Filtering SQL via the Principles of Pandemonium

John Lusth, Nenad Jukic
The University of Alabama
Tuscaloosa, AL 35487
lusth@cs.ua.edu, njukic@cs.ua.edu

Pandemonium -

1: the capital of Hell in Milton's *Paradise Lost*
2: the infernal regions
3: a wild uproar (when not capitalized)

...*Webster's 9th Collegiate Dictionary*

Abstract - In this paper, we re-introduce Pandemonium, an early specification for parallel processing through semiautonomous agents. The biggest advantage of the Pandemonium approach is its simplicity, which is achieved by dividing tasks among many computational units. Using Pandemonium as a metaphor, we design an interpreter for general-purpose filtering of text. To demonstrate the applicability of our design, we show how to partially parse and correct a sequence of badly ordered SQL commands. This example is not artificial; the badly ordered commands were generated by a commercial CASE tool for database development and the command set was too large for manual correction. A Pandemonium-style approach has many advantages over using full-blown parser generators and rule-based systems for such tasks.

1 Introduction

In 1958, Oliver Selfridge [1] proposed a forward-looking concept of using semiautonomous agents for problem solving. Individual agents would be tuned to certain aspects of the input and, upon recognizing such an aspect, an agent would proclaim that it had the answer (modulo its limited world view). The more confident the agent, the louder the agent's proclamation. Other agents, in turn, would hear these proclamations, sort them out, and proclaim their own opinion of the "true" answer. A master agent would then select the best answer from the highest level of agents. Selfridge called his system, *Pandemonium*, and his agents, *demons*, invoking meanings one and three of the definition at the beginning of this paper.

Although Pandemonium, as originally conceived, corresponds more closely to the neural nets of today, and indeed, still has its followers [2], the principles involved provide a useful metaphor for a large class non-neural net problems, such as data correction, data filtering, or data mining. We have a particular interest in data correction and filtering due to numerous requests to "write a program to fix up this [large] flawed data file". Our desire is to design a general filter based upon the principles outlined by Selfridge. We believe such a filter will be powerful yet easy and intuitive to customize.

Although we use the Pandemonium metaphor as a starting point for specifying a filter, we diverge from Selfridge in four important ways. Firstly, we make each demon fully autonomous; there is no hierarchy of demons and no master demon deciding which subdemons are to be believed. Secondly, since we are primarily concerned with collections of textual data, we employ a streaming approach for funneling data to the demons. Thirdly, we explicitly order demons such that higher ordered demons have first crack at the streaming data. Finally, we use the term *daemon* [3], rather than demon, lest readers think nefarious instructions can be discerned by reading this paper backwards.

In the remainder of this paper, we discuss a specific filtering problem in Section 2, describe the syntax and semantics of a general Pandemonium-influenced [4] interpreter in Section 3, present a Pandemonium program for correcting badly ordered SQL generated by a commercial CASE tool in Section 4, and discuss the advantages of the Pandemonium approach over other potential solutions in the last section.

2 The problem

The relational model of data [5] is based on a simple and uniform data structure: the relation. A relational database usually contains many relations, with tuples in relations that are related in various ways [6]. The referential integrity constraint is specified between two relations and it is used to maintain the consistency among tuples of the two relations. The best known language used to implement relational database model on commercial database management systems is called SQL [7], an acronym for *structured query language*. The following example written in SQL describes a referential integrity constraint between two relations:

```
CREATE TABLE A
   (Aname      CHAR(10),
    Aaddress   CHAR(20),
    PRIMARY KEY (Aname));
```

```
CREATE TABLE B
    (Bname       CHAR(10),
     Baddress    CHAR(20),
     Aname       CHAR(10),
     PRIMARY KEY (Bname),
     FOREIGN KEY (Aname)
        REFERENCES A(Aname));
```

Every **B** is associated with an **A**, and the referential integrity constraint "FOREIGN KEY (Aname) REFERENCES A(Aname))" makes sure that every **B** is associated with an existing **A**.

The CREATE TABLE statements are often generated automatically from the high level conceptual database models (such as ER diagrams) by a data dictionary-based computer-aided software engineering (CASE) tools. It is not unusual to generate hundreds of CREATE TABLE statements at once, by using a CASE tool command. We have identified a potential problem that arises during the SQL generation phase of some commercial CASE tools. The following example illustrates the problem. Suppose this code was generated by a CASE tool.

```
CREATE TABLE A
    (Aname       CHAR(10),
     Aaddress    CHAR(20),
     Bname       CHAR(10),
     PRIMARY KEY (Aname),
     FOREIGN KEY (Bname)
        REFERENCES B(Bname));

CREATE TABLE B
    (Bname       CHAR(10),
     Baddress    CHAR(20),
     PRIMARY KEY (Aname));
```

The first CREATE TABLE statement would be rejected by the SQL interpreter, because table **A** refers to a table **B** that has not been created yet. That makes this code unusable in this shape.

This problem can be solved by sorting table creation statements (using topological sort, for example), so that every CREATE TABLE statement which refers to another table, appears after the creation statement for the referred table. Sorting fails, though, if a circular reference appears. An alternate approach is to strip CREATE TABLE statements of referential integrity constraints, by filtering FOREIGN KEY constraints out. After the last table creation command is processed, ALTER TABLE commands can be used to add the referential integrity constraints back to the tables whose CREATE TABLE statements originally contained a FOREIGN KEY constraint, as illustrated by the following example:

```
CREATE TABLE A
    (Aname       CHAR(10),
     Aaddress    CHAR(20),
     Bname       CHAR(10),
     PRIMARY KEY (Aname));

CREATE TABLE B
    (Bname       CHAR(10),
     Baddress    CHAR(20),
     PRIMARY KEY (Bname));

ALTER TABLE A ADD FOREIGN KEY
    (Bname) REFERENCES B(Bname);
```

If the number of generated CREATE TABLE statements is large, a parsing mechanism is necessary on order to automate the process of filtering out and, later, adding back in FOREIGN KEY referential integrity constraints. We propose using Pandemonium to implement this alternate approach.

3 The solution

For correcting badly ordered SQL, we can imagine our daemons strung out along the input stream. As the CREATE TABLE commands pass by, the daemons will modify the commands and collect information for later use. The data that streams past the last daemon will the collection of CREATE TABLE commands sans their integrity constraints and will be written to the output data stream. A daemon which is triggered by the end of the input data stream performs an action which causes the ALTER TABLE commands to be written to output.

To implement such a solution, we have devised a general language for specifying daemons. In the remainder of the section, we discuss the syntax and semantics of daemons and give a specific Pandemonium program for performing the task at hand.

3.1 Daemons

A daemon is bipartite, being composed of a trigger clause and an action clause. Syntactically, the two clauses are separated by a colon and terminated with a semicolon. The grammar rule specifying a daemon is quite simple:

```
daemon  : trigger COLON action SEMICOLON
        ;
```

A daemon tests its trigger clause whenever a data item comes into view. Like a rule in a rulebase, if the trigger evaluates to true, the daemon executes its action clause. As to actions, a daemon may remove data, modify it, add to it, or modify a global or static local variable. Global variables correspond to Selfridge's demonic shrieks and wails, with the greater the magnitude of the variable, the louder the noise made by the demon. Static local variables correspond a daemon's memory.

We assume that the data streams by all the daemons in a sequential fashion. That is, the last daemon has a chance to act upon the i^{th} piece of data before the first daemon sees the i^{th+1} data item. This is an arbitrary constraint we place upon the model for simplicity's sake; the model itself can be naturally viewed as a pipelined processor of sequential data. Should a more parallel implementation be desired, the same techniques for managing a pipelined CPU [8] would apply here as well.

3.2 Triggers

Triggers look very similar to premises in rules; and are specified by the following grammar rules...

```
trigger : expr
        | expr -> expr  // implication
        ;
expr    : tokenList
        | arithmetic
        ;
```

where a *tokenList* is a list of tokens, possibly specified with regular expressions, and *arithmetic* is any integer-valued arithmetic expression. Triggers evaluate to true or false. We use a similar method as C in interpreting whether an expression is true or not: integer zero is considered false; all other integers are considered true.

The one exception to the above rule concerns *embedded implications*, a feature which distinguishes triggers from typical rule premises and is extremely useful for writing succinct and robust applications. Implications are used to ensure that the daemon is in a well-determined state, much the same way asserts are used in C programs. Examples of this use are...

```
'create' -> 'table'
'foreign' 'key' ->
GettingConstraint
colors > 16 ||
   limitedTextures -> textures < 8
commaExpected() -> ','
```

The first example states that if the data token in front of the daemon is 'create', the next token must be 'table'. The second example trigger states that, if the next two tokens are 'foreign' and 'key' in that order, then the *gettingConstraint* flag should be true. The third example states that if the there are more than 16 colors or the *limitedTextures* flag is true, there should be less than 8 textures. The final example states that if the *commaExpected()* function returns true, the data token should be a comma. If an embedded implication fails, an appropriate message is generated and the application terminates. Note the syntax of triggers (and actions) borrows heavily from C.

Tokens in a token list are enclosed with either single quotes or double quotes. Single quotes call for case-insensitive matching and double quotes call for case-sensitive matching. A plus sign is used to concatenate the strings and is useful for constructing a token with case-sensitive and case-insensitive portions.

3.3 Actions

An action is a comma separated list of commands. In general, actions either modify the data stream, or the local / global state, or both. Examples of commands are

```
remove
replace with '(' $* ')'
cleanUp()
++Level
pending = 0;
```

The first command specifies that the input tokens matched by the trigger should be removed from the data stream. The second command specifies that the matched tokens should removed, then reinserted into the data stream with enclosing parentheses. The term $* is shorthand for all matched tokens. The third command calls the cleanUp() procedure, while the last two commands modify variables in standard C fashion.

Like YACC [9], our system uses a shorthand notation for referring to matched tokens. The term $$i$ refers to the i^{th} matched token, while $* refers to the entire ordered list of matched tokens. For example, a daemon of the form

```
('hickory'|'filbert')
('bush'|'tree') :
   replace 'nut' $2;
```

replaces phrases such as *hickory tree* and *filbert bush* with *nut tree* and *nut bush*, respectively. The $2 in the action clause refers to the second token matched in the trigger.

3.4 A Pandemonium program

The entire list of daemons for correcting the badly ordered SQL is as follows...

```
'create' 'table' '*':
   Table = $3;

',' 'foreign' -> 'key' '(':
   remove,
   GettingConstraint = 1,
   GettingKeys = 1;

'foreign' -> 'key' '(':
   remove,
   GettingConstraint = 1,
   GettingKeys = 1,
   Comma = 0; // remove trailing commas

')' && GettingKeys :
   remove, GettingKeys = 0;

GettingKeys && '*':
   remove,
   Keys = addKey(Keys, $1);

GettingKeys && ',':
   remove;

'references' '*' ->
gettingConstraint:
    remove, ForeignTable = $2;

',' && GettingConstraint &&
!Comma:
    remove,
    GettingConstraint = 0,
    Comma = 1;
    addConstraint(Table, Keys,
        Foreign);

(')' || ',') && GettingConstraint:
    GettingConstraint = 0;

addConstraint(Table,Keys,Foreign);

EOF:
    generateAlterTableCommands();
```

Note that the program is quite short, even considering the addition of the two bookkeeping functions to handle multi-keys and to keep track of the elided integrity constraints. The program would be even shorter (and somewhat easier to understand) if not for the need to delete, in some cases, the comma preceding or trailing an integrity constraint. To handle the four different cases, though, only two additional demons are needed.

When a daemon matches on a token, the streaming of input data is temporarily halted at the location of the daemon. A useful analogy is that the daemon throws a temporary dam across the input stream. Only when the daemon's action is performed, or its trigger fails on subsequent tokens, does the daemon remove the dam. In the case of the first daemon in the above program, the daemon dams the stream upon seeing the token 'create'. If the next token is 'table', the daemon copies the value of the next token to the global variable *Table*, and then sequentially releases all three tokens to the downstream daemons. If the token after 'create' is not 'table', the trigger fails and the daemon sequentially releases the two tokens it has seen.

4 Conclusion

The biggest advantage of the Pandemonium approach is its simplicity. In many cases, when parsing, it is not necessary to understand the entire grammar of the input. Parser generators, such as YACC, which are often used to construct filters, do not lend themselves to the use of partial grammars. On the other hand, the Pandemonium approach offers a very quick and pragmatic mechanism for parsing when only a subset of the grammar is known. Compared to a rule-based system, our approach has the notational convenience of embedded implications and can be parallelized through pipelining.

In this paper, we have demonstrated the explicitness and effectiveness of the Pandemonium approach on a concrete problem of filtering SQL statements. We do not claim that the Pandemonium approach is the quickest one, performance wise. However, we believe that this approach is a very practical alternative when the amount of time and effort that is to be spent on creating a parser is limited. In our future work, we will address more complex problems with this approach, and consider the role that master daemons could play in improving it, without increasing the complexity of the Pandemonium programs.

5 References and notes

[1] Selfridge, O., "Pandemonium: A paradigm for learning", *Mechanisation of Thought Processes: I*, pp. 511-526, 1959.

[2] Smieja, F., "The Pandemonium System of Reflective Agents", *IEEE Transactions on Neural Networks*, **7 (1)**, January 1996.

[3] *daemon*, which is defined by Webster's as a "tutelary deity or spirit" does not have the evil connotation of demon and is used to denote an autonomous agent in the UNIX* operating system. We find it an appropriate replacement.

[4] Herein, we will use the term *Pandemonium* for brevity instead of the more accurate, but awkward, phrase *Pandemonium-influenced.*

[5] Codd, E., "A Relational Model for Large Shared Data Banks", *Communications of the ACM*, **13 (6)**, June 1970.

[6] Elmashri R. and B. S. Navathe, *Fundamentals Of Database Systems*, The Benjamin/Cummings Publishing Company, Inc., 1994.

[7] ANSI (1986), American National Standards Institute: The Database Language SQL, *Document ANSI X3.133*, 1986.

[8] Hwang, K., *Advanced Computer Architecture*, published by McGraw-Hill, pp. 265-321, 1993.

[9] Johnson, S., "Yet another compiler-compiler", *Bell Labs Technical Report* TM-75-1273-6, July 1975.