NAME

smacq — System for Modular Analysis and Continuous Queries

DESCRIPTION

The System for Modular Analysis and Continuous Queries (SMACQ) is an extensible component system for analyzing streams of structured data. Users with a familiarity of SQL will immediately be comfortable using the basic features of the system. However, there are additional object-relational and extensibility features that are described below. This manpage describes the query syntax and the modules included with the system by default, which are grouped into sections for I/O, Boolean, Annotation, and Miscellaneous Analysis functions. Queries are executed using SMACQ's command-line utility, **smacqq(1)**, or the SMACQ C++ API.

The primary difference from standard relational databases is that data is not stored in preloaded tables, but is instead produced by data source modules. Also, the select operation does not automatically print fields. If printable output is desired, use the **print** command.

The following example prints the "srcip" and "dstip" fields from a stream of packets stored in a tcpdump-format file named "/tmp/dump":

```
print srcip, dstip from pcapfile("/tmp/dump")
```

Nested queries are also supported. For example:

```
print srcip, dstip from (uniq srcip, dstip from pcapfile("/tmp/dump"))
```

When queries are nested deeply, the syntax described above can become complex. As a result, commas are optional between arguments, and SMACQ also supports the | symbol used in Unix shells. The result is a syntax much more familiar to Unix shell users. Thus the following queries are equivalent:

```
print srcip, dstip from (uniq srcip, dstip from pcapfile("/tmp/dump"))

pcapfile "/tmp/dump" | uniq srcip dstip | print srcip dstip
```

"Where" clauses are supported for both boolean tests based on <, <=, >, >=, =, !=, or arbitrary filtering functions.

```
print dstip from pcapfile("/tmp/dump") where srcip = 128.129.1.2

pcapfile /tmp/dump | where srcip = 128.129.1.2 | print dstip

print dstip from pcapfile("/tmp/dump") where mask(srcip, 128.129.0.0/16)

pcapfile /tmp/dump | mask srcip 128.129.0.0/16 | print dstip
```

Aliasing with "as" is supported: `print −v dstip, sum(len) as totalbytes from pcapfile("/tmp/dump"))`

```
pcapfile /tmp/dump | print −v dstip sum(len) as totalbytes
```

Joins are supported as well:

```
print a.srcip, b.srcip from pcapfile("/tmp/dump") a, b where
```

```
a.srcip != b.srcip
```

When used with streams, joins require very large amounts of memory. The "until" term can be used to define when elements can be removed from the pool of possible values in a join:

```
print a.srcip, b.srcip from pcapfile("/tmp/dump") a until (new.a.ts > a.ts), b where a.srcip != b.srcip
```

Extended relational algebra provides aggregate functions and the "group by" and "having" terms. This syntax is supported, but with slightly different semantics. For example, the following query would behave as it would in SQL:

```
print dstip, sum(len) from pcapfile("/tmp/dump") group by dstip
```

```
pcapfile /tmp/dump | print dstip sum(len) group by dstip
```

but is also the same as:

```
pcapfile /tmp/dump | groupby dstip -- sum len | print dstip sum
```

However the use of a function (such as "sum()") as one of the arguments will result in the "sum" module processing the data before the "print" module. A module called in this way is expected to annotate the data object with a field of the same name as the function. Thus, the sum field will be part of the object from now on. Thus, it can also be used in subsequent arguments. In addition, however, the module can cause other side-effects to the data. Finally, functions can be used whether or not "group by" is used.

SMACQ is an extensible system that the user can add modules to. See the **smacqp**(1) manpage for a detailed description of modules. Many modules take flags which, like all arguments, can be separated with commas or spaces:

```
print -v, srcip, dstip from pcapfile("/tmp/dump")
```

INPUT/OUTPUT FUNCTIONS

Print

print [−x] [−v] [−B] [−d *delimiter*] [fields ...]

Print the specified fields for every record that has them If the **-v** option is given, then warnings are printed when fields aren't present, and field values are preceded by the field name. If **-x** option is given, fields are surrounded with XML-style tags. If the **-B** option is specified, then output is not buffered (output is flushed after each record). Fields are separated by a delimiter string which can be specified by **-d** and defaults to TAB. If no fields are specified, then all fields are printed.

Tabular Input

tabularinput [−d *delimiter*] [−f *filename*] [field [:type] ...]

Read records from STDIN by default, or a filename specified with the **-f** option, one per line, with fields delimited by TAB or an alterate delimiter specified by **-d**. If field names are specified, data columns are assigned those names sequentially. If no field names are specified, or there are more columns that field names, unnamed fields are named numericaly starting at 1.

Field types can be specified by appending a colon and the type name to the end of the field name. If no type is sepecified for a field, it is treated as a double if possible, or a string otherwise.

Packet Capture

pcaplive [−i *interface*] [−s *snaplen*] [−p] [*filter ...*]

The **pcaplive** module reads packets from a network interfaces using libpcap. It can only be used at the beginning of a pipeline. Root privileges are typically required to run this module.

The **-i** option specifies an interface to listen on (default is **any**). The **-s** option specifies the maximum number of bytes per packet to capture (default is 68). The **-p** specifies that the interface should NOT be placed in promiscuous mode.

An optional filter string is a BPF filter string (see **tcpdump**(1)).

Packet Capture File

pcapfile *filename ...*

pcapfile −w *filename* [−l] [−s *megabytes*]

The **pcapfile** module either reads from or writes to a tcpdump-style, libpcap packet trace file. If the module is at the beginning of a pipeline, it reads from a file. Otherwise it writes data to a file.

When reading, one or more files must be specified. Use **-** for stdin. Input files that are compressed with **gzip** are supported automatically. If the **-l** option is specified then files are read from STDIN instead of the arguments.

When writing, a single output file must be specified with **-w**.

The **-s** option specifies the maximum file size (in megabytes) for output files. If specified, the output file will have a two-digit suffix number appended and output will be split between as many files as necessary.

cflowd Raw Flow File

cflow *filename ...* [−l]

The **cflow** module reads from a raw flow file as created by cflowd. One or more files must be specified. Use **-** for stdin. Input files that are compressed with **gzip** are supported automatically. If the **-l** option is specified then files are read from STDIN instead of the arguments.

Socket

socket [−p *port*] [−h *host*] [−d]

The **socket** module is used to send records across the network to another instantiation of the **socket** module. It can be used in two different ways: as a producer who receives data from the network, or as a consumer that writes data to a network. If the module is at the beginning of a pipeline, it is assumed to be a server. Otherwise it is a consumer that writes data to the network.

The **-h** and **-p** options specify a host and port, respectively. The host option is required for a consumer. The default port is 3000.

The **-d** option is only valid in the server context. If specified, the module will continue to accept new connections forever and will never exit. Without this option, the server will accept a single connection, process it until it closes, and then terminate.

BOOLEAN FUNCTIONS

Boolean functions immediately either filter-out or pass-on each data object they are given.

IP Address Mask Lookup

iplookup*field*

The "addr/cidr" argument is a CIDR netmask. An object is filtered out if and only if the specified field does not exist or does not match the given netmask.

Unlike the mask module, this module uses an efficient Patricia Trie to efficiently lookups in large vectors of masks.

IP Address Mask

mask*field* [!]*addr/cidr*

The "addr/cidr" argument is a CIDR netmask. If the mask size is not specified, 32 is assumed. An object is filtered out if and only if the specified field does not exist or does not match the given netmask. If the address begins with a '!', then the logic is reversed and the object is filtered out if the field does match the netmask.

See also the iplookup module.

Substring

substr [*field*] *string* [; *string* ...]

Search for each byte string in the specified field, or in the whole data object if no field is given. If multiple strings are given, then each string corresponds to an output channel, and the object will be output only on the channel(s) that match.

Filter

filter *field [[<=>] value] ... ...*

Filter out all objects in the stream that do not satisfy all of the specified criteria. Expressions can be arbitrarily complex and include AND and OR statements and parentheses for grouping.

This is the select (sigma) operation from relational algebra ("where" in SQL).

Unique Filter

uniq [−m *megabytes*] *fields ...*

Treat the specified field(s) as a tuple and filter out all occurrences of duplicate values of that tuple.

The **-m** option specifies that a probabilistic algorithm using a fixed amount of memory (specified in megabytes) should be employed. Some records may be mistakenly filtered, but some large datasets cannot be processed with a perfect algorithm.

Top

top [−m *megabytes*] [−r *deviation*] *fields ...*

Treat the specified field(s) as a tuple and count the number of occurrences of each values of that tuple. Filter out all records except those whose occurrence deviates from the average by more than a factor of **deviation**. If no **-r** option is specified, the default deviation threshold is 1.

If **-m** is specified, then probabilistic counters are used, consuming a max of **megabytes** memory, at the expense of some records not being filtered even though they’re value is rare.

It is often useful to follow this module with **uniq** in order to get exact counts for all records that pass this filter.

Head

head *number*

Pass the first **number** records through and then end the pipeline. Those records will be processed by all subsequent modules in the pipeline and the program will then terminate.

ANNOTATION FUNCTIONS

An annotation function always adds a field to every data object and the name of that field is identical to the name of the function.

Clock

clock [−t *seconds*] *field*

The clock module is used to bin input data into discrete clock periods. Each object is annotated with a clock field containing the numerical value of the current clock. The current clock value is determined by keeping track of the largest value seen for the specified field (presumably a time) and dividing that value by the optional time period, which defaults to 1. The input is assumed to be sorted in increasing order.

Constant Annotation

const [−t *type*] [−f *field*] *string* [*field*]

Annotate each object with a field containing the specified constant. The default field name is "const" and the default type is "string".

Delta

delta *xfield*

For each data object seen, compute the delta from the previous x field to this current xfield. The data object is annotate with a "**delta**" field of type "**double**" containing the result. The x field must be convertable to doubles as well.

Derivative

derivative *yfield xfield*

For each data object seen, compute the derivative of the y field with respect to the x field between this point and the last object seen. The data object is annotate with a "**derivative**" field of type "**double**" containing the result. The x and y fields must be convertable to doubles as well.

Div

div [−d *divisor*] [*field*]

The div module annotates each object with a field of type "int" and the name "div". The field is computed by dividing the speficied field by the specified divisor (or 1 by default). The result is then truncated. See the "clock" module for similar functionality.

Flow ID

flowid [−t *time*] [−r] *fields ...*

Treat the specified field(s) as a tuple and assign a unique flow id number to each object based on the typle value. The annotated field is called "flowid". All but the first packet will be filtered out.

The **-r** option specifies that the same flow id should be assigned to packets in the reverse direction. Separate flow statistics will be kept for each direction.

The **-t** option specifies a number of seconds idle time before a flow is timed out. When it times out a REFRESH record with the flows identifying fields (as specified in the arguments), the current time (timeseries) and the packet and byte counters ("packets", "packetsback", "bytes", "bytesout") and the "start" and "finish" times.

Now

now [−f *field*]

The now module annotates each object with an object of type "timeval" (a struct timeval) with the given name, or "now" by default.

MISCELLANEOUS ANALYSIS FUNCTIONS

Counter

count [−a] [−f *countname*] [−p] [*fields ...*]

If no fields are specified, simply count the number of records seen. If one or more fields are specified, treat those fields as a tuple and count the number of occurrences of each value for that tuple.

Unless the "−p" flag is specified, then a double value named "probability" is annotated instead. The "−f" flag can still be used to specify an alternate field name.

Normally an annotation is made to only the final object and all other objects are filtered out. However, if the "−a" flag is given, then every object is passed and annotated with a running value.

Deskew

deskew [−s *secondaryfield*] [−b *min*] [−e *max*] [*fields ...*]

The deskew module is designed to take a stream of in-order timeseries data where some of the time values are incorrect and outside a range of possible values. The range is specified with "−b" and "−e". Any such illegal values are replaced with the previous valid value in the stream. This operation is performed on the specified field. If a second field is specified with "−s", then that field is adjusted by an equal amount.

Sort

sort [−r] [−b *batchsize*] [*fields ...*]

Buffer-up the input datastream and output a sorted stream. If "−b" is not specified, no data will be output until the input stream closes. If "−b n" is specified, then sorted data will be output after every n records.

Any fields that are specified that have "double" fields, will be sorted numerically. All other fields will be sorted byte by byte in their native storage format.

The −r option specifies descending order instead of the default, ascending order.

Take

take [*field*]

This module outputs the object specified field of every input object. For example, if an input stream consists of objects with a "timestamp" field (and any number of other fields), and "timestamp" is the specified field name for take, then the output stream will consist of those timestamp fields.

This is similar to the "project" function, and the project operator in relational algebra (select in SQL), except that a single object is returned rather than a tuple of values.

Stateful Matching

dfa *statefile*

The DFA module takes a input file describing transitions in a state machine. Each line contains a current state, a subsequent state, and a predicate for the transition between those states. The predicate is in normal SMACQ syntax for a "where" clause. States named START and STOP are required. All other states can be named with any non-whitespace word.

The DFA module will create multiple instantiations of the given state machine. However, a given input object is used by at most 1 of those instantiations. When the DFA module receives an input object, any existing state machines are checked for possible transitions that would be satisfied by the object. If none of the transitions from the current state of that machine are matched, then that machine will remain in the current state. After a machine does match and transition on an input, no other machines will receive that input. If no existing machines can use the input, then transitions from the START state are checked. If the START state can be left, then a new machine is created.

Last

last [−t *time*] [*fields ...*]

If any fields are specified, treat those fields as a tuple and keep track of the last object seen with that tuple value. After there is no more data, output the objet for each tuple value.

The **-t** option specifies, as a real number, the number of seconds between periodic updates. After the specified amount of time, the last object seen for each tuple value will be emitted (just as is done at the end of the data stream). At the end of the update, an object of type "**refresh**" will be sent with a "**timeseries**" field of type "**timeval**" containing the time. Note: Time is not the wall-clock time, but is instead the time stored in the record in the "**timeseries**" field of type "**timeval**". The **-t** cannot be used with records that do not have this field.

Discrete Probability Density Function

pdf

Assemble a stream of input records with "count" fields. When a "refresh" record is received or the data flow ends, then use the "count" fields to calculate the fraction of the total that each record is responsible. Attach this value as a "probability" field of type "double". calculate then use the

Private Field Namespace

private

Return a new object that shares the same data, but has its own namespace for fields. The namespace is initially the same as the original, but new fields that are added are private to the new copy.

Project

project *fields ...*

Replace all objects in the input stream with new objects containing only the specified fields. This is the project (Pi) operation from relational algebra ("select <fields>" in SQL).

Rename

rename *oldfield newfield ...*

Given a list of alternating old and new field names, make a copy of the old field with the new name. Combined with the Project module, this can implement the rename (rho) operation from relational algebra ("as" in SQL).

Entropy

entropy

This module expects a series of data objects with "**probability**" fields and computes the Shannon entropy for that series. When the data stream ends or a "**refresh**" object is seen, it is assumed that every ocurring value has been seen and the entropy for the series is calculated and added as an annotation of type **double** to a refresh object. See the "**last**" module for more information on **refresh** objects.

Group-By

groupby *fields ...* −− *query ...*

Treat the specified field(s) as a tuple and instantiate the specified query for each tuple. If a record of type "refresh" is received, then the pipeline for that tuple will be gracefully terminated.

Time Sort

fifodelay [−t *time*] [−i *input-time-field*] [−o *output-time-field*]

Sort a series of input records and output them sorted by an output time field that is specified with the **-o** option and defaults to "timeseries". All records that are past the edge time are immediately updated. The edge time is determined by the input time field (specified with the **-i** option and defaullting to "timseries") and a time delay which is specified with the **-t** option which defaults to 0 seconds.

QUERY SYNTAX

SMACQ queries are specified using the following SQL-like grammer:

query:
action from [alias, joins] [WHERE boolean] [GROUP BY args [HAVING boolean]]
| action [WHERE boolean]
| WHERE boolean
| query '|' action [WHERE boolean] [GROUP BY args [HAVING boolean]]

action:
function args
| function (args)
| (query)
| (parenquery + parenquery)

joins:
[parenquery] alias [, joins]

parenquery:
(query)

| function (args)
| function

from:
FROM action [from]

Arguments can be given in a space separated list or a comma separated list. Any argument can be followed by the phrase "AS alias" to be given the specified alias.

argument:
word
| function (args)
| '[' expression ']'

boolean:
(boolean)
| boolean AND boolean
| boolean OR boolean
| NOT boolean
| operand
| subexpression op subexpression
| function (args)

SEE ALSO

smacqq(1), **DTS**(3), **SmacqGraph**(3)