Tigon SQL Contributor 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

Contributor Manual

1. Introduction

This manual describes Tigon SQL, the SQL component of Tigon. The goal of this manual is to provide documentation for developers who are writing extensions for Tigon SQL.

2. External Functions and Predicates

The material in this section also appears in the Users Manual, but is repeated here for the the reader's convenience.

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 **Error! Reference source not found.**.

2.1. 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
```

3. Writing Functions and Predicates

User-defined functions and predicates are quite similar. A predicate is a function which returns a Boolean value, and which uses a special declaration in the **external_fcns.def** file (see the User Manual). With a couple of exceptions discussed below, the C or C++ prototype matches the declaration in the **external_fcns.def** file. For example, the prototype corresponding to this declaration

```
ullong FUN [LFTA_LEGAL] PACK (uint, uint);
```

is

```
unsigned long long int PACK(unsigned int, unsigned int);
```

There are two libraries for user-defined functions:

- The lfta library, normally stored in tigon/tigon-sql/src/main/c/**lib/gscplftaaux/**. Put your C code here (*not C++*), and modify the makefile to include the object file in the **libgscplftaaux.a** library. Put the prototypes in tigon/tigon-sql/**include/lfta/rts_external.h** (for functions and predicates) or tigon/tigon-sql/**include/lfta/rts_udaf.h** (for aggregation functions). You must provide lfta code for your function if you declare the function to be LFTA_LEGAL.
- The hfta library, normally stored in tigon/tigon-sql/src/main/c/**lib/gscphftaaux/**. Put your C++ code here (most C code is acceptable) and modify the makefile to include the object file in **libgscphftaaux.a**. Put the prototype in tigon/tigon-sql/**include/hfta/ hfta_runtime_library.h** or **tigon/tigon-sql/include/hfta/ hfta_udaf.h**.

If the function is simple enough, it can be written as '`#define`' in the **.h** file.

3.3.1. String Types

String data is passed by a special structure. This structure is defined for the hfta code in tigon/tigon-sql/**include/vstring.h**:

```
struct vstring {
    gs_uint32_t length;
    gs_p_t offset;
    gs_uint32_t reserved;
};
```

The offset field, when cast to char*, points to the start of the string, which consists of length bytes. The reserved field indicates how the string has been allocated. tigon/tigon-sql/**include/hfta/host_tuple.h** defines the following possible values that a function writer might see:

- INTERNAL : The string is allocated on the heap, there is no sharing.

- SHALLOW_COPY : the string points to a block which is managed by some other entity.

For the lfta, the string structure is defined in tigon/tigon-sql/**include/lfta/rts_external.h** as follows:

```
struct string {
    gs_uint32_tlength;
    gs_sp_t data;
    struct FTA * owner;
};
```

The length and data fields of struct string have the same meaning as the length and offset fields of struct vstring. Do not use the owner field as it is for internal memory management.

String parameters are always passed by reference. For example, the prototype of

```
PRED [LFTA_LEGAL]str_exists_substr[string, string];
```

is

```
int str_exists_substr(vstring *, vstring *);
```

3.3.2. Partial Functions

If a function is declared to be PARTIAL, then its return value is an integer returning zero if the function succeeds, and non-zero if the function fails. The variable which holds the return value is passed by reference to the function. For example, the prototype corresponding to the declaration

```
uint FUN [PARTIAL,COST HIGH] strtoi(string)
```

is

```
int strtoi (gs_uint32_t *ret, vstring *);
```

3.3.3. String Return Values

If a function returns a string, it will be treated as a partial function (whether or not the function is declared to be partial).

If an lfta function returns a string, it **MUST** be a substring of one of its arguments (must be a SHALLOW_COPY).

3.3.4. HANDLE Parameters

If a parameter has a HANDLE modifier, then the code will generate parameter registration and de-registration functions. The function writer must provide these functions, and also adjust the prototype of the actual function (function handles are of type int). The format of the handle registration function name is 'register_handle_for_F*unctionName*_slot_*ParameterNumber'*, with the format of the de-registration function as 'deregister_handle_for F*unctionName*_slot_*ParameterNumber*'.

Parameter numbering starts at zero. As usual, there are different libraries for the lfta and the hfta. For example, consider the declaration below:

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

The function prototypes required by this declaration are as follows:

```
gs_param_handle_t register_handle_for_getlpmid_slot_1(vstring *);
deregister_handle_for_getlpmid_slot_1(gs_param_handle_t);
int getlpmid(unsigned int, gs_param_handle_t);
```

The GS-defined type gs_param_handle_t is large enough to store a pointer. When the query starts, the register_handle functions are called for all handle parameters. The 'register_handle' function performs whatever processing is required and returns an gs_param_handle_t value (generally a pointer cast as an gs_param_handle_t). This value is passed to the function in place of the actual parameter. When the query terminates, the deregister functions are called to release resources.

In the query, parameters occupying the slot of a HANDLE parameter must be a literal, an interface property, or a query parameter. If the parameter is a query parameter, then the deregister_handle/register_handle sequence is called whenever the parameter value id is updated.

3.3.5. Predicates

Predicates are functions that return true/false values. The actual return value is always an integer: zero indicates false and non-zero indicates true. Predicates are indicated by the 'PRED' keyword. In addition their parameter list is enclosed in square brackets. For example, the prototype of

```
PRED [LFTA_LEGAL]str_exists_substr[string, string];
```

is

```
Int str_exists_substr(vstring &, vstring&);
```

3.3.6. Function Modifiers

Function modifiers change how functions and predicates are handled. The modifiers are listed between the FUN/PRED/etc. keyword and the function name. Any number of modifiers can be listed, separated by commas. Some modifiers take a value, in which case the syntax is as follows:

Modifier value

Some modifiers currently recognized by Tigon SQL are as follows:

- PARTIAL : the function may fail to return a value.

- LFTA_LEGAL : The function can be pushed to an lfta query node (by default, functions are hfta-only).
- LFTA_ONLY : The function can only be executed in an lfta query node.
- SAMPLING : The function is used for semantic load shedding.
- COST : An estimate of the cost of executing the function. The possible values of COST from lowest to highest are; FREE, LOW, HIGH, EXPENSIVE, and TOP. Tigon SQL uses the function cost to determine the order in which to evaluate predicate clauses. By default, the COST of a function is LOW. If the COST of a function is HIGH or larger, then Tigon SQL will perform function caching if possible. If the COST of the function is FREE, then Tigon SQL will try to use the function in the lfta prefilter, if possible.

Please refer to Section 2 for a more complete list of function modifiers, such as the example below:

```
    string FUN [PARTIAL, COST HIGH] str_extract_regex( string, string
HANDLE);
```

3.3.7. Function Prototypes

The prototypes of all functions and predicates must be listed in the **external_fcns.def** file for Tigon SQL to be able to use them. Tigon SQL does not use type promotion (e.g. from long int to long long int) when inferring whether a function prototype fits a particular use of the function. However, functions can be overloaded, so you can define the following:

```
FLOAT FUN [LFTA_LEGAL] UMAX (FLOAT,FLOAT);
IP FUN [LFTA_LEGAL] UMAX (IP,IP);
uint FUN [LFTA_LEGAL] UMAX (uint,uint);
```

3.3.8. Combinable Predicates

Some predicates can be combined for a more efficient evaluation. An example of such a predicate is the match_str[data,sub_str,offset] predicate, which returns true if it finds sub_str at offset bytes in the data string. If the length of sub_str is, say, 4 bytes or less, then the predicate has a very fast evaluation, as it is just testing an integer for equality.

Suppose, however, that we have 30 match_str predicates to evaluate. The cost of calling a function, extracting an integer from data, and word-aligning it, can be expensive. By combining all of these predicates into a super-predicate and returning a bitmap indicating success or failure, we can avoid paying the overhead costs for each individual predicate.

The location where we are likely to gather together such a large collection of predicates is in the prefilter, and that is where combinable predicates will be combined. In this section, we discuss how to define and implement a combinable predicate.

A combinable predicate is declared in external_fcns.def by giving a COMBINABLE annotation, and declaring one or more of its parameters to be CLASS parameters. To enable the predicate to be pushed to the prefilter, it should be declared to be FLTA_LEGAL (or LFTA_ONLY), and to be COST FREE:

```
PRED [LFTA_LEGAL, COST FREE, COMBINABLE] match_str[string
        CLASS, string CONST, uint CONST];
```

Only those predicates with identical scalar expressions for their CLASS parameters will be combined. All other parameters must be constants declared to be either CONST or HANDLE (they will be installed in the combined predicate at query initialization time). The data values to be examined must therefore be CLASS parameters. For the other parameters, the implemented can choose whether or not to make them CLASS. For example, the offset parameter determines which portion of the input string is to be extracted and word-aligned. A simpler combined predicate can be written by declaring it to be CLASS.

If a predicate is declared COMBINABLE, then two functions in addition to the regular function must be written and installed in the lfta library. The first additional function registers the constants for a collection of combinable predicates. Its prototype is

```
void *register_commonpred_handles_<pred_name>( {constants from
  predicate}, void *handle, unsigned integer bit_position);
```

If handle is NULL, the handle registration function must allocate a new handle. The bit_position indicates which bit of the output (an unsigned long int) represents the predicate with the registered constants. Since at most 32 predicates can be represented by an integer, the Tigon SQL compiler will register at most 32 predicates in a single handle.

Lets consider an example. Suppose that the query set has the following three predicates:

```
match_str[TCP_data,'foo', 0]
match_str[TCP_data,'bar', 0]
match_str[TCP_data,'boo', 3]
```

If this is the case, the Tigon SQL compiler will generate the following set of handle registration calls:

```
str_constructor(&(t->complex_literal_0),"bat");
str_constructor(&(t->complex_literal_1),"bar");
str_constructor(&(t->complex_literal_2),"foo");
    pref_common_pred_hdl_0_0 = (void
*)register_commonpred_handles_match_str(t->complex_literal_0, 3UL,
                NULL,0);
  register_commonpred_handles_match_str(t->complex_literal_1, 0UL,
                pref_common_pred_hdl_0_0,1);
  register_commonpred_handles_match_str(t->complex_literal_2, 0UL,
                pref_common_pred_hdl_0_0,2);
```

Note that the first call allocates a handle, and subsequent calls add to that handle.

The second additional function is the one that evaluates a collection of common predicates on its CLASS parameters. The prototype for this function is as follows:

```
unsigned int  eval_commonpred_<pred_name>{void *handle, {CLASS
                        parameters});
```

For example, the three predicates would be evaluated using the following call:

```
                    pref_common_pred_val_0_0 =
eval_commonpred_match_str(pref_common_pred_hdl_0_0,unpack_var_TCP_data_
                                18);
```

4. Writing your own UDAFs

Introduction

A User Defined Aggregate Function (UDAF) is a user-written module that computes special purpose aggregates (i.e., instead of AVG, SUM, etc.). Examples of UDAFS are approximate top-K, approximate count distinct, and so on.

A UDAF may take more than one parameter. For example, an approximate "heavy hitters" udaf (which computes the source IP addresses) that sends the most bytes would be invoked by the following:

```
approx_hh_udaf(sourceIP, len)
```

If a UDAF naturally returns a single value, its return value can be used directly. For example,

```
SELECT tb, sourceIP, approx_count_distinct(source_port)
FROM IP
GROUP by time/3600 as tb, sourceIP
```

If the UDAF can produce multiple values, the recommended practice is to write an *extraction function* that can interpret the return value of a UDAF. For example, to get the top three heavy hitters, we would write the following:

```
SELECT  tb, sourceIP,
 approx_hh_extract(approx_hh_aggr(source_port, len),1),
 approx_hh_extract(approx_hh_aggr(source_port, len),2),
 approx_hh_extract(approx_hh_aggr(source_port, len),3)
FROM IP
GROUP BY time/3600 as tb, sourceIP
```

The GSQL compiler will recognize that the three `approx_hh_aggr` references are the same and only need to be computed once. Complex UDAFs can expensive to compute, so this approach leads to significant performance improvements. However the query is cumbersome and places an excessive burden on the user. GSQL provides an alternative facility (described below) to let the user write the following:

```
SELECT  tb, sourceIP,
 approx_hh(source_port, len, 1),
 approx_hh(source_port, len, 2),
 approx_hh(source_port, len, 3)
FROM IP
GROUP BY time/3600 as tb, sourceIP
```

Declaring a UDAF

The file **external_fcns.def** (in the *cfg* directory) contains the prototypes of all functions, predicates, and UDAFS that a query can reference. The GSQL translator uses these prototypes to verify that queries are correctly written, and to generate code.

A UDAF prototype is as follows:

```
<return type> UDAF <name> <storage type> (<param list>);
```

Where

`<return type>` is the data type returned by the UDAF

`<name>` is the name used to invoke the UDAF

`<storage type>` is the data type used to store intermediate results.

`<param list>` is a list of the data types of the parameters.

For example,

```
uint UDAF approx_count_distinct fstring100 (uint);
```

The storage type is the data type of the scratch space provided to the UDAF for storing intermediate results. While any GSQL type can be specified, normally one would use either a variable length string (`string`) or a fixed-length string (`fstring`).

The `fstring` type is a special data type which can only be used as the storage type for a UDAF. The `fstring` is always followed by an integer, which is the number of bytes in the string. For example, `fstring100` is 100 bytes long. Using `fstring` instead of `vstring` is more efficient because the `fstring` does not need to be separately malloc'd, and because it is stored with the rest of the tuple (increasing locality). When an `fstring` is referenced, only a pointer is provided, not the length. By contrast, the string data type is a structure containing both a pointer to the data and the length of the allocated buffer.

GSQL provides a facility by which an extraction function can be called on the return value of UDAF in a convenient way. An *extractor* is defined by the following

```
<return type> EXTR <name> <udaf name> <fcn name> (<param list>)
```

where:

`<udaf name>` is the name of the UDAF

`<fcn name>` is the name of the extraction function

`<param list>` are the types of the calling parameters.

For example, `approx_hh` is declared by:

```
uint EXTR approx_hh approx_hh_aggr approx_hh_extract (uint, uint, uint)
```

The parameters of the extractor are portioned out as follows:

- The parameters of the udaf must be the first parameters of the extractor.
- The first parameter of the extraction function must be the return type of the UDAF.
- The remaining parameters of the extraction function must be the remaining parameters of the extractor.

Also, the return value of the extractor must be the same as the return value of the extraction function. For example, the UDAF and the extraction function of the **approx_hh** extractor are declared as follows:

```
string UDAF approx_hh_aggr fstring1000 (uint, uint);
uint FUN approx_hh_extract (string, int)
```

A basic optimization in Tigon SQL is to split queries into high-level queries and low-level queries. Low-level queries are simple, fast queries that read directly from NIC buffers and are used to reduce the amount of data that needs to be copied. High-level queries complete any remaining processing. The low level queries must be fast; otherwise, the NIC buffers will overflow.

Low-level aggregation queries are processed using a fixed-size direct-mapped buffer to store the groups. Because buffer collisions are possible the result of the low-level aggregation, they must be re-aggregated by a high-level query. If a UDAF specifies its sub aggregate (used by the low-level query) and its super aggregate (used by the high-level query), then a query that references the UDAF can be split. The sub and super UDAFS are specified in the optional modifier list of the source UDAF. For example,

```
string UDAF [SUBAGGR approx_hh_lfta, SUPERAGGR
approx_hh_hfta] approx_hh_aggr fstring1000 (uint, uint);
string UDAF approx_hh_lfta fstring250 (uint, uint);
string UDAF approx_hh_hfta fstring1000 (string);
```

The parameter list of the sub aggregate must be the same as that of the source UDAF, while the only parameter of the super aggregate is the return value of the sub aggregate.

Either both a sub and a super aggregate must be specified, or neither. If they are specified, they must be declared with properly formed parameter lists in **external_fcns.def.** A sub or super UDAF may a sub or super aggregate may not have other sub or super aggregates within its declaration in **external_fcns.def**.

Example 1:

```
SELECT  tb, sourceIP,
 approx_hh(source_port, len, 1)
 approx_hh(source_port, len, 2)
 approx_hh(source_port, len, 3)
FROM IP
GROUP BY time/3600 as tb, sourceIP
```

By putting example one together, the query is changed into the two queries below. (Note: Tigon SQL performs the optimization of computing an aggregate only once, so only one UDAF is listed in the low-level query.)

_lfta_query:

```
            SELECT tb, sourceIP, approx_hh_lfta(source_port, len)
as f3
            FROM IP
            GROUP BY time/3600 as tb, sourceIP
```

hfta_query:

```
            SELECT tb, sourceIP,
             approx_hh_extract(approx_hh_hfta(f3), 1),
             approx_hh_extract(approx_hh_hfta(f3), 2),
             approx_hh_extract(approx_hh_hfta(f3), 3)
            FROM _lfta_query
```

GROUP BY tb, sourceIP

User-written functions

To implement a high-level UDAF, the user must supply the following four routines:

1. **init** : initialize the UDAF scratchpad
2. **update** : add a value to the scratchpad
3. **output** : create an output value from the scratchpad
4. **destroy** : clean up state (release malloc'd memory)

Put the function prototypes in tigon/tigon-sql/**include/hfta/hfta_udaf.h** and the code in tigon/tigon-sql/**src/lib/gscphftaaux/hfta_udaf.cc.**

When a new group is detected, a structure for its aggregate values is created. At this point, the UDAF scratchpads are initialized. During the lifetime of the group, values are presented to the UDAF via the update function. When the group closes, the output function is called once. The result may be referenced several times (e.g., by extractor functions). Finally, the destroy function is called to clean up after the output result is last accessed.

The names of these functions are the UDAF name with one of the following suffixes:

```
init : _HFTA_AGGR_INIT_
update : _HFTA_AGGR_UPDATE_
output : _HFTA_AGGR_OUTPUT_
destroy : _HFTA_AGGR_DESTROY_
```

For example, to implement the `approx_hh_aggr` UDAF, the user needs to supply the following for procedures:

```
approx_hh_aggr_HFTA_AGGR_INIT_,
approx_hh_aggr_HFTA_AGGR_UPDATE_,
approx_hh_aggr_HFTA_AGGR_OUTPUT_,
approx_hh_aggr_HFTA_AGGR_DESTROY_,
```

The functions have some calling conventions:

- variable length strings (buffers) are represented using the following structure (see tigon/tigon-sql/**include/vstring.h**):

```
struct vstring {
      unsigned int length;
      void *offset;
      unsigned int reserved;
};
```

- The 'length' is the length of the buffer.
- The 'offset' is a pointer to the buffer (when cast as a `char *`).
- The `reserved` field is used to indicate how the buffer is stored.
- The 'enum vstring_type' is defined in tigon/tigon-sql/**include/hfta/host_tuple.h**

  The `reserved` field takes values drawn from `vstring_type and` defines two relevant storage types:

  - `INTERNAL` : allocated on the heap, required de-allocation.
  - `SHALLOW_COPY` : does not require de-allocation (copy of a string, etc.)

- The fixed-length strings are stored as character arrays. For example, `fstring100` corresponds to `char[100]`. The length of the `fstring` buffer is NOT passed to the UDAF functions at runtime.
- The "structured" types are passed by reference. Currently the only structured type is the 'string' type.
- The scratchpad is always passed by reference. The `fstring` types are always passed as **`char *`**.

```
void <udaf_name>_HFTA_AGGR_INIT_( <scratchpad type> );
```

The above function should set the UDAF to an initial and empty state. Please note that the INIT function is not passed the first value to aggregate. Instead, the INIT function will be immediately followed by a call to the UPDATE function.

```
void <udaf_name>_HFTA_AGGR_UPDATE_( <scratchpad type>, <param1
type>, ...);
```

Add a value to the UDAF. In general the value can be a vector. Non-structured types (int, unsigned int, etc.) are passed by value; structured types (string) are passed by reference.

```
void <udaf_name>_HFTA_AGGR_OUTPUT_( <return type>, <scratchpad
type>);
```

Extract a value from the UDAF. This function will be called only once, and afterwards no more values will be added. Both the return type and the scratchpad type are passed by reference whether they are structured types or not.

```
void <udaf_name>_HFTA_AGGR_DESTROY_( <scratchpad type> );
```

Perform cleanup, free (destroy) any malloc'd (new'd) buffers.

For instance, the declaration listed under example one, will generate code which references the functions under example two.

Example 1

```
        string UDAF approx_hh_aggr fstring1000 (uint, uint);
```

Example 2

```
        void approx_hh_aggr_HFTA_AGGR_INIT_(char *);
        void approx_hh_aggr_HFTA_AGGR_UPDATE_(char *, unsigned
int, unsigned int);
        void approx_hh_aggr_HFTA_AGGR_OUTPUT_(vstring *, char
*);
        void approx_hh_aggr_HFTA_AGGR_DESTROY_(char *);
```

The declaration of passing strings as parameters (see Example 3), will require the function listed under Example 4.

Example 3

```
        string UDAF approx_hh_hfta fstring1000 (string);
```

Example 4

```
        void approx_hh_hfta_HFTA_AGGR_UPDATE_(char *, vstring
*);
```

Implementation notes:

1) These functions will be compiled with a C++ compiler. If you are uncomfortable with C++, just write C code but avoid using C++ keywords (new, delete, bool, etc.) as variable names.
2) Memory management: Tigon SQL applications are usually very long running, so take care to plug up memory leaks.
    a) vstring management:
        i. Use the function '`hfta_vstr_destroy(vstring *)`', declared in **include/hfta/hfta_runtime_library.h**, to de-allocate a vstring. The 'hfta_vstr_destroy' function will perform a 'free' only if the length is nonzero and the reserved field is INTERNAL.
        ii. Since '`hfta_vstr_destroy`' uses '`free`' for deallocation, use '`malloc`' to allocate a buffer in a `vstring`.
    b) `vstring` return values: Tigon SQL will perform an '`hfta_vstr_destroy`' on `vstring` return values. If the return value is contained in the internal state, set the reserved field to '`SHALLOW_COPY`'**.** The '`_HFTA_AGGR_DESTROY_`' will be called only after last use of the return value.
    c) Once '`_HFTA_AGGR_DESTROY_`' is called, delete any allocated buffers. If nothing has been allocated, do nothing and return.
3) Empty UDAFs: Because of outer joins, it is possible for an UDAF to be empty. That is, the OUTPUT function is called after INIT but without any UPDATE calls. This consideration extends to extractor functions. A SUPERAGGREGATE UDAF or an extractor function might also be passed an empty string. While GSQL supports outer join, it does not have NULL values, so NULL values are replaced by an "empty" default value. In the case of strings, the default is an empty (`length == 0`) vstring. As a result, it is possible to pass empty strings to extractor functions and even to super aggregate UDAFS (but that might mean that the query is badly written). The extractor functions and the SUPERAGGREGATE UDAFs should interpret empty strings as empty UDAF return values.

Implementing a low-level UDAF is nearly identical to implementing a high-level UDAF. The differences are as follows:

1) Put the function prototypes in tigon/tigon-sql/**include/lfta/rts_udaf.h** and the code in tigon/tigon-sql/src/main/c/**lib/gscplftaaux/rts_udaf.c**
2) Low-level UDAFS must be written in C.

3) In low-level queries, variable length strings are represented as shown below:

```
struct string {
      int length;
      char * data;
      struct FTA * owner;
};
```

(Defined in tigon/tigon-sql/**include/lfta/rts_external.h**) strings are always passed by reference, but since there is no typedef, you must specify as shown below:

```
struct string *
```

3) Five procedures must be implemented.

init : _LFTA_AGGR_INIT_

update : _LFTA_AGGR_UPDATE_

flushme : _LFTA_AGGR_FLUSHME_

output : _LFTA_AGGR_OUTPUT_

destroy : _LFTA_AGGR_DESTROY_

The calling conventions for the **init**, **update**, **output**, and **destroy** procedures are the same as for the high-level queries, except that strings are passed as **`struct string *`** instead of **`vstring *`**.

The 'flushme' call is performed after every update (except the first). If the scratchpad is "full", the UDAF can request to be flushed (close the group and emit a tuple for processing by the high-level query). The prototype for the flushme call is as follows:

```
int <udaf_name)_LFTA_FLUSHME_( <scratchpad type> );
```

For example, the declaration under example five requires the functions listed under example six:

Example 5

```
string UDAF approx_hh_lfta fstring250 (uint, uint);
```

Example 6

```
void approx_hh_lfta_LFTA_AGGR_INIT_(char *);
void approx_hh_lfta_LFTA_AGGR_UPDATE_(char *, unsigned
int, unsigned int);
int approx_hh_lfta_LFTA_AGGR_FLUSHME_(char *);
void approx_hh_lfta_LFTA_AGGR_OUTPUT_(struct string *,
char *);
void approx_hh_lfta_LFTA_AGGR_DESTROY_(char *);
```

4) Memory management:

A significant difference between low-level queries and high-level queries is that functions which return strings are not supposed to malloc the string (they are supposed to return substrings). Therefore no string return value is freed. If you need to malloc a buffer for the return value, you must record the buffer in the scratchpad and free it during the 'DESTROY' call (this is possible because 'OUTPUT' will be called only once).

5) Advanced topics

It is possible to declare that UDAF and extractor function parameters are *pass-by-handle*. The pass-by-handle allows you to perform complex processing on a parameter when the query starts. Since pass-by-handle parameters must be constant, this is a mechanism for requiring that a UDAF parameter be a constant. Please note that the only parameter of a SUPERAGGREGATE is the output of the SUBAGGREGATE, so the handle parameter will not be passed to the SUPERAGGREGATE. Please consult section 5-GSQL Users Manual for more information on pass-by-handle parameters.

The GSQL parser will not perform type conversion to call functions. That is, given the declaration under example seven, the expression **`approx_hh(destIP, len, -1)`** will be rejected as a type mismatch error (-1 is not a uint).

Example 7

```
uint EXTR approx_hh approx_hh_aggr approx_hh_extract (uint, uint, uint)
```

In this case, rejecting the -1 parameter is the intended behavior. In other cases, a variety of parameter types (int, uint, ushort) might be acceptable because of type conversion performed by the C/C++ compiler. Creating a function signature with every desired pattern of call types will prevent the overloading of function names.

Example

For a simple example, we'll create a UDAF which computes the average of an unsigned int. The declaration in **external_fcns.def** is as follows:

```
          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);
```

The **avg_udaf_lfta** low-level aggregate needs the following five functions, whose prototypes are as follows:

```
void avg_udaf_lfta_LFTA_AGGR_INIT_(char *);
void avg_udaf_lfta_LFTA_AGGR_UPDATE_(char *,unsigned int);
int avg_udaf_lfta_LFTA_AGBGR_FLUSHME_(char *);
void avg_udaf_lfta_LFTA_AGGR_OUTPUT_(struct string *,char *);
void avg_udaf_lfta_LFTA_AGGR_DESTROY_(char *);
```

Their implementation is as follows:

```
typedef struct avg_udaf_lfta_struct_t{
        long long int sum;
        unsigned int cnt;
} avg_udaf_lfta_struct_t;

void avg_udaf_lfta_LFTA_AGGR_INIT_(char *b){
        avg_udaf_lfta_struct_t *s = (avg_udaf_lfta_struct_t *)b;
        s->sum = 0;
        s->cnt = 0;
}

void avg_udaf_lfta_LFTA_AGGR_UPDATE_(char * b,unsigned int v){
        avg_udaf_lfta_struct_t *s = (avg_udaf_lfta_struct_t *)b;
        s->sum += v;
        s->cnt++;
}

int avg_udaf_lfta_LFTA_AGGR_FLUSHME_(char *b){
        return 0;
}

void avg_udaf_lfta_LFTA_AGGR_OUTPUT_(struct string *r,char *b){
        r->length = 12;
        r->data = b;
}

void avg_udaf_lfta_LFTA_AGGR_DESTROY_(char *b){
        return;
}
```

The **avg_udaf_hfta** high-level udaf needs the following functions:

```
void avg_udaf_HFTA_AGGR_INIT_(char *b);
void avg_udaf_HFTA_AGGR_UPDATE_(char *b, vstring *v);
void avg_udaf_HFTA_AGGR_OUTPUT_(vstring *r,char *b);
void avg_udaf_HFTA_AGGR_DESTROY_(char *b);
```

The implementation is as follows:

```
//              struct received from subaggregate
struct avg_udaf_lfta_struct_t{
        long long int sum;
        unsigned int cnt;
};
```

```
//              sctarchpad struct
struct avg_udaf_hfta_struct_t{
        long long int sum;
        unsigned int cnt;
};

//                      avg_udaf superaggregate functions
void avg_udaf_hfta_HFTA_AGGR_INIT_(char *b){
        avg_udaf_hfta_struct_t *s = (avg_udaf_hfta_struct_t *) b;
        s->sum = 0;
        s->cnt = 0;
}

void avg_udaf_hfta_HFTA_AGGR_UPDATE_(char *b, vstring *v){
        if(v->length != 12) return;
        avg_udaf_hfta_struct_t *s = (avg_udaf_hfta_struct_t *) b;
        avg_udaf_lfta_struct_t *vs = (avg_udaf_lfta_struct_t *)
(v->offset);
        s->sum += vs->sum;
        s->cnt += vs->cnt;
}

void avg_udaf_hfta_HFTA_AGGR_OUTPUT_(vstring *r,char *b){
        r->length = 12;
        r->offset = (unsigned int)(b);
        r->reserved = SHALLOW_COPY;
}

void avg_udaf_hfta_HFTA_AGGR_DESTROY_(char *b){
        return;
}

//              Extraction function
double extr_avg_fcn(vstring *v){
        if(v->length != 12) return 0;
        avg_udaf_hfta_struct_t *vs = (avg_udaf_hfta_struct_t *)
(v->offset);
        double r = (double)(vs->sum) / vs->cnt;
    return r;
}
```

The implementation of the **avg_udaf** aggregate is the same as for the **avg_udaf_hfta** aggregate, except that the 'UPDATE' function accepts a uint parameter:

Example 8

```
void avg_udaf_HFTA_AGGR_UPDATE_(char *b, unsigned int v){
        avg_udaf_hfta_struct_t *s = (avg_udaf_hfta_struct_t *) b;
        s->sum += v;
        s->cnt ++;
}
```

An example query which uses this udaf is the following:

```
          SELECT  tb,destIP,(1.0*sum(len))/count(*),
extr_avg(len)
          FROM TCP
          WHERE ipversion=4 and offset=0 and protocol=6
          GROUP BY time/10 as tb, destIP
```

Running Aggregates

To support running aggregation queries, a UDAF may be declared to be RUNNING.

```
  ullong UDAF [RUNNING] moving_sum_udaf fstring12 (uint, uint);
```

The RUNNING modifier has two effects on the treatment of the UDAF:

- The UDAF must support a REINIT callback, which is made when the aggregation epoch changes. For example,

```
  void moving_sum_udaf_HFTA_AGGR_REINIT_( char * buf);
```

- The 'OUTPUT' will, in general, be called many times.

Additional Modifiers

Some additional modifiers of UDAFS are the following:

- MULT_RETURNS : In some operators, the 'OUTPUT' function of the UDAF will be called many times. The 'MULT_RETURNS' modifier declares that the UDAF supports this type of usage.
- LFTA_BAILOUT : If the 'LFTA_BAILOUT' modifier is set, then when a tuple is to be output at the lfta level, Tigon SQL will first make a call to '<udaf_name>__LFTA_AGGR_BAILOUT_(char *)'. If this function returns non-zero, the output tuple will be discarded.

Note: the LFTA_BAILOUT modifier is used for special performance optimizations, and breaks regular Tigon SQL semantics. This Modifier should only be used by someone who is proficient with this system.

Stateful Functions

Stateful functions allow a user to implement a variety of stream sampling algorithms. A query that uses stateful functions is evaluated and processed by Stream Sampling Operator.

Introduction

To implement some of the algorithms, a number of functions need to have shared access to a number of control variables used to implement a particular sampling algorithm. These control variables are stored in a structure we call a *state*. The specially defined functions that share this state are called *stateful functions*. Stateful functions are very similar to UDAFs, except for the following differences:

- They can produce output a number of times during the execution.

- The state can be modified only from within the functions which share that state, i.e. functions that have access to all control variables of the state.

The control variables stored within the state structure maintain necessary information about the sampling process throughout the entire computation; therefore, the state of a stateful function is always associated with a supergroup rather than a group. In other words, every supergroup maintains its own state structure. If an algorithm uses a number of states, every supergroup will have that number of states associated with it.

To illustrate use of stateful functions, consider the following query:

```
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
```

This query expresses the dynamic subset-sum sampling algorithm which collects 100 samples per 20 seconds, and it uses a number of stateful functions. For example, '`ssthreshold()`' returns the current value of one of the parameters of the sampling algorithm, the dynamically adjusted threshold that determines whether a certain tuple is sampled or discarded.

State Declaration

The file **external_fcns.def** (in the **cfg** directory) contains the prototypes of all states, functions, predicates and UDAFs that the query can reference.

A state prototype is as follows:

```
<storage type>  STATE  <name>;
```

For example,

```
fstring100 STATE smart_sampling_state;
```

The storage type in the declaration should allocate enough memory to store the C structure of the state. While any GSQL type can be specified, normally one would use either a variable length string (`string`) or a fixed-length string (`fstring`). In the example above, the storage type **fstring100** allocates 100 bytes for the state structure, so the state structure must not exceed that size.

The state should be implemented as a C/C++ structure, with the control variables as structure fields. Here is an example of the state structure for the subset-sum sampling algorithm:

```
struct SSstate {

  int count;
  double gcount;
```

```
  double fcount;
  double z;
  double z_prev;
  double gamma;
  int do_clean;
  int bcount;
  int s_size;
  int final_z;
  int time;

};
```

State Initialization

Associated with each state are a number of initialization functions. Their prototypes should reside in **include/hfta/hfta_sfun.h :**

```
void <function name><state name> (void *s);
```

where **s** is the pointer to the state structure.

The pointer should be cast into the appropriate state type within the function implementation code.

For example,

```
void _sfun_state_clean_init_smart_sampling_state(void *s) {
          ...
          state_struct_type *state = (state_struct_type *)s;
          ...
}
```

Most sampling algorithms follow a common pattern of execution:

- A number of tuples are collected from the data stream according to certain criteria (perhaps with aggregation).
- If a condition on the sample is triggered (e.g. the current sample is too large), a cleaning phase is invoked 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.

There are three initialization functions for each state, each corresponding to a certain step within the process of Stream Sampling Operator query evaluation:

- `clean_init` – called when a new state is being created, the initialization values are assigned independently of any other existing state.
- `dirty_init` – called when a new state is being created and the initialization of some of the control variables need to be derived from the state of the old (previous) time widow.

- `final_init` – called right before final cleaning phase, the condition for which is specified in the HAVING clause.

Here is an example of implementation of the '`clean_init`' state initialization function for subset-sum sampling algorithm:

```
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 = 100;
    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;
    state->count_closed = 0;
    state->count_closed_flows = 0;
    state->count_notsampled_new = 0;

};
```

Stateful Function Declaration

Stateful function prototypes reside in **external_fcns.def** file. The declaration of a stateful function ties it to the state the function should have access to:

```
<type>  SFUN [modifiers] <function_name> <state_name>  (<param_list>)
```

For example,

```
    bool  SFUN ssample smart_sampling_state(uint, uint);
```

In the C/C++ implementation of a stateful function, the first parameter is always the pointer to the state structure, followed by the rest of the parameters if any. The C/C++ prototype of a stateful function resides in **hfta_sfun.h** file and has the following form:

```
          <return type> <name>(void *s, int cdistinct,
<param_list>);
```

where **`s`** is the pointer to the state structure and **`cdistinct`** is the default built-in superaggregate which calculates the number of distinct groups on a per supergroup basis.

For example,

```
int ssample(void *s, int cdistinct, unsigned long long int
length, unsigned int sample_size);
```

Definitions of all relevant states, state initialization functions and stateful functions should be included in the user created .cc file(s) in the tigon/tigon-sql/src/main/c/**lib/gscphftaaux** directory.

5. Tigon SQL API

This segment outlines various Tigon SQL APIs. As there are Tigon SQL Tigon SQL APIs, they will be documented as needed. If a particular API is not documented in this manual, it may be listed in the include files in tigon/tigon-sql**/include**. If the include files are not helpful, contact the Tigon SQL team to update this manual with the necessary documentation.

5.1. Application API

Applications can directly instantiate, stop and consume data from Tigon SQL using the Application API. Please consult the **app.h** include file in tigon/tigon-sql**/include** for details. Studying the source code of 'gsprintconsole' (found in tigon/tigon-sql/src/main/c**/tools/gsprintconsole.c)** may also be of use.

6. GDAT File Format

The GDAT file format is self-documenting. Every GDAT file consists of a header which defines the contents of the file, followed by a sequence of records in the defined format.

6.1. GDAT Header Format

The GDAT header consists of three '\n'-terminated lines, followed by a text string which defines the format of records (the *schema*). The first three lines are generated by the following sprint format string:

"GDAT\nVERSION:%u\nSCHEMALENGTH:%u\n"

For example

```
GDAT
VERSION:4
SCHEMALENGTH:536
```

This part of the header means that the schema was generated by version 4 of the GS Tool software, and is 536 bytes long. The next 536 bytes in the file are the schema-defining string, and the first record immediately follows. In this example, these three lines of the header take up 32 bytes, and are followed by 536 bytes of the schema string. Then the first record starts at byte 32+536=568 (using a starting position of byte 0).

The schema-defining stream must be one that can parsed by the GSQL parser and which returns a STREAM parse tree. These strings have the format

```
FTA{
STREAM <name>{
 <list of fields>
}
<query text>
}
```

For example,

```
FTA{
STREAM example {
        UINT systemTime get_field_systemTime ( INCREASING  ) ;
        UINT uintInPosition1 get_field_uintInPosition1;
        ULLONG ullongInPosition2 get_field_ullongInPosition2;
        IP ipInPosition3 get_field_ipInPosition3;
        IPV6 ipv6InPosition4 get_field_ipv6InPosition4;
        V_STR stringInPosition5 get_field_stringInPosition5;
        BOOL boolInPosition6 get_field_boolInPosition6;
        INT intInPosition7 get_field_intInPosition7;
        LLONG llongInPosition8 get_field_llongInPosition8;
        FLOAT floatInPosition9 get_field_floatInPosition9;
        UINT Cnt get_field_Cnt;
}
DEFINE{
```

```
            query_name 'example';
            _referenced_ifaces '[default]';
            visibility 'external';
}
Select systemTime AS systemTime, uintInPosition1 AS uintInPosition1,
ullongInPosition2 AS ullongInPosition2, ipInPosition3 AS ipInPosition3,
ipv6InPosition4 AS ipv6InPosition4, stringInPosition5 AS
stringInPosition5, boolInPosition6 AS boolInPosition6, intInPosition7
AS intInPosition7, llongInPosition8 AS llongInPosition8,
floatInPosition9 AS floatInPosition9, SUM(_t0.Cnt) AS Cnt
From 'dwarf9'.CSV0._fta_example _t0
Group By _t0.systemTime AS systemTime, _t0.uintInPosition1 AS
uintInPosition1, _t0.ullongInPosition2 AS ullongInPosition2,
_t0.ipInPosition3 AS ipInPosition3, _t0.ipv6InPosition4 AS
ipv6InPosition4, _t0.stringInPosition5 AS stringInPosition5,
_t0.boolInPosition6 AS boolInPosition6, _t0.intInPosition7 AS
intInPosition7, _t0.llongInPosition8 AS llongInPosition8,
_t0.floatInPosition9 AS floatInPosition9

}
```

6.2.GDAT Record Format

A GDAT record may be a *data record*, a *trace record* (also called a *temporal tuple*), or an EOF record

A record has four parts

1. A 4-byte unsigned integer, which is the length of the record *not* including this length field.
2. Field data, with formats corresponding to the entries in the field list, in order.
3. A 1-byte field which indicates the type record : 0 for a data record, 1 for a trace record, and 2 for an EOF record.
4. Variable-length data (e.g. string field payloads).

The field formats are:

Data type	**Length in bytes**	**Interpreted as**
INT	4	int
UINT	4	unsigned int
USHORT	4	unsigned int
BOOL	4	unsigned int

IP	4	unsigned int
LLONG	8	long long int
ULLONG	8	unsigned long long int
FLOAT	8	double
IPV6	16	ipv6_str
VSTRING	12	vstring32

The field formats can be found in gstypes.h, vstring.h (vstring32), and packet.h (ipv6_str).

In a data record, string data is packed at the end of the record. A vstring32 record has the format

int32 length
int32 offset
int32 reserved

The offset field is the starting position of the string computed as an offset from the start of the fixed-length data record. The length field is the length of the string field in bytes.

All records are packed sequentially, so there is no guarantee of word alignment when accessing the data stream. Data records carry stream data in the format defined above. Trace records are system-generated and have an opaque format. An EOF record indicates the last record in the stream or file.

For example, a record might be defined by

```
INT foo
STRING bar
```

The a data record (35, “foobar”) would have the representation

- Length: 23 (4 bytes)
- INT FOO: 35 (4 bytes)
- STRING bar: (7,17,0) (12 bytes)
- Record type: 0 (1 byte)
- String payload: “foobar” (6 bytes)

7. GSHUB Service

Tigon SQL relies on GSHUB RESTful service to register and discover Tigon SQL instances, data sources and sinks at runtime. GSHUB implements the following API:

1. Tigon SQL clearinghouse announcement and discovery:

   POST : http://<gshub-hostname>:<gshub-port>/v1/AnnounceInstance with application/ json body { "name" : "<gsinstance_name>", "ip" : "<clearinghouse_ip>", "port": "<clearinghouse_port>" }

GET : http://<gshub-hostname><gshubport>/

   v1/DiscoverInstance/<gsinstance_name> will return application/json response { "ip" : "<clearinghouse_ip>", "port": "<clearinghouse_port>" }

2. Announcement and discovery of initialized Tigon SQL instance:

   POST : http://<gshub-hostname>:<gshub-port>/v1/AnnounceInitializedInstance with application/json body { "name" : "<gsinstance_name>"}

GET : http://<gshub-hostname><gshubport>/

   v1/DiscoverInitializedInstance/<gsinstance_name> will return application/json response { "ip" : "<clearinghouse_ip>", "port": "<clearinghouse_port>" }

3. Data source discovery

GET : http://<gshub-hostname><gshubport>/

   v1/DiscoverSource/<data_source_name> will return application/json response { "ip" : "<data_source_ip>", "port": "<data_source_port>" }

4. Data sink discovery

   GET : http://<gshub-hostname><gshub-port>/v1/DiscoverSink/<data_sink_name> will return application/json response { "ip" : "<data_sink_ip>", "port": "<data_sink_port>" }

Note that names of the Tigon SQL instances, data sources and sinks registered with particular GSHUB must be unique.

A sample implementation of GSHUB service is included in Tigon SQL distribution in tigon/tigon-sql/`bin/gshub.py`. `runit` scripts generated by query translator automatically include the invocation of builtin GSHUB service. Applications that want to implement custom GSHUB service are free to do so as long as they implement GSHUB API described above.

8. Tigon SQL Functionality Guidelines for Internal Usage

When Tigon SQL is used internally, follow these guidelines:

Converting saved stream files to HEX

Synopsis

tigon/tigon-sql/bin/gdat2hex -v File

Description

The executable ‘gdat2hex’ converts a binary gdat file produced by ‘gsgdatprint’ into a separated ASCII representation. In contrast to ‘gdat2ascii’, ‘gdat2hex’ displays strings as hexadecimal values. This is useful if the strings represent binary data rather than printable characters. Only a single uncompressed file name can be specified. If compressed or multiple files need to be converted, ‘gdat2hex’ should be used in a UNIX pipe in conjunction with ‘gdatcat’. In this case the File name should be set to a hyphen (**-**).

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

Example

tigon/tigon-sql/bin/gdat2hex -v 1109173485ping.gdat

tigon/tigon-sql/bin/gdatcat 1109173485ping.gdat 1109173495.ping.gat | *tigon/tigon-sql*/ bin/gdat2hex -v –

See Also

gdatcat, gdat2ethpcap, gdat2pospcap, gsgdatprint, gsprintconsole, gdat2ascii