AT&T Technical Journal
Vol. 64, No. 9, November 1985
Printed in U.S.A.

Datastream—A Language for Large Files

By D. SWARTWOUT*

(Manuscript received April 9, 1984)

Datastream is a simple language designed for writing programs that perform computations on large data files in the *UNIX*™ operating system. Datastream handles larger files and a wider variety of computations than most *UNIX* database management systems, but it provides no explicit support for updates. Its features and performance characteristics are particularly good for working with infrequently updated, mostly statistical data. This paper describes the Datastream language, outlines its evolution, and summarizes users' experience with a prototype implementation.

I. INTRODUCTION AND CURRENT STATUS

Datastream is a simple language designed for writing programs that perform computations on large data files in the *UNIX* operating system. It features simple control structures, built-in input/output, and arithmetic expressions in the style of the C programming language.[1] Datastream users can concentrate on defining computations without spending much time on other aspects of programming. The system has been implemented on the VAX 11/780[†] computer system under AT&T *UNIX* System V and University of California, Berkeley *UNIX* 4.1 and 4.2 Berkeley System Distribution (BSD), and on the AT&T 3B20 Simplex computer. It is written in C and has roughly 11,000 lines of source code.

Datastream processes larger files than most *UNIX* database management systems that run on comparable hardware. The largest data

* AT&T Information Systems.
† Trademark of Digital Equipment Corporation.

file that Datastream has handled contained roughly 70M bytes of data. Several applications use files of 40 to 50M bytes. Sets of data files can be interconnected and used as a database; the largest database totals 140M bytes. Datastream is a tool for working with databases, but it is not a database *management* system: it provides no explicit support for updates. Datastream programs can build new data files from old ones, however. These characteristics make the system particularly suitable for analysts and others whose files are large and infrequently updated.

Section II of this paper gives a brief history of Datastream's evolution, and Section III summarizes the resulting architecture. Section IV describes the main features of the language, Section V discusses experience with it, and Section VI draws some conclusions.

II. HISTORY AND MOTIVES

Datastream has passed through three distinct stages since work began on it in late 1980. It was originally intended to manage distributed analytical databases. Several interesting, large-scale, single-site applications appeared before any distributed ones, so our attention shifted to large nondistributed databases. At that time "large-scale" meant roughly 2 to 20M bytes, but initial successes with databases in this range encouraged users to try still larger files. Changes made to accommodate these "very-large-scale" files removed the last similarities to ordinary database management systems; Datastream had evolved into a special-purpose programming language. The rest of this section discusses these changes in more detail.

2.1 Distributed database system

We set out originally to develop a system that would maintain distributed analytical databases efficiently. The *UNIX* operating system was expected to be used at each site, no database-oriented changes would be made to the operating system kernel or file system, and no special features were expected from the network connecting the sites. At the database designer's discretion, a given data file could be present at two or more sites, but no site was expected to have all the data. Queries were to be decomposable so that several sites could process parts of a query in parallel whenever possible. Distributing partial queries to multiple processors can be a complex process, and we did not wish to spend much time developing software to solve it. Datastream's query language was designed to permit easy decomposition of queries.

Concentrating on analytical databases had several important consequences. First, analytical databases change infrequently. New data may arrive quarterly or annually or even never, and updates are usually appended to the database; they almost never destroy existing data.

Second, analytical databases are large. It took several years of experience to fully understand this point, but it was clear from the beginning that we would need hardware of at least the VAX computer class. Finally, not all analysts should be considered "naive users." Many are experienced, if not expert, programmers. Some have written tens of thousands of lines of Fortran, and virtually all of them are comfortable with such things as operator precedence, assignment, and the difference between integer and floating-point arithmetic.

2.2 Statistical database system

An earlier paper discusses adjustments that were made to accommodate large single-site databases.[2] Those adjustments are summarized here. The general goal was to make Datastream a tool that would extract interesting subsets of databases for further analysis by a system such as S.[3,4] The original prototype of Datastream performed rudimentary computations such as conversion between English and metric units, but it could not do aggregate computations such as sums and averages. Datastream's computational facilities needed to be much stronger to make it useful as an analytical tool. We added three main features to strengthen the query language: conditional expressions, `collect` statements, and `compress` statements, all described in Section IV. We did not add specific aggregate functions such as sum, count, or average. Analysts can write a wide variety of aggregate computations with `collect` and `compress` statements, including, but by no means limited to, the standard aggregates.

2.3 Special-purpose programming language

An epitaph for phase two in Datastream's evolution might read, "Give them a megabyte, and they take a hundred." Initial success with databases up to about 20M bytes encouraged users to try still larger data sets. The largest in use at this writing has about 70M bytes. Datastream could not handle databases of this scale in phase two because building new databases required too much computing and file handling. "Raw" data had to be preprocessed to make it intelligible to the query-processing software. The results were stored in specially formatted database files. These "cooked" files could be processed efficiently, but they had several disadvantages. They were expensive to create. The preprocessing program sorted large amounts of data; its running time was proportional to $n \log n$, where n is the number of records in the database. It also created large temporary files. The configurations and load patterns of the machines running Datastream were such that these costs became excessive as the size of the data files approached 20M bytes. Cooked files could be used only by Datastream. Numeric fields were stored in binary (not ASCII) form, character strings were terminated by null (that is, zero) bytes, and records had Datastream-specific headers with assorted binary fields. Ordinary commands such as `grep` and `sort` were useless for handling cooked files, and users who tried to display them risked putting their terminals into a hopelessly confused state.

We solved these problems by replacing the lowest-level file-handling routines in the query-processing software with similar routines that could handle ordinary files. By "ordinary" we mean that records correspond to lines, data fields are delimited by some fixed character such as the tab, and numeric fields are in a printable form. "Ordinary" does not mean "any." It is hard to get Datastream to produce useful results from a file that contains more than one kind of record, for example.

Datastream originally constructed relationships among database objects in the cooking process, so we had to find a suitable way to set up connections between uncooked files. This was the most difficult part of the conversion. We chose a descendant of the original relationship mechanism that sacrifices some generality but works with ordinary files. This approach to connections is described in Section IV.

As a result of the switch to ordinary files, Datastream ceased to be a database management system. We regard it as a special-purpose language tailored to writing programs that perform computations on large, analytical data files. Datastream also can be used to build new analytical files from old ones. For historical reasons, Datastream programs are usually called "queries."

III. ARCHITECTURE

3.1 Software

Datastream is implemented as three major modules: the query compiler, the execution supervisor, and the statement processor. Each is discussed in a subsection below. The command

`stream` *qname*

starts query processing. *Qname* is a file that contains the text of the query. Usually a query file is created by the user with a text editor, but queries have been written by C programs, `awk`, `m4`, and even other queries. `stream` writes its results on the standard output, so they can be displayed on the user's terminal, saved in a file, or piped to another program for further processing. Several variations on this theme can be requested by additional command-line arguments.

3.1.1 Query compiler

The Datastream query compiler translates queries into an intermediate code. This code is kept in a file and used again without recompilation the next time *qname* is run, unless the source text has changed. The intermediate code consists of sections that correspond closely with statements in the source text. They are called "compiled statements" from here on. Each compiled statement describes input and output formats, temporary variables, constants, and the processing required to transform input to output. Most of the latter consists of reverse-Polish expressions that define computations and assign results to temporary variables or fields in output records. The intermediate code also contains a small amount of information about control flow.

The code for the query compiler includes a scanner generated by `lex`,[5] a parser generated by `yacc`,[5] and assorted support routines. It has been rewritten extensively twice, with numerous minor revisions. It is a one-pass compiler with respect to statements (that is, the compilation of a statement is independent of statements that follow it), but it makes two passes over each statement. The first pass parses the statement and generates code for expressions. The second pass formats output records, identifies temporary variables, and writes the remainder of the intermediate code. Most of these characteristics resulted from evolutionary pressure, not explicit design. The query compiler is the largest and most complex of the modules. At this writing it requires about 100K bytes of main memory at run time. In some unusual cases the statement processor can grow larger than this via dynamic storage allocation.

3.1.2 Execution supervisor

When the execution supervisor starts working, it searches for an up-to-date compiled query file and runs the query compiler to make one, if necessary. Then the execution supervisor starts one copy of the statement processor for each compiled statement in the query. It breaks the compiled query into statements, passes each compiled statement to the corresponding process, creates pipes through which the statement processors communicate, and waits for them to terminate. It checks for abnormal terminations and writes an error message when something goes wrong.

Figure 1 illustrates this process. Boxes represent *UNIX* system processes and ovals represent data files. Arrows trace the movements of data and queries through the system. The source text of a query is represented in the upper left-hand corner. Each statement begins with one of the key words `get`, `collect`, or `compress`, which represent Datastream's three control structures. In simplified form, they work as follows. The `get` statement tells its statement processor to get selected lines from a data file and transmit them through a pipe to the next statement processor. The `compress` statement reads lines from

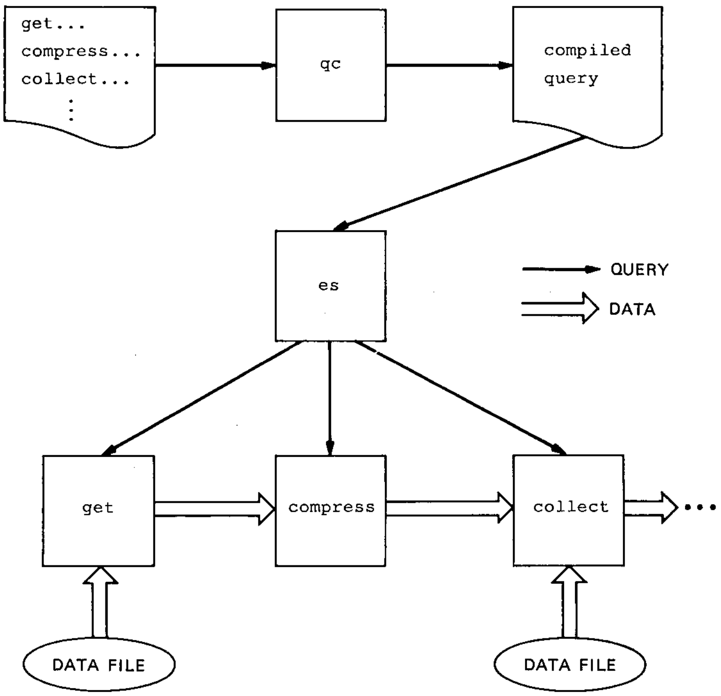

Fig. 1—Datastream architecture.

its inbound pipe and compresses them into aggregate values (for example, averages) that are written into its outbound pipe. `compress` statements do not read from the database. Like the `get` statement, the `collect` statement gets lines from a data file, but instead of writing them immediately into its outbound pipe, it computes aggregate values from them, attaches these values to the inbound lines, and writes the augmented lines into its outbound pipe. Known collectively as the *mainstream*, these pipes are the data paths that connect the `get`, `compress`, and `collect` processes in Fig. 1.

Most *UNIX* systems impose a limit on the number of concurrent processes a user may have. When a query has too many statements to process at once, the execution supervisor divides the query into blocks of statements and then processes the query a block at a time. The Datastream user can control the number of statements per block, but in practice the default value of ten statements per block is rarely overridden.

Although it is not discussed further in this paper, it is possible to specify postprocessing of query output by including a *UNIX* system shell script[6] in the source text of a query. When this feature is used, the execution supervisor has to start a shell, give it the script, and connect it with a pipe to the appropriate statement processor. The execution supervisor's duties are "just" routine management of detail, but its code totals six hundred lines of C.

3.1.3 Statement processor

A statement processor reads from two sources of input and writes two kinds of output. Its inputs are an inbound mainstream pipe and data file. The first statement of a query has no inbound mainstream to read, of course. No statement processor reads more than one data file, and `compress` statements read only the mainstream. Each statement in a query except the last one writes into an outbound mainstream pipe that becomes the inbound mainstream for the next statement processor. Some statement processors write on the standard output, specifically, the last statement in each query. Sometimes intermediate statements print with standard output directed into files. Anything that can be transmitted through the mainstream can be printed on standard output, and vice versa.

3.2 Files

A Datastream data file is a sequence of lines. Each line is subdivided into *fields*, which can be one of four *types*: integer, real, character string, or pointer. Fields of all types appear as variable-length ASCII strings in the data file, and they are separated by a single-character *delimiter*. The delimiter is usually a tab, but it can change from file to file. When a numeric field is used in a query statement, the ASCII text from the data file is converted to a suitable binary form. On the hardware that currently supports Datastream, integers and pointers are 32 bits long, and reals are 64 bits.

Each line in a data file represents an *entity*, and all the lines in a given data file represent entities of some entity type.[7] *Entity types* usually correspond to interesting persons, places, or things in the external world, for example, telephone calls or telephone customers. The user assigns a name to every entity type. In this paper, names for entity types will be ordinary nouns with the first letter capitalized to emphasize status as a component of the database: Call and Customer, for example. Datastream knows nothing about an entity except the values of the fields in the line that represents it. Lines in a given data file all have the same fields in the same order. If a line has too few fields, the missing fields are assumed to have appropriate null values; extra fields, if any, are ignored.

Data files have to be described in a configuration file before Datastream can use them. The *configuration file* contains a description of each entity type in the database, including the name of the entity type, the name of its data file, and a list of its field names and types.

The query compiler reads the configuration file to get information about names and types. The execution supervisor extracts data file names and passes them to statement processors that need to know which data file to open.

For example, a data file might be described as follows:

```
Call            :   /usr/don/phone/Callfile
    telno       :   string
    minutes     :   real
    miles       :   integer
```

Lines of `/usr/don/phone/Callfile` represent telephone calls, and each contains three fields: a character string named `telno`, a real named `minutes`, and an integer names `miles`, in that order. `Call` is the entity type name. One can think of a data file as a table with a fixed number of named, typed columns and an unspecified number of unnamed rows. Here are some Call data displayed in that form:

telno	*minutes*	*miles*
111-2233	1.65	55
111-2233	25.1	561
444-5566	1.88333	34
444-5566	19.45	455
444-5566	31.6333	736
777-8899	44.9167	1040

IV. FEATURES

A Datastream query consists of one or more *statements*. The simplest kind of query has just one statement that prints the value of some field from each line of some data file.

```
get each Call
    print .miles, .telno ;
```

results:

```
55        111-2233
561       111-2233
34        444-5566
455       444-5566
736       444-5566
1040      777-8899
```

The phrase `get each` *entity-type* is an iterator; it causes the rest of the statement to be executed once for each line in the entity type's data file. Lines are processed in the order in which they appear in the data file; reordering the lines in `/usr/don/phone/Callfile` would cause a corresponding change in the output shown above. The key word `print` introduces a list of values to be printed. The values can be simple fields as above, or more complicated *expressions*. The dots in `.miles` and `.telno` mean that `miles` and `telno` are fields in lines from Call's data file. They are redundant information in this query, but as we will see later, such dots play an important role in more complex ones. This convention is meant to suggest `Call.miles`, which is a familiar notation to users of C and Pascal.[8] The query compiler ignores most white space (that is, blanks, tabs, and newlines). Every statement ends with a semicolon.

4.1 Expressions and assignments

Typical Datastream queries are full of expressions. The arithmetic operations available are addition, subtraction, multiplication, division, and modulus. All are infix binary operators, and they are represented by the symbols used in C: + – * / %. The precedence rules from C are used, and arbitrary groupings can be specified with parentheses. Datastream and C part company on the issue of string concatenation. In Datastream the symbol ~ (tilde) is an infix binary operator that concatenates string arguments. C does not have string concatenation, and in C the tilde is a unary operator that computes the one's complement of its argument.

Conditional expressions are an important part of the language. A conditional expression has the form

`if` (*condition*) *expression1* `else` *expression2*

If the condition is true, then *expression1* is evaluated and its value is the value of the whole expression. If the condition is false, then *expression2* is evaluated and its value becomes the value of the whole expression. This is exactly the `?:` construction from C, set in a syntax that Datastream's users find more comfortable.

A condition can be any Boolean combination of comparisons of expressions. The Boolean operations are `or`, `and`, and `not`, in order of increasing precedence. Comparison operators are mostly taken from C: == != > >= < <=. In addition, a comparison can be a regular expression match:

string-expression `matches` "*regular-expression*"

A simple regular expression is used in the following example.

```
get each Call
     print if ( .telno matches "444-" )
         "city" else "suburb", .telno ;
```

results:

```
suburb     111-2233
suburb     111-2233
city       444-5566
city       444-5566
city       444-5566
suburb     777-8899
```

In general, strings can be matched against regular expressions of the same form as those recognized by the editor `ed`.

Early in Datastream's evolution, users who were not familiar with C complained that the conditional expression

```
if (a=b) 1 else 2
```

contained a syntax error (the symbol for equality comparison was supposed to be ==, not =). The two symbols are now treated as synonyms.

Properly installed functions written in C can also be used in expressions:

```
get each Call
    print .miles, log( .miles );
```

results:

```
55        4.00733
561       6.32972
34        3.52636
455       6.1203
736       6.60123
1040      6.94968
```

The `print` clause uses simple formats that are adequate for most purposes. Sometimes a query writer needs more precise control over the format of the output. The `printf` clause is available for this purpose. The query writer simply makes a `printf` call as it would appear in a C program:

```
get each Call
    printf( "phone number: %s time: %12.2e\n",
        .telno, .minutes );
```

results:

```
phone number:    111-2233    time:       1.65e+00
phone number:    111-2233    time:       2.51e+01
phone number:    444-5566    time:       1.88e+00
phone number:    444-5566    time:       1.94e+01
phone number:    444-5566    time:       3.16e+01
phone number:    777-8899    time:       4.49e+01
```

4.2 Control structure, part 1

The `get` statements we have described can be adjusted by inserting a `such that` clause:

```
get each Call
     such that .miles > 500
         print .telno, .miles ;
```

results:

```
111-2233      561
444-5566      736
777-8899      1040
```

The `such that` clause allows the rest of the statement to be executed only on lines of the data file for which the condition is true. There is no mechanism for optimizing data file access in the presence of a `such that` clause; every line of the data file is read and tested.

The `collect` statement is Datastream's second control structure.

```
initialize count = 0, totalmin = 0.;

collect each Call
     count = count + 1,
     totalmin = totalmin + .minutes
then
     print count, totalmin ;
```

results:

```
6    124.633
```

Like `get`, `collect` reads every line in a data file. The assignments between `collect` and `then` (known as *inner assignments*) are executed once for each line (in the order in which they appear in the query). When processing reaches the end of the data file, the `print` clause is executed. The `initialize` statement is executed once before anything else is done.

The query above would be shorter if one could write something like

```
print count( Call ), sum( Call.minutes ) ;
```

The main problem with functions such as `count` and `sum` is that they are never enough; analysts are adept at inventing questions that require endless subtle variations on functions they have used before. Rather than try to stay ahead of users' creativity, we designed the `collect` statement, with which one can write computations including `sum`, `count`, `average`, `maximum`, and so on, all in a similar way. It also allows more exotic things, such as the following stratified sum:

```
initialize shortmin = 0., medmin = 0., longmin = 0.;

collect each Call
     shortmin = if ( .miles < 100 )
                  shortmin + .minutes else shortmin,
     medmin = if ( .miles >= 100 and .miles < 500 )
                  medmin + .minutes else medmin,
     longmin = if ( .miles >= 500 )
                  longmin + .minutes else longmin
then
     print shortmin, medmin, longmin ;
```

results:

```
3.53333    19.45    101.65
```

The analysts who use Datastream are almost always comfortable writing this kind of computation. Initializations, counters, and partial sums are familiar to those with some programming experience (most of them), and conditional expressions are easy to learn. No one seems to miss programming the things that Datastream does automatically in a query like this: opening the file, reading and parsing the lines, recognizing the end of file, closing the file, and so on. Query writers are free to spend their time describing solutions to problems rather than programming input/output operations.

4.3 Mainstream variables

Some queries require more than a single statement. The mainstream provides a way to pass data values from one statement to the next for further processing. Wherever a `print` clause appears, a `keep` clause can be used instead.

```
print expression1, expression2, . . .

keep expression1 as name1, expression2 as name2, . . .
```

The print clause writes a stream of lines on the standard output, while the keep clause writes a stream of mainstream records into a pipe. A *mainstream record* is a set of values for a set of *variables*. The values can be any of the four Datastream types: integer, real, string, or pointer, but they are represented in an internal form that is not accessible to the user. To maintain this contrast between data in files and data in mainstream pipes, we will observe the following convention: data files are filled with lines, and lines consist of fields; mainstream pipes are filled with records, and records consist of variables.

Replacing print with keep in the long-haul query above produces

```
get each Call
     such that .miles > 500
            keep .telno as T, .miles as M ;
     .
     .
     .
```

This statement writes mainstream records as follows:

```
T               M
111-2233        561
444-5566        736
777-8899        1040
```

4.4 Control structure, part 2

At the receiving end of a mainstream pipe, one can write a compress statement.

```
get each Call
     such that .miles > 500
          keep .telno as T, .miles as M ;

initialize longest_call = 0, longest_telno = "";

compress
     longest_telno = if ( M > longest_call ) T else
     longest_telno,
     longest_call = if ( M > longest_call ) M else
     longest_call
then
     print longest_telno, longest_call ;
```

results:

```
777-8899      1040
```

One can think of the compress statement as a collect statement that operates on records from the mainstream rather than lines from a data file. The initialization is done once, after which the inner assignments are executed on each inbound mainstream record. The print clause executes when the inbound mainstream ends. The last example was contrived to make a simple illustration of the compress statement. It is possible (exercise for the reader) to rewrite it with a single collect statement. The following query uses a compress on clause to find the shortest-distance call for each Customer.

```
get each Call
    keep .telno as T, .miles as M ;
initialize shortest_call = 1000000;

compress on T
    shortest_call = if ( M < shortest_call ) M else
    shortest_call
then
    print T, shortest_call ;
```

results:

```
111-2233      55
444-5566      34
777-8899      1040
```

compress on is the most elaborate of Datastream's control structures. The stream of inbound mainstream records is divided into *compression sequences*. Each compression sequence is a maximal sequence of successive inbound mainstream records, all with the same values for the variables in the compress on clause (in this example, just T). The mainstream records written by the first statement are shown below, separated into compression sequences by blank lines.

```
T             M
111-2233      55
111-2233      561

444-5566      34
444-5566      455
444-5566      736

777-8899      1040
```

The initialization executes once at the beginning of the compression sequence. Then the inner assignment executes for each member of the sequence. Finally, the `print` clause executes once at the end.

Ordering is crucial here, because the statement processor does *not* look ahead in the inbound mainstream. If the Call file were rearranged as follows

telno	*minutes*	*miles*
111-2233	1.65	55
111-2233	25.1	561
444-5566	1.88333	34
777-8899	44.9167	1040
444-5566	19.45	455
444-5566	31.6333	736

then the compression sequences would be

```
T               M
111-2233        55
111-2233        561

444-5566        34

777-8899        1040

444-5566        455
444-5566        736
```

and the query's output would be

```
111-2233        55
444-5566        34
777-8899        1040
444-5566        455
```

The `compress on` clause has been valuable in constructing new data files as well. This use of it will be discussed further in Section 4.6.

4.5 Connected files

The original Datastream included a program called `build` which constructed "cooked" databases from "raw" data. It was capable of building arbitrary binary relationships between entity types. When `build` was scrapped for performance reasons, some means for connecting ordinary *UNIX* system files had to be found. The approach we have taken is not as general as the original, but it provides high performance and does not make specially formatted copies of large data files.

Datastream's current connection mechanism is based on pointers. A *pointer* is an offset from the beginning of a data file to the beginning of some data line. Pointers can be stored as fields in one file and used in queries to provide fast access to interesting lines in some other file. In particular, pointers are used to implement entity lists. An *entity list* associates one entity with an ordered list of entities of some other type. For example, assuming that the `telno` field in our Call file is the telephone number of the customer who dialed the call, consider the following data about Customers:

Configuration:

```
Customer                 :    /usr/don/phone/Customerfile
     telno               :    string
     Call_count          :    integer
     total_minutes       :    real
     total_miles         :    integer
     Call_pointer        :    Call control by telno
```

Data:

telno	*Call_ count*	*total_ minutes*	*total_ miles*	*Call_ pointer*
111-2233	2	26.75	616	0
444-5566	3	52.9666	1225	35
777-8899	1	44.9167	1040	95

`Call_count` is the number of calls the Customer made, `total_minutes` and `total_miles` are self-explanatory, and `Call_pointer` is the number of bytes (including new lines) of `/usr/don/phone/Callfile` that have to be skipped to reach the Customer's first Call.

The following example shows how a pointer behaves in a query.

```
get each Customer
     such that .telno = "444-5566"
          keep .Call_pointer as it ;
get each Call for it
     print .telno, .minutes ;
```

results:

```
444-5566     1.88333
444-5566     19.45
444-5566     31.6333
```

The keep clause defines it to be a pointer-valued variable. The second statement behaves just as it would without the for it, except that the first Call retrieved from the data file is the one it points to, and Calls are processed so long as they have the same value for telno. Readers familiar with relational database systems will recognize Call_pointer as a way of storing the results of joining Call and Customer on telno. The configuration file declares that telno is the *control field* for Call_pointer; that is, a change in its value signals the end of the entity list. The above syntax for declaring control fields is new, unsettled, and likely to change.

The first statement in the previous example writes only one outbound mainstream record. If it wrote more than one, the second statement would repeat its action for each inbound mainstream record.

```
get each Customer
     such that .Call_count >= 2
          keep .Call_pointer as it ;

get each Call for it
     print .telno, .minutes ;
```

results:

```
111-2233      1.65
111-2233      25.1
444-5566      1.88333
444-5566      19.45
444-5566      31.6333
```

Datastream supports a similar collect each . . . for clause, as the following query shows.

```
get each Customer
     such that .Call_count >= 2
          keep .telno as telno, .Call_pointer as it ;

initialize longest = 0. ;

collect each Call for it
     longest = if ( .minutes > longest ) .minutes else
     longest
then
     print telno, longest ;
```

results:

```
111-2233      25.1
444-5566      31.6333
```

The initialization is done once for each mainstream record. Then the inner assignment is done for each Call starting with the one `it` points to and continuing until `telno` changes. When the end of file or a Call with a different `telno` is encountered, the `print` clause executes and processing resumes with the initialization for the next inbound mainstream record.

Pointers can be printed, and they can be used in some expressions, notably conditional expressions. They cannot be used in arithmetic. Pointers can be compared for equality but not magnitude. The general rule is that algebraic manipulation of pointers should be discouraged because a pointer that does not point to the beginning of a data line can lead to seriously mangled results. Of course, pointer arithmetic and comparison are legal in many languages, and they may find their way into Datastream if a pressing need arises.

4.6 Constructing pointers

Users can make data files containing pointers any way they wish. It has been done with special-purpose C programs, but it is more common to use queries. As the statement processor traverses a data file, it keeps track of where it is. In any statement of the form

```
get each X . . .
```

or

```
collect each X . . .
```

the phrase `the` X is a pointer-valued expression whose value is the offset to the beginning of the current data line. This feature can be used as follows (note the use of `the Call` in the `keep` clause).

```
get each Call
     keep the Call as Call__inbound, .telno as telno ;

initialze Call__ptr = null(Call);
compress on telno
     Call__ptr = if ( Call__ptr = null(Call) )
          Call__inbound else Call__ptr

then
     print telno, Call__ptr ;
```

results:

```
111-2233        0
444-5566        35
777-8899        95
```

The mainstream records written by the first statement are shown below, separated into compression sequences with respect to `telno`.

```
Call_inbound          telno
0                     111-2233
17                    111-2233

35                    444-5566
55                    444-5566
74                    444-5566

95                    777-8899
```

An expression of the form `null`(*entity-type*) is always unequal to ordinary *entity-type* pointers in comparisons. This means the comparison in the conditional expression above is true only for the first member of each compression sequence, when `Call__ptr` has its initial value of `null(Call)`. The conditional expression evaluates to the ordinary Call pointer `Call__inbound`, which is assigned to `Call__ptr`. For all succeeding members of the compression sequence, `Call__ptr` is unequal to `null(Call)`, so its value does not change. Thus whenever `Call__ptr` is printed, its value is just a pointer to the first Call for some Customer.

The query above was written by a program called `canon`. `Canon` takes a configuration file and the name of a pointer field (`Call_pointer` in this example) as input. It writes a "canonical" pointer-construction query as output. Such a query always has two statements, a `get` and a `compress`. The `get` statement gets each *X*, where *X* is the entity type pointed to. The pointer's control fields and `the` *X* are kept. The `compress` statement compresses on the control fields, computes `X__ptr` as above, and finally prints the control fields and `X__ptr`. The user can run `canon`'s output as is, or augment it to compute summary information. The double underscores are a simple-minded way to make names generated by `canon` look different from names chosen by the user.

To compute the Customer data used in this paper, the canonical query shown above was modified by hand to compute `Call_count`, `total_minutes`, and `total_miles`.

```
get each Call
     keep .minutes as time, .miles as distance,
          the Call as Call__inbound, .telno as telno ;
initialize Call__ptr = null(Call),
     tdistance = 0, ttime = 0., count = 0;
compress on telno
     count = count + 1,
     ttime = ttime + time,
     tdistance = tdistance + distance,
     Call__ptr = if ( Call__ptr = null(Call) )
          Call__inbound else Call__ptr
then
     print telno, count, ttime, tdistance, Call__ptr ;
```

Pointer construction queries such as the above are sometimes used with the *UNIX* system command `join` to construct data files. For example, one might join the Customer data above to the results of a survey to make a more elaborate Customer file, or the output of two pointer construction queries could be joined to produce `Customer` data with, say, a list of Calls made and a list of Equipment installed.

V. EXPERIENCE

5.1 Positive

5.1.1 Size

At the time of this writing, the largest database accessible with Datastream contains about 70M bytes. A little more than half of that is original data; the rest was constructed by Datastream queries. Some databases are smaller than the original data. In the most dramatic case a file of 50M bytes turned out to be 60-percent blanks and insignificant zeroes in numeric fields. A query that simply printed every field reduced the original file to an equivalent one with only 20M bytes. This is common with data that analysts acquire from systems that use fixed-length fields.

5.1.2 Speed

Query processing performance is adequate, although users would be happier if every query finished in five seconds. Timing experiments found that Datastream's basic overhead on a VAX 11/780 computer running University of California, Berkeley *UNIX* 4.1 BSD was about nine microseconds of CPU time per byte of data read. Basic overhead was measured by timing a query that does nothing but count the records in a file. This query ran about 40 percent faster than the `cat` command on the same file with standard output ignored.

In another performance test, an expert programmer wrote C programs that answered three simple queries. These programs required 1 percent, 40 percent, and 250 percent more CPU time to process a 20M-byte file than was needed for equivalent Datastream queries. The programmer took about 40 minutes to write his programs, compared to ten minutes for the queries. The queries were chosen to show Datastream in a good light, of course, and they were written by the author of this paper, an expert query writer. When an analyst was asked to write such queries, his performed slightly *better* than the author's.

5.1.3 Use of pipes

Datastream's use of separate statement-handling processes communicating through pipes made implementing the query-processing code simpler than it might have been otherwise, but it carried a risk of bad performance. Specifically, we were concerned about the possibility of thrashing among statement processors competing for the same CPU, and swapping of pipes competing for the same system buffer space. The first has not been a serious problem because of the large (10K-byte) buffers used in the statement processors. In general, a statement processor has room to read or write 10 to 100 mainstream records or data file lines at a time. This means it can do considerable processing without giving up the CPU. Second, we have never found swapped pipes in our performance experiments. Pipes do not fill beyond a system-specific limit (usually 10K bytes), and processing a query requires at most nine of them at any one time. Most queries need four pipes or fewer. Our experience has been on machines with three or four megabytes of main memory, and these systems have had no trouble accommodating the pipes. Of course, the same large memories support large buffers in the statement processors as well.

5.1.4 Use of an intermediate code

The decision to separate the query compiler from the statement processor with an intermediate code turned out to be a good one. It has simplified development, debugging, maintenance, and tuning: to date no bug has required changes to both modules, and several times major changes have been made in one without affecting the other. A human expert can read the intermediate code without much difficulty, and that has proved helpful in debugging.

5.1.5 Query language

Users find that it takes some work to learn the control structures, especially `compress on`. Once grasped, however, the language seems easy to use. It allows analysts to concentrate on specifying computations without spending much time on the details of I/O or control structure.

5.2 Negative

5.2.1 Updates

The switch to ordinary files has made it much easier to correct errors or append new data than it used to be. However, derived data (summaries, pointers) sometimes account for a large fraction of an analytical database (as much as 50 percent), and it has to be rederived when the underlying data changes. This can be a real nuisance; some applications have decided not to use Datastream because they needed to do too many updates. Others have considered the possibility of using Datastream for queries and other utilities or ad hoc code to handle updates.

5.2.2 Connections

From time to time an application would like a more powerful mechanism for connecting files than the sort-and-point technique available now. For example, data files are often sorted on the control fields for some pointer, and secondary indices into such files would be helpful for some applications. Some extensions in this direction seem reasonable and may be implemented in the future.

5.2.3 Functions

Some systems (notably S[3,4]) put procedures supplied by users in processes separate from the basic code. This simplifies installation and protects the basic code from name conflicts and bugs in the code supplied by users. Unfortunately, it is difficult to make such a scheme work efficiently in Datastream because overhead for context switching and interprocess communication can be large compared to calling a function. At present we maintain two versions of the statement processor. One is the basic statement processor with no functions at all, while the other is the basic statement processor loaded with a set of functions that are shared by all the Datastream users on a given system. The shared library had 17 functions at last count. It can probably grow to several times that size before it becomes unmanageable. Datastream also provides a utility that allows analysts to make statement processors loaded with personal libraries of functions that can be called from queries.

Another problem with functions results from evolution: `printf` clauses and expressions of the form `null`(*entity-type*) look like function calls, but they do not behave exactly like functions. Fortunately, this inconsistency does not seem to bother the users.

VI. SUMMARY AND POSSIBLE IMPROVEMENTS

Experience with Datastream has shown that a language designed to simplify access to large analytical databases can be useful in a *UNIX* system environment. Datastream meets that need except for some defects and omissions. Its users routinely use it to get information from files that are five to ten times larger than we thought possible when work on Datastream began. In spite of some initial reservations about the architecture of the query-processing software, we have achieved an efficient implementation. The conversion to ordinary files has been well worth the effort; it eliminated much redundant use of disk space and allowed Datastream to work with, rather than against, other software tools.

Datastream might benefit from a facility for efficient random sampling from data files and a way to define (not just call) functions in queries. New data structures and connection mechanisms could be supported by constructing a set of statement processors, each with a different routine for handling data files. This is feasible because the query-processing software is almost independent of the low-level file handler. The new structures could be fully updateable, at least in principle, and the new file handlers might even be those of some other database system.

VII. ACKNOWLEDGMENTS

I would like to thank the original Datastream team of Ed Fisher, Steve North, and Ray Yanofchick; others who provided suggestions and encouragement, including John Chambers, Dave Belanger, Rick Becker, John Walden, Bob Kayel, Bill Keese, and Jim Downs; and the courageous users of the prototype: without them none of this would make any difference.

REFERENCES

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Englewood Cliffs, N.J.: Prentice-Hall, 1978.
2. D. Swartwout, "How Far Should a Database System Go? (to Support a Statistical One)," Proc. Second Int. Workshop on Statistical Database Management, Los Altos, Calif., September 27–29, 1983.
3. R. A. Becker and J. M. Chambers, *S: An Interactive Environment for Data Analysis and Graphics,* Belmont, Calif.: Wadsworth, 1984.
4. R. A. Becker and J. M. Chambers, "Design of the S System for Data Analysis," Commun. ACM, *27,* No. 5 (May 1984), pp. 486–95.
5. S. C. Johnson and M. E. Lesk, "Language Development Tools," B.S.T.J., *57,* No. 6 (July–August 1978), pp. 2155–75.
6. S. R. Bourne, "The UNIX Shell," B.S.T.J., *57,* No. 6 (July–August 1978), pp. 1971–90.
7. P. P.-S. Chen, "The Entity-Relationship Model—Toward a Unified View of Data," ACM Trans. Database Syst., *1,* No. 1 (1976), pp. 9–36.
8. N. Wirth, "The Programming Language PASCAL," Acta Informatica, *1,* No. 1 (1971), pp. 35–63.

AUTHOR

Don Swartwout, B.A. (Mathematics) 1974, Kalamazoo College; Ph.D. (Mathematics) 1979, University of Michigan; AT&T Bell Laboratories, 1979–1985; AT&T Information Systems, 1985—. Mr. Swartwout's dissertation dealt with the mathematical foundations of database systems. At AT&T Bell Laboratories, he has worked on concurrency control and query languages for database systems, programming environments, source-to-source programming language translation, and compiler development. He continues working on compiler development at AT&T Information Systems.