Web Log Session Analyzer: Integrating Parsing and Logic Programming Into a Data Mart Architecture

Michel C. Desmarais[1]
École Polytechnique de Montréal
Computer Engineering
C.P. 6079, succ. Centre-Ville, Montréal Québec, Canada, H3C 3A7
michel.desmarais@polymtl.ca

Abstract

Navigation and interaction patterns of Web users can be relatively complex, especially for sites with interactive applications that support user sessions and profiles. We describe such a case for an interactive virtual garment dressing room. The application is distributed over many web sites, supports personnalization and user profiles, and the notion of a multi-site user session. It has its own data logging system that generates approximately 5GB of complex data per month. The analysis of those logs requires more sophisticated processing than is typically done using a relational language. Even the use of procedural languages and DBMS can prove tedious and inefficient. We show an approach to the analysis of complex log data based on a stream processing architecture and the use of specialized languages, namely a grammatical parser and a logic programming module, that offers an efficient, flexible, and powerful solution.

1. Introduction

MVM Inc. developed a virtual dressing room that allows a user to create her own personalized virtual model and allows one to try on garments from a retailer over that model (see `www.mvm.com` and the example in Figure 1). The personal virtual model can be made to resemble as close as possible to the user's own body, namely her body measurements, but also including hair style, skin colour, etc. A Web questionnaire of about 12 items allows the user to tailor her model to her will (see Figure 2).

This application is sold to garment retailers such as Lands'End and Sears Inc. However, personal attributes of the user is kept by a central server managed by MVM, such that the user can register and create his/her model once and retrieve it for every retailer once logged in.

Figure 1. A virtual model.

MVM developed a data collection system that logs relevant usage information. Here are a few examples of the questions we want the system to be able to answer:

- How many users visited Lands'End during the month and how many garments did they try on?
- What is the breakdown of the number of MVM network retailer sites visited by registered users this month?
- What proportion of users stop the registration process and at which step is the highest dropout rate? How many times do they modify their model during creation? How many times an unfulfilled field error is reported per model creation process?
- What garment is most often tried on with a specific garment? What is the ratio of garments tried on over

[1]Formely at MVM Inc., 80 Queen St Suite 502 Montreal, Quebec, Canada H3C 2N5.

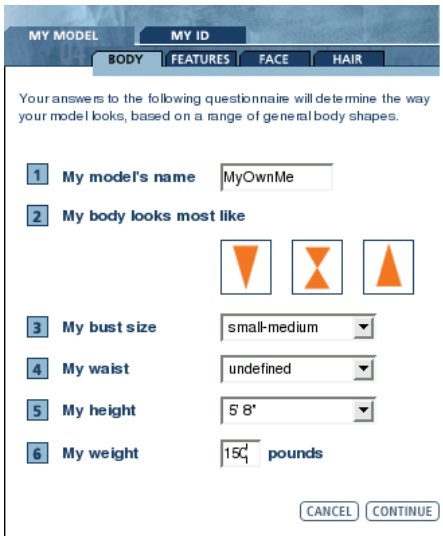

Figure 2. Virtual model creation questionnaire.

garments bought? In what coulour is this garment most often bought?

These questions are critical to many departments, namely the marketing people, the user experience group, and even the accounting department because some billing information relies on the data collection system.

The questions impose a number of challenges to the data collection and analysis system:

Multi-site: Sessions can start on MVM's portal server (where the questionnaire is hosted) and continue over one to many retailer sites (servers are sometimes hosted by the client site themselves).

Large number of events: To accommodate the diverse amount and sometimes changing information that needs to be collected, many types of events need to be collected. Different versions also need to coexist.

Representational power and flexibility: In order to recreate the full user experience, some of the information stored requires flexible representations such as embedded lists (eg. the list of garments currently worn by the model, or the list of answers provided, both of which are not fixed numbers). Moreover, Because the application evolves and the data collection requirements are not fixed, the data collection scheme needs to accommodate many changes over time. It has to be flexible.

```
Time  Event   Site User  Sess. Seq. Data
stamp Type    ID   ID    ID    #
_____________________________________________________________
[ts]  Start   vmis GUEST [sid] 0  {|}
[ts]  LocSes  vmis GUEST [sid] 1  {|sid=MVM022154533728|}
[ts]  Out     vmis GUEST [sid] 2  {|}
[ts]  Star    vmis GEN   [sid] 3  {|s=F|}
[ts]  GetVM   vmis GEN   [sid] 4  {|p=F|}
[ts]  Body    vmis GEN   [sid] 5  {|v=02HDL|}
[ts]  Out     vmis GEN   [sid] 6  {|}
[ts]  CreaVM  vmis GUEST [sid] 7  {|p=F|}
[ts]  Quest   vmis GUEST [sid] 8  {|d=F|qs=A|}
[ts]  Quest   vmis GUEST [sid] 9  {|d=F|qs=A|}
[ts]  Quest   vmis GUEST [sid] 10 {|d=F|qs=A|}
[ts]  Quest   vmis GUEST [sid] 11 {|d=F|qs=A|}
[ts]  ModMV   vmis GUEST [sid] 12 {|a={LipShape=2902,
                                   DOMAIN=F,
                                   QUESTION_SET=A,
                                   [...]
                                   HairStyle_F_40=2119}|}
[ts]  GetVM   vmis GUEST [sid] 13 {|p=F|}
[ts]  Body    vmis GUEST [sid] 14 {|v=06HDL|}
[ts]  Quest   vmis GUEST [sid] 15 {|d=AU|qs=UP|}
[ts]  LocSes  vmis GUEST [sid] 16 {|sid=MVM022154533728|}
[ts]  Quest   vmis GUEST [sid] 17 {|d=F|qs=A|}
[ts]  ModMV   vmis GUEST [sid] 18 {|a={FudgeFactor=4902,
                                   [...]
                                   HairStyle_F_40=2119}|}
[ts]  ModMV   vmis GUEST [sid] 19 {[...]}
[...]
[ts]  Fai     vmis GUEST [sid] 47 {|fd=le nom moon
                                   [...]
                                   |e=SignInCreate|}
[ts] Fai     vmis GUEST   [sid] 48 {|fd=User exists [...]
                                   [...]
                                   |e=SignInCreate|}
[ts] End     vmis GUEST   [sid] 49 {|}
```

Figure 3. Raw data example.

2. Log Data

Figure 3 provides a sample excerpt from a user session. Although it has a fixed number of fields (7), the last field (Data) allows the logging of structured data. The fields are describe below:

Time Stamp: Time in milliseconds

Event Type: The type of the event. About 50 different event types are defined.

Site ID: The ID of the site where the event is recorded.

User ID: The user ID of registered users (GUEST for unregistered users)

Session ID: A unique session ID that is global across sites.

Sequence ID: The sequence number of the event for that unique session.

Data: A structured data field that extends the event's specific data to be collected. The data field can contain any number of sub-fields. Some events contain up to 10 fields and some fields can contain list structured data such as question answers or garment lists.

3. Data Mart Challenges

The data collection system can easily answer the relevant questions if the data is correctly aggregated and detailed accordingly. This is the essence of a data mart: Aggregating data into specialized repositories [1]. For example, many questions require the data to be aggregated on a global session basis with all relevant information stored in the corresponding fields and in a proper relational scheme. In our data mart, the *session* table contains 41 fields. Another table is the *user* table which contains about the same number of fields for answering questions in relation to the registered user aggregation perspective.

However, processing the raw data of Figure 3 directly into such tables poses three challenges:

1. The algorithms are non trivial. For example, classifying a user session into about 10 types (eg. *new user complete*, *new user incomplete at step 2*, *registered failed,* , etc.), involves approximately 30 rules.

2. Performance is important. Monthly analysis involves about 5GB of data and 50M user sessions. Data access and memory usage must be optimized.

3. Flexibility is required. In any new application, requirements are bound to change frequently and new questions will appear. The system must adapt to frequent changes and be flexible enough to accommodate them.

These challenges can be difficult to meet with standard approaches. Much like telecommunication and credit card transaction data, Web logs can be considered as stream data. Storing this type of data in a database for further processing is known to cause difficulties. For example, Dobra et al. [2] note that:

> "In most such applications, the data stream is actually accumulated and archived in the DBMS of a (perhaps, off-site) data warehouse, often making access to the archived data prohibitively expensive".

Some of the difficulties stem the use of a standard procedural language, such as PL/SQL, that involves a relatively laborious implementation task for non trivial algorithms. Also, much care must be taken to organize the data and the algorithms for performance. Finally, in a context of frequent changes, program code rewriting and schema data reengineering can prove tedious and slow.

4. Stream processing architecture

The approach that we adopted is to use a stream processing architecture with specialized programming languages for the different phases of processing. Stream processing can be thought of as a sequence of programs joined through what is known as *pipes* in a UNIX environment: The output of a program is directly fed into the input of the following program. Our approach takes advantage of stream processing to aggregate data efficiently before storing the aggregated results into a data mart, thereby avoiding the problems of accumulating of raw data into a DBMS as noted by [2]. The stream architecture is described in Figure 4.

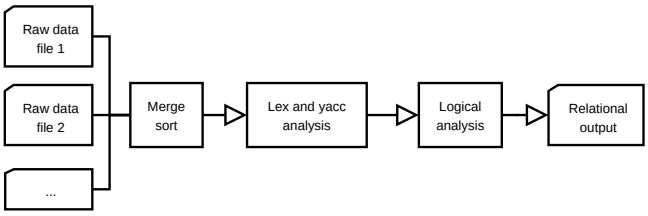

Figure 4. Data flux.

The raw input files are collected from the server sites. Records (one per line) are sorted by the global session ID key. A merge sort is performed, using the standard `sort` utility, as the first step of the stream processing sequence. The merge sort has the advantage of having an order of growth $O(n)$, linear with the number of records n, and a constant memory size.

The second step is a lexical and grammatical analysis program that parses the input stream into a sequence of tree structured records (in fact, a doubly linked list in the syntax of Prolog lists), one for each record (line). This program is written with a combination of the standard `lex` and `yacc` (`bison` in fact) parsing utilities available in UNIX. `lex` generates a C program that represents a finite state automaton and corresponds to a set of regular expressions defined with `lex`. `yacc` generates an efficient C implementation for a grammar description of an LALR context-free grammar. The result of the parsing process is a transformation of the raw data into Prolog predicates, which are equivalent to relational tuples.

The third step is a Prolog program that sequentially reads the predicates (tuples). There is one predicate per record (which corresponds to a raw log file input line). Prolog has the advantage of combining a high level logic programming language with database relational operations that makes the implementation of the algorithms very efficient and condensed. Any programmer who has programmed both Prolog and PL/SQL can appreciate the elegance of Prolog for quickly implementing logic rules over relational data.

As an example of the convenience of Prolog to implement logic rules, consider Figure 5 which contains a tree structured list of events that is used for classifying user sessions. It is relatively straightforward to write a small

```
[seq,
 [event 'Start', [], droupout0],
 [seq,
  [event, 'CreaVM', [], droupout1],
  [event, 'ModVM', []], dropout2],
  [or,
   [event, 'CreaAcc', [], complete1],
   [seq,
    [event, 'Fai', [e='SignInCreate'], error1],
    [event, 'End', [], dropout3]]],
 [seq,  ...], ...]
```

Figure 5. A Prolog decision tree session classifier (partial).

Prolog routine that traverses this tree to verify the ordered presence of the events, including tests of some event properties in the Data section, and returns the corresponding session type. For example, a session will be classified as `droupout3` if the session contains the events (in that order) `Start`, `CreaVM`, `ModVM`, *not* `CreaAcc`, `Fai`, and `End`, and if the event `Fai` has the property-value condition `e=SignInCreate`. Such succint and flexible representation of decision tree-like structures make the analysis of data highly flexible and simple to implement.

The output of the Prolog program is a flat file that represents tuples. It can be readily exported to a DBMS and provide the necessary data mart information. There is one tuple for the user session aggregation analysis, and another one for the user profile aggregation.

5. Performance

The complete analysis of a 5Mb month data set can be done on a standard Pentium IV workstation with 512Mb of RAM memory in about 2 hours. As we discuss below, we were able to compare this performance with a standard PL/SQL with an Oracle 8i database and estimated that the stream implementation was approximately 5 times faster than the database implementation. We did not investigate the reasons, but we can presume that it was due to the sequential disk access that the stream approach entails, and potentially also the high speed of the C program for parsing the raw data (using `lex` and `yacc` for C code generation). The use of constant memory of the stream architecture may also be a factor.

6. Conclusion

However, the most interesting quality of the approach is not so much in its performance, but in its facility for rapid implementation and the resulting flexibility. That quality was the decisive factor in deciding to use this approach instead of the traditional DBMS data mart architecture.

In fact, the stream approach was first used as a prototype before the implementation of a standard data mart architecture. It was not intended to become the final approach. Its design and implementation took about 5 months for a single programmer. After this first prototype, a prototype of standard data mart implementation was started by a DBMS specialist. After about three months, about only half of the modules were implemented, even though all the design had been done for the first prototype. Meanwhile, a full team of 4 DBMS specialists started the design and implementation of what was envisioned as the full data mart. After four months, the DBMS data mart project was canned because it was deemed too expensive. The stream prototype, that meanwhile provided marketing and the other departments with the analyzed data, ended up as the final operational data mart!

However, the stream approach is not without drawbacks, the most important of which is undoubtedly the difficulty of finding people who could support the Prolog and the `lex` and `yacc` code. Not only is it difficult to find experts in both fields, but it also has been difficult to bring the two cultures together. The company did face some resistance in introducing other Artificial Intelligence technologies in its products, such as a recommender system for garment size fit and a Prolog engine for implementing a rule based system for the garment mix and match process, but the strongest resistance was found in introducing Prolog and formal parsing techniques for data processing. There appears to be a strong divide between the world of AI and data warehousing!

Nevertheless, the interesting finding here is that we stepped aside the mainstream data mart/warehouse industrial approach, which relies mostly on database processing, and introduced a stream processing architecture that allows the integration of heterogeneous and specialized technologies such as parsers and logic programming to better tackle specific subproblems. The experience showed that it can be highly efficient both in processing and in implementation time and effort.

References

[1] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *SIGMOD Records*, 26(1), 1997.

[2] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In M. Franklin, B. Moon, and A. Ailamaki, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, June 3–6, 2002, Madison, WI, USA*, pages 61–72, New York, NY 10036, USA, 2002. ACM Press.