A QUERY-ORIENTED APPROACH TO CORPUS ANALYSIS WITH SQL AND MODERN RELATIONAL DATABASE MANAGEMENT SYSTEMS

A Thesis

Presented to

The Honors Tutorial College

Ohio University

In Partial Fulfillment

of the Requirements for Graduation

from the Honors Tutorial College

with the degree of

Bachelor of Science in Computer Science

---

by

Nathaniel Ekoniak

June 2006

ABSTRACT

This paper explores the applications of relational database management systems to some common tasks in Corpus Linguistics, including various statistical analyses, concordances, collocations and searching. Simple table design for corpora is discussed and techniques for examining corpus data and statistics are presented using the SUSANNE corpus (an annotated subset of the Brown corpus). Programmer efficiency in the relational database approach are compared with traditional iterative programming techniques, and the advantages and disadvantages of the relational approach are analyzed.

Table of Contents

1. Introduction

A common complaint in the field of Computer Science is that programmers, engineers and researchers are constantly "reinventing the wheel"—wasting countless hours writing programs to do things that have already been done. Recently, the prevalence of so-called "scripting" languages—Perl, Python, and others—has alleviated some of the burden with task-oriented modules, powerful built-in data structures and high-level interfaces which remove the programmer from many of the nitty-gritty details often associated with programming. However, for non-programmers, the theory and implementation of even relatively simple programs can be a daunting task requiring intimate knowledge of algorithms and data structures. Additionally, the use of these languages is not, in and of itself, truly a move away from redundancy; it reduces the time spent in development and debugging, but in the end we have merely streamlined the wheel-invention process. Unfortunately, for all but the most common and pedestrian of tasks there is often no easy solution.

For some types of tasks, however, there is an oftentimes-overlooked solution: modern relational database management systems. Relational databases offer a consistent interface to data and stores it in a regular fashion. Consequently, the user of a relational database does not need to be concerned with the actual data structures in which the data is stored or the algorithms used to access or update them or optimize these operations—these are all taken care of by the RDBMS itself. Whereas scripting languages still, at their root, use the same traditional style of iterative programming, relational databases offer SQL, a standard language for relational data access and

manipulation, which emphasizes the relational model and data abstraction. This paper investigates the advantages and disadvantages of the relational model and using relational database management systems to accomplish tasks in the analysis of annotated corpora.

This paper offers to those faced with this problem of gathering data from corpora a basic model for the creation and use of relational database management systems with an annotated corpus. Although the use of relational databases has been occasionally investigated, implemented successfully and documented (Davies *Advantages*), the relational model has been considered as something that must be worked around and thought of as ill-suited for storing strings of text (Lawler and Dry). The approach in this paper, however, embraces the relational model for the storage of sequential textual data; as a result, the user can take the maximum advantage of relational database technologies for the abstraction of data, efficient access and the powerful built-in functionality of modern relational database management systems.

2. Background

Corpus linguistics, a sub-field of linguistics, is an expanding discipline which merges empirical data—corpora—with linguistic theory. Observations gathered from this data are used for many purposes, like probabilistic modeling, machine translation, static and dynamic description of stylistic language, and improving the development of accurate, real-world linguistic models (Kennedy 8-9). Analysis of corpora generally involves the annotation, (be it manual, computer-assisted or automatic) of

text with relevant linguistic information (e.g., part-of-speech tagging or lemmatization) and subsequent statistical analyses on the annotated text (McEnery and Wilson 32-53). Researchers have at their disposal many different kinds of tools for this analysis, but they fall mainly into three categories: commercial tools which may be expensive and are often, by their nature, closed to modification, open-source tools which are freely available but frequently lacking in depth of features, and custom tools created by individual linguists themselves (Hammond 2).

While the ability to create custom programs is a useful, even necessary skill for language researchers wishing to leverage the ability of computers to automate tasks such as annotation and analysis of large bodies of data, it is unreasonable to expect all excellent researchers to also be excellent programmers. Unfortunately, the exploratory nature of research can often require tools which are not readily available to researchers. While this is not a problem for researchers positioned within large companies with in-house programmers and deep pockets, it can be a prohibitive obstacle to many others. As such, linguists are forced to become programmers and software architects, taking crash courses in introductory programming just so they can get the data that they need to continue their work (Mason 4). Tellingly, there are a number of books available(Hammond, Mason) which attempt to give linguistic researchers a "crash course" in programming. The knowledge and skill needed for most tasks, however, is beyond the scope of an introductory book; computer programmers study for several years to learn them.

The skills required to create good software are themselves numerous and varied, spread across their own fields like Software Engineering and Computer Science. Those coming from other fields may not be familiar with the breadth of tools available, the possibilities and limitations of those various tools, and the most efficient way to use these tools to accomplish a task. Linguistic researchers without a computer programming background are fundamentally limited by this lack; programming can be, even for students and professionals in the field, a laborious and difficult task.

This kind of cross-discipline effort also rarely results in efficient (or even functional) coding. A common problem with any kind of data parsing is what may be termed the "shampoo-bottle method": parse, process, repeat. For every question, a new program must be written or adapted from an old one, and that program must read in the data, parse it into its internal data structures, and process the data for the information it is looking for. When the program is finished, it outputs the results and exits. If the researcher wants to ask another question, no matter how slightly it differs from the one just answered, the whole process must begin again. Larger, more complicated programs may avoid this by keeping the data in memory and allowing the researcher to adjust queries and even see results in real time, but the resources required to design such a program can be staggering—perhaps requiring a team of experienced programmers several years. When the question is not something that can be readily answered by existing tools, or those tools are not available due to lack of funding or restrictive licensing or for other reasons, however, the only recourse may

be to create or adapt a single-purpose program to accomplish the task. Even this adaptation, while it may reduce the time needed to program a solution, can result in wasted time and effort when an inexperienced programmer does not understand the original code, or does not consider a more efficient or correct solution because it would require starting over.

This is where modern relational database management systems come in to play. With powerful relational database management systems freely available for download, researchers can leverage their power to improve their ability to perform linguistic research with large corpora. The algorithms needed for data manipulation and access such as searching, sorting, storing, retrieving, displaying and more, have already been implemented, optimized and given a standard interface. Although the use of relational databases for storing corpora has been investigated (Nerbonne 2), the potential of the RDBMS to enhance the efficiency, flexibility and simplicity of certain corpus research has not been investigated in depth.

3. Related Work

Mark Davies, Professor of Corpus Linguistics at Brigham Young University, has created several searchable online corpora using relational database technology, most notably the *Corpus del Español*[1], using Microsoft SQL Server (Davies *N-gram*). However, the format for the storage of corpus data I implement in this paper is somewhat different from his approach. The actual text of the Corpus del Español is

[1] Accessible online at «www.corpusdelespanol.org»

not stored in the same way as the annotation data; rather it is stored in a "1000-2000 word chunks of text…not annotated in any way, apart from a code that indicates the source of each block of text."(23) Not only does does this not conform to the relational model, but for corpora in languages with a high degree of polysemy it becomes difficult to disambiguate word senses. However, the "vertical" format (*Advantage* 318) that is implemented and used in this paper allows us to store the corpus text on the same level as the annotation. By storing each token in its own row, there are a number of benefits which would are not available in the format Davies uses, such as per-token annotation and finer-grained searching. Furthermore, Davies imports n-gram data from an external program, WordSmith, with frequency data grouped by the century of the source material (319). Since the database does not contain a fine-grained copy of the original text, frequency information cannot be obtained for categories more specific than those imported originally. Using the techniques described in this paper, however, additional frequency information can be obtained using SQL queries directly inside of the database application at any time using arbitrary annotation categories.

4. The Database

The relational database management system used for this research is PostgreSQL 8.1, available at «http://www.postgresql.org/». As the aim of this paper is to show how this method can be used by any researcher regardless of their programming background or available resources, an Open Source database which can be

freely downloaded and will run on most modern architectures was chosen. In addition to meeting these requirements, PostgreSQL can be used freely for any purpose, being released under the BSD license; this means that derivative works can be freely redistributed, including for use in commercial products, without restriction.[2] This paper does not cover installation and setup of PostgreSQL. However, setup instructions can be found at the PostgreSQL website, and for many common operating systems there are programs to accomplish the steps necessary to install it automatically. Other RDBMSes such as MySQL, MS SQL, ORACLE, SQLite and others could also be used. All interaction with the database is done using the bundled PostgreSQL client `psql`. Clients with graphical interfaces, such as phpPgAdmin[3], are also available freely available.

5. The Data

5.1 The SUSANNE Corpus

In order to provide any useful examples, a reasonably-sized corpus with at least one level of annotation was necessary. I have also chosen a freely available tagged corpus, the SUSANNE[4] corpus. The SUSANNE corpus is a subset of the Brown corpus, a well-known megaword corpus of American English. The SUSANNE corpus consists of about 130,000 words, lemmatized and tagged for part of speech

---

[2] see http://www.postgresql.org/about/licence

[3] http://phppgadmin.sourceforge.net/

[4] SUSANNE stands for "**S**urface and **u**nderlying **s**tructural **an**alyses of **n**aturalistic **E**nglish" (Sampson)

and structure. This corpus, along with several other related corpora, are available at «http://www.grsampson.net/Resources.html» (Sampson). Although the SUSANNE corpus is tagged with extensive information including tree structure, it is beyond the scope of this paper to investigate the use of all fields. Rather, this paper focuses on demonstrating how to accomplish several useful types of tasks with the data available.

5.2 Table Design

The basis of the all relational databases is, unsurprisingly, *relations*, more commonly known as tables. Data is organized into columns and rows, with each column representing the type of data and each row representing a record. Relational database management systems are tools for storing, managing, accessing and manipulating the data in these tables (PostgreSQL 6). Using the SQL query language, one can perform a wide array of functions.

Figure 1 shows a small excerpt of data from the SUSANNE corpus, as distributed. As a text file, this information is difficult to process without a solid knowledge of programming and text processing; as a table in a relational database, however, many complex tasks can be carried out almost immediately using SQL, skipping quite a few steps that would be necessary before even getting started when gathering data using a language like Perl or Java. For most of the data in this project, however,

```
A01:0030.06 -  NN2   irregularities   irregularity   .Np:s]
A01:0030.09 -  VVDv  took             take           [Vd.Vd]
```

Figure 1. An excerpt from the SUSANNE corpus, as distributed.

File *text*	Index *integer*	Status *char(1)*	Tag *text*	Token *text*	Lemma *text*	Parse *text*
A01	3006	—	NN2	irregularities	irregularity	.Np:s]
A01	3009	—	VVDv	took	take	[Vd.Vd]

Figure 2
An excerpt from the SUSANNE corpus represented in a single database table.

there is a very simple 1-to-1 relationship between each field and its associated index. The main database table has the same structure as the distributed corpus data (figure 2). Figure 3 shows how additional data may not have the same 1-to-1 relationship, and should be stored in a separate table, linked to the main table by the unique pair of file and index.

5.3 Importing

File *text*	Start *integer*	Stop *integer*	Sentence *integer*
A01	1006	3015	1
A01	3021	6048	2

Figure 3
A theoretical "sentence" table defining an additional level of annotation to the corpus.

Unfortunately, there is no magic bullet for converting data from one format into another; the task can be trivially simple or extremely difficult depending on the type of data and the difference between the original and target formats. Fortunately this step need only be done once; after the data has been imported into relational tables, it need not be parsed again, and can even be exported and distributed in a database-independent manner. When writing single-purpose programs, this step must be repeated every time the program is run and every time another program is written.

In this case, the source and target formats are quite similar, and PostgreSQL, along with most other databases, has import functionality for regular, delimited data distributed as one-record-per-line text files In fact, with PostgreSQL's COPY functionality, the tab-delimited SUSANNE files could be imported directly into a suitable database table exactly as distributed. However, since it is somewhat useful to split the first field, originally a concatenation of the filename, some minor reformatting was done on the original text to replace the 4th character (always ':') with a tab ('\t') and to replace the line number with successive integers (it will be made clear later why this is useful). In this case was done using a small perl script (see figure 4) which only needed to be created and run a single time.

```
while (<>)
{
  s/^(...):.{7}/$1\t$line/;
  print;
  $line++;
}
```

Figure 4. Basic PERL script for reformatting the SUSANNE corpus files.

```
CREATE TABLE main (
      file text,
      index integer,
      status char(1),
      tag text,
      token text,
      lemma text,
      parse text
);
```

Figure 5. Main table creation.

```
susanne=# \copy main FROM A01.tab DELIMITER '\t'
```

Figure 6. Data import with \copy.

Table creation is fairly straightforward. Figure 5 shows the usage of the CREATE TABLE command along to create the main database table corresponding to the structure of SUSANNE corpus files seen in figures 1 and 2. Since the newly-created table and the data files have the same fields in the same order, PostgreSQL

`COPY` or psql's `\copy`[5] functionality is capable of importing the data from the text files (figure 6).

5.4 Annotations

Additional annotation can be added to the corpus in the form of additional database tables which reference the original table by certain columns. Included with the SUSANNE corpus, for example, is source data for each of the source texts

```
CREATE TABLE sources (
      file text,
      title text,
      author text,
      source text,
      year integer
);
```

Figure 7. Source table creation.

```
G01 Edward P. Lawton, "Northern Liberals and Southern Bourbons", The Georgia Re-
view, 15 (1961), 254-259
G02 Arthur S. Miller, "Toward a Concept of National Responsibility", The Yale Re-
view, LI:2 (December 1961), 186-191
```

Figure 8. Source: The SUSANNE Corpus: Documentation.
Excerpt from source listing.

File *text*	Title *text*	Author *text*	Source *text*	Year *date*
G01	Northern Liberals and Southern...	Edward P. Lawton	The Georgia Review	1961
G02	Toward a Concept of National...	Arthur S. Miller	The Yale Review	1961

Figure 9. A table describing source attributes. Data is truncated here for display only.

```
susanne=# INSERT INTO sources VALUES ('G01', 'Northern Liberals and Southern
Bourbons','Edward P. Lawton','The Georgia Review','1961');
INSERT 0 1
susanne=# INSERT INTO sources VALUES ('G02', 'Toward a Concept of National
Responsibility','Arthur S. Miller','The Yale Review','1961');
INSERT 0 1
```

Figure 10. Inserting data into the source annotation table.

[5] PostgreSQL `COPY` requires database superuser permissions. The psql client's `\copy` uses the same syntax but requires no special permissions (PostgreSQL 1041).

(figure 8). Note that instead of the typical one-to-one relationship with each row in the main table, this table has one row for each original file (figure 9). Completely new data can be entered using INSERT statements (figure 10).

```
susanne=# SELECT tag, token, lemma
FROM main LIMIT 5;
  tag   |  token   | lemma
--------+----------+--------
 YB     | <minbrk> | -
 AT     | The      | the
 NP1s   | Fulton   | Fulton
 NNL1cb | County   | county
 JJ     | Grand    | grand
(5 rows)
```

Figure 11. Using SELECT … FROM … LIMIT to retrieve a few rows from the database.

To look at the data, use the `SELECT` command. SELECT, followed by a list of columns, functions, or `*` for all columns, followed by FROM and the name of a table, will display the data in that table. When just looking at a table for an example it may be useful to place a `LIMIT` clause at the end of your query; e.g., `SELECT tag, token, lemma FROM main LIMIT 5;` will return only the tag, token and lemma columns for the first 5 rows (figure 11). It is worth noting that the order of rows in a table is never guaranteed and should not be considered important unless explicitly sorted by an `ORDER BY` clause (see section 6.2).

```
susanne=# SELECT token, author FROM
main NATURAL JOIN sources LIMIT 5;
  token   |      author
----------+------------------
 <minbrk> | Edward P. Lawton
 NORTHERN | Edward P. Lawton
 liberals | Edward P. Lawton
 are      | Edward P. Lawton
 the      | Edward P. Lawton
(5 rows)
```

Figure 12. Using NATURAL JOIN to combine the annotation table to the main table.

5.5 JOIN

The most important feature of the relational database is the `JOIN`. Although there are several types of joins, for simplicity's sake this paper. first discusses only `NATURAL JOIN`, which allows two (or more)

tables to be linked based on columns that have the same name. To link the original table, `main`, with the new annotations, the `FROM` clause must be, instead of one table name, a table name followed by `NATURAL JOIN` followed by a second table name (see figure 12). This allows us to see what annotations are associated with each line from the corpus. When joining tables, rows which are not linked to a corresponding row in another table will not appear by default. This is because by default all joins are `INNER JOIN`s. For information on other types of `JOIN`s that see the PostgreSQL Documentation, available at «http://www.postgresql.org/docs/».

6. Corpus Analysis

6.1 Aggregate Statistics

Some of the simplest tasks involve retrieving a single piece of information about the corpus from the database—for example, the number of words total. By using some of PostgreSQL's aggregate functions—functions that combine data from multiple rows, various statistics about the data in the corpus can be obtained. The function `count()`, for example, counts the

```
susanne=# SELECT count(*) FROM main;
 count
--------
 156622
(1 row)
```

Figure 13. Using the aggregate function count() to obtain a count of the number of words in the table.

```
susanne=# SELECT count(DISTINCT to-
ken) FROM main;
 count
-------
 17345
(1 row)
```

Figure 14. Using the aggregate function count() to obtain a count of unique tokens in the table.

```
susanne=# SELECT avg(char_length(to-
ken)) FROM main;
        avg
--------------------
 4.5747276883196486
(1 row)
```

Figure 15. Combining aggregate and non-aggregate functions to find the average token length.

number of rows returned by a query. `count()` can take several arguments; the most common is `*`, which counts all rows, as seen in figure 13. It can also be given a specific column name, in which case it will count the number of rows in which that column's value is not `NULL`[6]. Finally, `DISTINCT column` will count the unique values in that column, as in figure 14. Functions can also be combined for even more powerful queries: `char_length()` returns the number of characters in a string, and the aggregate function `avg()` returns the average value of a set of rows. To get the average length of tokens in the main database table, simply use `SELECT avg(char_length(token)) FROM main;` as shown in figure 15.

```
susanne=# SELECT count(*) FROM main
WHERE token = 'Friday';
 count
-------
    13
(1 row)
```

Figure 16. Using WHERE to narrow down results.

The `WHERE` clause narrows down the rows returned to a specific subset of rows which cause the given expression to evaluate to true. The `WHERE` clause takes any expression returning true or false, which can be composed of multiple expressions joined together by `AND` and `OR`. Expressions can be negated with `NOT`. To get a count of only rows which have the token "Friday", for example, simply attach `WHERE token = 'Friday'` to the

[6] NULL indicates the non-existence of a value. Due to how SUSANNE corpus data was imported, there are no NULL rows in the database.

original query using `count()`, as seen in figure 16.

6.2 Word Lists

Included in the 5th release of the SUSANNE corpus is a file, `lexicon`, which contains "an alphabetized list of all pairs of wordform and word-tag that occur at least once in the Corpus" (Sampson). While it is convenient to be provided with such a list, it is also obvious that the data necessary to create it is already in the database.

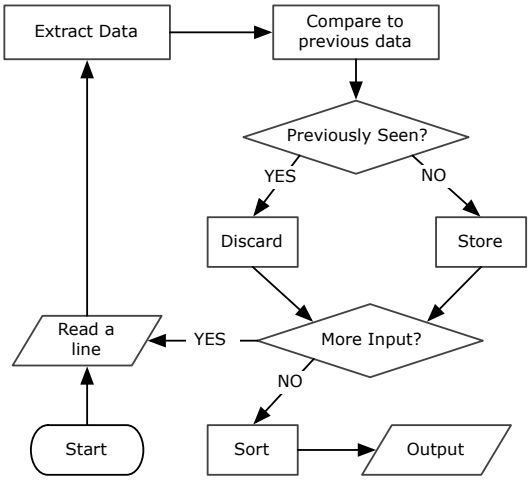

Figure 17.
A flowchart description of an iterative program for listing unique items read in from a file.

Unfortunately, creating a program to accomplish even this simple task is non-trivial for a non-programmer in a language such as Perl or Java. A high-level description of such a program might look something like figure 17, with many of the individual steps shown representing extremely difficult programming problems in and of themselves. Considerations include how to extract the data from the file, how to store data in memory, how to sort and display the data, and myriad other minutiae. Finding unique items in a list is a common enough task, but there may be no ready-made functionality, that can operate, for example, on pairs of items, or triplets. Even in Perl, where there are many, many pre-made modules available to download, if a function does not accept and return data in the formats needed, there is the additional overhead of reformatting data. If an algorithm does not provide exactly what is

```
                        susanne=# SELECT token, tag FROM main GROUP
                        BY token, tag ORDER BY token ASC;
lexicon                        token       |  tag
--------                ------------------+--------
%       NNUp22           %                | NNUp22
&       CC               &                | CC
&       NP1j32           &                | NP1j32
(       YPL              (                | YPL
(1)     MCb              (1)              | MCb
(1,1)   FOx              (1,1)            | FOx
```

Figure 18. A partial side-by-side comparison of the `lexicon` *file and the output of the SQL query.*

needed, a new one must be created or another adapted, which can still require in-depth programming knowledge. With a corpus in a relational database, however, data input and storage are already done, and output is handled by the client. The fully standardized and generic structure of relational database tables allows for a wide array of data manipulations which can be done with incredibly flexibility and efficiency, without traditional iterative coding.

The same task, done in our database, is astoundingly simple: using SELECT, the GROUP BY clause which allows us to merge rows with duplicate fields, and the ORDER BY clause which allows us to sort rows based on a certain column, the same information that is in `lexicon` can be retrieved with the SQL query seen in figure 18.

6.3 Frequency

Some of the advantages of relational databases and RDBMSes such as Postgres has already become apparent in even the simple task of creating a list of words.. A common and related task is frequency counting. Though there is no lack of software to find word frequencies in a textfile, a relational database has a few advantages

with respect to constraining results. When using `GROUP BY`, aggregate functions, instead of working on all the rows in the table, compute values on all the merged rows for each row returned. In order to get a frequency list of all the unique pairings of token and tags, simply add the count(*) aggregate function to the list of columns to return for the word list from section 5.1, as seen in figure 19. It is rather trivial to extend this to include a percent count of words in the corpus as a whole, but another concept is needed: subqueries.

```
susanne=# SELECT token, tag, count(*)
FROM main GROUP BY token, tag ORDER
BY count DESC LIMIT 5;
  token   | tag | count
----------+-----+-------
 the      | AT  |  8488
 +,       | YC  |  6877
 +.       | YF  |  6584
 of       | IO  |  4433
 and      | CC  |  3253
```

Figure 19. The aggregate function count() allows us to count the number of rows in each grouping.

```
susanne=# SELECT token, tag, count(*), 100. * count(*) / (SELECT
count(*) FROM main) as proportion FROM main GROUP BY token, tag OR-
DER BY count DESC LIMIT 10;
  token   | tag | count |      proportion
----------+-----+-------+--------------------
 the      | AT  |  8488 | 5.4194174509328191
 +,       | YC  |  6877 | 4.3908263206956877
 +.       | YF  |  6584 | 4.2037517079337513
 of       | IO  |  4433 | 2.8303814278964641
 and      | CC  |  3253 | 2.0769751375924200
 -        | YG  |  3068 | 1.9588563547905147
 a        | AT1 |  2776 | 1.7724202219356157
 in       | II  |  2404 | 1.5349056965177306
 <minbrk> | YB  |  1897 | 1.2111963836498065
 to       | TO  |  1763 | 1.1256400761068049
```

Figure 20. The 10 most common word+tag pairs in the corpus, with their associated count and the proportion of the corpus they represent.

```
susanne=# SELECT token AS word, tag
AS part_of_speech, 5 * 10 AS fifty
FROM main;
   word   | part_of_speech | fifty
----------+----------------+-------
 <minbrk> | YB             |    50
 The      | AT             |    50
 Fulton   | NP1s           |    50
 County   | NNL1cb         |    50
 Grand    | JJ             |    50
```

Figure 21. Renaming columns.

The formula `100. * count / total`[7] will find the percent of all words in the corpus for each token+tag combination. The total (the count of words in the corpus as a whole) is necessary for the calculation, but because of the `GROUP BY` clause the aggregate function `count()` operates on each grouping of rows. One way, of course, would be to issue two separate queries: one to find the total word count, and then a second to find the proportion of each word to the total, with the total en-

```
susanne=# CREATE TABLE twograms AS SELECT one.file, one.index AS
one, one.tag AS tag1, one.token AS token1, one.lemma AS lemma1,
two.index AS two, two.tag AS tag2, two.token AS token2, two.lemma AS
lemma2 FROM main AS one JOIN main AS two USING (file) WHERE
two.index = one.index + 1;
```

Figure 22a. Using self joins to generate a 2-gram table.

```
susanne=# CREATE TABLE threegrams AS SELECT twograms.*,index AS
three, tag AS tag3, token as token3, lemma AS lemma3 FROM twograms
JOIN main USING (file) WHERE index = two + 1;

susanne=# CREATE TABLE fourgrams AS SELECT threegrams.*,index AS
four, tag AS tag4, token as token4, lemma AS lemma4 FROM threegrams
JOIN main USING (file) WHERE index = three + 1;
```

Figure 22b. Using self joins to generate 3-gram and 4-gram tables.

---

[7] Note the period after `100`—PostgreSQL, like most computer programs, uses integer division for dividing two integers. By forcing 100 to be a floating point number, the percent column is calculated correctly using floating point math.

```
susanne=# SELECT token1, token2,
count(*) FROM twograms GROUP BY
token1, token2 ORDER BY count DESC
LIMIT 10;
  token1  |  token2  | count
----------+----------+-------
 +.       | <minbrk> |  1708
 of       | the      |  1395
 in       | the      |   774
 +,       | and      |   658
 -        | to       |   611
 +.       | The      |   593
 +,       | the      |   509
 +<rdquo> | +.       |   471
 to       | the      |   437
 +.       | He       |   364
(10 rows)
```

Figure 23a. Finding the 10 most frequent 2-grams.

tered by hand from the previous query. However, if the original `count()` query is surrounded in parentheses to indicate that it is a subquery, this information can be retrieved all in one query, as seen in figure 20. Note that PostgreSQL the new column has been renamed `proportion`. A new column name can be specified for any column by following the column specification with `AS new_name`, as shown in figure 21.

6.4 N-grams and Collocations

Mark Davies, in *Relational n-gram databases as a basis for unlimited annotation on large corpora,* proposes the annotation of textual databases with n-gram tables for fast and powerful searches. For the *Corpus del Español,* Davies uses the WordSmith program to generate n-gram tables for importation into

```
susanne=# SELECT tag1, tag2, count(*)
FROM twograms GROUP BY tag1, tag2 ORDER
BY count DESC LIMIT 10;
 tag1 | tag2 | count
------+------+-------
 II   | AT   |  2325
 AT   | NN1c |  2199
 YF   | YB   |  1815
 JJ   | NN1n |  1708
 AT   | NN1n |  1656
 AT   | JJ   |  1644
 JJ   | NN2  |  1523
 IO   | AT   |  1337
 JJ   | NN1c |  1249
 NN1n | IO   |  1223
(10 rows)
```

Figure 23b. Finding the 10 most common tag bigrams.

```
susanne=# SELECT token1, token2, token3, token4, count(*)
FROM fourgrams WHERE tag1 NOT LIKE 'Y%' AND tag2 NOT LIKE
'Y%' AND tag3 NOT LIKE 'Y%' AND tag4 NOT LIKE 'Y%' GROUP BY
token1, token2, token3, token4 ORDER BY count DESC LIMIT 5;
  token1  |  token2  |  token3  | token4 | count
----------+----------+----------+--------+-------
 of       | the      | United   | States |     9
 the      | radio    | emission | of     |     9
 that     | the      | United   | States |     8
 for      | the      | first    | time   |     8
 the      | end      | of       | the    |     8
```

Figure 23c. The 10 most frequent 4-grams, without punctuation (tags starting with Y).

```
susanne=# SELECT token1, token2, token3, count(*) FROM threegrams
WHERE token2 LIKE 'and' AND tag1 NOT LIKE 'Y%' AND tag2 NOT LIKE
'Y%' AND tag3 NOT LIKE 'Y%' GROUP BY token1, token2, token3 ORDER
BY count DESC LIMIT 5;
    token1     | token2 |    token3    | count
---------------+--------+--------------+-------
 <formul>      | and    | <formul>     |    19
 1             | and    | 2            |     5
 medical       | and    | dental       |     4
 physiological | and    | pathological |     4
 more          | and    | more         |     4
```

Figure 23d. The 5 most frequent 3-grams consisting of two non-punctuation tokens joined by 'and'.

```
susanne=# SELECT token1, token2, tag1, tag2, count(*) FROM
twograms WHERE tag1 = 'JJ' AND tag2 LIKE 'N%' GROUP BY
token1, token2, tag1, tag2 ORDER BY count DESC LIMIT 5;
    token1    |  token2  | tag1 | tag2  | count
--------------+----------+------+-------+-------
 United       | States   | JJ   | NN2   |    52
 New          | York     | JJ   | NP1t  |    24
 bronchial    | artery   | JJ   | NN1c  |    20
 pulmonary    | artery   | JJ   | NN1c  |    16
 New          | England  | JJ   | NP1c  |    15
```

Figure 23e. The 5 most frequent 2-grams consisting of an adjective followed by a noun.

the database (Davies *Advantages* 319). However once again this information is already in the database itself. By joining the main table to itself, a list of all n-grams, unique n-grams with frequency information, or other types of annotation can be generated. The SQL query in figure 22a will generate every 2-gram in the database, with associated annotations and unique column names. Although it is arguably most correct to generate this table anew for each query, since the underlying database does not change this information can be inserted into a new table by prefixing a `SELECT` query with `CREATE TABLE ... AS`, turning the result of the query into its own table. This table is also useful for generating 3-grams, 4-grams, etc. by joining it with the main table only once each additional time, which offers the additional advantage of being no slower to create each additional level than it was to create the previous. (figure 22b). The technique discussed in section 5.3 to retrieve frequency information can be used for n-grams, too, as seen in figures 23a and 23b. Text comparison operators such as

```
susanne=# SELECT token, tag,
count(*) FROM main GROUP BY
token, tag HAVING count(*) >
1000 ORDER BY count DESC;
  token   | tag  | count
----------+------+-------
 the      | AT   |  8488
 +,       | YC   |  6877
 +.       | YF   |  6584
 of       | IO   |  4433
 and      | CC   |  3253
 -        | YG   |  3068
 a        | AT1  |  2776
 in       | II   |  2404
 <minbrk> | YB   |  1897
 to       | TO   |  1763
 to       | IIt  |  1261
 was      | VBDZ |  1225
 is       | VBZ  |  1222
 <ldquo>  | YIL  |  1157
 +<rdquo> | YIR  |  1148
 that     | CST  |  1069
 The      | AT   |  1035
 for      | IF   |  1014
(18 rows)
```

Figure 24a. Using HAVING to narrow down results based on aggregated count.

LIKE and SIMILAR TO[8] can narrow down search results for certain tags, tokens or lemmas (figures 23c, 23d, 23e).

```
susanne=# SELECT token, tag, count(*), 100. * count(*) /
(SELECT count(*) FROM main) AS percent FROM main GROUP BY
token, tag HAVING 100. * count(*) / (SELECT count(*) FROM
main) > .5 ORDER BY count DESC;
   token   |  tag   | count |        percent
-----------+--------+-------+-----------------------
 the       | AT     |  8488 |     5.4194174509328191
 +,        | YC     |  6877 |     4.3908263206956877
 +.        | YF     |  6584 |     4.2037517079337513
 of        | IO     |  4433 |     2.8303814278964641
 and       | CC     |  3253 |     2.0769751375924200
 -         | YG     |  3068 |     1.9588563547905147
 a         | AT1    |  2776 |     1.7724202219356157
 in        | II     |  2404 |     1.5349056965177306
 <minbrk>  | YB     |  1897 |     1.2111963836498065
 to        | TO     |  1763 |     1.1256400761068049
 to        | IIt    |  1261 | 0.80512316277406750010
 was       | VBDZ   |  1225 | 0.78213788612072378082
 is        | VBZ    |  1222 | 0.78022244639961180422
 <ldquo>   | YIL    |  1157 | 0.73872125244218564442
 +<rdquo>  | YIR    |  1148 | 0.73297493327884971460
 that      | CST    |  1069 | 0.68253502062290099731
 The       | AT     |  1035 | 0.66082670378363192910
 for       | IF     |  1014 | 0.64741862573584809286
 +<hyphen> | YH     |   937 | 0.59825567289397402664
 he        | PPHS1m |   884 | 0.56441623782099577326
 with      | IW     |   818 | 0.52227656395653228793
 it        | PPH1   |   805 | 0.51397632516504705597
 his       | APPGm  |   804 | 0.51333784525800973043
 be        | VB0    |   788 | 0.50312216674541252187
(24 rows)
```

Figure 24b. Using HAVING to narrow down results based on percentage of the corpus.

[8] LIKE allows basic wildcard matching with '%' to indicate 0 or more unknown characters and '_' to indicate a single unknown character. SIMILAR TO allows for full POSIX regular expression support (PostgreSQL 144-146)

6.5 Thresholds

So far limiting the returned rows to only the top results has been done using `ORDER BY` and `LIMIT`. It is tempting to use `WHERE` to attempt to limit the results of a frequency analysis to entries with, say, more than 1000 occurrences, or entries that comprise more than .5% of the corpus. Since these use data from aggregate functions calculated on grouped rows, the `WHERE` clause cannot be used in the main query itself and instead `HAVING` must be used (PostgreSQL 82). Applying `HAVING` to the original word frequency count (section 5.3) can limit the results to more than 1000 occurrences (figure 24a). The proportional analysis from figure 19 can be limited to words comprising more than than .5% of the corpus by using `HAVING` and the > (greater than) operator (figure 24b).

6.6 Concordances with 5-grams

Another common task is concordances, or listings of words in context. Creating traditional concordances with a certain number of characters of context is not a

```
susanne=# SELECT file, three AS index, token1, token2, token3 AS
center, token4, token5 FROM fivegrams WHERE token3 LIKE 'fly';
 file | index | token1 |  token2   | center | token4  |  token5
------+-------+--------+-----------+--------+---------+----------
 N15  |    31 | -      | could     | fly    | was     | sent
 G11  |   261 | +;     | we        | fly    | through | the
 A13  |   804 | of     | pop       | fly    | hits    | <mdash>
 A13  |   823 | a      | sacrifice | fly    | +.      | <minbrk>
 G04  |  1756 | +.     | A         | fly    | would   | crawl
 G03  |  2313 | -      | to        | fly    | without | specific
(6 rows)
```

Figure 25a. Using 5-grams to view concordances for a literal token 'fly'.

task well-suited to relational databases (though it is not difficult, there are no advan-

```
susanne=# SELECT file, three AS index, token1, token2, token3 AS
center, token4, token5 FROM fivegrams WHERE lemma3 LIKE 'fly' AND
tag3 LIKE 'V%';
 file | index |  token1  |   token2   | center | token4  |  token5
------+-------+----------+------------+--------+---------+----------
 N15  |    31 | -        | could      | fly    | was     | sent
 N15  |    62 | the      | enemy      | flew   | or      | floated
 N04  |   184 | They     | all        | flew   | into    | action
 G11  |   261 | +;       | we         | fly    | through | the
 N15  |   705 | and      | instrument | flying | +,      | and
 A11  |  1101 | +<rdquo> | +,         | flew   | here    | late
 N04  |  1512 | young    | warrior    | flew   | over    | its
 A09  |  1943 | <bmajhd> | HOOD       | FLIES  | OVER    | HOUSE
 A09  |  1955 | the      | car        | flying | over    | the
 N15  |  2273 | the      | enemy      | flew   | into    | them
 N15  |  2309 | +<apos>s | propeller  | flew   | off     | in
 G03  |  2313 | -        | to         | fly    | without | specific
 N14  |  2487 | loose    | dirt       | flying | behind  | him
 N05  |  2517 | +tailed  | hawk       | flew   | in      | behind
(14 rows)
```

Figure 25b. Using 5-grams to view concordances for all instances of 'fly' as a verb using lemma and POS annotation.

tages offered by the relational model which would make it significantly better than other methods), but with n-gram tables (or a number of joins) it is trivial to generate concordances with a certain number of tokens on each side by simply searching the middle field of an n-gram table. Likely the minimally useful amount of context would be two words on each side, requiring a 5-gram table. However, more context can be added with additional joins; joining a 5-gram table to itself can give four words of context on each side, and with an arbitrary amount of joins one can have an arbitrary amount of context). Example concordances using 5-gram tables can be seen in figures 25a and 25b.

7. Conclusions

7.1 Disadvantages

Unfortunately, some tasks and types of annotation are unsuited for relational databases. Some tasks are by nature better suited to iterative programming techniques which, while they can be implemented and linked to the database using PostgreSQL's procedural language support (PostgreSQL 521), are not made easier by the use of a relational database and SQL. Tasks that involve complicated string processing such as stemming, tagging, etc., while not impossible, are simply unsuited for a language like SQL, which is primarily used for data access and manipulation, not iterative calculation and creation. Additionally, hierarchical data, including treebanks, are not easily represented by the relational model and hence is not easy to examine using SQL inside of an RDBMS (Davies *Advantages* 329).

7.2 Advantages

For searching and analyzing corpus statistics, relational databases as the basis for storing and analyzing corpora have a number of advantages over plain text storage formats such as XML and pre- or custom-made programs for analysis. A great deal of general-purpose functionality is available in modern RDBMSes that can be applied to many tasks in corpus linguistics. Aggregate functions and GROUP BY make tasks such as frequency analysis which would normally require an in-depth knowledge of programming and data structures extremely simple. Additionally, PostgreSQL has extensive support for mathematical functions, logical operators and string manipulation including full support for POSIX regular expressions (Post-

gresSQL). Since data storage and access is highly regular, relational databases, especially those with a modular table design, can be extremely fast for even complex queries, when combined with indexes (which are not discussed in this paper) as used by Davies's *Corpus del Español*(Davies *Advantages* 332-333). Comparative statistical analyses on texts can also be carried out based on arbitrary annotations such as genre, time period, author and more, without significant extra work as might be required by a programmer writing a general-purpose corpus analysis tool.

Additionally, the approach discussed in this paper has advantage even over other relational database techniques in use as described in Davies's "Relational N-Gram Databases as a Basis for Unlimited Annotation on Large Corpora" and "The advantage of using relational databases for large corpora: Speed, advanced queries, and unlimited annotation", since they allow for more finely-grained annotation and analysis and embrace the relational model for storage of corpus data at all levels.

7.3 Summary

The techniques in this paper can empower a languages researcher with the ability to perform advanced searches and statistical analyses on large corpora, without having to become a full-fledged programmer. While it cannot replace the need for custom programming for all possible tasks, the RDBMS can greatly lessen the burden on researchers in terms of both time and effort for many investigative tasks that do not fit squarely with available software. Storing annotated corpus data in an RDBMS also allows the researcher to avoid the "parse, process, repeat" trap—data is

instead parsed once and then stored in the database where it can be queried with new questions in an efficient manner. This eliminates wasted resources and improves program and programmer efficiency. The fixed structure of relational tables and the SQL language allow a researcher, to a certain point, to focus on what questions to ask and how to ask them, rather than how to parse, process, store and retrieve the answer. In turn, this can encourage further investigation and open-ended questions; expanding the possibility for discovery.

Works Cited

Davies, Mark. "The advantage of using relational databases for large corpora: Speed, advanced queries, and unlimited annotation." *International Journal of Corpus Linguistics* 10.3 (2005).

---. *Relational N-Gram Databases as a Basis for Unlimited Annotation on Large Corpora*. < citeseer.ist.psu.edu/560613.html>

Hammond, Michael. *Programming for Linguists: Perl for Language Researchers*. Malden: Blackwell, 2003.

Kennedey, Graeme. *An Introduction to Corpus Linguistics*. New York: Longman, 1998.

Lawler, John, and Helen Dry. *The Nature of Linguistic Data: Using Databases*. 1999. 2 June 2006. <http://www.sil.org/computing/routledge/simons/databases.html>

Mason, Oliver. *Programming for Corpus Linguistics: How to Do Text Analysis with Java*. Eds. Tony McEnery and Andrew Wilson. Edinburgh Textbooks in Empirical Linguistics. Edinburgh: Edinburgh UP, 2000.

McEnery, Tony, and Andrew Wilson. *Corpus Linguistics: An Introduction*. 2nd ed. Edinburgh Textbooks in Empirical Linguistics. Edinburgh: Edinburgh UP, 2001.

Nerbonne, John, ed. *Linguistic Databases*. Stanford: CSLI, 1998.

Oakes, Michael. *Statistics for Corpus Linguistics*. Edinburgh Textbooks in Empirical Linguistics. Edinburgh: Edinburgh UP, 1998.

The PostgreSQL Global Development Group. *PostgreSQL 8.1.0 Documentation*. 27 Feb 2006 <http://www.postgresql.org/files/documentation/pdf/8.1/postgresql-8.1-US.pdf>

Samson, Geoffrey. *The SUSANNE Corpus: Documentation*. Release 5. Aug 2000. University of Sussex, Brighton. 4 May 2006 <http://www.grsampson.net/Resources.html>.

The SUSANNE Corpus. Release 5. Aug 2000. University of Sussex, Brighton. 4 May 2006 <http://www.grsampson.net/Resources.html>.