (on the order of 10s of users, plausibly), we will implement explicit locking on nodes. Storing locks in the database will enable us to show it in the UI.

- Lock(id, node_id, user_id, from)
    - *Locking is binary, but if the application becomes unresponsive for a user, we must be able to time a lock out, so we maintain the time at* from
    - *Additionally, a user may not edit more than one node at once*

Locks would be places by the application based on the interaction; in the current case, locking makes sense at the level of depth=2 nodes.

The proposed design embeds time/version information in the structure of the data itself, rather than at a sequence of edits; this is a much simpler problem, taking an approach similar in goal to [2]. Because in addition to maintaining a record of edits, it will be necessary to enable the editors of translation to carry out conversations as they collaborate on changes. In this way, the challenge is similar to that addressed by DB-Wiki [1], but limited to text; the proposed design allows adding additional metadata to edits, in the Edit relation, such as reference to a discussion thread, and so on.

EVALUATION

The main concern is first to determine whether this works *quickly enough*. Because a UI would become burdensome if operations take more than 10ms, and each write and read operation is expected to be performed upon interaction, I will time those operations to verify that they stay within the responsiveness goal of 10ms. Ideally, they would not exceed this, as even more time would be added by UI rendering/networking overhead, as well, but this is a passable initial metric. I will time the performance of the queries discussed in this document on an increasingly large dataset in response to artificial, random edits that approximate the expected rates of real users of each kind of edit, based on prior studies of this interaction. The text to be edited will go from a dozen to several thousand texts. The starting dataset contains segments from Project Gutenberg texts by various authors; I have parsed these to exclude irrelevant artifacts that are not part of the text (legal notices, illustrations, etc.). The first test will be to insert these texts into the database, timing how long it takes to insert each next text, and how large the database gets on a natural dataset like this, relative to theoretical worst-case. I will time each insertion of a new text, as well as the retrieval of a few random texts from the database. The cleaned version dataset has 100k chunks taken from 6k distinct literary texts with some associated metadata; I also have, for reference, the originals of every document with complete available metadata. Each of the chunks has an average of 8k words; this count varies widely across chunks and documents. In the database, these chunks will be mapped onto depth=3 nodes of each document, depth=2 paragraphs and depth=1 sentences within each part as described. I will have a Java program to then apply random edits to this document.

GOALS

This work has thus far involved design of a UI, an initial prototype, and preliminary experiments with the web-based prototype [3]. The existing application is built using PHP, using mysqli to access a MySQL database (on a UW webserver) and create appropriate abstractions for the application layer. A new implementation of the database may require changes to the specific queries needed, but the abstractions will remain unchanged, allowing the consistent use of the same front end design. As described in evaluation section, I have compiled an initial text dataset. As I noted in the proposal, by this point I have fleshed out the design of the database and begun to implement the evaluation application. I will complete the implementation, and test with my sample data, in the next two weeks.

REFERENCES

- Buneman P., Cheney J., Lindley S., Mueller H. The Database Wiki Project: A General-Purpose Platform for Data Curation and Collaboration. ACM SIGMOD Record 2011.
- Buneman P., Khanna S., Tajima K., Tan Wang-Chew. Archiving Scientific Data. ACM Translations on Database Systems 2004.
- Kuksenok K., Brooks M., Bangalore S., Fogarty J. Collecting High-Quality Fine-Grained Human Feedback on Machine Translation Errors with *ChoiceWords*. NW-NLP2012.