DSS: Data Stream Scan

Glenn Fowler and Balachander Krishnamurthy

Abstract— ***dss* (data stream scan) is a framework for describing, transforming, reading, querying, and writing streams of record oriented data. *dss* is implemented as a command and library API and used extensively to aid network measurements in our organization. The API is extended by DLLs (shared libraries) that define data domain specific I/O, type and query functions. We provide a best-in-class repository for data scanning, along with up-to-date documentation as a side effect of coding to the API. Many large scale network applications have used *dss* significantly reducing the time spent in coding and in querying data. The reasons for the success of *dss* are its lightweight and extensible architecture, generic usage template, scope of supported data domains, scalability, and speed. *dss* compares extremely favorably against *perl*, the typical recourse in the networking community, and against customized C/C++ code written to deal with specific data sets. *dss* has been successfully applied over three years over a wide range of applications, including large volumes of `NETFLOW` data, `BGP` tables, `HTTP` proxy and server logs, $SMTP$ logs, and `OSPF LSA` (Link State Advertisements.) The use of *dss* has led to fearless exploration of ideas across very large volumes of data.**

I. Introduction

Parsing is a critical part of record oriented data analysis. It involves describing the record structure and splitting the record fields into a form suitable for the analysis tools. Each tool may have its own idiosyncratic record description and field identifier syntax and format, and some tools may be better at hiding the details than others. For example, tools like *grep*, *awk*, and *perl*, intermix low and high level definition details within the same program. Although relational databases provide high level abstractions and GUI interfaces suitable for an end user, data stream management and initial loading typically require more expertise. Loading expense may be worthwhile only for data that will be scanned more than a few times.

It is common to have many analysis tools, each with an independent parser, applied to the same data within a single group of analysts. This in itself is not a bad situation. A relational database may be overkill for an application that needs to visit each record only once; hand coded programs may be inappropriate for data that is too large to fit in memory, or that may benefit from indexed access, or that may undergo many record format changes. However, subtle parsing differences between independent analyzers can make comparisons difficult. The absence of read or load errors does not necessarily mean that the data was read correctly. While data access bugs can be fixed as they are discovered, in the worst case this could mean translating the fix to every analysis tool. In some cases it may even be impossible to recover from bugs that affected intermediate legacy data.

dss provides a data abstraction that can be used to mitigate parsing differences between independent analysis tools. The key is that the abstraction provides efficient (in space and time) I/O support *and* a uniform interface for all data. This approach has multiple benefits: (1) allows data analysts to focus on analyzing data; (2) provides a target API for data I/O coders; (3) third parties familiar with the data abstraction model are automatically proficient in new data domains.

The authors are with AT&T Labs–Research, New Jersey. E-mail: gsf@research.att.com, bala@research.att.com

dss arose out of experience with `BGP` data analysis in our lab. A few in-house `BGP` tools gained popularity with the analysts. Each new `BGP` feed and file format (*cisco* router table dumps, `MRT` format data, and ad-hoc flat files) necessitated a recoding effort, which meant changing a handful of related but slightly different implementations that had evolved over a four year period. Merging the `BGP` I/O and parsing into a unified API was a big time saver, and also uncovered several embarrassing bugs that had remained hidden for years. Also, by employing best of class coding practices, the merged interface produced noticeably more efficient code. For example, the *dss* `BGP` `MRT` parser is 10 times faster than the one published by the originators of the format. By the time the analysts requested help on `NETFLOW` data the lesson had been learned, and *dss* was designed, implemented, and extended to several additional data formats. Overall *dss* shows that a tool can handle a wide variety of data formats while being accurate and as fast or faster than tailored solutions.

II. Data Abstraction Model

dss partitions data stream access into components that form a uniform model for all data. The model guides both the implementation and user interfaces. It supports independent data methods, each with one or more physical formats. For example, the `BGP` method supports *cisco* router dump and `MRT` formats, and the `NETFLOW` method handles all versions of the *cisco* dump format. Each user visible item, whether in the base API or in DLL extensions, is placed in a dictionary. A dictionary entry has a name and descriptive text suitable for inclusion in manual pages. This information can also be listed by the runtime interfaces, so up-to-date documentation is always available. Figure 1 illustrates the component relationships in the *dss* model. Some components are part of the default *dss* library, but most are implemented as independent *dss* API DLLs. Applications link against the base *dss* library which handles the mechanics of locating and loading the DLLs at runtime. Old applications thus have access to new methods without requiring recompilation.

A. TRANSFORM Components

Transforms are generic data independent filters that are applied to the raw data. They are automatically detected and applied on input, and are selectively applied on output, as determined by user and/or data specific options. *dss* currently supplies compression transforms for *gzip*, *pzip* [1], *bzip* and *compress*; support for deltas (*vcdiff* [2]) and encryption is planned. A transform may be implemented within the main *dss* process (e.g., the *sfio* [3] *gzip* discipline) or as a separate process (e.g., *gzip* or *gunzip* connected by a pipe.) In some cases it may be possible to delay the current vs. separate process choice until runtime. For example, on a multi-cpu architecture with idle processors, running *gzip* as a separate process may take less real time than decompressing and scanning in a single process.

Transforms allow data to be stored in the most efficient (or se-

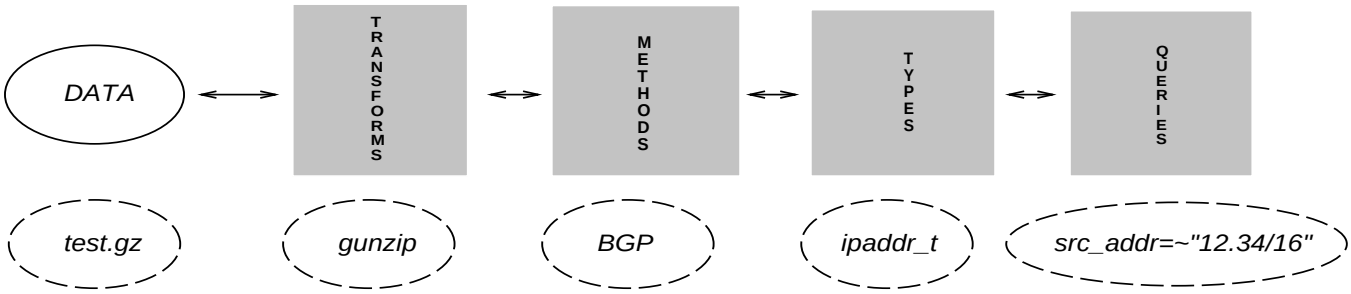

Fig. 1. dss Component Architecture

cure) manner. The data need only be converted when accessed. Given the volumes of available network data, compression or deltas, along with sampling, are essential for archiving.

B. METHOD Components

A method describes the data records for a specific data domain and must be specified before *dss* accesses any data. It provides a uniform view of all data domains supported by *dss*, both from the user command line and API perspectives. Lessons and techniques learned in one method most likely will apply to other methods as well. The method data description includes:

- A dictionary of file storage formats with three functions: an identification function that automatically determines input formats (a format may be a different version (`NETFLOW` versions 1, 5 and 7) or a completely different structure (`BGP MRT` files vs. *cisco* router dumps.), and record read and write functions, both of which also handle format specific headers and trailers.
- A dictionary of record field types and names that includes the union of fields available in all supported formats. Fields absent in a particular format will have empty values. For example, the *dss* `NETFLOW` method supports all popular `NETFLOW` formats, and the same queries can be used on any of them. The `source mask` field, not in the version 1 format, is still available for version 1 queries, but with a default value of 0.
- An optional canonical record that provides access to a C structure representation of each record. A method that provides a canonical record can simplify ad-hoc C queries by allowing the queries to be implemented as dynamic *dss* QUERY component functions, described below.

C. TYPE Components

A *dss* type provides functions that convert data between internal and external data representations. Some types may also have a base type that provides alternate access to the same data. Types are method independent and are often shared between methods. For example, the `ipaddr_t` type in the *dss* `ip_t` type library implements an `IP` address as a 32 bit unsigned integer (the base type), and converts between the integer representation and quoted dotted-quad or `DNS` names as the application requires. Often logs contain a mixture of dotted-quad and `DNS IP` addresses. Both are handled by automatic conversions depending on the context.

D. QUERY Components

Queries are the user visible part of *dss*. They are used to select, filter, and summarize data stream records. There are two forms of queries. Interpreted queries provide named access to the data fields using C style operators, and dynamic queries provide C-level access to the record stream. Interpreted queries allow users to experiment on small sets of data with many different queries, and then to switch to a dynamic query for large data sets. The impact of switching is mostly on running time, i.e. the only changes are coding the query in a DLL (about 100 lines of C), and changing the dss command line query string to access the DLL. Queries may be composed similar to a UNIX pipeline. Interpreted queries are generic and may be used with any method. Dynamic queries may be generic or method specific.

III. QUERY EXPRESSIONS AND COMPOSITION

Interpreted queries are `C` style expressions on the data fields defined by the current method. The `C` expression grammar is extended to allow string operands for the `< <= == != >= >` operators and string regular expression matching for the `=~ !~` operators, where the regular expression pattern is the second operand. Non-numeric constants are represented as quoted strings; e.g., the `IP` address `"12.34.56.78"` and the `AS` path `"123 456 789"`. A *dss* type may optionally provide a match function that overrides the default regular expression string match for the `=~` and `!~` operators. For example, the `ipaddr_t` type match function does `IP` prefix matching: `(src_addr =~ "12.34/16")` selects all records where `src_addr` matches the `IP` prefix `12.34/16`. The `aspath_t` type match function does `AS` path regular expression matching where `AS` numbers are treated as individual tokens (as opposed to the inefficient alternative of converting the `AS` path to a string and applying a string regular expression match.) `(path =~ "^ 123 [ 456 789 ] - 8765$")` selects all records where `path` starts with `AS 123` followed by `456` or `789` and ends with `8765`.

A dynamic query is similar to a UNIX command, except that it is executed within the *dss* process. Dynamic queries accept options (including `--man` for online documentation) and arguments, and are implemented by four functions:

- An *init* function that is called once before the first record is read. *init* checks the options and arguments and may allocate private data to be used by the *select*, *act*, and *done* functions.
- A *select* function that is called for each record. If it returns `true` then the *act* function is called, and if it returns `error` then the scan terminates with an error.
- An *act* function that is called for each record for which the *select* function returns `true`.

• A *done* function that is called once after the last record is read. *done* may list reports or summaries, and also releases any private resources allocated by the *init*, *select* and *act* functions.

The basic *awk* paradigm (begin, match, act, done) is evident in the dynamic query structure. There is also some similarity between *perl* modules and DLLs containing dynamic queries. The narrow focus of the *dss* API, however, allows for a much more efficient implementation.

The default *dss* library provides a few generic dynamic queries. The simplest is `{count}` which prints the number of selected records and total number of records after all records have been read. The `{stats}` query computes the count, mean, and unbiased standard deviation for the named numeric field arguments, and supports grouping by field value:

```
{stats --average --group=prot bytes}
```

This `NETFLOW` example computes the average number of bytes per packet (the numeric `bytes` field value) grouped by protocol (the `prot` field.)

Dynamic queries enclosed in `{...}` and interpreted queries enclosed in `(...)` may be composed using the `|`, `?:` and `;` operators. Query composition allows for efficiently applying many queries to a data stream using just one pass over the data. Only one copy of each record is generated; the composition operators simply pass a record reference to the appropriate operands.

`A|B` specifies that query `B` processes records selected by query `A`. `A?B:C` specifies that query `B` processes the records selected by query `A` and query `C` processes the records not selected by query `A`; query `B` may be omitted. `A;B` specifies that queries `A` and `B` process the same records.

This `NETFLOW` query

```
(src_addr =~ "123.45/16") |
{stats --average --group=prot bytes}
```

lists the average number of bytes per packets, grouped by protocol, for the records with source `IP` address matching the `IP` prefix `123.45/16`. This `NETFLOW` query

```
(prot == "TCP") ? {write > tcp.out} :
(prot == "UDP") ? {write > udp.out} :
{write > other.out}
```

splits the input record stream into three files, `tcp.out` for `TCP` records, `udp.out` for `UDP` records, and `other.out` for all other protocols.

When dynamic queries are composed the evaluation order of *init* and *done* functions is important since one query may reference resources allocated by another query. The order is resolved using the composed query parse tree. The *init* functions are evaluated using a post-order depth first traversal, and the *done* functions are evaluated using a pre-order depth first traversal.

IV. Command and Library API

The `dss` command provides generic access to all *dss* transform, method, type and query components. It can be used to test new components under development, as well as to explore interpreted queries that may be candidates for implementation as method-specific dynamic queries. The command syntax is:

`dss` [*options*] *query* [*file* ...]

The `dss` command scans a record-oriented data stream, applies a *query* expression to each record, and writes the matching records to the standard output. The *query* `-` matches all records. If *query* does not contain `|{write format}` then the output format is the same as the format of the first input *file*.

The `-x` *method* option specifies the data method and `--man` lists the manual page (with details for all options) on the standard error. `dss -x netflow '{write --man}'` lists the formats supported by the `netflow` method, while `dss -x dss '{stats --man}'` lists the `stats` query manual page, and `dss -x bgp {count} *.gz` counts the number of `bgp` records in all `.gz` files in the current directory.

The *dss* library API is based on the discipline and method architecture [4] where disciplines control library diagnostics and methods control data access. The user opens a session handle by first initializing a discipline and method pointer, and then calling the `dssopen()` function. Data files are accessed by opening one or more file streams using `dssfopen()` and reading a record with `dssfread()` or writing new records with `dssfwrite()`. Handle-specific resources are freed by calling the corresponding *close* function: `dssclose()` closes a *dss* session, and `dssfclose()` a file stream. More than one file stream may be open within a single *dss* session handle to do file format conversion. Similarly, more than one session may be open within a single program; e.g., programs that collect data from more than one method type.

V. Generic Flat File Method

The `FLAT` method provides generic support for files of records that contain fixed width or delimited fields, with optional (and ignored) header and trailer records. The record fields are described in a separate file using XML tags. XML was chosen to avoid writing yet another tiny description language (*and* lexer *and* parser); it is table driven and easily extended, and is also self documenting (`dss -xflat,man` describes the `FLAT` method tags.) `FLAT` methods are useful for accessing intermediate data files and file formats typically found in the UNIX and networking communities: the password file, `HTTP` proxy/server logs. Legacy COBOL and fixed width binary formats are also supported. This example illustrates basic `FLAT` method concepts:

```
<FLAT>
 <NAME>http-url</><DESCRIPTION>http url log file</>
 <LIBRARY>ip_t num_t time_t</>
 <HEADER><DELIMITER>&newline;</><COUNT>1</></>
 <FIELD>
  <NAME>time</><TYPE>time_t</>
  <DESCRIPTION>request time</>
  <PHYSICAL>
   <QUOTEBEGIN>[</><QUOTEEND>]</><DELIMITER>&space;</>
  </>
 </>
 <FIELD>
  <NAME>addr</><TYPE>ipaddr_t</>
  <DESCRIPTION>request IP address</>
  <PHYSICAL><DELIMITER>&space;</></>
 </>
 <FIELD>
  <NAME>customer</><TYPE>string</>
  <DESCRIPTION>customer ID</>
  <PHYSICAL><CODE>ebcdic</><WIDTH>64</></>
 </>
```

```
 <FIELD>
  <NAME>accesses</><TYPE>integer</>
  <DESCRIPTION>access count</>
  <PHYSICAL><TYPE>le_t</><WIDTH>4</></>
 </>
 <FIELD>
  <NAME>url</><TYPE>string</>
  <DESCRIPTION>request URL</>
  <PHYSICAL><QUOTEBEGIN>"</><DELIMITER>&newline;</></>
 </>
</>
```

- `<FLAT>` encapsulates the entire `FLAT` method description
- `<LIBRARY>` tags name the required type component libraries
- `<FIELD>` tags specify the record fields
- `<PHYSICAL>` tags specify the physical record syntax: quoting, delimiters; the last field delimiter also serves as the record delimiter
- `<CODE>` tags specify alternate character code sets
- `<WIDTH>` tags specify fixed (non-delimited) field widths
- `<TYPE>` tags within `<PHYSICAL>` tags specify the physical storage type: here `le_t` specifies a little endian integer

Typically this description would be placed in a file, say `small-log.dss`, and copied to a common *dss* directory. The `-xflat:small-log dss` command option would then access it as a `FLAT` method.

The `FLAT` method can be used to convert data from an old schema to a new one. The old schema and data files are passed as *dss* arguments, and the new schema file is passed as a `{flat}` conversion query argument. Fields omitted in the new schema are ignored, and additional fields not in the old schema are synthesized with default values. All field attributes are taken into account during the conversion. The `{flat}` conversion query has been used in scripts that accept mainframe (EBCDIC,BCD) COBOL copybooks (schema descriptions) and data and convert the data to field-delimited newline-terminated flat files suitable for UNIX database loading.

VI. Applications

We have fine tuned the *dss* command and libraries to suit the growing needs of a range of applications. The basic nature of *dss*, a stream model that simplifies the process of separating parsing from semantic-actions that are unique to the application at hand, remains unchanged. It is important to stress the role of *dss* in removing one of the key bottlenecks for a data analyst: the fear of not being able to test out ideas on significant amounts of data (e.g., a full day's worth of netflow dump from a router in a large ISP) in close to real time. The different categories of use of *dss* include parsing data, parsing and processing data into specific formats for different uses later, and finally, parsing and processing data via specific DLL queries with application-specific semantics encoded in the queries without loss of generality. In each of these cases the generality of the *dss* model actually helped to simplify both existing application implementations and new application designs.

dss has been ported and tested on UNIX, Windows, Mac and MVS architectures. All reported times, unless otherwise mentioned, are for a solaris machine with 48 sparc 900 MHz cpus.

A. Change Detection

To examine large scale changes in traffic as a means to identifying anomalies [5], we needed to process large amounts of netflow data and output a subset of valid fields in binary form to enable subsequent sorting. We processed several billions of netflow records using 20 lines of new C code added to a boilerplate netflow dynamic query. Typical processing time was over 220K records/second for most kinds of queries, as shown in Table I. The first column shows the duration of netflow (in minutes, ranging from 10 minutes to a full day) while the second column shows the number of records (ranging from 2.5 Million to 328 Million records). Note that all the netflow files are in `.gz` format and *dss* reads them in `.gz` format. The first query, `dss -xnetflow '{count}'` simply counts the number of records, with the numbers in parentheses indicating the thousands of records processed per second. The fourth column shows the time for printing the number of packets in typical DNS transactions, via the query:

```
dss -xnetflow '(prot==17&&dst_port==53)|
       {print "%(packets)d"}'
```

The last column shows the times for printing destination addresses of typical Web transactions (protocol is TCP and destination port number is 80) where the TCP SYN_FLAG is set, when restricted to a single (unsampled) router interface:

```
dss -xnetflow '(input==9 && prot==6 &&
      dst_port==80 && (tcp_flags&0x02))|
      {print "%(dst_addr)s"}'
```

Note that in all cases, with significant increase in the number of records processed, the rate of processing (in K records/second) remains largely constant, indicating the scalability of the *dss* approach. It should be clear that *perl* is not a practical option for files of this size.

B. OSPF LSA

Not all projects using *dss* do so for speed. The OSPF monitor project [6] collects LSA (Link State Advertisements) data from the network. *dss* is used to parse and select LSAs, based on user-specified query expressions, for subsequent processing and analysis. OSPF topology changes and OSPF routing table record types emanating from the LSA processing are also parsed by *dss*. The use of canonical records in *dss* allowed queries and analysis routines to be written against them, making it easer to add support for additional LSA formats. Prior to using *dss*, this project compiled queries directly into the analysis code; separating the tasks without sacrificing speed led to significant improvements. On a recent LSA trace of nearly 100 files representing 120 routers, *dss* was able to complete in under 7 seconds.

C. BGP Forwarding Tables

BGP forwarding tables dumped daily from a large number of routers are parsed to extract specific attributes for a variety of applications. We use an illustrative example: a routing emulation tool[7] that performs "what-if" analysis to predict how changes in BGP routing policies affect the flow of traffic in a large IP backbone. The largest dataset used by the tool are the

Log length (minutes)	gzip'd netflow file size(MB)	Number of flow records	Counting occurrences	Packet counts of DNS records	TCP flows on 1 interface on port 80 with SYN set
10	5.9	2502960	5.20s (481 Krec/s)	10.95s (228 Krec/s)	10.42s (240 Krec/s)
20	12.1	5125860	10.89s (470 Krec/s)	23.16s (221 Krec/s)	20.80s (246 Krec/s)
30	18.1	7664850	16.47s (465 Krec/s)	33.62s (228 Krec/s)	32.11s (239 Krec/s)
720	3304	147537990	5m 9.4s (476 Krec/s)	10m 38.43s(231 Krec/s)	10m 4.7s(244 Krec/s)
1440	7513	328198200	12m 52s (425 Krec/s)	25m 20.94s (216 Krec/s)	23m 6.58s (237 Krec/s)

TABLE I
DSS SYS+USER TIME FOR DIFFERENT ACTIONS IN NETFLOW DUMPS

Log ID	# of records	Perl script	dss
B-L1	358339	840.88	16.99
B-L2	353538	1024.22	20.04
B-L3	339437	905.91	18.39
B-M1	290074	926.42	18.22
B-M2	284394	17.29	2.76
B-M3	284701	8.37	2.64
B-S1	127778	119.42	3.22
B-S2	104443	17.78	1.17

TABLE II
PERL SCRIPT VS. DSS CPU TIME COMPARISON FOR EXTRACTING BGP ATTRIBUTES IN FORWARDING TABLES

Log	# of records	C code	dss
P-L1	8807651	4m 4.32s	2m 40s
P-L2	8475763	3m 55.08s	2m 6.5s
P-L3	8358513	3m 41.67s	2m 32.98s
P-M1	6054027	3m 5.58s	1m 54.49s
P-M2	5476062	2m 7.23s	1m 43.76s
P-M3	5347157	2m 3.08s	1m 43.37s
P-S1	569255	14.25s	10.66s
P-S2	380288	11.34s	7.10s

TABLE III
C CODE VS. DSS SYS+USER TIME COMPARISON FOR PRINTING URL QUERY IN HTTP PROXY LOGS

BGP routing tables for the edge routers in the network. Parsing and loading the BGP tables contributed most of the latency and lowering this overhead was critical. In Table II we present a comparison between running *dss* and the *perl* script in use earlier.[1].

D. SMTP data

We have access to lengthy SMTP logs of a very large ISP. The SMTP implementation is not *sendmail* but a tailored local variant. The *dss* log analysis filters out suspected spam email—a key problems on the Internet today. Most fields are *name=value* to handle sparse records, but the semantics of *name* may depend on the values of other fields. The `FLAT` method has sub-structure constructs to qualify and disambiguate fields. Typical queries involved extracting `From` and `To` address pair counts, examining error records, correlating mail sizes and suspected addresses, and correlating SMTP error codes and email ids. Space constraints prevent us from presenting detailed numbers on this application except to state that the average processing time is around 200 K records/second.

E. HTTP Data

We have extensive collections of HTTP Web server and proxy logs, and one author has considerable experience in parsing them over the years. Although this work is fairly standardized, it was instructive to code a *dss* parser for HTTP logs and compare both the speed with which such a parser can be built and the run times. Corresponding *dss* `FLAT` method descriptions were coded and debugged in several hours. The HTTP proxy log description was 102 lines and the server log description was 86 lines (both with documentation.) The result was that Web server and proxy logs could be parsed as well or better than C code specifically tailored to the data. One of the reasons for this is that *dss* analyzes queries at runtime and controls the parser to compute details only for fields accessed by the query. Other parsers, e.g., in the *scanf* style, may convert all numeric and string field values, even for fields ignored by the current query.

When compared with the publicly available webalizer [8] tool (also specifically coded for HTTP log files), the webalizer parsing alone was twice as slow as simple *dss* `{count}` queries.

The proxy log comparisons were done on an SGI machine with 16 250 Mhz and 8 300 Mhz CPUs. The server logs comparisons were carried out on a different SGI machine with 12 500 Mhz CPUs; the choice of machines had to with where the data resides. As Tables III— VI show, *dss* times are consistently significantly less than tailored C code for proxy and server logs. The smallest improvement is 22% and in some cases *dss* is nearly twice or more than twice as fast.

VII. EXPERIENCE, RELATED AND FUTURE WORK

The *dss* tool consists of five software components (the four components already mentioned plus a base glue component): the base library and command with 10K lines of code, the methods with 13K lines, the types with 4K lines, the standard dynamic queries with 2K lines, and the transform components (open source code not written by the authors and not counted here), for a total of about 28K lines of C code. While sizes for the support code needed to handle each method varies, it ranges

[1] We thank Nick Feamster for his help in getting us the benchmarking numbers

Log	# of records	C code	dss
P-L1	8807651	2m 9.96s	1m 15.22s
P-L2	8475763	1m 48.89s	1m 12.91s
P-L3	8358513	2m 4.17s	57.52s
P-M1	6054027	1m 39.26s	45.86s
P-M2	5476062	1m 23.21s	53.88s
P-M3	5347157	1m 24.62s	50.85s
P-S1	569255	9.27s	4.18s
P-S2	380288	6.10s	3.44s

TABLE IV
C CODE VS. DSS SYS+USER TIME COMPARISON FOR COUNTING RECORDS IN HTTP PROXY LOGS

Log ID	Log size	C code	dss
S-L1	8919932	2m 10.24s	1m 21.07s
S-L2	8911376	2m 9.55s	1m 20.91s
S-L3	8707094	2m 7.59s	1m 19.03s
S-M1	7750026	1m 52.49s	1m 10.33s
S-M2	7531624	1m 49.15s	1m 8.30s
S-M3	7518351	1m 49s	1m 8.11s
S-S1	4550081	1m 6.37s	40.97s
S-S2	4419307	1m 4.05s	39.87s

TABLE V
C CODE VS. DSS SYS+USER TIME COMPARISON FOR PRINTING REQUESTS IN HTTP SERVER LOGS

from 1.3K lines for `NETFLOW` to 3.3K lines for the more general `FLAT` method. A ballpark estimate is that a simple C-struct type data method implementation would be about 1K lines and more complex formats (`MRT`, *cisco*) would be around 3K lines.

As stated in Section VI, handling new flat file formats (such as HTTP proxy/server logs), is of the order of 100 lines of code. The run times for *all* methods are faster than any comparable *perl* or C code that we have been able to compare against. While it is possible that highly tailored, format-specific fast code might exist for some domains, we are fairly sure that it will not be as general as *dss*. If faster code materialized then we would simply commandeer it into an existing method, transparent to the users (except for runtime speedups.) The built-in documentation enables analysts to learn about the tool's applicability to all supported domains.

New networking data domains and formats are added as the need arises, with a reasonable degree of confidence that they will fit our model. Among related tools that deal (only) with netflow, are Flowscan [9] and flow-tools [10]. FlowScan relies on the slow *flex* lexer as their documentation indicates and the filter expression is re-evaluated on a per-flow basis. Flowscan is capable of reading a variety of formats including *cflow* [11] and *flow-tools*. We added the *flowtool* format (thanks to Mark Fullmer for useful information) to the *netflow* method in a couple of hours, so as to compare relative speed. A quick test showed that *dss* was twice as fast as *flowtools* with gzipped data. A direct comparison of run times against the same dataset showed that *dss* was more than five times faster on a comparable architecture. In [12], the authors describe module composition, which roughly corresponds to dynamic queries, except that inter-module relationships are hard-wired within the modules themselves. Unlike *dss*, this rules out interpreted control over query composition.

Log ID	Log size	C code	dss
S-L1	8919932	1m 9.65s	37.31s
S-L2	8911376	1m 9.64s	37.29s
S-L3	8707094	1m 7.1s	36.32s
S-M1	7750026	1m 0.58s	32.46s
S-M2	7531624	58.65s	31.36s
S-M3	7518351	58.43s	31.29s
S-S1	4550081	34.94s	18.73s
S-S2	4419307	34.09s	18.22s

TABLE VI
C CODE VS. DSS SYS+USER TIME COMPARISON FOR COUNTING RECORDS IN HTTP SERVER LOGS

We also plan to add a *dss* builtin to *ksh93* and to extend the *ksh93* type system with *dss* type components.

dss has saved a considerable amount of work in our organization. It has made it easier for new analysts to come on board to existing projects; it has simplified access to new data formats, both in coding parsers and queries; and it has improved the integrity of data analysis by providing a common repository for the sometimes mundane tasks that nonetheless must be correct. We expect to make at least the command, library API, `FLAT` file method, and HTTP proxy/server log schemas open source.

REFERENCES

[1] A. L. Buchsbaum, D. F. Caldwell, K. W. Church, G. S. Fowler, and S. Muthukrishnan, "Engineering the compression of massive tables: An experimental approach," in *Proc. 11th ACM-SIAM Symp. on Discrete Algorithms*, pp. 175–84, 2000.

[2] D. Korn and K.-P. Vo, "Engineering a Differencing and Compression Data Format," in *Proceedings of Usenix'2002*, USENIX, 2002.

[3] D. G. Korn and K.-P. Vo, "SFIO: Safe/Fast String/File IO," in *Proc. of the Summer '91 Usenix Conference*, pp. 235–256, USENIX, 1991.

[4] K.-P. Vo, "The discipline and method architecture for reusable libraries," *Software, Practice and Experience*, vol. 30, pp. 107–128, 2000.

[5] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based change detection: Methods, evaluation, and applications," in *Internet Measurement Conference 2003*, October 2003. `http://www.research.att.com/~bala/papers/nad-imc03.ps`.

[6] A. Shaikh and A. Greenberg, "Experience in black-box ospf measurement," in *Proc. of the Internet Measurement Workshop*, Nov 2001.

[7] N. Feamster, J. Winick, and J. Rexford, "A model of BGP routing for network engineering," in *ACM SIGMETRICS*, June 2004.

[8] "The webalizer. what is your web server doing today?." `http://www.mrunix.net/webalizer/`.

[9] D. Plonka, "Flowscan: A network traffic flow reporting and visualization tool," in *Proc. USENIX 14th System Administration Conference*, Dec 2000.

[10] M. Fullmer. `http://www.splintered.net/sw/flow-tools/`.

[11] cflow. `http://net.doit.wisc.edu/~plonka/Cflow/Cflow.html`.

[12] G. V. Cristian Estan, Stefan Savage, "Automated measurement of high volume traffic clusters," in *Proc. of the Internet Measurement Workshop*, Nov 2001.