Tigon SQL Users Manual

January 2014

Authored by: The Tigon SQL Team, AT&T Research

Table of Contents

List of Appendixes

No table of figures entries found.

List of Tables

Name	Title	Phone number
Oliver Spatscheck	Lead Member of Technical Staff	(908) 901-2076
Theodore Johnson	Lead Member of Technical Staff	(212) 341-1010
Vladislav Shakepnyuk	Principle Member of Technical Staff	(212) 341-1813
Divesh Srivastava	Director	(908) 901-2077

Table 1 Tigon SQL Team Points of Contact

Document Title	Authors	Last Updated:
Gigascope™*: Building a Network Gigabit Sniffer*	Charles D. Cranor, Yuan Gao, Theodore Johnson, Vladislav Shkapenyuk, Oliver Spatscheck *AT&T Labs — Research*	
Streams, Security and Scalability	Theodore Johnson , S. Muthukrishnan , Oliver Spatscheck, and Divesh Srivastava	
GSQL Users Manual	Theodore Johnson	August, 2014
Sampling Algorithms in a Stream Operator	Theodore Johnson, S. Muthukrishnan, Irina Rosenbaum	16 June 2005

Table 2 Referenced Documents

Font	Indicates
Bold	Device, Field, Host, File or directory names.
Italicized	Introducing a new concept or definition.
‘in single quotes’	Titles, executables, commands, predicates and functions.
`In Courier New`	Interface Set names, and definition examples.
`IN CAPS`	Clause titles

Table 3 Document Conventions

1. Introduction

This manual describes Tigon SQL the SQL component of Tigon. Tigon SQL can be used in conjunction with the Tigon system or in stand-alone mode.

1.1. Background

The phenomenal growth of the Internet has had a tremendous effect on the way people lead their lives. As the Internet becomes more and more ubiquitous it plays an increasingly critical role in society. Indeed, in addition to leisure-time activities such as gaming and Web browsing, the Internet also carries important financial transactions and other types of business communications. Clearly, our dependency on the correct operation and good performance of the Internet is increasing and will continue to do so.

For network operators, understanding the types and volumes of traffic carried on the Internet is fundamental to maintaining its stability, reliability, security, and performance. Having efficient and comprehensive network monitoring systems is the key to achieving this understanding. The process of network monitoring varies in complexity from simple long term collection of link utilization statistics to complicated ad-hoc upper-layer protocol analysis for detecting network intrusions, tuning network performance, and debugging protocols. Unfortunately, rapid Internet growth has not made monitoring the network any easier. In fact three trends associated with this growth present a significant challenge to network operators and the network monitoring tools they use.

This introductory excerpt was taken from the reference paper "Gigascope™: Building a Gigabit Network Sniffer." A review of the rest of this paper is strongly recommended, as it covers extensively the ideas, concepts, and architecture behind the initial Tigon SQL implementation as conceived in 2001.

As described within the context of this document, Tigon SQL is a high-speed stream database, the purpose of which is network monitoring. To provide additional insight into why Tigon SQL was built and what it is primarily used for, it is advisable to read *Streams, Security and Scalability*. This document provides examples of Tigon SQL uses in the security sense, and highlights some of the more advanced Tigon SQL features.

> **Note:** Not all features described in this paper are available in the current public Tigon SQL release. For more information on these features, contact the Tigon SQL team (see Table 1 Tigon SQL Team Points of Contact on page i of this document).

1.2. Theory of Operation

Tigon SQL is a *stream database*, and while its user interface is in many ways similar to that of a conventional stored-table database management system (dbms), its operation is very different. Let us briefly review how a dbms operates.

1.2.1. Stored Table Databases

A dbms typically stores several tables on persistent storage. Each of these tables consists of a collection of records, and each record has a collection of named fields. For example, a database might contain tables FOO and BAR, where FOO has fields (String a, Float b, Int c). These tables might be stored as files in the database server's file system and named FOO.dat and BAR.dat, respectively.

A dbms generally makes use of *indices* to accelerate the performance of queries. For example, table FOO might have an index on field c, named Index_FOO_c and stored as Index_FOO_c.idx in the file system.

Users submit queries to access the data in a dbms. A popular language for expressing queries is SQL. For example, the user might wish to fetch the a and b values of all records in FOO where c=15. The SQL query for this query is

```
Select a, b
From FOO
Where c=15
```

The dbms translates this query into a sequence of steps for evaluating the query (the *query plan*). There are many possible ways for evaluate even this simple query; one possibility is

1. Open the index file Index_FOO_c and determine the record locations in FOO.dat such that c=15.
2. Open FOO.data and, using the result of step 1, fetch the indicated records.
3. Using the result of step 2, format a new record comprised of the fields (a, b).
4. Using the result of step 3, output the records (e.g. to the user's terminal).

Steps 1 through 4 are usually implemented using a pre-written program called an *operator*, that has been parameterized to evaluate the particular processing that is needed. For this simple example, the operators are linked (by the "using the result of" relationship) into a list. The operator graphs of more complex queries will be Directed Acyclic Graphs.

The database administrator (dba) is responsible for loading tables FOO, BAR, and others, into the dbms. The dba defines the *schema* of FOO – its fields, indices, and other properties. The dba then loads data into FOO. Data loading might occur once (for a

static data set), or might occur e.g. daily (for a data warehouse). Data loading might occur continually, as in a transaction processing system, but OLTP systems often limit the scope of permitted data analysis queries to keep them from interfering with transaction processing.

Because the data is loaded, the dbms has the opportunity for collect statistics about its tables, which can be used for query plan optimization. For example, the statistics might indicate that 15 is a very common value of FOO.c, occurring in 50% of the records. In this case, a better query plan than the one above is to fetch all records from FOO and test if c=15 (indexed access has a high overhead as compared to sequential access).

The query evaluation plan has the leisure to *fetch* records from permanent storage. If the server is slow, the query evaluation program fetches records at a slower pace than usual. If the dbms determines that it is low on resources (e.g., memory), it can delay the evaluation of new queries until existing ones terminate.

1.2.2. Stream Databases

New applications, such as network packet monitoring, produce data on a continual basis. The traditional approach to analyzing this kind of data is to gather it, load it onto permanent storage, and then analyze it. However, this kind of processing has a lot of inefficiencies and delays built into it. A more promising approach is to analyze data as it flows past. The data is only touched once, useless data is not gathered, and the results are produced rapidly.

For example, we might be monitoring packets flowing through a router. By using the router's SPAN port, we get a copy of each of these packets and direct them to our streaming analysis server. The packets are read by a *Network Interface Card*, or NIC, and then presented to a system such as Tigon SQL for analysis.

A stream database such as the Tigon SQL structures the raw data sources it receives for the convenience of analysis. The dba defines the *interfaces* that provide data – in this example, the NIC. The dba also defines a schema on the packets that arrive, parsing the data contents for convenient analysis. For example, a network packet will typically have a source IP address, a destination IP address, a time of arrival, and so on.

Suppose that the user wishes to extract the source IP address and timestamp of IPV4 packets such that the destination address is 1.2.3.4. This query can be expressed in an SQL-like language as

```
Select SourceIP, TimeStamp
From IPV4
Where DestIP=IPV4_VAL:'1.2.3.4'
```

The stream database translates this query into a plan for evaluating the query, for example

1. Receive a new record
2. Test if DestIP is 1.2.3.4
3. If so, format a record consisting of (SourceIP, TimeStamp)
4. Output the record

As is the case with the stored procedure database, each of these steps can be performed as separate *pipelined* operators - meaning that one step does not need to run to completion before the next step can start. Step 1 receives records and pipes them to step 2. Step 2 performs its test and pipes the qualifying records to step 3, and so on.

The output of a stream database such as the Tigon SQL is another data stream. A Tigon SQL query set must specify what is to be done with the output: save it to a collection of files, or pipe it to a consuming application.

Tigon SQL uses an SQL-like language, GSQL for specifying its queries – the sample query is valid GSQL. A significant restriction of GSQL is that all queries must have a pipelined evaluation plan. For example, the following query does not have a pipelined plan:

```
Select SourceIP, count(*)
From IPV4
Where DestIP=IPV4_VAL:'1.2.3.4'
Group By SourceIP
```

The "Group By" clause means that the qualifying records should be organized by their SourceIP value, and the query reports the number of records observed for each observed SourceIP value. Since no part of the final answer is known before all of the data has been processed, the query cannot produce piecemeal results.

However, if the input data is partially sorted by a field in the Group By clause, then we can perform pipelined processing. Each packet has a TimeStamp, which is its time of observation. Successive records are have nondecreasing values of the TimeStamp, so the TimeStamp can serve as the sort order. Therefore this query

```
Select SourceIP, TimeStamp, count(*)
From IPV4
Where DestIP=IPV4_VAL:'1.2.3.4'
Group By SourceIP, TimeStamp
```

has a pipelined query plan. Whenever Timestamp increases from say 1:00 pm to 1:01 pm, we know the final value of all records with Timestamp 1:00 pm and earlier. Therefore the query plan can remove all such records from its internal tables and produce them as output. In general, all joins must have a predicate which matches the timestamps of its sources, and every aggregation query must have a timestamp in its list of group-by fields.

Requiring pipelined query plans has three advantages. First, Tigon SQL can continually produce output. Second, internal tables (for joins and aggregations) remain small as they are continually cleaned of obsolete data. Third, queries can be connected together into complex data processing systems because the output of every stream is another data stream, suitable as the input to another query.

For an example, suppose we wish to count the number of distinct SourceIp addresses observed for every tick of the timestamp. Tigon SQL queries are named; suppose the example pipelined aggregation query is named SourceCount. We can use the output of SourceCount as follows

```
Select TimeStamp, count(*)
From SourceCount
Group By TimeStamp
```

A Tigon SQL instantiation generally does not run a single query, it runs a collection of queries, organized into a DAG of communicating processed. The parts of the queries which access data directly from the interfaces are grouped into a single executable called RTS (for run time system). This grouping is done so that data from the interface does not need to be copied, instead each query is invoked on each successive packet. Complex downstream processing is performed in downstream programs, named hfta_[0-9]+. Each program (the RTS or the hfta) executes a large collection of operators, to avoid unnecessary data copying.

A Tigon SQL installation can support a significant amount of parallelization. A typical installation will process from multiple NICs simultaneously; each NIC is represented as an interface, and there is an RTS for each interface. For example, a Tigon SQL installation might process data from two Gigabit Ethernet ports bge0 and bge1. Tigon SQL will contain an RTS for the bge0 interface and the bge1 interface, and will combine the results in downstream processing (the hfta's).

High volume interfaces (e.g. 10 gigabit Ethernet) might produce such large volumes of data that we need to parallelize their RTSs. This parallelization is accomplished using *virtual interfaces*: records from a NIC are hashed to virtual interfaces before entering Tigon SQL. If bge0 and bge1 both have four virtual interfaces, then the Tigon SQL installation will have eight RTS processed running, with the results combined properly in the downstream hfta processing.

Some of the queries in the downstream hfta processing might require significant CPU processing, and therefore it might be necessary to execute multiple copies of an hfta process. Records from the input stream are hashed to copies of the hfta, with partial results getting merged with further downstream processing.

The processing graph for a large parallelized Tigon SQL instantiation might become very large with complex record routing. However Tigon SQL transparently computes the proper query plan.

1.2.3. Hybrid Processing

In many large stream processing installations, the stream of data arrives as periodic chunks of records, generally in the form of data files. For example, a logging application might monitor a web server and collect all URL requests that arrive. These URLs are collected in the server's main memory, and then dumped to a file once per minute. Igon will by default produce a similar sequence of files. These sequences of files form a hybrid stream of continual bulk arrivals.

When Tigon SQL operates in file processing mode, it uses one or more file sequences as its data source. New files are pushed into Tigon SQL processing queue. A Tigon SQL component accesses the files in their order of production, extracts data records, and pushes them into the conventional Tigon SQL stream processing. Except for the bulk arrivals, the processing is the same.

Data records can also be delivered to Tigon SQL though a TCP socket. This style of processing delivers a continual one-at-a-time flow of records to a Tigon SQL query set, but over an explicit TCP connection.

The manner in which data is delivered to a Tigon SQL query set is described by the *interface definitions* for the installation. Tigon SQL will automatically determine how to load data from an interface based on the interface definition.

1.3. Tigon SQL Operation

The version of the Tigon SQL described in this manual is a data stream management system that primarly uses hybrid processing. Data can be delivered to Tigon SQL either through a stream of files, or through a TCP socket connection.

2. Tigon SQL Installation Instructions

The Tigon SQL installation instructions in this document will be sufficient for audiences familiar with UNIX. Users should note, however, that due to contributions from many sources since its initial release, the build process of Tigon SQL has grown considerably.

2.1. Unpacking the Source

The sources should arrive as a single tar ball. To unpack it, type the following:

```
tar -xzf STREAMING.tar.gz
```

This will generate sub directory tigon.

Alternatively, one can access github.

2.2. Compiling the Source

```
cd tigon/tigon-sql/src/main/c
make clean
make
make install
```

2.3. Define an Interface

At this point all required sources have been compiled and installed in the right place. The next step is to define the interface(s) that will be used to connect to data sources. For a more complete discussion of interfaces, please consult Section 4.

To get started, let us assume that you wish to read a file stream consisting of a sequence of files named exampleCsv on machine dwarf9.research.att.com. To define this interface on a machine you first have to add the interface to **tigon/tigon-sql/`cfg/ifres.xml`** as an additional resource. One simple definition would be the following:

```
<Resources>
        <Host Name='localhost'>
                <Interface Name='CSV0'>
                        <Class value='Main'>
                        <InterfaceType value='CSV'/>
                        <CSVSeparator value='|'/>
                        <Filename value='exampleCsv'/>
                        <StartUpDelay value='10'/>
                        <Verbose value='TRUE'/>
                </Interface>
       </Host>
</Resources>
```

After defining the interface, map it to a Tigon SQL interface set for localhost. This is done by placing **`default: Contains[InterfaceType,GDAT] and Equals[Host,'localhost'];`** in the following file: **tigon/tigon-sql/`cfg/localhost.ifq`.**

The Tigon SQL tarball should contain an ifres.xml file with sample interfaces defined for localhost, and a localhost.ifq file with a couple of sample definitions. These sample interfaces include the one illustrated above.

2.4. Define a Schema

A Tigon SQL query needs to be able to parse the records it processes into fields with names and data types. The file tigon/tigon-sql`/cfg/packet_schema.txt` contains these record definitions. The Tigon SQL tarball will contain a tigon/tigon-sql`/cfg/packet_schema.txt` file with some sample record definitions which are used by the sample queries. For more information on defining a record schema, see Section 5.35.

2.5. Completing the Installation

In directory **tigon/tigon-sql/`bin`,** run the following

```
perl parse_cpuinfo.pl > ../cfg/ cpu_info.csv
```

The **tigon/tigon-sql/`cfg/cpu_info.csv`** file now contains a map of the processing cores available on the server. This information is used by the Tigon SQL performance optimizer.

At this point the installation is complete. For instructions on making the first GSQL query operable, please reference Section 4.1-Interface Definition and Use for more details.

3. Tigon SQL Quick Start Guide

3.1. Compiling a Query Set

Query sets are collections of GSQL queries stored in files with the .gsql extensions. These query sets should be stored in subdirectories of tigon/tigon-examples/tigon-sql. For example, one query set included in the distribution is in tigon/tigon-examples/tigon-sql/`CSVEXAMPLE`. It contains two query files (`example.gsql` and `example2.gsql`) which contain one query each (see section 11-Example Queries). The first query reads records defined by the `CSV_EXAMPLE` schema from the `csv` query set, and computes per-second aggregates. The second query reads the library query `csv_example/ex2_src` and computes other per-second aggregates. See Section 4.1.8 for more information about defining and using library queries.

Tigon SQL allows the user to define complex chains of query nodes, perhaps accessing library queries. For performance optimization, many of these query streams are inaccessible, being wrapped up into larger operators or perhaps transformed to accommodate better performance. The file `output_spec.cfg` lists all accessible query outputs, with descriptions of how the output is generated. In the output_spec.cfg file in tigon/tigon-examples/tigon-sql/`CSVEXAMPLE`, `example` and `example2` are accessible while library query `csv_example/ex2_src` is not. See Section 6.4 for more information about the output_spec.cfg file.

To build the Tigon SQL binaries implementing these queries, type the following:

```
cd tigon/tigon-examples/tigon-sql/CSVEXAMPLE
tigon/tigon-sql/bin/buildit.pl
```

For more information on the 'buildit' script, see section 9.1-Automated Build Script.

You will notice multiple configurations, .c, .cc and script files being generated in addition to the Tigon SQL binaries for this query set.

3.2. Starting a Query Set

To start the Tigon SQL implementing the tigon/tigon-examples/tigon-sql/`demos/CSVEXAMPLE` query set, use the auto generated 'runit' script by typing the following:

```
cd tigon/tigon-examples/tigon-sql/CSVEXAMPLE
./runit
```

There will be some debugging output on the screen, and when the prompt returns, all Tigon SQL processes will be up and running. If an error message is received during this process, check for the following discrepancies:

- The interface definition is incorrect.

3.3. Starting a Data Source

The query set needs an input data source. The queries in tigon/tigon-examples/tigon-sql/`CSVEXAMPLE` read from the CSV0 interface, which reads data from file `exampleCsv`. A running Tigon SQL instance will scan for the `exampleCsv` file and if found, will unlink it and then process each record in sequence. An external data mover process can now create a new `exampleCsv` file with new data.

The directory tigon/tigon-examples/tigon-sql/`CSVEXAMPLE` contains a script `gen_feed` which creates a new exampleCsv file once per second. Notice the sequence of actions – first a new file is created from a template using `cp`, and then the file is renamed to `exampleCsv` using `mv`. By renaming the file to the target name only when it is complete, we avoid the problem of Tigon SQL trying to consume a partial file.

The directory tigon/tigon-examples/tigon-sql/`CSVEXAMPLE` contains a script, `runall`, which starts Tigon SQL using `runit`, and then starts the data feed using `gen_feed`.

3.4. Instantiating a Query

The next step is to instantiate a query within the running Tigon SQL. In our example the running Tigon SQL instance contains two possible queries we could instantiate (example and example2). A query can either be instantiated by your own application using our API (see section 8-Gigascope™ API) or by one of the two tools provided (gsprintconsole or gsgdatprint). To use the 'gsprintconsole' tool, one must determine proper the correct Tigon SQL instance (several can be running simultaneously). To do this, execute cat gshub.log:

```
[dpi@dwarf14 CSVEXAMPLE]$ cat gshub.log
127.0.0.1:58398
```

Use the response as a parameter to gsprintconsole as follows:

```
tigon/tigon-sql/bin/gsprintconsole -v 127.0.0.1:58398 default example
```

Tigon-SQL needs a start signal to ensure that all data is read and synchronized. To start processing, run the script

```
tigon/tigon-sql/bin/ bin/start_processing
```

If you have already started gen_feed, or you used runall, you will get an output record once per second.

The output of the query `example` can be redirected to data files using the gsgdatprint application. If you want to save the output of the query `example`, you must use a tool such as gsgdatprint because in `output_spec.cfg`, its output is defined to be a stream.

The `example2` query has an output specification of file in `output_spec.cfg`, so its output is automatically saved in a file, in this case in directory output_dir as specified in `output_spec.cfg`. The query `example2` still needs to be instantiated by a tool such as gsprintconsole. There will be no output because `output_spec.cfg` has does not have any line specifying a stream output for `example2` – but one can be added if desired.

3.5.Stopping a Query Set

To stop all processes of a query set which were started using '`./runit`', execute the autogenerated '`./stopit`' script by typing the following:

```
cd tigon/tigon-examples/tigon-sql/CSVEXAMPLE
./stopit
```

The stopit script does not know about the applications which are accessing Tigon SQL output, nor does it know about any feed-generating processes. It is often convenient to write a script which stops these applications also, as the killexample script in the tigon/tigon-examples/tigon-sql/`CSVEXAMPLE` directory does.

4. Referencing Data

A Tigon SQL query set runs off of one or more streaming data sources. The fundamental data source in Tigon SQL is a file stream, defined as an *Interface*. A particular query might reference data that is output from another query. In particular, a query might reference the output of a library query. We describe these methods of accessing data in this section.

4.1. Interface Definition and Use

4.1.1. Introduction

The data sources for any Tigon SQL query set are one or more *interfaces*. These correspond to data from the outside world – a stream of data files. As discussed in Section 1, data from an interface is interpreted through a *protocol* specification. This still leaves the question of how one specifies which interfaces are available on a host and which interfaces a query reads from. Issues related to interface management and specification are discussed extensively in this section.

4.1.2. Interface Management

In Tigon SQL, a file stream (or other input stream) is represented by an interface. A list of all known interfaces is assumed to be recorded in the `ifres.xml` file, normally located in the tigon/tigon-sql`/cfg` directory. (See section 2.3, “Define an Interface,” for more information on this configuration step). As the extension implies, this file is in XML, with the following schema:

```
Resources
      Host+
            Interface+
                  [Properties]*
```

For example,

```
<Resources>
        <Host Name='localhost'>
                <Interface Name='GDAT0'>
                        <InterfaceType value='GDAT'/>
                        <Filename value='exampleGdat'/>
                        <StartUpDelay value='10'/>
                        <Verbose value='TRUE'/>
                </Interface>
                <Interface Name='CSV0'>
                        <InterfaceType value='CSV'/>
                        <Class value=’Primary’/>
                        <CSVSeparator value='|'/>
```

```
                        <Filename value='exampleCsv'/>
                        <StartUpDelay value='10'/>
                        <Verbose value='TRUE'/>
                        <Source value='ExampleData'/>
                </Interface>
                <Interface Name='CSV0TCP'>
                        <InterfaceType value='CSVTCP'/>
                        <CSVSeparator value='|'/>
                        <TcpPort value='45678'/>
                        <StartUpDelay value='10'/>
                        <Verbose value='TRUE'/>
                </Interface>
        </Host>
</Resources>
```

For the host server, say dwarf14.research.att.com, there are one or more interfaces (e.g., GDAT0). The GDAT0 interface is of type GDAT and is sourced from the file exampleGdat. There is a second interface, CSV0 of type CSV and is sourced from file exampleCsv, and a third CSV0TCP which is of type CSV but is sourced from a TCP connection at TCP port 45678. Some interface properties are required (as noted below), while others are optional (e.g., Class). Note the format of the file. The Host and the Interface take a single parameter Name, while the properties take the single parameter value. The Host parameter becomes the property host, while the Interface parameter becomes the property Name.

In some installations it is convenient to define the interfaces of multiple hosts in a single ifres.xml file. In this case, the Host value property should be the host name. For example <Host value='dwarf14.research.att.com'>. Other installations make use of a single server and do not need the complexity of specific host names. The single-server case can use host name localhost, as is done in the example above, and in the samples provided with Tigon SQL. The buildit scripts in tigon/tigon-sql/bin make use of host name localhost.

The required interface properties are:

- InterfaceType : specifies the nature of the format. GDAT means that the data source is a file stream in GDAT format and CSV means that the source is a file stream in delimited-ascii format.
- Filename : is the name of the file that sources the data for the interface. This may be a relative or an absolute path name.

Optional interface properties are:

- CSVSeparator : is the delimiter for CSV interfaces. The default value is ','.

- StartUpDelay : is a start up delay to ensure that all data consumers have started before data streaming begins. The default value is 0.
- Verbose : if True, triggers informational tracing messages. The default value is False.
- SingleFile : if True, a single file is processed and then the Tigon SQL instance shits down, flushing all results. The processed file is not deleted.
- TcpPort : if a TcpPort property exists, the interface reads records from the tcp port specified as the value. Any FileName property is ignored. The records must be in CSV format. A TcpPort property overrides the Filename property.
- Gshub: If a Gshub property exists, then Tigon SQL will connect to the GShub process to determine the tcp port over which to receive data. Tigon SQL will use the value of the Filename property as the name of the interface when communicating with GShub. Tigon SQL will expect to receive GDAT records over the socket connection.

Additional interface properties can also be specified, and can be helpful for two reasons

- Interface properties can be used to define interface sets (see below)
- Interface properties can be referenced in queries, to help establish the provenance of a record. See Section 4.1.7 for more details on referencing interface properties in GSQL queries.

4.1.3. File Stream Processing

Tigon SQL processes data in a file stream as follows:

1. Tigon SQL waits until it can find and open the file specified by the Filename property.
2. Tigon SQL unlinks the file (removing the directory entry but preserving the now-anonymous file).
3. Tigon SQL processed each record in the file.
4. Tigon SQL closes the file – which deletes the file from the filesystem (assuming no other process has the file open).
5. If SingleFile is false, go to step 1 and repeat.

The normal method of feeding a file stream to Tigon SQL is

1. Collect data in a new file.
2. Wait for the new file to be closed.
3. Wait until the file specified by the Filename property has been unlinked.

4. mv the new file to the filename specified by the Filename
5. go to step 1.

4.1.4. TcpPort Processing

Tigon SQL processes data from a TcpPort stream as follows:

1. Tigon SQL opens the tcp port associated with the interface.
2. Tigon SQL receives CSV formatted records from the port and processes them.

An example of TcpPort processing is in tigon/tigon-examples/tigon-sql/CSVTCPEXAMPLE. Ensure that the interface definition in ifres.xml that the query example.gsql reads from (CSV0) has the following property:

```
<TcpPort value='45678'/>
```

Build the example, and start the processing using the runalls script. The gendata.pl script will open tcp port 45678 and feed data to it.

4.1.5. Interface Sets

GSQL queries must specify interfaces and protocols. Per Section **Error! Reference source not found.**, "**Error! Reference source not found.**," there are two ways to specify the interfaces:

- Specify the particular interface from which to read.
- Use an *interface set*.

Specifying the particular interface from which to read can be explained by the following example: Suppose that we compile the following query on **dwarf9**:

```
SELECT systemTime, uintInPosition1
FROM CSV0.CSV_EXAMPLE
WHERE ullongInPosition2=6
```

GSQL will direct Tigon SQL to read from the CSV0 interface for this query. The GSQL compiler will check the **ifres.xml** to ensure that the CSV0 interface actually exists on the host machine. Attempting to run this query on host **dwarf9** instead of **dwarf14** will likely result in an error since CSV0 interface might not be defined on that host.

Using an *interface set* can be explained by the following example: Suppose that the `exl` interface set consists of all interfaces with a Source property of '`ExampleData`'. Then the following query reads from these interfaces:

```
SELECT systemTime, uintInPosition1
FROM [exl].CSV_EXAMPLE
```

```
WHERE ullongInPosition2=6
```

If this query is run on dwarf9, it would read from, e.g., CSV0. If it is run on dwarf14, it would read from e.g., GDAT0. It is possible that an interface set might contain more than one interface. For example, suppose that the 'csv' interface set consists of all interfaces with an `InterfaceType` property of CSV. If we run the following query on dwarf9:

```
SELECT systemTime, uintInPosition1
FROM [csv].CSV_EXAMPLE
WHERE protocol=6
```

It will read from CSV interfaces – say CSV0 and CSV1. (Tigon SQL will handle the details of merging the data from the interfaces into a single stream). If this query is run on dwarf14, it will read only from interface CSV0.

If you read from a protocol but do not specify any interfaces, GSQL assumes that the query reads from the interface set 'default'. The following query,

```
SELECT systemTime, uintInPosition1
FROM CSV_EXAMPLE
WHERE protocol=6
```

is equivalent to

```
SELECT systemTime, uintInPosition1
FROM [default].CSV_EXAMPLE
WHERE protocol=6
```

4.1.6. Defining Interface Sets

The file **<host>.ifq** contains the specification of the interface sets for that host. If running in a single-host mode (which is the default), the file is localhost.ifq. For multi-server configurations, substitute the host name for <host>. For example, the file **dwarf9.research.att.com.ifq** contains the interface set specifications for host **dwarf9.research.att.com**. These files are normally kept in the tigon/tigon-sql/**cfg** directory.

An ifq file contains interface set specifications in the format of the interface set name followed by a colon, then the defining predicate, with a semicolon separating the specifications, as in the example below:

```
[interface set name 1] : [predicate 1] ;
[interface set name n-1] : [predicate n-1] ;
[interface set name n] : [predicate n]
```

The interface specification uses a simple language for its predicates, consisting of the Boolean connectives AND, OR, and NOT, parentheses to force evaluation order, and the following three predicates:

- `Contains[ <property> , <value> ]` : Evaluates true if one of the values of <property> is <value>, else false.
- `Equals[ <property>, <value> ]` : true if there is only one value of <property>, and it is <value>, else false.
- `Exists[ <property> ]` : true if the property is defined, with any value, otherwise false.

Both the property and the value must be strings (with the single quote 'delimiter') or names (alphanumeric, starting with a letter with no spaces or punctuation). As with other GSQL files, both '**--**' and '//' are comment indicators.

Let's consider an example. Suppose that **localhost.ifq** contains the following:

```
default : Equals[Host,'dwarf9'];
csv : Contains[InterfaceType,CSV] and Equals[Host,'dwarf9'];
gdat: Contains[InterfaceType,GDAT] and Equals[Host,'dwarf9'];
other: Contains[InterfaceType,FooBar] and Equals[Host,'dwarf9'];
```

The predicate '`Equals[Host,'dwarf9']`' for the interface set default is redundant; it is assumed for all interface sets in the **dwarf9.ifq** file. The '`default`' interface set contains (GDAT0, CSV0), the '`csv`' interface set contains (CSV0), and the '`gdat`' interface set contains (GDAT0). The '`other`' interface set is empty. Using it will result in an error.

4.1.7. Using Interface Properties in Queries

The values of the interface properties can be referenced in a query. This is useful for determining a packet's origin and its properties. We can reference an interface property in a query by preceding it with a '@' symbol. For example,

```
SELECT systemTime, uintInPosition1, @Name
FROM CSV_EXAMPLE
WHERE ullongInPosition2=6
```

This query will return the name of the interface (GDAT0 or CSV0) that the packet came from, along with its timestamp and source IP.

Some issues with using interface properties in queries are as follows:

- 'Name' must be used to reference the interface name, and 'Host' to reference the interface host.
- The data type of an interface property is always a string.
- A reference to an interface property must be bound (implicitly or explicitly) to a protocol source.

- For every interface in the interface set, the property must be defined, and defined only once.

The code generation system replaces interface properties by their corresponding constants in the low-level (LFTA) queries (which execute on data sent by specific interfaces).

4.1.8. Library Queries

As sample query `example2.gsql` in tigon/tigon-examples/tigon-sql/`CSVEXAMPLE` demonstrates, a Tigon SQL query can use the output of another Tigon SQL query as its data source. Furthermore, the source query can be a *library* query. In example2.gsql

```
DEFINE {
query_name 'example2';
}
select systemTime,uintInPosition1,sum(Cnt)
from csv_example/ex2_src
group by systemTime,uintInPosition1
```

the data source is csv_example/ex2_src. The GSQL compiler interprets a data source with a slash character '/' as a reference to a library query. Library queries are stored in the directory tigon/tigon-sql/qlib. The GSQL compiler will search directory tigon/tigon-sql/qlib/csv_example for a file named ex2_src.gsql, and will try to extract a query named ex2_src (by default, the first query in a file is named by the file prefix). Library queries can reference other library queries, but ultimately the data sources must resolve to interfaces.

5. Protocols and Interfaces

One of the basic functions of the Tigon SQL runtime is to intercept data records on *interfaces* and present them to LFTAs. In some installations, these data records correspond to network data packets (e.g. IPV6 packets), but in other installations they correspond to formatted data records received from a data feed. The resource specification and software component that defines these data records is called a *protocol* (from Tigon SQL roots in processing network packets). One of the primary services of a protocol is to provide a collection of functions for interpreting the contents of a packet. For example, a protocol for IPV4 packets would have a function 'get_ipv4_dest_ip' returning an unsigned integer representing the destination IP address.

GSQL uses a *protocol schema* to allow named access to the values returned by these functions. Let's look at an example:

```
PROTOCOL base{
  uint systemTime get_system_time
      (required,increasing, snap_len 0);
}

PROTOCOL CSV_EXAMPLE (base) {
  uint uintInPosition1  get_csv_uint_pos1;
  ullong ullongInPosition2 get_csv_ullong_pos2;
  IP ipInPosition3 get_csv_ip_pos3;
  IPV6 ipv6InPosition4 get_csv_ipv6_pos4;
  string stringInPosition5 get_csv_string_pos5;
  bool boolInPosition6 get_csv_bool_pos6;
  int intInPosition7 get_csv_int_pos7;
  llong llongInPosition8 get_csv_llong_pos8;
  float floatInPosition9 get_csv_float_pos9;
}


PROTOCOL CSV_TCPEXAMPLE (base) {
  uint ivalue get_csv_uint_pos1 (increasing);
  uint value get_csv_uint_pos2;
}

PROTOCOL GDAT_EXAMPLE (base) {
  uint uintOldTime get_gdat_uint_pos1;
  uint uintInPosition1  get_gdat_uint_pos2;
  ullong ullongInPosition2 get_gdat_ullong_pos3;
  IP ipInPosition3 get_gdat_ip_pos4;
  IPV6 ipv6InPosition4 get_gdat_ipv6_pos5;
  string stringInPosition5 get_gdat_string_pos6;
  bool boolInPosition6 get_gdat_bool_pos7;
  int intInPosition7 get_gdat_int_pos8;
  llong llongInPosition8 get_gdat_llong_pos9;
  float floatInPosition9 get_gdat_float_pos10;
}
```

The schema for `CSV_EXAMPLE` has nine explicit fields, uintInPosition1 through floatInPosition9. The field named uintInPosition1 has a uint (unsigned 32-bit integer) data type, and is extracted from a source record using the built-in extraction function get_csv_uint_pos1.

Network protocols tend to be layered (e.g., an IPV4 packet is delivered via an Ethernet link). As a convenience, the protocol schemas have a mechanism for field inheritance. In this schema, CSV_EXAMPLE is layered on top of base, which has a single field systemTime. In total, CSV_EXAMPLE has ten fields. A protocol schema can inherit from any number of other protocol schemas, subject to the following two restrictions:

- There must be no cycles in the inheritance.
- No field name can appear twice in any protocol schema, whether directly listed or through inheritance.

The systemTime field has several field attributes associated with it, in parentheses (see Table 5 for a list of recognized attributes).

The following is a list of data types. Some of these data types might not be available at the LFTA level (due to resource limitations in specialized hardware). The IP type stores an IPv4 address in an unsigned integer, but its annotation as an IPv4 address allows access to IPv4 functions and specialized printing routines.

Type name	Data type
bool Bool BOOL	Boolean
ushort Ushort USHORT	Unsigned short integer
uint Uint UINT	Unsigned integer
IP	Unsigned integer
IPV6 IPv6	IPV6
int Int INT	Signed integer
ullong Ullong ULLONG	Unsigned long long
llong Llong LLONG	Signed long long
float Float FLOAT	Floating point (double)
string String STRING v_str V_str V_STR	String (variable length)

Table 4: Data Type Names

The field attributes give hints to GSQL about the meaning of the field, typically used for optimizations. The `increasing` and `decreasing` attributes tell GSQL about ordering properties of the stream. This information is critical for the correct operation of the queries. Fields which are marked `increasing` or `decreasing` are called *temporal fields*. The following is a list of recognized field attributes:

Attribute name	Meaning
Required	Field is present in every data packet
increasing Increasing INCREASING	Field value never decreases
Decreasing Decreasing DECREASING	Field value never increases
snap_len	The parameter of the attribute is the maximum length of the packet that must be searched to find the attribute (for optimizations).
subtype	An attribute that can be associated with the type. The subtype is not used by GSQL, but it is reported in the output schema.

Table 5: Field Attributes

5.1. Group Unpacking

Tigon SQL operates at extremely high data rates; consequently, the cost of extracting fields from packets is a serious optimization concern. Our experience in writing optimized field unpacking code has led us to observe that often it is cheaper to unpack a group of fields at the same time than it is to unpack each field individually. For example, to extract a field, (e.g. the sequence number) in a TCP header, the unpacking function must navigate several layers of network protocols, and only then can the sequence number be extracted.

If several TCP fields are needed by queries in the query set, then field unpacking is more efficient if all the TCP fields are extracted at once. If only the sequence number is needed, then extracting it alone is more efficient.

To support the automatic optimization of field extraction, Tigon SQL supports *extraction functions* in packet_schema.txt. Each unpacking function has a name, a function that is called to perform the unpacking, and a cost. An example of an unpacking function declaration is as follows:

```
UNPACK_FCNS{
```

```
    unpack_TCP unpack_tcp_group 2;
    foo unpack_foo_group 1;
    bar unpack_bar_group 1
}
```

Here, in the example above, three unpacking functions are declared:

- unpack_tcp, with unpacking function unpack_tcp_group and cost 2
- bar, with unpacking function unpack_bar_group, and cost 1.
- foo, with unpacking function unpack_foo_group, and cost 1.

The fields that a group unpacking function unpacks are indicated by a comma-separated list of the names of the unpacking functions which unpack a field, in square brackets and at the end of the field declaration. As a convenience, a PROTOCOL can be marked in the same manner with the names of the unpacking functions which unpack all fields in the PROTOCOL. For example,

```
PROTOCOL TCP (DataProtocol) [unpack_TCP]{
    uint sequence_number get_tcp_sequence_number (snap_len 138) [foo];
    uint ack_number get_tcp_ack_number (snap_len 138) [foo, bar];
    bool urgent get_tcp_urgent_flag (snap_len 138);
            …
```

The field sequence_number can be unpacked by unpack_TCP and foo; ack_number can be unpacked by unpack_TCP, foo, and bar; and urgent can be unpacked by unpack_TCP.

In the lfta code, the prototype for an unpack function is as follows:

```
void unpack_function(void *)
```

It is called with a pointer to the packet data. An example of the generated code is as follows:

```
unpack_tcp_group(p);
```

If a field does not have an unpacking function, the field access function (e.g., get_tcp_sequence_number for sequence_number) must also act as its unpacking function. If a field does have an unpacking function, the field access function is still used within the query processing code.

5.2. Built-in Field Access Functions

The purpose of a protocol specification is to define field names and data types in records, and to specify the function used to extract a particular field from a record. In some applications (e.g., network monitoring), these functions are highly specialized and need to be carefully written to extract bit-packed data. However, many data sources provide

data in a standard and regular format, so that field access functions can be pre-generated and provided with the Tigon SQL run time.

Tigon SQL provides a large collection of field access functions to assist with developing a custom Protocol. The name of one of these access functions is highly structured, reflecting its action. The format of a built-in field access function name is

```
get_<record_format>_<data_type>_pos<field_position>
```

where

- record_format is either csv or gdat. A csv record is better described as delimited ascii; the record is ascii text with a '\n' record delimiter. Fields are separated by a field delimiter character, which is by default a comma ',', but can be overridden by the `CSVSeparator` property in the Interface specification (see Section 4.1.24.1). A gdat record is a binary record encoded using Tigon SQL default format (both gsgdatprint and a file output specification in output_spec.cfg produce gdat files).
- data_type is one of (uint, ullong, ip, ipv6, string, bool, int, llong, float) and corresponds to the data types listed in Table 4: Data Type NamesTable 4.
- field_position is the position of the field in the record, starting with position 1. Postion 0 is always occupied by systemTime.

For example,

get_csv_string_pos3

extracts the 3rd field of a csv record and interprets the value as a string.

These functions can be found in tigon/tigon-sql/include/lfta/csv_macro.h and tigon/tigon-sql/include/lfta/gdat_macro.h. For each record format and each data type, there is a field access function for field positions 1 through 1000. If more access functions are needed, the scripts gencsvinclude.pl and gengdatinclude.pl generate these functions (actually, macros).

The gdat access functions access records which are packed using the internal Tigon SQL record format, which by convention us the .gdat filename extension.

The csv access functions provide limited field parsing functionality. In particular:

- Bool: the string TRUE (all upper case) is interpreted as true, all other values are interpreted as false.
- Short, Unsigned short : is not supported.
- Uint: read with sscanf using %u
- Int: %d

- Ullong : %llu
- Llong: %lld
- Float : %lf
- Ipv4 : %u.%u.%u.%u
- Ipv6 : %x:%x:%x:%x:%x:%x:%x:%x
- String : the text between the delimiters is used as the string value.

An additional built-in field access function is `get_system_time` function, which does not access any data in the record, but instead samples the current unix time by calling time().

5.3. Writing a Protocol

The example Protocols provided with Tigon SQL and listed in Section 5 are somewhat artificial. Let's examine a more realistic one. Suppose that we need to ingest a data feed in csv format with fields (dest_ip, bytes_transferred, url) with data types (ipv6, ullong, string) respectively.

```
PROTOCOL User_Downloads (base) {
  IPV6 dest_ip  get_csv_ipv6_pos1;
  ullong bytes_transferred get_csv_ullong_pos2;
  string url get_csv_string_pos3;
}
```

Inheriting from base gives access to systemTime. The remaining entries are a straight-forward translation of the record format specification into a Protocol specification.

5.4. Interfaces

If a GSQL query reads data using a protocol, it must read the data from a particular *interface* -- that is, from a particular source of data. For a full discussion, please see Section 4.1-Interface Definition and Use. See also the discussion of the FROM clause in Section 6.2.2 below.

6. GSQL

One of the important features of Tigon SQL is the ability to specify packet monitoring programs in a high-level, SQL-like language (GSQL). Tigon SQL transforms a GSQL query into C-language and C++ language modules which are integrated into the Tigon SQL run time. This manual documents the tools used to generate FTA modules and supporting code.

This document discusses the GSQL language features. See Section 6-Example Queries and Section 7.2-Writing your own UDAFS.

6.1. Background

Tigon SQL is a stream query processing system. Data from an external source arrives in the form of packets at one or more *interfaces* that Tigon SQL monitors. These records are interpreted by a *protocol* and used as input for the queries which are running on the interface. Each of these queries creates a *stream* of *records* as its output. The stream may be read by an application, or by another query (which in turn creates its own output stream).

A query that reads from a protocol is a *Low-level query*, or *LFTA*, while a query which reads only from streams is a *high-level query*, or *HFTA*. One reason for distinguishing between LFTA and HFTA queries is to ensure high performance. All LFTA queries are compiled into the Tigon SQL run-time system, thus allowing external packets to be presented to the LFTA queries with minimal overhead. In some cases, the LFTAs might execute even in part or in whole on the Network Interface Card (NIC). Because the LFTAs are compiled into the runtime system, the set of available LFTAs cannot be changed without stopping and re-linking the runtime. HFTAs execute as independent processes, and can be added on the fly. Another reason for the distinction between LFTAs and HFTAs is that LFTAs read data from *interfaces* (see Section 4.1), and therefore requires some special runtime support.

The distinction between HFTA and LFTA is almost transparent to the user. In some cases, the GSQL optimizer will break a user's query into an HFTA component and one or more LFTA components. In other cases, the user's query executes entirely as an LFTA. In either case, the query output is accessed using the query name.

The output of a query is always a stream, which is a sequence of records (sometimes called *tuples*) that are delivered to the application. This output is very different than what a conventional DBMS produces. First, the records are *pushed* to the application rather than the application *pulling* them (e.g., there is no cursor). There are some methods by which the application can tickle the query system to obtain some records (for use in some special cases). Second, the size of the output is roughly proportional to the length of time

that the stream is monitored, rather than being a fixed size. The records in the stream are usually in a sorted order on one or more of its fields.

6.2. Query Language

Queries are specified using a SQL-like language. Several different query types are possible, but they reuse some common language components.

6.2.1. Selection Queries

A selection query filters out a subset of the tuples of its input stream, computes a transformed version of these tuples, and outputs the transformed tuples in its output stream (the Filtering and Transformation of an FTA). Consider the following example:

```
SELECT  sourceIP, destIP
FROM IPV4
WHERE protocol=1
```

The data source is a stream derived from an IPV4 protocol. The 'filter' predicate is the condition `protocol=1` (i.e., the ICMP protocol). The transformation is to output only the source and destination IP addresses.

The meaning of a very simple query such as this is pretty clear even to a novice user (however, more than 150 lines of C code are generated from it). All three of the components of a selection query can have a more complex structure, however.

6.2.2. FROM Clause:

The FROM clause specifies the data source(s) of the query. In general, its format is as follows:

```
FROM table1 [tablevar1], …, tablen [tablevarn]
```

That is, the argument of the FROM clause is a comma separated list of table names and the corresponding table variables. The *table variable* represents the item being queried, as shown in the FROM clause below:

```
FROM IPV4 I, PacketSum P
```

There are two table variables, I and P. Every field reference is to one of these two table variables (e.g. `I.protocol` or `P.bytecount`). It is possible for two different table variables to be drawn from the same source:

```
FROM PacketSum P1, PacketSum P2
```

This construction allows us to compare tuples in **`PacketSum`** to other tuples in **`PacketSum`**.

At the risk of some confusion, GSQL allows you to drop references to the table variable as long as it can deduce which table variable supplies each of the fields referenced in the query. If table variables are not supplied, GSQL will impute them. For example, the selection query at the start of this section is translated into the following:

```
SELECT _t0.sourceIP, _t0.destIP
FROM IPV4 _t0
WHERE _t0.protocol=1
```

If the table variable of a field reference is ambiguous, GSQL will reject the query with an error message. If the query references more than one table source, it is best to use explicit table variables to eliminate possible errors due to unexpected table variable imputation. While all selection queries use only a single table variable, other query types such as stream merge and stream join use multiple data sources.

6.2.3. Scalar Expressions

Every reference to a particular value is through a scalar expression. The query at the start of this section references the following three scalar expressions:

- `sourceIP`
- `destIP`
- `protocol`

All three of these scalar expressions are simple field references. More complex scalar expressions can be built using the following rules:

1. **Literals:** A literal is an expression which has a constant value known at compile time. Examples include `3`, `4.0`, and `FALSE`. The data type of the literal is determined by the form of the literal. A list of literals accepted by GSQL is shown in Table 6: Literals
2. **Defined Literals:** A literal can take its value from the DEFINE block (see section 5.5.2-Options). The value of a defined name can be referenced by a hash mark (#) in front of the name. For example, if the define block contains the line **`foo'bar';`**, then a reference to **`#foo`** is equivalent to **`'bar'`** in the query. Defined literals always have the string data type. This feature is useful if you need to reference the same literal several times in a query.
3. **Query parameters:** A query parameter is a value which does not depend on the contents of any data stream, but is supplied at the start of the query and can be changed as the query executes. A query parameter is indicated by a dollar sign ($) followed by a symbolic name, e.g. `$MinCount`. The data

type of the query parameter is declared in the PARAM declaration of the query. We discuss query parameters in greater detail in a later section.

4. **Interface properties:** An interface property is a literal which takes its value at the LFTA level from the defined properties of the interface it reads from. An interface property is indicated by an 'at' symbol (@) followed by a symbolic name, e.g. `@Dir`. For a more detailed discussion of interface properties, see Section 4.1-Interface Definition and Use.

5. **Field references:** A field reference takes its value from a tuple field in a table variable. All field references must be associated with a table variable. However, explicit table variable references can be dropped (for convenience in simple queries) and GSQL will attempt to impute table variables. As is noted in Section 6.2.2, "**FROM** Clause:," it is best to use explicit table variables if there is any possibility of confusion. The table variable imputation mechanism allows three ways to specify a field reference:

   a) `TableVariable.FieldName` : The table variable is explicit. For example, `_t0.destIP`.

   b) `TableName.FieldName` : The table variable is imputed to be the one bound to a table of the same name. If there is more than one table variable bound to this table, the imputation fails and the query is rejected. An example of this kind of field reference is `IPV4.destIP` or `eth0.IPV4.destIP`.

   c) FieldName : The table variable is identified as the one bound to the table which contains a field of the same name. If no such table is listed in the FROM clause, or if more than one table variable can be matched, the imputation fails and the query is rejected. An example of this kind of field reference is `destIP`.

6. **GROUP-BY variable reference:** A GROUP-BY variable is a component of the entity which defines a group. GROUP-BY variables are discussed in greater detail in the section on GROUP-BY/aggregation queries. We note here that GSQL tries to impute a field to be a GROUP-BY variable before it tries to impute a table variable for the field.

7. **Unary operators:** A unary operator computes a value from the value of a single sub expression. For example, `~destIP`.

8. **Binary operators:** A binary operator computes a value from the value of two sub expressions. For example, `total_length-offset`.

9. **Functions:** A function takes zero or more parameters and returns another value. The file **external_fcns.def**, which normally resides in the tigon/

tigon-sql/**cfg** directory, defines the set of available functions and their prototypes. The data types of the function parameters must exactly match the function prototype; no type escalation is performed. However, functions can be overloaded. We also note that some of the function parameters must be expressions involving only literals and query parameters (i.e., no field references or GROUP-BY variable references). Functions are discussed in greater length in Section 11 (External Functions). An example of a function call is: '`PACK(sourceIP,source_port)`'.

10. **Aggregate functions:** An aggregate function is syntactically similar to a regular function, but it is evaluated over a set of input values rather than over a single value. See section 5.4.10 GROUP-BY/Aggregation for more discussion.

Every scalar expression (and each of its sub expressions) has a data type. As mentioned above, the parameters of a function call must match its prototype. Because operators are really overloaded functions expressed using infix notation, the parameters must match some prototype of the operator. The rules for type imputation are shown below in Table 6: Literals

Data Type	Regular Expression	Example
Unsigned integer	([0-9]+)|([0-9]+UL)	35, 17UL
Unsigned integer (hex)	HEX' [0-9A-Fa-f]+ '	HEX'7fff'
IP	IP_VAL'[0-9]{1-3}.[0-9]{1-3}.[0-9]{1-3}.[0-9]{1-3}'	IP_VAL'135.207.26.120'
IPv6	IPV6_VAL' ([0-9abdcef]{1-4}.){7}([0-9abdcef]{1-4}'	IPV6_VAL'0000.2222.4444.6666. 8888.aaaa.cccc.eeee'
Unsigned long long	[0-9]+ULL	1000000000000ULL
Unsigned long long (hex)	LHEX' [0-9A-Fa-f]+ '	LHEX'7abcdef012'
Float (double precision)	[0-9]+”.”[0-9]*	35.0
Boolean	TRUE|FALSE	TRUE FALSE
String	'[^'\n]*'	'foobar'

Table 6: Literals

Operator	Left Operand	Right Operand	Result
! (logical negation)	int, uint, ushort, ullong llong, bool		Same as operand.
~ (bitwise negation)	int, uint, ushort, ullong llong		same as operand
- (unary minus)	Int, uint, ushort, ullong, llong, float		Same as operand
+ (plus)	Int, uint, ushort, ullong, llong, float	Int, uint, ushort, ullong, llong, float	Same as larger operand
- (minus)	Int, uint, ushort, ullong, llong, float	Int, uint, ushort, llong, float	Same as larger operand.
- (minus)	Int, uint, ushort, ullong, llong, float	Ullong	Llong
* (multiply)	Int, uint, ushort, float	Int, uint, ushort, float	Same as larger operand.
/ (divide)	Int, uint, ushort, float	Int, uint, ushort, float	Same as larger operand.
| (bitwise or)	Int, uint, ushort, ullong, llong, IP	Int, uint, ushort, ullong, llong, IP	Same as larger operator
| (bitwise or)	IPV6	IPV6	IPV6
| (bitwise or)	bool	bool	bool
& (bitwise and)	Int, uint, ushort, ullong, llong, IP	Int, uint, ushort, ullong, llong, IP	Same as larger operator
& (bitwise and)	IPV6	IPV6	IPV6
& (bitwise and)	bool	bool	bool
>> (shift right)	int, uint, int, ullong, llong	Int, uint, int	Same as larger operand

<< (shift left)	int, uint, int, ullong, llong	Int, uint, int	Same as larger operand

Table 7: Operator Prototypes

In addition to a conventional data type, each value has a *temporal* type, which indicates how the value changes in successive packets in its data stream. Possible values of the temporal type are constant (never changes), increasing (never decreases), decreasing (never increases) and varying (no information). Table 8, "Temporal Type Imputation," shows the rules for temporal type imputation (rules resulting in varying are not listed).

Operator	Left Operand	Right Operand	Result
+, *	Increasing	Constant, increasing	increasing
+, *	Decreasing	Constant, decreasing	Decreasing
+, *	Constant	Constant, increasing, decreasing	Same as right operand
-	Constant	Constant	Constant
-	Constant, decreasing	Increasing	decreasing
-	Constant, increasing	Decreasing	increasing
/	Constant, decreasing	Constant, increasing	Decreasing
/	Constant, increasing	Constant, decreasing	Increasing
/	Constant	Constant	Constant

Table 8: Temporal Type Imputation

6.2.4. Selection List

A selection list is a list of (optionally named) scalar expressions following the SELECT clause. The output of the query is exactly this selection list, in the order listed. Each entry in the selection list is a field in an output stream, and is named, either implicitly or explicitly, using the AS clause. For example,

Scalar_expression as name

If the field is not explicitly named, a name is pre-populated for it. For instance, if the scalar expression is a field reference, then the output field name is the same as the input field name. Aggregates of field references are formed by *aggregate_field*, e.g., SUM_len for SUM(len). In other cases, the field name is not so obvious. For example, the following produces tuples with output fields Field0 and *MaskedDestIP:*

```
    Select sourceIP & IP_VAL'255.255.255.0', destIP &
IP_VAL'255.255.255.0' AS MaskedDestIP
```

6.2.5. Predicate Expressions

A predicate represents a true or false value used for the purpose of filtering records (not to be confused with the TRUE and FALSE values of a Boolean type). An *atomic* predicate is a source of a true/false value, similar to a field reference in a scalar expression. The three types of atomic predicates are as follows:

- **Comparison:** A comparison predicate has the form *scalar_expression relop scalar_expression*, where *relop* is one of {=, <>, <, >, <=, >=}. The two scalar expressions must have comparable types. Any pair of numeric types is comparable, while Boolean, string, and timeval can only be compared to each other. This type of predicate has the obvious meaning. An example is '`protocol=1`'.
- **IN:** An IN predicate is of the form *scalar_expression* `IN [` *literal_list* `]`. This predicate evaluates to true if the value of the scalar expression is one of the values in the literal list. The type of the scalar expression must be the same as the types of all of the literals. An example is

  '`source_port IN [80, 8000, 8080 ]`'.
- **Predicate Function:** A predicate function has syntax similar to that of a regular function, except that the parameters are enclosed in square braces '[]' and its return value is used as a predicate. The prototype of the predicate function is defined in **external_fcns.def**. An example is '`http_port[source_port]`'.

Atomic predicates can be combined into predicate expressions using the connectors `AND`, `OR`, and `NOT`, which have the expected meaning. An example is '`protocol=4 AND http_port[source_port]` '.

6.2.6. WHERE clause:

The predicate of the WHERE clause is used to filter the tuples of the input stream. If the tuple does not satisfy the predicate, it is discarded.

6.2.7. Comments

A comment can appear anywhere in a query, schema definition, or external function definition. All text following two dash characters "--" or two slash characters "//" is ignored until the end of line. For example,

```
//      A simple query
SELECT  sourceIP, destIP   // just source and dest
FROM IPV4
```

```
        WHERE protocol=6                    -- All TCP
```

6.2.8. Selection Summary:

The syntax of a selection query is as follows:

```
SELECT list_of_select_expressions
FROM table
[WHERE predicate]
```

That is, the WHERE clause is optional. Recall the transformed version of the query:

```
SELECT  _t0.sourceIP, _t0.destIP
FROM eth0.IPV4 _t0
WHERE _t0.protocol=1
```

The meaning of the selection query is this:

- Monitor the packets from eth0.
- Interpret the packets using the IPV4 protocol.
- If the value of the protocol field is 1, marshal the sourceIP and destIP fields into a tuple and put the tuple into the output stream.

6.2.9. Join

A *join* query is similar to a selection query, except that in a join query there can be more than one table in the FROM clause. A join query will emit a tuple for every pair of tuples from its sources which satisfy the predicate. This can be a very large number of tuples, but normally a correctly written join query will filter out almost all of the pairs. An example of a join query is as follows:

```
SELECT R.sourceIP, R.destIP, R.tb, R.length_sum,S.length_sum
OUTER_JOIN from Inpackets R, Outpackets S
WHERE R.sourceIP = S.destIP and R.destIP = S.sourceIP and
R.tb = S.tb
```

This query associates aggregate measurements from an in-link and an out-link to create a combined report.

Notice the keyword OUTER_JOIN in front of the FROM clause. There are two types of join semantics:

- *inner join,* which generally conducts filtering
- *outer join*, which generally conducts report generation

6.2.9.1. INNER_JOIN

An inner join will create a tuple only when it can match a pair of tuples from R and S. Unmatched tuples are discarded (and hence inner join does filtering). An example query which uses inner join is the following, which computes the delay between a syn and a synack:

```
SELECT S.tb, S.sourceIP, S.destIP,S.source_port, S.dest_port
(R.timestamp-S.timestamp)
INNER_JOIN from tcp_syn_ack R, tcp_syn S
WHERE S.sourceIP=R.destIP and S.destIP=R.sourceIP and
      S.source_port=R.dest_port and S.dest_port=R.source_port
AND
      S.tb=R.tb and S.timestamp<=R.timestamp
```

6.2.9.2. OUTER_JOIN

An outer join will create an output tuple for every matched pair from R and S, and in addition it will create an output tuple for each *unmatched* tuple from both R and S. The reporting example above is a typical use of an outer join. If there are Inpackets but no Outpackets (or vice versa) for a particular sourceIP, destIP pair, there still needs to be a report about the Inpacket traffic (or Outpacket traffic). The query will generate a meaningful output tuple in spite of the missing information, but there are some subtleties to be aware of.

1. The SELECT clause will probably contain scalar expressions that reference both R and S. When an output tuple is generated for an unmatched input tuple, there will be missing information. If creating an output tuple for an unmatched R tuple, use default values for fields from S (and vice versa). Integers become 0, floats become 0.0, and strings become a empty ('').
2. There is an exception to the above rule. If there is a predicate in the WHERE clause of the form *R.fieldr=S.fields*, the value R.fieldr is substituted for S.fields when processing an unmatched tuple from R, and conversely for S. This is the only condition under which GSQL will infer non-default values for the missing field values. The motivation is that equality predicates on fields identify the join key, and you are gathering a report about that key. The source, R or S, is irrelevant, although you are forced to choose one when writing the SELECT clause.

6.2.9.3. LEFT_OUTER_JOIN

The LEFT_OUTER_JOIN is like an outer join, except that only the matched pairs and the unmatched tuples from R will generate an output tuple. Unmatched tuples from S are discarded.

6.2.9.4. RIGHT_OUTER_JOIN

The RIGHT_OUTER_JOIN is like an outer join, except that only the matched pairs and the unmatched tuples from S will generate an output tuple. Unmatched tuples from R are discarded.

6.2.9.5. Restrictions on Join Queries

The following are restrictions on join queries in the current implemetation of Tigon SQL:

1. No more that two tables can be in the FROM list (2-way join only)
2. Tigon SQL has to be able to figure out a window on the input streams over which to evaluate the join. To do this, there must be a predicate in the query which equates a temporal value (increasing or decreasing) from one table to a temporal value from the other table. In the example on the previous page, the predicate is R.tb = S.tb (assuming that R.tb and S.tb are both increasing). The temporal predicate can involve scalar expressions, so that R.tb+1=S.tb/60 will also work; however, the following predicates will not work:
    a. R.tb - S.tb = 2
    b. R.tb < S.tb+1 and R.tb >= S.tb
3. Tigon SQL will deduce the temporalness of fields in the SELECT clause only under the following conditions:
    a. There are predicates R.tb=S.tb, which imply that R.tb and S.tb are temporal. The temporalness of these fields can be used to impute the temporalness of a scalar expression in the SELECT clause.
    b. There is a predicate (temporal scalar expression in R) = (temporal scalar expression in S). If a scalar expression in the SELECT clause exactly matches one of the scalar expressions in the predicate, the output field is temporal. For example, R.tb+1 = S.time/60 means that R.tb+1 and S.time/60 are temporal.
4. The current implementation of the join operator uses a *hash join* : that is, it puts potentially matching tuples into the same hash bucket, and then applies any remaining predicates. Tigon SQL looks for predicates of the form (scalar expression in S) = (scalar expression in R) to define the hash bucket. In the example, Tigon SQL will use the predicates 'R.SourceIP = S.SourceIP' and 'R.DestIP = S.DestIP' to define its hash buckets. The more selective these predicates are, the more efficient the join will be. Since scalar expressions are allowed, the predicate 'UMIN(R.SourceIP,R.destIP)=S.sourceIP&IP_VAL'255.255.0.0' will also

be used for hashing. However the following predicates will not be used for hashing:

a. S.length_sum > R.length_sum
b. S.length_sum - R.length_sum = 500
c. R.length_sum = 55

6.2.10.Filter Join

A filter join is a special type of join that can be evaluated at the LFTA. A common use scenario is to look for packet in protocol P whose payload contains a particular string. An efficient way to perform this task is to first look for the start of a protocol P flow. Then perform regular expression matching on subsequent packets of the same flow. That is, we are joining a stream (the *match* stream) which identifies packets that start a protocol P flow with all packets from the same interface (the *result* stream), for some period of time.

The syntax of a filter join is the same as that of regular join, with the following four exceptions:

1. Field references in the Select list must be from the result stream, not the match stream.
2. There must be at least one hashable join predicate, e.g. of the form (scalar expression in R) = (scalar expression in S).
3. There must not be any join predicate on temporal scalar expressions.
4. The from clause is written as
   FILTER_JOIN(temporal_field, duration) From Protocol1 R, Protocol2 S
   where
   a. R is the result stream and S is the match stream. Protocol1 and Prototocol2 may be different.
   b. temporal_field is an increasing temporal field, and duration is a positive non-zero integer.

After identifying a packet from the match stream, the filter join will look for joining packets from the result stream for duration ticks of the temporal field. The FILTER_JOIN keyword is augmented with the temporal_field and duration as a convenience, replacing the otherwise required expression, which is as follows:

R.temporal_field>=S.temporal_field and R.temporal_field<=S.temporal_field+duration

The filter join implementation uses an approximate set membership algorithm to perform the join. There are two algorithm options:

- Bloom filter

- (Limited-size) hash table

The Bloom filter algorithm has false positives (it accepts packets from the result stream that don't actually have a match in the match stream) but no false negatives. The hash table algorithm has false negatives (it fails to accept result stream packets that match a match stream packet). The default algorithm is the Bloom filter, but define `algorithm hash` to use the hash algorithm.

By default, the Bloom algorithm uses 10 Bloom filters each covering 1/10 of the temporal duration. Define `num_bloom` to use a different number of Bloom filters. By default, each Bloom filter has 4096 bits and uses three probes per hash value. To use a different size bloom filter, define `bloom_size` to the $\log_2$ of the desired number of bits. For example, define `bloom_size 16` to use 65536 bits per bloom filter. By default the hash algorithm uses 4096 hash table entries. Define `aggregate_slots` to change the hash table size. The following is an example of a filter join query:

```
     DEFINE{
             algorithm hash;
     }
       SELECT  R.subinterface,  R.time,  R.timewarp,  R.srcIP,  R.srcPort,
R.destIP, R.destPort,
             R.sequence_number, R.ack_number
     FILTER_JOIN(time, 15) from   TCP R,TCP S
      WHERE  R.srcIP = S.srcIP and        //key
             R.destIP = S.destIP and      //key
             R.srcPort = S.srcPort and    //key
             R.destPort = S.destPort and  //key
             R.protocol = 6 and
             S.protocol = 6 and
             R.offset = 0 and
             S.offset = 0 and
             R.data_length <> 0 and
             S.data_length <> 0 and
             str_match_offset[0,'HTTP',S.TCP_data] and
                 str_regex_match[R.TCP_data,'.(mov|wav|mp3|mpg|mpeg|wma|
wmv']
      //          expensive single relational predicate
```

6.2.11.GROUP-BY/Aggregation

A GROUP-BY/aggregation query divides its input stream into a set of *groups*, computes *aggregate functions* over all tuples in the group, and then outputs a tuple based on the group definition and the values of the aggregate function. The syntax of a GROUP-BY/ aggregation function is as follows:

```
SELECT list_of_scalar_expressions
FROM table
[WHERE predicate]
GROUP BY list_of_groupby_variables
[Having predicate]
```

An example query is as follows:

```
SELECT sourceIP, tb, count(*), max(offset)
FROM eth0.IPV4 T
WHERE T.protocol=1
GROUP BY T.sourceIP, T.time/60 as tb
HAVING count(*) > 5
```

6.2.11.1. GROUP-BY Variables

The collection of GROUP-BY variables defines the group. All tuples with the same values of their GROUP-BY variables are in the same group. The syntax for defining a group is as follows:

Scalar_expression as name

When a tuple arrives, the scalar expression is computed and its value is assigned to the GROUP-BY variable *name*. At the risk of some confusion, GSQL provides a shortcut expression for defining a GROUP-BY variable. If the value of the input stream field is the value of the GROUP-BY variable, then the syntax

Fieldname

defines a GROUP-BY variable with the same name as the field name, and whose value is computed to be the value of the field; therefore, the following three GROUP-BY clauses are equivalent:

```
GROUP BY sourceIP
GROUP BY T.sourceIP
GROUP BY T.sourceIP as sourceIP
```

As previously noted, when GSQL imputes the meaning of a fieldname, it will prefer to impute the name as referring to a GROUP-BY variable. Therefore use `T.sourceIP` to refer to the field value, and `sourceIP` to refer to the GROUP-BY variable if there is any possibility of confusion.

6.2.11.2. Extended Group-by Patterns

In some applications, the user wishes to compute aggregations at multiple granularities. For example, the user might wish to compute, the number of packets flowing from each source IP address to each destination IP address over a five minute period – and also the number of packets flowing from each source IP address, and the number of packets

flowing to each destination IP address. This task could be accomplished using three queries, but for convenience, maintainability, and performance optimization, we can use a single query:

```
SELECT sourceIP, destIP, count(*)
FROM eth0.IPV4 T
GROUP BY T.time/60 as tb, Cube(sourceIP, destIP)
```

The CUBE keyword indicates that grouping should be performed using all subsets of the group-by variables in its argument. For a two-parameter argument, four groups are affected by each record that is processed: (sourceIP, destIP), (sourceIP,-), (-, destIP), and (-,-). All output records must have the same schema, so the missing group-by variables are assigned a default value (see Table 6: Literals). The default value for the IPv4 type is 0.0.0.0, so if a packet arrives with time=6,000,000, sourceIP=1.2.3.4 and destIP=4.3.2.1, then the count is incremented for the following four groups: (6,000,000, 1.2.3.4, 4.3.2.1), (6,000,000, 1.2.3.4, 0.0.0.0), (6,000,000, 0.0.0.0, 4.3.2.1), (6,000,000, 0.0.0.0, 0.0.0.0).

GSQL offers three types of extended group-by patterns: Cube, Rollup, and Grouping_Sets.

Cube: is specified with a CUBE, Cube, or cube keyword, followed by a list of group-by variables as defined in Section 6.2.11.1. When a record arrives, a collection of groups is created, one for each element of the set of all subsets of the group-by variables in the Cube's argument. If there are *n* group-by variables in the Cube's argument, then 2^n patterns of group-by variable assignment are created. Group-by variables not in a generated pattern receive a default value.

Rollup: is specified with a ROLLUP, Rollup, or rollup keyword. The Rollup keyword also takes a list of group-by variables as an argument, but creates a hierarchical pattern. For example, Rollup(sourceIP, destIP) creates the patterns (sourceIP, destIP), (sourceIP,-), (-,-). If there are *n* group-by variables in the Rollup's argument, then *n+1* patterns are created.

Grouping_Sets: is specified with a GROUPING_SETS, Grouping_Sets, or grouping_sets keyword. The Grouping_Set keyword takes a list of lists of grouping variables as its argument. The patterns created are exactly those specified by the list of lists. For example, Grouping_Sets((sourceIP, destIP), (sourceIP), (destIP)) creates the patterns (sourceIP, destIP), (sourceIP,-), (-,destIP).

A Group By clause can contain a mixture of Cube, Rollup, and Grouping_Sets keywords as well as individual grouping variables. The number of patterns that result is the product of the number of patterns of each entry. For example, the following Group By clause has 16 patterns:

```
GROUP BY T.time/60 as tb, Cube(substr(customer_id,8) as cid,
product_id), rollup(state, city, zip)
```

A group-by variable can be referenced in only one of the entries of the entries in the Group-By clause. So for example, the following is illegal because state is referenced twice in two different components:

```
GROUP BY T.time/60 as tb, state, rollup(state, city, zip)
```

6.2.11.3. Aggregate Functions

An aggregate function is one whose value is computed from all tuples in the group. The syntax of an aggregate function is the same as that of a regular function, except for the aggregate function '`count(*)`'. Table 9 below contains a list of aggregate functions in GSQL. In addition, Section 12-User-Defined Aggregate Functions can be referenced, as discussed below.

Aggregate function	Meaning	Operand	Result
Count	Number of tuples in group.	*	Int
Sum	Sum of values of operand	Unsigned short, unsigned int, signed int, unsigned long long, signed long long, float	Same as operand
Min	Minimum of values of operand	Unsigned short, unsigned int, signed int, unsigned long long, signed long long, string, IP, IPV6, float	Same as operand
Max	Maximum of values of operand	Unsigned short, unsigned int, signed int, unsigned long long, signed long long, string, IP, IPV6, float	Same as operand

And_Aggr	AND of values of operand	Unsigned short, unsigned int, signed int, unsigned long long, signed long long, bool	Same as operand
Or_Aggr	OR of values of operand	Unsigned short, unsigned int, signed int, unsigned long long, signed long long, bool	Same as operand
Xor_Aggr	Exclusive OR of values of operand	Unsigned short, unsigned int, signed int, unsigned long long, signed long long, bool	Same as operand

Table 9: Built-in Aggregate Functions

6.2.11.4. HAVING clause

The HAVING clause has the syntax below:

HAVING predicate

The HAVING clause defines an optional postfilter to apply after the group and its aggregates are computed. An output tuple is created only if the group and its aggregates satisfy the predicate.

6.2.11.5. Restrictions on Scalar Expressions

A GROUP-BY/aggregation query defines a two-stage process. First, gather the groups and compute the aggregates. Second, apply the postfilter and generate output tuples. The different components of a GROUP-BY/aggregation query have restrictions on the entities that can be referenced, depending on which part of the process they affect:

- **GROUP-BY variable definition:** The scalar expression that defines the value of a GROUP-BY variable may not reference any aggregate function or any other GROUP-BY variable.
- **Aggregate function operands:** The operand of an aggregate function may not reference the value of any other aggregate function.
- **WHERE clause:** No scalar expression in a WHERE clause may reference the value of any aggregate function.

- **HAVING clause:** No scalar expression in a HAVING clause may reference any field of the input table (e.g., only GROUP-BY variables, aggregate function values, literals, query parameters, and interface properties).
- **SELECT clause:** No scalar expression in the SELECT clause may reference any field of the input table – the same as for the HAVING clause.

6.2.11.6. Temporal Aggregation

In the conventional definition of GROUP-BY/aggregation, no output can be produced until all of the input has been processed. Because the stream input (in general) does not end, we have incorporated optimizations into GSQL to unblock the GROUP-BY/ aggregation operator.

If one or more of the GROUP-BY variables can be imputed to be temporal (i.e., either increasing or decreasing), then the group is *closed* (no further tuples will be a member of the group) when an incoming tuple has a different value for one of the temporal GROUP-BY variables. When the group is closed, its output tuple can be computed and put on the output stream, and the group's memory reclaimed.

6.2.11.7. GROUP-BY and Aggregation Summary

Let us recall the example query:

```
SELECT sourceIP, tb, count(*), max(offset)
FROM eth0.IPV4 T
WHERE T.protocol=1
GROUP BY T.sourceIP, T.time/60 as tb
HAVING count(*) > 5
```

This query will interpret data packets from the eth0 interface using the IPV4 protocol. If the protocol field of the packet has the value 1, the query will compute the GROUP-BY variables sourceIP and tb. If this group does not exist in memory, a new group will be created. The query will count the number of tuples in this group, and also the maximum value of the offset field. When the group is closed (a tuple with a larger value of tb arrives), the query will check if the number of tuples in the group is larger than 5. If so, the query will marshal a tuple as specified by the SELECT clause, and put it on the output stream.

6.2.12.Running Aggregation

A limitation of regular aggregation is that it can summarize data from a single time window only, and that time window is the reporting interval. However, in many cases we

would like to report aggregates from data over multiple time windows. Here are two examples:

1. One might want to compute a moving average of the number of bytes per connection over the last five time windows.
2. One might want to compute the number of duplicate sequence numbers in a TCP/IP connection. It is possible to write a user-defined aggregate function to compute this aggregate, but its state should span the entire time during which the connection is active.

GSQL provides *running aggregates* to enable aggregation across multiple time windows. A running aggregate is a user-defined aggregate which is labeled as RUNNING in the **external_fcns.def** file (normally located in the tigon/tigon-sql/**cfg** directory). For example, the declaration of a moving average function could look like the following:

```
float UDAF [RUNNING] moving_avg(uint, uint HANDLE);
```

(The second parameter is the window size. It is declared to be HANDLE to ensure that it is a constant in the query.)

A running aggregate is an aggregation query that references a regular aggregate. The processing is similar to that of a regular aggregation query. In particular, when the time window changes, the groups that satisfy the `HAVING clause` are output. A running aggregation differs from a regular aggregation in the following ways:

1. Groups are not automatically deleted when the time window changes. If a group does not satisfy the `Closing_When` clause when the time window changes, it is created in the new time window, with updated values of its temporal GROUP-BY variables.
2. Groups which satisfy the `Closing_When` clause when the time window changes are discarded.
3. If a group is carried over to the new time window,
    a) Regular (non-running) aggregates are initialized to an empty value.
    b) Running aggregates receive a notification that the time window changed.

The syntax of a running aggregation query is as follows:

SELECT list_of_scalar_expressions
FROM table
[WHERE predicate]
GROUP BY list_of_groupby_variables
[HAVING predicate]
[CLOSING_WHEN predicate]

For example,

```
SELECT tb, srcIP, moving_avg(len,5)
FROM TCP
WHERE protocol = 6
GROUP BY time/10 as tb, srcIP
HAVING moving_avg(len,5)>0
CLOSING_WHEN count(*)=0
```

This query computes the moving average of the lengths over the last 5 time periods of 10 seconds each, closing the group and discarding state whenever the source IP emits no packets during a 10 second interval. This query will generate a report every 10 seconds; acting like a continuous query.

6.2.13.Stream Sampling and Aggregation

Stream Sampling Operator is used to implement a wide variety of algorithms that perform sampling and sampling based aggregation over data streams. The operator collects and outputs sets of tuples which are representative of the input.

6.2.13.1. Query Definition

Most sampling algorithms follow a common pattern of execution:

1. A number of tuples are collected from the original data stream according to certain criteria (perhaps with aggregation).
2. If a condition on the sample is triggered (e.g. the sample is too large), a cleaning phase is initiated and the size of the sample is reduced according to another criteria.

This sequence can be repeated several times until the border of the time window is reached and the final sample is outputted.

Following this pattern of execution, a GSQL sampling query has the following format:

```
SELECT <select expression list>
FROM <stream>
WHERE <predicate>
GROUP BY <group-by variables definition list>
        [ : SUPERGROUP < group-by variables definition list>]
[HAVING <predicate>]
CLEANING WHEN <predicate>
CLEANING BY <predicate>
```

A predicate in the “CLEANING WHEN” clause defines the condition for invocation of the cleaning phase. For example, a cleaning phase might be triggered when the size of the

current sample is too big, i.e. exceeds a predefined threshold value for the desired size of the sample.

A predicate in the "CLEANING BY" clause defines the criteria according to which the current sample is reduced to the desired size and will be evaluated only when the predicate from the "CLEANING WHEN" clause had been evaluated to true.

SELECT: may reference constants, query parameters, group-by variables, aggregate values and stateful functions.

WHERE: may reference constants, query parameters, tuple attributes and stateful functions.

CLEANING WHEN: may reference constants, query parameters, group-by variables and stateful functions.

CLEANING BY: may reference constants, query parameters, group-by variables, aggregate values and stateful functions.

HAVING: may reference constants, query parameters, group-by variables, aggregate values and stateful functions.

In order to implement a sampling algorithm using this type of query, user must be familiar with the definition of stateful functions and supergroups. Next section gives a light overview of these two concepts. More detailed information is available in the next segment (Stateful Functions).

6.2.13.2. Supergroups

As was mentioned earlier, a state structure stores various control variables of the sampling algorithm. Since a user might wish to obtain a sample on a group-wise basis, the sampling state is associated with supergroups, and samples are associated with the groups in a supergroup.

The variables in the SUPERGROUP clause must be a subset of group-by variables defined in the GROUP BY clause, not including temporal variables. By default, the supergroup is ALL, which in most cases mean that a single supergroup is associated with a time window specified by a query.

Along with sampling state variables, the supergroup can compute superaggregates (aggregates of the supergroup rather than the group). We use the dollar sign (`$`) to denote that an aggregate is associated with the supergroup rather than the group. Superaggregate **`min$(len)`** for instance will return a tuple with the smallest length attribute for every supergroup. One of the most useful superaggregates is **`count_distinct$()`**, which returns the number of distinct groups in a supergroup. This is a built-in superaggregate that is maintained regardless of whether it was explicitly used in a query.

6.2.13.3.3. Query Evaluation Process

The Sampling query is designed to express stream sampling algorithms that follow a common pattern of execution. First a number of items are collected from the original data stream according to a certain criteria, and perhaps with aggregation in the case of duplicates. If a condition on the sample is triggered (e.g., the sample is too large), a cleaning phase is initiated and the size of the sample is reduced. This sequence can be repeated several times until the border of the time window is reached and the sample is outputted.

The semantics of a sampling query are as follows:

When a tuple is received, evaluate the WHERE clause. If the WHERE clause evaluates to false, discard the tuple.

If the condition of the WHERE clause evaluates to TRUE, do the following:

- Create and initialize a new supergroup and a new superaggregate structure, if needed; otherwise, update the existing superaggregates (if any).
- Create and initialize a new group and a new aggregate structure, if needed; otherwise, update the existing aggregates (if any).
- Evaluate the CLEANING_WHEN clause.
- If the CLEANING_WHEN predicate is TRUE
    - Apply CLEANING_BY clause to every group.
    - If the condition of CLEANING_BY clause evaluates to FALSE
        - Remove group from the group table, and update superaggregates

When the sampling window is finished, do the following:

- Evaluate the HAVING clause on every group.
- If the condition in the HAVING clause is satisfied, then the group is sampled; otherwise, discard the group.
- Evaluate SELECT clause on every sampled group. Create an output tuple.

6.2.14.Stream Merge

A stream merge query combines two streams in a way that preserves the temporal properties of one of the attributes. The syntax of a stream merge query is as follows:

Merge field1 : field2: ... : fieldn

FROM table1, table2, …, tablen

For example,

```
Merge P1.tb : P2.tb
FROM PacketSum1 P1, PacketSum2 P2
```

The merge query is restricted in the following ways:

- The two input tables must have the same layout, the same number of fields, and fields in corresponding positions must have the same data type. The field names, however can be different.
- The two merge fields must be temporal and in the same way (with both either increasing or decreasing). They must be in the same position in both input tables.

Please note that input streams are merged on fields of the input streams. Computed values are not accepted. The output stream is temporal in the merge field.

6.3. Query Specification

A GSQL query has the following syntax:

[Parameters definition]
[Options definition]
query_text

A query file has the following format:

query1;
...
queryn

As is shown, the format is a list of queries (with optional parameters and options) separated by semicolons. For example,

```
SELECT tb, srcIP, count(*)
FROM tcp_pkts
GROUP BY tb, srcIP;

DEFINE{
      query_name tcp_pkts;
}
SELECT time/60 as tb, srcIP
FROM TCP
WHERE protocol = 6 and offset = 0
```

The first query reads from the second query, which is named **tcp_pkts** by the option in the define block. By default, the name of the first query in a file is the file name up to the

'.gsql' suffix. For example, if the file above were named count_tcp.gsql then the first query would be named **count_tcp**.

Some queries produce externally-accessible output, while others perform internal processing. The file `output_spec.cfg` specifies which queries produce output, and the nature of the output. See Section 6.4 for more information about the output_spec.cfg file.

The previous section covered the query language. In this section we discuss how to set the query options and parameters.

6.3.1. Parameters

A query can be parameterized, as discussed in Section 6.2.3, “Scalar Expressions.” A parameter has the format *$name*, and acts in most respects like a literal. All parameters used in a query must be defined in the *parameter block* of the query. The parameter block has the following format:

```
PARAM{
  Parameter_name parameter_type;
        …
  parameter_name parameter_type;
}
```

The parameter name has the following format: [A-Za-z_][A-Za-z0-9_]*, while the parameter type is one of the data type names listed in Table 4: Data Type Names. The parameter value then has the corresponding data type. All parameters for the query must be supplied when the query is started. In addition, the parameters can be changed while the query executes.

The Tigon SQL run time library provides a suite of functions that simplify the process of creating a parameter block for a query. Given a query definition, the following two library functions parse the query to enable functions which pack parameter blocks:

```
int ftaschema_parse_string(char *f);
int ftaschema_parse_file(FILE *f);
```

These functions return a handle, referred to as 'sh' in the following functions:

```
int ftaschema_parameter_len(int sh); // number of parameters
char * ftaschema_parameter_name(int sh, unsigned int index); // parameter name
//          set the parameter, pass in textual representation of the param value
int ftaschema_setparam_by_name(int sh, char *param_name,
        char *param_val, int len);
int ftaschema_setparam_by_index(int sh, int index,
        char *param_val, int len);
//          Create a parameter block to be passed to the query
int ftaschema_create_param_block(int sh, void ** block, int * size);
```

If one query reads from another, they must have identical sets of parameters. The top level queries will instantiate the queries they read from and pass along the parameter block.

6.3.2. Options (DEFINE Block)

A query can accept a set of options, which affect how the query is processed. These options are defined in the *define block*, which has the following format:

```
DEFINE{
  Option_name option_value;
 …
  option_name option_value;
}
```

The option name has the format [A-Za-z_][A-Za-z0-9_]*, while the option value has the same format as a string literal, '[^'\n]*'. However the enclosing single quotes around the option value can be dropped if the value has the same format as a name. The following are the options currently accepted by GSQL:

- **real_time**: If the option real_time is set to any non-empty string, the aggregation LFTAs will try to flush their aggregation buffers as soon as possible. This entails computing the temporal GROUP-BY variables for each tuple regardless of whether the tuple satisfies the Where predicate.
- **aggregate_slots**: The number of slots to allocate in the LFTA aggregation table. The default value is 4096. This define will be overridden if there is an entry in the lfta_htsize file, see Section 7.5.2.
- **query_name**: The name of the query (used to access the query output).
- **slow_flush**: The number of input packets processed per output tuple flushed in LFTA aggregation processing (default 1).
- **lfta_aggregation**: Generate only the LFTA part of an aggregation query, and give the LFTA the query name (instead of a mangled name)
- **select_lfta:** If an aggregation query can't be split in a way that creates an LFTA-safe aggregation subquery, create a selection LFTA instead.
- **algorithm** : if set to `hash`, use a hash table to process the filter join instead of the default Bloom filter.
- **num_bloom** : The number of Bloom filters to use when processing a filter join using Bloom filters (i.e., the number of distinct temporal intervals covered).

- **Bloom_size** : The $\log_2$ of the number of bits in each Bloom filter used to process a filter join. E.g., a value of 10 means that each Bloom filter has 1024 bits.
- **comment**: uninterpreted comment text.
- **title** : text interpreted as the title of the query for auto-documentation.
- **namespace** : interpreted as the namespace of the query for auto-documentation.
- **print_warnings** : If present, causes the merge operator to print out-of-order warnings.
- **max_lfta_disorder** : Tigon SQL uses "out-of-order" processing to eliminate spikes in resource usage when aggregate epochs change (which triggers a buffer flush). However, out-of-order processing increases the delay before producing a report for aggregate queries. Set max_lfta_disorder to 1 to reduce the reporting delay, at the cost of spikes in rts processing.

The DEFINE block can also define the value of string literals. For example, if `foo 'bar';` is in the DEFINE block, then in the query text #foo is string literal with value 'bar'. See Section 6.2.3.

6.3.3. Query Name

Each query in the system has a name, which the application specifies to access the output stream. GSQL uses the following procedure to determine the name of a query:

1. Use the value of query_name option, if this value is non-empty.
2. If this is not the case, use the name of the file containing the query, between the last '/' character and the first '.' character following it.
3. If the file name can't be determined or if the name extraction results in an empty string, use the name *default_query*.

6.4. Defining a Query Set

A typical use of Tigon SQL generally executes multiple queries simultaneously. A collection of queries can be specified by

- Putting multiple queries in a file, separated with semicolons (see Section 6.3).
- Providing multiple queries as input to the query compiler (see Section 6.5). The buildit script (see Section 10.1) uses all files ending in .gsql
- Referencing library queries (see Section 4.1.8).

Not every query query in the set will produce output – ofttimes a complex query is built from a collection of simple subqueries. The collection of queries that can produce output is specified by the `output_spec.cfg` file.

The format of the output_spec.cfg file is:

```
query_name,operator_type,operator_param,output_directory,bucketwidth,
                    partitioning_fields,n_partitions
```

Where

query_name : is the name of the query.

operator_type : how output is generated. Currently supported types are `stream`, `file`, `zfile`.

`stream` means, the hfta will stream its output (if there is not another hfta consuming its output). Use this option if you want use gsgdatprint to generate files.

`file` : generate uncompressed files (but see the gzip option)

`zfile`: generate compressed files, using zlib.

operator_param : parameters for the output operator. Currently the only parameter that is recognized is `gzip`, and only by the `file` output operator. If `gzip` is specified, the file is gzipped after creation.

output_directory : where the files are written (operator type file or zfile). The path can be a relative directory.

bucketwidth : the change in the temporal field required before creating a new file. The default is 60. Set to 1 to use the natural bucket width.

partitioning_fields : if the file output operator is asked to produce multiple output streams, use these fields to hash the output among the output streams. Only for operator_type file and zfile.

n_partitions : number of (file or zfile) output streams to create.

If an hfta that creates output is parallelized (as specified in hfta_parallelism.cfg), then each file output stream is mangled with __copyX. If the hfta parallelism is larger than n_partitions, then the hfta is still parallelized as specified, and the number of file streams is equal to the hfta parallelism. If the hfta_parallelism is equal to n_partitions, then each parallel hfta produces a single output file stream. If n_partitions is larger than the hfta parallelism, then the parallelism must divide n_partitions. Each hfta copy produces its share of the file output streams, mangled by fileoutputX.

The specification of which queries can produce output is a critical optimization for Tigon SQL, as helper subqueries can be combined into processes (minimizing communications costs) and optimized by query rewriting.

While the gsgdatprint tool (Section 10.5) can be used to write streaming data to files, using a file output operator is significantly more efficient. A query can produce multiple types of output, e.g. both stream and file output, by specifying both types of output in the output_spec.cfg file.

If a query has file or zfile output, it must still be instantiated before it can execute. The gsprintconsole tool (Section 10.4) can be used to instantiate a query. No output will be sent to the gsprintconsole process (unless the query also has stream output), instead the query will write its data to files.

6.5. Invoking the GSQL Compiler

Normally, a user will use one of the provided buildit scripts to create the Tigon SQL executables. However, an advanced user might wish to adjust how translate_fta operates.

Usage: translate_fta [-B] [-D] [-p] [-L] [-l <library_directory>] [-N] [-H] [-Q] [-M] [-C <config_directory>] [-Q] [-S] [-h hostname] [-c] [-f] [-R path] [schema_file] input_file [input file ...]

[-B] : debug only (don't create output files)

[-D] : distributed mode.

[-p] : partitioned mode.

[-L] : use the live_hosts.txt file to restrict queries to a set of live hosts.

[-C] : use <config_directory> for definition files

[-l] : use <library_directory> for the query library.

[-N] : output query names in query_names.txt

[-H] : create HFTA only (no schema_file)

[-Q] : use query name for hfta suffix

[-M] : generate Makefile and runit and stopit scripts.

[-S] : enable LFTA statistics (alters the generated Makefile).

[-f] : Output schema summary to schema_summary.txt

[-h] : override host name (used to name the rts).

[-c] : clean out any existing Makefile and hfta_*.cc files before doing doce generation.

[-R] : path to the root STREAMING directory (default is ../..)

The output is an lfta.c file and some number of hfta_*.cc files. The –N option will output the names of all generated queries in the file query_names.txt. The root (output) query is

marked by an 'H' following the name. Child queries follow the root query and are marked by an 'L' following the name.

See also Section 10.3.

6.5.1. Files Used by translate_fta

Translate_fta depends on the following definition files. external_fcns.def, ifres.xml, <hsot_name>,ifq, and the packet schema file are all assumed to be in the directory specified by the –C switch. The other files are assumed to be in the current directory.

- **external_fcns.def** : prototypes of predicates, functions, UDAFS, etc.
- **ifres.xml** : descriptions of all interfaces available to Tigon SQL.
- <**host_name>.ifq** : query set predicates over the interfaces available at <host name>.
- **output_spec.cfg** : defines the collection of queries which generate output.
- Optimization hint files as described in Section 7.5.

6.5.2. Files Generated by translate_fta

A successful invocation of translate_fta will create the following files

- Source code: <hostname>_lfta.c and hfta_[0-9]+.c. <hostname> is the value returned by the shell tool hostname. Use –h to override the hostname.
- preopt_hfta_into.txt, postopt_hfta_info.txt : information about system configuration before and after distributed optimizations are performed.
- qtree.xml : detailed query operator configuration in xml format.
- gswatch.pl : a tool for monitoring the processes in the Tigon SQL processing system.
- set_vinterface_hash.bat : a command for setting virtual interface settings in DAG cards.
- Makefile, runit, stopit : are generated if translate_fta is invoked with the –M parameter. Makefile is the makefile used for compiling the system. runit starts all Tigon SQL processes – it DOES NOT start any subscribing applications. stopit kills all Tigon SQL processes and any of the common subscribing applications.

7. Optimizations

GSQL uses a number of optimizations to improve performance. Here we list the optimizations to keep in mind when using Tigon SQL.

7.1. Query splitting

GSQL will try to execute as much of the query as possible as an LFTA; however, some queries cannot execute as LFTAs even if they reference only protocols as data sources. These queries are split into an HFTA and an LFTA component. The HFTA inherits the query name, while the LFTA is given a name which is created by prepending '_fta_' to the query name. For example, if the query name is 'Foo', then the HFTA will be named 'Foo' while the LFTA will be named '_fta_Foo'.

A query will be split for a variety of reasons, one example being access to a function which is not available at the LFTA level. In addition, all aggregation queries are split. The LFTA will perform *partial aggregation*, while the HFTA computed the exact result from the partially summarized output of the LFTA. The LFTA performs partial aggregation to achieve the data reduction benefits of aggregation, but using limited space. The number of GROUP-BY slots is set by the **aggregate_slots** option, which has a default value of 10. The LFTA uses a set-associative hash table, so collisions are possible even if the number of groups is less than the number of GROUP-BY slots.

7.2. Prefilter

After computing the collection of LFTAs to execute, GSQL finds a prefilter for the query. If a packet satisfies the WHERE clause of some LFTA, it will satisfy the prefilter. A packet is presented to the LFTAs only if it satisfies the prefilter. The prefilter provides a shortcut that allows the Tigon SQL RTS process to avoid invoking a query on a packet that it will reject.

7.3. Group Unpacking

See Section 5.1.

7.4. Process Pinning

Query processing in a streaming system forms a Directed Acyclic graph of data records flowing from one process to another. By careful placement of Tigon SQL processes, we can minimize the costs of data transfers, e.g. by making use of 2nd level and 3rd level cache. Processes can be pinned to particular cores using the `taskset` command. Tigon SQL provides a self-optimization facility (Section 7.5.3), which will create a file `pinning_info.csv` with recommendations for process placement. The script tigon/

tigon-sql/`bin/pin_processes.pl` will perform the process pinning recommended by pinning_info.csv. The Tigon SQL instance must be executing when pin_processes.pl is run.

7.5. Optimization Hints and Self-Optimization

The translate_fta query compiler (Section 6.5) uses several configuration files to set parameters for performance optimization. In this section we discuss these files, and a mechanism by which these files can be automatically generated and tuned.

7.5.1. HFTA Parallelism

The query compiler translate_fta consults the file `hfta_parallelism.cfg` to determine the number of copies of an HFTA to create. The format of hfta_parallelism.cfg is

```
Query_name, level_of_parallelism
```

For example,

```
example, 2
example2, 1
```

The level of parallelism must divide the level of parallelism of the hfta's sources – and ultimately must divide the number of "virtual interfaces" which scan an interface. The default level of parallelism is 1. Since virtual interfaces are not yet implemented for file input, the hfta parallelism must always be 1.

7.5.2. LFTA Aggregation Buffer Sizes

Tigon SQL has a two-level query processing architecture. As described in Section 1, records from a streaming source are processed by all subscribing queries in the run-time system. This processing is simple and fast, performing only selection, projection, and *partial* aggregation.

In the run-time system, an aggregation query is preprocessed using a small fixed-size hash table. If the number of groups is larger than the number of hash-table entries, one of the groups is evicted and sent to an HFTA, where all partial aggregates are summed up in a hash table that can expand to the size that is needed.

Tuning the LFTA aggregation buffer size is a balance between two costs. To improve cache locality, the buffer size should be as small, but to minimize the eviction costs, the buffer should be large.

The default buffere size is 4096 entries. This value can be overridden by the **aggregate_slots** define (see Section 6.3.2). Both of these values are overridden by entries in the file `lfta_htsize.cfg`, which has the format

lfta_query_name,hash_table_size

The lfta query names in the file lfta_htsize.cfg are (usually) mangled names of query fragments executing in the run-time system. The names of all lfta queries are listed in the file qtree.xml, which is generated by translate_fta (see Section 6.5.2). However, lfta_htsize is intended to be part of the automatic optimization process.

7.5.3. Self Optimization

Tigon SQL Tigon SQL provides a self-optimization facility which generates files that help to optimize Tigon SQL performance: `lfta_htsize.cfg`, `hfta_parallelism.cfg`, and `pinning_info.csv`. In this section, we document how to use the self-optimization facility.

Optimization requires statistics, which in the case of a streaming system is available by running the system on the actual data streams. In general, self-optimization is an iterative process: run an unoptimized version of Tigon SQL, collect statistics, determine an improved configuration, and repeat.

1. The translate_fta query compiler accepts –S as a flag, indicating that it should collect statistics (see Section 6.5). The buildit script buildit_with-stats (see Section 10.1) uses the –S flag.
    a. By default, statistics are logged to file /var/log/gstrace. The script tigon/tigon-sql`/bin/get_last_trace.pl` will extract the trace of the last run.
2. When running a query set generated with the –S flag, Tigon SQL generates internal statistics, but not performance statistics. The script tigon/tigon-sql`/bin/monitor_gs.pl` will collect performance statistics about the running Tigon SQL instance, putting them in the file resource_log.csv
3. For process placement, the optimizer needs a map of the server's cores. This information is normally kept in the file tigon/tigon-sql`/cfg/cpu_info.csv`. Use the script tigon/tigon-sql/bin/parse_cpu_info.pl to generate this file, as described in Section 2.5.
4. The program tigon/tigon-sql`/bin/process_logs` will process qtree.xml (generated by translate_fta, see Section 6.5.2), cpu_info.csv, resource_log.csv, and a selected portion of the gslog (e.g. selected by get_last_trace.pl) to make a variety of analyses and performance recommendations.
5. The script tigon/tigon-sql`/bin/accept_recommendations.bat` accepts the configuration recommendations made by process_logs.

The process_logs program produce the following.

1. Performance_report.csv : a report on data traffic, record loss, and resource utilizations of various components of the Tigon SQL instance. This file is in csv format and is readily loaded into tools such as Excel.
2. hfta_parallelism.cfg.recommended : a recommended level of parallelization for query operators executed at the hfta level. The script accept_recommendations.pl will copy this file to hfta_parallelism.cfg (see Section 7.5.1).
3. lfta_htsize.cfg.recommended : a recommended lfta hash table size for lfta query fragments which perform aggregation. The script accept_recommendations.pl will copy this file to lfta_htsize.cfg (see Section 7.5.2).
4. rts_load.trace.txt : a record of previously attempted lfta hash table size allocations. This file is used to speed convergence in hash table size allocation.
5. pinning_info.csv : a recommended placement of run time systems nad hfta processes to cores (see Section 7.4)

The recommended procedure for self-tuning is

1. Compile the query set using buildit_with-stats
2. Start up the query set (runit) and all clients.
3. tigon/tigon-sql/bin/pin_processes.pl (if a pinning_info.csv file has been previously produced).
4. Run tigon/tigon-sql/bin/monitor_gs.pl
5. Let the system run for a while.
6. Stop the system (stopit)
7. tigon/tigon-sql/bin/get_last_trace.pl
8. tigon/tigon-sql/bin/process_logs last_tracefile.txt
9. tigon/tigon-sql/bin/accept_recommendations.bat

This procedure will iteratively optimize a Tigon SQL query set. Running the accept_recommendations.bat script will cause the next run to use the generated recommendations.

8. External Functions and Predicates

GSQL can be extended with externally defined functions and predicates. The prototypes for these functions and predicates must be registered in the file **external_fcns.def**, normally kept in tigon/tigon-sql/**cfg**. Each entry in this file is one of the following declarations:

- **Function:** *return_type FUN [optional list of modifiers] function_name(list of parameter data types);* This declaration indicates that the function accepts parameters with the specified data types and returns the specified data type.
- **Predicate:** *PRED [optional list of modifiers] predicate_name[list of parameter data types];* This declaration indicates that the predicate accepts parameters with the indicated data types. Predicates evaluate to true or false in a predicate expression.
- **User-defined Aggregate:** *return_type UDAF[optional list of modifiers] udaf_name storage_type (list of parameter types);* This declaration indicates that 'udaf_name' is an aggregate function returning the specified data type, using a block of type 'storage_type' for its scratchpad space, and taking the specified list of parameters.
- **Aggregate Extraction Function:** *return_type EXTR function_name aggregate_name extraction_function (list of parameter types);* This declaration indicates that when 'function_name' is referenced in a query, it is replaced by the call 'extraction_fcn(aggregate_name(…),…)'.
- **State:** *storage_type STATE state_name;* This declaration indicates that the storage block 'state_name' has the specified storage type. All stateful functions which declare this state name as their state share the storage block.
- **Stateful Function** : *return_type SFUN function_name state_name (list of parameter types) ;* This declaration indicates that the stateful function 'function_name' returns the indicated type, takes the specified list of parameters, and uses the indicated state as its storage block.
- **Comment** : a comment starts with two dash "—" or two slash "//" characters.

The optional list of modifiers of a function or predicate set the properties of that function or predicate. The modifiers are as follows:

- COST : Indicate the cost of evaluating the function or predicate to the optimizer. Legal values are FREE, LOW, HIGH, EXPENSIVE, and TOP (in increasing order of cost). The default value is LOW. FREE functions

and predicates can be pushed to the prefilter. Tigon SQL uses the function cost to determine the order in which to evaluate predicate clauses. If the COST of a function is HIGH or larger, then Tigon SQL will perform function caching if possible.

- LFTA_LEGAL : the function or predicate is available in an LFTA (by default, functions and predicates are not available in an LFTA).
- LFTA_ONLY : the function or predicate is available in an LFTA, but not in an HFTA (by default functions and predicates are available in an HFTA).
- PARTIAL : the function does not always return a value. In this case the function has a special call sequence.
- SUBAGGR : indicates that the user-defined aggregate can be split; use the SUBAGGR in the lfta.
- SUPERAGGR : indicates that the user-defined aggregate can be split; use the SUPERAGGR in the hfta.
- HFTA_SUBAGGR : Used to support aggregation in distributed mode.
- HFTA_SUPERAGGR : to support aggregation in distributed mode.
- RUNNING : indicates that the aggregate is a running aggregate.
- MULT_RETURNS : indicates that the aggregate doesn't destroy its state when asked to produce output, and therefore can produce output multiple times. Aggregates used in Cleaning_When and Cleaning_By clauses must have this property.
- LFTA_BAILOUT : indicates that the aggregate accepts the _LFTA_AGGR_BAILOUT_ callback.
- COMBINABLE : Indicates that the predicate is combinable at the prefilter (the predicate value is the same).
- SAMPLING : Used for load shedding.

The list of parameter data types completes the prototype. Function, stateful function, user-defined aggregate, and predicate names can be overloaded by changing the list of parameter data types. The properties of a parameter can be modified by following the data type name with one or more of the following modifiers:

- HANDLE : the parameter is a *handle* parameter (see below).
- CONST : the parameter must be a constant expression (a scalar expression involving unary or binary operators, literals, query parameters, and interface properties only).

- CLASS : the parameter is used for classifying COMBINABLE predicates at the prefilter. Predicates with identical scalar expressions for their CLASS parameters can be combined. All other other parameters (i.e., non-CLASS parameters) must be CONST or HANDLE.

A parameter can be designated a *handle* parameter by following the data type with the keyword HANDLE. Handle parameters are not passed directly; instead, they are registered with the function to obtain a parameter handle. Instead of passing the parameter value, the generated code will pass the pa*rameter handle*. This mechanism is provided to accommodate functions which require expensive preprocessing of some of their attributes, e.g. regular expression pre-compilation.

Some examples of function and predicate prototypes are as follows:

```
    bool FUN  [LFTA_LEGAL] str_exists_substr(string, string HANDLE);
    string FUN  [PARTIAL] str_between_substrings( string , string ,
string );
    PRED [LFTA_LEGAL] is_http_port[uint];
    float EXTR extr_avg avg_udaf extr_avg_fcn (uint);
    float FUN extr_avg_fcn (string);
    string UDAF[SUBAGGR avg_udaf_lfta, SUPERAGGR avg_udaf_hfta]
avg_udaf fstring12 (uint);
    string UDAF avg_udaf_hfta fstring12 (string);
    string UDAF avg_udaf_lfta fstring12 (uint);
    fstring100 STATE smart_sampling_state;
    BOOL SFUN ssample smart_sampling_state (INT, UINT);
    BOOL SFUN ssample smart_sampling_state (UINT, UINT);
```

For more information about user defined functions, predicates, and aggregates, see Section 7-Writing Functions and Aggregates.

8.1. User-defined Operators

Each HFTA is an independent process; therefore, it is possible to write a “user defined operator” which makes use of the HFTA API (see section 8- Gigascope™ API). A GSQL query can reference the user-defined operator as long as the interface of the operator is defined in the schema file. For example,

```
    OPERATOR_VIEW simple_sum{
        OPERATOR(file 'simple_sum')
        FIELDS{
            uint time time (increasing);
             uint sum_len sum_len;
        }
        SUBQUERIES{
            lenq (UINT (increasing), UINT)
        }
```

```
        SELECTION_PUSHDOWN
    }
```

The name of the view is **simple_sum** (which is the name to use in the FROM clause). The OPERATOR keyword indicates the nature and location of the operator. Currently, the only option is 'file', and the parameter of this option is the path to the operator's executable file. The FIELDS keyword indicates the schema which the operator exports. The SUBQUERIES keyword indicates the source queries for the operator. In the example, there must be a query in the query set named **lenq** whose schema matches that indicated by the schema in parentheses. An operator can read from more than one subquery. In this case, separate them by a semicolon (;). SELECTION_PUSHDOWN will be used for optimization, but currently nothing is done. An example of a query which references simple_sum is

```
    SELECT time, sum_len
    FROM simple_sum
```

9. Example Queries

9.1. A Filter Query

This query extracts a set of fields for detailed analysis from all TCP and UDP packets. The timestamp field has nanosecond granularity, so it can be used for detailed timing analysis. Note the following:

- The query is reading data from the default set of interfaces.
- When reading data from a protocol source, such as TCP, the schema only says how to interpret the data packet. The system does not perform any explicit tests to verify that the packet is in fact from the TCP protocol. There is an implicit test, however. All fields referenced in the query must be part of the packet. Since all the referenced fields are part of both the TCP and UDP schemas, this query can extract information about both types of packets. Using DataProtocol would be better, since it contains all the referenced fields.

```
SELECT time, timestamp, protocol, srcIP, destIP,
         source_port, dest_port, len
FROM TCP
WHERE (protocol=6 or protocol=17) and offset=0
```

9.1.1. Using User-Defined Functions

This query uses the 'getlpmid' function, which does longest prefix matching to extract data from IPv4 packets about local hosts. The getlpmid function returns with -1 if it is unable to match the IP address in its prefix table.

```
DEFINE {
      query_name 'local';
}
SELECT time as timebucket, srcIP as localip
FROM i2.IPV4
WHERE getlpmid(srcIP,'./localprefix.tbl') > 0
```

Note the following:

- The SELECT clause uses field renaming; the output fields are named timebucket and localip.
- The query reads from a specific interface, named i2.
- The second parameter of the 'getlpmid' function is the path to a file containing the local prefixes. This parameter is a *handle* parameter, meaning that it is processed when the query starts. The processing creates a data structure, which is passed to 'getlpmid' whenever it is called, making this type of processing very efficient.

9.1.2. Aggregation

This query computes the number of IPv4 packets and bytes seen in five-minute increments. The results are returned at the end of each increment.

Note the following:

- The query defines **aggregate_slots** to be 2000. The LFTA portion of the query will round up 2000 to the nearest power of two, and allocate a 2048 slot hash table. If there is an entry in the `lfta_htsize.cfg` file for query src, thus define will be overridden.
- The computed group value time/300 is renamed as tb.
- The query reads from the '`default`' set of interfaces. This is the same interface set you receive if you don't specify the interfaces.

```
DEFINE {
      query_name 'src';
      aggregate_slots 2000;
}
SELECT tb*300 as timebucket, srcIP, sum(len) as total_bytes, count(*)
as packets
FROM [default].IPV4
WHERE ipversion=4
GROUP BY time/300 as tb, srcIP
```

9.1.3. Aggregation with Computed Groups

This aggregation query produces a per-minute report of web pages referenced at least twice during that minute. Note the following:

- The group variable hostheader is computed using the 'str_extract_regex' function. This function is *partial*. If it can't match the regular expression, it fails. If it fails, the tuple is discarded and no group is created (as intended).
- The WHERE clause contains the predicate 'str_match_start', which ensures that the payload of the packet starts with 'GET'. This function is very fast because it's a one-word mask and compare. It is pushed to the LFTA and acts as a fast and highly selective filter.
- The HAVING clause ensures that all of the reported groups have at least two members. (The web page is referenced at least twice).

```
SELECT tb*60, min(timestamp), max(timestamp),
          destIP, dest_port, hostheader, count(*)
FROM TCP
WHERE ipversion=4 and offset=0 and protocol=6 and
          str_match_start[TCP_data, 'GET']
GROUP BY time/60 as tb, destIP, dest_port,
      str_extract_regex(TCP_data,'[Hh][Oo][Ss][Tt]: [0-9A-Z\\.:]*') as
      Hostheader
```

```
HAVING count (*) > 1
```

9.1.4. Join

This query reads from two stream queries, *http_request* and *http_response*. It computes the delay between http request and response by matching request and response pairs that occur within the same time period, tb (a 10 second time interval). Although this loses some of the request/response pairs, it captures most of them. The `S.tb=R.tb` predicate in the WHERE clause is required to localize the join of the two streams.

```
SELECT  S.tb as tb, getlpmid(S.destIP,'../../mappings/cpid.tbl') as
cpid,
        (A.timestamp - S.timestamp)/4294967 as rtt
INNER_JOIN from http_request S, http_response A
WHERE   S.srcIP = A.destIP
        AND S.destIP = A.srcIP
        AND S.srcPort = A.destPort
        AND S.destPort = A.srcPort
        AND S.tb = A.tb
        AND S.timestamp <= A.timestamp
        AND S.to_ack = A.ack_number
```

9.1.5. A Query Set

Complex analyses are often best expressed as combinations of simpler pieces. In the example below, four queries work together to compute traffic flow by application. These queries are as follows:

- traffic_baseflow
- traffic_portflow
- total
- traffic

All four of the queries listed above can be specified in the same file. By default, the first query can be subscribed to by applications, while the other queries are purely internal. This default can be changed by properly setting the visibility 'define'. Note the following:

- Each query is separated by a semicolon.
- The query set uses different protocol mappings depending on the type of traffic. The results are merged together in the '`total`' query. The merge is legal because both source queries have the same field types in the same order. The '`total`' query merges its streams on their timebucket field; consequently, the output of the merge is ordered on timebucket.

```
DEFINE{
      query_name 'traffic';
```

```
}
SELECT tb*300 as timebucket, application, localID, remoteID, direction,
        SUM(packets) ,SUM(total_bytes)
FROM total
GROUP BY
        timebucket/300 as tb,
        application,
        localID,
        remoteID,
        direction
;

DEFINE{
      query_name 'total';
}
merge t1.timebucket : t2.timebucket
FROM traffic_baseflow t1 ,  traffic_portflow t2
;

DEFINE {
      query_name 'traffic_baseflow';
      aggregate_slots '8192';
}
SELECT  time as timebucket, application,localID,remoteID,direction,
        ULLONG(COUNT(*)) as packets,
        ULLONG(SUM(len)) as total_bytes
FROM IPV4
WHERE ipversion=4 and not (offset=0 and (protocol=6 or protocol=17))
GROUP BY
        time,
        prot2app(protocol,'../../mappings/protocolclass') as
application,
        getlpmid(srcIP,'./remoteprefix.tbl') + INT(1) as localID,
        getlpmid(destIP,'./remoteprefix.tbl') + INT(1) as remoteID,
        INT(1000) as direction
;

DEFINE {
      query_name 'traffic_portflow';
}

SELECT  time as timebucket, application, localID,remoteID,direction,
         ULLONG(COUNT(*)) as packets,
        ULLONG(SUM(len)) as total_bytes
FROM DataProtocol
WHERE ipversion=4 and offset=0 and (protocol=6 or protocol=17)
GROUP BY
        time,
        port2app(protocol, srcPort, destPort,
                 '../../mappings/port2class.txt')  as application,
        getlpmid(srcIP,'./remoteprefix.tbl') + INT(1) as localID,
        getlpmid(destIP,'./remoteprefix.tbl') + INT(1) as remoteID,
        INT(1000) as direction
```

10. Tool References

10.1. Automated Build Script

10.1.1. Synopsis

tigon/tigon-sql/bin/buildit

tigon/tigon-sql/bin/buildit.pl

tigon/tigon-sql/bin/buildit_test.pl

tigon/tigon-sql/bin/buildit_with-stats

10.1.2. Description

The buildit script is a simple shell script which automates the task of building a Tigon SQL instance from a set of GSQL files. Since the buildit script needs to find the Tigon SQL tools and libraries the script expects to be executed in a sub directory of the tigon/tigon-examples/tigon-sql/**demos/** directory such as the **CSVEXAMPLE** directory. From its executed directory the buildit script will combine all files with an ending of .gsql into a Tigon SQL instance. The Tigon SQL instance will include binaries for a run time system, and HFTAs as well as runit and stopit scripts.

The buildit.pl script allows Tigon SQL instances to be compiled in any directory under the tigon root.

Buildit_with-stats is similar to buildit, with the exception that its executables log statistics that help with auto-optimization. The logging slows down processing so for regular processing we advise against enabling the extra logging.

Buildit_test.pl is used for the testing suite and should not be used to develop application instances.

10.1.3. Example

cd tigon/tigon-examples/tigon-sql/CSVEXAMPLE

buildit .pl

10.2.Auto-generated Start and Stop Scripts

10.2.1. Synopsis

./runit

./stopit

10.2.2. Description

The runit and stopit scripts are auto generated when the 'buildit.pl' script is executed. There is one 'runit' and one 'stopit' script for each Tigon SQL instance executing on a single Tigon SQL machine. The scripts combine the knowledge of which binaries need to be started or stopped and in which order to start or stop them. They also know which physical network interfaces need to be instantiated to support the queries of a particular Tigon SQL instance on a particular host. This information is deducted by analyzing the FROM clause in the GSQL statements, the host name of the host and section 4.1-Interface Definition and Use.

10.2.3. Example

cd tigon/tigon-examples/tigon-sql/CSVEXAMPLE

/tigon/tigon-sql/buildit .pl

./runit

... Tigon SQL instance starts up ...

./stopit

... Tigon SQL instance shuts down

See Also

buildit, interface definition file

11. FTA Compiler

11.1.1. Synopsis

Usage: translate_fta [-B] [-D] [-p] [-L] [-l <library_directory>] [-N] [-H] [-Q] [-M] [-C <config_directory>] [-Q] [-S] [-h] [-c] [-f] [-R path] [schema_file] input_file [input file ...]

11.1.2. Description

The command **translate_fta** launches C and C++ programs. The C and C++ programs implement the queries given as the input files. These files must be compiled with the Tigon SQL libraries in order to run properly. The **–M** option will generate a makefile.

The command line options are as follows:

[-B] : debug only (don't create output files)

[-D] : distributed mode.

[-p] : partitioned mode.

[-L] : use the live_hosts.txt file to restrict queries to a set of live hosts.

[-C] : use <config_directory> for definition files

[-l] : use <library_directory> for the query library.

[-N] : output query names in query_names.txt

[-H] : create HFTA only (no schema_file)

[-Q] : use query name for hfta suffix

[-M] : generate Makefile and runit and stopit scripts.

[-S] : enable LFTA statistics (alters the generated Makefile).

[-f] : Output schema summary to schema_summary.txt

[-h] : override host name (used to name the rts).

[-c] : clean out any existing Makefile and hfta_*.cc files before doing code generation.

[-R] : path to the root of tigon

The command line parameters are as follows:

- **schema_file** : describes the Protocols available to be queried.
- **input_file(s)** : queries written in GSQL.

translate_fta creates the following files:

- **<host name>_lfta.c** : The source code for the run time system to be executed at <host name>.
- **hfta_[0-9]+.cc** : the source code for the higher-level query nodes.
- **qtree.xml** : a description of the query tree, in XML.
- **gswatch.pl** : a tool for monitoring the processes in the Tigon SQL processing system.
- **Makefile, runit, stopit :** if the –M switch is set.

Translate_fta depends on the following definition files:

- **external_fcns.def** : prototypes of predicates, functions, UDAFS, etc.
- **ifres.xml** : descriptions of all interfaces available to Tigon SQL.
- <**host_name>.ifq** : query set predicates over the interfaces available at <host name>.

11.1.3. Example

translate_fta –C ../../cfg –M packet_schema *.gsql

See Also

Automated Build Script, GSQL manual

11.4.Printing Streams to the Console

11.4.1. Synopsis

/tigon/tigon-sql/bin/gsprintconsole [-t tcp_port] [-r *bufsize*] [-v] [-X] [-D] query_name param ... param

11.4.2. Description

The command ‘gsprintconsole**’** will instantiate a query within an already running Tigon SQL instance on the local machine. The tuples produced by the query instance are printed to STDOUT as delimited records.

If the **-v** argument is given the first output line contains the field names. The line starts with‘#’. This argument is optional. If –v is specified twice (-v –v) additional diagnostics are printed.

The **–X** argument causes a timestamp to be printed for each output record.

The **-r** argument allows the user to specify the size of the ringbuffer used between the gsprintconsole process and the hfta or rts process which produces the tuples displayed by 'gsprintconsole'. This argument is optional.

The –t argument directs the output to the indicated tcp port instead of to standard output.

After the optional command line arguments the user needs to specify the query name which should be instantiated as well as all parameters required by the query. Parameters are specified as *parametername=value*.

11.4.3. Example

cd tigon/tigon-examples/tigon-sql/CSVEXAMPLE

tigon/tigon-sql/bin/buildit.pl

./runit

... Tigon SQL instance starts up...

tigon/tigon-sql/bin/gsprintconsole -v ping

... ping the main Tigon SQL interface of the machine used...observe the results of the ping query on STDOUT ..../STOPIT

See Also

gsgdatprint

11.4.4. Known Bugs

The directory **gsprintconsole** does not correctly support string parameters which contain non-printable characters.

11.5.Saving streams to files

11.5.1. Synopsis

tigon/tigon-sql/bin/gsgdatprint [**-r** <int>] [**-v] -f -s** [**-z] -c** <int> -q <int> **-b** <field> **-t** <int> **-e** <string> <query_name> <parameters>

Description

The command **gsprintconsole** will instantiate a query within an already running Tigon SQL instance on the local machine. The tuples produced by the query instance are stored in files in gdat format. (The gdat format is an efficient binary format for Tigon SQL Tigon SQL streams). The gdat format can be subsequently processed by such tools as 'gdatcat', 'gdat2ascii' and 'gdat2ethpcap'.

The **gsgdatprint** command has the following optional arguments:

- **-v** makes the STDOUT and stderr output verbose. Has no impact on the stream data in the files.
- **-r** allows the user to specify the size of the ringbuffer used between the 'gsgdatprint' process and the hfta or rts process which produces the streams consumed by gsgdatprint.
- **-z** compress the files generated by 'gsgdatprint' (using gzip)
- **-f** flush each record individually.
- **-s** use in streaming mode (not compatible with –v, -z, -b, -t, or –e).
- **-c** terminate after the indicated number of records are written.
- **-q** terminate after the indicated number of seconds.

The remaining arguments (**-b**, **-t**, **-e**) are monitory and are used as follows:

- In order to generate output files from the data stream, 'gsgdatprint' monitors the values of the field specified by the **–b** option in the data stream.
- It is assumed that the values of that field are increasing monotonically.
- The field type needs to be one of int, uint, llong, ullong.
- Whenever the value of the field specified by **-b** has increased by the value specified in the **-t** option, a new output file is generated.

The output files are named using the following naming convention:

- Each file name starts out with an integer followed by the file extension specified with the **-e** argument.
- If files are being compressed with the **-z** option, 'gsgdatprint' does NOT add the .gz extension automatically. This extension should be part of the extension specifie with **-e**.
- The integer is computed based on the values of the field specified by the **-b** option. The first value of this field seen by gsgdatprint determines the integer part of the first file name.
- Subsequent integer parts of the file name are computed by adding this value to a multiple of the increment specified by the **-t** option.
- Tuples within the data stream are stored in the file with the largest integer component in its name. That integer is smaller or equal to the value of the field in that tuple.

Note: Files are generated only if they are not empty. If the field value progresses more than a single increment per step between two tuples in the data stream, some files may be missing. Missing files do NOT indicate that the Tigon SQL is not functioning properly.

11.5.2. Example

cd tigon/tigon-examples/tigon-sql/demos/ping

tigon/tigon-sql/bin/buildit

./runit

... Tigon SQL instance starts up ...

tigon/tigon-sql/bin/gsgdatprint -b time -t 10 -e ping.gdat ping

... ping the main Tigon SQL interface of the machine used ...

... observe the results of the ping query in files ending in ping.gdat...

... one such file will be generated if time progresses 10 seconds ...

... if there are tuples available ...

./stopit

See Also

gdatcat, gdat2ascii, gdat2ethpcap, and gsprintconsole.

11.5.3. Known Bugs

In addition to the arguments described here, ‘gsgdatprint’ supports a series of experimental arguments. Execute tigon/tigon-sql/**bin/gsgdatprint** to see the current list of supported experimental and production arguments.

11.6.Concatenating Saved Stream Files

11.6.1. Synopsis

tigon/tigon-sql/bin/gdatcat File ... File

11.6.2. Description

The executable ‘gdatcat’ needs to be used to concatenate gdat files generated by gsgdatprint. This executable can only concatenate gdat files which contain data with identical schemas (files produced by one query). Executable ‘gdatcat’ verifies that the aforementioned is true, and stops with an error message if one of the files specified has a different schema. The command ‘gdatcat’ can concatenate a mix of compressed or uncompressed gdat files; however, the concatenated output is always returned to STDOUT without compression.

11.6.3. Example

tigon/tigon-sql/bin/gdatcat 1109173485ping.gdat 1109173495.ping.gat > ping.gdat

See Also

gdat2ascii (section), gdat2ethpcap (section 9.8), and gsgdatprint (section).

11.6.4. Known Bugs

There are no known bugs at this time.

11.7.Converting Saved Stream Files to ASCII

11.7.1. Synopsis

tigon/tigon-sql/bin/gdat2ascii -v File

11.7.2. Description

The executable ‘gdat2ascii’ converts a binary gdat file produced by ‘gsgdatprint’ into a separated ASCII representation identical to the one produced by ‘gsprintconsole’. Only a single uncompressed file name can be specified. If compressed or multiple files need to be converted, ‘gdat2ascii’ should be used in a UNIX pipe in conjunction with ‘gdatcat’. In this case the File name should be set to ‘-‘.

If the **-v** argument is given, the first output line contains the field names. The line starts with a hash mark (#). This argument is optional.

11.7.3. Example

tigon/tigon-sql/bin/gdat2ascii -v 1109173485ping.gdat

tigon/tigon-sql/bin/gdatcat 1109173485ping.gdat 1109173495.ping.gat | tigon/tigon-sql/bin/gdat2ascii -v -

See Also

gdatcat, gdat2ethpcap, gsgdatprint, gsprintconsole

11.7.4. Known Bugs

There are no known bugs at this time.

---

12. External Functions and Predicates

12.1.Conversion Functions

str_exists_substr [string, string] : returns true if the second string is contained within the first.

str_compare [string,string] : returns true if both strings are equal

str_match_offset [uint,string,string] : returns true if the string passed as the second argument matches a substring of the third argument at the offset passed as first argument; otherwise it returns false.

byte_match_offset [uint,uint,string]: matches a byte (passed as second argument) at the offset (passed as first argument) in the string (passed as third argument).True is returned if a match is found; otherwise false is returned.

byte_match_reverse_offset [uint,uint,string] : same as byte_match_offset except that the offset is calculated from the end of the string backwards.

net_word_match_offset [uint,uint,string] : identical to byte_match_offset except that second argument contains a word size number which is matched in network byte order

little_endian_word_match_offset [uint,uint,string] : identical to net_word_match_offset except that the network data is first transfered to little endian order before the match is performed

str_regex_match [string, string HANDLE] : returns true if the string passed as first argument matches the regular expression passed as the second argument.

str_partial_regex_match(string, string HANDLE, uint) : similar to str_regex_match, but match only up the the number of characters specified by the 3rd parameter.

str_extract_regex (string, string HANDLE): similar to str_regex_match except that the first matched string is returned. If no match is found, the function fails.

str_extract_regex_null: Similar to str_extract_regex, but return an empty string if no substring found.

str_file_regex_match(string, string HANDLE, uint, uint): Similar to str_regex_match, but the regular expression is loaded from the file whose path is the second argument. The regex is reloaded every timeout seconds, where timeout is the 3rd argument.

12.2.Conversion Functions

get_int(string s, uint n): extract the 4 bytes which are at positions n-1 through n+3 in string 3, interpret then as an unsigned integer, and return the result.

get_bool(string s, uint n): extract the 4 bytes which are at positions n-1 through n+3 in string 3, interpret then as a boolean, and return the result.

get_suffix(string s, uint n): Extract the suffix of string s starting at byte position n-1.

LLMIN(ullong, ullong): return the minimum of the two arguments.

LLMAX(ullong, ullong): return the maximum of the two arguments.

UMIN(uint, uint): return the minimum of the two arguments.

UMAX(uint, uint): return the maximum of the two arguments.

UMIN(ip, ip): return the minimum of the two arguments.

UMAX(ip, ip): return the maximum of the two arguments.

EQ(uint, uint): return true if the arguments are equal.

GEQ(uint, uint): return true if the first argument is greater than or equal to the second.

LEQ(uint, uint): return true if the first argument is less than or equal to the second.

IF(bool, uint, uint): if the first argument is true, return the second, else the third argument.

non_temporal(int), non_temporal(uint), non_temporal(llong), non_temporal(ullong): cast away any temporal properties of the argument.

INT(uint), INT(ullong), INT(llong): cast the argument as an int.

UINT(int), UINT(ullong), UINT(llong), UINT(ip): cast the argument as an uint.

STRID(string): interpret the first 4 bytes as a uint.

FLOAT(uint), FLOAT (int), FLOAT (ullong), FLOAT (llong): cast the argument as a float.

ULLONG(uint), ULLONG (int): cast the argument as a ullong

strtoi(string): interpret the argument as a uint.

strtoip(string): interpret the argument as an ip.

strtoi_c(string): interpret the constant argument as a uint.

strtoip_c(string): interpret the constant argument as an ip.

12.3.Prefix Functions

getlpmid(ip, string): load a file of ipv4 prefixes and identifiers using the filename specified buy the first parameter, match the ip in the first parameter, and return the corresponding id.

getv6lpmid(ipv6, string): similar to getlpmid but use ipv6 addresses.

12. User-Defined Aggregate Functions

12.1. Moving Sum Functions

moving_sum_exp(uint, uint, float) : computing the sum of the first parameter exponentially decaying over the number of windows specified by the second parameter; the third parameter is the rate of decay.

12.2. String Matching and Extraction

12.2.1. Synopsis

```
PRED [LFTA_LEGAL]str_exists_substr[string, string];
PRED [LFTA_LEGAL]str_compare[string,string];
PRED [LFTA_LEGAL]str_match_offset (uint,string,string);
PRED [LFTA_LEGAL]byte_match_offset (uint,uint,string);
PRED [LFTA_LEGAL]byte_match_reverse_offset (uint,uint,string);
PRED [LFTA_LEGAL]net_word_match_offset (uint,uint,string);
PRED [LFTA_LEGAL]little_endian_word_match_offset(uint,uint,string);
PRED [LFTA_LEGAL] str_regex_match( string, string HANDLE);
PRED [LFTA_LEGAL] str_partial_regex_match( string, string HANDLE,
uint);
string FUN [PARTIAL]str_extract_regex( string, string HANDLE);
string FUN [LFTA_LEGAL]str_truncate (string, uint);
```

12.2.2. Description

This is a collection of functions and predicates that can be used to analyze fields of the type string (such as the payload of a TCP segment). All predicates defined above are also available as functions of type **`uint FUN`**, which return 1 instead of true, and 0 instead of false. The different functions and predicates perform the following:

- str_exists_substr[string, string] : returns true if the second string is contained within the first.
- str_compare[string,string] : returns true if both strings are equal
- str_regex_match[string, string HANDLE] : returns true if the string passed as first argument matches the regular expression passed as the second argument.
- str_match_offset [uint,string,string] : returns true if the string passed as the second argument matches a substring of the third argument at the offset passed as first argument; otherwise it returns false.

- byte_match_offset [uint,uint,string]: matches a byte (passed as second argument) at the offset (passed as first argument) in the string (passed as third argument).True is returned if a match is found; otherwise false is returned.
- byte_match_reverse_offset [uint,uint,string] : same as byte_match_offset except that the offset is calculated from the end of the string backwards.
- net_word_match_offset [uint,uint,string] : identical to byte_match_offset except that second argument contains a word size number which is matched in network byte order.
- little_endian_word_match_offset[uint,uint,string] : identical to net_word_match_offset except that the network data is first transfered to little endian order before the match is performed.
- str_extract_regex(string, string HANDLE): similar to str_regex_match except that the first matched string is returned
- str_truncate (string, uint): returns the first n bytes of the string given as first argument. The n is the minimum of the second argument and the string length of the first argument.

12.2.3.Example

The following example matches TCP payloads to check if some common P2P signatures are present: `select time, srcIP, destIP,`

`srcPort, destPort,`

```
// Here we calculate the bit vectore. This is conceptually the
// same statement as in the WHERE clause, except that we
// calculate a bit vectore not a predicate
// kazaa
(((str_match_offset(0,'GET',TCP_data)
| str_match_offset(0,'HTTP',TCP_data))
& str_regex_match(TCP_data,'[xX]-[Kk][Aa][Zz][Aa][Aa]'))*32)
//gnutella
| str_match_offset(0,'GNUTELLA',TCP_data)*64
// next statement used for Gnutella signal
| ((byte_match_offset(16,HEX'00',TCP_data)
| byte_match_offset(16,HEX'01',TCP_data)
| byte_match_offset(16,HEX'40',TCP_data)
| byte_match_offset(16,HEX'80',TCP_data)
| byte_match_offset(16,HEX'81',TCP_data))
&net_word_match_offset(19,data_length-23,TCP_data))*2
// next statement have to be both matched for directconnect
| (byte_match_offset(0,36,TCP_data)
& byte_match_reverse_offset(1,124,TCP_data)
```

```
& str_regex_match(TCP_data,'^[$](types|MyNick|Lock|Key|Direction|
GetListLen|ListLen|MaxedOut|Error|Send|Get|FileLength|Canceled|HubName|
ValidateNick|ValidateDenide|GetPass|MyPass|BadPass|Version|Hello|
LogedIn|MyINFO|GetINFO|GetNickList|NickList|OpList|To|ConnectToMe|
MultiConnectToMe|RevConnectToMe|Search|MultiSearch|SR|Kick|OpForceMove|
ForceMove|Quit)'))*4
// next statements for bittorent have to be both matched
| (byte_match_offset(0,19,TCP_data)
& str_match_offset(1,'BitTorrent protocol',TCP_data))*8
// next statement for edonkey
| (byte_match_offset(0,HEX'e3',TCP_data)
& little_endian_word_match_offset(1,
data_length-5,TCP_data))*16 as app_class
//, TCP_data
FROM TCP
WHERE (ipversion=4 and protocol=6 and offset=0 )
and ( ( getlpmid(destIP,'./mesaprefix.tbl') = 18 ) or
( getlpmid(srcIP,'./mesaprefix.tbl') =18))
and
// next two statements are used for Kazaa
(((str_match_offset(0,'GET',TCP_data)=1
or str_match_offset(0,'HTTP',TCP_data)=1)
and (str_regex_match(TCP_data,'[xX]-[Kk][Aa][Zz][Aa][Aa]')=1))
// gnutella
or
str_match_offset(0,'GNUTELLA',TCP_data)=1
// next statement used for Gnutella signal
or ((byte_match_offset(16,HEX'00',TCP_data)=1
or byte_match_offset(16,HEX'01',TCP_data)=1
or byte_match_offset(16,HEX'40',TCP_data)=1
or byte_match_offset(16,HEX'80',TCP_data)=1
or byte_match_offset(16,HEX'81',TCP_data)=1)
and net_word_match_offset(19,data_length-23,TCP_data)=1)
// next statement have to be both matched for directconnect
or (byte_match_offset(0,36,TCP_data) =1
and byte_match_reverse_offset(1,124,TCP_data)=1
and str_regex_match(TCP_data,'^[$](types|MyNick|Lock|Key|Direction|
GetListLen|ListLen|MaxedOut|Error|Send|Get|FileLength|Canceled|HubName|
ValidateNick|ValidateDenide|GetPass|MyPass|BadPass|Version|Hello|
LogedIn|MyINFO|GetINFO|GetNickList|NickList|OpList|To|ConnectToMe|
MultiConnectToMe|RevConnectToMe|Search|MultiSearch|SR|Kick|OpForceMove|
ForceMove|Quit)')=1)
// next statements for bittorent have to be both matched
or (byte_match_offset(0,19,TCP_data) = 1
and str_match_offset(1,'BitTorrent protocol',TCP_data)=1)
// next statement for edonkey
or (byte_match_offset(0,HEX'e3',TCP_data)=1 and
little_endian_word_match_offset(1,
data_length-5,TCP_data)=1))
```

12.2.4.Known Bugs

Our fastest regex matching algorithms are not available in the AT&T external release. In fact we replaced them with libc regex functions for the release.

12.3.Longest Prefix Match

12.3.1.Synopsis

```
int FUN [LFTA_LEGAL] getlpmid(IP,string HANDLE);
```

12.3.2.Description

This function is used to perform longest prefix matches against an IP address. The prefix table has the following format: `0.0.0.0/0|0`

```
1.2.3.4/29|5
2.3.4.5/29|6
5.6.7.8/27|6
9.10.10.11/29|7
11.11.11.11/27|7
```

The table needs to be stored in a file on the local filesystem of Tigon SQL. The filename is passed as the string parameter in either function. Currently, the function only evaluates the file if a query is instantiated; therefore, changes to the prefix table file are not reflected in already running queries.

Given an IP address in the first argument the function will return the id of the longest matching prefix. The id is stored after the pipe '|' in the prefix table and must be a uint.

12.3.3.Example

The following query collects IP and TCP header fields of all traffic which either originates or targets an IP address (in prefixes with a prefix id of 1) in the **fitler.tbl** prefix table file.

```
SELECT
time,timestamp,ttl,id,srcIP,destIP,srcPort,destPort,sequence_number,ack
_number,flags,len,data_length
FROM TCP
WHERE protocol=6 and
(getlpmid(srcIP,'./filter.tbl')=1
or getlpmid(destIP,'./filter.tbl')=1)
```

12.3.4.Known Bugs

After a query has been instantiated, the changes to the prefix table are not reflected in the running query.

12.4.Static Subset-Sum Sampling:

12.4.1.Synopsis

Subset-Sum sampling algorithm estimates the sum of object sizes which share a common set of properties. This section describes a basic version of the algorithm, where the threshold of the tuple size is set by the user and doesn't change throughout execution of the program. The result of the algorithm is a sample of arbitrary size.

12.4.2.Description

S – Sample

R – Stream of objects to sample from

t – An object

t.x - The value of the size attribute of an object

z – Initialized by the user, doesn't change during execution

The subset-sum algorithm collects a sample (S) of tuples from 'R' in such a way that accurately estimates sums from the sample. In the static version of the algorithm, the user sets the threshold (z), which determines the sample size. Each tuple (t) is sampled with probability: $p(x) = \min\{1,t.x/z\}$. The following is an implementation of the static subset-sum sampling algorithm:

```
bool static_sample(data t, int z){
  static int count = 0;
  if (t.x > z ){
      //sampled
      return true;
  }
  else{
      count += t.x;
       if (count > z){
            count -= z;
           //sampled
           return true;
       }
   }
   // not sampled
   return false;
}
```

To estimate the sum, the measure ‘t.x’ of the sampled small tuple (less than the value of the threshold ‘z’) is adjusted to: z: t.x = max{t.x,z}.

This static version of the algorithm can be expressed as a stateful function which can access a state structure with relevant control variables.

12.4.3.Example

The following query makes use of a stateful function which implements the sampling procedure of the static version of the algorithm:

```
SELECT time, srcIP, destIP, len
FROM TCP
WHERE nssample(len,100)= TRUE
```

The stateful function ‘`nssample(len, 100)`’ accepts the value of the attribute to sample on (length of the tuple in this case) as well as the value of the threshold ‘z’. This query is evaluated as follows:

- for each incoming tuple, evaluate the WHERE clause. If the function ‘`nssample(len, 100)` ’returns true, the tuple is sampled and outputted. Otherwise start processing the next tuple.
- on every sampled tuple evaluate the SELECT clause and output listed attributes of the tuple.

See also **s**tateful functions, Supergroups, Dynamic Subset-Sum Sampling, and Flow sampling query.

12.4.4.Known Bugs

There are no known bugs at this time.

13.MIN, MAX

13.1.1.Synopsis

```
ullong FUN [LFTA_LEGAL] LLMIN (ullong,ullong);
ullong FUN [LFTA_LEGAL] LLMAX (ullong,ullong);
uint FUN [LFTA_LEGAL] UMIN (uint,uint);
uint FUN [LFTA_LEGAL] UMAX (uint,uint);
IP FUN [LFTA_LEGAL] UMIN (IP,IP);
IP FUN [LFTA_LEGAL] UMAX (IP,IP);
FLOAT FUN [LFTA_LEGAL] UMAX (uint,FLOAT);
FLOAT FUN [LFTA_LEGAL] UMAX (FLOAT,FLOAT);
FLOAT FUN [LFTA_LEGAL] UMAX (FLOAT,uint);
```

13.1.2.Description

The various MIN and MAX functions compute the pairwise MIN and MAX.

13.1.3.Known Bugs

There is no definition of MIN and MAX for all type combination.

13.6.Typecast

13.6.1.Synopsis

```
int FUN [LFTA_LEGAL]INT(uint);
int FUN [LFTA_LEGAL]INT(ullong);
int FUN [LFTA_LEGAL]INT(llong);
uint FUN [LFTA_LEGAL]UINT(int);
uint FUN [LFTA_LEGAL]UINT(ullong);
uint FUN [LFTA_LEGAL]UINT(llong);
float FUN FLOAT(llong);
float FUN FLOAT(ullong);
float FUN FLOAT(int);
float FUN FLOAT(uint);
ullong FUN [LFTA_LEGAL]ULLONG(uint);
ullong FUN [LFTA_LEGAL]ULLONG(int);
uint FUN [LFTA_LEGAL]TIMESTAMPTOSEC(ullong);
uint FUN [LFTA_LEGAL]TIMESTAMPTOMSEC(ullong);
```

13.6.2.Description

These functions perform the appropriate type casts.

13.6.3.Known Bugs

Only typecasts needed for this document are defined; not all possible typecasts.

13.7.Conditional assignment

13.7.1.Synopsis

```
uint FUN [LFTA_LEGAL]IF(uint,uint,uint);
int FUN [LFTA_LEGAL]IF(int,int,int);
string FUN [LFTA_LEGAL]IF(int,string,string);
string FUN [LFTA_LEGAL]IF(uint,string,string);
```

13.7.2.Description

Each 'IF' function returns the second parameter only if the first parameter is not 0, and the third parameter is 0.

13.7.3.Known Bugs

Currently only the functions in the type subsets used are defined.

13.8.Local Triggers

13.8.1. Synopsis

```
PRED [LFTA_LEGAL,COST TOP] set_local_trigger[uint, uint];
        PRED [LFTA_LEGAL,COST HIGH] check_local_trigger[uint];
        bool FUN [LFTA_LEGAL,COST TOP] set_local_trigger(uint, uint);
        bool FUN [LFTA_LEGAL,COST HIGH] check_local_trigger(uint);
```

13.8.2. Description

This function provides a local trigger which can be set by one query by calling 'set_local_trigger' and can be read by other queries by calling 'check_local_trigger'. The first argument in either function or predicate is the trigger ID. Currently up to 127 triggers are supported. The second argument for 'set_local_trigger' indicates how many times 'get_local_trigger' should return true after 'set_local_trigger' was called.

In the example below the collection of 1000 packets is triggered after a packet is visible with a broken Ipv4 checksum using trigger number 1.

```
DEFINE {
query_name 'type_header' ;
real_time 'true';
visibility 'external';
}

SELECT @Host, @Name,time,len,timewarp,raw_packet_data
FROM IPV4
WHERE ipversion = 4 and total_length < hdr_length
;
DEFINE {
query_name 'type_checksum' ;
real_time 'true';
visibility 'external';
}

SELECT @Host, @Name,time,len,timewarp,raw_packet_data
FROM IPV4
WHERE ipversion =4 and ipchecksum(IPv4_header)<>0 and
set_local_trigger(1,1000)=TRUE

;

DEFINE {
query_name 'trailer' ;
```

```
real_time 'true';
visibility 'external';
}

SELECT @Host, @Name,time,len,timewarp,raw_packet_data
FROM IPV4
WHERE ipversion =4 and check_local_trigger[1]
```

13.8.3. Known Bugs

Triggers only function within a single process. Therefore great care must be taken to assure that both queries are compiled into the same binary.

13. User Defined Aggregate Functions

13.1. POSAVG

13.1.1. Synopsis

```
float UDAF POSAVG fstring16(float);
```

13.1.2. Description

A user defined aggregation function which computes the average of all positive values passed to it. This is useful in cases such as processing the output of 'TRAT' which sets its measurement value to -1 if it is not valid. In that circumstance, 'POSAVG' will compute the average of all valid measurements.

13.1.3. Example

13.1.4. Known Bugs

There are no known bugs at this time.

14. Sampling

14.1. Dynamic Subset-Sum Sampling:

14.1.1. Synopsis

The *dynamic subset-sum sampling* algorithm estimates sums of the sizes of flows (objects sharing a common set of properties) from a sampled subset of objects. Unlike the static version of the algorithm, the dynamic version produces a sample of a constant size defined by the user.

14.1.2. Description

S: Sample

R: Stream of objects to sample from

N: Desired size of the sample

T: An object

t.x: The value of the size attribute of an object

z: Dynamically adjusted threshold for the size of the objects to be sampled

B: Number of objects whose size t.x exceeds current threshold z

The algorithm works in the following manner:

- Collect samples, each with probability $p(x) = min\{1,t.x/z\}$
- If $|S| > \gamma N$ *(e.g.,* $\gamma=2$*),* estimate a new value of z which will result in N objects. Subsample S using new value of z, and continue the sampling process.
- When all tuples from R have been processed, if $|S| > N$ then adjust z and subsample S.

Parameter z can be adjusted in a number of ways.

- Conservative
- Aggressive
- Root finding: recursive method (not applicable).

To estimate the sum, the measure 't.x' of the sampled small objects is adjusted to: z: t.x = max {t.x, z}. Then the estimate of the flow size is the sum of all 't.x' in the sample that belongs to the flow. For more details please refer to Sampling Algorithms in a Stream Operator.

Example:

When applied to a data stream, subset-sum sampling occurs in successive time windows. The following query expresses the dynamic subset-sum sampling algorithm which collects 100 samples per 20 seconds:

```
SELECT uts, srcIP, destIP, UMAX(sum(len),ssthreshold())
FROM PKTS
WHERE ssample(len,100)= TRUE
GROUP BY time/20 as tb, srcIP, destIP, uts
HAVING ssfinal_clean(sum(len))=TRUE
CLEANING WHEN ssdo_clean()=TRUE
CLEANING BY ssclean_with(sum(len))=TRUE
```

In this case a single supergroup is created every 20 seconds. The state structure for the algorithm is defined as follows:

```
struct SSstate {
  int count;           // count to sample small packets with
                       // certain probability
  double gcount;       // count for clean_with() function
  double fcount;       // count for final_clean() function
  double z;            // z is the threshold for a size of the
                       // packet
  double z_prev;       // z from the previous iteration of the
                       // cleaning phase
  double gamma;        // tolerance parameter for emergency
                       // control over the number of samples,
                       // should be >= 1
  int do_clean;        // set to 1 when cleaning phase is being
                       // triggered
  int bcount;          // count for number of packets that exceed
                       // threshold, need it for threshold
                       // adjustment
  int s_size;          // need to remember sample size for
                       // _sfun_state_init()
  int final_z;         // indicates if z was adjusted to its
                       // final value before the final clean
  int time;            // remember timestamp from the previous
                       // iteration
};
```

14.1.2.1. Detailed Query Evaluation Process

When the first tuple is received, a supergroup for the time window is created and its state is initialized with '`clean_init`' initialization function call. The function is called only once at the beginning of the execution:

```
void _sfun_state_clean_init_smart_sampling_state(void *s){
struct SSstate *state  = (struct SSstate *)s;
   state->count = 0;
   state->gcount = 0;
   state->fcount = 0;
   state->z = 200;          //initial value for z
   state->z_prev = 0;
   state->gamma = 2;
   state->do_clean = 0;
   state->bcount = 0;
   state->s_size = 0;
   state->final_z = 0;
   state->time = 0;
   };
```

On every packet, evaluate the '`ssample(len,100)`' function, where the parameters passed to the function are the length of the packet in bytes (packet attribute) and the desired final size of the sample for the current time frame of 20 seconds. The function implements the condition for admitting the tuple into the sample with probability p(x) = min{1, len/z}:

```
int ssample(void *s, int curr_num_samples, unsigned long long int
len, unsigned int sample_size){
  struct SSstate *state  = (struct SSstate *)s;
  int sampled = 0;

  //initialize s_size to 100
  state->s_size = sample_size;

  //evaluate when just returned from the cleaning phase
  if(state->do_clean == 1){
    state->gcount = 0;
    state->do_clean = 0;
  }

  //sampling condition
  if(len > state->z){
    state->bcount++;
    sampled=1;
  }
  else{
    state->count += len;
    if(state->count >= state->z){
      sampled=1;
      state->count -= state->z;
    }
  }
  return sampled;
```

```
};
```

If the function returns as false, then the predicate condition for admitting a tuple to the sample failed and the next tuple needs to be processed. If the function passes as true, the tuple is admitted into the sample.

14.1.2.2. Cleaning Phase

On every sampled packet, evaluate '`ssdo_clean()`'. This function implements the condition for triggering the cleaning phase on the current sample and returns true whenever the size of the current sample exceeds the threshold 'γN'. The new value of the threshold is then calculated and the cleaning phase is triggered; otherwise, proceed to the next tuple:

```
int ssdo_clean(void *s, int curr_num_samples){
    struct SSstate *state  = (struct SSstate *)s;

    if(curr_num_samples > (state->gamma*state->s_size)){
       state->do_clean = 1;
      state->z_prev = state->z;
      state->z=(double)state->gamma*state->z;
      state->bcount = 0;
      state->count = 0;
      state->gcount = 0;
    }
    return state->do_clean;
};
```

In this case, the initial threshold can be estimated for the new time window based on the value of the threshold in the previous window, adjusting its value to obtain an estimated 'N' sample during the new time window.

If the cleaning phase was triggered, evaluate '`ssclean_with(sum(len))`' function on every tuple in the current sample. As a result of this evaluation process, the current sample will be subsampled using the previously calculated value of the threshold.

```
int ssclean_with(void *s,int curr_num_samples, unsigned long long
int glen){
    struct SSstate *state  = (struct SSstate *)s;

    //cleaning condition
    int sampled = 0;
    double new_len = 0;

    if (glen < state->z_prev)
      new_len = state->z_prev;
    else
      new_len = glen;
```

```
        if(new_len > state->z){
           state->bcount++;
           sampled = 1;
        }
        else{
           state->gcount += new_len;
           if(state->gcount >= state->z){
              sampled = 1;
              state->gcount -= state->z;
           }
        }
        return sampled;
    };
```

If the function returns as false, then the predicate condition for leaving a tuple in the sample failed and the tuple is deleted; otherwise, the tuple is sampled.

14.1.2.3. Clean Initialization

When the border of the time window is reached, the state of each supergroup (in this example only one) is first finalized with the '`final_init`' function. The new time window is detected when the first tuple from the next time window is received.

```
    void _sfun_state_final_init_smart_sampling_state(void *s, int
    curr_num_samples){
      struct SSstate *state  = (struct SSstate *)s;

      if(state->final_z == 0){

        state->z_prev = state->z;

        if(curr_num_samples < state->s_size){
          state->z = state->z*((max((double)curr_num_samples-
          (double)state->bcount,1))/((double)state->s_size-
          (double)state->bcount));

        }
        else {
          if(curr_num_samples >= state->s_size){
          state->z = stat->z*
         ((double)curr_num_samples/(double)state->s_size);
          }
        }

        if(state->z <= 0)
          state->z = 1;
```

```
        state->bcount = 0;
        state->final_z = 1;
        state->do_clean_count++;

      }
    };
```

Evaluate 'ssfinal_clean(sum(len)' on every tuple that is currently in the sample. The threshold 'z' is adjusted according to the aggressive method of the algorithm. The function makes a final pass through the sample and subsamples it to the desired size.

```
int ssfinal_clean(void *s, int curr_num_samples, unsigned long
long int glen){
    struct SSstate *state  = (struct SSstate *)s;

    state->do_sooth = true;

    // for ssample() where just returned from the clening
    //  phase
    state->do_clean = 1;

    int sampled = 0;
    double new_len = 0;

    if (glen < state->z_prev)
      new_len = state->z_prev;
    else
      new_len = glen;

    //no need to clean
    if(curr_num_samples <= state->s_size){
       return 1;
    }
    else{
       if(new_len > state->z){
          sampled = 1;
        state->bcount++;
       }
       else{
          state->fcount += new_len;
             if(state->fcount >= state->z){
                sampled = 1;
                state->fcount -= state->z;
             }
       }

       return sampled;
```

```
        }
    };
```

If the function returns as true, the tuple is in the final sample for the current time window and its attributes listed in the SELECT clause of the query are outputted to the user. Otherwise the tuple is deleted.

14.1.2.4. Dirty Initialization

When the first tuple from the next time window is received, the new state for this supergroup is created and initialized with '`dirty_init`' (instead of `clean_init`) initialization function. This function uses some values of the state from the previously processed time window. If there is no need in initializing a new state structure with the old state values, the function will be similar to the '`clean_init`' initialization function.

```
    void _sfun_state_dirty_init_smart_sampling_state(void *s_new,
    void *s_old, int curr_num_samples){
        struct SSstate *state_new  = (struct SSstate *)s_new;
        struct SSstate *state_old  = (struct SSstate *)s_old;

        if(curr_num_samples < state_old->s_size){
          state_new->z = state_old->z*
          ((max((double)curr_num_samples-(double)state_old-
          >bcount,1))/((double)state_old->s_size-
          (double)state_old->bcount));

        }
        else {
          if(curr_num_samples >= state_old->s_size){
             state_new->z = state_old-
            >z*((double)curr_num_samples/(double)state_old-
            >s_size);

          }
        }

        if(state_new->z <= 1.0)
          state_new->z = 1;

        state_new->gamma = state_old->gamma;
        state_new->do_clean = state_old->do_clean;
        state_new->s_size = state_old->s_size;
        state_new->bcount = 0;
        state_new->gcount = 0;
        state_new->count = 0;
        state_new->fcount = 0;
        state_new->final_z = 0;
```

```
            state_new->time = 0;
        };
```

From this point on all the tuples in the current window are processed according to the evaluation process described above.

See also the following:

- Stateful functions
- Supergroups
- Static Subset-Sum Sampling
- Flow Sampling Query

14.1.3.Known Bugs

There are no known bugs at this time.

15. Flow Subset-Sum Sampling:

15.1. Synopsis

This section describes a flow sampling query. The packet sampling step in the flow collection process of the Dynamic Subset-Sum sampling approach is replaced by a more sophisticated flow sampling approach which combines the flow aggregation with flow sampling.

15.1.1. Description

The query described in the Dynamic Subset-Sum sampling section is a high level sampling query, which is fed by a low level flow aggregation query. Another approach is to integrate flow aggregation into sampling and do them simultaneously on the traffic at the packet level:

```
SELECT tb, srcIP, destIP, COUNT(*),
       UMAX(sum(len),ssthreshold())
FROM TCP
WHERE flow_ssample(100) = TRUE
GROUP BY time/20 as tb, srcIP, destIP
HAVING flow_ssfinal_clean(P, sum(len)) = TRUE
CLEANING WHEN flow_ssdo_clean(max$(time))= TRUE
CLEANIN BY flow_ssclean_with(P, sum(len)) = TRUE
```

This query uses a new set of stateful functions and is evaluated by the stream sampling operator in the following manner:

1. When a tuple is received, call state initialization function (‘`clean_init`’ if it’s a first tuple being evaluated, ‘`dirty_init`’ otherwise). Evaluate the WHERE clause. Call ‘`flow_ssample(100)` ’, which always returns true and admits all incoming tuples into the sample without performing any preliminary filtering.
2. Evaluate the CLEANING WHEN clause. Call ‘`flow_ssdo_clean(max$(time))`’. This function implements two phases of query evaluation; the *counting* phase and the *cleaning* phase. The counting phase is triggered every second. During this phase the numbers of closed flows which are currently in the group table are counted (see below). The count of closed flows is used to trigger the cleaning phase. The cleaning phase is triggered when the current number of closed flows, which was obtained during the most recent counting phase, exceeds the threshold for the number of samples. If the function returns false, neither of the two conditions is met so proceed to the next tuple.
3. Evaluate the ‘CLEANING BY’ clause whenever ‘CLEANING WHEN’ returns true. Call ‘`flow_ssclean_with(P, sum(len))`’ function, wher‘`P`’ is a set of conditions which indicate whether a flow is closed. For instance, a flow can be

considered closed if we have received FINISH or RESET or there was no packet from this flow within the last 15 seconds:

```
(Or_Aggr(finish)|Or_Aggr(reset)),15,max(time)
```

4. If the function is called during the counting phase, `P` is applied to every group to determine whether the flow is closed. The counter of closed flows which is not evicted from the sample is incremented accordingly. The function always returns true during the counting phase. When the function is called during the cleaning phase, the current set of closed flows is subsampled by applying to each flow the newly estimated value of the size threshold of the tuple and deleting those flows which do not meet the cleaning condition. The function returns as true if the flow satisfies the condition.
5. When the sampling window is closed, call '`final_init`' state initialization function. Evaluate HAVING clause. At this point all flows are considered closed. Call '`flow_ssfinal_clean(P, sum(len))`', which performs the final subsampling of the current sample only if it exceeds the desired size. If the function returns false, the flow is evicted from the sample. Otherwise, the flow is sampled.
6. SELECT is applied to every sampled group while it is output as the answer to the query.

15.1.2. Examples

```
SELECT tb, srcIP, destIP, COUNT(*),
       UMAX(sum(len),ssthreshold())
FROM TCP
WHERE flow_ssample(1000) = TRUE
GROUP BY time/60 as tb, srcIP, destIP
HAVING flow_ssfinal_clean((Or_Aggr(finish)|Or_Aggr(reset)),
       5,max(time), sum(len)) = TRUE
CLEANING WHEN flow_ssdo_clean(max$(time))= TRUE
CLEANIN BY flow_ssclean_with((Or_Aggr(finish)|
           Or_Aggr(reset)),5,max(time), sum(len)) = TRUE
```

This query will return a sample of 1000 tuples per 1 minute time window. A flow is considered closed if FINISH or RESET was received, or if there was no packet from this flow within the last five seconds.

See also the following:

- **S**tateful functions
- Supergroups
- Static Subset-Sum Sampling

- Dynamic Subset-Sum Sampling