Main database

When other scholars are using the Java application we build, they'll be working with a database of "volume parts" that is not broken down to the page level. A page-level database is in practice too large to distribute to humanists.

But we can't create a volume-part database without segmenting volumes at the page level — and in some cases, even below that level. So most of our work at present will be based on a table of word occurrences broken down by volumeID and pageNumber.

Formally, this is a "sparse table." It has rows only for words that actually occur on a given page of a given volume. It doesn't have a lot of zeroes for the words that *don't* occur.

The data is represented as a comma-separated text file: i.e., fields are separated by commas and each line is terminated by a new-line character. In practice we will transmit these files after compression by bzip2, which can achieve tenfold compression with this sort of highly-redundant text. The fields are

```
volumeID, pageNumber, formFlag, word, numOccurrences
```

`volumeID:` a string that maps onto a HathiTrust volumeID

`pageNumber:` a positive integer, starting at zero because we're living in JavaWorld and integers start at zero. It will never be larger than 32,000 and could be represented as a "short" within Java.

`formPart:` will be either 0, -1, or -2. Rows containing zero are features that apply to the whole page. FormPart -1 means that the row contains a feature count that applies only to the *prose portion* of a page. FormPart -2 means that the row contains a count that applies only to the *line-capitalized* portion of a page.

> For the purposes of this database, a line of text counts as *line-capitalized* if it is in a sequence at least five lines long where the first alphabetic character of each line is upper-case. When we're measuring the length of a sequence, we can follow it across page boundaries, but in the database, the rows for page N are of course only describing word counts on page N. Moreover, we never count the *final* line of a capitalized sequence as part of the line-capitalized portion. We assume it's

the beginning of a prose sentence after the passage of verse, or what have you, that was capitalized. See `/tokenizer/Volume.py` for relevant code.

`word:` a string representing the feature that's being counted. My tokenizing script ignores case when counting words. It doesn't do any stemming, except that apostrophe-s is generally truncated. Words will not be more than thirty characters long.

There are several special "words" that should not be taken literally. `Arabicnumeral` and `romannumeral` are catch-all designations for numbers. Also, there's a "word" called `titlecasenodict` that is a catch-all designation for words in Titlecase that I can't find in the dictionary. Those are often proper nouns, and the frequency of proper nouns can be a significant clue about genre.

Two other features that apply to the page as a whole: `#textlines` and `#caplines`. The `#textlines` feature tells you how many lines of text there are on a page, and the `#caplines` feature tells you how many of those are capitalized.

We can use these integers to create a couple of ratios that we might use for purposes of classification or clustering.

> #textlines for this page / mean #textlines for the whole volume
> is one interesting ratio: it tells you how much print or white space is on a page

> #caplines on this page / #textlines on this page
> is another interesting ratio

both of those ratios may tell us something about page structure. They'll help us recognize tables of contents, etc. The third ratio we'll want to construct for clustering pages is

> this page number / total number of pages in the volume

`occur:` an integer providing the number of the specified "words" that occur in the specified combination of volume, page, and formal subsection of the page. It will never be larger than 32,000 and could be represented as a "short" within Java.

How to represent the fields that are strings internally in Java:
Possibly, after we load data in to Java, we'll want to convert the strings into a sequence that we can refer to using only the integer indices -- or possibly not. It depends on

whether you think that's helpful for performance. When I'm writing programs, I usually find that integer values make it easy to index into arrays, but I guess we could just use hash tables for the same purpose.

Converting strings into a sequence of integers may be slightly tricky, because in order to come up with an integer sequence we would need to know the full range of DocIDs and words included in a given dataset.

But it's only slightly tricky. For instance, the first stage of a workflow might be a join, to select certain DocIDs from the larger dataset.

Then we could do a map-reduce stage whose sole function was to count distinct DocIDs and words, mapping them onto integers. (At this stage we might also choose to work with only (say) the 5,000 most common words in this particular corpus. 5000 is just an example; the actual number might vary.)

Then we could write our intermediate stage of data as a sequencefile, and proceed with whatever algorithm we wanted to run on the corpus.