1981

The Flat File Database Generator Ffg

Douglas E. Comer
Purdue University, comer@cs.purdue.edu

Report Number:
81-379

The Flat File Database Generator Ffg

Douglas Comer

Computer Science Department
Purdue University
West Lafayette, IN 47907

September 1981

ABSTRACT

A flat file is the simplest possible database. It consists of a single, unformatted text file in which each line corresponds to a record. k-1 occurrences of a separator character divide each record into k variable length fields; the separator character does not otherwise appear in the file. Unlike most database systems, the flat file system is not a single, large program. Instead, it consists of a set of small, independent programs, called primitives, that each perform one basic operation. The user composes a subset of the primitives by directing the output of one to the input of the next in order to perform complex retrieval or update operations. Because they are independent, primitives are easily modified or replaced, and one can add programs to the set of primitives. Both the selection of primitives as well as their implementation are discussed.

CSD-TR-379

1. Introduction

A flat file is the simplest possible database. It consists of a single text file, F, containing zero or more lines, where each line is thought of as a record. Records are further divided into k fields, $f_1, f_2, \ldots, f_k$ by k-1 occurrences of a distinguished separator character, S. Although k is fixed over all records, the length of individual fields is not. The flat file generator, ffg, is a database system that provides facilities to create, query, and modify a flat file database.

Unlike most commercial database systems that consist of one or two large programs to process queries and modify the stored data (see DATE75, ULLM80), ffg consists of many small programs, called primitives, that each perform one basic operation. A user composes a subset of the primitives by directing the output of one to the input of the next in order to perform a complex task. Because the primitives each perform one basic operation, selecting an appropriate combination is straightforward and natural. Because primitives are independent programs, they can be modified or replaced, and one can add programs to the set. The ease of extension and modification is important in achieving flexibility because it allows one to tailor ffg for each application.

Constructing systems as a set of primitives is not new. Kernighan and Plauger [KEPL76] describe primitives for program and text manipulation; Hanson [HANS79] extends them. Borden et. al. [BOGS79] describe an electronic mail preparation and reading system implemented in primitives.

Interestingly enough, most of the experiments with primitives have their roots in the UNIX operating system [RITH78,KEMA79]. Unlike most systems which encourage one to build large integrated programs, UNIX encourages one to build independent programs and connect them together. It provides a simple and efficient mechanism for passing data between running programs. It

treats I/O to files, devices (like terminals), and other programs uniformly, so one does not need to know how a program will be used when writing it. UNIX contains sets of primitives for text processing and language development.

The next section of this paper describes pertinent parts of the UNIX environment in more detail, and gives the reader some appreciation of how UNIX influenced the flat file design. The following sections describe the flat file primitives, give an example of using flat files, and discuss their implementation. The paper concludes by discussing the merits of systems constructed from primitives.

2. The UNIX Environment

UNIX contains a large set of independent primitives, called *commands*, and a mechanism for composing them, called a *shell*. The UNIX shell [BOUR78] is a simple programming language that can be used interactively (as a command interpreter would be), or invoked to read input from a file (as a programming language interpreter would be). Shell programs are called *shell scripts*, or just *scripts*. If one tries to execute a file that contains a shell script, the system automatically invokes the shell to interpret it. Thus, a shell script functions just like a compiled program. In fact, some of the system commands are implemented as shell scripts.

The shell has facilities to invoke commands, direct the output of one command to the input of the next, or direct the input (output) of a command to a file or I/O device. Control statements (e.g., *while*, *for*, *if*, etc.) provide indefinite iteration, conditional execution, and definite iteration much like conventional programming languages. Unlike conventional languages, the shell only supports one data type — that of character string. It relies on commands to evaluate numeric expressions, test file status and accessibility, and handle complex computations.[1] One learns quickly that the art of constructing shell programs lies in

composing and invoking commands, not in using the shell exactly as one would use Algol 60 or Pascal.

UNIX includes a mechanism for composing primitives called a *pipe*. Pipes, denoted by "|" in shell scripts, connect the output from one program to the input of another. One writes

```
a | b
```

to invoke programs *a* and *b* with the output from *a* connected to the input of *b*. The line

```
a arg1 | b arg2 arg3 | c
```

specifies a pipeline connecting the output of program *a* to the input of program *b* and connecting the output of program *b* to the input of program *c*. Program *a* has one argument (arg1), program *b* has two (arg2 and arg3), while program *c* has none. *a*, *b*, and *c*, could be the names of shell scripts or compiled programs.

UNIX contributed to the construction of ffg in several other ways:

1. All system services are available at the command level. One can create files, change protection modes, trap exceptions, and perform other tasks directly from the shell in UNIX. On many systems such tasks require special programs, often in assembler language.
2. UNIX provides a rich set of text manipulation primitives that ffg uses extensively.
3. UNIX is a late binding system. There is little distinction between data and program; one can write a file and invoke the shell to run it as a script.

The UNIX environment is not perfect, but it contributed nicely to the exper-

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.